Prototyping of embedded systems

van der Steen, Twan H.J.

Award date:
2001

TU/e technische universiteit eindhoven

Faculty of Electrical Engineering
Section Design Technology For Electronic Systems (ICS/ES)
ICS-ES 772

Master's Thesis

PROTOTYPING OF EMBEDDED SYSTEMS.

T.H.J. van der Steen

Coach:	ir. I.C. Kang (Philips Research Laboratories)
Supervisor:	prof.dr.ir. J.L. van Meerbergen
Date:	August 2001

Prototyping Of Embedded Systems

Twan van der Steen
Monday, 20 August 2001

Graduation professor: Prof. dr. ir. J.L. van Meerbergen
Eindhoven University of Technology
Department of Electrical Engineering
Information- and communications System Group (ICS)
ES (Design Automation)

Daily supervisor : ir. I.C. Kang
Philips Research Laboratories Eindhoven

Author : ing. T.H.J. van der Steen
Philips Research Laboratories Eindhoven

Abstract

In the old days when people were developing ICs the first silicon was (almost) never working. It was (and still is) an iterative process for making ICs. But when the sizes in IC technology were decreasing the mask set cost were increasing rapidly. Because of the iterative process every iteration needs a new mask set. A method for decreasing the number of expensive mask sets is very welcome.
This thesis proposes a strategy for decreasing the number of mask sets by prototyping the desired functionality on a PC prototyping system. This prototyping system is based on reusable PCI boards with FPGAs on it. The goal is to develop reusable components or IP Blocks and map it onto FPGAs, these FPGAs are part of a bigger heterogeneous multi-processor architecture. If you want to develop reusable code for a specific application domain without knowledge of the future-system bus its wise to read this thesis. This method will show how to develop reusable components or IP blocks on a reusable PC prototyping platform with a reusable standard interface stack, and C-HEAP as synchronisation protocol.

Keywords: C-HEAP, real-time, multiprocessor architectures, FPGAs, hardware-software co-design, System prototyping.

Acknowledgements

Over the last 10 months I had the privilege of working in an excellent and inspiring environment, which eventually resulted in this thesis. I was fortunate to work in the Embedded Systems Architectures on Silicon (ESAS) group at Philips Research. I look back on a very pleasant time and I would like to thank all people who contributed to this.

In the first place I would like to thank my Prof. Jef van Meerbergen for his supervision and for giving me the opportunity to perform my Master research at Philips Research. Though he has a busy schedule, which prevented him to play an active role in the early stages of the research, he did manage to find the time at the end to understand the fundamentals of the presented work and helped me to finalise the project. I also want to thank Jeffery Kang for his contribution at our weekly meetings. Both offered me the freedom to choose my own research directions, while their suggestions guided me to take the right paths. Likewise, I would like to thank Albert van der Werf for his contribution in the beginning of my research.

Many people at Philips Research have contributed to this thesis by discussing and commenting on my work. I would like to thank O.P. Gangwal in the ESAS group, for his clarification about the C-HEAP protocol. Further, Hennie van de Poel for his helpfulness with Renoir, PCI, and Local Bus protocols and Jennifer Blijlevens for the FPGA-Board.

Contents

List of Figures

List of Acronyms

ADC	Analogue-to-digital converter
ASDPE	Application Specific Domain Prototyping Environment
ASIC	Application Specific Integrated Circuit
BIOS	Basic input/output System.
C-HEAP	CPU-controlled Heterogeneous Embedded Architectures for signal Processing
CPU	Central Processing Unit
DAC	Digital-to-Analogue converter
DMA	Direct Memory Access
DSP	Digital Signal Processing
FIFO	First In First Out
FPGA	Field Programmable Gate Array
HW	Hardware
IC	Integrated Circuit
LM	Local Memory
LUT	Look-up table
PCI	Peripheral Component Interconnect
PLD	Programmable Logic Device
pSOS	Real-time operating system
RAM	Random Access Memory
SoC	System on Chip
SRAM	Static RAM
SW	Software
VHDL	Very High speed Integrated Circuit Hardware Description Language
WWICE	Window to the World of Information, Communication and Entertainment

Chapter 1

1. Introduction

1.1 Introduction

What is characteristic of the last two decades of the previous millennium is the shift from analogue electronics to digital electronics. More and better functionality and low-cost digital solutions have gradually replaced functions that used to be implemented with analogue components. The great amount of processing power that has become available in digital implementations as a result of the high integration density of modern ICs has even allowed for entirely new products with complex signal processing requirements. Complex functions that previously could not be implemented efficiently by means of analogue circuitry can now be implemented digitally in an economical manner, although analogue electronic implementations remain present in certain types of applications.

Because of the high integration density of ICs it is possible to map multiple processors and hardware IP on one chip, this is called a System on a Chip (SoC). But for reducing the risk of erroneous chips and a better re-use level, all subparts (component-level) and the complete integrated system (system-level) have to be tested. This thesis describes my research efforts for developing a prototyping environment board for system-level testing. This thesis deals with hardware/software systems, which are systems that contain both hardware components and embedded software. Such systems are typically found in applications domains such as tele and data communications systems, consumer electronic products, industrial control systems, automotive systems, and aerospace systems. In general these systems contain one or more processors to execute software and dedicated hardware components like ASICs (Application-Specific Integrated Circuit) and FPGAs (Field Programmable Gate Arrays).

The objective of this thesis is to provide solutions for problems that are related to testing and debugging of embedded (hardware/software) systems[1]. Since the introduction of IC technology and software programmable devices, we are facing an exponential increase in the complexity of hardware/software systems. Gordon Moore, co-founder of Intel, predicted in 1965 that the density of transistors on ICs such as memory chips and microprocessors would double roughly every 18 months. This prediction has been proven to be valid until today and is likely to remain valid in the near future. Handling this increasing complexity in hardware/software systems

[1] Although there exists no single and widely accepted definition of what an embedded system exactly is, it is generally characterised by heterogeneous implementation technology containing various hardware and software components, which performs dedicated functions in a larger 'host system'. This is the definition used in this thesis.

demands improvement of all technologies, tools and methods required for the design, implementations and verification of these systems. Consequently, there is a growing complexity gap between process (+58% a year), hardware (+21% a year), and software (+8% a year)-technology and the designer's ability to design and verify complex hardware/software systems.

As leading edge semiconductor manufacturing technology transitions from .35µ through 0.15µ and beyond, System-on-a-Chip is becoming reality. Improvements in process technology enable the fabrication of low power, high performance and, multi-million gate designs. This capability enables mass integration which results in the ability to fabricate System-on-a-Chip. The same technology trend that enables manufacturing of SoC is stressing the methods used in attempts to design highly integrated systems. The result is what we refer to as the widening Design Productivity Gap and is illustrated in Figure 1 below. The Design Productivity Gap means the industry's level of integration from the design side is far below the industry's manufacturing capability. As we try to integrate more and more, design cycles grow longer and longer, thus delaying the introduction of new products into the market.

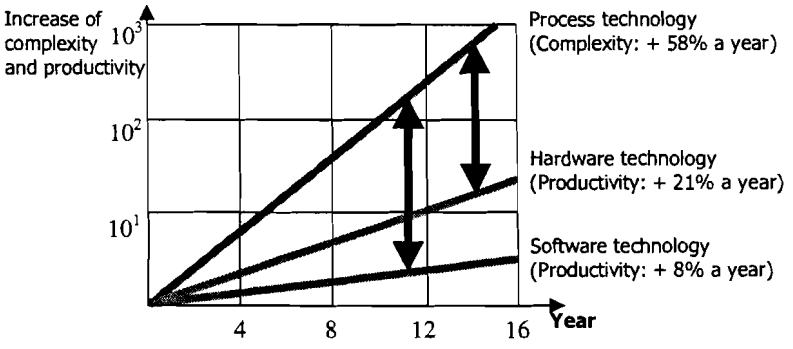

Figure 1 : Design productivity gap.

To fill the hardware productivity gap it is possible to design some new architecture, for example multi-processor architectures or add (big) embedded memories or reconfigurable components (flexibility in hardware). It is also possible to reuse proven hardware so there is time to develop other (extra) hardware. The key issue nowadays is to design and verify systems while meeting time-to-market constrains. The task of verification is to check the correctness of a system. Various techniques for verification are used in successive steps of the design process, such as simulation or formal verification of system models and testing of the system implementation. The focus of this thesis is on a prototyping environment board for system-level testing and debugging. System-level testing is defined as verifying the correctness of a system by applying test stimuli to the hardware/software implementation of the system and observing the response. System-level testing implies verification of the system to check the correctness of the system as a whole, with all its hardware and software components. System-level testing should yield a verdict on the correctness of the system. Whenever testing reveals the presence of an error, debugging is required to determine the exact fault mechanism in the system that caused the error. Hence, an improved method for system-level testing and debugging will directly result in reduced development costs and a shorter time-to-market.

The increasing difficulty of system-level testing and debugging is closely related to the increasing complexity of hardware/software systems. First of all, exhaustive

testing of a hardware/software system is generally impossible to achieve because the required number of test cases is astronomical. Second, testing and debugging a system through its external interfaces does usually not provide sufficient visibility into the internal operation of a system. Consequently, it may be very difficult to observe and control some specific parts of the system, such as the interaction between hardware and software components.

1.2 Objectives

The problem to be solved is that of implementing a circuit into a set of FPGAs which are combined onto one prototyping board possibly with some other hardware. This board provides a hardware environment that can be used either as a prototyping board, allowing to test circuit design, or as a hardware support for practically test the algorithms.
The objective of this thesis is to develop a generic prototyping environment, in terms of a generic method and a generic hardware platform that deals with the problems of system-level testing and debugging in hardware/software systems. I primarily concentrate on the design of hardware architecture on a PCI-board for verification of a hardware/software system. I do not pay any attention to problems related to test case generation.

The key element of my method is to improve the design for hardware/software systems in such a way that testing and debugging of the implementation is supported. The basic principle of my method is improving system-level testing. This provides an external tester/debugger in the system environment that can observe, control, and initialise the operation of the system for testing and debugging purposes. In my case the host processor is used as observer, controller, and initialiser for the internal operation of the system it also runs some part of the application.

This thesis describes a generic method towards prototyping of embedded systems used for a specific application domain that should fulfil the following objectives:

- The method should provide a solution to handle the complexity of system-level testing and debugging in hardware/software systems.
- The method should be fully Integrated into the hardware/software co-design process of the system groups
- I should demonstrate the architecture and methods in practice and provide experimental evidence that the method indeed improves the system-level testing and debugging of hardware/software system.

For everybody who reads this thesis, always remember:

No complex system can be optimum to all parties
(Eberhart Rechtin)

1.3 Rapid System Prototyping

Originally, the word prototyping is derived from the Greek word *protos-typos,* which means "first model". Webster's dictionary defines a prototype as "an original model or pattern from which subsequent copies are made, or improved specimens developed". Throughout modern history, people have frequently built a physical (scale) model before constructing the "real thing". Since a model permits cheap and fast construction, the respective concept or design can be analysed, demonstrated and tested without having to build the final object. Furthermore, changes can be incorporated quickly and at low cost. By means of these properties, prototyping (i.e. building and analysing a prototype) provides means to exclude errors and misunderstandings with respect to the final object. Consequently, prototyping can reduce and even eliminate expensive and time consuming rework on the final object.

This old reasoning is still valid for the design of modern electronic systems. Within the context of electronic design, a prototype provides a functional equivalent of (a part) the final application, i.e. the prototype behaves in the same manner as the target implementation. By means of this behavioural equivalence, the functionality of an electronic system can be assessed in an early stage of the design. Additionally, the functionality can be changed rapidly due to the programmable nature of most prototype implementations. In addition to an early availability and a flexible implementation, the observability of a prototype implementation is in general superior to the target implementation whereas the functionality can be verified much more quickly than by means of simulation.

Due to the complexity and the short time-to-market of embedded systems prototyping is required. Furthermore, due to the cost requirements and timing constraints of systems, application-specific hardware solutions are often needed, this makes the co-design of hardware and software a topic for the design automation of embedded systems.

A lot of different approaches for simulation have been developed, hardware can be simulated at different levels (e.g., electrical circuits, logic gates, or register-transfer level) or behavioural VHDL descriptions.
To close the gap between specification and implementation and to support the co-design and validation of large embedded systems, rapid prototyping with configurable and programmable hardware/software systems is allowing to validate the complete system in a (almost) real time environment [4]. Because of this the observed testing time in a prototyping system is longer than when using simulation. Furthermore the implementation of the system under development is executed, with the software parts running on processors and the hardware parts are mapped on a FPGA.

Today's digital IC engineers grapple with:

- Design Productivity Gap (already explained in paragraph 1.1)
- Verification Dilemma

Since the hardware and software behaviour together defines the system, both must be verified as conformant and compatible with existing systems, also known as the verification dilemma. A new style is clearly needed to enable proper verification of systems and to reduce the cost associated with this.
There is a need for a better testing method before releasing the product. But it is impossible to completely eliminate bugs before the SoC is released. The problem is really that when a product is released and errors do occur (at a consumer's home), there are no really good ways of discovering what went wrong and how to fix it. It is possible to report the problem to a programmer/IC-designer, but typically that person does not have very good ways of finding out what happened either. At this point it is necessary to go back to the simulation level which takes a long time (or probably impossible to test the application for 100% (see Table 1)). That's why some bugs do not get fixed and a prototyping system is needed. See Table 1 for more debugging times [4].

Table 1 : Verification Dillema.

technology	Debug (Hrs)	Debug time
Prototyping	1	1 Hour
FPGA	10	~ 1 working day
HW Emulator	100	4 Days
Throughput Model	1000	1.4 Month
Transaction Model	10000	1.2 Years
Cycle Accurate Sim. Model	100000	~12 Years
RTL Model	1000000	> 1 lifetime
Gate level Model	10000000	~1 Millennium

During the embedded system development the next points must be considered, see also Figure 2:

- Prototyping environment development (System Level Architecting group)
- Iterative process of system partitioning and concurrent software development (System groups)
- Cost effective hardware implementation (IC group)
- System verification (System group)

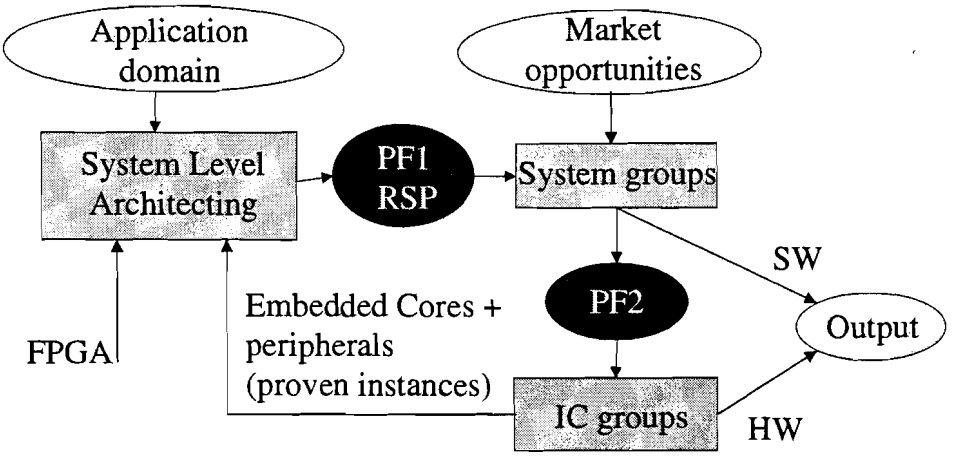

Figure 2 : Rapid System Prototyping.

The System Level Architecting Group is operating in a certain application domain. For this application domain they develop an application specific platform (PF1) for Rapid Silicon Prototyping (RSP). Furthermore they describe a set of rules for using this platform, this is for supporting the System Groups. One of the inputs comes from the IC groups that already developed cost effective hardware, which is re-used from other projects. The other input are the FPGAs which are available on market. The System Group hands over the platform to the IC group which can use this platform for developing a new IP block.

If a new product has to be developed, driven by market opportunities, the System Groups use the RSP-platform provided by the System Level Architecting group. An example is shown in Figure 3.

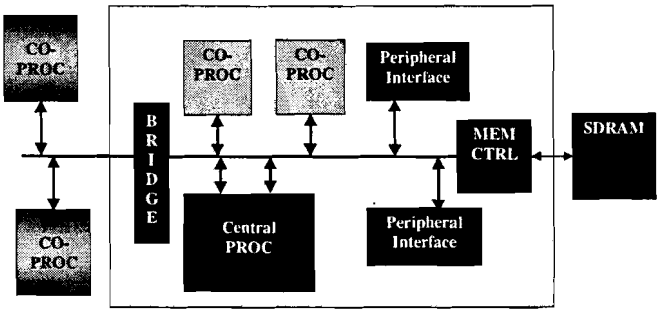

Figure 3 : Platform for Rapid Silicon Prototyping.

Most systems are too large to fit on a single FPGA or whatsoever, that's why a system must be partitioned so that it will fit into the allowed FPGA (or co-processor). Notice that when systems are partitioned, resulting communication delays, limitations on interconnection size, and speed of an FPGA must be taken into account because these are affecting performance. These are some reasons why a prototyping platform (probably) can not run on the desired frequency.
When the System Groups gets the RSP-platform they can start by making a functional description in C of the design. Because some parts of the design does not met the specified performance and other requirements (like power consumption) these must be mapped onto hardware, while some parts can be mapped onto a processor, that is why a partitioning have to be made. This partitioning is an iterative process and can be very time consuming. During the partitioning the software (e.g.

everything in C, or a part in C and a part in VHDL or Verilog) is written, this high-level source code may have design errors that were introduced by a designer. If everything is written in C, some parts of the functional specification (functional specification is never cycle true because the specification is focussed on functionality and not on performance) must be mapped onto FPGAs. The C-code must be transformed to VHDL (this can be done manually or automatically with an Art-compiler[9]). When the mapping is done the debugging can start. The proven instance does not have to be tested, only the software (on the programmable processor) and the new hardware have to be tested. When all tests are passed the platform for RSP can be used as a demonstrator. There are still undetected design errors even when the complete system is debugged, but there are significantly fewer bugs than debugging without an RSP platform.

The output of the System Groups is the tested software, which is divided into software for the programmable processor and software (functional specification) which have to be converted into cost effective hardware by the IC Group. Now it is possible for the IC Groups to make a cost effective (requirements are met) SoC or component in a sort time. This hardware part is the output of the IC Groups, when the new hardware is debugged this IP block can be re-used in future applications.

The design of new IP blocks must always be done with re-use and an application specific domain in mind because if the design is done with re-use in mind this IP block can be added in another application without any trouble. In this way the hardware productivity gap can be filled. Furthermore an application domain must be known for the performance optimisation.

When the software is completed and the IC-Groups implemented a cost effective IP block it is known which IP blocks, on the RSP platform, are un-used and can be de-configured, as shown in Figure 4.

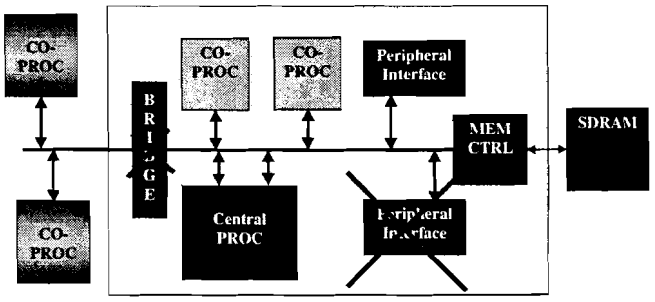

Figure 4 : De-configuring of un-used IP.

At last the developed IP can be mapped with the proven IP to one System-on-a-Chip.

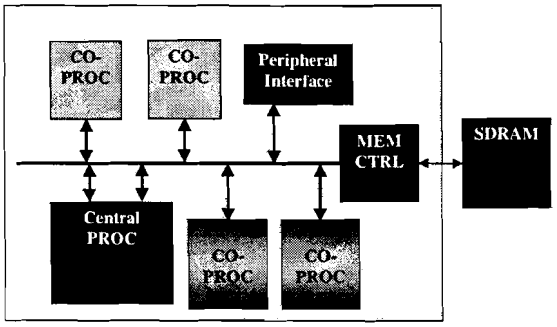

Figure 5 : IC Design.

The SoCs that are manufactured contain manufacturing errors caused by spot defects and parametric errors. Manufacturing tests such as stuck at testing are used to separate the good dies from the bad ones [3]. If the manufacturing test is not perfect (because the fault models are not accurate enough, for example), some faulty devices may pass the manufacturing test. Manufacturing tests are used to check whether realisations of ICs are consistent with their netlist. They do not check whether the netlists are correct. If, for a given design, all the devices that have passed the manufacturing test, fail in the application, the design probably contains an error (because design errors are always present on all chips, yet manufacturing tests does not catch them).
We are left with devices that may contain undetected design errors and undetected manufacturing errors that only become visible in an application. A silicon debugger can be used to locate these manufacturing errors electrically, it is recommended to read [3] for more information about Silicon Debugging.

The advantages of a Rapid Prototyping System are:

- Design works real time.
- Provides a platform for early implementation and verification of the system.
- Can serve as an architectural template that enhances design re-use and reduces the effort required developing a system.
- A standard interface stacks is used on the RSP platform for the communication architecture, this stack is useful for improving the re-usability of the components.
- Fast design of a cost effective one chip solution.
- Long term partnership between the groups.
- Delivers a demonstrator for the System on Chip.
- Can quickly react to changing customer environment or requirements.
- Systems are too complex to simulate real-world operation in "bounded" time (need to build to test).
- Customers won't put up with unreliable products.

Some disadvantages of a Rapid Prototyping System can be:

- Not the same performance as final product (slower).
- Not the same size as final product (bigger or more ICs required).
- Prototype more expensive than final production unit.

In summary, prototyping of a part of an embedded system under design, can lead to three primary design benefits:

1. Reduction of IC design risk
2. Early system integration
3. Improved parallel development of hardware and software

Prototyping allows for a reduction of design risk, as it facilitates an early and thorough functional verification and validation of a new defined hardware part via high-speed execution of an observable prototyping implementation.
The other two benefits basically allow for a speed-up of the design process by means of concurrent engineering. By using prototyping, system integration can be carried out during design instead of after fabrication. Likewise, software testing does not have to wait upon availability of the 'first silicon' implementation of the target. Prototyping allows to test the execution of software programs on programmable hardware combined with application specific hardware. It also allows evaluating the trade-offs that have been made during the design (such as hardware/software partitioning) before sending the design to a foundry.

1.4 Thesis organisation

To conclude the introduction, this final section describes the organisation of this thesis. This thesis is divided into seven chapters and one appendix.

- **Chapter 2** named "Prototyping platform" describes why it is needed to develop a Rapid Silicon Prototyping board and an introduction about the used PCI board. Furthermore, a very small introduction is given about FPGAs and the Local Bus protocol that is used.

- **Chapter 3** named "Introduction C-HEAP" is an introduction of the used synchronisation protocol (C-HEAP) and a programming model is given.

- **Chapter 4** named "C-HEAP Synchronisation hardware" describes C-HEAP in full detail. A full description of the complete developed architectural template is given. And as last the initialisation and how to use is fully described.

- **Chapter 5** named "Architectural template" decribes the developed architecture in detail. Furthermore it is described how to initialise and use the architecture in association with the PCI board.

- Finally in **chapter 6** the conclusions and recommendations are discussed. Also, further performance improvement is discussed and how this can be accomplished.

The appendix is a description of the Local bus protocol.

Chapter 2

2. Prototyping platform

2.1 Introduction

Large and complex embedded systems are often composed of multiple and heterogeneous processing blocks. Some of these blocks may be involved in computations, where hardware implementations in ASIC may provide the best speed performance, and FPGAs provide easy programmability, flexibility, and also good speed performance. Other blocks may need to handle (complex) control flows or multiple standards (H263, MPEG, etc.), where a software programming approach is most flexible and cost effective. Therefore, in contrast to pure software programming or pure hardware implementations, an alternative approach is to partition the system into software and hardware sections. Each selection can be implemented using respectively software or hardware technology. For the hardware technology an FPGA-Board (described in section 2.2) can be used for functional verification of the designed hardware. Because the hardware must communicate with software and other hardware there is a need for a synchronisation protocol (introduction of a synchronisation protocol (C-HEAP) is given in chapter 3). The software can be executed on the host-processor.
Since hardware and software interact with each other they cannot be designed independently. A new design methodology must be developed to develop hardware and software concurrently to meet specified performance and cost constraints. This is known as hardware/software co-design.

2.2 FPGA-Board

The verification procedure of a SoC becomes increasingly more important and time consuming. One of the verification methods is to use the PCI-board which is developed by the Storage Systems and Applications group, although this board is not developed for SoC prototyping it can be used for this purpose. With an FPGA prototyping board, it is possible to verify the functionality of the SoC at much higher speed compared to software simulation. Furthermore, it is possible to test the complete SoC at a system level, which includes external memory, other ICs and I/Os before the SoC is fabricated into an IC.

This paragraph describes a prototyping board (this board is taken from [1]) based on Altera FPGAs. This board is specially developed for recording and playback of (partial) DVB transport streams. Combined with the PC it formed a storage device for the WWICE (Window to the World of Information, Communication, and

Entertainment) project. The task of the PC is to record and playback (also concurrently) of multiple streams. Furthermore a stream has to be recorded to a harddisk of the PC using the IEEE 1394 interface. The infrastructure of the PCI board is especially developed for receiving data from IEEE1394 input. There is no DMA protocol implemented and that's why a polling based communication protocol is used. This means that the software checks the status of the on-board memory of the PCI board on a regular basis. If the memory is getting full it reads this data. DMA would enhance the performance of the complete system.

Each of the PCI boards have four re-programmable FPGAs on board, on which the designed VHDL has to be mapped, see also [1].

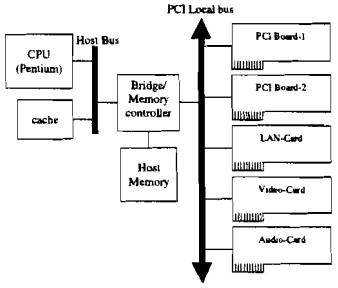

Figure 6 : Prototyping hardware set-up.

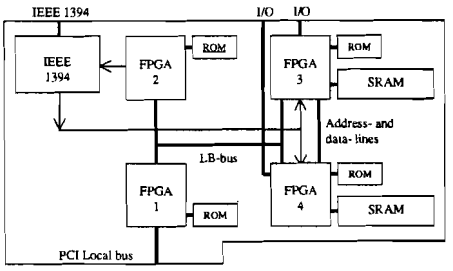

Figure 7 : PCI Board.

FPGA-2 contains some registers for configuring the IEEE 1394 protocol. FPGA-2, -3, and -4 are available for use, FPGA-3 and -4 are also connected with a 16 bits wide bus, and it is possible to specify a communication protocol yourself.

FPGA-1 implements a PCI-to-Local Bus[2] Wrapper, this interface currently supports either master or slave operation, but not both. Since the CPU at start-up programs the hardware, we need at least a slave interface. Consequently, the PCI card cannot act as a master on the PCI bus, which means that it cannot access data outside the board. This is not a problem for now since it can still access its internal RAMs.

Every PCI board has 6 *base addresses* which are used to access each of the four FPGAs, and the two RAMs (see Figure 7), this is done to be able to address every FPGA in a easy way.
All operations are performed memory mapped, that's way there is a need for mapping the assigned PCI base addresses (done at PC start-up) to user space base addresses. In this way, all reads and writes to the devices are done as if they are ordinary memory operations (e.g. memcpy), which is exactly what we want. To do this mapping, the system function *mmap* is used. This function asks to map a piece of memory space (expressed as a number of bytes) onto user space and returns a pointer to the beginning of the mapped area.

For the ease of implementation, we may want to map one hardware device on one FPGA, and let the devices communicate via the local bus. However, the operation of the local bus is such that all transactions involving any of the FPGAs on the board is always to or from the PCI interface. It is not possible for two FPGAs to communicate with each other. This limits the number of hardware devices we can put on a PCI card to one if they have to communicate.

[2] See appendix A.1 for Local Bus protocol.

There are many memories distributed throughout the PCI board, every FPGA has 20kbits, furthermore there are 2*1MB of memory. On-chip memories cannot be under-used because it are the on-chip memory that determine the overall chip size. The computational units only represent a small(er) percentage of the total chip area. Ideally, the mapping of an application on a multiprocessor chip should therefore always be memory-limited. Off-chip memory bandwidth (amount of information per second) is expensive and power consuming and is already a major bottleneck in nowadays systems. Therefore, on-chip communication will play a very important role in the future. As can be seen in Figure 1 it is possible to develop much bigger chips than they can utilise, this remaining part of gates can be filled by memories or re-configurable hardware components. FPGA-1 (see Figure 7) contains a PCI initiator/target interface needed to communicate with the CPU and other cards.

The PCI Board already developed:

- Communication on-board is with LB protocol (see Appendix LB), which is not a standardised protocol. Adhering to a standard ensures easy reuse and better chances to get readymade interfaces for variety of busses;
- Especially made for one application with partial DVB streams;
- FPGAs can not communicate with each other.
- FPGAs are to small (100k gates)
- The board only can be target

Since embedded systems have become increasingly digital, components that interact with the analogue environment, such as analogue-to-digital and digital-to-analogue converters (ADCs and DACs), can be necessary. At the already developed FPGA-board these ADCs or DACs are missing. Although not every SoC needs these components it can be useful to have some PCI-Boards with only ADCs and/or DACs on it.

2.2.1 FPGA

The meaning of this paragraph is to do a very short general introduction about FPGAs. A PLD (Programmable Logic Device) is a (re)configurable hardware component that can be programmed on-side by the end user without involvement of the manufacturer of the device.

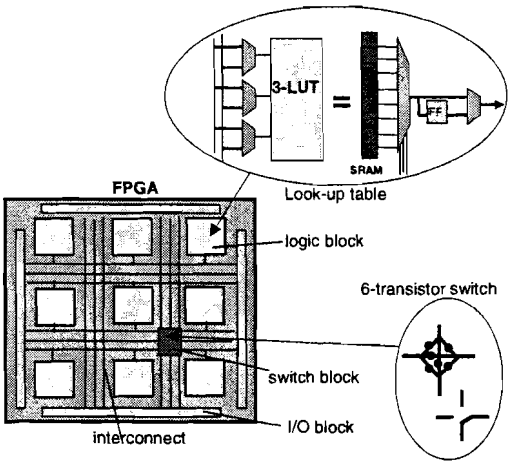

Figure 8 : FPGA.

All Altera families (used by [1]) consists of an array of logic blocks embedded in a configurable interconnect structure and surrounded by configurable I/O blocks.

Figure 8 shows a three-input look-up-table (LUT) or *function generator*. A lookup table implements combinatorial logic as a 2^n x 1 memory, composed of configuration memory cells.

The memory is used as a lookup table, addressed by the *n* inputs. When the FPGA is programmed, the memory (LUT) is loaded with the bit pattern corresponding to the truth table of the function.

The switch blocks controls the connection of wiring segments in the programmable interconnect, connects a one column interconnect with one row interconnect. The switch bock is a pass transistor controlled by a configuration memory cell. Wire segments on each side of the transistor are connected or not depending on the value in the memory cell.

The logic blocks contains a single lookup table with a maximum of 8 input signals and 16 output signals.

For further information and datasheets : www.altera.nl

2.2.2 Local Memory

There are two memories assembled on the PCI board that are together 1MB of memory. Both memories are 32 bits wide (4 Bytes). As a consequence of this there is a need for an 18 bits address bus, 17 bits for the appropriate memory line and 1 bit for the appropriate memory. Possibly 2 bits are needed for selecting the appropriate byte from the 32 bits (4bytes). In total $2^{17} * 2^2 * 2^1 = 2^{20}$ = 1MB can be addressed.

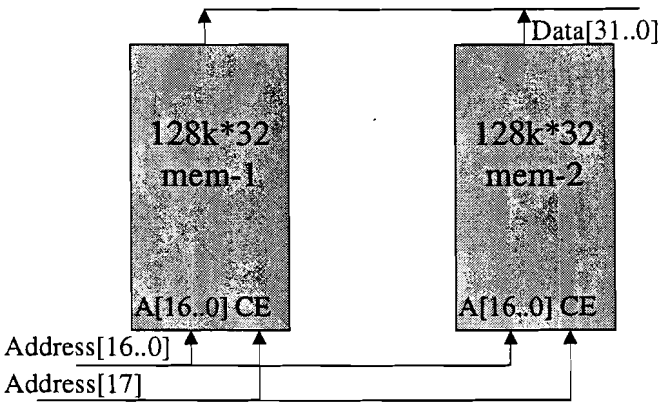

Figure 9 : Principle of memory design.

Chapter 3

3 Introduction C-HEAP

3.1 Introduction

The C-HEAP protocol is intended for signal processing applications where infinite streams of data have to be processed. It is based on a *Kahn*-model [2], which is often referred to as *dynamic data flow*. In *Kahn process networks*, a number of concurrent processes communicate by passing data *tokens* through unidirectional FIFO channels, where writes to the channel are non-blocking, and reads are blocking. This means that writes to the channel always succeed immediately, whereas reads block until there is sufficient data in the channel to satisfy them.
In this model, the overall function is decomposed in a number of parallel tasks communicating via *point-to-point* channels with first-in-first-out (FIFO) behaviour. No data can be lost on these channels. But in practice only finite FIFO channels can be made, that's why a task will block when a task wants to read data from a channel, and there is no data available. A task will also block when it wants to write to a channel if the associated FIFO is full. Synchronisation between the tasks is derived from the status of the FIFOs: a blocked reading task is unblocked when the writing process writes on the channel and a blocked writing task is released when the reading task reads from the channel. This synchronisation takes place on a token basis. Although a token is unit of synchronisation, the amount of data associated with a token can vary.

There is chosen to clearly separate synchronisation from communication:

- Communication is the set of activities required to transport the data from one task to another;
  - Synchronous if the write and read operations must occur simultaneously.
  - Asynchronous, otherwise. In this case there can be a finite or and infinite number of buffer locations, where the information is stored between the instant in which it is written and the instant in which it is read.
- Synchronisation primitives are used to notify other task of the input/output activity of the calling task, and blocking if channel empty/full.

This separation of inter-task synchronisation and data transportation provides flexibility to perform data transportation at a different granularity than synchronisation. This separation is especially advantageous in shared memory architectures since only synchronisation primitives are needed, no copying of data is required.
Synchronisation is probably the most difficult and error-prone type of programming for multi-processor architectures that exists. It is difficulty arises because it involves the understanding of the potential simultaneous actions of multiple processors. The huge number of possibilities to consider is beyond the capability of most people. Two forms of synchronisation occur in concurrent programs: *mutual exclusion* and *conditional synchronisation*. Mutual exclusion is concerned with ensuring that *critical*

sections of statements that access shared objects are not executed at the same time. Conditional synchronisation (like C-HEAP) is concerned with ensuring that a process delays if necessary until a given condition is true. This is used to ensure that a message is not received before it has been sent and that a message is not overwritten before it has been received.

3.2 When is C-HEAP useful?

Multimedia applications require the execution of a large number of tasks on a variety of multimedia data. The amount of processing power demanded by such applications is very large. The performance of single processor solutions exploiting fine-grain instruction-level parallelism is insufficient to meet such demands. Much higher performance is obtained by making use of coarse-grain task-level parallelism. This enables parallel execution of large independent tasks and can be exploited in powerful multiprocessor solutions.

Depending on the application, task parallelism can be exploited when two or more programmable processors and/or hardware blocks are combined into one system, also called a heterogeneous multi-processor system. Sometimes it is even needed to map the application on a multi-processor because then the requirements can be met.

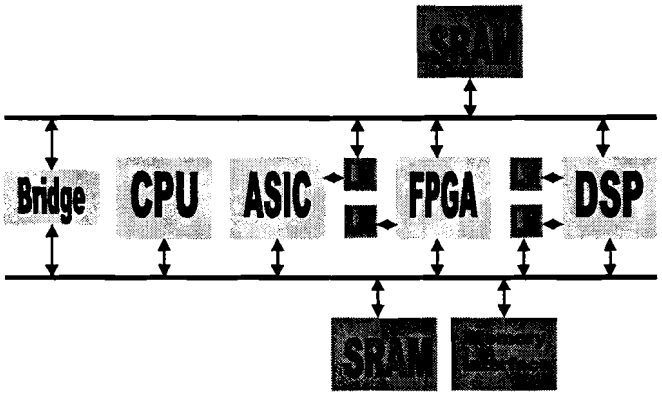

Figure 10 : Heterogeneous multi- processor system.

If there is only one task at the processor or hardware there is nothing to synchronise and C-HEAP is not useful at this moment. But when multiple processors and/or hardware blocks are combined they have to transfer data between them, so there is a need for a synchronisation protocol and a communicating protocol. For this synchronisation C-HEAP is used.

But why do we need hardware accelerators?

A programmable processor[3] (PP) is very flexible but very inefficient, see Table 2 at page 18 what this means. But on the other hand hardware IP is very inflexible but

[3] Programmable processors can be subdivided into general purpose CPU-cores (e.g. ARM and MIPS) and programmable DSP-cores.

very efficient and also cheaper. It is possible to exchange some flexibility into some efficiency and lower cost, this is possible with mapping some tasks on hardware IP.

The *degree of adaptability* or *flexibility* characterises the scope of reusability. Closed models cannot be modified before the use; their structure and behaviour is fixed forever. If the designer wants to use a closed component, it must adapt the surrounding parts of the system to this component. Generic models tend to be open for the modifications/adaptations concerning the processing width or internal delays. In principle the generic models preserve their initial function provided by the model developer. The configurable models are still more flexible. In addition to the adaptations provided by the generic models, the configurable models offer some degree of functional adaptability through the selection of the appropriate sub-components and their internal architectures. However the external interface of a configurable model stays fixed.

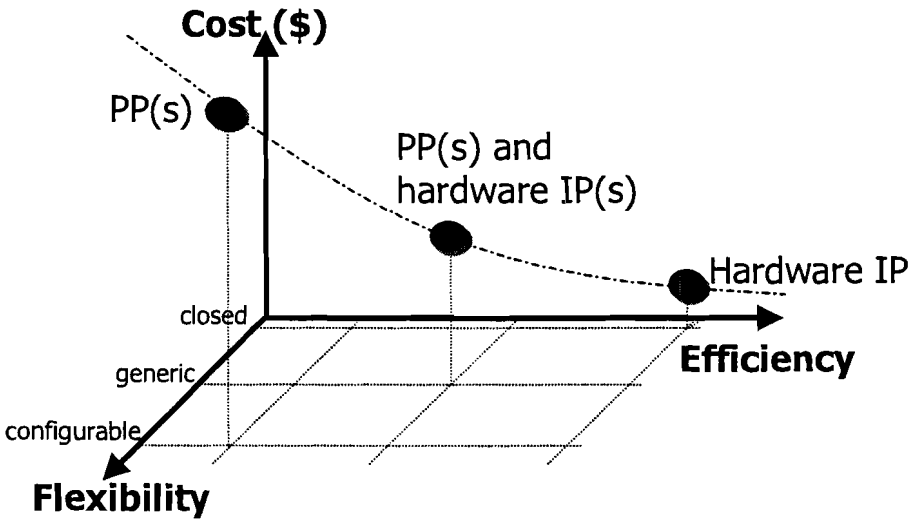

Figure 11 : 3D-Design Space.

Normally you do not want to exchange all flexibility into efficiency because your system is not flexible (future proof/scalable) anymore. What you want is an optimal combination between flexibility, efficiency, and cost. This can be reached through a combination between (multiple) programmable processors and (multiple) hardware accelerators. A hardware accelerator can be seen as a hardwired implementation in an ASIC or programmable FPGA device. These hardware accelerators are some dedicated pieces of hardware with only one configuration on one device, but on the other hand programmable processors can have many configurations on one device.

What is meant with flexibility, efficiency, and cost can be seen in the next table.

Table 2 : 3D Design Space.

Flexibility	Efficiency	Cost
Field updates Standard upgrades Multipurpose Adaptability Scalability	High speed Low power Small area High bandwidth	Reduced development costs Reduced manufacturing costs Early and fast designs(*)

(*) see Figure 12.

ASICs or FPGAs may provide the best speed and power performance (compared to a programmable processor), other blocks may need to handle complex control flows or communication protocols, where a software programming approach is most flexible and cost effective. Since hardware and software blocks interact with one another, they cannot designed independently. At this moment a new design methodology is rising, were the hardware and software blocks are developed concurrently to meet specified performance and cost constraints.

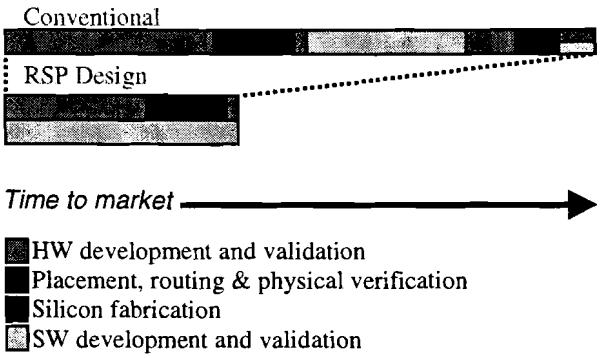

Figure 12 : Time to market.

A problem is to find a way of dividing an application into multiple tasks. The granularity of tasks depends heavily on the application domain. Normally, granularity is chosen such that the complexity of each task is manageable for the application designer and that efficient re-use of tasks and modular design is possible. This can be achieved by making it an explicit part of the application specification process. As it turns outs, tasks are often a natural way of expressing functionality. Normally designer's starts drawing block diagrams, connecting the blocks with arrows that represent information data flow. In a later stage, the design is refined by expressing the blocks themselves as a network of lower level blocks. When this is done you can determine which part can be best mapped onto hardware or programmable processor.

3.3 C-HEAP programming model

Because a high-throughput process requires large amount of data, buffering of data on communication channels is an important issue. Due to the buffering it is possible to separate different tasks, so high parallelism can be achieved. Due to bandwidth limitations such buffering cannot always be implemented in shared background memory. A stream concept implies that the order of the communicated data is preserved, a natural implementation of a communication channel between tasks is a FIFO buffer.

As can be seen in Figure 13 between tasks are FIFOs mapped onto channels. For C-HEAP it is not important whether these tasks is a hardware accelerator or a piece of software mapped on a programmable processor.

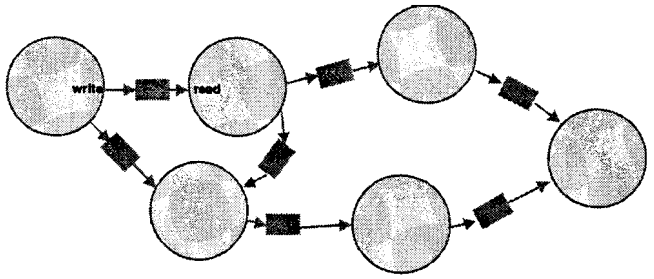

Figure 13 : C-HEAP Programming model.

A task writes its processed data to a token in the FIFO and has to notify the other task. Once the tasks are synchronised the other task can read the data from the FIFO. Of course the FIFOs are bounded, this means that they have a finite number of tokens. The scheduling is based on availability of empty and filled tokens. This means that a task blocks if there is no free token to write the data to (FIFO completely filled), or when there is no more data to read (FIFO completely empty).

3.3.1 Synchronisation

As can be seen in Figure 14, the tasks have to read the data from the input-channel, at the left hand side. Notice that the data is **not** read from the other task but from the FIFO in the channel, because of this the task does not has to know whether the communicating task is on a programmable processor or on a hardware accelerator. After the data is read the task puts the emptied token back into the input-channel. When the task processed the data, it writes this data in the requested empty token from the output-channel (right hand side).

In a nutshell, a task has to perform the following synchronisation actions to get input or to produce output. To get input data from an input channel, a task has to synchronise with the producing task on the input channel. This is done using the

primitive *chp_get_data* on the input channel. This primitive will return a reference (pointer) to the oldest not yet consumed data send by the producer. Note that this primitive will block when the FIFO on the channel is empty. When *chp_get_data* "falls through", the reference to the data can be used to process the data. When the data is completely used, the task has to notice that the data space (buffer) can again be used by the producing task. This is done by the primitive *chp_put_space* on the input channel. Note that *chp_put_space* does not block. On an output channel similar primitives exist: *chp_get_space* to acquire a reference to a buffer which can be filled and *chp_put_data* to notice the consumer at the other side of the channel that data has been produced. The *chp_get_space* primitive blocks when the FIFO is full, the *chp_put_data* primitive will not block.

Notice that it is possible to request a token even before the task can handle it. It is also possible to request tokens and handle them out of order. But in both cases the tokens must be released in FIFO order.

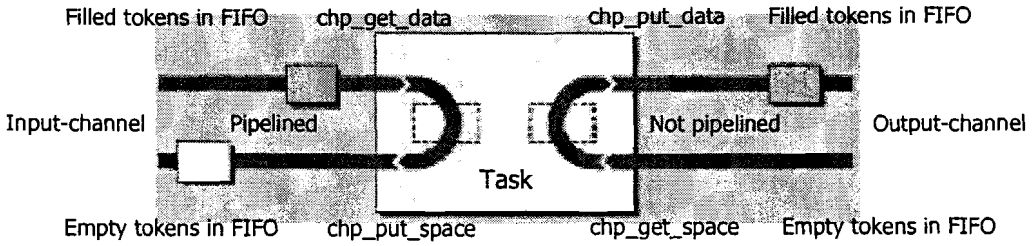

Figure 14 : Synchronisation.

For the input-channel :

1. Get a full token from the input-channel (chp_get_data);
2. Read data from this token for processing;
3. Put the empty token back onto the input-channel (chp_put_space).

For the output-channel :

1. Get an empty token from the output-channel (chp_get_space);
2. Fill token with the processed data;
3. Put the filled token back onto the output-channel (chp_put_data).

Notice that one complete synchronisation transaction is a chp_get_space with *chp_put_data* and a chp_get_data and chp_put_space action.

As can be seen in Figure 14 it is also possible to have a degree of pipelining. At the left hand side the task can process the data when the other communicating task again refills the emptied token in the channel. And at the right hand side two tasks alternately can fill or empty the only token in the FIFO, so no parallelism can arise.

3.3.2 C-HEAP Synchronisation in Shared memory

In the previous paragraph I talked about a FIFO that is mapped on a channel, what does this mean?

This means that the FIFO is mapped onto a channel which is mapped onto a shared memory, this has to be a shared memory space for the communicating tasks because if one tasks writes data into the FIFO the other tasks must be able to read this data. This shared memory can be any memory that can be accessed by both tasks.

In Figure 15 can be seen that during the initialisation process a FIFO with two tokens is defined in the shared memory. When the producer (P) has some data for the consumer (C), C-HEAP delivers the producer a pointer to the first free FIFO token, shown in Figure 16. When the producer has a valid pointer(1) it fills the token with the processed data(2). When the token is filled the producer puts the filled token back onto the output-channel (put_token), the consumer gets notified that something is changed in the channel.

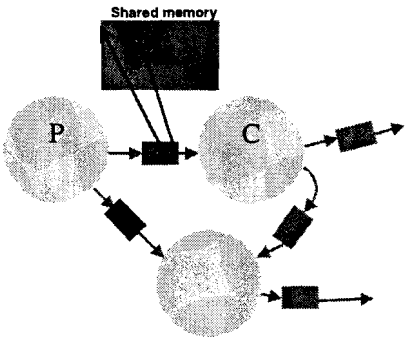

Figure 15 : Initialisation and start state.

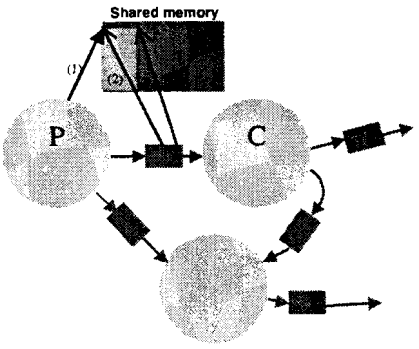

Figure 16 : Get_space, fill space, and put_data.

At this point it is possible for the producer to fill the next token (not shown). When the consumer is woke-up by C-HEAP it can read the data from the token(3).

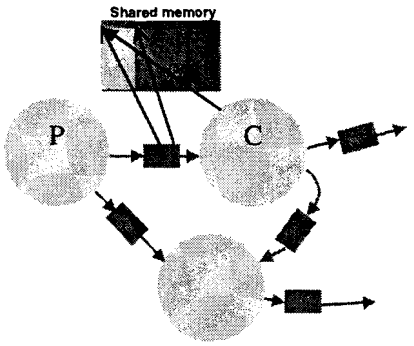

Figure 17 : Get_data, read data, and put_space.

When the task read all the data it put the empty token back into the channel (Figure 15). If the second token is filled and the consumer does not free some tokens the producing task is blocked(!), on other hand when the FIFO is empty the consumer task is blocked(!).

3.3.3 C-HEAP Architectural template

To obtain high performance, concurrent execution of tasks is exploited by using multiple autonomous processors (programmable or accelerators) that can perform tasks independently, and thus in parallel with other processors, see Figure 18.
The template consists of two main parts. First, a general purpose microprocessor (CPU) is responsible for control oriented tasks, such as the interaction with the environment, and possibly low to medium performance signal processing tasks. Secondly, a number of application domain/application processors (mapped onto an ASIC or FPGA) implement the time or power critical tasks.
As stated earlier tasks can be mapped onto e.g. CPUs, DSPs, ASICs, or FPGAs because of that it is a heterogeneous architecture. Also one or more bus structures can be implemented, these busses can be of different types. If one wants to use C-HEAP there is a need for a shared memory between the processors which have to communicate with each other. These shared memories can be distributed. Furthermore local memories are allowed.

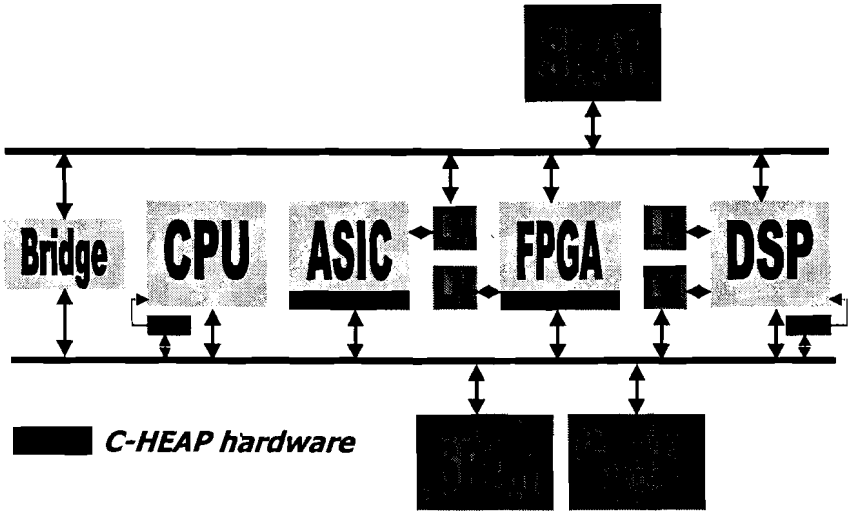

Figure 18 : C-HEAP Architectural template.

The hardware implementation of the synchronisation control can be centralised that means one controller is used for all devices or can be distributed that means every device has it is own controller. There are advantages and disadvantages of both schemes, which are discussed below.

Centralized control:
+ Only one controller is needed to be instantiated, may result in small area
- Complexity of controller grows as number of clients increases because frequency of synchronisation calls increases
- Latency to synchronisation actions may increase when synchronisation network is shared.

Distributed control:
+ Easy addition of the number of clients
+ Distributed and simple control
+ No sharing of controller results in lower latency
- Controller need to be instantiated in every device, may result in large area

Due to the positive points distributed control is implemented for C-HEAP protocol.

Chapter 4

4 C-HEAP synchronisation hardware

4.1 Introduction

This chapter describes C-HEAP in all forms, this means how C-HEAP has to communicate with other C-HEAP Blocks in hardware or software. In section 4.2 the C-HEAP Task Shell is described, section 4.3 gives an overview of the C-HEAP pSOS/MIPS software. In the last section of this chapter C-HEAP Block is described.

4.2 C-HEAP Task Shell

In multiprocessor systems everything runs in parallel, if one processor finishes its job, the data or a pointer to that data is stored in one of the FIFOs tokens. In this model FIFOs are bounded, this means that a task will block when someone writes to a channel if the associated FIFO is full. The other parallel task should read the buffer from the FIFO without communication with the originating task. Since two concurrent tasks modify the same buffers, access to that buffer needs to be atomic either by means of atomic read-modify-write or guarded by semaphores. A fine-grained atomic action is one that is implemented directly by the hardware on which a concurrent program executes. C-HEAP takes care of the synchronisation of tasks. If data is stored in FIFO buffers the size of the buffers should be properly determined, based on the volume of expected traffic. If the buffers are too small, and the rate of incoming or outgoing cells is to high, the FIFO may over/under-flow. Only one FIFO buffer means that the tasks have to be executed alternately, whereas more than one buffer allows pipelining of the tasks. The FIFO can be accessed by the producer and the consumer because the FIFO is in the address space. Furthermore, for efficiency reasons, no dynamic memory allocation is performed during operation. All communication memory is allocated at set-up and is re-used during operation. This implies the re-use of physical channel buffers during operation.

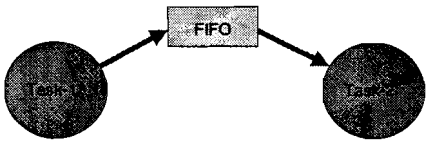

Figure 19 : Principle of C-HEAP.

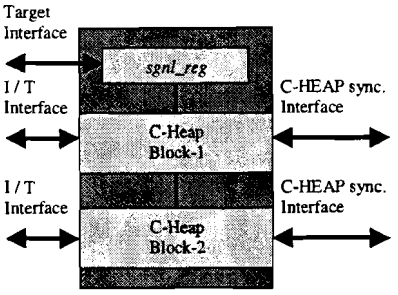

Figure 20 : C-HEAP Shell.

- Every hardware task needs one C-HEAP Shell for synchronis4ing.
- Every task at a programmable processor needs one channel record (explained later in detail).
- A C-HEAP Task Shell only has one signalling register (*sgnl_reg*) in spite of the fact that it may contain more C-HEAP Blocks. This *sgnl_reg* is used to signal the device about available tokens, this is for waking up the C-HEAP synchronisation controller.
- The C-HEAP Block requires system bus accesses to access administrative information from the other devices.
- All C-HEAP Block registers are mapped in a shared address space.
- A C-HEAP Channel is a point-to-point channel.

Synchronisation between the tasks is derived from the status of the FIFOs: a blocked reading task is released when the writing process writes on the channel and a blocked writing task is released when the reading task reads from the channel. This synchronisation takes place on a per token basis. Although a token is unit of synchronisation, the amount of data associated with a token can vary. In C-HEAP, communication and synchronisation are clearly separated where, in this context, communication is the set of activities required to transport the data from one task to another and synchronisation is set of activities required to inquire and notice the data transport actions to other tasks.

In our implementation, it is allowed to claim (reserve) a number of tokens and process them out of order before releasing them in FIFO order. This means that multiple claim primitives can be called before the corresponding tokens are released by release call. Note that in case, the number of buffers on the channel should be greater or equal to the number of consecutive claim calls, otherwise deadlock occurs.

4.3. Overview of the C-HEAP pSOS/MIPS software

This section gives an overview of the C-HEAP software, which was originally intended to run on a MIPS processor running the pSOS operating system.

C-HEAP tasks that are mapped on the MIPS are implemented as pSOS tasks. It was chosen to give the tasks different priorities, which are assigned, in descending order of creation (in pSOS[4]). The scheduling policy used by pSOS is priority based, pre-emptive, with time-slicing among tasks with equal priority.

There are four synchronisation primitives in C-HEAP: two for claiming buffer data and space (CHP_get_data/CHP_get_space), and two for signalling data and space (CHP_put_data/CHP_put_space). A task may be blocked while trying to claim data or space, and it can be unblocked by the communicating task when it signals the availability of new data or space. This unblocking mechanism may be implemented either by interrupts or polling of the buffer status. The interrupt mechanism for different hardware/software producer/consumer pairs are discussed in the next sub-paragraphs.

4.3.1 Software – software

The synchronisation between two software tasks is done through a pSOS message queue. This queue has depth 1 and is solely used for this purpose, i.e. the content of the queue is of no importance. A task can unblock another task by writing into the message queue. In C-HEAP, the synchronisation interrupts may be *masked* if the receiving task is able to continue processing (i.e. is not blocked on a read/write). This is done to avoid unnecessary interrupt overhead. In software, this is realised by writing into the queue regardless of its fullness (and ignoring the returned error code in case it is full). If the other task is not blocked on this queue, it will not notice this action.

4.3.2 Software – hardware

A hardware task is unblocked by writing the channel ID to the signalling register in the C-HEAP Task Shell of a hardware task. Since this register is memory mapped and is visible in the global memory map, a simple memory store operation is enough. The hardware task will ignore this event if it can continue processing (masked interrupt).

[4] In pSOS, the tasks are created and started immediately with one system call. This is done when the software C-HEAP task is started. However, from the C-HEAP point of view, the tasks are created much earlier. In this step, the memory is allocated to hold the task information.

4.3.3 Hardware – software

Sending an interrupt to a software task by a hardware device is done by writing to a memory mapped *interrupt register*. This register can collect interrupt requests from multiple hardware tasks and is connected to the MIPS interrupt controller. The interrupt service routine reads a bit vector from the interrupt register and determines in which message queues to write, thereby unblocking the tasks that are waiting for data or space.

4.3.4 Hardware – hardware

A hardware device can write directly into the signal register of another device to signal the availability of data. The address is present in a register inside the device shell.

4.4 C-HEAP Block

This C-HEAP Block is the heart of the C-HEAP Shell. The C-HEAP Generic Interface ports & device interface ports are shown in Figure 21. There are two more signal added to the C-HEAP Block named *sgnl_reg* and *chg_sgnl*. The *sgnl_reg* is the value of the global signalling register and the *chg_sgnl* indicates that the value of the *sgnl_reg* has been changed. These both signals are used to wake up a C-HEAP block (more precisely to wake up the C-HEAP Synchronisation Controller). The C-HEAP synchronisation interface signals are shown in Figure 21. Basically, the implementation of C-HEAP block can be divided in four different parts.

1. Administrative information of the channels (paragraph 4.4.1 C-HEAP Administrative Information)
2. The C-HEAP Synchronisation Controller (paragraph 4.4.2 C-HEAP Synchronisation Controller)
3. The C-HEAP generic bus interface initiator (paragraph 4.4.3 Generic Bus Interface)
4. The C-HEAP generic bus interface target (paragraph 4.4.3 Generic Bus Interface)

A C-HEAP Block contains administrative information of the channel, and takes care of the C-HEAP synchronising protocol. This channel and task information (e.g. semaphore counter, base address of buffer, size of buffer, number of buffer etc.) is kept in registers. The synchronisation transaction starts when the controller has a valid buffer pointer. The hardware version of C-HEAP checks itself if there is some

data available at the other side of the channel. But the software version of C-HEAP does not perform this check automatically. Because of this there are two kinds of C-HEAP Blocks necessary namely a version for hardware-hardware communication and a version of hardware-software communication (this is explained later in more detail).

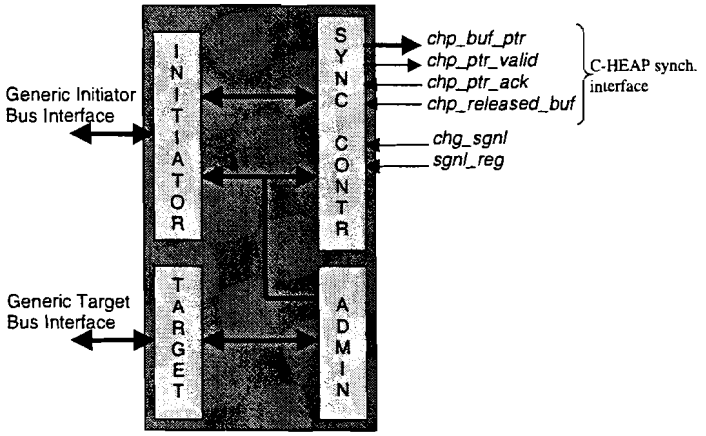

Figure 21 : C-HEAP Block.

Why is there a need for different C-HEAP Blocks?

In the original software implementation, the administrative information of a channel is stored in a so-called *channel-record*, centrally located in memory. The synchronisation protocol can be optimised by storing the channel information in memory mapped registers in the C-HEAP Task Shell. Two hardware devices that communicate with each other via a channel both have a copy of the channel information. In this way, the load on the chosen-bus-interconnect network can be minimised.

A task has one C-HEAP Block for every input- or output- channel. As a matter of fact any device can read/write data from/to the C-HEAP Block Administrative Information because the Administrative Information registers are in the address space. *Sgnl_reg* is the value of the global signalling register and the *chg_sgnl* indicates that the value of the *sgnl_reg* has been changed. These both signals are used to wake-up the C-HEAP Block, more precisely to wake-up the C-HEAP Synchronisation Controller.

The *chp_put_data/space* calls are implemented with only one signal called *chp_released_buf*. The device asserts this signal when it wants to notify the release of the buffer to the connected device. After receiving *chp_released_buf* signal the CSC controller takes care to notify this to the other device. Note that the synchronization interface is synchronous i.e. all signals are sampled/changed only at the clock edge.

Chp_released_buf is used by the task to notify C-HEAP Block that a token is released for the other communicating task. A token is released if the task read or wrote data to it. For more information see paragraph 4.4.2 C-HEAP Synchronisation Controller.

4.4.1 C-HEAP Administrative Information

In order to provide FIFO behaviour of a buffer, some administrative information has to be maintained. The administrative information contains some static values and some dynamic values. These C-HEAP Block registers are mapped in the address space. This is configured at configuration time. When the administrative information is located closely to the task the memory access latency minimises, which in turn improves memory efficiency. But because of the address space this information can be stored in every memory in that memory map. As can be read in paragraph 4.4.2 C-HEAP Synchronisation Controller, it is possible that there are some timing constrains. *Size_buf* depends on the use of memory space (larger *Size_buf*) and synchronisation overhead (smaller *Size_buf*), for every application a choice has to be made. Latency and throughput are both very important, latency for control signals and throughput for processing.

In Table 3 the administrative information of hardware register in the C-HEAP Task Shell is given.

Table 3 : Administrative chanel information of a hardware task.

Memory mapped registers	Function
Input	If (TRUE) channel is input else channel is output
Size_buf	Size of a token.
sgnl_reg_addr	Address from the signalling register (*sgnl_reg)* of the other communicating device. This address is for writing the *sgnl_value* to, this is for waking up the other communicating device.
sgnl_value	Unique signalling value fixed for every channel. Equal *sgnl_reg* value & *sgnl_value* indicates wake-up signal for this channel.
RSMPR_addr	Local semaphore address of the same channel but located in other device.
Nbuf	Number of tokens.
LSMPR_reg	Local semaphore value with a flag indicating rolling of semaphore. The flag is most significant bit of the register, rest of the bit is a counter.
Buf_ptr	Base pointer to the array of buffers.
Mem_LSMPR_addr	Address where the updated LSMPR has to be written when communicating with a software task. (This register is not implemented for communication between two hardware tasks, or two hardware tasks whom are communicating without a channel record, see Table 4.)

In Table 4 the administrative information of channel record in memory is given, channel records are used for a software implementation of C-HEAP.

Table 4 : Administrative chanel record of a software task.

Channel record fields	Function
Channel_ID	Unique value fixed for every channel.
Nbuf	Number of tokens.
Channel mode	Synchronisation polling or interrupt based, and whether the channel buffers are allocated directly or indirectly.
Producer task address	Pointer to the producer record.
Consumer task address	Pointer to the consumer record.
Size_buf	Size of a token.
Buf_ptr	Base pointer to the array of buffers.
Producer channel synchronisation counter	This value is used in combination with consumer channel synchronisation counter to determine the full/emptiness of the channel.
Consumer channel synchronisation counter	This value is used in combination with producer channel synchronisation counter to determine the full/emptiness of the channel.

When communication with software is needed there is a need for one channel record field, even if there are two software tasks. Every hardware tasks needs memory mapped registers for the ADMIN information about the channel. This is shown in Figure 22.

As can be seen in Figure 22, in case of hardware-hardware communication, the channel information is located in registers and possibly in a record in shared memory, it is recommended for having a copy of channel information in memory. This has some advantages:

- If, for e.g. debug reasons, we want to know what the status of the channel is at any given time, keeping a copy in memory allows us to access this information easily (it is not needed to read the data from both shells).
- In the future dynamic reconfiguration of the channels must be supported. A channel may be switched from between two hardware tasks to between a hardware task and a software task, so we need a software copy anyway.

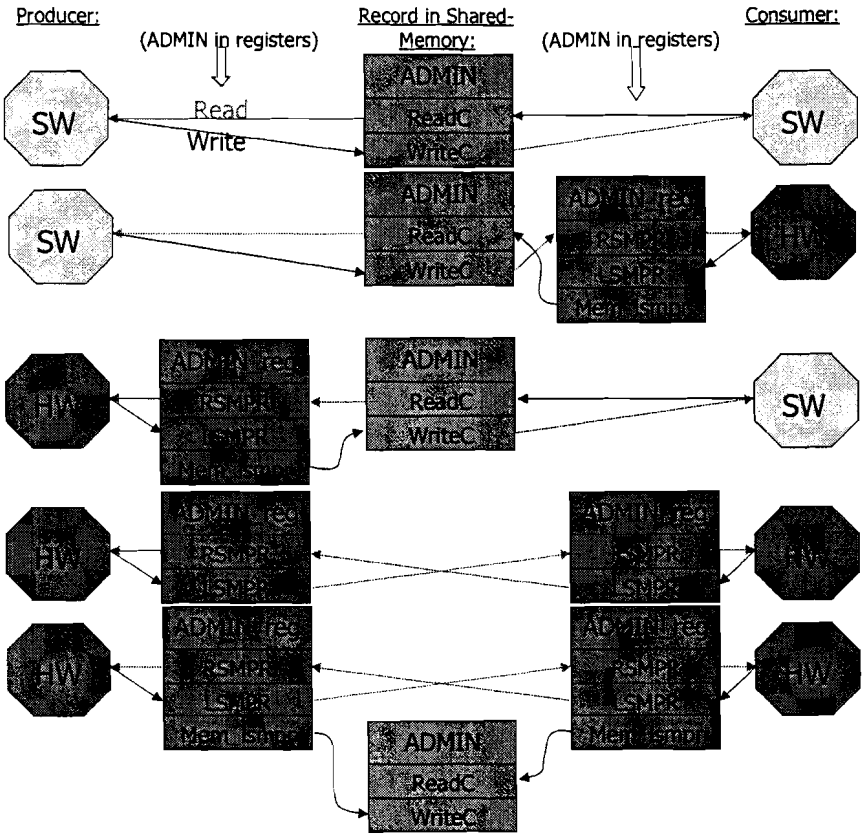

Figure 22 : Location of channel information.

When setting up a channel, first a structure is created in memory. Then, depending on the producing and consuming tasks, none (SW-SW), one (SW-HW), or two (HW-HW) copies of different format are written in the device shell registers. At a later stage, these copies may be changed by the software (e.g. when allocating the channel buffers). In order to do this, the software copy of the channel structure should have some additional information, such as the locations of the device shell registers in which to write the channel information. This channel record now looks as follows:

Table 5 : Administrative channel record for communication with hardware task(s).

Channel record fields	Function
Channel_ID	Unique value fixed for every channel.
Nbuf	Number of tokens.
Channel mode	Synchronisation polling or interrupt based, and whether the channels buffers are allocated directly or indirectly.
Producer task address	Pointer to the producer record.
Consumer task address	Pointer to the consumer record.
Size_buf	Size of a token.
Buf_ptr	Base pointer to the array of buffers.
Producer channel synchronisation counter	This value is used in combination with consumer channel synchronisation counter to determine the full/emptiness of the channel.
Consumer channel synchronisation counter	This value is used in combination with producer channel synchronisation counter to determine the full/emptiness of the channel.
Pchan	Pointer locations to the possible hardware copies of the producer. These field is NULL if there is no copy.
Cchan	Pointer locations to the possible hardware copy of the consumer. These field is NULL if there is no copy.

The software copy of the channel record should be updated, i.e. when the channel fullness changes. The software is developed so that this update is automatically done when two software tasks are communicating, but the hardware is made only for hardware to hardware communication without a memory record. Of course SW-HW communication is needed that is why a special version of the hardware is implemented. When the producer or the consumer is a hardware task, then, when the hardware task releases a buffer, the task shell must update the channel synchronisation value in memory after updating its local register.

For calculate the number of free tokens next code can be used:

```
flags_equal := (tmp_rsmpr_flag = lsmpr_flag);
if input_type = input
   then
        if flags_equal
          then  nr_buffer_available <= RSMPR_reg – LSMPR_reg;
          else  nr_buffer_available <= Nbuf – (LSMPR_reg – RSMPR_reg);
        end if;
end if;
```

```
if input_type = output
   then
        if flags_equal
         then nr_buffer_available <= Nbuf – (LSMPR_reg – RSMPR_reg);
         else nr_buffer_available <= RSMPR_reg – LSMPR_reg;
         end if;
end if;
```

The next registers have to be initialised at start up:

- *Sgnl_value;*
- *RSMPR_addr;*
- *Nbuf;*
- *Buf_ptr.*

Sgnl_addr, RSMPR_addr, and *Buf_ptr* are determined by the C-HEAP root task at start up (when the tasks and channels are created). *Input* and *Size_buf* are fixed values, yet simply adjustable in the VHDL source-code, both should be programmable eventually. *LSMPR_reg* is a variable that is initiated at zero.

Table 6 : Register order in C-HEAP Block.

Memory mapped registers	Register	Address [4..0]
Input	Fixed	-
Size_buf	Fixed	-
sgnl_reg_addr	Reg_0	00000
sgnl_value	Reg_1	00100
RSMPR_addr	Reg_2	01000
Nbuf	Reg_3	01100
Buf_ptr	Reg_4	10000
Mem_addr_LSMPR	Variable	10100
LSMPR_reg	Variable	11000

4.4.2 C-HEAP Synchronisation Controller

The synchronisation transactions to the device start automatically when C-HEAP Synchronisation Controller has a valid buffer pointer. This is signalled through *chp_ptr_valid* and *chp_buf_ptr*, and is accepted by *chp_buf_ack*. On a *chp_released_buf* call the C-HEAP Block signals the other task that something is changed.

Table 7 : Synchronisation interface signals.

Port	Source	Function
Chp_ptr_valid	C-HEAP Block	C-HEAP is providing a valid buffer pointer.
Chp_buf_ptr	C-HEAP Block	Buffer pointer.
Chp_ptr_ack	task	Process accepted buffer pointer.
Chp_released_buf	task	Process signals that the buffer is ready to be released. Then, C-HEAP Block takes care for actually writing wake-up signal value over the bus to the other task.
Sgnl_reg	*sgnl_reg*	Access to signal register data, used to wake up channels / tasks.
Chg_sgnl	*sgnl_reg*	High for one clock cycle when a write action is performed on *sgnl_reg*.

When the controller has a valid buffer pointer, it informs the task by putting the buffer pointer at *chp_buf_ptr* bus and asserting the *chp_ptr_valid* signal, otherwise it simply waits for a wake up signal from the other communicating device. If the task wants to read/write data, C-HEAP Block gives the task a pointer to the start address where data can be read/written. The task acknowledges the receipt of the pointer by asserting *chp_buf_ack* signal. If *chp_buf_ack* signal is not asserted by the task, *chp_ptr_valid* and *chp_buf_ptr* signals remain the same.
On accepting the acknowledged signal from the task the controller checks the administrative information of the channel and comes up with a new valid value of buffer pointer. The task asserts *chp_released_buf* signal for one clock cycle when it wants to notify the release of the buffer to the connected device. If the task asserts *chp_released_buf* the oldest not yet released token is released. This provides freedom to the task/device to continue execution without waiting to complete this action. After receiving *chp_released_buf* signal, the controller takes care to notice the other task. A task can use as many tokens as made available without releasing previously occupied tokens but they have to be released in FIFO order.

Note:

- All signals are sampled/changed at the clock edge;
- Releasing a token requires a system bus write operation.

It is possible that the input of a producing task has a very tight timing constraint (*deadline*). If a task deadline is critical to system functionality and must be met, it is classified as a *hard deadline*. With other words, the producer has to read and write its process input continuously else data could be lost. So, C-HEAP has to take care

for sufficient tokens for writing the data to. It is not allowed that the producer blocks because at that moment it can not send the processed data to the consumer even though it has to process the next data. Since these hard deadlines are critical to the functioning of the system, they must be met under worst-case conditions.
Sometimes loss of data may be tolerated as long as the error does not accumulate, this is classified as a *soft deadline*. To make it complete, there are also tasks with *no deadline*, for example maintenance and testing activities.

Notice that there are many definitions of hard- and soft deadlines. They are all based on the usefulness of late/missed data.

4.4.3 Generic Bus Interface

At Philips a "Sea of IP" is available, but most IP is only re-usable in its native context! (See Figure 23). A standardised generic interface makes it possible to develop IP that can be used with in different busses and IC architectures. In modern complex ICs the main bus is often system specific. Different ICs use different buses to meet different requirements. In the future, it is likely that new busses will emerge that are tuned to various applications or specific memory types. Therefore, a bus in a SoC IC should be considered a system specific communication channel. In order to avoid redesigning IP for different buses, the interface to the IP can be made more abstract. If a good generic interface is defined for bus-based IP, that IP can be easily adapted to different buses using a wrapper. The generic interface must isolate the IP from the specific details of the system bus. The bus interface must be a point-to-point interface without arbitration or multiplexing. Furthermore it must be a symmetric interface, this means that two IP blocks can be connected glue-less together with no bus in-between.

Notice that when bus infrastructures (bus control units, bridges, etc.) are standardised it is possible to re-use these infrastructures and also the developed bus wrappers.

The concept is that, when connecting blocks of IP to various busses, only the wrapper functionality will need to be different. The IP implementation will not change, and thus the block design needs to be done only once and can be easily deployed over a variety of busses and applications. Exchange of IP cores between platforms is needed for better re-use so for a shorter time-to-market and a lower risk, see Figure 25. This can be done through making IP cores with a DTL interface [5] and bus wrappers, see Figure 24.

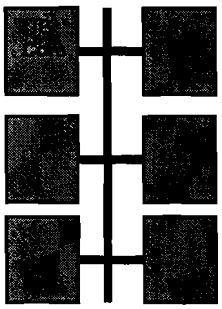

Figure 23 : IP core with bus interface.

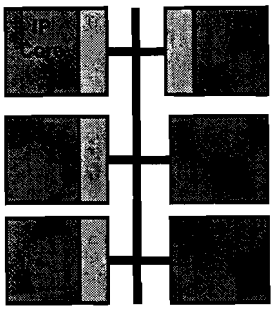

Figure 24 : IP cores with DTL interface and adapter to bus.

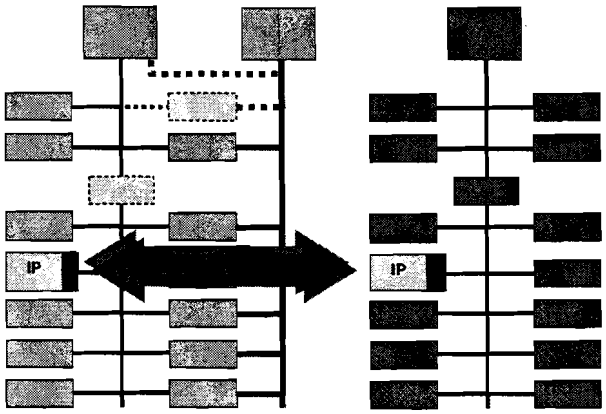

Figure 25 : IP re-use.

DTL helps to separate fixed IP core functionality from the variable system environment. The goal of this is to achieve bus independent IP cores. Once there is a bus independent core which communicates with DTL the only thing what is needed next is a bus wrapper. Once the whole department uses DTL there are also wrappers (DTL-to-x-bus) be developed which of course also can and will be re-used. For a matter of fact at Philips the "re-use shop" exists which can deliver wrappers and other IP.

One of the Re-Use Principles: separate communication types at the IP core boundary, this gives the system designer the choice to *merge* or *separate* communication flows at system level (shown in Figure 26)

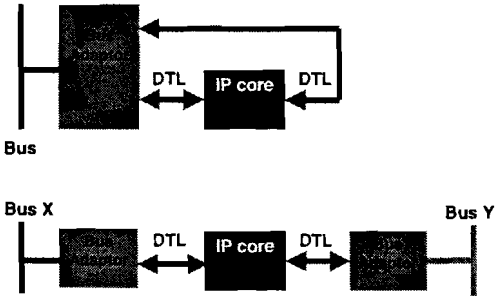

Figure 26 : Choice of merging or separating communication.

The DTL protocol is used for data communication between an initiator and a target. The initiator always initiates a transaction and the target always responds. The DTL protocol is a point to point protocol; a single initiator is connected to a single target at all times. The target receives commands and possibly write data from an initiator, and may drive read data back to the initiator.

The objectives of the DTL protocol include the following:
- Provide a simple, efficient, point-to-point synchronous interface for IP-to-bus or IP-to-IP communication;
- Allow for unplanned IP re-use by reducing concern for system issues in the IP development phase;
- Promote the development of bus-independent IP;
- Support low latency traffic as well as high bandwidth.

The DTL protocol includes the following features:
- Generic definition: IP, system, and technology independent;
- Support for different types of traffic (MMIO, streaming, etc.);
- Point-to-point interface using unidirectional signals;
- The DTL protocol is a synchronous protocol. All signals in the DTL protocol (except **rst_an**) are synchronous to the rising edge of **Clk**;
- Independent flow control for command, write data and read data;
- Scalable address, data, and block size;
- Write data tagging;
- Buffer management facilities;
- Optimisation towards specific communication requirements through defined DTL subsets.

An adapter is used to interface a DTL port to a bus. Adapters typically include data buffering in order to optimise bus transactions for performance. They may provide an interface between two different clock domains. Adapters typically have one bus interface and one or more DTL interfaces.
With DTL, IP cores are system independent defined and the cores 'talk' DTL in a fashion that is natural to the IP, and not to the system. System adaptation is done in an adaptation layer ("adapter").

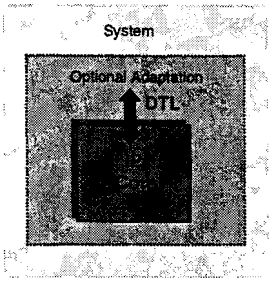

Figure 27 : How to Abstract IP Cores from the System?

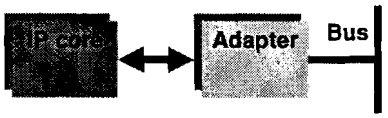

Figure 28 : Bus Adapter.

The DTL target must accept any command issued by an initiator even if the command cannot be handled by the target, otherwise dead-lock occurs. The target issues an error upon unsupported commands.

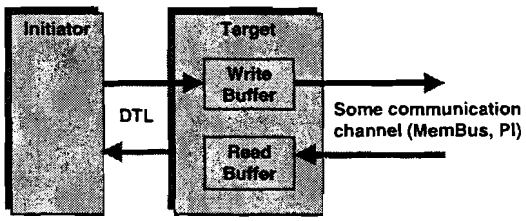

Figure 29 : Buffers.

Buffers are commonly used in systems to de-couple two communication channels, e.g. DTL from DVP Memory Bus or PI-Bus. Buffers also can be used for:

- Clock domain separation.
- Gathering of write data to create write data packets for a bus.
- Capture read data packets from the bus.
- Pre-fetch read data.

Recall, normally there are multiple C-HEAP Blocks in C-HEAP Task Shell and only one (system) bus because of this (de-)multiplexers are needed. These (de-)multiplexers are needed because of the multiple DTL-initiator ports in C-HEAP Blocks and in the Application which have to be connected to the Local-Bus (LB). When there are multiple DTL-initiator ports an arbiter is needed. Note that this arbiter can be omitted if there is only one DTL-initiator port.
A de-Multiplexer is needed because the LB signals have to be carried through to one of the multiple DTL-target ports in the C-HEAP Task Shell or Application. Note that an address decoder is needed for selecting the appropriate DTL-target.
A multiplexer is needed for multiplex all C-HEAP Task Shell and Application DTL-initiator ports to one DTL-port, an arbiter is needed for selecting a future initiator.

Due to simplicity reasons a minimal DTL protocol is implemented. The Task can read and write only one 32-bit register at once, as shown in Figure 30.

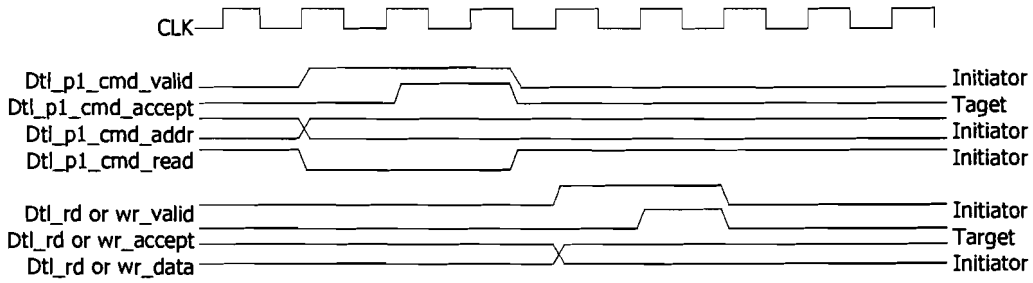

Figure 30 : DTL Initiator read and write.

For a solid explanation of DTL read the internal rapport about it: Philips Semiconductors DTL Protocol Specification. On Philips Intranet the newest version of the DTL protocol can be found at [6].

4.4.4 C-HEAP Signalling Register

The C-HEAP Signalling Register is implemented to provide all configured C-HEAP Blocks a unique signalling value for waking up the blocked C-HEAP Blocks.

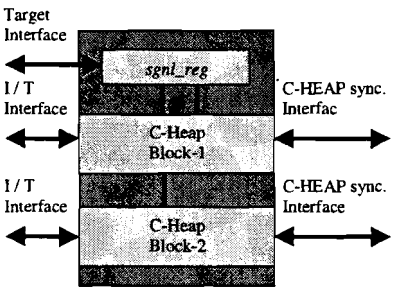

Figure 31 : C-HEAP Task Shell.

Although a C-HEAP Task Shell can be configured with multiple C-HEAP Blocks there is only one waking-up signal.

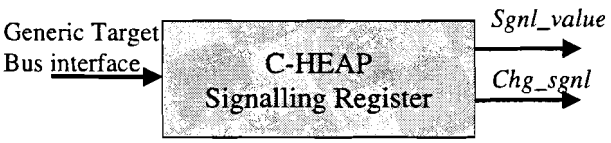

Figure 32 : C-HEAP Signalling Register.

To write the waking-up signal to the C-HEAP Signalling Register a generic bus target interface is needed. The communicating C-HEAP Block is writing *sgnl_value* (unique channel number) to the address (*sgnl_reg_addr*) of the C-HEAP Signalling Register. When *sgnl_value* is written to the C-HEAP Signalling Register this value is written to

the *sgnl_value* output which is connected to every C-HEAP Block. When the value is written to the output port *chg_sgnl* is high for only one clock cycle. When a C-HEAP Block detects the *chg_sgnl* it is reading the *sgnl_value* and compares this with the internally stored *sgnl_value*. When these signals match the C-HEAP Synchronisation Controller is woken-up.

Notice that the C-HEAP Synchronisation Controller only blocks when there are no buffers available to process. So when all buffers are processed the C-HEAP Synchronisation Controller blocks and waits for a wake-up signal.

4.4.5 Compounded C-HEAP Shell

Buses are present in systems because they are a cost-effective and versatile means of connecting several devices together. The cost-effectiveness and versatility of busses stems the fact that a bus has only one communication medium, which is shared by multiple devices. In other words, at most one communication can occur on a bus at a given time.
As can be seen in Figure 31 there are (2N+1) DTL Initiator- and Target- ports needed for every C-HEAP Shell, with N is number of C-HEAP Blocks, multiplied by two because there is a need for an Initiator- and Target port, add one for the target interface for the signalling register. Usually there are also two ports needed for the Task. For every DTL port is a DTL-2-"bus" wrapper needed, to minimise the amount of wrappers it is possible to combine DTL ports. This combining can be done with multiplexers, see next two figures.

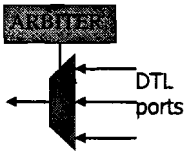

Figure 33 : Multiplexer with arbiter.

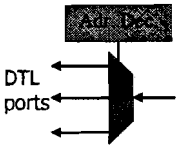

Figure 34 : De-Multiplexer with Address decoder.

In this project a *centralised arbitrator* is used, which is in charge of granting control to one of the initiators (One of the C-HEAP Blocks or the Task). The central arbiter is connected to each of the bus initiators through private two-way connections (request and grant signals). Because the bus request can be made independently and in parallel by the bus initiators, this connections is also known as *centralised independent request arbitration* or *centralised parallel arbitration*. Various arbitration policies can be implemented in the arbiter, making this a flexible scheme. This scheme is fast because there are direct connections between any bus initiator and arbiter.

The address decoder is needed for sending the incoming DTL signals to the appropriate target. The address decoder must be initialised when the system starts-up because it must "know" which address ranges it must send to which consumer.

When one multiplexer and de-multiplexer is combined with C-HEAP Shell a compounded C-HEAP Shell exists, the advantage of this is that there is only one DTL Initiator port and one DTL Target port, see next figure. Notice that the DTL target

Interface is needed for initialising the C-HEAP Blocks and sgnl_reg, furthermore it is possible to read the memory mapped registers.

Note: AR means ARbiter and AD means Address Decoder.

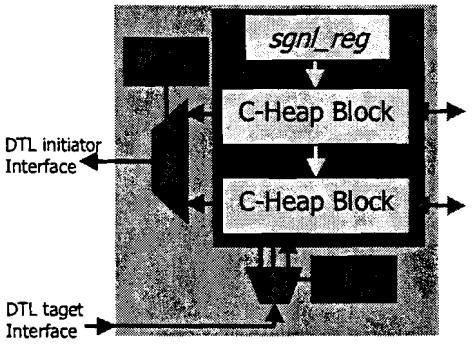

Figure 35 : Compounded C-HEAP Shell.

De-multiplexer-1 combines both C-HEAP Block Initiator ports and arbiter-1 is arbitrating between both initiator ports. At this moment a Round-Robin arbiter is implemented because it is an easy arbitration scheme and it is enough to prove the C-HEAP concept.
Multiplexer-2 and Address decoder-2 are selecting the appropriate C-HEAP Block Target port. This is for configuring the administrative register of both C-HEAP Blocks and writing the *sgnl_value* to wake-up one of the C-HEAP Blocks.

4.5 Problem and Solution

One problem is that when the C-HEAP Block wants to initiate a bus transaction. The PCI Initiator/Target asks for the PCI bus (PCI_REQ#), when it gets the PCI bus (PCI_GNT#) the address has to be present at once, but this is impossible. This is impossible due to the latency of the communication stack (shown in Figure 36).

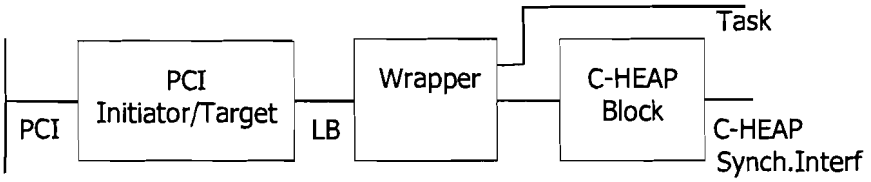

Figure 36 : Standard communication stack overview.

If the task wants to initiate a transaction it gives a valid signal and the appropriate address to the LB-2-DTL wrapper. The PCI I/T asks for the bus by asserting PCI_REQ. When the PCI I/T is granted to the PCI-bus this has to be translated into a Local Bus signal and this local bus signal into an accept for the task. But the wrapper writes the address to the PCI I/T, the PCI I/T has to convert the local bus signals into a PCI address. So it is impossible to have the address in the PCI Initiator/Target in time (in time means within 30us), this is shown in Figure 37.

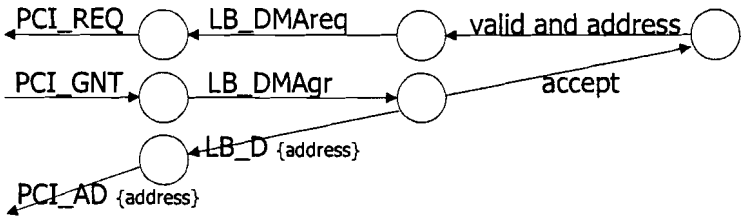

Figure 37 : PCI-Board problem.

The solution for this problem is shown in Figure 38. When the task wants to initiate a transaction, *valid* gets active. The LB-wrapper[5] has to know from which C-HEAP Block the address is (whose coming from the application). Because, at that moment the other communicating C-HEAP Block is known and it is possible to signal the PCI Initiator/Target this with *LB_CHEAPinfo*.

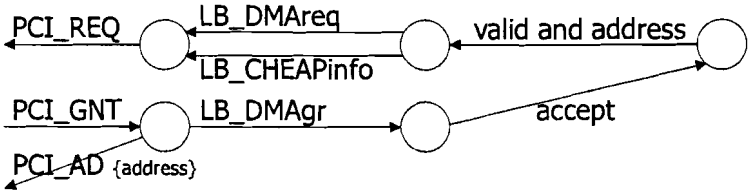

Figure 38 : PCI-Board solution.

But now the PCI Initiator/Target has to know which address is meant with *LB_CHEAPinfo*, a look-up table must be implemented in the wrapper and one in the PCI I/T. This is shown in Figure 39. The local bus protocol has 8 spare lines for communicating between PCI I/T and wrapper this means that both look-up tables can have 256 rows.

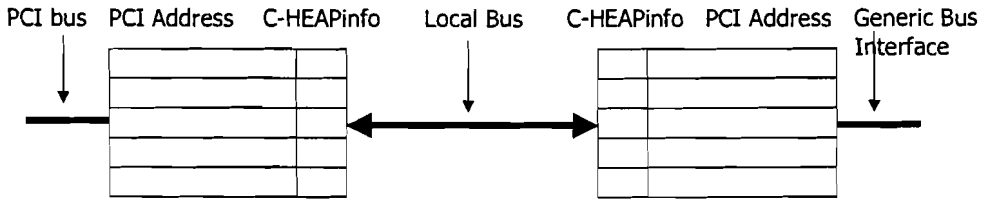

Figure 39 : Solution with LB-Spare lines.

The filling of this table has to be done by the driver but this is not described in this report.

[5] The basic function of a wrapper is to convert the generic interface protocol to the specific protocol of the system bus.

Chapter 5

5 Architectural template

5.1 Introduction

This chapter describes the developed architecture in detail. Furthermore it is described how to initialise and use the architecture in association with the PCI board. In section 5.2 the developed architectural template is explained and in section 5.3 how the template must be initialised.

5.2 Architecture.

The basis of this architectural template is the Compounded C-HEAP Shell. The Compounded C-HEAP Shell is of course connected with the Task. Furthermore this Task has one DTL Initiator port and one DTL Target port. There is only one Initiator port because in this way the Task does not have to choose between multiple Initiator ports. E.g. if the Task writes an address it does not have to know whether it has to write to local memory or the PCI bus, this address decode is done outside the Task.

As can be seen in Figure 40 there are two Target ports (one to de-multiplexer-2 and one to the Task). One target port is used for initialising and reading[6] the C-HEAP Administrative registers and the other for initialising and reading the registers in the Task.

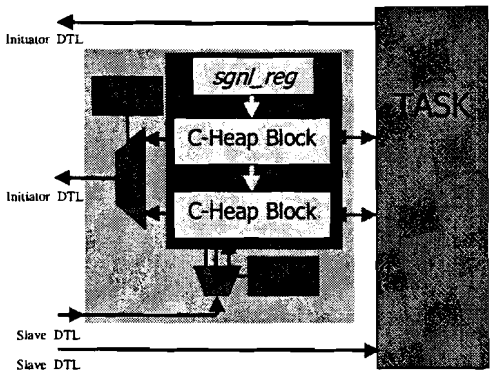

Figure 40 : Compounded C-HEAP Shell with a Task.

Initialisation of the administrative registers is handled by C-HEAP Root task at start-up. Reading of the registers is done by the CPU or another board. Now there is a

[6] Notice that the C-HEAP Administrative registers only can be read, writing the C-HEAP Administrative registers is only done at the initialisation-phase at start-up.

need for a de-multiplexer-3 for choosing between Compounded C-HEAP Shell and Task. The selection is done with an address decoder, see Figure 41.

Both multiplexers (4 and 5) and address decoders (4 and 5) are needed because it must be possible to start a transaction to local memory (memory on the FPGA board) or to the local bus.

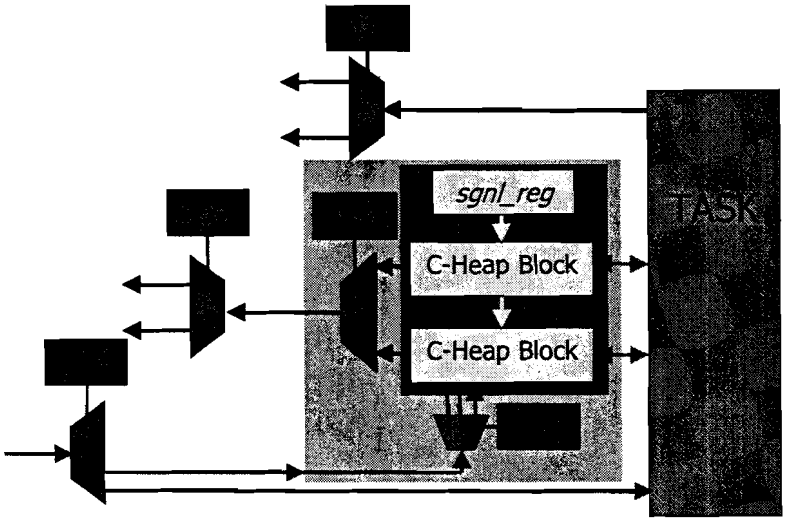

Figure 41 : Multiplexer and address decoders added.

Both de-multiplexers (6 and 7) and arbiters (6 and 7) are needed. De-multiplexer-7 because the memory interface only has one communication port and the local bus, the application, and the Compounded C-HEAP must be connected to it. De-multiplexer-6 is needed because the local bus only has one master communication port, see Figure 43.

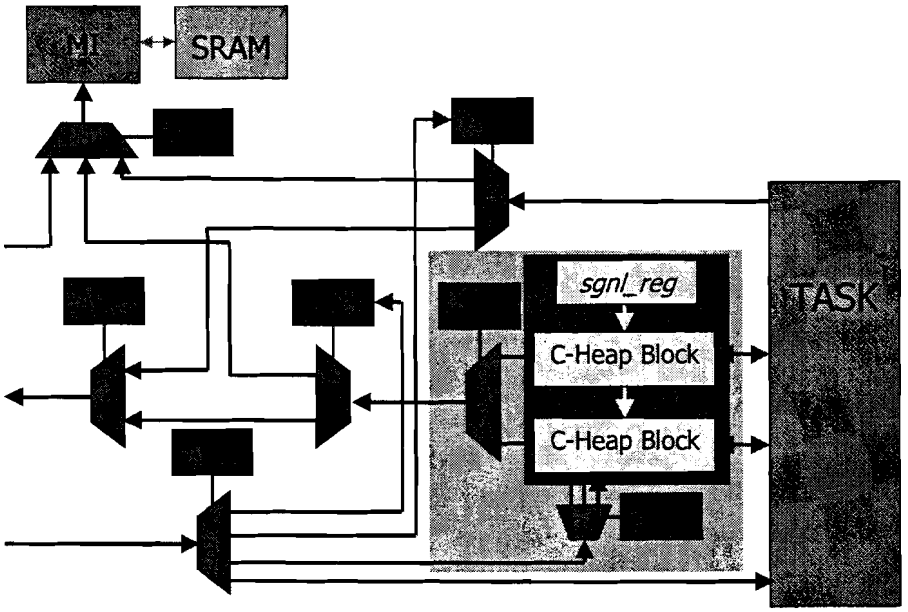

Figure 42 : De-multiplexers and arbiters added.

Notice that both DTL-target input ports cannot be multiplexed because both the SRAM and the registers have there own base-address.

What's left are three (!) LB-2-DTL bus-wrappers.

1. LB-Initiator to memory;
2. LB-Initiator from compounded C-HEAP and Task;
3. LB-target to compounded C-HEAP and Task.

Which bus wrapper must be used depends of the used board, in this case the LocalBus protocol is used, but every protocol can be used. In the future the LocalBus protocol must be replaced by a re-usable bus architecture (the system bus must support a superset of busses it can serve as siblings). This re-usable bus architecture is necessary because at that point it is possible that the developed PCI boards can be used in differed application domains and it is more future proof.

The complete architecture can be found in Figure 43.

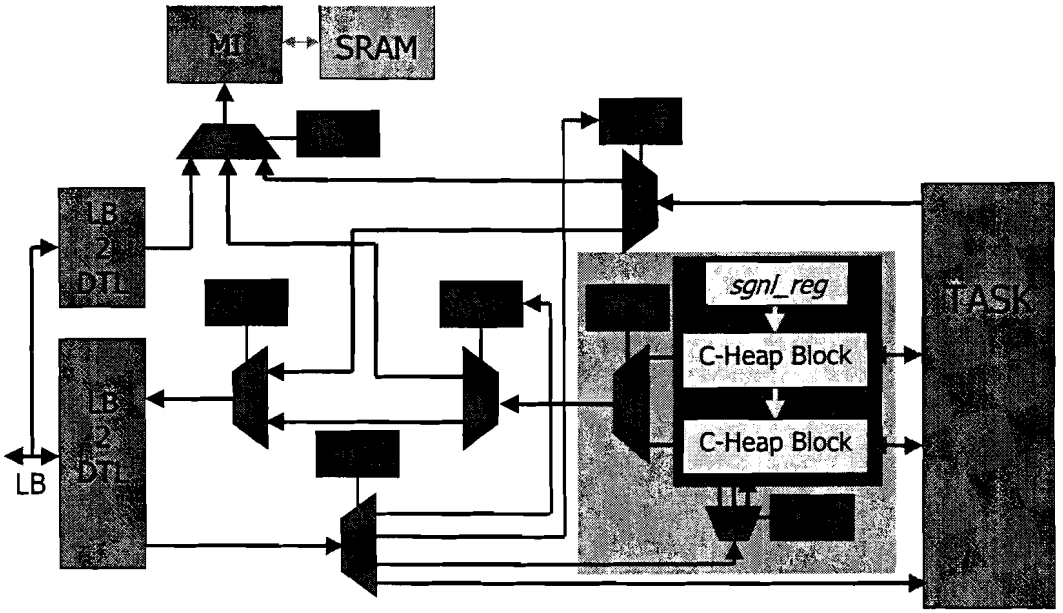

Figure 43 : Architecture for 2 C-HEAP Blocks.

The complete implemented architecture can be found in Figure 44. The main difference between Figure 43 and Figure 44 is the Master-react Block. In first instance this block was implemented for simulation purpose only because with this block it was possible to check the functionality of C-HEAP as initiator to the PCI bus. Later this part was also synthesised for debugging reasons because when C-HEAP wants to read or write to memory and something went wrong in the standard interface stack you got an error code from the CHEAP_react block. This is of course only working when the data is routed to the CHEAP_react block.

All address decoders that are used have to be initialised in the init-phase. This can be done in the same way as all C-HEAP Administrative registers are initialised.

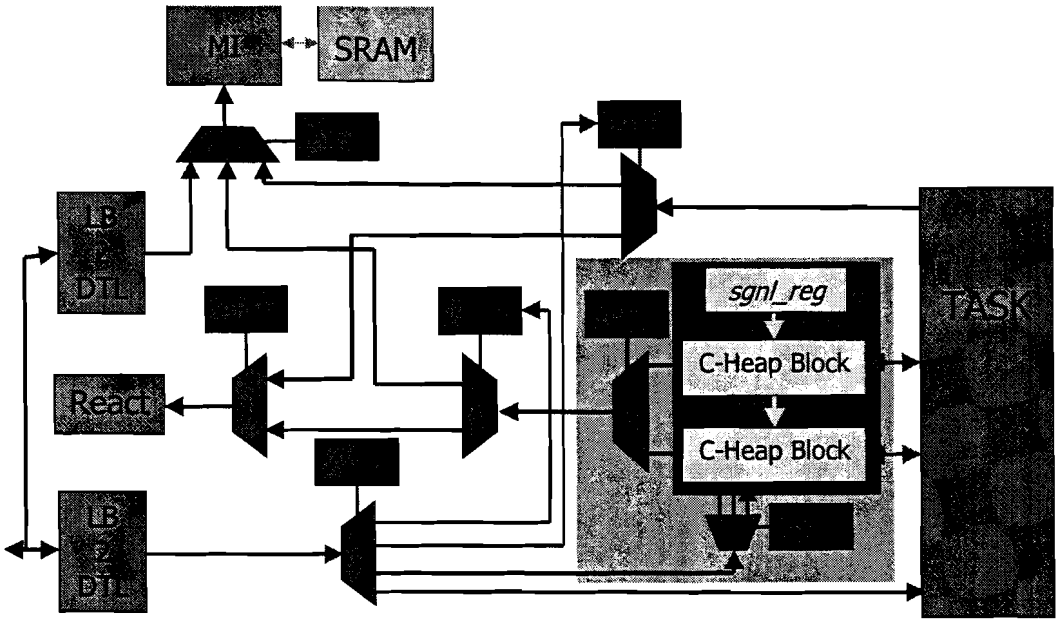

Figure 44 : Implemented architecture for 2 C-HEAP Blocks.

The memory bus has a very limited application field: the connection of a range of bus initiators to a single slave (memory). The Memory Interface (MI) depends (protocol, addressing scheme, clocking, or frequency) of the type of memory (SRAM, SDRAM, or DRAM), so for every type of memory a new interface must be used. There are standard configurable memory interfaceses available but the problem is the bus interface to the memory interface, the answer to this problem is the standard DTL protocol.

5.2.1 Device overview

The used FPGAs on the PCI-board are the EPF10K50EQC240 from Altera. (note: EPF10K is the type, 50 means 50k gates, EQC is the kind of package, and 240 are the number of pins.) See Table 8 for FPGA utilisation, and [8] for more information about Altera devices. Leonardo Spectrum also checks if the design fits into the selected FPGA, if it is not possible to put the design into the FPGA Leonardo gives an alternative. In this case it recommends the EPF10K200EGC599 device. In this case it is not possible to swap the EPF10K50EQC240 for an EPF10K200EGC599, this is because the firsts one has 240 pins and the second one needs 599 pins.

Table 8 : FPGA utilisation generated by Leonardo Spectrum.

```
*********************************************
Device Utilisation for EPF10K50EQC240
*********************************************
Resource        Used    Avail    Utilisation
-----------------------------------------------
IOs             115     189      60.85%
LCs             7542    2880     261.88%
DFFs            4558    3184     143.15%
Memory Bits     0       40960    0.00%
CARRYs          1106    2880     38.40%
CASCADEs        1105    2880     38.37%

-----------------------------------------------
This design does not fit in the device specified!
Trying an alternate device ...
*********************************************
Device Utilization for EPF10K200EGC599
*********************************************
Resource        Used    Avail    Utilization
-----------------------------------------------
IOs             115     470      24.47%
LCs             7542    9984     75.54%
DFFs            4558    10160    44.86%
Memory Bits     0       98304    0.00%
CARRYs          1106    9984     11.08%
CASCADEs        1105    9984     11.07%
-----------------------------------------------
```

Now the question is rising which Altera Device can be implemented instead of the EPF10K50EQC240 without redesigning the PCI board. This can be read in [7], but the devices which can replace the EPF10K50EQC240 are :

- EPF10K20 EQC240 (Smaller than EPF10K50EQC240);
- EPF10K30 EQC240 "
- EPF10K30A EQC240 "
- EPF10K40 EQC240 "
- EPF10K50 EQC240 (Equal to EPF10K50EQC240);
- EPF10K50V EQC240 "
- EPF10K70 EQC240 (Bigger than EPF10K50EQC240);
- EPF10K100A EQC240 "

So, it is not very helpful to replace the EPF10K50EQC240 for the bigger version because Leonardo calculated that device EPF10K200EGC599 (twice as big as EPF10K100A) is needed.
At this moment the question is how big an FPGA must be to use it now and in the near future. Because Philips Research almost always uses Altera devices I searched what is possible now a days. At this moment the APEXII EP2A90 is the biggest device Altera has, it has 4.000.000 typical gates (7.000.000 maximum system gates) and 1,5Mbits of RAM. And now we can question is this big enough?

I think it is (of course for the near future), why do I think this?

At the ESAS group a slave ART processor is implemented, when this processor is synthesised it needs approximately 300k gates, they think the Initiator ART processor also needs approximately 300k gates, with some application specific functional blocks (approximately 200-300k gates) a device of 800-900k gates is needed. This means that a minimum of an APEXII EP2A25 is needed, it has a typical gate count of 900k gates. In the near future or when the approximations are not good this device is completely filled, no more extra hardware can be implemented. But when the APEX II in a 724-pin BGA 33 x 33 package is used it is possible to swap the devices for bigger ones without redesigning the PCI board. In this range an APEX II EP2A25, EP2A40, EP2A70, and EP2A90 (biggest Altera device) can be used (from 900k till 4.000k typical gates).

5.2.2 Components overview

It is interesting to know which the bigger resource consumers are, in Table 9 a summary is given.

Table 9 : Comparation of needed Logic Cells and D-FlipFlops.

Master_react_CHEAP	declarations	1 x	5	5 F1_LUT
		5		5 CASCADEs
		1		1 VCC
		1		1 GND
		24		24 LCs
		18		18 DFFs
Mux_1_to_2_rechts	multiplexers	2 x	7	14 F1_LUT
		1		2 VCC
		210		420 DFFs
		227		454 LCs
		15		30 CASCADEs
Mux_1_to_3_int_compoun_cheap	multiplexers	1 x	38	38 F1_LUT
		74		74 CASCADEs
		1		1 VCC
		335		335 LCs
		147		147 DFFs
Mux_1_to_4_with_2_ad_ports	multiplexers	1 x	10	10 F1_LUT
		15		15 CASCADEs
		1		1 VCC
		250		250 LCs
		228		228 DFFs
Mux_2_to_1	multiplexers	3 x	7	21 F1_LUT
		7		21 CASCADEs
		1		3 VCC
		171		513 LCs
		162		486 DFFs
Mux_2_to_1_with_busy_to_mem_12	multiplexers	1 x	3	3 F1_LUT
		4		4 CASCADEs
		1		1 VCC
		101		101 LCs
		94		94 DFFs
Mux_2_to_1_with_busy_to_mem_17	multiplexers	1 x	3	3 F1_LUT
		4		4 CASCADEs
		1		1 VCC
		106		106 LCs
		99		99 DFFs
Mux_3_to_1_20bits	multiplexers	1 x	10	10 F1_LUT
		10		10 CASCADEs
		1		1 VCC
		274		274 LCs

```
                                194   194 DFFs
        TRI       flex10e   94 x    1    94 TRIs
        VCC       flex10e   1 x     1     1 VCC
Write_fifo_to_dtl_44_12    declarations    1 x    3    3 VCC
                                54    54 LCs
                                54    54 DFFs
Write_fifo_to_dtl_49_17    declarations    1 x    3    3 VCC
                                59    59 LCs
                                59    59 DFFs
Write_fifo_to_dtl_with_chp_released_63_31    declarations    2 x    3    6 VCC
                                74   148 LCs
                                74   148 DFFs
cheap_hw_to_mem_polling_based      cheap_cpu    2 x   274   548 CASCADEs
                                 3     6 VCC
                               573  1146 DFFs
                               963  1926 LCs
                                15    30 CARRYs
                                14    28 F1_LUT
dtl_to_fifo    renoir_shared    1 x     1     1 F1_LUT
                                 2     2 CASCADEs
                                 1     1 VCC
                                 1     1 GND
                               122   122 LCs
                                97    97 DFFs
 dtl_to_ram_be_notri    renoir_shared    1 x     1     1 F1_LUT
                                 2     2 CASCADEs
                                 1     1 GND
                                 3     3 VCC
                               121   121 LCs
                                98    98 DFFs
 lb_reg_data_access_notri    renoir_shared    2 x    38    76 LCs
                                 2     4 CASCADEs
 lpm_counter_10_299_0       OPERATORS    1 x    10    10 CARRYs
                                10    10 LCs
 lpm_counter_15_299_0       OPERATORS    1 x    15    15 CARRYs
                                15    15 LCs
 lpm_fifo_mod_45_1_OFF_UNUSED_LPM_FIFO    declarations    1 x    29    29 F1_LUT
                                51    51 CASCADEs
                                 1     1 VCC
                               156   156 DFFs
                                93    93 CARRYs
                               308   308 LCs
 lpm_fifo_mod_50_1_OFF_UNUSED_LPM_FIFO    declarations    2 x    29    58 F1_LUT
                                51   102 CASCADEs
                                 1     2 VCC
                               166   332 DFFs
                                93   186 CARRYs
                               321   642 LCs
 lpm_fifo_mod_64_1_OFF_UNUSED_LPM_FIFO    declarations    2 x    29    58 F1_LUT
                                51   102 CASCADEs
                                 1     2 VCC
                               194   388 DFFs
                                93   186 CARRYs
                               349   698 LCs
 producer     application    2 x     1     2 F1_LUT
                                 1     2 CASCADEs
                                 1     2 VCC
                               104   208 DFFs
                               136   272 LCs
                                32    64 CARRYs
 sgnl_reg       sgnl_reg    1 x     1     1 VCC
                                39    39 LCs
                                39    39 DFFs
```

As can be seen in general the (de-)multiplexers and C-HEAP are the biggest components.

5.3 Initialisation.

The latest PCI specifications can be found at the internet page of the Peripheral Component Interconnect-Special Interest Group (www.pcisig.com), but now some important items are explained.

At system power-up, device independent software must be able to determine which devices are present, build a consistent address space, and determine if a device has an expansion ROM. Notice that the total size of the memory in the complete system has to be known for building a consistent address space. Power-up software needs to build a consistent address map before booting the machine to an operating system. This means that it has to determine how much memory is in the system, and how much address space the I/O controllers in the system require. Power-up software can determine how much address space the device requires by writing a value of all 1's to the register and then reading the value back. The device will return 0's in all don't-care address bits, effectively specifying the address space required. After determining this information, power-up software can map the I/O controllers into reasonable locations and proceed with system boot. The PCI design implies that all address spaces used are a power of two in size and are naturally aligned. Devices are free to consume more address space than required, but decoding down to a 4KB space for memory is not suggested. For instance, a device that has 64 bytes of registers to be mapped into memory space may consume up to 4KB of address space in order to minimise the number of bits in the address decoder. Devices that do consume more address space than they use are not required to respond to the unused portion of that address space.

For example (shown in Figure 45), if the system needs five address spaces (for example <4KB) they are mapped onto the global address space, even if not the complete address space is needed. When a PCI Initiator/Target, which is implemented on each PCI board, detects a valid address for a specified range (e.g. between 10110_000h and 10110_FFFh (address space-3)) the appropriate address space is selected and the address is converted into a local address space address. For example, if address 10110_013h is detected the PCI Initiator/Target converts this address into 013h, as can be seen in this instance the above address bits are used for decoding the address.

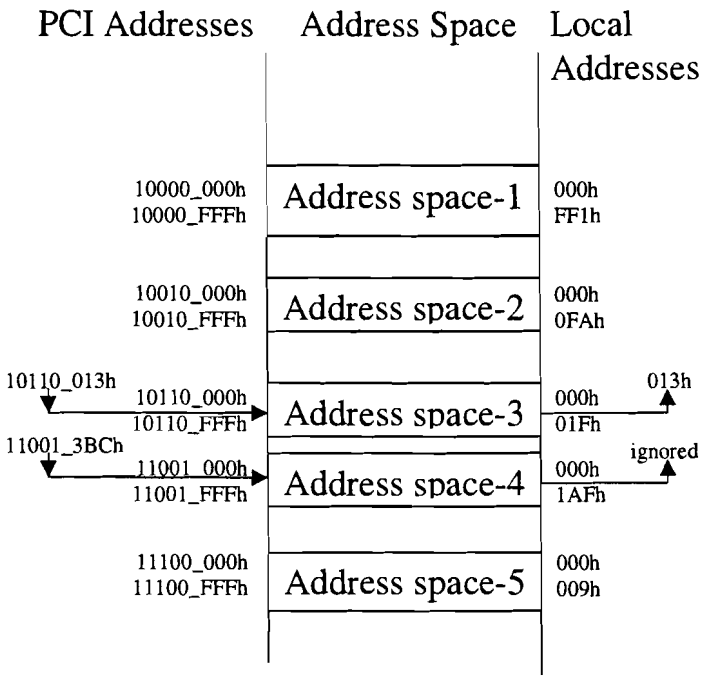

Figure 45 : PCI to Local Address.

There is chosen to divide one 8KB address space over the Compounded C-HEAP, Task, and address decoders, see Table 10. In other words there is one address space for the PCI board (excluding memory). In this way there is a lot of space to add more registers for future implementations. The address decoder of multiplexer-number-3 is fixed.

Table 10 : Address space of address decoder number-3.

Part		Amount	Address bits	
			12,11	10..0
Address decoders	Number 4(*)	2KB	00	0..0 address. Rest of addresses are spare.
	Number 5(*)	2KB	01	0..0 address. Rest of addresses are spare.
Compounded C-HEAP		2KB	10	Table 11
Task		2KB	11	Make an application specific address space.

(*) see Figure 41.

Address decoder-2 is fixed, see Table 11, this means there is no need for some initialisation, everything is known (fixed) in the address decoder.

Table 11 : Address space of address decoder number-2.

Part	Address bits		
	10..7	6,5	4..0
Signal register	*Spare*	00	00000
C-HEAP Block-1	*Spare*	01	See Table 6, at page 33.
C-HEAP Block-2	*Spare*	10	
Spare	*Spare*	11	*Spare*

Address decoder-4 and –5 have to be initialised because it has to distinguish between communicating with local bus port or local memory. The first thing what have to be done is to write the PCI memory address to both address decoders.

Table 12 : Address space of address decoder number-4 and –5.

Part	Address bits		Destination
	31..18	17..0	
Address decoder-4	PCI memory address	Don't care	Memory
	Not PCI memory address	Don't care	Local Bus
Address decoder-5	PCI memory address	Don't care	Memory
	Not PCI memory address	Don't care	Local Bus

5.4 Developed applications

The architecture as shown in Figure 43 is completely implemented and simulated. When I synthesised the design it was not possible to fit it in a FPGA. Because of this I had to remove much of my work for trying to fit it into one FPGA. In a later stage (when bigger FPGAs are available) it is possible to add these pieces back into the design. The result can be seen in the next two figures. I split up the producer (output-channel) and the consumer side (input-channel), so now there is one producer application and one consumer application as described below, this is done so save some FPGA space.

5.4.1 Producer side

The simplest producer produces a row of numbers starting from 0 to 0xFFFFFFFF (32 bits counter). The C-HEAP Block used is a hardware to software synchronisation protocol.

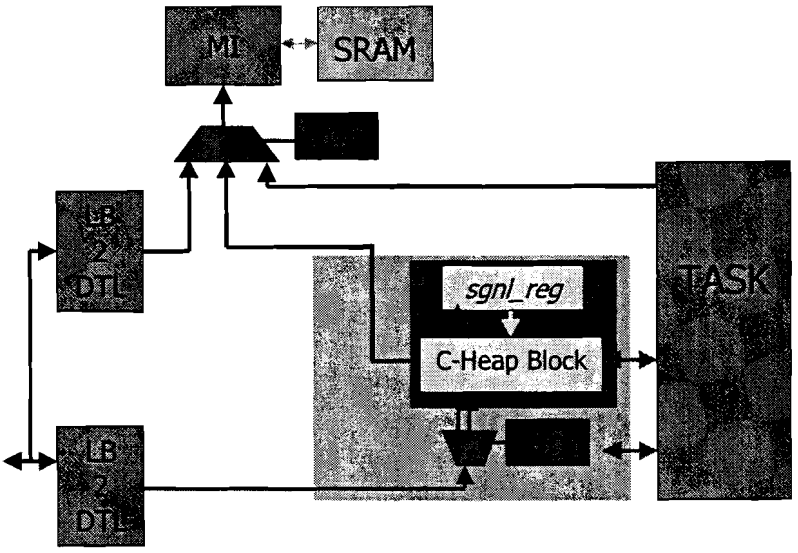

Figure 46 : Producer.

The C-HEAP channel is mapped onto the memory on board, this is because it is impossible for the board to become a master. In this architecture the board is a pseudo-master and the processor is a pseudo-target. This means that the Task initiates a transaction to the SRAM but it will not notify the processor, the processor also initiates a transaction to read the channel (this is called polling).
The initialising is about the same way, but at this moment there is no Address-Decoders with have to be initialised. Furthermore there is only one C-HEAP Block, the decodation of the Address-Decoder from the Compounded C-HEAP can be seen in the next table:

Table 13 : New address decoder-table of the producer board.

Part	Address bits		
	10..7	**6,5**	**4..0**
Signal register	*Spare*	00	00000
C-HEAP Block-1	*Spare*	01	See Table 6, at page 33.
Spare	*Spare*	10	*Spare*
Spare	*Spare*	11	*Spare*

This architecture is also designed and simulated, but when I synthesised it is still was still too big to fit in the used FPGA. At this moment I had a very big problem because it is impossible to remove more functional blocks because it is impossible to test C-HEAP.

5.4.2 Consumer side

Also in the consumer side the FIFO channel is mapped onto the SRAM on the PCI-board. Now the producer (processor) is writing the data into the channel and the Task (consumer) is reading it from the channel. To check if everything works the consumer writes the data in a register, which has to be read by the producer. The C-HEAP Block used is a hardware to software synchronisation protocol.

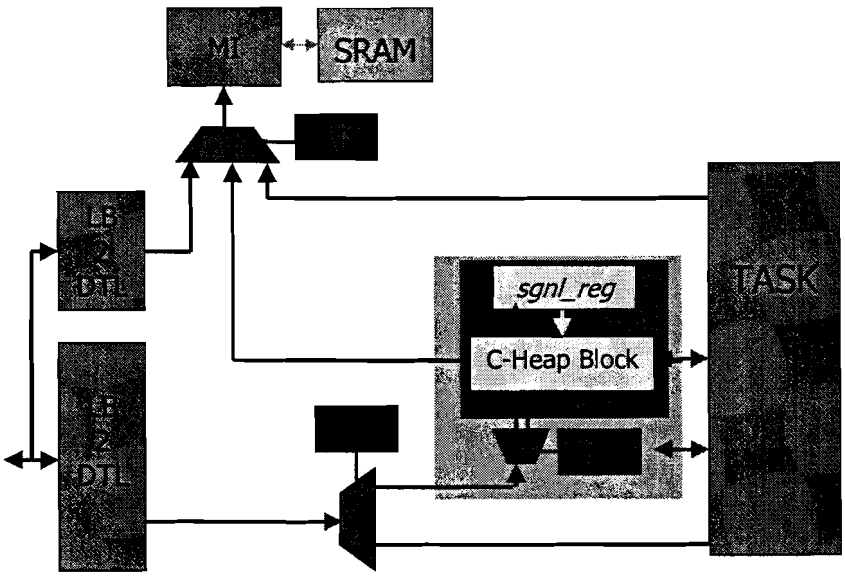

Figure 47 : Consumer.

The demultiplexer has to change from a 1-to-3 demultiplexer into a 1-to-2 demultiplexer, also the address decoder changes. The new address decoder can be seen in Table 14 (compare it with Table 10).

Table 14 : Address space of address decoder number-3.

Part	Amount	Address bits	
		12,11	10..0
Compounded C-HEAP	2KB	10	Table 11
Task	2KB	11	Make an application specific address space.

So the Consumer is the producer plus a multiplexer, but it is already impossible to fit the producer in the used FPGA so this design is not simulated and I did not try to map it onto the used FPGA.

Chapter 6

6 Conclusions and recommendations

6.1 Introduction

This thesis describes a strategy to build a rapid system prototyping system for a heterogeneous multiprocessor architecture with the use of C-HEAP. In section 6.2 the conclusions are given, and in section 6.3 recommendations are given for future developments.

6.2 Conclusions

For smaller designs, FPGAs prototypes are adequate. FPGAs has less gate count capacity and speed than ASICs. FPGAs are good for smaller applications, but not suitable for the entire SoC because most applications needs a kind of programmable processor. Although it is possible to use Altera FPGAs with a standard processor (ARM, MIPS, or NIOS) implemented on it.

Another problem to be solved is that of implementing a circuit into a set of FPGAs (called partitioning) because the FPGAs and memories must be joined together by an interconnect, a study to this kind of interconnect must be done before a new board is going to be developed.
The bus used now (local bus protocol) is not a good protocol (due to the address problem when the application can be a initiator, and the one way communication between the PCI I/T with the other FPGAs). It is better to have some other application specific communication protocol, e.g. PI, AHB, VPB, or PCI. But the best solution is to develop a system bus which supports a superset of busses it can serve as siblings. This means that it must be able to support all types of transactions that the siblings busses would require (not necessarily all transactions), and it should be able to work off-chip as well as on-chip. Furthermore if a "superset"-protocol is used the prototyping boards can be used for other applications domains aswel. All developed boards provides a hardware environment that can be used as a prototyping board, allowing to test circuit designs, or as a hardware support for test an optimise the application.

One advantage about using a PC system is about the PCI bus, this is a proven systembus on a proven PC system, and it is easy to buy and add PCI-Boards to the PCI-bus. The PCI bus is the real architecture. Any board can be added via expansion boards. The PCI platform, can reduce risk and development time.
Furthermore at Philips lots of proven IP for PCI is available (e.g. memory interfaces, power management, I2C, and IEEE1394).

Why is it good to do prototyping on a PC with a PCI system-bus?

- Most supported system world-wide is the PC,
- PCI bus is a widely used standard,
- Already boards and components for PC on the market,
- PCI software drivers are available for PC OS's (Windows, Linux),
- The PCI bus is just one level in the standard communication stack.

When the basic bugs have been eliminated, an extensive application run will be required to find run-time bugs. Simulation may not be able to run an application for very long time periods so a prototyping environment can be used. Although emulating is still slower than the actual silicon, prototyping is very useful because it can find bugs quick and these bugs can be fixed very fast. Only new IP needs validation, so make IP reusable (saves time) with a standard interface stack. Every time the reusable IP is used there is a chance of finding some minor bugs, because of this the hardware evolves in time to almost bug free hardware.

There is an Application Specific Domain Architecture "box" which evolves with time, driven by a domain of products. It is variable because the architecting group can choose how many and what kind of PCI boards are needed (can be re-used), and how many FPGAs are possibly necessary. Normally it is not necessary to make project dependent PCI boards, only if a new application domain is added. In the "box" are one Pentium computer, the necessary Application Specific Domain PCI-boards and FPGAs, all documents about the boards and FPGAs, and one document about the complete Application Specific Domain system-level-prototyping design phase.

Furthermore, the focus of a prototyping board must be on re-use and has more than one product as a target. Rapid prototyping board is not a solution for analog IP.

It is not very helpful to replace the EPF10K50EQC240 FPGAs for a bigger version because Leonardo calculated that device EPF10K200EGC599 (twice as big as EPF10K100AEQC240 which can replace the EPF10K50EQC240) is needed.
So, there is needed:

- a complete redesign of the PCI-Board;
- or a study to different kind of PCI Boards which are commercially available on the market.

As last, before prototyping a SoC a study must be done of which external data must be handled (IEEE1394, LAN, MPEG, etc), for these kind of protocols it is best to develop a separate PCI board. This for the reusability of all developed PCI-boards. Furthermore a study must be done to the off-the-shelf-ICs which are needed for the project, these external ICs must communicate with the RSP-system, so they must be placed onto the application specific PCI board or on an external PCI board.

The first point of the objectives (chapter 1.2) is met, a platform for handling the complexity of system-level testing and debugging is described in detail in chapter 5. The second point (the method should be fully integrated into the hw/sw co-design process of the system groups) is not completely met but the interest in rapid prototyping is growing. It would takes months/years to convince the different design groups that this method is necessary for cutting down the cost and development time. Cutting down the development time is necessary because of the personnel cost

but more important is that it is impossible to find enough highly educated personnel on the market.
The third point (prove the developed architecture) was not possible because the complete and minimized architecture were to big to fit into the FPGAs (chapter 5.4).

6.3 Recommendations

In this thesis some future research topics and implementation recommendations are given, in this section a summary is given:

Possible enhancements for this board:
- C-HEAP and PCI can work with interrupts, develop the complete interrupt mechanism;

- PCI board as both initiator and target, with interrupt support.

- Put some bigger FPGAs on the PCI-board, this is possible without any re-design. But it is not very useful due to the maximum size of the FPGA which can be used. See Altera documentation about this [8].

For new board development:
- It is not optimal to map a ARM or MIPS processor on an FPGA, so if a new board is going to be developed it is good to do a study at the ALTERA Excalibur FPGAs. The ALTERA Excalibur combines logic, memory, and a processor core, Altera's Excalibur embedded processor solutions allow engineers to integrate an entire system on a single programmable logic device (PLD). It is possible to have a ARM, MIPS, or NIOS embedded processor standard at the FPGA, see Altera webside [8].

- At this moment the FPGAs are working on 33 MHz (PCI speed) maybe it is interesting to let the FPGA work at higher speeds.

- Since embedded systems have become increasingly digital, components that interact with the analogue environment, such as analogue-to-digital and digital-to-analogue converters (ADCs and DACs), can be necessary. At the already developed FPGA-board these ADCs or DACs are missing. Although not every SoC needs these components it can be useful to have some PCI-Boards with only ADCs and/or DACs on it.

- To complete the complete design and verification trajectory do a study of Incide [3], which stands for Integrated circuit debugging environment. The purpose of Incide is to allow quick debugging of first silicon. It enables clock control and allows access to on-chip clock signals, flip-flops and RAMs. Incide is different from traditional software debuggers because it can be used to access *all* on-chip registers whereas software debuggers are meant to debug the software that executes on chips, and only provide a view of the registers and memory that are visible to the programmer. When a new PCI boards is going to be developed it is possible to add Incide. I think that the advantage of a rapid silicon prototyping platform with Incide is that it is not needed to go to first silicon before using

silicon debugging, FPGAs are almost real-silicon. I think it is possible to use Incide at the RSP platform for finding more and easier design errors, although Incide must also be used with the real first silicon for finding more design errors but mainly for manufacturing errors (but this is not our problem). At this moment a separation between design errors and manufacturing errors is made.

- At this moment the Task works with single read/write DTL Memory Mapped input/output this has to become a streaming or block data flow protocol.

- It is needed that FPGAs can communicate with each other, so a design must be developed with a re-usable bus architecture (the system bus must support a superset of busses it can serve as siblings).

- *Sgnl_addr, RSMPR_addr,* and *Buf_ptr* are determined by the C-HEAP root task at start up (when the tasks and channels are created). *Input* and *Size_buf* are fixed values, yet simply adjustable in the VHDL source-code, both should be programmable eventually. Because at that moment it is possible to (re-)configure the configuration during operation.

Bibliography

[1] J. Blijlevens, J. Kahlman, N. Lambert and I. Paulussen, *The WWICE homearchive, on the design of a Linux based hard disk recorder embedded in a IEEE1394 network*, Nat.Lab. Technical Note 2000/503.

[2] G. Kahn, *The Semantics of a Simple Language for Parallel Programming*, in Proceedings of the IFIP Congress 74, 1974, North-Holland Publishing Co.

[3] P. van de Haar, G.J. van Rootselaar, M.E.F. Roelofs and T.A. van Weelie, Incide User Manual by Philips Electronic Design and Tools/Verification, Nat.Lab. not yet officially released future Technical Notee.

[4] J.Jacot, Introduction to Rapid Silicon Prototyping: Hardware-Software Co-Design for Embedded Systems on-a-Chip ICs.

[5] T.Pontius, Philips Semiconductors DTL Protocol Specification

[6] http://pww.rtg.sc.philips.com/CoReUse/arch/rabpublic/dtl/dtl.html (possibly only within Philips).

[7] http://www.altera.com/literature/ds/dsf10k.pdf

[8] http://www.altera.com

[9] http://www.aldelantetech.com

[10] P.E.R. Lippens and A.K. Nieuwland, C-HEAP CPU-controller Heterogeneous Embedded Architectures for Signal Processing, Nat.Lab. Technical note 175/97

[11] I.C. Kang and P.E.R. Lippens, C-HEAP ++ Architecture and API extensions supporting dynamic network reconfiguration, Nat.Lab Technical note 2000/303

Samenvatting

Nadat er is besloten om een System-on-a-Chip te ontwikkelen wordt de ontwerp methodologie van boven naar beneden doorlopen. Een belangrijk punt zijn de kosten van een System-on-a-Chip. Om de prijs zo laag mogelijk te houden is het van belang om een combinatie van programeerbare processoren en hardware componenten te ontwerpen, die aan de gestelde eisen voldoen. Eisen die men zou kunnen stellen aan een System-on-a-Chip zijn bijvoorbeeld: hoge snelheid, laag vermogen, zo klein mogelijk, bandbreedte moet zo hoog mogelijk zijn, moet makkelijk uitbreidbaar te zijn in de toekomst, enz. Een programeerbare processor is bijvoorbeeld erg flexibel maar totaal niet efficiënt, door deze flexibiliteit is een programmeerbare processor relatief een duur component. Hardware daarentegen erg efficiënt maar totaal niet flexibel, door het missen van de flexibiliteit wordt een hogere efficiëntie behaald, hardware relatief goedkoop om te gebruiken maar het geheel moet nu nog wel zelf ontworpen worden. Dus op het moment dat een System-on-a-Chip ontwikkeld moet gaan worden is het, om aan de eisen te voldoen, erg aantrekkelijk om een combinatie van programmeerbare processoren met hardware componenten te ontwikkelen, dit noemt men ook wel hardware-software codesign. Wordt er voor het ontwikkelen van een System-on-a-Chip alleen maar naar de prijs van het uiteindelijke component gekeken? Nee natuurlijk niet, maar waarom zou je een systeem veel flexibeler en efficiënter maken dan dat de komende jaren verwacht wordt wat je nodig hebt.
Doordat het IC ontwerp een iteratief proces is moeten er regelmatig dure masker sets worden gemaakt om de uiteindelijke System-on-a-Chip in de testfase te kunnen testen. Om het aantal masker sets tot een minimum te beperken zou je graag een platform ontwikkelen voor de systeem en IC groepen waar deze hun ontwerp op zouden kunnen testen. Doordat een programeerbare component samen met hardware componenten gebruikt gaat worden krijg je op dit moment een scheiding van systeem niveau ontwerp en component niveau ontwerp. Het systeem niveau wordt als prototype platform gebruikt en wordt door de systeem architecten aangeboden aan de component ontwerpers die daar hun ontwerp op kunnen testen en verbeteren.

Om het ontwikkelen van een prototype platform zo snel en efficiënt mogelijk uit te kunnen voeren moeten er prototype kaarten worden ontwikkeld. Deze ontwikkelde kaarten (die in een Pentium systeem worden geïnstalleerd) kunnen natuurlijk in elk gewenst project worden ingezet. Waarom is het goed om een prototype platform op een PC met PCI bus te ontwikkelen? (i) PC is het meest ondersteunde systeem ter wereld, (ii) PCI bus is werldwijd een standard, (iii) Er zijn vele PCI kaarten op de markt verkrijgbaar, (iv) PCI software drivers zijn beschikbaar voor verschillende operating systemen, (v) De PCI bus is maar een onderdeel van de standaard communicatie stack. Er is wel een protocol nodig om de verschillende kaarten onderling en met de Pentium te kunnen laten synchroniseren en communiceren. Hiervoor moet een standaard interface stack ontwikkeld worden. Door deze stack communiceren en synchroniseren de verschillende kaarten met de buitenwereld met het DTL protocol. Het grote voordeel van deze methode is dat de ontwerpers geen kennis meer hoeven te bezitten van allerlei verschillende bus protocollen. Zij ontwikkelen de applicatie waarna met een bus vertaler het DTL protocol wordt omgezet in een systeem specifiek protocol (bus of een punt-naar-punt protocol). Let wel dat de ontwerpers kennis dienen te bezitten van het gebruikte synchronisatie

protocol, in het geval van dit project is voor C-HEAP gekozen. Door deze methode te volgen kunnen de verschillende ontwikkelde applicaties zeer eenvoudig in meerdere projecten worden hergebruikt, dit komt de ontwikkel tijd zeer ten goede. De hergebruikte (stukken) van de applicatie hoeven nu niet meer ontwikkeld en getest te worden want deze stappen zijn al uitgevoerd. Je hoeft dus geen specifieke kennis meer te hebben van de applicatie maar wel van het gebruikte synchronisatie en communicatie protocol, daar deze (standaard) DTL en C-HEAP zijn bezit de betreffende ontwikkelaar genoeg kennis om het hergebruikte (stuk) van de applicatie meteen in te kunnen zetten.

Het onderzoek dat gepresenteerd wordt in deze scriptie voorziet in een standaard interface stack die gebruikt kan worden voor het synchroniseren en communiceren tussen verschillende PCI kaarten met de Pentium. Deze methode kan worden gebruikt om snel en effectief een (gedeelte) van een applicatie te ontwerpen zonder verder iets van de rest van het systeem te weten. De problemen die in deze scriptie behandeld worden zijn (i) de ontwikkeling van een standaard interface stack, (ii) synchronisatie tussen verschillende hardware blokken op verschillende PCI kaarten, (iii) synchronisatie tussen hardware- en softwareblokken.

Ik heb vooral de architectuureigenschappen onderzocht die typerend zijn voor een communicatie/synchronisatie stack voor een multiprocessor systemen. Om de effecten van deze stack te onderzoeken hadden wij eigenlijk graag de stack op een FPGA willen testen. Deze test zou hebben ingehouden dat het communicatie gedeelte samen met het synchronisatie gedeelte op een FPGA gemapt en getest zou worden. Aangezien het board wat ik moest gebruiken dusdanig kleine FPGAs bevatten kon dit niet in de praktijk worden getest, er was te weinig tijd om een andere kaart voor deze experimenten te regelen. Door mij is het geheel met behulp van Modelsim wel compleet gesimuleerd en daarna met Leonardo compleet gesynthetiseerd. Toch kan ik samenvatten dat ik geslaagd ben in het ontwikkelen van een standaard interface stack ook al kon de structuur alleen door middel van simulatie worden bewezen. Verder heb ik ook verschillende types C-HEAP ontwikkeld die ook door middel van simulatie bewezen zijn.

Summary

When there is decided that a System-on-a-Chip will be developed, the developing methodology will be walked through from top to bottom. A very important point of a System-on-a-Chip is the cost. To keep the cost as low as possible there is a need of combining processors and hardware components into one system which are meeting the requirements and constraints. Constraints which can be demanded for a System-on-a-Chip are : high speed, low power consumption, small area, bandwidth as high as possible, must be future proof (expandable), etc. For example a programmable processor is very flexible but the efficiency is very low, because of the flexibility a programmable processor is relatively an expensive component. On the other hand, hardware is very efficient but totally not flexible, efficiency is very high but everything must be developed through the hardware development engineers. So when a System-on-a-Chip is going to be developed it is very attractive to combine programmable processors and hardware components, this is also called hardware-software co-design. Is the cost of a System-on-a-Chip which is going to be developed the most important issue? Of course not, but when a System-on-a-Chip is going to be developed why would you design a more flexible and efficient system than needed in the few next years.
Because the development of an IC is an iterative process, expensive mask-sets must be made for test the System-on-a-Chip. To minimise the number of needed mask-sets you want a system for testing the complete functionality of the System-on-a-Chip. This rapid system prototyping system can be used by system groups for developing the needed software and hardware. Because of the hardware and programmable processors a separation between system-level-design and component-level-design can be made. The system level is the prototyping environment and is handed to the system groups by the system level architecting group. The system groups and the IC groups use this platform for developing and testing the designed components.

For developing a prototyping system as fast and efficient as possible some prototyping cards must be developed. It is possible to use these developed card for every project where the cards can be needed but especially for a application specific domain. These cards will be installed into a Pentium system. Why is it good to do prototyping on a PC with a PCI system-bus? (i) Most supported system world-wide is the PC, (ii) PCI bus is a widely used standard, (iii) Already boards and components for PC on the market, (iv) PCI software drivers are available for PC OS's (Windows, Linux), (v) The PCI bus is just one level in the standard communication stack.
At this moment a communication and synchronisation protocol is needed for all different cards which are developed, it is also needed to communicate with the Pentium processor. It is best to develop a standard interface stack for all kinds of data transfer. Through this stack all cards and processors can communicated through a DTL interface with every card and processor in the system. One very big advantage of this method is that the hardware developers does not have to know something about the application specific bus which is needed in the final IC. Now it is possible to develop some IP without knowing something about the application specific bus which is used, one thing that is needed are the bus wrappers, which are converting the DTL signals into the application specific bus or point-to-point protocol. Notice that it is needed that the developers have knowledge of the used synchronisation protocol, in this project C-HEAP is used. If this method is followed it is very simple to

re-use the already developed applications in other projects, this is very positive for decreasing the development costs. It is not needed to develop and test the re-used (parts) of the application because everything is already done. When a part is re-used you do not have to have some application specific knowledge of that specific part, only the knowledge of the used synchronisation and communication protocol is necessary. For this project only knowledge of C-HEAP and DTL is needed.

The research which is presented in this thesis gives a standard interface stack which can be used for synchronising and communication for different developed PCI cards with the Pentium system. This method can be used for developing applications without having some application specific knowledge.
The problems which are treated in this thesis are: (i) the development of a standard interface stack, (ii) synchronisation between different hardware blocks on different PCI cards, (iii) synchronisation between hardware-hardware, software-software, and hardware-software blocks.

My research was done in the direction which is typical for a communication / synchronisation stack for a multiprocessor system. In first instance we wanted to do a study to effects of the standard interface stack when we mapped the communication-and synchronisation stack on a FPGA. Because the FPGAs on the PCI board I had to use where to small it was impossible to map and test the standard interface stack on the FPGA. At this moment in time it was impossible for me to search or develop an other PCI board for doing the mapping and testing of the synchronisation and communication stack. I developed the complete architecture and did functional testing with ModelSim and everything was working. After ModelSim I used Leonardo for synthesising the complete design. Finally I can say that I succeeded in developing a standard interface stack although it was impossible to test the stack on an FPGA. Furthermore I have developed several kinds of C-HEAP which are proven by simulation (ModelSim) and not by mapping.

Appendix

A.1 Local Bus

The definition of the LB used for interfacing with other devices on the board is a compromise between ease-of-use, number of pins and performance. Don't think of it as a fixed standard, the whole idea of the EPLD solution is the flexibility to modify it. Current definition is based on a 32 bit, bi-directional address/data bus closely linked to the PCI-bus. However, the PCI target core takes care of configuration and address decoding, and by adding data FIFO's makes the requirements on hand shake signals a lot easier to deal with.

A.1.1 LB signals

Table 15: LB signal descriptions.

Name	target	Description
#LB_Arst	In	Asynchronous HW or SW programmed reset (active low).
LB_D (31. .0)	InOut	Bi-directional address and data bus.
LB_BA(6. .0)	In	One-hot base address select. *LB_BA* (6 downto 1) correspond to the actual PCI base address registers. *LB_BA* (1) = 1 selects registers in PCI target core. The 5 others are available for back end targets. *LB_BA* (0) does not correspond to a true PCI base address: *LB_BA* (0) = 1 is used for selecting configuration registers in the PCI target core in configuration mode.
LB_Sadr	In	Indicates that *LB_D* must be copied synchronously to local address generator, regardless of the rest of the bus state. Happens for all PCI transfers, therefore usually in vain.
LB_BEn (3. .0)	In	Byte Enable-not signals for the individual bytes of the *LB_D* data bus.
LB_RWn	In	Read (from target), Write-not (to target) direction signal for the *LB_D* data bus. Target I must drive *LB_D* whenever its *LB_BA* (I) = 1 and *LB_RWn* = 1.
LB_Mrdy	In	Initiator ready. Signals that data written to target is valid when *LB_RWn* = 0, or that initiator is ready to receive data when *LB_RWn* = 1. Allows prefetched reading to optimise speed.
LB_Mgrdy	In	Initiator Guaranteed ready. Same as *LB_Mrdy* when *LB_RWn* = 0, but more careful (thus slower) when reading since it tries to avoid unwanted side effects

		by preventing prefetch.
LB_TBusy	Out	Target Busy. Driven by *LB_BA* (I) = 1 selected target I, tri-stated by all non-selected targets. Signals not ready to receive data when *LB_RWn* = 0, or valid data on *LB_D* when *LB_RWn* = 1.
LB_DMAreq (4. .0)	Out	DMA request. Reserved for signalling target DMA readiness.
LB_DMAgr (4. .0)	In	DMA granted. Reserved for signalling DMA transfer active.
LB_Int (4. .0)	Out	Interrupt requests. Any high bit, if enabled in IntMASK register of PCI target core, will generate an INTA signal on PCI bus.
LB_ClkEna		Signal from PCI target core to back end clock buffer. Mechanism to prevent cross talk from clock when downloading firmware in back EPLDs.
LB_Spare (7..0)	InOut	For future expansions.

A.1.2 LB protocol

The protocol is as roughly as follows. When *LB_Sadr* = 1, every target on the LB copies the relevant bits of *LB_D* to its local address generator, regardless the rest of the LB state. An individual target is selected afterwards by the one-hot *LB_BA* base address selection. Whenever selected, the target must drive *LB_TBusy* to indicate whether or not it is busy. *LB_RWn* indicates the direction of the data transfer, *LB_RWn* = 0 for writing *LB_D* data to the target, and *LB_RWn* = 1 when reading from the target. A selected target must drive *LB_D* whenever *LB_RWn* = 1. *LB_Mrdy* indicates initiator ready: whenever *LB_Mrdy* = 1 and *LB_TBusy* = 0, data is actually transferred. FIFO's automatically handles pipeline delays in the PCI target core, see section of FIFO's. At the end of the transfer, the LB is always parked in write mode (*LB_RWN* = 0) at the last selected target. In practice this means that a local target typically has no knowledge of the 'end' of the transaction. It simply reacts as a target to the LB control signals LB_BA, *LB_RWn*, and *LB_Mrdy* and should not make any further assumptions. (If needed you can use the trailing edge of *LB_RWn* for the end of a read transaction, and/or *LB_Sadr* = 1 as the potential start of new transaction hence the definite end of any previous transaction.) Also, any target should be prepared to handle a burst of data.

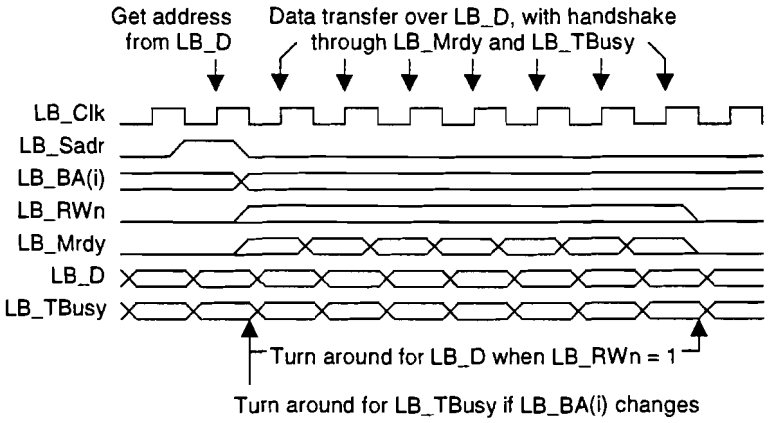

Figure 48 : LB data transfer protocol.

When *LB_Sadr* = 1, the data bus *LB_D* and byte enables LB_BEn contain the complete PCI address and command, but again, this information is handled by the PCI target core and should normally be ignored by the local target.