Master of Science in Mechatronics Engineering

Dissertation

HARDWARE/SOFTWARE CO-DESIGN OF EMBEDDED SYSTEMS

By

Marcelo Juan Peña Zúñiga

Mads Clausen Institute for Product Innovation
University of Southern Denmark
Sønderborg, Denmark

June 2008

Master of Science in Mechatronics Engineering

Dissertation

HARDWARE/SOFTWARE CO-DESIGN OF EMBEDDED SYSTEMS

By

Marcelo Juan Peña Zúñiga

Mads Clausen Institute for Product Innovation
University of Southern Denmark
Sønderborg, Denmark

June 2008

ACKNOWLEDGEMENTS

It has been an interesting journey since I finished high school and decided to study electronics engineering. I would like to think that this thesis is the perfect way to summarize my work over the past few years. I would like to take this opportunity and express my deepest gratitude to all the people that has helped me along this path.

This thesis could not be completed without the guidance of my supervisor, Assist. Prof. Krzysztof Sierszecki. I am grateful for his enthusiasms, patience and dedication to this project. I admire his commitment to embedded systems innovation. I appreciate all the counseling time and the constructive feedback that has driven me to overcome frontiers throughout the research over the last year.

I express my gratitude to all the people or the Mads Clausen Institute for Product Innovation at the University of Southern Denmark for the financial support.

I would like to thank my parents and my family for keep believing in my dreams and helping me to move forward with their crystallization. Thank you for all your teachings and advices, they my most valuable tool during difficult moments where critical decisions have been taken while being far away from home. This work is dedicated to all of you.

Finally, but not least I thank God for letting me live long enough to conclude this goal in my life. I thank you for enlighten me and showing the path.

- Marcelo Juan Peña Zúñiga

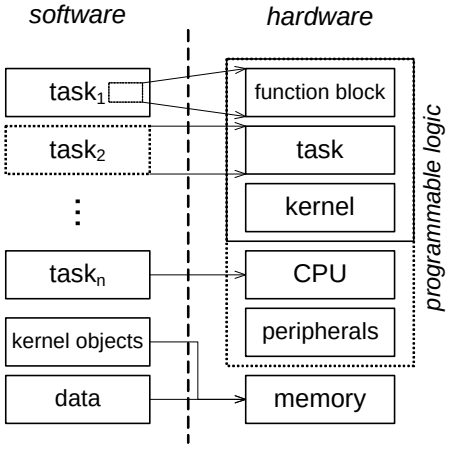
ABSTRACT

The present stage of computer development can be described as the post-PC era or pervasive computing with many interconnected tiny devices embedded into various equipment. An example of such system is wireless sensor network that is used in many application areas, including environment and habitat monitoring, healthcare applications, home automation, and traffic control.

Control systems, previously built around central computer, have not avoided the transformation either and today they are implemented as a network of embedded control systems. Each embedded system is a mixture of software and hardware, where software is used for features and flexibility, while hardware is used for performance.

Typically, in such setting, hardware consists of predefined microcontroller with a micro-processor, memories and a set of peripherals. However, recent advances in the field of programmable logic devices allow flexible and simultaneous design of both hardware and software. This approach is called hardware/software co-design.

The Software Engineering Group at the Mads Clausen Institute addresses the current challenges in the area of embedded control systems design by developing suitable design methods and implementation mechanisms. The COMDES (Component-based Design of Embedded Systems) framework and HARTEXμ (Hard Real-Time Executive) family of kernels are the main contributions of the group.



The goal of the project is to explore the possibility of implementing parts of the COMDES/HARTEXμ systems in hardware. Even though COMDES/HARTEXμ applications are implemented as a mixture of software and hardware, the COMDES/HARTEXμ platform itself is entirely implemented as software. However, it would be beneficial to move the time-critical and computationally expensive calculation to hardware counter-parts. For example, the kernel overhead could be practically eliminated by moving the kernel functionality into hardware; or, task execution time may be drastically reduce by implementing the whole task or some of the encapsulated function blocks in hardware, as demonstrated in the figure.

As a result, a suitable platform should be selected that will be used to carry on further developments in the area of hardware/software co-design. The project will investigate possible

ways of implementing the HARTEXμ operational environment in hardware, while keeping the application tasks as software entities for flexibility reasons. The tasks will be executed in an industry-accepted type of CPU, e.g. the MicroBlaze soft-processor from Xilinx. The ultimate solution shall target embedded applications, and therefore, the work has to take into account limitations of embedded systems.

Marcelo Juan Peña Zúñiga
Sønderborg, Denmark
2nd June 2008

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

The journey of embedded systems

1.1 INTRODUCTION

The embedded system development process has been subject of research over the past decades. Design methodologies have been proposed to speed up the development process, dealing with diverse issues, such as hardware platform performance, software reusability, higher design abstraction levels, software/hardware verification and validations, among other. All this issues convey towards the same goal: shorter time-to-market, higher performance and low cost[1][12][15].

Economics has been and always will be an essential factor in embedded systems development. The ubiquity of embedded systems demands shorter launching time and higher performance. Hardware development has been an excellent example when it comes to mass production, while incorporating emerging technologies. It has not been the case for software. In principle, the software development is based on fitting the hardware while satisfying the embedded system requirements. Dr. Makimoto expresses the hardware-software relationship as a pendulum cycle: "When large number of new technologies, such as devices, architectures, and software, appear, the semiconductor industry as a whole moves towards standardization[24]". This principle is called "the digital wave" (also known as Makimoto's wave) which has dictated the hardware/software development relationship since the 1960's (Figure 1.1.1).

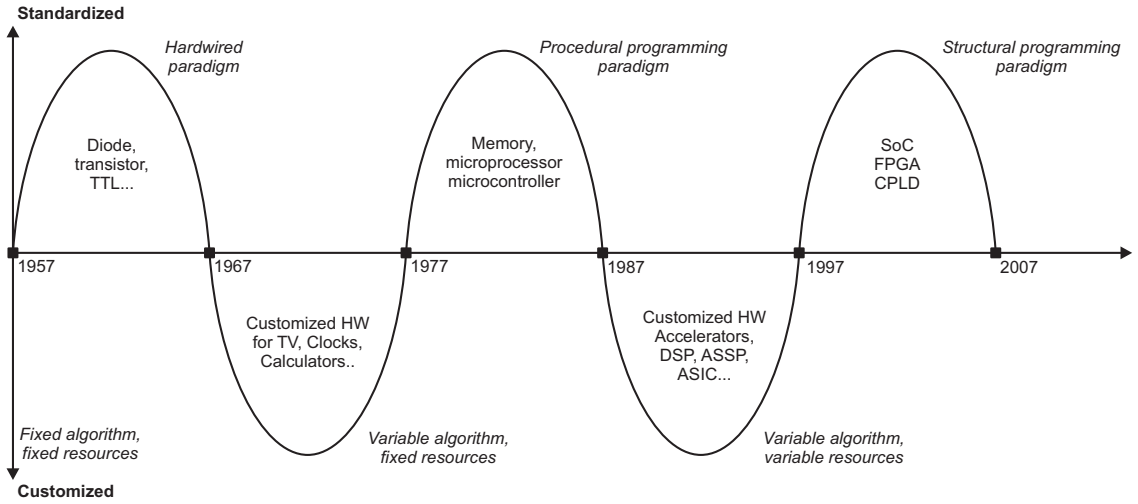


Figure 1.1.1 Makimoto's wave.

This alternated wave process also indicates that once the semiconductor industry has moved to standardization, this leads to a customization where new computational paradigms for embedded systems have to be developed.

Traditionally, in the first waves of this cycle, computing execution of algorithms is accomplished with a couple of methods: the first method is the execution of algorithms using von Neumann architecture[16] where microprocessor and microcontrollers are programmed using software. The second method consists in fabricating an application specific processor or integrated circuits. The former method is a more flexible solution with performance degradation, the later method has been used for particular applications, it may not be cost-effective to modify or add more features but it presents higher performance.

Advances in silicon technology had made possible to embed inexpensive processor, sensor, and actuators in just about everywhere and anything. Pervasive computing holds the promise of erasing the barriers between the world of hardware and the world of software. This promise is starting to take shape with advances in programmable logic devices.

Programmable logic devices provide the opportunity of hardware acceleration through custom logic functions. One of early examples is the coprocessing unit which allows the computation of mathematical functions in a fraction of the time that it would take by using the same resources as in the microprocessor/microcontroller. This open a whole new field for embedded system, resulting in today's digital signal processing (DSP) units where the hardware is dedicated to perform complex computations like audio and speech signal processing, spectral estimation, image processing, biomedical signal processing among other fields. The hardware acceleration performed by these devices leads to real-time processing, a highly desirable characteristic in embedded systems.

Research is being conducted on these areas by the software engineering group at the University of southern Denmark[11][10], especially on operational software design and component-based design resulting in the development of HARTEXμ family of reconfigurable real-time kernel and COMDES framework[12][13], and the associated software design methods. At the current stage with the emerging programmable logic devices technology and the developed software design methodologies, posses the question: can the components be implemented in hardware? If so, which components can be implemented? Ultimately leading to the question, can a kernel be implemented in hardware?

This thesis provides a comprehensive guide towards the implementation of software components in hardware. The research presented here tries to assemble the "big puzzle". This first chapter provides an introduction to embedded systems and the associated development cycle, followed by an overview of the research work at SDU on HARTEXμ and COMDES, and the software/hardware co-design challenge. Also a brief introduction to reconfigurable computing and the possible impact on the next wave of customization in the programming paradigm will be presented.

The next piece in the puzzle, chapter two, is an introduction to the programmable logic devices, the current devices that can be used to implement custom logic devices, and their basic structure. Chapter three presents the programming languages for the programmable logic devices, and the possible abstraction levels from where custom logic can be implemented.

It is not enough to have programmable devices; the application software containing the embedded systems tasks needs to be executed by a processing unit. Microprocessor's core can be synthesized and implemented in logic devices introducing flexibility in to the system. This discussion is extended in chapter four along with various cores to consider for this project. During this research, the capabilities and features of the soft-processor are explored. Chapter five presents an introduction to the Spartan-3 boards family used in this project. Chapter six presents a review of all the tools provided by Xilinx used in this project. Chapter seven discuses the porting of HARTEXμ to the MicroBlaze soft-processor and the HW/SW co-design process. Chapter eight presents the project conclusions and the road map for future work.

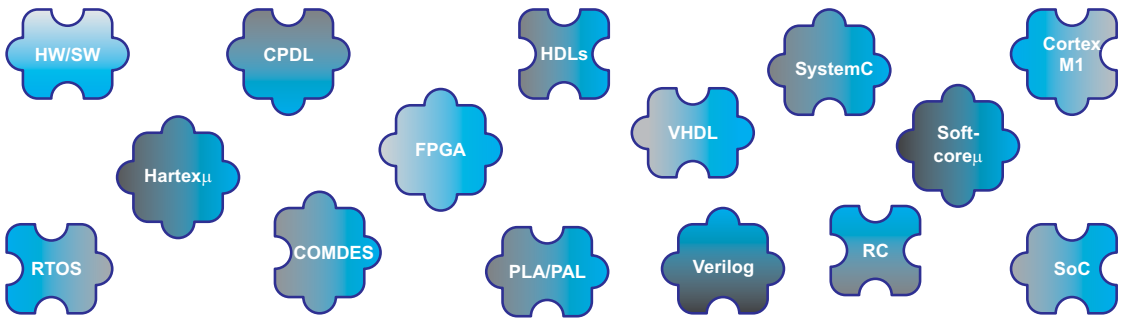


Figure 1.1.2 Project puzzle.

1.2 SOFTWARE FOR EMBEDDED SYSTEMS

What is an embedded system? This has been a controversial topic over the decades, making it difficult to define such a system. Traditionally computer science has tended to view embedded systems as "messy", consequently software for embedded systems has not benefited from the highly developed abstractions, i.e. personal computing software. Engineers writing embedded software are rarely computer scientist; instead they are experts in the targeted application with a good understanding of the embedded system architecture. The experts have the best understanding of how the embedded system interacts with the physical world, after all that is the main goal[1].

Developing embedded systems is not an easy task. Conventionally hardware is selected with the software application in mind; however the software has to be tailored to meet the application requirements while fitting the hardware environment. Figure 1.2.1 shows the typical development cycle for an embedded system. Through the process, the selected processing unit (microprocessor, microcontroller, DSP, etc) remains the same, whereas software will go under a continuous development process (i.e. V-cycle) until all the requirements are satisfied.

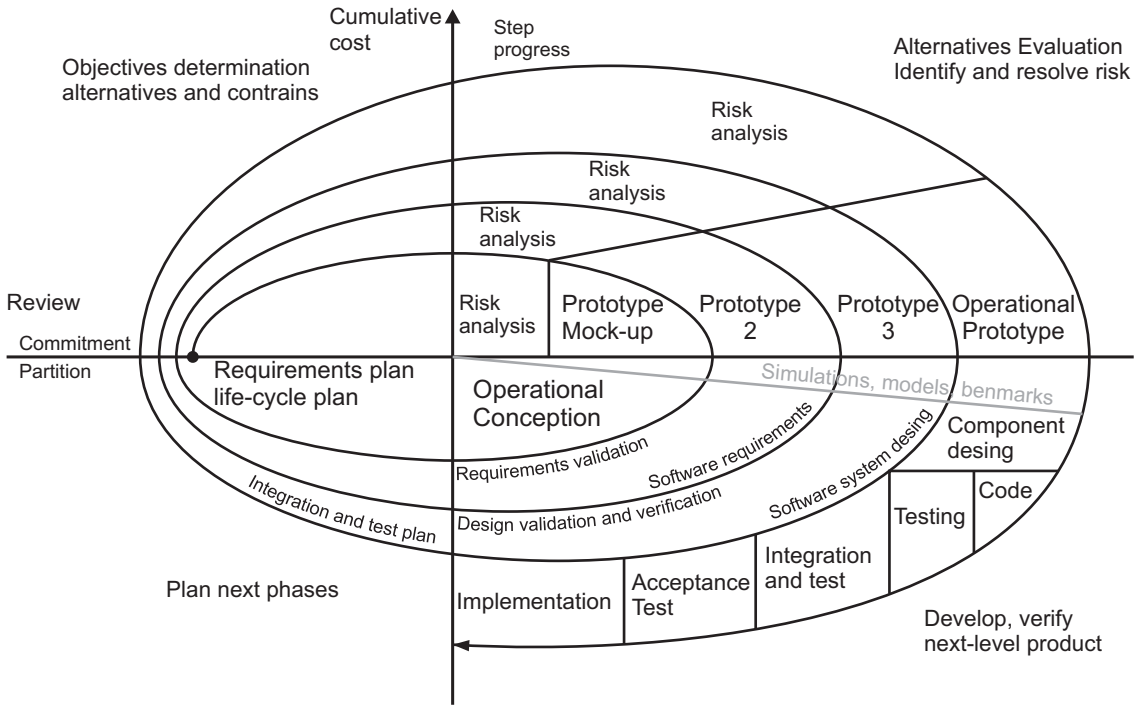


Figure 1.2.1 Embedded system development cycle.

Since embedded systems interact with the physical world, there is a need to provide an application environment where all the different interaction processes (tasks) are assigned with computation resources and allocated to specific computation time, making the embedded system responsive and giving concurrency to the tasks. This is crucial when dealing with real-

time systems. The application environment comes in the form of an operating system or a kernel. Essentially a kernel is a task schedules and an event manager (where events can be from an external source or internal generated by the operation of the system). In the following subsection a short discussion is presented regarding the design and development of embedded system using HARTEXμ kernel.

1.2.1 DESIGN USING REAL-TIME OPERATION SYSTEMS – HARTEXμ

Most embedded systems involve real-time computations; some of these have hard deadlines, requiring not only real-time throughput, but also low latency. Tasks need to compete for the resources in order to go through required real-time computations.

Embedded systems incorporate a real-time operating systems, which offered scheduling services highly specialize on real-time needs, besides the standard services such as I/O, task and memory management. The schedule is based on priorities; they can be assigned base on the principles of rate monolithic, deadlines, or even ad hoc.

Modern Fixed-Priority Schedule Theory (FPST) supplies a widespread framework for reliable real-time systems[10]. Enhancing operating systems with FPST methods leads to a new generation of so called safe real-time kernels. These hard real-time kernels support predictable scheduling of hard real-time tasks; predictable interaction between real-time tasks; predictable and deterministic operation of the kernel subsystem. These guidelines were used to develop the distributed real-time kernel HARTEX at the University of Southern Denmark, Mad Clausten Institute Software Engineering Group under the supervision of Prof. Christo Angelov[10][11].

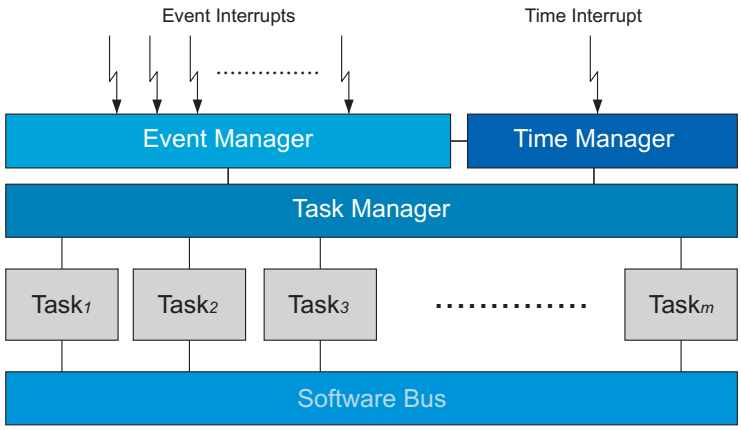


Figure 1.2.2 HARTEX component-based architecture.

The HARTEXμ framework consists of a component-based kernel architecture, where the event manager, time manager, task manager and software bus are encapsulated. The main feature of HARTEXμ is the use of Boolean vectors[10] reducing to a fraction of the time that would take if queues were used, leading to a jitter-free kernel.

HARTEXμ is the kernel considered in this project. The subsequent research is done towards the partial or full implementation of the different components of this kernel. There are different versions of HARTEX being developed, however as a starting point, the simplest version is used, the so called "HARTEXμ Student edition".

1.2.2 DESIGN VIA COMPONENT BASED FRAMWORK - COMDES

The ubiquity of embedded systems mandates the use mass production methods for software by means of pre-fabricated components and higher levels of abstraction. While kernel frameworks structurized embedded software design, they do not couple with the reusability issues and abstraction levels required for industrial production of embedded systems. The main problem to be addressed is the systematic development of a software framework for embedded applications while considering the true nature of embedded systems, which are predominantly real-time control and monitoring systems.

A solution has been developed at the Mads Clausen Institute (SDU), a domain specific framework for embedded control systems; Component-based design of software for Distributed Embedded Systems (COMDES). In principle, COMDES is defined as "a set of executable models that are due to specify relevant aspects of system structure and behavior". A system represented in COMDES consists of three types of components: function units, function unit activities and function blocks[12].

Functions units are conceived as "software integrated circuits" which encapsulated dynamically scheduled activities. Communication signal driving other function units are also encapsulated. Function blocks are considered activities which have been built from reusable executable components. Hybrid state machines are use to specify the behavior of the activities. The hybrid state machine is a hierarchical executable model that takes into account reactive and transformational phases of system behavior[12][13].

1.3 THE HARDWARE-SOFTWARE CO-DESIGN CHALLENGE

The continuous progress in semiconductor technology has enable customize logic devices to become a reality. This opens a new possibility of design for embedded systems; a co-design process where hardware and software can be tailored to fit specific requirements. Current methods for embedded systems require specifying and designing hardware and software separately. The hardware-software partition is decided a priori and remains trough the entire development process, since changes in this partition will require extensive re-design[28].

In principle the co-design process can be composed by the following phases: modeling, partitioning, co-synthesis, co-simulation, and verification. The modeling phase determines the specific hardware and software to be use along with a suitable methodology. There are three

different paths to start the modeling phase; an existing software implementation, and existing hardware implementation, or starting from zero. In the partitioning phase, the different components are mapped to software or hardware according to the specified constrains such as time, size and cost.

Co-synthesis makes use of tools to synthesize the software, hardware and their interaction. Co-design tools shall generate hardware/software inner-process communication and schedule software processes to meet the time specifications. The co-simulation and verification phase executes all the system components functioning together in real-time.

It should be notice that co-design still is a very new field and researchers in this area have rapidly evolved co-design methodologies. The barriers between software and hardware start to "blur" when software can configure hardware, which leads to reconfigurable computing.

1.4 THE EMERGING RECONFIGURABLE COMPUTING PARADIGM

Reconfigurable computing (RC) is emerging as the new paradigm for satisfying the simultaneous demand for application performance and flexibility. Reconfigurable computing promises an intermediate tradeoff between flexibility and performance, by making use of hardware that can be adapted at run-time to facilitate grater flexibility without compromising performance. Reconfigurable architectures can exploit fine-grain and coarse-grain (section 2.6) parallelism available in the application due to the adaptability[15]. The RC paradigm is the potential candidate to bring stability in the digital wave alternation (Figure 1.4.1).

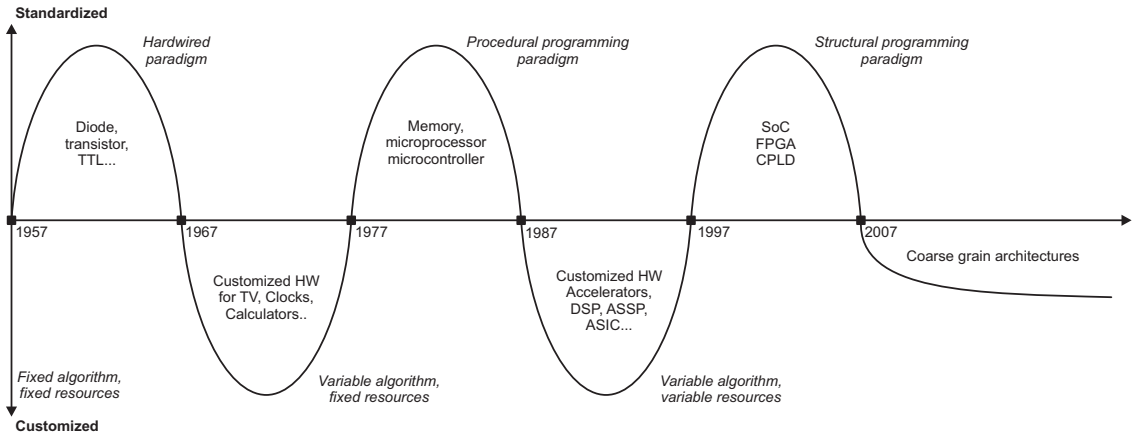


Figure 1.4.1 Makimoto-Hartenstein wave.

The reconfigurability of the hardware allows the adaptation of itself for specific computations in each application to achieve higher performance compared to software. Complex functions can be mapped onto the architecture achieving higher silicon utilization and reducing the

instruction fetch and execute bottleneck[14] [15], this is referred as "von Neumann syndrome[16]". The main components of a reconfigurable computing system are: data path units, configware, morphware and flowware. In the following subsections a short overview is provided along with a comparison to their von Neumann system counterpart.

1.4.1 DATA PATH UNIT

The data path unit (DPU) executes computation operations and data-stream-driven data transfer among processing units inside data path arrays (DPA). In the case of von Neumann architectures, the DPU includes an instruction sequencer and a program counter, however in the reconfigurable computing paradigm; the DPU stands alone, since reconfigurable computing systems are data-stream-driven, where program counters are replaced by data counters (Figure 1.4.3). In this case the data counters are co-located with the memory blocks[24].

1.4.2 CONFIGWARE VS. SOFTWARE

Software is considered to be structural programming; in the von Neumann architecture software means instruction-stream-base programming in time, due to the CPU nature, where the computational resources are centralized, only one DPU can be executed at any given time. Configware means procedural programming where data-stream-based happens in time and space (Figure 1.4.1). A configware compiler generates two different kinds of codes: a mapper doing the placement and routing generating configware code from the so called "source program". The other kind of code is scheduler generating data stream code (flowware). The configware code is applied before the run time, so the platform (FPGA, CPDL, etc) can be configured. During the run time there is no instruction fetch. The flowware code is a data schedule to organize data-streams running through the previously configured platform[24].

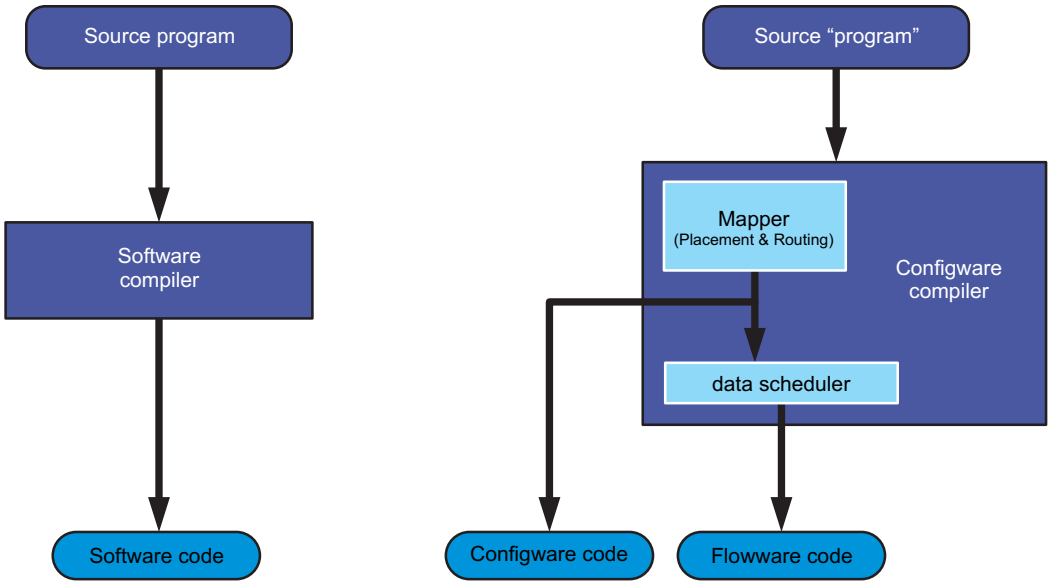


Figure 1.4.2 Software vs. Configware engineering.

1.4.3 MORPHWARE VS. HARDWARE

In the reconfigurable computing paradigm, there is a division in hardware platform. The soft, reconfigurable platform is called morphware, which is configured by the configware. The classical hard platform remains named hardware[24].

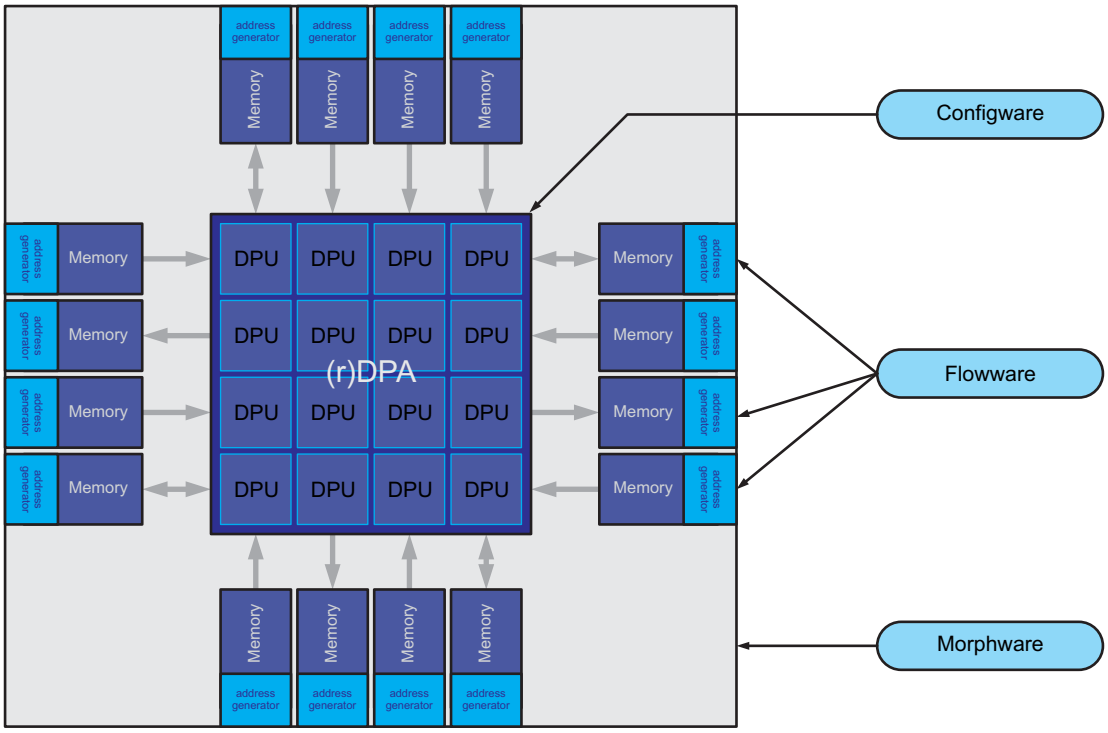
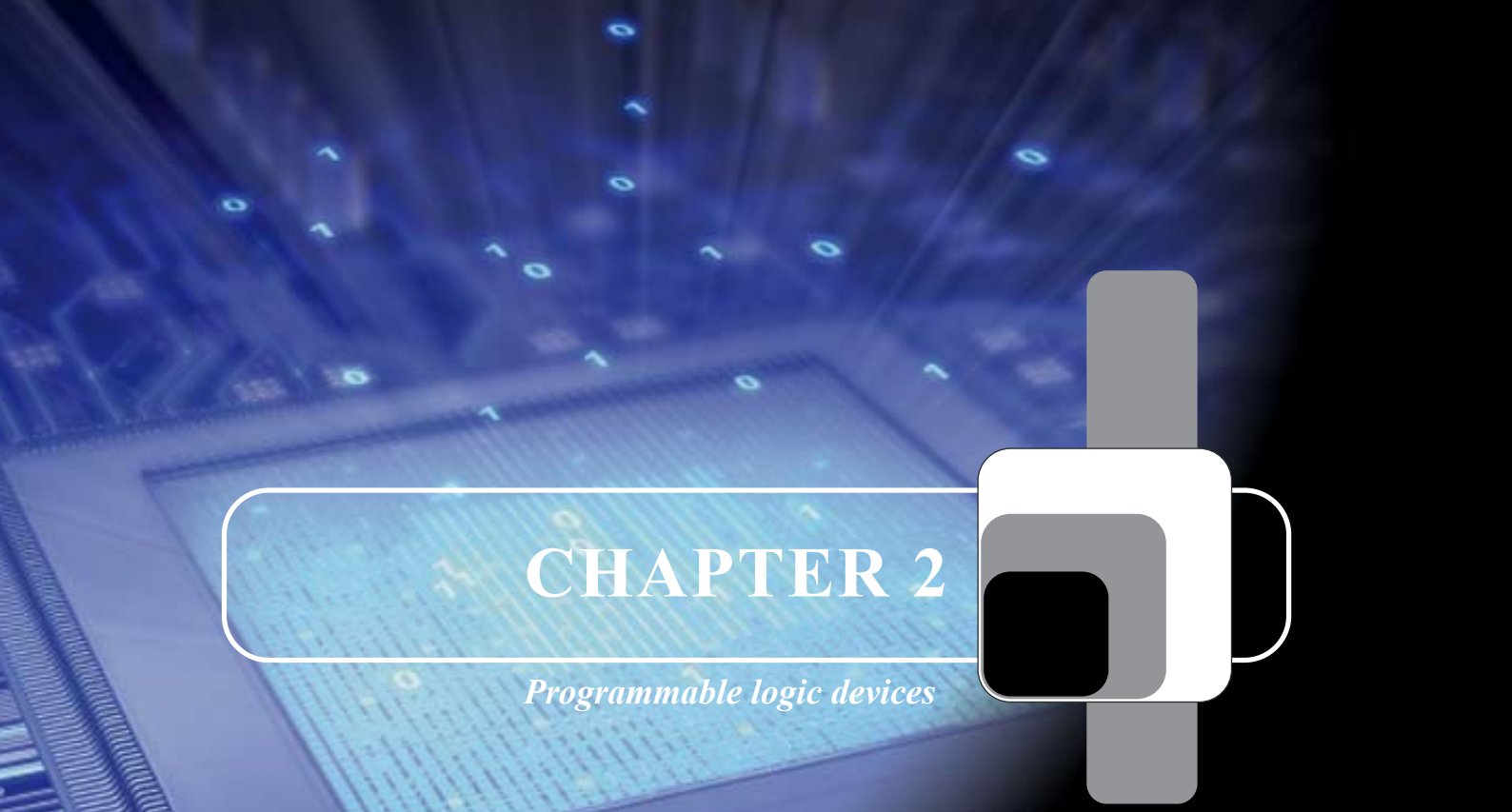


Figure 1.4.3 Reconfigurable computing processing machine.

1.5 SUMMARY

This chapter has presented the cornerstones for the project. The fundamental component to work in this project is the HARTEX real-time kernel. Several directions can be followed from here, the two most appealing are: the possible implementation of HARTEXμ kernel as a hardware accelerator, or the implementing of a reconfigurable computing system and "emulating" the kernel functionality with the true parallelism offered by this paradigm. The subsequent chapters will provide a clearer picture about the feasibility of these options for this project.



CHAPTER 2

Programmable logic devices

2.1 INTRODUCTION

The goal of this chapter is to provide a general overview and basic understanding of the logic devices technologies and options available. Programmable logic devise (PLD) are standard "off-the-shelf" integrated circuits that are composed in such a way that design-engineers can configured them to perform certain custom functions.

PLDs were introduced in the mid 1970's. They can contain anywhere form a few hundred to tens of thousands of logic gates, such is the case with programmable logic arrays going up to complex programmable logic devices, this devices will be discuss in greater detail in this chapter, including the devices in the middle range like generic logic arrays. Advances in very large integration scale technology have brought devices with hundred of thousands of logic, such as field-programmable gate arrays, introduced later in this chapter.

During the 1980's, another approach was taken in the circuit customization area. Integrated circuits were design targeting specific applications, leading to the so called application-specific integrated circuits, which are changing the way embedded systems are design. This topic is also covered in this chapter along with the design methodologies, trades and drawbacks.

2.2 PROGRAMMABLE LOGIC ARRAY

A way to design combinational logics is to use the digital gates and connect them with wires. The main disadvantage of this technique is the lack of portability; Programmable Logic Arrays (PLA) are the one-chip solution to implement these logics. A PLA is a circuit that allows the implementation of Boolean functions in a sum-of-products or rarely product-of-sums forms; these forms are also known as canonical forms. A PLA implementation consists mainly of an input buffer for all the device inputs, a programmable AND matrix (programmable logic plane) followed by the programmable OR matrix and then the output buffers[9].

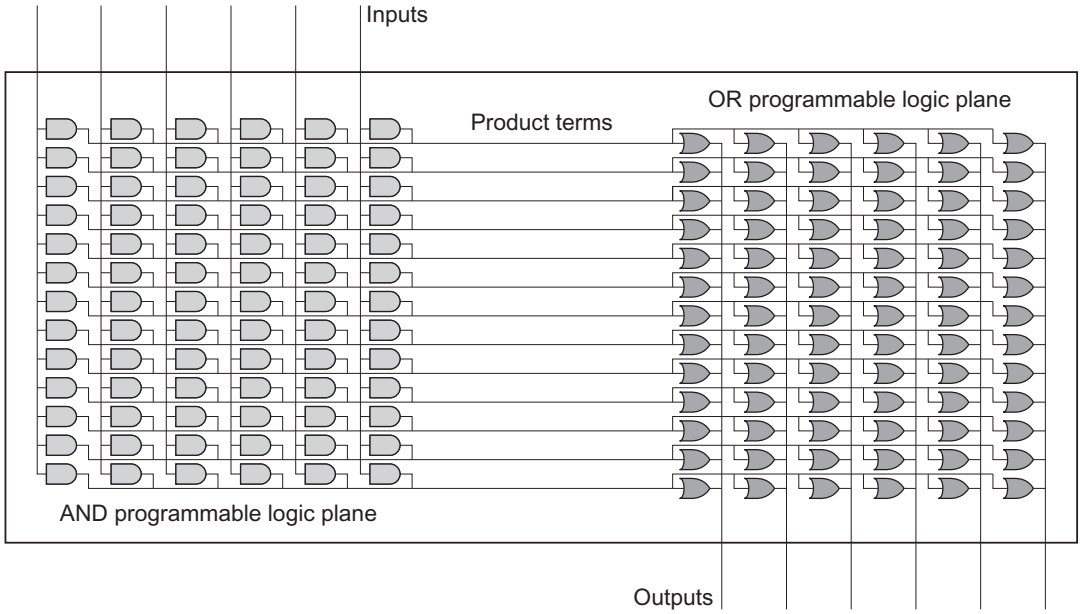


Figure 2.2.1 Conceptual PLA structure.

The programmable logic plane is a programmable read-only memory array that allows the routing of the digital signals in the device pins to the output logic macrocell (OLMC), which are usually registers and buffers.

PLA programming was done electrically by means of binary patterns; however software products were available from the vendor-supplier or sometimes third-party software like DATA/IO and PALSAM. The latter was a very popular hardware description language (Section 3.2) in the early 1980's; it was used to express the Boolean equations in a text file which was converted into a "fuse map" file. This file was use in combination with the vendor-supplier software.

A variant of PLAs was implemented by Monolithic Memories, Inc. which had the same programmable AND matrix plane, however the OR matrix is fixed. This logic device is known as programmable array logic (PAL).

Figure 2.2.2 represents a PLA implementation of Boolean functions: F_1, F_2, F_3 and F_4. Basically the set of inputs A, B and C in conjunction with their inverse are use as entries for the programmable AND matrix giving the product terms which are later used with the programmable OR matrix generating the sum-of-products output. Fundamental Boolean algebra tools to be use in the implementation of PLA/PLA are Karnaugh maps and De Morgan's laws.

Both tools can help to reduce the size of a PLA/PAL implementation. Physically PLA and PLA consist of programmable matrices of NAND or NOR gates which are easier to implement with transistor logic.

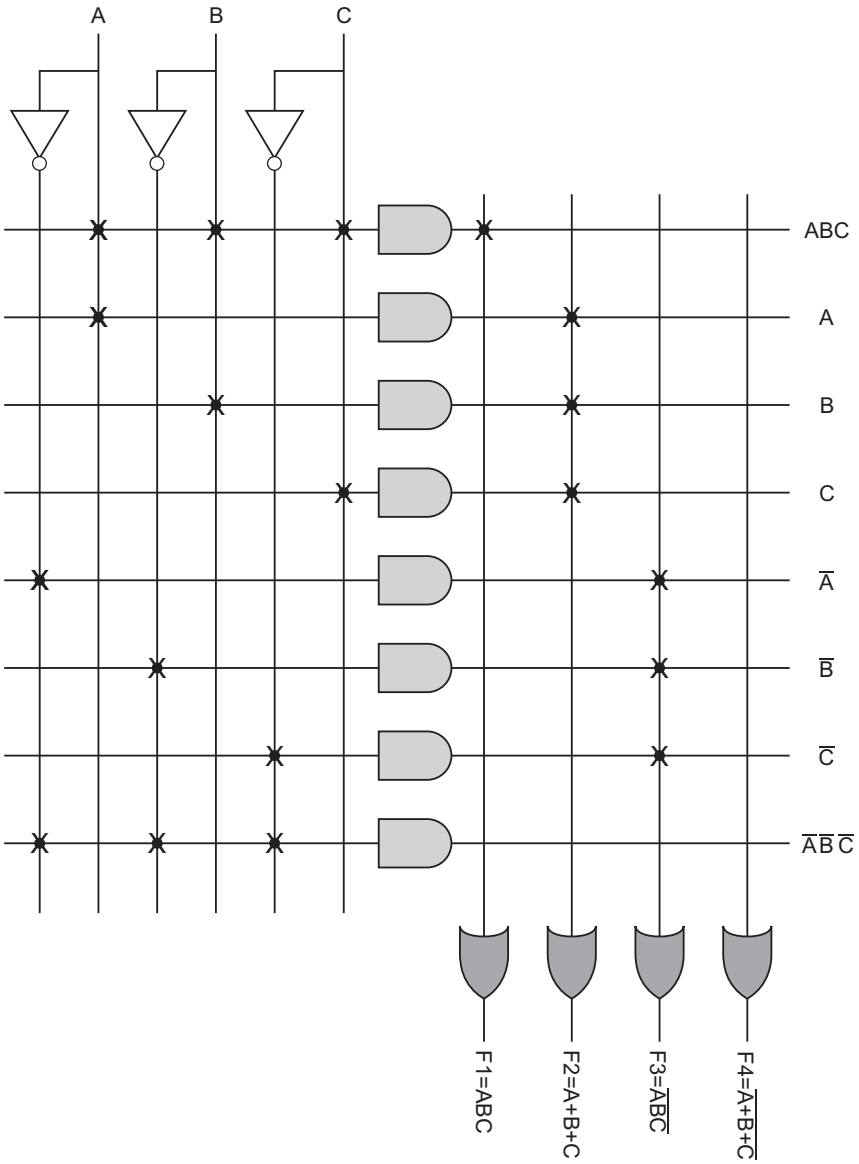


Figure 2.2.2 Implementation of Boolean functions in PLA.

2.3 GENERIC ARRAY LOGIC

The generic logic array (GAL) was developed by Lattice semiconductor in 1985 and latter on it was licensed to other manufactures. This device improves the concept of programmable array logic's. It keeps the same structure as a PAL, however the programmable AND matrix is reprogrammable, since the logic plane consists of electrically erasable CMOS cell. It also has a fix programmable OR matrix and programmable output logic macrocells which provide more flexibility.

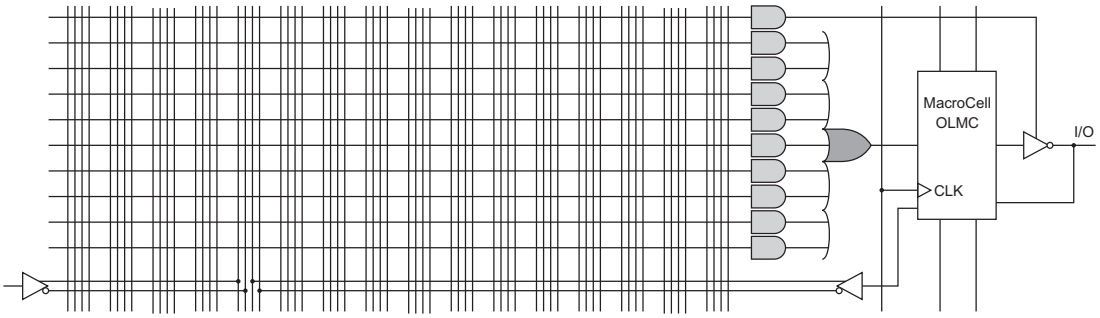


Figure 2.3.1 Typical GAL structure.

GAL programming is done via cell activation by means of connecting it to the corresponding row-column intersection. This can be achieved by PAL programmers or in-circuit programming. GAL proves to be useful in the prototyping stages of a design process.

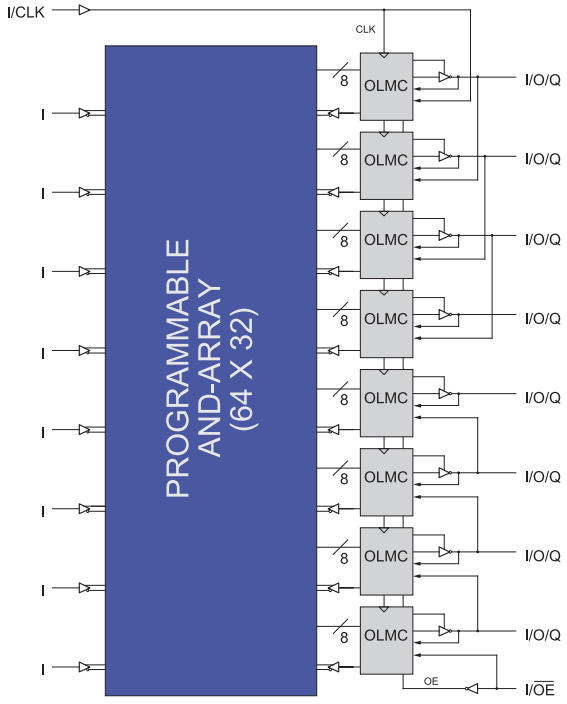


Figure 2.3.2 GAL functional block diagram[36].

2.4 COMPLEX PROGRAMMABLE LOGIC DEVICE

The complex programmable logic device (CPLD) extends the concept of programmable logic devices to a higher integration density allowing the implementation of systems efficiently in a smaller silicon area. A CPLD consists on multiple so called logic array blocks (LAB). The routing and connection of blocks is done by means of a programmable interconnect matrix, which makes an efficient use of the silicon leading to a better performance[9].

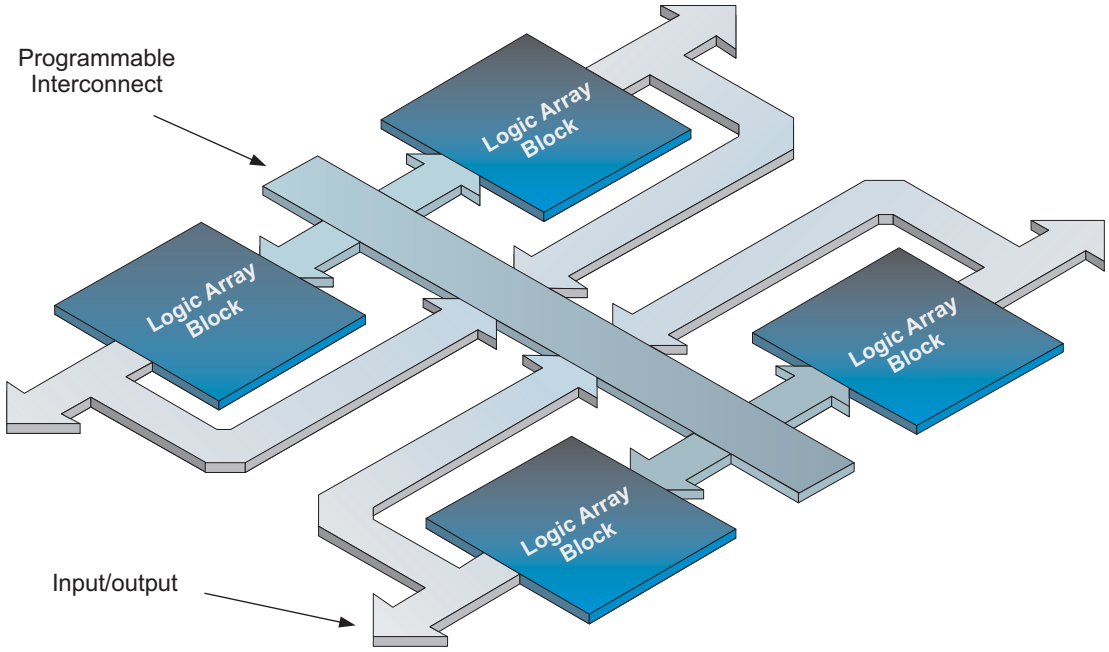


Figure 2.4.1 Typical CPLD structure.

The programmable interconnect matrix (PIM) allows the connection of the input/output device pins to the entry or output of a logic array block, PIM also allows the interconnection of different LABs, even more it can connect the LAB to it self. There are two possible matrix configurations that PIM can use: interconnection by arrays or by multiplexers. The former is based on a row-column EECMOS cell matrix; it has the same activation process as a GAL. This configuration allows a total interconnection between the inputs and outputs among the different LABs. This flexibility has a price; the device performance is diminished, requiring higher energy consumption and a bigger silicon area.

On the other hand, interconnection by multiplexes requires at the entry of each LAB a multiplexer. The programmable interconnection paths are connected to the entry of a fix number of multiplexer in each LAB. The channels of the multiplexers are selected in such manner that only one path of the interconnection matrix can be connected. The drawback with this configuration is that the multiplexers do not have access to all the paths in the interconnection matrix. To overcome this routeability issue, multiplexer with more channels can be used,

allowing paths combination of the interconnection matrix to be routed to any logic block. This solution also has a price, as the number of channels per multiplexer increases the power consumption raises, the device performance is diminished and the integration scale in the silicon area is reduced.

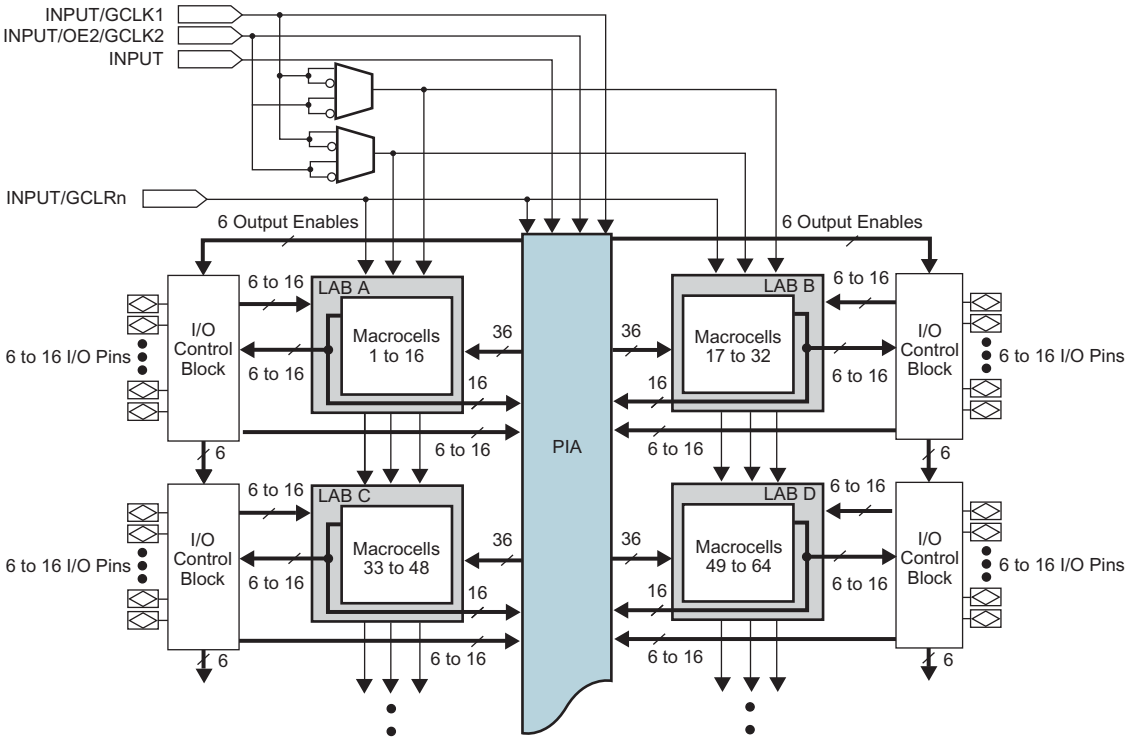


Figure 2.4.2 CPLD functional block diagram using interconnection arrays[33].

The logic array blocks resemble the structure of a PAL, each one of them contain a AND gate matrix, OR gate matrix and a configuration for the product terms allowing their distribution throughout the different macrocells of the LAB. The LAB size is measurement used to indicate the implementation capacity of the CPLD.

The CPLDs macrocells are very similar to the ones present on the PLA/PAL regarding structure. CPLDs have two types of macrocells: input/output macrocells and buried macrocells. The buried cells can not be connected to the device pins; instead they are directly connected to the PIM. This feature allows the storage and handling of intermediate computation stages for other macrocells to use.

CPLDs had increased their popularity in recent years. They can be tailored to meet any particular function, proving to be excellent coprocessing units, furthermore low power consumption CPLDs have been developed, targeting portable applications and replacing CPU's, such as ALTERA's MAX IIZ. Other applications include: Pin/Port improvement and expansion, peripheral control and interface bridge for different I/O protocols.

2.5 APPLICATION-SPECIFIC INTEGRATED CIRCUIT

With the advances in very large integration scale technology during the 1980's engineers had the possibility of customizing integrated circuits according their own applications. These so called application-specific integrated circuits (ASIC) were approached from two design methodologies. The first use chips containing arrays of prefabricated gates (gate arrays). In second methodology chips are base on libraries of standard functions cells (standard cell). These methodologies allowed the ASIC design to focus on meeting time-to-market and customer specific requirements.

ASIC chips are not just application specific, they are in fact costumer specific. The term ASIC is quite controversial, it can be argued that devices like memories and microprocessor can be considered as application specific, however on the other hand manufactures of customizable logic devices argue that this type of devices are the only true ASIC. As a rule of thumb, if a device can be found in a data-book, then is most likely not an ASIC. Since 1980's semiconductor technology has undergone exponential changes allowing higher integration density. This has increased the design options of ASIC technologies, there are three types: Full-custom, standard-cell based and gate-array based ASICs[4].

FULL-CUSTOM ASIC

In a full-custom ASIC an engineer designs part or the entire logic cell, circuitry and the specific ASIC layout. This approach diverges from the idea of using pre-tested and pre-characterized cells. This makes sense if only there are no appropriated cell libraries fitting the entire design. It could be the case that existing libraries are not fast or small enough; even more they could have a higher power consumption that the design requires. When designing full-custom ASICs the engineer has to "manually" design and optimize each single primitive logic function or transistor, manipulating individual geometric shapes which represents the features in the chip; hence the term "polygon pushing". This process results in compact chips with the highest possible speed and lowest power dissipation. The ASIC manufacturing is a declining market since fewer and fewer full-custom ICs are being designed, mainly due to two factors: high cost of the design process and once an ASIC is manufactured it can not be changed or updated.

STANDARD-CELL BASED ASIC

Cell-based ASIC or cell-based integrated circuit (CBIC) makes use of pre-defined logic functions or cells known as standard cells. The functions are made available to the designer via cell libraries. Typically these libraries begin with gate level primitives such as AND, OR, NAND, NOR, XOR, inverters, flip-flops, registers and so on. Cell libraries also include more complex functions like adders, multiplexer, decoders, ALUs, shifters and memory. In some cases they may include even more complicated microprocessor/microcontroller support

functions such as dividers, multipliers, serial port, event timers, real-time clock, etc. These last functions are known as Megacells or system-level macros (SLM).

Schematic capture or HDL synthesis is used to create standard cells. The ASIC designer defines only the placement of the standard cells and the interconnections across the CBIC; however the placement can occur in any part on the silicon, meaning customization of the CBIC to any particular design. Each standard cell in the library is constructed using the full-custom methodology therefore it is used by the designer knowing that it has been tested and characterized reducing time to market and risk.

Standard cell are design to fit together like wall bricks. Power busses run horizontally on metal lines inside the cells. This allows process automatization when assembling cells by placing them in groups, fitting horizontal rows. The rows stack vertically to form flexible blocks, allowing the possibility of connecting between blocks built from several rows of standard cells to other standard-cell blocks, even more to other full-custom logic blocks.

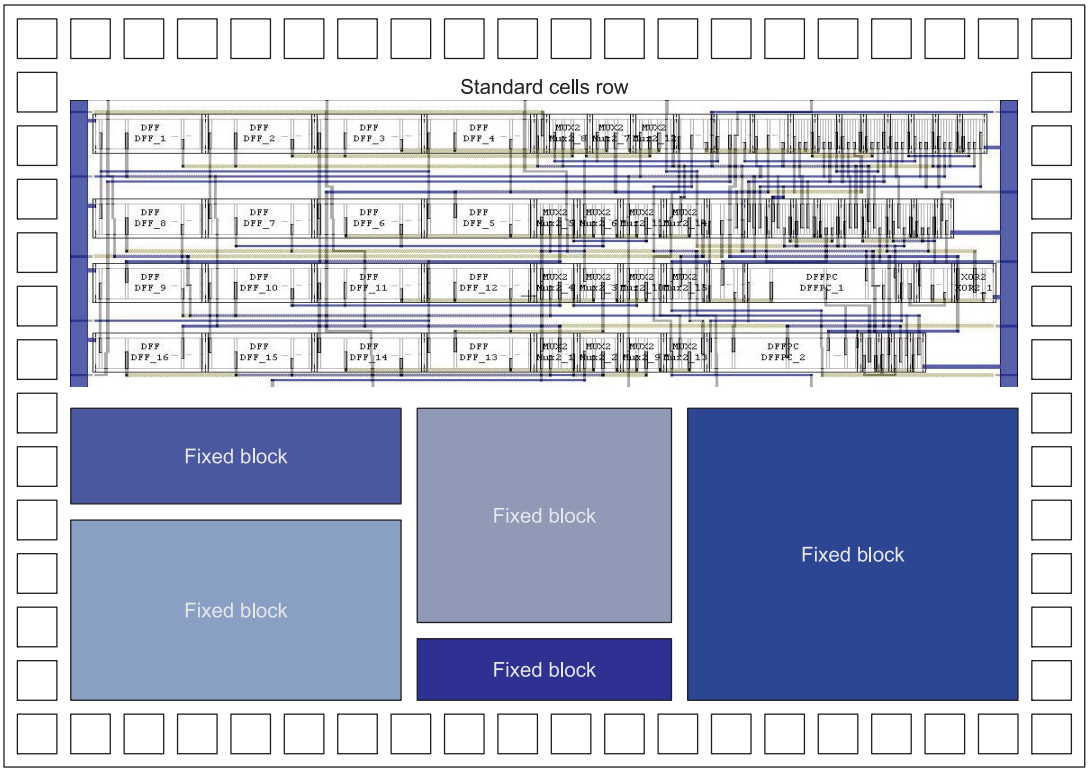


Figure 2.5.1 Standard-based cell ASIC architecture.

All the mask layers of a CBIC are customized allowing placement of Megacells in the same integrated circuit. Megacells are supplied by an ASIC or library vendor complemented with behavioral models and a testing methodology. Layer customization leads to efficient and denser memory designs.

GATE-ARRAY BASED ASIC

In Gate-array based ASICs the transistors are predefined on the silicon. A chip design can be created using custom interconnection patterns on an array of uncommitted logic gates. The pattern of logic cell transistors on the gate are the base array, and the smallest element that is replicated to make the base array is the base cell (primitive cell). In this IC only the top few layers of metal are defined by the designer using custom masks, also the designer can select from a gate-array library of pre-designed and pre-characterized logic cells (usually known as macros). This gate arrays are known as masked gate array (MGA).

The base cell layout is the same for each logic cell, being only the interconnections subject to customization. There are three types of MGAs. *Channeled gate arrays* where only the interconnections are customized, which uses predefined spaces between rows and base cells. The second type is *channelless gate arrays* where only top few mask layers of the interconnections are customized, and finally *structured gate arrays*, have customized interconnections and custom blocks with fixed functions which can be embedded.

Structured ASIC is a bit different from traditional gate-array due to the conceptual nature of the device. Gate arrays contain a "sea-of-gates" across the entire area, which are basically uncommitted. While structured ASICs offer an array of partial or complete macro blocks. The macro blocks are fully optimized, hence it can be said that structured ASICs contain "sea-of-macros" which are embedded within IP blocks such as delay lock loops (DLL), phase lock loops (PLL), double data rate (DDR) RAM and a variety of configurable I/O. In addition structured ASIC support high performance initializable RAM, both single and dual port (DP) SRAM. Structured ASICs are considered the foundation for FPGAs, the next level in customization.

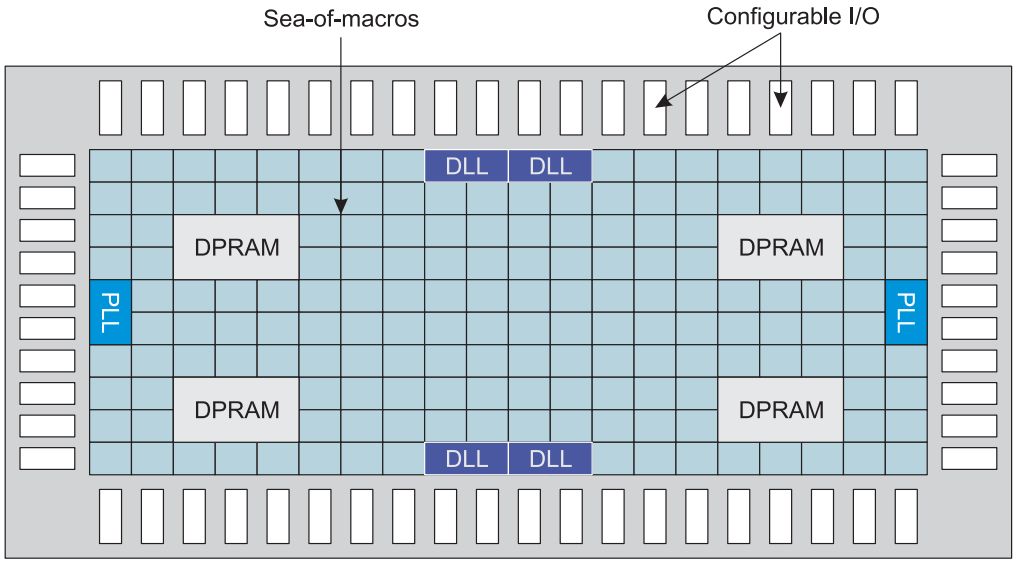


Figure 2.5.2 Structured ASIC architecture.

2.6 FIELD-PROGRAMMABLE GATE ARRAYS

FPGAs present a hierarchical architecture, the building blocks of this architecture are the logic cells or elements (the given name may vary from vendor to vendor), which are grouped in blocks so called programmable logic blocks (PLB). To achieve any particular function the PLB can be interconnected by a programmable connection matrix (PCM), very similar to the ones exhibited by CPLDs[17][18][22][42].

The reason for having this type of logic-block hierarchy is that similar interconnection hierarchy is used. Logic cell inside PLBs have a faster interconnect than interconnected PLBs. The goal is to achieve the optimum tradeoff between easy making cells and blocks connections without incurring in excessive interconnect delays.

The programming of an FPGA can be done by electrically programmable switch matrices (PSM). FPGAs can reach higher levels of integration in the same silicon area than the PLDs discussed before; however this raises the complexity level of the logics implementation and the routing path architecture.

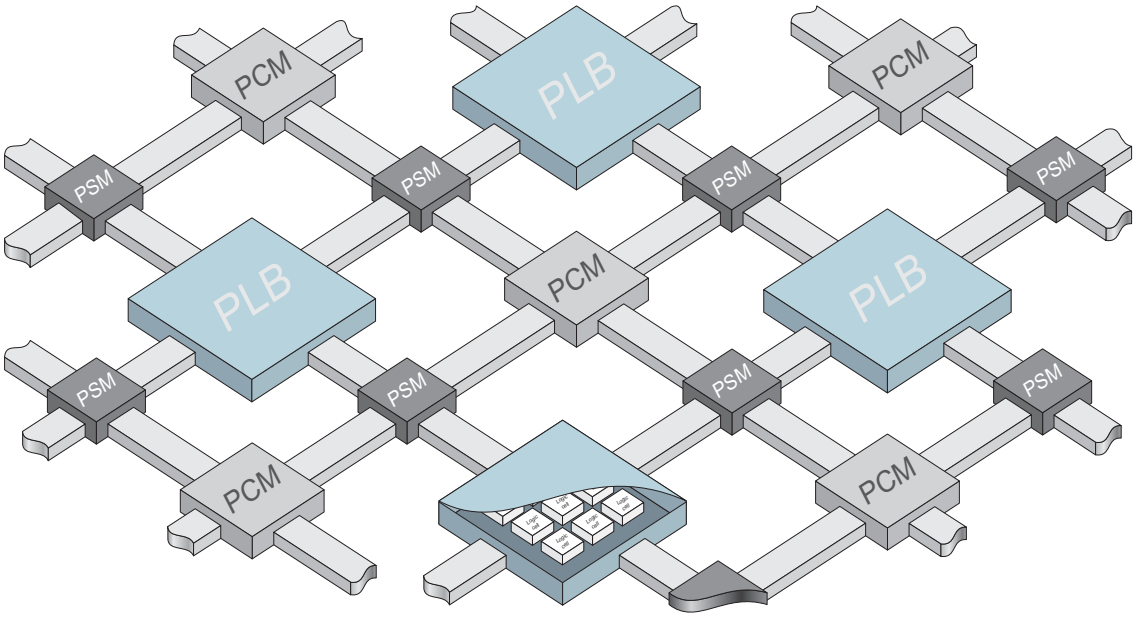


Figure 2.6.1 Conceptual FPGA architecture.

Physically logic cells can be implemented with one or more of the following components: a transistor pair, two-input NANDs or X-ORs, multiplexers (MUX) and look-up tables (LUT). The last two components are commonly used by FPGA manufactures. The idea behind LUTs is fairly simple. A group of input signals (usually four) are use as an index in to a look-up table, usually an array (table) used to replace a runtime computation with an extremely simple look-up operation. The contents of the table are arranged in such a fashion that the specified cell

provides the desired value according the combination set in the cell's entry. The grouping of logic cell depends on the vendor, i.e. Xilinx FPGAs group 2 logic cells into a "slice", then going up in hierarchy, the slice are arrange in groups of 4 resulting in a PLB, or as Xilinx calls them "configurable logic blocks" (CLB). Altera groups eight logic elements in to one logic array block (LAB). Actel names their building blocks as simple logic modules, which are multiplexer-based cells.

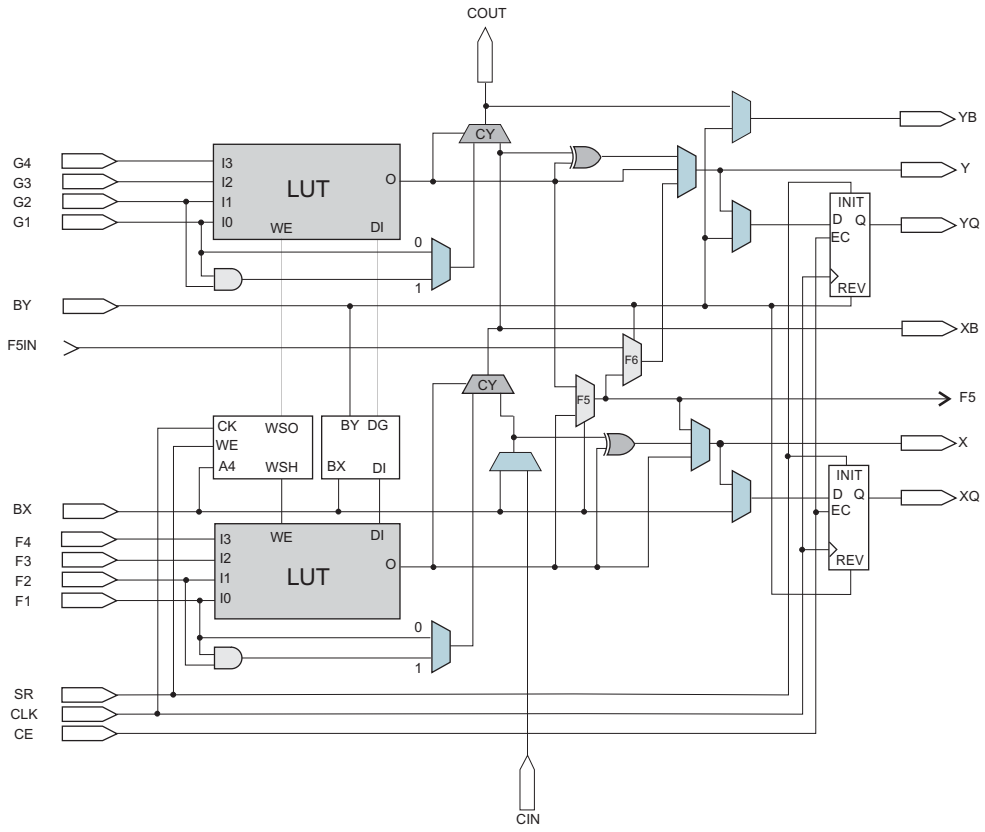


Figure 2.6.2 Xilinx's slice diagram with two logic cells[33].

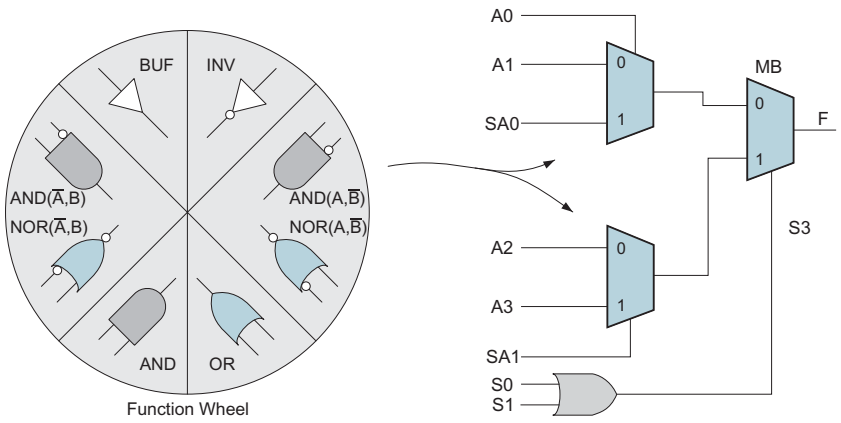


Figure 2.6.3 Actel's logic module[34].

The routing architecture of an FPGA incorporates wire segments of different lengths which can be connected by the PSM. The number of wires used affects the possible achieved density by an FPGA. The distribution of the wire segment lengths has an affect not only in the density but also impacts the FPGA's performance.

There are several technologies widely used by FGPAs manufactures to implement PSMs to retain logic cells functions and values, the most popular ones are: SRAM, Antifuse and EPROM. The properties of PSMs such as size, on-resistance and capacitance may dictate of the tradeoff in FPGA's architecture[18][42].

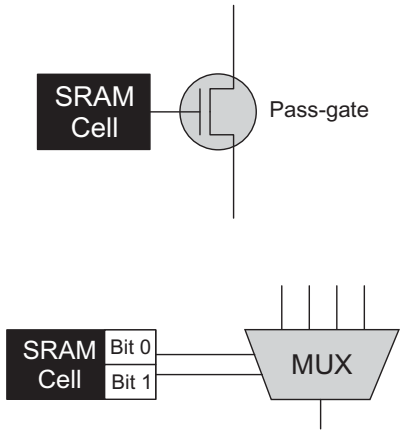


Figure 2.6.4 SRAM-base programming technology.

The SRAM technology uses static random access memory cells to control pass-gates and multiplexers in the logic cell. The functionality is as follows: when a one is stored in the SRAM cell, the pass-gate emulates a closed switch, which can be used to assemble connections between the wire segments. Likewise when a zero is stored the pass-gate is open creating a high resistance level between the wire segments. In the case of the multiplexer, the SRAM cells connected to the selection lines enables one of the inputs to be channeled to the output.

Since the SRAM cells are volatile, the FPGA must be loaded and configured at the chip power-up, requiring a permanent external memory, such as PROM, EPROM or EEPROM. The major disadvantage with this technology is the large area cover in the silicon, since a SRAM cell requires at least five transistors; while the two major advantages of SRAM cells are: fast re-programmability and the technology required is standard integrated circuit-process.

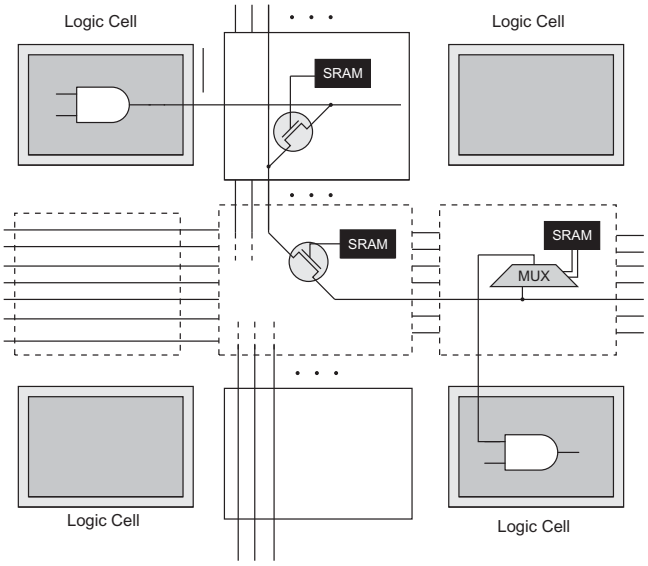


Figure 2.6.5 SRAM-controlled programmable switches.

An antifuse is basically a two terminal device with an unprogrammed state, holding a high resistance between its terminals. When a high voltage is applied across the antifuse terminals, it

will “melt” creating a low resistance connection. This link is permanent and it can only be done once. The major advantage is the small size required by the antifuse, however to provide the necessary current for the “melting” the FPGA will have to allocate large size transistors to handle the programming current.

The EPROM-based technology uses the unprogrammed transistors gate to determine the connection state by means of ultraviolet (UV) light. Basically the transistor has 2 gates: a floating gate and a control gate. The connection between gates can be disabled by injecting a charge on the floating gate using high voltage among the control gate and the transistors drain. The charge can be removed by exposing the floating to UV light reestablishing the connection.

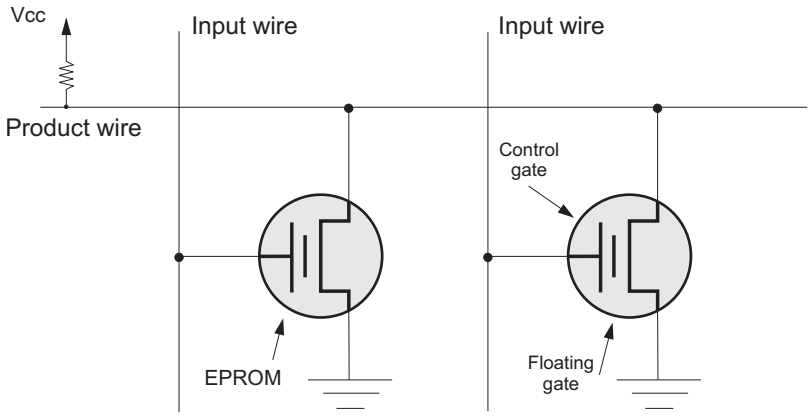


Figure 2.6.6 EPROM-based programming technology

EEPROM-based is similar to the EPROM approach, the only difference is that the connection is reestablished electrically, in circuit, without UV light. The main advantage of these technologies is that the configuration will be stored by the cell eliminating the external memory use. On the other hand the on-resistance of EPROM/EEPROM transistors is high along with the static power consumption.

FPGAs are categorized according to the architectural granularity offered: fine-grained and coarse-grained. In the fine-grained case, each PLB can be used to only implement a very simple function, such as primitive logic gate (AND, OR, XOR, NAND, etc.) or a storage element (Flip-flop, latch, etc.). Besides implementing this logic and irregular structures like state machines, fine-grained architectures offer advantages in compare to traditional logic synthesis, especially when executing functions with massive parallel implementations (systolic algorithms[30], a cornerstone in reconfigurable computing).

The coarse-grained architecture has a relatively large amount of logics compared to fine-grained counterpart per each logic block. It can be the case that a single coarse-grained PLB has several LTUs, multiplexers and flip-flops. An important consideration is that fine-grained require large number of connections compare to the functionality supported by the block; as

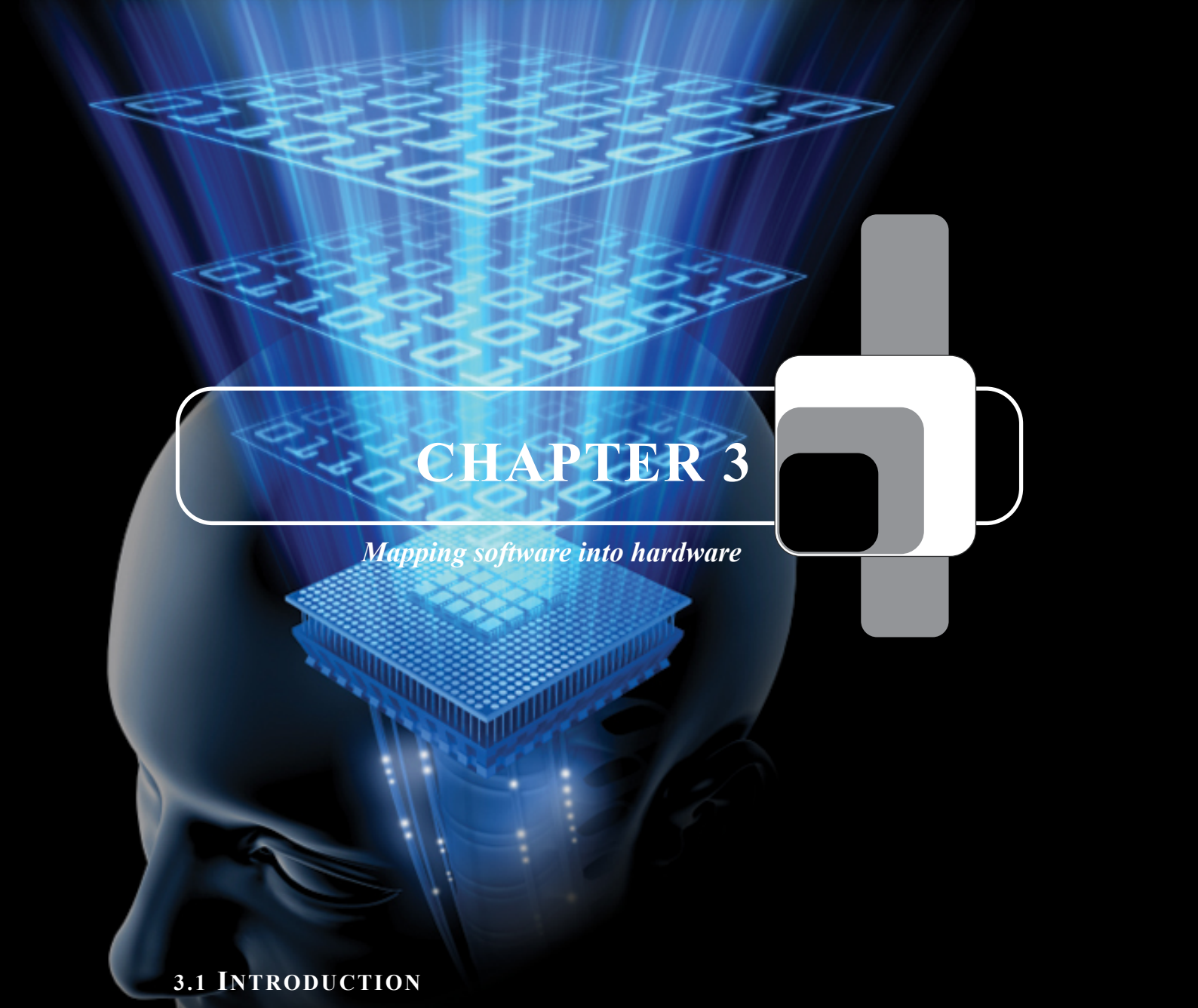
blocks move towards higher granularity the number or connections decreases, this is an important design issue since signal delays throughout the FPGA are directly related to the amount of connections.

FPGAs are the fastest growing market of logic devices. A very popular application is software acceleration though hardware implementation. Large and complicated algorithms can be "translated" in to hardware drastically reducing the computation time. CPU can be implemented in FPGA's by means of synthetization (section 4.1), rapidly replacing MCUs and starting to gain popularity in DSP applications.

2.7 SUMMARY

This chapter has presented the gamma of possibilities regarding programmable logic devices, their basic technology and the integration scale achieved by these devices. PLAs, PALs and GALs are rather short for the project intention. FGPAs are the most promising devices to fit the project requirements. As discussed before, CPLDs are a possibility that cannot be ruled out, especially if the implementation is a small version of the kernel with very few tasks.

ASICs can be another possibility, once the kernel implementation is stable in hardware; FPGA vendors offer the option of creating the ASIC mask from the FPGA implementation. ASICs has the highest performance of all the researched devices, however in the down side ASICs are expensive when adding new functionalities.



CHAPTER 3

Mapping software into hardware

3.1 INTRODUCTION

Digital systems can be described at different levels of abstraction and from different points of view. There are three views from where a system can be described, *Behavioral view* which describes the functionality of a system; *Structural view* which describe the internal implementation of a system; *Physical view* describes the physical characteristics of the system and adds additional information to the structural view. The main purpose of the abstraction levels is to provide a simplified model of the system by only showing the selected features and ignoring the associated details. There are four levels to be considered in digital systems: transistor level, gate level, registers transfer level, and process level[8].

The incremental design process of digital systems requires changes in abstraction and view levels. It is desirable to have a common framework to exchange information along this process. Description languages serve this process. This chapter provides an introduction to the different description languages for hardware design, according to the view and abstraction levels. This chapter also includes an overview of hardware translator tools.

3.2 HARDWARE DESCRIPTION LANGUAGES

Digital systems can be described at different abstraction levels and from different points of view, this is where HDLs come in to the picture, modeling and describing accurately a circuit, whether the circuit is already built or under development, from either structural or behavioral views at the required abstraction level. The main modeling object of HDLs is hardware; hence their semantics and use are very different from the traditional programming languages.

Most traditional general-purpose programming languages are modeled after sequential processes where operations are performed consecutively, one operation at the time. However the characteristics of digital hardware are very different from sequential models. As seen in the past chapter, digital systems (such as logic devices) are built from smaller parts with customized wiring that interconnects their inputs and outputs. Operations throughout these parts are performed concurrently and have associated propagation delays and timing. This uniqueness of digital systems characteristics cannot be capture by traditional programming. HDLs fill this gap left out by traditional programming.

Fundamentally digital circuits are characterized by the following conceptual definitions: entity, connectivity, concurrency and timing. It is considered as entity the basic building block that has been modeled after a real circuit, it is self-contained and independent. Connectivity deals with the wiring connections among entities. Concurrency describes the parallelism of entities operating at the same time. Timing specifies the initiation and completion of all the possible operations providing a schedule and order for concurrent operations.

In the following sections a couple of HDLs are discussed: VHDL and Verilog. These languages encapsulate the previous reviewed concepts, covering description of circuits at structural and behavioral levels. They include constructs that resemble sequential processes in order to ease engineers with experience in traditional programming languages in to HDLs.

3.2.1 VHDL

The VHSIC (very high speed integration circuit) HDL was initially developed by the US Department of Defense as a hardware documentation standard in the 1980's, and later transfer to the IEEE. In 1987 VHDL was ratified as IEEE standard 1076 (VHDL-87). After the initial release various extensions have been developed to facilitate design and modeling requirements, i.e. VHDL analog and mixed signal extensions, VHDL mathematical package, VHDL initiative towards ASIC libraries and so on[8].

VHDL is intended for circuit synthesis as well as circuit simulation. Once a VHDL code has been written there two immediate applications: downloading the code into PLDs (from Altera, Xilinx, Actel, Lattice, Atmel, Cypress, QuickLogic, Aeroflexor, Achronix, etc.) or submitting it to an ASIC manufactures (such as ChipX, AMCC, Freescale, AMI, LSI, etc.).

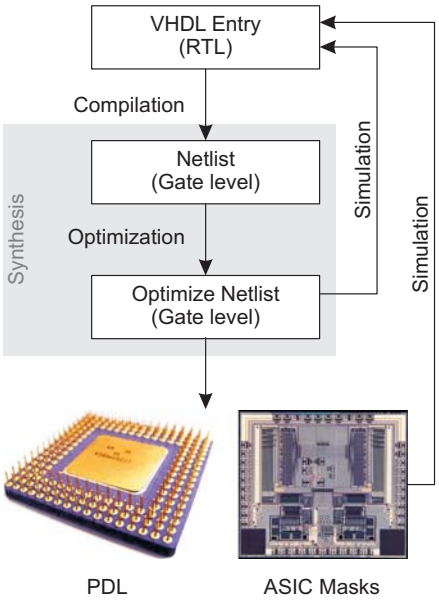


Figure 3.2.1 VHDL design flow.

The VHDL design flow can be summarized as follows: A VHDL code (saved as .vhd and the same name as its entity's name) containing the circuit description at the register transfer level (RTL). The code is compiled, converting the high-level language into a netlist at the gate level. The next step is optimization at gate-level, it could be either for speed or silicon area; at this stage simulation is possible. The final step, a place and route application will generate the physical layout in case of a PLD or the manufacture masks for an ASIC. There are several electronic design automation (EDA) tools available following this design flow that are being offered by the vendor.

A standalone portion of VHDL code is composed of at least three fundamental sections: *Library declarations* which contain the listing of all used libraries, the structure of the libraries resemble the ones in C languages. *Entity* which is responsible for specifying the I/O pins configuration. *Architecture* contains the behavioral description of the circuit.

A full adder is shown as an example. The entity is named full_adder. The architecture is described by the adder function itself. Having a, b and cin (carry input) as entries; s and cout as outputs, the circuit behavior can be described by expressions 3.2.1 and 3.2.2 and implemented using the Algorithm 3.2.1.

$$s = a \oplus b \oplus cin \qquad \text{Expression 3.2.1}$$

$$cout = a \cdot b + a \cdot cin + b \cdot cin \qquad \text{Expression 3.2.2}$$

```
ENTITY full_adder IS
PORT (a, b, cin: IN BIT;
            s, cout: OUT BIT);
END full_adder;
-----------------------------------------
ARCHITECTURE adder_example OF full_adder IS
BEGIN
  s <= a XOR b XOR cin;
  cout <= (a AND b) OR (a AND cin) OR (b AND cin);
END adder_example;
```

Algorithm 3.2.1 VHDL full adder example.

There are several ways to implement the actual physical circuit corresponding to expressions described in the architecture. This depends mostly on the compiler and optimizer being used, furthermore on the target technology. Nevertheless whatever final circuit is inferred from the code, the operation shall be always the same.

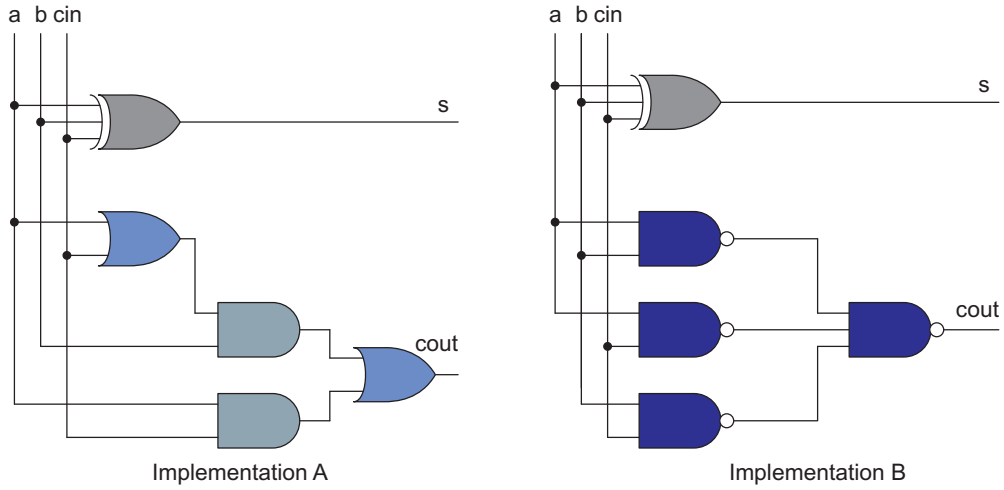


Figure 3.2.2 Possible inferred VHDL circuitry.

3.2.2 VERILOG

This HDL was developed in the early 1980's by Gateway. In 1990 it was purchased by Cadence and made public under the open verilog international (OVI). Verilog became an IEEE standard in 1995. This language is fairly simple to learn, especially if the engineer is familiar with C programming language. In 2005 a superset of verilog languages was created under *SystemVerilog*, combining HDL and hardware verification language (HVL). The HVL part is based on OpenVera, a language created by Synopsis. SystemVerilog includes structures, pointers and recursive subroutines. The full adder example from the last section is revisited; now implemented using Verilog.

```
module full_adder (a, b, cin, s, cout);
  input a, b, cin;
  output s, cout;
  reg s;
  reg cout;

always @(a or b or cin)
  being
    s <= a ^^ b ^^ cin;
    cout <= (a && b) || (a && cin) ||(b && cin);
  end
endmodule
```

Algorithm 3.2.2 Verilog full adder example.

VHDL VS. VERILOG

Hardware structures can be implemented equally efficiently in both VHDL and Verilog. When modeling abstract hardware the choice is not base solely on technical capabilities of the languages, but on personal preference, EDA tool available and commercial, business & marketing issues.

VHDL presents multiple design units that could be compiled separately if desired producing the same result in the end. Verilog is rooted in its native interpretative mode, care must be taken with the compilation order, results can change if done in different order.

Data types can be user define in VHDL, this allow multitude of the language when placing procedure and functions in packages so that they are available to any design unit that wishes to use them, making design reusable. Verilog has very simple data types, easy to use much geared towards modeling hardware. There is no package concept so procedures and functions must be defined in the module. To overcome this reusability obstacle, procedures and functions shall be place in a separate system file making them generally accessible. VHDL is a more robust language than Verilog; however the latter is easier to learn if starting with zero knowledge of either language.

3.3 SYSTEM DESCRIPTION LANGUAGES

The previous languages are bottom-up constructions. They describe a circuit from at register transfer or gate levels. System description languages (SDL) approach circuit description from a top-down point of view. In this case a high level language is used to implement digital circuitry at specification level. In the following subsections two of the most relevant SDLs will be discussed; *SystemC* was developed by Synopsys and made public domain in 1999 throug Open SystemC organization. *SpecC* was developed at the University of California in the center for embedded computer systems in 2000.

3.3.1 SYSTEMC

SystemC is a library of C++ classes, global functions, data types and a simulation kernel that can be used for creating cycle accurate simulators of hardware architecture. This library is used for support system level modeling. By making use of this library, executable specifications or a model can be created in order to simulate, validate and optimize the system[48].

SystemC design flow is an incremental refine process that allows modeling whole systems, including hardware and software parts. SystemC's core language is based on C++. Data types are defined in SystemC which are dedicated to hardware modeling. Core language elements such as modules, processes, events, channels, event driven simulation kernel are also present. Communication mechanisms between concurrent objects are also supported by SystemC.

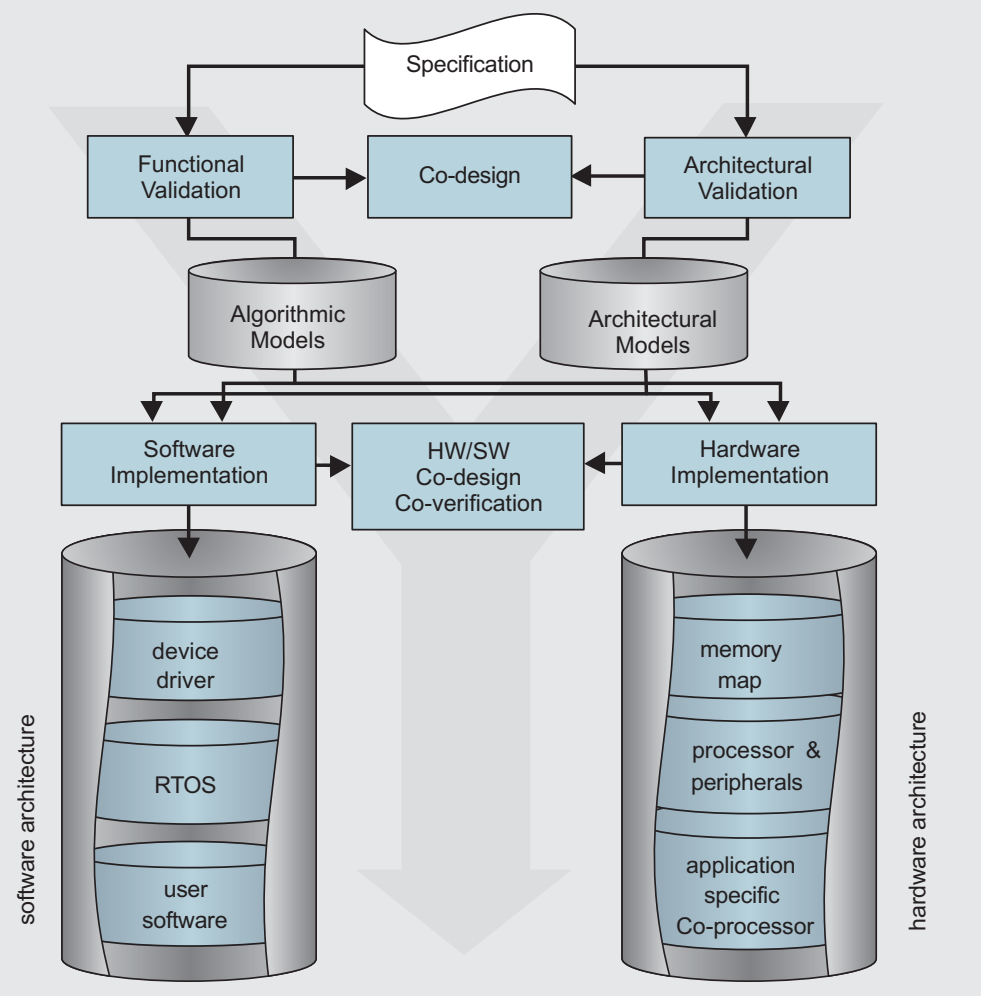


Figure 3.3.1 SystemC design flow[48].

Modules in SystemC have to be an inheritance of the existing class sc_module. The concept of module is pretty much the same as in VHLD or Verilog. Typically a module contains numerous concurrent processes used to implement circuit behavior.

When it comes to processes, SystemC has two existing types: sc_method and sc_thread. To some extent, both processes are similar, since they contain sequential statements and own execution thread, hence they operate concurrently. By definition sc_method cannot be suspended during its execution. On the other hand sc_thread can be suspended and resumed at a later stage.

Channels are the communication medium in SystemC. They can be seen more as generalized form of signals. The language provides wide selection of predefined channels such as: sc_signal, sc_fifo, sc_semaphore and so on. In addition SystemC allows the creation of user defined channels.

3.3.2 SPECC

SpecC is a language which uses most of the C-language semantics. In addition system specification and modeling features have been included. This language is intended for formal notation for specification and design of digital embedded systems, especially for systems on chip (SoC). SpecC is top-build for the ANSI-C programming language concept of embedded systems such as: behavioral and structural hierarchy, concurrency, communication, synchronization, state transition, exception handling, and timing[49].

The SpecC design flow is based on four levels of abstraction: specification, architecture, communication, and implementation. The system synthesis process is divided into two tasks: architecture maps and communication synthesis. The final result of the system design process is the implementation model. At any of the four levels the design models are represented by a corresponding description written in the SpecC language.

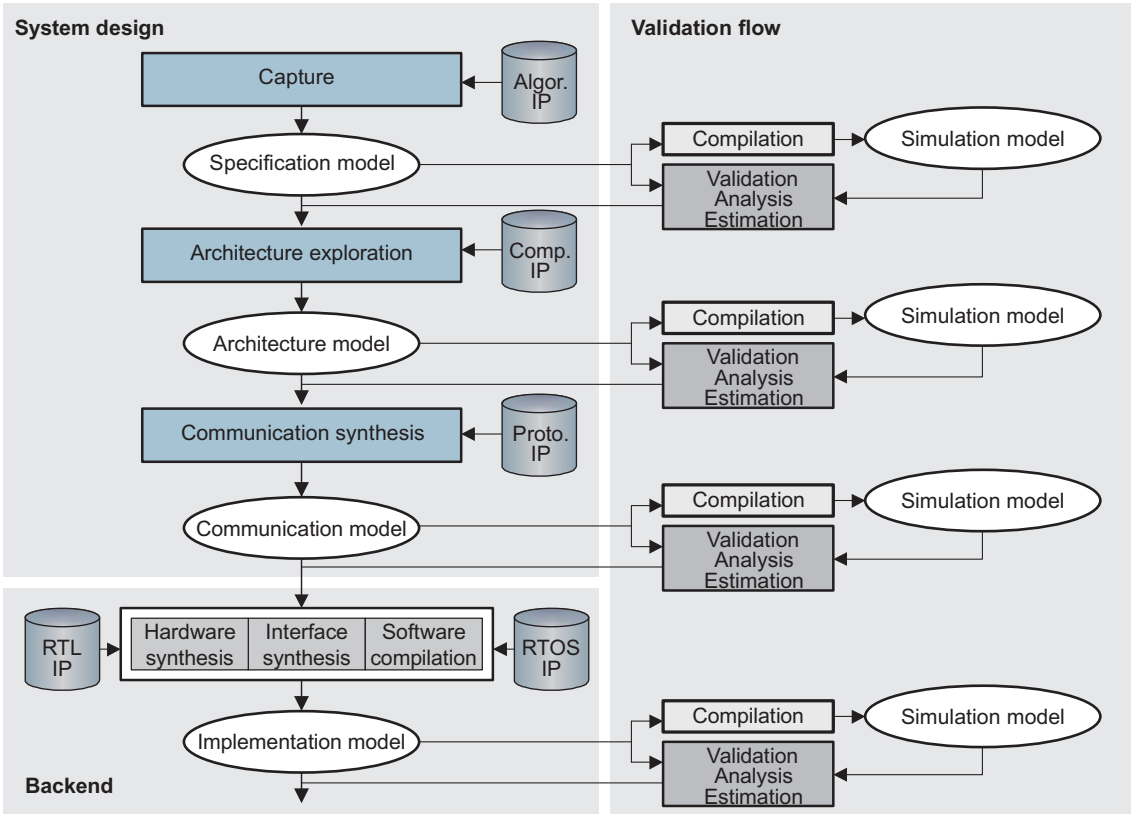


Figure 3.3.2 SpecC design flow[49].

The SpecC design flow approach contains three tasks: system synthesis where circuitry behavior is mapped. In the backend task the components are implemented by synthesizing hardware, software and bus interfaces. Once the model is formalized, automatic refinement, synthesis and verification is applied.

3.4 HLD CONVERTERS

Research is being conducted to convert high-level design languages to HDL such as MatLab Sim2HDL, Altera's DPS Builder and Xilinx's System Generator; however the main focus is around general-purpose programming languages (namely C) to HDL, i.e. ImpulseC, C2Verilog, HandelC, CatapultC, DIME C, Cynthesizer, FpgaC, and MitronC. This research is extended to other programming languages, i.e. Tools like JHDL and Jove which convert Java to HDL, or MyHDL which translates Python to HDL.

In the following subsection a couple of these approaches will be discussed. ImpulseC from Accelerated technologies, basically targeting FPGA applications with a hardware/software co-develop approach. Sim2HDL developed at the University of Brasov, which translates MatLab Simulink diagrams into HLD.

3.4.1 IMPULSEC

ImpulseC extends standard ANSI C making use of a predefined C-compatible library which supports communicating process in parallel programming models. Conceptually, the programming model is similar to a dataflow. The programming model supports a wide range of applications and parallel process topologies. In ImpulseC, the programming model makes use of buffered data streams as the primary communication method among independent synchronizable processes. Data buffers are FIFO implementations, making possible to write parallel applications at higher abstraction levels, without the use of the clock cycle-by-cycle synchronization that otherwise would be required[50].

The ImpulseC compiler automatically generates parallel logic by scheduling C operators, including entire code blocks, and is capable of generating loop pipelines to increase parallelism and performance. The ImpulseC platform support libraries for specific logic devices targets such as Xilinx and Altera. ImpulseC can also be used to generate hardware modules that do not interface to software processes; there is no need to include an embedded processor to make use of ImpulseC.

3.4.2 SIM2HDL

Sim2HDL is an EDA tool that converts a non-specific MatLab Simulink model into HDL. This uses a limited set of Simulink blocks from the original library. Sim2HDL supports Altera and Xilinx devices as well as various ASIC technologies. Sim2HDL offers support for MatLab variables from the workspace; they can be introduced directly into the Simulink blocks[31].

The design flow starts with two files: the Simulink file that contains the logic device algorithm description, and a library file with the corresponding HDL behavioral description. There are four stages which the files have to go through in order to get the synthesis and simulation of the model: parser, database, netlist generator, and message handler & HDL writer.

3.5 SUMMARY

This chapter has presented various design implementation options for digital circuitry. Hardware description languages are the first design approach. The HDLs presented are compatible with all the programmable devices manufactures. These HDLs are not the only ones, i.e. Altera has their own HLD, "Altera HDL" (AHDL). However these manufacture specific HLDs are only for their programmable devices, they cannot be used for other manufactures. HDLs design digital circuitry at low abstraction levels (transistor, gate and register transfer levels) and at physical and behavioral views.

The second set of description languages presented, can design digital circuitry at specification and process levels, at structural and behavioral view. This approach is called top down design methodologies, where as HDLs are bottom up methodologies. The last option presented, are translators to HDL, where a high language is taken and transform by software tools, parsers or libraries into HDL.

From the research presented in this chapter, the most appealing languages to implement any of the HARTEXμ functions are HDLs (VHDL or verilog). In the case of kernel primitives and services, SystemC and ImpulseC are more appropriated, since they offer a higher level of abstraction.

CHAPTER 4

FPGA + CPU → The soft-processor

4.1 INTRODUCTION

Today's FPGAs are available with the equivalent of over a million logic gates. With these resources is possible to implement complex digital systems, including one or more microprocessors inside a single FPGA.

Several programmable logic companies have introduced devices that allow the user to combine general purpose processing with programmable logic on the same chip. This is considered at the time being as the state-of-the-art. Some of the devices include a dedicated hardcore-processor with FPGA-based logic, others rely on a soft-core processor that can be implemented (synthesized) inside an FPGA with the possibility of implementing user define logic.

This chapter provides an introduction and features review to various soft-core processors: Cortex-M1 by ARM, Nios II by Altera, Leon2 by Gaisler Research, MicroBlaze by Xilinx and OpenRISC 1200 by the OpenCores foundations. Technical characteristics are discussed in detail, such as instruction set architecture, cache architecture, communication bus, memory management, and extended features for co-processing units. The end of the chapter is focused on Xilinx MicroBlaze and its main advantages offered to this project.

4.2 BASIC ARCHITECTURE

Is not hard to imagine customizable devices with very specific peripherals, memory interfaces and processing functions; with the current technology these devices are not out of reach. As discussed in section 2.5, ASIC designers have been trying to do this for the past decades, however economically is not the best option. Until the late 1990's customizable embedded processor started to be feasible with FPGA devices which had enough on-chip memory and raw performance.

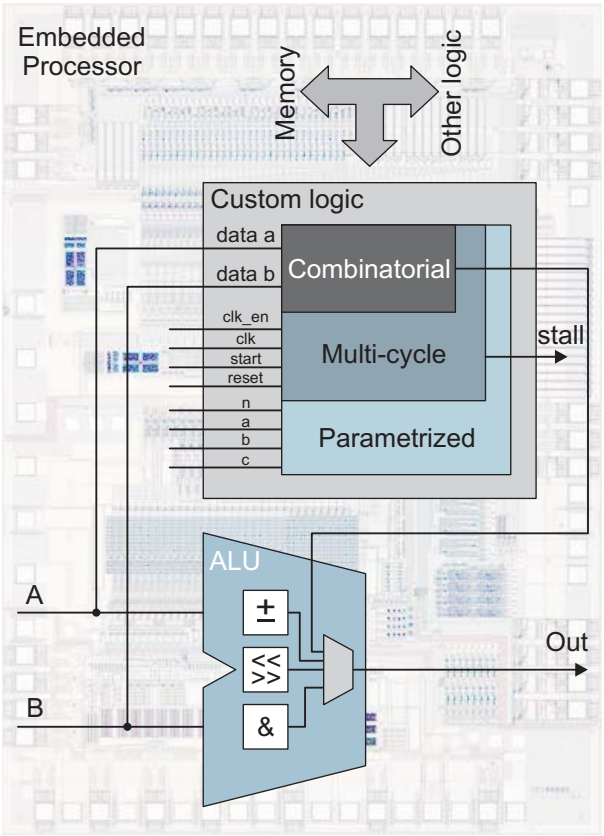


Figure 4.2.1 CPU extended instruction set, FPGA-based.

There are various CPU architectures that can be implemented into an FPGA. Intuitively the first approach is to extend the instruction set of the CPU, with the possibility of customizing logic functions which is offered by FPGAs. This can be achieved by specific algorithms implemented in hardware offered by several processor IP vendors.

Using the processor's normal arithmetic-logic unit (ALU), data can be fed to a custom block that essentially becomes one with the ALU. In some cases, custom instructions can support multi-cycle operation and access other system resources.

Custom instruction can provide significant performance benefits over running the same algorithm using the normal ALU resources.

Other FPGA-based architectural approaches are multiple processors where different CPUs in a system are embedded into one FGPA, this reduces the size of the board and requires less signal routing. It could be implemented as multi-channel or serially linked processors. The former can be used to meet system troughput by using multiple processor, each dedicate to handle a portion of the overall channel troughput. The later case allow the designers to treat each CPU as a stage in larger processing pipeline, so the different CPUs are taking one piece of the overall processing task. Another approach is DSP chips and discrete processors connected to FPGAs getting the benefit of hardware acceleration, peripheral expansion and interface bridging. This approach is used to extend the product life of a system by updating the FPGA to new demands.

4.3 SYNTHESIZABLE CPU'S

Synthesizable CPUs are soft-core processors (or softprocessors, σprocessor) that can be implemented entirely in the logic primitives of an FPGA. In the subsequent subsections four of the most noticeable σprocessor are presented (Cortex-M1, NiosII, Leon2 and OpenRISC 1200) followed by a section dedicated to discuss in greater detail one of the most mature σprocessor available in the market (MicroBlaze).

4.3.1 CORTEX-M1

The Cortex-M1 processor is the first ARM processor synthesized specifically for implementation in FPGAs[32][34]. This σprocessor targets all the major FPGA devices, such as Actel Fusion, Altera Cyclone II & III, Xilinx Spartan-3 and Xilinx Virtex-2 trough Virtex-5. The σprocessor synthesis is also supported by third-party vendor tools such as Mentor Precision by Mentor Graphics, Actel Libero, Altera Quartus II, and Xilinx ISE.

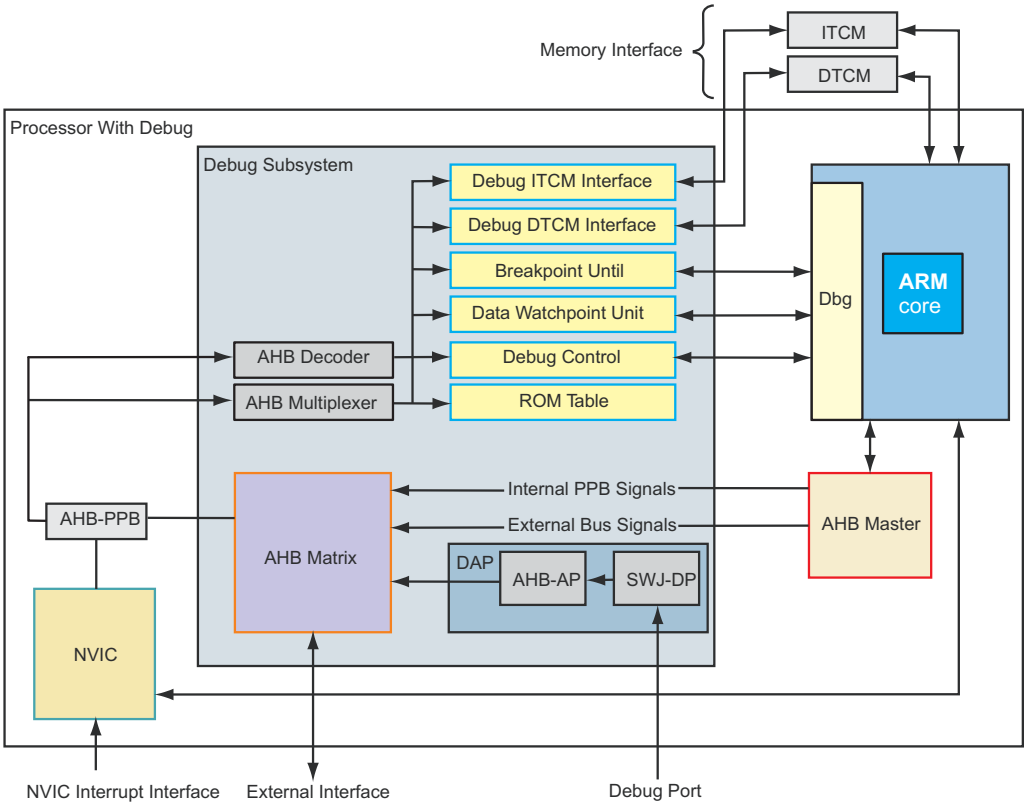


Figure 4.3.1 Cortex-M1 system architecture[34].

The Cortex-M1 σprocessor is a streamlined three-stage 32-bit processor, von Neumann based architecture. It can be configured for a big-endian or little-endian byte ordering. The instruction set architecture (ISA) is based on the Thumb-2 instruction set. This set introduces 32-bit instructions that are intermixed with 16-bit instructions. Thumb-2 is compatible with the rest of

ISAs from ARM, however there is one important difference; Thumb-2 introduces a new conditional execution instruction, IT, that is a logical if-then-else function. In addition several new 32-bit and 16-bit instructions are introduced. The main enhancements are support for exception handling in Thumb state, access to coprocessors, DSP and media instructions. In the 16-bit area Thumb-2 has added compare and branch on zero (CBZ), and compare and branch on non zero (CBNZ), they replace the two-instruction sequence.

Cortex-M1 interface consists of the advance microcontroller bus architecture (AMBA) together with the advance high performance bus-lite (AMBA-AHB-lite) protocol, advance peripheral bus (APB) interface, and AHB to APB bridge. This σprocessor accesses memory through a dedicated core memory interface which is composed by instructions to access the tightly couple memory interface (TCM). In addition Cortex-M1 can interact with several IP-cores such as: CoreGPIO, CorePCIF, CoreI2C, CorePWM, CoreTimer, CoreAI, CoreWatchdog, CoreInterrupt, Core429, CoreMemCtrl among others.

4.3.2 NIOS II

Nios II is a 32-bit synthesizable σprocessor that can be assigned in either little-endian or big-endian word format. Nios II has been specially design to target Altera FPGA devices, such as Stratix and Cyclone. Nios II σprocessor is defined in AHDL, which can be implemented using Quartus II CAD system.

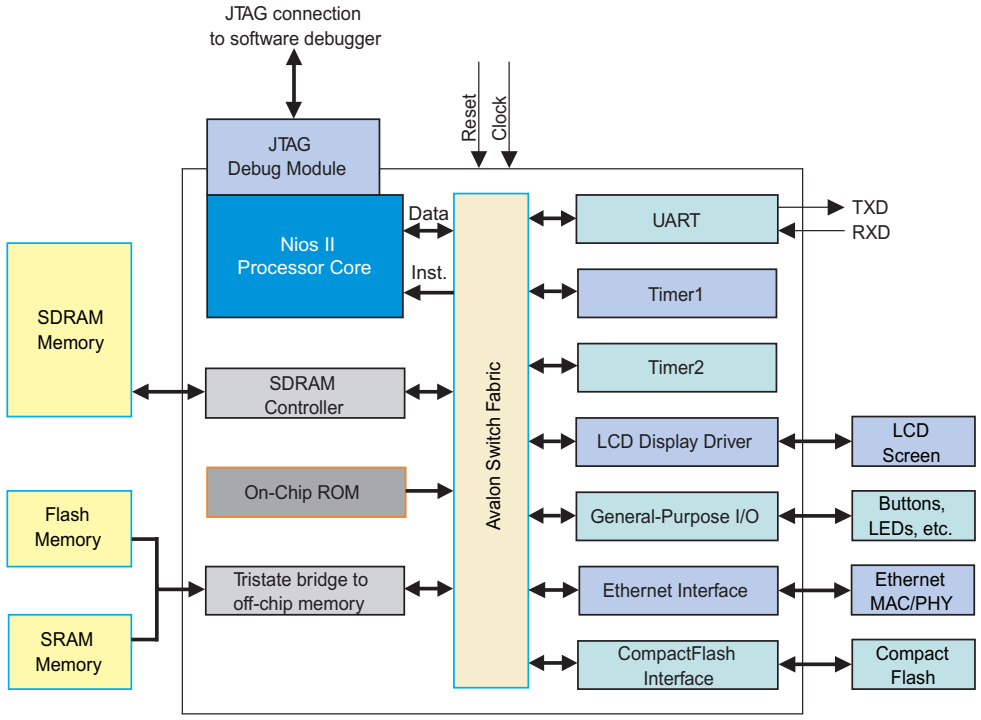


Figure 4.3.2 Nios II system architecture[35].

The Nios II σprocessor can be streamlined in one, five or six stages, according to the version of the core, which are Harvard based architectures. The σprocessor presents a RISC ISA. The instruction set has been designed to easily incorporate custom logic into the ALU.

The custom instructions behave similar to the native instruction of the σprocessor. Both types of instructions can accept values from up to two 32-bit source registers and write back a result to 32-bit destination register. Custom instructions can fine-tune hardware to meet specific performance requirements. These instructions can be handled in C language as macros.

Nios II uses the Avalon switch fabric (ASF) as the interface bus to the embedded peripherals. The main feature of the ASF is the use of a slave-side arbitration scheme which allows the simultaneous operation of multiple masters over the same bus. ASF offers integration flexibility for custom design peripherals. ASF also supports multiprocessor environments by coordinating the access to shared resources via the mutex core.

MMU is not present in Nios II; however memory and I/O access are managed by the Avalon memory mapped interface (Avalon-MM). The instruction and data buses are implemented as 32-bit Avalon-MM master pipelined ports. Both ports are connected to the peripherals via interconnect fabric.

Nios II memory driver supports SDRAM, CompactFlash and On-chip FIFO. In addition to these units, the following can be added to the σprocessor: EPCS device controller core, Scatter-Gather DMA controller core, DMA controller core, JTAG UART core, UART core, SPI core, Avalon streaming peripherals cores among others.

4.3.3 LEON2

Leon2 is a 32-bit RISC SPRAC V8 compliant architecture. This synthesizable σprocessor was developed by the European space center (ESA), currently maintained by Gaisler Research. Originally the σprocessor was developed as fault-tolerant for space applications, however Leon2 is available for free as a non fault-tolerant under the GNU LGPL license at Gaisler Research website[27][37].

The ISA utilizes the SPRAC V8 instruction set, which design goal is to make software optimization easily in the compiler and to simplify hardware pipeline implementations. Leon2 ISA has three different instruction formats and three addressing modes: immediate, displacement and indexed. It also includes instructions for multiply-accumulate (MAC) and divide operations.

The ISA includes instructions for two coprocessors, one FPU and one custom user defined coprocessor. The FPU can be from one of the following: GRFPU provided by Gaisler Research, Meiko FPU from Sun Microsystems, and LTH FPU from Lunds Tekniska Högskola.

The integer unit implements the SPRAC V8 ISA as a single issue five-stage pipeline. The cache system is a Harvard architecture with 1x64 Kbytes per way for the instruction and the data cache.

The memory management unit can be enabled providing support for memory protection mechanisms required fro advanced operating systems. The interface between the integer unit and memory/peripherals is via the AMBA-AHB and AMBA-APB.

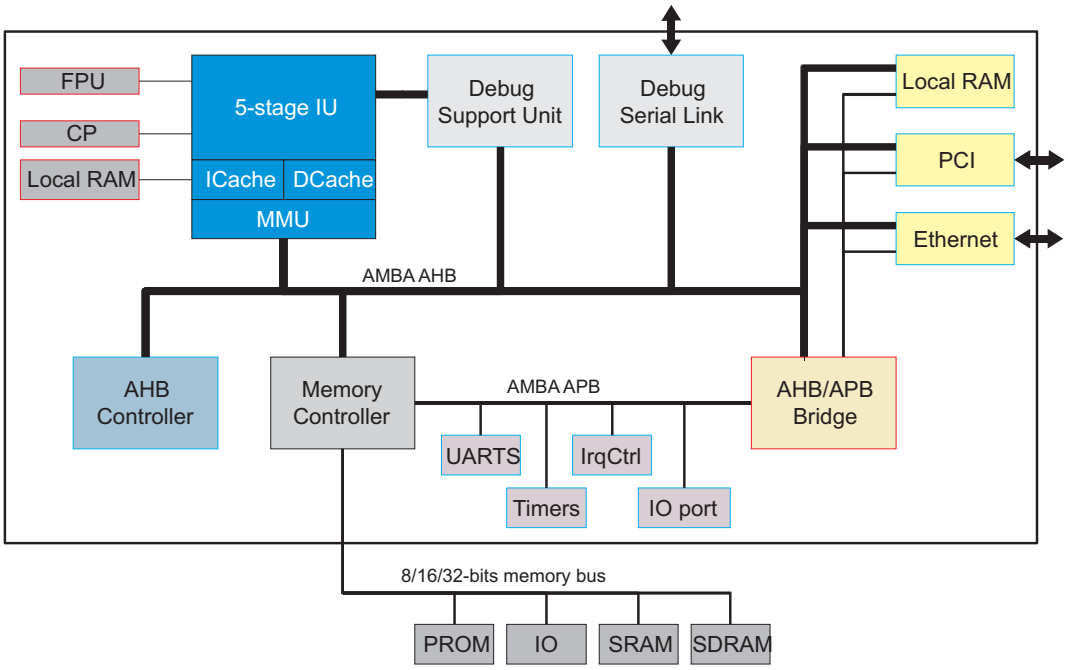


Figure 4.3.3 Leon2 architecture.

Leon2 memory controller supports several types, including: PROM, SRAM and SDRAM. In addition to the units mentioned before, other units can be connected: Debug support unit (DSU), PCI interface, Ethernet MAC and on-chip RAM.

4.3.4 OpenRISC 1200

OpenRISC 1200 is a synthesizable σprocessor developed and managed by a team of developers at OpenCores. This processor is a 32-bit RISC implementation. It uses big-endian byte ordering. This σprocessor is intended for embedded, portable and network applications. OpenRISC 1200 is an open source IP-core available at the OpenCore website, modeled in Verilog and licensed under the GNU LPGL license[27].

The ISA is an implementation of the ORBIS32 instruction set. It has five instruction format and three addressing modes: immediate, displacement and pc-relative. The ISA includes instructions for multiplication and division; however they are implemented in hardware.

The integer unit is a single five-stage pipeline implementation of the ORBIS32 ISA. The cache architecture is Harvard where the instruction cache can vary in size from 512 bytes to 8 Kbytes, and the data cache from 1 to 8 Kbytes. Both caches are direct-mapped. Memory management unit is implemented and can be enabled to provide support for memory protection.

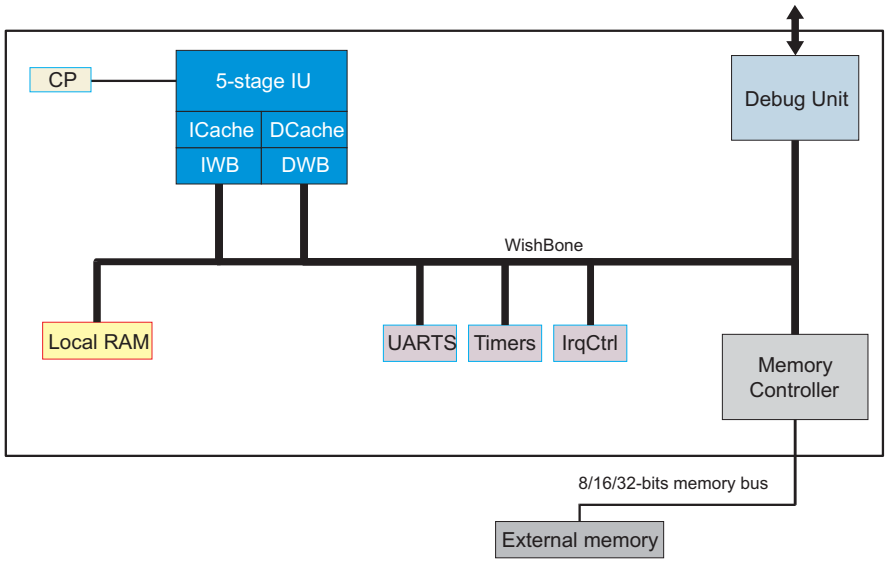


Figure 4.3.4 OpenRISC 1200 architecture.

The interface with the memory and peripherals is achieved via two Wishbone compliant 32-bit bus interfaces. This interface supports point-to-point, shared bus, and crossbar switch and dataflow interconnections. The memory controller supports among other, SDRAM, SSRAM, FLASH and SRAM. In addition OpenRISC 1200 can interact with the following IP-cores: FFT cores, Ethernet MAC, I2C controller core and cryptographic cores.

4.4 MICROBLAZE

MicroBlaze is a 32-bit RISC σprocessor from Xilinx, optimized for use in any Xilinx FPGA devices such as Spartan-3 and any of the Virtex series. Microblaze is a highly configurable σprocessor, many aspects can be fine-tuned: cache size, embedded peripherals, and bus interfaces can be customized. In addition certain instructions can be selectively added or removed, e.g. multiplication, division, and floating-point arithmetic.

MicroBlaze is distributed with the Xilinx embedded development kit (EDK) as a parameterizable netlist without any royalties. The VHDL source code can be acquired from Xilinx at a higher cost (The Verilog version of MicroBlaze, AEMB is being developed by Æste Engineering as open source project). The latest version of MicroBlaze is 7.0 under Xilinx EDK version 10.1.

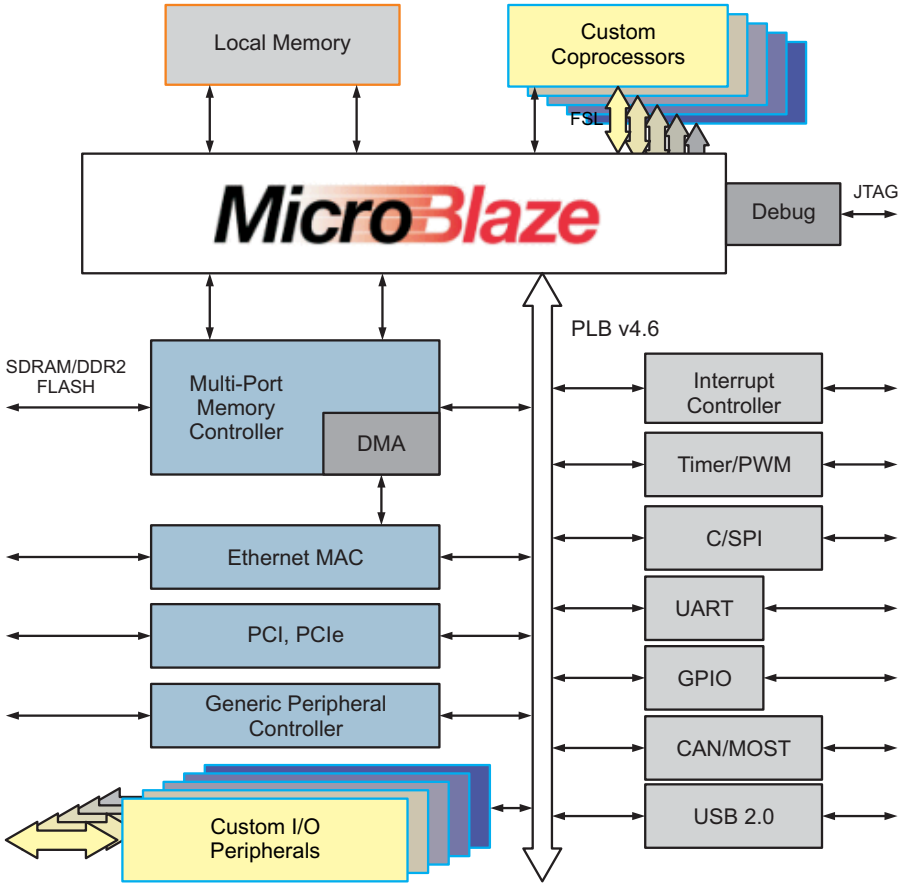


Figure 4.4.1 MicroBlaze system.

4.4.1 ARCHITECTURE

The MicroBlaze architecture is highly configurable, allowing the selection of specific features required by a particular design. However the following features are architecture-fixed: thirty-two 32-bit general purpose registers, 32-bit instruction word with three operands and two addressing modes, 32-bit address bus and single issue pipeline[33].

Instruction execution is pipelined. When area optimization is enabled, the pipeline is divided into three stages to minimize hardware cost; when disabled, the pipeline is divided into five stages to maximize performance. MicroBlaze uses big-endian bit-reversed format to represent data. Word, half word and byte types are supported by the hardware.

MicroBlaze has its own ISA specially designed for the σprocessor. The instruction set is orthogonal since any instruction can use data of any type via any addressing mode. There are two types of instructions: Type A, which have up to two source register operands and one destination register operand; and Type B, which have one sources register and 16-bit immediate operand and one a single destination register operand. There are two addressing modes supported by MicroBlaze: immediate and displacement.

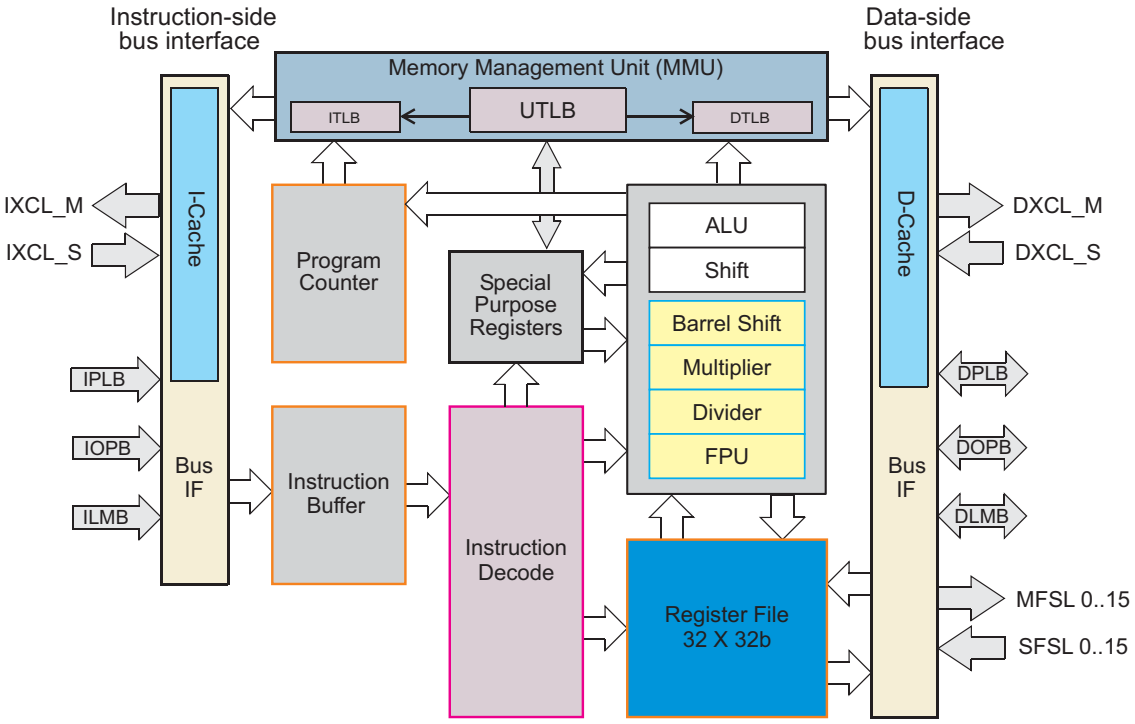


Figure 4.4.2 MicroBlaze core architecture[33].

Harvard memory architecture is implemented in the MicroBlaze σprocessor. Each address space has a 32-bit range. The instruction and data memory ranges can be made to overlap by mapping them to the same physical memory.

General purpose registers

MicroBlaze has 32 general purpose registers divided into three categories: volatile, non-volatile and dedicated.

- Volatile register are temporary registers and do not retain their values across function calls. These registers go from R3 to R12. R3 and R4 are used to return values to the caller function. R5 through R12 are used to pass parameters.
- Non-volatile registers keep their values across function calls. These registers go from R19 through R31.
- Dedicated registers are used to store return addresses from interrupts, sub-routines, traps and executions. These register go from R14 through R17. R0 is always zero and R1 is used to store the stack pointer. These register should not be used.

Special purpose registers

There are five special purpose registers: machine status register, program counter, exception address register, exception status register and floating point status register:

- *Machine status registers (MSR)*. The MSR holds control and status bits for the processor. The MSR contains the enable and disabling of interrupts, exceptions and data/instruction cache. It also contains bits for error such as divisions by zero and FSL.
- *Program counter (PC).* The PC is a read only register which contains the address of the executing instruction.
- *Exceptions address register (EAR).* The EAR stores the full address of the exception source.
- *Exception status register (ESR).* The ESR contains the exceptions status bits for the processor.
- *Floating point status register (FSR).* The FSR contains the status bits for the floating point unit.

4.4.2 SYSTEM INTERFACE

The MicroBlaze core is organized as a Harvard architecture with the following memory interfaces: Local memory bus (LMB), IBM processor local bus (PLB) or On-chip peripheral bus (OPB), and Xilinx cache link (XCL). The XCL interface is intended for use with specialized external memory controllers. The PLB and OPB interfaces provide a connection to both on-chip and off-chip peripherals and memory. The LMB provides single-cycle access to on-chip peripheral bus.

MicroBlaze also support up to 16 fast simplex link (FSL) ports, each with one master and one slave FSL interface. In general the bus interfaces can be configured as follow:

- A 32-bit version of the PLB V6.6 interface.
- A 32-bit version of the OPB V2.0 bus interface.
- LMB provides a simple synchronous protocol for efficient block RAM transfers.
- FSL provides a fast non-arbitrated streaming communication mechanism.
- XCL provides a fast slave-side arbitrated streaming interface between caches and external memory controllers.
- Debug interface for use with the microprocessor debug module (MDM) core.
- Trace interface for performance analysis.

MicroBlaze system interface has been developed to support a high degree of user configurability, allowing specific cost-performance tailoring to meet any requirements. These configurations can be done via parameters that enable size or processor features, e.g. the instruction cache is enabled by setting the C_USE_ICACHE parameter.

4.4.3 COPROCESSOR AND CUSTOM LOGIC

One advantage of σprocessors is their customization flexibility by using only the processor features required for a specific application. Another advantage is the integration of intellectual

property (IP) cores resulting in a drastic acceleration in the software execution-time due to algorithms being parallel-hardware executed.

MicroBlaze offers two ways to integrate a customized IP core: OPB connection which is a standard on-chip bus; and FLS channel connection which is a dedicated bus for time-critical custom applications.

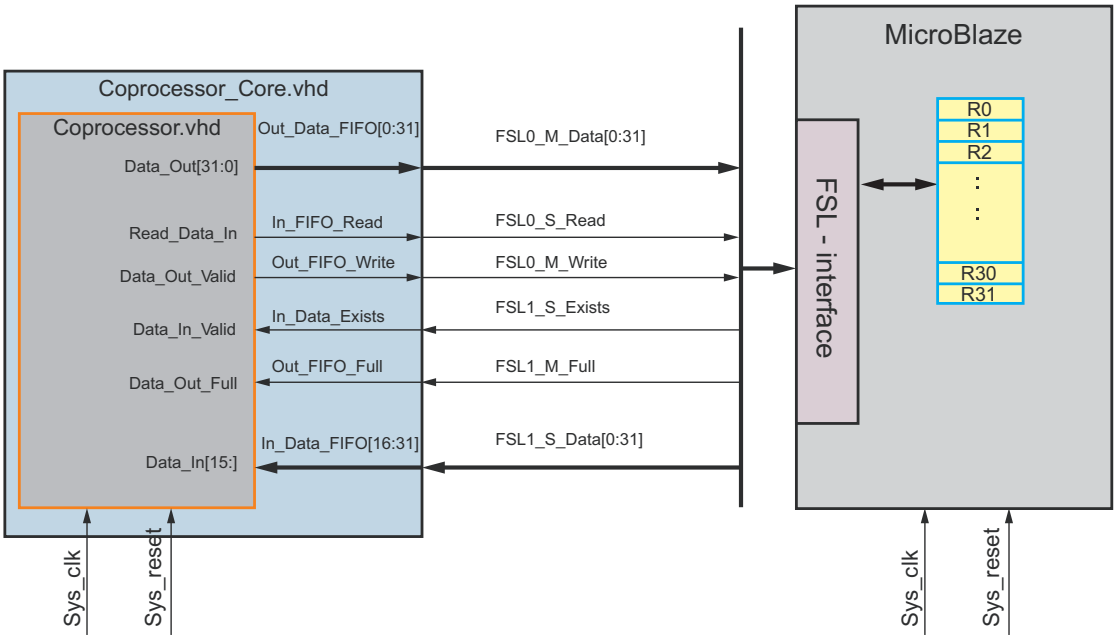


Figure 4.4.3 MicroBlaze coprocessor implementation[33].

FSL channels are dedicated unidirectional point-to-point data streaming interfaces. These interfaces are 32-bit wide and can be use to transmit or receive either control or data words. A separate bit indicates whether the transmitted or received data is either control or data words. FSL can achieve speeds up to 300 MB/sec which makes it perfect for σprocessor-to-σprocessor streaming I/O communications. The main FSL characteristics are[56]:

- Unidirectional point-to-point communication.
- Unshared non-arbitrated communication mechanism.
- Control and data communication support.
- FIFO-based communication.
- Configurable data size.
- 600MHz standalone operation.

FSL peripherals may be created as master or a slave to the FSL bus. All peripherals that act as a master to the bus should create a bus interface of the type “master” for the bus standard FSL in the microprocessor peripheral description (MPD) file. A peripheral connected to the slave ports

of the bus reads and pops data and control signals from the FSL; they should create a bus interface of the type "slave" for the bus standard FSL in the MDP file. The put and get instructions of MicroBlaze can be used to transfer the contents of a MicroBlaze register onto the FSL bus and vice-versa. The FSL bus configuration can be used in conjunction with any of the other bus configurations.

4.5 SUMMARY

This chapter has presented five of the most important soft-core processors available in the market; they have been selected because of the wide spread used, however they are not the only ones, e.g. CoreABC by Actel, Leon3 by Gaisler Research, LatticeMico32 by Lattice Semiconductors, Core8051 by Intel, among others.

The integration of a customized IP core within the execution unit (ALU) is very restrictive in the presented σprocessors. One of the biggest restrictions is due to the nature of RISC processors architecture itself. Modern RISC architectures have two-input and a one-output execution unit. Custom packet processing applications require different dynamically changeable inputs (mask bits, registers) and outputs.

Another bottleneck is the customized instruction itself. If the critical path of the whole system is through the user IP, the whole σprocessor will decrease in performance because the user IP is include within the σprocessor itself. If the architecture doesn't allow the designer to stall the pipeline, the σprocessor can't run at the higher frequency than the critical path would allow.

Software integration of customized instruction can't be handled directly form the complier, thus the user has to use inline pragmas assembly to work with them, producing a C application code which is neither very clean nor portable.

MicroBlaze offers the fully dedicated FSL interface solution, which fits better this project. It is important to make the following remark; soft cores have a lower performance and clock frequency than their hard cores counterpart. This is in part due to the placing and routing that has to be done in a FPGA to synthesize the soft core. On the other hand, custom logics accelerate the system performance, leaning the balance in favor of soft cores.

CHAPTER 5

Embedded development kits

5.1 INTRODUCTION

There are several embedded platforms were MicroBlaze can be crafted to a design with the perfect combination of feature set, performance and cost. Xilinx offers two families of platform boards: Spartan and Virtex. The main difference between these families is the Spartan is only a FPGA where MicroBlaze can be synthesized whereas Virtex also include a “hard” processor, namely IBM PowerPC.

This chapter presents and discusses two boards from the Spartan 3 family: The Spartan-3E Starter kit and the Spartan-3A DSP S3D1800A MicroBlaze Edition. The Spartan boards are sufficient for this project. They can easily allocate a MicroBlaze core leaving plenty space for custom logic implementations.

The Spartan-3E starter kit gathers all the minimum requirements to implement and run a MicroBlaze core, making it ideal for beginners to get familiarized with MicroBlaze. The Spatan-3A DSP S3D1800A kit presents a robust set of features that explores the “flexibility” of the MicroBlaze soft processors.

5.2 SPARTAN 3

The Spartan 3 family of FPGAs from Xilinx is specifically designed to meet high volume needs and cost-effectiveness. The density presented by this family ranges from 50,000 to five million system gates. Spartan 3 builds on the legacy of the earlier Spartan-IIE family by increasing the amount of logic resources, the capacity of internal RAM, the total number of I/Os, and the overall level of performance as well as by improving clock management functions. Spartan 3 family presents the following features:

- SelectIO interface signaling, up to 712 I/O pins with transfer data rates at 622 Mb/s. Support of DDR, DDR2 and SDRAM up to 33 Mbps.
- Abundant logic cells with shift register capability, wide and fast multiplexers and dedicated 18x18 multipliers.
- SelectRAM hierarchical memory.
- Digital clock manager with eight global clock lines and abundant routing.

5.2.1 ARCHITECTURE

The Spartan 3 family architecture consists of five fundamental programmable elements: *Configurable logic blocks* (CLBs) which contain RAM-based LUTs to implement logic and storage elements that can be used as flip-flops or latches. CLBs can be programmed to perform a wide variety of logical functions as well as to store data[33].

The second elements are *Input/Output blocks* (IOBs) which control the flow of the date between the I/O pins and the internal logic of the device. Each IOB supports bidirectional data flow in addition to three-stage operations. DDR registers and twenty-six different signal standards are included. The digitally controlled impedance (DCI) feature provides automatic on-chip terminations which simplify board designs.

The next elements are *Block RAM* providing data storage in the form of 18-Kbit dual-port blocks. *Multiplier blocks* accept two 18-bit binary numbers as inputs and calculate the product. The last fundamental element are the *digital clock manager* (DCM) blocks, providing a self-calibrating, fully digital solutions for distributing, delaying, multiplying, dividing, and phase shifting clock signals.

The discussed elements are organized in the following mode: A ring of IOBs surrounding a regular array of CLBs. The array has RAM blocks embedded which are associated with a dedicated multiplier. The DCMs are position at the ends of the outer RAM blocks segments.

Spartan 3 FPGAs are programmed by loading configuration data into robust, reprogrammable, static CMOS configuration latches (CCLs) that collectively control all functional elements and routing resources. Before powering on the FPGA, configuration data is stored externally in a

PROM or some other nonvolatile devices. After applying power, the configuration data is written to the FPGA using any of the five different modes: master parallel, slave parallel, master serial, slave serial, and boundary scan (JTAG).

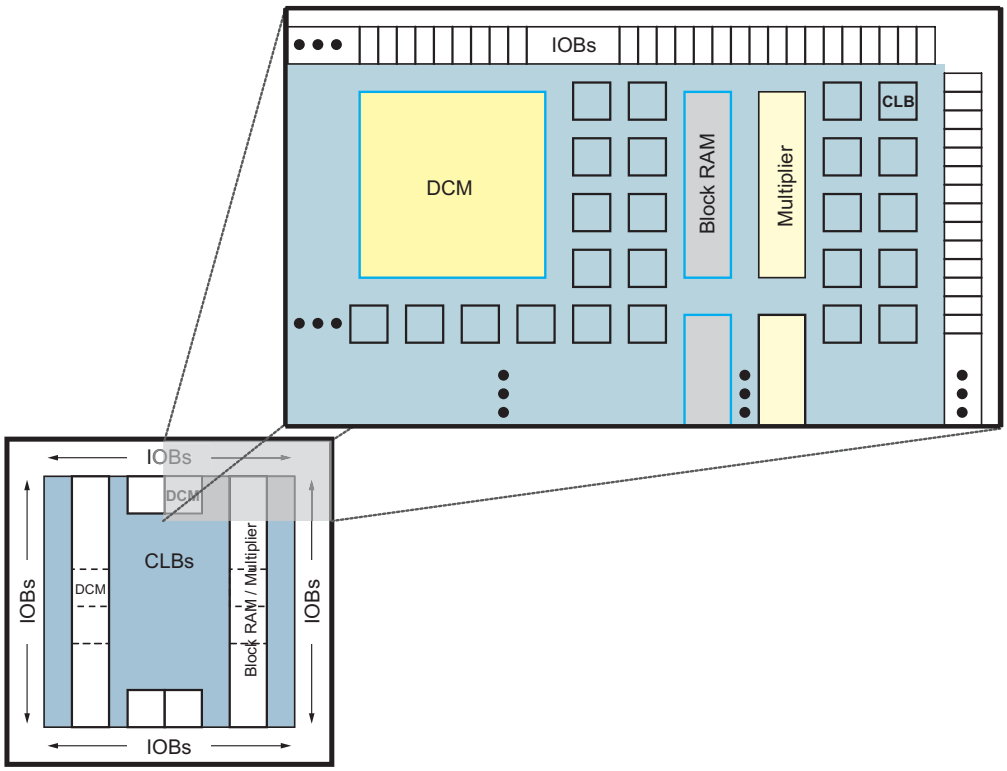


Figure 5.2.1 Spartan 3 architecture.

5.3 SPARTAN-3E STARTER KIT

The Spartan-3E family builds on the legacy of the Spartan 3 family by increasing the amount of the logic per I/O, significantly reducing the cost per logic cell. Spartan-3E new features improve system performance and reduce the cost of configuration. This family presents enhancements, combined with advanced 90nm process technology, delivering more functionality and bandwidth[33].

5.3.1 OVERVIEW

The Spartan-3E starter kit board highlights the unique features of the Spartan-3E FPGA family and provides a convenient development board for embedded processing application. The board presents parallel NOR Flash configuration, multiBoot FPGA configuration from parallel NOR Flash and SPI serial Flash configuration. This board is also capable of the following embedded development features: MicroBlaze 32-bit embedded RISC processor, PicoBlaze 8-bit embedded controller and DDR memory interfaces.

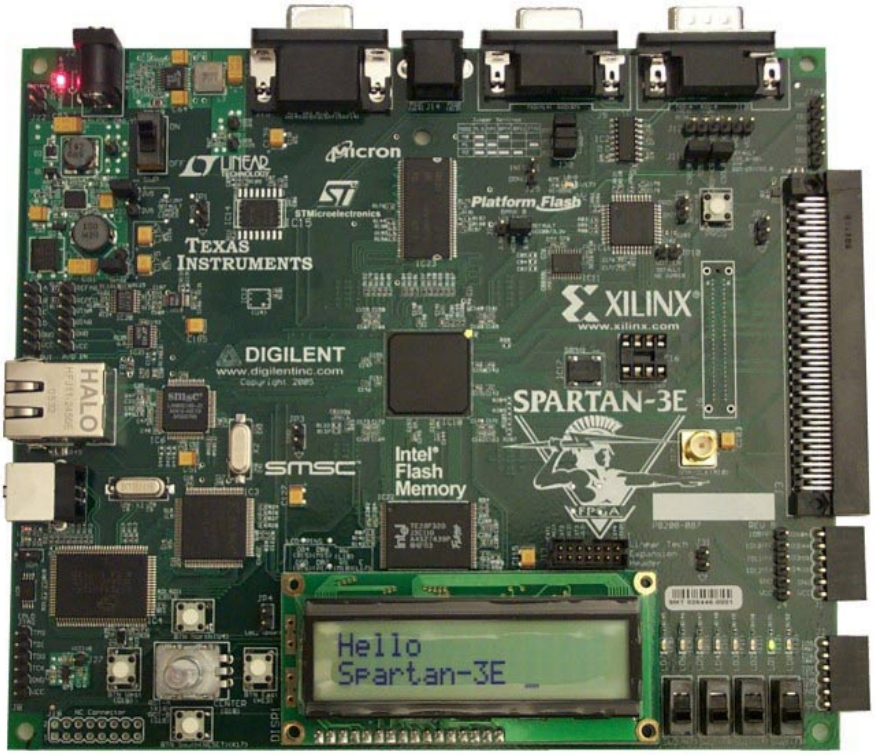


Figure 5.3.1 Spartan-3E Starter Kit[33].

5.3.2 ON-BOARD RESOURCES

Spartan-3E starter kit presents the following on-board features[33]:

- Xilinx XC3S500E Spartan-3E FPGA with 232 user I/O pins in a 320-pin package and over 10,000 logic cells.
- Clock at 50MHz.
- Xilinx 4Mbit platform flash configuration PROM.
- Xilinx 64-macrocell XC2C64A coolRunner CPLD.
- 64 MB of DDR SDRAM, 16 MB of parallel NOR Flash (Intel Strata Flash) and 16Mbs of SPI serial Flash (STMicro) for FPGA configuration storage and MicroBlaze code storage.
- 2-line, 16 character display.
- PS/2 mouse or keyboard port andVGA display port.
- 10/100 Ethernet PHY.
- Two 9-pin RS-232 ports in DTE and DCE style.
- On-board USB-based FPGA/CPLD download and debug interface.
- 50MHz clock oscillator.
- Three digilent 6-pin expansion connectors.
- Four-output SPI-base DAC, two-input SPI-base ADC with programmable gain pre-amplifier.

- Rotary-encoder with push-button shaft.
- Eighth discrete LEDs, four slide switches and four push-button switches.
- SMA clock input.

5.4 SPARTAN-3A DSP S3D1800A MICROBLAZE EDITION

Spartan-3A FPGA family is especially design for intense I/O electronics applications and low power consumption modes (e.g. Suspended, Hibernate). This family builds on the Spartan-3E and Spartan 3 legacy, by increasing the amount of I/O per logic, improving system performance and reducing the cost of configuration. The Spartan-3A DSP targets floating point Matlab algorithms to fixed-point logic. This kit also includes a license to MicroBlaze royalty free using the Xilinx platform studio.

5.4.1 OVERVIEW

The Spatan-3A DSP SD31800 Kit highlights the unique features of the Spartan-3A and Spartan-3A DSP families, providing a convenient development board for embedded digital signal processing and intense I/O electronics. This board provides the necessary hardware to evaluate advanced features of the Spartan-3A DSP family. User application can be implemented using extended plug-in modules into the EXP connectors[33].

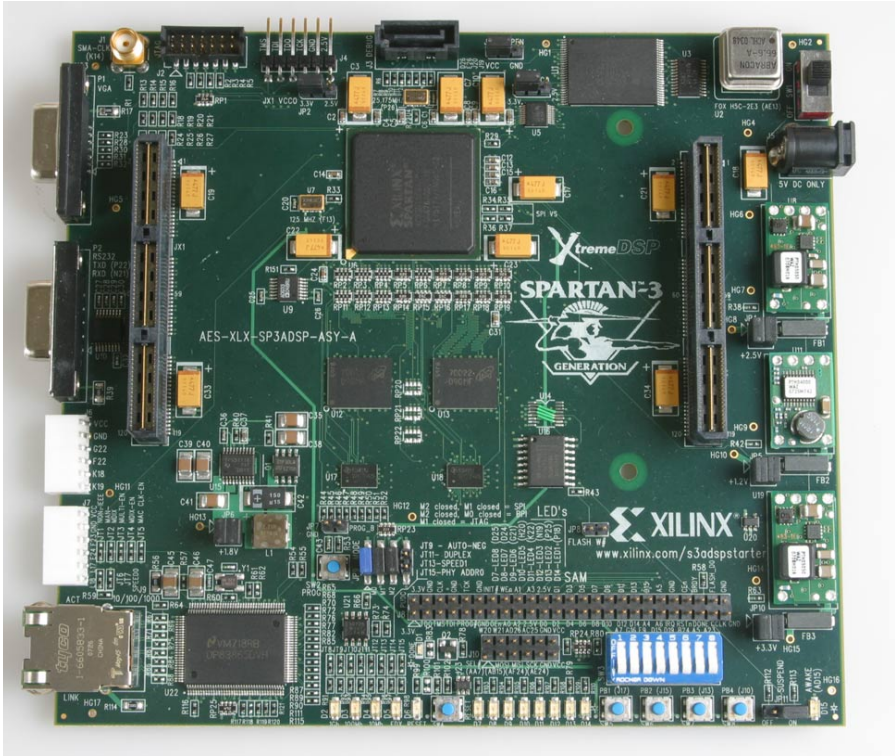


Figure 5.4.1 Spartan-3A DSP 1800[33].

5.4.2 ON-BOARD RESOURCES

Spatan-3A DSP SD31800 Kit presents the following on-board features[33]:

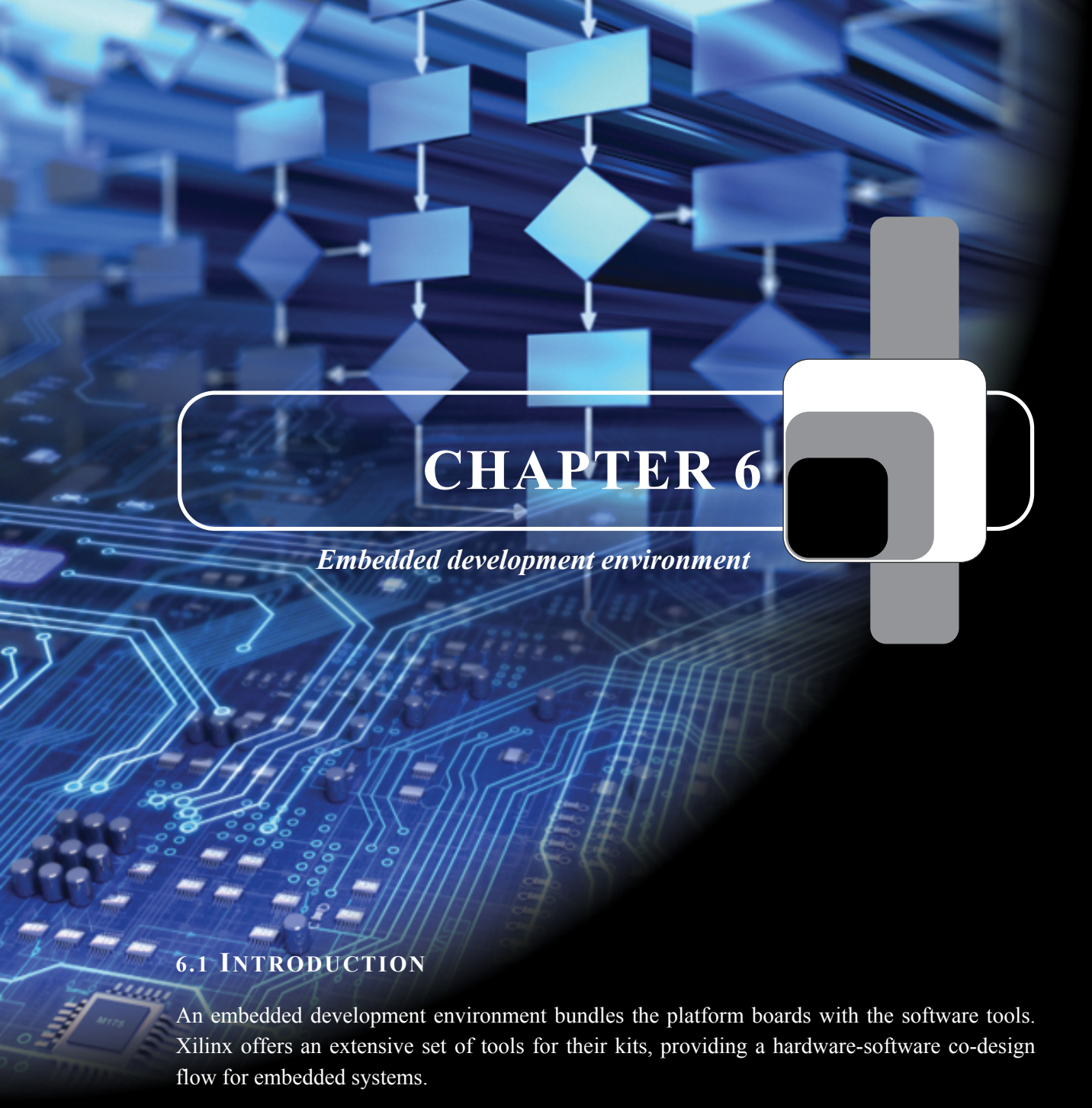
- Xilinx XC3SD1800A-4FGG676C Spartan-3A DSP FPGA.
- Clocks at 125 MHz, LVTTL SMT Oscillator for video clock.
- 25 MHz Ethernet clock.
- 128 MB of DDR2 SDRAM, 16Mx8 parallel-configuration Flash and 64 Mb SPI configuration/storage Flash.
- 10/100/1000 PHY.
- RS232, VGA, JTAG ports.
- Four SPI select lines.
- Eighth User LEDs, 8-position user DIP switch, four user push-button switch, and reset push-button switch.
- Two digilent 6-pin headers and two EXP expansion connectors.
- 30 pin general purpose I/O connector.

5.5 SUMMARY

This chapter has presented two of the several boards in which a MicroBlaze core can be implemented. These boards are based on the Spartan 3 FPGA family. The boards provide the sufficient components to implement embedded systems for the experimentation purposes of this project.

In both boards it is possible to implement several MicroBlaze cores running at the same time, the number of cores depends on the features selected; however the MicroBlaze debugging module can run up to eight cores at the time.

The Spartan 3E starter kit will be used to get familiarized with the MicroBlaze core and the initial porting of HARTEXμ into this architecture. Deeper performance analysis will be conducted on the Spartan-3A DSP S3D1800A MicroBlaze Edition while using the custom logic functions generated from the subsystems of HARTEXμ. This board has higher operation frequency that the Spartan 3E starter kit.



CHAPTER 6

Embedded development environment

6.1 INTRODUCTION

An embedded development environment bundles the platform boards with the software tools. Xilinx offers an extensive set of tools for their kits, providing a hardware-software co-design flow for embedded systems.

The two main tools for a MicroBlaze implementation are the Xilinx embedded development kit and the Xilinx integrated software environment. The former tool is targeted to create a MicroBlaze core according to the selected development board, creating all the necessary hardware parameters for its physical implementation and at the same time creation software support for any possible user application. The later tool takes the previously created parameters in physically implements them into the development board.

This chapter focuses its main discussion on the embedded development environment provided by the Xilinx tools, their different system input and the generated system outputs toward the design and configuration of processor cores and coprocessor units. The tools reviewed in this chapter are under the 9.2i released version.

6.2 XILINX EMBEDDED DEVELOPMENT KIT

The Xilinx embedded development kit (EDK) provides a design tools suite which is based on a common framework that allows the complete hardware/software co-design of an embedded processor system for implementation in a Xilinx FPGA device. Xilinx EDK suite includes The Xilinx platform studio GUI interface, the embedded system tools suite, embedded processing intellectual property cores, and the platform studio software development kit.

The tools provided with the EDK suite are especially design to assist in all the phases of the embedded system co-design process. Basically this process consists on four phases: software development, hardware development, verification and target device configuration. In the hardware development phase the EDK allows the configuration of hardware platforms which consist on one more processors and peripherals connected to the processor buses. EDK captures the hardware platform description in the microprocessor hardware specification (MHS) file.

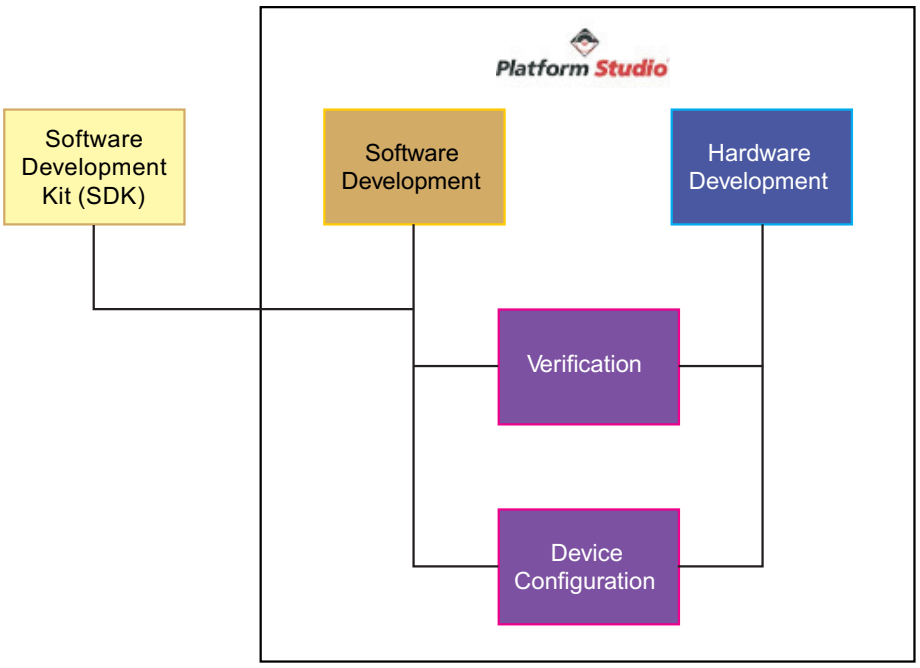


Figure 6.2.1 EDK design flow.

In the software development phase, the EDK builds a software platform which consists of software drives and, optionally, the operating system on which applications can be built. This platform is described in the microprocessor software specification (MSS) file, consisting on portions of the Xilinx library that have bee use in a particular design. Multiple applications can be created during this phase to run on the software platform. The EDK verification phase supports hardware and software validation. The correct functionality of the hardware platform is verified using a model run on an HDL simulator. Software execution is added to the hardware simulation. The hardware simulation model can be behavioral, structural or timing-

accurate model. Software verification presents the following options: software can be loaded into the development board and use the debugging tools provided by the EDK to control the target processor. Another option is to use a virtual platform, which consist of an instruction set simulation running in a host computer where the code can be debugged. System performance gauging can be accomplished by profiling the code execution.

The final phase is the device configuration. Completed hardware and software platforms are combined to create a configuration bit-stream for the target FPGA device. During the prototyping stage, the EDK download the bit-stream to run on the embedded system board while connected to a host computer. Once the system is ready for production, the configuration bit-stream is stored in a non-volatile memory connected to the targeted FPGA.

6.2.1 XILINX PLATFORM STUDIO INTERFACE

The Xilinx platform studio (XPS) is an integrated environment for software and hardware flow specification development for embedded system based on MicroBlaze and PowerPC processor. XPS offers customization of tool flow configuration options, providing a graphical system editor for processor, peripheral and bus connections.

From XPS all the embedded system tools required can be lunched, for all the hardware process and software system components. In general XPS offers the following features[33]:

- IP cores addition by editing parameters and creating bus and signal connections to generate the appropriated MHS file.
- Generation and modification of MSS files.
- Generation and view for processor system block diagram and design report.
- Multiple-user software application support and project management.
- Process and tool flow dependency management.
- Support of System ACE-CF tools, GNU software tools and board support packages (BSP)

6.2.2 XILINX SOFTWARE DEVELOPMENT KIT

The Xilinx software development kit (SDK) is a complementary graphical user interface (GUI) to XPS. Provides a development environment for software application projects according to the software libraries built for a specific processor implementation. SDK is based on the Eclipse open-source standard. SDK features include[33]:

- Feature-rich code editor and GNU compiler compilation (GCC) environment.
- Project management.
- Error navigation.
- Integrated environment for debugging and profiling embedded targets.
- Source code version control.

6.3 XILINX INTEGRATED SOFTWARE ENVIRONMENT

The Xilinx integrated software environment (ISE) is a design tool that consists of a set of programs to create, capture, simulate and implement digital designs into FPGAs and CPLDs. This collection of tools makes use of a GUI that allows all the programs to be executed from toolbars, menus or icons. ISE includes the following tools[33]:

- ISE Foundation software.
- ISE WebPACK software.
- ISE Foundation software with the ISE simulator.
- PlanAhead design and analisys tool and PlanAhead Lite
- ChipScope Pro tool.
- ChipScope Pro serial I/O toolkit.
- System generator for DPS.
- AccelDSP Systhesis tool.

ISE is essential in the EDK design process, it has to be installed in order for the EDK to run and download the embedded processor design into the target device. ISE has its own design flow process, which is incorporated in to the EDK design flow. ISE flow steps are: design entry, design synthesis, design verification, design implementation and device configuration.

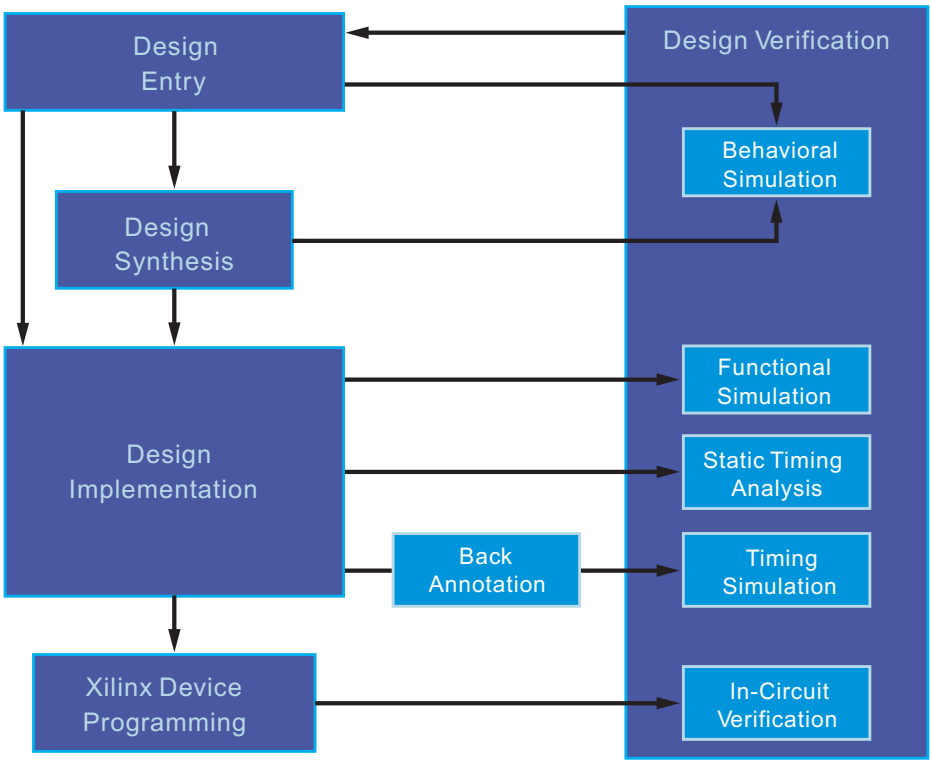


Figure 6.3.1 ISE design flow.

The first step in the ISE design flow is the design entry which can be done by creating a source file. The design entry can be captured in different formats such as schematic or HDL (VHDL, Verilog or ABEL). Typically a project design will consist of top-level source files and various low-level source files, in both cases they can be schematic or a HDL entry file.

The synthesis phase creates netlist files from the different entry source files. The netlist files can be used as the input to the implementation step. The verification stage is an important step that should be executed among the other design steps. Simulation is used to verify the functionality, timing and behaviour or a design circuit.

The implementation phase converts the previously created netlist fillies into a physical file that can be downloaded on the target device. This phase involves three sub-steps: translation the netlist into the physical file (bitstream), mapping and place&route. The final phase, device configuration, refers to the actual programming of the target FPGA or CPLD by performing the physical download of the generated bitstream.

6.4 XILINX HW/SW CO-DESIGN FLOW

Xilinx provides a robust platform for hardware-software co-design of embedded systems by combining EDK/XPS/SDK and ISE. The platform design flow merges the standard embedded software development flow with the standard FPGA hardware development flow into a single embedded system development flow. The entry point to conjoint development flow is selecting a target FPGA device and creating a microprocessor core (base system) in Xilinx EDK. Board support packages, software libraries and system neltlits are generated at this point as entries for the software standard flow and the hardware standard flow.

The embedded software flow can take any C/C++ code as an entry. This code could contain specific applications or tasks, furthermore kernel objects and libraries can be integrated to execute any given RTOS. The board support package and libraries tailor the C/C++ cross compiler. Changes in hardware are automatically reflected in the compiler. The compiled executable file can be loaded directly into a Flash memory.

The hardware development flow can take VHDL or Verilog code as entries. These codes can be used as co-processing units and hardware accelerators. Xilinx ISE can simulate and synthesize these entries together with the previously generated system netlist; they can be compiled into a bitstream file and downloaded into an FPGA.

The Xilinx EDK presents an option to merge the compiled executed and bitstream files into a Data2MEM file which can be downloaded to the base system. This file contains the information necessary to build the processor core and program the application source.

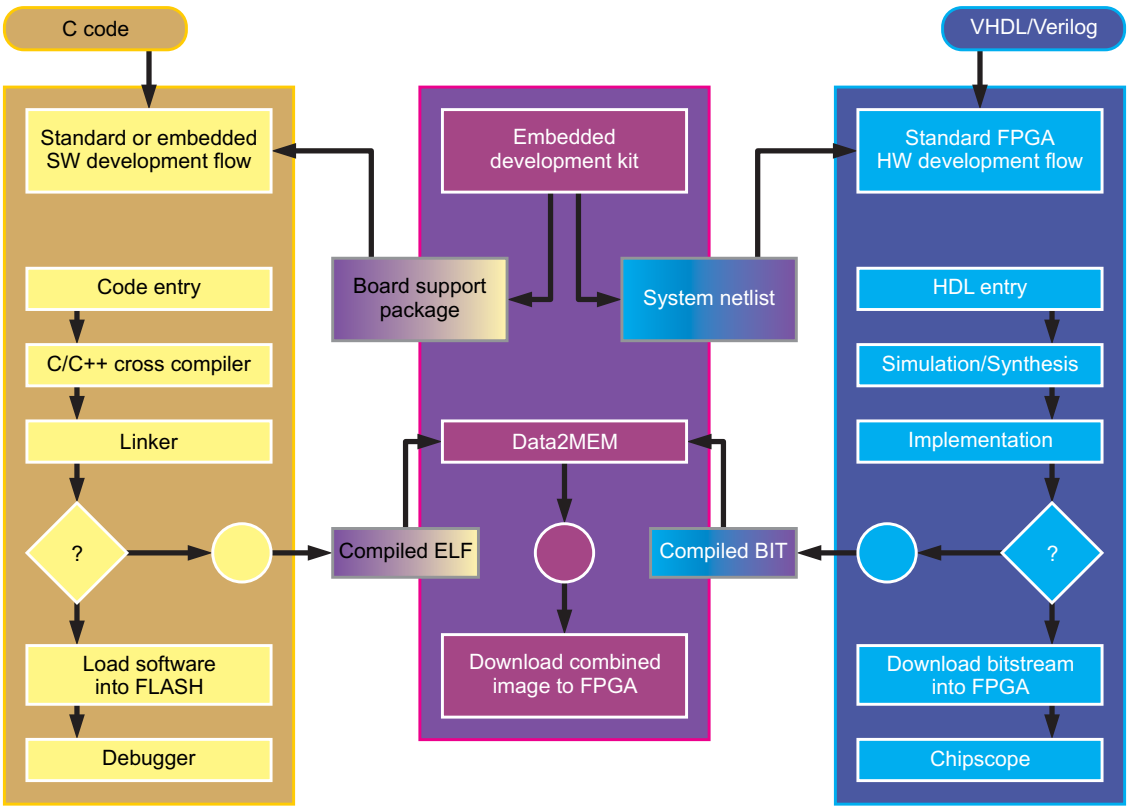


Figure 6.4.1 Xilinx HW/SW co-design flow.

6.4.1 BASE SYSTEM BUILDER

The base system builder (BSB) wizard is an assistant tool incorporated in the Xilinx EDK which helps to quickly build a working system. Simple embedded systems can be completed with the use of the BSB alone. For complex embedded systems, BSB provides a baseline system than can be further customized to specific embedded requirements.

The BSB allows the selection and configuration of basic system elements such as processor type, debugger interface, cache configuration, memory type and size, and peripherals. These features are based on the selected target device. Functional default values are pre-selected in the wizard for each one of the feature.

MHS and MSS files (Code segment 6.4.1) are generated as outputs of the BSB wizard, along with software projects. These outputs can be loaded in XPS\SDK for further enhance and fine tune the embedded system. Optionally the BSB generates system block diagrams (Figure 6.4.2), illustrative simple test applications and linker scripts than can be complied and run on the hardware for the target development board. The content of each test application might vary depending on the components of the system. XPS supports multiple software projects for every hardware system designed in the EDK. Each of the projects contains its own set of source files and their corresponding linker script.

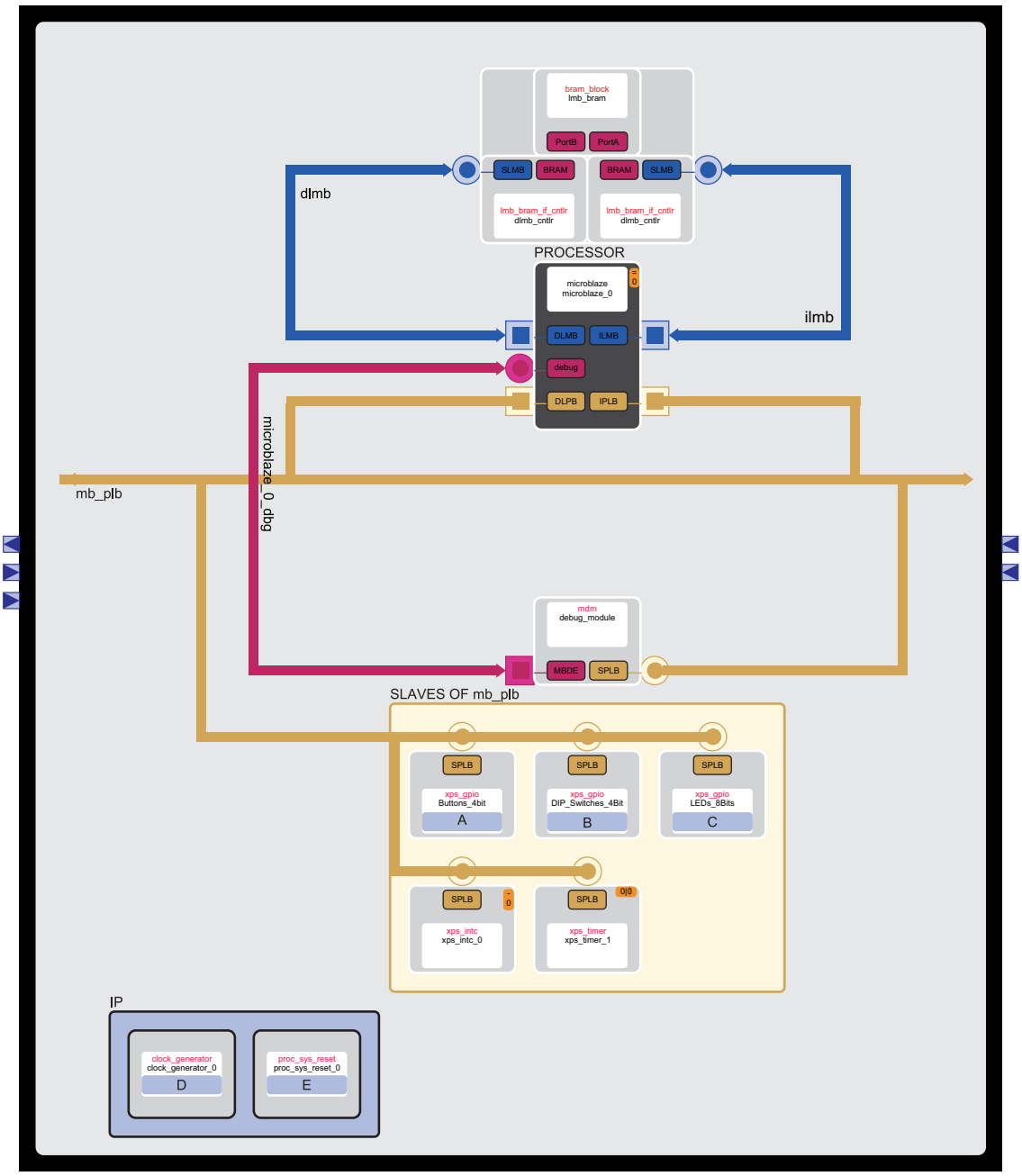
MHS FILE

```
PORT
fpga_0_DIP_Switches_4Bit_GPIO_in_pin =
fpga_0_DIP_Switches_4Bit_GPIO_in, DIR
= I, VEC = [0:3]
 PORT fpga_0_Buttons_4Bit_GPIO_in_pin
= fpga_0_Buttons_4Bit_GPIO_in, DIR =
I, VEC = [0:3]
 PORT sys_clk_pin = dcm_clk_s, DIR =
I, SIGIS = CLK, CLK_FREQ = 50000000
 PORT sys_rst_pin = sys_rst_s, DIR =
I, RST_POLARITY = 1, SIGIS = RST

BEGIN microblaze
 PARAMETER INSTANCE = microblaze_0
 PARAMETER C_INTERCONNECT = 1
 PARAMETER HW_VER = 7.10.b
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_AREA_OPTIMIZED = 1
 BUS_INTERFACE DLMB = dlmb
 BUS_INTERFACE ILMB = ilmb
 BUS_INTERFACE DPLB = mb_plb
 BUS_INTERFACE IPLB = mb_plb
 BUS_INTERFACE DEBUG =
microblaze_0_dbg
 PORT MB_RESET = mb_reset
 PORT Interrupt = Interrupt
END

BEGIN plb_v46
 PARAMETER INSTANCE = mb_plb
 PARAMETER HW_VER = 1.02.a
 PORT PLB_Clk = sys_clk_s
 PORT SYS_Rst = sys_bus_reset
END

BEGIN lmb_v10
 PARAMETER INSTANCE = ilmb
 PARAMETER HW_VER = 1.00.a
 PORT LMB_Clk = sys_clk_s
 PORT SYS_Rst = sys_bus_reset
END

BEGIN lmb_v10
 PARAMETER INSTANCE = dlmb
 PARAMETER HW_VER = 1.00.a
 PORT LMB_Clk = sys_clk_s
 PORT SYS_Rst = sys_bus_reset
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = dlmb_cntlr
 PARAMETER HW_VER = 2.10.a
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00007fff
 BUS_INTERFACE SLMB = dlmb
 BUS_INTERFACE BRAM_PORT = dlmb_port
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = ilmb_cntlr
 PARAMETER HW_VER = 2.10.a
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00007fff
...
```

MSS FILE

```
BEGIN PROCESSOR
 PARAMETER DRIVER_NAME = cpu
 PARAMETER DRIVER_VER = 1.11.a
 PARAMETER HW_INSTANCE = microblaze_0
 PARAMETER COMPILER = mb-gcc
 PARAMETER ARCHIVER = mb-ar
 PARAMETER CORE_CLOCK_FREQ_HZ =
50000000
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = bram
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = dlmb_cntlr
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = bram
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = ilmb_cntlr
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = lmb_bram
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = gpio
 PARAMETER DRIVER_VER = 2.12.a
 PARAMETER HW_INSTANCE = LEDs_8Bit
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = gpio
 PARAMETER DRIVER_VER = 2.12.a
 PARAMETER HW_INSTANCE =
DIP_Switches_4Bit
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = gpio
 PARAMETER DRIVER_VER = 2.12.a
 PARAMETER HW_INSTANCE = Buttons_4Bit
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = tmrctr
 PARAMETER DRIVER_VER = 1.10.b
 PARAMETER HW_INSTANCE = xps_timer_1
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE =
proc_sys_reset_0
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = intc
 PARAMETER DRIVER_VER = 1.11.a
 ...
```

Code segment 6.4.1 Generated MHS and MSS files by the BSB wizard.



SPECS	
EDK VERSION	10.1.01
ARCH	spartan3e
PART	xc3s500efg320-4
GENERATED	Wed May 07 12:45:52 2008

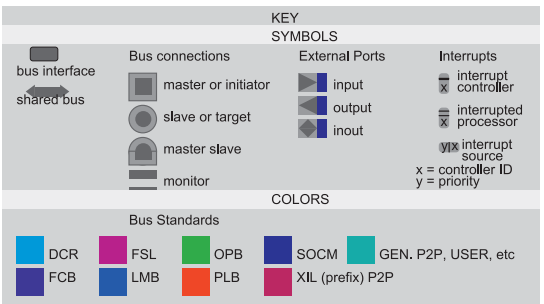


Figure 6.4.2 Embedded hardware block diagram.

6.5 SUMMARY

This chapter has reviewed all the necessary tools to implement a MicroBlaze core and the development of software applications for this processor. Xilinx presents a comprehensible embedded development framework in which the co-design is possible and changes can be reflected in either side of the development flow easily.

Xilinx EDK is the primary tool for co-development; the entire system can be build inside this environment. However for experience VHDL/Verilog programmers it is possible to fine-tune the hardware configuration with Xilinx ISE. Software application can be developed in EDK or separately in the SDK and later be imported to the embedded project in EDK. The porting of software applications into the different Spartan 3 boards can be achieve effortless as long as the generated libraries are used to link the different processor functionalities.

A base system can be built with the assistance of the BSB wizard, making the first design stages of an embedded system simple. The EDK allows easy integration of VHDL/Verilog code to the embedded system, either as a co-processing unit or an IP core, though the general o dedicated buses.

CHAPTER 7

HARTEXμ porting & HW/SW co-design

7.1 INTRODUCTION

The research presented in this thesis leads to the experimentation phase presented in this chapter with the porting of HARTEXμ kernel to the MicroBlaze soft-core processor platform and the hardware/software co-design and implementation.

An introduction to HARTEXμ has been provided in chapter one; a detailed general overview of the kernel is presented in this chapter, including the organization and different subsystems which integrate this operational environment.

A MicroBlaze framework is presented, in which HARTEXμ can be fully operational and can be used throughout different core configurations and Spartan-3 families. A performance analysis is conducted to review the execution rates of the kernel primitives, services and functions, providing pointer to find the most suitable candidates to be implemented in hardware. Based on the performance results with the hardware/software co-design implementation experiments, a HARTEXμ multi-core architecture is proposed, which takes advantage of today's high density integration scale of FPGA.

7.2 HARTEXμ KERNEL

The HARTEXμ is a small RTOS implementation for a single microprocessor which usually has very limited resources. This kernel was originally designed and implemented for the Atmel AVR microcontroller family. HARTEXμ has been successfully ported to other micro-controllers such as Hitachi H8/300, Motorola ColdFire and ARM7.

Minimum memory and response overhead are the stronger features of this kernel, without compromising any of the basic services and functionality. HARTEXμ supports basic tasks following specific task-modeling with fixed priority and preemptive scheduling policy.

HARTEXμ operation mode can be switched between preemptive and non-preemptive. This kernel supports non-blocking synchronization and communication mechanisms. The memory footprint of the kernel is extremely small, having a high responsiveness with a minimum jitter execution.

This kernel is tick and event driven, it makes used of binary vectors where the bit position identifies the task and its priority. Semaphores follow this mechanism to synchronize and communicate among the different task the occurrence of event or message arrival. HARTEXμ presents a uniform treatment for tick interrupts and all extern events where task can be released when the appropriated time has lapsed or external events thresholds have occurred.

All the basic tasks under this kernel share a common stack which is reflected in a lower memory overhead. Resource management is handled by a system ceiling protocol ensuring safe access to the system resources. HARTEXμ exhibits a content-oriented message addressing implementation which addresses the messages among the different task only by the communicated variable name providing a transparent communication mechanism while freeing the user of the communication details.

These characteristics make HARTEXμ a very scalable and compilable kernel[54]. This kernel can be scaled to meet any application requirements by configuring the different kernel subsystems. Furthermore, data structures can be compiled to be allocated according to the number of kernel objects required by the applications.

7.2.1 KERNEL ORGANIZATION AND SUBSYSTEMS

HARTEXμ is organized into various subsystems which encapsulate several system objects, i.e. tasks, messages, semaphores, resources, events that are set up during the system configuration. Each subsystem provides calls to other subsystems, these set of subroutine calls and services are known as the kernel primitives. HARTEXμ is conformed by the following subsystems: task manager, resource manager, software bus and integrated event manager. To avoid full

dependency of these subsystems on the hardware, an adaptation layer (HAL) is provided to establish the corresponding links.

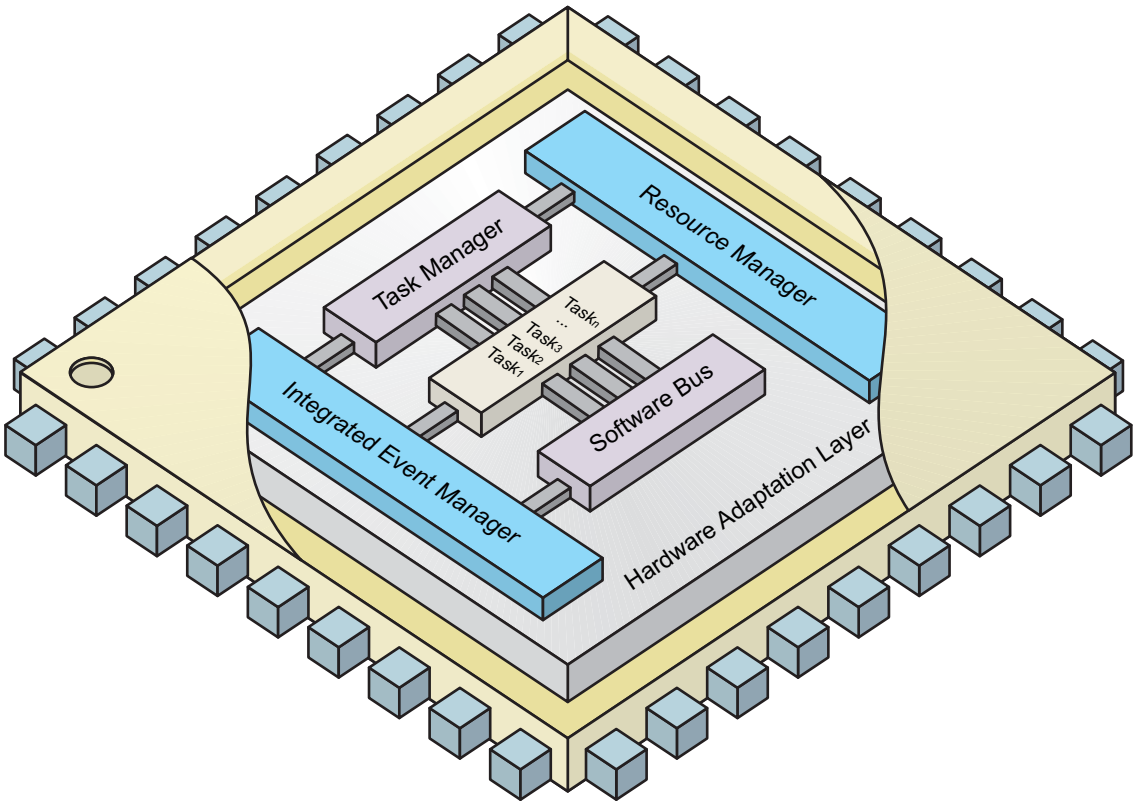


Figure 7.2.1 HARTEXμ kernel structure.

The *task manager* is responsible for preparing and managing the different tasks according to the scheduling policy. The task manager encapsulates data structures while providing interfaces to the rest of the subsystems. It is also responsible for task context switching during preemption.

The *resource manager* controls the access to shared resources in a mutually exclusive fashion. This implementation results in an integrated task and resource management. The *software bus* provides the necessary inter task synchronization and communication. The *integrated event manager* handles the timing and external events by invoking the appropriated primitives.

7.2.2 TASK MANAGEMENT

HARTEXμ supports only basic task without any significant limitation for practical applications. Single processor systems require multitasking which is carried out by the kernel by allocating processing time and resources to the system tasks, however this is done one task at the time. To do so, tasks are labeled with their current priority order processing condition,

which is referred as state. The task model in HARTEXμ consists of the following states: running, ready, preempted and inactive.

The running state corresponds to the executed task at the current time, whereas the ready state corresponds to the task prepared to start execution. The preempted state is assigned to a task when it has been preempted by a higher priority task. Finally the inactive state is used to indicate that the task has finished its execution and is no longer required in any of the other states. Figure 7.2.2 illustrates the state transitions handled by the task manager.

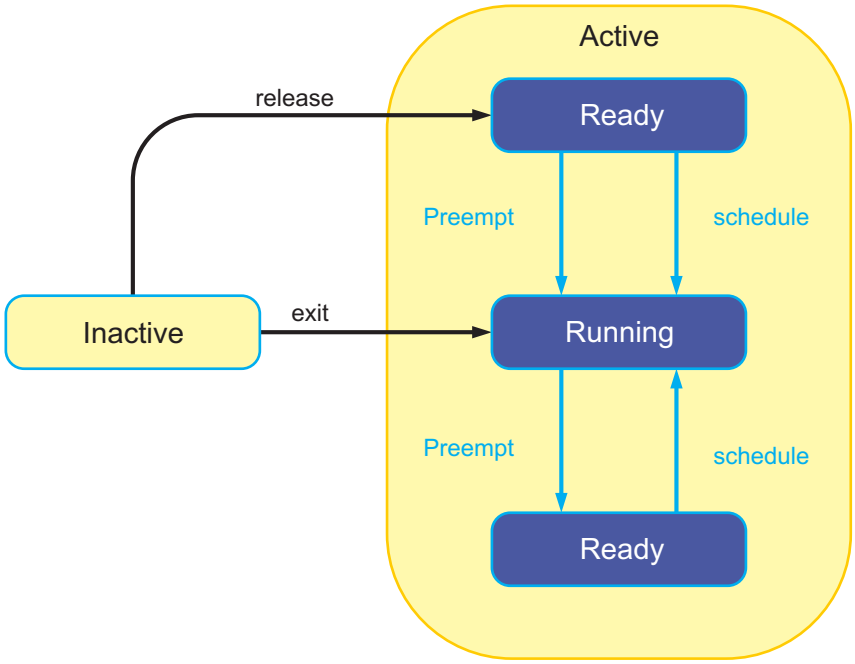


Figure 7.2.2 Task state transition diagram.

INTEGRATED TASK AND RESOURCE MANAGEMENT

In multitasking kernels data and shared resources shall remain consistent throughout the different accesses by tasks. These accesses can be conducted in an atomic fashion and are performed in two styles: use of resource reservation and use of execution locking.

When using resource reservation a task locks a resource so the rest of the task using the same resource should execute another activity. This kind of operation can be implemented with mechanisms such as mutex or semaphores. On the other hand, the use of execution locking implies that the task lock the shared resource and prevents execution time for the rest of the tasks as long as the locking is required.

The first approach suffers from deadlocks and unbounded priority inversion. The second style does not have any of these problems; since HARTEXμ is equipped with a stack-based ceiling

priority protocol that prevents these scenarios from happening while achieving an integrated resource and task management.

The resource management service provides locking and unlocking to the shared resources; it is done by using the lock and unlock primitives. The tasks sections with shared resources enclosed under the lock and unlock primitives are known as critical regions.

There are two possible scenarios for multiple locking resources: *Sequential locking*, where resources are lock and unlock by the task in series, and *nested locking* where a task uses multiple shared resources simultaneously so all the lock take place first and the unlocking sequence stars with the last resource locked by the tasks.

7.2.3 TASK INTERACTION

This kernel employs asynchronous producer-consumer interaction which is non-blocking since the tasks are basic. Task communication can be achieved by message communication via content oriented message or asynchronous event notification via vector semaphore. Task synchronization is achieved by vector semaphores to asynchronously notify an event.

Vector semaphores are objects or data entities than can be review and modify by any task. In HARTEXμ the semaphores are 8-bit Boolean vector that are used to indicate which task is signaled and released. Task synchronization using this mechanism in this kernel is presented in the following variants: one-to-one event notification, one-to-many event notification, many-to-one event notification, many-to-many event notification and broadcast event notification.

The content-oriented message method implies the sender, receiver and message buffer are specified by the name of the variable or message being sent or received. Message queues are eliminated, achieving a very fast, transparent and time-fixed communication system, releasing the user from implementation details.

7.2.4 EVENT MANAGEMENT

This kernel presents a simple and uniform treatment of all interrupt, whether they are generated by timer interrupts (ticks) or external hardware interrupts. The integrated event manager subsystem provides the integrated time and event management.

The timing event processing in this kernel is done by means of a basic tick interrupt with a time period of 1 millisecond (basic time unit). To achieve larger time periods the tick routine has four flags corresponding to the following resolution: basic time unit, one second, one minute and one hour. The external event processing takes place when the external interrupt request is handled by the interrupt service routine which carries out the basic processing and invokes the integrated event manager through the associated event primitives and objects.

7.3 MICROBLAZE FRAMEWORK FOR HARTEXμ KERNEL

The first step toward the experimentation phase of this project is the porting of the HARTEXμ kernel to the MicroBlaze processor architecture. The necessary libraries are created when a MicroBlaze core is configured and generated (see Appendix A).

The processor parameters are placed in "xparameters.h". Hardware changes are directly reflected into this file, it is recommended to use the labeling provided by this file to keep the embedded system updated. This framework activates the following processor components:

- Configuration of general purpose input/output (GPIO) ports.
- Configuration and initialization of timer tick.
- Configuration and enabling the tick interrupt request and service routine.

Driver functions are provided in the libraries. GPIO driver functions are located in the "xgpio_l" files, timers are in "xtmrctr_l" and interrupts are in "xintc_l"[55]. HARTEXμ provides a HAL where all this drives can be configured and initialized. Inside the hardware initialization routine, the following functions are called:

- *XGpio_mSetDataDirection*. This function sets the data direction for the GPIOs. The Spartan 3E starter kit has one 8-bit LED port which is addressed by XPAR_LEDS_8BIT_BASEADDR and it is set as an output, and one 4-bit DIP switch port addressed by XPAR_DIP_SWITCHES_4BIT_BASEADDR and it is set as in input.
- *XTmrCtr_mSetLoadReg*. This function set the number of cycles in the register corresponding to the timer. In this case the register can be address by the label located in the parameters file as XPAR_XPS_TIMER_1_BASEADDR.
- *XTmrCtr_mSetControlStatusReg*. The timers are reset and the interrupts are cleared in this function, the following masks are enabled: XTC_CSR_ENABLE_TMR_MASK, XTC_CSR_ENABLE_INT_MASK which enable timers and interrupts respectively, XTC_CSR_AUTO_RELOAD_MASK enables the automatic reloading of the counter after the timeout and XTC_CSR_DOWN_COUNT_MASK set a decreasing count.
- *XIntc_RegisterHandler*. A pointer of the timer handler is passed as a parameter to this function and registers it in an interrupt handler to the timer interrupt.
- *XIntc_mMasterEnable*. This function starts the interrupt controller for the defined interrupt, in this case XPAR_XPS_INTC_0_BASEADDR.
- XIntc_mEnableIntr. The timer interrupts are enabled in the interrupt controller in this function, linking the XPAR_XPS_INTC_0_BASEADDR to the corresponding timer mask labeled as XPAR_XPS_TIMER_1_INTERRUPT_MASK.

An interrupt service routine has to be created to handle the timer interrupt request where the kernel tick shall be allocated. Basically this routine reads the timer register and checks for the

timer interrupt, in case the interrupt has occurred, the XTC_CSR_INT_OCCURED_MASK is cleared, the interrupt is acknowledge and the kernel tick is invoke.

```
// Set LED as outputs
XGpio_mSetDataDirection(XPAR_LEDS_8BIT_BASEADDR, 1, 0x00000000);

// Set DIP Swithces as inputs
XGpio_mSetDataDirection(XPAR_DIP_SWITCHES_4BIT_BASEADDR, 1, 0xffffffff);

// Serve all interrups
XIntc_SetIntrSvcOption(XPAR_XPS_INTC_0_BASEADDR,
                       XIN_SVC_ALL_ISRS_OPTION);

// Register timer interrupt handler
XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
                      XPAR_XPS_INTC_0_XPS_TIMER_1_INTERRUPT_INTR,
                     (XInterruptHandler)timer1_int_handler,
                     (void*)XPAR_XPS_TIMER_1_BASEADDR);

// Start the interrupt controller
XIntc_mMasterEnable(XPAR_XPS_INTC_0_BASEADDR);

// Set the number of cycles before an interruption
XTmrCtr_mSetLoadReg(XPAR_XPS_TIMER_1_BASEADDR, 0, ONE_MS_TIME);

// Reset timers and clear interruptions
XTmrCtr_mSetControlStatusReg(XPAR_XPS_TIMER_1_BASEADDR, 0,
                             XTC_CSR_ENABLE_TMR_MASK |
                             XTC_CSR_ENABLE_INT_MASK |
                             XTC_CSR_AUTO_RELOAD_MASK |
                             XTC_CSR_DOWN_COUNT_MASK);

// Enable timer interrupts in the interrupt controller
   XIntc_mEnableIntr(XPAR_XPS_INTC_0_BASEADDR,
                     XPAR_XPS_TIMER_1_INTERRUPT_MASK);
```

Code segment 7.3.1 MicroBlaze hardware initialization for HARTEXμ.

```
unsigned int timer_csr;
//  Read timer 0 CSR to see if it raised the interrupt
timer_csr = XTmrCtr_mGetControlStatusReg(ref, 0);

if ( timer_csr & XTC_CSR_INT_OCCURED_MASK ){
     // Clear the timer interrupt
     XTmrCtr_mSetControlStatusReg(ref, 0, timer_csr );
     // Acknowledge interrupt
     XIntc_mAckIntr(XPAR_XPS_INTC_0_BASEADDR,
     tick();
    }
```

Code segment 7.3.2 Timer interrupt service routine.

For optimal performance and control, the enable and disable interrupts are handled manually by clearing and setting the interrupt enable bit in the machine status register (rmsr) and temporarily storing it in one of the general purpose register, in this case register R12.

```
#define DISABLE_INTERRUPTS {  \
   asm volatile ( \
         "mfs  r12, rmsr      \n\t"     /* Read the MSR              */\
         "andni r12, r12, 2   \n\t"     /* Clear the IE bit          */\
         "mts   rmsr, r12     \n\t"     /* Write to the MSR          */\
         : \
         : \
         : "r12" ); \
}

#define ENABLE_INTERRUPTS {   \
   asm volatile ( \
         "mfs r12, rmsr       \n\t"     /* Read the MSR              */\
         "ori r12, r12, 2     \n\t"     /* Set the IE bit            */\
         "mts rmsr, r12       \n\t"     /* Write to the MSR          */\
         : \
         : \
         : "r12" ); \
}
```

Code segment 7.3.3 Enable and disabling MicroBlaze interrupts.

As mention earlier, HARTEXμ can be configured to preempt tasks. In order to prevent critical sections from being preempted, interrupts shall be disabled during this sections and reestablished in according to the previous state of the interrupt flag (Code segment 7.3.4, 7.3.5).

```
#define ENTER_CRITICAL EnterCritical()

void EnterCritical(void)
{
 if ( !nestedCritical++ )
    STORE_INTERRUPTS_CONTEXT(interruptFlag);
}

#define STORE_INTERRUPTS_CONTEXT (IEFLAG) {  \
    asm volatile ( \
           "mfs   r12, rmsr         \n\t"    /* Read the MSR          */\
           "mfs   %0, rmsr          \n\t"                               \
           "andi  %0, %0, 2         \n\t"                               \
           "andni r12, r12, 2       \n\t"    /* Clear the IE bit      */\
           "mts   rmsr, r12         \n\t"    /* Write to the MSR      */\
           : "=r" ((ULONG)IEFLAG)\
           :: "r12" ); \
    }
```

Code segment 7.3.4 Enter critical section function.

```
#define EXIT_CRITICAL ExitCritical()

void ExitCritical(void)
{
 if ( !--nestedCritical )
    RESTORE_INTERRUPTS_CONTEXT(interruptFlag);
}

#define RESTORE_INTERRUPTS_CONTEXT (IEFLAG) {  \
    asm volatile ( \
           "mfs  r12, rmsr       \n\t"     /* Read the MSR            */\
           "or   r12, r12, %0    \n\t"     /* Set the IE bit          */\
           "mts  rmsr, r12       \n\t"     /* Write to the MSR        */\
           : \
           : "r" ((ULONG)IEFLAG)  \
           : "r12" ); \
}
```

Code segment 7.3.5 Exit critical section function.

7.4 HARTEXμ PERFORMANCE ANALYSIS

The implementation of a kernel introduces execution overhead into the system, which can lead to execution jitter and reduce the task responsiveness of the system. HARTEXμ is no exception to these effects; however they are significantly reduced by using Boolean vector processing among the different kernel primitives.

In this section HARTEXμ is implemented and executed in a Spartan-3A DSP 1800 board at 125MHz with four tasks accessing the LED port at different rates. The attention is focused in the kernel primitives, services and functions, and their respective execution times. In particular one of the functions call by the preempt and schedule primitives: *find_msb( )* which finds the most significant bit of a Boolean vector and returns the number of the highest priority task to be executed. Software algorithm implementations of *find_msb( )* has been researched previously by the Software Engineering group at the MCI[10][11]; furthermore it has been implemented into an FPGA and memory-mapped to a microcontroller using the extended memory addresses[52].

SOFTWARE PROFILING USING GNU GPROF

To measure the execution rates of these functions Xilinx EDK/SDK includes a profiling feature based on the GNU gprof tool; however this technique is considered extremely intrusive since it requires periodic interruptions of the software application execution to obtain samples of its program counter locations. The software application has to be built with the –pg option, indicating to the compiler to insert a call to _mcount function after every function call. A timer interrupt is also required when compiling under the profiling mode, which provides the sampling rate. At higher frequencies more sample are obtained, by default the frequency is set

to 10KHz[51]. This technique presents several problems when profiling a software application in the presence of an RTOS. Measurements become inaccurate when the sampling frequency is in the same range as the kernel tick. At lower sampling frequencies the execution data is insufficient to profile the application properly. Another drawback it that the time spent in sections where interrupts are disabled can not be measured, so primitives under atomic execution can not be shown by the profiler.

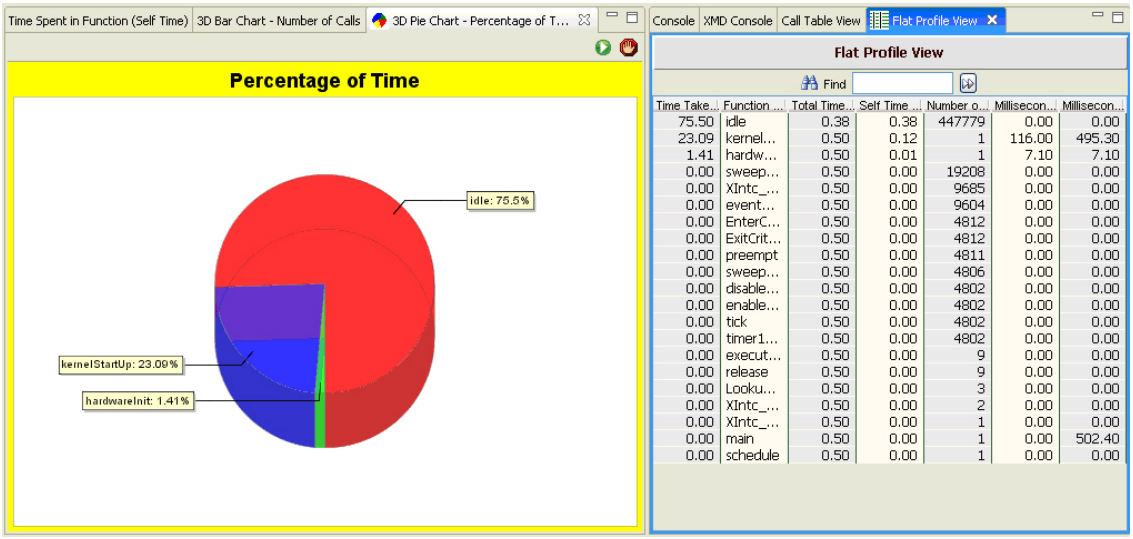


Figure 7.4.1 Kernel profiling at 10KHz.

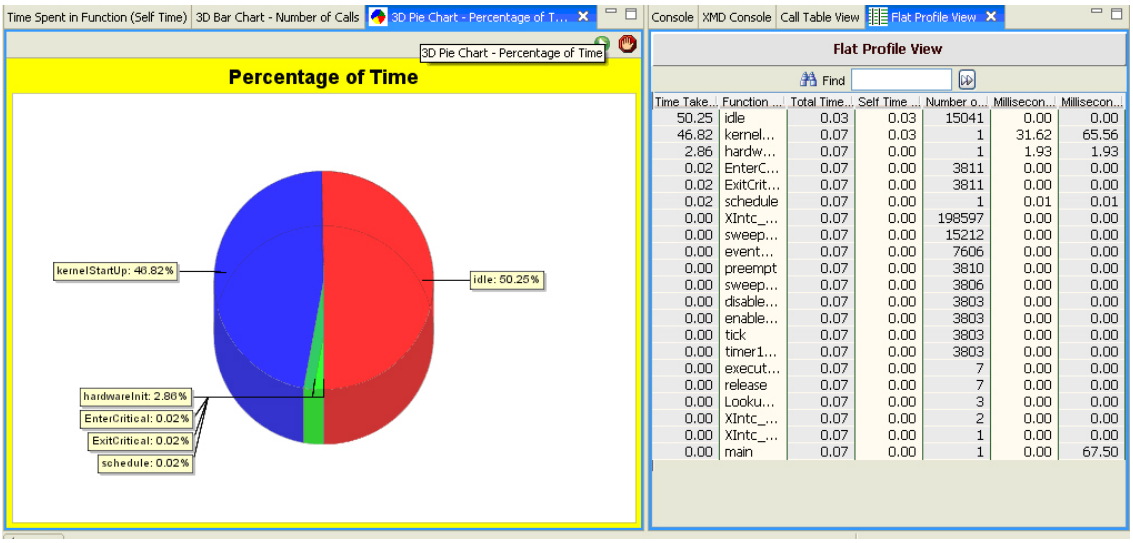


Figure 7.4.2 Kernel profiling at 100KHz.

SOFTWARE PROFILING USING HARDWARE-BASED MEASUREMENTS

A more traditional and less intrusive approach has been taken to measure the execution rates of the kernel. A pin is turn on when the function is entered and it turns off when the function is

exited, an extra port has been created and configure in HDL for this purpose (Appendix A). The following functions are monitored: *tick( ), preempt( ), schedule( ), find_msb( ), release( ), task_exit( ) and enable_preempt( )* along with the execution of the four illustrative tasks, providing the execution trace shown in Figure 7.4.3.

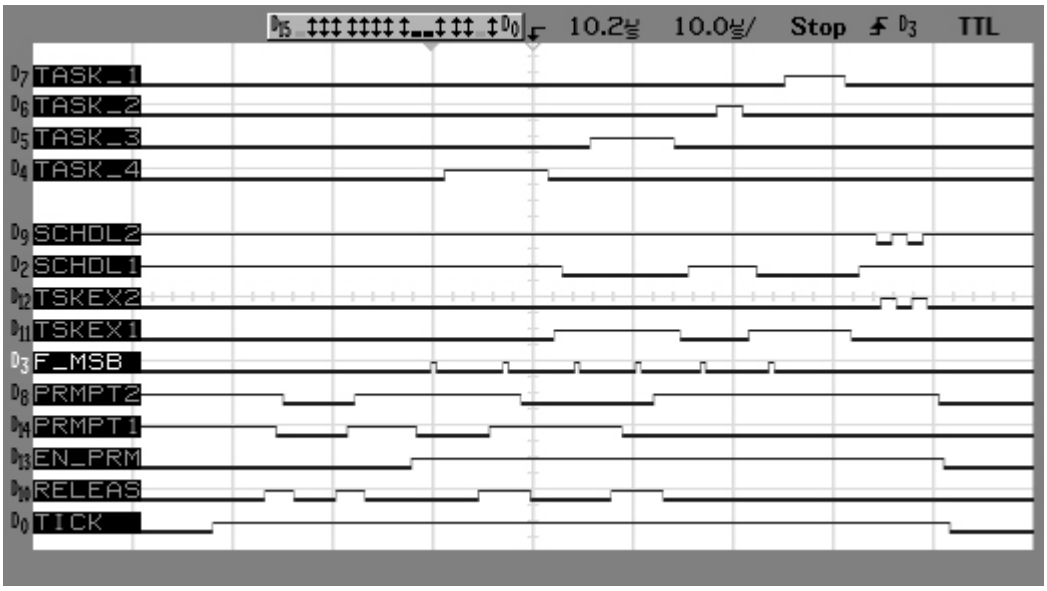


Figure 7.4.3 HARTEXµ execution trace.

The execution time for this trace takes around 78µs, in which all the kernel primitives, services and functions are executed. In this trace it has been taken into account that some kernel primitives can be called recursively, such cases are *preempt( )*, *schedule( )* and *task_exit( )* primitives, in which a toggling pins is added at the entrance and another toggling pin is added at the exit of the primitives. In Figure 7.4.3 preempt is labeled as prmpt1 (entrance) and prmpt2 (exit), the same convention is applied for taks_exit (tskex) and schedule (schdl).

The scenario shown in this trace corresponds to the "super period[11]" where all the tasks shall be released simultaneously. While *tick( )* processes all the possible external events, *preempt( )* is disabled. Once *preempt( )* is enabled, *find_msb( )* is executed returning the number of the highest priority task, in this case task number four.

When task four has been served, it invokes the *task_exit( )* primitive which invokes the *schedule( )* primitive. The later primitive calls *find_msb( )* in order to determine the following task to be executed (at this point task number three) while making a recursive call to *preempt( )* which again calls *find_msb( )*. Once task three has been serve it call *task_exit( )* which makes a recursive call to *schedule( )*. This process continues until all the tasks are served. It can be seen that *find_msb( )* is called frequently with a constant execution rate. The *find_msb( )* algorithm compares the input with the middle value of the vector range to placed the input in one half of

the range, which is taken as the new range for the next comparison iteration. The middle value of the new range is compared with the input placing it a smaller range. This process is repeated until the comparison range has only two values.

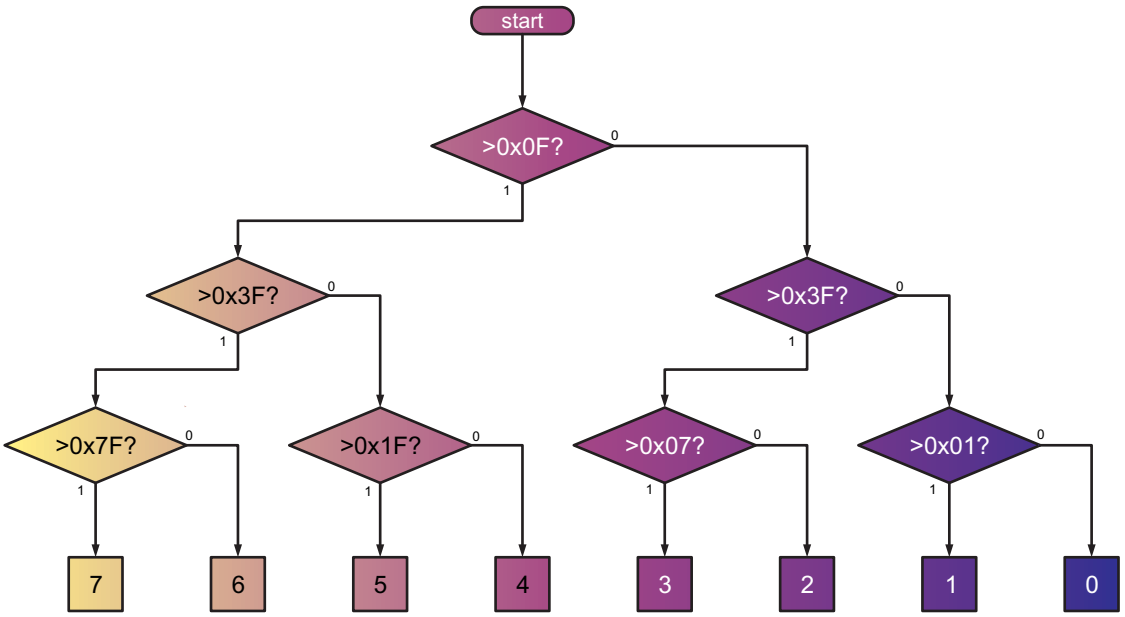


Figure 7.4.4 Algorithm for *find_msb( )* function.

The algorithm execution rate is measured in the following cases: when processing an 8 bits and 32 bits Boolean vectors. The measurements are shown in figures 7.4.5 and 7.4.6 giving an execution time of 592ns and 868ns respectively.

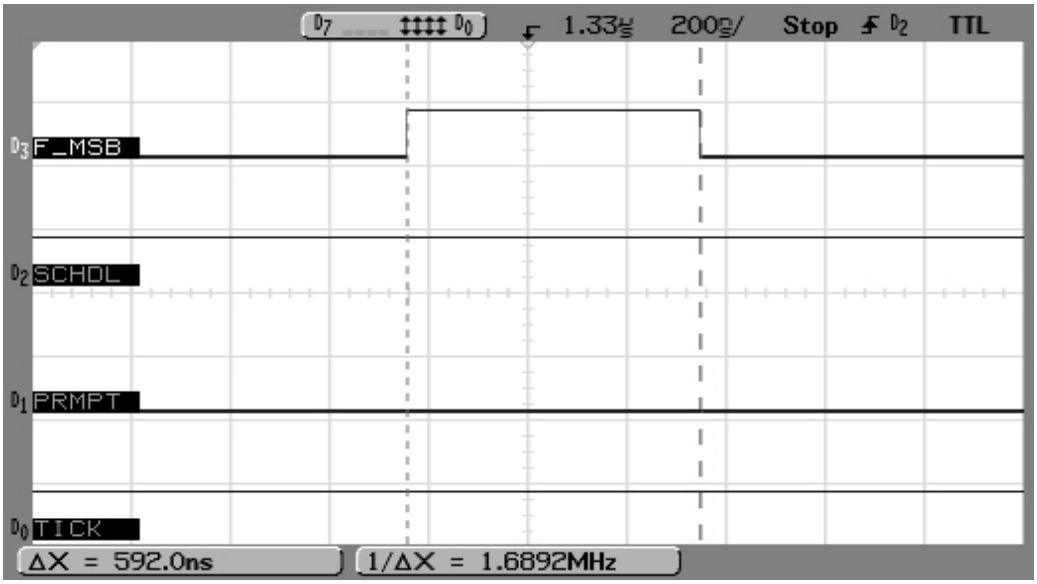


Figure 7.4.5 Execution time of *find_msb( )* for 8bits.

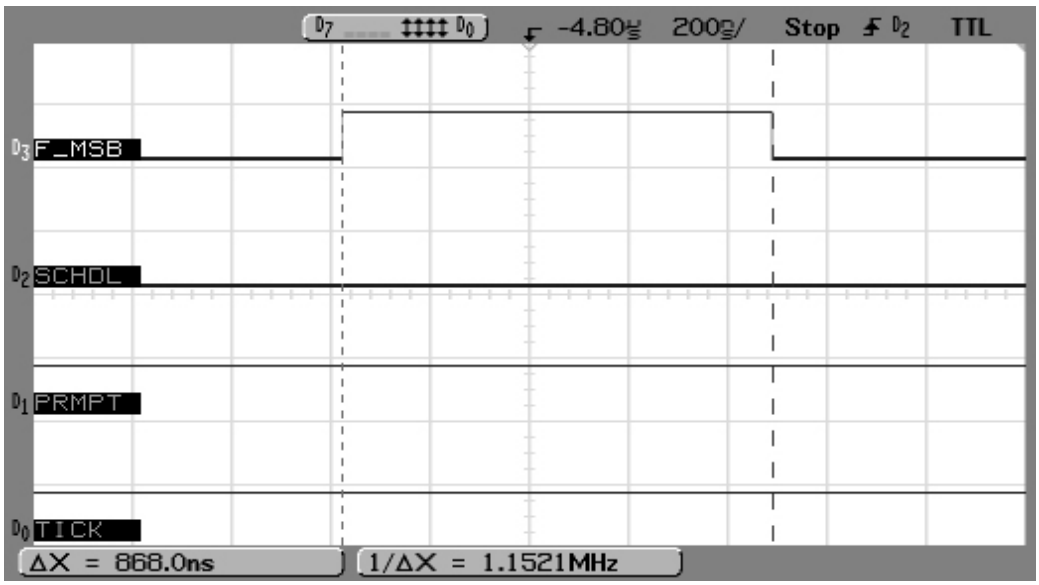


Figure 7.4.6 Execution time of *find_msb( )* for 32bits.

7.5 HARTEXμ HW/SW CO-DESIGN

From the previous results and due to the simplicity of its algorithm, the *find_msb( )* function is the first candidate to be moved into hardware. Based on previous research work at MCI[52], this function can be implemented in HDL (in this case Verilog) as an IP core and create a FSL interface "wrap" around it (see Appendix B).

Xilinx tools offer a coprocessor and IP core wizard that takes care of the communication interface. For this project a new IP core is created under the name of find_msb_core (Figure 7.5.1). This core contains the necessary HDL code to establish the communication link between the IP core and the MicroBlaze processor, including the signal and sequence interface.

The coprocessor and IP wizard creates a project for Xilinx ISE in which the algorithm for the *find_msb( )* function can be inserted in the form of a HDL code. The manual translation of C to HDL is simple for this case, due to the nature of this algorithm. Essentially the hardware algorithm behaves in the same manner as its software counterpart. A 32bit vector is sent from the processor to the coprocessor/IP core and it returns a 32-bit vector containing the most significant bit in the form of an index.

Once the find_msb_core is added, small changes on the software side are required (Code segment 7.5.1). The libraries and BSBs shall be generated again. The communication interface is in the mb_interface.h file. The communication over FSL can have blocking or non-blocking nature. For this project, communication has been chosen in blocking mode, because the execution of this function can not proceed without getting the most significant bit.

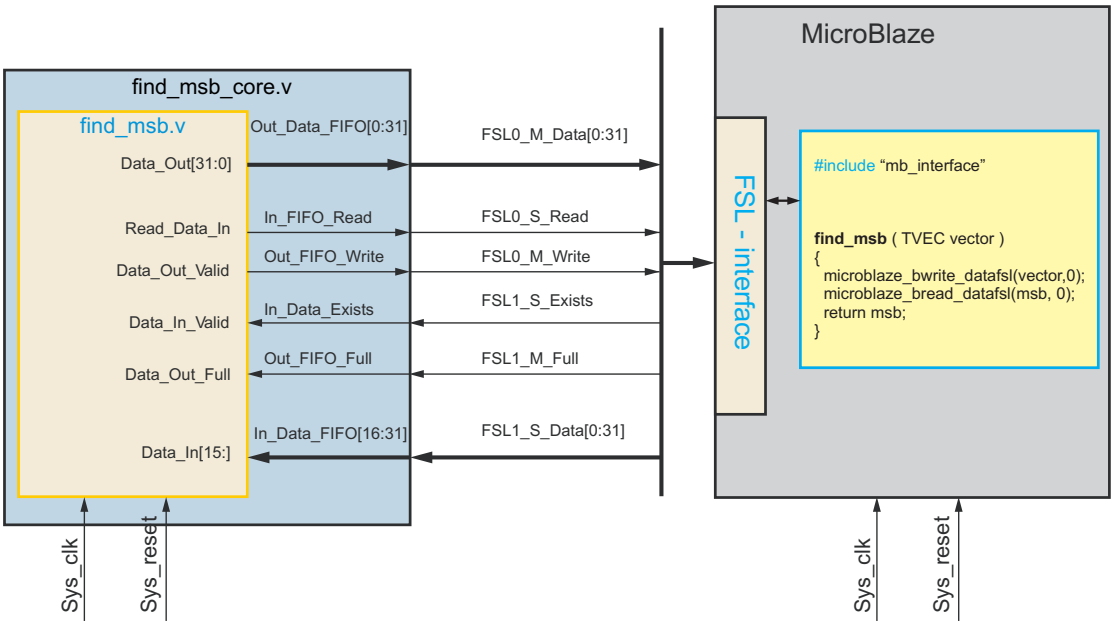


Figure 7.5.1 Co-design of the *find_msb( )* function.

```
#include "mb_interface"
...

UBYTE find_msb( TVEC vector)
{
  #ifdefine find_msb_core

  UBYTE msb;

   microblaze_bwrite_datafsl(vector,0);
   microblaze_bread_datafsl(msb,0);
   return msb;

  #endif //find_msb_core
}
```

Code segment 7.5.1 Software interface for the find_msb_core.

Following the same measurement technique the execution rate is obtained for the *find_msb( )* using the find_msb_core (Figure 7.5.2). The time that takes to execute the hardware algorithm and the communication interface is 492ns, which is a constant execution time, since it follows the same behaviour as the software algorithm.

These results imply a 16.89% speed improvement over the 8 bit software algorithm and a 43.31% speed improvement over the 32 bit software algorithm. This last result proves significant performance acceleration thanks to the FSL communication bus which has been designed by Xilinx as a fully dedicated bus for this type of applications.

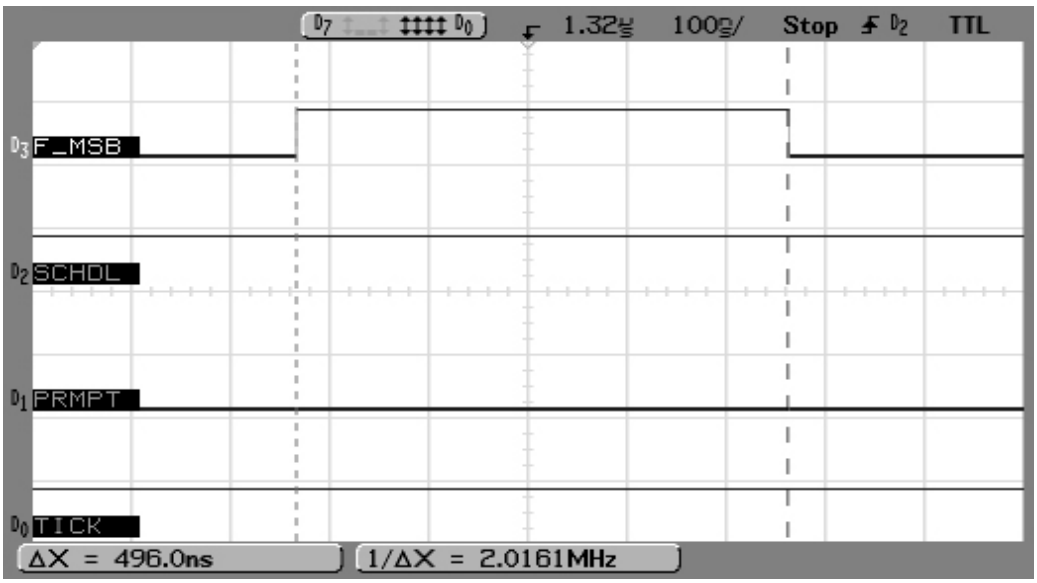


Figure 7.5.2 Execution time of *find_msb( )* for 32bits with find_msb_core.

The effectiveness of hardware implementations has been proven since the addition of the coprocessor units to microprocessor. Hardware performs faster than software, since software applications and execution is built on top of the hardware.

The subject of discussion in this section is the efficiency of the link between the processor and the coprocessor units. In previous co-design attempts, the problem has been the communication between these two units, the hardware acceleration gained was lost in the communication process in software.

Xilinx FSL can achieve speeds up to 300MB/sec, providing a full dedicated bus to link processors with coprocessor units and IP cores. The experiments conducted in this section have proven the effectiveness of moving software functions into hardware bounding them with a FSL bus. The results show a dramatic performance improvement when execution 32bit applications, due to the fact that FSL has been design with the 32bit market in mind.

Functions are relatively easy to move into hardware. This is in part due to the nature of software functions. Simple functions take data, performed the required computations and give back a result. This is in fact, the operational principle of coprocessors and DSPs. With this in mind, MicroBlaze RISC architecture is no longer limited by its instruction set, since expensive computations can be moved into hardware. Furthermore, tasks can perform heavy and complex computation just by attaching a coprocessing unit or IP core to the processor. With these results, is inevitable to ask: What about moving HARTEXμ kernel primitives and services with the shared data structures into hardware? Well let's just say: "that is a complete different story".

7.6 HARTEXμ MULTI-CORE KERNEL

HARTEXμ kernel primitives and services operate on shared date structures. These data structures can be accessed by the different subsystems in the kernel, including the task. Making the proper partition of primitives and services between hardware and software has been proven not an easy task.

Several counseling hours and analysis have been put into this problem; every possible scenario at the end presents the same problems: how to handle shared data structures between the hardware and software implementations? Which primitives and services should be placed in hardware and which ones should remain in software?

Implementing a number of primitives and services in hardware implies moving data structures also in to hardware, which need a synchronization mechanism to keep these structures updated with the ones remaining in software that can introduce a communication overhead.

Based on the analysis of the current embedded system technology presented in this thesis, it has been realized that sometimes the simplest solution is the best: "if the kernel operation introduces an overhead into the tasks execution, why not take out the kernel?"

This solution presents a "bold" concept, however it is completely feasible. High density of gates on FPGAs has provided the opportunity to have several processors running in the same silicon area at the same time. HARTEXμ kernel can be placed in one of the processors and task execution can be allocated in the rest, creating a network. Studies have been conducted on how to connect the network of several MicroBlaze cores[53]. Since the kernel primitives, services and data structures are centralized in one of the processors; the best network topology is the star connection.

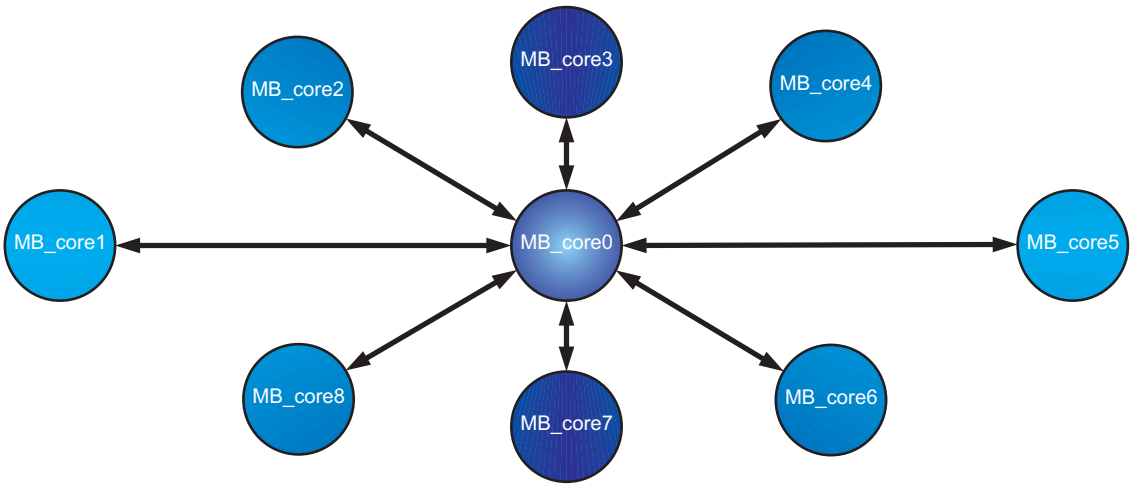


Figure 7.6.1 MicroBlaze star network topology.

In a star network topology each node is connected to a central node, in this case HARTEXμ kernel. The drawback of this configuration is that the whole system is totally dependent on the central node, in case this one fails the entire system "crashes". This weakness is not very relevant in an embedded system where all the nodes are located inside the same chip.

The advantage of this system is the massive parallelism achieved by the cores; several tasks can be running concurrently under one kernel, while a virtual "kernel-free" execution environment is provided to the tasks.

Implementation issues shall be addressed for this architecture. This solution introduces a new dimension of complexity into the kernel operation environment: computations in space. It is well known the main resource handled by kernels is time. Running tasks concurrently adds the space allocation issue, in which core should the task be executed? Other issues that can be foreseen in this architecture are:

- Communication between the core hosting the kernel and its subsystems and the cores hosting the tasks.
- Synchronization mechanisms to execute kernel primitives.
- A set of primitives and services for space allocation of tasks.
- Access to the centralized shared data structures.

7.6.1 MATERIALIZING HARTEXμ MULTI-CORE WITH MICROBLAZE

Taken advantage of the Xilinx FPGAs and MicroBlaze soft-core processor previously discussed in this thesis, an implementation guideline is presented in this subsection for the HARTEXμ multi-core kernel architecture.

One MicroBlaze core shall be fully dedicated to execute the kernel primitives, services and functions. The shared data structures can be placed on external memory where all the task cores can access them through an arbitrated bus. For this purpose Xilinx offers the OPB interface.

Task cores are built with local RAM where applications can be allocated. For fast access to the kernel primitives and services each core is connect to the kernel's core trough a couple of FSL interfaces, since it has been demonstrated that this interface achieves high speed data transfer rates and is ideal for critical inter-processor communication[33][53]. An OPB interface can be established to access the kernel data structures.

All external events are handled by the core hosting HARTEXμ. The kernel core and task cores are equipped with timers that can generate local processor interrupts for periodic task execution. Every task core can have access to coprocessing units or IP core, the only limitation is that each MicroBlaze core can only support up to sixteen FSL interfaces.

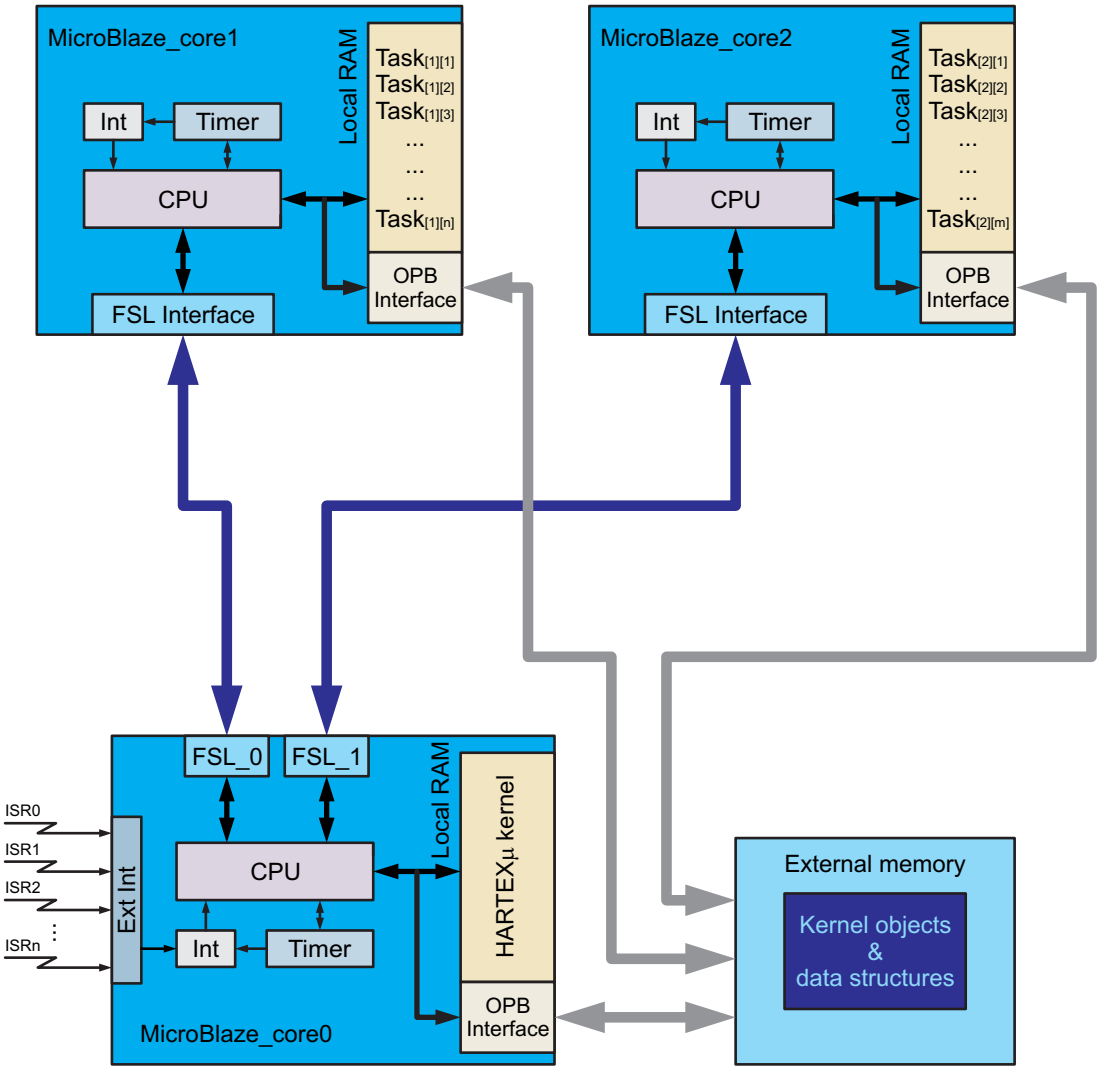


Figure 7.6.2 HARTEXμ multi-core architecture.

This architecture condenses and exploits the features of MicroBlaze, the Spartan-3 family of FPGAs and the experimental results obtained from the hardware/software co-design implementations. The HARTEXμ multi-core kernel architecture requires further experimental testing and verifications.

7.7 SUMMARY

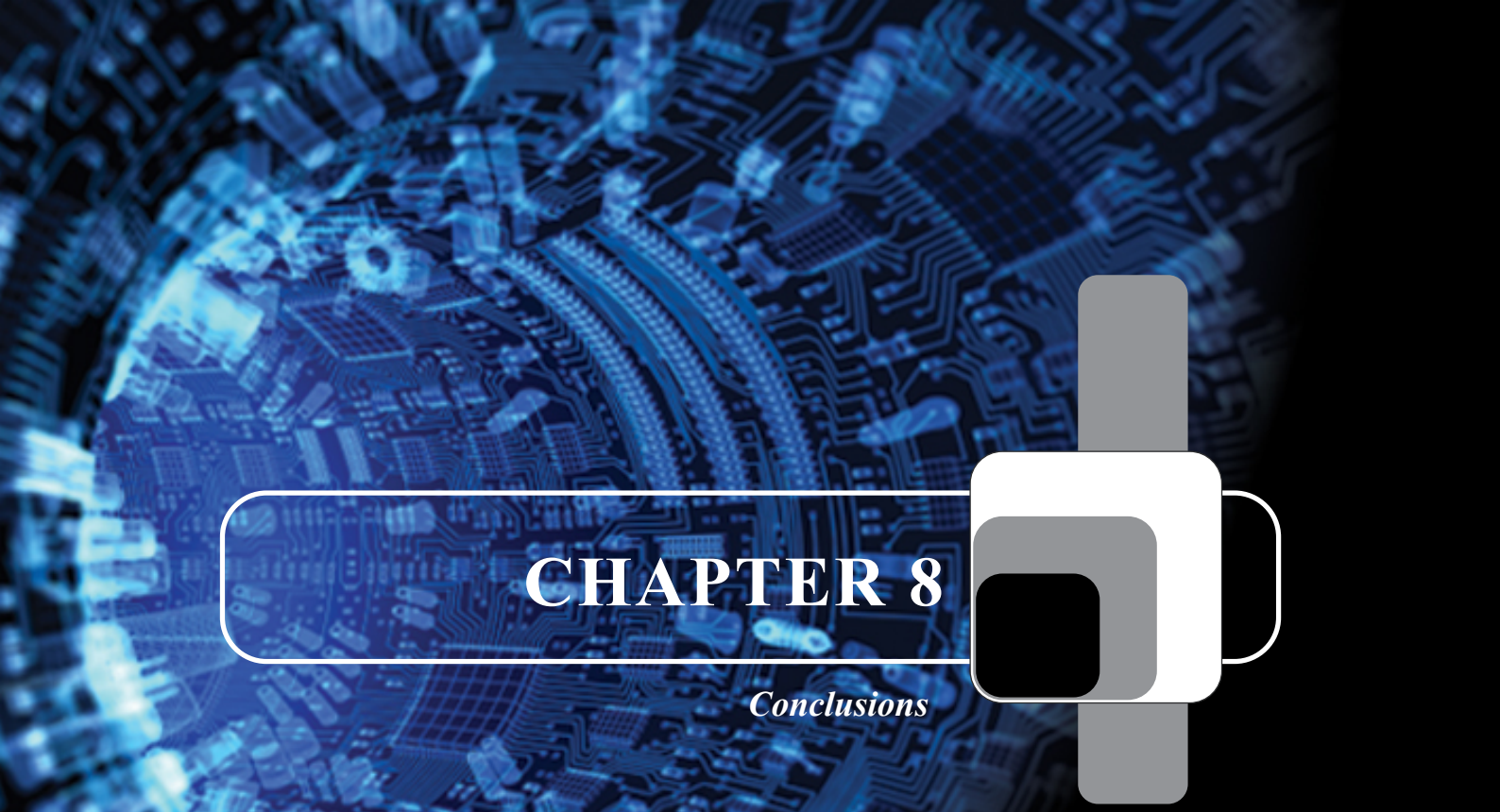
This chapter has presented the materialization of all the concepts, theories and devices presented throughout this thesis. HARTEXμ has been successfully ported to the MicroBlaze soft-core processor platform with use of the Spartan-3 FPGA family by Xilinx. The initial porting was made with the Spartan-3E starter kit board, which is suitable platform to get

familiarized with MicroBlaze. For analysis performance and testing, Spartan-3A DSP 1800 platform was used. This board presents higher clock frequency operation and more on-board general purpose I/O connectors.

The HDL cores are target specific, unlike the software that can be used in any board of the Spartan-3 family, as long as it makes use of the labels created by the libraries. The software framework or adaptation layer for HARTEXμ presented in this chapter can be used with any of the Spartan-3 boards.

Besides introducing HARTEXμ into the soft-core processors world, it has been demonstrated how the kernel performance can be accelerated using a hardware/software co-design approach. In this case one the functions invoke by the kernel has been translated to HDL and implemented in hardware. It can be concluded with this experiment that a dramatic increase in performance can be achieved with the current FPGA architecture offered by Xilinx.

Based on the hardware/software co-design approach and the research presented in this thesis, an HARTEXμ multi-core solution is proposed. The main advantage is the task execution parallelism and the "kernel-free" execution environment, some initial issues have been addressed for this architecture, extensive experimentation and testing will be required.



CHAPTER 8

Conclusions

8.1 SUMMARY

This thesis has presented a guide to hardware/software co-design of embedded systems, in particular taking HARTEXμ kernel and exploring the possibilities offered by soft-core processors. As shown by Makimoto's digital wave, the current embedded technology has driven the design of embedded systems into a paradigm shift. Taking advantage of the flexibility and compatibility of programmable logic devices, several paradigms have been proposed and developed over the years without any of them being the final stabilizing solution to the alternating digital wave.

Hardware acceleration through co-design implantations and reconfigurable computing systems are the most attractive paradigms that can achieve true computational parallelism. This project has focused on hardware acceleration experiments towards a HARTEXμ hardware/software solution taking some of the reconfigurable computing principles.

An extensive research has been presented about the current stage of programmable logic devices, their capabilities and integration scale. PALs, PLAs and GALs present a low integration scale of gates, falling short for the purposes of this project. CPLDs have enough

logic components to implement coprocessing units or small processors. FPGAs provide the proper scale integration to implement several processor cores, coprocessing units and IP cores. ASIC technology has been also considered, however due to the fixed nature of this device it is not well suited for this project.

Focusing the attention on FPGA, several programming options have been presented. HDLs are the native programming languages for these devices, in either VHDL or Verilog. Both languages can achieve the same goal; however VHDL is fully hardware-oriented language while Verilog is a C-friendly language making easy the transition for software into hardware. SDLs have been also discussed briefly. The main advantage of these languages (such as SystemC or ImpulseC) is the friendly environment where the programmer does not need to learn HDL and the tool takes care of the proper translation from C to HDL.

For this project HDLs are better suited. SDL are appropriated when implementing function with heavy computations into hardware. One of the goals is to implement the HARTEXμ kernel subsystem which do not behave entirely as functions, translating these subsystem into hardware is not a straight-forward task.

An entire processor can be described in HDL and synthesize into an FPGA, this processor are known as soft-core processor. A general introduction and features overview of the most noticeable soft-core processors has been provided in this thesis. Detail research has been conducted on the MicroBlaze soft-core processor, which is a 32 bit RISC architecture, highly configurable and customizable, specially designed by Xilinx for the FPGA devices.

MicroBlaze presents a distinct feature which makes it the ideal candidate for this project: a fully dedicated bus for custom logic. The FSL interface can connect any coprocessing unit or IP core to the MicroBlaze processor, avoiding the hassle of restrictive integration of custom instructions to the ALU. The FSL interface unlike custom instructions is easy to integrate in software applications.

Xilinx provides a couple of families where a MicroBlaze core can be synthesized: Virtex and Spartan. The basic difference between these two families is that Virtex includes a hard-core IBM PowerPC processor. This project has tipped the scale in favor of Spartan devices, since it only explores the hardware/software co-design of HARTEXμ and its capabilities on MicroBlaze.

This project makes use of two boards of the Spatan-3 family: Spartan-3E starter kit, which is used to get familiarized with the MicroBlaze embedded development environment, and Spartan-3A DSP 1800 which is used to analyze the performance of HARTEXμ running in MicroBlaze and co-design of the kernel subsystems. Both boards are able to run HARTEXμ, however Spartan-3A offers higher frequency clock operation and more general purpose I/O.

It is important to keep in mind that the tool generated HDL code for the MicroBlaze core is target specific, so a core configured for Spartan-3E can not be synthesized in a Spartan-3A board; however from the software point of view it is virtually transparent.

One of the strongest points of MicroBlaze is the environment tool support provided by Xilinx. It is a comprehensible framework in which a co-design process is possible and configuration changes are easily reflected on any of the hardware/software design flows. The primary tool offered by Xilinx is EDK/XPS, where hardware can be configure and built using the BSB wizard, while software is handled by the SDK. In order to download the implementation into the FPGA, the embedded environment makes use of ISE. EDK also offers a wizard to create and configure a coprocessing unit or IP core and attached to the MicroBlaze processor through FSL or OPB interface.

The experimentation phase conducted with HARTEXμ and MicroBlaze during this research has lead to the following results (Chapter seven):

- A software MicroBlaze framework on which HARTEXμ kernel can be executed.
- It has been demonstrated the efficiency of the FSL interface for hardware/software co-design, ergo hardware acceleration of kernel functions has been achieved.
- A conceptual model of a multi-core implementation of HARTEXμ kernel.

These contributions probe the possibility of hardware/software co-design; however this is just the beginning stage towards a new paradigm shift in embedded systems.

8.2 FUTURE WORK

The direction of the hardware/software co-design set by this project is clear: computational parallelism of HARTEXμ kernel throughout a distributed network of processors inside the same FPGA silicon area. Further research has to be conducted on this field, incorporating some of the parallelism concepts of reconfigurable computing. This research has triggered interesting concepts for kernel theory and implementation. How to handle the space computation dimension introduced by parallel execution of tasks among several processors? What sort of primitives and services will be required?

A Virtex board is the most likely to be used to carry on this research, the processor hosting the HARTEXμ can the implemented in the PowerPC hard-core processor offered by this family of boards and the tasks can be allocated in several MicroBlaze soft-core processors, which can ultimately lead to the implementation of an ASIC task processor with a fix hard-core processor containing HARTEXμ kernel and an embedded development environment which configures the number of soft-core processors according to the number of tasks.

REFERENCES

[1] Qing Li, Caroline Yao. *Real-time concepts for embedded systems*. CMPBooks 2003.

[2] Enoch O. Hwang. *Digital logic and microprocessor design with VHDL*. Thomson 2004.

[3] Vijay K. Madisetti. Chonlameth Arpikanondt. *A platform-centric approach to system-on-chip (SoC) design*. Springer 2005.

[4] Michael John Sebastian Smith. *Application-specific integrated circuit.* Addison-Wesley professional 1997.

[5] Volnei A. Pedroni. *Circuit design with VHDL*. MIT press 2004.

[6] Richard Munden. *ASIC and FPGA verification: a guide to component modeling*. Morgan Kaufmann publishers 2005.

[7] Ahmed Amine Jerraya, Wayne Wolf. *Multiprocessor System-on-chip*. Morgan Kaufmann publishers 2005.

[8] Pong P. Chu. *RTL hardware design using VHDL*. Wiley-interscience 2006.

[9] Robert Dueck. *Digital design with CPLD applications and VHDL*. Thomson 2000.

[10] C.K. Angelov, I.E. Ivanov, A. Burns. *Hartex - a safe real-time kernel for distributed computer control systems*. 2002.

[11] C.K. Angelov, J. Berthing. *Lecture Notes in Computer Science - A Jitter-Free Kernel for Hard Real-Time Systems*. Springer 2005.

[12] Krzysztof Sierszecki. *Component-based design of software for embedded systems*. MCI University of southern Denmark 2007.

[13] C.K. Angelov, K. Sierszecki. *Component-based design of software for distributed embedded systems*. University of southern Denmark 2006.

[14] K. Bondalapati, V.K. Prasanna. *Reconfigurable computing systems*. Proceedings of the IEEE, vol 90. No.7, 2002.

[15] Katherine Compton, Scott Hauck. *An introduction to reconfigurable computing*. IEEE Computer 2000.

[16] R. Hartenstein. *The von Neumann syndrome*. Stamatis Vassiliadis memorial symposium 2007.

[17] Steve Kilts. Advance *FPGA Design – Architecture, implementation, and optimization.* Wiley-interscience 2007.

[18] Jonathan Rose, Abbas El Gammal, Alberto Sangiovanni-Vincentelli. *Architecture of field-programmable gate arrays*. IEEE 1993.

[19] B. Osterloh, H. Michalik, B. Fiethe, F. Bubenhagen. *Enhancements of reconfigurable System-on-Chip Data Processing Unit for Space Application.* Second NASA/ESA Conference on Adaptive Hardware and Systems, IEEE 2007.

[20] Dave Landis. *Programmable Logic and Application Specific Integrated Circuits*. The Pennsylvania State University.

[21] Techbites interactive team. *EDA: Where electronics begin.* 2001.

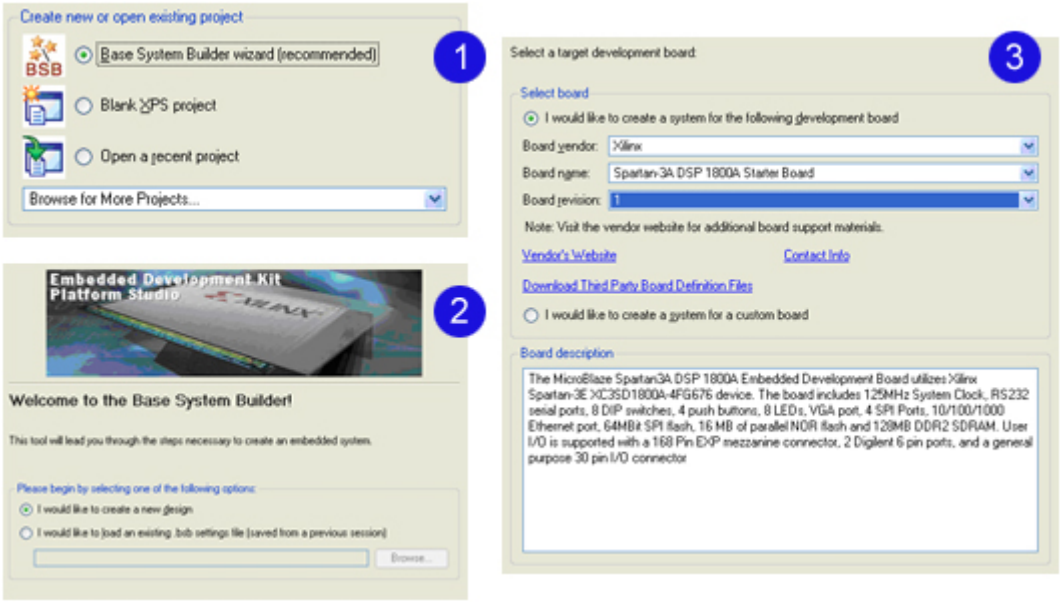
[22] Stephen Brown, Jonathan Rose. *Architecture of FPGAs and CPLDS: A tutorial*. Department of Electrical and Computer Engineering University of Toronto.
[23] B. Fiethe, H. Michalik, C. Dierker, B. Osterloh, G. Zhou. *Reconfigurable System-on-Chip Data Processing Units for Space Imaging Instruments*. IDA TU Braunschweig.
[24] Reiner Hartenstein. *A Decade of Reconfigurable Computing: a Visionary Retrospective.* University of Kaiserslautern, Germany
[25] David Pellerin, Scott Thibault. *Practical FPGA programming in C*. Printice Hall 2005.
[26] Randall Hyde. *The art of assembly language programming*. http://www.arl.wustl.edu/~lockwood/class/cs306/books/artofasm/toc.html
[27] Daniel Mattsson, Marcus Christensson. *Evaluation of synthesizable CPU cores*. Chalmers University of technology 2004.
[28] Juanjo Noguera, Rosa M. Badia. H*W/SW codesing techniques for dynamically reconfigurable architectures*. IEEE transactions on VLSI 2002.
[29] Donald E. Thomas. Jay K. Adams, Herman Schmit. *A model and methodology for hardware-software codesign.* Carnegie Mellon University. 1993.
[30] Reiner Hartenstein. Reconfigurable computing: the roadmap to a new business model - and its impact on SoC design. University of Kaiserslautern, Germany
[31] Bogdan Sbarcea, Dan Nicula. *Automatic conversion of MatLab/Simulink models to HDL models*. Dept. of Electronics and Computers, Transilvania University of Brasov.
[32] ARM microprocessor intellectual property. www.arm.com
[33] Xilinx Inc. www.xilinx.com
[34] Actel power matters. www.actel.com
[35] Altera. www.altera.com
[36] Lattice Semiconductors Corporation. www.latticesemi.com
[37] Gaisler Research. www.gaisler.com
[38] Cypress Semiconductor. www.cypress.com
[39] FPGA and structures ASIC Journal. www.fpgajournal.com
[40] The official site of the embedded development community. www.embedded.com
[41] Global news foe the creators of technology. www.eetimes.eu.
[42] Clive Maxfield. *FPGA Architectures from 'A' to 'Z'*. www.pldesignline.com
[43] Navanee Sundaramoorthy. *Put a configurable 32-bit processor in your FPGA.* Embedded system design – Europe. Issue May 2007.
[44] Rob Irwin. Making FPGAs work for embedded developers. Embedded system design – Europe. Issue June/July 2007.
[45] Barry O'Rourke, Richard Taylor. *Turbocharging your CPU with an FPGA-Programmable coprocessor.* Xilinx ESL Initiative. Issue third quarter 2006.
[46] Embedded computing design. *2007 Resource guide*. Issue August 2007.
[47] Jack Ganssle. *The embedded muse*. www.ganssle.com
[48] Open System C initiative. www.systemC.org
[49] SpecC system. http://www.cecs.uci.edu/~specc/
[50] Impulse accelerated technologies. www.impulsec.com

[51] EDK profiling user guide. *A guide to profiling in EDK*. UG448. Xilinx
[52] Peng Jiang. *HARTEXμ Kernel Hardware/Software co-design*. University of southern Denmark 2006
[53] P. Huerta, J. Castillo, J. Martinex, V. Lopez. Multi MicroBlaze system for parallel computing. Universidad Rey Juan Carlos. Spain
[54] Krzysztof Sierszecki. *HARTEXμ Kernel User Manual.* MCI University of Southern Denmark 2004.
[55] Paul Glover. *Using and creating interrupt-based systems*. Xilinx application note 773.
[56] Hans-Peter Rosinger. *Connecting customized IP to the MicroBlaze soft processor using the Fast Simplex Link Channel*. Xilinx application note 529

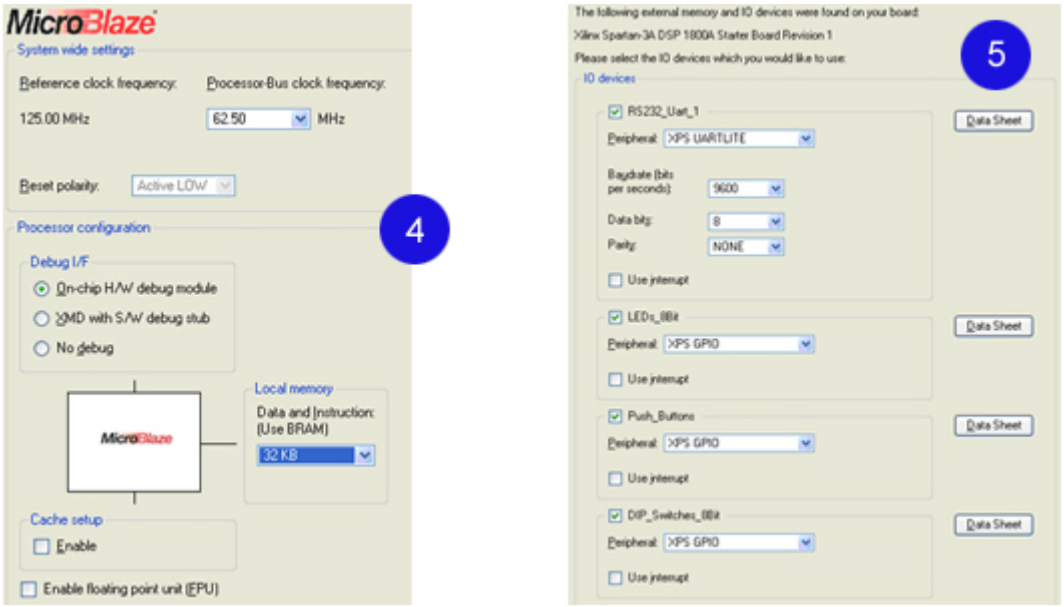
APPENDIX A

Creating a MicroBlaze core

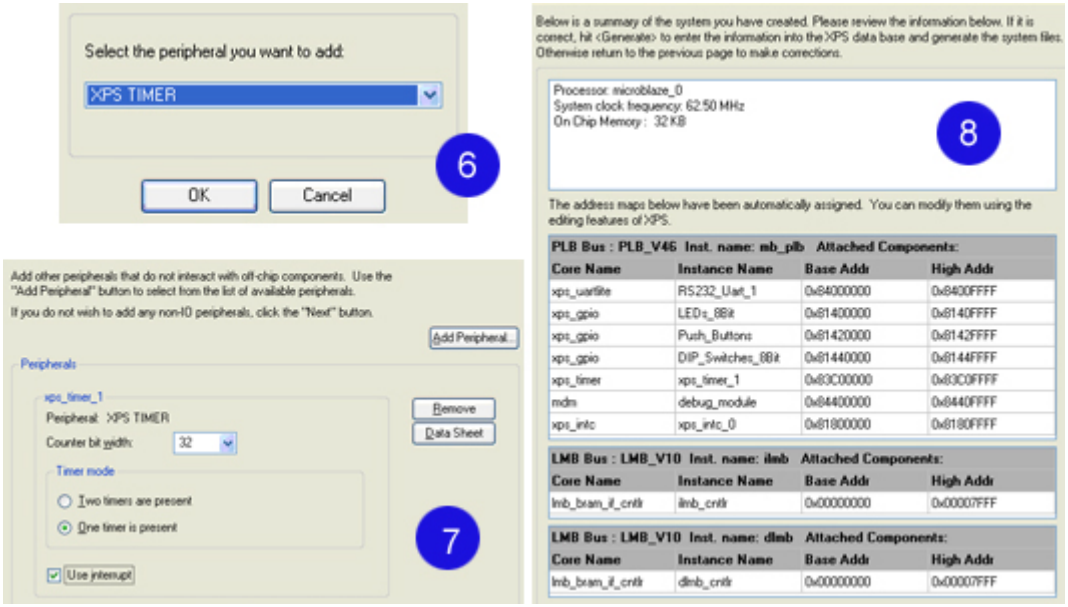
This section provides a guide to create and configure a MicroBlaze core. Open the EDK/BSB wizard (1) and select a project folder. In the welcome screen (2) select "I would like to create a new design". In the next screen select the targeted development board (3).



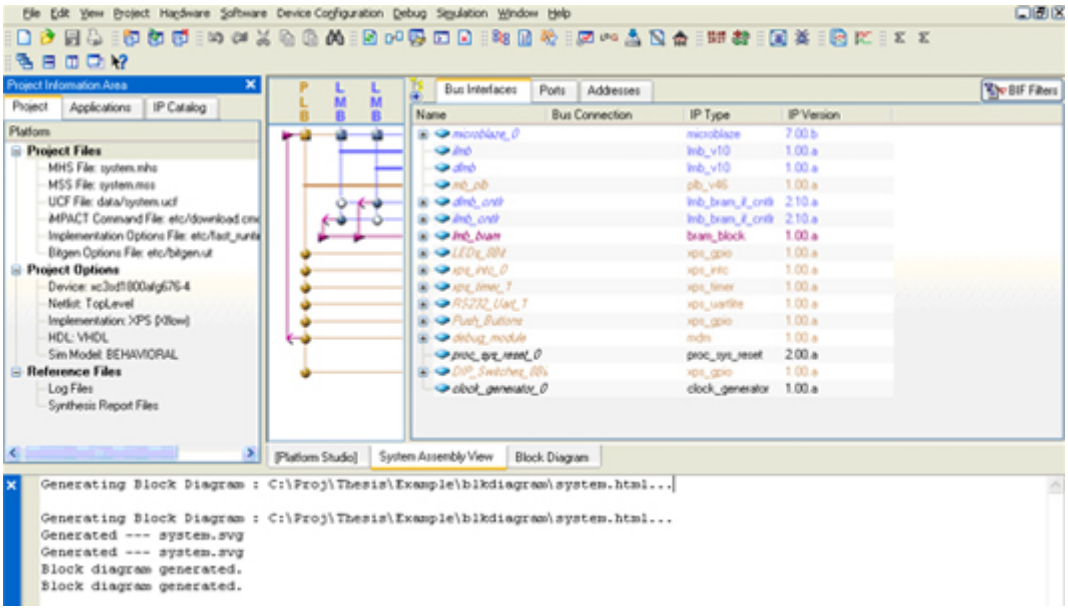
In the following screen select the required configuration for the MicroBlaze core (4).



Selected the I/O interfaces required for the specific design (5). Peripherals can be added or removed. Interrupts can be also added and associated to the peripherals.

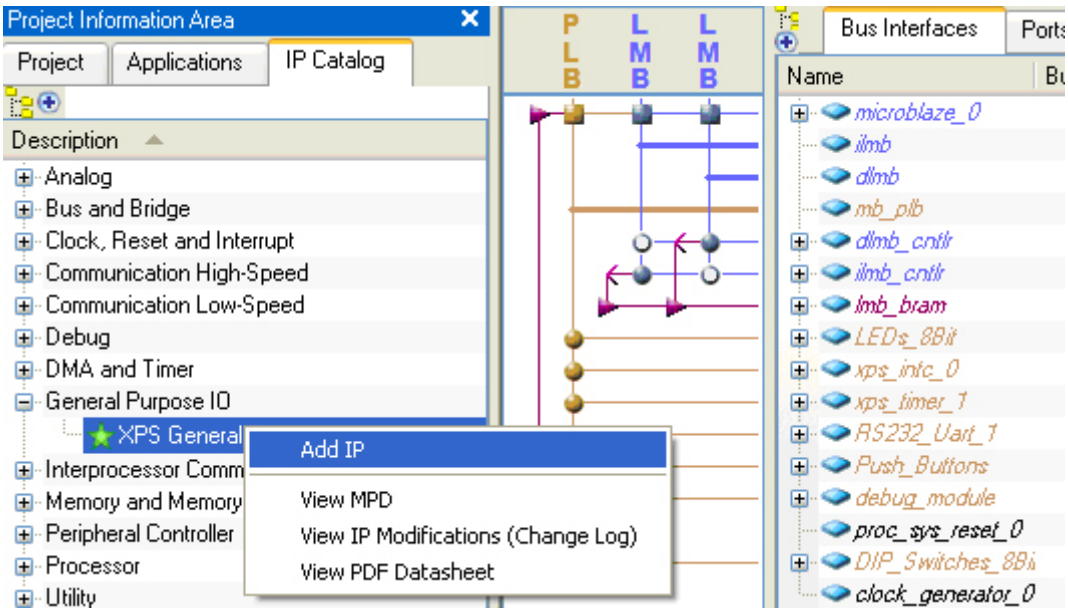


Timers can be also integrated into the core (6) and enable the corresponding interrupt (7). Once the required peripheral are configured, a screen with the core summary is shown (8), if the configuration is appropriated, press generate. EDK/XPS opens the system assembly view containing all the information regarding the core. The project information area contains three tabs: the project tab contains the MHS, MSS and UCF files. The Application tab contains all the software applications for this core, and the IP catalog tab which contains all the possible IP cores that could be implemented in MicroBlaze.

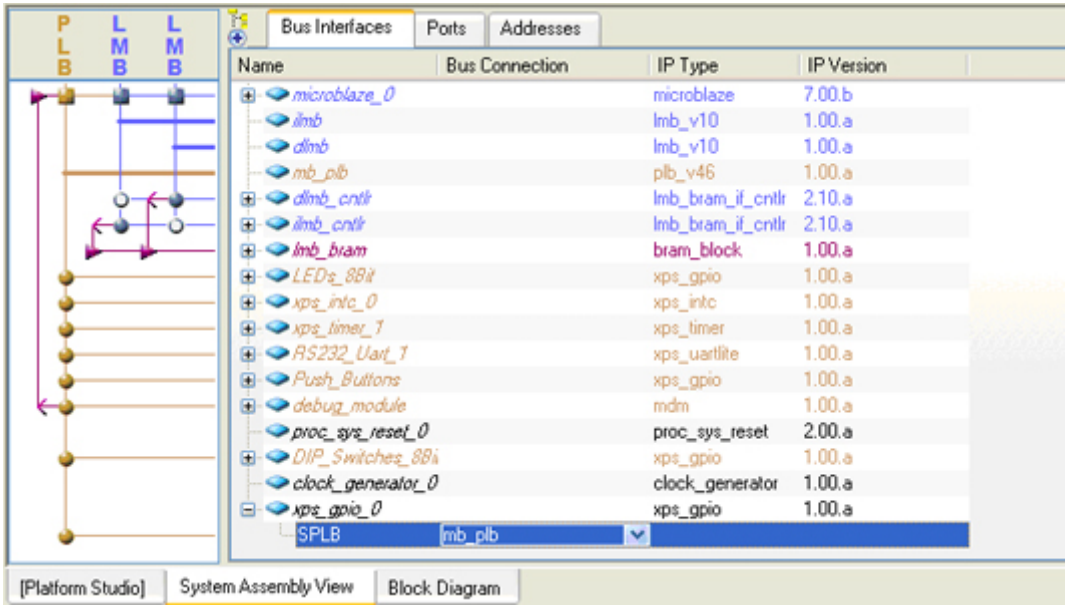


ADDING A PERIPHERAL

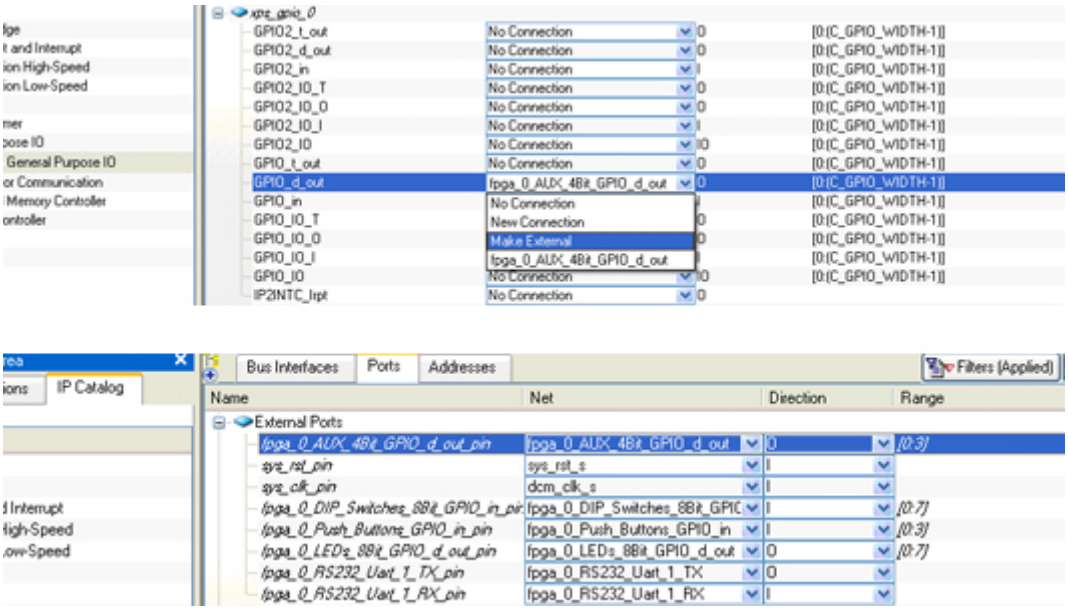
In this section and general purpose I/O port is added. Go to the IP catalog tab and select the General Purpose IO catalog, add the XPS GPIO core.



The system assembly view is updated with the new core, in this case xps_gpio_0. In the SPLB row, connect the GPIO core to the MicroBlaze bus by selecting mb_plb. Right click over the xps_gpio_0 core and click Configure IP. Configure the data width in bits, the pins direction and the number of channels supported by the GPIO core. Once these parameters are established, go to the addresses tab on the system assembly view and generate the address for the xps_gpio_0 by clicking the generate addresses button.



Go to the Ports tab on the system assembly view and expand the xps_gpio_0 and set a label for the GPIO_d_Out net and make it external. The new label should be updated in the External ports.



Once the port has been configured go to the project information area under the project tab and open the UCF file. The following code configures a 4bit output port using the SAM connector on the Spartan-3A DSP1800 board.

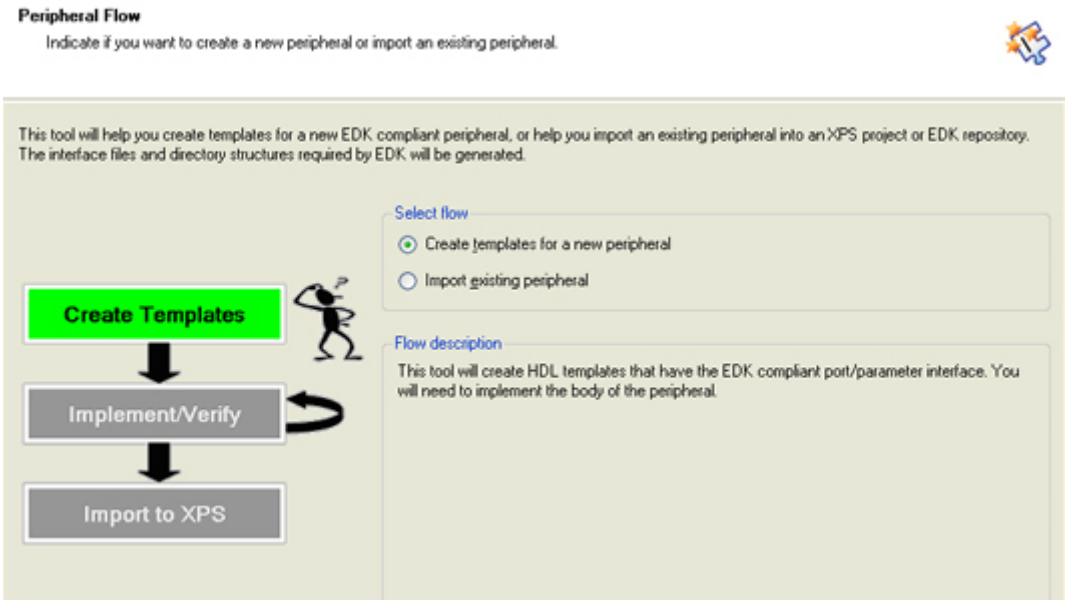
```
#### Module AUX_4Bit constraints

Net fpga_0_AUX_4Bit_GPIO_d_out_pin<0> LOC = U18;
Net fpga_0_AUX_4Bit_GPIO_d_out_pin<0> IOSTANDARD=LVTTL;
Net fpga_0_AUX_4Bit_GPIO_d_out_pin<1> LOC = Y23;
Net fpga_0_AUX_4Bit_GPIO_d_out_pin<1> IOSTANDARD=LVTTL;
Net fpga_0_AUX_4Bit_GPIO_d_out_pin<2> LOC = T20;
Net fpga_0_AUX_4Bit_GPIO_d_out_pin<2> IOSTANDARD=LVTTL;
Net fpga_0_AUX_4Bit_GPIO_d_out_pin<3> LOC = Y25;
Net fpga_0_AUX_4Bit_GPIO_d_out_pin<3> IOSTANDARD=LVTTL;
```

To generate the bitstream and netlist, go to hardware and click on Generate Bitstream. This process could take up to 30 minutes.

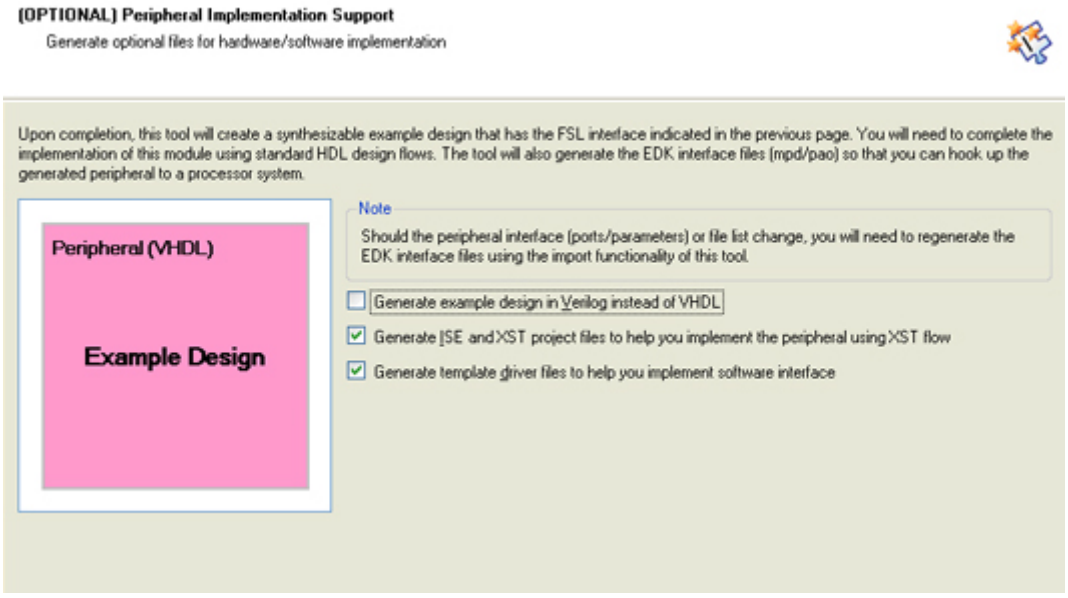
APPENDIX B

Creating an IP core

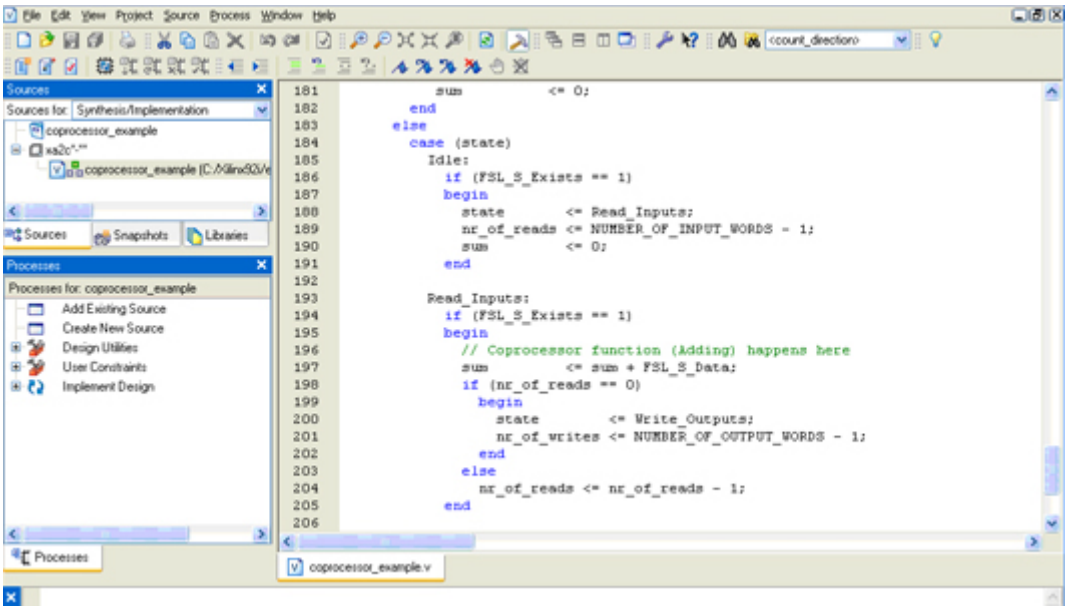
This section presents a guide to the creation of coprocessing units and IP cores. Go to the hardware menu in the EDK/XPS and click on create or import a peripheral. A screen with process flow is opened. Select create templates for new peripheral



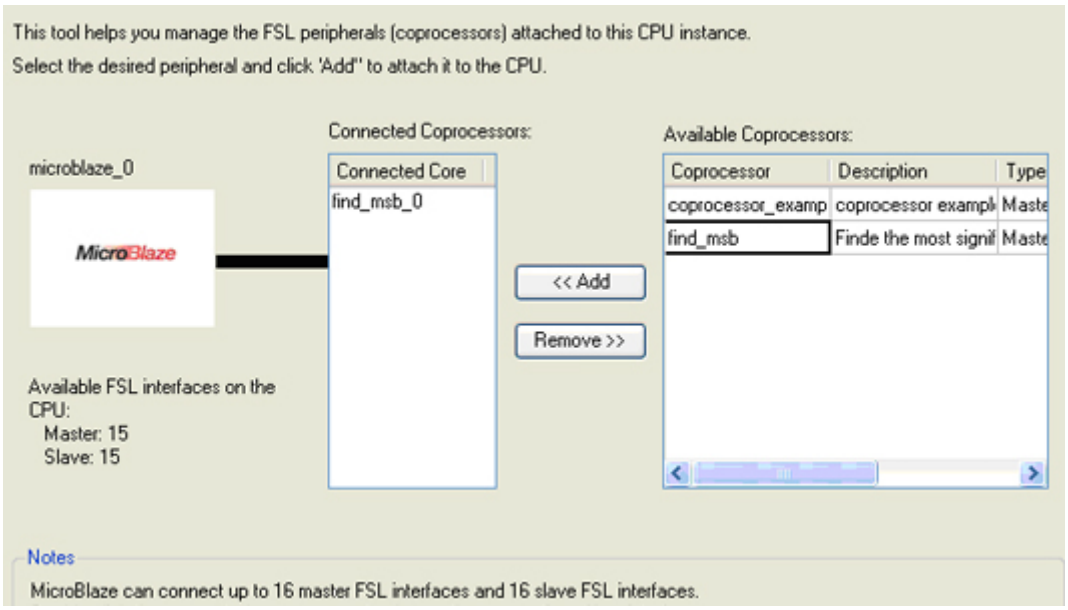
Select a working directory and a name for the new IP core. In the next screen select the type of communication bus. Select the number inputs and outputs of 32bit words.



This wizard creates the ISE templates and the drivers to implement the software interface. Open the project development file in Xilinx ISE and import the HDL file.



The HDL file contains the necessary signal interface for the selected communication protocol. Insert the HDL code containing the required application and check the syntax. Once the IP core is finished, go to the hardware menu and click on Configure Coprocessor



Select the required IP core and the system assemble view should be updated with the new core.