Alma Mater Studiorum
Università degli Studi di Bologna

ARCES – ADVANCED RESEARCH CENTER FOR ELECTRONIC SYSTEMS

Dottorato di Ricerca in Ingegneria Elettronica
Tecnologie dell' Informazione
ING-INF/01

Ciclo XX

Memory Hierarchy and Data Communication in Heterogeneous Reconfigurable SoCs

Tesi di Dottorato di:
ARSENIY VITKOVSKIY

Coordinatore Dottorato:
Prof. Ing. RICCARDO ROVATTI

Relatore:
Prof. Ing. ROBERTO GUERRIERI

Corelatore:
Dott. Ing. FABIO CAMPI

Anno Accademico 2007-2008

ABSTRACT

The miniaturization race in the hardware industry aiming at continuous increasing of transistor density on a die does not bring respective application performance improvements any more. One of the most promising alternatives is to exploit a heterogeneous nature of common applications in hardware. Supported by reconfigurable computation, which has already proved its efficiency in accelerating data intensive applications, this concept promises a breakthrough in contemporary technology development.

Memory organization in such heterogeneous reconfigurable architectures becomes very critical. Two primary aspects introduce a sophisticated trade-off. On the one hand, a memory subsystem should provide well organized distributed data structure and guarantee the required data bandwidth. On the other hand, it should hide the heterogeneous hardware structure from the end-user, in order to support feasible high-level programmability of the system.

This thesis work explores the heterogeneous reconfigurable hardware architectures and presents possible solutions to cope the problem of memory organization and data structure. By the example of the MORPHEUS heterogeneous platform, the discussion follows the complete design cycle, starting from decision making and justification, until hardware realization. Particular emphasis is made on the methods to support high system performance, meet application requirements, and provide a user-friendly programmer interface.

As a result, the research introduces a complete heterogeneous platform enhanced with a hierarchical memory organization, which copes with its task by means of separating computation from communication, providing reconfigurable engines with computation and configuration data, and unification of heterogeneous computational devices using local storage buffers. It is distinguished from the related solutions by distributed data-flow organization, specifically engineered mechanisms to operate with data on local domains, particular communication infrastructure based on Network-on-Chip, and thorough methods to prevent computation and communication stalls. In addition, a novel advanced technique to accelerate memory access was developed and implemented.

KEYWORDS

Reconfigurable architectures

Heterogeneous Systems-on-Chip

Memory organization

Data structure

Memory access pattern

ACKNOWLEDGEMENTS

This research was sponsored by the European Commission under the 6th Framework program within the MORPHEUS project (IST FP6, project no. 027342). The PhD study was carried out within European Doctorate program in Information Technology (EDITH) at the joint research laboratory of FTM/CCDS STMicroelectronics and Advanced Research Centre for Electronic Systems (ARCES) of the University of Bologna.

I would like to express thanks to my supervisors: Fabio Campi, whose essential guidance and valuable practical advices helped me to increase the scientific quality of my research, and Roberto Guerrieri, who provided me with the continuous support along the whole period of my study. I would also like to thank Georgi Kuzmanov and Georgi Gaydadjiev for the fruitful collaboration during my practical research at CE/EEMCS Delft University of Technology.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF TERMS AND ACRONYMS

AG	Address Generator
AHB	Advanced High-performance Bus
ALU	Arithmetical Logic Unit
AMBA	Advanced Microcontroller Bus Architecture [1]
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CEB	Configuration Exchange Buffer
CLB	Configurable Logic Block
CPMA	Configurable Parallel Memory Architecture
DEB	Data Exchange Buffer
DFG	Data-Flow Graph
DMA	Direct Memory Access
DNA	Direct Network Access
DPM	Dynamical Power Management
DRAM	Dynamic RAM
DSP	Digital Signal Processor
DSS	Dynamic Storage Scheme
DVFS	Dynamic Voltage and Frequency Scaling
eFPGA	Embedded FPGA
FFT	Fast Fourier Transform
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
FSM	Final State Machine
GCD	Greatest Common Divisor
GOPS	Giga Operations

GPP	General Purpose Processor
Granularity	Refers to the size of the computational data element (see section 2.1.2)
HDL	Hardware Description Language
HRE	Heterogeneous Reconfigurable Engine
HW	Hardware
I/O	Input/Output
ILP	Instruction Level Parallelism
IP	Intellectual Property
ISA	Instruction Set Architecture
KPN	Khan Process Net
LCM	Least Common Multiple
LUT	Look-Up Table
MIMD	Multiple Instructions, Multiple Data is a technique employed to achieve parallelism.
MPSoC	Multi-Processor SoC
MUX	Multiplexer
NoC	Network-on-Chip
PN	Petri Net
QoS	Quality of Service
RA	Reconfigurable Architecture
RAM	Random Access Memory
RC	Reconfigurable Computing
RF	Register File
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
RTOS	Real-Time Operating System
RTR	Run-Time Reconfiguration

SIMD	Single Instruction, Multiple Data is a technique employed to achieve data level parallelism.
SoC	System-on-Chip
SPR	Special-Purpose Register
SRAM	Static RAM
SRF	Stream RF
SW	Software
TCM	Tightly Coupled Memory
VLIW	Very Long Instruction Word
XR	eXchange Register

CHAPTER 1

INTRODUCTION

In this chapter, a formal definition of the research goal is presented. The discussion follows the reasons that made the research area to arise and outlines its significance in the modern technology development. In the end, a brief overview of the following chapters is given.

1.1 MOTIVATION

The primary force that pushes the hardware development sector, enabling to appear new devices with faster speed and higher complexity, is better application performance. For four decades, the *Moore's Law* that predicts a doubling of transistor density every 18 months has been providing the necessary conditions to satisfy ever-increasing computation requirements. Unfortunately, nowadays we are reaching the limit when constantly mounting transistor density no longer delivers congruous improvements in application performance. The reasons are known well enough, but a clear alternative solution is not yet available. When increasing the amount of transistors, wire delays and speed-to-memory issues acquire greater side-effects. Aggressive single-core designs unavoidably lead to greater complexity and larger power consumption. On top of it all, scalar processors suffer from a fundamental limitation: their design is based on serial execution, which makes it almost impossible to extract more *instruction-level parallelism* (ILP) from applications.

By now, there are outlined new strategies to supplement Moore's law and, exploring innovative architectures and design concepts, to overcome the limitations of conventional systems. These strategies include:

- Multicore systems that use a set of cores of the similar type on a die to continue delivering steady performance gains.
- Special-purpose processors, including application specific and reconfigurable architectures, that provide enhanced performance in areas where conventional processors perform much poorly.
- Heterogeneous architectures, where computational engines of various nature, complexity and programming approach work cooperatively.

Each of these strategies has potential to deliver substantial performance improvements. However, on the long term, heterogeneous computing has prodigious means for accelerating applications beyond what Moore's law can offer [64], in the same time getting over many of the obstacles that limit conventional architectures. Thus, it is expected for heterogeneous architectures to become extremely important over the next several years.

Although heterogeneous systems have great potential to obtain significant performance, they might be totally ignored by programmers unless they are able to make use of heterogeneous specifics in applications relatively easy. This requires not only dedicated software environment, but – what is even more important – thoroughly implemented *data structure* in the system. Heterogeneous architectures may incorporate data-flows of various natures inside the system, such as computation data, configuration data, control data, different synchronization signals, I/O communications and many others, depending on the type and amount of devices integrated into the system. Managing all these flows within a single organization may become a nightmare. The data structure is primarily based on the interconnection strategy inside the system as well as data storage allocation. Communication and memory subsystems are responsible for the efficient traffic routing, fast and timely data access, and prevention of computational stalls induced by bottle-neck(s) in communication interface(s). Such interfaces are very design critical, being able to introduce great benefits, if enhanced with specific mechanisms to provide the amount and kind of data specifically required by the running application. Thus, intelligent distribution of the data-flows in the system supported by communication and storage means can provide clear programming model for the user. Because of the predefined and well regulated interaction between computational engines, a number of application mapping stages can be easily automated by the compiler or other software tools. In the aggregate, all described above mechanisms enable not only stable system functionality, but also lighten programmer's effort of mapping complex applications.

1.2 OBJECTIVES

This research targets complex reconfigurable Systems-on-Chip with heterogeneous organization. Such systems unify computational engines of various natures, such as general-purpose, application specific, reconfigurable and other devices, within a common architecture. In order to benefit from running complex applications, the integrated engines feature different computational densities and purposes. Application mapping for such systems becomes a very sophisticated task, requiring much manpower and time resources.

The objective of this research is to develop a memory organization that would hide the heterogeneous nature of the system from the programmer, providing a user-friendly interface for application mapping. From the user point of view, the system should represent as close as possible a conventional single-processor architecture with distributed storage organization. Such systems exploit the traditional programming model, thus having the advantage of a clear and well-understood concept of application mapping.

Simultaneously, this approach should preserve the inherent advantages of the reconfigurable architectures that are primarily designed for data intensive computations. Therefore, it is important to provide the required data bandwidth to these devices, supporting respective parallelization and run-time adjustability of the memory access according to the target application needs. In this scope, it is prerequisite to optimize the memory access in order to provide the required data-flow for the most data-hungry computational engines.

1.3 STATEMENT OF ORIGINALITY

Correct organization of the memory subsystem as well as data structure is the crucial task in heterogeneous reconfigurable architectures. However, this context emerged quite recently and thus, generalized methodologies and solutions are not yet state-of-the-art. This research aims at exploring novel approaches and trade-offs to extend the huge opportunities of heterogeneous architectures to the memory infrastructure.

In this scope, topical matter relates to a memory access optimization. Automated addressing FSMs have been successfully used for a long time with high-end Digital Signal Processors (DSPs). However, it is an open research field for massively parallel systems based on GPPs and FPGAs of various natures [10].

1.4 OVERVIEW

The thesis structure is organized as follows. Chapter 2 presents a retrospective of the reconfigurable computing and specifics of the heterogeneous nature of hardware architectures. The main stress is put on the memory subsystem and its significance in the context of the whole system performance. The basic terms, concepts, notions and definitions are given.

Chapter 3 introduces the MORPHEUS target architecture, making emphasis on the most relevant concepts of this research.

Chapter 4 describes the exploration on memory organization and its detailed implementation in the context of the target system.

Chapter 5 discusses the further enhancements of the research topic, presenting the advanced parallel memory access acceleration technique. Together with the previous chapter, this one presents numerical results and comparison with the related works.

Finally, Chapter 6 concludes the thesis by summarizing the developed techniques and solutions, outlining the contribution of this work in the scope of the total project, and showing questions arose by this research that can be basement for the future work.

CHAPTER 2

MEMORY ORGANIZATION IN HETEROGENEOUS RECONFIGURABLE ARCHITECTURES

This chapter presents backgrounds and related research in the area of heterogeneous reconfigurable architectures. The special emphasis is done on the role which memory subsystem plays in the target hardware architectures. Simultaneously, a number of important definitions and general concepts are given which will appear throughout the entire work. All along the discussion, the references to the related work are provided in order to complete a picture of the state-of-the-art research and development in the target area.

2.1 STATE-OF-THE-ART RECONFIGURABLE COMPUTING AND RECONFIGURABLE ARCHITECTURES

Because of its capability to significantly speed-up a large variety of applications, reconfigurable computing has become a subject of a large amount of research. Its most important feature is the ability to run computations in hardware with high performance, while keeping a lot of the flexibility peculiar to software solutions.

Traditionally in computing, there are two primary methods for the execution of algorithms. The first method is to use purely hardware technology, either an *Application Specific Integrated Circuit* (ASIC) or a group of individual components (or *Intellectual Properties*, IPs) integrated in a complex system. ASICs are designed targeting specific computational task. Therefore, they are very efficient when executing the exact computation they were designed for. However, after fabrication the device cannot be adjusted if any part of it requires further improvement, and complete circuit is forced to be redesigned and refabricated. This is an expensive process in terms of resources and time-to-market, especially considering the difficulties imposed by replacing ASICs in a big amount of deployed systems.

The second method is to use software-programmed microprocessors, which is a much more flexible solution. In order to perform a computation, processors operate on a specific *Instruction Set Architecture* (ISA), peculiar to the processor architecture. Execution of different instructions from the ISA alters the system functionality without changing its hardware. However, this method has its own drawback, which consists in significant performance degradation because of low clock speed and/or work rate, being far below that of an ASIC. The processor reads every instruction from memory, decodes it, and only after that executes it. This brings about a large execution overhead for each individual operation. *Pipelined* organization of the instruction execution can reduce the latency overhead by means of additional circuit complexity, and thus higher power consumption, but is not capable to eliminate it completely. In addition, the ISA is defined at the fabrication time of the processor. Any other operations that are to be implemented must be built out of existing instructions.

Reconfigurable computing (RC) in its turn fills the gap between hardware and software solutions, offering potentially much higher performance than software-programmable architectures, while keeping a higher level of flexibility than hardware technologies. Reconfigurable devices, organized in the form of *Field-Programmable Gate Arrays* (FPGAs), consist on an array of computational elements whose functionality is specified by multiple programmable configuration bits, called *configuration bitstream*. These elements are interconnected by means of a set of programmable routing resources. Thanks to such organization, custom digital circuits, composed by a set of logic functions, can be mapped to the reconfigurable hardware by computing the these functions within the computational elements, and using the programmable routing resources to connect the elements together in order to form the necessary circuit.

RC has been shown to accelerate a variety of applications. Taking as example a set of software kernels for multimedia applications, DREAM reconfigurable processor [21] shows average performance about 30 GOPS and 1.8 GOPS/mm^2. For comparison, an ARM926EJ-S processor [9] in the same technology node achieves the performance up to 0.5 GOPS and 0.32 GOPS/mm^2. Neglecting some minor overheads, it would thus be necessary to provide up to 60 ordinary processors to match the performance delivered by DREAM on computation intensive kernels.

In order to achieve such performance supporting a wide range of applications, reconfigurable systems are usually organized as a combination of reconfigurable logic and a *General-Purpose Processor* (GPP). The processor performs various data-dependent control operations that cannot be done efficiently in the reconfigurable logic, while the computation intensive cores are mapped to the reconfigurable hardware. This reconfigurable logic is usually composed of either commercial FPGAs or custom configurable hardware.

Compilation environments for Reconfigurable Architecture (RA) range from tools to assist a programmer in performing a hand mapping of an appropriate functionality to the hardware, to automated systems that map circuit functionality, described in a high-level language, to a reconfigurable system. The design process involves a number of stages. First, a program is partitioned into blocks to be

implemented on hardware, and those which are to be implemented in software on the host processor. The computations intend for the reconfigurable hardware are synthesized into a gate level or register transfer level (RTL) circuit description. This circuit is mapped onto the computational elements within the reconfigurable hardware which are connected using the reconfigurable routing. After compilation, the circuit is ready for configuration onto the hardware at run-time. These steps, when performed using an automatic compilation environment, require relatively small effort, whereas, performing these operations by hand can result in a more optimized circuit for performance-critical applications.

However, the efficiency of the mapping depends not only on the carefully developed manual task distribution or on the advance of automated tools, but also it depends – to a certain extant, even more – on the exact architecture of the target reconfigurable system, its data structure and memory access facilities. This problem acquires even more significance when the matter concerns complex systems unifying two or more reconfigurable devices. Intensive data traffic between reconfigurable units, memory and main processor faces the prospects of communication bottle-necks and computation stalls. In order to avoid such problems, reconfigurable system should represent clearly organized architecture, transparent for compilation tools and end-user.

Since FPGAs are forced to pay an area penalty because of their reconfiguration capability, device capacity can sometimes be a concern. Systems that are configured only at power-up are capable to elaborate only as much of functionality as will fit within their programmable structures. In order to accelerate additional program sections, the reconfigurable hardware should be reused during program execution. This process is known as *run-time* (or *dynamical*) *reconfiguration* (RTR). While this concept allows for the acceleration of a greater portion of an application, it also introduces the configuration overhead, which limits the amount of acceleration possible. Methods such as configuration compression and the partial configuration can be used to reduce this overhead.

Consideration must be given to a fact that reconfigurable computing is very young and rapidly developing concept. Its classifications are still being

formed and refined as new architectures are developed. No unifying taxonomy has been suggested to date. However, several inherent parameters can be used to classify these systems.

2.1.1 HOST INTEGRATION

Host integration is often encountered in practice, when the reconfigurable hardware is coupled with a GPP. The reason is that programmable logic tends to be inefficient at implementing certain kinds of operations, such as variable-length loop and branch control. In order to increase the efficiency, the program sections that cannot be sufficiently mapped to the reconfigurable logic are executed on a host processor. Whereas, the sections with a high computation density, that can benefit from implementation in hardware, are mapped to the reconfigurable logic. For the systems that use a GPP in combination with reconfigurable logic, there are several methods to couple these two computation structures.

1. Reconfigurable hardware may be implemented solely to provide reconfigurable capabilities within a host processor [61], [40]. This solution provides a traditional programming environment with the addition of custom instructions that may vary over time. The reconfigurable units operate as functional units on the main processor data-path using registers to hold the input and output operands.
2. A reconfigurable unit can be used as a coprocessor [80], [54], [63], [20], [8]. In general, a coprocessor is larger and more independent than a functional unit, being able to perform computations without the constant supervision of the host processor. After initialization phase of the reconfigurable hardware, the processor either sends the necessary data to the logic, or provides an address in the memory where this data might be found. The reconfigurable unit performs the necessary computations independently, and returns the results after completion. Such integration strategy allows the reconfigurable unit to operate for a large number of cycles without interfering from the host processor, and potentially permits the host processor and the reconfigurable logic to operate in parallel. This, in its turn, decreases the overhead induced by the use of the reconfigurable logic, compared to the first approach when the reconfigurable unit

communicates with the host processor each time a reconfigurable instruction is used.

3. An integrated reconfigurable unit [78], [7], [39] behaves as if it is another processor in a multiprocessor system or an additional computation device accessed through external I/O. The data cache of the host processor is not visible to the attached reconfigurable hardware. Therefore, there is longer communication latency between the reconfigurable unit and the host processor or main memory. This communication is performed though specific primitives similar to multiprocessor systems. In spite of the latency drawback, this type of RA allows for a great deal of computation independence, by shifting large segments of a computation over to the reconfigurable unit.

All of the described organizations target various communication and computation models. The tighter the integration of the reconfigurable unit, the more frequent is its computation/control data exchange with the host system due to a lower communication overhead. However, tightly coupled hardware is unable to operate with more or less significant segments of data without intervention from a host processor, and the amount of available reconfigurable resources is often quite limited. The more loosely coupled solutions allow for greater execution parallelism, but suffer from higher communications overhead.

2.1.2 GRANULARITY

Reconfigurable hardware is assembled from a set of similar computation elements that form a matrix. These basic elements, traditionally called *Configurable Logic Blocks* (CLBs), features various complexity from a simple block that can perform computation on a bit-level with few inputs (usually, up to three), to a structure operating on a word-level, organized as a small Arithmetical Logic Unit (ALU). The size and complexity of the basic CLBs is referred to as the block's (or RA's) *granularity*.

An example of a *fine-grained* RA is the Xilinx6200 series of FPGAs [6]. Its functional unit can implement two- or three-input functions. Although such architecture is efficient for bit manipulations, it can be too fine-grained to rationally implement a number of circuits, such as multipliers. Similarly, finite

state machines (FSMs) frequently are too complex to be reasonably mapped on fine-grained logic blocks.

Other RAs use a granularity of logic block that can be referred as *medium-grained* [41], [42], [46]. For instance, Garp [41] was designed to perform a set of operations on up to four 2-bit inputs. Another medium-grained structure is embedded inside a general-purpose FPGA to implement multipliers of an adaptive bit width [42]. The CLB used in this architecture is capable of mapping a 4 x 4 multiplication, or being cascaded into bigger structures. The CHESS architecture [46] also operates on 4-bit inputs, with each of its CLBs acting as a 4-bit ALU. In general, medium-grained architecture may be utilized to implement data-path circuits of varying widths, which makes it more similar to the fine-grained structures. However, having the possibility to perform more complex operations of a greater number of values, medium-grained structure can be efficiently used for a wider range of operations.

Coarse-grained architectures are mainly used for the implementation of word-width data-path circuits. Since their CLBs are optimized for large computations, they perform these operations much faster and consume fewer resources than a set of smaller elements connected together to form similar structure. However, because of their fixed internal architecture, it is impossible to make optimizations in the size of operands. The RaPiD [35] and the Chameleon [3] architectures are examples of this which are composed of word-sized adders, multipliers, and registers. If, e.g., only three 1-bit values are required to process, the efficiency of these architectures suffers from a redundant area and speed overhead, since the computation is performed on all of the bits in the full word size.

The granularity of the RA also implies a significant effect on the reconfiguration time of the device. This is a noticeable issue for run-time reconfiguration, when the length of the reconfiguration phase influences the performance of the complete system, causing computation stalls and communication traffic jams. A fine-grained array requires many configuration points to perform very small computations, and thus requires longer bitstream during configuration.

Thus, the granularity parameter effects not only the internal RA functionality, such as efficient computation mapping, execution speed, resource occupation, and power consumption; but also it influences the host system performance in terms of computational efficiency and data distribution. This problem can be solved by having both coarse- and fine-grained arrays and on the same chip.

2.1.3 HETEROGENEOUS ARCHITECTURES

In order to achieve greater performance and computation flexibility, the RAs of mixed granularity can be integrated together within a common system. Such systems are called *heterogeneous* because they unify hardware of various complexity and computational capability.

The most relevant advantage of the heterogeneous approach is that it potentially allows each integrated device to perform the tasks to which it is best suited. This model can employ the dedicated RAs to accelerate some operations up to 100 times faster than what conventional processors can achieve, and consequently expanding the applicability of conventional processor architectures. In the situation when applications include both code segments that could benefit from acceleration and code segments that are better suited for conventional processing, there is no single type of processor which would equally satisfy computation strategies. Heterogeneous organization makes it possible to map each operation of a given application on the right hardware type.

However, there are two primary obstacles hampering heterogeneous RA to be widespread [64]: the programming complexity required for efficient distribution of workloads across multiple computational engines, and the additional effort, required to map a code segment on the appropriate computational device, induced by a specificity of its hardware architectures. These issues can impose significant difficulties, so that any possible advantage of a heterogeneous approach should be compared with the costs to overcome them.

2.2 MEMORY AND COMMUNICATION SUBSYSTEMS IN RECONFIGURABLE ARCHITECTURES

For a long time, the integration of RA in embedded systems has been considered a very attractive alternative for the pure Application Specific Integrated Circuits (ASICs). It allows obtaining software level programmability having the performance of conventional ASICs. The increasing performance and the usability of RA make them a feasible solution for computationally intensive tasks, thus guaranteeing the required data bandwidth for the RA becomes the key implementation issue.

Typically, the software programmable architectures have sequential organization, i.e. a small set of data is computed at a time (usually two, a few more for VLIW or superscalar processors). In these architectures, a HW computational unit is strictly separated from a memory region, providing flexible and programmable addressing mechanism based on single random access. Such architectures also feature a local register file (RF) and a memory access pattern, so that addressing of computation data is performed by specific instructions. Compilers support this addressing mechanism, decomposing the data patterns in terms of collections of single load/store operations, which are described by control flow statements (e.g. for/while loops). The high sophistication degree of modern compilation techniques and the appropriate skillfulness in high level programming languages allow covering almost all kinds of regular data addressing patterns required by applications.

Another feature of ASIC is its space-oriented computation where a theoretically infinite number of computation operands may be required for any given operation. Therefore, data transfer organization in an ASIC is not a trivial task, and it is crucial for its performance. On the other hand, it is always possible to implicitly remove programmability and to implement any data flow and operand retrieving pattern as a part of the design itself. Temporary storage resources, which are usually distributed between computational units and dedicated data channels, are specifically designed where it is required.

In order to benefit from natural parallelism of RA, it is essential to provide a data bandwidth comparable to that of ASICs. In addition, to justify their

integration in a given embedded system, RAs need to be capable of changing the addressing pattern in order to match the reconfigurable computation. Therefore, one of the most critical parts of the system architecture is to assure a suitable data structure which would feed with data such computation dense core. Thus, the bottleneck is passed on to the rapidly growing requirements of the respective memory infrastructure. Thoroughly designed memory subsystem, organized in a layered manner with multi-port local buffers, programmable DMA, efficiently implemented main memory controller and intelligently distributed temporal storage repository, become mandatory. Pure off-chip solutions, without essential on-chip aids, lack the required bandwidth and are unacceptable regarding power consumption and system costs. Memory architecture in reconfigurable SoCs poses additional challenges to the digital designer, since target applications often require flexible memory access patterns (e.g. various word sizes, parallel access or different addressing modes).

2.2.1 THE PROBLEM OF MEMORY SHARING IN EMBEDDED RECONFIGURABLE ARCHITECTURES

In the beginning of the era of reconfigurable architectures, the first designs were oriented at extending the instruction set of standard RISC processors. Machines like PRISM [15], PRISC [61], OneChip [80] in a way or another all used reconfigurable hardware to extend the processor computation capabilities, while still relying on the processor for handling memory accesses.

As the reconfigurable computing development leads to the design of more and more dense extension blocks, these blocks also become more data hungry. One issue that rose from this efforts is that the function unit paradigm, although elegant and compiler friendly, hardly provides enough computation parallelism to justify the reconfigurable hardware utilization. Following proposals in the area of reconfigurable architectures showed a specific interest on the data feeding mechanism. XiRisc [22], if similar in many aspects to the cited designs, provides a three-way VLIW configuration to enhance memory access bandwidth. Garp [20] presents a specific data addressing mechanism implemented in the reconfigurable array. The Molen programming paradigm and HW architecture [74] distributes data in the system in two ways:

- Small amounts of data are exchanged between the processor core and the reconfigurable unit through specific Exchange Registers (XRs).
- Intensive data streams can be connected directly to the extension unit, but in this case either the data flow is routed to the unit with a streaming pattern, or the unit itself is provided with a HDL coding that specifies the type of addressing required

In this case, the user is required to design the addressing pattern of the extension instruction as part of the computation design, as it is the case for ASIC implementation. This solution has the relevant advantage of minimizing exchanges between the unit and the external memory hierarchy, as temporary variables can be handled locally. On the other hand, the approach of mapping memory addressing as part of the micro-coded extension segments could be costly in terms of resources and will make any kind of co-compilation impossible creating two different and separate compilation domains. Also, this solution can only be valid for fine grained reconfigurable units such as embedded FPGAs, while it is hardly applicable to coarse-grained logics. An example of embedded reconfigurable processor where memory access patterns are implemented on the eFPGA fabric is described in [18].

With respect to all architectures mentioned above, coarse grained reconfigurable fabrics feature a significantly different computational grain of the ISA extension segments. As a consequence of this shift, connection between reconfigurable units and the system memory in order to provide enough data to exploit the extension segment potential appears as even more severe. Most coarse-grained data-paths such as Pact XPP [77] or PipeRench [39] do not actively intervene on the data layout: they simply consume data streams, provided by standard external sources or appropriately formatted by an external processor core or by specific DMA logic. Morphosys [66] is only slightly more sophisticated, featuring a dedicated frame buffer in order to overlap data computation and transfers over orthogonal chunks of variable width.

Another interesting solution is the ADRES architecture [51]. ADRES exploits a reconfigurable unit that is similar to that of Morphosys, based on coarse-grain processing elements. In contrast to Morphosys, the ADRES

reconfigurable hardware is used as a functional unit in the frame of a VLIW processor. Data exchange with external memory is realized through the default path of the VLIW processor, and data exchanges take place on the main register file, as it was the case for XiRisc. The programming model is thus simplified because both processor and reconfigurable unit share the same memory access. ADRES is completed by a compilation environment [50] that schedules Instruction Set Architecture (ISA) extensions in order to exploit maximum concurrency with the VLIW core and handles data addressing towards the VLIW register file for both reconfigurable array and hardwired core. A relevant additional value of the compiler, that made the RF-oriented micro-architecture, is that extension instructions are generated by the same compilation flow that produces code for the hardwired core. The provided data is thus randomly accessed and it is not limited to data streaming. However, the VLIW register file may remain a bottleneck for intensive data-flow. A different solution is provided by Montium [67], a coarse-grain reconfigurable processor composed of a scalable set of Tile Processors (TP). Each TP is equiped with RAM buffers, feeding each ALU input. Buffers are driven by a configurable automated Address Generation Unit (AGU). Montium is affected by the same bottleneck as most of architectures overviewed above: in order to exploit its computational density, it needs to fetch from a data repository several operands per clock, and possibly each of them featuring an independent, if regular, addressing pattern.

2.2.2 STREAMING COMMUNICATION MODEL AND AUTOMATED MEMORY ACCESS

In conventional multi-processor embedded systems with high level of computational parallelism, data throughput becomes a significant issue on the path to achieving the expected performance and exploiting the available computational power. State-of-the-art DSP architectures, and massively parallel FPGA devices have dealt extensively with such problems.

A general solution is to structure communication between data storage and processing units in a streaming way. The memory subsystem is based on configurable stream units that move data while computation is performed. Stream units are specialized addressing and data communication blocks that are optimized

for contiguous data transfers. They are organized as a set of stream descriptors, which define the memory access pattern, to pre-fetch and align data in the order required by the computational block. By utilizing the stream units in the memory subsystem, the architecture effectively decouples communication from computation and allows dealing with their implementation and optimization individually [14]. The stream units benefit from pre-fetching data before it is needed and, consequently, the system performance becomes dependent on the average bandwidth of the memory subsystem with less sensitivity to peak latency to access a data element. Chai S.M. et al [25] outline the following main properties of streaming computation model:

- Software computation kernels are independent and self contained. They are localized in such a way that there are no data dependencies between other kernels. The user annotates portions of a program that exhibit this behavior for mapping onto a stream processor or accelerator.
- Computation kernels are relatively static. The processing of each computation kernel is regular or repetitive, which often comes in the shape of a loop structure. There are opportunities for compiler to optimization the computation and the access patterns.
- Explicit definition of communication. Computation kernels produce an output data stream from one or more input streams. This stream is identified explicitly as a variable in a communication data-flow or a signal between computation kernels.
- Data movement completely controlled by programmer. A programmer can explicitly define data transfer from memory to other computation kernels. Hardware mechanisms, such as a DMA or stream unit, provide this capability without intercepting the main processor. The stream communication model allows either minimization of data movement by localizing the computation, or pipeline of computation with data movement. If memory bottlenecks arise, the programmer can retune the memory access.

Automated address generation based on regular patterns can be considered a promising option for providing high performance reconfigurable hardware with

the required bandwidth. In fact, embedded applications, and especially those that benefit most from mapping on reconfigurable architectures, typically feature kernels based on regular addressing patterns, where addressing is more often is generated and incremented with regularity as part of a loop. A convenient way to retrieve data at high parallelism, commonly used in state of the art highly parallel DSPs, is to utilize programmable stridden addressing generation FSMs.

This is achieved in Morpheus by making use of programmable Address Generators (AG), that are appropriately set at the beginning of each computation kernel, and will produce a new local address at each clock or, more precisely, at each request from the data-path, depending on the issue delay of the required computation. Automated addressing FSMs thus add a new level of configurability to the system, providing an adaptive addressing mechanism for reconfigurable units, thus greatly enhancing their potential exploitation of inherent parallelism. As it is the case with reconfigurable computing in general, automated addressing can be considered a viable alternative only if supported in the long term by solid compilation tools that could spare the end user from manual programming. In fact, it is possible to automatically extract from a high level specification of the algorithm (typically C/C++) regular addressing patterns to be applied to automated addressing FSM.

2.2.3 LOCAL REPOSITORY AND ITS IMPLEMENTATION TRADE-OFFS

On-chip static memories (SRAMs) tend to become slower when the variability effect is present [79]. The bigger the design margins that have to be added in the design with negative implications in meeting specific performance constraints, the slower is the actual performance of the system. This effect becomes more evident at higher frequencies, where not only more power is being consumed, but also the slack between the required performance and the one offered by the memories becomes tighter at every technology node. To address this problem, it is necessary to relax operating frequency while still meeting the application real time constraints. One of the advantages of reconfigurable computing is that frequency requirements can be relaxed utilizing parallel architectures for both the functional units and the local layer of the memory hierarchy [53]. For this reason, it is useful to define a memory layer local to the

RA, which can be addressed as a single item by the system, but is organized in small local banks that can be accessed concurrently by all ports of the reconfigurable unit.

The physical structure of such banks can organized as a traditional MUXed SRAM. Built around a standard, off-the-shelf memory manufactured in high volume, a MUXed SRAM offers a very attractive cost structure in terms of power, area and timing. However, this advantage might be misleading: although a standard MUXed SRAM costs less than a specialized dual-port memory on a per-bit basis, the total cost from a system perspective might be significantly higher. Any architecture with shared memory access built around a standard RAM device will require additional facilities in order to enable access to a common memory block for two computational units. This will require additional design resources and most likely will elongate the development cycle. Furthermore, after layout stage, the additional logic may occupy a physical area comparable to that of integrated dual-port memory while requiring additional place and route cost. Considering the performance implications, it appears that a single port MUXed SRAM suffers from a severe disadvantage relative to a multi-port alternative. Since the single port has always to be switched from one device to another, each device accessing the MUXed SRAM will be limited to less than half of the maximum theoretical bandwidth. On the other hand, a multi-port memory capable of supporting simultaneous access, often across different bus widths and frequencies, imposing no delay on either port during a read or write operation. Consequently, its maximum performance will exceed the traditional MUXed SRAM by a factor of at least two [16]. While the technological density is continuously increasing, it is important to move the performance closer to its theoretical capability, narrowing the gap between HW density and practical functionality.

2.2.4 RELATED RESEARCH ON DATA COMMUNICATION AND MEMORY SUBSYSTEMS

There is a number of streaming processor architectures presented in literature over recent years. We can mark out several examples: RAW [73], IMAGINE [28], Merrimac [29], and the RSVP™ architecture [27], [48]. A set of

computational models supporting streaming concept, such as SCORE [24], ASC [52], and Streams-C [38] was also developed.

The IMAGINE processor represents a programmable architecture that achieves the performance of special purpose hardware on graphics and image/signal processing. This is realized by exploiting stream-based computation at the application, compiler, and architectural levels. IMAGINE supports 48 ALUs organized as 8 SIMD clusters. Each cluster contains 6 ALUs, several local register files, and executes completely static VLIW instructions. The stream register file (SRF) is the basement for data transfers on the processor. The memory system, arithmetic clusters, host interface, microcontroller, and network interface all interact by transferring streams to and from the SRF. IMAGINE is programmed at two levels, kernel-level and stream-level. Kernels may access local variables, read input streams, and write output streams, but may not make arbitrary memory references, whereas at the stream-level supported by Stream-C basic functions are provided for manipulating streams and for passing streams between kernel functions. All memory references are deployed utilizing stream load/store instructions that transfer entire streams between memory and SRF. This stream load/store concept is similar to the scalar load/store RISC architecture; it simplifies programming and optimizes the memory system for stream throughput, rather than the throughput of individual accesses. The memory system provides 2.1GB/s of bandwidth to off-chip SDRAM storage via four independent 32-bit wide SDRAM banks operating at 143MHz. The system can perform two simultaneous stream memory transfers. To support these parallel transfers, four streams (two index streams and two data streams) connect the memory system to the SRF. IMAGINE addressing modes support sequential, constant stride, indexed (scatter/gather), and bit-reversed accesses on a record-by-record basis (see Fig. 1).

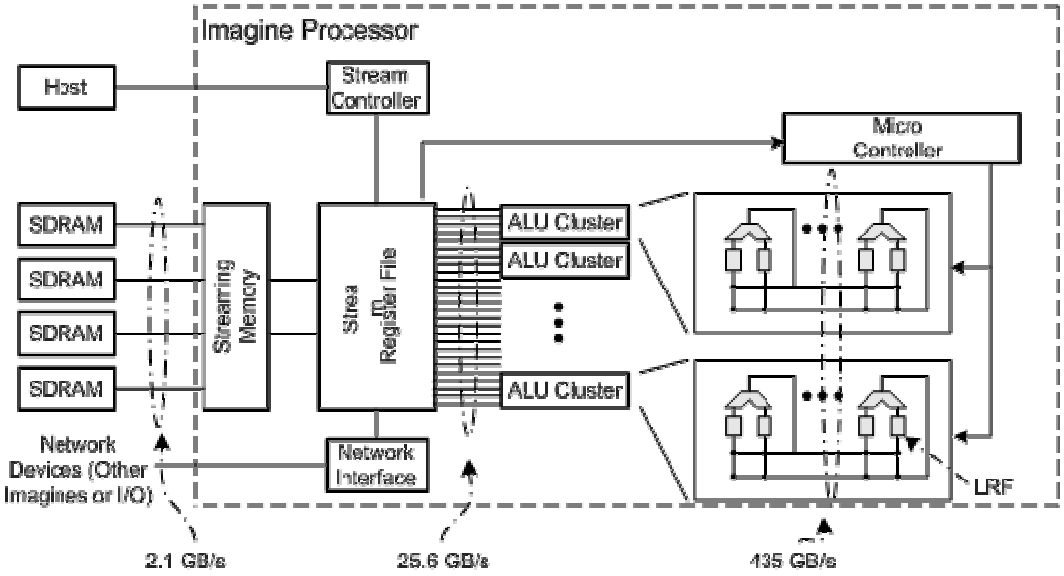


Fig. 1. Subsystem and bandwidth hierarchy of IMAGINE processor [28].

Along with homogeneous stream multi-processor platforms there exists a number of heterogeneous reconfigurable systems, whose memory organization provides different trade-off solutions between computation power and flexibility: examples are Chameleon [3], BAZIL [72], Pleiades [81], etc.

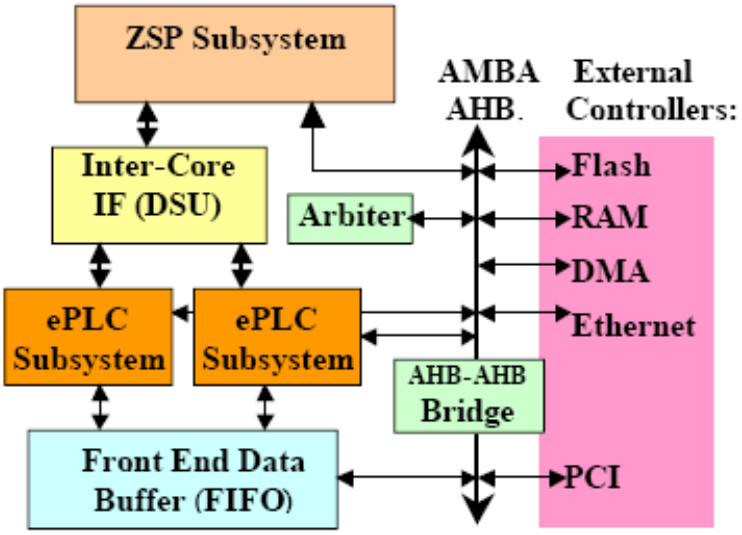


Fig. 2. BAZIL architecture block diagram [72].

The BAZIL architecture is based on the concept of intra-chip DSP and programmable logic block co-processing. Key blocks in BAZIL are LSI Logic ZSP400 (ZSP) and ePLC cores, which are integrated together with other peripherals for external memory and data access. BAZIL's heterogeneous

architecture can be controlled by bus masters, which include the ZSP and DMA controller, and AHB-bridge. These alternative methods of on-chip cores and peripherals control allow robust solution for system programming. BAZIL supports combined boolean and DSP processing through a flexible core interconnect scheme. There are two types of inter-core communication supported in BAZIL architecture:

- An AMBA High-speed Bus (AHB) provides an arbitrated mean for inter-core communication.
- The DSP and ePLC blocks are additionally interconnected through a higher bandwidth inter-core interface (DSU) that allows higher throughput and data sharing between cores.

Since BAZIL interfaces are AMBA based, they represent straightforward and convenient concept. The ZSP400 Core exploits in parallel two independent interfaces for memory and peripherals (see Fig. 3): an Internal Port interface for close coupled, single cycle program and data memory; and an External Port for Instruction Unit (IU) and Data Unit (DU) alternative access to external memory and peripherals. Both internal and external ports contain instruction and data interfaces that support either single-port or dual-port memories. The ePLC sub-systems (see Fig. 4) are intended as loosely coupled co-processors for algorithm acceleration. Each ePLC is made up of:

- The Multi Scale Array (MSA), containing user programmable portions of the ePLC and consisting of an array of configurable ALU (CALU) Cells.
- The Application Circuit Interface (ACI), providing the signal interface between the MSA and the application circuitry.
- The PLC Adapter, loading the ePLC configuration data and interfaces to test circuitry, clock and reset control through a Configuration Test Interface. It allows the ePLC programming to be handled over the on chip AHB from flash or other external memory.

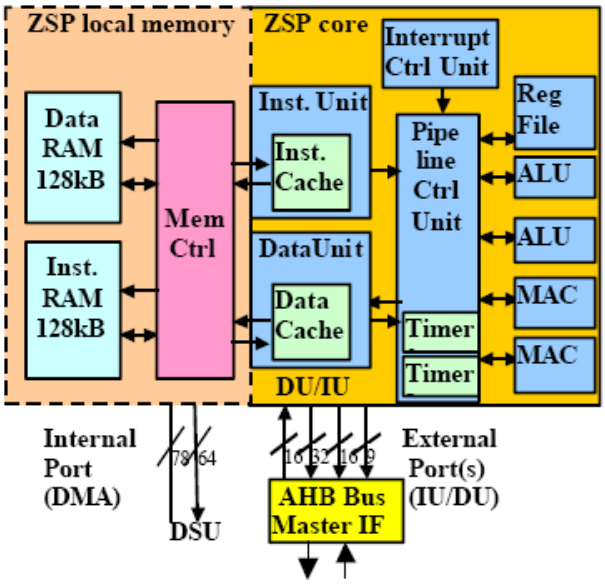


Fig. 3. ZSP-core memory subsystem [72].

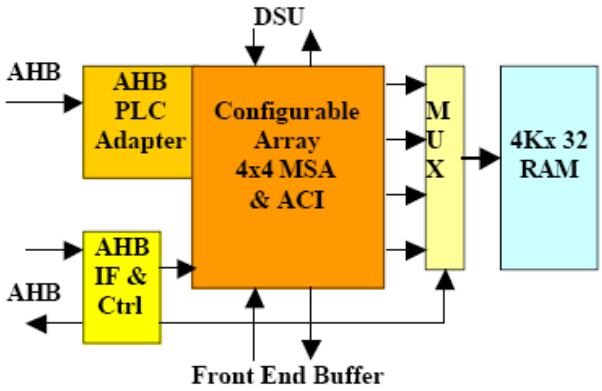


Fig. 4. ePLC-core memory subsystem [72].

In typical broadband processing (see Fig. 5), data is transmitted in a batch or streaming mode through a high-bandwidth buffered interface (PCI port). The data buffer simplifies the caching mechanism of the on-chip data bursts. ePLC blocks are used to perform a range of pre-processing and data reduction operations. Then data is passed to the ZSP, either through shared memory or directly from the DSU for DSP operation. The DSP output data can be either exported off chip or to the ePLC for further post processing via the shared ZSP internal memory. While the DSU does not provide a communication channel between the ePLC subsystems, the ePLC bocks can communicate via the shared ZSP internal memory or FEB. It is also possible to move data between ePLC systems via ZSP controlled AHB traffic. The amount of data available and used in different processing steps (pre-DSP and post-processing) typically is reduced with

each step. As a result, interfaces required for export of processed data (Ethernet) can have significantly lower bandwidth than those needed during import stages (i.e. PCI).

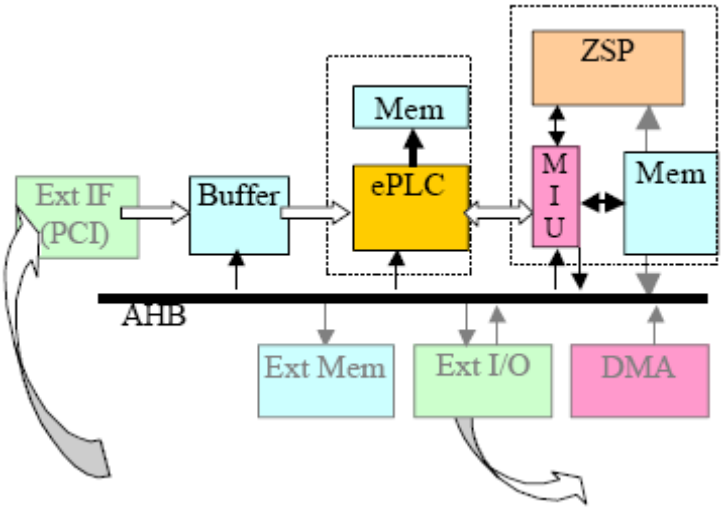


Fig. 5. BAZIL dataflow [72].

2.3 SUMMARY

To date, reconfigurable computation reached high level of development and proved its feasibility in a great range of implemented devices. Target applications become more and more complex, including dense computation for multimedia kernels, data intensive flows for communication tasks and support for Run-Time Operating Systems (RTOS). Heterogeneous architectures are meant to deal with such applications.

One of the most tricky and critical tasks in heterogeneous hardware design is to well organize the memory susbsystem. In contrast to homogeneous systems, the memory storage of a heterogeneous architecture must deal with data traffic of various nature, granularities and densities. Moreover, it is essential to delimit computational engines from each other in order to fully exploit particular features of each hardware component. In this scope, management of the local storages acquires an additional importance.

There are various methods to provide computational engines with the required data dandwidth. Streaming data-flow, parallel memory access, local data buffering, programmable data pattern – all of them aim at a more efficient and flexible data distribution. Comprehensive utilization of these techniques will allow releasing the full potential of heterogeneous reconfigurable architectures.

CHAPTER 3

OVERVIEW OF THE MORPHEUS HETEROGENEOUS RECONFIGURABLE PLATFORM

Most of the research was performed in the framework of the MORPHEUS project, which includes an architecture design and hardware development of the heterogeneous reconfigurable platform. Since the project is an integration of the standard blocks, custom-design IPs and conceptual innovations from various industrial and academic partners, it is essential to make a general overview of these contributions and summarize the related to this research features. Consequently, this chapter presents the most relevant features of the MORPHEUS platform which are significant in the scope of the research area of this thesis. Namely, the overview of the platform is detailed with the descriptions of the reconfigurable engines provided by MORPHEUS partners. The presented specifics will be used in the following chapters as a basis for the design concepts' development and justification.

3.1 MORPHEUS DESCRIPTION

The large-scale deployment of embedded systems, through cooperating objects for example, is raising new demanding requirements in terms of computing performance, cost-efficient development, low power, functional flexibility and sustainability. This trend results in an increasing complexity of the platforms and an enlarging design productivity gap: current solutions outlived their potential while current development and programming tools do not support the time-to-market needs.

MORPHEUS is a *Multipurpose Dynamically Reconfigurable Platform for Intensive and Heterogeneous Processing* – a technology breakthrough for embedded computing. It copes with the above introduced challenges by developing a global solution based on a modular SoC platform providing the disruptive technology of embedded dynamically reconfigurable computing completed by a software (SW) oriented design flow. These "Soft Hardware" architectures will enable huge computing density improvements (GOPS/Watt) by a factor of 100x, reuse capabilities by 5x, flexibility by more than 100x and time-to-market divided by 2, thanks to a convenient programming toolset.

Hence, MORPHEUS provides a new concept of flexible "domain focused platforms", positioned between general purpose flexible hardware and general purpose processors. It delivers:

- A modular silicon demonstrator composed of complementary run-time reconfigurable building blocks to address the different types of application requirements
- The corresponding integrated design flow supporting the fast exploration of HW and SW alternatives

3.1.1 OBJECTIVES

From a business perspective, embedded systems are facing tough cost-effectiveness issues: the mitigation of increasing developments costs of silicon platforms dedicated by markets (customer-centric development with stringent time to market and short lifecycle constraints) is imperative: the customization after fabrication by SW techniques promises to give the response provided that

challenges in architecture and design tools are overcome. The first objective of this project is to provide best of class solutions in these domains and to support these key markets.

Existing commercial products (mainly low architecture level FPGA from United States vendors, completed by some Intellectual Property products) bring limited benefits in combining flexibility (field programmability) and efficiency (computing density, development time) due to the lack of hybrid architecture and late binding capabilities. On the other hand, cutting edge research programmers in Europe and R&D programmers in the US demonstrate decisive improvements through dynamic reconfiguration on coarse grain architectures provided that ambitious associated tools exist. Thus, MORPHEUS objective to provide a new type solution summarized in Fig. 6.

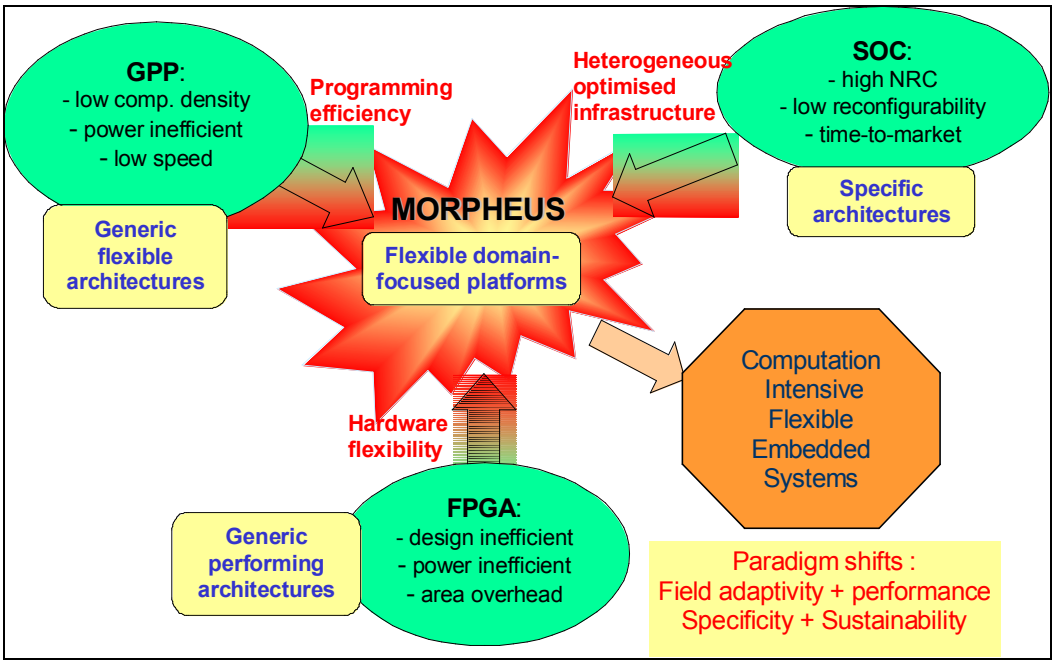


Fig. 6. MORPHEUS objectives.

The figure shows the position of the MORPHEUS platform in the range of computing solutions for embedded systems, between generic, programmable but inefficient General Purpose Processors (GPP), optimized but inflexible ASIC and flexible generic but inefficient FPGA. In summary, the MORPHEUS aims at enabling flexible ”Domain Focused Platforms” providing breakthroughs in performance and cost-effectiveness to embedded computing systems.

As high intrinsic cost of FPGA dominated by wires routing is caused by their general-purpose character, to solve this problem, MORPHEUS proposes the concept of domain focused platform with specialization of reconfigurable logic. According to this concept, the cost reduction can be achieved if reconfigurable logic architecture is optimized towards requirements of processing kernels from a target application domain instead of being made fully general-purpose. The different applications domains are representative of data-path-oriented, random-logic-oriented, or memory-oriented: it will thus provide an efficient validation of the suitability of the proposed platform for a wide range of embedded systems.

Dynamical re-configurability [17] leads to multifunctional hardware at low cost, low power consumption and high performance, making the platform attractive and cost effective for a new breed of embedded applications. Finally, for most identified functions, the main benefits of run-time reconfiguration in embedded products should be real estate (i.e., reduction of the area) and power consumption (i.e., reduction of the interconnection).

Reconfigurable devices with their regular structure are good candidates for deeper sub-micron technologies (and even beyond CMOS). Speed and clock distribution make a strong case against long distance interconnection, hence favoring mesh-connected designs with fast communications between neighbors, as well as the optimized distribution of memory and processing resources. Using hierarchical architectures and wiring by abutment in that spirit could potentially solve the current scalability problem of FPGAs. As an effect of limited routing resources, the wiring congestion worsens with the size of the circuit. Implementing a Network-on-Chip with capabilities for reconfiguration is another way to increase scalability.

Coarse granularity facilitates time flexibility (whereas other solutions are also possible to implement dynamic run-time re-configurability) and, associated to well defined high level architecture and abstraction layers, also facilitates, in some extent, design exploration and compilation. On the other hand, fine granularity facilitates spatial flexibility and probably high level synthesis. Thus, a heterogeneous mixed-grained programmable architecture constitutes the HW platform target for MORPHEUS.

In order to prioritize and quantify the objectives, an analysis by domains has been made to take into account application dependant metrics. It results in the following list of quantified and prioritized global objectives compared with today's state-of-the-art FPGAs:

- Faster time to market is the primary goal for all application test cases. FPGAs typically require each of the multiple tasks of a complex application to be broken down to RT-level. Higher abstraction, better adaptation to memory and interconnect requirements and better support of reuse shall decrease the design/implementation/validation/debug phase by at least 50%. By the way this would result in cutting by half the number of re-designs and the associated cost due to protocol/standard changes. In term of overall cost-effectiveness it can also be mentioned that the unit prices should be reduced by 5 to 10 compared to future (3 years) FPGA, as a result of flexibility (best use of HW resources) and of an optimized computing architecture.
- Second major goal is an increased flexibility. It is mandatory to map multiple tasks and or applications to the same processing platform. This needs to take place during runtime (e.g. to adapt to a new image format or decoding scheme) – i.e. without reboot of the entire unit or any loss of data. This objective can be achieved in combination with error correction mechanism if the reconfiguration time is locally below 20µs (coarse grain level) and globally 1ms. (Several seconds are typical for today's larger FPGAs).
- Processing power associated with high data rates (e.g. up to 40Gbit/s per wavelength for telecommunications systems) are another concern for the targeted applications. Today's FPGAs already support 40 Giga multiplication/accumulation operations per second and more. In absolute numbers, this should be sufficient to implement most required algorithms. The processing power needs to be high enough to process all incoming data streams without any loss of data. It is required that such numbers can be sustained for schemes more complex than FIR-filters though.

- Power consumption is another goal. The efficiency of FPGAs should be met. Increases in efficiency (in terms of GOPS/Watt) by at least a factor of two are considered desirable and realistic.

3.1.2 TARGET APPLICATIONS

These techniques can be applied whenever systems adapt to changing requirements, especially when there are severe constraints upon the cost/size/power of the computer hardware. Examples of these applications include communications, multimedia, instrumentation, and robotics. MORPHEUS specifically addresses:

- Broadband Wireless Access Systems, where IEEE 802.11a and 802.16-2004 implementation on same SoC is expected to lead to important silicon area savings and design time improvement.
- Network routing systems, where RC-enhanced network processors designs are poised to make an impact on future packet-processing systems in so-called active networks.
- Professional video, where it is expected that typical image processing operations can be mapped very efficiently on coarse grain reconfigurable architectures.
- Homeland security, where improved detection and identification require intelligent camera systems where algorithms require sustained performance in the tens of GFLOPS while providing cost-effectiveness and sustainability.

An example of a target application is presented hereafter in order to examine the system requirements in more detail. The application is a general purpose (multi applications) image processing system for vision applications. Traditionally, the implementation of image processing consists in mapping a hardwire RTL algorithm over FPGA. Such an approach is useful as far as only very simple image processing algorithms are considered. The typical example is a pre-processing algorithm for use in image display systems. However for more sophisticated cases, the lack of low cost reconfiguration capability of such approach leads to significant shortcomings for the following reasons:

- Only a very limited number of operating modes can be provided due to the cost of implementing a new mode.
- Hardware resource is very poorly used since every new mode requires extra hardware resources which are most often sleeping since only one mode is active at a time.

Such systems can be viewed as a large collection of real time algorithms which are activated (or not) by a function of non predictable events such as the content of the image or an external information or a request from the user.

Consequently, the approach should be to design a single multi-application reconfigurable platform architecture on the top of which a set of different image processing applications can be implemented:

- A set of basic image processing operators is programmed onto the MORPHEUS architecture
- Reconfiguration consists in reconfiguring the chip so as to implement one of these operators
- Reconfiguration is decided by the Molen processor which in fact decides the sequence of operators to be applied to an image

Such an implementation is convenient for a large class of image processing algorithms, namely those algorithms which can be viewed as sequence of operations on an image (or a window within an image).

The typical figures of complexity required by image processing applications are:

- Pixel resolution:
    - 16 bits for monochrome images;
    - 3 x 8 bits for colour images.
- Image size:
    - 768 x 576 for conventional digital TV;
    - 1920 x 1080 for HDTV.
- Frame rate:
    - 25 Hz, 30 Hz, 50 Hz or 60 Hz.
- Input data rate:
    - 11 Mega pixels per second for conventional digital TV;

  - 52 Mega pixels per second for HDTV.
- Processing complexity:
  - 10 to 100 operations per pixel for simple processing;
  - 1000 to 10000 operations per pixel for complex processing.
- Processing power:
  - From 0.1 GOPS to 500 GOPS.
- Throughput data rate
  - depends both on the algorithm and the way it is implemented;
  - twice the input data rate when all the operators can be applied sequentially on each pixel or on each window of an image;
  - 10 to 100 times the input data rate when the full image has to be stored every time a new operator is run.

It is important to note that the upper bound of these figures is clearly beyond the scope of present technology. In other words, the needs are far above what is feasible at a reasonable price. As a consequence, today's requirements usually reflect the limitations of the technology, not the performances wished by the customer. Hence, having the best technology at low price and low power consumption is clearly a competitive advantage in this domain.

Reconfiguration requirements strongly depend on the implementation strategy which itself depends on the performances of the architecture. In this scope, two types of implementation can be proposed. A trade-off has to be made in the sense of the reconfiguration rate, which can be reduced at the expense of the increased computational data throughput and vice versa.

By the first implementation philosophy (referred as a throughput intensive option), the full image is processed through a first operator and the result is stored in an external memory (external due to the size of the image). Then, this result is read from the external memory and processed through a second operator and so on. In this case, for a typical algorithm where 20 successive operators have to be applied:

- The reconfiguration rate is 20 x 25 Hz = 500 reconfigurations / second.
- The throughput is 20 x 2 x 11 Mega pixels / second = 440 Mega pixels / second

By the second implementation philosophy (referred as a reconfiguration intensive option), only a window of the image is processed through a first operator and the result is stored in an internal memory. The size of the window is chosen so that it can be stored inside the internal memory of the chip. Then, this result is read from the internal memory and processed through a second operator and so on. After the window is processed through all the operators, the result is stored in an external memory. Then the next window is processed through all the operators and so on. In this case, for a typical algorithm where 20 successive operators have to be applied and for a chip having an internal memory leading to split the image into 100 windows,

- The reconfiguration rate is 100 x 20 x 25 Hz = 50000 reconfigurations / second.
- The throughput is 2 x 11 Mega pixels / second = 22 Mega pixels / second

It should be noted that for the reconfiguration intensive option, the throughput is very often higher since the windows have to overlap in order to avoid side effects.

For both philosophies, a design goal is to minimize the processing power due to the combined impact of the reconfigurations and the throughput.

3.2 DESIGN CHALLENGES

Unless in some specific and very simple situations, today's reconfigurable computing platforms cannot be used as the sole computing resources in a given system. In general, reconfigurable resources are used in combination with standard computing resources and other devices in a system that resembles the sketch drawn on Fig. 7. The MORPHEUS architecture target, as far as it has to comply with a broad range of applications, is intended to be a complete and heterogeneous platform.

Typically such a platform consists of a hardware system architecture and design tools including methodologies which allow application engineers to utilize the hardware architecture. In order to develop a smart and flexible reconfigurable computing hardware and to increase the efficiency of today's reconfigurable computing systems the following issues have to be considered.

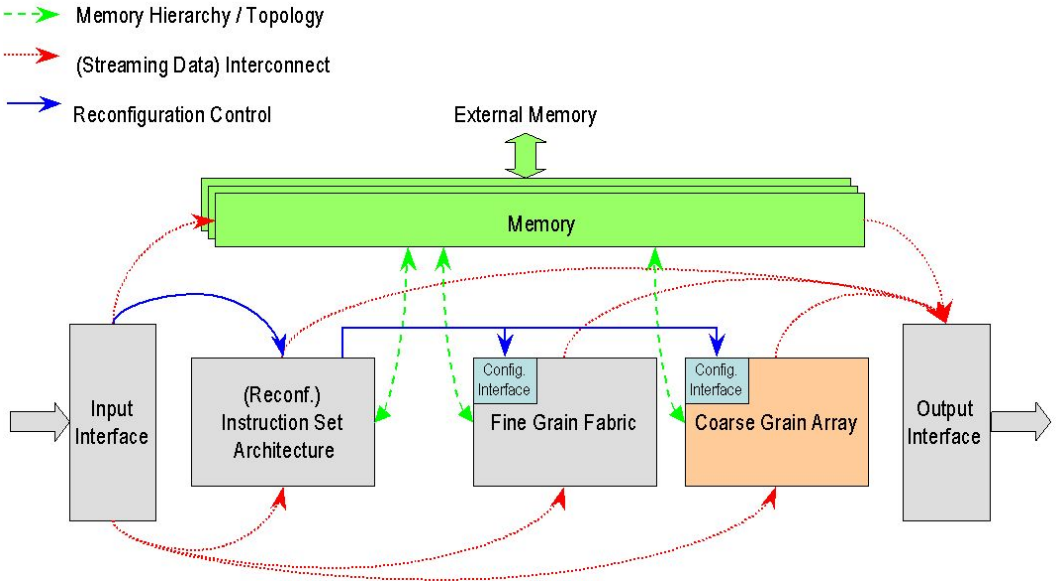


Fig. 7. Conceptual view of the MORPHEUS SoC.

Control and (dynamic) reconfiguration concept. The requirements of embedded computing solutions (cost, mobility, functionalities) are typically translated by designers in area, energy and performances constraints and thus often lead to the specification of dedicated chips. In the same time, the explosion of the cost of development results in the need for flexible architectures taking advantage of high-level programming tools. Static reconfiguration is used to adapt

the architecture to the application. Then, computing resources and communications can be configured according to the application requirements. However, performances, energy constraints and low cost demand a clear breakthrough which can only be achieved through a stronger adaptation of the architecture to the application. For this purpose, dynamic reconfiguration is compulsory. It enables "on the fly" architecture optimization taking into account the current pattern of calculation, to implement either loop kernels, pipeline stages or taking advantage of data locality. Such kind of reconfiguration is only relevant if and only if mechanisms are established to speed the reconfiguration process.

Modularity. Modularity is a key aspect of the MORPHEUS approach. The heterogeneous nature of the architecture gives huge opportunities for scalability and modularity. Another important aspect of modularity is the possibility to easily integrate the scalable and modular block into one architecture. For this reason generic interfaces have to be provided by the modules. It is denoted in Fig. 7 that the definition of interfaces for logical and physical interconnection of the modules integrated into the reconfigurable architecture is one of the main challenges. The link of this modular HW platform with the toolset should be ensured by "tool-interfaces" providing the important aspects and requirements like simulation, debugging, verification, and monitoring.

Architectures for coarse- and fine-grain reconfigurable computing. Coarse-grained reconfigurable architectures fill the gap between General Purpose Processors (GPP), DSPs, fine-grained FPGAs, and specialized hardware (ASICs). Reconfigurable architectures are flexible and provide a high degree of parallelism. They are built from a large number (typically in the range between 10 and 100) of processing elements with ALUs for signal processing algorithms. Applications are mapped for a certain time to the array while data flows through the network of operators (i.e. ALUs). After a certain number of data has been processed, the array can be reconfigured, thus the functionality of the nodes and the interconnection network is changed. This approach is well suited for *streaming* data with limited control flow. The intention of MORPHEUS is to improve the application space towards more control-flow oriented architectures. A more flexible coarse-grained architecture needs to communicate very efficiently with the steering unit –

typically a GPP - and must be integrated with low latency into the memory hierarchy (including dynamic reconfiguration). Coarse-grained architectures are designed for algorithms operating on word-level (e.g. 16 bit). However several algorithms (e.g. entropy encoder in video codecs) demand fine-grained architecture such as FPGAs. MORPHEUS utilizes legacy eFPGAs in the SoC and having efficient interfaces to the coarse-grained architectures. Thus, if the algorithm was properly partitioned each of the architectures can operate in its optimal application space. The benefit is a better ratio of area vs. performance for the overall application without sacrificing flexibility.

Efficient interconnection infrastructure. While fine- as well as coarse-grained IPs have progressed significantly during the last years, the resulting requirements on interconnect in terms of bandwidth, flexibility and efficiency have hardly been targeted by reconfigurable architecture research. Especially the huge opportunities of run time reconfiguration of interconnect are only marginally exploited so far. In order to do so, today's dominant bus architectures need to be extended by reconfigurable high bandwidth point-to-point connections as well as suitable Network-on-Chip (NoC) approaches. The heterogeneous and mixed-grain SoC architecture with its different possibilities to run tasks on the chip and also the flexibility for tasks to be migrated from one architecture tile to another, forces to integrate a high performing and adaptive interconnection infrastructure. For this, a run-time adaptable network with the possibility of changing the topology and protocol, e.g. exploiting also dynamically the trade-offs between packet and circuit-switched communication parts/phases, is necessary. In addition, the connection of the different cores with parallel memory modules has to be considered. To provide a fast data-throughput it has to be enabled, that bottlenecks for parallel memory access and inter-tile communication have to be avoided. To exploit the parallel mixed-grained architecture efficiently it is necessary to integrate more than one memory module connection resulting in determining a suitable trade-off in central/decentral (e.g. global/local) memory access interconnect topologies.

Memory topologies. Since reconfigurable SoCs offer the potential to drastically increase processing power and efficiency especially in data oriented

processing schemes, the bottleneck is passed on to the simultaneously growing requirements on the respective memory infrastructure. Intelligently organized on-chip memories – configured as local memory with user controlled DMA access or as transparent caches – become mandatory, because off-chip solution lack the required bandwidth and are unacceptable regarding power consumption and system costs. In addition, memories in reconfigurable SoCs pose particular challenges to the digital designer, since typical applications often require certain flexible access patterns (e.g. different word sizes, parallel access or different addressing modes). Hence, on-chip memories are probably the most mission critical components of today's embedded signal processing systems. Generalized solutions and methodologies are not yet state of the art. It is one of the goals of MORPHEUS approach to develop such methodologies and to extend the huge opportunities of (dynamic) reconfiguration to the memory infrastructure. This issue will be described in the following sections in more detail.

System integration. The integration of a large number of different units (coarse- and fine-grained reconfigurable units, GPPs, high bandwidth I/O and peripherals) demands the efficient simulation capabilities. Therefore, the silicon-proven IP are delivered with new extensions and test benches. During back-end processing, tight cooperation ensures fast design iteration cycles in case of problems to reach the objectives. Special focus is given to the interconnect between the individual SoC modules (all arrows in Fig. 7) as well as on the topology and respective SoC integration of memories.

3.3 HARDWARE ARCHITECTURE

The MORPHEUS architecture is based on an *ARM9 embedded RISC processor*, which is responsible for data, control and configuration transfers between all resources in the system, memory, I/O peripherals, and a set of *Heterogeneous Reconfigurable Engines* (HREs) each residing in its own clock domain with a programmable clock frequency (see Fig. 8). Each HRE is composed of a reconfigurable IP seen as a memory-mapped co-processor or peripheral. The following three HREs are integrated in the system:

- The PACT XPP is a coarse-grain reconfigurable array primarily targeting algorithms with huge computational demands but mostly deterministic control- and dataflow. Further enhancements based on multiple, instruction set programmable, VLIW controlled cores featuring multiple asynchronously clustered ALUs also allow efficient inherently sequential bitstream-processing.
- The PiCoGA core is a medium-grained reconfigurable array consisting of 4-bit oriented ALUs. Up to four configurations may be kept concurrently in shadow registers. The architecture is mostly targeting instruction level parallelism, which can be automatically extracted from a C-subset language called Griffy-C.
- The M2000 is a lookup table based, fine grain reconfigurable device – also known as embedded Field Programmable Gate Array (eFPGA). As any FPGA, it is capable to map arbitrary logic up to a certain complexity provided the register and memory resources are matching the specifics of the implemented logic. The M2000 may be scaled over a wide range of parameters. The internals of a reconfigurable logic block may be modified to a certain degree according to the requirements. Flexibility demands may favor the implementation of multiple smaller M2000 eFPGAs instead of a single large IP.

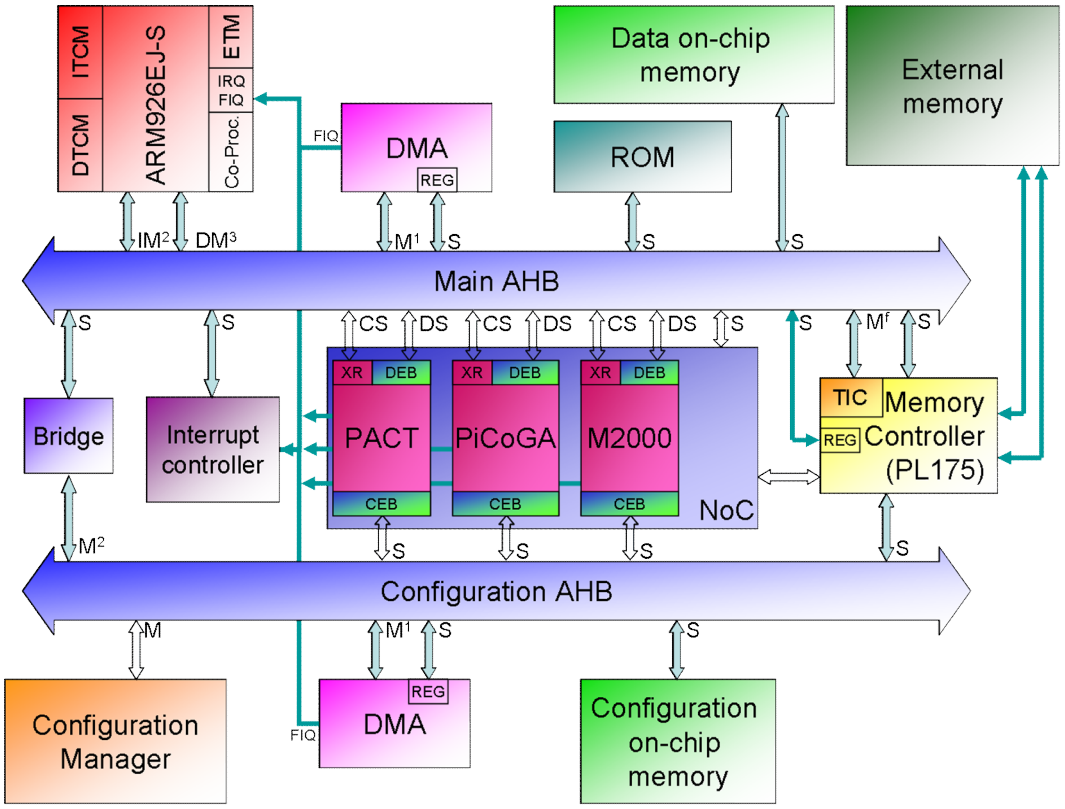


Fig. 8. MORPHEUS block diagram.

All control, synchronization and housekeeping is handled by an ARM9 processor. As dynamic reconfiguration imposes a significant performance demand for the ARM processor, a dedicated reconfiguration control unit provides a respective offload capability. All system modules are interconnected via multilayer AMBA busses. Separate busses are implemented for reconfiguration and data/control access. As the required bandwidth for high performance data intensive processing is quite huge, a circuit switched Network-on-Chip (NoC) is implemented which – with regards to the implementation – avoids the disadvantages of wide conventional bus systems. As NoCs imply a significant implementation risk, the AMBA busses serve as proven fallback solution.

As the HREs operate on differing clock domains, they are decoupled from the system clock domain by *Data Exchange Buffers* (DEB) consisting of dual-port dual-clock memories either configured as FIFOs or *ping-pong buffers* [5]. The HREs have access to further on-chip SRAMs for buffering of local data. These SRAMs may be either used as cache or scratchpad RAM. A state of the art multi-channel SRAM/DRAM controller provides access to external system memories

(volatile or non-volatile). Standard DMAs are used in order to increase the utilization of the bus architecture and further improve data transfer bandwidth between on-chip resources. All data transfers between HREs, on- and off-chip memories may be either HRE triggered or managed by a DMA control unit. Furthermore, a set of standard I/O peripherals (UART, USB, timers, I2C etc.) is provided. Configuration and task triggering between the ARM9 core and HREs is performed according to the *Molen paradigm* [74], through specific *eXchange Registers* (XRs).

3.4 COMMUNICATION INFRASTRUCTURE

Network-on-Chip (NoC) concept is used as communication means for the MORPHEUS platform when aiming at the top performance. STNoC [70] is a specific approach to implement a generic NoC technology. It contains three different types of building blocks, appropriately interconnected to each other, and a patented network topology that promises to deliver the best price/performance trade-off for the future Multi-Processor System-on-Chip (MPSoC) applications. STNoC technology comprises packet-based communications protocol and delivers significant advantages to system designers, leveraging a powerful Quality of Service (QoS) support, without having to evaluate different network topologies for each application.

The key building blocks are:

- The network interface (NI), which connects individual IP blocks or subsystems to the on-chip network;
- The Router, which is responsible for the data transfer across the network and for the QoS offered by the network;
- The physical link, which is responsible for the actual propagation of the signals across the network and to/from the external IPs and subsystems.

Fig. 9 represents how the three STNoC building-blocks are related to the ISO-OSI protocol layers. At the lowest level, the STNoC physical link implements the physical layer of the NoC protocol. It is responsible for the connections between routers and between routers and NIs. There are several possible ways of implementing physical links, including all permutations of synchronous/asynchronous and serial/parallel links. Of course, the choice of an appropriate physical link technology involves trade-offs between mutiple issues such as clock distribution over a wide silicon area, amount of on-chip wiring and the chip area required. In this respect, the decoupling of layers provided by the NoC paradigm is a major advantage, as changes to the physical layer can be subsequently made without affecting the packet transport and transaction layers [71].

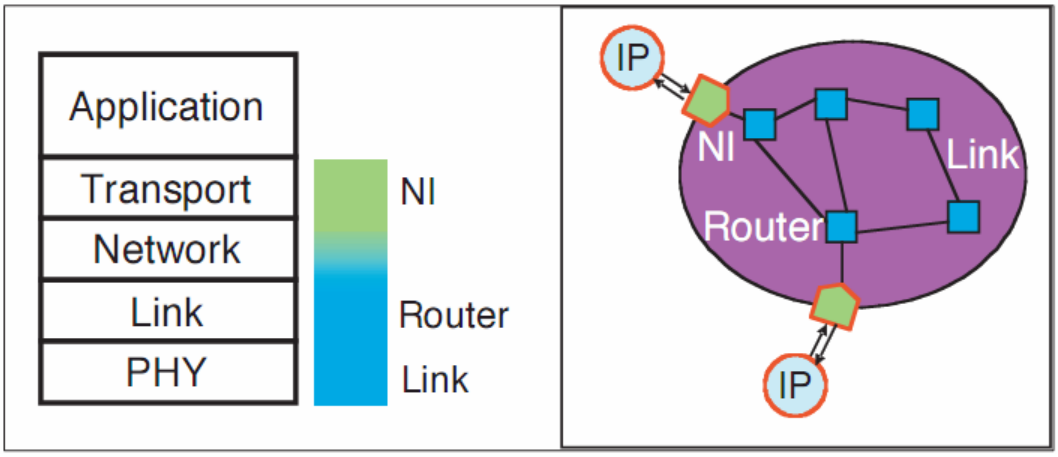


Fig. 9. STNoC implementation of the ISO-OSI protocol layers.

The NI is the access point to the NoC, converting the transactions generated by the IP or the subsystem connected to it into packets that are transported over the network. The NI hides network-dependent aspects, allowing IP blocks to be reused without further modification no matter how the NoC architecture subsequently evolves. This is a crucial benefit in terms of MPSoC design time. The NI is also responsible for performing size conversion when the IP or the subsystem have a different data bus size compared to the NoC, and frequency conversion when the IP or the subsystem work at a different clock frequency than the NoC.

The STNoC Router implements both the network and the data link layers of the NoC protocol, offering "best effort" delivery (where responsibility for recovering lost or corrupted packets lies with the intended receiver of the packet rather than with the network) and also the possibility to provide QoS in terms of latency and throughput. It is responsible for the transmission of the *flits* – the elements into which packets are logically divided. The STNoC router is designed to support ST's proprietary *Spidergon* topology (Fig. 10, [70]), therefore it is capable of routing packets via three different links: left, right and across. However, depending on the application traffic requirements, links can be removed, which gives STNoC the key benefit of being able to support with one homogeneous component all the possible topologies from the simple ring to the *Spidergon*.

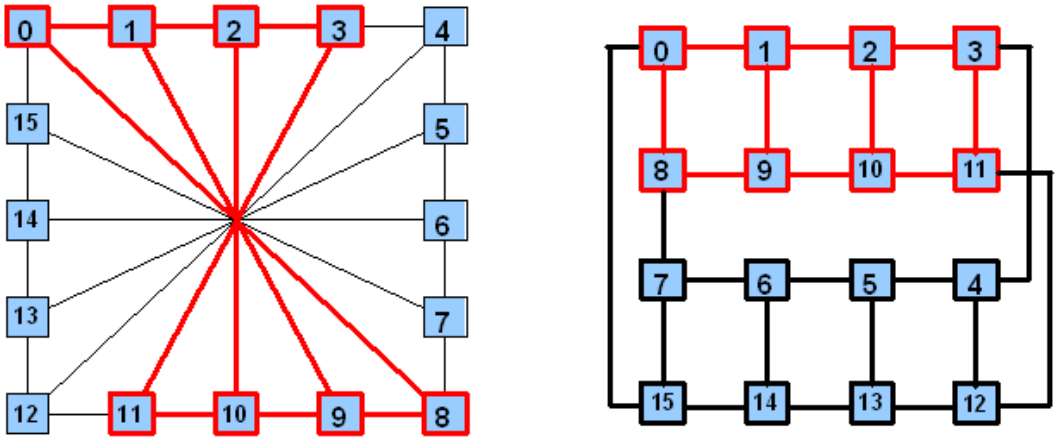


Fig. 10. Spidergon topology.

3.5 DESCRIPTION OF THE IPS

3.5.1 PACT XPP

The XPP array is a coarse-grained reconfigurable tile, specialized on data flow type of algorithms. Compared to fine-grained reconfigurable architectures, the coarse-grained ones consume less power and need smaller configuration stream, which allows more flexible dynamic reconfiguration. With respect to instruction-set processors, coarse-grained reconfigurable architectures don't require a new instruction every clock cycle. Several coarse-grained reconfigurable architectures have been developed in academia and industry. Two of them are considered in MORPHEUS.

The XPP architecture provides parallel processing power combined with fast reconfiguration. The new version which is named XPP-III is currently under development and will be adapted to meet the MORPHEUS requirements. XPP-III integrates the new Function PAEs (FNC-PAE) which extend the application space of the XPP also towards high performance control flow oriented applications.

XPP is a coarse-grained scalable architecture designed not only to provide maximum performance combined with low power consumption but also to simplify algorithm design and programming tasks. The XPP can process both basic categories of application software: data-flow oriented sections and control-flow oriented sections. The sections are handled by two basic types of processing resources:

1. The reconfigurable course grained XPP-array processes the data-flow sections of the application:
    a. Configurable Processing Array Elements (ALU-PAEs and RAM-PAEs) are arranged in an array and communicate via point-to point communication links
    b. Programs are mapped as flow graphs to the array of ALUs and RAMs
    c. Communication is packet-oriented with auto-synchronization
    d. Control of programs is handled by an independent event network
    e. The array provides fast dynamic reconfiguration

f. I/O supports streaming and memory mapped I/O.

2. The FNC-PAEs process the Control-Flow sections of the application:
   a. VLIW-type PAEs are tightly integrated into the XPP-array
   b. Data exchange with the XPP-array is data-flow synchronized
   c. The FNC-PAEs may steer the reconfiguration sequencing of the XPP-array
   d. FNC-PAEs I/O may use the XPP-array streaming I/O and shared external memory.

Fig. 11 shows an array with 5 x 8 ALU-PAEs, 2x8 RAM-PAEs and 8 FNC-PAEs. The array-I/O is integrated in the RAM-PAEs at the four corners of the array. In the following sections the fabric which is built from RAM-PAEs and ALU-PAEs is named the "XPP-array". The size of the array is scalable in X and Y direction.

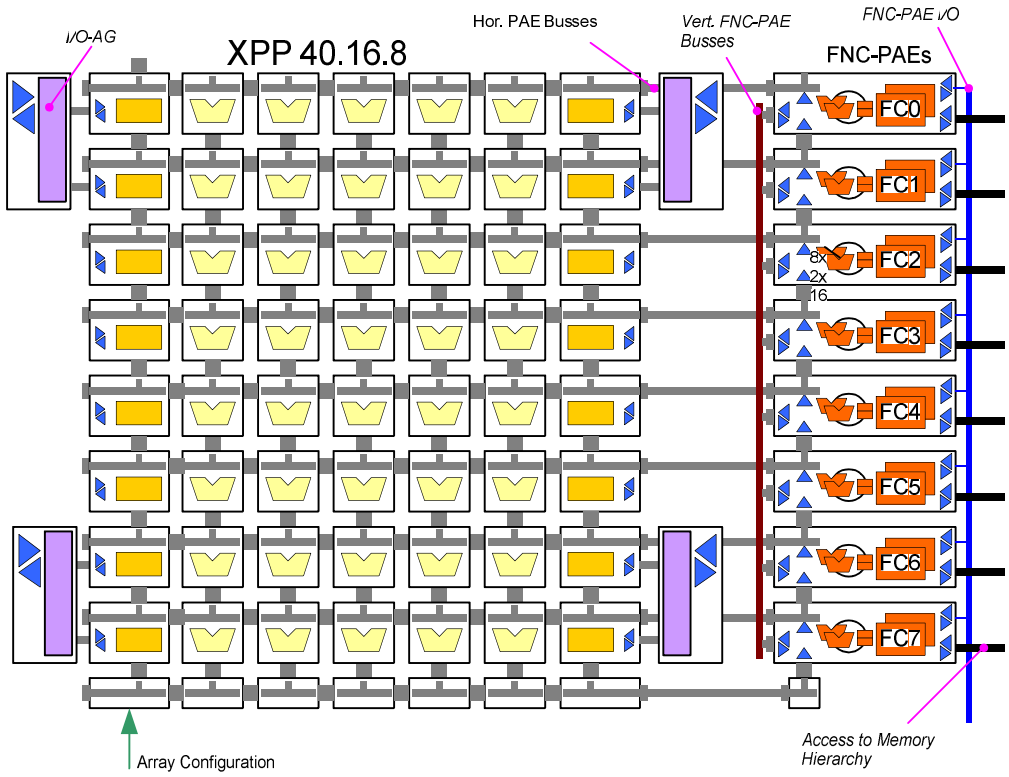


Fig. 11. An XPP array with 6x5 ALU-PAEs.

Arithmetic and logical operations are executed in the ALU-PAEs; data can be stored locally in the RAM-PAEs. Communication is done by transmission of

data packets[1] through the configured communication network. A configuration specifies the communication paths between the PAEs the function of the ALUs and initial values of registers and RAMs. The configuration is not changed as long as data flows through the network. Data I/O to the array is performed by means of the ports at the corners of the array. The FNC-PAEs may access the outside world via direct access to the external memory hierarchy or through the streaming ports.

The algorithm is defined by means of a flow graph, which is statically mapped (spatial mapping) onto the array during one configuration.

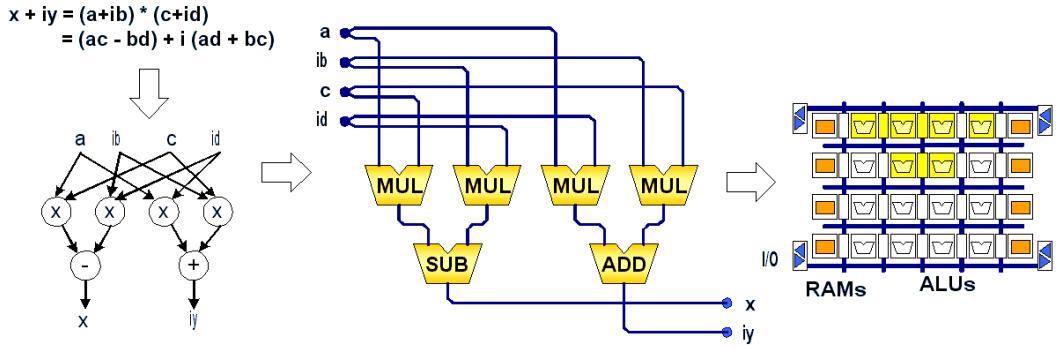


Fig. 12. Flow-graph of a complex multiplication and spatial mapping.

Fig. 12 shows the flow-graph of a complex multiplication. With XPP, each operator (MULT, ADD, SUB) is mapped onto an ALU-PAE and the connections between the PAEs are statically wired. Data flows pipelined through this network, which is not changed until a certain amount of data has been processed and - optionally - has been buffered in the RAM. After execution, the PAEs are released and can be used for the next configuration, which performs the next step of the computation.

This strategy is efficient for algorithms, where a large number of data must be processed in a relatively uniform way. Since the reconfiguration of the array requires several hundred clock cycles and extra energy, a single configuration should be active for a certain amount of processed data. Most multimedia and wireless applications process data streams and require lots of processing power exactly for this type of algorithms.

[1] A *data packet* is a single 16-bit word

In XPP, a data stream is a sequence of single data packets traveling through the flow-graph that defines the algorithm. A data packet is a single machine word (e.g. 16 or 24 bit). Streams can, for example, originate from natural streaming sources such as A/D converters. When data is located in a RAM, the XPP may generate packets that address the RAM producing a data stream of the addressed RAM-content. Similarly, calculated data can be sent to streaming destinations, such as D/A converters or to integrated or external RAMs.

In addition to data packets, state information packets are transmitted via an independent event network. Event packets contain one bit of information and are used to control the execution of the processing nodes and may synchronize external devices.

The XPP network enables automatic synchronization of packets. An object (e.g. ALU) operates and produces an output-packet only when all input data and event packets are available. The benefit of this auto-synchronizing network is that only the number and order of packets traveling through a graph is important – there is no need for the programmer or compiler to care about absolute timing of the pipelines during operation. This hardware feature provides an important abstraction layer allowing compilers to effectively map programs to the array.

XPP-arrays interface to external devices and the FNC-PAEs with:

- Data streaming channels with one processor word by means of a hardware handshake protocol that maintains the stream-synchronization capabilities also to the outside world (i.e. SoC Busses, AMBA, NoC and FNC-PAEs).
- The array I/O interface can alternatively be configured to provide addresses and data for connection to external RAMs (not to FNC-PAEs)
- Event streaming ports transfer one-bit information similarly to the data channels
- The Reconfiguration Port provides a streaming interface that allows sequential loading of configuration into the array. Typically an external DMA controller may performs this task.

The Function PAEs (FNC-PAE) which are tightly coupled to the reconfigurable XPP-array are sequential 16-bit cores which are optimized for algorithms requiring a large amount of conditions and branches. One FNC-PAE

comprises two columns of four small non-pipelined 16-bit ALUs[1]. This is on the first view similar to VLIW DSPs. However there are substantial differences which enhance the condition and branch performance. First of all, any ALU can access results of the rows above and the register file within a single clock cycle. Based on results, subsequent ALUs in a column can be disabled conditionally. This allows conditional operations and branching to different targets to be evaluated within the current clock cycle. In parallel, the Special Function Unit (SFU) comprises a parallel multiplier and bit-field operations. Code is stored in a small local associative Instruction Cache. Data is stored in a fast tightly coupled local RAM and the large external System RAM[2]. Both are accessed through a 32-bit address generator (AG) comprising stack and pointer arithmetic.

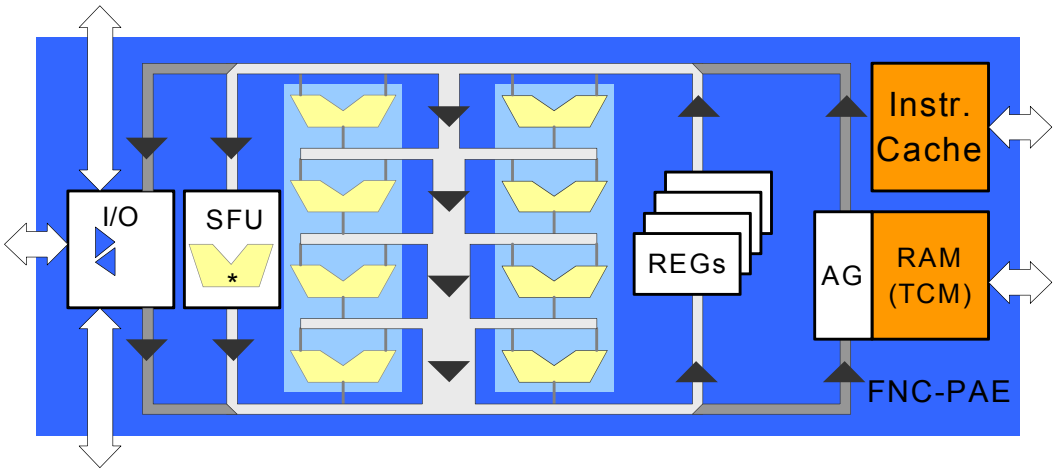


Fig. 13. FNC-PAE block diagram.

The communication with the XPP-array (Fig. 13 left ports) is data flow synchronized: a port suspends its operation until data can be transferred. Thus programs running on the XPP array and the FNC-PAEs are implicitly data synchronized. Furthermore, FNC-PAEs may exchange data through vertical data flow busses. Synchronization on operating system level (e.g. loading a new XPP configuration) can be achieved with XPP events and FNC-PAE interrupts.

Due to the fact that XPP array is not a standard sequential processor and also not a fine-grained FPGA, specialized development tools are provided. A tool

[1] ALU operations: boolean add/sub, barrel shift, branching etc.

[2] The System RAM is SoC specific and shared by the Function PAEs.

suite is available which allows describing the algorithm as flow graph. The tools feature automatic place and route, clock accurate simulation and an API that allows the integration into System-C based simulations. A vectorizing C-compiler simplifies porting of sequential algorithms to the XPP array.

The FNC-PAEs can be programmed in assembler language and/or with ANSI C. The tools provide co-simulation and debugging features for programs utilizing both, the XPP-array and programs running on several Function PAEs. The simulation is cycle accurate within the XPP-array. Access to the external memory hierarchy which is required for the FNC-PAEs is performed by means of a simplified memory model.

The FNC-local memories D-MEM and I-MEM provide the first level of the memory hierarchy. Time critical sections of algorithms should be executed using only those local resources. The I-MEM is organized as a 4-way set associative cache (4 x 64 * 256 bit). The sets can be locked and prefetched under program control. The D-MEM is organized as a linear 1024 x16 bits.

Since several FNC-PAEs will access the memory, an arbiter is required. However, most inner loops will be executed from the local I-MEM, thus only minimal external code access is expected. Local variables should be stored in the local D-MEM.

3.5.2 M2000 EMBEDDED FPGA

FlexEOS macros are SRAM-based, re-programmable logic cores to be integrated into SoC designs. The logic function of the core can be re-configured simply by downloading a new bitstream file. FlexEOS is available in different capacities and multiple macro instances can be implemented in one device to achieve the required configurability while accommodating area and performance constraints.

3.5.2.1 OVERVIEW

A FlexEOS macro is a FPGA to be embedded in a SoC design. The FlexEOS package contains a hard macro of the FPGA core, plus the software necessary to configure the FPGA core with the required functionality. Each FlexEOS package contains the following items:

- A hard macro, the so-called macro core, which is the actual re-configurable core to be included in a SoC design.
- A soft block which is the synthesizable RTL description of the 'Loader', a controller which manages the interface between the macro core and rest of the SoC. Multiple macro instances in one device require multiple Loaders, one per macro. The main functions of the Loader are to:
    - load the configuration bitstream, and verify its integrity at any time
    - simplify the silicon test procedure
- A software tool suite to create
    - files required during the integration of the macro into the SoC design,
    - a bitstream file to configure the hard macro for a particular application.

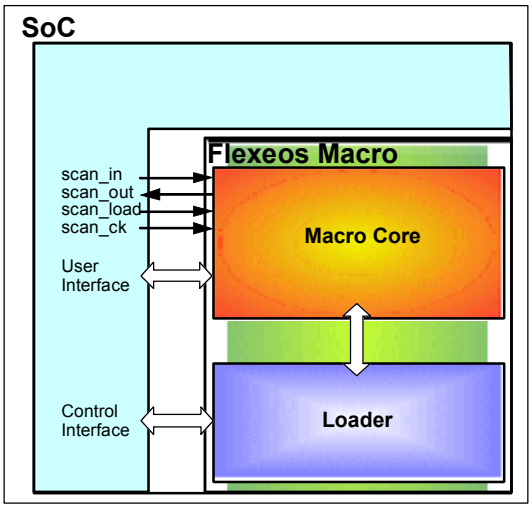


Fig. 14. FlexEOS macro block diagram.

Fig. 14 shows a block diagram of a FlexEOS macro when embedded in a SoC, and its interfaces to the rest of the system. It has to be noted that each FlexEOS macro contains a macro core and a Loader. Furthermore, the control interface in Fig. 14 is only used for accessing the system functions of the FlexEOS macro, i.e. for writing commands and configuration words to the Loader and reading back status information from the macro core. The user interface signals correspond to the macro core input and output signals, and are the only ports which can be instantiated by a design mapped into the core during run-time.

The FlexEOS macro is a LUT-based FPGA technology which needs to be re-configured with a design each time the power is turned on, or each time the application requires a change of its functionality. The Loader ensures the proper loading of a configuration bitstream. Its design is optimized to simplify the interactions between the rest of the SoC and the macro core, and to allow predictable and reliable control of the core configuration and operation modes. It verifies the integrity of the bitstream while it is being loaded by computing a CRC signature which is checked against a reference CRC previously calculated by the FlexEOS compilation software. The CRC signature of the loaded configuration is also continuously computed when the application is running, so that if an error occurs in the eFPGA configuration, the SoC controller can be interrupted to reload the bitstream and re-initialize the related system functions. The time required for a CRC signature computation is about 2 ms for a 4K-MFC macro, depending on the Loader clock frequency.

A typical example for a bitstream corruption during application run-time is a software error. Thereby, one or more configuration memory bit-cells may switch to their respective opposite value due to surrounding noise. The functionality mapped to the eFPGA is then modified and not predictable.

In addition to handling the configuration, the Loader includes specific functions which speed up the silicon test time. The FlexEOS architecture is highly parallel, so only a minimal set of configuration and test vectors are needed to test each unique internal structure. The Loader uses this information to test any similar structure by simultaneously replicating a basic set of configuration and test vectors for the whole core. It then analyzes the result of all the tests in parallel and stores the result in its own status register. The external controller, which in this case should be the tester, can read this status register back at the end of each test sequence to find out if it failed or passed.

The Loader is presented as a synthesizable VHDL design, which requires between 10K and 20K ASIC gates, depending on the customer implementation flow and target manufacturing technology. Its typical operating frequency is 100MHz and below.

3.5.2.2 ARCHITECTURE

FlexEOS uses a highly scalable architecture which permits gate capacities from a few thousands to multiple millions. The basic FlexEOS macro includes 4096 MFCs (Multi-Function logic Cells). Several macros can be unified in more complex structure. Furthermore, it also includes the following:

- 8 DPRAM blocks (either 4K bits or 8K bits)
- 32 MACs
- 128 x 8 bit adders

The basic FlexEOS building block is the MFC which is a programmable structure with seven inputs and one output. It combines a four-input LUT (Look-Up Table) and a D flip-flop (see Fig. 15).

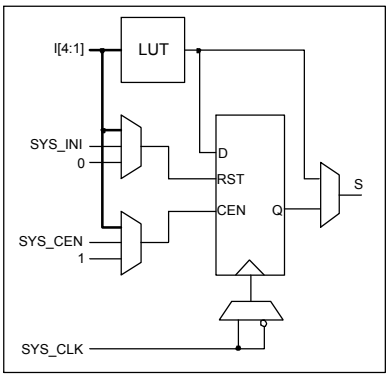


Fig. 15. MFC schematic.

The storage element has clock, clock enable, and reset input signals. The clock signal always comes from the system clock tree, and can be inverted, whereas the clock enable and reset signals can either come from the interconnect network via a regular signal input or from the system interconnect network. The FlexEOS compilation software selects the appropriate source according to the nature of the design to be implemented.

The MFCs are organized in groups of 16 units and are all located at one hierarchical level in the core architecture. A FlexEOS macro with 4K MFCs has an equivalent ASIC gate capacity of up to 40,000 gates. The design configuration file (bitstream) size is 36Kbytes, and the loading time is around the range of 600μs when the FlexEOS Loader operates at 100 MHz. The data bus interface is 32-bits wide.

Most control designs and all signal processing designs use classic arithmetic operators such as add, subtract, increment, decrement, equal to, inferior to and superior to. By default, they can be mapped to classic structures such as "carry propagate" or "carry look-ahead". The first is more compact and uses fewer MFCs, whereas the second shows better timing performance but poor MFC mapping efficiency. In many cases, the carry chains are part of longer logic paths (critical paths), which results in slower maximum operating frequency for the whole design, especially if the chain is 8+ bits long.

The FlexEOS architecture can optionally include optimized 8-bit carry-chain operators (one per group of MFCs). They provide:

- better timing performance (comparable to ASIC design),
- optimal mapping efficiency (requires one MFC per operator bit)

It has to be noted that a partial utilization of a carry-chain block is possible. Thereby, the range from 1 to 8 bits can be used, while the others are ignored and not connected to the interconnect network. Furthermore, carry-chain blocks are automatically chained by the FlexEOS compilation software using dedicated interconnect resources located between the blocks. As a consequence, the timing delay remains minimal and optimal.

Third party FPGA synthesis software can automatically infer the carry chains with the proper functionality from an RTL description. Nevertheless, the designer can manually instantiate such operators if necessary.

Two sizes of synchronous true dual-port RAM block are available for FlexEOS cores:

- 4K-bit block,
- 8K-bit block.

As shown in Fig. 16, each port has its own control signals (clock, enable, write) so that it can be read or written independently from the other port at anytime. This means that the ports operate asynchronously from each other. The input and output data bus width must be the same for a given port, but can be different from the other port (see Table 1 for the different options depending on the memory block size).

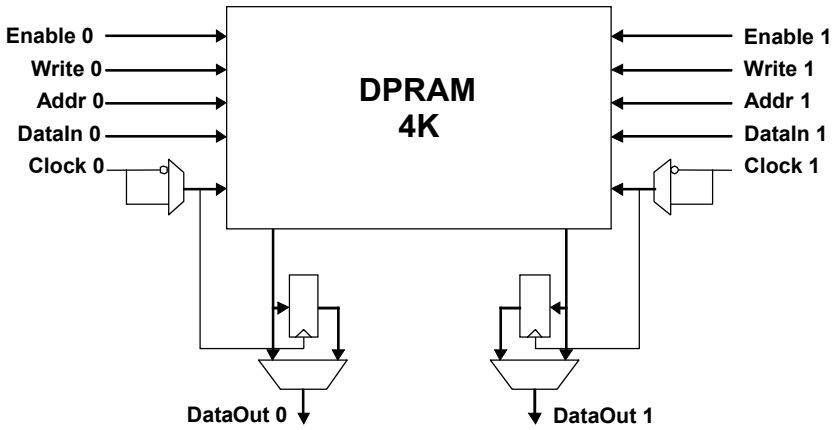


Fig. 16. Embedded DPRAM schematic.

Furthermore, Fig. 16 shows two other configuration options available to the designer:

- Data output on positive or negative clock edge.
- Registered or non-registered data output.

There are two ways to select these options:

- by modifying the description of the input design, assuming the RTL synthesis software can recognize it correctly
- by manually instantiating the proper memory block model in the RTL code.

Each port can be independently clocked and independently controlled. They can be configured as shown in Table 1. Table 2 lists the signals available for each port.

Table 1. eDRAM size and configuration options.

4K		*8K*	
256 words	x 16 bits	512 words	x 16 bits
512 words	x 8 bits	1024 words	x 8 bits
1024 words	x 4 bits	2048 words	x 4 bits

Table 2. DPRAM interface signals.

Control Signal	*Function*
Input Data (DataIn)	Data input bits
Output Data (DataOut)	Data output bits
Clock	Clock, active on the rising edge
Enable	Block validation, active high
Address (Addr)	Word selector
Write	Write access, active high.

Each port features four configuration memory cells:

- two for SRAM depth and word width,
- one for clock polarity,
- one for output mode (pipelined or not).

The MAC block is a basic multiply/accumulate operator with the following features:

- 16x16-bit signed/unsigned multiplier with registered/non-registered inputs
- 32-bit adder
- 32-bit accumulation register
- 32-bit registered/non-registered input to the adder if the accumulator feedback loop is not used
- synchronous reset in accumulation mode.

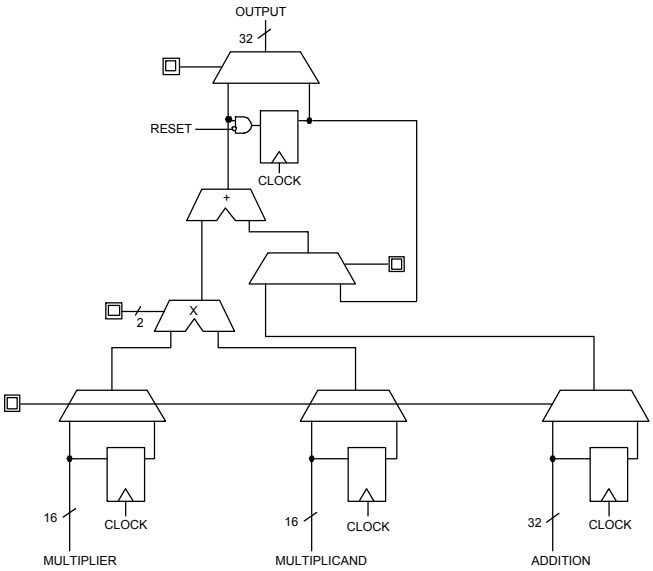


Fig. 17. MAC schematic.

As shown in Fig. 17, the output accumulation register can be bypassed in order to connect the adder output directly to the MAC output bus. It has to be pointed out that only the accumulation register is connected to the reset signal.

FlexEOS eFPGA technology is based on a multi-level, hierarchical interconnect network which is a key differentiation factor in terms of density and performance when compared to other LUT-based FPGA technologies. The interconnect resources are based on a full crossbar switch concept (see Fig. 18), which provides equivalent routing properties to any element inside the macro and gives more freedom for placing and routing a given design to the FlexEOS

compilation software. The interconnect network can only be configured statically, meaning that the clock must be stopped.

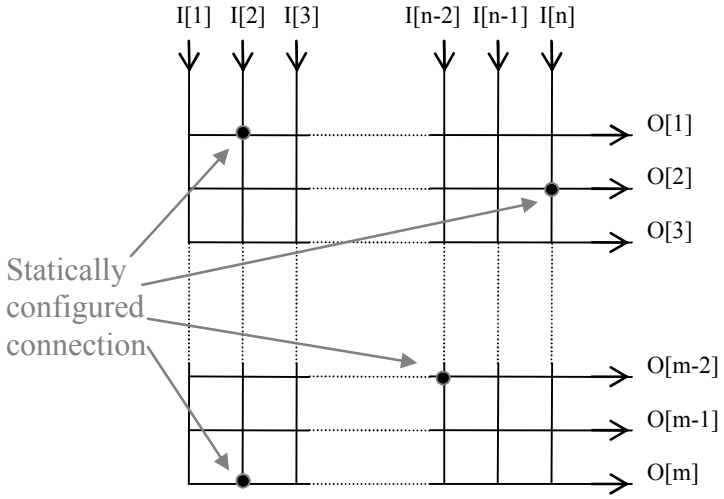


Fig. 18. Full crossbar switch.

Fig. 19 shows the organization of the macro with the different building blocks. It also shows the symmetry of the architecture which provides more flexibility for mapping and placing a design. Each computing element of the macro can either be connected to its neighbor by using a local interconnect resource, or to another element via several interconnect resources.

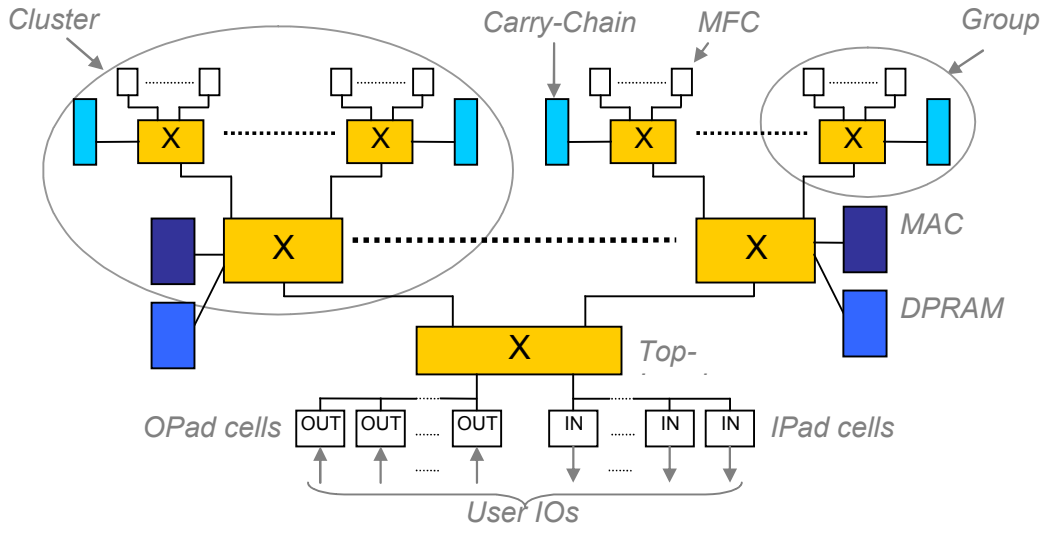


Fig. 19. FlexEOS core architecture.

In addition to the regular interconnect network, a low-skew low-insertion-delay buffer tree network (system interconnect network) starts from 8 dedicated

user input ports (SYS_IN) and connects to all the synchronous cells. Its usage is recommended for high fan-out signals such as reset signals, or high speed signals such as clock signals. If parts of the system interconnect network is not used by the design, the FlexEOS compilation software automatically uses portions of it to improve the final design mapping and performance.

At any level of the hierarchy, the interconnect resources are unidirectional, including the user I/O interface signals. The standard 4K-MFC macro block includes 512 input ports and 512 output ports. Each of them is connected in the same way to the interconnect network, which gives the following properties:

- Any input port can access a given computing resource inside the core
- Any input port can be used as a system signal such as clock or reset
- Any output port can be reached by a computing resource

These three points are meaningful when considering the integration of the eFPGA macro into a SoC architecture and defining the physical implementation constraints.

During the SoC design phase, several potential applications should be mapped to the eFPGA to:

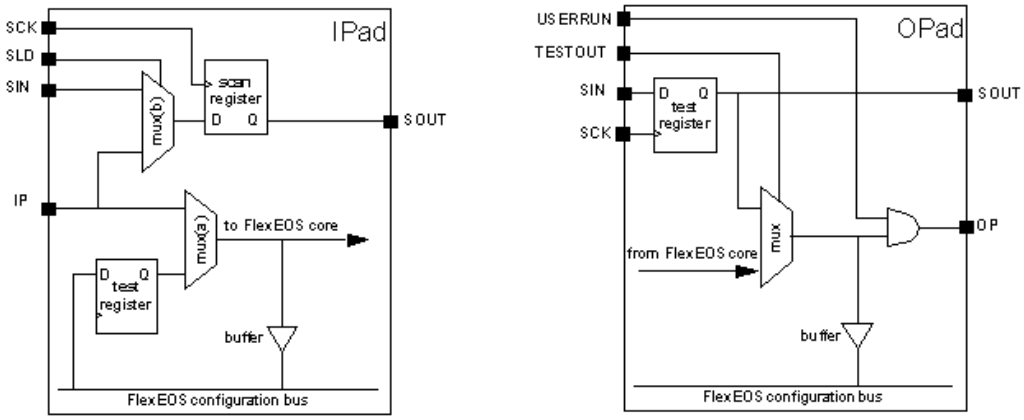
- Evaluate the system constraints of the IP
- Refine the different parameters of the IP (number of MFCs and I/Os, need for carry chains, memory blocks, MACs)
- Evaluate its connectivity to the rest of the system. This is made easier by the flexibility of the eFPGA interconnect network and its I/O port properties: the FlexEOS macro does not add any routing constraints on SoC signals connected to the user I/Os as they can reach any resource inside the macro core.

The core I/O cells are connected together internally to form 2 boundary scan chains:

- one for the input ports;
- one for the output ports.

They can be included in the SoC scan chains when implementing the chip to test the random logic connected the macro core I/Os. The boundary scan chain models are delivered as VHDL files and are compatible with standard Automatic

Test Pattern Generator (ATPG) tools. Fig. 20 shows an IPad cell and OPad cell, with their respective scan logic and path. The usual scan interface signals are available for a seamless scan insertion. When the scan test mode is enabled, the macro core is completely isolated from the rest of the SoC.



a) IPad b) OPad

Fig. 20. IPad and OPad cell with scan logic.

The Control Interface bus is directly connected to the FlexEOS Loader (see Fig. 14). It is used to access the internal Loader resources for test purposes and bitstream loading. This interface behaves similarly to a synchronous SRAM block. It comprises the following signals (see Fig. 21):

- Clock (100MHz and below)
- Reset (active high), needs to be activated at power-on to reset the Loader and the core
- Data In (usually 32 bits, depending on the system bus width)
- Data Out (usually 32 bits, depending on the system bus width)
- Address (4 bits)
- Chip Select (active high)
- Write Enable (active high)
- Busy (active high)
- Done (active high)

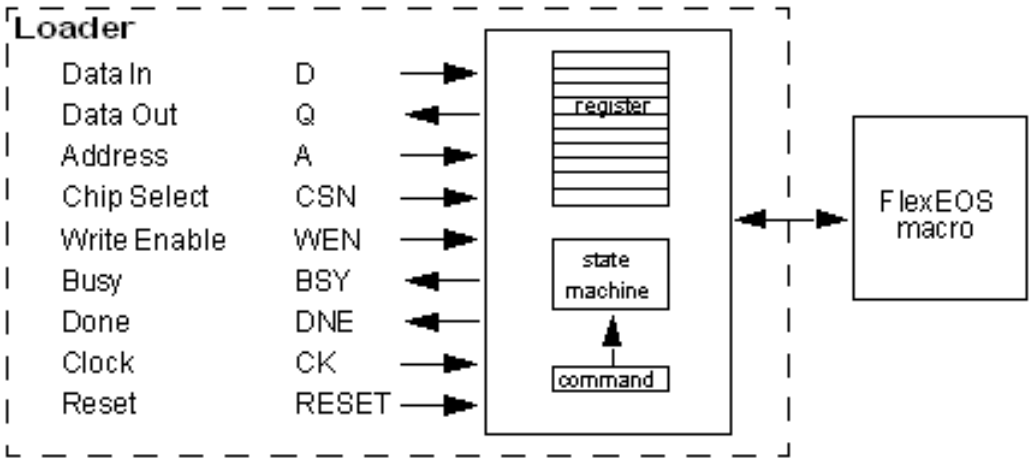


Fig. 21. FlexEOS Loader overview.

A typical operation starts by writing a command and data to the appropriate registers. The state machine then executes the command, and sets the Busy signal to high. When the operation has completed, the Busy signal goes to low, and a subsequent command can be executed. This is illustrated in the timing diagram depicted in Fig. 22. The eFPGA macro, together with its Loader, can be implemented multiple times on the chip, connecting to the system and/or peripheral busses.

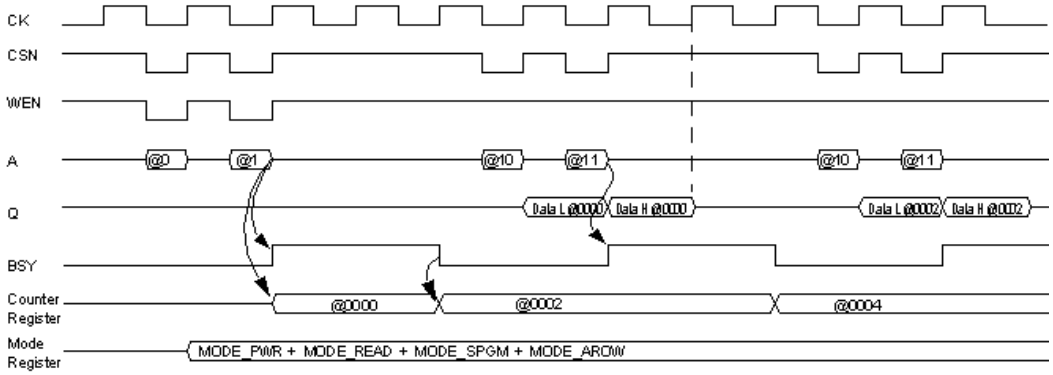


Fig. 22. FlexEOS Loader control interface waveform.

3.5.2.3 SIZE AND TECHNOLOGY

Table 3 shows the dimensions of a 4K FlexEOS macro in 90nm CMOS technology with 7 metal layers.

Table 3. FlexEOS 4K-MFC features and size.

Equivalent ASIC gates:	40,000 (estimated when considering MFCs only)
LUTs/DFFs (MFCs):	4096
I/Os:	504 x IN, 512 x OUT, 8 x SYS_IN
Silicon area for 4K MFCs only:	2.97 mm2 (CMOS 90nm)
Size of bitstream configuration file:	36 Kbytes (4K-MFC only block)
Silicon area for 4K MFCs + 8 x 8Kbytes RAM + 32 MACs + 128 x 8-bit carry-chains:	4.5mm2 (CMOS 90nm)
Size of bitstream configuration file:	Apx. 60 KBytes (4K-MFC + features)

Table 4 shows several design examples mapped onto the FlexEOS eFPGA macros. It also provides the correspondence between the ASIC gate count derived from Synopsys Design Compiler and the MFC capacity required mapping the same designs onto a FlexEOS macro.

Table 4. Example of design mapping results.

	ASIC Gates	*Equivalent MFCs (LUT + FF)*	*FlexEOS eFPGA macro size granularity*
160 x 16 bit counters	29742	3982	4096 MFCs
UART 16550	8096	1459	1536 MFCs
Viterbi Decoder	10028	2245	3072 MFCs
Dynamic synchronous cross-bar bus	5788	1431	1536 MFCs
Ethernet MAC	20587	3995	4096 MFCs

It should be highlighted that FlexEOS macros can be ported to any standard CMOS process. Even multiple identical macros can be implemented in one SoC.

3.5.2.4 FLEXEOS SOFTWARE TOOL SUITE

The FlexEOS proprietary software tool suite provides a design flow which is complete, easy to use, and designed to interface with the main standard FPGA synthesis software packages. It takes the following files as inputs:

- A design structural netlist (mapped to DFFs, LUTs and optional blocks such as memory, MAC and carry-chains), generated by FPGA synthesis software

- An I/O pin assignment file, i.e. assignment of specific input or output I/O cells to each input and output port of the design
- The design constraints such as clock definition, input and output timing delays, false path.

The FlexEOS compilation software provides implementation options such as timing-driven place-and-route, automatic design constraint generation (very useful the first time a design is mapped).

The output files are:

- The configuration bitstream to be loaded in the eFPGA core
- A configuration bitstream reference signature to be provided to the Loader
- A functional Verilog netlist for post-implementation simulation
- Timing annotation file (SDF: Standard Delay File) to perform further timing analysis on a given mapped design with third party software, or to run back-annotated simulation when used in combination with the generated Verilog netlist
- Simple timing report for each clock domain critical path for a pre-selected corner (Best, Typical or Worst case)
- Macro wrapper (Verilog file) which instantiates the mapped design and connects its I/O ports to the physical core ports. This file is useful for in-context (i.e. in the SoC environment) timing analysis or simulation of applications

The FlexEOS software flow is illustrated in the following diagram. The RTL front-end design tasks are executed using commercial tools such as Mentor Graphics Precision RTL or Synplicity Synplify CLS (Custom LUT Synthesis).

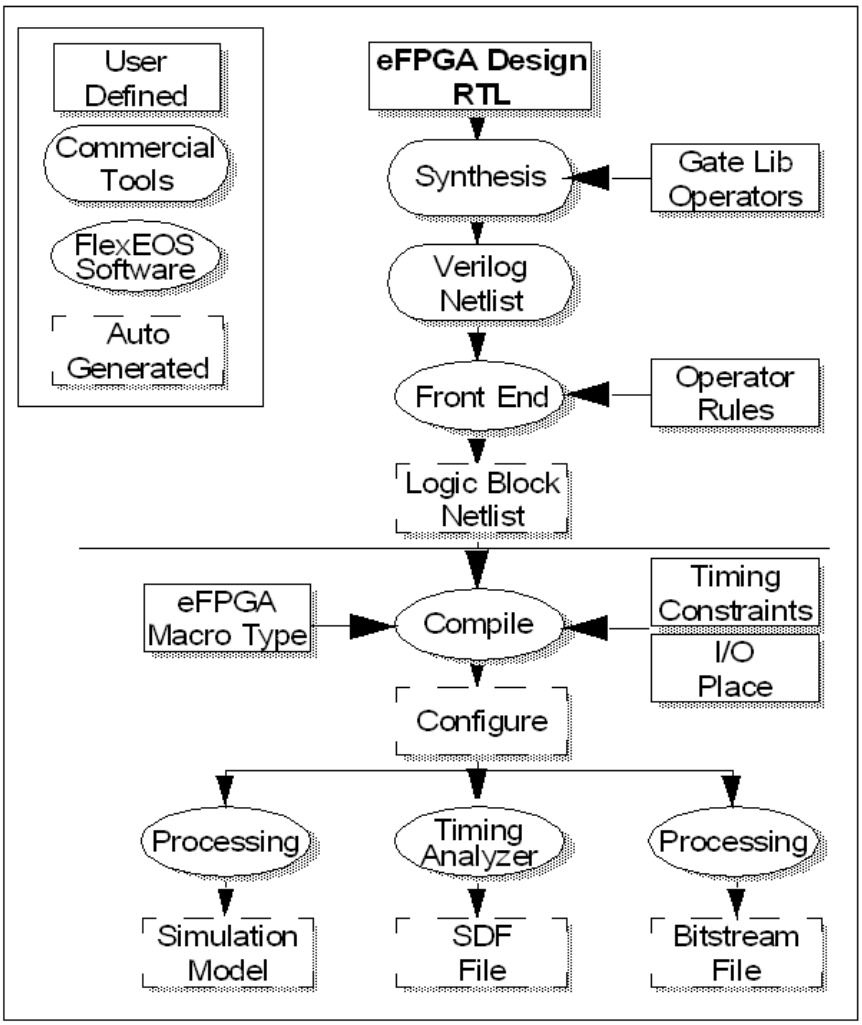


Fig. 23. FlexEOS software flow.

3.5.3 PICOGA

The PiCoGA, Pipelined Configurable Gate Array, is a programmable gate array especially designed to implement high-performance algorithms described in C language. The focus of the PiCoGA is to exploit the Instruction Level Parallelism (ILP) present in the innermost loops of a wide spectrum of applications (e.g. multimedia, telecommunication and data encryption). From a structural point of view, the PiCoGA is composed of 24 rows, each 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 (e.g. 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 is composed of three main sub-parts, highlighted in Fig. 24:

- A homogeneous array of 16x24 RLCs with 4-bit granularity (capable of performing operations e.g. 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 (e.g. data availability, stall generation, etc.)

In terms of I/O channels, the PiCoGA features twelve 32-bit inputs and four 32-bit outputs, thus allowing for each operation (PGAOP) to read up to 384 bits and to write 128 bits.

The PiCoGA 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.

If we exclude the interface block, the PiCoGA is a custom designed array, thus scalability and modularity is limited and requires additional work. The PiCoGA interface supports the propagation of the dataflow paradigm used inside the PiCoGA at an instance level, thus obtaining a hierarchical pipeline.

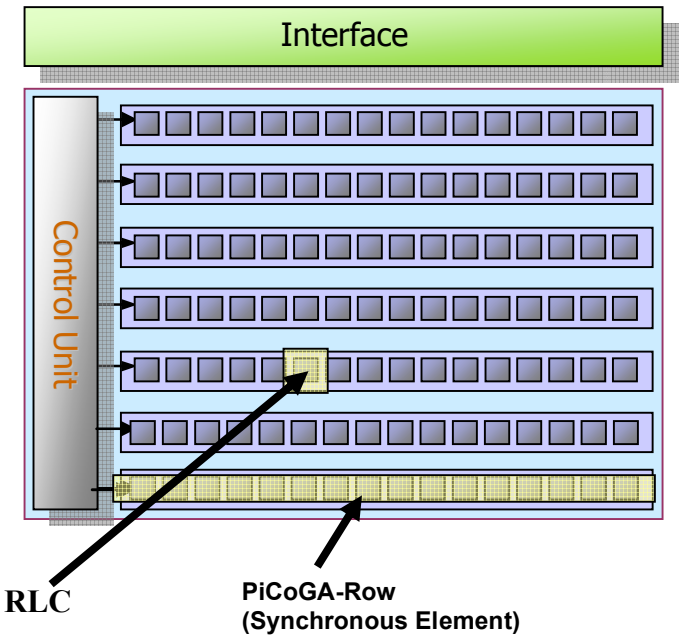


Fig. 24. Simplified PiCoGA achitecture.

3.5.3.1 PICOGA ARCHITECTURE

The main features of the PiCoGA architecture are:

1. A medium-grained configurable matrix of 16x24 RLCs
2. A reconfigurable Control Unit, based on 24 Row Control Units (RCUs) that handle the matrix as a data path (see Fig. 24).
3. 12 primary 32-bit inputs and 4 primary 32-bit outputs
4. 4 configuration contexts are provided as a first-level configuration cache
    a. only 2 clock cycles are required to change the active context (context switch)
    b. only one configuration context can be active at a time.
5. Up to 4 independent PiCoGA operations can be loaded in each context, featuring partial run-time reconfiguration

Each RLC can compute algebraic and/or logic operations on 2 operands of 4 bits each, producing a carry-out/overflow signal and a 4-bit result. As a consequence, each row can provide a 64-bit operation or two 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.

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 each operation output. This storage element cannot be bypassed cascading different cells. Thus PiCoGA can be considered a pipelined structure where each elementary operator composes a 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. Fig. 25 shows an example of pipelined DFG mapped onto PiCoGA.

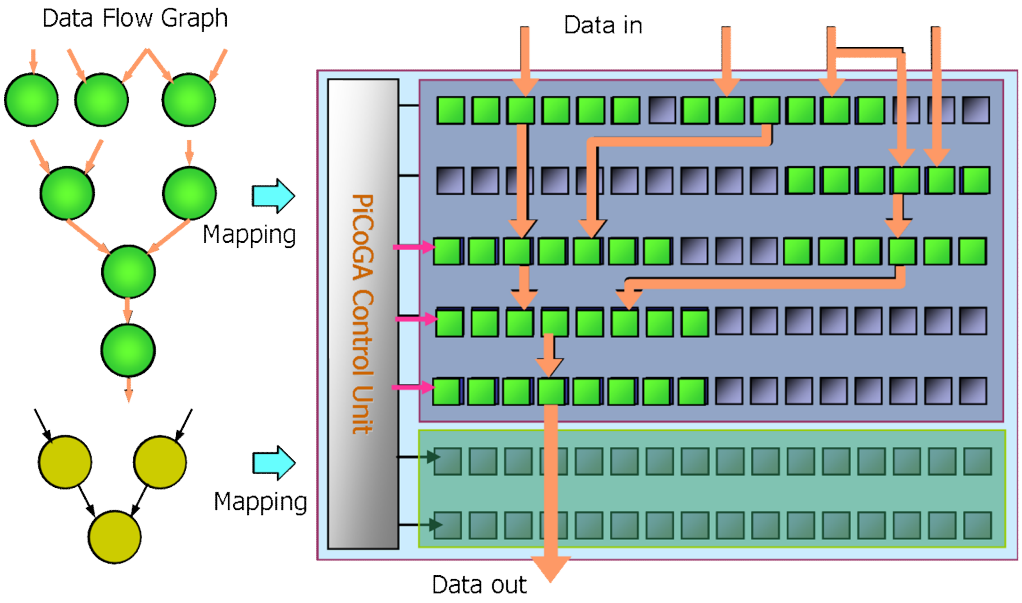


Fig. 25. Pipelined DFG in PiCoGA.

The set of elementary operations composing a PGAOP and their data dependencies are described by a DFG (Data Flow Graph). PiCoGA is programmed using Griffy-C. As shown in the example in Fig. 26. Griffy-C is a subset of the C language that is used to specify a set of operations that describe the DFG. Automated tools (Griffy-C compiler) are used to:

1. Analyze all elementary operations described in the Griffy-C code composing the DFG, determining the bit-width and their dependencies. Elementary operations are also called DFG nodes.
2. Determine the intrinsic ILP (Instruction Level Parallelism) between operations (nodes). Fig. 27 shows an example of Pipelined DFG automatically extracted from a Griffy-C description. In this representation, nodes are aligned for pipeline stage. A special class of nodes is routing-only nodes. Routing-only nodes do not require computational logic to be performed: they are implemented just exploiting interconnection resources (e.g. shifts, comparisons with zero) or may be collapsed in the following operation (logic operations by constants) and thus do not occupy a pipeline stage. In Fig. 27, routing only operations are depicted with dotted nodes, while pipeline stages are aligned by rows.
3. Map the logic operands on the hardware resources of the PiCoGA cells (a cell is formed by a Lookup Table, an ALU, and some additional multiplexing and computational logic). Each cell features a register that is used to implement pipelined computation. Operations can not be cascaded over two different rows. Fig. 28 shows a typical mapping on PiCoGA.
4. Route the required interconnections between RLCs using the PiCoGA interconnection channels.
5. Provide the bitstream (in the form of a C vector) to be loaded in the PiCoGA in order to configure both the array and the control unit (the PiCoGA Interface does not require a specific configuration bitstream). Configurations can be loaded in any configuration layer starting from any available row.

```
#pragma pga SAD4 1 2 out p1 p2 {
  unsigned char p10, p11, p12, p13;
  unsigned char cond0, cond1, cond2, cond3;
  …
  #pragma attrib cond0, cond1, cond2, cond3, SIZE=1
  …
```

```
  sub0a = p10 - p20; sub0b = p20 - p10;

  cond0 = sub0a < 0;

  sub0  = cond0 ? sub0b : sub0a;

  …

  out = acc1 + acc2;

}

#pragma end
```

Fig. 26. Example of Griffy-C code representing a SAD (Sum of Absolute Differences).

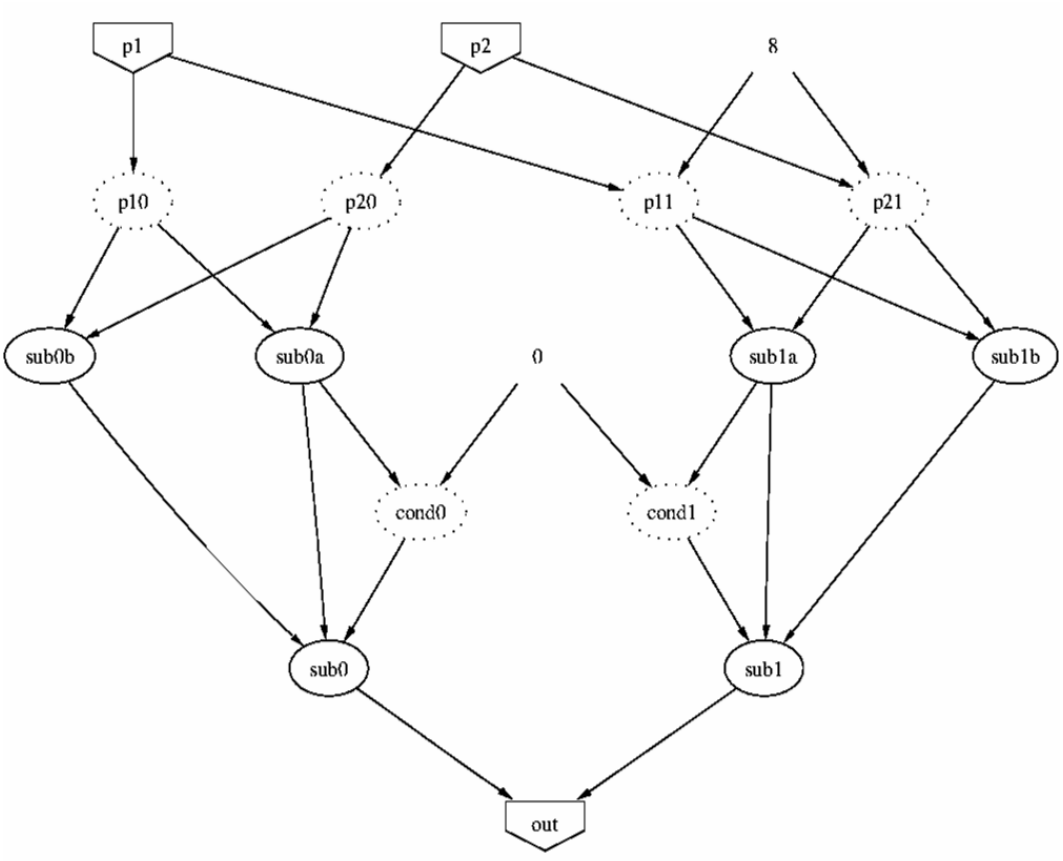


Fig. 27. Example of Pipelined DFG.

Fig. 28 represents a typical example of mapping onto PiCoGA. As explained in previous sections, after a data-dependency analysis, the DFG is arranged in a set of pipeline stages (thus obtaining the Pipelined DFG). Each of pipeline stage is placed in a set of rows (typically they are contiguous rows, but this is not mandatory).

In Fig. 28, different colors represent different pipeline stages. Depending on the row-level granularity of the PiCoGA Control Unit, one row can be assigned only to one single pipeline stage, and it cannot be shared among different pipeline stages.

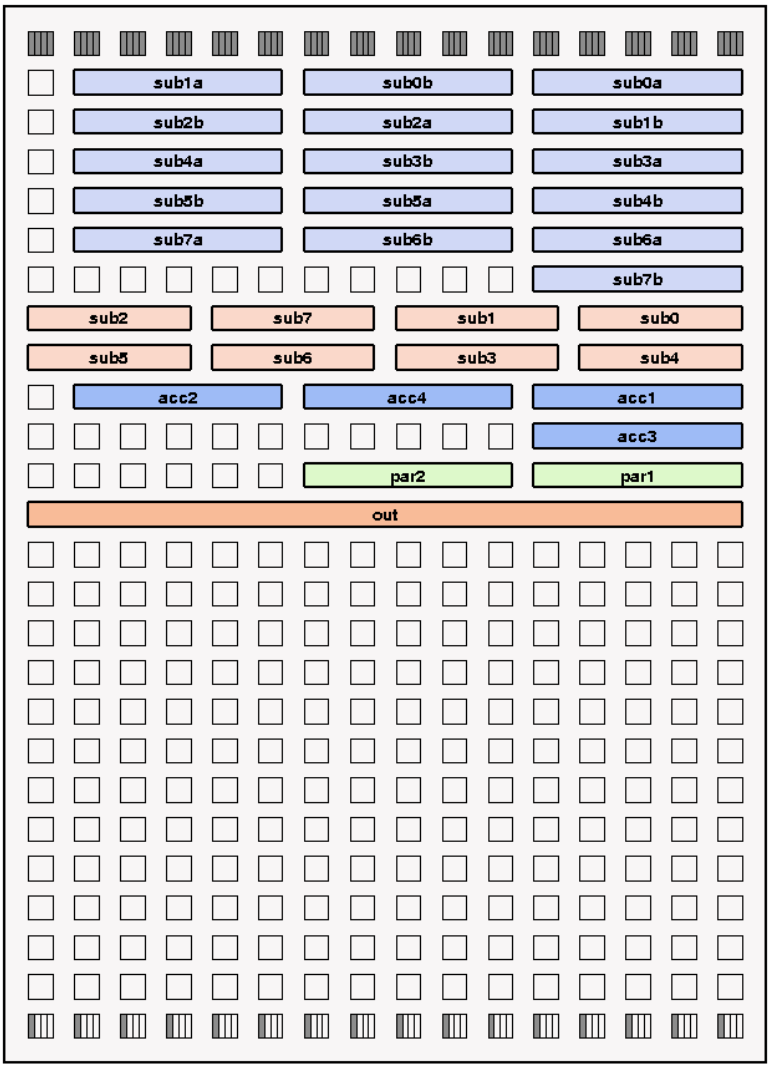


Fig. 28. Example of PGAOP mapping on PiCoGA.

The internal architecture of the Reconfigurable Logic Cell is depicted in Fig. 29. Three different structures can be identified:

1. The input pre-processing logic, which is responsible to internally route inputs to the ALU or the LUT and to mask them when a constant input is needed
2. The elaboration block (ALU & LUT), which performs the real computation based on the operation selected by the RLCop block

3. The output manager, which can select outputs from the ALU, the LUT, and eventually from the Carry-Chain and synchronize them through Flip-Flops. The output block samples when enabled by the Row Execution Enable signal provided by 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 and 4/5/6-bit inputs
    - 2-bit outputs and 4/5-bit inputs
    - 4-bit outputs and 4-bit inputs
    - a couple of independent lookup tables featuring:
        - 1-bit output and 4-bit inputs
        - 2-bit outputs and 4-bit inputs
- Up to 256-bit configurable memory module. Each configuration context provides 64-bit LUTs (see the previous point) and this special memory module can be implemented flattening in a single-context configuration the memory amount of all the LUTs. This special memory configuration can be applied for every RLC in the array, and the addressing is internal, and performed through other RLCs.
- 4-bit Multiplier module; in more detail, it is a multiplier module with 10-bit (in case of $A \times B$. 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/medium multiplier on PiCoGA resource.
- 4-bit Galois Field Multiplier – GF(2^4)

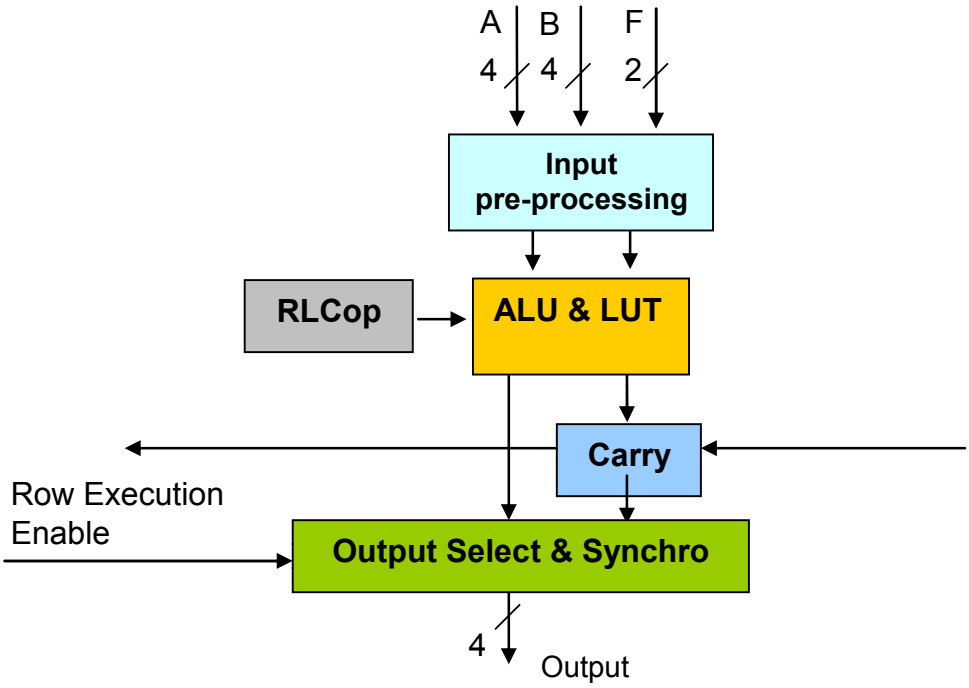


Fig. 29. Reconfigurable Logic Cell: simplified architecture.

Furthermore, lookup tables can be used to implement operations that require carry propagation, such as the comparison between two variables. LUTs can be programmed to use the carry chain while the carry-out can be re-directed to standards outputs. While standard RLC inputs (A, B in Fig. 29) are 4-bitwise (compliant with the cell granularity), the F inputs are 2 additional bits, that are used only when the multiplier module or some customized configuration is used.

The PiCoGA Control Unit handles the pipeline evolution, triggering the execution of a pipeline stage (implemented as a set of rows) when:

- input data are available
- output data can be overwritten
- writeback channels are available,

A data-flow graph directly represents dependencies among computational nodes through the data dependency graph, and it is possible to check both forward and feedback arcs to handle an optimal pipelined execution.

A pipelined data-flow computation can be modeled using timed Petri Nets associating an inverse data arc and a placeholder to each data arc (representing a data dependency). Each node computation is “taken” when all input arcs have a token in the placeholder and it produces a token for each output arc. The

activation of each node, or transition in terms of Petri Nets specific language, depends on each preceding node's completion and on each successive node's availability through a producer/consumer paradigm.

Under this pattern, the dedicated programmable control unit can be used to handle the pipeline activity, to start new PGAOPs or to stall them when requested resources are not available yet (e.g. when write back channels are already used by another PGAOP).

To save area, the dedicated control unit works with a granularity of one array row, thus 16 RLCs are the minimum number of active cells. More than one PiCoGA row can be used to build a wider pipeline stage, but, in order to maintain a fixed clock frequency cascaded RLCs are better mapped on different pipeline stages.

When a pipeline stage computes, it produces a "token" which is sent to preceding and successive nodes through a dedicated programmable interconnection channel. Each RCU receives "tokens" from the preceding and successive connected nodes which represent placeholders of the equivalent timed Petri Net that manages the pipelined DFG computation. Under this pattern, we schedule computational nodes to build pipeline stages, according with the earliest firing rule, and then we map pipeline stages on a contiguous set of rows. Fig. 30 shows a possible pipelined data-flow graph and the corresponding simplified control unit configuration.

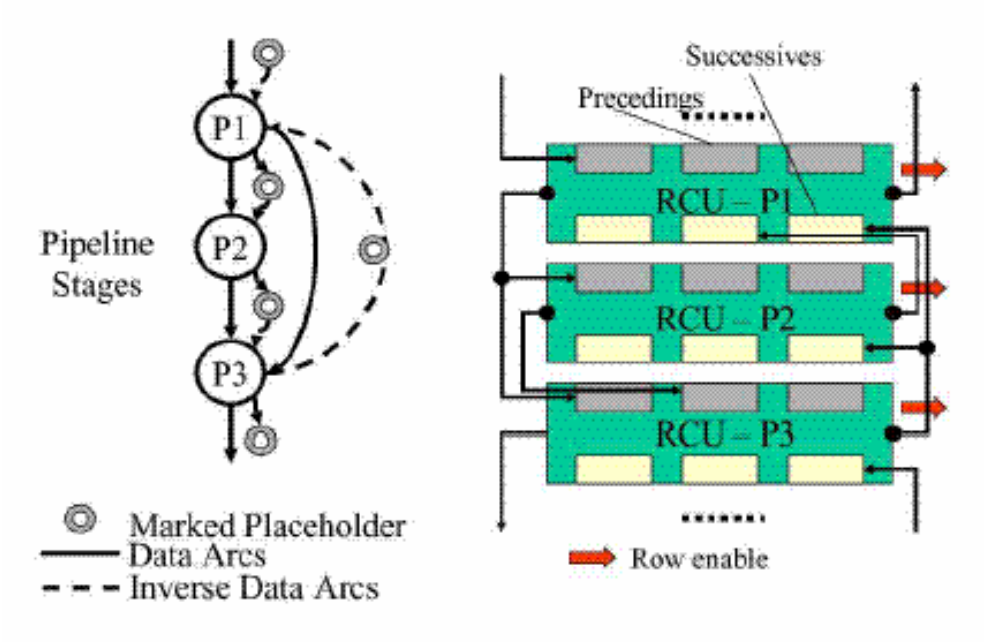


Fig. 30. Pipeline management using RCUs.

3.5.3.2 PICOGA PROGRAMMING APPROACH

The language used to configure the PiCoGA in order to efficiently implement pipelined DFG is called Griffy-C. Griffy-C is based on a restricted subset of ANSI C syntax enhanced with some extensions to handle variable resizing and register allocation inside the PiCoGA: differences with other approaches reside primarily in the fact that Griffy is aimed at the extraction of a pipelined DFG from standard C to be mapped 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 it is handled separately by the hardwired control unit, thus allowing much smaller resource utilization and easing the mapping phase. This also greatly enhances the placing regularity.

Griffy-C is used as a friendly format in order to configure the PiCoGA using hand-written behavioral descriptions of DFGs, but can also be used as an intermediate representation (IR) automatically generated from high-level compilers. It is thus possible to provide different entry points for the compiling flow: high-level C descriptions, pre-processed 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. 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 hypotheses 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.
- Single assignment: each variable is assigned only once, avoiding hardware connection ambiguity.
- Manual dismantling: only single operator expressions are allowed (similarly to intermediate representation or assembly code).

Basic Griffy-C operators are summarized in Fig. 31, while special intrinsic functions are provided in the Griffy-C environment in order to allow the user to instance non-standard operations, such as for example the "multiplier module".

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

Fig. 31. Basic operations in Griffy-C.

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. Operator width is automatically derived from the operand sizes. Variables defined as static are used to allocate static registers inside the PiCoGA, which are registers whose value are maintained across successive

PGAOP calls (i.e. to implement accumulations). All other variables are considered "local" to the operation and are not visible to successive PGAOP calls. Once critical computation kernels are identified through a code profiling step in the source code, they are rewritten using Griffy-C and can be included in the original C sources as atomic PiCoGA operations. *#pragma* directives are used to retarget the compiling flow from standard assembly code to the reconfigurable device.

```
#pragma picoga name n_outs n_ins <outs> <ins>{

  [declaration of the variables]

  [PiCoGA-function body]

}

#pragma end
```

Starting from the Griffy-C description, DFGs are placed and routed into the PiCoGA, while the array control unit is programmed in order to perform a pipelined execution. Hardware configuration is obtained by direct mapping of predefined Griffy-C library operators. Thanks to this library-based approach, specific gate-array resources can be exploited for special calculations, such as a fast carry chain, in order to efficiently implement arithmetic or comparison operators. Logic synthesis is kept to a minimum, implementing only constant folding (and propagation) and routing-only operand extraction such as constant shifts: those operations are implemented collapsing constants into destination cells, as library macros have soft-boundaries and can be manipulated during the synthesis process.

The functional validation of a PGAOP is carried out in a standard C environment. It allows the user to debug a PGAOP in order to verify the correctness of the code. The PGAOP, described in Griffy-C, is compiled by PiCoGA tools that provide an ANSI C emulation.

The emulation is functionally equivalent to Griffy-C, taking into account both standard operations and instruction set extension, such as direct LUT specification or multiplier modules. Furthermore, the emulation takes into account the scheduling performed by the compiler when pipeline stages are built.

Debugging is facilitated by a Graphical User Interface (GUI) that can be associated to a standard debugging tool in order to provide an easy way to inspect intermediate results in the Griffy-C part. While the standard C code can be suspended through breakpoint, the execution on the PiCoGA is emulated as if it was an atomic instruction (it is a functional model).

3.5.3.3 CONFIGURATION CONTROL

Since the PiCoGA was intended to be used in a processor-oriented environment (as a configurable functional unit in the first release, as a configurable co-processor in the MORPHEUS context), the language used is a processor-oriented language. We call *instructions* all the operations performed to control the PiCoGA (e.g. trigger a PGAOP or load a configuration) even if they are implemented through read/write operation in the system memory or in a more specific configuration register. The real integration of the PiCoGA in a system, and thus the PiCoGA control, is an ongoing issue in MORPHEUS and several considerations will be provided in the last section of the PiCoGA description.

Computation on the PiCoGA is explicitly triggered by a specific instruction (PGAOP) from the main processor. Each PGAOP may feature up to 12 primary 32-bit inputs and 4 primary 32-bit outputs. The *latency* of each PGAOP is the number of cycles between the PGAOP trigger (reading of the 12 inputs) and the generation of the 4 outputs.

The *issue delay* of each PGAOP is the number of cycles that must divide two different issues of the same PGAOP. It depends on data dependencies across pipelines stages, and it represents the maximum “distance” between a producer (the pipeline stage that provides/write a data) and a consumer (the pipeline stage that consumes/read a data).

Both such parameters are determined by the Griffy-C compiler. The latency could be higher than the predicted one due to hardware stalls; hardware stalls could be caused by many different reasons, one of those being the need to respect issue delay constraints. There is never any issue delay between different PGAOPs, provided they reside on the same context layer.

PiCoGA is a multi-layer device: it features 4 layers, or contexts. This means that each configuration memory device inside the array is multiplied by 4:

at any instant, 4 different configurations are available on the array. In turn, each layer may contain up to 4 different PGAOPs (sharing the 24 available rows between them).

A single row can not be shared between different PGAOPs. At any given time the PiCoGA may hold up to 4x4=16 active PGAOPs, provided the sum of rows occupied by the 4 PGAOPs in a layer is less or equal to 24. PGAOPs residing in the same layer can be run concurrently (only one PGAOP can be issued in a single cycle, but PGAOPs have latency more than one cycle) while before changing layer (that is, running a PGAOP residing in a layer different from the current) it is necessary to conclude all PGAOPs in the current layer. An automatic hardware mechanism ensures this, at a price of inserting stalls in the processor flow.

The PiCoGA is not configured one-shot as a whole, but each PGAOP can be uploaded or erased to/from the array as a separate entity. For this reason, the location on the array where a given PGAOP should be saved depends dynamically on the PiCoGA state (that is, which PGAOPs were previously loaded and where).

The position of an active PGAOP on the PiCoGA is specified by the following parameters:

1. *Layer*: The configuration layer where the PGAOP was located.
2. *Region*: Each layer allows a maximum of 4 PGAOPs, regardless their size. For this reason, each PGAOP in a layer is identified by an integer range [0..3] called region. Region is NOT topological (geographic) information specifying a physical portion of the array, but only a logic identifier to distinguish different PGAOPs sharing the same layer.
3. *Starting Row*: Each PGAOP is composed by a fixed number of rows. The physical position of a PGAOP on a layer is determined by the StartingRow. Obviously, $Starting\_Row + PGAOP\_Size \leq 24$.

These three parameters allow univocally referring to a given PGAOP on the PiCoGA.

A configuration is a bitstream residing in the system memory and containing a set of PGAOPs. There is no restriction on the number of PGAOPs contained, and very often they are more than what can be contained in the

PiCoGA. The configuration is a potentially infinite (the only limit being the available system memory) repository of PGAOPs, between whom the user can select a maximum of 16 active PGAOPs and load them on the array. The bitstream is produced in C format by the Griffy-C compiler.

In order to compute a PGAOP, it is necessary to load the relative bitstream on the Gate-Array. As described above, PiCoGA is not configured one-shot as a whole, but each PGAOP is uploaded/erased separately from the array. At any time the loading of a PGAOP is required, it is necessary to determine if and where there is a suitable space on the array, thus attributing a layer, region and starting row to the PGAOP. At the moment of loading a PGAOP on the array, the PGAOP receives the specification of $layer + region + starting\_row$ that will be then used to access the PGAOP.

Before the loading, the PGAOP is accessed through a pointer in the processor addressing space, after the load by the above described information. Following their definition, $layer + region + starting\_row$ are packed in a specific integer value, defined PGAOP Descriptor (PD), which is defined as follows:

Table 5. Parameter format for PGAOP position specification (bits).

15:14	13:12	11:10	9:5	4:0
Region	Layer	--	Size	Starting_Row

The PD can be determined by two different means:

1. *User Defined Load (Pga_Force)*. In this case the programmer has the responsibility to determine an available location, and will specify Layer, Region and StartingRow. If the specified location is illegal (Layer or region > 3, StartRow > 23) or it refers in all or in part to a location already occupied the PiCoGA will generate an exception. A specific C function computed by the processor will update the PGAOP description struct in the system memory and generate the PD. The function syntax is: *Int PD = Pga_Force(int Layer, int StartRow, int Region, int pgaop_name)*
2. *Automated Load (Pga_Allocate)*. In this case a C function will search an internal database (located on the system memory) to determine the first available space on the array and then update the PGAOP description struct

in the system memory and generate the PD. The function syntax is: *Int PD = Pga_Allocate(int pgaop_name)*

In both cases, at the end of the C function a memory transfer will be triggered, utilizing the PGAOP pointer as a base, to transfer the PGAOP bitstream into the chosen location of the array. This process is triggered by the built-in instruction:

Pga_Load(int Pointer,int PD,int load_mode)*

Pointer is a pointer to the PGAOP bitstream in the configuration cache, and is retrieved in the PGAOP struct. PD is calculated according to the $layer + region + starting\_row$ parameters determined by *pga_allocate* or specified by the user with *pga_force*.

The PiCoGA may run a *pgaload* (load a bitstream) on one layer, while a PGAOP computation is running on a different layer. If a *pgaload* is issued on an active layer (a layer computing one or more PGAOPs) the load will be stalled until the end of all active PGAOPs. If a *pgaload* is already running, following PGAOPs will be stored in a specific 8-slot load queue until the load is finished. If a *pgaload* is issued and the queue is full a stall will occur. If a PGAOP is required before being loaded, an exception will occur. On the contrary, the issue of a PGAOP that is currently being load will cause a processor stall, again affecting the system performance.

As described above, the bitstream size is $3024 \times Rows\_Number$. As the load operation can be performed while both the processor and a different layer of the PiCoGA are computing, it is often possible to hide the *pgaload* latency while performing different elaborations.

Following the above description, every PGAOP is composed by a self-contained, pre-placed and pre-routed set of rows designed by the Griffy-C compiler to compute a given DFG. The location of this set of rows on the array is not determined at compilation time, but negotiated any time the PGAOP is required on the array. For this reason, the location of the PGAOP will depend on the previous history of the program. Any *pgaload* will have an influence on the following. It should be observed that the order of *pgaload* issues (that is

independent from the issue order of the relative PGAOPs) have a strong influence on the program performance.

As described above, a C function will look for the first available location on the array in order to perform a *pgaload*. As a consequence, two PGAOPs that are tightly coupled in the program code may be placed in the same layer or not according to the overall *pgaload* order. The two will be able to run concurrently if they share the same layer, whereas will run sequentially if they reside on different layers. In order to achieve the maximum performance it might then be necessary to carefully architect the location of the various PGAOPs on the available layers. When the bitstream relative to a given PGAOP is no longer needed on the array, it is necessary to explicitly erase it from the array in order to make space for future loading. This is performed with the C function: *pga_deallocate(int pganame)*. It will erase the specified PGAOP from the internal database (used by pga_allocate) and force the deletion of the bitstream from the array through the built-in function *pga_free(int pganame)*.

Once the bitstream relative to a given pgaop has been successfully loaded on the array, the pgaop can be computed an indefinite number of times on different inputs. A fundamental difference between PiCoGA and embedded FPGAs is that PiCoGA is explicitly issued by an external command, and inputs are explicitly selected for each issue.

PiCoGA features 384 input bits and 128 output bits, organized in 12/4 words. The PiCoGA computation is entirely synchronous. The issue of the C built in function: *PGAOP (int pga_name)*, causes the device to sample all inputs and start the pipelined computation of the DFG on the array. If the PGAOP is currently being loaded a HW stall mechanism will stop the processor computation until the load is concluded, while if it has never been scheduled for loading an exception will be issued.

Only one PGAOP can be triggered in each cycle if this is compatible with the PiCoGA internal status which can raise a stall signal. Normally the 128 result bits will be produced after the latency specified by the Griffy compiler. In case different PGAOPs are active (currently computing on the array), there may be congestion on the output channels, resolved by a priority mechanism (the oldest

PGAOP in the array gets higher priority) and latencies could be higher than the expected ones. Each output can feature a specific latency, so that the 4 results of a PGAOP can be produced in 4 different clock cycles, and can be stalled independently due to congestion on the output channels.

3.6 SUMMARY

The MORPHEUS platform represents a heterogeneous architecture which includes three reconfigurable engines (HREs) of fine, medium and coarse granularities, controlled by a general-purpose ARM processor.

PACT XPP is a coarse-grain reconfigurable engine targeting computation intensive applications with high level of data parallelism. Once configured, it operates well with video processing tasks on a pixel level, performing complex arithmetical operations on a data flow in a SIMD manner.

PiCoGA is a medium-grain (4-bit width) reconfigurable engine. Unlike XPP, it deals equally well with both data and reconfiguration intensive flows (SIMD and MIMD) thanks to the four contexts that can be dynamically reconfigured. Consequently, PiCoGA can be efficiently utilized for multimedia, wireless, as well as cryptography applications, mostly benefiting from advanced dynamical reconfiguration mechanism.

M2000 is a fine-grain reconfigurable engine, architecturally structured as an embedded FPGA inheriting all its features. It meant less for computation intensive applications, in contrast to XPP; or reconfiguration intensive applications, in contrast to PiCoGA. However, M2000 benefits from the great usage of LUTs and MUXs in, e.g., network routing applications and various communication protocols.

Thus, by unifying three types of reconfigurable engines in one platform, MORPHEUS covers a wide range of applications, providing the most efficient hardware for each computational concept.

Each of the HREs requires handling of a set of data types (computation, configuration, control, etc.) featuring various intensities. Thus, MORPHEUS is a typical example of system targeted by this research. The development of memory organization in the context of MORPHEUS can be accounted as an important case study in order to verge towards creating a general methodology for heterogeneous reconfigurable architectures design.

CHAPTER 4

HIERARCHICAL MEMORY ORGANIZATION AND DISTRIBUTED DATA STRUCTURE

This chapter thoroughly discusses novel methods, techniques and solutions for memory organization and data communication. The target architecture and its basic features are presented in the previous chapter. The chapter starts with the description of the generic bandwidth requirements followed by the developed data structure and memory hierarchy descriptions. Being based on these architectural concepts, a computational model is presented. In addition, an example of the integration of one of the reconfigurable engines is given. The discussion marks out the data exchange mechanisms with the host platform. Finally, the features that distinguish this proposal from the related work are summarized.

4.1 GENERIC BANDWIDTH REQUIREMENTS

In this section, the theoretical bandwidth which can be achieved by the IPs will be recalled and analyzed. The estimation will not outline the bandwidth needed by the applications, but the theoretical maximum. Fig. 32 shows a block diagram of these theoretical bandwidth capabilities.

The theoretical maximum is calculated by adding the theoretically possible bandwidths of all main data consuming and producing modules of the MORPHEUS chip. These are literally the three different HREs as well as a memory controller and on-chip memory. The total bandwidth results are roughly obtained by multiplying the number of I/O ports, the bit width of I/O ports and the frequency. However, the following annotation to the given calculation has to be kept in mind.

This estimation does not take the impact of the HREs' internal subsystems into account. Depending on, e.g., the granularity of the task or communication interfaces among different clock domains, it is uncertain that the sub-system impacts the performance, introducing a specific overhead. This overhead is supposed to be negligible in case of large tasks, i.e. mappings on the HREs, because of the dominance of computation time over communication time. If, on the contrary, small kernels are mapped on the HREs, probably a loss of peak performance due to communication time can be observed, which includes e.g. synchronization overhead or memory latencies. Although these local considerations presumably do not impact the final performance of the system, an impact of the subsystem with regard to the kernel size cannot be excluded. Therefore, the given bandwidths are considered as a best-case estimation, which nevertheless is a reasonable justification of the decisions, described in this chapter. One should also note that most interfaces can either be used as input or output. This is indicated by the "or" statements below the outgoing arrows on Fig. 32. An "and" statement on the other hand means that input and output ports are separated and can therefore be used completely in parallel.

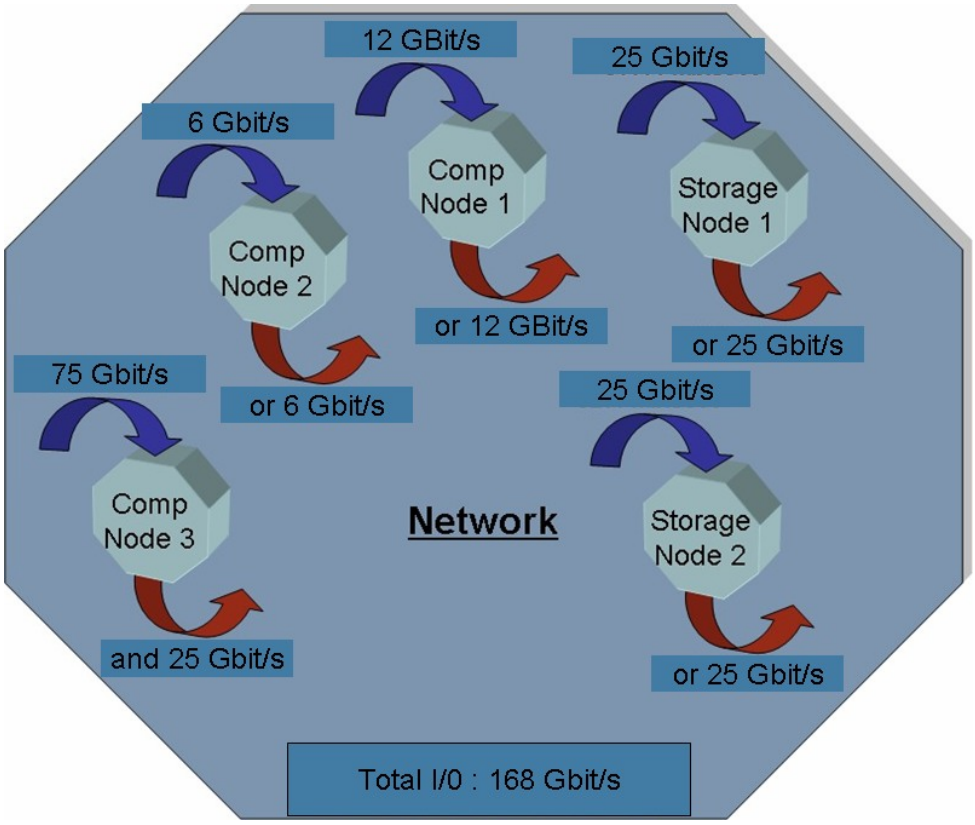


Fig. 32. Schematic view on the theoretical bandwidth constraints.

Each computational node (Comp Node 1,2,3), as well as each storage block (Storage Node 1,2) mirrors a major data generating or data consuming physical module of the MORPHEUS platform. The allocation to the IPs is illustrated in Table 6. However, this schematic view shows only the logical channels seen by the applications, instead of the actual physical channels of each module. It means that only the total bandwidth as a sum of all physical channels is exposed. In such a way, it is taken into account that the accurate bandwidth of the physical channels, which strongly depends on the applications, is not yet predetermined at this point.

Table 6. Nodes annotation.

Node	*IP Name*	*Frequency in MHz*
Comp 1	XPP	100
Comp 2	eFPGA	50-150
Comp 3	PiCoGA	200
Storage 1	On-Chip RAM	200
Storage 2	Memory Controller	200

It is also important that the computational nodes will run each at its own clock domain. Hence, there are different clock speeds assumed for each of the

computational nodes. Table 6 shows the different frequencies, which are taken to calculate the bandwidths.

Fig. 32 presents only the theoretical bandwidth of the stand-alone nodes; therefore it might be the case that the total input bandwidth is not equal to the total output bandwidth. Of course, practically achievable bandwidth between the nodes will be rather smaller and will significantly depend on the chosen hierarchical and interfacing solutions. It is essential to discover the extreme cases (*Pareto-points*) of dataflow distribution imposed by the target applications in order to specify the system requirements.

A summary of inter-nodes communication bandwidth requirements is presented in Table 7. These requirements will be specified more precisely together with the detailed description of system architecture. Fig. 32 and Table 7 do not take into account the ARM microcontroller, as well as the other peripherals, since their communication capability can be neglected regarding data streaming and data intensive applications. Table 4 presents a general idea about the traffic range: low, high, very high; which would help to imagine better the general view of system communication requirements.

Table 7. Inter-node bandwidth requirements.

	Comp node 1	*Comp node 2*	*Comp node 3*	*Int. mem.*	*Ext. mem.*
Comp node 1	x	High	Low	High	Very high
Comp node 2	High	X	High	High	High
Comp node 3	Low	High	x	Very high	Low
Int mem	High	High	Very high	x	High
Ext mem	Very high	High	Low	High	X

The legend for Table 7:

- “Low” equals about 1 Gbit/s
- “High” equals about 4 Gbit/s
- “Very High” equals about 10 Gbit/s

4.2 DATA STRUCTURE IN THE SYSTEM

The set of different communication flows and data storage levels that are presented in the system can be summarized as follows:

1. *Internal Computation Data Flow*: On the data side, the architecture is composed of the main processor, I/O peripherals, user programmed DMA, as well as of two types of communication infrastructures – AMBA AHB bus and NoC backbone, which connects HREs data ports. AMBA AHB can be used efficiently when only one of the available HREs is used at full bandwidth, and especially during the chip verification phase to address each HRE and to verify independently its performance and functionality. On the other hand, the NoC can be used to handle high-speed, large bandwidth communication to achieve relevant peak performance with all HREs computing concurrently.
2. *I/O Interfaces*: Access to external resources is performed through a Memory Controller Device, which can be mapped on the main Data AHB bus and interfaced on the NoC also. The memory controller allows handling off-chip Flash, SRAM and SDRAM memory banks and can be internally interfaced to up to 6 internal AHB bus (thus allowing to share the resource between Data, Control and Configuration flows). Finally, I/O data that feature small bandwidth but may benefit from some degrees of pre-processing can be acquired through the eFPGA HRE that in this case can act as configurable I/O engine.
3. *Configuration data flow*: In the overall system, the flow of data should be strictly separated from the flow of HRE reconfiguration bit-streams, in order to provide efficient memory management and avoid bottlenecks. The configuration of HREs is independent from the data flow and is handled by a specific Configuration Management unit (CM) that controls DMA and HRE's configuration memories. Connectivity between configuration resources is provided by a second, separate AMBA AHB bus. For testability reasons, the processor core can have access to the configuration bus through a bridge mapped on the main AMBA bus.

4. *Control Flow*: The main task of the ARM processor in the MORPHEUS architecture is to provide overall management and "coarse-grained" task synchronization. Consequently, a third set of internal transfers, apart from computation data and configuration bitstreams, will be related to the control of all on-chip resources, i.e. HRE control words or synchronization operations. During HRE computation, the ARM core monitors the state of each HRE and produces the appropriate commands to load configuration, start computation, route data chunks to the required destination, manage DEB consistency and so on. Control transfers share the same bus with computational data, and additionally utilize an interrupt subsystem which includes Interrupt Controller directly connected to the ARM and HREs.

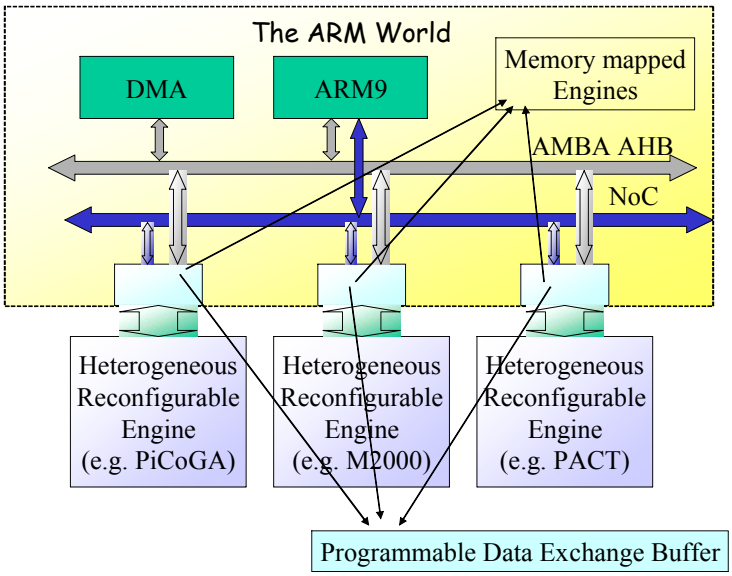


Fig. 33. Simplified MORPHEUS architecture.

4.2.1 COMPUTATIONAL DATA STORAGE

The MORPHEUS architecture supports two different basements for computational data transfers:

1. A first option is to use the main AMBA AHB bus for direct data communication between the ARM and HREs. This is a low-cost, low performance option that can be utilized during the system verification phase or for non-critical tasks. In case a higher performance is required, it

is possible to implement a multi-layer bus. However such option can be considered as redundant.

2. A second option is provided by the NoC infrastructure, which can offer high bandwidth inter-HREs communication. The NoC supports a high throughput data transfer mechanism compliant with the streaming computation model.

The interconnect architecture is used to connect the slow speed, high density, possibly non-volatile off-chip memory banks with the small and fast memories that are local to the ARM core and the HREs, defined Tightly Coupled Memories (TCM) for the ARM core, and DEB in the case of the HREs. Local memories (TCM or DEB) can be in the range of 1K to 64K bytes with access times around 2 ns, while off-chip memories may offer storage capabilities of several mega bytes but access time of tens of ns (see Table 8).

Table 8. Memory hierarchy levels.

Memory Level	*Bandwidth*	*Size*	*Notes*
Level 1: Local Memory (one channel)	6,4 Gb/s	1K :16K	Embedded dual port, dual clock
Level 2: On-Chip Memory	4,0 Gb/s	64K:1M	Embedded, single port, Multi-cycle access
Level 3: Off-chip Flash	400 Mb/s	1M:256M	Non volatile
Level 3: Off-chip SRAM	640 Mb/s	1M:256 M	Static memory
Level 3: Off chip SDRAM	400:560Mb/s	64M:512M	Dynamic memory

In case (1), in order to exploit the bus full performance, an intermediate sized memory is needed on the AMBA AHB buffer to act as communication repository, or software cache between the two hierarchy levels described above. The size of such memory is in the range of 64K to 1M bytes.

The utilization of the NoC as an extension to the bus infrastructure, as described by point (2), is due to its natural effectiveness in transporting large amounts of data. The IPs included in the architecture can to process large quantities of data with a relevant performance. Therefore, the streaming information delivery is crucial. A relevant advantage of the NoC approach is due to the fact that the routing blocks that compose the network feature local storage capability. The NoC itself can than be considered a powerful temporary data

storage mean, distributed along the data transfer paths, thus relieving the necessity to add large memories in the design. The point-to-point interconnection of the NoC, with respect with the point-repository-point offered by a standard bus, may also offer performance enhancements and power consumption minimization due to the significantly smaller average length of interconnect wires.

The off-chip memory can be composed of Flash, SRAM, or SDRAM banks according to the application benchmark requirements. Access to external storage devices is ensured by a specific memory controller. State of the art controller can provide several AMBA AHB slave ports, so that a controller may be multiplexed between different AMBA instances (e.g. data bus and configuration bus). Should the benchmark analysis prove the need to augment the chip-to-memory I/O bandwidth it is possible to integrate more than one controller to provide fully parallel access to separate off-chip banks, but it would be necessary to pay a severe cost in terms of I/O pads required.

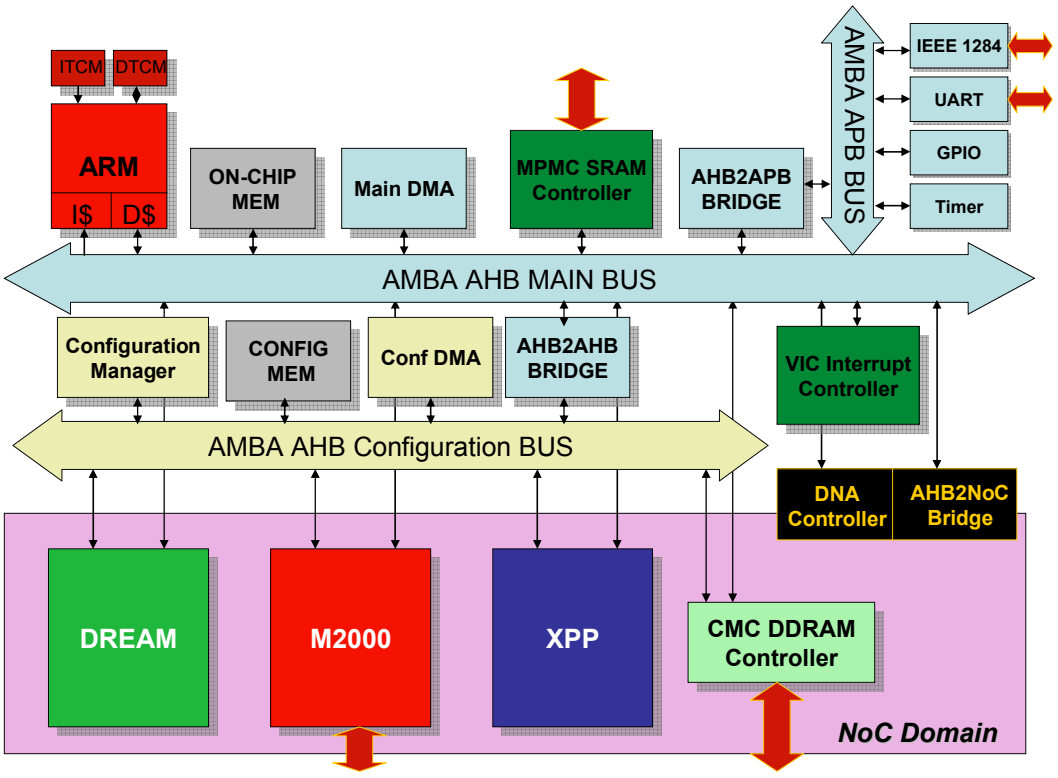


Fig. 34. MORHEUS SoC architecture.

In MORPHEUS system (Fig. 34), each HRE is seen by the ARM-centered system as a memory-mapped bus slave. The Network-on-Chip paradigm, though,

requires that each HRE should be considered as both initiator and/or target of data transfer according to the data flow determined between HREs by the application. Memories are passive devices: in order to connect the HRE to the NoC as initiator, each local memory should be connected to a DMA unit or a specific FSM for addressing the system-port. In this way HREs can interface to the NoC as transfer initiator. The DMA would be programmed by the user through the ARM processor or a specific controller as part of the HRE control. Its requirements are related to the NoC specifications and depend on the chosen NoC topology, switching type and pattern granularity.

The ARM processor is Harvard architecture, providing concurrent access to the data and instruction memories. The ARM9 core included in the MORPHEUS architecture features data and instruction caches (16Kbytes each), data and instruction TCMs of programmable width (typical TCM size is 32 to 64 Kbytes), and independent data and instructions AMBA AHB ports.

4.2.2 CONTROL DATA STORAGE

The critical task of the embedded ARM processor core in the MORPHEUS architecture is to support the centralized control of the different HREs. Control information includes the handling of the following resources:

- Issuing of configuration tasks and computation commands according to the Molen paradigm (*set*, *execute* primitives);
- Synchronization of task dependencies, according to the Molen paradigm (*break* primitive) ;
- Routing of data and transfers between architectural resources: programming of AHB and NoC DMAs;
- Synchronization of exclusive access to DEBs;
- Programming of HRE address generators when utilizing HREs in “Slave” addressing mode;
- Configuration control: CM task synchronization, configuration DMA(s) programming.

Control information is provided to HREs through a set of registers shared between the HRE and the system, defined as XRs. XRs are accessible both from

the system clock domain and the HRE clock domain. In order to preserve data access consistency, it is necessary to provide an explicit synchronization protocol for each access. The detailed description of XR synchronization functionality will be determined in the following sections. It is also possible to use the ARM core set of interrupts to provide real time events handling.

The control flow does not feature a significant traffic bandwidth, but it might become critical in terms of transfer latency. According to the chosen interconnect strategy, XRs can be connected to the configuration bus and communicate with CM through the configuration AHB bus, or be connected to the data bus. In this case, there is no necessity to include an additional bus and the bridge in the control data path. In this case control information that is always issued by the processor core should have priority with respect to data flows handled by DMA. It should be observed that the main AMBA AHB bus, which is connected to the ARM core data port, will be used as data exchange vehicle only in the testing phase, while during peak computation data transfers will be performed on the NoC architecture, leaving the processor data bus free for the handling of control information. A further option is the definition of a third AMBA AHB layer, specifically allocated to control flow.

4.2.3 CONFIGURATION DATA STORAGE

Table 9 describes bitstream requirements for each IP in the system. The XPP core as well as the FlexEOS core does not support partial reconfigurations, so that full bitstreams have to be downloaded into the HREs. The PiCoGA core supports frequent partial reconfigurations that gain to be handled inside the HRE, so that a large bitstream is provided into the local configuration memory at a coarse granularity, and the PiCoGA behaves independently.

Table 9. Configuration bitstream requirements for the MORPHEUS IPs.

IP	*Bitstream Size*	*Array size*
M2000 eFPGA	60Kbytes	4Kgates
Pact Xpp	16Kbytes	8x8 Xpp Array
PiCoGA	72Kbytes	4layers x (24x16 RLC Array)

The configuration bus will have access to the off-chip banks through the memory controller device (see Fig. 35); it is possible to use a dedicated memory controller for the configuration bus to enhance bandwidth, but that would require a heavy penalty in terms of I/O pads (70 per each controller).

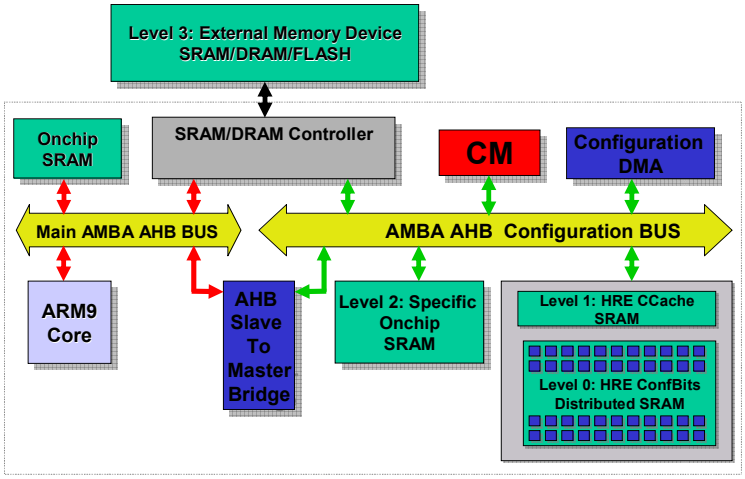


Fig. 35. Configuration data hierarchy.

As IP reconfiguration requires fast and low latency memory-to-IP communication rather than large bandwidth point-to-point communication, the use of the NoC as a mean to carry configuration is not considered reasonable. On the contrary, the bus width can be determined, according to the AMBA protocol, from 32 to 256 bits. In case the architecture simulation should underline a bottleneck in configuration traffic, it is possible to build a multi-layer bus structure.

The Configuration Bus (see Fig. 35) should be mastered by the specific Configuration Manager (CM) that synchronizes task configuration according to inputs and synchronization signals received by the ARM core. For testability purposes, the processor can have access to the configuration bus as an alternative master through an AHB-to-AHB bridge residing on the main processor bus. CM (or the processor core) may take advantage of local DMA(s), one for each eventual bus layer, to speed-up data transfers.

As it was the case with the data storage hierarchy, the configuration storage hierarchy also has to deal with the handling of different clock domains.

There are two solutions which could be proposed for a Configuration Interface (CI) implementation across the system/HRE clock domains boundary (see Fig. 36).

1. The first option provides a fine-grain streaming approach. In this case the CI is based on the clock domain crossing synchronization registers or asynchronous FIFOs. The configuration data flows directly through these registers. There is no local storage of configuration bits in the HRE domain, thus decreasing area occupation, at the price of a very relevant overhead in terms of performance since each synchronization will require 2 to 4 clocks for crossing the clock domain boundary.
2. The second option provides a coarse-grain solution for the configuration streams transportation that is analogous to what is described for data in section 4.2.1. A CEB is utilized in the same way as it was done with DEB, crossing clock domains with a coarse-grained synchronization that is very friendly for configuration data. Configurations are, in fact, typically organized in large chunks corresponding to a set of functionalities to be mapped on the IPs. The size of the CEB may depend on the IP requirements and the number of configurations that the application needs to have readily available. A reasonable choice is to replicate one (or a few) full configurations, but that may lead to relevant area occupation as described in Table 10.

It should be pointed out that PiCoGA and XPP provide embedded addressing capabilities for the configuration bitstream.

Table 10. Area requirements for dual clock CEBs.

IP Configuration	*Memory Size*	*Area Occupation vs IP Size (ST CMOS090 Technology)*
PiCoGA 4 layers	72Kb	1.6 mm^2 vs 12 mm^2
M2000 4K full configuration	60Kb	1.2 mm^2 vs 4.5 mm^2
4xPact Xpp 8x8 full configuration	64Kb	1.2 mm^2 vs 11 mm^2

Note: Single port memories feature comparable access time, while the area is roughly 60% of the figures described above.

The case of the embedded FPGA is different as the configuration mechanism is not bound to the same frequency specifications as the computation. The M2000 device is capable to perform computation on up to 16 different concurrent clock domains (it should be observed that the entry language for M2000 is VHDL/Verilog, so it is possible to program an application on the IP making use of different clock domains). Consequently, M2000 CI will be different from the others IP in a way that the M2000 configuration can be fed to the HRE directly from the bus ports at system speed. Therefore, there is no specific need to cross clock domain boundaries when transferring configuration on to the M2000 IP. From the system control and synchronization point of view, though, M2000 can feature the same API software interface for configuration as the other HREs, thus providing to system programmers a homogeneous approach.

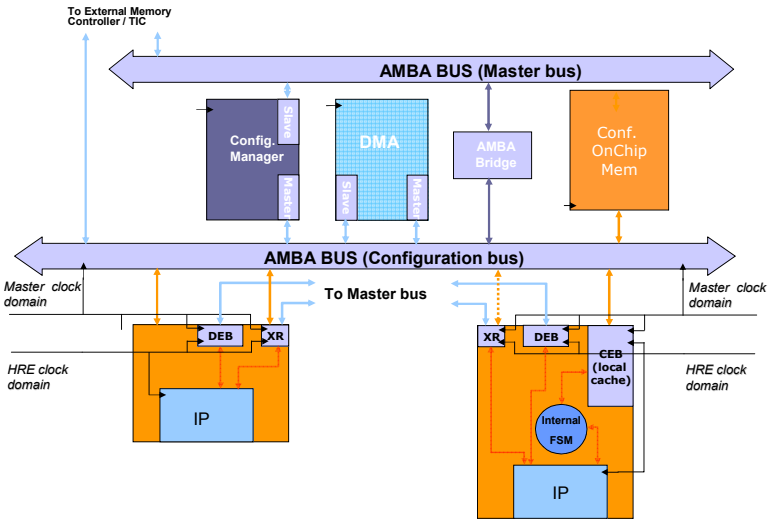


Fig. 36. HRE configuration.

4.3 MEMORY ARCHITECTURE DEVELOPMENT

A hierarchical structure of the memory subsystem is ought to deal with the issues posed by moving data from the external world to the computational engines (see Fig. 37). There are three memory storage levels presented in the system:

- Level 3 is an off-chip memory. It provides a global shared storage of the data, needed for the system.
- Level 2 is an on-chip memory. This memory is used as a temporal repository of the frequently used data blocks between level 3 and level 1. On this level, configuration data is strictly separated from the other data types.
- Level 1 is a dedicated data/configuration exchange buffer (DEB/CEB). It is tightly coupled with each HRE and has two main functions: 1) to store the data which is currently processed in HRE, and 2) to separate HRE clock domain from the system clock domain.

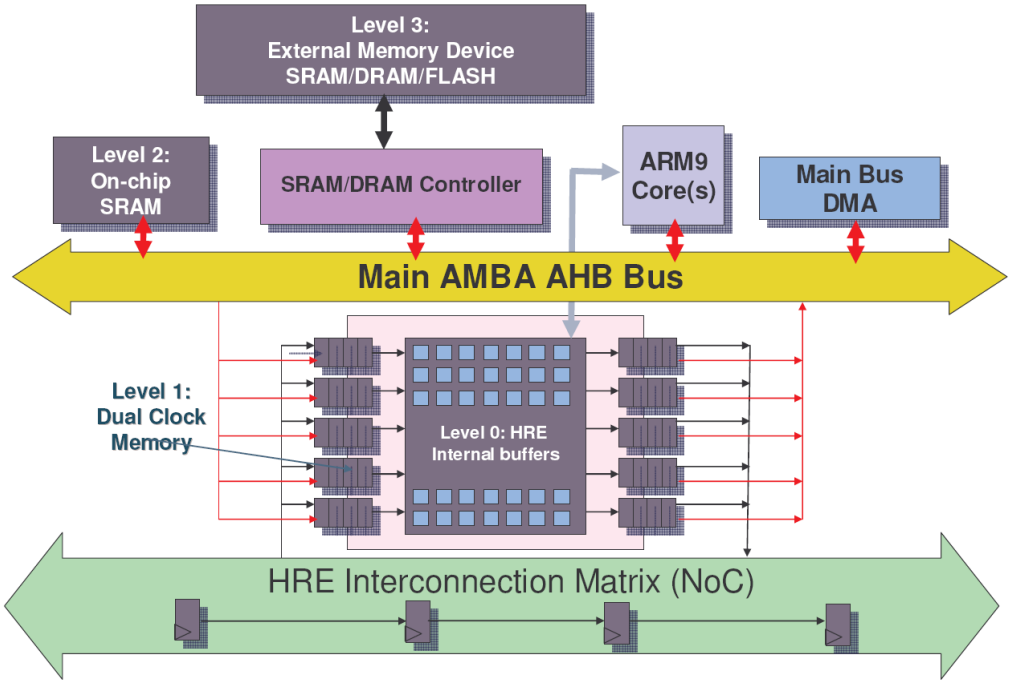


Fig. 37. MORPHEUS memory hierarchy.

The following sections give more detailed description of the memory subsystem providing quantitative specifications.

4.3.1 LEVEL 3: OFF-CHIP MEMORY

Off-chip memories serve to store large amount of data which does not fit in on-chip memory. Today's off-chip solutions support already up to few gigabytes of the storage size, however their bandwidth is about one order of magnitude smaller comparing to the memories implemented inside a SoC. Huge sizes of the storage blocks make them also much more complex, therefore a special memory controller is used to manage an access to the external data. Memory controllers contain the logic necessary to read and write dynamic RAM, and to "refresh" the DRAM by sending current through the entire device. Without constant refreshes, DRAM will lose the data written to it as the capacitors leak their current within a number of milliseconds (64 milliseconds according to JEDEC standards). Reading and writing to DRAM is facilitated by use of multiplexers and de-multiplexers, by selecting the correct row and column address as the inputs to the multiplexer circuit, where the de-multiplexer on the DRAM can select the correct memory location and return the data (once again passed through a multiplexer to reduce the number of wires necessary to assemble the system). Bus width is the measure of how many parallel lanes of traffic are available to communicate with the memory cell. Memory controllers' bus width ranges from 8-bit in earlier systems, to 256-bit in more complicated systems and video cards (typically implemented as four, 64-bit simultaneous memory controllers operating in parallel, though some are designed to operate in "gang mode" where two 64-bit memory controllers can be used to access a 128-bit memory device).

In MORPHEUS, an ARM PrimeCell Multi-Port Memory Controller (MPMC) PL175 is integrated. A choice of the given device rides on the following main features of MPMC:

- AMBA AHB 32-bit compliancy.
- Dynamic memory interface supports DDR-SDRAM, SDRAM, and low-power memories.
- Asynchronous static memory interface supports RAM, ROM, and Flash with or without asynchronous page mode.

- Designed to work with non-critical word first and critical word firs processors, such as the ARM926EJ-S.
- Read and write buffers to reduce latency and to increase performance.
- 6 AHB interfaces (+2 optional) for accessing external memory.
- 16-bit and 32-bit wide data-bus SDRAM and SyncFlash memory support. 16-bit wide DDR-SDRAM memory data support.
- 4 chip-selects for synchronous memory and 4 ship-selects for static memory devices.
- Power saving modes dynamic control.
- A separate AHB interface for programming the MPMC registers.
- Support for all AHB burst types.
- Integrated Test Interface Controller (TIC), etc.

For more details of MPMC specification, refer to [2]. All these characteristics make the memory controller to be a general purpose device with a high level of parameterization which on the one hand, gives sufficiently easy way to integrate it inside MORPHEUS SoC and on the other hand, leaves a possibility to adjust it for the concrete tasks. A modular architecture enables disconnecting of dormant interfaces and sub-blocks, thus reducing die area and saving power consumption.

Though two or more memory controllers had provided higher bandwidth, the joint integration in the SoC heavily influences not only on the chip area and power consumption, but also on the number of I/O pads and, thus, a necessity of a different package form-factor usage. Therefore, the decision is to implement only one memory controller with the following features:

- Dynamic memory interface connected to 32-bit wide SDRAM memory on 200MHz.
- Asynchronous static memory interface is disabled.
- External memory bandwidth for different types of AHB bursts, see in Table 11.
- AHB interfaces are connected to the main and configuration busses (plus 1 or 2 interfaces may be connected to the NoC). The other AHB interfaces are disabled.

- Programming and TIC AHB interfaces are connected to the main bus to be easily accessed by the main processor.
- Area occupation 276636 μm2.
- Number of I/O pads for MORPHEUS integration: 72

Table 11. An external bandwidth provided by a general purpose memory controller.

Access type	*Bandwidth MB/s Burst-1*	*Bandwidth MB/s Burst-4*	*Bandwidth MB/s Burst-8*	*Bandwidth MB/s Burst-16*	*Bandwidth MB/s Burst-32*
SDRAM Page-Hit Read access	94	265	379	482	559
SDRAM Page-Miss Read access	66	204	312	424	518
SDRAM Page-Hit Write access	664	664	664	664	664
SDRAM Page-Miss Read access	664	664	664	664	664

The bandwidth of memory controller is shared between its AHB ports. To meet given bandwidth requirements, a *TimeOut* register must be programmed. When a memory request is made to the port the value of the counter is loaded. Every cycle where the port transaction is not serviced the *TimeOut* register counts down. When the *TimeOut* counter reaches zero, the port is increased in priority. This functionality enables each AHB port to be programmed with a deterministic latency. This also enables the amount of bandwidth that a port consumes to be indirectly defined.

Thus, the reuse of the state-of-art memory controller gives a proven, highly parameterized solution, and saves the design time that might be spent on a custom device developed from scratch.

4.3.2 LEVEL 2: ON-CHIP MEMORY

In case of MORPHEUS, main system memory is dedicated to a second hierarchical memory level. Typically, this kind of storage contains the programs that are currently being run and the data the programs are operating on. In modern computers, the main memory is the electronic solid-state random access memory. It is directly connected to the CPU via a memory bus and a data bus. The arithmetic and logic unit can transfer information very quickly between a processor register and locations in main storage. The memory bus is also called an address bus and both busses are high-speed digital "superhighways". Access methods and speed are two of the fundamental technical differences between memory and mass storage devices.

In MORPHEUS, main memory acquires even more significance, since together with the data for the central processor it contains a temporal data currently used by HREs. This data has bigger size and different structure comparing to the ARM data. Moreover, the main system storage is physically separated in two parts: computational/control data, and configuration data.

On-chip memory organization depends very much on target applications and in many respects defines the performance of the whole system integrally. In section 3.1.2, it is proposed to consider all applications running on reconfigurable platform from two points of view: throughput intensive processing and reconfiguration intensive processing. The decision to choose one approach or another will be taken at the end of a design space exploration phase as a consequence of the platform architecture compared to the characteristics of the application. Thus, investigating two extreme sides of software functionality, it is possible to determine memory specifications, targeting the most sufficient trade-off.

In the throughput intensive scenario, the full image is processed by the first operator and the result is stored in an external memory. The external memory is considered since the size of the image from a typical MORPHEUS application is large enough to be stored elsewhere but the external memory. Then, this result is read from the external memory and processed by a second operator and so on. In this scenario, a typical algorithm accumulates about 20 successive operators.

In the reconfiguration intensive scenario, only a window of the image is processed by a first operator and the result is stored in on-chip memory. The size of the window is chosen so that a reasonable number of iterations will be needed to process the whole image. Then, this result is read from the internal memory and processed by a second operator and so on. After the window is processed through all the operators, the result is stored in an external memory. Then the next window is processed through all the operators and so on. In this scenario, a typical algorithm where 20 successive operators have to be applied, one image may be split into 100 windows.

Relying on these two approaches it is possible to evaluate lower borders of on-chip data and configuration memories. Consider a typical MOPHEUS application, such as film grain noise reduction for HDTV. The following parameters are taken from 3.1.2:

- Pixel resolution:
  - color channels for color images;
  - 16 bits per color channel.
- Image size:
  - 1920x1080 for HDTV.
- Frame rate:
  - 24 fps

Thus, the minimum on-chip memory size, required for the chosen application under the second scenario is described by the next equations:

$$S_{data\_\min} = W \times N_{HREs} + D_{ARM}, \tag{1}$$

$$W = F/100,\ F = \mathrm{Re}s \times N_{ch} \times Ch_{depth},$$

$$S_{conf\_\min} = 2 \times \left(S_{conf\_PACT} + S_{conf\_PiCoGA} + S_{conf\_M2000}\right), \tag{2}$$

where:

- $S_{data\_\min}$ - a minimum on-chip data memory size;
- W - a size of window;
- N_{HREs} - a number of HREs in the system;
- D_{ARM} - a size of program and data used by ARM;

- F - a size of image;
- Res - an image resolution for HDTV;
- N_{ch} - a number of color channels;
- Ch_{depth} - a color channel depth;
- $S_{conf\_\min}$ - a minimum on-chip configuration memory size which is able to store two configurations for each HRE;
- S_{conf_PACT} - PACT configuration size;
- S_{conf_PiCoGA} - PiCoGA configuration size;
- S_{conf_M2000} - M2000 configuration size;

Consequently, on-chip memory size selection strategy is presented as follows:

- On-chip data memory:
  - Lower border: $F = 1920 \times 1080 \times 3 \times 16bits = 95Mbits \approx 12MB$, $W = 12MB/100 = 120KB$, $S_{data\_\min} = 120KB \times 3 + D_{ARM} = 360KB + D_{ARM}$, refer to (1). D_{ARM} depends on the exact software and may vary from tens to hundreds kilobytes. Area occupation around 8 mm2.
  - Upper border: depends on the available area.
- On-chip configuration memory:
  - Lower border: $S_{conf\_\min} = 2 \times (72KB + 64KB + 60KB) = 256KB$, refer to equation (2) and Table 10. Area occupation around 4 mm2.
  - Upper border: depends on the available area.

4.3.3 LEVEL 1: DATA/CONFIGURATION EXCHANGE BUFFERS

Internally to each separate clock island, each reconfigurable IP has visibility and access only to its own exchange registers (XRs) for control and local memories (DEBs and CEBs) for data and configuration, as shown in both Fig. 37 and Fig. 38.

Each HRE DEB consists of a dual-port, dual-clock memory device. The "system-port" is connected to the ARM clock domain, and is accessed from the AMBA bus (or the NoC Interface). This port is used by the ARM and/or DMA unit(s) to store/retrieve data to/from the IP. The "IP-port" is connected to the HRE clock domain. It is utilized by the IP to access data for the local computation. Thus, each local memory provides a uniform mean for the system to feed inputs to and load results from the computational blocks hiding the heterogeneity of each HRE own frequency domain and internal architecture. Data consistency and access synchronization on the DEB is handled by software and is based on a programmable exclusive access policy (portions of the DEB are dynamically reserved for external access and some others to internal access, and this allocation is switched by explicit commands issued by the ARM9 processor).

In the HRE internal clock domain, DEBs are seen as an addressing space where computation inputs and outputs and temporary variables reside. According to the HRE features, two access models can be utilized (see Fig. 38).

- "Processor-oriented computation": If the HRE is capable of acting as MASTER, it independently will access the DEB "IP-Port" (on the regions indicated as safe by the ARM core. These are the regions not currently accessed by the "System Port"). When results are available, the IP will notify the occurrence to ARM, and the portion of memory holding results will be then accessed by core/DMA/NoC and become unavailable for the IP. This configuration is more suitable for applications where the addressing patterns depend on the processed data.
- "Stream-oriented computation": If the addressing pattern for the input data contained in the DEB is regular, it is possible to obtain higher performance relieving the IP from the addressing burden, configuring the same IP to perform computation as a data-crunching SLAVE without addressing capabilities. For this reason, a set of programmable address generators can be added on the "IP port" to each DEB bank in the HRE, thus ensuring a higher data bandwidth through the IP. Each address generator can be programmed independently by the ARM core and provides specific vectorized addressing patterns.

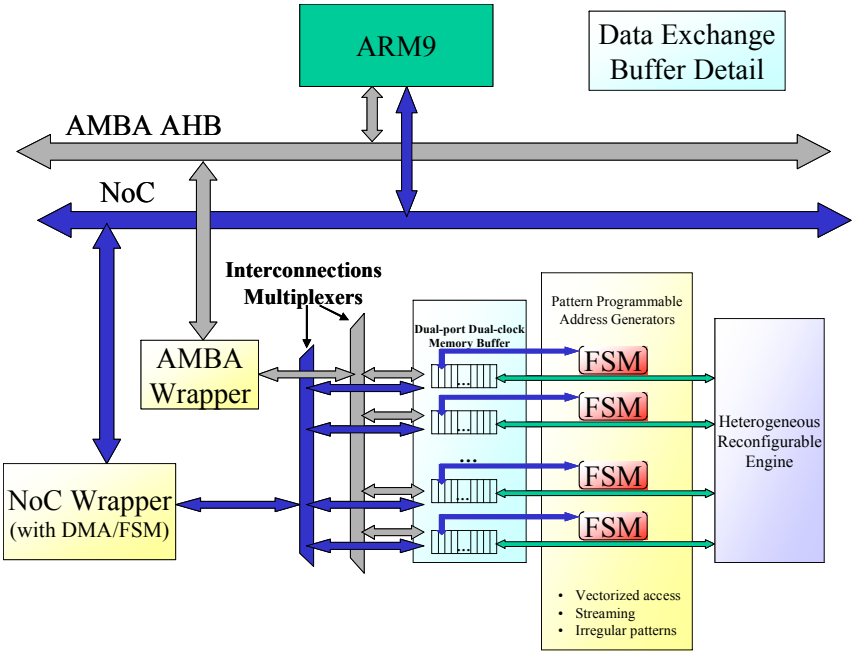


Fig. 38. Clock domain data storage organization.

4.3.4 EXCHANGE REGISTERS

All synchronizations with the host system are ensured by asynchronous interrupts and a cross-domain eXchange Registers (XRs). Two types of the XRs are implemented:

- *Hardware oriented registers* handle specific hardware signals such as reset, interrupts, clock enable, and other control signals. HRE can issue only interrupt signals to the host system, one part of which is related to the predefined services and another is managed by the user. The host system issues three types of signals to HRE via hardware oriented registers, they are: 1) computation control; 2) clock control (clock enable, clock select, and PLL control); 3) stream computation control.
- *Software oriented registers*, on the contrary, are managed by the user. Their contents and meanings are completely programmable from both host system side and HRE side.

Structurally, all HREs have similar XR organization. Namely, each XR is a 32-bit word: the higher 16 bits are reserved for HRE-to-HOST communication, so they can be read by both but can only be written from the HRE side; the lower

16 bits are reserved for HOST-to-HRE communication so they can be read by both but can only be written from the host system side. The following tables describe the functionality of the XRs:

4.3.4.1 HARDWARE ORIENTED REGISTERS

XRh0: COMPUTATION CONTROL REGISTER

Address Offset + 0x000 Physical control of the HRE

31:17	16	15:3	2	1	0
0	End of computation (HRE Idle request)	0	HRE enable '0': HRE Idle '1': HRE Active	HRE resetn 2	HRE resetn 1

XRh1: CLOCK CONTROL REGISTER (1)

Address Offset + 0x004 PLL Handling

31:17	16	15:4	2	1:0
0	PLL_LOCK: '0' PLL powered down or unlocked '1' PLL Locked, clock can be used for HRE	PLL Multiplication Factor Definition P VALUE	PLL PowerDown '1': Switch on PLL '0': Power Down PLL	Clock MODE (IN) 00 Stuck at gnd 01 Stuck at gnd 10 External clock 11 PLL Output

XRh2: CLOCK CONTROL REGISTER (2)

Address Offset + 0x008 PLL Handling

31:16	15:8	7:0
	PLL Multiplication Factor Definition M VALUE	PLL Multiplication Factor Definition N VALUE

XRh3: DATA STREAMING CONTROL REGISTER

Address Offset + 0x00c 4-Way Streaming computation control and synchronization

31 :28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
	HRE_LOCK		ARM_LOCK_ACK		HRE_LOCK_ACK		ARM_LOCK

XRh4: INTERRUPT REGISTER

Address Offset + 0x010 Interrupt Signals

31:17	16	15:0
0	HRE: Printf Dump Request	0

XRh5: INTERRUPT ACKNOWLEDGE REGISTER

Address Offset + 0x014 Interrupt Acknowledge Signals

31:16	15:1	0
0	0	DREAM: Printf Dump acknowledge

4.3.4.2 SOFTWARE ORIENTED REGISTERS

In this case the functionality is not specified in hardware but the programmer is free to make any use of them via software.

XRsn : Address 0x20 + n*4 General Purpose XRs

31:16	15:0
User Defined (HRE2ARM)	User Defined (ARM2HRE)

4.4 COMPUTATIONAL MODEL

MORPHEUS architecture was designed to process streaming data-flows under given real time constraints. Fig. 39 depicts a general view of the data-flow in the MORPHEUS platform. The computational demands of the target application are manually partitioned over available computational units – HREs. The aim of the mapping task aims at building a balanced pipeline flow in order to induce as few stalls of HREs as possible in order to sustain the required run-time specifications. It should be noted that the order and the direction of the traffic between the HREs (and the I/O facilities) is completely flexible and can be easily adapted to various applications. Also the bandwidth related to a given data stream may change significantly throughout different stages of the same computation.

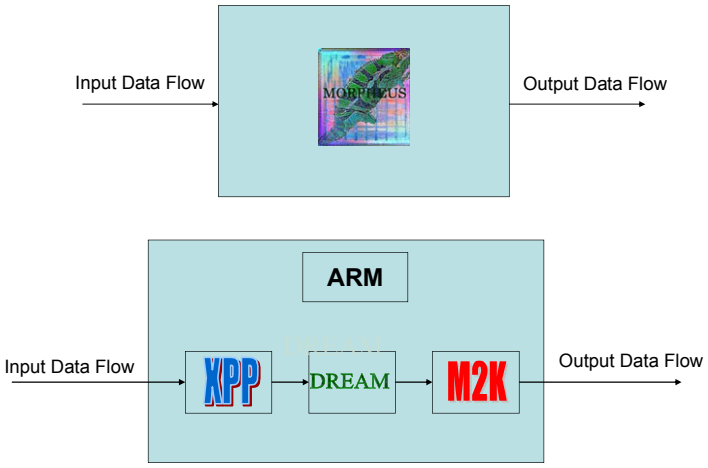


Fig. 39. General view of the data-flow on the MORPHEUS platform.

In such a way, there are the following requirements for the application mapping:

- The computational kernels should be *distributed* as much as possible among the three different HREs in a *balanced* way (the throughput of the overall system will be that of the slowest HRE), depending on the mapped application.
- Data traffic between HREs (and I/Os) should ideally not induce stalls and thus should be hidden by the computation (of course this may depend a lot

on the time correlation imposed by the application between different data elements in the stream).

The fundamental task for the MORPHEUS application designer is to develop a balanced pipeline where data transfers and computation can interact concurrently without inducing bottlenecks and/or stalls. Fig. 40 provides a generic example of application mapping on MORPHEUS, utilizing only two HREs for simplicity. The example makes it evident how the overall performance will be driven by the slowest stage, where a stage can be induced by either computation or data transfers.

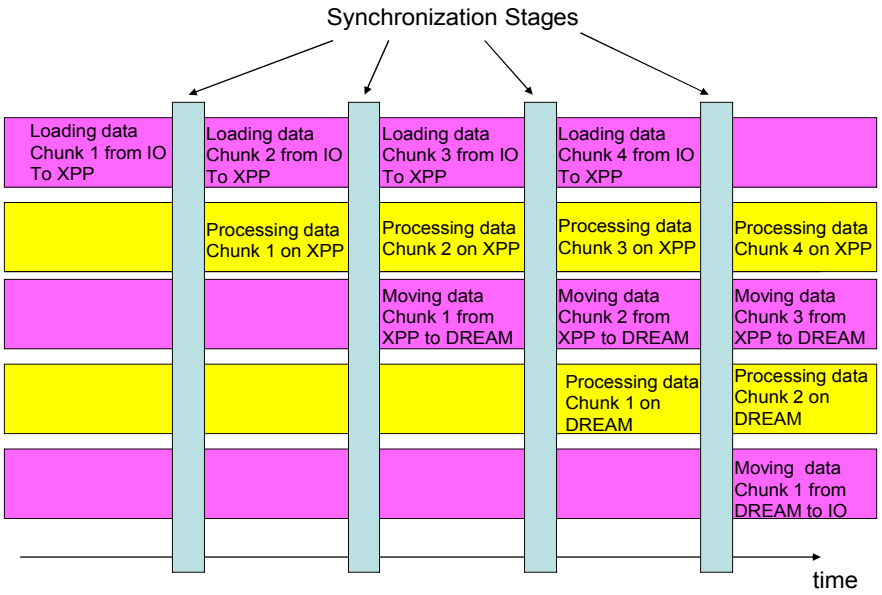


Fig. 40. Example of a possible data flow organization on the MORPHEUS architecture.

On the other hand, due to the intrinsic heterogeneity of the HREs, the *timing budget* of each stage is extremely flexible, and can be determined by the user, much depending on the application features. It is likely that the transfers will be more flexible and it will be easily adaptable with respect to computation in the design optimization phase.

The granularity of data-chunks is the size of either a single computation event on the HRE or the specific transfer in the communication infrastructure. From the hardware point of view, this is adaptability is maintained both in terms

of addressing details and size to allow the user the widest possible design space in mapping the application.

4.4.1 MATHEMATICAL REPRESENTATION

In order to deploy the concept of the *synchronization stage* in the streaming data-flow without applying strict constraint on the size and nature of each application kernel mapped on the HREs, the overall computation flow can be modeled according to two different design description formalisms:

- Petri Nets (PN) [37];
- Khan Process Nets (KPN) [45], [69].

In the first case, the synchronization is made explicit, as each computation node in the network is triggered by a specific set of events. In the second case, the synchronization is made implicit by the presence of FIFO buffers that decouple the different stages of computation/data transfer.

In fact, KPN are mostly suited to "hardwired" implementation since the dimensioning of FIFOs is very critical to avoid stalls, but it is entirely related to the application. In such reconfigurable architecture as MORPHEUS, the application of the KPN processing pattern may require some tuning of the kernels granularity to match the granularity of FIFOs.

The choice of the most suitable design formalism for a specific application depends on both the targeted application features and the nature of the targeted HRE. Since it is possible to model a KPN through a PN (but not vice versa), from the hardware point of view the PN model has been kept as a reference in the design, although full support for KPN-oriented computation is maintained.

One important detail of the computation deployment is that the MORPHEUS architecture is a completely programmable device. That is why each HRE must be configured before starting the computation. The configuration stage of a given HRE must be considered as a *triggering* event for computation in the context of Petri Nets. Therefore, a pure KPN pattern can not be applied to MORPHEUS, unless the case where the configuration is completely static (all HREs and transfers are programmed only once in the application lifetime). In case of dynamic reconfiguration (the number of nodes in the KPN/PN is higher than

the available HREs) KPN can be implemented as sub-nets, or second level nets of a larger PN triggered by the configuration events. In this case, the application nodes must be time-multiplexed and scheduled over available HRE. It is possible to define this computational model as a controlled Petri net, and the configuration must be considered as one of the triggering events for computation.

Looking more precisely on the HREs, the XPP array is suited for a KPN-oriented flow because its inputs are organized with a streaming protocol. Unlike the XPP array, DREAM is a computation intensive engine: typically, input data is iteratively processed and written back in DEBs with the generation of a significant share of temporal results. Thus, computation on DREAM can be more appropriately described as a collection of *iterations* triggered by specific events, rather than a purely streaming computation. Therefore, PNs are more suitable to model its computation and its interaction with the rest of the system. Finally, M2000 is an eFPGA device and, from the architectural point of view, any computation running on it can be modeled according to either formalism.

In order to keep the *balance* between computation kernels and relative data transfers to fit in the "extended heterogeneous pipeline" concept, each HRE is provided with local proprietary input and output buffers – DEBs. The DEBs provide the synchronization stage described on Fig. 40. According to the information described above, XPP DEBs are modelled as FIFOs, while the DEBs connected to DREAM and M2000 are run-time programmable and support both FIFO mode and random memory access mode. In order to exploit the hardware features of each HRE at their maximum performance, the MORPHEUS chip is organized according to the Globally Asynchronous Locally Synchronous (GALS) clock distribution model. Each HRE is provided with a local programmable PLL. DEBs are implemented as dual-port, dual-clock memories (DPDC) in order to implement the clock domain crossing (this concept is presented in more detail in section 4.3.3).

4.4.2 ORGANIZATION OF DATA-FLOW IN THE SYSTEM

From the computational point of view MORPHEUS architecture provides two types of I/O:

- Very high speed I/O channel is provided by the external memory controller. The device was added to the design in order to sustain the peak requirements of the most demanding applications (for the quantitative specifications refer to section 4.3.1). In particular, the controller ports are organized with input/output reordering buffers in order to enable the controller utilization in the context of a KPN-oriented data-flow.
- On the other hand, the M2000 eFPGA fabric is supported with the direct connection to I/O pads in order to guarantee the deployment of low-bandwidth but configurable specific hardware protocols (e.g. various external interfaces).
- Other standard I/O protocols (IEEE1254, UART, JTAG) are available as part of the ARM processor environment but, their utilization is only planned as control/debug facility and not as part of the MORPHEUS data-flow.

Data-flow in MORPHEUS, can be represented as a set of synchronized data transfers from I/O, through the DEBs, possibly through on-chip memory, and finally to I/O again (see Fig. 39). As described in section 3.3, there are two physical means for data transfer:

- A multi-layer AMBA bus, which is used for all control, synchronization and configuration of the system components but can also be used for transferring data at low bandwidth.
- A communication infrastructure based on a Network-on-Chip.

According to the PN/KPN concept each node in the computation network must be provided with the means of forwarding its result to the following node, possibly in a concurrent way in order to avoid bottlenecks and exploit parallelism. For this reason every HRE Network Interface (HRE-NI) in the NoC is provided with an embedded DMA-like data transfer engine. The user can organize data-flow according to three different approaches:

1. The ARM processor acts as “full-time” traffic controller. The code running on ARM monitors the status of each HRE through the exchange registers (XRs) and triggers the required transfers over the HRE-NIs in order to maintain the desired stream through the system. This is useful in

the first stages of application mapping in order to evaluate the cost of each step in the computation, maintain full programmability, and find bottlenecks.

2. The ARM processor acts as "batch" controller and enabler. After the configuration phase, when ARM programs all HREs and relative transfers on the HRE-NIs, it goes to the idle mode waiting for interrupts. By handling XRs, ARM maintains the synchronization between the various stages of the pipeline described in Fig. 40. This approach is useful in case of controlled computation network (application that requires dynamic reconfiguration to schedule different PN nodes over the same HRE) or in any other case when the user prefers to utilize a PN, which is an event driven network on contrast to KPN.
3. The modeled network is self-synchronized: the ARM processor only provides the initial configuration phase, and after that the HRE-NI will iterate over circular buffer addressing implementing a fixed data-flow through the system. This case can be modeled for static applications or, most likely, for a limited time-share of the application as a second level KPN, included in a larger PN network.

A sample of the C control source code for the ARM processor, managing computational data exchange with the DREAM and M2000 HREs, is presented in Appendix A.

4.4.3 NETWORK-ON-CHIP AS THE DATA COMMUNICATION MEAN

The proposed NoC topology is presented in Fig. 41. It consists of eight nodes, each of which is connected to a computational or a storage unit. Note that PACT XPP HRE and CMC (Central Memory Controller) use in twos network nodes due to their architectural specifics.

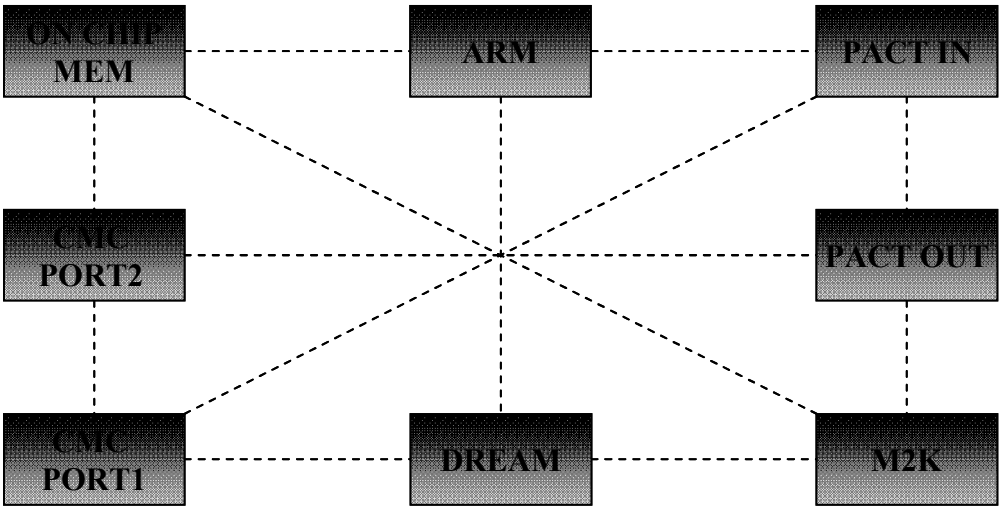


Fig. 41. NoC Spidergon topology.

In order to provide ARM processor with the "full-time" control over data traffic in the system, the generic NoC communication concept requires a slight refinement. NoC is by definition a distributed communication infrastructure with a set of *initiator* nodes (e.g. computational cores) issuing transfers, and a set of *target* nodes providing information storage and responding to the transfer request (e.g. memory units). Consequently, from the NoC point of view the nodes have peculiar functionality simultaneously combining initiator and target behaviors. This is implemented in the NoC by means of so called "distributed DMA" pattern: in order to act as a traffic initiator, a Network Interface of each HRE node (HRE-NI) in the NoC is enhanced with a local data transfer engine defined as a *Local DMA* (see Fig. 42). Local DMAs also feature flexible addressing patterns that include step/stride and circular buffer functionality. NIs are capable to load data chunks from HREs and store them through the NoC into the target repository and vice-versa. From the core/user point of view this concept enhances the NoC with an enlarged and highly parallel DMA architecture. The user has possibility to handle computation on HREs as C-level functions mapped on a specific processing unit. Operands for this function are referenced by their DMA transfer information, composed by base address and addressing pattern details.

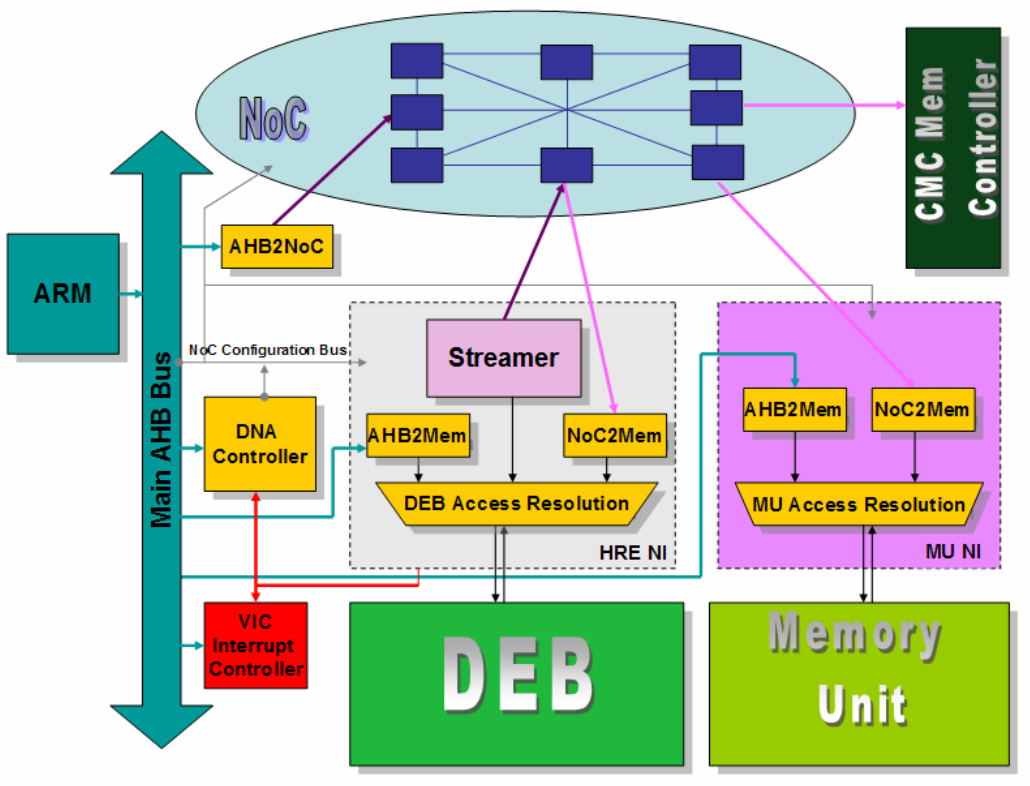


Fig. 42. Initiator and target HRE-NI.

Data Transfers are initiated by ARM through specific configuration registers on the HRE network interface. The configuration process is performed via a dedicated NI configuration channel reaching all HRE NIs, mapped as a slave on the AMBA bus. The HRE NIs can support multi-channel transfers with variable priority scheme, also programmed through the same configuration channel. End-of-transfer notification for each channel in the HRE NI can be read both as a status register or handled as interrupt by the core.

Programming of the distributed engines integrated in the NoC is handled by means of C-based drivers. They support single transfers and multi-block transfer for stream access, such as:

- Auto-reload Multi-Block transfer
- Auto-reload Multi-Block transfer with contiguous Source address
- Auto-reload Multi-Block transfer with contiguous Destination address

A channel is selected and programmed using two C structures called respectively *config* and *lli.* For each one several parameters are defined. Refer to Appendix A for the single- and multi-block transfer source code examples.

As described in section 4.4.2, quasi-static data transfers are expected to be by far the greatest volume of communication load on the network. This positively affects the overall system control performed by ARM under the RTOS, since block transfers can be handled using DMA methodology. In addition to data transfer control, system synchronization activities may limit the application performance. Since the MORPHEUS platform is a single master system, it may suffer synchronization overload due to a tremendous amount of interrupt sources (dependent on the granularity of the tasks, mapped on the HREs). A possible solution is to offload ARM from a part of the RTOS services by mapping them on a dedicated hardware. Such network controller, DNA (Data Network Access), besides the setup mechanisms for data transfers, features a hardware handshaking protocol to support synchronization between communication and computation. Its programming model is very similar to a common DMA interface.

The DNA is programmed by ARM through AMBA AHB. Eight parallel programmable channels are supported, each one able to control multi-block transfers. The DNA programs NoC transferes by means of configuring HRE-NIs. In spite of introducing an additional latency, this approach provides a twofold benefit. On the one hand, the DNA can handle interrupt-based multi-block transfers with block sizes of up to 128KBytes by monitoring channel states and STNoC interrupts. One particular and most commonly used multi-block mode is multi-block chaining. Thereby configuration items for channel setup can be stored in a dedicated memory beforehand and will be pre-fetched and executed by the DNA independently from further ARM interference. In addition, the auto-reload mechanisms can be utilized to support the streaming communication approach.

On the other hand the hardware handshake mechanism acting between the HREs and the DNA has been implemented can automatically handle the workload due to consistency synchronization of transferred data.

4.4.4 KPN MODELING

An important requirement for building efficient KPN is that each node of the net is implemented on independent channel and that each channel is implicitly synchronized by FIFOs. If the target application corresponds to the KPN formalism, it is relatively easy to map it over MORPHEUS platform. All HREs

and the memory controller feature hardware FIFOs are mapped on the chip addressing space. The interconnect infrastructure was designed in order to provide independent channels for each XPP port (four 16-bit inputs and four 16-bit output FIFOs) towards the memory controller. Moreover, each channel is featured with dedicated data transfer engine, which allows maintaining of full bandwidth at the expense of a marginal control from the side of the ARM. This is not necessary for M2000 as it was evaluated that the maximum computational bandwidth that the device can provide both as computation and I/O engine can be served by a single 32-bit data transfer channel (see section 1.1).

As it was mentioned in section 4.4.2, ARM configures all HREs and all data-transfers on each HRE-NI and can essentially remain idle during the computation phase, while FIFOs will implicitly synchronize the data-flow. The NoC was dimensioned in order to minimize congestion on critical nodes such as the memory controller to XPP link. If the circular addressing is not possible on the I/O lines it will be necessary to add a synchronization stage with an interrupt line to ARM in order to refresh the HRE-NI transfers.

As for DREAM, a pure KPN approach is not applicable due to the HRE internal structure. On the other hand, it is always possible to describe a KPN through a PN. In such a way, it is possible to emulate a FIFO data exchange model through the DREAM DEBs with a specific interaction between the DREAM embedded processor and ARM, exploiting interrupts and XRs. This will induce a little overhead in the data-flow and require an interaction with ARM but from the theoretical point of view the KPN model will be maintained.

4.4.5 PN MODELING

As it was presented above, in some cases it is convenient to manage the computation on different HREs as a set of iterative executions over a given set of data. The data must be locked by the HRE for a given time before being released for the successive computation stage. During the computation of an input stream, this process will be repeated iteratively; therefore, it is necessary to abstract this computational model by building a sort of “virtual streaming flow”. The data flow is maintained consistent across the successive iterations with the “supervision” of the controlling processor, implementing a PN.

This concept is realized by means of utilizing the XRs and interaction between the HRE and the main program running on ARM, to build a mechanism defined as *ping-pong buffering*. Each computation is intended as a node of the PN, and the data chunks required by the iteration (that in this case must be finite, although they can be portion of an infinite stream) are associated to preceding/successive tokens. The rules of a generic PN can be briefly described as follows:

- A given node can be triggered when:
    - All preceding nodes are available (have terminated computation);
    - All successive nodes have read the results of the previous computation.

  In the context of MORPHEUS these rules can be rewritten as follows:
- A given computation can be triggered on a given HRE when:
    - The bit-stream for the application was successfully loaded;
    - All input data chunks have been successfully uploaded to the HRE local DEB;
    - All output data chunks that would be rewritten by the current iteration have been successfully copied from the HRE local DEB to their respective destination(s).

Hence, ARM programs the PN consistency and produce the preceding/successive tokens triggering a given computation stage. Of course, if data chunks are large enough, this event monitoring step will not be required very often.

Each HRE computation event will be applied to a finite input data chunk, and will create an output data chunk. In order to ensure maximum parallelism, during the HRE computation event N, the HRE-NI should load input chunks N+1, N+2, N+3 filling all available space in the DEB but ensuring not to cover unprocessed chunks. Similarly, it should download output chunks N-3, N-2, N-1, ensuring not to access to chunks not yet processed.

4.4.6 LOCAL DATA SYNCHRONIZATIONS

4.4.6.1 COMPUTATIONAL DATA

The synchronization is based on signal handshakes from ARM to the HRE, so that it can be performed by either a local FSM or by a software procedure mapped on the ARM processor. The handshake signals are controlled by XRs (refer to section 4.3.4). The mechanism (referred as *ping-pong buffering*) is based on the concurrent switching of two virtual buffers (VBUF) mapped on the DEBs. The VBUF can be a part of a DEB, a whole DEB or a set of DEBs. It is only referred by the start and finish address over the DEB addressing space. The idea is that while the first VBUF is being accessed by the host system (Write/Read from ARM/AMBA or from the NoC infrastructure), the second VBUF is being accessed by the HRE (see Fig. 43). The two processes should be concurrent. When both accesses are finished, the two VBUFs should be switched. Altogether, ARM, DREAM and XPP are software programmable machines (only M2K can be considered as a hardware programmable device). Thus, any signal between the two can be handled via a specific software handshake

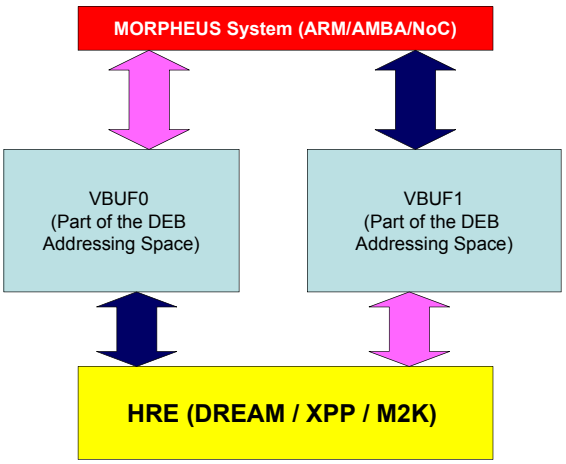


Fig. 43. Representation of the ping-pong buffering.

Fig. 44 shows the synchronization mechanism between the host system and the HRE. Before accessing the virtual buffers, the host processor sends the *lock* signal to the HRE and waits for *acknowledge*. When *acknowledge* is received, the ARM processor can start load/store procedure from VBUF0, while

the HRE is elaborating the VBUF1. After finishing with VBUF0, ARM releases the buffer issuing the *UnLock* signal. In its turn, the HRE sets the *UnLock acknowledge* after finishing with VBUF1. Then, the whole synchronization procedure is repeated with new data stored on the VBUFs.

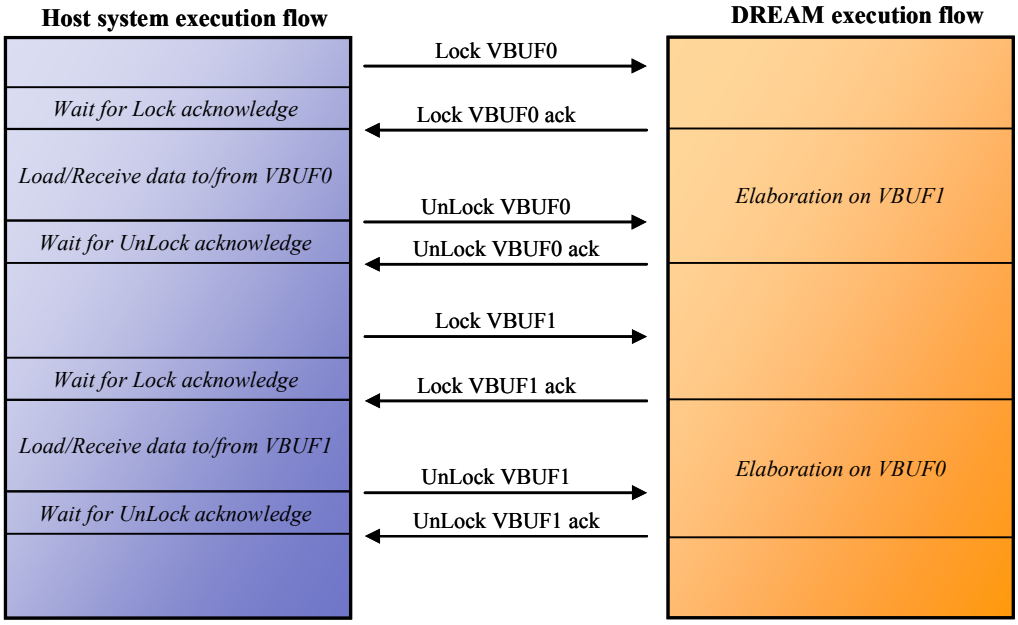


Fig. 44. Synchronization scheme.

Fig. 45 and Fig. 46 show an example of the ARM C-code that implements the handshake procedure, and the corresponding C-code running on DREAM.

```
for(block=0;block<BLOCK_SIZE+2;block+=2) {

   // Load/retrieve VBUF0

   qprintf("Block %d \n",block,0,0,0);

   dream_XR->arm_lock0=1;while(dream_XR->arm_lock_ack0!=1);

   if(block<=BLOCK_SIZE-1) load_debs  ((int*)dream_debs->pair01, DEB64, 0, FRAME_SIZE ,
(int)((int)frame+(block)*64));

   if(block>=2) retrieve_debs((int*)dream_debs->pair23, DEB32, 0, FRAME_SIZE/4,
(int)((int)result+(block-2)*16));

   dream_XR->arm_lock0=0; while(dream_XR->arm_lock_ack0!=0);



   // Load/retrieve VBUF1

   qprintf("Block %d \n",block+1,0,0,0);

   dream_XR->arm_lock1=1;while(dream_XR->arm_lock_ack1!=1);

   if(block<=BLOCK_SIZE-1) load_debs  ((int*)dream_debs->pair45, DEB64, 0, FRAME_SIZE ,
(int)((int)frame+(block+1)*64));

   if(block>=2)retrieve_debs((int*)dream_debs->pair67, DEB32, 0, FRAME_SIZE/4, (int)((int)result
+(block-1)*16));

   dream_XR->arm_lock1=0;while(dream_XR->arm_lock_ack1!=0);

  }
```

Fig. 45. ARM C-code representing HW handshake.

```
while(XR->arm_lock0!=0); XR->arm_lock_ack0=0;

while(1){

   // Process current VBUF0 while ARM is working on VBUF1

   while(XR->arm_lock1!=1);XR->arm_lock_ack1=1;

   process_frame(16,17,18,0);

   while(XR->arm_lock1!=0);XR->arm_lock_ack1=0;



   // Process current VBUF1 while ARM is working on VBUF0

   while(XR->arm_lock0!=1);XR->arm_lock_ack0=1;

   process_frame(20,21,22,0);

   while(XR->arm_lock0!=0); XR->arm_lock_ack0=0;

  }
```

Fig. 46. DREAM C-code representing HW handshake.

4.4.6.2 CONFIGURATION BITSTREAMS

The AMBA bus hierarchy features a sub-bus for handling configuration bitstreams on the HREs (refer to section 4.2.3). Bitstreams are seen by the programmer as predefined libraries, either provided by the IP vendors or synthesized by the MORPHEUS toolset. The binary format - produced by the IP toolset - is integrated into ASCII ANSI-C source code by means of the C *#include* directive, where the bitstream is described as an integer vector, and is compiled in the main program. In this way, the bitstream is handled as part of the main code. During computation, the bitstream can be referenced by ARM through its address and its size. ARM may load the bitstream word by word with load/store operation, or more conveniently utilize DMA transfers. In order to avoid traffic congestion on the main AMBA bus, that should be reserved for control operations (XRs handling, NoC transfer requests), ARM may utilize the configuration DMA residing on the configuration bus. The configuration DMA may access the bitstream on the off-chip or on-chip configuration memories, and write on the local HRE configuration buffers. XPP and DREAM feature self-configuration capabilities to ensure high rates of dynamic run-time configurability, so they will

access the CEB according to their internal status, negotiating bitstream transfer requests with ARM via the XRs. On the contrary, M2K does not feature any mean for self-reconfiguration, hence it does not require any CEB, and configuration bits are written directly on the M2000 bit-stream input port.

During the peak computation, it is expected that a dynamic monitoring of the HRE status and the handling of their reconfiguration requests could be too demanding for the ARM core, which may not be capable to handle such occurrences in real-time conditions, thus inferring stalls on the reconfiguration process. For this reason, hardware logic has been added as master of the configuration bus, called "Predictive Configuration Manager (PCM)". The PCM can be considered as a coprocessor that will not only leverage the core processor with low level bitstreams transfers and configuration memory hierarchy management when the reconfiguration directives are issued, but will also offer pre-fetch prediction services. Fig. 47 describes a short example where a bistream is loaded over the DREAM HRE. *Load_dream_bitstream* is a library function that handles DMA transfers over the configuration bus. Similar libraries are available for all HREs.

```
File interpolate.dreambistream.h:


// DREAM Data Memory content
unsigned int dm_segment[] = {
0x00000200,0x00E00000,[…]
int dm_size = 41;
unsigned int pm_segment[] = {
0xE000021E,0xE000111E,[…]
int pm_size = 1838;
unsigned int cm_segment[] = {
0x00A50001,0x02A50221,[…]
int cm_size = 5612;
```

```
Main Program:



#include "interpolate.dreambitstream.h"

// Load bitstream on DREAM CEBs (Note DREAM Does not need to be active)

 load_dream_bitstream(0, (unsigned int*)cm_segment,cm_size,

              (unsigned int*)dm_segment,dm_size,

              (unsigned int*)pm_segment,pm_size);
```

Fig. 47. SW support for the configuration load procedure.

4.5 HRE INTEGRATION BY THE EXAMPLE OF PICOGA

This section presents the design strategy and implementation of the embedded reconfigurable device (named *DREAM*) for heterogeneous SoCs. The IP targets data intensive computation exploiting a medium grained multi-context run-time reconfigurable unit PiCoGA (described in section 3.5.3) to build custom pipelined function accelerators. DREAM IP block should be distinguished from the PiCoGA unit, which is actually only a reconfigurable part of the DREAM. The design is completed with a full software tool-chain providing the application algorithmic analysis and design space exploration in an ANSI C environment using cycle accurate simulation and profiling.

4.5.1 INTEGRATION STRATEGY

DREAM was designed in order to function as one of HREs in the MORPHEUS platform. However, its generalized design strategy allows utilizing DREAM as a stand-alone IP in other reconfigurable platforms controlled by conventional RISC processor. The following design specifications aim at supporting this feature:

- homogeneous integration approach;
- efficient communication with the host system by means of local storage sub-system;
- local clock and power managements detached from the host system domain;
- programming model, compliant with high-level languages.

The homogeneous integration was achieved by organizing DREAM as a microprocessor, providing the uniform architectural approach and programming model. Control and communication with the host system is handled by the embedded RISC processor, which guarantees maximum flexibility and well known usability. The computational data flow is exchanged with the main system by means of communication buffers, accessed independently and concurrently by the host system and the local data path (see Fig. 48). The data consistency in these buffers is maintained in software in the manner presented by *Molen paradigm* [74].

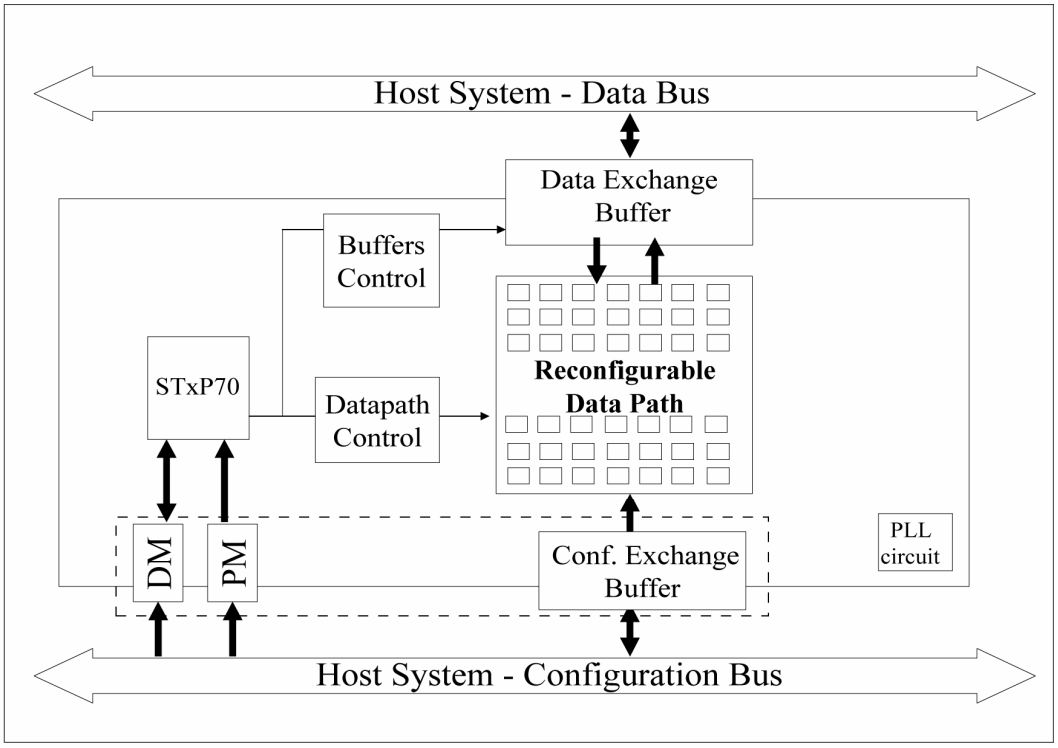


Fig. 48. DREAM integration.

Reconfigurable architectures allow hardware resources dynamically adapt to the degree (data granularity) and nature (SIMD, MIMD) of parallelism given by the application. The reconfigurable architectures (RAs) with bigger granularity provide the required computational density to exploit such parallelism at maximum performance. Therefore, it is important to provide a data communication mechanism capable to support the available computation bandwidth. In such a way, the IP should have enough capabilities to address local buffers with the required parallelism and feed the reconfigurable data-path with computational data, while maintaining flexibility such as of a standard processor. This is even more significant since the chosen data-path features scarce storage support, compensated by zero time overhead switch between multiple contexts and a relatively small size. Many applications require multiple iterations of different operations on the same set of data. This is realized in DREAM by a flexible access to the local memory, i.e. directly reading and writing from data buffers in a more efficient way with respect to a purely streaming pattern. Another issue is that DREAM I/O buffers most probably will not contain ordered data. Whereas embedded applications, and especially those who mostly benefit from

mapping on RAs, typically feature kernels based on regular addressing patterns. A convenient way to retrieve data at high parallelism, commonly used in state-of-the-art highly parallel DSPs is to utilize programmable vectorized and modulo addressing generation FSMs. In DREAM, this is achieved by means of programmable Address Generators (AGs) that are set at the beginning of each computational kernel and produce a new local address at each request from the data-path.

Moreover, the local buffers serve as a bridge between IP and host system clock domains. The use of dual-clock dual-port memory modules assures stable and convenient – from implementation point of view – boundary. The synchronization mechanism provides data consistency in the buffers and controls accesses to both ports. The most suitable synchronization mechanism for such organization is *ping-pong buffering* (or double-buffering), when the buffer is virtually separated in two parts. While data is being processed in one part, the next set of data is read into another virtual part. The synchronization is performed on the software level using dedicated eXchange Registers described in section 4.3.4.

The local buffers isolate the integrated IP from the host system enabling an on-site *dynamical power management* (DPM), such as *Dynamic Voltage and Frequency Scaling* (DVFS), and *Sleep Mode*. A clock control circuit allows dynamical trading of energy consumption with computation speed, depending on the required data processing bandwidth, without any impact on the working frequency of the rest of the chip. In this context, *voltage level shifters* could be used to create a mixed-voltage system enabling implementation of DPM schemes.

As it was outlined in section 3.5.3, computation on the reconfigurable data-path is based on *macro-instructions* organized in a Data Flow Graph (DFG). After an automated extraction of the Instruction Level Parallelism (ILP), each DFG is mapped on reconfigurable hardware as a custom pipelined unit working at the top processor frequency with variable latency and issue delay. Hardwired pipeline control logic preserves data dependencies described by the DFG. It is implemented as a Petri net [37]. Each configuration issue and/or execution of a given macro instruction must be explicitly triggered by the user. The main processor controls both data and configuration transfers. Zero reconfiguration

overhead of DREAM prevents long computation stalls. Moreover, the RISC core handles the control services: reading/writing XRs, programming AGs and sending interrupts.

4.5.2 CONTROL UNIT

Control tasks in DREAM are mapped on a 32-bit RISC processor – the STMicroelectronics STxP70 internal core, which is responsible for instructions fetch, program flow handling, and providing appropriate control signals to the other blocks. These signals are generated by particular coprocessor operations. The processor is relatively small, being composed of 20K gates of logic plus a specific memory module acting as an embedded 32-slot register file (RF). It features arithmetical-logical operations, 32-bit shifts and a small embedded multiplier.

Synchronization and communication between the IP and main processor of the host system is ensured by asynchronous interrupts on the local core, and a cross-domain XRs. Program code and program data for STxP70, as well as the configuration bitstream for PiCoGA, are considered as parts of the DREAM programming pattern. The configuration bitstream is loaded by the host system on the CEBs, implemented on dual-port, dual-clock memories of the total size 36KBytes. The program code and data are transmitted to STxP70 through the Program Memory (PM) and the Data Memory (DM). PM and DM are also implemented on dual-port, dual-clock memories of the size 16KBytes each. Input data and computation results are exchanged through the DEBs using a coarse-grained handshake mechanism (defined as ping-pong buffering).

There are two DPM techniques supported by the IP block: DVFS and Sleep Mode. They are performed locally and do not affect the host system voltage and frequency, provided that the set of voltage level shifters are implemented. The decision about the voltage scaling or entering the Sleep Mode in the IP is made either centrally, by RTOS relying on activity scanning results, or adaptively, by the STxP70 itself relying on the idle time conditions. For this purpose, the STxP70 supports up to four IDLE modes for fine clock domain control and more than 80% gated clocks.

The choice of utilizing a small processor allows the user to exploit a sophisticated program control flow mechanism, writing commands in ANSI-C, and utilizing a reliable compiler to optimize code and schedule task efficiently. The processor function accelerators can also act as computation engines in some cases, concurrently to the reconfigurable data-path. Computation kernels are re-written as a library of macro-instructions, and mapped on the reconfigurable engine as concurrent, pipelined function units. Computation is handled by the STxP70 in a fashion similar to the Molen paradigm: the core explicitly triggers the configuration of a given macro-instruction over a specific region of the data-path, and when the loading of the configuration is complete, it may run any desired issue of the same functionality in a pipelined pattern. Fig. 49 shows an example of a function that configures the data-path, executes a loop and releases the configuration. Up to four macro-instructions can be loaded on each of the four available contexts. These contexts can not be computed concurrently but context switch requires only two clock cycle. A sophisticated stall and control mechanism ensures that only correctly configured operations can be computed on the array, and manages context switches.

```
my_function() {
 PD=Set_Configuration(picoga_op);
 IN=Set_Interconnect_Matrix_Channel(buffer,port);
 Program_AG(buffer,base,step,stride,count,mask,rw);
 for (i=0; i<N; i++)
  Execute(PD,IN);
 Wait_for_Pipeline_Empty();
 Unset_Configuration(PD);
}
```

Fig. 49. Example of the programming code for DREAM.

In addition, STxP70 features a hardware loop mechanism. It allows zero overhead execution of branches when looping over a predefined code segment. It gives the benefit in terms of the latency needed for the branch execution, or in terms of the costs for building the branch prediction tables. Up to two independent and symmetric hardware loops are possible with independent loop counters and

programmable loop-entry and loop-exit addresses. The predefined loops may be nested or not.

For the test purpose, on-chip emulation with standard low pin count JTAG interface is implemented. It provides the in-line debugging capability similarly to the standard on-chip emulator (OCE). In such a way, a programmer can use well-known debugging tools and techniques for comfortable interaction with the IP.

4.5.3 ADDRESS GENERATOR

The solution is based on the stream memory architecture proposed to be implemented in PiCoGA based reconfigurable device. A MUX-based, run-time programmable interconnection matrix (IM) is used to connect DEBs with the reconfigurable logic I/O ports. This matrix is programmed by the STxP70 processor.

As it was mentioned in section 4.3.3, DEBs may function in two modes: random accessed memory (RAM) or a FIFO memory. The main difference is that read operations are destructive on a FIFO while they are not on RAM. A FIFO can only be accessed once and sequentially, while RAM allows repeated and indexed access. On the other hand, a FIFO may be programmed to load a potentially infinite stream, while RAM can only handle fixed-length streams.

Each DEB is also linked with the IM through the set of address generators. For computational kernels of embedded applications, which are typically enclosed in iterative cycles of fixed length, it is possible in most cases to predefine the addressing pattern and relieve the controlling processor of the data transfer phase, that is performed to the address generator.

The described system is controlled by a small subset of the processor's instruction set, implemented in the compiler through built-in functions. Dedicated hardware implements their functionality. Triggering an operation on the reconfigurable logic means feeding the logic through the appropriate input blocks and preparing the output blocks to receive a write back. The latency of a given operation on the reconfigurable logic may be unknown or downright unpredictable, so a locking logic was designed to ensure the consistency of the data flow. An operation can not be triggered if it reads from an empty FIFO, or if the register is scheduled for a write from a preceding operation. In case a write

operation is required on a saturated FIFO, the whole system is frozen until the FIFO becomes available. The proposed solution aims further development of FIFO programming flexibility.

DEBs consist of 16 buffers, 32-bit wide by 1 Kword in depth, which are represented (see Fig. 50). The storage blocks are connected with PiCoGA by means of programmable interconnection network. PiCoGA supports up to 12 inputs and 4 outputs, each of which might be programmatically connected with any of 16 DEBs. This flexibility makes possible to change PiCoGA inputs and outputs at the runtime. Moreover, it allows feeding yet unused DEBs with the future data, simultaneously with reading the current data from the other registers.

DEBs are fed with data from the on-chip memory through the host system bus. The DMA unit of the host system is used to accelerate this process. Furthermore, the DMA may be programmed in such a way that it prevents repeated transfer of the same data through the system bus. A data block is read from the memory into the DEB only once (even if it is needed to be used a number of times in PiCoGA). Then, user programmable AG controls which data word should be read from FIFO and when it should be deleted. Thus, the same data word can be read several times from FIFO into PiCoGA.

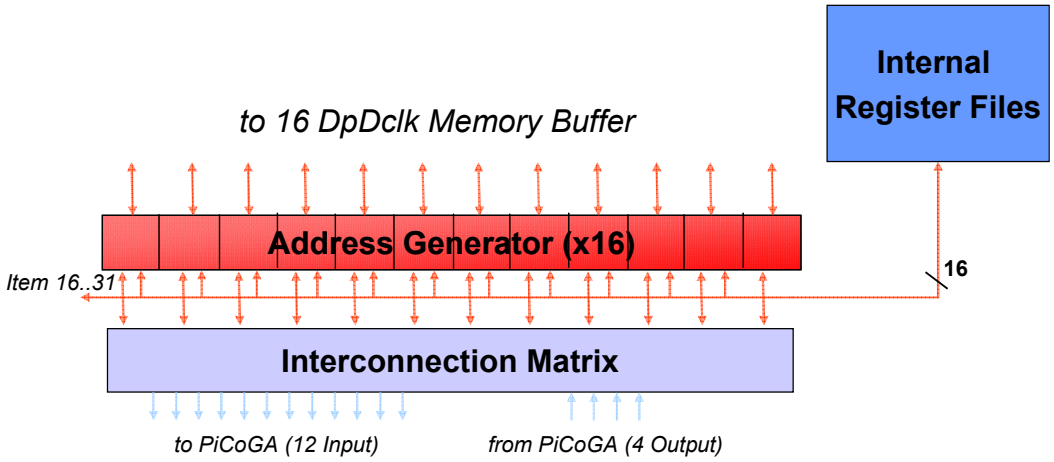


Fig. 50. Integration of the AG in DREAM architecture.

AGs provide standard *step* and *stride* capabilities to achieve non-continuous vector addressing. A specific *mask* functionality allows also power-of-two modulo addressing in order to realize variable size circular buffers with programmable start point.

In order to program the Address Generator, two different functions can be used inside a .c/.cpp program file. First of them provides standard memory access. Its syntax is represented in Table 12. Using these five parameters it is possible to generate a set of access patterns, classified in two series with two subgroups each. The simplest group of the access patterns is the so called *solid data chunks* access, with or without overlapping (see Fig. 51: I, II). Fig. 51-I represents an example to perform memory access starting from the address zero, with a block size of 3 elements, with the stride equal to 5: *set_DF(Item, 0, 3, 5, 1, 0)*. The previous example can be modified by setting the stride equal to 2, instead of 5, in order to perform an overlapped access: *set_DF(Item, 0, 3, 2, 1, 0)*. In this case, the last element of each chunk is overlapped with the first element of the successive chunk (see Fig. 51-II). Another group of memory accesses includes *fragmented data chunks* access (see Fig. 51: III, IV), i.e. when the step is greater than 1, a non-contiguous access is performed. While the stride is greater or equal to the product of count and stride, two separated chunks are generated (Fig. 51-III): *set_DF(Item, 0, 2, 6, 2, 0)*. By decreasing the stride it is possible to overlap two chunks (Fig. 51-IV): *set_DF(Item, 0, 3, 1, 3, 0)*.

Table 12. Standard memory access.

```
set_DF(Item, Addr, Count, Stride, Step, rw);

//where:

//  Item - is an index value (range 16..31) that points to the Identifier of the according address
generator;

//  Addr - is the base address inside the memory (range 0...4095);

//  Count - number of data words inside a chunk (range 0...4095);

//  Stride - distance between data chunks (range -127..+128);

//  Step – distance between two successive data words inside a chunk (range -127..+128);

//  rw - read/write access.
```

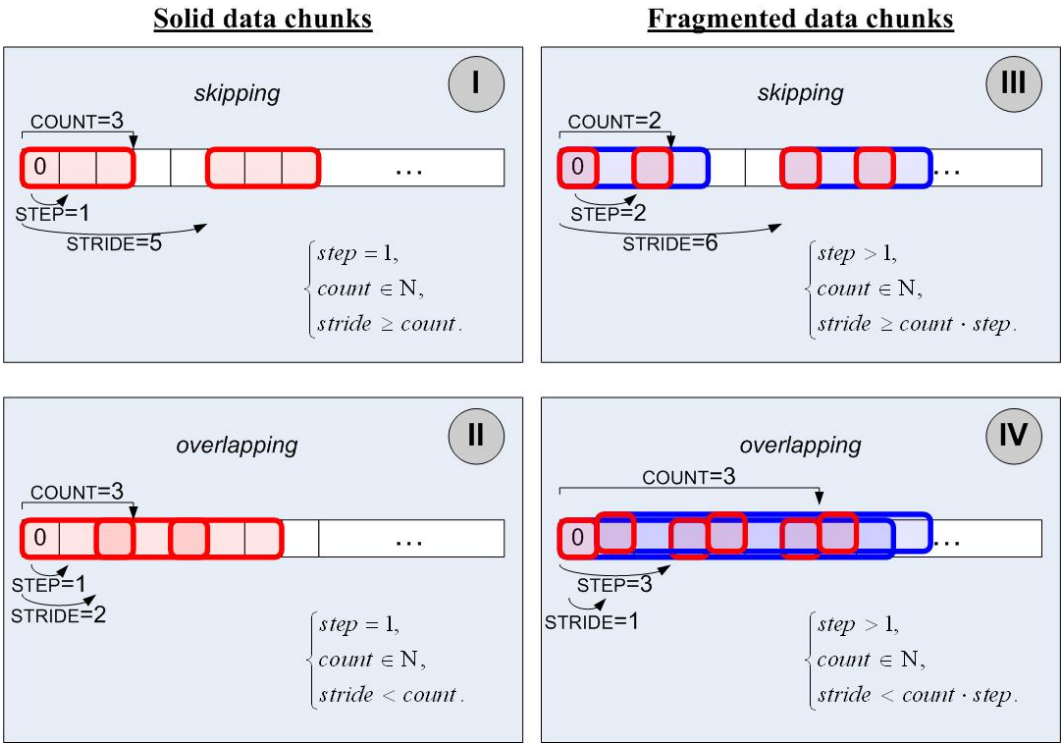


Fig. 51. A classification of memory access types enabled by AG.

With respect to the standard memory access, the memory access provided by the function from Table 13 is enhanced by the *mask* parameter. The mask can be used to implement *modulo addressing* (e.g. for the *circular buffer* implementation) where the *modulo* function is obtained from the mask (this is not a real modulo, but for the power-of-two the functionality is the same). From a functional point of view, the addressed memory location is determined from the following expressions:

```
Local_address = next_address;

Base_address  = next_base_address;

address = (Base_address & ~mask) + (local_address & mask);

next_address      = local_address + step;

next_base_address = (base_address + stride) if (Count == EndOfCount);
```

Table 13. Masked memory access.

```
set_DF(Item, Addr, Count, Mask, Stride, Step, rw);

//where:
```

```
//  Item - is an index value (range 16..31) that points to the Identifier of the according address
generator;

//  Addr - is the base address inside the memory (range 0...4095);

//  Count - number of data words inside a chunk (range 0...4095);

//  Mask - size of the circular buffer  (range 2^0...2^16);

//  Stride - distance between data chunks (range -127..+128);

//  Step – distance between two successive data words inside a chunk (range -127..+128);

//  rw - read/write access.
```

The following examples represent the behavior of the masked access:

```
Example 1:

  Base = 0x0; Step = 1; Stride = 0; count = 8; mask = 0x07;

  Generated address sequence:

  0x00 – 0x01 – 0x02 – 0x03 – 0x04 – 0x05 – 0x06 – 0x07 – 0x00 – …


Example 2:

  Base = 0x1; Step = 1; Stride = 4; count = 4; mask = 0x03;

  Generated address sequence:

  0x01 – 0x02 – 0x03 – 0x00 – 0x05 – 0x06 – 0x07 – 0x04 – 0x09 - …


Example 3:

  Base = 0x1; Step  = 1; Stride = 5; count = 4; mask = 0x03;

  Generated address sequence:

  Address = 0x1

  Address = 0x2

  Address = 0x3

  Address = 0x0           // masking effect!!! (and EndOfCount sent)

  Address = 0x6           // stride sum

  Address = 0x7
```

```
Address = 0x4          // masking effect!!!

Address = 0x5          // EndOfCount sent

...
```

When the mask is written in the form of 0b0..01..1 (i.e. with two series of consecutive 0s and 1s representing the standard form), the addressed location is the concatenation of bits in the base address (for the bits corresponding to 0 in the mask), and of bits in the local address (for the bits corresponding to 1 in the mask). Step and stride have the same functionality of the standard cases, while the obtained result (addressed location) is affected by a *quasi*-modulo operation:

- the local_address is calculated using the step, but the result is masked;
- the stride is used to update the base address, but in fact can also be used to determine/change the starting point of the circular buffer;
- the circular buffer is defined as:
    - Bottom_address = Base_address & (~mask)
    - Top_address = Bottom_address + mask

It should be noted, that when the mask is set to "all 1s" (0b1111..111), the functionality is the same as was for the standard memory access. Step and stride might be less then zero, which allows making a step in both directions from the base address. However, stream length is always greater then zero.

The block diagram Fig. 52 outlines the AG implementation.

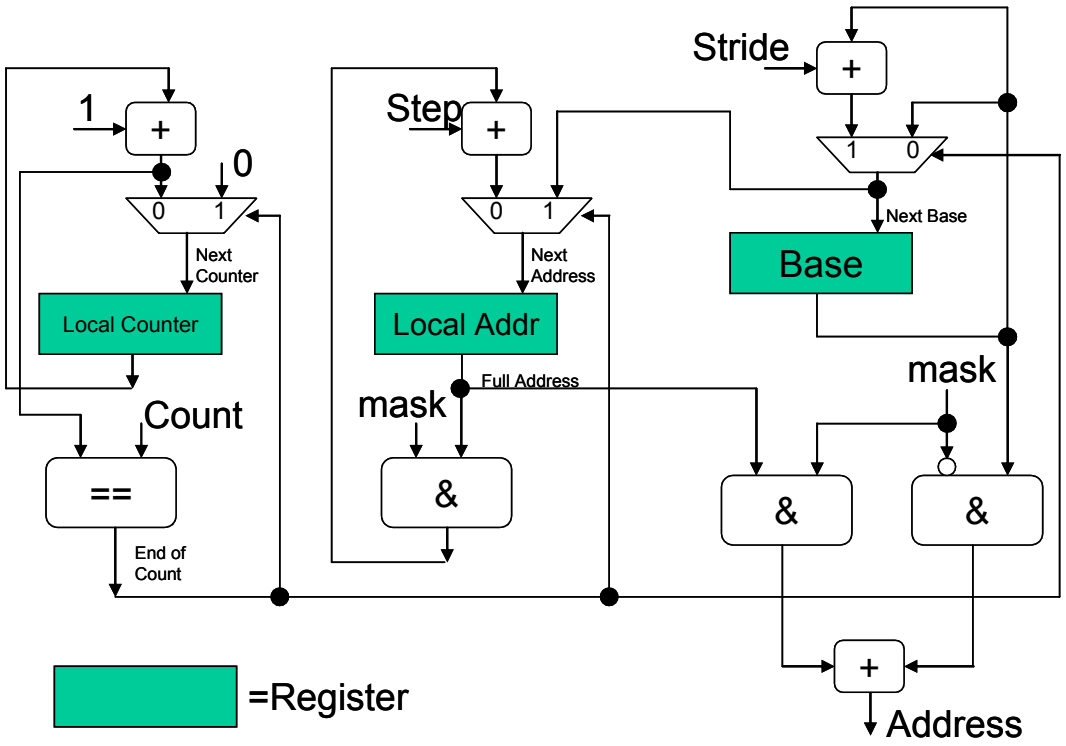


Fig. 52. AG block diagram.

4.5.4 SOFTWARE TOOL-CHAIN

The DREAM architecture is programmed at two levels: PiCoGA level and STxP70 level. PiCoGA-III is programmed using a simplified single-assignment ANSI-C syntax, which is translated into configuration bitstreams by the specific mapping tool [55]. Bitstreams are provided as C vectors that can be included in the global application code for the embedded STxP70. The application code itself, including references to configuration and computation of macro-instructions as well as AG patterns, is compiled with a retargeted version of the Open64 compiler [33] and proprietary tools [60]. Furthermore, cycle accurate simulation, debugging and performance evaluation tools are available [60] under the STxP70 development environment, based on the Eclipse platform. The complete design space exploration is thus performed by the user in a high-level software environment, where performance speed-ups typical of hardware oriented implementations can be obtained requiring no specific hardware expertise, which is a significant advantage of the introduced architecture.

An example of the C control source code for the host ARM processor, which manages data exchange with the DREAM IP, is presented in Appendix A.

4.5.5 RESULTS

The proposed architecture was implemented in CMOS 90nm technology. Technical results are outlined in Table 14. The total area is 16 mm^2, taking into account 100% density of the custom layout circuits and 70% density of the RTL logic after the final layout. Top frequency in worst-case commercial conditions (125C, 0.9V) is 200MHZ, while the device can deliver up to 250MHZ in typical conditions (25C, 1V). The PiCoGA-III was designed with a mixed custom/semi-custom design flow, while the control and memory addressing sections were designed in HDL and mapped on standard cells libraries.

Table 14. Area occupation and energy consumption.

Unit	*Area*	*Dynamic energy*	*Leakage energy*
STxP70 with 16KB+16KB memories	1.18 mm^2	30 µW/MHz	0.4 mW
PiCoGA-III, interf. logic	10.31 mm^2	340 µW/MHz	15.6 mW
DEBs (64KB), AGs, interconnect matrix, RF	2.09 mm^2	283 µW/MHz	6.7 mW
CEBs (36KB)	1.15 mm^2	negligible	4.1 mW
Others (PLL, co-proc. interf., glue logic)	0.17 mm^2	negligible	Negligible
Total	16 mm^2	652 µW/MHz	26.8 mW

Processor efficiency was measured on a set of computational kernels, oriented towards multimedia and communication applications. In particular, four highly-parallel kernels were selected from the open-source H.264 coding standard [49], as well as an OFDM Constellation Encoder or Mapper [68] (implemented at three levels of unfolding) and a well-known symmetric-key cipher AES with 128 key size [57], whose implementation is thoroughly described in [56]. Performances were evaluated at 200MHZ, and they are parameterized with respect to the *interleaving factor*, intended as the number of data blocks concurrently elaborated. Table 15 describes the performance of the selected kernels. In fact, most multimedia and communication kernels feature thread-level parallelism (i.e. image processing transforms show no correlation across macro-blocks), and interleaving of the elaboration of more than one block allows deeper

level of pipelining in computation. The interleaving factor applicable depends also on the available DEB memory budget. All the benchmarks reach a saturation point, where further computation unfolding is made impossible by lack of storage capacity on local memory.

Table 15. Performance of several application kernels.

Kernel	*GOPS*	*GOPS/mm²*	*GOPS/mW*
Add4x4idct	32.54	2.03	0.15
Sub4x4dct	48.23	3.01	0.22
Sad4x4	59.28	3.70	0.33
Satd4x4	44.06	2.75	0.22
Ofdm-Mapper1	4.10	0.25	0.04
Ofdm-Mapper4	16.10	1.00	0.10
Ofdm-Mapper8	31.50	1.96	0.15
AES-128	4.90	0.30	0.03

Fig. 53 describes performance statistics, in terms of processed bits per second. Fig. 54 depicts the speed-up in terms of computation cycles with respect to a standard embedded RISC core.

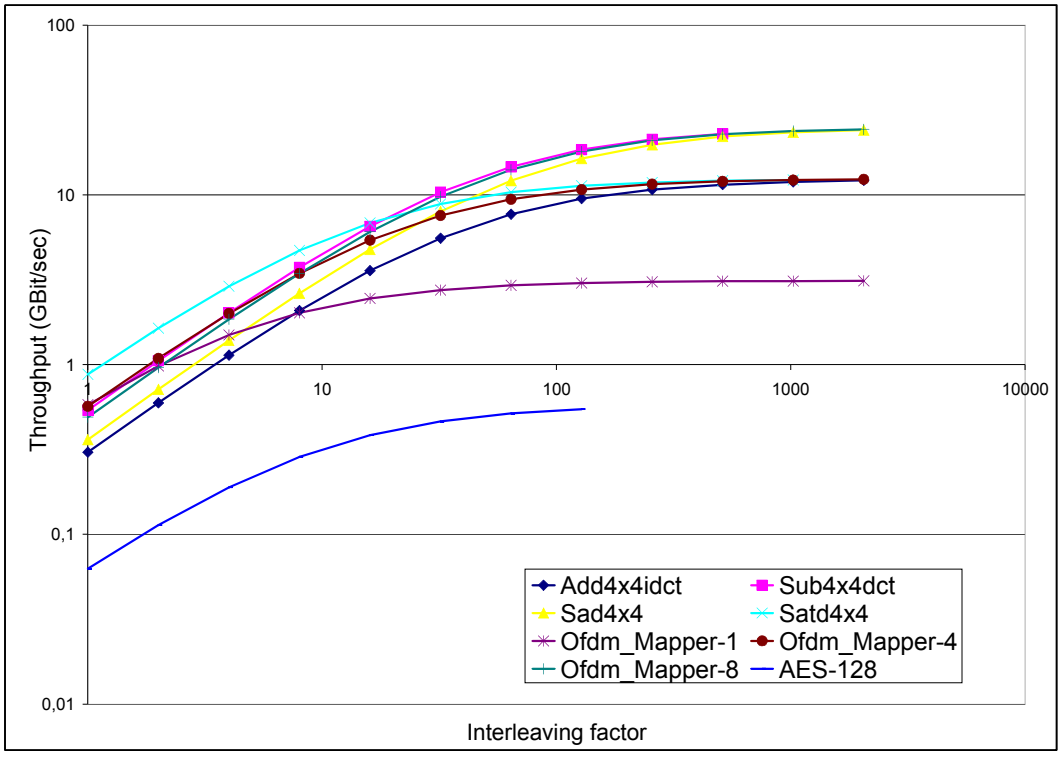


Fig. 53. Throughput vs. interleaving factor.

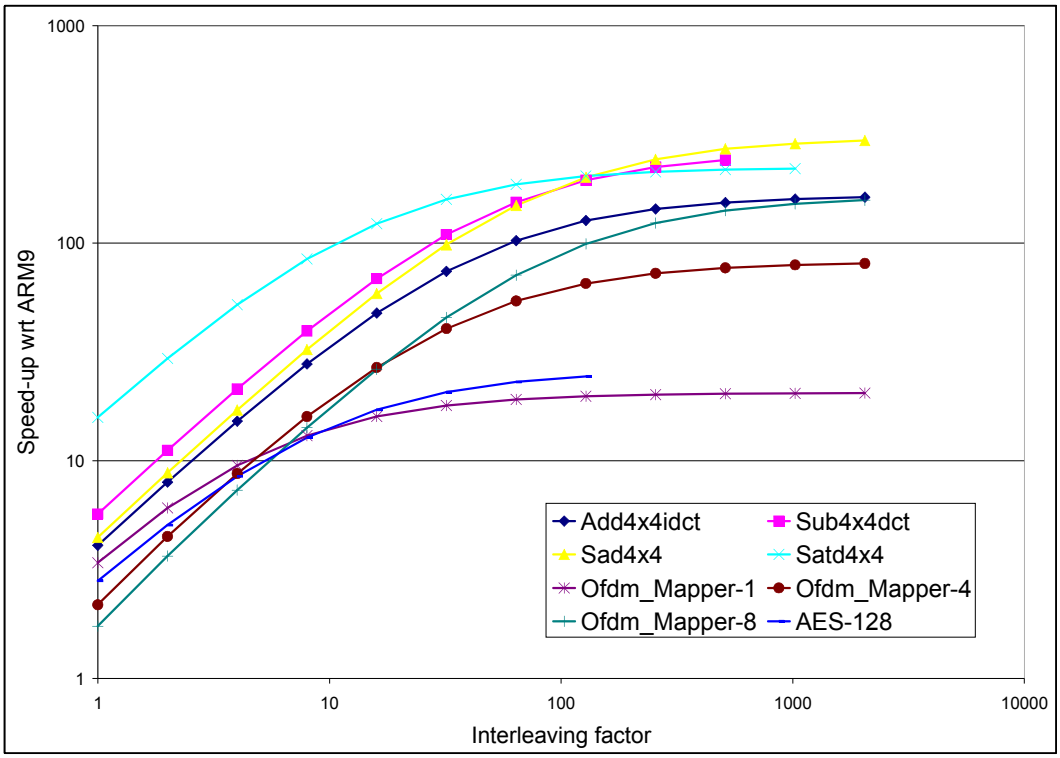


Fig. 54. Speed-up wrt. ARM9 processor.

4.6 COMPARISON WITH THE RELATED WORK

The MORPHEUS memory organization inherited the advantages of the addressing modes described in section 2.2.2, taking into account the specifics of HREs and the kind of computation required:

- Dual clock asynchronous FIFOs based on Gray Code, fed by system-level DMAs or the NoC DNA paradigm will be used to implement data streaming to/from HREs across different clock domains.
- The HREs, where possible (i.e. in the case of the M2000 eFPGA) will be used to generate run-time configurable addressing patterns on local memories (DEBs) to feed their own computation.
- A set of programmable hardwired address generators are made available to implement highly parallel, parametric high speed addressing on DEBs. Their addressing pattern is based on the Base + Step + Stride pattern presented, among others, by [27].

The local to each HRE memories (DEBs) on dual-port, dual-clock memories, are available as off-the-shelf items in the targeted technology node. In addition to the dual-port advantages described in section 2.2.3, the dual-clock structure offers the significant advantage of isolation of each HRE clock domain, that in some cases could be also run-time programmable and application-specific. Thus, it enables each component of the MORPHEUS architecture running at its ideal speed. From the system side, communication to the DEBs is performed both through the NoC and the AMBA bus infrastructures, whereas from the HRE side communication is performed making use of the mechanism described in section 4.3.3.

The streaming concept, presented in section 2.2.2, matches very well the features of the MORPHEUS computing platform, thus making finite streams of variable width (depending on the given application) the ideal units for data transfers between ARM processor, HREs, off-chip memories and main on-chip memories. Stream-oriented transfers can be driven by the processor core using a collection of load/store operations or by embedded DMAs on the main data bus. In this context, a relevant additional value could be given by the utilization of the NoC. Apart from obvious advantages in enhancing data throughput via multiple

channels and shortening large data transfer latency due to its pipelined nature, the NoC can add a stream-oriented connotation to the communication mechanism, thanks to parametric FIFO memories that are resident in NoC routing nodes. Such memories, which should be considered as a specific layer of the peculiar to the MORPHEUS memory hierarchy, provide a powerful internal storage resource that alleviates the stress on off-chip and on-chip memories. Moreover, they are used as a mean to decrease the stalls induces by congestions in communication and peaks/stalls in HRE computation. Thus, the NoC exploits the data streaming approach not only to decouple computation from data transfers, but also to decouple different steps in the computation providing a very flexible and bottleneck free hierarchical transfer mechanism. Finally, a further added value of the NoC approach deployed in MORPHEUS is that it appears very compiler friendly because it presents the same Advanced Programming Interface (API) as the one for handling standard DMAs. The only drawback – the non-predictable latency sometimes introduced by the NoC communication – can be tackled with worst case quality of service analysis, and reserving specific NoC channels for critical communications. Interrupts and specific synchronization instructions (according to the Molen paradigm) is used by the ARM processor to synchronize computation and communication and maintain data flow consistency.

Comparing with the IMAGINE architecture (refer to section 2.2.4), MORPHEUS shares the concept of utilizing streams as atomic units for transfers. Moreover, it provides similar differentiation between stream-level and kernel-level programming:

- At the ARM processor level, the application is described by abstractions: operands are not scalar data but data chunks, or streams. Operators, in turn, are HRE operations, or libraries of HRE operations stored in the configuration memory and utilized as microcode, according to the Molen paradigm. One should note that since MORPHEUS is stream-oriented architecture, the standard Molen programming paradigm (*set-execute*) is adapted to the stream behavior. The Molen *set* operation correlates with MORPHEUS *set configuration* procedure, while the Molen *execute* operation is implicitly integrated in the data transfers to/from HREs, as

starting with HRE initialization it automatically begins to process input data and send the results to the output. Stream transfer details are implemented via DMA/NoC and hidden to the programmer, and NoC/Bus/DMA control instructions can be considered as microcode in the Molen programming context. As a result, all computation (HREs) and communication (Bus/NoC) means are seen in a totally homogeneous way and handled by Molen-like microcode and software libraries.

- Dual clock DEBs are used to cross frequency domains and to transfer operands (streams) from the communication engines to the HREs.
- At HRE level, the operator is described by the reconfigurable fabric specific microcode (bitstream). The bitstream is produced according to the HRE design flow and entry language. The HRE may then process the stream providing scalar access to each one of its components. A significant added value of the MORPHEUS memory hierarchy is that this access offers a very relevant degree of flexibility. HRE may read sequentially the stream as a FIFO, utilize programmable address generators for vector addressing, or describe the access to stream elements as a part of the reconfigurable operator itself. This choice enables the stream communication, similar to the ones described in section 2.2.2, but offers a much higher level of flexibility from the HRE side, allowing also the usage of the DEBs as temporary computation repository.

Similar to the BAZIL system (refer to 2.2.4), MORPHEUS utilizes two types of communication infrastructure. However, in contrast to BAZIL, MORPHEUS includes NoC instead of the direct connection. This solution is more flexible and allows parametric interconnection between an indefinite number of reconfigurable cores, thus providing a modular architectural template that can be refined and adapted beyond the specific demonstrator implementation. Furthermore, the NoC itself represents intermediate data storage for HREs. The local FIFOs in the routing nodes allow a streaming approach representing a way to hide delays due to local congestions and to hide latency due to peaks in HRE calculation. Also, MORPHEUS provides DMA controllers for the bus hierarchy

and DNA controller for the NoC, both based on the same API, which allows distributing data inside the system more efficiently.

4.7 SUMMARY

The memory organization of the MORPHEUS platform features a hierarchical structure, enabling distributed data-flow and providing transparent homogeneous integration for the end user. More precisely, it supports the following features:

- The external memory serves as main system storage for configuration data and application specific data streams.
- The main on-chip memory connected to both NoC infrastructure and multi-layer AMBA bus provides high density and low power, low latency access to the smaller frames of the data stream.
- Local FIFOs on the NoC enable the streaming data transfer mechanism, which minimizes congestions on communication channels and stalls of local computation.
- Local dual-port, dual-clock DEBs represent the most innovative and critical part of the overall memory organization, where the most significant effort was aimed on tighter integration with the HREs. According to the MORPHEUS specifications, DEBs provide a uniform interface to the programmer, making the interconnect infrastructure more homogeneous. Their flexible access interfaces to the HREs allow integration of embedded devices with different nature and various granularities.

The presented memory organization unifies the most promising features from conventional architectures, putting the accent on finding optimal trade-offs between: distributed data-flow and interconnect integration; stream memory access and programming complexity; hierarchical data storage and die area occupation. The following features distinguish MORPHEUS memory subsystem from the state-of-the-art solutions:

- Four parallel internal and external data-flows: computational data, configuration data, control data, and I/O data. A distributed organization of these flows prevents them of interfering and creating communication bottle-necks.

- Hierarchical data storage organization separates computation from communication, provides fast access to the currently processed data, and relaxes off-chip traffic.
- Thoroughly engineered mechanisms operate with data frames of various granularities on all levels of storage hierarchy and provide reconfigurable resources with flexible memory access.
- Network-oriented interconnect infrastructure provides a powerful local storage mechanism in addition to the state-of-the-art communication capabilities.

That the most promising feature of the MORPHEUS memory architecture resides in its dual level organization, i.e.:

1. At the system level, the programmer handles data transfers using the stream approach. This provides a homogeneous and user friendly utilization of the interconnect resources, similar to the approach offered by the Molen programming paradigm.
2. At HRE level, the designer of the instruction set extension, that will become a library for the ARM processor, must be very well acquainted with the HRE architectural details. In this case, the designer will be able to utilize in a very efficient manner different addressing methods on the local streams at high parallelism, in order to exploit the computational capabilities of the specific HRE at its maximum.

This dualism between high level homogeneous programmability and low level hardware-specific exploitation of available performance is the key for the success of the MORPHEUS approach.

CHAPTER 5

TWO-DIMENSIONAL PARALLEL MEMORY ACCESS WITH MULTIPLE PATTERN

This chapter presents a novel multi-pattern parallel addressing scheme in two-dimensional (2D) addressing space and the corresponding 2D interleaved memory organization with run-time reconfiguration features. The proposed architecture targets mainly multimedia and scientific applications with block cyclic data organization running on computing systems with high memory bandwidth demands, such as vector processors, multimedia accelerators, etc. The prior research on 2D addressing schemes is substantially extended introducing additional parameters, which define a large variety of 2D data patterns. The proposed scheme guarantees minimum memory latency and efficient bandwidth utilization for arbitrary configuration parameters of the data pattern. The presented mathematical descriptions prove the correctness of the proposed addressing schemes. The design and wire complexities, as well as the critical paths are evaluated using technology independent methodology and confirm the scalability of the memory organization. These theoretical results are confirmed by the synthesis for both ASIC and FPGA technologies. Comparison with the related works shows the advantages of reported addressing scheme. The RTL implementation of the memory organization represents the complete platform-independent IP and can be integrated in any architecture.

5.1 INTRODUCTION

5.1.1 RESEARCH CONTEXT AND GOAL DESCRIPTION

With modern increase of technology development, the performance of memory subsystems lags more and more behind the processing units. This trend becomes increasingly evident for architectures with massively parallel data processing, such as multimedia accelerators, vector processors, SIMD-based machines, etc. There are several techniques developed to reduce the processor versus memory performance gap, including various caching mechanisms, memories advanced with extra wide data word or multiple ports. But most of all, the parallelism phenomenon is utilized in *parallel memory* organizations, where the storage subsystem consists of a set of memory modules working in parallel. The main advantages of this organization are: relatively small overheads, low latency, efficient interconnection usage and possibility of accessing specific *data patterns*. The data patterns depend very much on the target application and might have various shapes, sizes and *strides* (distances between the successive elements).

The design challenge is to ensure *conflict-free* parallel data access to all (or maximum possible number of) memory modules for a set of different data patterns. This is obtained by means of a *module assignment function*. According to the data pattern format (in other words template), various module assignments can be implemented, such as linear functions [19], XOR-schemes [36], rectangular addressable memories [44], periodic schemes [65] and others. *Row address function* specifies physical address inside a memory module. Together, module assignment and row address functions form the class of *skewing schemes*.

However, there is no single skewing scheme which would support conflict-free access for all possible data patterns [11]. Two solutions that deal with such limitation are: *Configurable Parallel Memory Architecture* (CPMA) [43], [59] and *Dynamic Storage Scheme* (DSS) [11], [32], [30], [47]. CPMA provides access to a number of data templates using a single relatively complex hardware when the number of memory modules is arbitrary. A more dedicated DSS unifies multiple storage schemes within one system. The appropriate scheme

is chosen dynamically according to the specific data pattern in use. DSS restricts the amount of memory modules to the power of two and considers only interleaved memory system [32], [30].

The goal of this research is to develop a memory hierarchy with dynamically adjustable regular 2D access patterns, which would improve the data throughput between the main memory and processing units. Our approach is to split the problem into six trivial sub-problems, which would require hardware implementation with rather low complexity and short critical path. We consider an exhaustive set of pattern definition parameters and propose a performance efficient, interleaved memory organization. More specifically, the main contributions of the current proposal are as follows:

- Extended set of 2D pattern access parameters: *base address; vertical and horizontal strides, group lengths, and block sizes*.
- Support for the complete set of the 2D data patterns described by the above parameters.
- Run-time programmability of the memory access pattern by means of Special-Purpose Registers (SPRs).
- Independency of the data pattern size from the number of the interleaved memory modules;
- Minimal memory access latency for arbitrary strides and group lengths.
- Modular implementation, which can be easily simplified to a restricted subset of 2D data patterns (if the target application does not require full flexibility), thus reducing the design complexity and critical path.
- High design scalability confirmed by hardware synthesis results.

The proposed memory organization targets highly data-parallel applications with 2D block cyclic data distribution [26]. The matter concerns mainly scientific operations on matrices for e.g. synthetic aperture radar (SAR) software [58] or applied aerodynamics (Actiflow), as well as multimedia applications such as audio/video compression (ADPCM, G721, GSM, MPEG4, JPEG).

5.1.2 RELATED WORK

A DSS for a strided vector access was presented in [32]. The stride value is detected by the compiler and sent directly to the pipelined address transformation hardware. In such a way, the scheme supports conflict-free accesses for non-restricted vectors with constant arbitrary stride.

In [30] the authors extended their scheme with block, multistride and FFT accesses. To decrease the latency of multistride vector access when conflict-free access is not achieved, it was proposed to use dedicated buffers to smooth out transient non-uniformities in module reference distribution. Block access supported only restricted set of blocks sizes equal to power of two. Finally, in order to improve a radix-2 FFT algorithm, authors proposed non-interleaved storage scheme and a constant geometry algorithm for which they identified three data patterns. For all three types of address transformations, the same hardware was used.

CPMA from [43] supports generate, crumbled rectangle, chessboard, vector, and free data patterns. Virtual address is used to read appropriate row address and access function from the page table which are further transformed into row and module addresses. The authors presented complexity, timing and area evaluations for their architecture.

A buffer memory system for a fast and high-resolution graphical display system was proposed in [59]. It provides parallel access to a block, horizontal, vertical, forward-diagonal, and backward-diagonal data pattern in a two-dimensional image array. All pattern sizes are limited to power of two. The address differences of those patterns are specifically prearranged and saved in two SRAMs so that later they can be added to the base address in order to obtain memory module addresses.

Other researches explore memory scheduling of DRAM chips by addressing locality characteristics within the 3D (bank, row, column) memory structure [62]. The solution consists of reordering memory operations in such a way that allows saving clock cycles on precharging banks and accessing successive rows and columns.

5.2 THEORETICAL BASIS

In parallel memories data can be referenced using predetermined patterns called memory access patterns or data patterns. The data distribution among the parallel memory modules is called a module assignment function m which is also known as a skewed scheme. The module assignment function determines data patterns that can be accessed conflict-free. A data element with a linear address a is assigned to a memory module according to $m(a)$. A row address function A determines the physical address of a data element inside a memory module.

Our task is to develop a memory hierarchy with dynamically programmable data patterns to minimize the main memory access latency from the vector processing units using interleaved memory modules organized in a two-dimensional matrix with parallel access. Fig. 55 depicts an example of a data pattern block including six groups of size $VGL \times HGL = 2 \times 4$ and strides $(VS,HS)=(4,5)$. The parameters used to describe the data pattern are explained in Table 16.

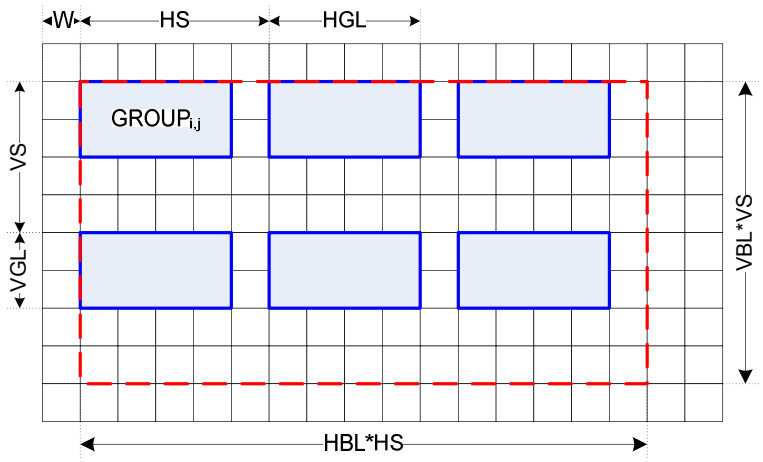


Fig. 55. Proposed memory access pattern.

Table 16. Memory access pattern parameters.

Parameter	Description
$W \in \mathbb{N}$[1]	- data word length in bytes;
$w_A \in \mathbb{N}$	- row address width in bits;
$b'(vb, hs) \in [0, \mathbb{N}]$	- base address of the accessed block;
$VS, HS \in \mathbb{N}$	- vertical and horizontal strides;
$VGL, HGL \in \mathbb{N}$	- vertical and horizontal group lengths – group size;
$VBL, HBL \in \mathbb{N}$	- vertical and horizontal block lengths – block size;
$M \times N \in \mathbb{N}$	- size of a data block stored in the memory;
$VD \times HD \in \mathbb{N}$	- size of the matrix of memory modules;
$0 \le i = \left\lfloor \frac{id_V}{VGL} \right\rfloor < VBL$, $0 \le id_V < VBL \cdot VGL$, $id_V \in \mathbb{N}$.	- vertical i and horizontal j group indices;
$0 \le j = \left\lfloor \frac{id_H}{HGL} \right\rfloor < HBL$, $0 \le id_H < HBL \cdot HGL$, $id_H \in \mathbb{N}$.	
$0 \le k = id_V \bmod VGL < VGL$ $0 \le l = id_H \bmod HGL < HGL$	- vertical k and horizontal l element indices.

The first step is to translate the linear address of the main memory into a two-dimensional one. The transformation equation (3) is used in order to translate linear base address b' into a two-dimensional one (vb, hb) with the vertical and horizontal constituents.

$$b' = vb \cdot N + hb,$$
$$vb = \lfloor b' / N \rfloor, \tag{3}$$
$$hb = b' \bmod N$$

Any data element with linear address a' belonging to an accessed data block has the following two-dimensional address (va, ha):

[1] Though the word length can have any natural value, on practice it usually has a value of power of two.

$$a' = va \cdot N + ha,$$

$$va = VB + i \cdot VS + k = a_{i,k}, \tag{4}$$

$$ha = HB + j \cdot HS + l = a_{j,l}.$$

where indices $i \in [0, VBL-1], k \in [0, VGL-1]$, and $j \in [0, HBL-1], l \in [0, HGL-1]$

As follows from (4), the two-dimensional address is completely separable, i.e. its vertical and horizontal constituents are independent from each other. Therefore, in the following discussion we consider the address constituent along only one dimension.

The *stride* parameter as any other natural number might be represented in the following expansion:

$$S = \sigma \cdot 2^s \in \mathbb{N}, \tag{5}$$

where $\sigma = 2x + 1$, $\forall x \in \mathbb{N}$ and $s \in [0, \mathbb{N}]$. Consequently, stride S is *odd* when $s = 0$, and it is *even* when $s \in \mathbb{N}$.

Before proceeding to the description of our solution we would need to consider the following theorems.

Theorem 1: No single skewing scheme can be found that allows conflict-free access for all the constant strides and group lengths when the data pattern can be unrestrictedly placed. The theorem is valid for arbitrary number of memory modules, when at least two data elements are accessed concurrently[1].

Proof: Let $m(a)$ be the module assignment function and $a_{id} = b + j \cdot S + l$ the first accessed data element. Then, the next accessed data element is

$$\begin{cases} a_{id+1} = b + j \cdot S + l + 1 = a_{id} + 1, & if\ l < GL - 1; \\ a_{id+1} = b + (j+1) \cdot S = a_{id} + S - l, & if\ l = GL - 1 \end{cases},$$

[1] This theorem follows the theorem 1 from [11] but in our case it has wider application since it considers the group length together with the stride.

where GL is a group length, and S is a stride. The two elements can not be accessed conflict-free if there are stride S and group length GL such that $m(a_{id}) = m(a_{id+1})$. □

Theorem 2: All the *odd strides* can be accessed conflict-free with the low-order interleaved scheme if the number of memory modules equal to power of two: $D = 2^d$, $d \in \mathbb{N}$.

Proof: The proof is presented by M. Valero et al. in [47]. □

Theorem 3: Let stride S', group length GL, and number of memory modules D along one dimension equal to power of two, i.e. $S' = 2^s$, $GL = 2^{gl}$, and $D = 2^d$, where $s, gl, d \in \mathbb{N}$. Also let $s \geq d$ which means that the strides have less than one access per row. Then the module assignment function defined by

$$m(a) = \left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D, \tag{6}$$

allows conflict-free parallel accesses to D memory modules.

Proof: First we find a period P of function (6). If function $m(a)$ maps address a to its module address for a stride S' and group length GL, then $m(a) = m(a + P \cdot S')$, $\forall\, a$. According to (6), this corresponds to

$$\left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D = \left((a + P \cdot S') + GL \cdot \left\lfloor \frac{\lfloor (a + P \cdot S')/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D;$$

$$\left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D = \left(a + P \cdot S' + GL \cdot \left\lfloor \frac{\lfloor a/D + P \cdot S'/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D;$$

$$\left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D = \left(a + P \cdot S' + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor + P \cdot 2^{s-d}}{2^{s-d}} \right\rfloor \right) \bmod D;$$

$$\left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D = \left(a + P \cdot S' + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor + P \cdot GL \right) \bmod D;$$

$$\left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D = \left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor + P \cdot (S' + GL) \right) \bmod D.$$

According to the properties of modulo operation, this equality holds when $(P \cdot (S' + GL)) \bmod D = 0$. The last formula corresponds to the following set of periods:

$$\begin{cases} P\cdot(S'+GL)=x\cdot D; \\ P\cdot S'=y\cdot D; \\ P\cdot GL=z\cdot D, \end{cases} \Rightarrow \begin{cases} P=x\cdot\dfrac{D}{S+GL}; \\ P=y\cdot\dfrac{D}{S'}; \\ P=z\cdot\dfrac{D}{GL}, \end{cases}$$

where $x,y,z\in\mathbb{N}$ and $x,y,z:\ P\in\mathbb{N}$.

Now we will find the minimum period $P_{\min}>1$. This means to solve the next set of minimization problems:

$$\begin{cases} P_{\min}=\min\left(x\cdot\dfrac{D}{S'+GL}\right); \\ P_{\min}=\min\left(y\cdot\dfrac{D}{S'}\right); \\ P_{\min}=\min\left(z\cdot\dfrac{D}{GL}\right), \end{cases} \quad \text{where } P_{\min}\in\mathbb{N}.$$

$$\begin{cases} x=\dfrac{S'+GL}{GCD(D,S'+GL)}=\dfrac{S'+GL}{GCD(2^d,2^s+GL)}\overset{\substack{GCD\\property\\s\geq d}}{=}\dfrac{S'+GL}{GCD(D,GL)}; \\ y=\dfrac{S'}{GCD(D,S')}=\dfrac{2^s}{GCD(2^d,2^s)}\overset{s\geq d}{=}\dfrac{2^s}{2^d}=\dfrac{S'}{D}; \\ z=\dfrac{GL}{GCD(D,GL)}. \end{cases}$$

$$\begin{cases} P_{\min}=\dfrac{S'+GL}{GCD(D,GL)}\cdot\dfrac{D}{S'+GL}=\dfrac{D}{GCD(D,GL)}; \\ P_{\min}=\dfrac{S'}{D}\cdot\dfrac{D}{S'}=1; \\ P_{\min}=\dfrac{GL}{GCD(D,GL)}\cdot\dfrac{D}{GL}=\dfrac{D}{GCD(D,GL)}. \end{cases}$$

Thus, the minimum period equal to $P_{\min}=\dfrac{D}{GCD(D,GL)}$.

Now we will find number of accesses performed to the distinct memory modules. Harper and Jump showed in [31] that for a basic skewing storage scheme the number of distinct modules referenced during a vector access is

$A' = \min(P, D)$. For our case with group length presented, the number of distinct modules is $A = \min(P \cdot GL, D)$ and $GL = 2^{gl}$, $gl \in \mathbb{N}$.

$$A = \min(P \cdot GL, D) = \min\left(\frac{D \cdot GL}{GCD(D, GL)}, D \right) = \min\big(LCM(D, GL), D\big) = D .$$

Thus, any vector of length D, having the form of Fig. 55 and (4), inside the sequence of module addresses generated by $m(a)$ has exactly D distinct addresses. This is the definition of conflict-free accesses. □

Theorem 4: Let stride S', group length GL, and number of memory modules D along one dimension equal to power of two, i.e. $S' = 2^s$, $GL = 2^{gl}$, and $D = 2^d$, where $s, gl, d \in \mathbb{N}$. Also let $s < d$ which means that the strides have at least one access per row. Then the module assignment function defined by

$$m(a) = \left(a + \left(GL \cdot \left\lfloor \frac{a}{D} \right\rfloor \right) \bmod S' \right) \bmod D , \tag{7}$$

allows conflict-free parallel accesses to D memory modules.

Proof: The proof mainly repeats the one of *Theorem 3*. First we find a period P basing on the fact that $m(a) = m(a + P \cdot S')$, $\forall\, a$.

$$\left(a + \left(GL \cdot \left\lfloor \frac{a}{D} \right\rfloor \right) \bmod S' \right) \bmod D = \left(a + P \cdot S + \left(GL \cdot \left\lfloor \frac{a + P \cdot S'}{D} \right\rfloor \right) \bmod S' \right) \bmod D .$$

Using properties of modulo operation we derive the following combined equations:

$$\begin{cases} (P \cdot S') \bmod D = 0; \\ \left(a + \left(GL \cdot \left\lfloor \frac{a}{D} \right\rfloor \right) \bmod S' \right) \bmod D = \left(a + \left(GL \cdot \left\lfloor \frac{a}{D} + \frac{P \cdot S'}{D} \right\rfloor \right) \bmod S' \right) \bmod D. \end{cases}$$

The first equation gives us the set of periods equal to $P = x \cdot \frac{D}{S'}$, where $x \in \mathbb{N}$: $P \in \mathbb{N}$. Now we substitute period P in the second equation with its value derived from the first equation:

$$\left(a + \left(GL \cdot \left\lfloor \frac{a}{D} \right\rfloor \right) \bmod S' \right) \bmod D = \left(a + \left(GL \cdot \left\lfloor \frac{a}{D} + x \cdot \frac{D}{S'} \cdot \frac{S'}{D} \right\rfloor \right) \bmod S' \right) \bmod D ;$$

$$(a+\left(GL\cdot\left\lfloor\frac{a}{D}\right\rfloor\right)\bmod S')\bmod D=(a+\left(GL\cdot\left\lfloor\frac{a}{D}\right\rfloor+GL\cdot x\right)\bmod S')\bmod D.$$

Using the same property of modulo operator we obtain the following equation: $(x\cdot GL)\bmod S'=0 \Rightarrow x=y\cdot\frac{S'}{GL}$, where $y\in\mathbb{N}$: $x\in\mathbb{N}$.

Substitute x in the equation which gives the set of periods P:

$$P=x\cdot\frac{D}{S'}=y\cdot\frac{S'}{GL}\cdot\frac{D}{S'}=y\cdot\frac{D}{GL},\ y\in\mathbb{N}:\ P\in\mathbb{N}.$$

This set of periods exactly repeats the one from the proof of *Theorem 3*. Therefore, the minimum period equal to $P_{\min}=\frac{D}{GCD(D,GL)}$ and, as follows from the same proof, the number of distinct accessed addresses equals to D. Thus, module assignment function (1) is conflict-free. □

Theorem 5: If a vector is to be accessed with even stride $S=\sigma\cdot 2^s$, where $\sigma=2x+1,\ \forall x\in\mathbb{N},\ s\in\mathbb{N}$ and $s\neq 0$, and its elements are arranged in memory according to the storage scheme appropriate for a stride $S'=2^s$ access the accesses are conflict-free [32].

Proof: The proof repeats the one from [32] with only difference that in our case we should examine the sequence of groups of module addresses instead of the sequence of single addresses. □

Now we are ready to present the proposed solution.

5.3 PROPOSED MEMORY ACCESS SCHEME

Since there is no any single scheme for all the strides and group lengths according to *Theorem 1*, we propose to partition the problem in a number of cases, thus reducing the problem to a set of trivial sub-problems, and examine each of them independently.

We propose to partition the problem according to the *stride oddness* criterion on two subtasks. The following partition is done according to the theorems in section 5.2. According to *Theorem 2* odd strides can be accessed conflict-free using a basic skewing scheme [11], [44]. Further splitting on cases I and II is made for the purpose of memory latency minimization. Cases V and VI refer to *Theorem 3* and *Theorem 4* respectively. The remaining situations with even stride refer to cases III and IV and the skewing scheme from [32] is used. The problem partitioning is depicted on Fig. 56.

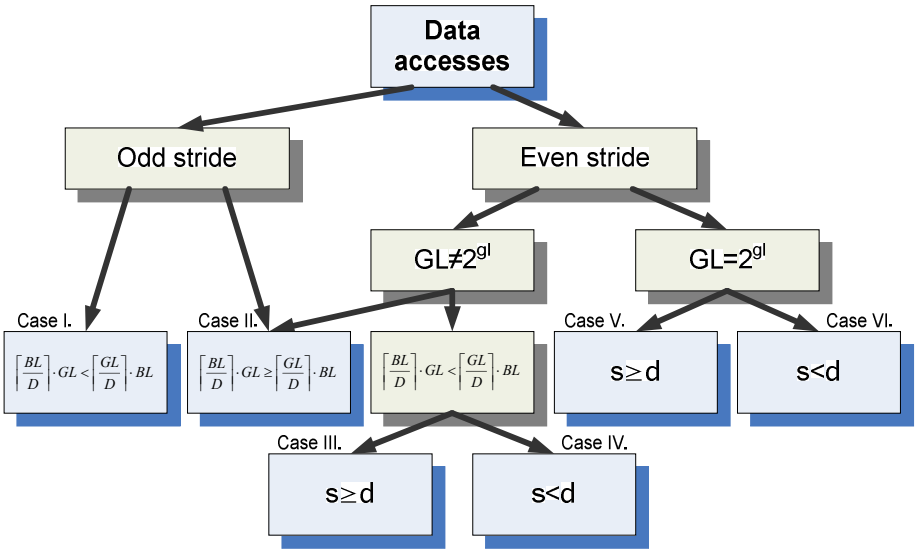


Fig. 56. Problem partitioning.

Now we need to build the module assignment and row address functions for all the cases.

5.3.1 MODULE ASSIGNMENT FUNCTION

We partition the design problem, imposed by the multiplicity of access patterns, into trivial subtasks. A module assignment function is devised for each of six different cases with respect to particular initial conditions.

5.3.1.1 CASE I

Initial conditions:

$$S = \{\sigma \cdot 2^s \mid \sigma = 2x + 1; s = 0\},$$
$$\left\lceil \frac{BL}{D} \right\rceil \cdot GL < \left\lceil \frac{GL}{D} \right\rceil \cdot BL. \quad (8)$$

An access to the data pattern is performed on a basis of sets of elements; i.e. ab init, the set of the first elements of all groups is accessed followed by the set of the second elements of all groups and so on. When $BL > D$ then more than one access is required to reed/write a whole group of data. If $(BL \bmod D) \neq 0$ then the remaining memory modules stay unused. The relation $\left\lceil \frac{BL}{D} \right\rceil \cdot GL < \left\lceil \frac{GL}{D} \right\rceil \cdot BL$ again guarantees that the number of accesses required to access the whole pattern is minimal. This case represents a conventional interleaved scheme with stride access which can be implemented conflict-free according to *Theorem 2*.

The module assignment function has the same representation as for a common interleaved scheme:

$$m(a) = a \bmod D. \quad (9)$$

The indices iterate according to the following sequence:

(i,k) = ((0,0);(1,0);...;(VBL-1,0);
(0,1);(1,1);...;(VBL-1,1);
...;
(0,VGL-1);(1,VGL-1);...;(VBL-1,VGL-1)). (10)

The number of the required accesses to read/write the whole data pattern in this case is equal to:

$$t = \left\lceil \frac{BL}{D} \right\rceil \cdot GL. \quad (11)$$

5.3.1.2 CASE II

Initial conditions:

$$S = \{\sigma \cdot 2^s \mid \sigma = 2x+1; s = 0\};$$

$$S = \{\sigma \cdot 2^s \mid \sigma = 2x+1; s \neq 0\} \ \& \ GL \neq 2^{gl}, \ gl \in \mathbb{N}; \tag{12}$$

$$\left\lceil \frac{BL}{D} \right\rceil \cdot GL \geq \left\lceil \frac{GL}{D} \right\rceil \cdot BL.$$

An access to the data pattern is performed group-wise, i.e. one group is accessed at a time. When $GL > D$ then more than one access is required to reed/write a whole group of data. If $(GL \bmod D) \neq 0$ then the remaining memory modules stay unused. The relation $\left\lceil \frac{BL}{D} \right\rceil \cdot GL \geq \left\lceil \frac{GL}{D} \right\rceil \cdot BL$ guarantees that the number of accesses required to access the whole pattern is minimal. The fact that any separate group inside the block can be accessed conflict-free is shown by G. Kuzmanov et al in [44].

The module assignment function has the same representation as for the case I (9):

$$m(a) = a \bmod D,$$

but the indices iterate according to the different sequence:

(i,k) = ((0,0);(0,1);...;(0,VGL-1);
(1,0);(1,1);...;(1,VGL-1);
...;
(VBL-1,0);(VBL-1,1);...;(VBL-1,VGL-1)). (13)

The number of the required accesses to read/write the whole data pattern in this case is equal to:

$$t = \left\lceil \frac{GL}{D} \right\rceil \cdot BL. \tag{14}$$

5.3.1.3 CASE III

Initial conditions:

$$S = \{\sigma \cdot 2^s \mid \sigma = 2x+1; s \neq 0\} \ \& \ GL \neq 2^{gl}, \tag{15}$$

$$\left\lceil \frac{BL}{D} \right\rceil \cdot GL < \left\lceil \frac{GL}{D} \right\rceil \cdot BL,$$

$$s \geq d.$$

In this case, the indices iterate as in (10) but here we use module assignment function from [32]:

$$m(a) = \left(a + \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D, \tag{16}$$

In fact, all initial conditions for this case exactly repeat the ones presented in [32].

The number of the required accesses to read/write the whole data pattern is described by formula (14).

5.3.1.4 CASE IV

Initial conditions:

$$S = \{\sigma \cdot 2^s \mid \sigma = 2x+1; s \neq 0\} \;\&\; GL \neq 2^{gl},\ gl \in \mathbb{N};$$

$$\left\lceil \frac{BL}{D} \right\rceil \cdot GL < \left\lceil \frac{GL}{D} \right\rceil \cdot BL; \tag{17}$$

$$s < d.$$

Again, memory access repeats the sequence (10), and the module assignment function from [32] is appropriate to the initial conditions:

$$m(a) = \left(a + \left\lfloor \frac{a}{D} \right\rfloor \bmod S' \right) \bmod D. \tag{18}$$

The number of the required accesses to read/write the whole data pattern is described by formula (14).

5.3.1.5 CASE V

Initial conditions:

$$S = \{\sigma \cdot 2^s \mid \sigma = 2x+1; s \neq 0\} \;\&\; GL = 2^{gl},\ gl \in \mathbb{N};$$

$$s \geq d. \tag{19}$$

An access to the data pattern is performed element-wise, that allows the maximum utilization of the memory modules. Power of stride is not smaller than power of the array size: $s \geq d$.

The module assignment function has the following representation:

$$m(a) = \left(a + GL \cdot \left\lfloor \frac{\lfloor a/D \rfloor}{2^{s-d}} \right\rfloor \right) \bmod D, \tag{20}$$

The sequence of indices *(i, k)* is not important in this case since all memory modules are accessed conflict-free (refer to *Theorem 3*).

The number of the required accesses to read/write the whole data pattern in this case is equal to:

$$t = \left\lceil \frac{GL \cdot BL}{D} \right\rceil. \tag{21}$$

5.3.1.6 CASE VI

Initial conditions:

$$S = \{\sigma \cdot 2^s \mid \sigma = 2x + 1; s \neq 0\} \;\&\; GL = 2^{gl},\ gl \in \mathbb{N}; \qquad s < d\ . \tag{22}$$

An access to the data pattern is performed element-wise, as in case V. Power of stride is smaller than power of the array size: $s < d$.

The module assignment function has the following representation:

$$m(a) = (a + \left(GL \cdot \left\lfloor \frac{a}{D} \right\rfloor \right) \bmod S') \bmod D, \tag{23}$$

The sequence of indices *(i, k)* is not important again since all memory modules are accessed conflict-free (refer to *Theorem 4*).

Number of the required accesses to read/write the whole data pattern in this case, as in case V, is equal to:

$$t = \left\lceil \frac{GL \cdot BL}{D} \right\rceil. \tag{24}$$

5.3.2 ROW ADDRESS FUNCTION

The row address function determines the linear address inside a memory module. An important characteristic of the proposed solution is that, in spite of having four different representations of the module assignment function, there is only one row address function which is valid for all cases described above. This feature enables large hardware design simplification as well as reduces the design time. The row address function is described by the following formula:

$$A(va,ha)=\left\lfloor \frac{va}{VD} \right\rfloor \cdot \left(\frac{N}{HD} \right)+\left\lfloor \frac{ha}{HD} \right\rfloor . \tag{25}$$

Equation (25) shows that the function is completely separable, which means that we are still able to examine vertical and horizontal constituents independently.

5.3.3 MEMORY ACCESS LATENCIES

Number of accesses that are needed to read/write the whole data pattern is described by the following equations, representing the best and the worst cases:

$$t_m^{best}=\left\lceil \frac{VGL\cdot VBL}{VD} \right\rceil \times \left\lceil \frac{HGL\cdot HBL}{HD} \right\rceil , \tag{26}$$

$$\begin{aligned} t_m^{worst} &= \left(\min(VGL,VBL)\cdot \left\lceil \frac{\max(VGL,VBL)}{VD} \right\rceil \right)\times \\ &\times \left(\min(HGL,HBL)\cdot \left\lceil \frac{\max(HGL,HBL)}{HD} \right\rceil \right). \end{aligned} \tag{27}$$

From the equations above we can derive an optimal choice for the matrix size:

$$D_{opt}=\{D:D\mid \max(GL,BL)\}. \tag{28}$$

5.4 DESIGN IMPLEMENTATION AND COMPLEXITY EVALUATION

In order to evaluate our scheme, we have verified it using a MatLab model, implemented it in VHDL and performed technology independent complexity evaluation in terms of wire complexity and logic complexity. The parallel memory controller which exploits the proposed scheme is implemented between the main memory and the processing unit (see Fig. 57). It aims at shadowing the high-latency channel to the main memory by means of a wide bus connection to the processing unit which performs parallel transmissions of data. The concurrently accessed matrix of memory modules is placed inside the memory controller. The pattern parameters are transmitted to the memory controller via programmable Special Purpose Registers (SPRs).

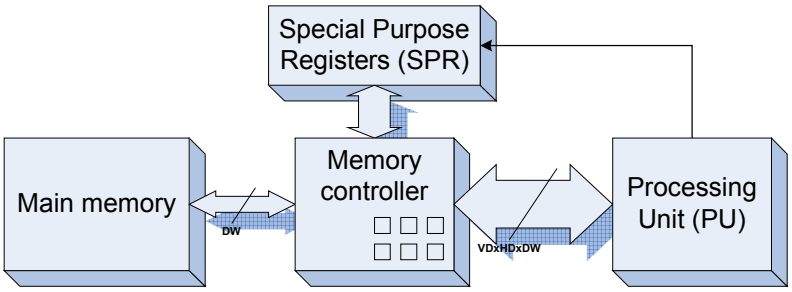


Fig. 57. Integration of parallel memory controller.

Structurally, the memory controller consists of an address generation part, a data routing part and a matrix of memory modules (see Fig. 58). The address generation part is split in vertical and horizontal sides that completely mirror each other. It includes the following blocks: mode select, address generator, set of row address generators, module assignment and address shuffle. The data routing part consists of a number of input shuffles and output de-shuffles. Refer to Appendix B for the VHDL source code of each sub-module of the design.

The critical path passes through the address generator, module assignment unit and decoding part of the shuffle. Hereafter the memory controller blocks are described in details.

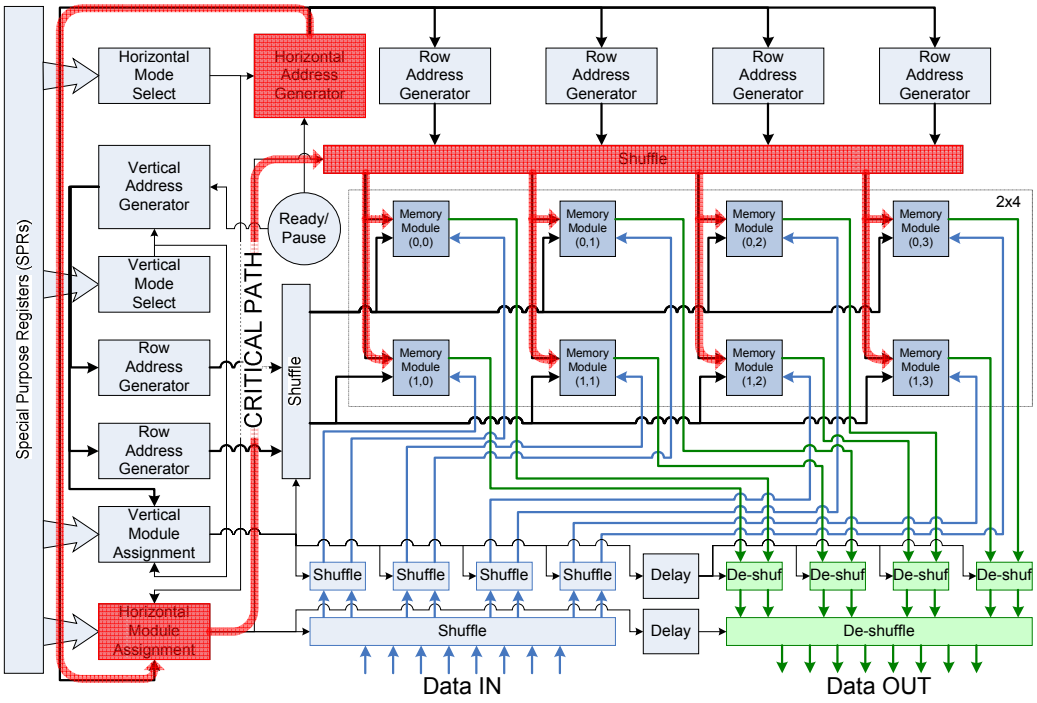


Fig. 58. Parallel memory controller block diagram.

5.4.1 MODE SELECT

The mode select unit sets the address generation logic to a mode, corresponding to the six cases of the problem partitioning (see Fig. 56). The pattern parameters stride S, group length GL and block length BL are read from the programmable SPRs.

The block implements stride oddness check and resolving of two inequalities: $\left\lceil \frac{BL}{D} \right\rceil \cdot GL \geq \left\lceil \frac{GL}{D} \right\rceil \cdot BL$ and $s \geq d$ (see Fig. 59). The *Counter** includes logic for power of two equality check, i.e. it counts number of logic '1' in the input signal: if there is only one logic '1', then the input is equal to power of two and the output is set to logic '1', otherwise the output is set to logic '0'. The *Coder* block performs coding of four 1-bit signals according to the problem partition diagram (Fig. 56) in order to create 3-bit *Mode* signal. The correspondence between cases and *Mode* signal is outlined in Table 17.

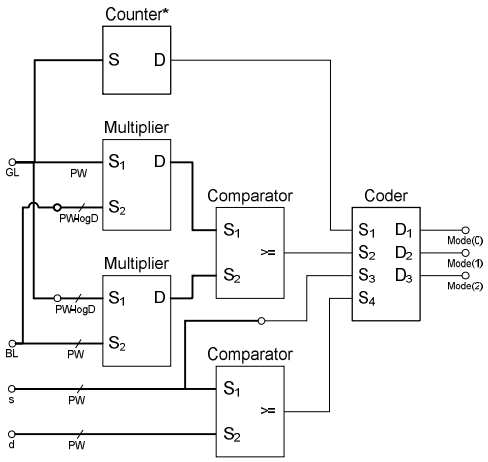


Fig. 59. Mode select block diagram.

The hardware complexity almost does not depend on the size of the matrix of memory modules $VD \times HD$, nor on the data word length W since the width of the input signal is constant and equals to 16 bits in our implementation. The wire complexity is constant and does not depend on the quantity of memory modules, data or address widths.

Table 17. Correspondence table.

Case #	*Mode signal*		
	bit 2	bit 1	bit 0
Case I.	0	0	0
Case II.	0	0	1
Case III.	0	1	0
Case IV.	0	1	1
Case V.	1	0	0
Case VI.	1	0	1
Reserved	1	1	0
Reserved	1	1	1

5.4.2 ADDRESS GENERATOR

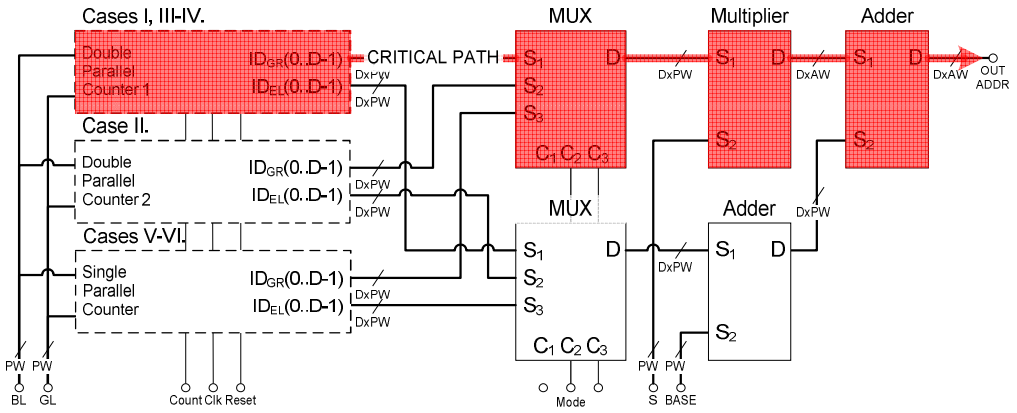
The address generator produces vertical/horizontal constituents of two-dimensional addresses of the accessed data pattern according to formula (4). Data pattern parameters are read from SPRs and an address mode is loaded from the mode select block. The address generator consists of two double parallel counters and one single parallel counter (see Fig. 60) that generate the sequence of pairs of

indices *(i, k)* or *(j, l)* (refer to Table 16, section 5.3.1 and equations (10),(13)). The double counters generate group and element indices separately (for cases I-IV), and the single counter generates group and element indices on the base of a common index by implementing respectively division and modulo by group length (for cases V-VI). Note that the group length is equal to power of two for cases V-VI therefore division and modulo operations become possible. One side of a parallel double counter is presented on Fig. 61.

The complexity of the address generator block is $O(w_A \cdot D)$ because of the multiplier with input signal width depending on the size of the matrix of memory modules.

The critical path passes from the register inside a double counter and goes through the double counter, one multiplexer, one multiplier[1] and one adder to the address output. The critical path is equivalent to $O(\log D)$.

The wire complexity is equal to $O(w_A \cdot D)$ since the address generator produces D addresses of width w_A for each of the two dimensions.



[1] In the actual VHDL implementation, this multiplier was unrolled in a set of adders and pre-calculated in parallel with the counters. This allowed to reduce the critical path though its length still equals to $O(\log D)$.

Fig. 60. Address generator block diagram.

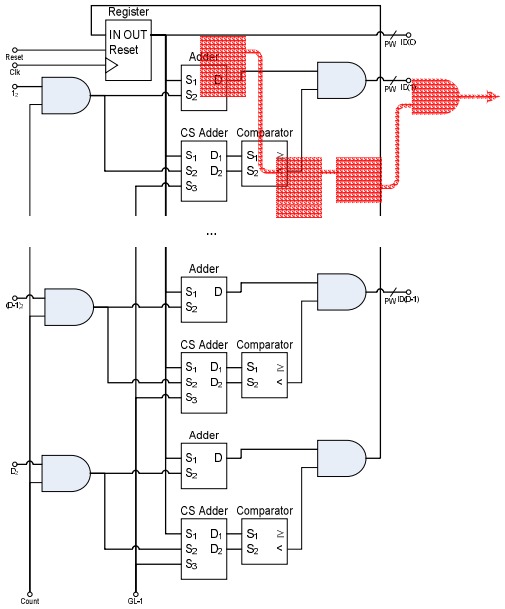


Fig. 61. Parallel counter block diagram.

5.4.3 ROW ADDRESS GENERATOR

The row address generator translates vertical/horizontal constituents of two-dimensional addresses into the physical addresses inside memory modules according to equation (25). Since formula (25) is completely separable, vertical and horizontal row address generators are implemented in separable blocks (see Fig. 58). Consequently, vertical blocks generate upper bits of the row address, and horizontal blocks generate the lower bits. No additional logic is needed to implement this block. The wire complexity is equivalent to the address width: $O(w_A)$.

5.4.4 MODULE ASSIGNMENT UNIT

The module assignment unit translates vertical/horizontal constituents of two-dimensional addresses into memory module addresses inside the matrix of memory modules according to equations (9), (16), (18), (20), and (23). Data pattern parameters are read from SPRs and an address mode is loaded from the mode select block. The equations are implemented in parallel and their outputs are multiplexed according to the address mode (see Fig. 62).

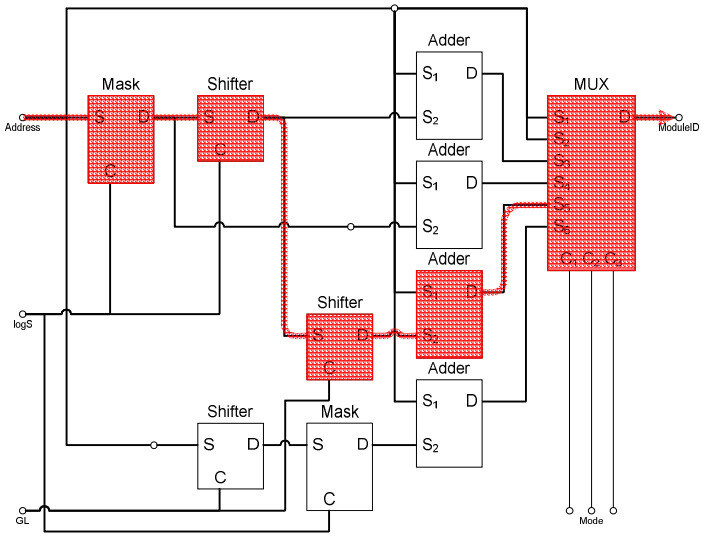


Fig. 62. Module assignment unit block diagram.

Complexity of the module assignment function for all cases (refer to equations (9), (16), (18), (20), and (23)) is presented in Table 18. The notation $x_{ms:ls}$ represents the bit interval from the least significant bit ls to the most significant bit ms. The complexity of the complete block is proportional to $O(\log D)$ because of the adders with input signals of the maximum width equal to $\log D$.

The wire complexity is equal to $O(w_A \cdot D)$ since the module assignment unit produces D results basing on the input address of width w_A.

The critical path passes through the mask unit, shifters, one adder and one multiplexer. Its length is mostly influenced by the adder of width $\log D$ and is equivalent to $O(\log(\log D))$.

Table 18. Module assignment function complexity for different cases.

Case #	*Complexity*	
Cases I-II.	$m(a) = a_{d-1:0}$	(29)
Case III.	$m(a) = \left(a_{d-1:0} + a_{s+d-1:s}\right)_{d-1:0}$	(30)
Case IV.	$m(a) = \left(a_{d-1:0} + a_{s+d-1:d}\right)_{d-1:0}$	(31)
Case V.	$m(a) = \left(a_{d-1:0} + a_{s+d-1:s} \cdot 2^{gl}\right)_{d-1:0}$	(32)
Case VI.	$m(a) = \left(a_{d-1:0} + \left(a_{8 \cdot W-1:d} \cdot 2^{gl}\right)_{s-1:0}\right)_{d-1:0}$	(33)

5.4.5 SHUFFLE UNIT

The shuffle unit is used to reorder row addresses, received from the row address generators, according to the module assignment function. It consists of a parallel set of de-multiplexers and output OR-gates (see Fig. 63).

Its complexity is $O(w_A \cdot D)$. The biggest shuffle in the design is the Data IN shuffle (see Fig. 58). Its wire complexity is equal to $O(D^2 \cdot W)$ since it has D^2 inputs and the same amount of outputs of width W.

The critical path passes through a multiplexer via its select port and does not depend on D.

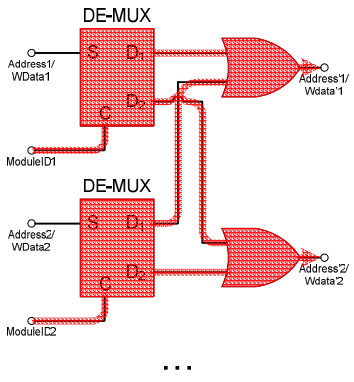


Fig. 63. Shuffle unit block diagram.

5.4.6 DE-SHUFFLE UNIT

The de-shuffle unit is needed to reorder the data from memory modules back to the initial sequence. It consists of a set of parallel multiplexers (see Fig. 64).

The complexity of the shuffle unit is $O(w_A \cdot D)$. The widest de-shuffle is situated at the Data OUT. Its wire complexity is similar to the shuffle's one and equals to $O(D^2 \cdot W)$.

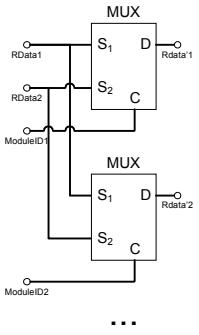


Fig. 64. De-Shuffle unit block diagram.

5.5 RESULTS

The technology independent complexity estimations from Table 19 indicate that the critical path complexity is weakly sensitive to the size of the memory matrix and thus the design is well scalable to any matrix size. In fact, the throughput is directly proportional to the matrix size *VD*×*HD*, and inversely proportional to the critical path, i.e. $throughput \propto W \cdot D^2 / \log D$.

Table 19. Summary of the technology independent design complexity evaluation.

Design unit	*Logic complexity*	*Wire complexity*	*Critical path*
Mode select	$O(const)$	$O(const)$	-
Address generator	$O(w_A \cdot D)$	$O(w_A \cdot D)$	$O(\log D)$
Row address generator	0	$O(w_A)$	-
Module assignment unit	$O(\log D)$	$O(w_A \cdot D)$	$O(\log(\log D))$
Shuffle	$O(w_A \cdot D)$	$O(D^2 \cdot W)$	$O(const)$
De-shuffle	$O(w_A \cdot D)$	$O(D^2 \cdot W)$	-
Total	$O(w_A \cdot D)$	$O(D^2 \cdot W)$	$O(\log D)$

5.5.1 ASIC SYNTHESIS

The synthesis was performed for an ASIC 90 nm CMOS technology. The results for six different matrix sizes, word widths *W* of 32 and 64 bits, and 12-bit addresses are presented in Table 20 and Fig. 65. In fact, data word width of 8 Bytes corresponds to utilization of two concurrently coupled 32-bit wide memory modules. Generally speaking, the address width ranging from 8 till 16 bits is enough for the most of practical applications, which would give the complexity range of 45.5-53.9 Kgates for 4×4 matrix with *W* = 32 bits. Considering that the memory modules used in the design have 4096×32bits size and occupy 43.8 Kgates, the logic complexity overheads vary from 14.5% for 2×2 32-bits matrix to 3.8% for 8×8 64-bits matrix with respect to the total hardware complexity. The presented synthesis results confirm the linear increase of the design complexity and the quadratic increase of the throughput, derived from our theoretical estimations. As it was expected, the critical path is proportional to the

logarithm of the matrix size along one dimension and the design complexity depends linearly on it.

Table 20. Synthesis results for ASIC 90 nm.

Matrix size	*Complexity (KGates)*		*Frequency (MHz)*		*Throughput (Gbits/sec)*	
	W=4	*W*=8	*W*=4	*W*=8	*W*=4	*W*=8
2×2	25.34	26.73	377	371	44.94	88.45
2×4	33.81	39.11	341	336	81.30	160.21
2×8	58.48	70.19	314	321	149.72	306.12
4×4	46.60	53.27	336	333	160.21	317.57
4×8	88.07	101.57	321	314	306.12	598.90
8×8	176.83	211.06	313	310	597.00	1182.55

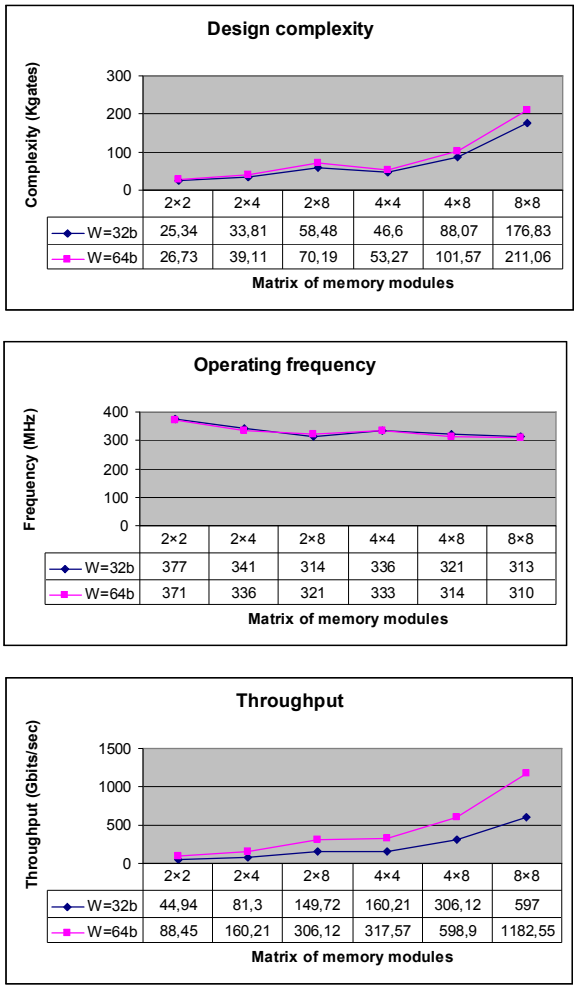


Fig. 65. Synthesis results for ASIC 90 nm: design complexity, frequency and throughput.

5.5.2 FPGA SYNTHESIS

The FPGA synthesis was performed with Xilinx ISE 8.2i toolset for Virtex2P xc2vp30-7ff896 device with speed coefficient -7. The results are presented in Table 21.

Table 21. FPGA synthesis results.

Matrix size	IO ports utilization (%)			Frequency (MHz)			Throughput (Gbits/sec)		
	W=1	W=2	W=4	W=1	W=2	W=4	W=1	W=2	W=4
2×2	16	22	58	128.8	128.8	108.4	4.12	8.24	13.87
2×4	22	34	-	123.3	123.3	-	7.89	15.78	-
2×8	35	58	-	134.4	134.4	-	17.20	34.40	-
4×4	35	58	-	124.6	124.6	-	15.94	31.89	-
4×8	61	-	-	133.2	-	-	34.09	-	-

In contrast to ASIC, design for FPGA is much more sensitive to the wire complexity which is significant for the parallel systems. Therefore, FPGA technology enables implementation of a restricted variety of configurations outlined in Table 21. On the other hand, FPGA technology allows mapping the design within a short timeframe and performing the experiments on real-applications to prove the concept of parallel memory access with configurable 2D data patterns.

5.6 COMPARISON WITH THE RELATED WORK

Studying the related research it is striking that, although a lot of work has been carried on the subject of memory access acceleration and many mechanisms have been proposed (refer to section 5.1.2), only few of them were implemented in hardware, also having the results published. At least to our knowledge, the most significant of them are presented in Table 22. The table shows the comparison with the related schemes from [13] and [12]. The operating frequencies are normalized to 90nm technology node according to [4].

Usually, the address calculation is a very design critical mechanism because it performs complex arithmetical calculations, such as division and multiplication. The problem becomes more complicated when a divisor or multiplier factors are not equal to power of two. Therefore, it becomes extremely important to optimize these calculations on both algorithmic and hardware levels. The authors of [13] and [12] decided to divide the address computation on pipelined stages, thus reducing the critical path of the computational block and rising the total system frequency. The main drawback of this solution lies in the fact that, in spite of the high frequency, the *address calculation latency* becomes longer. For example, in [13] it takes 11 clock cycles (excluding 6 cycles of DRAM access) to calculate the address in order to read data from memory. In comparison, an imaginary mechanism with the operating frequency 10 times slower than the one from [13] but having the address calculation latency equal to one clock cycle will operate faster. This situation is very well illustrated by comparing the technique from [12] and our mechanism. Although the total operating frequency of the device proposed by [12] is higher, it takes $3clk/539MHz \approx 5.6ns$ to calculate the read address (see Table 22), while in our case it will take only $1clk/393MHz \approx 2.5ns$, which is more than two times faster.

Table 22. Comparison to the schemes with 8 memory modules and 8 bits data width.

Design	*Supported data patterns*	*Address calculation latency (r/w cycles)*	*Frequency, normalized to 90nm CMOS (MHz)*	*Complexity scaling*	*Area (Kgates)*
CPMA [13]	Generate, crumbled rectangle, chess board, vector, free	11/8 pipelined	~32	$O(w_A \cdot N_{total})$	~27.0
PMAS [12]	Stridden vector	3/2 pipelined	539	$O(w_A \cdot N_{total})$	5.5
This proposal	Block cyclic	1/1	393	$O(w_A \cdot \max(VD, HD))$ $N_{total} = VD \cdot HD$	26.9

We will define the *memory access latency* as the time (in clock cycles) which is needed to move the complete data pattern to/from the memory. The memory access latency unifies the address calculation latency and the amount of transfers required to read/write the complete data pattern. For the proposed memory organization, the memory access latency is calculated by formulas (26) and (27). If the complete data pattern fits the matrix of memory modules (i.e. the amount of the accessed data elements is equal to the number of available memory modules), then the memory access latency is equal to one clock cycle in the best case and four cycles in the worst case. If the data pattern does not fit in the matrix, then the memory access latency will be as long as it is required to read/write all data elements. The latest is also true for the other mechanisms from Table 22.

The situation with "area *vs.* complexity scaling" reminds of the one with "frequency *vs.* address calculation latency" described above. The demolition return of less optimized complexity scaling factor will significantly influence the total design area, especially for the larger devices with very wide memory channel. Table 22 presents area occupation for the designs with 8 memory modules (in our case it corresponds to $VD = 2$ and $HD = 4$), 10-bit address width and 8-bit data width. In our architecture, the logic complexity scales equivalently to the longest side of the matrix of memory modules (for the case examined it Table 22, it is $HD = 4$), while for the other designs, the logic complexity is equivalent to the total number of memory modules. Note, that the address and data widths have only minor influence on the total design complexity (compare

values for $W = 4$ and $W = 8$ from Table 20). For example, consider the proposed architecture with $D = VD = HD$; thus, the total number of memory modules is $N_{total} = VD \cdot HD = D^2$, meaning that the length of any side of the matrix is $D = \sqrt{N_{total}}$. Now the difference of the complexity scaling factors is evident: $O(w_A \cdot \sqrt{N_{total}})$ for the proposed architecture, and $O(w_A \cdot N_{total})$ for the other architectures.

One also should take into account that the die area occupation is not the primary ASIC design objective nowadays. With the technology development, gate density is constantly increasing, while the functionality does not keep pace with such advance. In other words, contemporary chips provide great deal of hardware resources that are not completely utilized. Consequently, it seems more reasonable to expand device flexibility and reusability at the expense of additional gates occupation, thus reducing the design cycle, than to press towards development of the smallest device with the restricted application domain and to redesign everything from scratch when the requirements are slightly changed.

Another important feature of the proposed memory organization is *modularity*. At the design time, it is possible to limit the variety of supported data patterns. This will reduce the logic complexity of the Mode select unite, which is in charge with memory access mode, as well as Address generator and Module assignment unit; plus this will shorten the critical path of the latest two blocks. In addition, there is a possibility to adjust address width, data width, and the size of the matrix of memory modules at the design time. Thus, the proposed memory organization can be very well adapted to a specific application domain being at the same time significantly optimized.

5.7 SUMMARY

High throughput memory accesses with flexible data patterns are widely used in many different areas such as multimedia, telecommunications, and scientific applications. We presented a parallel memory organization that accumulates the advantages of the previous solutions. In addition, it allows access to a data pattern with more complex structure and relaxes the limitations of the data pattern parameters. Runtime programmability by means of SPRs enables flexible data management. Our theoretical conclusions were proved, and additionally confirmed by mathematical modeling. The design implementation and synthesis showed expected results according to our theoretical estimations. As a result, our memory organization provides minimum latency between main memory and processing unit for a given type of schemes.

CHAPTER 6

CONCLUSION

This chapter summarizes all achievements of the presented research. The distinguished contributions of this work are listed in the respective section. Finally, a possible direction of the future work is outlined and briefly discussed.

6.1 SCIENTIFIC RESULTS

This thesis has examined memory organization and data structure in heterogeneous reconfigurable Systems-on-Chip. The target framework lies within the context of the MORPHEUS project.

As a result, data in the system was structured in four separated flows: computational, configuration, control and I/O data. Such distributed organization makes the computational model of the system very clear and supports the solution for interconnects.

A hierarchically organized memory subsystem makes it possible to separate communication tasks and computation tasks, allowing computation accelerators to process data without interrupting on data routing tasks. Currently processed data is placed into local storages and transferred between the computation resources without interacting with slow and power consuming off-chip memories.

Specially developed mechanisms allow operating with data chunks of various granularity on all levels of memory hierarchy and supplying computational engines with flexible memory access. In this scope, the Address Generator integrated in DREAM architecture provides the reconfigurable data-path with the access to a programmable data pattern for all its inputs and outputs in parallel.

A more general mechanism for memory access was developed and implemented: starting from theoretical basis until the synthesis and optimization phases. This mechanism provides for a computational unit parallel two-dimensional access to the memory with programmable data pattern. The technique supports an advanced set programming parameters enabling greater variety of the supported data patterns. The mechanism is implemented as a stand-alone IP being compliant with wide range of complex hardware architectures requiring fast parallel programmable access to the memory, including MORPHEUS platform.

6.2 CONTRIBUTION STATEMENT

Since the research was performed within the framework of a bigger project, it is necessary to specify the contribution of this work. The following main tasks were performed:

1. Definition of the data structure in the system and proposal of communication means for the data-flows ([34]).
2. Proposal on the three-level hierarchical memory organization with functional definition and architectural allocation of each layer ([34]).
3. Definition of the number and sizes of the memory units implemented in the system on the basis of target applications analysis ([34]).
4. Implementation of the complete MORPHEUS platform on the RTL level, including fine tuning of the integrated IP from the project partners and simulation of the architecture ([34] and [23]).
5. Definition of data exchange on the local buffers and synchronization means between reconfigurable engines ([23] and [21]).
6. Development and integration of the Address Generator for the DREAM architecture ([23] and [21]).
7. Taking part into DREAM and M2000 integration into MORPHEUS platform ([23] and [21]).
8. Development of two-dimensional memory architecture with multiple data patterns. The work included all stages starting from proof of the theoretical concepts, through the mathematical modeling in MatLab to the VHDL implementation, simulation, verification, optimization and synthesis for two target technologies: ASIC and FPGA ([75], [76], and another article is currently under review for IEEE Transactions on Circuits and Systems II).
9. Application mapping the reconfigurable engines for the verification purpose and architecture efficiency proof ([23] and [21]).

Thus, the work was performed on various stages of the design cycle and levels of application, however, targeting the same objective of data storage development and data structure organization.

6.3 FUTURE WORK

As it always happens, one step forward in any scientific field opens more questions than it actually answers. Reconfigurable architectures and heterogeneous systems are an ever evolving research topic, therefore there are many directions in which the work of this thesis can be extended upon. One of the possibilities is presented below.

A hardware design should be followed by an application approval. There are a number of tasks within MORPHEUS that would be interesting and useful to test, such as application mapping. It might be of particular importance since it proves system functionality and shows memory organization efficiency. The feedback from this work may influence both the resource mapping decisions as well as hardware improvements. As a consequence, the data flow optimization and application mapping together with the system design implementation might be beneficial as a proof of the overall system efficiency.

Among the applications offered by MORPHEUS partners, systems for intelligent cameras seem to be the most suitable for this purpose. On the one hand, it has a sufficient level of concurrency introducing the wide specter of mapping varieties. This feature opens a large area for the research and decision justifications. On the other hand, the application has reasonable complexity to serve as a test application. The sources of the application kernels are open and can be easily found from the open sources.

Application mapping workflow can be divided on several stages.

1. *Application study.* On the first stage, the application is defined and organized for the following consideration. The kernels are thoroughly examined for the purpose of data bandwidth requirements and logic complexity. The result of this stage should be the table with the characteristics of the kernels.
2. *Kernels mapping.* On the mapping stage, the dependencies between separate kernels within the application are disclosed and outlined on the dataflow graph. Moreover, the reconfiguration requirements should be taken into account, i.e. static and/or dynamic reconfiguration overheads.

The conclusions should present available tradeoffs for kernels mapping and one (or a number of) suggested mapping(s).

3. *Tool set organization.* The HRE models from all partners must be put together and integrated in one simulation system (written on SystemC). All models must be tuned in order to function as stand-alone models as well as in a common simulation environment. This stage is particularly important as it provides the tool set for the whole further research.
4. *Source code porting.* During the application porting stage, kernels' source codes are adapted to a particular hardware engine. Each kernel should be maximally optimized for its engine in order to achieve the best performance. This phase includes performing simulations on the stand-alone simulators (PiCoGA, XPP, and M2000) and analyzing the results. In case of dynamical reconfiguration, the particular emphasis must be placed on configuration flow management. This work should be performed in close cooperation with the mapping stage. As a result, an adopted simulation ready code must be produced.
5. *Experiment planning.* The simulation strategy planning phase includes experimental setup and metrics evaluation. The experimental setup will define the initial conditions and hardware configuration for the following simulation process. It is necessary to define metrics which will be used in the simulation stage as well as a basis for the final results analysis and overall efficiency evaluation. Also some theoretical estimation should be made. Thus, the output of this stage will include the definition of simulation flow and evaluation metrics.
6. *Simulation.* After the preparation work the set of the simulations should be performed. The number of simulations and simulations parameters are taken from experiment planning phase. The produced results should be organized into visual forms and be prepared for the following analysis.
7. *Results evaluation and design space exploration (DSE).* Usually it is the most time consuming part of a project. On this stage the simulation results are analyzed in order to produce the feedback to the mapping and porting stages. Essential corrections might be done in the source code as well as

kernels mapping. After this, the experiments are repeated. Such iterations should be continued until the obtained results suit theoretical estimations. The final results should give an evaluation of the overall hardware architecture efficiency and application mapping complexity, as well as the results of the particular mapping characteristics.

APPENDIX A. EXAMPLES OF C-SOURCE CODES FOR THE MORPHEUS PLATFORM

ARM control code

```
#include "morpheus.h"
#include "gpio.h"
#include "frames.h"
#include "mpmc.h"
#include "vic_pl190.h"
#include "interpolate.dreambitstream.h"
#include "exapp.m2kbitstream.h"
#include "string.h"

void C_Entry()
{
  int i;

  // INITIALIZATION PHASE
  sram_init();                          // Initialize the SRAM controller
  intctlInit();                         // Initialize the Interrupt Controller
  intctlIntEnable(0x0000FFFF);          // Enable IRQs
  intctlIntRegister((1<<9), p_nIRQ_9, 1); // Vectored IRQ number 9 (dream printf Interrupt)


  // Set M2K Parameters ------------------------------------------------
  qprintf("Launch M2K.. \n",0,0,0,0);
  m2k_XR->clock_mode = global_clock;
  m2k_XR->resetn1 = 1;
  m2k_upload_bitstream(m2k_exapp);

  // Program M2K DEBs in FIFO mode
  for(i=0;i<8;i++)
    (m2k_XR->deb[i]).dontusefifo = 0;

  // Setting direction for used FIFOs
  // Input FIFOs
  m2k_XR->deb[0].direction = 1;
  m2k_XR->deb[1].direction = 1;
```

```
  m2k_XR->deb[2].direction = 1;
  // Output FIFOs
  m2k_XR->deb[3].direction = 0;
  // ------------------------------------------------------------------------

  // Set DREAM Parameters ------------------------------------------------
  // Load bitstream on DREAM CEBs (Note DREAM Does not need to be active)
  load_dream_bitstream(0,  (unsigned int*)cm_segment,cm_size,
                           (unsigned int*)dm_segment,dm_size,
                           (unsigned int*)pm_segment,pm_size);

  // Lauching Computation on Dream
  qprintf("Launch DREAM.. \n",0,0,0,0);
  dream_XR->ARM_GP0 = BLOCK_SIZE;

  dream_XR->clock_mode = global_clock;
  dream_XR->resetn1 = 1;
  dream_XR->enable = 1;

  // ------------------------------------------------------------------------

{
   // This code implements the PN pattern exploiting Ping-pong buffering on the DREAM XR,
   // while M2K DEBs are programmed as FIFOs according to the KPN paradigm
   int block,frame;
   for(block=0;block<STREAM_SIZE/BLOCK_SIZE+2;block+=2)
    {

      // Loading M2K Input DEBS 01
      if(block<=STREAM_BLOCKS-1)
         mem_to_debs((int)input_frames+block*BLOCK_SIZE*INPUT_FRAME_SIZE,
                     2*BLOCK_SIZE*INPUT_FRAME_SIZE,
                     (int)m2k_debs->pair01, DEB64, 0);

      // VBUF 0: Loading DREAM Input DEBS 01  ---------------------
      //         Moving from DREAM DEB 2 to M2K DEB 2
      qprintf("Block %d \n",block,0,0,0);
      dream_XR->arm_lock0=1;while(dream_XR->arm_lock_ack0!=1);
      if(block<=STREAM_BLOCKS-1)
```

```
        mem_to_debs((int)input_frames+block*BLOCK_SIZE*INPUT_FRAME_SIZE,
                    BLOCK_SIZE*INPUT_FRAME_SIZE,
                    (int)dream_debs->pair01, DEB64, 0);
      if(block>=2)
        for(frame=0;frame<BLOCK_SIZE;frame++)
          debs_to_debs((int)dream_debs->pair23, frame*INPUT_FRAME_SIZE/8,
                       (int)m2k_debs->pair23, 0,
                       DEB32, AVG_FRAME_SIZE);
      dream_XR->arm_lock0=0; while(dream_XR->arm_lock_ack0!=0);
      // ---------------------------------------------------------------------------

      // VBUF 1: Loading DREAM Input DEBS 45 ---------------------
      //         Moving from DREAM DEB 6 to M2K DEB 2
      qprintf("Block %d \n",block+1,0,0,0);
      dream_XR->arm_lock1=1;while(dream_XR->arm_lock_ack1!=1);
      if(block<=STREAM_BLOCKS-1)
        mem_to_debs((int)input_frames+(block+1)*BLOCK_SIZE*INPUT_FRAME_SIZE,
                    BLOCK_SIZE*INPUT_FRAME_SIZE,
                    (int)dream_debs->pair45, DEB64, 0);
      if(block>=2)
        for(frame=0;frame<BLOCK_SIZE;frame++)
          debs_to_debs((int)dream_debs->pair67, frame*INPUT_FRAME_SIZE/8,
                       (int)m2k_debs->pair23, 0,
                       DEB32, AVG_FRAME_SIZE);
      dream_XR->arm_lock1=0;while(dream_XR->arm_lock_ack1!=0);
      // ---------------------------------------------------------------------------

      // Retrieving M2K Output DEB 3
      if(block>=2)
        debs_to_mem((int)m2k_debs->pair23+4,DEB8,0,
                    (int)result_frames+(block-2)*BLOCK_SIZE*OUTPUT_FRAME_SIZE,
                    2*BLOCK_SIZE*OUTPUT_FRAME_SIZE);
    }
  }
#endif

  // Switching off computation
  dream_XR -> clock_mode = grounded_clock;
  m2k_XR   -> clock_mode = grounded_clock;
```

```
  clk_stop();
}
```

NoC transfer code

```
//Initialization phase-------------------------------------
struct config channel_cfg;
struct lli    channel_lli;

channel_cfg.cfgl    = 0;
channel_cfg.cfgh    = 0;
channel_cfg.sstatar = 0;
channel_cfg.dstatar = 0;
channel_cfg.sgr     = 0;
channel_cfg.dsr     = 0;
channel_cfg.channel = #_channel;

channel_lli.sar   = source_address;
channel_lli.dar   = destination_address;
channel_lli.ctll  = 0;
channel_lli.ctlh  = 0;
channel_lli.sstat = 0;
channel_lli.dstat = 0;

//Configuration phase--------------------------------------
//Set the transfer size in Bytes
changeBits(&channel_lli.ctlh, BLOCK_TS, BLOCK_TS_S, 1024);

//Set the transaction burst lengths for source and destination ports
changeBits(&channel_lli.ctll, SRC_MSIZE, SRC_MSIZE_S, 2);
changeBits(&channel_lli.ctll, DST_MSIZE, DST_MSIZE_S, 2);

//Set the data width for src. and dest. ports
changeBits(&channel_lli.ctll, SRC_TR_WIDTH , SRC_TR_WIDTH_S, 2);
changeBits(&channel_lli.ctll, DST_TR_WIDTH , DST_TR_WIDTH_S, 2);

//Set the DMA master channels
//SMS - Source Master Select, is connected to the memory unit
changeBits(&channel_lli.ctll, SMS, SMS_S, 1);
//DMS - Destination Master Select, is connected to the NoC initiator port
```

```
changeBits(&channel_lli.ctll, DMS, DMS_S, 0);

//SB - Single block transfer-----------------------------------
while (e != OK)
  e = transfer(SB, &channel_cfg, &channel_lli);

//Complete the transfer and disable the channel-----------
setBits(&channel_lli.ctll, INT_EN);          //Enable interrupts
maskInt(DMA_engine_ID, I_BLOCK, 0, TRUE);    //Block-complete interrupt
maskInt(DMA_engine_ID, I_TFR, 0, TRUE);      //Transfer-complete interrupt

=========================================================

//Initialization phase-----------------------------------
…

//Configuration phase------------------------------------
…

//AR_MB - Multi-block transfer
while (e != OK)
  e = transfer(AR_MB, &channel_cfg, &channel_lli);

//Upon completion of the transfer the configuration is automatically reloaded and a HW interrupt is
set. It then stalls until the block-complete interrupt is cleared by software. If the interrupts are
disabled or masked, the hardware does not stall until it detects a write to the block-complete
interrupt clear register; instead, it immediately starts the next block transfer. In this case,
software must clear the reload bits in the Configuration register.
```

APPENDIX B. VHDL SOURCES

Mode select

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.gsma_package.all;

entity mode_select is
  generic (
    logD : natural := 1);              -- Logarithm to the base 2 of number of
                                       -- modules along one dimention
  port (
    logS    : in  std_logic_vector(logParam_width - 1 downto 0);  -- Logarithm to
                                                                  -- the base 2 of
                                                                  -- S
    GL      : in  std_logic_vector(param_width - 1 downto 0);  -- Group length
    BL      : in  std_logic_vector(param_width - 1 downto 0);  -- Block length
    mode    : out std_logic_vector(2 downto 0)                 -- Case enable
    );
end mode_select;

architecture Behavioral of mode_select is
  signal stride_oddness             : std_logic;  -- '1' = odd stride
  signal GL_norm, BL_norm           : unsigned(2 * param_width - logD - 1 downto 0);
  signal pot                        : boolean;

begin  -- Behavioral
  stride_oddness <= '1' when conv_integer(unsigned(logS)) = 0 else  -- S(0);
                    '0';
  GL_norm <= conv_unsigned(unsigned(BL(param_width - 1 downto logD)) * unsigned(GL), 2 * param_width
- logD) when
             conv_integer(unsigned(BL(logD - 1 downto 0))) = 0 else
             conv_unsigned((unsigned(BL(param_width - 1 downto logD)) + 1) * unsigned(GL), 2 *
param_width - logD);
  BL_norm <= conv_unsigned(unsigned(GL(param_width - 1 downto logD)) * unsigned(BL), 2 * param_width
- logD) when
```

```
            conv_integer(unsigned(GL(logD - 1 downto 0))) = 0 else
            conv_unsigned((unsigned(GL(param_width - 1 downto logD)) + 1) * unsigned(BL), 2 *
param_width - logD);
  mode <= "000" when (stride_oddness = '1') and (GL_norm < BL_norm) else  -- Case I.
          "001" when ((stride_oddness = '1') and (GL_norm >= BL_norm)) or
          ((stride_oddness = '0') and (not po2(GL)) and (GL_norm >= BL_norm)) else  -- Case II.
          "010" when (stride_oddness = '0') and (not po2(GL)) and
          (GL_norm < BL_norm) and (unsigned(logS) >= logD) else  -- Case III.
          "011" when (stride_oddness = '0') and (not po2(GL)) and
          (GL_norm < BL_norm) and (unsigned(logS) < logD) else  -- Case IV.
          "100" when (stride_oddness = '0') and po2(GL) and (unsigned(logS) >= logD) else  -- Case
V.
          "101" when (stride_oddness = '0') and po2(GL) and (unsigned(logS) < logD) else  -- Case
VI.
          "XXX";
end Behavioral;
```

Address generator

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.gsma_package.all;

entity AddrGen is
  generic (
    D    : natural;                    -- Number of memory modules along one dimension
    logD : natural
    );
  port (
    clk          : in  std_logic;
    RESETn       : in  std_logic;
    ready        : in  std_logic;       -- Enables addr generation process
    mode         : in  std_logic_vector(2 downto 0);
    B            : in  std_logic_vector(addr_width - 1 downto 0);  -- Linear base address
    S            : in  std_logic_vector(param_width - 1 downto 0);  -- Stride
    GL           : in  std_logic_vector(param_width - 1 downto 0);  -- Group length
    BL           : in  std_logic_vector(param_width - 1 downto 0);  -- Block length
    output_valid : out std_logic_vector(D - 1 downto 0);  -- Output velidness
```

```
    last_output  : out std_logic;       -- Signals the last output address
    a            : out addr_bus(D - 1 downto 0));  -- Output addresses
end AddrGen;

architecture Behavioral of AddrGen is
  type index_type is array (natural range <>) of std_logic_vector(param_width - 1 downto 0); --
natural range 0 to 2**param_width - 1;

  signal i, k                           : index_type(D downto 0);
  signal ind                            : std_logic_vector(param_width - 1 downto 0); --natural
range 0 to 2**param_width - 1;
  signal carry_i, carry_k, carry_ind    : std_logic_vector(param_width - 1 downto 0); --natural
range 0 to 2**param_width - 1;
  signal carry_last_output, last_output_i : std_logic;
  signal output_valid_i                 : std_logic_vector(D - 1 downto 0);
  signal ibyS, iS_prec                  : index_type(D downto 0);
  signal carry_ibyS                     : std_logic_vector(param_width - 1 downto 0);

begin  -- Behavioral

  TPC_REGs: process (clk, RESETn)
  begin  -- process
    if RESETn = '0' then               -- asynchronous reset (active low)
      carry_i     <= (others => '0');
      carry_k     <= (others => '0');
      carry_ind   <= (others => '0');
      carry_ibyS  <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      -- Counter indices
      if ready = '1' then
        carry_i   <= i(D);             -- i + 1
        carry_k   <= k(D);             -- k + 1
        carry_ind <= ind;              -- (i * k) + 1
        carry_ibyS <= ibyS(D);         -- (i * S) + S
      end if;
    end if;
  end process TPC_REGs;

  -- Preculculation of the (i * S) product
```

```
iS_prec(0) <= carry_ibyS;
ibyS(0) <= carry_ibyS;
MULT_PREC: for id in 1 to D generate
  iS_prec(id) <= iS_prec(id - 1) + S;  -- precalculated value
  ibyS(id) <= iS_prec(id)  when i(id) > i(id - 1) else
              ibyS(id - 1) when i(id) = i(id - 1) else
              (others => '0');
end generate MULT_PREC;

-- Tripple Parallel Counter
TPC: process (mode, GL, BL, carry_k, carry_i, carry_ind)
  variable i_id, k_id, ind_id, temp_id   : natural;
  variable save_i : natural;
  variable lst_output, output_valid_i_id : std_logic;
  variable temp : std_logic_vector(param_width - 1 downto 0);
begin

  k(0)              <= carry_k;
  i(0)              <= carry_i;
  output_valid_i(0) <= '1';

  case mode is

    when "001" =>                    -- Group-wise access
      -- Current indices
      for id in 1 to D - 1 loop
        if carry_k + id > GL - 1 then
          k(id) <= GL - 1;
          output_valid_i(id) <= '0';
        else
          k(id) <= carry_k + id;
          output_valid_i(id) <= '1';
        end if;
        i(id) <= carry_i;
      end loop;  --id
      -- Next carry indices
      if carry_k + D > GL - 1 then
        k(D) <= (others => '0');                    -- reset k syncronously
        if carry_i = BL - 1 then
```

```
      i(D) <= (others => '0');                -- reset i syncronously
      last_output_i <= '1';
    else
      i(D) <= carry_i + 1;
      last_output_i <= '0';
    end if;
  else
    k(D) <= carry_k + D;
    i(D) <= carry_i;
    last_output_i <= '0';
  end if;
  ind <= carry_ind;

when "000" | "010" | "011" =>     -- Access based on the set of elements
  -- Current indices
  for id in 1 to D - 1 loop
    if carry_i + id > BL - 1 then
      i(id) <= BL - 1;
      output_valid_i(id) <= '0';
    else
      i(id) <= carry_i + id;
      output_valid_i(id) <= '1';
    end if;
    k(id) <= carry_k;
  end loop;  --id
  -- Next carry indices
  if carry_i + D > BL - 1 then
    i(D) <= (others => '0');
    if carry_k = GL - 1 then
      k(D) <= (others => '0');
      last_output_i <= '1';
    else
      k(D) <= carry_k + 1;
      last_output_i <= '0';
    end if;
  else
    i(D) <= carry_i + D;
    k(D) <= carry_k;
    last_output_i <= '0';
```

```
      end if;
      ind <= carry_ind;

    when "100" | "101" =>             -- Element-wise access
      -- Current indices
      for id in 1 to D - 1 loop
        if (carry_k + id = GL - 1) and (carry_i + id = BL - 1) then
          k(id) <= GL - 1;
          i(id) <= BL - 1;
          output_valid_i(id) <= '0';
        else
          k(id) <= carry_k + id; -- mod2(carry_ind + id, GL);
          i(id) <= div2(carry_ind + id, GL);
          output_valid_i(id) <= '1';
        end if;
      end loop;  -- id
      -- Next carry indices
      if (carry_k + D = GL - 1) and (carry_i + D = BL - 1) then
        ind  <= (others => '0');
        k(D) <= (others => '0');
        i(D) <= (others => '0');
        last_output_i <= '1';
      else
        ind <= carry_ind + D;
        k(D) <= carry_k + D; -- mod2(carry_ind + D, GL);
        i(D) <= div2(carry_ind + D, GL);
        last_output_i <= '0';
      end if;

    when others =>
      k             <= (others => (others => 'X'));
      i             <= (others => (others => 'X'));
      ind           <= (others => 'X');
      output_valid_i <= (others => 'X');
      last_output_i  <= 'X';

  end case;

end process TPC;
```

```
  -- Outport connections
  ADDR_BUS : for m_id in 0 to D - 1 generate
    a(m_id) <= std_logic_vector(conv_unsigned(unsigned(B + ibyS(m_id) + k(m_id)), addr_width)); --
implicit multiplication and base address
  end generate ADDR_BUS;
  last_output  <= last_output_i;
  output_valid <= output_valid_i;

end Behavioral;
```

Row address generator

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.gsma_package.all;

entity RowGen is
  generic (
    logD :     natural := 1);          -- Logarithm to the base 2 of number of modules along one
dimention
  port (
    a    : in  std_logic_vector(addr_width - 1 downto 0);   -- One dimention constituent of the
input address
    row  : out std_logic_vector(addr_width - 1 downto 0));  -- One dimention constituent of the row
address
end RowGen;

architecture Behavioral of RowGen is

  signal row_un, a_un : unsigned(addr_width - 1 downto 0);

begin  -- Behavioral

  -- Shifter is implemented afterwords as high and low memory module addresses (simplified)
  row(addr_width - logD - 1 downto 0)          <= a(addr_width - 1 downto logD);
  row(addr_width - 1 downto addr_width - logD) <= low_vector(logD - 1 downto 0);
```

```
end Behavioral;
```

Module assignment unit

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.gsma_package.all;

entity ModuleAssign is

  generic (
    logD : natural := 1);
  port (
    a        : in  std_logic_vector(addr_width - 1 downto 0);  -- One dimention constituent of the
input address
    logS     : in  std_logic_vector(logParam_width - 1 downto 0);  -- Logarithm to the base 2 of S
    GL       : in  std_logic_vector(param_width - 1 downto 0);  -- Group length
    mode     : in  std_logic_vector(2 downto 0);  -- Case enable
    ModuleID : out std_logic_vector(logD - 1 downto 0));  -- Module select
end ModuleAssign;

architecture Behavioral of ModuleAssign is

  signal GL_un                 : unsigned(param_width - 1 downto 0);
  signal m1, m2, m3, m4, m5, m6 : unsigned(logD - 1 downto 0);
  signal prod5, prod6          : unsigned(addr_width - 1 downto 0);
  signal a_msk3, a_msk6, a_sh3  : std_logic_vector(addr_width - 1 downto 0);
  signal logS_nat              : natural;
  signal ModuleID_tmp          : std_logic_vector(logD - 1 downto 0);

begin  -- Behavioral

  logS_nat <= conv_integer("0" & logS);

  -- CASE I.
  m1 <= unsigned(a(logD - 1 downto 0));  -- m1 = a[d - 1 : 0]

  -- CASE II.
```

```
  m2 <= unsigned(a(logD - 1 downto 0));  -- m2 = a[d - 1 : 0]

  -- CASE III.
  masking3 : for i in 0 to addr_width - 1 generate
    a_msk3(i) <= a(i) when i < logS_nat + logD else  -- a_msk3 = a[s + d - 1 : 0]
                 '0';
  end generate masking3;
  a_sh3       <= to_stdLogicVector(to_bitVector(a_msk3) srl logS_nat);  -- a_sh3 = a[s + d - 1 : s]
  m3          <= conv_unsigned(unsigned(a(logD - 1 downto 0)) + unsigned(a_sh3), logD);
                                       -- m3 = (a[d - 1 : 0] + a[s + d - 1 : s])[d - 1 : 0]

  -- CASE IV.
  m4 <= conv_unsigned(unsigned(a(logD - 1 downto 0)) + unsigned(a_msk3(addr_width - 1 downto logD)),
logD);
                                       -- m4 = (a[d - 1 : 0] + a[s + d - 1 : d])[d - 1 : 0]

  -- CASE V.
  prod5 <= conv_unsigned(unsigned(to_stdLogicVector(to_bitVector(a_sh3) sll log2(GL))), addr_width);
                                       -- prod5 = a[s + d - 1 : s] * 2^gl
  m5    <= conv_unsigned(unsigned(a(logD - 1 downto 0)) + prod5, logD);
                                       -- m5 = (a[d - 1 : 0] + a[s + d - 1 : s] * 2^gl)[d - 1 : 0]

  -- CASE VI.
  prod6       <= conv_unsigned(unsigned(to_stdLogicVector(to_bitVector(a(addr_width - 1 downto
logD)) sll log2(GL))), addr_width);
                                       -- prod6 = a[addr_width - 1 : d] * 2^gl
  masking6 : for i in 0 to addr_width - 1 generate
    a_msk6(i) <= prod6(i) when i < logS_nat else  -- a_msk6 = (a[addr_width - 1 : d] * 2^gl)[s - 1 :
0]
                 '0';
  end generate masking6;
  m6          <= conv_unsigned(unsigned(a(logD - 1 downto 0)) + unsigned(a_msk6), logD);
                                       -- m6 = (a[d - 1 : 0] + (a[addr_width - 1 : d] * 2^gl)[s - 1
: 0])[d - 1 : 0]

  -- Case multiplexing
  ModuleID_tmp <= std_logic_vector(m1) when mode = "000" else
                  std_logic_vector(m2) when mode = "001" else
                  std_logic_vector(m3) when mode = "010" else
```

```
                std_logic_vector(m4) when mode = "011" else
                std_logic_vector(m5) when mode = "100" else
                std_logic_vector(m6) when mode = "101" else
                (others => 'X');

  ModuleID <= ModuleID_tmp;

end Behavioral;
```

Shuffle

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.gsma_package.all;

entity Shuffle is
  generic (
    sub_bus_num     : natural := 2;
    log_sub_bus_num : natural := 1;
    sub_bus_width   : natural := 1);
  port (
    I_bus : in  std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0);  -- Input busses
    S_bus : in  std_logic_vector(sub_bus_num * log_sub_bus_num - 1 downto 0);  -- Select signal
busses
    O_bus : out std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0));  -- Shuffled busses

end Shuffle;

architecture Behavioral of Shuffle is

  type inner_subtype is array (sub_bus_num - 1 downto 0) of std_logic_vector(sub_bus_width - 1
downto 0);
  type inner_type is array (sub_bus_num - 1 downto 0) of inner_subtype;
  type inner_nat_type is array (sub_bus_num - 1 downto 0) of natural;

  signal O_bus_tmp : std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0);
  signal inner_sig : inner_type;
  signal sel : inner_nat_type;
```

```
begin  -- Behavioral


  -- Set of deMUXs
  SET_OF_deMUXs : for demux_id in 0 to sub_bus_num - 1 generate
    -- Select signal decoder
    sel(demux_id) <= conv_integer(unsigned(S_bus((demux_id + 1) * log_sub_bus_num - 1 downto
demux_id * log_sub_bus_num)));
    -- DeMUX
    DeMUX : for output_id in 0 to sub_bus_num - 1 generate
      inner_sig(demux_id)(output_id)(sub_bus_width - 1 downto 0) <=
        I_bus((demux_id + 1) * sub_bus_width - 1 downto demux_id * sub_bus_width) when output_id =
sel(demux_id) else
        low_vector(sub_bus_width - 1 downto 0); --(others => '0');
    end generate DeMUX;
  end generate SET_OF_deMUXs;

  -- Set of OR-gates
  OUTPUT_OR_GATES: process (inner_sig)
    variable O_bus_var : std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0);
  begin  -- process OUTPUT_OR_GATES
    for or_id in 0 to sub_bus_num - 1 loop
      -- OR-gate
      O_bus_var((or_id + 1) * sub_bus_width - 1 downto or_id * sub_bus_width) :=
        inner_sig(0)(or_id)(sub_bus_width - 1 downto 0);
      for input_id in 1 to sub_bus_num - 1 loop
        O_bus_var((or_id + 1) * sub_bus_width - 1 downto or_id * sub_bus_width) :=
          O_bus_var((or_id + 1) * sub_bus_width - 1 downto or_id * sub_bus_width) or
          inner_sig(input_id)(or_id)(sub_bus_width - 1 downto 0);
      end loop;  -- input_id
    end loop;  -- or_gate_id
    O_bus_tmp <= O_bus_var;
  end process OUTPUT_OR_GATES;
  O_bus <= O_bus_tmp;

end Behavioral;
```

De-shuffle

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use work.gsma_package.all;

entity De_Shuffle is
  generic (
    sub_bus_num     : natural := 2;
    log_sub_bus_num : natural := 1;
    sub_bus_width   : natural := 1);
  port (
    I_bus : in  std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0);  -- Input busses
    S_bus : in  std_logic_vector(sub_bus_num * log_sub_bus_num - 1 downto 0);  -- Select signal
busses
    O_bus : out std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0));  -- Shuffled busses

end De_Shuffle;

architecture Behavioral of De_Shuffle is

  signal O_bus_tmp : std_logic_vector(sub_bus_num * sub_bus_width - 1 downto 0);

begin  -- Behavioral

  MUXs           : process (I_bus, S_bus)
    variable sel : natural;             -- natural_vector(sub_bus_num - 1 downto 0);
  begin  -- process MUXes
    O_bus_tmp <= (others => '0');
    for mux_id in 0 to sub_bus_num - 1 loop  -- Block of MUXes
      sel := conv_integer(S_bus((mux_id + 1) * log_sub_bus_num - 1 downto mux_id *
log_sub_bus_num));  -- Select signal
      for input_id in 0 to sub_bus_num - 1 loop  -- MUX inputs
        if input_id = sel then
          O_bus_tmp((mux_id + 1) * sub_bus_width - 1 downto mux_id * sub_bus_width) <=
            I_bus((input_id + 1) * sub_bus_width - 1 downto input_id * sub_bus_width);
        end if;
      end loop;
```

```
    end loop;
  end process MUXs;

  O_bus <= O_bus_tmp;

end Behavioral;
```

BIBLIOGRAPHY

[1] *AMBA specification*, rev. 2.0 edition.

[2] *ARM PrimeCell Multi-Port Memory Controller (PL175). Technical reference manual.*

[3] Chameleon CS2000. www.cmln.com.

[4] International technology roadmap for semiconductors. itrs.net.

[5] Wikipedia. the free encyclopedia. www.wikipedia.org.

[6] XC6200: Advance product specification. Technical report, Xilinx, Inc., San Jose, CA, 1996.

[7] Wildfire reference manual. Technical report, Annapolis Microsystems, Inc, Annapolis, MD., 1998.

[8] CS2000 advance product specification. Technical report, Chameleon Systems, Inc., 2000.

[9] *ARM9E-S Core Technical Reference Manual*, revision: r2p1 edition, 2004.

[10] *Processor Design*. Springer Netherlands, 2007.

[11] Eero Aho, Jarno Vanne, and Timo D. Hämäläinen. Parallel memory architecture for arbitrary stride accesses. *IEEE Design and Diagnostics of Electronic Circuits and Systems*, pages 63–68, 2006.

[12] Eero Aho, Jarno Vanne, and Timo D. Hämäläinen. Parallel memory implementation for arbitrary stride accesses. In *IC-SAMOS'06: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 1–6, Jul. 2006.

[13] Eero Aho, Jarno Vanne, Kimmo Kuusilinna, and Timo D. Hämäläinen. Address computation in configurable parallel memory architecture. *IEICE Transactions on Information and Systems*, E87-D(7):1674–1681, Jul. 2004.

[14] Saman Amarasinghe and Bill Thies. Architectures, languages, and compilers for the streaming domain. In *PACT '03: The 12th International*

Conference on Parallel Architectures and Compilation Techniques, Sep.-Oct. 2003.

[15] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11-18, 1993.

[16] Bill Beane. Multi-port memories evolve to meet SoC demands. *EE Times*, Apr. 2003.

[17] Jürgen Becker. Dagstuhl-seminar "dynamically and partially reconfigurable architectures". *IT - Information Technology*, 46(4):218-225, 2004.

[18] Michele Borgatti, Francesco Lertora, Benoit Forêt, and Lorenzo Calí. A reconfigurable system featuring dynamically extensible embedded microprocessor, fpga and customizable i/o. In *IEEE Journal of Solid-State Circuits*, volume 38, pages 521-529, Mar. 2003.

[19] P. Budnik and D. J. Kuck. The organization and use of parallel memories. *IEEE Transactions on Computers*, 20(12):1566-1569, Dec. 1971.

[20] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The garp architecture and C compiler. *IEEE Computer*, 33(4):62-69, 2000.

[21] Fabio Campi, Antonio Deledda, Matteo Pizzotti, Luca Ciccarelli, Pierluigi Rolandi, Claudio Mucci, Andrea Lodi, Arseni Vitkovski, and Luca Vanzolini. A dynamically adaptive DSP for heterogeneous reconfigurable platforms. In *DATE '07: Proceedings of the International Conference on Design, Automation and Test in Europe*, pages 9-14, Apr. 2007.

[22] Fabio Campi, Mario Toma, Andrea Lodi, Andrea Cappelli, Roberto Canegallo, and Roberto Guerrieri. A VLIW processor with reconfigurable instruction set for embedded applications. In *ISSCC '03: Digest of Technical Papers of the IEEE International Conference on Solid-State Circuits*, volume 1, pages 250 – 491, 2003.

[23] Fabio Campi, P. Zoffoli, Claudio Mucci, Massimo Bocchi, Antonio Deledda, M. De Dominicis, and Arseni Vitkovski. A stream register file unit for reconfigurable processors. In *ISCAS '06: Proceedings of the 2006 IEEE International Symposium on Circuits and Systems*, May 2006.

[24] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and André DeHon. Stream computations organized for reconfigurable execution (score). In *FPL '00: Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, pages 605–614, London, UK, 2000. Springer-Verlag.

[25] Sek M. Chai, Nikolaos Bellas, Malcolm Dwyer, and Dan Linzmeier. Stream memory subsystem in reconfigurable platforms. In *WARFP '06: 2nd Workshop on Architecture Research using FPGA Platforms,*, Feb. 2006.

[26] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. Technical report, Knoxville, TN 37996, USA, 1995.

[27] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. The reconfigurable streaming vector processor. In *MICRO-36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 141–150, 2003.

[28] William J. Dally. Imagine: A high-performance image and signal processor.

[29] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, 15–21 Nov. 2003.

[30] III David T. Harper. Block, multistride vector, and FFT accesses in parallel memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):43–51, 1991.

[31] III David T. Harper and J. R. Jump. Vector access performance in parallel memories using a skewed storage scheme. *IEEE Transactions on Computers*, 36(12):1440–1449, 1987.

[32] III David T. Harper and Darel A. Linebarger. Conflict-free vector access using a dynamic storage scheme. *IEEE Transactions on Computers*, 40(3):276–283, 1991.

[33] Benoit Dupont de Dinechin. GCC for embedded VLIW processors: Why not? In *GREPS '07: International Workshop on GCC for Research in Embedded and Parallel Systems*. ST Microelectronics, Grenoble, France, Sep. 2007.

[34] A. Deledda, C. Mucci, A. Vitkovski, M. Kuehnle, F. Ries, M. Huebner, J. Becker, P. Bonnot, A. Grasset, P. Millet, M. Coppola, L. Pieralisi, R. Locatelli, G. Maruccia, F. Campi, and T. DeMarco. Design of a HW/SW communication infrastructure for a heterogeneous reconfigurable processor. In *DATE '08: Proceedings of the International Conference on Design, Automation and Test in Europe*, 2008.

[35] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. RaPiD - reconfigurable pipelined datapath. In *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 126–135, London, UK, 1996. Springer-Verlag.

[36] Jean Marc Frailong, William Jalby, and Jacques Lenfant. XOR-schemes: A flexible data organization in parallel memories. In *ICPP '85: International Conference on Parallel Processing*, pages 276–283, 1985.

[37] Guang R. Gao, Yue-Bong Wong, and Qi Ning. A timed petri-net model for fine-grain loop scheduling. In *CASCON '91: Proceedings of the 1991*

conference of the Centre for Advanced Studies on Collaborative research, pages 395–415. IBM Press, 1991.

[38] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented FPGA computing in the streams-C high level language. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 49–56, 17–19 Apr. 2000.

[39] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. PipeRench: A reconfigurable architecture and compiler. *IEEE Computer*, 33(4):70–77, 2000.

[40] Scott Hauck and Gaetano Borriello. Pin assignment for multi-FPGA systems. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, volume 16, pages 956–964, 1997.

[41] J. R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, page 12, Washington, DC, USA, 1997. IEEE Computer Society.

[42] Simon D. Haynes and Peter Y. K. Cheung. A reconfigurable multiplier array for video image processing tasks, suitable for embedding in an fpga structure. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 226–234, 15–17 Apr. 1998.

[43] Kimmo Kuusilinna, Jarno Tanskanen, Timo Hämäläinen, and Jarkko Niittylahti. Configurable parallel memory architecture for multimedia computers. *Journal of Systems Architecture*, 47(14–15):1089–1115, 2002.

[44] Georgi Kuzmanov, Georgi Gaydadjiev, and Stamatis Vassiliadis. Multimedia rectangularly addressable memory. *IEEE Transactions on Multimedia.*, 8:315–322, Apr. 2006.

[45] Edward A. Lee and Thomas M. Parks. Dataflow process networks. pages 773–799, May 1995.

[46] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays*, pages 135–143, New York, NY, USA, 1999. ACM.

[47] Montse Peiron Eduard Ayguadé Mateo Valero, Tomás Lang. Conflict-free access for streams in multimodule memories. *IEEE Transactions on Computers*, 44(5):634–646, 1995.

[48] Sek M.Chai, Silviu Chiricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, James M. Norris, Mike Schuette, and Abelardo López-Lagunas. Streaming processors for next-generation mobile imaging applications. *IEEE Communications Magazine*, 43(12):81–89, Dec. 2005.

[49] VLC media player. x264 - a free h264/avc encoder. http://developers.videolan.org/x264.html.

[50] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *FPT '02: Proceedings of the IEEE International Conference on Field-Programmable Technology*, pages 166–173, 16–18 Dec. 2002.

[51] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *FPL'03: Proceedings of the 13th International Conference on Field-Programmable Logic and Applications*, volume 2778, pages 61–70. Springer, 2003.

[52] Oskar Mencer, David J. Pearce, Lee W. Howes, and Wayne Luk. Design space exploration with a stream compiler. In *FPT '03: Proceedings of the IEEE International Conference on Field-Programmable Technology*, pages 270–277, 15–17 Dec. 2003.

[53] Miguel Miranda. System-level design methodologies for memory organization and communication in technology-aware design. Scientific report, IMEC, 2005.

[54] Takashi Miyamori and Kunle Olukotun. A quantitative analysis of reconfigurable coprocessors for multimedia applications. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 2, Washington, DC, USA, 1998. IEEE Computer Society.

[55] Claudio Mucci, Carlo Chiesa, Andrea Lodi, Mario Toma, and Fabio Campi. A c-based algorithm development flow for a reconfigurable processor architecture. In C. Chiesa, editor, *SoC '03: Proceedings of the IEEE International Symposium on System-on-Chip*, pages 69–73, 2003.

[56] Claudio Mucci, Luca Vanzolini, Andrea Lodi, Antonio Deledda, Roberto Guerrieri, Fabio Campi, and Mario Toma. Implementation of AES/Rijndael on a dynamically reconfigurable architecture. In L. Vanzolini, editor, *DATE '07: Proceedings of the International Conference on Design, Automation and Test in Europe*, pages 1–6, 2007.

[57] NIST. Aes. FIPS PUBS 197.

[58] Coert Olmsted. Scientific SAR user's guide. Technical Report asf-sd-003, Alaska Satellite Facility (ASF), July 1993.

[59] Jong Won Park. An efficient buffer memory system for subarray access. *IEEE Transactions on Parallel and Distributed Systems*, 12(3):316–335, 2001.

[60] Pierre G. Paulin and Miguel Santana. FlexWare: A retargetable embedded-software development environment. *IEEE Design & Test of Computers*, 19(4):59–69, 2002.

[61] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *MICRO 27: Proceedings of the 27th annual International Symposium on Microarchitecture*, pages 172–180, New York, NY, USA, 1994. ACM.

[62] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *ISCA '00: Proceedings of the 27th International Symposium on Computer Architecture*, pages 128–138, 2000.

[63] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold, and M. Gokhale. The NAPA adaptive processing architecture. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 28, Washington, DC, USA, 1998. IEEE Computer Society.

[64] Amar Shan. Heterogeneous processing: a strategy for augmenting moore's law. *Linux Journal*, Jan. 2006.

[65] H.D. Shapiro. Theoretical limitations on the efficient use of parallel memories. *IEEE Transactions on Computers*, 27(5):421–428, May 1978.

[66] Hartej Singh, Ming-Hau Lee, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, and Eliseu M. Chaves Filho. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.

[67] Gerard J.M. Smit, Paul M. Heysters, Michèl Rosien, and Bert Molenkamp. Lessons learned from designing the MONTIUM - a coarse-grained reconfigurable processing tile. In *SoC '04: Proceedings of the IEEE International Symposium on System-on-Chip*, pages 29–32, 16–18 Nov. 2004.

[68] IEEE standard. Air interface for fixed broadband wireless access systems. IEEE 802.16-2004.

[69] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette. System design using Khan process networks: the Compaan/Laura approach. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, pages 340–345Vol.1, 16–20 Feb. 2004.

[70] STMicroelectronics. Stmicroelectronics unveils innovative network-on-chip technology for new system-on-chip interconnect paradigm. www.st.com, December 15 2005.

[71] STMicroelectronics. Building a new system-on-chip paradigm. *Ferret, www.ferret.com.au*, January 12 2007.

[72] N. Stollon and B. Sihlbom. BAZIL: A multi-core architecture for flexible broadband processing. In *Proceedings of the Embedded Processor Conference*, 2001.

[73] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. *ACM SIGARCH Computer Architecture News*, 32(2):2, March 2004.

[74] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. The MOLEN polymorphic processor. *IEEE Transactions on Computers*, 53(11):1363–1375, 2004.

[75] Arseni Vitkovski, Georgi Kuzmanov, and Georgi Gaydadjiev. Two-dimensional memory implementation with multiple data patterns. In *ProRISC: Anual workshop on Circuits, Systems and Signal Processing*, 2007.

[76] Arseni Vitkovski, Georgi Kuzmanov, and Georgi Gaydadjiev. Memory organization with multi-pattern parallel accesses. In *DATE'08: Proceedings of the Internation Conference of Design, Automation & Test in Europe*, 2008.

[77] Martin Vorbach and Jürgen Becker. Reconfigurable processor architectures for mobile phones. In *IPDPS'03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 6pp., 22-26 April 2003.

[78] Jean E. Vuillemin, Patrice Bertin, Didier Roncin, Mark Shand, Hervé H. Touati, and Philippe Boucard. Programmable active memories: reconfigurable systems come of age. *IEEE Transactions on Very Large Scale Integration Systems*, 4(1):56–69, 1996.

[79] Hua Wang, Miguel Miranda, Wim Dehaene, Francky Catthoor, and Karen Maex. Systematic analysis of energy and delay impact of very deep submicron process variability effects in embedded sram modules. In *DATE '05: Proceedings of the International Conference on Design, Automation and Test in Europe*, pages 914–919, 2005.

[80] Ralph D. Wittig and Paul Chow. OneChip: An FPGA processor with reconfigurable logic. In Kenneth L. Pocek and Jeffrey Arnold, editors, *FCCM '96: IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[81] Hui Zhang, Vandana Prabhu, Varghese George, Marlene Wan, Martin Benes, Arthur Abnous, and Jan M. Rabaey. A 1V heterogeneous reconfigurable processor IC for baseband wireless applications. In *ISSCC '00: Digest of Technical Papers of the IEEE International Solid-State Circuits Conference*, pages 68–69,448, 7–9 Feb. 2000.