Università degli Studi di Bologna

FACOLTÀ DI INGEGNERIA

Dottorato di Ricerca in Ingegneria Elettronica,
Informatica e delle Telecomunicazioni

XX Ciclo

ING-INF/01

Exploration of Communication Strategies for Computation Intensive Systems-On-Chip

Tesi di Dottorato di

Antonio Deledda

Relatore

Chiar. mo Prof. **Roberto Guerrieri**

Coordinatore

Chiar. mo Prof. **Paolo Bassi**

Anno Accademico 2006-2007

Keywords:

Reconfigurable architectures

Heterogeneous MPSoC

MORPHEUS

Network On Chip

Digital Signal Processing

Contents

List of Figures

List of Tables

Introduction

Data intensive processing in embedded systems is receiving relevant attention, due to rapid advancements in multimedia computing and high-speed telecommunications. Applications demand high performance under realtime requirements, and computation power appetite soars faster than Moore's law (see Figure 1, [13]). Processor efficiency is impaired by the memory bandwidth problem of traditional Von Neumann architectures. On the other hand, the conventional way to boost performance through Application Specific Integrated Circuits (ASIC) suffers from sky-rocketing manufacturing costs and long design development cycles. This results in an increasing need of post-fabrication programmability at both software and hardware level. Field Programmable Gate Arrays (FPGA) bring maximum flexibility with their fine grain architecture, but imply severe overheads in timing, area and consumption. Word or sub-word oriented Run-time Reconfigurable Architectures (RAs) [1] offer highly parallel, scalable solutions combining hardware performance with software flexibility. Their coarser granularity reduces area, delay, power consumption and reconfiguration time, but introduces tradeoffs in the design of the processing elements, that need to be tailored for a given application domain.

A possible way to mitigate this aspect for building a flexible yet efficient signal processor is to substitute each ASIC accelerator with a specific domain oriented RAs, inducing a graceful shift of SoCs from application specific circuits to domain oriented platforms, where different flavors of reconfigurable hardware, each more suited to a given application environment, are merged with ASIC and general purpose processors to provide ideal tradeoff between performance and post-fabrication programmability. The immediate advantage is that the higher computational density of RAs allows to build networks composed of a significantly smaller number of nodes. The immediate drawback is the need to synchronize units that are intrinsically different and

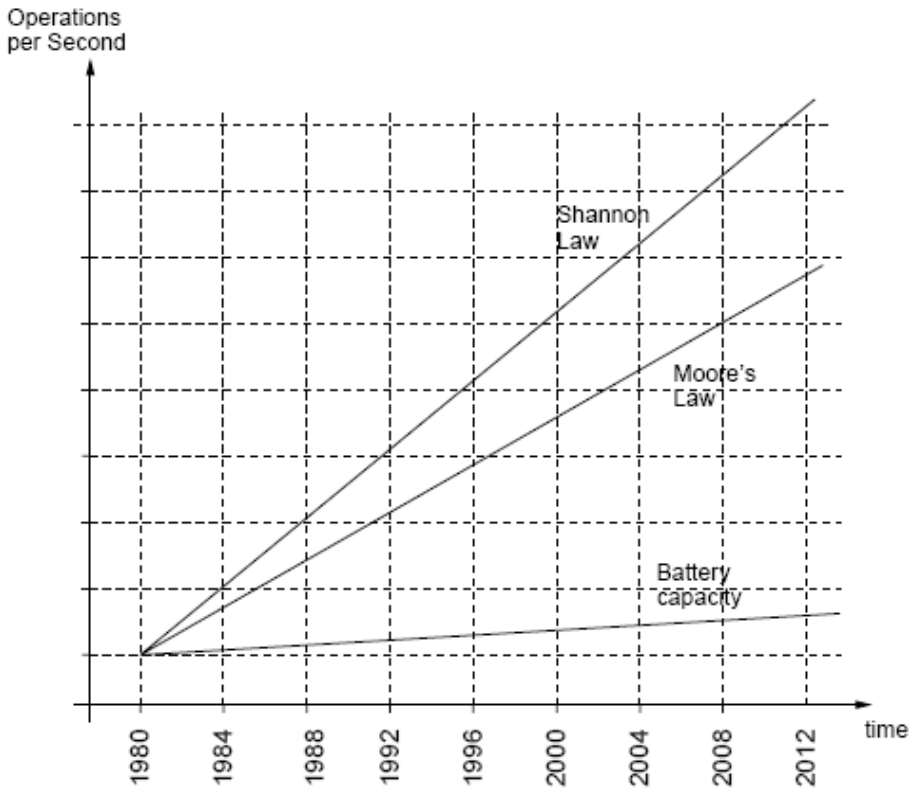

Figure 1: Computational requirements vs. Moore's law and battery storage

provide independent application mapping styles and entry languages. In this context, critical issues are related to the definition of

- a toolset that must be capable to hide RA heterogeneity and hardware details providing a consistent and homogeneous programming model to the end user
- a data interconnect infrastructure, that must sustain the bandwidth requirements of the computation units while retaining a sufficient level of programmability to be adapted to all the different data flows defined over the architecture in its lifetime.

These aspects are strictly correlated and their combination, together with the strategy deployed for RA computation synchronization represents the signal processor interface toward the end-user. In particular, the architecture view shall be abstracted as much as possible for the user, providing a programming

model looking like purely functional code. Program parts requiring acceleration on RAs should be identifiable in the easiest possible way. A toolset can then handle and program the code corresponding to data movements and reconfigurations related to these accelerating parts.

In this context not only computation but also communication aspects must indeed be considered. This will enable performance optimization by masking communication time by computation time through a "pipelined" behavior. The scheduling of these accelerating parts among each other, including loading configuration and execution, may be managed at compilation time based on RTOS-oriented services.

This thesis presents the definition and the design of a heterogeneous reconfigurable SoC platform, where state-of-the-art RAs of different size and nature are grouped together in a processor-controlled system. In particular, this work aims at describing the most significant challenges and design choices that have been faced in the deployment of a well known NoC infrastructure (the ST Spidergon NoC approach [2]) and the consequent impact on the architecture and toolset definition. I believe that the most relevant innovation aspects of this work are:

1. A significant milestone in the field of Heterogeneous-MultiCore SoCs;
2. The first design-case challenging the deployment of the NoC concept to a network of high-bandwidth computation intensive RAs.

The rest of this thesis is organized as follow. In Chapter 1, the terminology and basic foundation of Computation Intensive System-on-Chips are revisited to pave way to the rest of the thesis. An overview of the European project, where this work was done, is also given. Chapter 2 presents the defined architecture for the MORPHEUS project. Chapter 3 and 4 details the main choices given in the definition of the memory hierarchy and the adopted communication

infrastructure. The last chapter presents design and performance results achieved in the frame of this project.

Chapter 1 Multiprocessing and Reconfigurable Computing

Up to the 1990s processor designers mainly focused their work on boosting single processor performance. This evolution was conducted constantly increasing clock rates extending instruction-level parallelism (ILP). This was made possible by technology scaling which reduced physical delays and device sizes allowing for a larger area to be utilized by new logic. The result is a variety of superscalar architectures employing different hardware solutions to concurrently process different instructions. Performance of future embedded systems, according to the ITRS estimation shown in figure 1.1, will require the execution of an increasing number of instructions per clock cycle, but the cost of extracting such parallelism from a single thread is becoming prohibitive both in terms of area and energy consumption.

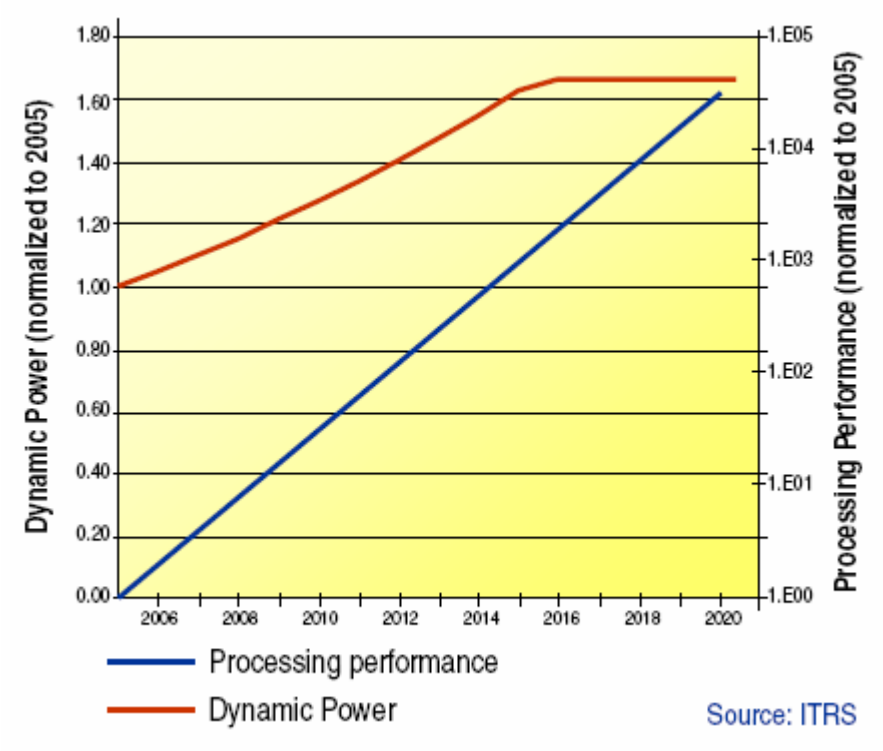

Figure 2: MPSoC trends

It has nevertheless been stated that trying to further increase ILP is not the best choice as explained by D. Patterson in [17]. As an alternative to superscalar architectures, processor designers and researchers are proposing a different approaches based on Tread Level Parallelism (also named Task Level Parallelism, TLP) that seem to enable significant speed-up and proves more flexible than ILP.

From an architecture point of view, we can distinguish MPSoC architectures in two main classes:

1. Homogeneous MPSoC, where all the processing elements are usually identical or at least share a common ISA,
2. Heterogeneous MPSoC, characterized by the integration of different computational core: i.e. processor with different ISAs, several ASICs or DSPs, etc.

One of the main aspects in heterogeneous MPSoC is that software modules have to interrelate with hardware modules. In [18] the authors show the use of an high level programming approach for the abstraction of HW-SW interfaces. The proposed programming model is based on a set of functions (primitives) that can be used by the SW engineer to interact with HW modules. In the reconfigurable computing domain, alternative approaches have also been investigated; in [19] a scalable programming model (named SCORE) is presented and used for a homogeneous scalable reconfigurable architecture. The model allows indifferently computing a set of tasks in time or in space, following the resources available: the advantage is that software is reusable for any generation of component based on that model.

From the memory point o view we can investigate two main programming models:

1. SMP (Symmetric Multi Processing) where all the processors have a global vision of the memory (shared memory) and

2. AMP (Asymmetric Multi Processing) where the processors are loosely coupled and have generally dedicated local memory resources.

Task control management is also an issue to consider. Threads can be handled at execution time (dynamically on a single processor) by an operating system or at design time (statically) by complex scheduling techniques. A part of the MP-SOC community focuses on static task placement and scheduling in MP-SOC. Indeed, having a complex operating system in memory taking care of run-time mapping is often not feasible for a SOC, because of the restricted memory resources and associated performance overhead. Moreover, these systems are often heterogeneous and dedicated to a few tasks, and a single but efficient scheduling of tasks may be more adapted. For instance in [20], the authors summarize the existing techniques (ILP based or heuristics) and have developed a new framework based on ILP solvers and constraint programming to solve at design time the task allocation/scheduling problem.

1.1 MPSoC state of the art

Multiprocessor systems-on-chip are mostly suitable for high-volume products with stringent constraints in terms of performance, power consumption and cost. Many application domains are covered by these features, including multimedia, communications, automotive and networking. This section, referring to a survey proposed by Wolf [16], describes some state-of-the-art MPSoCs, suitable for different application domains. A common feature of all the analyzed architectures resides in the integration of a standard processor which operates as the main system controller. This choice generally simplifies system programmability allowing multiple processing elements and customized ASIC blocks to be programmed as co-processors, maintaining a central control task on the control processor.

1.1.1 TI OMAP™

The Open Multimedia Application Platform (OMAP) [21][22] proposed by Texas Instruments is a combined RISC/DSP architecture targeted to 3G wireless applications. The platform supports mainly baseband processing and voice services, in addition multimedia, gaming and other application at user level.

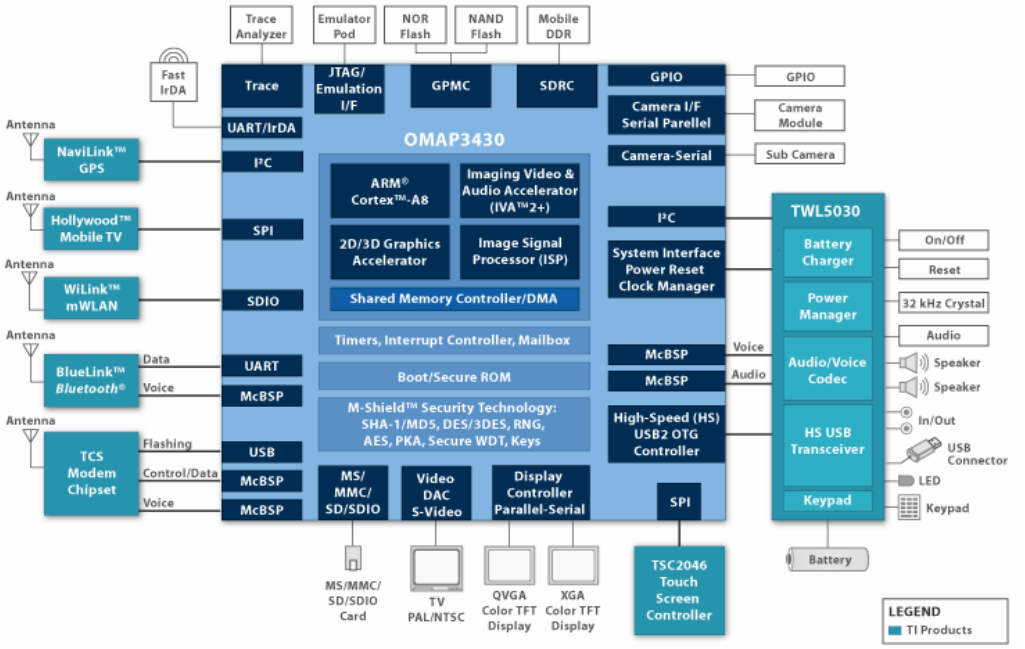

Figure 3: TI OMAP 3430 block diagram

Figure 3 shows the architecture overview of the TI OMAP 3430. The inclusion of a standard ARM Cortex A8 processor ensures the compatibility with different commercial operating systems, while the additional DSPs provide the platform with additional computational power to process the previously cited applications. The ARM core and the DSP use a shared external DRAM interface, while a consistent amount of SRAM are internally integrated. A complete set of peripherals is also included (USB, I2C, UARTs, GPIOs). Outstanding gaming capabilities will also be possible, thanks to ARM's integrated vector floating-point acceleration working with a dedicated 2D/3D graphics hardware accelerator.

To simplify software development on the heterogeneous multi-core architecture, the RISC is defined as the system master and a DSP resource manager runs on the ARM. Tasks executed on the DSP are controlled through a DSP/BIOS™ bridge which adds support for inter-processor communication, based upon the mailbox mechanism. The DSP/BIOS allows the ARM to initiate DSP tasks, to exchange messages and data streams with the DSP and to control the DSP status. This hardware support simplifies system programmability treating the DSP and the accelerators as system co-processors.

1.1.2 ST Nomadik™

STMicroelectronics Nomadik platform [23] is designed for 2.5G/3G mobile phones, personal digital assistants (PDAs) and, more in general, portable wireless products with multimedia capabilities. The architecture is focused at delivering ultra low power consumption enabling audio and video applications. The result is a 20mW typical power consumption with the computational power required by MPEG-4 encoding and decoding with display sizes ranging from 160x160 pixels to 640x480 pixels.

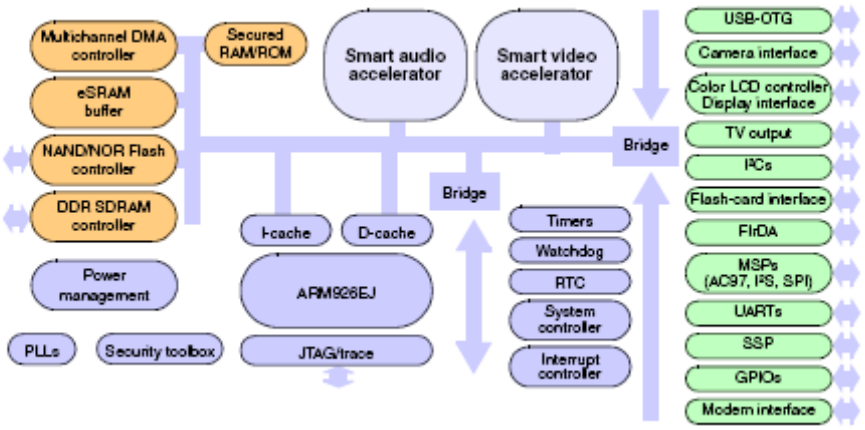

Figure 4: ST Nomadik multimedia processor architecture

Again, the architecture is based on a standard ARM926E-JS processor. This 32-bit processor core supports 32-bit ARM and 16-bit Thumb instruction sets,

enabling the user to trade off between high performance and high code density. The cached ARM CPU features a memory management unit (MMU) and is clocked at a frequency up to 350 MHz. It has a 16-Kbyte instruction cache, a 16-Kbyte data cache, and a 128-Kbyte level 2 cache, and supports the Jazelle™ extensions for Java acceleration.

In addition to the ARM core a series of accelerators are included for dedicated task:

- smart video accelerator for SDTV video encoding and decoding, with MIPI and SMIA camera interfaces.
- smart audio accelerator containing a comprehensive set of digital audio decoders and encoders, and offering a large number of 3-D surround effects.
- A smart imaging accelerator, providing real-time, programmable image reconstruction engine.
- A smart graphics accelerator

A multi-layer AMBA crossbar interconnect for optimized data transfers between the CPU, accelerators, memory devices and peripherals is also integrated. A set of hardware semaphores for flexible inter-process management is used. A wide range of peripheral interfaces (GPIO, USB-OTG high speed, UART, I²C, FIrDA, SD/high-speed MMC/Memory Stick Pro, fast serial ports, TV output, color LCD and camera interfaces, scroll-key encoder, key-pad scanner) is available.

1.1.3 Philips Nexperia™

The Nexperia NXP1500 [24] is a MPSoC architecture developed by Philips Semiconductors and designed for digital video applications, including digital television, home gateway and networking, and set-top box applications.

At the heart of all PNX1500 processors is a TriMedia TM3260 CPU core delivering top performance through the implementation of a very-long instruction word (VLIW) architecture. Five issue slots enable up to five simultaneous RISC-like operations to be scheduled into only one VLIW instruction. These operations can simultaneously run on five of the CPU's 31 pipelined functional units within one clock cycle.

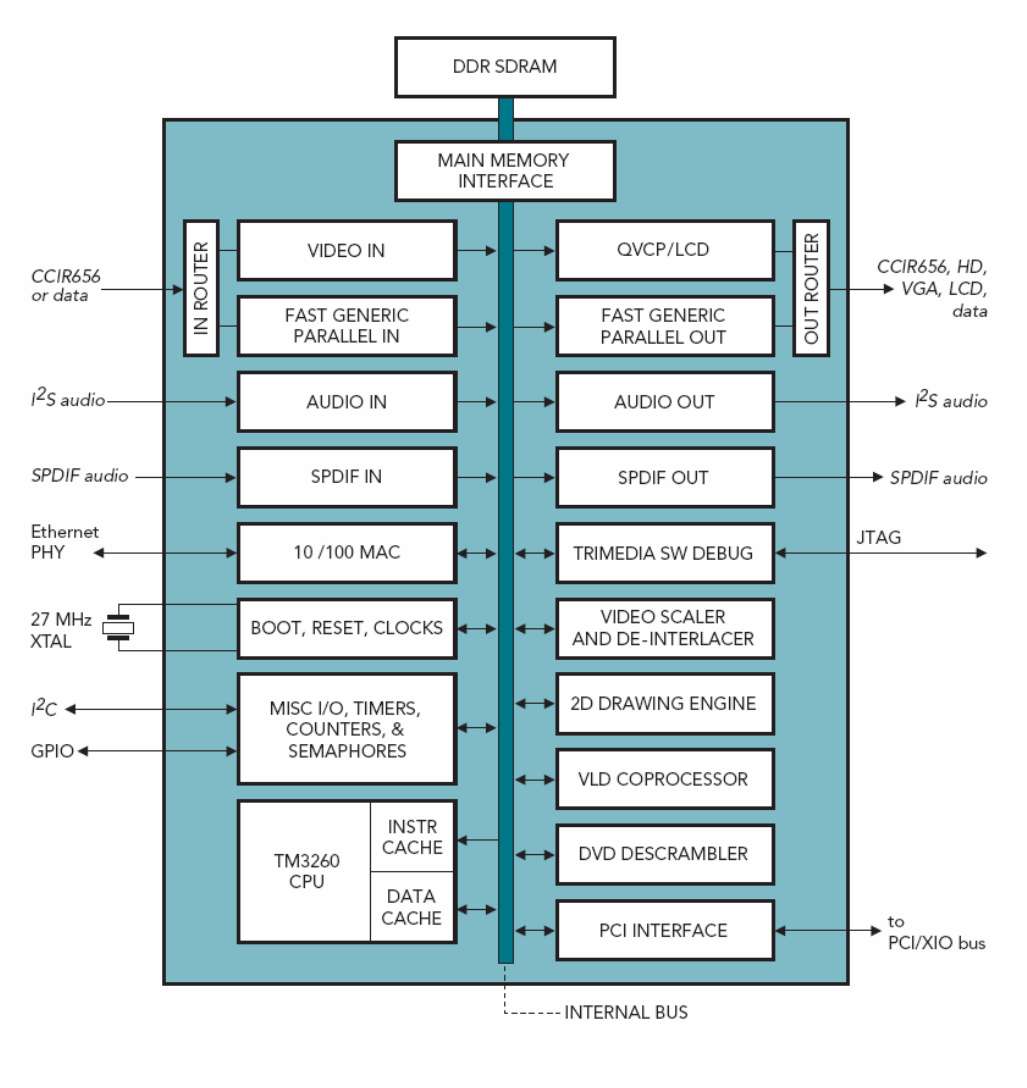

Figure 5: Philips Nexperia PNX1500 block diagram

In addition to the Trimedia core a set of specific accelerators are integrated. A video input processor (VIP) captures and processes digital video for use by on-chip units. During capture of a continuous stream of data, the VIP unit can

crop, horizontally down-scale, or convert the YUV video to one of many standard pixel formats as needed before writing data to memory. An additional versatile, programmable memory-based scaler unit applies a wide variety of image size, color, and format manipulations to improve video quality and prepare it for display.

A set of Audio input/output programmable device are also integrated. Audio units provide all signals needed to read and write digital audio datastreams from/to most high-quality, low-cost serial audio oversampling A/D and D/A converters and codecs. The AI unit supports capture of up to eight channels of stereo audio. The AO unit outputs up to eight channels and directly drives up to four external, stereo I^2S or similar D/A converters or highly integrated PC codecs. Additional On-chip hardware accelerators are targeted to 2D and 3D graphics processing, MPEG decoding, image scaling and filtering, and display channel composition. All coprocessors read input and write results to memory.

A PNX1500's CPU and processing units access external memory through an internal bus system comprising separate 64-bit data and 32-bit address buses. Arbitrated by the MMI unit (Main Memory Interface), the internal buses maintain real-time responsiveness in a variety of applications. The system also includes an external DRAM interface, a DMA for each processor and several I/O interfaces.

1.2 Reconfigurable computing

The first idea of a reconfigurable computing machine was conceived by Gerald Estrin in the early 1960s [30] when he presented the "fixed plus variable structure computer" [31]. It would consist of a standard processor, augmented by an array of reconfigurable hardware blocks controlled by the main processor. The reconfigurable hardware could be programmed to perform a specific task with performance comparable to a dedicated hardware block. Once the task was performed, the reconfigurable unit could be set up again to

perform a new different task. This first example of hybrid computer, combining the flexibility of a software programmable processor core with the performance of dedicated hardware, failed to become an interesting solution for commercial products. For many years, in fact, microprocessors combined with Application Specific Integrated Circuits (ASICs) have represented a state of the art solution able to meet application requirements when stand alone processor computational power was not adequate.

The combination of higher silicon integration degree and the need of flexibility imposed by the continuous algorithmic innovation, has generated a tremendous attention on the reconfigurable computing. The term *Reconfigurable Computing* (RC) is broadly intended as the capability to couple software based programmability with dynamic hardware programmability. The most common devices utilized as reconfigurable units are the Field Programmable Gate Arrays (FPGAs), but current scenario of reconfigurable devices is being crowded by a variety of reconfigurable architectures, with different reconfiguration granularities (coarse/fine/mixed grain fabrics), VLIW processing, systolic arrays, processor networks and so on. RC has long been considered [27][28][29][32][33][34][35][36] a feasible alternative to tackle the requirements described before. As shown in [28][29][36][37][38], reconfigurable architectures are classified depending on their *grain*, intended as the bitwidth of their interconnect structure and the complexity of their reconfigurable processing elements (PEs). Field Programmable Gate Arrays (FPGAs) are typically regular architectures where PEs are based on lookup tables (LUTs) and merged in a bit-oriented interconnect infrastructure. Featuring small LUT cells and 1-bit interconnect FPGAs are typically described as *fine-grained.* Their very symmetrical and distributed nature makes FPGAs very flexible and general purpose, and they can be used to tackle both computation-intensive and control-oriented tasks, to the point that large commercial FPGAs are often used to build complete Systems-On-Programmable-Chip (SoPCs) [39]. Arithmetic-oriented datapaths feature regular structures, so when targeting computation intensive applications it is

possible to achieve higher efficiency designing PEs composed of hardwired operators such as ALUs, multipliers or multiplexers. Such kind of archietcture are usually defined *coarse-grained*. These devices trade part of the flexibility of FPGAs in order to provide higher performance for specific computations. There exist also a set of devices that fall in between the above two classifications, featuring bit widths of 2 or 4 bits, and small computational blocks that are either large LUTS or small arithmetic blocks as 4-bits ALUs. These can be classified as *medium-grained*.

All this broad category of digital architectures fall under the cumulative name of *"Reconfigurable Architectures" (RAs)*, underlining their capability to reconfigure at execution time part of their hardware structure to support more efficiently the running application. In this broad domain, we use the definition *"Reconfigurable Instruction Set Processor (RISP)"* for those reconfigurable architectures that are tightly integrated in order to compute as a single adaptive processing unit according to the Athanas/Silverman paradigm [25] regardless of their hybrid nature. The next two sections will describe the evolution of the reconfigurable processor concept in the last 10 years, through the description of several significant contributions in the field by both industry and academia.

1.2.1 Run-time reconfigurable instruction set processors

The first significant attempt at deploying instruction set metamorphosis to embedded systems taking advantage of run-time configurable hardware is P-RISC (PRogrammable Instruction Set Computer), proposed by Razdan/Smith in 1994 [26]. The architecture is depicted in Figure 9.3.

The PRISC micro-architecture is composed of a fixed RISC (a MIPS core) extended by instructions mapped on a standard FPGA embedded in the core and defined as a PFU (Programmable Function Unit). An efficient interface between the core and the PFU aims to fit the PFU into the core pipeline.

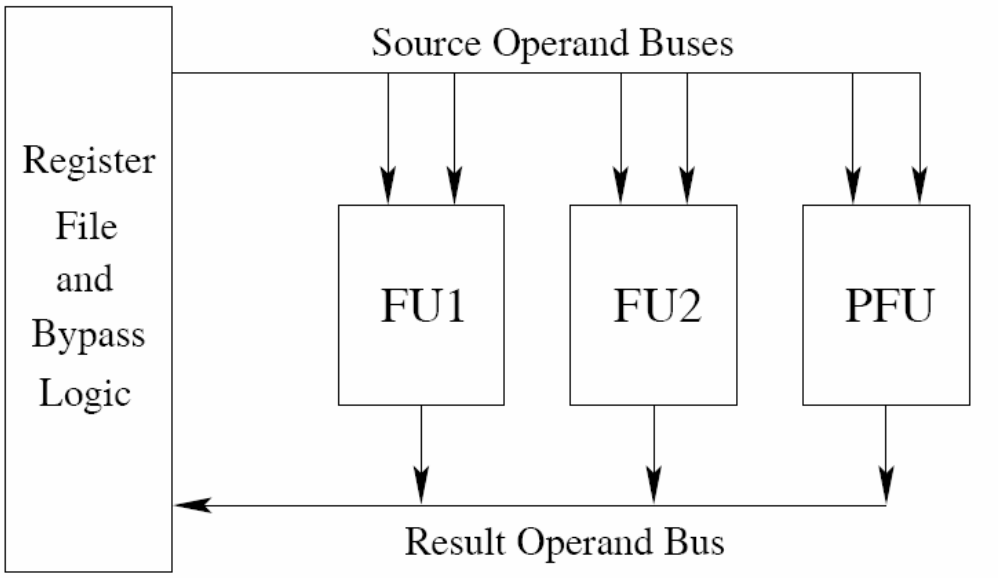

Figure 6: P-RISC Architecture

A compilation model for the ISA extension, starting from C specification is also proposed. To ease the physical interface between core and extension and to define a clear programming pattern for the compiler-based extraction of "interesting" extensions each PFU can handle 2-inputs 1-output functions. The most interesting concept of this architecture is that the PFU is considered as a function unit of the datapath, similar to an additional ALU, and PFU operands are read and written through the core register file providing a tightly coupled model. This is very friendly from the compiler and programmer point of view.

A significant novel step is represented by the "GARP" processor, which is shown in Figure 9.4. GARP [40] couples a MIPS core with a custom designed reconfigurable unit, connected as a coprocessor and addressed with explicit *Move* instructions while specific assembly instructions are implemented to trigger the configuration and computation on the reconfigurable unit. Unlike in the architecture P-RISC, the granularity of tasks mapped on the unit is quite coarse, to fully exploit the potentiality of the space-based computation approach. Another main difference respect the previous approach is that the coprocessor features direct access to memory allowing a larger data bandwidth

to the extension unit than that allowed by the core register file, although it raises relevant issues regarding memory access coherency.

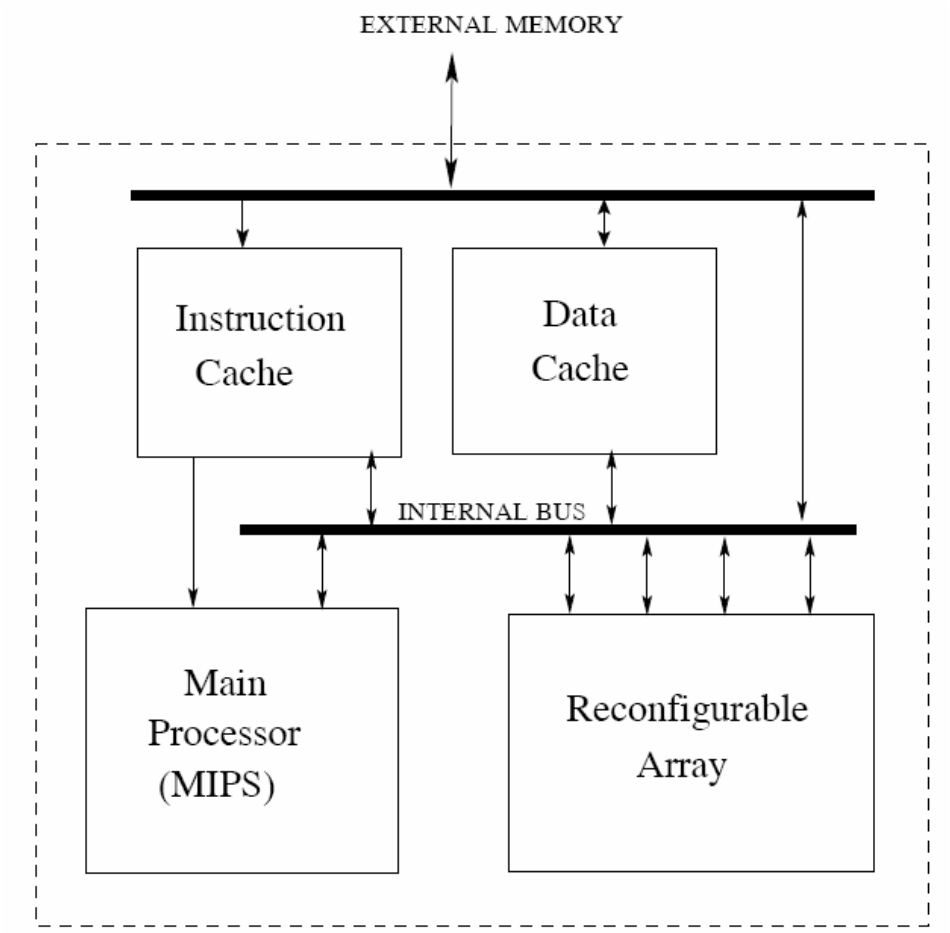

Figure 7 :GARP Architecture

The GARP reconfigurable unit is composed of an array of 24 rows of 32 LUT-based logic elements. Fast carry chains are implemented row-wide to provide efficient 32-bit arithmetical/logical operations on a single row. Each row can be approximated to a 32-bit ALU and an embedded hardware *sequencer* is added to the unit, in order to activate operators (one for each row) with appropriate timing to build a customized pipeline. Candidate kernels are described at C-level and decomposed in Data-Flow-Graphs (DFG), determining elementary operators and their data dependencies, and then mapped over the existing LUT resources. The sequencer embedded in the configurable hardware allows for an *imperative* computing pattern that matches

very well with C language and the GARP C compiler, thus easing a lot the gap between software and hardware programming indicated by Athanas/Silverman as the key issue in the deployment of instruction set adaptivity.

The MOLEN polymorphic processor [41] can be considered another important example of the reconfigurable instruction set processor paradigm. It has been implemented on a Xilinx Virtex-II FPGA coupling the on-chip PowerPC microprocessor with the Xilinx reconfigurable fabric, but the approach is quite independent from the device used to prove its feasibility. The significant contribution of MOLEN does not reside in its physical implementation, but in the theoretical approach to HW/SW co-processing and micro-architecture definition. The MOLEN contribution can be described as

- A microcode-based approach to the reconfigurable processor microarchitecture
- A novel processor organization and programming paradigm
- A compiler methodology for code optimization

A significant difference with all previously described architectures is that MOLEN does not attempt to propose a mean for "hardware/software co-compilation". Tasks to be mapped on the programmable hardware unit are considered as atomic tasks, primitive operations *Microcoded* in the processor architecture. Instruction set extensions are defined separately as libraries with an orthogonal HDL-based flow. Designing microcode for the adaptive extensions (*_μ-code*) consists in HDL design, synthesis and place & route of the extension functionality over third-party tools without any assistance from the reconfigurable processor compilation environment. This could raise issues for algorithmic developers not proficient with hardware design. On the other hand, this choice allows a large degree of freedom in the implementation allowing to extend the basic concepts to any technology. Also, as described in [41], the microcoded approach allows MOLEN-based RISPs to achieve speedups that are almost 100% of the theoretically achievable speedup

according to Amdahl's law, much higher than speedups achieved by hybrid compilation.

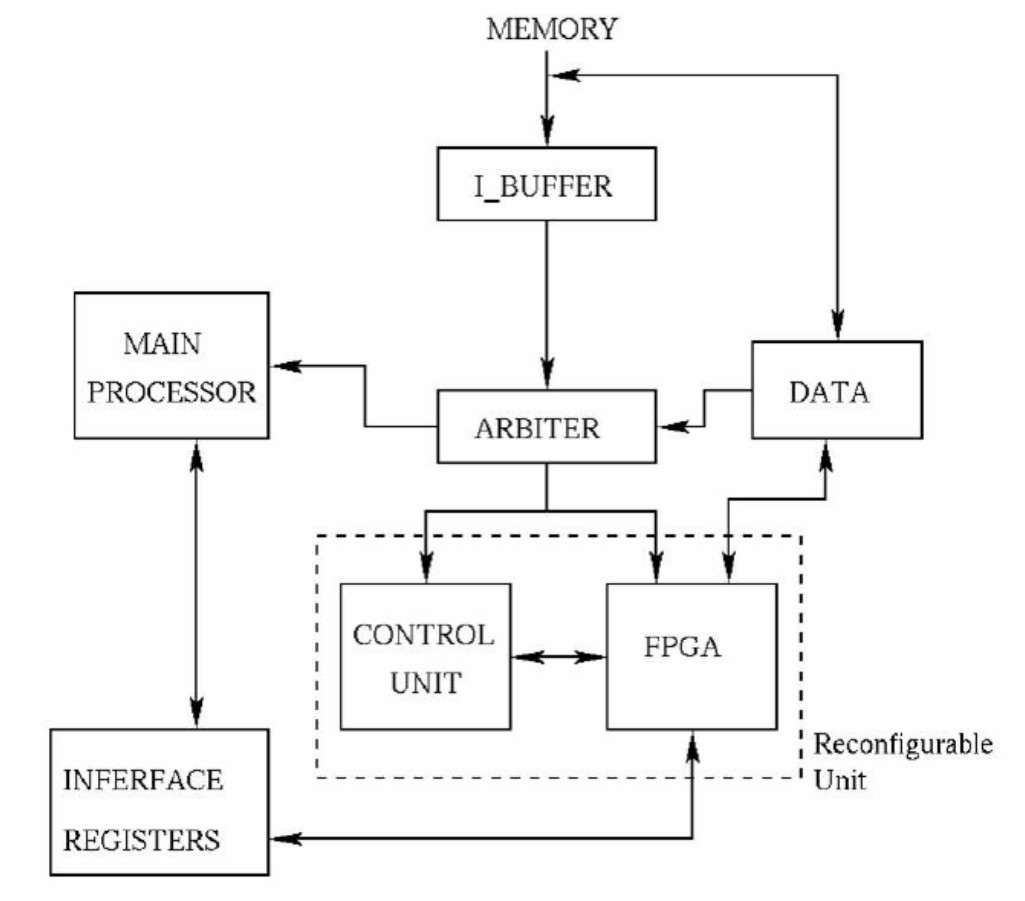

Figure 8: MOLEN Architecture

The MOLEN micro-architecture, shown in Figure 9.5., is organized as follows: instructions are decoded by an arbiter determining which unit is targeted. "Standard" instructions are computed by the *Core Processor* while instructions targeting the reconfigurable hardware are computed on the *Reconfigurable Processor* which is composed of a computational unit called *Custom Configured Unit* and a reconfigurable microcode control unit. The control unit allows partial reconfiguration. Exchange of data between the reconfigurable unit and the main processor is performed via specific exchange registers (XREGs).

As shown in the GARP design, in order to guarantee enough data bandwidth direct access from the extension segments to data memory is allowed, although there is no specific handling for multiple access consistency with respect to the core processor.

Although P-RISC, GARP and MOLEN represent from a theoretical standpoint the milestones that have brought to the formalization of the Reconfigurable Instruction Set Processor (RISP) concept, from the physical implementation side, an interesting attempt is described in [42][43] where the XiRisc architecture is introduced. XiRisc can be considered the first silicon implementation of a custom designed embedded reconfigurable instruction set processor. The design was performed at circuit level both for what concerns the core and the reconfigurable unit. XiRisc couples a Very Long instruction Word (VLIW) core, based on a five-stage pipeline, with an additional pipelined run-time configurable datapath (defined PiCoGA, see 2.2.2) acting as adaptive repository of application-specific functional units. While the VLIW core determines two symmetrical separate execution flows, the reconfigurable engine dynamically implements a third concurrent flow, extending the processor instruction set with *multi-cycle* pipelined functionalities of variable latency, according to the instruction set metamorphosis pattern.

Similar to P-RISC, extension segments are tightly integrated in the processor core receiving inputs and writing back results from/to the register file and direct access to memory is not allowed. In order to provide sufficient data bandwidth to the extension segments, PiCoGA features four source and two destination registers for each issued computation. Moreover, it can hold an internal state across several computations, thus reducing the pressure on connection to the register file.

1.2.2 Coarse grained reconfigurable processors

FPGAs have historically been used as programmable computing platforms, in order to provide high performance solutions to challenge NRE costs and time-to-market issues in the implementation of computationally intensive tasks. Even if in a first attempt, FPGA fabrics have been the immediate choice for implementation of reconfigurable hardware extensions, quite soon, it appeared evident that reconfigurable processors required computational features different from standard FPGAs. In several case application-specific logic such as hardwired multipliers has been utilize to achieve the necessary performance. When extension segments are very arithmetic-oriented, and bit-level computation is not necessary, the traditional LUT-based approach of standard FPGA can, as an extremes approach, be removed. Having in mind these concepts we can define Reconfigurable Processors (RPs) based on coarse grained hardwired operators rather than on fine grained LUTs as *coarse grained reconfigurable processors.* A significant benefit of this approach is the reduction of complexity in the place and route step, as well as the massive reduction of configuration memory and configuration time. The obvious drawback is that algorithm mapping is necessarily non-standard, and architecture-specific.

Shifting towards coarser grained Reconfigurable Architectures the definition of the PE internal structure becomes the most critical step in the design of the RP. It is obvious that the design of the PE has to be driven by an application domain in order to define the best trade-off between hardware complexity/features and application requirements. As a consequence, performance in that application field will be very impressive, but RPs will not scale well to different application environments. Coarse grained RPs will then be no more general purpose, as it was the case for the fine-grained architectures described earlier, but rather domain oriented.

PipeRench [44] is one of the first and more original run-time reconfigurable datapaths appearing in literature. It is composed by a set of configurable

blocks also called “stripes”. Each stripe maps a pipeline stage of the required computation, and is composed by an interconnect network and a set of PEs. In turn each PE contains one arithmetic logic unit and a pass register file that is used to implement the pipeline. ALUs are composed of lookup tables (LUTs) plus specific circuitry for carry chains while multipliers are built out of multiple adder instances. Each stripe can perform a different functionality per each cycle, thus providing an efficient time-multiplexing in the usage of each resource. The granularity of the computation fabric is parametric, but best performance results are obtained with 16 instances of 4- or 8-bit PEs per stripe, so that we can define the datapath as *average grained*.

A simplified format of C, defined *Dataflow Intermediate Language* (DIL), is used as entry language for the PipeRench programming environment. As in the case of XiRisc, operators to be mapped on the fabric are described at Data Flow Graph (DFG) level by a single-assignment C-based format, where variable size can be specified by the programmer, and then translated on one or more PEs on the stripe after an automated Instruction Level Parallelism (ILP) extraction.

Another interesting example of coarse grain reconfigurable architecture is the PACT XPP digital signal processor [46]. It is composed by an array of heterogeneous *Processing Array Elements* (PAEs) and a low level *Configuration Manager* (CM). Configuration Managers are organized in a hierarchical tree that handles the bit-stream loading mechanism. Communication between PAEs is handled by a packet-oriented interconnect network. Each PAE has 16-bit granularity and is composed by synchronization register and arithmetical/logical operations, including multiplication. Data exchange is performed by transmission of packets through the communication network, while I/O is handled by specific ports located at the four corners of the array. The PACT XPP architecture is depicted in Figure 9.7, a detailed description is presented in 2.2.1.

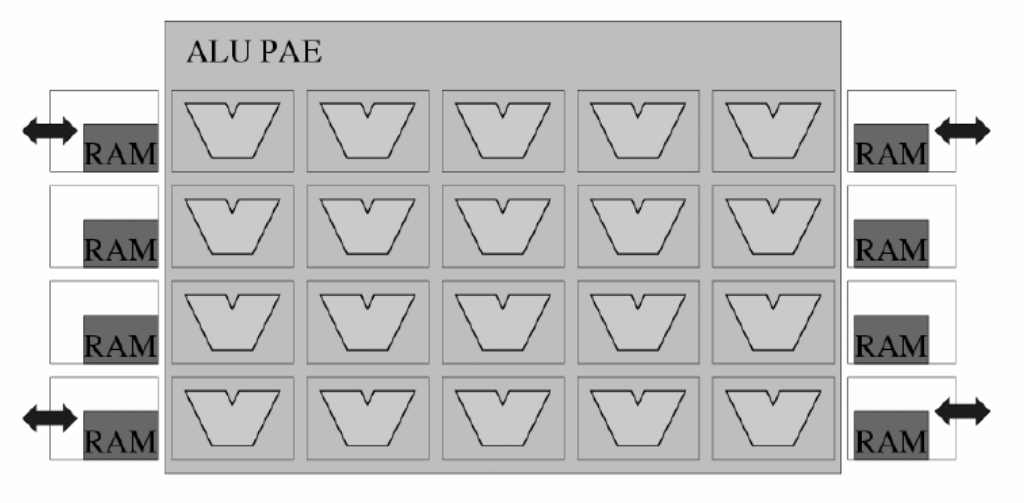

Figure 9: PACT Architecture

More or less both PACT and PipeRench explicitly propose to *replace* the concept of instruction sequencing (that is, cycle per cycle instruction fetching) by configuration sequencing (that is, spatial distribution, dynamically pipelined or not, of configuration bits) and they process data streams instead of single random accessed memory words. This concept of communication centric distributed computation is similar in principle to Transport Triggered Architectures (TTAs) [47], and it is indeed quite promising when applied to reconfigurable hardware because this micro-architectural paradigm, compared to the Von Neumann paradigm, appears more suitable to support a scalable number of function units, each with scalable latency and throughput. On the other hand, this promising approach has three main open issues:

- The communication infrastructure needs to be large yet flexible enough to allow the necessary throughput between the different function units (PEs).
- Tools and programming languages need also to describe synchronization between operators, and this requires structures and tools often unfamiliar to application developers.
- Lack of a memory addressing scheme: not all computation kernels in the embedded domain can be challenged with a streaming paradigm,

and for many cases it appears impossible to renounce to the addressing flexibility offered by standard cores.

Other coarse-grained devices are based on the concept of instruction set Metamorphosis introduced above, only utilizing a different architectural support for mapping extension segments: Morphosys [37], also shown in Figure 10, is a very successful RP that also been the base for a few successful commercial implementations.

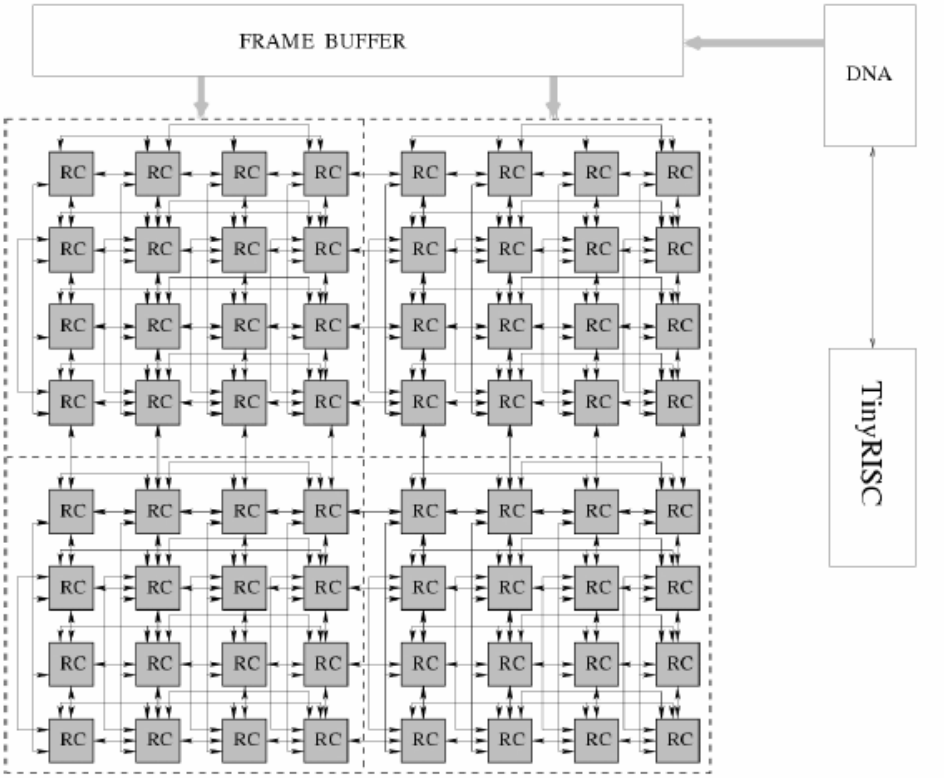

Figure 10: Morphosys Architecture

It is composed by a small 32-bit RISC core (TinyRisc), coupled to a so-called *Reconfigurable cell Array* composed of an 8x8 array of identical Reconfigurable Cells (RCs). Each cell is able to computes 16-bit words and contains multiplier, ALU, shifter, a small local register file and an input multiplexing logic. In order to minimize reconfiguration penalty, the architecture comprises a multi-context configuration memory, that is capable to overlap computation and configuration.

The array computation is organized in this way: all cells belonging to the same row receive the same control word and compute the same calculation over extended 128-bit words (8 x 16bit) as a purely Single Instruction Multiple Data (SIMD). Taking in account the array organization it's clear as the proposed architecture is characterized from a much higher area efficiency with respect to FPGA-based solutions described earlier, but it is also rather domain oriented: the machine is conceived for applications with relevant data parallelism, high regularity, and high throughput requirements such as video compression, graphics and image processing, data encryption and DSP transforms.

1.3 Interconnection Strategies

As shown above, Reconfigurable Instruction Set Processors have evolved from the mapping of combinatorial, single cycle functional extension of PRISC up to the very intensive hyper-parallel SIMD computational pattern of Morphosys-like architectures, but indeed the micro-architectural concept has remained more or less unchanged. The only aspect that has really changed is the computational grain of the ISA extension segments.

As a consequence of this shift, one architectural issue that is becoming more and more critical is the connection between reconfigurable units and the system memory in order to provide enough data to exploit the extension segment potential. Most coarse-grained datapaths such as PACT XPP or PipeRench do not actively intervene on the data layout: they simply consume data streams, provided by standard external sources or appropriately formatted by the RISC core or by specific DMA logic. Morphosys is only slightly more sophisticated, featuring a dedicated frame buffer in order to overlap data computation and transfers over orthogonal chunks of variable width. RPs based on FPGA fabrics, such as MOLEN, could map memory addressing as part of the microcoded extension segments, but this option could be costly in terms of resources and will make any kind of co-compilation impossible creating two different and separate compilation domains.

An interesting solution is that of ADRES (Architecture for Dynamically Reconfigurable Embedded systems) [50]. ADRES exploits a RFU similar to that of Morphosys, based on very coarse grained (32-bit) PEs implementing Arithmetical/Logical operations or Multiplications. Differently from Morphosys, the ADRES RFU is used as function unit in the frame of a VLIW processor. Data exchange with external memory is through the default path of the VLIW processor, and data exchanges take place on the main register file, as it was the case for the XiRisc processor described in section 4. The programming model is simplified because both processor and RFU share the same memory access. Even though the RFU has a grain comparable to PACT XPP or Morphosys, data feed is random accessed and very flexible, and it is not limited to data streaming. Still, the VLIW register file remains a severe bottleneck for RFU data access. A different solution is provided by Montium [51][52] a coarse grained reconfigurable processor composed of a scalable set of Tile Processors (TP). A TP is essentially composed by a set of 5 16-bit ALUs, controlled by a specific hardwired sequencer. Each TP is provided with 10 1Kbytes RAM buffers, feeding each ALU input; buffers are driven by a configurable *Address Generation Unit* (ATU). Montium can be seen rather as a flexible VLIW than a RP in the context described in this work, but it is affected by the same bottleneck shared by most RP overviewed above: in order to exploit its computational density, it needs to fetch from a repository several operands per clock, and possibly each of them featuring an independent, if regular, addressing pattern. In this respect, automated addressing generation based on regular patterns could be an interesting option: most applications that benefit from hardware mapping are based on loops, and addressing is more often than not generated and incremented with regularity as part of the loop. Automated addressing FSMs could add a new level of configurability to RPs, providing an adaptive addressing mechanism for adaptive units, enhancing potential exploitation of inherent parallelism. As it is the case with adaptive computation, automated addressing can be considered an option only if supported by solid compilation tools that could spare the end user from manual

programming. In fact, it appears theoretically possible to automatically extract from a high level (typically C/C++) specification of the algorithm regular addressing patterns to be applied to automated addressing FSM: the same issue has long been discussed for high-end Digital Signal Processors [53] and it is an open research field also for massively parallel systems based on discrete FPGAs [54]. These aspects are only very recently being evaluated in RP architectures.

DREAM [12][Section 3.3.1] is an example of reconfigurable processor that feeds its RFU through automated address generation. DREAM is an adaptive DSP based on a medium grained reconfigurable unit. Program control is performed by a standard 32-bit embedded core. Kernel computation is implemented on the RFU, composed of a hardware sequencer and an array of 24x16 4-bit PEs. The RFU accepts up to 12 32-bit inputs and provides 4 32-bit outputs per clock, thus making it impractical to access data on the core register file. For this reason, DREAM is provided with 16 memory banks similar to those of Montium. On the RFU side, an address generator (AG) is connected to each bank. Address Generation parameters are set by specific control instructions, and addresses are incremented automatically at each issue of an RFU instruction for all the duration of the kernel. AGs provide standard STEP and STRIDE [53] capabilities to achieve non continuous vectorized addressing, and a specific MASK functionality allows power-of-2 modulo addressing in order to realize variable size circular buffers with programmable start point.

1.4 General Outline of the MORPHEUS solution

The large-scale deployment of Embedded Systems is indeed raising new demanding requirements in terms of computing performance, cost-efficient development, low power, functional flexibility and sustainability. This results in an increasing complexity of the platforms and an enlarging design

productivity gap: current solutions are out of breath while current development and programming tools do not support the time-to-market needs.

MORPHEUS copes with these challenges by developing a global solution based on a modular heterogeneous SoC platform providing the disruptive technology of dynamically reconfigurable computing completed by a software oriented design flow and a consistent toolset. These "Soft Hardware" architectures will enable huge computing density improvements (GOPS/Watt, Giga Operations Per Second per Watt) by a factor of x100, reuse capabilities by x5, flexibility by more than 100 and time to market divided by 2 thanks to a convenient programming toolset.

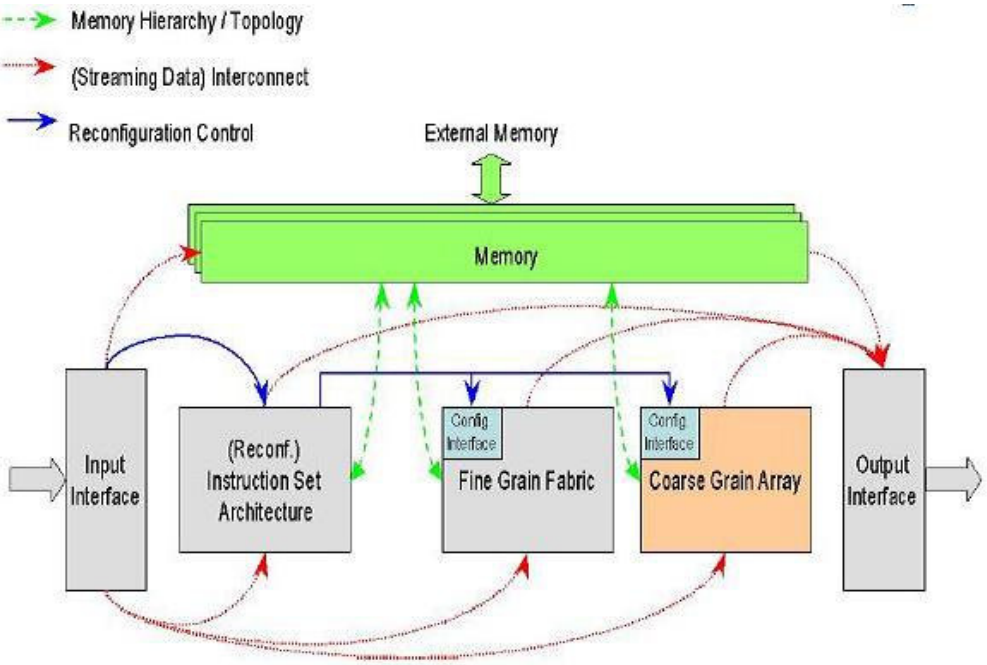

Figure 11: Architecture of a Heterogeneous reconfigurable device

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 Figure 11. 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 totalize the hardware architecture [3].

The MORPHEUS hardware architecture i.e. the MORPHEUS SoC is centered on three heterogeneous reconfigurable engines (HREs) targeting different types of computation:

- 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 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 favour the implementation of multiple smaller M2000 eFPGAs instead of a single large IP.

All control, synchronization and housekeeping is handled by an ARM 9< embedded RISC processor. As dynamic reconfiguration might impose a

significant performance demand for the ARM processor, a dedicated reconfiguration control unit is foreseen to serve as a respective off-load-engine.

Chapter 2 The MORPHEUS Design

2.1 The MORPHEUS Reference Architecture

Figure 12 describes a block diagram of the MORPHEUS architecture. The SoC architecture is organized in 3 main logic sub-blocks, which reflect the SoC programming model: a control and synchronization block centered on the ARM9 processor core, a computational intensive region formed by 3 separate computational engines defined Heterogeneous Computational Engines (HRE), and a data movement block composed by a multi-layered AMBA bus architecture and a Network-on-chip (NoC) infrastructure. The SoC programming model mirrors the physical architecture of the chip.

The end user interfaces with the ARM-centered region, handling the SoC as a single processor entity and making use of the standard state-of-art facilities offered by the processor–based environment that will be described in detail in the following. The innovative concepts and the heterogeneity of the computation and data movement regions are hidden by software libraries and hardware synchronization features as described by the Molen programming paradigm: the user works at high level of abstraction utilizing data chunks (streams) as operands and reconfigurable hardware operations as operators. The ARM processor handles computation and data transfer commands as "microcoded" instructions (defined as accelerated operations) that are then translated into bit-streams for reconfigurable hardware and Bus/NoC control statements for data transfers.

Other basic features of the toolset concept include the utilization of a Real-Time operating system (RTOS), that is strictly integrated with hardware services such as DMA control, interrupt management and hardware Configuration Manager (CM) to provide HRE/Data communication control and synchronization, and fast and smart handling of reconfiguration (bit-stream loading over HREs).

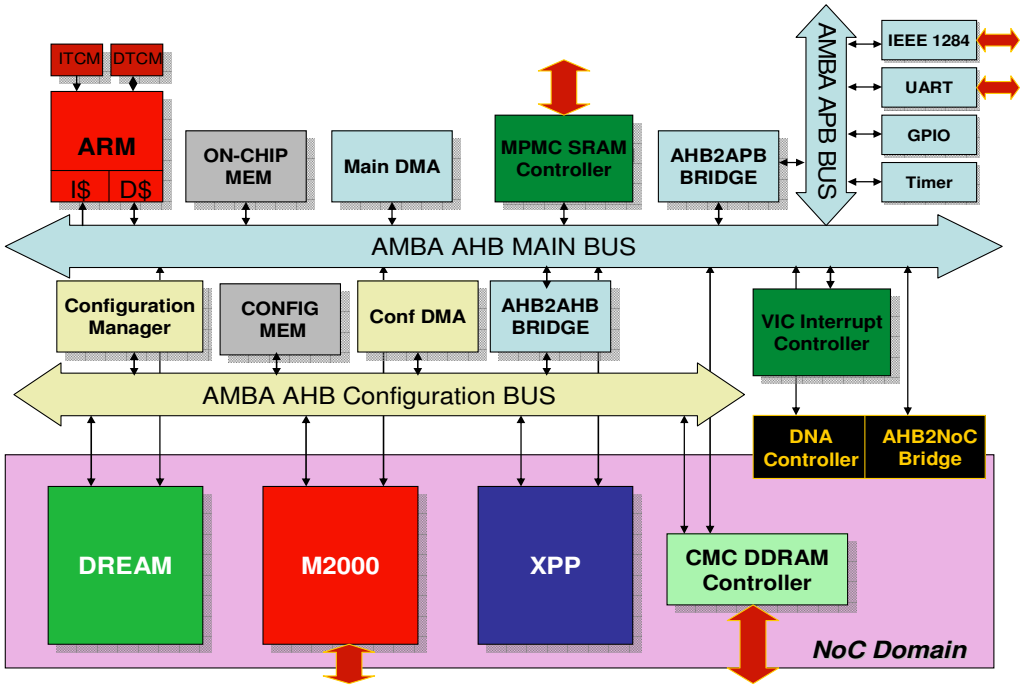

Figure 12: Morpheus Overall Architecture

To ensure ideal working frequency to all computational engines (that could also be dependent on the mapped application), HREs are integrated as independent clock islands. Local HRE buffer memories (Data Exchange Buffers, or DEBs) as used as a mean to propagate data across clock domains. Specific Configuration Exchange Buffers (CEBs) are used to propagate configuration bitstreams across clock domains (for details on DEBs and their connection to HREs see Section 3.3).

In this context, the design of the ARM-centered system infrastructure is a very critical step. ARM will be the main driver for most programming and control actions on all services offered by the Morpheus chip. Its criticality thus reside in the fact that in must be rock-stable, easy to use, easy to debug and well known and accepted both in terms of architecture and related tools. The usage of an unstable or not well known processor core could compromise the usability and user-friendliness of the whole system.

On the other hand, the innovation content of the overall system does not reside in this part, being rather centered on the computation engines and the relative data storage and transfer infrastructure. It is thus perfectly acceptable to make extensive use of state-of-art IP components and methodology. On the contrary, being this region the part of the system that is more closely interfaced with the off-chip world and the user point of view, it is mandatory to provide all the communication means, interfaces, and user utilities that may allow an easy, simple and efficient interface to the external word from the following points of view, roughly displayed in order of importance:

1. Application development, program compilation, RTOS utilization
2. Chip integration in a larger system to deploy peak computation efficiency
3. Chip utilization, test and verification
4. Clear Measurement of the chip performance

Figure 4 describes the MORPHEUS chip infrastructure, with detail on the ARM centered system control and user interface facilities; the infrastructure is composed by:

- One instance of ARM926EJS processor core
- A multilayered AMBA bus system
- A programmable DMA controller infrastructure
- An interrupt controller
- A set of IO peripherals to ease system control, communication, debug and test. They are not intended for fast communication during peak computation
- An external memory controller for off-chip communication

The reference architecture reflects the toolset organization specified based on the C language utilization and the *Molen* paradigm as a programming model

for the overall system organization. Through its bus architecture, exchange registers, and configuration and data exchange buffers (XR, CEB, DEB) the ARM core is capable to drive and controller for all hardware services and computational units in the SoC, including all data and configuration transfers, so that the RTOS is put in condition to control and synchronize data movementation and configuration and computation on the HREs. Through the specific configuration bus, DMA and connection through external memory it provide the means for the Dynamic configuration handling.

The following section will briefly describe the components outlined above, their configuration in the MORPHEUS architecture and the motivations that drove their selection.

2.1.1 ARM926EJ-S Embedded processor

The ARM926EJ-S processor is a member of the ARM9 family of general-purpose microprocessors. The ARM926EJ-S processor is targeted at multi-tasking applications. The main features that make it suitable to the MORPHEUS context are full memory management, high performance, low die size, and low power consumption.

ARM926EJ-S supports the 32-bit ARM and 16-bit Thumb instruction sets, enabling the user to trade off between high performance and high code density. The processor features a standard load/store RISC Harvard cached architecture and provides a complete processor subsystem, including:

- an ARM9EJ-S integer core
- a *Memory Management Unit* (MMU)
- separate instruction and data AMBA AHB bus interfaces
- separate instruction and data Tightly coupled memories

The ARM926EJ-S processor implements ARM architecture version 5TEJ. The TCM interfaces enable nonzero wait state memory to be attached, as well as providing a mechanism for supporting DMA access for fast reloading.

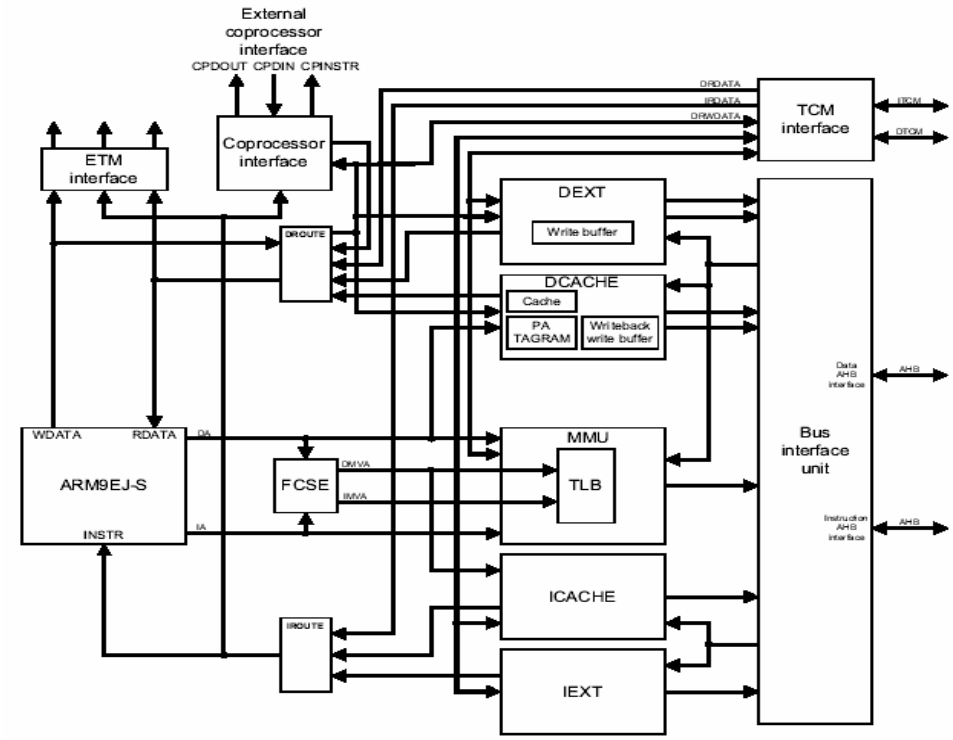

Figure 13: ARM926EJ-S block diagram

Figure 13 shows a block diagram of the ARM926EJ-S macrocell. Most important, the ARM926EJ-S supports the ARM debug architecture and includes logic to assist in both hardware and software debug through a specific JTAG connection. It will be thus possible to perform in-circuit source line debugging on the whole MORPHEUS system controlling the final board from a ARM debugger window. The device testability is enhanced by the ETM9 interface, which allows the user easy tracing of the code executed by the core in a specified timing window.

2.1.2 Multi-Layer AMBA bus system

Figure 14 describes the MORPHEUS bus system excluding the Network on Chip infrastructure. The bus architecture is based on the following layers/busses:

1. A main AMBA AHB bus controlled by the ARM processor, featuring a specific DMA controller. This bus is used by ARM for data and instructions (mostly ARM will work on caches and tightly coupled memories, so the impact of this transfers on the bus at peak computation will be negligible). Moreover, the main bus may access to HRE local memories for control and debug purposes. This bus is also used for all memory mapped control registers present in the system: HRE exchange registers, DMA control registers, Network-on-chip control registers, Configuration Manager control registers. Through bridges, this bus acts as a master on all secondary bus layers in the system. The main bus also features a dedicated access to the external memory controller.
2. A configuration AMBA AHB bus controlled by the Configuration Manager, featuring a second specific DMA controller. The bus will control all configuration ports relative to HREs, and have a dedicated access to the external memory controller to provide fast access to off-chip configuration repositories.
3. An AMBA APB bus driving chip peripherals, described in Section 2.1.5.

In the design of the MORPHEUS system, the bus architecture cannot be considered as an element of innovation; high speed, innovative interconnect strategies are implemented in the Network-on-chip design. The specifications for the bus are to provide necessary performance, and most of all to guarantee a low risk margin and offer good flexibility and programmability to the user.

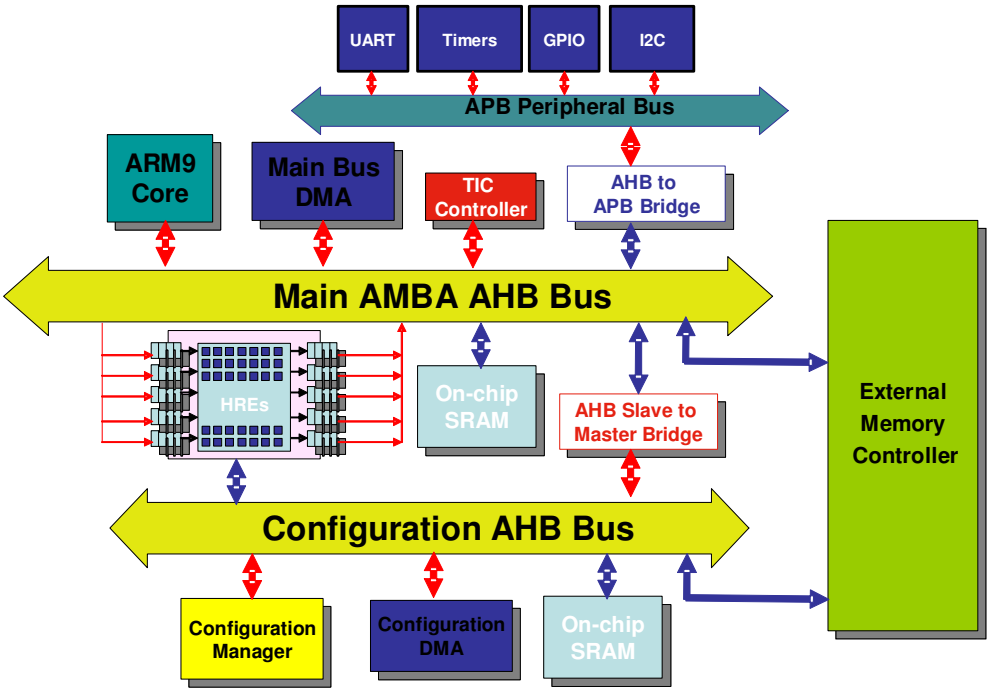

Figure 14: MORPHEUS multilayer bus hierarchy

The AMBA bus protocol, initially developed by ARM, is considered as state-of-the-art by most SoC developers both in industry and academia, and is present in a very wide range of SoC products in the market landscape. In the MORPHEUS context, it provides two essential advantages:

- The ARM processor, its MMU, and various peripherals distributed by ARM provide native support for the AMBA protocol.
- A lot of bus components, peripherals and utilities are distributed as pre-verified IP blocks both in the open-source world and in the IP market, not only in terms of HDL code for silicon implementation but also in terms of SystemC library for design exploration and system simulation

To minimize risk margin and to provide a state-of-art solution, the MORPHEUS bus architecture was built utilizing Synopsys DesignWare IP

components to implement both the bus architecture itself and the DMA controller (Section 2.1.3).

2.1.3 DesignWare DW_ahb_dmac DMA Controller

In order to implement an high level programming pattern where the MORPHEUS user handles macro-operations and data chunks and/or streams, it is necessary to relieve the ARM processor, that represents the user interface towards to MORPHEUS system, of the task of transferring data through the interconnect resources and in between the memory hierarchy and from/to the different HREs.

For this reason, programmable DMA controllers will be utilized for implementing data transfers both on the main bus and on the configuration bus. As described in Section 4.4 the network-on-chip architecture will provide the same application program interface (API) of the DMAs in order to offer homogeneous access to data transfers.

In order to profit from a well-known and "safe" architecture and minimize integration risk it was decided to utilize an IP library as DMA controller: the Synopsys DesignWare AHB DMAC. The main features of the controller are the following:

- AMBA AHB based DMA Controller core that transfers data from a source peripheral to a destination peripheral
- Supports multi-layer DMA transfers when the source and destination peripherals are on different AMBA layers (Figure 15)
- Multi-context: supports up to 8 concurrent channels (source/destination pairs). Channels are unidirectional (data transfers in one direction only)
- Programmable channel priority

- Channel buffering: one FIFO per channel, Configurable FIFO depth, Automatic data packing or unpacking to fit FIFO width

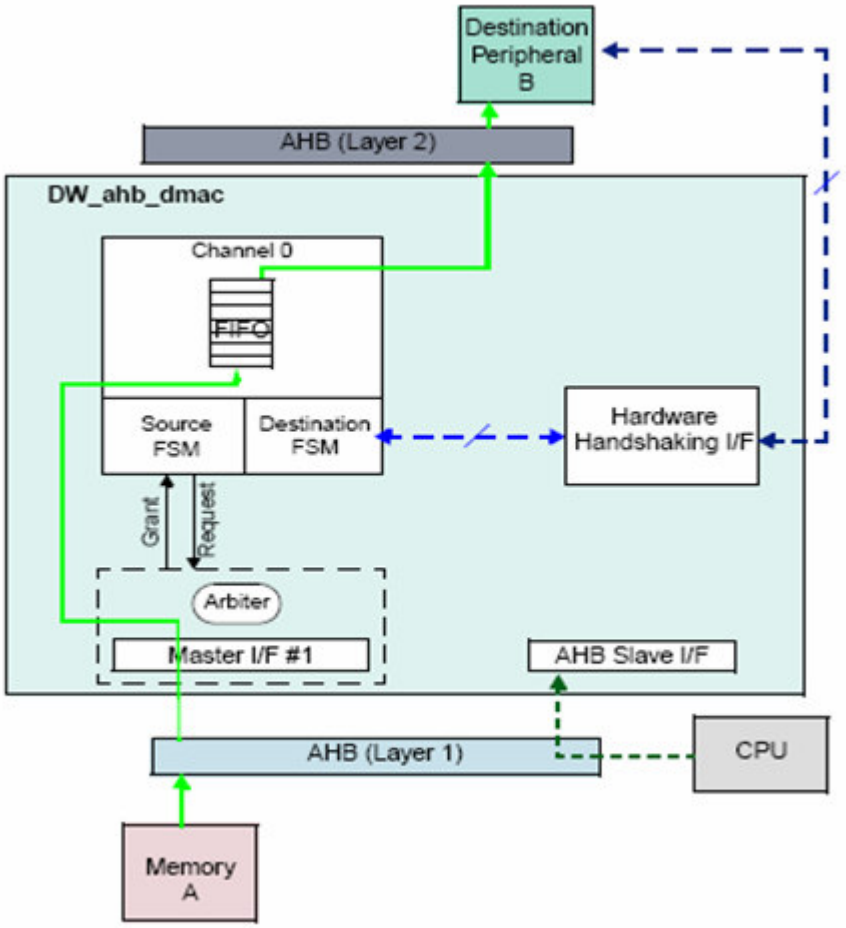

Figure 15: Scheme of a cross-layer DMA transfer

Figure 16 describes the hierarchy of any DMA transfer. Software programming by ARM controls the number of blocks in a given transfer. Once the DMA transfer has completed, the controller disables the channel and generates an interrupt to signal the DMA transfer completion. The amount of blocks and the block length is determined by the flow controller, an FSM integrated in the DMA unit that partition the overall transfer required by the driving processor in a suitable collection of partial transfers. For transfers between the DW_ahb_dmac and memory, a block is broken directly into a sequence of AMBA bursts and AMBA single transfers.

The DMA controller is provided with a set of software libraries that are used to hide details and provide high-level functionality to the user.

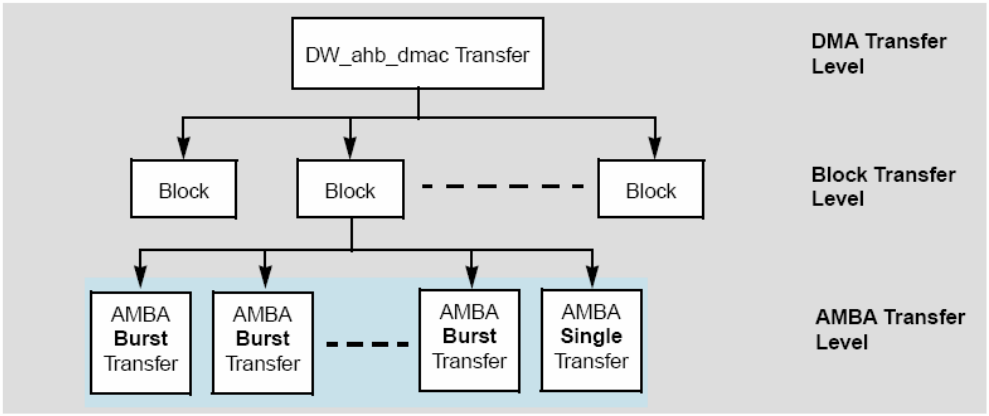

Figure 16: DMA transfer hierarchy

The DW_ahb_dmac is natively designed to support the AMBA AHB bus architecture. It is capable to exploit all transaction mechanisms featured by the protocol, thus ensuring the ideal utilization of efficient burst transfers. Moreover, the multi-layer support offered by this DMA controller is very useful in the context of the MORPHEUS architecture, i.e. to allow fast transfers between the main data bus and configuration data bus (this option may be used for testability to read-back the configuration memories of HREs, or to explicitly utilize the ARM core as configuration manager in some specific application or, again, for testability purposes).

Transfer type	Bandwidth (Mbit/sec)
Single layer transfer	1950
Dual layer transfer using different memory banks (2 master ports & 2 layers involved)	2700
Two independent transfers using two channels with different layers and memory blocks (4 master ports & 4 layers involved)	5250

Table 1: DesignWare DMA Bandwidth estimation

The previous table describes the bandwidth achieved by the DW_ahb_dmac, with a reference speed of 200MHZ, single block transfers of 1KB (256 words) between memory addresses, no other master device accessing the bus.

2.1.4 Interrupt Controller

The inclusion of an interrupt controller appears mandatory in the MORPHEUS architecture. On one hand, the flexibility offered by the ARM9 core in the interrupt handling is limited to two interrupt pins, one fast and one slow interrupt request. On the other hand, the MORPHEUS programming pattern relies a lot on interrupt handling for the synchronization of "microcoded" instructions, that is data chunk/stream transfers (macro-operands handling) and operation on HREs (macro-operations triggering).

The evaluations performed above on the convenience of utilizing pre-verified IP components are very much applicable also to the Interrupt controller selection. As part of the MORPHEUS reference architecture, it is made available the PrimeCell Vectored Interrupt Controller (PL190) by ARM.

As it is the case for the PL175 PrimeCell described in Section 2.1.3, this block is developed and distributed by ARM Ltd and it is especially designed to work with the ARM processor. Moreover, it is distributed as a pre-verified block for inclusion in SoC design. Moreover, PL190 has been used in many commercial products to support ARM-based real time operating systems.

The most relevant features of the PL190 are:

- Compliant to the AMBA bus protocol specification
- Control and status registers mapped on AHB for fast interrupt response
- Support for 32 standard interrupts, 16 vectored IRQ interrupts
- Hardware interrupt priority

- Software interrupt generation
- Interrupt masking, privileged mode support
- Vector interrupt controller daisy-chaining support

The PL190 provides essentially a software interface to the interrupt system. Through memory mapped register access to the interrupt controller, software (user routines/libraries/RTOS) can determine the source that is requesting service and where its service routine is loaded. It supplies the starting address, or vector address, of the service routine corresponding to the highest priority requesting interrupt source.

There are 32 interrupt lines. The PL190 controller uses a bit position for each different interrupt source. The software can control each request line to generate software interrupts.

2.1.5 MORPHEUS IO Peripheral Set

The main AMBA AHB bus matrix is provided with a bridge to an APB (Advanced Peripheral Bus) that will feature a set of IO peripherals for enhancing the chip observability and debugging. The APB bus can be driven by any master of the AMBA bus, so it will normally be ARM, but can be driven by DMA for chunk transfers or by TIC protocol for testability purposes. These peripherals include:

- A UART port for the implementation of a serial transmission protocol. This connection is used by the ARM processor to realize an external virtual terminal on a host test processor for easy remote control of the chip. This feature is particularly useful in the preliminary testing phase and to implement interactive demonstrations of the chip/board.

- An I2C connection that can be used to provide on-board connection between multiple instances of the MORPHEUS chip to build composite high performance systems.
- A set of programmable timers normally used to implement timeouts and watchdogs and to allow multithreaded elaboration by the Operating System.
- A set of general Purpose IO register multiplexed on a set of output pins, normally used to drive 7-segments or LCD displays to ease testability and verification of the architecture.

It should be noted that most of the described features are potentially suitable for a mapping on the eFPGA fabric rather than on std-cell technology. This would give an added-value to the demonstration of the flexibility of the MORPHEUS approach.

The eFPGA fabric will be provided with access to a set of IO Pads to support this design option.

2.1.6 MPMC PL175 Memory Controller

Note: The PL175 memory controller is part of the "MORPHEUS reference architecture", and is intended as a proposal reference at this stage of the project, but it could be substituted by different design option if a more suitable solution becomes available in the following course of the project.

The PrimeCell MPMC is an Advanced Microcontroller Bus Architecture (AMBA) compliant System-on-Chip (SoC) peripheral that is developed, tested and licensed by ARM Limited. The PrimeCell MPMC offers:

- AMBA 32-bit AHB compliancy
- Dynamic memory interface supports SDRAM, DDR-SDRAM and low-power variants

- Asynchronous static memory device support including RAM , ROM and Flash, with or without asynchronous page mode
- Read and Write buffer to reduce latency and to improve performance
- Eight AHB interfaces for accessing external memory with programmable priority mechanism
- 8-bit, 16-bit and 32-bit wide static memory support
- 16-bit and 32-bit wide databus SDRAM and SyncFlash memory support. 16-bit wide databus DDR-SDRAM support
- Separate AHB interface for programming the MPMC control registers
- Locked AHB transactions supported, Support for all AHB burst types
- Support for the *External Bus Interface* (EBI) that enables the memory controller pads to be shared
- Integrated *Test Interface Controller* (TIC) for monitoring bus activity on the internal MORPHEUS AMBA architecture

In the context of this section, the main features that suggest the utilization of the PL175 are its native compliancy with the ARM AMBA bus architecture, the bus monitoring and debug facility offered by the TIC test protocol an the large number of available channels with programmable priority. Moreover, being a pre-defined component distributed by ARM, and integrated in many existing products on the market in the technology targeted by MORPHEUS, it has a low integration risk margin that could be beneficial.

2.2 Reconfigurable Engines

2.2.1 XPP

The XPP array is a coarse grained reconfigurable tile, specialized for data flow type of algorithms. The following section provides a rough overview about XPP array. In the second part the Function PAEs and their integration into the XPP array are described. The XPP IP is scalable in terms of array size and routing capabilities. The available parameters are summarized in Table 3.

The XPP architecture provides parallel processing power combined with fast reconfiguration. The last version which is named XPP-III is currently under development and 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: Configurable Processing Array Elements (ALU-PAEs and RAM-PAEs) are arranged in an array and communicate via point-to point communication links; Programs are mapped as flow graphs to the array of ALUs and RAMs; Communication is packet-oriented with auto-synchronization; Control of programs is handled by an independent event network; The array provides fast dynamic reconfiguration; I/O supports streaming and memory mapped I/O.

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

2.2.1.1 The XPP-Array Overview

Figure 17 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".

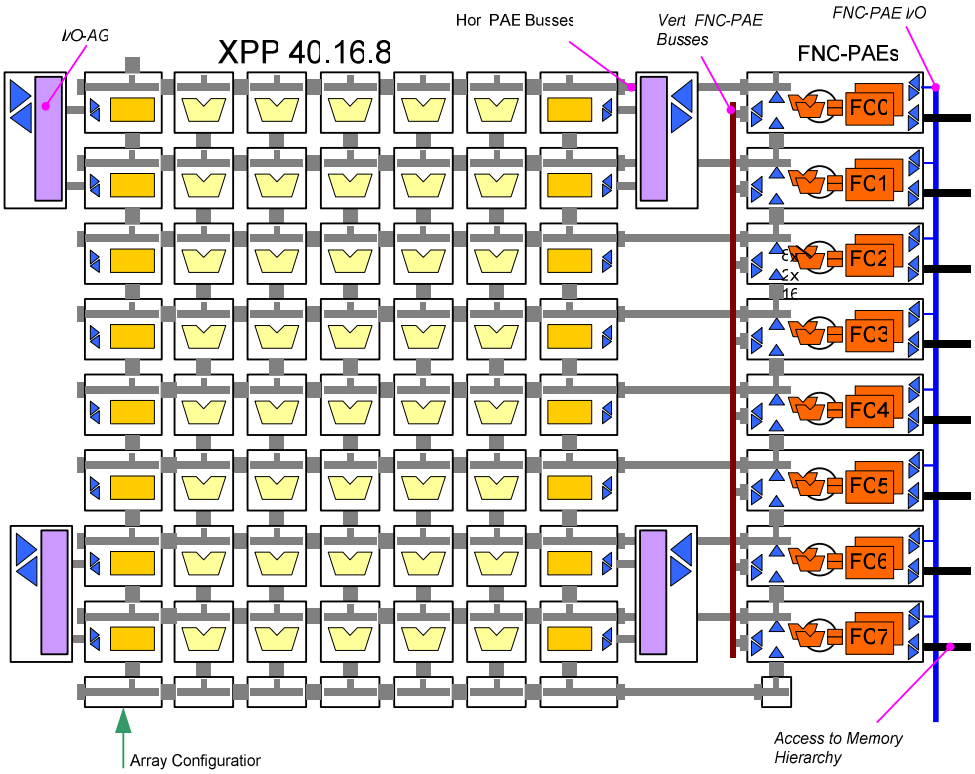

Figure 17: An XPP array with 6x5 ALU-PAEs

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 through the configured communication network. A c*onfiguration* 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.

2.2.1.2 Function PAE Overview

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.

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

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

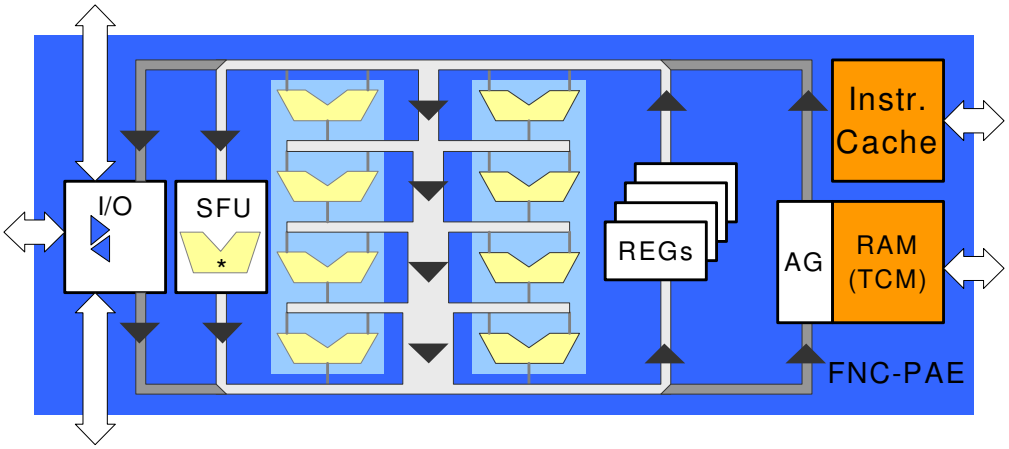

Figure 18: FNC-PAE

The communication with the XPP-array (Figure 18 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.

2.2.1.3 XPP- III Third Generation Core Details

As outlined in the previous overview, the XPP is built in a scalable and modular way. The following section describes roughly the concept and elements of the XPP technology. The XPP–III IP comprises

- the XPP-array
- the Function PAEs.

The XPP-III array uses only a handful different functional blocks: ALU-PAEs (processing array elements) perform the basic calculations, RAM-PAEs provide a static RAM together with an ALU and I/O interfaces. In addition, FNC-PAEs provide sequential processing capabilities. Each PAE contains several "objects". All objects are integrated with the communication channels

of the array, providing point-to-point connections. The configuration of the array is done by a pipelined bus-system.

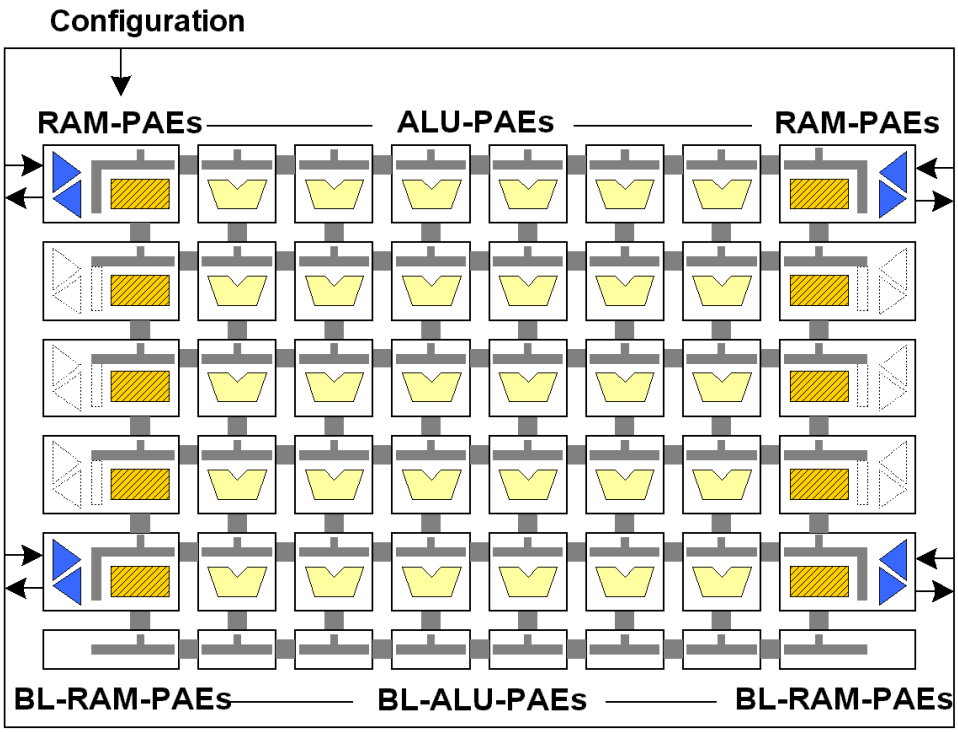

Figure 19: A sample XPP -array (6x5 ALU PAEs)

The XPP-III core is built from a rectangular array of ALU-PAEs, RAM-PAEs at the left and right side of the array. At the bottom line specialized PAEs (BL-ALU-PAE, BL-RAM-PAE) provide routing channels beneath the PAEs. Figure 19 shows a sample array with 30 ALU-PAEs, 10 RAM-PAEs, and I/O. Only the I/O of the RAM-PAEs at the corners of the array is used. The data word size is 16 bit.

ALU-PAEs comprise three objects and a connection-matrix. ALU-PAEs enclose an ALU-object featuring a typical DSP-command-set including multiplication. The BackRegister-object (BREG) is used for routing from bottom to top, for arithmetic and normalization. The ForwardRegister-object (FREG) provides routing channels from top to bottom and a specialized unit with data-flow operators. The objects have input registers and a one-stage transparent FIFO which can be preloaded during configuration. The output

Register (DF Register) is able to buffer one packet if the transfer to the next connected object stalls.

Vertically, each object can be connected to one or more horizontal busses. Configurable registered switch-objects are used for segmenting the communication lines horizontally to the neighboring PAEs. In parallel to the data connections, the similarly designed independent event-network (dotted connections in Figure 20) enables the transfer of status information from the ALUs. Events can be used to steer the data flow or to control the operation of ALU-opcodes. The BREG provides a look-up table for manipulation of several event streams.

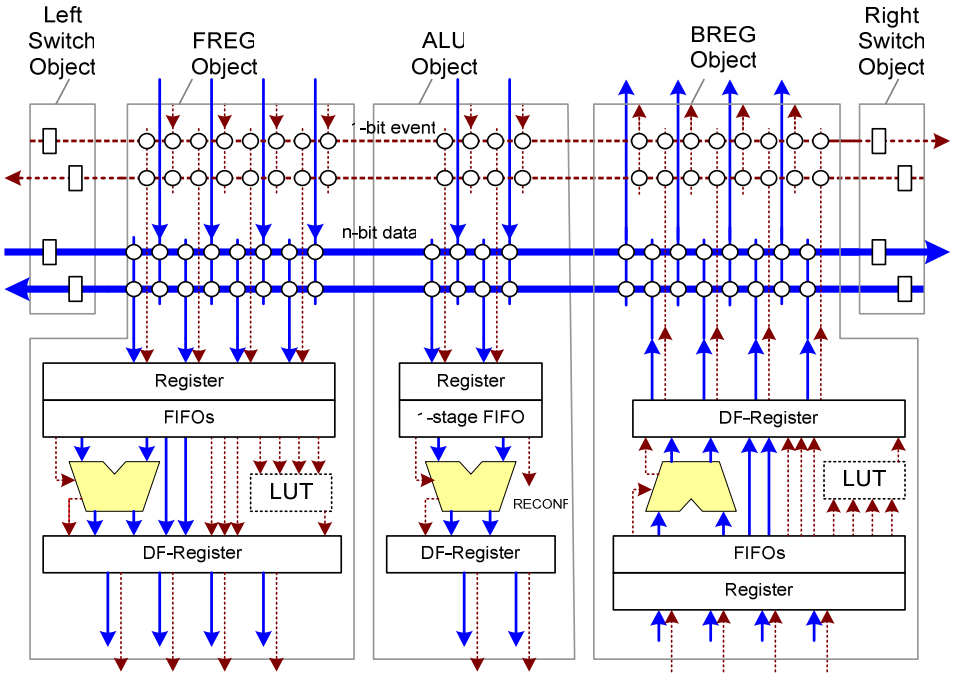

Figure 20: ALU PAE objects

RAM-PAEs are similar to ALU-PAEs, merely the ALU object is replaced by a RAM-object and the I/O Element is integrated.

The dual-ported RAM-object has two independent ports enabling simultaneous read and write operations. As with all XPP objects, the RAM offers packet-oriented data handling. To read from a RAM-object, a data

packet must be sent to its address input. As a result, the RAM-object generates an output packet with the content of the addressed RAM cell. Similarly, writing to RAM requires sending data packets to the address-inputs and the data-inputs of the write port. If the RAM is configured in FIFO-mode, no addressing is required and the FIFO generates output packets as long as packets are stored. RAMs and FIFOs can be preloaded during configuration. This allows using them as look-up tables or for storing coefficients and initialized parameters. Events may control read and write operations and inform about the status of the RAM-objects if they are concatenated to larger capacity RAMs.

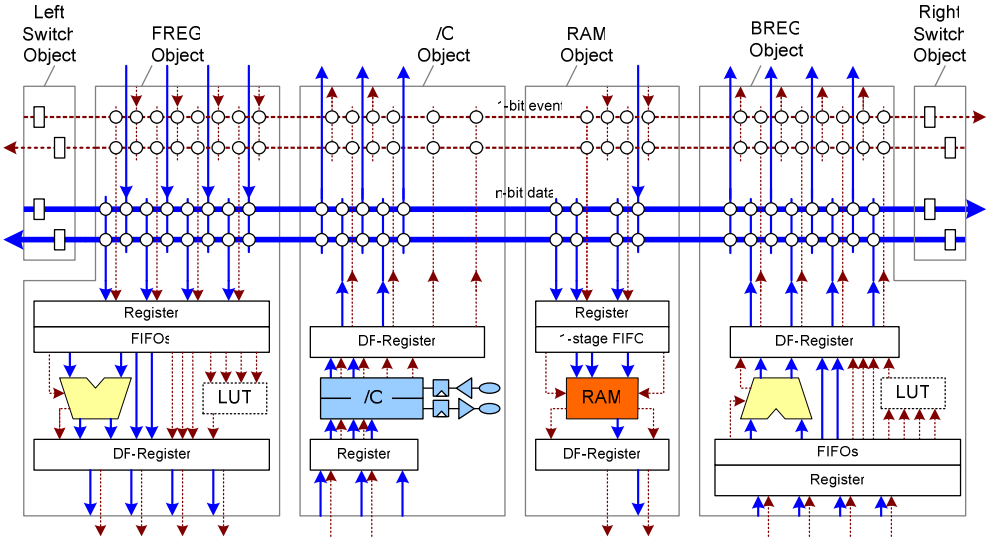

Figure 21: RAM PAE objects with I/O

Figure 22 shows the general structure of an I/O Interface in streaming mode. The I/O interface provides two data channels and two single bit event signal channels. All four channels can operate independently using an identical streaming protocol. Alternatively, the data channels can be combined to support external RAMs. This mode allows directly addressing an external memory module. Such memory can be used as larger external buffer RAM and for exchanging data blocks (scratch-pad). The RAM mode interface is mainly intended for algorithms which require random access to the storage repository or where the memory's FSM does not provide the required access pattern.

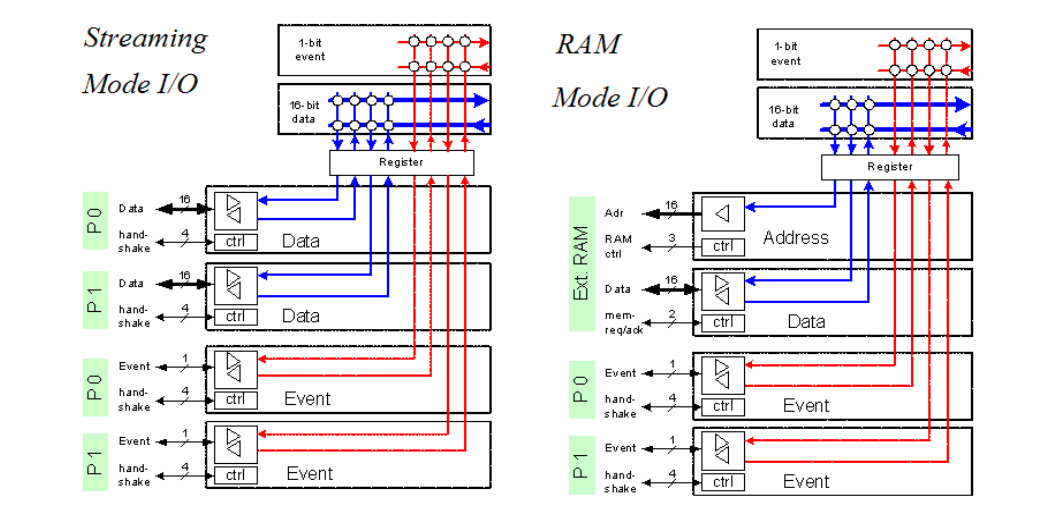

Figure 22: XPP I/O in Streaming mode & RAM mode

2.2.1.4 XPP III Function PAE Details

This section provides some details of the Function PAEs which handle the control-flow part of algorithms.

The FNC-PAE is based on a *load/store VLIW* architecture. Unlike VLIW processors, it comprises implicit conditional operation as well as sequential and parallel operation of ALUs within the same clock cycle. Program code is stored in a local cache which can be locked. Data is stored in a local tightly coupled memory (D-MEM) and (optionally) external RAM.

The ALU data-path comprises eight 16-bit wide integer ALUs arranged in four rows by two columns. Data processing in the left or right ALU column (path) occurs strictly from top to bottom. This is an important fact since conditional operation may disable the subsequent ALUs of the left or right path. The complete ALU datapath is executed within one clock cycle. The final result is written to the register file or other target registers within the very same clock cycle. Status flags of the ALUs are fed into the next row of ALUs. The status flags of the bottom ALUs are stored in the status register. Flags from the status register are used by the ALUs of the first row and the instruction decoder to steer conditional operations. This model enables the efficient execution of

highly sequential algorithms in which each operation depends on the result of the previous one.

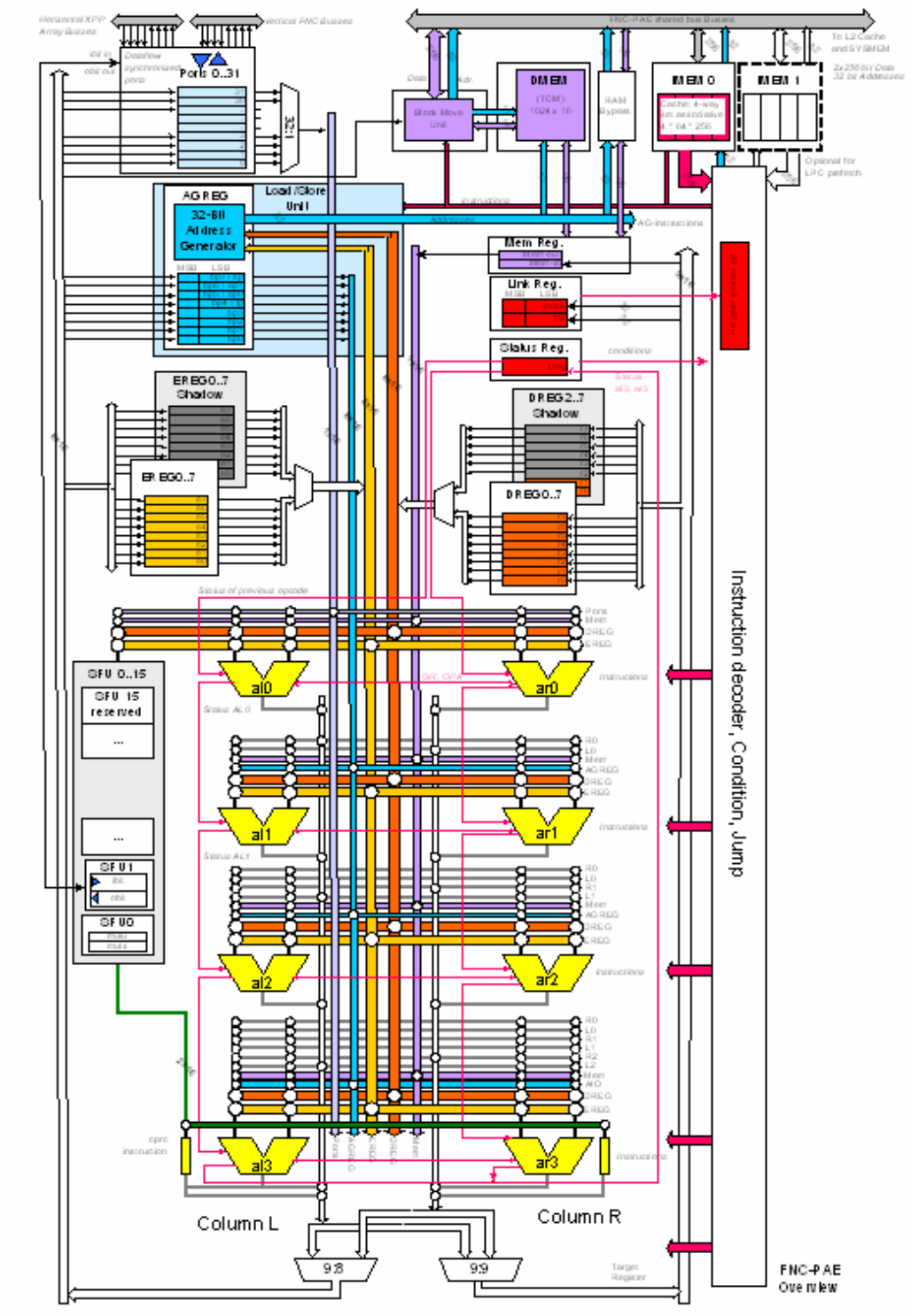

Figure 23: FNC-PAE overview the ALU data-path

All ALUs have access to the 16-bit register file. Additionally each ALU has access to the previously processed results of all ALUs above. In order to achieve low latencies within the ALU data-path, the ALUs support a restricted set of operations: addition, subtraction, compare, barrel shifting, and boolean functions as well as jumps. More complex operations are implemented separately as SFU functions. Most ALU instructions[3] are available for all ALUs, however some of them are restricted to specific rows of ALUs.

Conditional Operation and Branching

Many ALU instructions support conditional execution, depending on the results of the previous ALU operations, either from the ALU status flags of the row above or – for the first ALU row - the status register, which holds the status of the ALUs of row 3 from results of the previous clock cycle. When a condition is FALSE, the instruction with the condition and all subsequent instructions in the same ALU column are deactivated for this cycle. A deactivated ALU column can be reactivated again.

Three pointers are used for branching based on conditions. Without a condition, one *pointer* points to the next opcode. It is possible to select one of the three pointers based on results of a condition for relative branch targets between +-31. Long jumps are possible with dedicated ALU instructions or using *a* special register (*lnk*).

Multiple types of jump instructions are supported:

- Opcode implicit program pointer modifiers using the program pointers
- Long jump Instruction with immediate or register offset
- Subroutine calls and return with immediate or register offset (stack)

- Jumps via the 32-bit *lnk* register for subroutine call w.o. delay and stack operations
- Interrupt calls and return via *intlnk* register

Memory Hierarchy

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 pre-fetched under program control. The D-MEM is organized as a linear 1024 x16 bits.

The access to the external memory hierarchy depends on the overall SoC design.

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.

2.2.1.5 XPP Interfacing

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.

Protocol wrappers can adapt the streaming channels to any SoC infrastructure.

Configuration Interface

A pipelined configuration bus configures the objects within the PAEs. Figure 24 shows the configuration chain in the above-mentioned XPP-array.

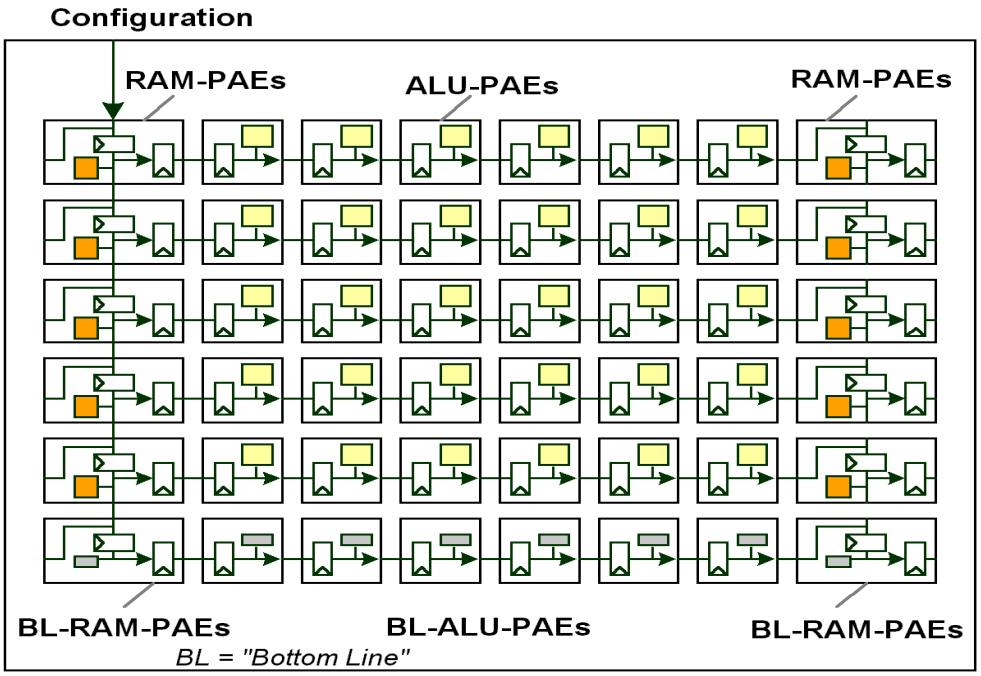

Figure 24: Configuration chain

Tokens with address and data are shifted pipelined through the array. Each configuration register has a unique address. The word width of the Configuration Bus is 43-bits. Thus, an external wrapper must adapt e.g. 32-bit DMA transfers to the required 43-bit words.

One should note that the configuration only programs the XPP-array and the horizontal busses which go to the FNC-PAEs (not shown in Figure 24). The

FNC-PAEs and their vertical dataflow busses are controlled by the FNC-PAEs itself by means of the FNC-I/O Bus.

Characteristics of the XPP-III IP

The following characteristics are derived from Synopsis tools. For the power estimation typical applications have been performed (i.e. MPEG2 inverse quantization and MPEG4 quarter pixel reconstruction) on an XPP 40.16.0. The algorithms are applied on a 16x16 pixel macro block and deliver one result / cycle.

The figures are intended to give a rough estimate about the required area and power budget. One should note that after backend processing more area will be required and the maximum frequency will drop.

XPP-III Array	Technical Data (Synthesis)			
XPP 40.8.0			dynamic & typical values	
Technology	area for 100 MHz design	f max	power @100 MHz	energy/ cycle
	[mm^2]	[MHz]	[mW]	[nJ]
GPLVT - 90nm, low treshold	11,7	400,0	88,0	0,88

Table 2: XPP-III array preliminary characteristics

The XPP-III hardware IP can be scaled by a number of parameters, which are defined in the following table.

Item	Parameter	Description	Range
XPP-core structure	General structure of XPP array and word width		
	PAE_COLUMN	Number of PAE columns per PAC (includes RAM PAEs)	>= 6
	PAE_ROW	Number of PAE rows. Specifies also the number of FNC-PAEs.	>= 2
	DATA_CH	Number of Data Channels per PAE for each direction Full featured I/O requires 6 or more	2 .. 8
	EVENT_CH	Number of Event Channels per PAE for each direction Full featured I/O requires 6 or more	2 .. 8
RAM Object	Internal RAM Object, RAM and FIFO Mode		
	IRAM_ADR_MASK	Address_mask_range : Address masking bits of the address mask	4
	IRAM_ADR_WIDTH	Number of address bits	8
BREG Object	Vertical bottom-up routing, Event Processing and special ALU		
	BREG_DATA_PORTS	Data routing ports (for BREG-ALU two routing ports are utilised)	2 .. 4
	BREG_EVENT_PORTS	Event routing ports	4 .. 5
FREG Object	Vertical top -down routing, Event Processing and special ALU fro dataflow steering		
	FREG_DATA_PORTS	Data routing ports (for FREG-ALU two routing ports are utilized)	2 .. 4
	FREG_EVENT_PORTS	Event routing ports	4 .. 5

Table 3: XPP-III array hardware IP parameters

2.2.1.6 Mapping of Algorithms to the XPP-array

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

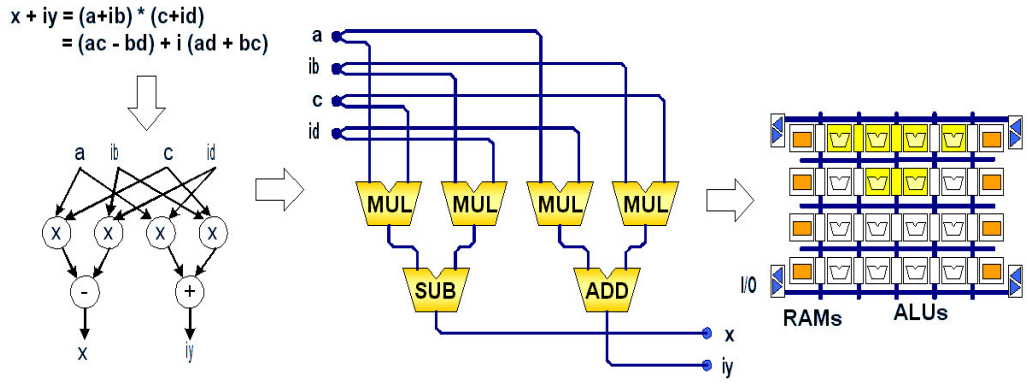

Figure 25: Flow-graph of a complex multiplication and spatial mapping

Figure 25 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.

2.2.1.7 Data and Event Streams

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.

2.2.1.8 Development tools

Due to the fact that XPP array is not a standard sequential processor and also no fine-grained FPGA, specialized development tools are provided. A tool 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.

2.2.2 PiCoGA

The PiCoGA 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 an homogeneous island-style architecture, the PiCoGA is composed of three main sub-parts, highlighted in Figure 26:

- An 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 12 32-bit inputs and 4 32-bit outputs, thus allowing for each PGAOP to read up to 384 bits and to write 128 bits.

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.

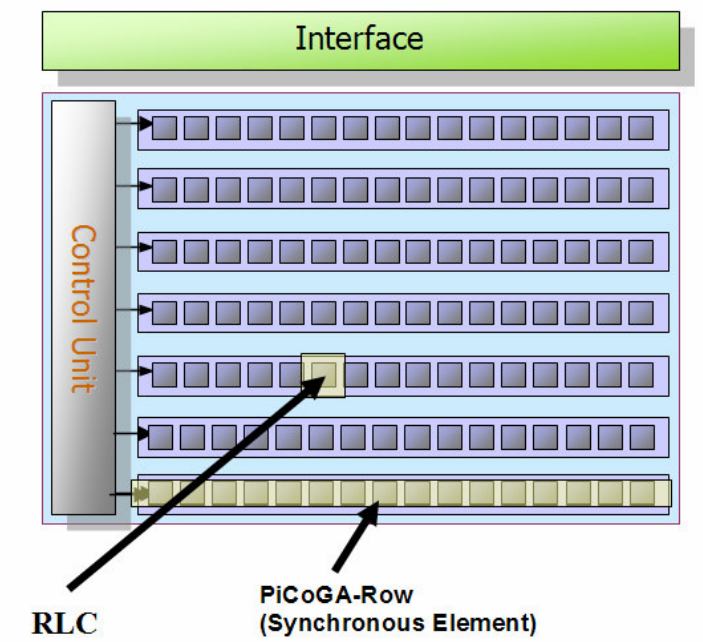

Figure 26: Simplified PiCoGA Architecture

If we exclude the interface block, the PiCoGA is a custom designed array, thus scalability and modularity is limited and requires additional work. In fact, the PiCoGA is a fixed-size architecture but more than one PiCoGA instances can be considered for the MORPHEUS design in order to further improve the

RLC

overall computational power. The PiCoGA interface supports the propagation of the dataflow paradigm used inside the PiCoGA at an instance level, thus obtaining a hierarchical pipeline.

2.2.2.1 PiCoGA Architecture

The main features of the PiCoGA architecture are:

- A fine-grained configurable matrix of 16x24 RLCs
- A reconfigurable Control Unit, based on 24 Row Control Units (RCUs) that handle the matrix as a datapath (see Figure 26).
- 12 primary 32-bit inputs and 4 primary 32-bit outputs
- 4 configuration contexts are provided as a first-level configuration cache; only 2 clock cycles are required to change the active context (context switch) and 1 configuration context can be active at a time.
- Up to 4 independent PiCoGA operations can be loaded in each context, featuring partial run-time reconfiguration

Each RLC can compute algebraic and/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 2 32-bit operations (or 4 16-bit, 8 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. **Figure** 27 shows an example of pipelined DFG mapped onto PiCoGA.

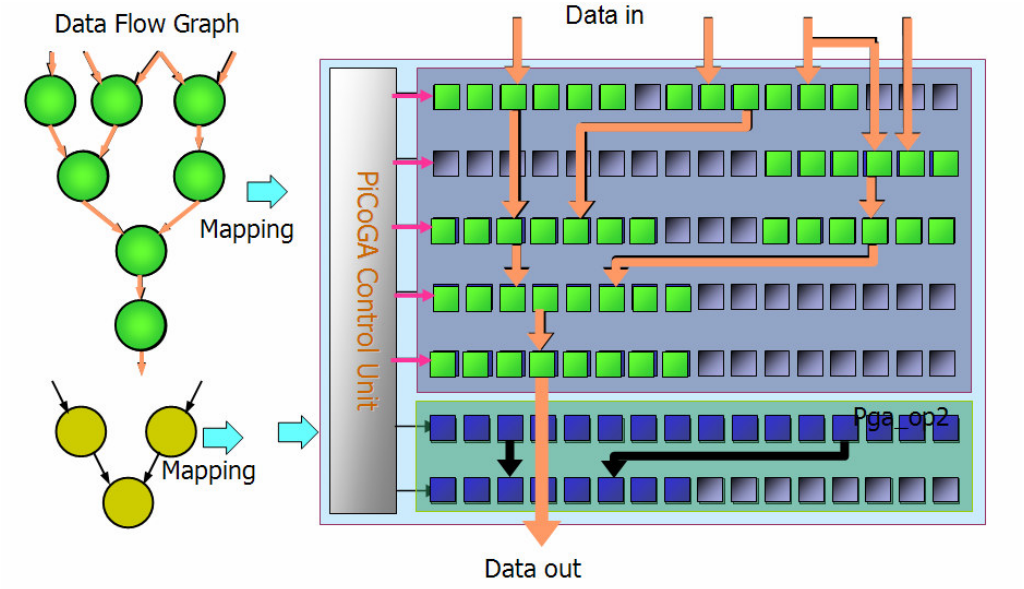

Figure 27: Pipelined DFG in PiCoGA

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. 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).
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*. Figure 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 are relocable, thus they can be loaded in any configuration layer starting from any available row.

Figure 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 Figure 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.

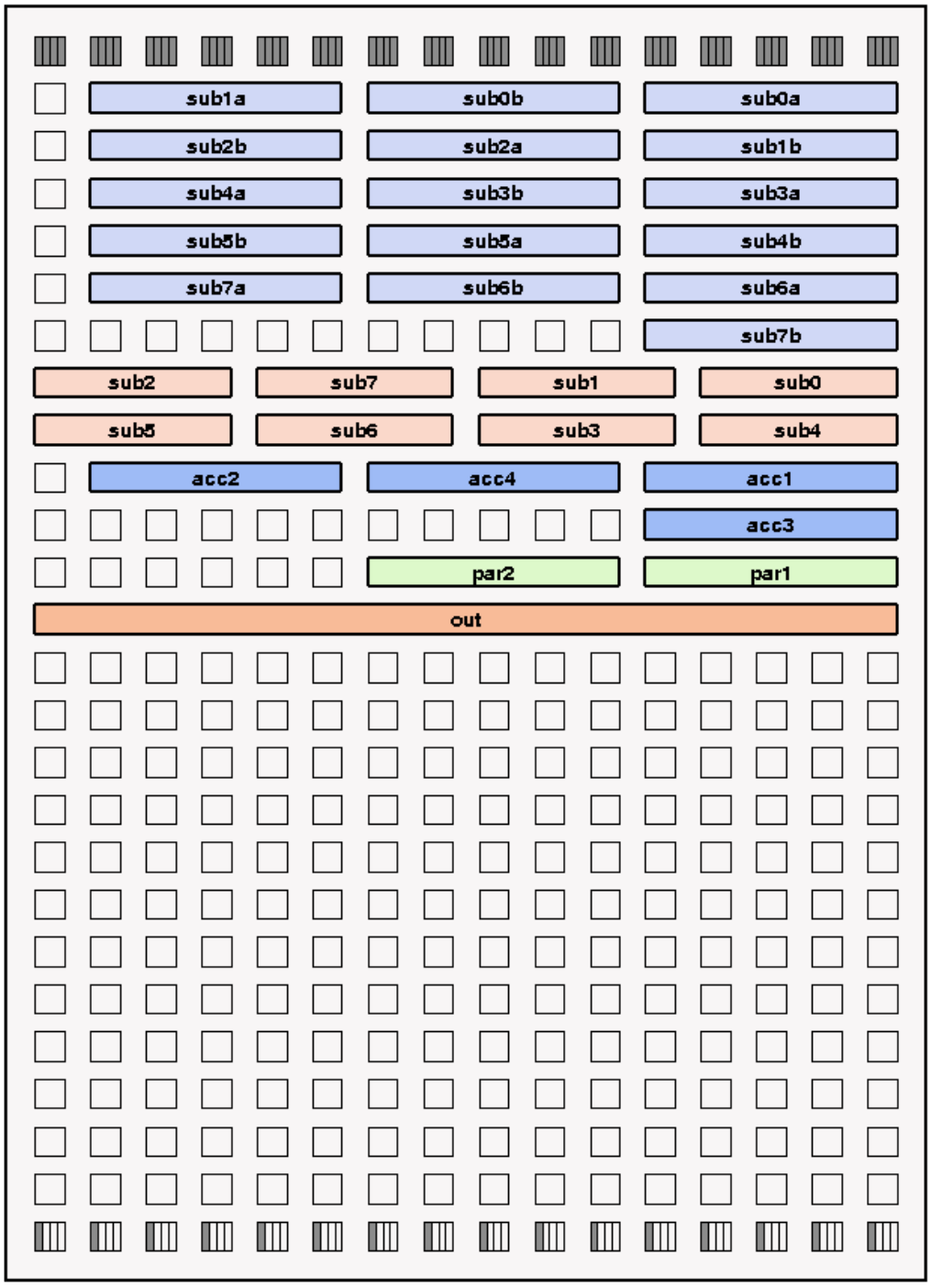

Figure 28: Example of PGAOP mapping on PiCoGA

Reconfigurable Logic Cell Architecture

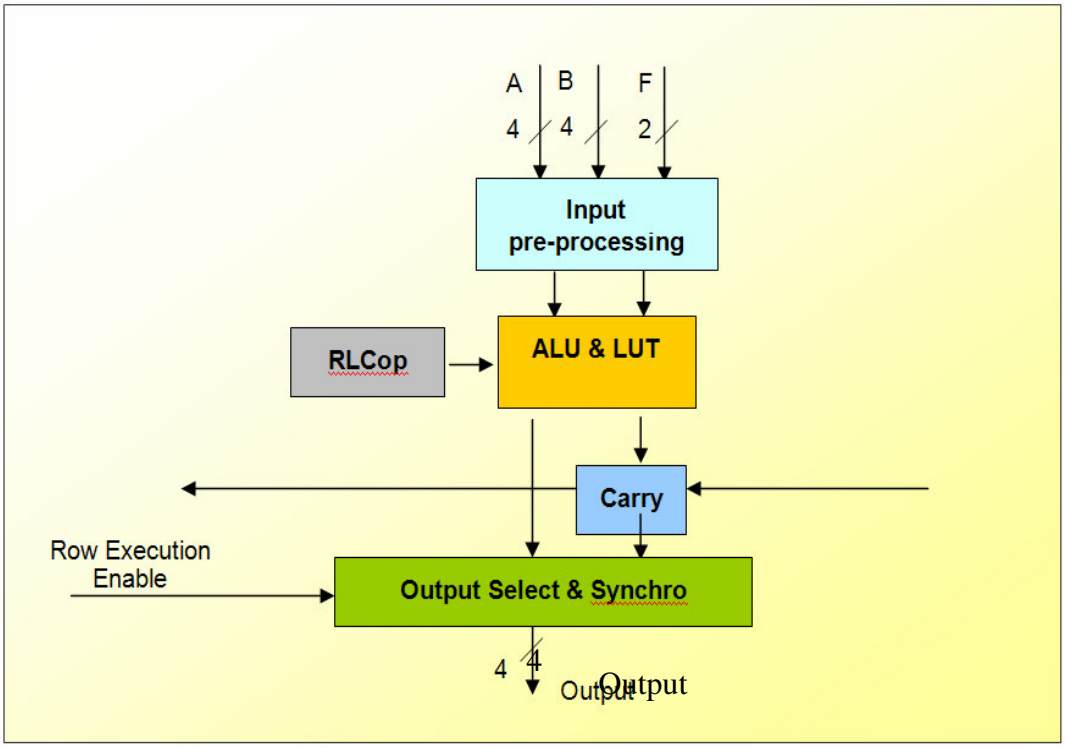

Figure 29: Reconfigurable Logic Cell: simplified architecture

The internal architecture of the Reconfigurable Logic Cell is depicted in Figure 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 RLC-op 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 4/5/6-bit inputs
    - 2-bit outputs 4/5-bit inputs
    - 4-bit outputs 4-bit inputs
    - a couple of independent lookup tables featuring:

      1-bit output/4-bit inputs or 2-bit outputs/4-bit inputs
- Up to 256-bit configurable memory module. Each configuration context provides 64-bit LUTs (see 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; more in detail, it is a multiplier module with 10-bit (in case of A * B. 6 bit are for the operand A and 4 bit for the operand B) of inputs and 5-bit output, including 12 Carry Select Adder and specifically designed to efficiently implement small/medium multiplier on PiCoGA resource.
- 4-bit Galois Field Multiplier – GF(2^4)

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 Figure 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.

PiCoGA Control Unit

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 completion and on each successive node 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 writeback 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.

30 shows a possible pipelined data-flow graph and the corresponding simplified control unit configuration.

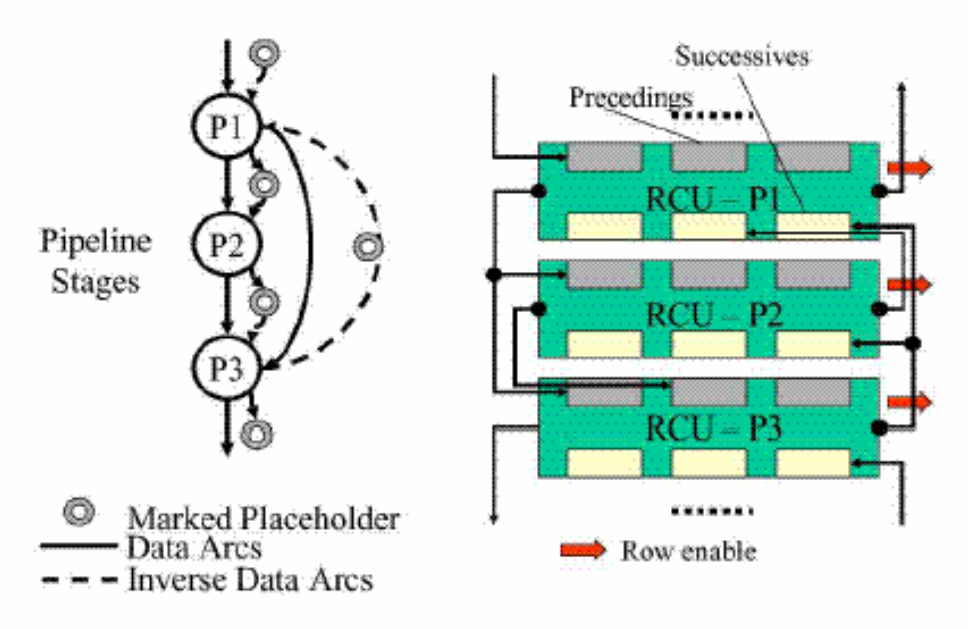

Figure 30: Pipeline management using RCUs

2.2.2.2 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 a 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, behavioural 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 Figure 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;*

Figure 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 is registers whose value is 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 PiCoGA directives are used to retarget the compiling flow from standard assembly code to the reconfigurable device.

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.

2.2.2.3 Validation of PGAOPs

The functional validation of a PGAOP is carried out in a standard C environment. The functional validation allows the user to debug a PGAOP in order to verify the correctness of the code. The PGAOP, described as usual 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).

2.2.3 Embedded FPGA

This section describes the different features and architectural options for the fine grained eFPGA block of the MORPHEUS SoC. This makes it possible to choose different architectural options for the MORPHEUS SoC design.

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.

2.2.3.1 Overview of the FlexEOS product

A FlexEOS macro is an 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.

2.2.3.2 FlexEOS Macro block diagram

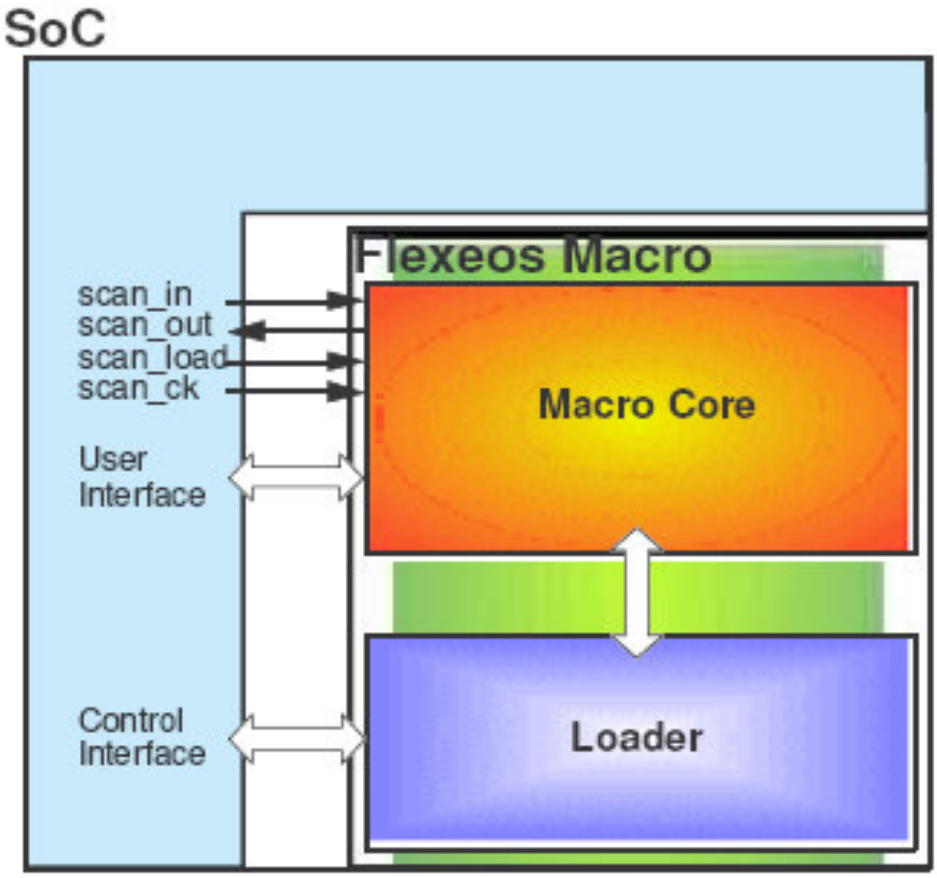

Figure 32: FlexEOS macro block diagram

Figure 32 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 Figure 32 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.

Loader

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 delivered 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.

2.2.3.3 Architecture

FlexEOS uses a highly scalable architecture which permits gate capacities from a few thousands to multiple millions.

A possible option for the MORPHEUS SoC is the FlexEOS 4K macro which includes 4096 MFCs (Multi-Function logic Cells). Furthermore, it can also optionally include the following:

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

The MFC

The basic FlexEOS building block is the MFC which is a programmable structure with 7 inputs and 1 output. It combines a 4 input LUT (Look-Up Table) and a D flip-flop (see Figure 33).

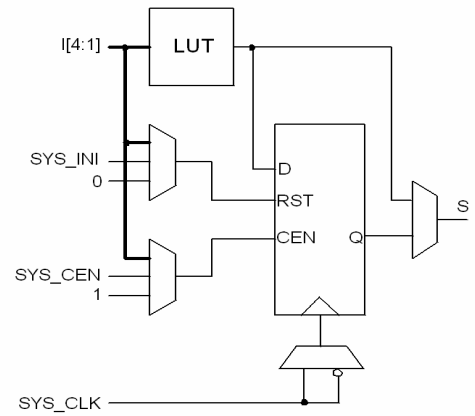

Figure 33: MFC schematic

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 by groups of 16 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.

Carry-chain block

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 1 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.

The embedded DPRAM

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

- 4K-bit block
- 8K-bit block

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** 4 for the different options depending on the memory block size).

Each port can be independently clocked and independently controlled. They can be configured as shown in

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 4: eDRAM size and configuration options

The MAC

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.

As shown in Figure 34, 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.

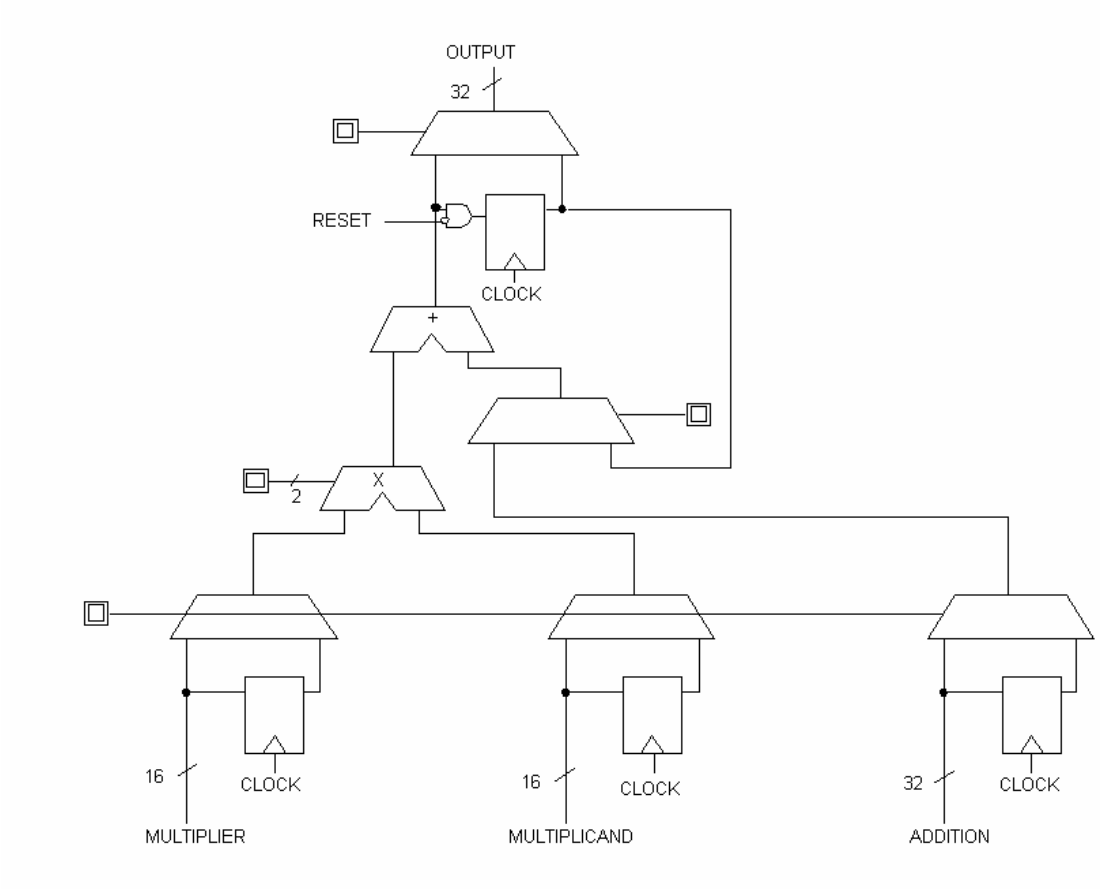

Figure 34: MAC schematic

Interconnect network

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 Figure 35), 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.

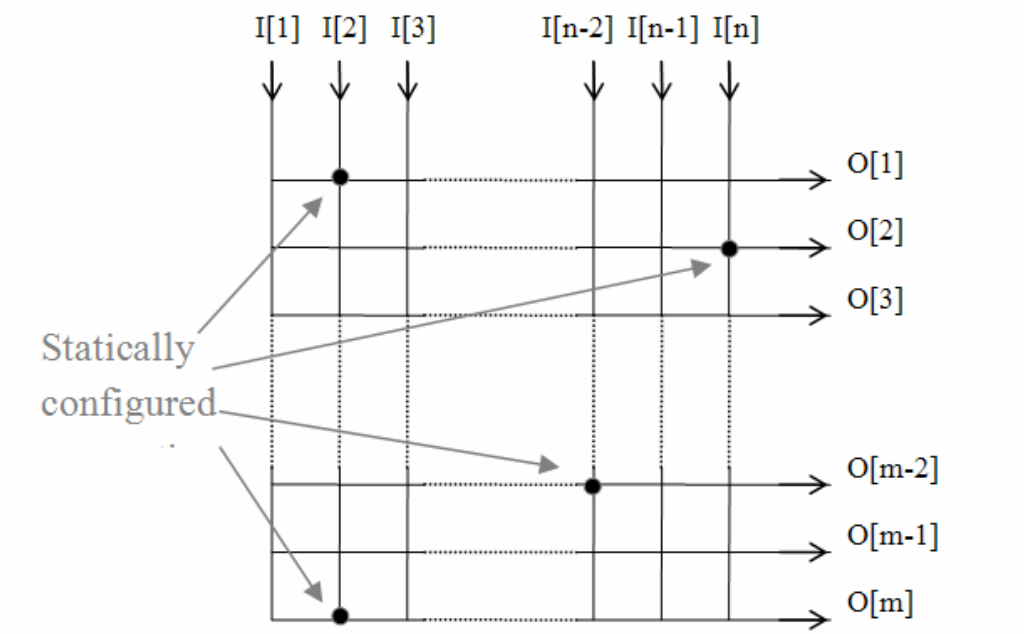

Figure 35: Full crossbar switch

Figure 36 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.

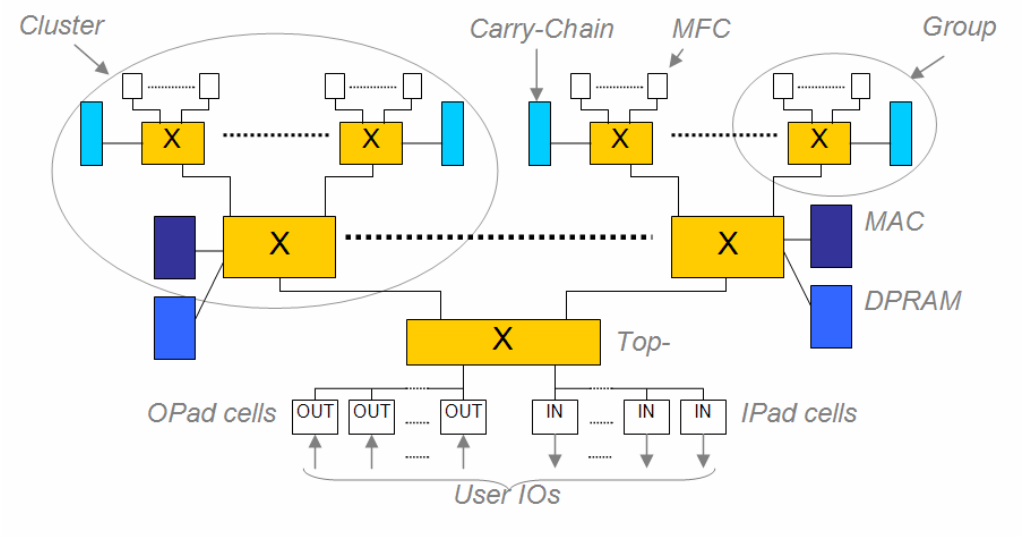

Figure 36: FlexEOS core architecture

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 fanout 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.

User I/O Interface

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.

Size and Technology

Table 5 shows the dimensions of a 4K FlexEOS macro in 90nm CMOS technology with 7 metal layers.

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 5: FlexEOS 4K-MFC features and size

Table 6 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.

	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

Table 6: Example of design mapping results

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.

Chapter 3 Memory Subsystem Definition

The MORPHEUS paradigm lies in assembling of three heterogeneous reconfigurable engines (HREs) under the control of a general-purpose processor (see 1.4). HREs serve for the intensive computation of data, in order to obtain high performance with different kind of computations for different application (e.g. with different granularities). As a consequence, the integration of heterogeneous engines in the same system allows the designer to cover a large number of different software tasks as the ones required in the context of this project.

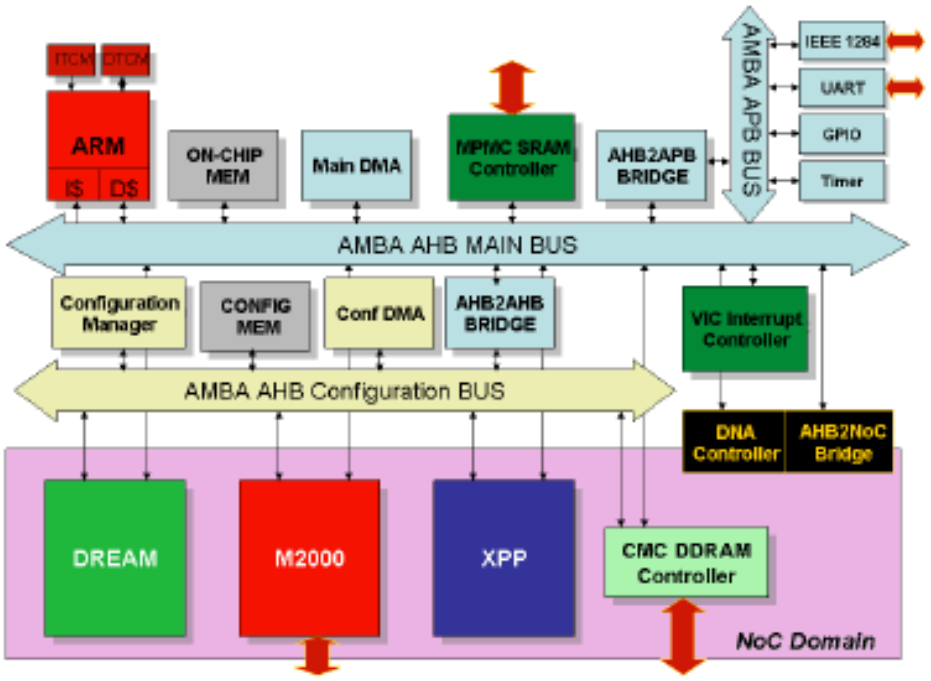

Figure 37: MORPHEUS SoC architecture

Alongside with the intensive computational data flow, the HREs are designed to support a dynamic reconfiguration. This feature is proposed in the today's architectures in order to increase the computational power of the system. Usually, reconfigurations are performed concurrently with

computations in order to mask reconfiguration overhead as much as possible (a sort of configuration pre-fetching), and configuration bandwidth requirements are typically more relaxed with respect to the ones of the computational data. Therefore, it is essential that these two flows would never interfere among each other (concurrency requirement). For this purpose, an embedded ARM9 RISC processor realizes a system control mechanism that coordinates the functioning of the HREs. ARM9 assigns the tasks to each HREs, handles the interrupts, distributes the data among the components, and so on. Task triggering is performed according to Molen paradigm by means of specific exchange registers (XRs). Moreover, a dedicated state-of-the-art interrupt controller (IC) provides a general interrupt handling from both the HREs and the other system components. State-of-the-art programmable DMAs help the efficient data distribution from the ARM9 to the remaining system. Since the dynamic reconfiguration might impose a significant performance demand for the ARM9 processor, an additional dedicated configuration manager (CM) can be used. Whereas the ARM9 processor usually controls both the computation data flows and the triggering of HREs, CM handles the configuration data flow. ARM9 processor still keeps the possibility of directly controlling the configuration process for testing purposes.

Off-chip communication is realized by means of a state-of-the-art multi-port memory controller (MPMC), supporting most of the fastest today's external memory devices (Flash, SRAM, DRAM, DDR, etc.) and several parallel data channels for higher data parallelization. For even higher data rates, a custom DDR-SDRAM controller was designed to access SDRAM and bypass the slower MPMC. Thanks to this flexibility provided by both controllers, they can be reprogrammed for different functional scenarios, like that ones proposed for the multi-purposing of MORPHEUS.

On-chip communication among computation islands, such as HREs, requires exchanging streams of data under weighty bandwidth requirements, usually dependent on the set of tasks and/or application kernels. Therefore,

MORPHEUS architecture includes a very flexible and powerful interconnection infrastructure, based on the combination of the state-of-the-art AMBA bus (for testing and basic controls) with a network-on-chip (NoC), providing to the system the necessary modularity and throughput required. In order to reduce/avoid any eventual communication bottlenecks among the different data flows, it is important to separate them physically from each other in such a way that computational and control data share one AHB bus, while configuration data are routed to the system using dedicated path. In this way, the interconnection infrastructure is designed to provide an optimal communication strategy between the main storage subsystem and the computational engines, as well as among the different computational engines. Each computational engine (or HRE) is provided of local storage, as the Data Exchange Buffer (DEB, see Section 3.3), that can be reused for the local computation (if required) in order to maximize the data locality and to reduce the overall system traffic.

Data flows	Functional units	Storage units	Communication means
Computational	HREs, ARM, main DMA, MPMC	DEBs, main on-chip memory, external memory	NoC, main AMBA AHB (for testability)
Configuration	HREs, CM, ARM, configuration DMA, MPMC	CEBs, configuration on-chip memory, external memory	Configuration AMBA AHB
Control	ARM, IC, XRs	main on-chip memory, external memory	Main AMBA AHB
External	MPMC, ARM, CM, DMAs	main on-chip memory, configuration on-chip memory, external memory	AMBA AHBs, NoC

Table 7: Distribution of data flows inside MORPHEUS architecture.

A hierarchical structure of the memory subsystem can be outlined by means of the main role played by the memory in the overall system. Three main memory levels can be identified:

- Level 3: Off-chip memory. It provides the mass storage subsystem, shared by all the components
- Level 2: On-chip memory. This memory layer include the memory storage required for the ARM9 code, the RTOS, and can be also used as a temporary repository of block of data read/write from/to the off-chip memory and sent/received to/from the computational islands. On this level, configurations are strictly separated from the other data types.
- Level 1: dedicated data/configuration exchange buffer (DEB/CEB).

  Level 1 is tightly coupled with each HRE (and usually designed around the main features of each HRE) and is responsible of three main actions:

  - to store the data currently processed by the HRE
  - to store the temporary data of the HRE computation
  - to separate HRE clock domain from the system clock domain

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

3.1 Level 3: Off-Chip Memory

Off-chip memory sub-system serves as mass-storage support. 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). 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).

As already indicated in Section 2.1.6 the MORPHEUS architecture contains one ARM PrimeCell Multi-Port Memory Controller (MPMC) PL175. 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 first 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.

- chip-selects for synchronous memory and 4 chip-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.

Due to the large bandwidth requirements of the HD digital film application in development to prove the efficiency of the reference architecture a custom controller with support for large data rates well beyond that of the MPMC was developed for access to external DDR-SDRAM.

3.2 Level 2: On-Chip Memory

On-chip memory contains the programs/tasks that are currently running (or will be run early) and the respective data. In MORPHEUS, main memory acquires more significance, since, together with the data for the central processor, it can contain temporary data currently used by HREs. These data usually show bigger size and different structure if compared to the basic ARM data-set. Moreover, the main system storage is physically separated in two parts: computational/control data, and configuration data.

On-chip memory organization usually depends on target applications and in many cases defines the performance of the whole system. We can consider all applications running on reconfigurable platforms from two points of view:

- throughput intensive processing
- reconfiguration intensive processing

The kind of processing approach is taken at the end of a design space exploration phase, investigating together the platform architecture functionalities and the requirements of the target application. For that, it is

required to support both the approaches, thus, investigating the two extreme sides of software processing, in order to determine memory specifications and their respective trade-offs.

As an example of application, one can consider the implementation of HD media digital technology, since it appear as the most memory demanding with respect to the other applications, especially for the main on-chip memory subsystem. The processing of an image requires usually the application of more than one basic operator (e.g. different filtering engines). It is thus possible to choose two main approaches: in the first, one can sequentially apply each operator to the whole image or, on the contrary, one can split the whole image in chunks (or windows) applying all the required operators for each chunk and then change the chunk under elaboration. From a computational point of view, there are two main scenarios:

- In the first case, *throughput intensive*, the full image is processed by the first operator and the result is stored in an external memory. The external memory is considered, since a whole image cannot be stored internally. Then, the result of each processing is directly read from the external memory and processed by the next operator.
- In the second case, *reconfiguration intensive*, only a sub-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 as a trade-off between the number of iterations required to process the whole image by each operator and the computational power provided by each HRE and the memory storage (global/local resource) available. In this scenario, the result of each elaboration is read from the internal memory and then processed by the next operator. Only at the end of the elaboration the result is stored in the external memory.

As an example of the trade-off occurring between throughput intensive and reconfiguration intensive approaches it can be considered an algorithm where

20 successive operators have to be applied on one image and a split factor of one hundred is considered. Hence it is possible to evaluate a lower bound of on-chip data and configuration memories. Considering the image size required for grain noise reduction for HDTV the following parameters must be considered:

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

In this case, the minimum on-chip memory size required for this application under is described by the next equations:

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

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

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

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
- $\mathrm{Re}s$ - 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 bound:
    $F = 1920 \times 1080 \times 3 \times 16 bits = 95 Mbits \approx 12 MB$,
    $W = 12 MB / 100 = 120 KB$,
    $S_{data\_\min} = 120 KB \times 3 + D_{ARM} = 360 KB + D_{ARM}$, refer to (1). D_{ARM} depends on the exact software and may vary from tens to hundreds kilobytes. Area occupation around 8 mm^2.
  - Upper bound: depends on the available area.
- On-chip configuration memory:
  - Lower bound:
    $S_{conf\_\min} = 2 \times (72 KB + 64 KB + 60 KB) = 256 KB$, refer to equation (2) and D3.1, section 5.1, table 10. Area occupation around 4 mm^2.
  - Upper border: depends on the available area.

3.3 Level 1: Data/configuration exchange buffer

Internally to each separate clock island, each reconfigurable IP has visibility and access only to its own synchronization registers (XRs) for control and local memories (DEBs and CEBs) for data and configuration.

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.

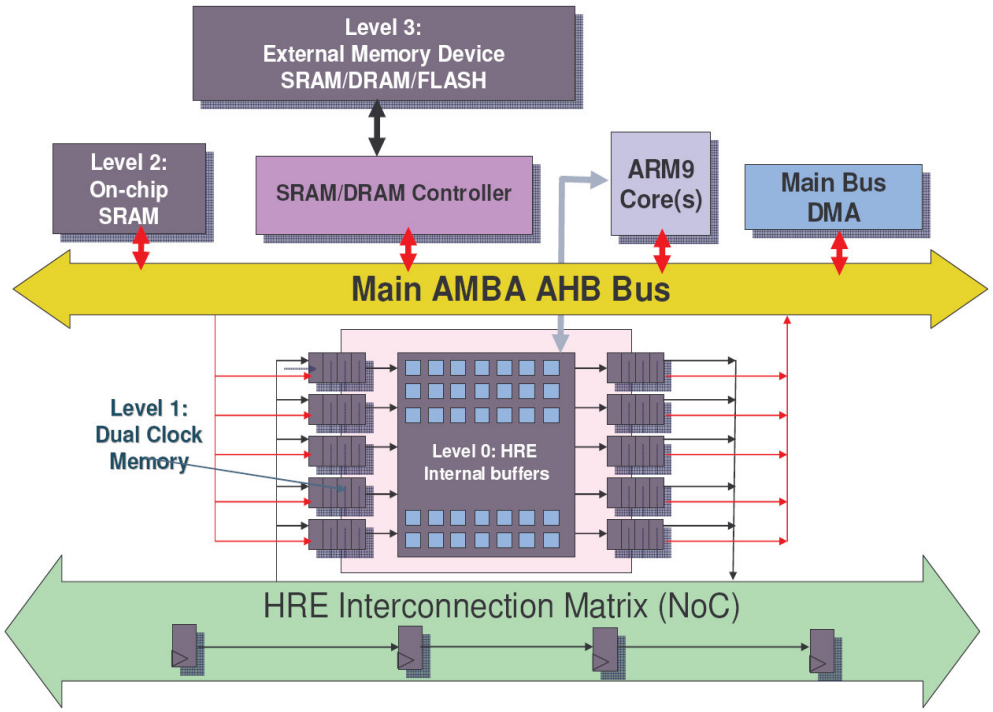

Figure 38: Data Storage Hierarchy

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.

1. *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). 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 pattern is depending on the processed data. This approach is fully explained in Section 3.3.1 there the DREAM integration scheme is presented and Section 3.3.2 where M2K is described.
2. *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.

Coming down to implementation details, this paradigm has been implemented on Morpheus in different manners for each HRE. In the next sections will be introduced the integration of each IP in the MORPHEUS SoC and in particular the interfacing to the memory sub-system. Depending on the features of the reconfigurable device a specific local memory topology and global memory connection will be considered.

3.3.1 PiCoGA integration: The DREAM Architecture

This Section describes the DREAM concept of the PiCoGA integration into MORPHEUS platform. DREAM architecture is a dynamically reconfigurable platform coupling the PiCoGA reconfigurable device with a RISC processor using a loosely-coupled co-processor scheme. A high bandwidth memory sub-system provides data communication with PiCoGA enabling high throughput and direct interface of the DREAM architecture with external computational blocks.

The DREAM is composed of three main entities: Control Unit, Reconfigurable Data Path, and Memory Access Block (see Section 2.2.2). Data transfers between DREAM and the host system are realized through exchange buffers, that also act as local repositories for data (DEBs, Data Exchange Buffers) and program code/configuration (CEBs, Configuration Exchange Buffers).

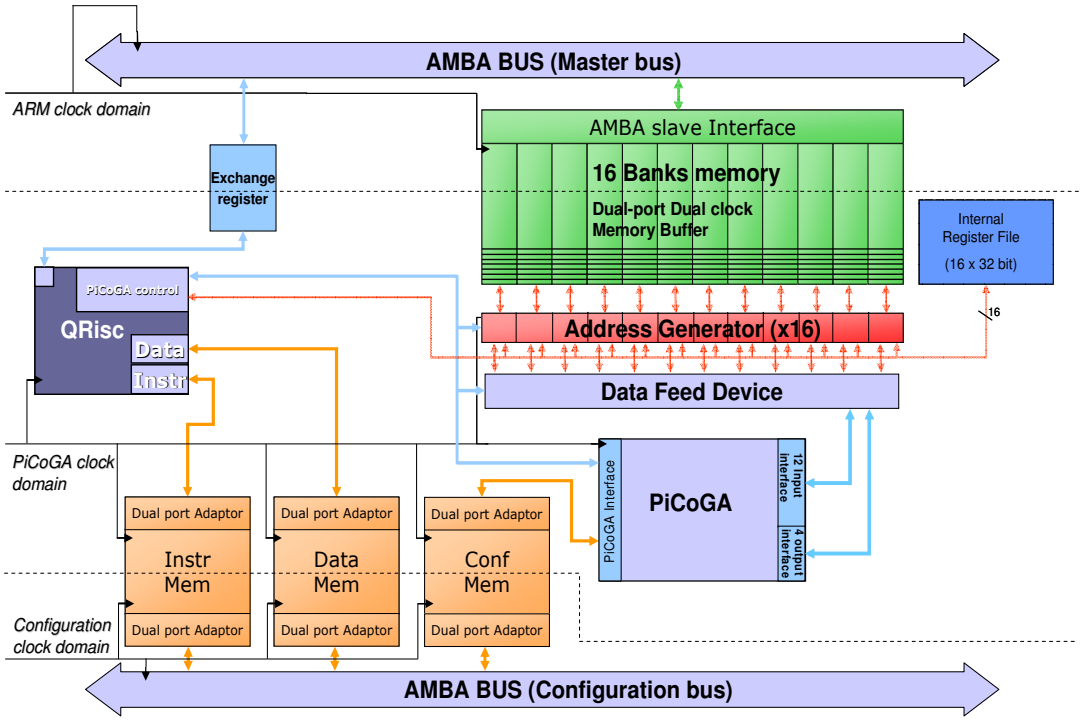

Figure 39. DREAM architecture.

3.3.1.1 Control Unit

The DREAM control unit is based on a 32-bit RISC processor, rather than a custom FSM, mainly because of the programmability, flexibility and reusability advantages provided by the state-of-the-art device. This unit is in charge for fetching instructions, handling program flow and providing appropriate control signals to the other blocks. The control signals are generated by specific coprocessor operations. Available operations that can be triggered by the control processor on the reconfigurable data path are listed in the Table 8.

DREAM Application Program Interface (API)
Set_Configuration(macro_instruction_identifier)
Unset_Configuration(macro_instruction_identifier)
Execute(macro_instruction_identifier,num_iterations)
Read/Write_Datapath_Registers(row)
Reset_Pipeline()
Wait_for_Pipeline_Empty()
Set_Interconnect_Matrix_channel(buffer,datapath_port)
Program_AG (buffer,base,step,stride,count,mask,rw)

Table 8: DREAM Application Program Interface

The processor occupies relatively small area: 20K gates of logic plus a dedicated memory module, which serves as an embedded 32-cell register file. It features arithmetical-logical operations, 32-bit shifts and a small embedded multiplier. Processor code and data, as well as the configuration bit-stream for the embedded data path are considered as part of the DREAM program code, and are loaded by the host system on the CEBs, implemented on dual port, dual clock memories. The addressing space of the control processor is configurable at design time, and in the current implementation is composed of 4KB of processor code and 4KB of data memory, plus 36KB of configuration memory for the reconfigurable data path. Input data and computation results are exchanged with the host system through a coarse-grained handshake mechanism on DEBs (also defined as ping-pong buffering).

Integration of a general-purpose processor gives to the user advantages of exploiting a sophisticated program control flow mechanism, writing commands in ANSI-C and utilizing a reliable compiler to optimize code and schedule task efficiently. 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 control processor in a fashion similar to the *Molen* paradigm, i.e. 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. Up to four macro-instructions can be loaded on each of the four available contexts. Contexts can not be computed concurrently but context switch requires only one cycle. A sophisticated stall and control mechanism ensures that only correctly configured operations can be computed on the array, and manages context switches. Besides the control functionality, the processor can also act as a computation engine in parallel with the reconfigurable data path.

3.3.1.2 Reconfigurable Data Path

The DREAM data path is composed by an array of RLCs. Each cell may compute two 4-bit inputs and provide a 4-bit result. RLC structure is described in Figure 40: it is composed of a 64-bit LUT, a 4-bit ALU, a 4-bit multiplier slice and a Galois Field multiplier over GF(2^4). A carry chain logic is provided row-wise allowing fast 8-, 16- and 32-bit arithmetic.

The ideal balancing between the need for high parallelism and the severe constraints in size and energy consumption suggested a size of 16x24 RLCs, and an IO bandwidth of 384 inputs (twelve 32-bit words) inputs and 128 outputs (four 32-bit words). This choice was mainly driven by the targeted MORPHEUS applications. The routing architecture features a 2-bit granularity, and is organized in three levels hierarchical levels: 1) global vertical lines carry only data path inputs and outputs; 2) horizontal global lines may transfer temporal signals (i.e. implementing shifts without logic occupation); and 3)

local segmented lines (three RLC per segment) handle local routing, while direct local connections are available between neighboring cells belonging to the same column.

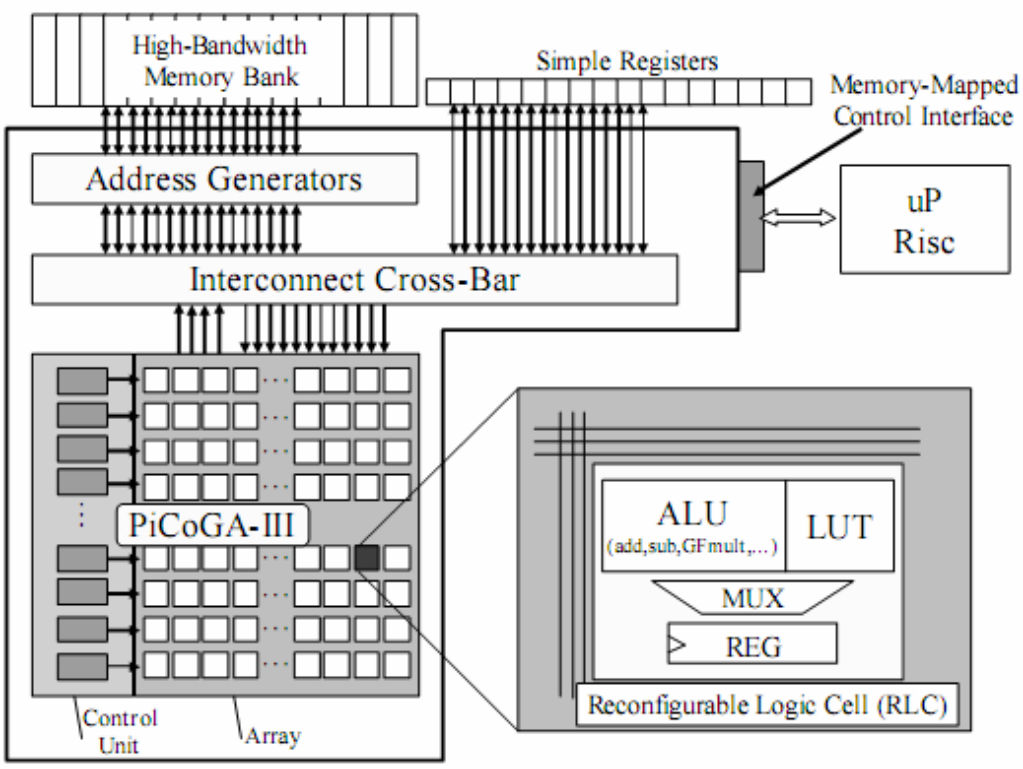

Figure 40. RLC in the DREAM reconfigurable data path.

The gate-array is coupled to an embedded programmable control unit, which provides synchronous computation enable signals to each row, or set of rows of the array, in order to provide a pipelined data-flow according to the data dependencies in the source data-flow graph. Due to its medium-grain and multi-context structure the DREAM data path provides a good trade-off between gate density (3Kgates/mm^2 per each context) and flexibility. Its deep pipelined nature allows very efficient resource utilization ratio (on average, more than 50% of the available resources per clock cycle) with respect to devices such as embedded FPGAs that need to map on reconfigurable fabrics the control logic of the algorithm. The full configuration of each context of the array is composed of 2Kb, which can be loaded in 300 clock cycles, besides each operation can be loaded and erased from the data path separately. To do

this, the reconfigurable unit is organized in 4 contexts; one context can be programmed while another is computing. An on-board configuration cache (36Kb in DREAM) and an high bandwidth configuration bus (288 bit/cycle) are used in order to hide the reconfiguration process of one context in the time consumed by computation on different contexts.

3.3.1.3 Computation and configuration data storage

In order to allow DREAM to function closer to its ideal frequency, regardless limitations imposed by the host system, dual clock embedded memory modules were chosen as physical support for DEBs and CEBs. This caused a 5% overhead in timing, 40% in area and 20% in power consumption, comparing to the single port solution. Such price is justified by the absence of multiplexing logic that would be required by the use of single port memories. This choice also implies a very straightforward physical implementation of the overall system, without need for explicit synchronization mechanisms. DEBs are composed by 16 dual port memory banks 4KB each. They are accessed as a single 32-bit memory device from the host system side and can provide parallel 16x32-bit bandwidth to/from the data path.

Due to their small granularity, DREAM macro-instructions often exchange information between successive issues, in form of temporary results or control information. For this reason a specific 16-cell multi-port register file (12 inputs, 4 outputs) was included as local data repository. As macro instructions feature variable latency, a specific hardware register locking logic was added to preserve access consistency, generating stalls to preserve the correct program flow.

Address generators

On the reconfigurable data path side, an address generator (AG) is connected to each DEB bank (see Figure 41). Addresses are incremented automatically at each cycle for all the duration of the kernel according to the programmed data pattern.

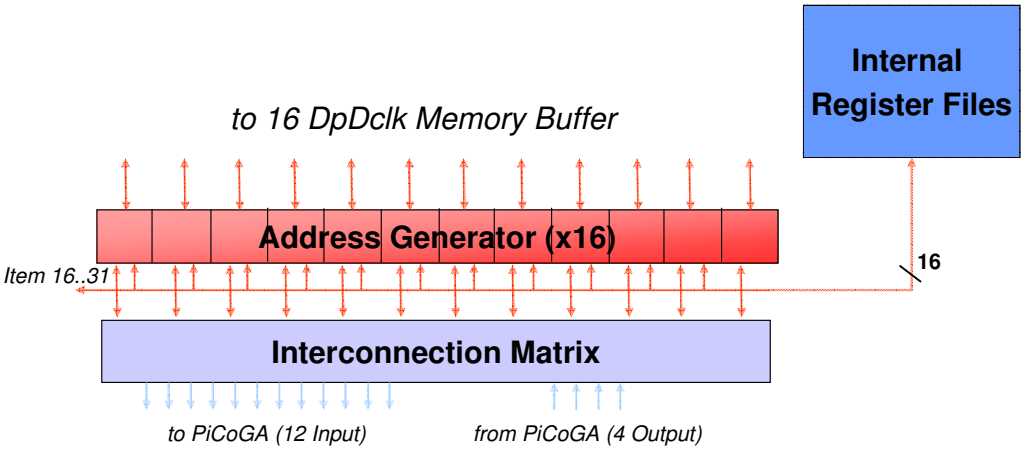

Figure 41. Integration of the address generators in DREAM architecture.

AGs provide standard STEP and STRIDE capabilities to achieve non-continuous parallel vector addressing. A specific MASK functionality allows power-of-2 modulo addressing in order to realize circular buffers of variable size with programmable start point.

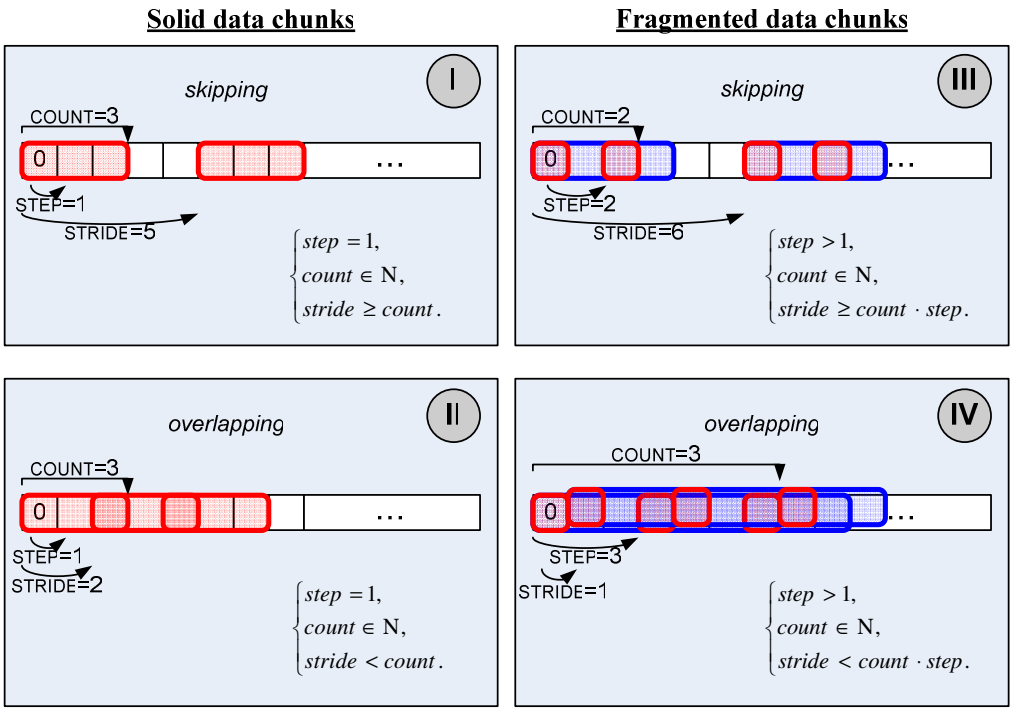

Figure 42. Classification of the available data patterns.

Address generation parameters are set by specific control instruction, enabling various types of memory access, which are classified on Figure 42. Such kind of data distribution is very often found in multimedia applications, being targeted by MORPHEUS. Thus, AGs provide stream data flow to the data path even for non-continuous vectors with regular pattern.

3.3.1.4 Integration overheads

The DREAM processor will be implemented for 90 nm CMOS technology process. The maximal operating frequency in the Morpheus context is 200MHz. The reconfigurable data path 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. Processor efficiency was measured on a set of computational kernels, oriented toward multimedia and communication applications. In particular, there we selected four highly-parallel kernels from the open-source H.264 coding standard, an OFDM Constellation Encoder or Mapper (implemented at three levels of unfolding), and well-known symmetric-key cipher AES with 128 key size. Performances were evaluated at 200MHz, and are parameterized with respect to the interleaving factor, intended as the number of data blocks concurrently elaborated. In fact, most of 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.

Figure 43 describes DREAM performance in terms of processed bits per second. For example, a single ARM-926EJ-S processor in the same technology node, according to the vendor data sheets, would provide up to 0.5 GOPS, 0.32 GOPS/mm^2, and 3.5-7.1 MOPS/mW. Neglecting overheads due to synchronization, it would thus be necessary to provide up to 60 ordinary

processors (thus much higher energy and silicon area) to match the performance delivered by DREAM on computation intensive kernels.

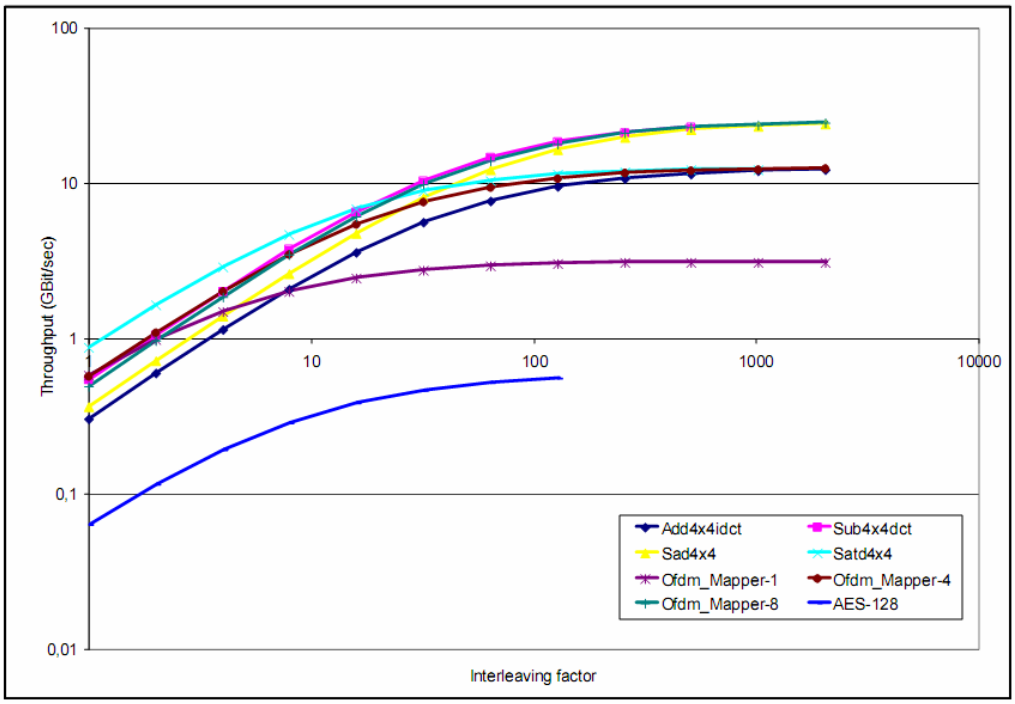

Figure 43. Throughput vs interleaving factor.

3.3.2 M2000 Integration

Figure 44 gives an overview over the M2000 sub-block in the top design.

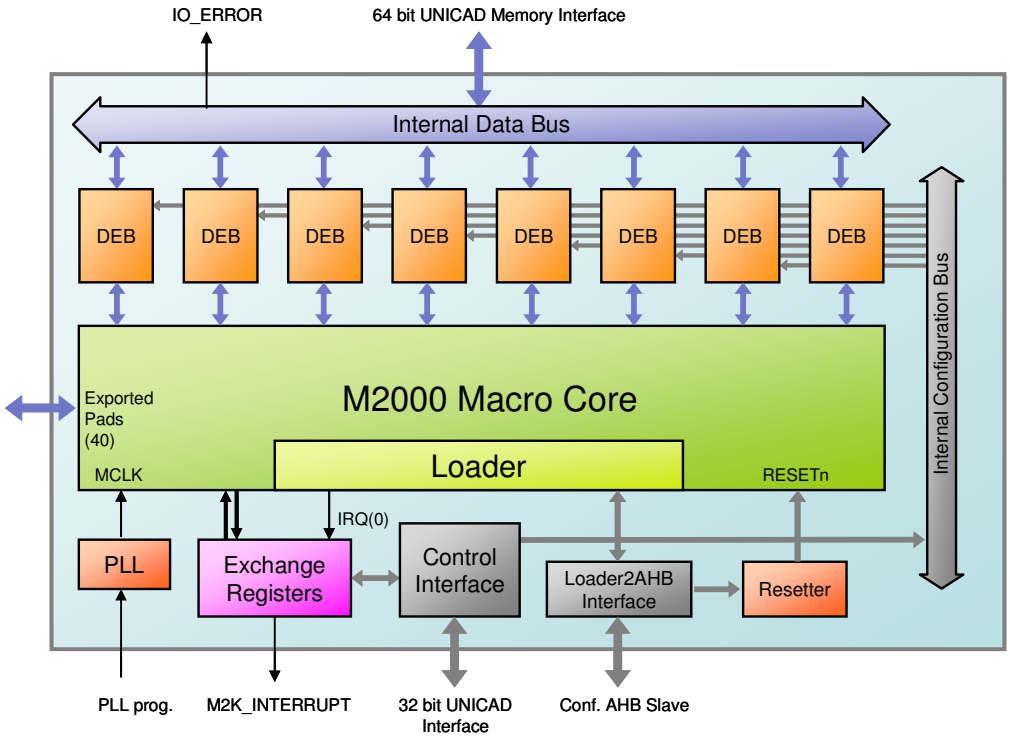

Figure 44: M2000 HRE sub-block as inserted in the top design

As shown in the figure, the sub-block is communicating with the system over several interfaces. A data interface mainly implemented via 8 32-bit dual port memories and a configuration and control interfaces wrapped via a standard AHB protocol. In the next chapters, the interconnection schemes for configuration control and data will be explained further in detail. Figure 45 lists the memory maps for the diverse interfaces of the M2000 sub-block:

- A 64-bit Data Interface that directly connect the M2000 HRE to the NoC communication engine via a dedicated port.
- A 32-bit AMBA Slave port connected to the Configuration Bus to manage the bitstream loading phase.
- A 32-bit AMBA Slave port connected to the Main Bus used by ARM to configure the Data Interconnection Interface.

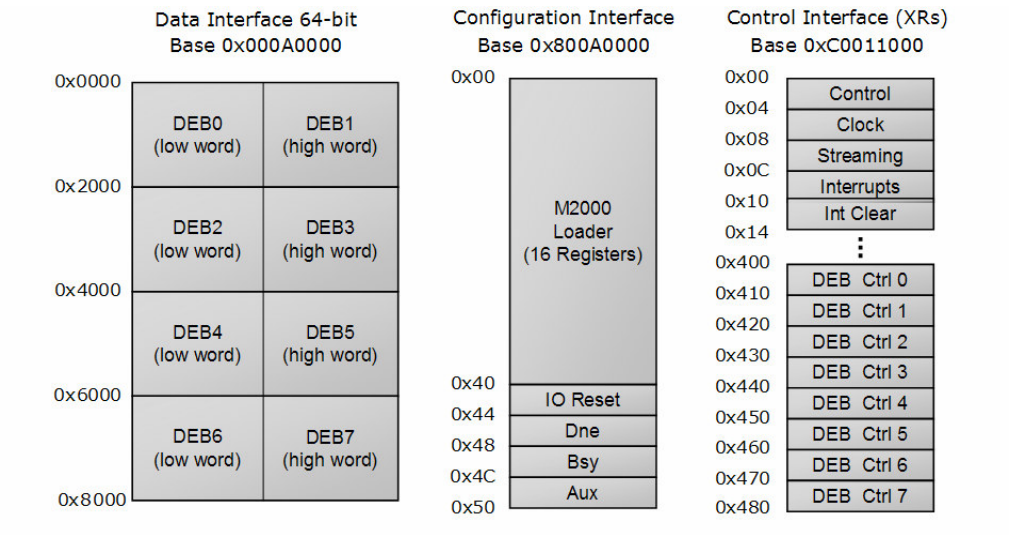

Figure 45: M2000 sub-block memory maps

3.3.2.1 Configuration Interconnection Scheme

The configuration bitstream is loaded by using the loader interface. All the loader registers can be directly accessed via the configuration AHB bus (base address 0x800A0000).

The bitstream is loaded using the following sequence[55]:

- configuring the loader in load mode by setting several parameters via a dedicated register (set Power = '1', Write Access = '1', Access Type = '00', Increment Mode = '001').
- storing the configuration words in alternation on the *data_32L* and *data_32H* registers, or in sequence on the *data_x64* register. Both can be done by using the Configuration DMA.
- An interrupt request (m2k irq 0) is set as soon as the upload is finished, but it is also possible to poll the *dne* or loader *status* register (bit 1) to detect this event.
- To verify if the bitstream has been uploaded correctly, a CRC check can be performed by storing the CRC integer provided by the FlexEOS tool into the *crc_ref* register. Upon completion of the

upload (*dne* = '1'), the *crc* bit of the *status* register (bit 2) must be '1' as well.

- The loader can now be set to run mode (set Power = '1', Run = '1', Write Access = '1', Activate Macro = '1') to start the application.

The *busy* register indicates that the loader is busy storing a configuration word, however, a wait state is inserted into the bus transfer automatically by the AHB wrapper, so the application designer does not need to take care of this.

3.3.2.2 Data Interconnection Scheme

Due to its bit-level programmability, M2000 has the greatest flexibility of all HREs inside the MORPHEUS SoC. This means that the data interface must be adaptable to a wide range of possible applications with different needs concerning the data exchange. Similar to DREAM and PACT, M2000 is connected to the chip infrastructure using a range of data exchange buffers (DEBs) based on dual-port dual-clock memories, in this case 8 cuts of 1024 32-bit words (see Figure 44). However, the DEBs also feature a programmable control logic that allows for different means of data buffering.

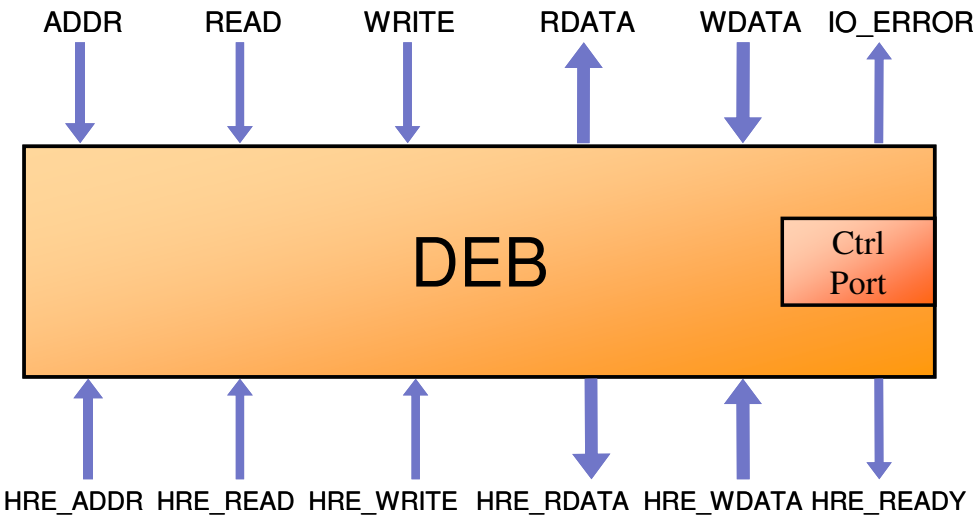

Figure 46: DEB I/O signals

The DEB interface signals of both the system and HRE side is shown in Figure 46. The HRE side ports are connected directly to the pads of the eFPGA core, while the system side ports can be driven by either the AHB bus (for debug purpose) or the NoC. From the addressing point of view, the DEBs are coupled to pairs of two to match 64-bit width of the NoC (see Figure 45).

Each DEB can be configured individually via the control port (mapped on the main bus at the address: 0xC0011000) providing access to the internal registers shown in Figure 47. The base addresses for each control port are listed in Figure 45.

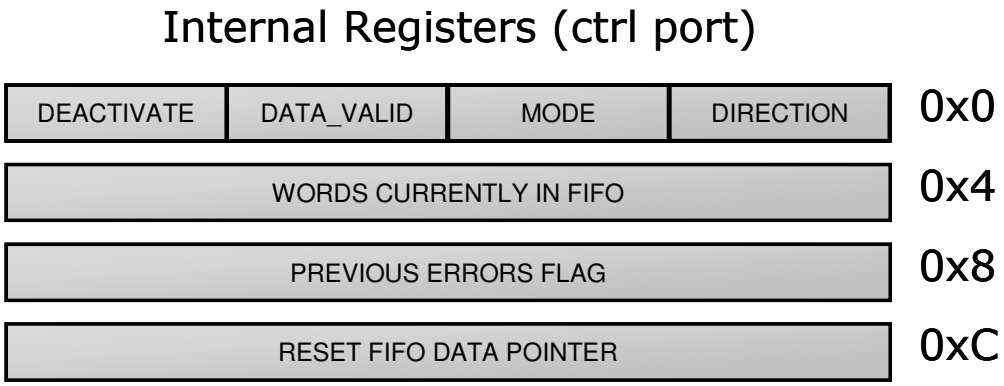

Figure 47: DEB Control Registers

Two operation modes are eligible:

1. *FIFO mode* (mode register = 0). Data streams are FIFO-buffered in the direction specified in the "direction" register. If data is buffered from the system to the HRE (direction = 1), HRE_READY indicates to the M2000 application that there is data available on the DEB, while IO_ERROR indicates push errors to the system. If data is buffered from the HRE to the system (direction = 0), HRE_READY indicates that there is space available while IO_ERROR indicates pop errors
2. *Direct addressing mode* (mode register = 1). In this case the DEB works as a normal dual port memory, either to buffer data exchange with external address generation or to serve as a local tightly

coupled memory for the application mapped on M2000. The HRE_READY signal is driven by the “data valid” register as an additional method of synchronization.

The FPGA macro core is connected to the outside world over 496 input and 512 output pads. For each one of the eight DEBs, 33 input pads and 44 output pads are assigned. 40 input and 80 output pads are used to directly drive bidirectional I/O pins of the chip, with M2000 as a reconfigurable I/O device in mind. In addition, 32 input and 31 output pads are used for interrupt and synchronization exchange registers. Clock inputs and reset signals are connected to the low-skew, low-insertion-delay buffer tree networks that start from the eight dedicated SYS inputs of the macro core.

IP[0-32]	DEB0
IP[33-65]	DEB1
IP[66-98]	DEB2
IP[99-131]	DEB3
IP[132-164]	DEB4
IP[165-197]	DEB5
IP[198-293]	DEB6
IP[231-263]	DEB7
IP[264-303]	PIN
IP[304-319]	pending irqs
IP[320-335]	arm2hre

OP[0-43]	DEB0
OP[44-87]	DEB1
OP[88-131]	DEB2
OP[132-175]	DEB3
OP[176-219]	DEB4
OP[220-263]	DEB5
OP[264-307]	DEB6
OP[308-351]	DEB7
OP[352-391]	POUT
OP[392-431]	PDIR
OP[432-446]	Irqs[1-15]
OP[447-462]	hre2arm

Figure 48: M2000 input/output pad distribution

3.3.2.3 Application Development Example

The current section describes the deployment of a simple example that can be considered a nice introduction to application deployment on the M2000 HRE.

In order to run an application into the M2K RA three main steps are required:

1. A global configuration step to activate the macro and to configure the interconnection infrastructure.
2. Program the configuration infrastructure in order lo load the bitstream and configure the M2K Macro.
3. Activate the Macro and program the data communication engine (NoC or AMBA based) to feed data in the input ports of the reconfigurable engine and read the results stored in the output ports.

Global configuration

In order to activate the macro the clock need to be activated via a store operation through the exchange register infrastructure. The reset is also deactivated and a dedicated function is executed:

```
m2k_XR->clock_mode = global_clock;
m2k_XR->resetn1 = 1;
m2k_power_on();
```

As shown in Section 3.3.2.1 the data interconnection scheme is based on 8 configurable DEBs. In the default configuration the first 4 DEBs are configured as input FIFO mode (SYSTEM → HRE) while the others as output FIFO (HRE → SYSTEM). In order to manually program such parameters a simple function is available in the drivers:

```
m2k_XR->deb[0].direction = 1; /* Input direction */
m2k_XR->deb[1].direction = 1; /* Input direction */
m2k_XR->deb[2].direction = 1; /* Input direction */
m2k_XR->deb[3].direction = 1; /* Input direction */
m2k_XR->deb[4].direction = 0; /* Output direction */
m2k_XR->deb[5].direction = 0; /* Output direction */
m2k_XR->deb[6].direction = 0; /* Output direction */
m2k_XR->deb[7].direction = 0; /* Output direction */
```

Moreover all DEBs can be programmed as standard memory deactivating the implemented FIFO controller:

```
for(i=0;i<8;i++)
(m2k_XR->deb[i]).dontusefifo = 1; */FIFO Deactivated*/
```

In this configuration the direction doesn't have any sense. DEBs in fact are implemented using a Dual Port Dual Clock memory, so parallel access can be performed form both side.

Macro configuration

In order to load the bitstream into the M2K RA a pre-built function is available. The functionality of this routine is to program the configuration DMA:

```
load_m2k_bitstream((unsigned int*)bitstream);
```

where `bitstream` is a 32 pointer of the bitstream stored in the on-chip or off-chip configuration memory.

In order to manage the loading procedure of the bitstream, the length of the bitstream is not required cause the macro is itself able to raise a DNE signal when a new configuration is fully stored and recognized. An interrupt bit on the Interrupt Register is reserved to manage the end of the loading procedure. In order to unmask such interrupt at the beginning of the configuration step a dedicated function need to be executed:

```
m2k_XR->irq_enable = 0x1; /* Interrupt 0 unmasked */
```

When the interrupt is recognized by ARM a dedicated routine is executed in order to deactivate the DMA transaction.

If necessary, all interrupt sources can be unmasked by programming the interrupt register in this way:

```
m2k_XR->irq_enable = 0xFFFF; Interrupt 0-15 unmasked */
```

Upon completion of the upload (DNE is raised), the loading procedure needs to be concluded and the application needs to be initiated. This means that the DMA transfer is stopped, the application is reset and the output pads of the eFPGA are activated. This is done by the following functions:

- m2k_conclude_confload(void);
- m2k_execute_app(void);

It might be a good idea to have those functions executed by the interrupt subroutine linked to the m2k irq 0. After this, the application is running.

Data Exchange and Synchronization

As displayed in Figure 45, the DEBs are coupled to pairs to fit the 64-bit width of the bus. Yet they these pairs consist of two independent 32-bit memories. While this is transparent as long as they, a little care must be taken in FIFO mode as both are controlled by separate FIFO controllers. To perform a push or pop operation from a single DEB, any 32-bit position inside the DEBs addressing space has to be addressed while performing a write or read access. However, it is also possible to access two FIFOs simultaneously by using a 64-bit access, but care has to be taken not to do so unintentionally.

Lets assume that the application receives four 32-bit input values from DEB0 and returns four results to DEB5:

```
/* Address low-word of DEB pair 01 */
*(volatile int*)(m2k_debs->pair01) = x0;
*(volatile int*)(m2k_debs->pair01) = x1;
*(volatile int*)(m2k_debs->pair01) = x2;
*(volatile int*)(m2k_debs->pair01) = x3;

/* wait for results to arrive on DEB5 */
while (m2k_XR->deb[5].wordsinfifo < 4) {};

/* Address high-word(!) of DEB pair 45 */
y0 = *((volatile int*)(m2k_debs->pair45)+1);
y1 = *((volatile int*)(m2k_debs->pair45)+1);
y2 = *((volatile int*)(m2k_debs->pair45)+1);
y3 = *((volatile int*)(m2k_debs->pair45)+1);
```

Of course the word count can only be read in FIFO mode, but it is also possible to use the arm2hre and hre2arm registers for synchronization:

```
/* (write some input data) */
(...)
/* set synchronization bit 5 */
m2k_XR->arm2hre |= 0x0040;

/* wait for computation to finish */
while (m2k_XR->hre2arm & 0x0040 == 0) {};

/* (read results) */
(...)
/* remove synchronization bit when input data is out of date */
m2k_XR->arm2hre &= ~0x0040;
```

On top of this, interrupts can be used for synchronization.

Chapter 4 Interconnect strategy

The aim of Morpheus is to exploit the available elaboration units (HREs-Heterogeneous Reconfigurable Engines) to provide a stream-oriented computation pattern that can be fully tailored, before and during the computation, to the requirements of the running application (or set of applications).

The interface between the user and the Morpheus hardware facilities is the ARM processor, and all hardware services are required and synchronized by software routines running on the ARM. From the hardware point of view this can be done in the same way by manual programming or RTOS.

The Morpheus programming model is based on to the Molen paradigm. Morpheus should be considered as a signal processor, where HREs are computation units providing instruction set extension, and tasks running on the HRE extensions should be seen as micro-operators of the processor. HREs are seen as the signal processor function units, Bit-streams represents the HRE instruction micro-code and the compiler's work is to schedule tasks in order to optimize the computation and ensure a familiar programming model to the user.

According to this paradigm, increasing the granularity of operators from ALU-like instructions to task running on HREs, we are forced to increase accordingly the granularity of the operands. Operands can not be any more scalar C-type operand data but become structured data chunks, referenced through their addressing pattern, be it simple (a share of the addressing space) or complex (vectorized and/or circular addressing based on multi-dimensional step/stride/mask parameters). Also operands can be of unknown or virtually infinite length, thus introducing the concept of stream-based computation.

From the architectural point of view we can then describe Morpheus handling of operands, [source, destination and temporary data] at two levels:

a) *Macro-Operand*, is the granularity handled by extension instructions, transferred by ARM and controlled by the end user through its "main" program written in C (possibly with the assistance of an RTOS). Macro-operands can be data streams, image frames, or different types of data chunks whose nature and size depends largely on the application.

b) *Micro-Operands* are the native types used in the description of the extension instruction, and tend to comply to the native data-types of the specific extension entry language that is C and GriffyC for DREAM, HDL for M2000, NML and FNCPAE-Assembly for XPP. Those micro-operands will only be handled when programming the extensions, or macro-operators, so they are meant to be handled by the user only when for optimization reason he will program manually extension operations on HREs. Otherwise will be handled by the Morpheus toolset.

Coming down to implementation details, this paradigm has been implemented on Morpheus at two levels:

4.1 Handling of micro-operands: local HRE interconnect strategy deployment

It was deemed as essential to provide locally to HREs a flexible addressing mechanism, especially for processing units that are not oriented to stream processing and make intensive use of temporary local data storage such as PiCoGA and M2000. This explains the tight interaction between the local memory hierarchy (DEB-based) and interconnect strategy. Local interconnect is the most critical aspect of data transfer and it has been given absolute

priority in the work partitioning and organization. *In fact, the aim of the local interconnect definition is to provide the abstraction between micro-operands and macro-operands thus is instrumental to the deployment of the homogeneous programming model*. The deployed strategy is three-fold, as it was considered that the different specific nature of the three HREs required specific handling of local data transfers.

1. For PiCoGA, a set of hardwired programmable address generators was added on the HRE side of the DEBs (see Section 3.3.1.3). The AGs are programmed (in C language) by the embedded processor core, whose code is part of the overall PiCoGA bit-stream. Details of the programming model for the AGs are tightly integrated with the programming model of PiCoGA itself and as described as part of the PicoGA programming. It should be noted note that this interconnect aspect can not be programmed at ARM level but is always part of the bit-stream specification.
2. For M2000, it was decided to capitalize on the HDL-oriented nature of the M2000 programming model. Address generator libraries similar to the ones described above are available, but are designed to be mapped on the M2000 fabric (see Section 3.3.2.2). Again, their specification is part of the M2000 bit-stream and is deployed as part of the application specification, where the DEB is seen as a SRAM memory macro resource “embedded” in the HRE. In alternative, it is possible to configure via ARM the DEBs in FIFO mode, and compute data in a stream-oriented pattern. Details on FIFO depth, and the number of available memory bits/fifo channel is maintained design-time configurable.
3. While DREAM is seen from Morpheus as a random access storage unit, M2000 is programmable between a RAM access and a streaming access, according to the HRE features for XPP it was

chosen to utilize pure data streaming. XPP was connected on a purely streaming pattern with four 1Kx16-bit input and four 1Kx16-bit output FIFO channels. Advanced data addressing patterns are handled internally in the XPP proprietary caching hierarchy, so similarly to the cases described above local interconnect is defined as part of the HRE programming.

4.2 Handling of Macro-operands: Global Interconnect strategy deployment

The organization of chip level data transactions is structured as a direct consequence of the theoretical approach outlined in Section XXX: the PN/KPN formalism and the Molen programming paradigm are applied to Morpheus so that applications are tackled by the ARM core instantiating a collection of coarse-grained Macro-Operators that are represented as micro-coded extension instructions whose microcode is the bit-stream itself. The handling of C-level operands and their interconnect local to the HRE, is embedded in the microcode of the instruction itself, much like the routing of single bits is embedded in the microcode of a shift or an add operation in a conventional 32-bit processor. In order to implement this pattern, macro-operands need to be referenced when calling macro-operations upon them. The flexibility in local addressing *inside* the HREs relaxes a lot the constraints on the macro-operator definition. In fact, a possible option could be to reference macro-operands only by (base_address, size) thus assuming the utilization of contiguous data chunks, leaving implementation of sophisticated addressing patterns to the HRE/DEB level. But this may not be feasible to all applications, and it is at odd with the scalability/flexibility required to Morpheus. The definition of the chip-level interconnect strategy face the following constraints:

1. Data-flow control is strongly processor-centered, HRE computation is seen as an extension of the Instruction Set of the controlling

processor. All transfers should be “initiated” by the main core, either explicitly or running a specific extension instruction. HREs may be capable to access memory independently, but in that case the HRE programmer must ensure consistency. Of course, a given transfer can be iteratively repeated to describe circular addressing and this may require only very sparse iteration with ARM.

2. The Interconnect mechanism should match the architectural scalability. One test chip is being produced, but that is only one possible instance of the architectural template. Interconnect should not be optimized on the test-chip parameters but must provide an infrastructure that can scale to different HREs configurations without compromising neither its performance nor, most important, its programming model. Also, the interconnect concept must be capable to scale between very different bandwidth requirements without changing the architectural template.
3. The Interconnect must provide a level of abstraction that matches the grain, the representation and the required flexibility of the macro-operands as described above. This level of abstraction (that represents the API towards the Morpheus toolset) must be described at C level to be fit in the Molen paradigm and Morpheus programming model.

In particular regarding point 1, it was established to limit HRE memory access only to DEBs, in order to allow the description of the HRE bit-streams as reusable macros. In order to provide a safe and consistent data transfer hierarchy, only ARM is capable to trigger transfer to and from DEBs, while HREs work on “locked” sections of the DEBs. The locking and unlocking of the DEBs is negotiated through an handshake mechanism implemented through the Exchange Registers between HREs and ARM.

4.3 Chip level Interconnect strategy deployment

The Interconnect mechanism should match the architectural scalability. One test chip is being produced, but that is only one possible instance of the architectural template. Interconnect should not be optimized on the test-chip parameters but must provide an infrastructure that can scale to different HREs configurations without compromising neither its performance nor, most important, its programming model. Also, the interconnect concept must be capable to scale between very different bandwidth requirements without changing the architectural template excluded the utilization of standard bus architectures, and suggested the utilization of a Network-on-Chip. The STNoC/Spidergon concept was adopted.

As a consequence of these design constraints the chip level Interconnect strategy was organized in the following components:

- A **communication kernel** implementing physical transfers between the interconnect nodes
- A **communication infrastructure** that is utilized to provide communication/synchronization towards the processor core (and thus the end user) and to inject/extract data to/from the communication kernel

4.4 Communication Kernel (Network-on-chip)

To ease the deployment of the Spidergon topology on the physical implementation of the Network-on-Chip ST will allow to Morpheus the utilization of the STNoC IP components, composed by the following entities:

- Router
- Target NI (Network Interface)
- Initiator NI (Network Interface)

Whenever possible the components will be distributed as pre-layouted macros. In other cases, similarly to IPs distributed by Synopsys DesignWare (such as the AMBA bus utilized in Morpheus) they are distributed through the Synopsys CoreConsultant tool that is generating gate-level net-lists. Those components are strictly confidential and functional details will not be distributed to the consortium partners.

A 10-nodes logic topology has been proposed to fit bandwidth requirement raised by the mapping of the target application on the reference design. Area issues may lead to the provision of a shrunk 8-nodes version as a possible backup solution. Of course, the NoC structure is specifically designed to hide such implementation details to architectural users, so that a consistent programming model can be developed without considering the above mentioned architectural issues. In particular, whole the number of nodes is fixed the number of routers in the topology will totally depend on the chip floor-plan and on timing analysis. This aspect is completely dependent on implementation issues but has no impact on the programming model and a marginal impact on performance.

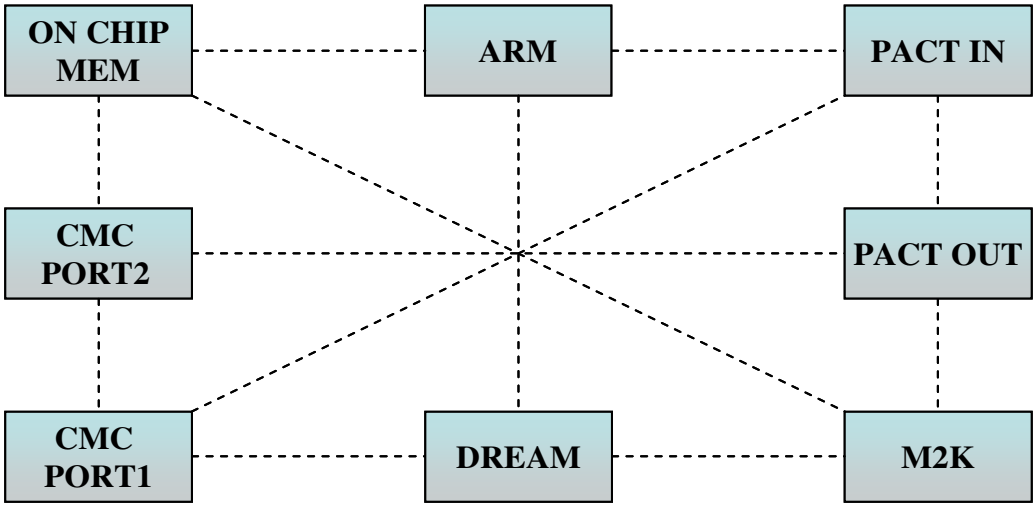

Figure 49: Proposed NoC Topology

4.4.1 Communication infrastructure

A flow of data in the Morpheus architecture can be described as a set of subsequent synchronized data transfers from IO, through the DEBs of the various HREs, possibly through on-chip memory, and finally to IO again. As described in Section 1.4, the available physical means for data transfer are

- A multi-layer AMBA bus hierarchy, that must be used for all control, synchronization and configuration of all system components but can also be used for transferring data at low bandwidth.
- A communication infrastructure based on the described 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 exploiting parallelism. For this reason every HRE Network Interface (HRE-NI) and Target Network Interface (Target-NI) also named MU-NI (Purely memory unit NI) is provided with an embedded DMA-like data transfer engine. These modifications are mostly related to the interface between memory hierarchy and Interconnect. No modification was performed on the STNoC standard components, in order to minimize risks and ensure high performance.

The user can design a given data flow according to 3 different approaches:

1. ARM can act as “full-time” traffic controller. In this case the code running on ARM monitor 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 very useful in the first stages of application deployment, to evaluate the cost of each step in the computation, maintain full programmability, and check for bottlenecks.

2. ARM can act as "batch" controller and enabler. After a "configuration" phase in which ARM configures all HREs and relative transfers on the HRE-NIs, it remains waiting for interrupts. This approach is necessary in case of a *controlled* computation network (application that requires dynamic reconfiguration to schedule different PN nodes over the same HRE) or in any case the user may prefer to deploy a PN, that is a event controlled network with respect to a KPN.
3. The deployed network can be self-synchronized: ARM 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 can be deployed 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.

4.4.2 The "load-α store-β" communication pattern

The proposed architecture required a processor-centered approach and a flexible referencing mechanism for macro-operands. In particular, spec 1 required a slight refinement of the NoC concept: a NoC is by definition a distributed communication platform with a set of initiator nodes (e.g. processor cores) issuing transfers and a set of target nodes providing information storage (e.g. memory units) and responding to the transfer request. In Morpheus, all transfers are supposed to be initiated (implicitly or explicitly) by Arm as macro-operands for a given macro-operation, much like the assembly for a standard processor is initiating transfers from the register file for an ALU operation. This is implemented through a "distributed DMA" pattern also defined "Load-α Store-β", where α, β are intended as Morpheus macro-operands (data-chunks): each HRE node Network Interface in the NoC is enhanced with a local, NoC-compatible data-transfer engine defined Local DMA. Local DMAs also provide the NI with very flexible addressing patterns that include 2D step/stride and circular buffer functionality. NIs "load" data

chunks from HREs and "store" them through the NoC to the target repository and vice-versa. From the core/user point of view the "Load-α Store-β" pattern describes the NoC as an enlarged and highly parallel DMA architecture. The user can than 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.

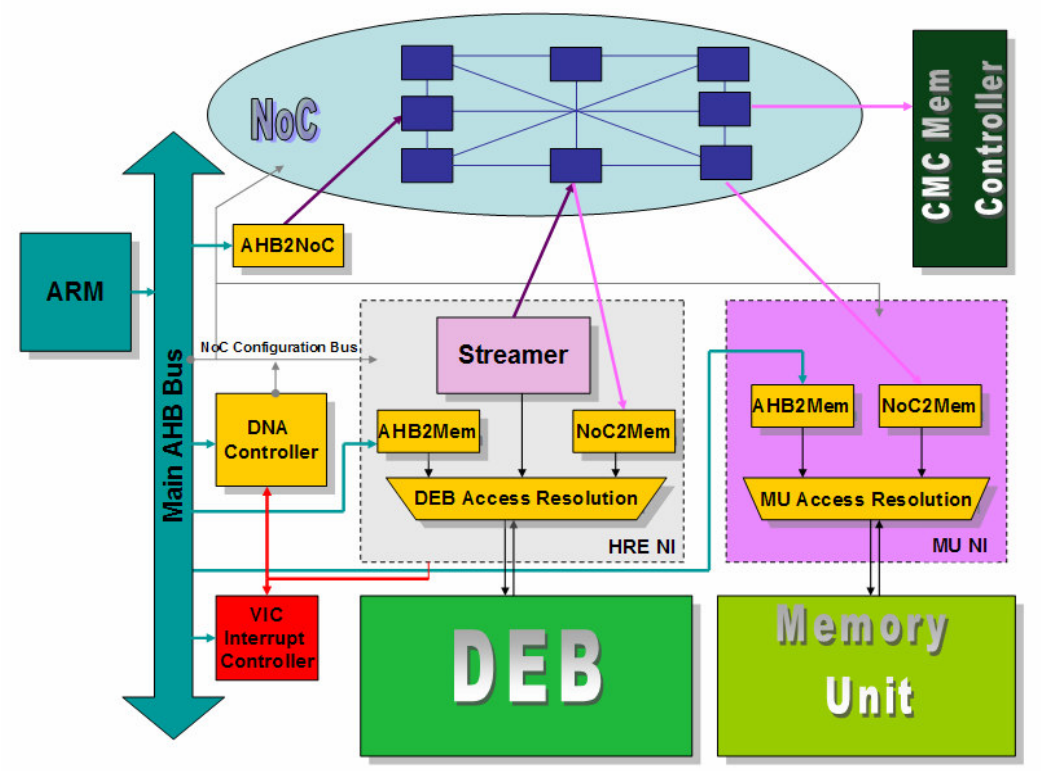

Figure 50: HRE and Target Network interfaces

4.4.2.1 The HRE Network interface

From the NoC point of view, HREs represent peculiar nodes: they should both be NoC initiators (require transfers from some storage units such as on-chip or off-chip RAM), or targets (process external requests such as a transfer request from another HRE or ARM). To deal with these design requirements, the standard Initiator interface was modified providing a "HRE-NI". This enhanced NI provides an initiator NI with the embedded Local DMA, as well

as a target NI multiplexed over the HRE DEBs (see). With this design option the ARM can require any transfer between HREs, as well as from any HRE to any storage unit (on-chip memory, Memory controller) according to the PN/KPN formalism.

Data Transfers are initiated by ARM programming specific configuration registers on the HRE network interface. This configuration is performed through on a specific NI configuration channel reaching all HRE NIs. This configuration channel is mapped as 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.

In order to increase the bandwidth and homogenize the “NoC programming model” a solution based only on store transaction is under investigation. In this scenario NOC Initiator always send data trough the NoC to a destination resource. This approach require to extend the usage of the modified HRE-NI interconnection scheme to several memory unit nodes (on-chip memories, off-chip memories) but permits to remove all the logic required to send load transaction trough the NoC without impacting the total area requirement.

4.4.2.2 The Target Network interface (MU-NI)

Similarly to what implemented for the HRE-NI, the memory unit interface has been implemented with a multiplexed solution (see) in order to manage parallel access from AMBA as well as from the NoC Target Interface.

4.4.3 Communication granularity

Bandwidth evaluations have suggested that the ideal granularity for the communication infrastructure is 64-bit. On the other hand, each HRE features a specific granularity: XPP has 16-bit IO granularity, DREAM 32-bits, while

M2000 IO granularity is not strictly related to the eFPGA structure but M2K DEBs have been fixed at 32-bits.

As a consequence, there might be some data reordering issues when organizing a stream of communication/computation requests. Of course, the communication infrastructure is capable to carry lower granularity data, but that comes at the price of a lower bandwidth. This effect can be mitigated with operands packing, but that will come at a cost. Very often, the unpacking cost is not significant on the HRE side, especially for PiCoGA and M2000, but on the processor side may become an issue especially if the data layout of inputs/outputs has specific application-related constraints. *In the communication infrastructure, data granularity information is transferred in the form of byte enable specification signals*.

The data granularity issue becomes particularly critical when dealing with FIFO oriented communication over HREs. In this case, the byte enable information must be used to trigger the FIFO read / FIFO write signals otherwise unwanted parasitic r/w operation may alter the HRE status. Another significant issue may arise when the organization of I/O buffers in the HRE do not match the NoC granularity. As an example, an application running on DREAM may require to fill only one 32-bit DEB, as the IO organization of the operation on PiCoGA is built as such. In this case, the DEB could not be filled at full granularity, and the bandwidth would necessarily decrease.

4.4.4 Chip level Interconnect strategy deployment

The Communication infrastructure is seen from ARM as a set of nodes. Through IO mapped commands (see following sections), ARM can issue transfer instances between nodes.

Each transfer features an initiator node and a target node. This definition does not describe the direction of the transfer (r/w) but the ownership: the

initiator is the entity that describes the addressing pattern and the transfer width.

The communication infrastructure is composed by:

- 3 HRE Nodes (Initiator and Target): XPP_out, M2K, DREAM. HRE Nodes can be programmed to issue transfers between any node to any other, so that these Nodes can either be transfer initiators
- 3 Memory Node wrapped as HRE-NI Nodes (Initiator and Target): On-chip memory, CMC Controller1, CMC Controller2. Target nodes can be programmed as Initiator in order to send chunk of data to a NoC Target port (all HRE input DEBS, and all memory in the system if necessary).
- 1 Initiator Node, connected to the AMBA bus. Issuing AMBA transfers, ARM or the Main AMBA DMA can initiate transfers on the communication infrastructure. This facility is only provided for test/verification
- 1 Target Node: XPP_in; This node can only be programmed as target.

There are two possible types of transfers over the communication infrastructure:

1. Data Chunk Regular Transfers
2. Single 32-bit ARM-induced transfers

Transfers of type (2) are only used for debugging/test purposes, and are performed by the user with simple IO access. The Morpheus data addressing space is replicated, so that the addresses [0X00000000] and [0x40000000] points to the same location, but in the first case accessed through the Bus hierarchy and in the second case through the NoC facilities. It is then sufficient

to trigger a bus operation in the second set of addresses to provide access ARM access through the NoC. It should be underlined that as ARM handles 32-bit data any ARM access will only utilize half of the NoC bandwidth but this is not significant for debug accesses.

Transfers of type (1) are regular Morpheus transfers utilized during peak computation. They are always triggered programming a set of control registers on each programmable HRE Network Interfaces. This programming action is performed through the specific NoC configuration bus that is mapped on the ARM addressing space (base address: 0x0xC0300000) and can be performed either by the user via ARM (Software Control) or by the DNA Controller (Stream-oriented automated control) according to a pre-defined pattern.

4.4.5 Programming NoC Transfers:

A regular NoC transfer requires an initiator and a target. Each initiator (HRE acting as Initiator and Memory unit acting as initiator) can program several write transfers from the local DEB/FIFO to any target. The HRE-NI allow the utilization of up to 2 write channels except the HRE-NI connected to the on-chip memory that is able to manage up to 4 write parallel channels. For each channel the data transfer is configured describing by the following set of parameters:

Name	Description	Address Offset	Bit Width	Reset Value
SAR	Source Address	0x000	64	0x0
DAR	Destination Address	0x008	64	0x0
CTL	Control Register	0x018	64	0x0
CFG	Configuration Register	0x040	64	0x0
SGR	Source	0x048	64	0x0

	Gather Register			
DSR	Destination Scatter Register	0x050	64	0x0

Table 9: Programming Registers for NoC Transfers

In order to program the distributed engines integrated in the NoC with an appropriate SW abstraction level, C-based drivers have been implemented. The implemented drivers support single transfers as well as multi block transfer for stream access as below:

- 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 programmed using two C structure called respectively *config* and *lli.* For each one several parameters are defined. The subsections below describe how to program the local DMA engine of an HRE-NI for a single block transfer and a multi block transfer with auto reload.

4.4.5.1 Single Block NoC Transfer

First of all several parameters of the *config* and *lli* structures need to be initialized. Source address and Destination address, as well as the programmed channel can be specified:

```
//defined variables:
struct config channel_cfg;
struct lli    channel_lli;

//part of code:
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;
```

The transfer size (defined in byte) for the selected channel is configured changing the field *ctlh* of the *lli* structure instance (*channel_lli*), representing the channel control register (CTLx[43:32]).

```
//part of code:
changeBits(&channel_lli.ctlh, BLOCK_TS, BLOCK_TS_S, 1024);
```

The burst transaction length both for source and destination ports must be defined changing the corresponding bits in the channel control register (CTLx[16:14] and CTLx[13:11]). Table 11 on page 104 of the DMA databook explains hot to set this field.

```
//part of code:
changeBits(&channel_lli.ctll,SRC_MSIZE,SRC_MSIZE_S,2);
changeBits(&channel_lli.ctll,DST_MSIZE,DST_MSIZE_S,2);
```

The transfer data width both for source and destination data must be defined changing the corresponding bits in the channel control register (CTLx[6:4] and CTLx[3:1]). Table 12 on page 105 of the DMA databook explains how to set this field.

```
//part of code:
changeBits(&channel_lli.ctll,SRC_TR_WIDTH,SRC_TR_WIDTH_S,2);
changeBits(&channel_lli.ctll,DST_TR_WIDTH,DST_TR_WIDTH_S,2);
```

In the Morpheus context, in order to increase the available bandwidth, local DMAs are generated with two master ports (two layers), one directly connected to the DEBs or memory units, while the other acts as Initiator of the NoC. Whit a multi layer configuration for each channel, source and destination layer must be defined. SMS (source master select) identifies the Master Interface layer from which the source device is accessed. DMS (destination master select) identifies the Master Interface layer from which the destination device is accessed. In the Morpheus configuration, where only write channel are used, SMS identifies always the Master interfaces connected to the local storage unit, while the DMS the Master Interface connected to the NoC Initiator port. In order to set the correct layer the channel control register (CTLx[26:25] and CTLx[24:23]) must be programmed:

```
//part of code:
changeBits(&channel_lli.ctll, SMS, SMS_S, 1);
changeBits(&channel_lli.ctll, DMS, DMS_S, 0);
```

With all configurations set, it is possible to request the transfer. The function *transfer* can be used, setting the type of transfer (SB, for simple-block transfer), the configuration structure address and the linked list item address. It returns OK if the transmission is correctly requested, or an error code (CH_BUSY, DMA_DISABLED, INVALID_DMA_NUM, UNKNOWN_TYPE).

```
//part of code:
while (e != OK)
   e = transfer(SB, &channel_cfg, &channel_lli);
```

When the block transfer has completed. Hardware disables the channel. If interrupt are activated and unmasked the DMA engine sets the block-complete interrupt and the transfer-complete interrupt. In order to activate the interrupt generation CTLx[0] must be set and the unmasking procedures need to be executed:

```
//part of code:
setBits(&channel_lli.ctll, INT_EN);
maskInt(DMA_engine_ID, I_BLOCK, 0, TRUE);
maskInt(DMA_engine_ID, I_TFR, 0, TRUE);
```

These instruction must be executed before the transfer function is called and the DMA is activated.

4.4.5.2 Auto Reload Multi Block NoC Transfer

In order to manage a streaming pattern access an Auto Reload mechanism combined with interrupt generation can be used:

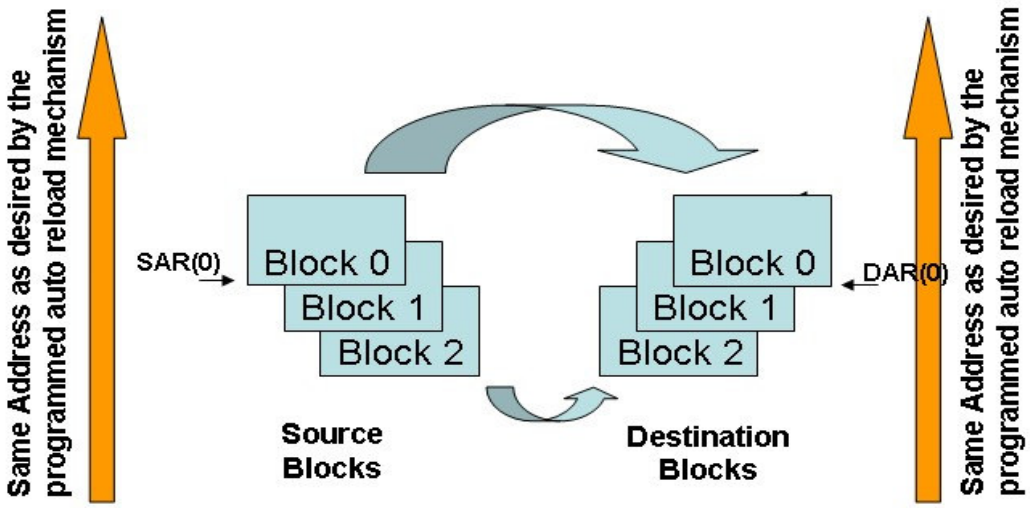

Figure 51: Multi-block Transfer with Source and Destination Address Auto-reloaded

During auto–reloading, the channel registers are reloaded with their initial values at the completion of each block. If the Contiguous Source address programming pattern is chosen only the DAR is reloaded while the SAR is contiguously incremented between sequential block. Same rules are used in the case of a Contiguous Destination address access.

As shown in the previous paragraph several parameters of the *config* and *lli* structures need to be initialized. Source address and Destination address, as well as the programmed channel can be specified:

```
//defined variables:
struct config channel_cfg;
struct lli    channel_lli;

//part of code:
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;
```

The transfer size (defined in byte) for the selected channel is configured changing the field *ctlh* of the *lli* structure instance (*channel_lli*), representing the channel control register (CTLx[43:32]).

```
//part of code:
changeBits(&channel_lli.ctlh,BLOCK_TS,BLOCK_TS_S,1024);
```

The burst transaction length both for source and destination ports must be defined changing the corresponding bits in the channel control register (CTLx[16:14] and CTLx[13:11]). Table 11 on page 104 of the DMA manual explains how to set this field.

```
//part of code:
changeBits(&channel_lli.ctll,SRC_MSIZE,SRC_MSIZE_S,2);
changeBits(&channel_lli.ctll,DST_MSIZE,DST_MSIZE_S,2);
```

The transfer data width both for source and destination data must be defined changing the corresponding bits in the channel control register (CTLx[6:4] and CTLx[3:1]). Table 12 on page 105 of the DMA manual explains how to set this field.

```
//part of code:
changeBits(&channel_lli.ctll,SRC_TR_WIDTH,SRC_TR_WIDTH_S, 2);
changeBits(&channel_lli.ctll,DST_TR_WIDTH,DST_TR_WIDTH_S, 2);
```

In the Morpheus context, in order to increase the available bandwidth, local DMAs are generated with two master ports (two layers), one directly connected to the DEBs or memory units, while the other acts as Initiator of the NoC. Whit a multi layer configuration for each channel, source and destination layer must be defined. SMS (source master select) identifies the Master Interface layer from which the source device is accessed. DMS (destination master select) identifies the Master Interface layer from which the destination device is accessed. In the Morpheus configuration, where only write channel are used, SMS identifies always the Master interfaces connected to the local storage unit, while the DMS the Master Interface connected to the NoC Initiator port. In order to set the correct layer the channel control register (CTLx[26:25] and CTLx[24:23]) must be programmed:

```
//part of code:
changeBits(&channel_lli.ctll, SMS, SMS_S, 1);
changeBits(&channel_lli.ctll, DMS, DMS_S, 0);
```

With all configurations set, it is possible to request the transfer. The function *transfer* can be used, setting the type of transfer (SB, for simple-block transfer), the configuration structure address and the linked list item address. It returns OK if the transmission is correctly requested, or an error code (CH_BUSY, DMA_DISABLED, INVALID_DMA_NUM, UNKNOWN_TYPE).

```
//part of code:
while (e != OK)
e = transfer(AR_MB, &channel_cfg, &channel_lli);
```

When the block transfer has completed, the local DMA reloads the SARx, DARx, and CFGx registers. If interrupts are enabled, which can be done by setting bit zero of the CFG register to '1', and the block-complete interrupt is un-masked hardware sets the block-complete interrupt when the block transfer has completed. It then stalls until the block-complete interrupt is cleared by software. If interrupts are disabled or the block-complete interrupt is masked (the MASKBLOCK[channel] = '1'), then 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.

4.4.6 Programming NoC Space address

The figure below represent the NoC programming space address.

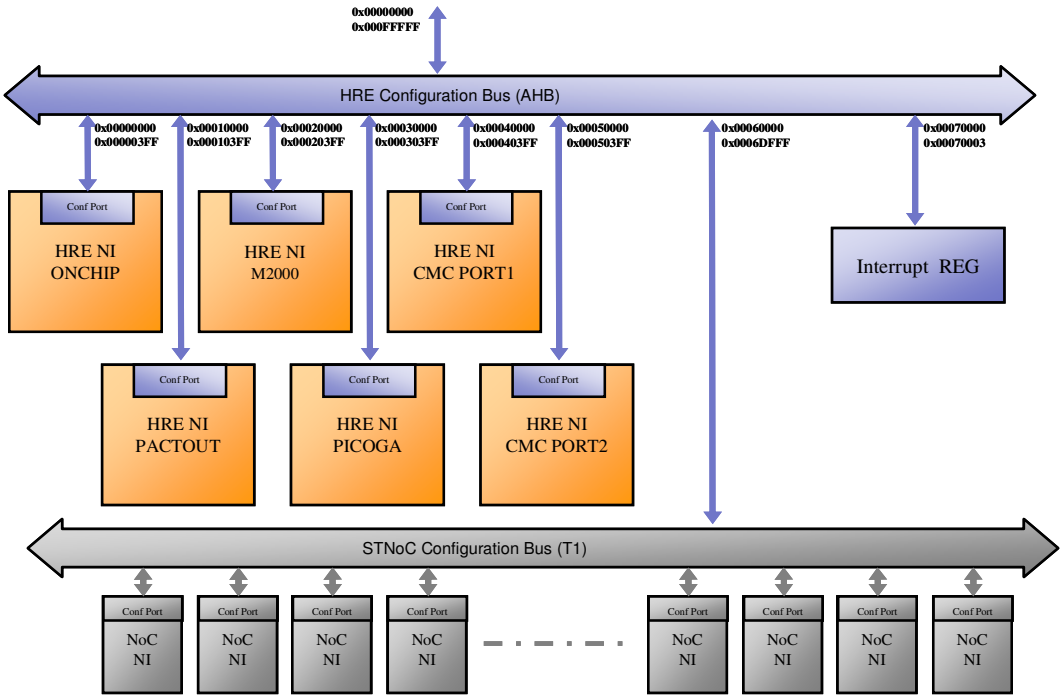

Figure 52: NoC Programming Space address

Each local DMA has a dedicated 128 byte space address. A dedicated STBUS T1 is directly connected to the HRE Configuration bus via a AMBA to T1 Bridge to set during the boot procedure some configurable parameters for each standard Network Interface.

In order to have a global accessible pointer to the interrupt status of the NoC an AMBA mapped 32-bit Interrupt register has been Implemented. Alternatively all the interrupt sources connected to this register are directly exported to the NoC Top in order to allow the designer to connect each source to a dedicated engine as a standard interrupt controller. Table 10: Interrupt Register connection scheme shows how all interrupt sources are connected to the Global NoC Interrupt Register.

Bit	Details
[0 .. 3]	On-Chip MEM local DMA: Transfer channel Interrupt [ch0 .. ch3]
[4 .. 7]	On-Chip MEM local DMA: Block channel Interrupt [ch0 .. ch3]
[8 .. 9]	PACT Out local DMA: Transfer channel Interrupt [ch0 .. ch1]
[10 .. 11]	PACT Out local DMA: Block channel Interrupt [ch0 .. ch1]

[12 .. 13]	M2K local DMA: Transfer channel Interrupt [ch0 .. ch1]
[14 .. 15]	M2K local DMA: Block channel Interrupt [ch0 .. ch1]
[16 .. 17]	DREAM local DMA: Transfer channel Interrupt [ch0 .. ch1]
[18 .. 19]	DREAM local DMA: Block channel Interrupt [ch0 .. ch1]
[20 .. 21]	CMC Port 1 local DMA: Transfer channel Interrupt [ch0 .. ch1]
[22 .. 23]	CMC Port 1 local DMA: Block channel Interrupt [ch0 .. ch1]
[24 .. 25]	CMC Port 2 local DMA: Transfer channel Interrupt [ch0 .. ch1]
[26 .. 27]	CMC Port 2 local DMA: Block channel Interrupt [ch0 .. ch1]

Table 10: Interrupt Register connection scheme

4.5 Results and Bandwidth Estimation

In this Charter several results of the NoC-based interconnection engine will be presented in order to show how the implementation choose of integrating the STNoC in the context of this design perfectly match with the application requirements of the target applications of the project. The main scope of this section is to introduce a quantitative study of the achieved bandwidth for several data transfer paths between HREs (HRE DEB to HRE DEB) and between HREs and on-chip/off-chip memories (HRE DEB to MEM). In order to have a complete overview of the performance of the communication engine a detailed analysis has been done considering data chunk of different size, starting from very small block of 64 bit to bigger block of 4KByte.

To activate a data transaction in the NoC a first programming phase is required to instruct the DMAs engine with the basic information required to control the transaction. Optionally a zero-overhead initialization phase to feed the memory is also necessary in order to verify the correctness of the transaction itself.

In order to validate the proposed approach several applications were investigated, as shown in Table 11:

- OUT-K frame processing, used for network routing application
- IEEE 802.11j, a well known wireless telecommunication protocol
- A Motion Detection algorithm used in High Definition Television protocols

Their dataflows were mapped on the described architecture, considering to implement critical kernels in the most appropriate RA. Each column of Table 11 represents the total bandwidth required for each physical link.

Apps	M2K/ OnChip Ram	DREAM/ OnChip RAM	M2K/ DDRAM	XXP/ DDRAM	DREAM/ DDRAM	XPP/ DREAM	XPP/ M2K	Dream/ M2K
OUT-K Frame	10Mb/s	10Mb/s						
IEEE 802.11j		312Mb/s	7Mb/s			288Mb/s	390Mb/s	24Mb/s
Motion Detection		124Mb/s		3.34Gb/s	1.73Gb/s			

Table 11: Application Bandwidth Requirements

4.5.1 Bandwidth Analysis

To analyze the achieved bandwidth the interrupt controller has been deactivated. In fact in this context the usage of an interrupt routine to trigger several data transfer introduce an overhead that cannot be attribute to the communication engine but to the Operative System. For our simulation a simple polling procedure has been implemented thanks to the integration fo a global interrupt register (see Table 1). In order to initialize all the memories of the system two different approach are possible:

1. All the data memories are connected to the NoC, and an Initiator test port has been implemented and connected to ARM in order to have a centralized test interface during the test chip phase. Thanks to this interface (see Figure 49) ARM is able to access each memories using the global space address shifted of a fixed quantity (0x4000 0000).
2. A second test mechanism has been implemented based on AMBA. Each modified NI in fact contains a bus-based bridge in order to connect each memory directly to the NoC and to the main system bus (see Figure 50).

The bandwidth estimation has been done programming several transfers single-channel and single-block. Each DMA engine integrated int the modified Initiator NI in fact can support up 2 concurrent channel (4 in the case of the on-chip memory NI) and each channel can manage single or multi block transfer.

In the case of a multi-block transfer the DMA automatically restarts the transfer of a second data chunk when the first is finished recalculating the source and destination address with different pattern based on the kind of parameter used to set the multi-block transfer. In this context 11 significant transfers has been analyzed:

- 8 write transfers (an Initiator trigger a write request in the NoC)
- 3 read transfers (an Initiator trigger a read request in the NoC)

In order to have a consistent number of bandwidth and analyze the achieved bandwidth I ripest of the data chunk size each transaction has been repeated several times with different data chunk size (8, 16, 32, 64, 1024, 2048, 4096 Byte). Table 12 summarized the achieved result for different pattern:

Source	Dest.	WR/RD	8	16	32	64	1024	2048	4096
M2K DEB	OnChip RAM	write	250	500	1000	1941	3385	4826	5872
BREAM DEB	OnChip RAM	write	350	696	1356	2712	3828	5260	6057
M2K DEB	DDRAM	write	365	731	1463	2438	4196	5277	6431
PACTIN DEB	DDRAM	write	345	689	1379	2758	3986	4728	6332
DREAM DEB	DDRAM	write	353	706	1412	2824	4055	5185	6326
PACTIN DEB	BREAM DEB	write	346	692	1384	2767	3996	5136	6332
PACTOUT DEB	M2K DEB	write	365	731	1463	2438	4196	5277	6425
DREAM DEB	M2K DEB	write	349	699	1398	2347	4035	5168	6314
PACTIN DEB	BREAM DEB	read	328	656	1113	1699	1988	2351	2529
PACTOUT DEB	M2K DEB	read	349	699	1174	1769	2371	2598	2859
DREAM DEB	M2K DEB	read	338	676	1135	1725	2330	2574	2849

Table 12: Bandwidth estimation (MB/s)

In order to better analyze this table it is necessary to take in account that the data base of the NoC is 64-bit and the operative condition should be 200 MHz. In an ideal scenario, where no programming step are required, o clock latency transfer, no traffic in the NoC, a maximum bandwidth of 12 Gb/s can be reached. Table 12 shows that programming the NoC to work with very small data chunk the achieved bandwidth is significantly reduced cause the overhead introduced by the programming phase cannot be neglected. Increasing the total

size of the transaction up to 4KByte it is possible to reach a maximum bandwidth of 6,5 Gb/s that is more than the half of the maximum ideal bandwidth (these numbers take in account of the possible traffic and conflicts that occurs in the NoC during the transfer of several blocks).

On the contrary best case scenarios occur if no interleaving of traffic is happening. Here one router is only responsible for one request. In the same way, one link is only used by one transfer. For simulation, the network has been set up with a source – destination distance of one (one hop) that means that the source node wants to send its data to the memory that is connected to the neighbor router. Also, only one channel is occupied, so that no interfering requests derange the results. Therefore, the bandwidth that is obtained here, is the maximal one that is possible in any case. Figure 53 illustrates the result of this simulation. The x-axis denotes the number of packets, where the y-axis shows the index of the packet size (form 2 to 64 Bytes).

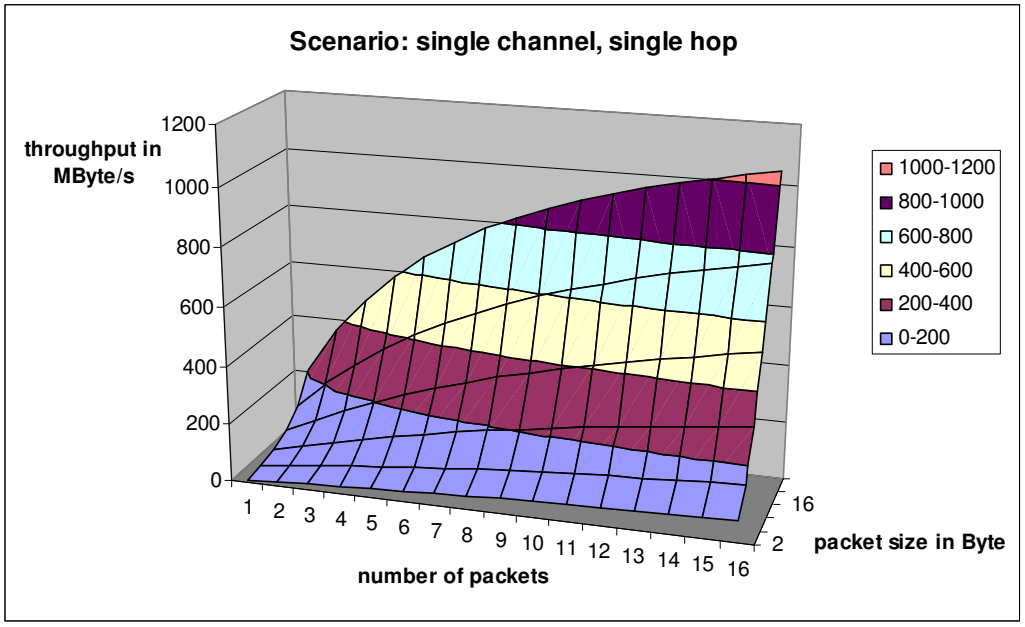

Figure 53: Throughput in MByte/s in a best case scenario

It can be seen from Figure 53 an exemplary maximal total throughput in the NoC of approximately 6200MByte/s for eight 16*32byte transfers and 8300Mbyte/s for eight 16*64byte transfers respectively. Still remember, that

no interference of data takes place here. Therefore this resembles the best case for an single channel scenario.

4.5.2 Conclusion

At the end of this chapter we can compare the result obtained by the simulation of several pattern transfers summarized in Table 12 and the bandwidth requirement presented in Table 11 for several applications that will mapped on the shown architecture.

The less bandwidth hungry application is the OUT-K frame processing used in routing protocol. In this case the implemented communication engine is able to cover the bandwidth requirements even in the case small amount of successive chunk of data are required. The peak bandwidth requirement is more or less 10 MByte/s fort eh two main communication path. As shown in Table 12 (rows 1 and 2) both communication pattern can be covered with each size of data chunk without compromise the application requirement.

Concerning the IEEE 802.11j, the analysis is a little bit more complex cause up to 5 parallel path are required. In any case the peak bandwidths are not very high and the communication engine is able to cover all the path with a minimum chunk size of 128-256 Kbytes.

The most bandwidth hungry application is the Motion Detection used in High Definition TV protocols. In this case in order to satisfy the bandwidth requirement of this application the maximum data chunk size is required and a dual port external memory interface is also necessary.

Chapter 5 Overall Implementation Results

This chapter represents a quantitative resource budgeting of the HDL database of the Morpheus chip RTL at the end of the functional specification phase.

The document refers to the schematic description of the logic hierarchy of the HDL database provided in the previous chapters and describes the expected timing performance, area occupation, and a rough power consumption estimation of the database.

The aim of this section is to provide a quantitative evaluation of the metrics on which the Morpheus prototype can be evaluated. As it is provided during the Front-End design phase, when designers have no knowledge of the technology support, they are bound to be inaccurate especially for what concerns dynamic power estimation. Moreover, feedback from technology implementation and bug fixing may induce slight changes on the results. Nevertheless, this quantitative evaluation effort represents a first valuable reference to estimate the potentialities of the Morpheus architectural design.

Area, timing and power estimations of the RTL code blocks were generated with *Synopsys Design_Compiler v.2005.09*. It should be considered that depending on the RTL database size and complexity, a degradation of around 20/40% could be introduced by the implementation phase on timing results. In turn, this figure heavily depends on the standard cell density ratio that normally ranges on 70/50% much depending on the design complexity and specifications.

Rough estimation of power consumptions have also been provided, making use of Switching Activity files annotation (SAIF) and the *Synopsys Power Compiler* tool, but it should be underlined that such evaluations are at Gate-level only, and do not take into account physical wire loads but generic wire

load models. They are thus bound to be strongly inaccurate. Leakage values for standard cells are less prone to variations, but it should be considered that the Morpheus design targets a technology that allow different threshold levels for std cells depending on the timing specifications of the relative paths, so that floor-planning related issues during P&R may significantly alter the so-called "threshold cocktail", with relevant impact on the final leakage figures.

Where applicable, power measurements for silicon proven macros have been provided.

- Timing evaluations are provided in Worst Case Commercial conditions, 0.9 V, 125C
- Power evaluations are provided in Nominal conditions, 1 V, 25C

The present chapter describes a quantitative resource budgeting of the Morpheus architecture and its RTL database at the end of the Front End phase. The status of the design at this point is after final functionality specification, but prior to P&R feedback, memory sizes and HRE size fine-tuning, and final bug fixing after verification.

Timing: At the current status of the design, the target frequency for the processor based infrastructure for all design components is greater than the 250MHZ mark in worst case conditions (WCCOM 125C 0.9V) after logic synthesis. This is a viable prerequisite for closing the implementation phase at a target of 200MHZ, although the large area of the chip and the presence of IPs featuring large size may lead to floor-planning issues that may impact on final timing. Computational engines in the chip (Heterogeneous Reconfigurable Engines – HRE) as shown in Chapter 3 are independent asynchronous clock islands. Their speed and consequently power consumption depends heavily on the mapped application. For this reason, each clock island features a software-programmable PLL to dynamically adapt HRE computation speed to the application specs and constraints.

Area: The current area estimations suggest a chip size of ~100 mm^2, including Pads. HREs will occupy around 60 to 70% of the overall area, the rest being divided between processor, communication infrastructure, on-chip memories and IO pads. This figure appears relevant in itself, but appears justified when compared to the overall computational power delivered by the Morpheus architecture. Further re-spins of the same architectural templates may offer space for optimization on timing [e.g. biasing threshold distribution according to feedbacks from measurements], whereas an optimization of the overall area above ~5/10% appears difficult without significantly impacting architectural choices or performance constraints.

Power Consumption: Power consumption estimation at this design stage (Gate-level netlist) is necessarily very inaccurate, especially for RTL logic, and floorplan-related IPs. Also, the selection of relevant application test-cases for power measurements is very difficult at half-way through the project. Preliminary evaluations show how leakage power for the chip should revolve around the 100 mW mark. This value may be significantly altered due to changes in threshold distribution in case the timing specs would prove very aggressive and timing closure for the architectural infrastructure or for the HREs would require massive use of low threshold logic. Any evaluation on dynamic power consumption is necessarily related to floor-plan choices and in particular mode on the selection of a relevant application test-bench. This would have such a large impact on overall consumption, to the point that it would be more significant to evaluate separate power profiles for different application domains. A significant contribution to overall dynamic consumption would also come from the chosen IO strategy: the inclusion of a high-speed large bandwidth memory controller will add a large overhead to the core power consumption that at the moment revolves around the 1.5 mW mark.

5.1 Overall Chip description

Table 13, from deliverable D4.5.1 [67], represents the top level pin-out of the Morpheus chip.

Pin Name	Direction	Functionality
HRESETn	IN	(Active Low) Overall System Reset
EOC	OUT	(Active Hi) Normally connected to external LED End of Computation: Control Signal that is triggered by the software routine exit() and signals the computation of a given software task
TEST_MODE[3 : 0]	IN	If != "0000" overrides the MPMC_data signal to produce some relevant debug signals from the internal bus architecture. It is normally connected on the test board to a set of switches.
VINIT_HI	IN	Selects boot type: '0' => ROM Boot through Parallel Port Interface '1' => RAM Boot: The RAM must be pre-loaded through TIC or JTAG connection.
PLL_CLKIN	IN	(Schmitt triggered) Normally connected to board oscillator Main Input signal for clock circuitry, receives external clock from board (range 0-80 MHZ)
PLL_ENABLE	IN	(Active Hi) Normally connected to External Switch or software-driven '0' : Utilize external PLL_CLKIN input as system Clock '1' : Utilize Main PLL output as system Clock
PLL_PD	IN	(Active Hi) Normally connected to External Switch or software-driven '0' : Power Down Main PLL to avoid unnecessary power consumption or to change PLL programming '1' : Power on Main PLL. In this case PLL is not operative until PLL_LOCK='1', which should take ~400 µs
PLL_LOCK	OUT	(Active Hi) Normally connected to external

		LED '0' : PLL is powered down or has not locked yet, it can not be used '1' : PLL is active and locked and can be used to drive system clock
PLL_MULFACT [1 : 0]	IN	Normally connected to external Switch Multiplication factor for Main PLL: "00" : PLL_CLKIN* "01" : PLL_CLKIN* "10" : PLL_CLKIN* "11" : PLL_CLKIN*
PLL_CLKOUT	OUT	Leaf of the System Clock Tree that is carried to output for testability purposes (Due to the Pad features this signal is filtered at ~180MHZ and is not significant above that figure)
ARM_nTRST	IN	(Active Lo) Test Reset Signal
ARM_TCK	IN	Test Clock Signal (Used for Jtag connection)
ARM_RTCK	OUT	Returned TCK, used to synchronize the Multi-ice controller
ARM_TMS	IN	JTAG Mode Select
ARM_TDI	IN	JTAG Serial Input
ARM_TDO	OUT	JTAG Serial Output
PP_DATA [7 : 0)	INOUT	Parallel Port (IEEE1284) bidirectional Data bus
PP_CONTROL [3 : 0]	IN	Parallel port (IEEE1284) Control bus
PP_DIRECTION	IN	Parallel Port Direction Signal (Schmitt triggered)
PP_NACKOUT_NSTROBEOUT	OUT	Parallel Port Asynchronous handshake signals
PP_NACKIN_NSTROBEIN	IN	
UART_RX_DATA	IN	RS232 Serial port rx signal
UART_TX_DATA	OUT	RS232 Serial port tx signal
M2K_IO [39 : 0]	INOUT	M2K Generic Bidirectional IO Signals (Direction is programmed via software)
MPMC_TESTIN	IN	TIC Test Mode Select: Note TIC is a synchronous, parallel, 32-bit wide methodology for on-chip bus verification that is part of the AMBA 2.0 bus protocol. Through the TREQa/TREQb control signals it is possible to take control

		of the onchip bus as bus master and thus access all devices (Memories, Peripherals etc) connected to the onchip bus. Address, Data Read and Data Write values are transmitted, according to the TIC protocol, through the TIC_DATA signals that in this case is multiplexed over the MPMC_DATA bus
MPMC_TREQA, MPMC_TREQB	IN	TIC control signals
MPMC_nBLSOUT[3 : 0]	OUT	MPMC Static Memory controller Byte Lane Select
MPMC_nWEOUT	OUT	(Active Lo) MPMC Static Memory controller Write enable. When TIC is active this signal behaves as TIC ACK
MPMC_nOEOUT	OUT	(Active Lo) MPMC Static Memory controller Output enable
MPMC_nSTCSOUT [3 : 0]	OUT	(Active Lo) MPMC Static Memory controller Chip (Bank) select
MPMC_ADDROUT [23 : 0]	OUT	MPMC Static memory controller Address out
MPMC_DATA	INOUT	MPMC Static memory controller Data bus
SD_CLK, SD_CLKN	OUT	(High speed differential dual Pad) CMC SDRAM Controller Differential Clock
SD_CLKE	OUT	CMC SDRAM Controller Clock enable
SD_WEn	OUT	(Active Lo) CMC SDRAM controller Write Enable
SD_RASn	OUT	(Active Lo) CMC SDRAM Controller Row Address Strobe
SD_CASn	OUT	CMC SDRAM controller Column Address Strobe
SD_CSn [1 : 0]	OUT	(Active Lo) CMC SDRAM controller Chip Select
SD_BANK [1 : 0]	OUT	CMC SDRAM Controller Bank Address
SD_ADDR [13 : 0]	OUT	CMC SDRAM Controller Address Bus
SD_DQM[7 : 0]	OUT	CMC SDRAM Controller Data Mask
SD_DQS[7 : 0]	OUT	CMC SDRAM Controller Data Strobe
SD_DQ[63 : 0]	INOUT	CMC SDRAM Controller Bidirectional Data Bus

Table 13: Top Entity Pinout

Figure 54 describes the overall structure of the Morpheus Chip, as described already in Chapter 2 and deliverables D3.1 and D3.2 [64][65].

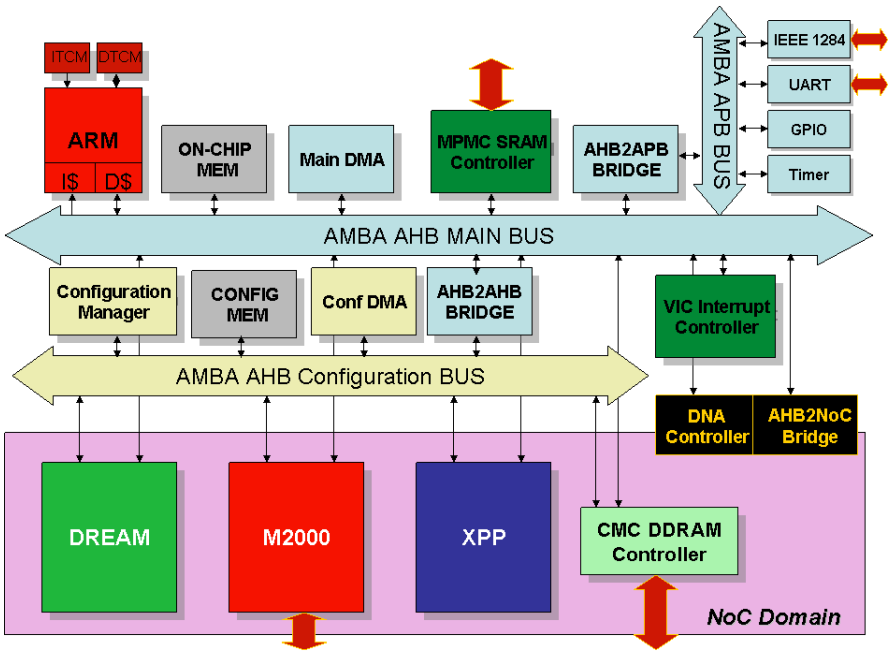

Figure 54: Overall description of the Morpheus chip architecture

The present section will provide performance estimations for each of the main Morpheus component blocks, following the organization outlined in the figure:

- ARM Processor Core, that is the main "interface" towards the end user and the overall system control and synchronization engine
- AMBA subsystem, comprising an AHB data bus (also utilized as control bus), an AHB configuration bus, an APB peripheral bus, two embedded DMAs (one per each AHB bus), bridges, a GPIO controller, a Timer, a IEEE1284 interface, an UART interface, a main onchip memory, configuration memory, interrupt handling system based on the ARM pl190 VIC vectorized interrupt controller, and a multi-port SRAM memory controller and TIC test engine based on the ARM pl175 MPMC

- Data communication infrastructure, composed of the NoC IPs (Network Interfaces + routers), a set of data transfer engines that collectively implement a distributed DMA structure, a traffic controller, multiplexing logic between AMBA and NoC-based access.
- Predictive configuration manager
- High speed, large bandwidth DDRAM memory controller
- Pact XPP HRE
- DREAM (PiCoGA-based HRE)
- M2000 FlexEOs-based HRE

5.2 Processor Based Infrastructure

5.2.1 ARM Core

The ARM A926EJS Core is a hard macro provided as layout library by ST. The macro contains the ARM 926EJS core running at the frequency of 380 MHZ@wc_0.9V_125C, the memory management unit, 16K+16K data and instruction caches, cache management logic. The macro is tightly coupled to two separate scratchpad memory modules (DTCM and ITCM) that provide single cycle fast access to the core. TCMs are not included in the macro but are instantiated at design time in the RTL database.

Block	Area (mm^2)
ARM core Macro including caches	2.11
16Kbytes ITCM	0.19
16Kbytes DTCM	0.19
Total	2.5

Table 14: Area of the ARM component

Block	Dynamic Power(μW/MHZ)	Leakage Power (mW)
ARM core Macro + caches	244.1	2.2
16Kbytes SP ITCM	20.2	0.76
16Kbytes SP DTCM	15.8	0.76
Total	280	3.72

Table 15: Rough Power Consumption estimations for the ARM926 core

5.2.2 AMBA Subsystem

Most of the AMBA bus system is composed by a gate-level Verilog library by *Synopsys Design_Ware*, so the following results will be estimations derived after logic synthesis: external components added to the design are two RTL IPs from ARM (the PL175 MPMC SRAM controller, and the PL190 VIC interrupt controller) and some small complementary components (on-chip memory interface, IEEE1284 interface) are distributed as open-source by ARCES under the GPL license.

Being the whole block with the only exception of memory cuts a soft IP synthesized on standard cells the Kgates metric has been considered more relevant than cell area. A rough evaluation of possible area after P&R is only provided on the Total figure. The same approach was maintained for all blocks designed at RTL level in the following sections of this document.

Table 16: Gate Count for the AMBA subsystem components

Block	Area (Kgates)
AHB Main Bus	2.5
Main bus DMA	43.8
Mpmc PL175	25.8
Vic PL190	13.2
System ROM	2
AHB2AHB bridge	1.3
AHB Conf Bus	0.9

Conf Bus DMA	41
APB Peripheral Bus	2
GPIO	0.8
Timer	2.2
IEEE 1284 Interface	0.2
Uart RS-232	2
Total	140
Estimated Area (70% Density)	0.9 mm^2

Table 17: Gate Count for the AMBA subsystem components

The AHB Subsystem also includes on-chip memories, whose area occupation is described below:

Block	Area (mm^2)
4x64K bytes Main Memory	3
4x64K bytes Configuration Memory	3
Total	6 mm^2

Table 18: Area occupation of memory cuts included in the AMBA bus system design

The AMBA Subsystem is able to run up to 290 MHZ@wc_0.9V_125C. The critical path of the overall logic resides in the Main_AHB_DMA block, due to the address generation and channel resolution mechanism in the DW_DMAC IP. Given the specifications, this delay was considered acceptable.

Block	Dynamic Power(μW/MHZ)	Leakage Power (mW)
Std Cells Logic	170	1.8
256K bytes Main Memory	120	10.4
256K bytes Configuration Memory	120	10.4
Total	410	22.6

Table 19: Rough Power Consumption estimations for the AMBA Subsystem

Note: The 64K bytes cut is the biggest memory model available. It is possible that larger memories will be necessary for both main and configuration bus (memory sizes will be fixed at M21. The current proposal is 256K for both main and configuration bus but overall chip area evaluations

may impose smaller figures) but in this case it will be necessary to join more cuts in the same memory block.

5.3 Hardware Services

5.3.1 The Predictive Configuration Manager block

The Predictive Configuration Manager (PCM) is an IP provided by CEA-List. By default, the component is off and does not issue interrupts nor does it access the configuration bus. The component wakes up after proper initialization procedure done by software on the ARM processor. The AHB slave interface has an addressing space of 64Kbytes. The Overall Area Estimation is around 160 Kgates.

The number of memory cuts is dependent on the number of memory ports utilized in the design. One buffer is required per each read port and write port. In the current configuration only 2 ports are used, thus 4 cuts are included. The Predictive Configuration Manager is targeted to run up to 350 MHZ@wc_0.9_125C.

Block	Area (mm^2)
Std Cells Logic (160Kg, 70% density)	1.05
14 SP/DP Memory Cuts	0.52
Total	1.57 mm^2

Table 20: Area occupation for the PCM

Block	Dynamic Power(µW/MHZ)	Leakage Power (mW)
Std Cells Logic	100	1
14 SP/DP memory cuts	62	0.71
Total	162	1.71

Table 21: Rough Power Consumption estimations for the PCM

5.3.2 The CMC DDRAM Memory Controller

The CMC is a high bandwidth dynamic memory controller that has been added to the Morpheus design to provide high bandwidth data access for stream-oriented applications. Since at the moment of writing all the results are relative to the closure of RTL functionality specification, all information reported here will be focused on the functional behavior of the CMC. Area/speed evaluations related to the design of the off-chip interface and data synchronization are not reported here as such details will be available only after the implementation phase. The CMC is designed to run up to 250 MHZ@wc_0.9_125C.

Block	Area (Kgates)
Cmc core	75.4
Configspace_top	1
Noc2cmc (2 instances)	2*26K = 52K
Total	128.4 Kgates
Area Occupation (70% density)	0.80 mm^2

Table 22: Main Building blocks of the CMC controller

Block	Area (mm^2)
2x(512x64) Read Buffer DP Memory	2x0.130629 = 0.261
2x(512x65) Write buffer DP Memory	2x0.132496 = 0.265
StdCells logic	0.80
Total CMC Area estimation	1.32 mm^2

Table 23: Area occupation of the CMC

Block	Dynamic Power(µW/MHZ)	Leakage Power (mW)
Std Cells Logic	113	1.07
Read Buffer Memories	2x(38.41)=76.82	0.897
Write buffer memories	2x(39.07)=78.14	0.905
Total	267.96	2.872

Table 24: Consumption estimations for the CMC

5.4 The Data Interconnect Infrastructure

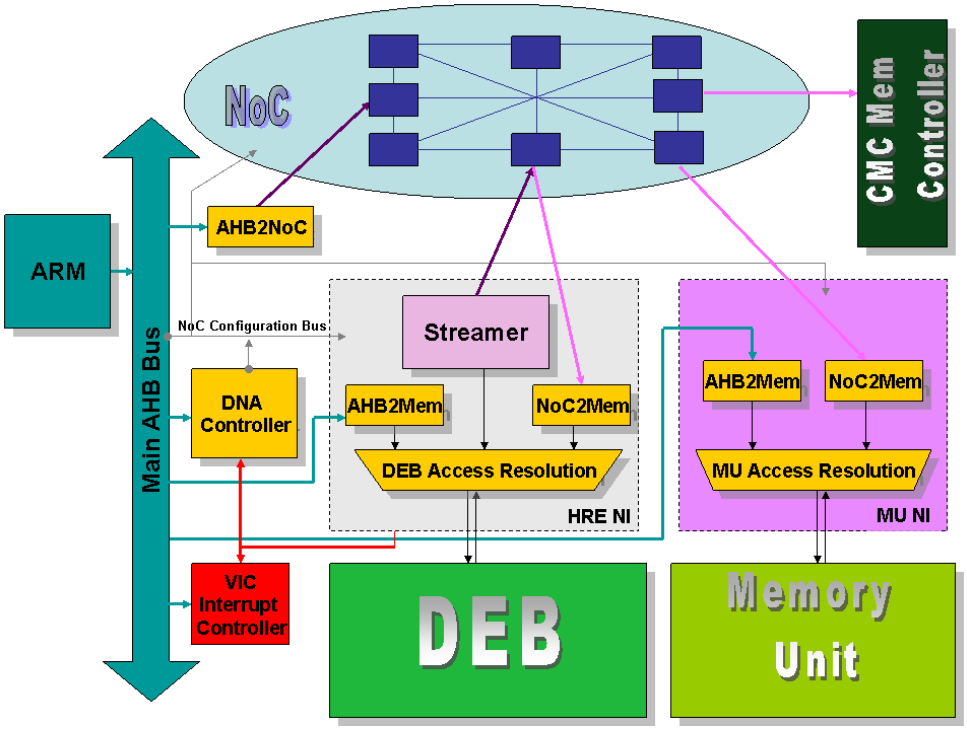

Figure 55: Schematic description of the MORPHEUS communication infrastructure

As explained in D4.5.1 [66], the specifications for the MORPHEUS communication infrastructure are

- Scalability, that is the possibility to add/remove nodes (HREs or storage units) in the system without affecting performance or programming model
- Performance, that is the capability to ensure the necessary bandwidth for relevant links between the nodes.
- In order to ensure testability and run-time verification the interconnect mechanism must provide low speed, standard AMBA-AHB access to all resources

As described in detail in [66] these specs, and in particular the scalability issue, suggested the utilization of a Network-on-chip oriented approach rather than a bus-based methodology. The STNoC “Spidergon” topology described in

Chapter 4 was chosen as reference for the NoC deployment and predefined NoC building blocks provided by ST (Initiator Network Interface, Target network interface, Router) were thus adopted as reusable IPs. In order to exploit the potentiality of the NoC approach, some modifications were performed in the HRE structure; these modifications are related to the interface between memory hierarchy and Interconnect, while no modification was performed on the STNoC standard components, in order to minimize risks and ensure high performance.

A Network-on-Chip is by definition a distributed communication platform where a set of independent initiator nodes (e.g. processor cores) issue transfers and a set of target nodes provide information storage (e.g. memory units) and respond to the transfer requests. On the contrary, in MORPHEUS, all transfers are supposed to be initiated (implicitly or explicitly) by ARM as macro-operands for a given macro-operation, much like the assembly for a standard processor is initiating transfers from the register file for an ALU operation. This centralized communication scheme is implemented through a "distributed DMA" pattern: each HRE node in the NoC is enhanced with a local data-transfer engine defined Streamer. Streamers also provide the HRE with very flexible addressing patterns that include 2D step/stride and circular buffer functionality. HREs "load" data chunks from DEBs and "store" them through the NoC to the target repository and vice-versa. From the core/user point of view this pattern describes the NoC as an enlarged and highly parallel DMA architecture. The user can than 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. Figure 55 describes the communication infrastructure. The encircled section represents the set of IPs provided by ST, and blue boxes represent IP target and initiator NIs. Routers are not depicted because their number and organization will totally depend on the physical topology of the chip which is not defined at this stage. From the NoC point of view, HREs represent peculiar nodes: they can both be NoC initiators (require

transfers from some storage units such as on-chip or off-chip RAM), or targets (process external requests such as a transfer request from another HRE or ARM). For this reason, the HRE Network interface is composed by the initiator NI connection (depicted in purple), the embedded DMA engine, as well as the target NI connection (pink) and the AMBA connection (blue) multiplexed over the HRE DEBs. ARM can require any transfer between HREs, as well as from any HRE to any storage unit (Onchip memory, Memory controller): transfers are initiated by programming specific configuration registers on the HRE network interface through a so-called network configuration bus (gray). This configuration is performed through a specific configuration channel reaching all HREs, mapped as slave on the AMBA bus.

During the verification phase all transfers can be issued by the ARM core. ARM is also connected as initiator to the Network-on-chip so it has full visibility of all network resources. During the computation phase, depending on the chosen programming pattern, the programmer may prefer to handle each data transfer configuration from the side of the driving processor. This is a more flexible approach, and safer as it allows a run-time programmable control of the operands flow in the application. During peak computation, in some cases, this may result in an excessive complexity form the user point of view or in an unnecessary demand for services from the side of the ARM core, especially for applications where a continuous flow of information needs to be implemented though the interconnect infrastructure. To provide a further level of automation in the deployment of the interconnect strategy a specific DNA (Data Network Access) controller is under design. This block will be used to handle end-of transfer notification automatically, issuing new transfers (when based on a regular pattern) without the need to resort to the core for handling each end-of-transfer request. In this way, it is possible to provide a regular streaming transfer, or a regular ping-pong buffering handshake in an automated pattern.

Block	Area (Kg)	Instances	Total Area (Kg)
DNA Controller	50	1	50
HRE Network Interface	100	6	600
Total			650
Estimated Area (70% Density)			3.7 mm^2

Table 25: Area occupation of main blocks composing the communication infrastructure

Block	Area (Kg)	Instances	Total Area (Kg)
Router	14.3	16	228,8
NI	10	7+7	144
Total			372.8
Estimated Area (70% Density)			2.12 mm^2

Table 26: Area occupation of STNoC IPs

Speed performance of the main blocks:

- The DNA controller currently performs at 160 MHZ but fixes is being upgraded to reach the target value of 250 MHZ
- Router, NI can be synthesized up to 800 MHZ, but the configuration chosen in the context of the Morpheus is of 250MHZ. Evaluations related to implementation issues may impose further speculations.
- The embedded DNA engine and glue/control logic can be synthesized up to 400 MHZ, but same considerations as above apply

One point emerging from this analysis is that the total Kgates count is relatively high, but composed by several instances of the same few blocks. A convenient option to mitigate P&R complexity would be to perform hierarchical P&R on the building blocks and reuse them at top level, although this option may be more resource-intensive.

It important to consider that of all components in the design, the communication infrastructure is the one that is more sensible to floor-planning issues that affect wire loads. Consequently, except where evaluating a self contained logic block whose communication is restricted at neighboring entities as is the case with the Streamers or the DNA controller, the FE dynamic power evaluations provided at this design stages are not reliable. In any case, a rough and very conservative estimation can range around 200/250 uW/MHZ dynamic power and 2/3 mW static power. When more stable, these figures should be added to the values described in the table above (as everywhere in this document leakage is estimated in nominal conditions).

5.5 Heterogeneous Reconfigurable Engines (HREs)

5.5.1 DREAM

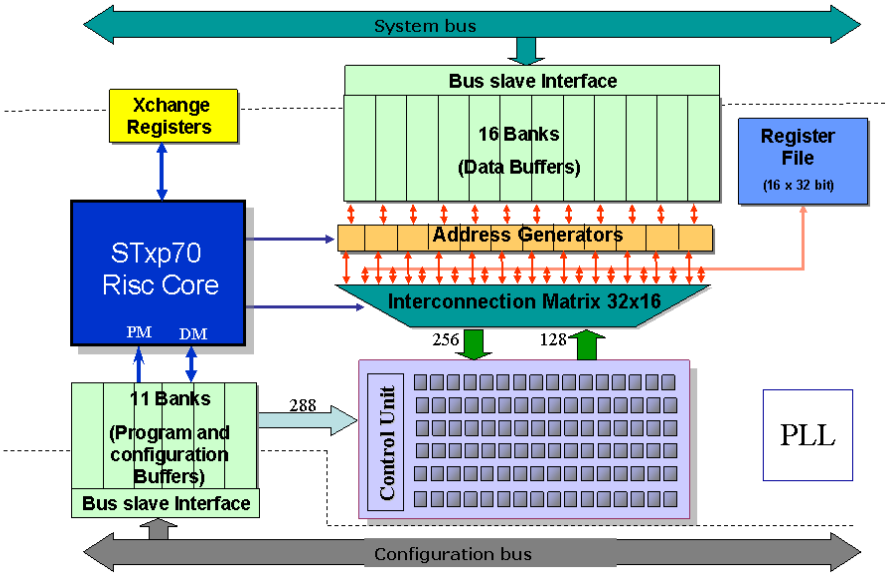

Figure 56: Description of the DREAM Architecture

As described in Figure 56, the PiCoGA-based HRE features a sophisticated communication and memory hierarchy that was designed, in the context of the Morpheus Project, to sustain the reconfigurable unit computation with the required data bandwidth and communication flexibility.

The main elements composing DREAM (see also 3.3.1) are the STxP70 Risc processor core (Control engine), the PiCoGA gate-array (Computation engine), a set of 16 dual port memory cuts, a set of address generators for supporting concurrent access to the 16 buffers, and an interconnect Matrix that provide programmable connection between the PiCoGA IO ports and the buffers.

From a technology point of view, thus, the DREAM HRE is a mix between custom layout circuits (the PiCoGA gate-array, custom memory cuts, PLL) and

RTL logic. In the following, area occupation figures will be described accordingly to the nature of each block

Block	Area (Kgates)
STxp70 Processor	40
PiCoGA Interface (Stall handling, configuration control, context switch)	50
Address Generators	12 (16*0.7)
Interconnect Matrix	45
Others	13
Total	170
Estimated Area (70% Density)	~1.1 mm^2

Table 27: Gate count of the main RTL sub-blocks composing DREAM

Block	Area (mm^2)
PiCoGA Gate-array (24x16 cells)	7.6
16x(1Kx32) DP Data Buffers (DEBs)	2.03 (16*0.127)
11x(1Kx32) DP Configuration Buffers (CEBs)	1.4 (11*0.127)
PLL	0.16
Total	10.2 mm
Overall DREAM Estimation(1)	11.4 mm^2

Table 28: Area of the hard macro blocks composing DREAM

Timing performance of the DREAM clock domain after logic synthesis are 300 MHZ @ wc_0.9V_125C. The critical path of the block is due to the interconnect matrix that connects the data buffers (DEBs) output ports with the PiCoGA inputs. The matrix provides full connectivity requiring 12x32 32:1 multiplexers on input signals, which impose a significant burden both in terms of timing and area occupation A more aggressive performance could be achieved renouncing to full connectivity, but it has been evaluated that the cost from the point of view of algorithm development would be unacceptable.

A second evaluated solution was that of pipelining the interconnect structure, but that would add to the latency of each PiCoGA operation and that also has been considered not convenient for application mapping.

Note: The size of the PiCoGA macro is significant, and its shape a relevant factor in the definition of the DREAM floor-plan. Also the 27 memory cuts used in the architecture impose restrictions on the floor-planning style. This evaluation and layout trials show that the FE estimation is not realistic and it would not be possible to meet the required timing constraint with an area value that matches the FE estimation. It should be thus considered that the area required by the DREAM IP will revolve around 14/16 mm^2.

Power estimations for the DREAM HRE are shown in the table below:

Block	Dynamic (µW/MHZ)	Leakage (mW)
STxp70 Processor	30	0.4
PiCoGA Interface	40	0.6
AG and Interconnect Matrix	70	0.8
Data Memory Buffers (DP, 64Kb)	212	5.92
Configuration Memory (DP, 44Kb)	3.2	4.07
PiCoGA Gate Array (24x16 cells)	300(1)	15
Total	655.2	26.8

Table 29: Main contributions to the estimated Power consumption of DREAM

The dynamic power consumption of PiCoGA has been measured from prototypes. The reference value is 25 µW/MHZ per each Row that is effectively computing. The ratio of active rows/MHZ depends strongly on the deployed application. For a very generic estimation it has been suggested here a ratio of 50%, that is 12 rows on the total 24 active per cycle, taking into account peak kernel computation (~24/24 active rows) and idle time.

5.5.2 M2000

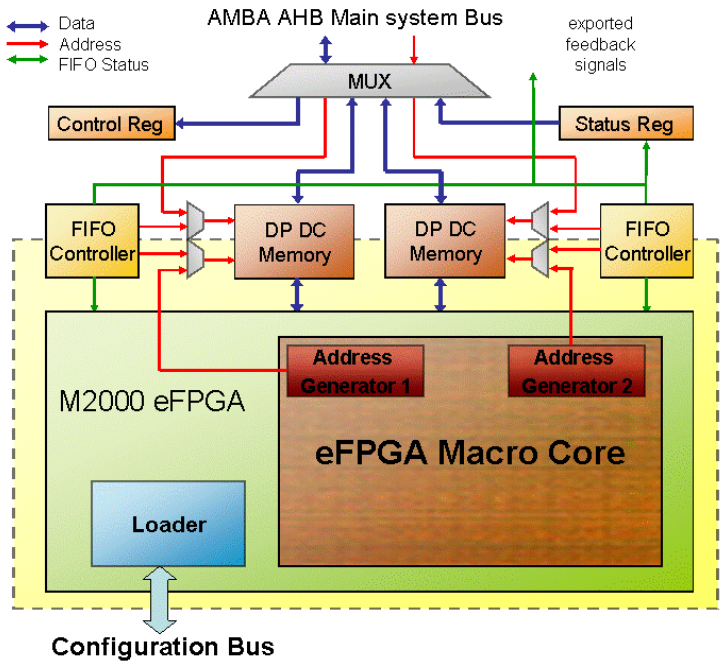

Figure 57: Block diagram of the M2000 HRE

The M2000 block is a computation engine centered on the M2000 FlexEOS FPGA. Similarly to the DREAM and XPP HREs, this HRE contains a data interconnect and communication logic aimed at providing flexibility and bandwidth for sustaining the IP computation capability.

Figure 57 describes the M2000 HRE: as it is the case for all Morpheus HREs clock domain crossing and local data storage is implemented on a set of buffers, DEBs. Address generation for concurrent DEB access can be performed according to two alternative patterns: either with a (asynchronous) FIFO paradigm for stream oriented applications, or generating DEB access directly on the eFPGA. For what concerns the configuration interface, the FlexEOS product features a memory mapped *loader*, described as a RTL IP. The loader is mapped on the configuration bus and works at the same frequency.

Block	Area (Kgates)
FIFO Controller (8 instances)	12
HRE Control and Synchronization	15
FlexEOS Loader	20
Total	57
Estimated Area (70% Density)	~0.4 mm^2

Table 30: Gate count of the main RTL sub-blocks composing the M2000 HRE

Block	Area (mm^2)
FlexEOS eFPGA Macro (4K cells)	2.9
Data Buffers (DEBs) 8x(1Kx32) DP	0.85
PLL	0.16
Overall M2000 HRE Estimation	4.55 mm^2

Table 31: Area of the hard macro blocks composing the M2000 HRE

Front end estimations for the maximum achievable performance are:

- M2000 Loader (Residing on configuration bus) -> 180 MHZ
- Data Interface (Residing on main AHB bus) -> 250 MHZ

As for power consumption, it is very difficult to estimate figures for the dynamic consumption of the FlexEOS HRE as this would strongly depend on mapped applications. In Table 32 an exemplar design was utilized to estimate such figures: an AES (Advanced Encryption Standard) application running on 80% of available cells. AES can be considered a good example of an energy-demanding application. Smaller applications would proportionally require a more reduced consumption.

Differently from XPP and DREAM M2000, being an eFPGA device, may feature working frequencies which are quite different from those of the Morpheus infrastructure. For this reason dynamic consumption contributions have been divided according to the relative clock domain:

- APP => Application frequency, in the range 40:120 MHZ
- CORE => Core frequency, set at 250MHZ (FE estimation)

Block	Dynamic (µW/MHZ)	Leakage (mW)
FlexEOS eFPGA Macro (20K cells cut) (1)	6170 @APP	30
Data Buffers (DEBs) 8x4K bytes DP	119.6@APP + 1.7@CORE	1.5
Std Cells logic	22@APP + 0.1@CORE	0.47
Total	6311@APP + 2.5@CORE	33.23

Table 32: Main contributions to the estimated power consumption of the M2000 HRE

5.5.3 Pact XPP

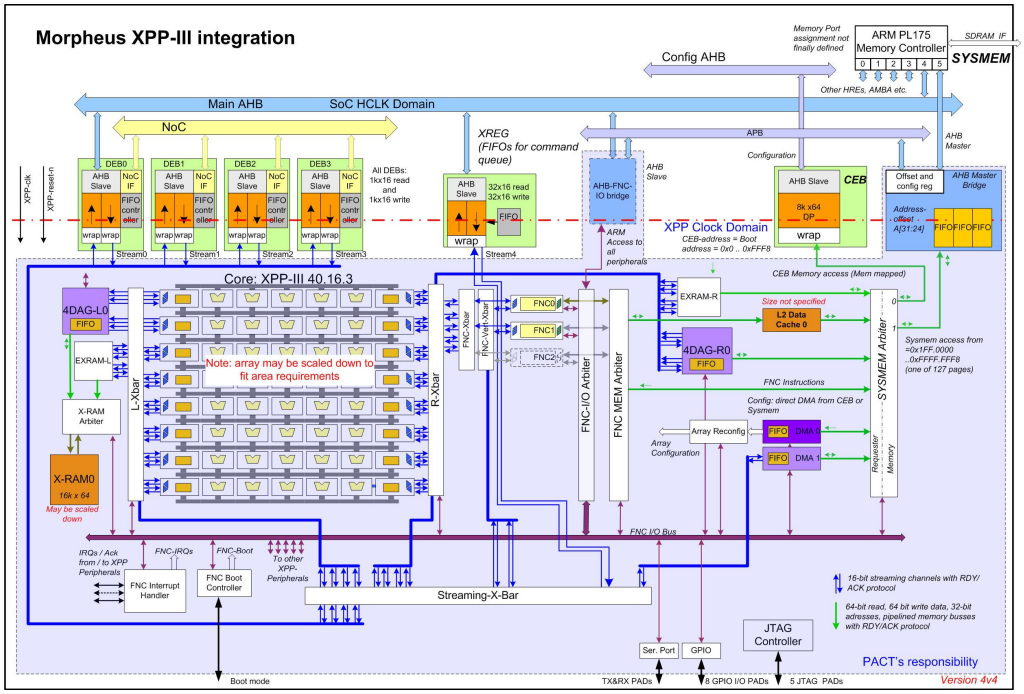

Figure 58: Description of the XPP HRE

Figure 58 (from [67]) describes the XPP HRE. The XPP HRE is a computation engine centered on the Pact XPP embedded signal processor. As XPP is oriented at streaming computation, this block contains the logic aimed at providing stream-based connection towards the system-level communication interface (FIFOs) for sustaining the IP computation capability.

Since the HDL coding of the Pact XPP is still under definition significant adjustment of the figures provided below are still possible.

5.5.3.1 XPP/System connection

XPP is connected to the Morpheus system via a set of DEBs used as mono-directional FIFOs to/from the XPP macro. The overhead due to this logic consists in 20Kgates for FIFO control and synchronization, plus 2 Kbytes DEBs and 64 Kbytes CEBs.

Block	Area (mm^2)
Data Buffers (DEBs) 8x(1Kx16) DP	0.85
Configuration Buffers 1x(8Kx64) DP	1.451
FIFO Control and synchronization (20Kg, 70% density)	0.125
PLL	0.16
Total	2.84

Table 33: Area estimations of the main blocks connecting the XPP Macro to the Morpheus System

Block	Dynamic (µW/MHZ)	Leakage (mW)
Data Buffers (DEBs) 8x(1Kx16) DP	168	0.802
Configuration Buffers 1x(8Kx64) DP	1.24	1.54
FIFO Control and synchronization	28.71	0.23
Overall XPP HRE Estimation	197.95	2.572

Table 34: Rough power consumption evaluations for the XPP/Morpheus connection

5.5.3.2 XPP core macro

As it is the case with the ARM Core, the PiCoGA gate-array array and the M2000 FlexEOS eFPGA, due to its relevant complexity, area occupation, and to the peculiar features of its design Pact XPP will be imported on the Morpheus design as a layout macro.

The estimation of area requirements of the XPP array are based on a trial layout which was done with 30 ALU-PAEs and 12 RAM-PAEs and 2 FNC-PAEs. Bottom line ALU and RAM PAEs has been included.

Block	Area (mm^2)
RAM PAE (x12)	0.60 (x12)
ALU PAE (x30)	0.41 (x30)
BL RAM PAE (x2)	0.33 (x2)
BL ALU PAE (x12)	0.22 (x12)
Total Array interconnection overhead	1.1
FNC PAE (x2)	2.72 (x2)
Reference Design	~8.26
Total	37.6

Table 35: Estimation of the area occupation of the main blocks composing the XPP

5.6 Padframe

The padframe is of course very liable to modifications due to implementation issues: at the current design stage, as described in Table 13 the design includes 249 signal pads, of which 104 optimized for high speed for supporting the CMC controller.

The design is not pad limited, featuring an area in the range of 90/100 mm^2. The contribution of the Padframe to the overall chip area can be roughly estimated as follows: supposing that the chip floorplan should be more or less regular in size (Chip height ~ Chip length) we have

Padframe Area ≈ Pad height * SQRT(Area) *4 ≈ 4.5 mm^2

(Note: Corners are included twice in this estimation, but that is done on purpose to ensure some flexibility for defining a padframe not perfectly square to accommodate large macros placement)

The Core reference voltage is 1V, the IO Ring reference voltage 3.3V . An analog voltage regulator is added to the design to provide stable reference voltage and minimize IR-Drop effects (given the chip area and the high consumption of particular chip regions such as the HRE, IR-Drop will have to be taken into account anyway in the course of the design.).

The number of power pads will be defined when more detailed power estimations will be possible. A theoretical reference value for the moment is ~150 voltage feed pads. The chosen IO package may impose restrictions on these number, although it is possible to bond two voltage feed pads to the same package pin.

Entity	Instances	Dynamic 1V (µW/MHZ)	Dynamic 3V3 (µW/MHZ)	Leakage (mW)
Power feed	~150	n.a.	n.a.	~0.2
Signal Pads (~20 MHZ)	145 (21in, 39 out, 84 inout)	21*.3 + 39*.14 + 84*.4= 45	21*10 + 39*60 + 84*90= 10110 (200 mW)	102*.095=9.6
High Speed Pads (200 MHZ)	104 (32 out, 72 inout)	39*.14 + 72*.4= 28.8	32*60+72*90 = 8400 (1600 mW)	104*.23= 23.9
Overall Padframe	~400	70 µW/MHZ, 1.4 mW	1800 mW	35 mW

Table 36: Rough power consumption estimation for the IO Ring

It should be noted that most Pads are control signals that do not commute regularly in the chip lifetime, excluded memory controller pads.

5.7 Final Consideration

In this final section overall estimations will be described for the whole chip. In order to ease the readability of this final report, the Morpheus database has been divided in 5 logic components: the 3 separated HREs described as

independent clock island, the data communication infrastructure, the processor-based subsystem (ARM core, AHB bus hierarchy, peripherals, on-chip memories and external memory controllers) and the Padframe.

Block	Area Estimation (mm^2)	Kgate (where applicable)	Kbytes (where applicable)	Macro Area (mm^2)
ARM+AMBA Interconnect Infrastructure	3.5	AMBA (140)	64 Kb (SP)	ARM (2.11) Memory (0.4)
NoC Based Interconnect + DNA	7	DNA (50) DMAs (270) STNoC NIs (140)		
DREAM clock island	18	170	132 KB (DP)	PiCoGA (10.2) Memory (3.92)
M2000 clock island	6.2	Loader + Control (57)	32 KB (DP)	M2K (2.9) Memory (0.85)
XPP clock island	36	Reference Design (350)	64 Kb (SP) 64 Kb (DP)	Array (24) Memory (2.5)
System Memory	7.5		512Kb (SP)	Memory (7.5)
PCM	2.2	160	14Kb (DP) 22Kb (SP)	Memory (0.7)
CMC	2	125	24 Kb (DP)	Memory (0.7)
PAD Ring + PLL	6.1			
Total	**92**			

Table 37: Main contributions to overall chip area

Note: To All area figures described in previous tables it has been added a slight overhead to take into account routing congestion due to the presence of large hard macro blocks

Conclusion

Many issues of current embedded systems design have been investigated in this work, and, having in mind the template of a Multi Processor System-on-Chip, different steps of architectural exploration and design were presented to eventually approach to a novel heterogeneous multiprocessor architecture, combining the advantages deriving from reconfigurable hardware, state of the art Network-on-Chip and parallel processing.

The known density advantage of reconfigurable hardware over standard processors has been extended by coupling a standard ARM RISC processor to several reconfigurable engines, achieving an improved flexibility and programmability compared to ASIC based platforms. The results described in this thesis have been validated by a complete design flow aimed to integrate the described concept in a silicon prototype.

Step by step, each chapter of this thesis presents different contributes integrated in the design of the proposed architecture. Chapter 1, which is dedicated to introduction, state of the art overview and general concepts on reconfigurable computing, multiprocessor architectures and onchip communication, is excluded from this summary,.

In chapter 2, we introduced the general outlines of the proposed SoC architecture, based on the integration of several reconfigurable architectures characterized by a different grain. The system is based on an ARM core, a complete hierarchy of AMBA busses for testing purposes and configuration management, and also features a set of standard memories including cache memories, on-chip memory and a dedicated external memory controller to integrate SRAM/FLASH memories. A set of different peripherals is also integrated.

In chapter 3, we gave a complete explanation of the main concepts adopted to define the complete memory hierarchy. Different Reconfigurable Architecture are characterized by different I/O interfaces, as well as different

working clock frequencies. In order to hide the heterogeneity of each block, a dedicated interface based on the usage of dual port dual clock memories has been adopted.

In chapter 4, we explained some details of the communication strategies adopted in the design of this architecture. A set of bus solutions is used to provide a secure and familiar medium for debugging and to manage the configuration transfer of the reconfigurable engines. On top of this, a novel approach based on the Spidergon NoC engine is adopted to manage high bandwidth data transfer paths between the reconfigurable unit and the on/off-chip memories. Finally, to provide the end user a single homogeneous interface when describing data transfers, the HREs have been equipped with local DMA-like data transfer engines, programmed and controlled at system level through the toolset. Communication synchronization and control may be handled in this way by software routines running on the main processor.

The work led to the implementation of a silicon prototype in 0.090µm technology provided by STMicroelectronics.. In Chapter 5, we present the implementation results achieved during the design of the MORPHEUS architecture. The chip aims to fit in a 90mm2 die.

Bibliography

[1] R. Hartenstein, "*A decade of Reconfigurable Computing: a visionary Retrospective*", Proceedings DATE 2001.

[2] M. Coppola et al, "*Spidergon: a novel on-chip communication network*", IEEE SOC 2004.

[3] The MORPHEUS reference website: http://www.morpheus-ist.org.

[4] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, R. Guerrieri, "*A VLIW processor with reconfigurable instruction set for embedded applications*", IEEE Journal of Solid-State Circuit, Nov. 2003.

[5] A. Lodi, A. Cappelli, M. Bocchi, C. Mucci, M. Innocenti, C. De Bartolomeis, L. Ciccarelli, R. Giansante, A. Deledda, F. Campi, M. Toma and R. Guerrieri, "*XiSystem: a XiRisc-based SoC with a Reconfigurable IO module"*, IEEE Journal of Solid-State Circuit (JSSC), Jan. 2006.

[6] M. Bocchi, C. De Bartolomeis, C. Mucci, F. Campi, A. Lodi, M. Toma, R. Canegallo, R. Guerrieri, "*A XiRisc-based SoC for Embedded DSP Applications"*, IEEE Custom Integrated Circuits Conferences (CICC'04), Oct. 2004

[7] A. Lodi, M. Toma, F. Campi, "*A Pipelined Configurable Gate Array for Embedded Processors*", Proceeding on FPGA 2003.

[8] A. Cappelli, A. Lodi, C. Mucci, M. Toma, F. Campi, "*A Dataflow Control Unit for C-to-Configurable Pipelines Compilation Flow*", IEEE Symposium on FCCM, Apr. 2004.

[9] C. Mucci, C. Chiesa, A. Lodi, M. Toma, F. Campi, "*A C-based Algorithm Development Flow for a Reconfigurable Processor Architecture*", IEEE International Symposium on System on Chip, November 2003.

[10] C. Mucci, F. Campi, A. Deledda, A. Fazzi, M. Ferri, M. Bocchi, "*A cycle-accurate ISS for a dynamically reconfigurable processor*

architecture", IEEE Reconfigurable Architecture Workshop (RAW), Apr. 2005.

[11] C. Mucci, M. Bocchi, P. Gagliardi, L. Ciccarelli, A. Lodi, M. Toma, F. Campi, "*A Case-Study on Multimedia Applications for the XiRisc Reconfigurable Processor"*, Proceedings on IEEE Int'l Symposium on Circuits and Systems (ISCAS), May 2006.

[12] F. Campi, A. Deledda, M. Pizzotti, L. Ciccarelli, C. Mucci, A. Lodi, A. Vitkovski, L. Vanzolini, P. Rolandi, "*A dynamically adaptive DSP for heterogeneous reconfigurable platforms"*, Proceedings on IEEE/ACM DATE 2007.

[13] Moore GE (1965), "Cramming More Components onto Integrated Circuits", Electronics, April 19, pp114-117.

[14] International Technology Roadmap for Semiconductors (2005) ITRS. Available at http://www.itrs.net.

[15] S. Brown and J. Rose, "*Architecture of FPGAs and CPLDs: A Tutorial*", IEEE Design Test f Computers, 1996, 13(2):42-57.

[16] W. Wolf, "*FPGA-Based System Design*", Prentice Hall 2004, Upper Saddle River, NJ.

[17] David A. Patterson, "*Future of Computer Architectures*", Berkeley EECS Annual Research Symposium (BEARS), College of Engineering, UC Berkeley, US, February 23 2006.

[18] Ahmed A. Jerraya, Aimen Bouchhima, Frédéric Pétrot, "*Programming Models and HW-SW Interfaces Abstraction for Multi-Processor SoC*", Proceedings of the 43rd annual conference on Design automation, San Francisco, USA, 2006.

[19] Caspi E., Chu M., Huang R., Weaver N., Yeh J., Wawrzynek J., and A. DeHon, "*Stream Computations Organized for Reconfigurable Execution (SCORE)*", *FPL'2000*, LNCS 1896, pp. 605-614, 2000.

[20] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti and M.Milano, "*Communication-Aware Allocation and Scheduling Framework for Stream-Oriented Multi-Processor Systems-on-Chip*", IEEE Design Automation and Test in Europe, Munich, Germany, March 2006.

[21] J. Chaoui, K. Cyr, S. de Gregorio, J. P. Giacalone, J. Webb, and Y. Masse, "Open multimedia application platform: enabling multimedia applications in third generation wireless terminals through a combined RISC/DSP architecture" in Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP2001), vol. 2, Salt Lake City, UT, 2001, pp. 1009–1012.

[22] OMAP 3 family of multimedia application processors. [Online]. Available at: http://focus.ti.com/pdfs/wtbu/tu_omap3family.pdf

[23] Mobile multimedia application processor. [Online].

Available at: http://www.st.com/stonline/products/literature/bd/14379.pdf

[24] Nexperia PNX1500 family of connected media processors. [Online]. Available at: http://www.nxp.com/acrobat_download/9397/75015926.pdf

[25] Athanas R., Silvermann H. (1993) "*Processor Configuration Through Instruction Set Metamorphosis*". IEEE Computer, 26(3): pp. 11-18.

[26] Razdan R, Smith M (1994) "*A High Performance Microarchitecture with Hardware-Programmable Functional Units*". Proc. Microarchitecture (MICRO-27), 1994. pp- 172-180.

[27] Rabaey JM (2000) "*Silicon Platforms for the next generation wireless systems what role does reconfigurable hardware play?*", Proc. Field Programmable Logic and Applications Conference (FPL), pp 277-285.

[28] Mangione-Smith WH, Hutchings B, Andrews D, DeHon A, Ebeling C, Hartenstein R, Mencer O, Morris J, Palem K, Prasanna VK, Spaanenburg HAE (1997) "*Seeking solutions in configurable computing*", IEEE Computer, 30(12):38-43.

[29] Hartenstein R (1997) "*The Microprocessor is no more general purpose*", Invited Paper, Proc. the international conference on Innovative Systems in Silicon, October 1997.

[30] G. Estrin, "*Reconfigurable computer origins: the UCLA fixed-plusvariable (f+v) structure computer*" IEEE Annals of the History of Computing, vol. 24, no. 4, pp. 3–9, 2002.

[31] G. Estrin , "Organization of computer systems-The fixed plus variable structure computer," in Proceedings of the Western Joint Computer Conference, New York, 1960, pp. 33–40.

[32] DeHon A (2000) "*The Density advantage of reconfigurable computing*", IEEE Computer, Vol 33, Issue 4, April 2000, pp 41 – 49.

[33] Bondalapati K, Prasanna VK (2002) "*Reconfigurable Computing Systems*", Proceedings of the IEEE, Vol 90, Issue 7, July 2002 , pp 1201-1217.

[34] DeHon A, Wawrzynek J (1999) "*Reconfigurable Computing: What, Why and Implications for Design Automation*", Proc. DAC, 1999, pp 610-615.

[35] Wong S, Vassiliadis S, Cotofana S (2002) "*Future Directions of (Programmable and Reconfigurable) Embedded Processors*", Proc. the 2nd Workshop on System Architecture MOdeling and Simulation (SAMOS).

[36] Barat F, Lauwereins R, Deconinck G (2002) "*Reconfigurable instruction set processors from a hardware/software perspective*", IEEE Transactions on Software Engineering, 28(9):847-862.

[37] Singh H, Lee MH, Lu G, Kurdahi FJ, Bagherzadeh N, Chaves Filho EM (2000) "*MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications*", IEEE Transactions on Computers, 49(5):465-481.

[38] Hartenstein R (2001) "*A decade of Reconfigurable Computing: a visionary Retrospective*". In Proc. DATE, pp 642-649.

[39] Bolsen I (2002) "*Challenges and opportunities of FPGA platforms*", Proc. FPL, pp 391-392.

[40] Callahan T, Hauser JR, Wawrzynek J (2000). "*The Garp Architecture and C Compiler",* IEEE Computer, 33(4):62–69.

[41] Vassiliadis S, Wong S, Gaydadjiev G, Bertels K, Kuzmanov G, Panainte EM (2004) "*The MOLEN Polymorphic Processor. In: IEEE Transactions on Computers*", Vol 53, no. 11, Nov 2004, pp 1363 – 1375.

[42] Campi F, Toma M, Lodi A, Cappelli A, Canegallo R, Guerrieri R (2003) "*A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications*", ISSCC Digest of Technical Papers, pp 250–251.

[43] Lodi A, Campi F, Toma M, Cappelli A, Canegallo R, Guerrieri R (2003) "*A VLIW processor with reconfigurable instruction set for embedded applications*", IEEE Journal of Solid-State Circuits (JSSC), 38(11):1876-1886.

[44] Goldstein SC, Schmit H, Budiu M, Cadambi S, Moe M, Taylor RR (2000) PipeRench: A Reconfigurable Architecture and Compiler. In IEEE Computer, April 2000, pp 70-77.

[45] Wilson R et al. (1994) "*SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compiler*". SIGPLAN Notices, page 31, Dec 1994.

[46] Vorbach M, Becker J (2003) "*Reconfigurable processor architectures for mobile phones*", Proc. the Int'l Parallel and Distributed Processing Symposium, April 2003.

[47] Corporaal H (1998) "*Microprocessor Architectures: From VLIW to TTA",* John Wiley & Sons, Chichester.

[48] Arnold JM (2005) "*S5: the architecture and development flow of a software configurable processor*", In Proc. IEEE International Conference on Field-Programmable Technology, pp 121-128.

[49] Sato T, Watanabe H, Shiba K (2005) "*Implementation of dynamically reconfigurable processor DAPDNA-2*". In Proc. IEEE VLSI-TSA International Symposium VLSI Design, Automation and Test, pp 323-324.

[50] Mei B, Vernalde S, Verkest D, DeMan H, Lauwereins R (2003) "*ADRES: An Architecture with Tightly coupled VLIW Processor and Corse-Grained Reconfigurable Matrix*". In Proc.FPL, pp 61-70.

[51] Rivaton A, Quevremont J, Zhang Q, Wolkotte P, Smit G (2005) "*Implementing non power-of-two FFTs on Coarse-Grain Reconfigurable Architectures*". Proc. Int'l Symposium on SOC, pp 74-77.

[52] Smit G, Heysters P, Rosien M, Molenkamp B (2004) "*Lessons Learned from Designing the Montium, a Coarse grained Reconfigurable Processing Tile*". In Proc. International Symposium on SoC, pp 29 – 32.

[53] Ciricescu S, Essick R, Lucas B, May P, Moat K, Norris J, Schuette M, Saidi A (2003) "*The Reconfigurable Streaming Vector Processor*". In Proc. Intl Symposium on Microarchitectures (MICRO-36), pp 141-150.

[54] Dezan C, Jegot C, Pottier B, Gouyen C, Lagadec L (2006) "*The case study of block turbo decoders on a framework for portable synthesis on FPGA*". In Proc. Hawaii International Conference on System Sciences.

[55] M2000 manual : "*The FlexEoS Loader*". Available at: http://www.m2000.com

[56] ARM ltd: "ARM926EJS Technical Reference Manual" Rev r1p5

[57] ARM ltd: "AMBA Specification" Rev 2.0

[58] ARM ltd: "PrimeCell Multi-Port Memory Controller" Rev r1p2

[59] Synopsys: "DesignWare DW_ahb Databook" Version 2.04a

[60] Synopsys: “DesignWare DW_apb Databook” Version 1.02b

[61] Synopsys: “DesignWare DW_ahb_dmac” Version 1.04c

[62] Synopsys: “DesignWare DW_apb_gpio Databook” Version 2.04

[63] Synopsys: “DesignWare DW_apb_uart Databook” Version 3.02a

[64] MORPHEUS, Deliverable D3.1: “Preliminary architecture definition”

[65] MORPHEUS, Deliverable D3.2: “Report on subtask specifications of WP3”

[66] MORPHEUS, Deliverable D3.4: “Architectural Metrics for Communication”

[67] Morpheus, Deliverable D4.5.1: “Report on HDL database after functional RTL description”

[68] MORPHEUS, Deliverable D5.1: “Specification of application test cases”

[69] MORPHEUS, Deliverable D5.2: “Definition of reference baselines and evaluation metrics”