P-3462

MULTI LEVEL COMPUTING ARCHITECTURE FOR PARALLELISM ON FPGA USING VHDL

By

K.SRI RAM

Reg No. 0920106015

of

KUMARAGURU COLLEGE OF TECHNOLOGY

(An Autonomous Institution affiliated to Anna University of Technology, Coimbatore)

COIMBATORE - 641049

A PROJECT REPORT

Submitted to the

FACULTY OF ELECTRONICS AND COMMUNICATION ENGINEERING

In partial fulfillment of the requirements

for the award of the degree

of

MASTER OF ENGINEERING

IN

APPLIED ELECTRONICS

APRIL 2011

BONAFIDE CERTIFICATE

Certified that this project report entitled "**MULTI LEVEL COMPUTING ARCHITECTURE FOR PARALLELISM ON FPGA USING VHDL**" is the bonafide work of Mr. **K. Sri Ram [Reg. No. 0920106015]** who carried out the research under my supervision. Certified further that, to the best of my knowledge the work reported herein does not form part of any other project or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

6/4/11.

Project Guide

Ms. M. Alagu Meenaakshi

6/4/11

Head of the Department

Dr. (Ms.) Rajeswari Mariappan

The candidate with university Register No. 0920106015 was examined by us in the project viva-voce examination held on ...21.04.2011.....

7/4/11

Internal Examiner

External Examiner

ACKNOWLEDGEMENT

A project of this nature needs co-operation and support from many for successful completion. In this regards, we are fortunate to express our heartfelt thanks to **Padmabhusan Arutselvar Dr.N.Mahalingam B.Sc.,F.I.E.,** Chairman and Co-Chairman **Dr.B.K.Krishnaraj Vanavarayar B.Com,B.L.,** for providing necessary facilities throughout the course.

I would like to express our thanks and appreciation to the many people who have contributed to the successful completion of this project. First I thank **Dr.J.Shanmugam Ph.D,** Director, for providing me an opportunity to carry out this project work.

I would like to thank **Dr.S.Ramachandran Ph.D,** Principal, who gave his continual support and opportunity for completing the project work successfully.

I would like to thank **Dr.Rajeswari Mariappan Ph.D,** Prof of Head, Department of Electronics and Communication Engineering, who gave her continual support for us throughout the course of study.

I would like to thank **Ms. M. Alagu Meenaakshi M.E. (Ph.D)**, Asst. Professor, Project guide for her technical guidance, constructive criticism and many valuable suggestions provided throughout the project work.

My heartfelt thanks to **Ms.R.Latha M.E. (Ph.D),** Associate Professor, Project coordinator, for her contribution and innovative ideas at various stages of the project to successfully complete this work.

I express my sincere gratitude to my family members, friends and to all my staff members of Electronics and Communication Engineering Department for their support throughout the course of my project.

ABSTRACT

The **Multi-Level Computing Architecture (MLCA)** is a novel parallel System-on-a-Chip architecture targeted for multimedia applications. Here the proposed novel architecture is used to efficiently exploit coarse-grain parallelism on FPGAs. The central component of the MLCA is its Control Processor (CP), which is analogous to an out-of-order scheduling unit of a superscalar Processor. The CP schedules coarse-grain units of computation, or tasks, onto Processing Units (PUs). In this paper, the control processors are implemented and demonstrate the scalability of the MLCA for applications. An 8-PU MLCA system is designed, tested and evaluated. The evaluation indicates that our CP design poses no bottlenecks to performance and has little overhead in terms of resource usage up to 8 processing units.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVATIONS

MLCA	--	Multi Level Computing Architecture
URF	--	Universal Register File
CRF	--	Control Register File
ALU	--	Arithmetic and Logic Unit
HDL	--	Hardware Description Language
FPGA	--	Field Programmable Gate Array
CP	--	Control Processor
PHYRF	--	Physical Register File
PU	--	Processing Unit
TID	--	Task Instruction Decoder
TI	--	Task Instruction
SOC	--	Systems On a Chip

CHAPTER - 1

INTRODUCTION

In the recent decades, two forces have been driving the increase in performance of processors: the advances in VLSI technology and microarchitectural enhancements. Processor performance was improved through clock speed increase and exploitation of instruction-level parallelism. Transistor counts on a die are still increasing, and will likely continue in foreseeable future. However, recent attempts of investing transistor count to further increase single-core performance have brought diminishing returns. In response, architects are building chips with multiple energy-efficient processing cores instead of investing the whole transistor count into a single complex and power-inefficient core. This brings many challenges, mainly because the shift towards multiprocessing requires new programming models and automatic parallelization techniques, in order to make the transition easier for developers.

At the same time, embedded systems are becoming ubiquitous. They are used in a myriad of domains, including video processing, networking, home entertainment, gaming, and wireless processing. While in the past, consumer embedded systems were mostly simple micro-controllers, nowadays it is commonplace for embedded systems to look more like high-performance computing devices. Modern embedded systems are designed as systems-on-a-chip (SOCs) that incorporate into a single chip, multiple programmable cores ranging from processors to custom designed accelerators. This paradigm allows the reuse of pre-designed cores, simplifying the design of billion-transistor chips

. In the past few years, parallel-programmable SOCs (PPSOCs) have become a dominant SOC architecture. Several commercial PPSOCs exist today, including the Daytona chip by Bell, the PCx series by picoChip, and Viper by Phillips.

Similar to the programming of general-purpose many-core chips, programming of embedded architectures such as PPSOCs is still a significant challenge. To utilize the raw processing power of these parallel architectures, coarse-grain parallelism in applications

has to be exploited. Developers have to partition applications into coarse grain units of computation and also take care of synchronization and communication. Inevitably, these tasks increase the time and cost of application development.

Another challenge is translating applications written in traditional software programming languages into efficient FPGA hardware has been met with considerable interest in recent past. Key to this challenge is the discovery, extraction and mapping of the parallelism in an application into efficient hardware. The key to addressing this challenge is the decoupling of computation on the one hand and synchronization and scheduling on the other. Such decoupling allows designers to reason about the performance of computation units and to better determine which computation units should map onto soft processors and which to dedicated hardware.

To this end, the novel architecture called the Multi-Level Computing Architecture (MLCA), as a template for efficiently exploiting application parallelism on FPGAs is used. The MLCA consists of two levels. At the lower level is a set of processing units (PUs), which can be programmable processors or custom FPGA accelerators. The PUs executes coarse-grain computation units called *tasks*. At the upper level, a dedicated Control Processor (CP) automatically extracts parallelism among tasks, synchronizes and schedules the tasks on the PUs. It does so using techniques that are similar to ones used in superscalar processors to extract and execute parallelism among instructions, including register renaming and out-of-order execution.

The MLCA completely decouples computations from synchronization and scheduling, while being able to support applications with complex control flow and data sharing. All synchronization and scheduling is performed by the upper level CP, which is logically and physically decoupled from the PUs. Tasks contain only computations; they have no synchronization code. Once a task is scheduled, it runs to completion. Thus, designers can focus on the performance of individual tasks and can easily determine execution of critical ones. These critical tasks can be accelerated using standard synthesis tools, such as Altera's C2H or Xilinix's AccelDSP.

Furthermore, the CP itself can be parameterized and its parameters can be customized to match the complexity of a single application or a set of applications. Thus, the MLCA presents an intuitive approach for designers to parallelize applications and to further accelerate them on FPGAs. The CP is a central component of the MLCA and hence may become a performance bottleneck of the system.

CHAPTER – 2

BACKGROUND

In this chapter, the background material relevant to the work is reviewed. Section 2.1 provides an overview of the MLCA architecture and corresponding programming model. It also describes the main benefits of the architecture. Section 2.2 presents a summary of the FPGA technology.

2.1 MLCA

The **Multi-Level Computing Architecture**, commonly known as the MLCA, is a multi- processor system-on-chip computing architecture. As such, the MLCA is template architecture: it is a minimal specification of an inter processor work-sharing and communication model, with an accompanying instruction set. This allows instances of the MLCA to be specialized for their specific application. The number and type of processing units can be varied to suit application requirements, as can the size and layout of memory. This customizability allows system designers to maximize efficiency by including only the hardware that their application needs.

The MLCA architecture is novel by its extension of superscalar techniques developed for instructions, which are fine-grain computations, to tasks, which are coarse-grain computations. The CP, the upper level of the hierarchy, uses similar techniques to those employed in high-performance processors such as parallel execution, out-of-order execution and register renaming. Similar to a superscalar processor, the MLCA can dynamically extract parallelism among tasks. Moreover, in MLCA, instruction level parallelism can be extracted within the PUs.

The MLCA is a generic architecture. It does not specify the on-chip arrangement of PUs, nor any type of interconnect between the PUs. Different implementations are possible, such as buses, cross-bars or on-chip packet-switched networks. In addition, as inter-task dependencies are enforced by the CP, there is no need to assume a particular memory hierarchy. There can be a single memory shared by all PUs, or the memory can be distributed into smaller memories each assigned to a PU.

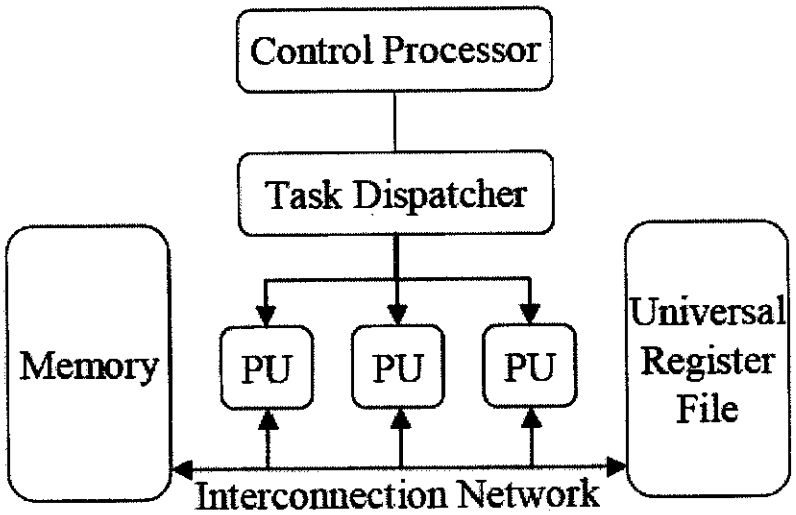
The basic unit of computation on the MLCA is a task. Tasks are a coarse-grained unit - they can perform complex operations, and generally have execution times ranging from several hundred cycles to tens of thousands of cycles. Tasks are analogous to (and are often implemented by) functions in a high-level programming language.

A minimal MLCA configuration includes one or more processing units (PUs), a universal register (URF), and a single control processor. PUs executes all task bodies, and can be general-purpose processors (such as PowerPC or ARM cores) or they can be special-purpose units (such as FPGAs or DSPs). A single MLCA system may have a heterogeneous set of PUs. The URF contains a set of registers, which can be read and written by tasks. The MLCA architecture does not specify any particular structure for the URF - registers may be fixed or variable length, however partial register access is not possible.

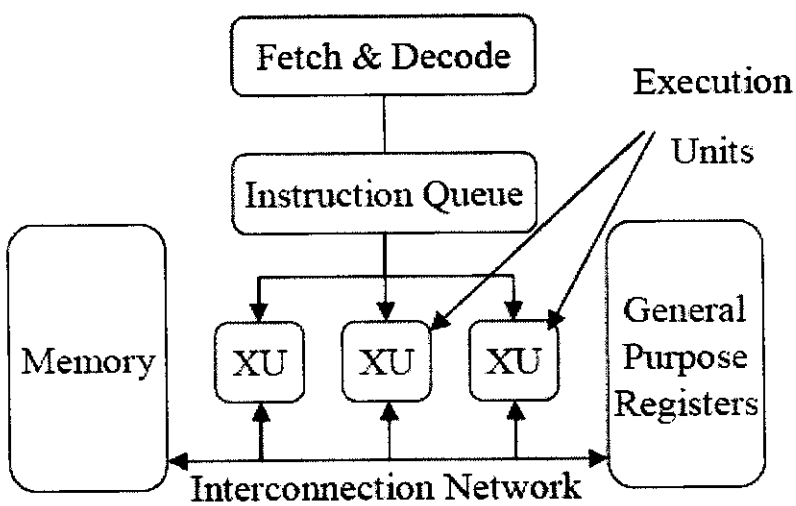
A task must read or write a URF register in its entirety. The control processor executes a control program, which consists of a sequence of control flow and task instructions. Each task instruction specifies a task to be executed as well as an explicit listing of the URF registers that it will read and write.

Figure 2.1(a) shows a block diagram of the MLCA architecture, and Figure 2.1(b) shows a diagram for a traditional superscalar processor. It can be seen that the two are virtually identical. The principal difference is the granularity of computation - the MLCA executes tasks in the same way that a superscalar processor executes instructions. A task instruction is considered ready to execute when all prior writes to its input registers have completed.

When a task is ready to execute it is passed to the control processor's task scheduling unit, which dispatches it to a PU. As in a superscalar processor, the control processor can perform URF register renaming to break false data dependencies, and it is capable of out-of-order task execution. Aside from the URF, the memory architecture of the MLCA is intentionally undefined. It could incorporate a global memory that is shared between all PUs, local memory for each PU, or a combination of both global and local memories. Likewise, interconnect between the PUs, URF and memory could be a shared bus, a hierarchical bus, or a crossbar.



(a) MLCA.



(b) Superscalar processor.

Figure 2.1 Comparison between (a) MLCA and (b) superscalar processor.

The MLCA generally have tight constraints upon cost, performance and power consumption. Leaving details such as the number and type of PUs, and the memory and bus architectures undefined allows the MLCA to be customized according to the specific

requirements of each application to which it is applied. For the purposes of this work we use an MLCA instance with 32-bit fixed-length URF registers, and a large global shared memory.

The novelty of the MLCA stems from the fact that the upper level of the hierarchy supports parallel execution of tasks, using the same techniques used in superscalar processors, such as register renaming and out-of-order execution. This leverages existing processor technology to exploit task-level parallelism across PUs, in addition to possible instruction-level parallelism within each task.

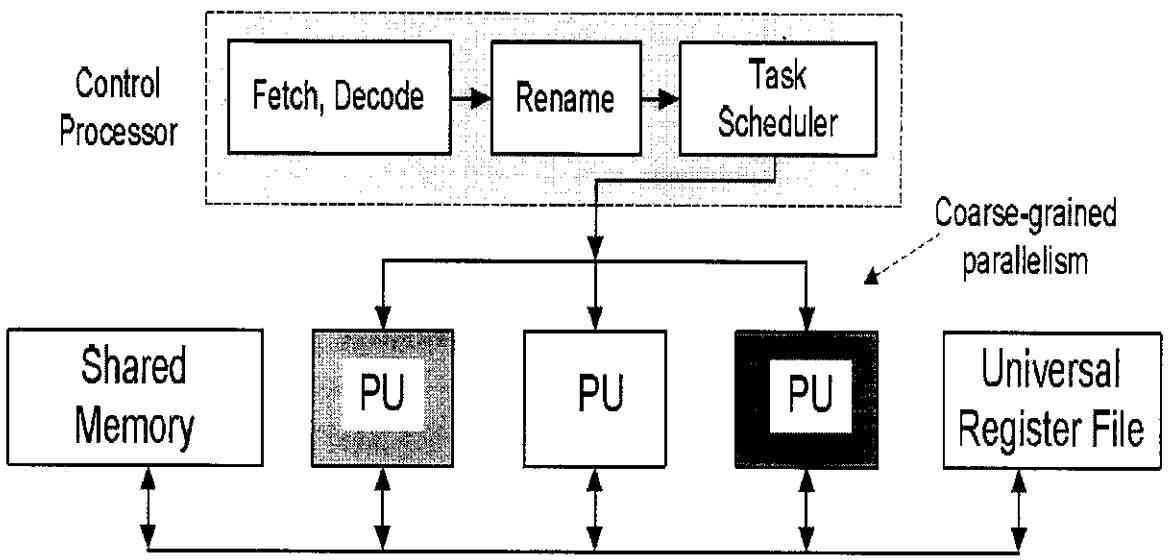


Figure 2.2 High-level organization of the MLCA

The overall organization of the MLCA is shown in Figure 2.2. The lower level consists of multiple *processing units* (PUs), and the upper level of a controller that automatically exploits parallelism among coarse-grain units of computation, or *tasks*. A PU can be a full-fledged processor core, a DSP, a block of FPGA, or any other type of programmable hardware. The top-level controller consists of a *control processor* (CP), a *task*

dispatcher (TD), and a *universal register file* (URF). A dedicated interconnection network links the PUs to the URF and to shared memory.

The PUs executes coarse-grain units of computation that are referred to as *tasks*. The top-level consists of a *Control Processor* (CP) and a *Universal Register File* (URF). The CP fetches *task-instructions* (TIs) that each represents an invocation of a task. The CP decodes each task-instruction and then schedules it to a PU where the corresponding task gets executed. Each TI (and the corresponding task invocation) takes its inputs from registers in the URF and deposits its outputs also to registers in the URF.

The novelty of the MLCA is that the CP uses techniques such as register renaming, out of- order-execution (OoOE), speculation and multiple-issue to extract parallelism among tasks. In this respect, it is similar to how a superscalar processor extracts parallelism among instructions. The MLCA gives rise to a simple and intuitive coarse-grain data-flow programming model.

Similar to the architecture, it consists of a set of *task functions* that each represents a task and a top-level *control program* that consists of task instructions (with URF inputs and outputs). The control program in itself is a sequential program and all parallelism among the tasks is extracted by the CP. In addition to TIs, the control program contains CP instructions that support control flow. These instructions access special *control registers* (CRs) in a Control Register File (CRF) and are executed directly by the CP.

2.2 OUT-OF-ORDER EXECUTION

Out-of-order execution is a technique used in superscalar processors, enabling the issue of instructions out of program order. With out-of-order execution, instructions are checked for issue in program order and an instruction may be issued to computation units regardless whether all the instructions previous to it have been issued or not. Thus, in contrast to in-order execution, a stalled instruction does not cause its subsequent instructions to be stalled.

2.3 REGISTER RENAMING

Register Renaming is a technique used to resolve false dependencies, i.e. output and anti dependencies, caused by register accesses. It eliminates these dependences by renaming all destination registers, including those with a pending read or write for an earlier instruction, so that the out-of-order write does not affect any instructions that depend on an earlier value of the operand.

2.4 FPGA TECHNOLOGY

An FPGA device comprises three fundamental types of components: logic cells, I/O block and programmable routing. The generic FPGA architecture shown in Figure 2.3 resembles a traditional well-structured island-style architecture in which array of logic blocks are surrounded by pre-fabricated programmable routing channels. Each of the logic clusters contains several logic cells which are interconnected by local routing. A logic cell is an elementary building block composed of a k-LUT and flip-flops.

A circuit is implemented in an FPGA by programming each of the logic cells to implement a small part of the overall logic functionality. Each I/O block is configured to act as either an input pad or output pad, as required by the circuit. The programmable routing is configured to make all the required connections between logic cells, and also from logic cells to I/O blocks.

Early FPGAs were not clustered, however as FPGAs matured and grew more dense, the clustering was introduced to reduce the compile-time, and at the same time to reduce the global routing requirements and increase operating speed. Unfortunately, there is still a significant logic density gap between FPGAs and ASICs. One of the approaches in FPGA architecture for narrowing this gap is by including hard circuits such as memories and multipliers. While the soft logic cells can implement any functionality (as well as memories and multipliers), some implementations are particularly inefficient. Hence, the hard blocks increase the efficiency of these circuits. The early FPGAs comprised only logic blocks and thus were homogeneous, modern FPGAs are evolving in the direction of a heterogeneous architecture.

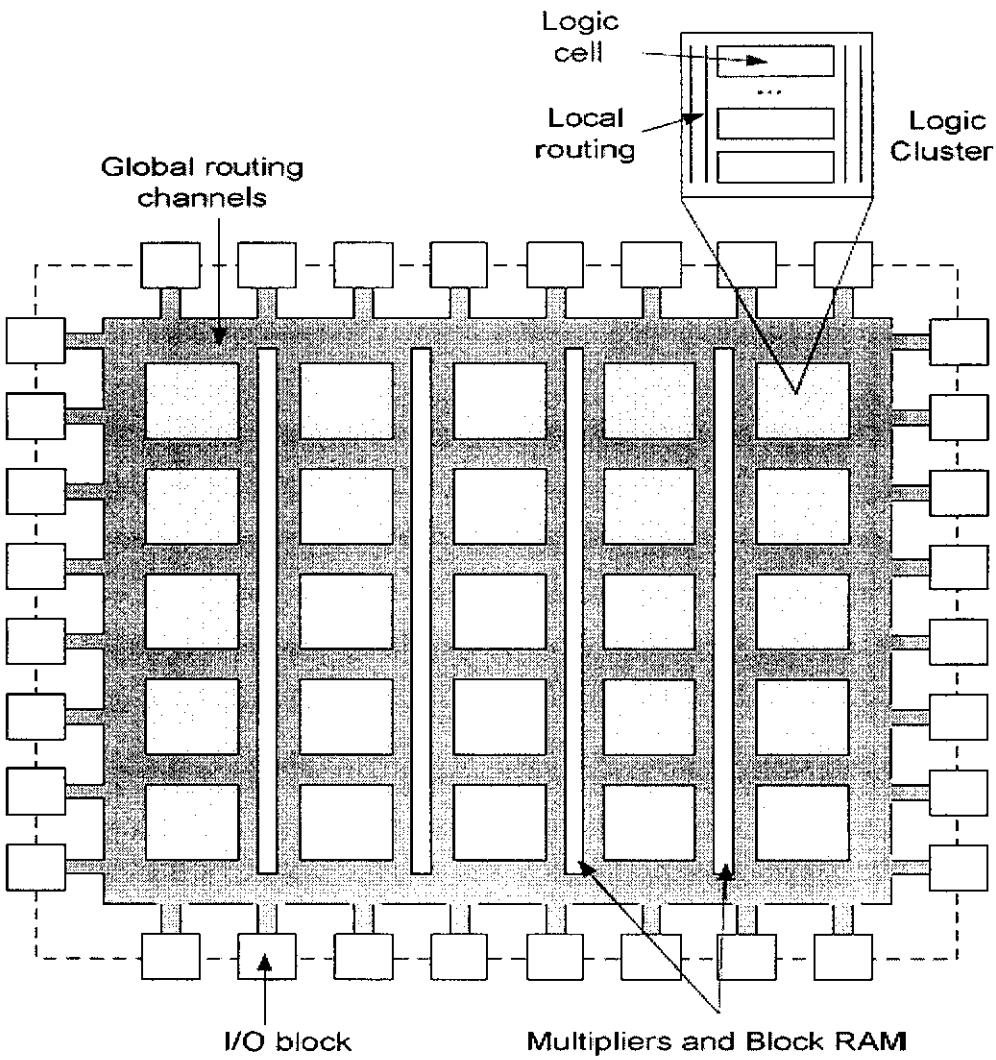


Figure 2.3 Generic island-style FPGA architecture

Implementation of large digital circuits such as entire systems-on-chips in modern high-density FPGAs is enabled by sophisticated CAD tools. The process of mapping a circuit on an FPGA is broken into a series of sequential sub problems which make the procedure tractable. In the first stage, a designer describes a circuit in a hardware description language (HDL, such as Verilog or VHDL).

The second stage is synthesis stage which contains several steps. In the first step HDL is converted into a netlist of basic gates which is subsequently minimized. Then, the basic netlist in mapped onto a netlist of FPGA logic cells. The third step is packing, in which the logic cells are packed into the logic clusters. In the third stage, for each logic cluster from the netlist it is decided where it is going to placed on an FPGA device. Modern placers are predominately based on a simulated annealing algorithm.

After locations of logic blocks are determined, a router finds a path between each connected logic block in the netlist and determines how the routing channels should be configured to implement the connection. Routing algorithms are usually timing-driven as most of the delay in FPGAs stems from the programmable routing. Once it is ascertained that the synthesized circuit meets all requirements of the specifications (such as area or timing) the chip programming file is generated.

2.5 BENEFITS OF THE MLCA

The combination of architecture and programming model of the MLCA give rise to a number of advantages for SoC designs:

- Reduced software complexity.

  The programming model of MLCA is close to that of sequential programming. An order of execution is given to the CP and the PUs, which when executed sequentially is considered to provide correct results. This alleviates the need for explicit parallel programming, which leads to considerable software complexity.

- Automatic extraction of the parallelism.

  Similar to instruction-level parallelism, speedup can be achieved through register renaming and out-of-order execution. For output and anti dependences, the CP allocates new registers, allowing the tasks to run in parallel on separate PUs.

- Multiple levels of parallelism.

  The MLCA combines task-level parallelism at the top-level and instruction-level parallelism in the PUs. Task-level parallelism is potentially higher than the instruction-level parallelism that can be extracted from sequential code.

- Separation of communication and synchronization from computations

  Synchronization and communication are often significant contributors to the complexity and cost of an embedded system. Both can lead to contention on interconnects, increased latency, and power consumption. In addition, software development is complicated by the need to insert synchronization and communication primitives. The MLCA programming model provides separation of synchronization and communication on the one hand, and computations on the other. Computations are done entirely within the tasks, i.e. they are accomplished by the PUs. Task communication and implicit synchronization are performed by the CP and the URF. This contrasts with most current parallel programming models used with existing multi-processor SoCs, where the communication and synchronization are explicit and are mixed in with the application code modules.

- Efficient communication.

  Tasks may communicate through the URF instead of relying on a shared memory, leading to potentially faster and more efficient communication.

- Fast architecture exploration.

  The modularity and flexibility of the template architecture that accommodates heterogeneity and the independence of the code from resource management, allow fast exploration of a MLCA configuration for a specific application.

CHAPTER – 3

DESIGN ISSUES

In this chapter the main challenges in designing and implementing the MLCA microarchitecture is reviewed. The challenges stem from the three different aspects. The first aspect is the flexibility of the MLCA programming model, which the MLCA microarchitecture implementation has to fully support. The second aspect is the set of differences in architectural models between a modern high-performance processor and the MLCA. The third aspect is the set of the constraints imposed on our implementation by the target FPGA device and the platform based on that device.

3.1 ARCHITECTURAL DIFFERENCES

In this section the differences between the architectural model of the MLCA and that of a high-performance processor is described. Furthermore, the impact of these differences on the proposed design is also described.

Table 3.1 summarizes the main differences of architectural models of a high-performance processor and the MLCA. The MLCA is founded on the similar techniques and mechanisms to those that extract parallelism at instruction granularity. The differences in implementation of the MLCA and a superscalar processor stems from the fact of applying them to the coarse granularity computational elements. Due to these differences the conventional superscalar microarchitecture cannot be taken and applied in a straightforward manner. The challenges that are faced makes the necessity to revisit the superscalar techniques so they can be efficiently implemented at coarse-granularity.

The granularity of execution in superscalar processors is on the order of 1 to 100's of clock cycles. On the other hand, MLCA tasks have granularity from 1000's to 100,000's of clock cycles. Modern superscalar processor proved non-scalable for more than 10 functional units. In case of the MLCA, the processing units are more complex units, such as RISC, VLIW, or DSP processors.

In contrast to superscalar processors, we believe that the MLCA can scale to a larger number of processing units. The proposed work shows that MLCA can scale up to 8 PUs. The scalability analysis with larger numbers of PUs was precluded by the limitations of our target platform.

An MLCA task can have variable execution time depending on run-time values; hence its execution time is unknown at issue-time. On the other hand, most of processor instructions have non-variable execution time, which is known at issue-time. Another difference is the number of inputs and outputs. Instructions usually have a small number of inputs and outputs (0 to 3). In contrast, MLCA tasks can have dozens of inputs and outputs (10 to 100). Furthermore, computation atomicity of RISC instructions in terms of their outputs is simple; the output is produced at the end of the execution. MLCA tasks produce their outputs at arbitrary times and in any order.

Feature	High-performance processor	MLCA
Computational element	Instruction	Task
Computation granularity (clock cycles)	Fine 1-100's	Coarse 1000's-100,000's
Number of processing units	4-10 FUs	4-32 PUs
Execution time variability	Lower variability	Higher variability
Number of inputs/outputs	Small (0-3)	Large (10-100)
Computation atomicity	Output is produced at the end of execution	Outputs are produced at arbitrary times and in any order

Table 3.1: Differences between architectural models of a high-performance processor and the MLCA

MLCA tasks during their execution can be written to the shared memory, which is also used for inter-task communication. However, the CP is unaware of this communication; it only observes the communication through registers. If tasks communicate data that is in the shared memory (e.g. a buffer or a structure), they have to be synchronized via a register dependency assigned to each memory dependency. It is the job of the MLCA compiler to associate shared memory dependencies with synchronization through registers.

3.2 CONSTRAINTS OF FPGA DEVICE AND PLATFORM

An additional influence on the proposed microarchitecture is the target FPGA technology which can be tailored with some microarchitectural decisions. Some choices at the logic level is made such that it is FPGA-specific. Nevertheless, microarchitecture and logic implementation is kept as platform-agnostic as possible and hope to port it in the future to other target technologies such as standard cell technology.

The constraints imposed in this FPGA platform can be roughly divided into two groups. The first group contains the constraints, which are encountered when implementing the microarchitecture of the CP. The second group of constraints posed limits on the size of a full large scale MLCA system with the CP and PUs when implementing on the target FPGA device.

Regarding the first group of constraints that influenced the design of our microarchitecture, most of them are driven by the architecture of FPGA devices and inherent properties of the FPGA building blocks. For a given microarchitectural unit, memory-based implementation or a logic-based implementation are the choices. The alternatives had to be weighed in terms of operating frequency and area.

For instance, some logic design choices that would be natural in standard cell ASIC design technology proved to be prohibitively expensive in FPGA logic, or they required long routing channels which had impact on maximum operating frequency. An example of a design that requires expensive implementation are the content addressable memories (CAMs).

They do not exist as embedded blocks and have to be implemented using generic FPGA logic resources which are prohibitively expensive. Similarly, FPGA memory blocks have only two ports, which dictate careful design and distribution of memory-based microarchitectural structures. If a memory structure requires more ports, it has to be replicated, which is an expensive and a very FPGA-specific approach. For this reason, in general, we avoided the choice of multi-ported structures.

On the other hand, the second group of constraints stems from the absolute limits of resources, that is, the number of logic and memory resources that are available on our target FPGA device. This was mostly manifested by the limited number of processing units to be fitted on the FPGA device. The total FPGA memory blocks were not enough to place the shared memory and application data on chip, so we used off-chip SDRAM as a shared memory, which limited our shared memory bandwidth. The limited on-chip memory also had impact on the type and the size of the memory hierarchy that we could implement on chip.

3.3 DESIGN OVERVIEW

Figure 3.1 shows the overall organization of the MLCA system in the FPGA. All of the components are placed on the FPGA device except the shared memory (SDRAM). The URF and CRF register communication between the tasks (running on PUs) goes through the CP which manages the CRF and URF register files

The MLCA control program is placed into a dedicated memory block. The code of the task functions, static, stacks and heap memory regions are all placed into the off-chip SDRAM. Each PU has L1 data and instruction caches. Since tasks can communicate through shared memory, its contents have to be kept coherent. The caches that are used are write-back and there is no cache-coherence mechanism. In this MLCA, system coherence is achieved using a software-managed mechanism that flushes the shared data from the cache before it is used by dependent tasks. The dependencies between the tasks are of two types.

The first type is scalar dependencies through which tasks only share a scalar value via a register. The second type is memory dependencies. Tasks share data through a memory buffer; however they synchronize and share the pointer to the memory buffer via a register. There is no flushing needed for the scalar dependencies.

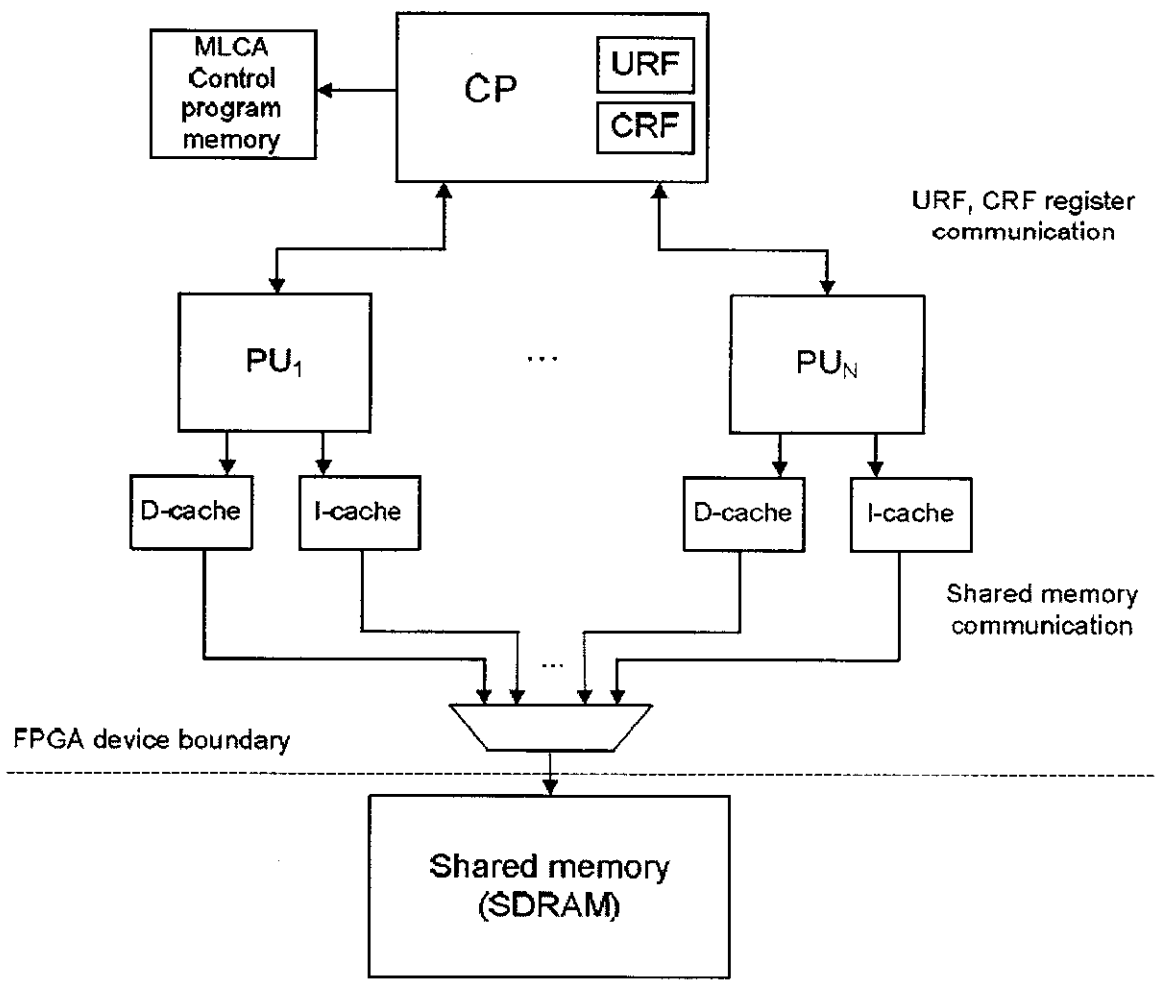


Figure 3.1 Overall organization of the MLCA system in the FPGA

3.4 THE MLCA SYSTEM DESIGN AND IMPLEMENTATION

Here implementation of the MLCA system is carried out using 8 PUs. All of the components are placed on the FPGA device except the shared memory (DDR2 SDRAM). The off-chip memory is accessed via Altera's High- Performance DDR2 controller. The URF and CRF register communication among tasks goes through the CP, which also manages the CRF and the URF. The control program is placed into a dedicated on-chip memory. The off-chip memory is logically divided among PUs, such that each PU has its own stack, heap and a copy of the task functions. PUs can share data through the heap. There is no operating system or virtual memory.

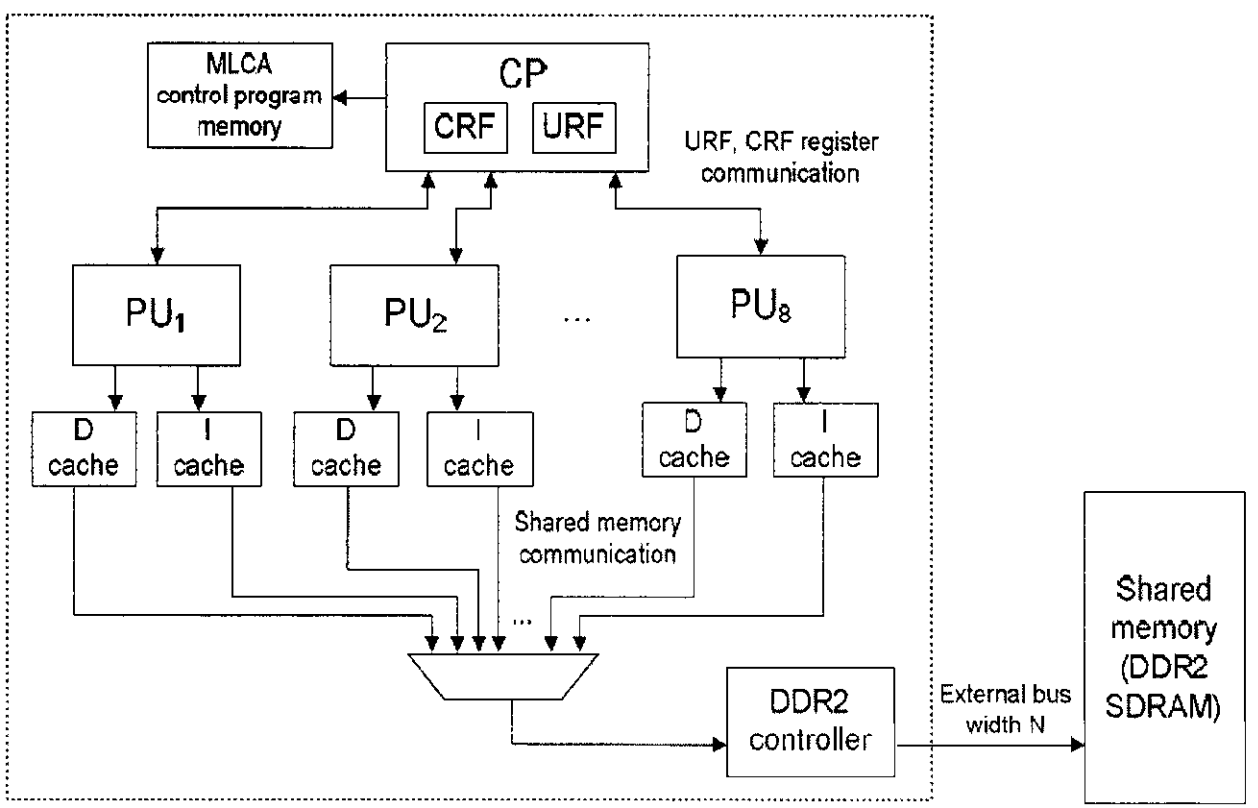


Figure 3.2 Organization of the MLCA system on our platform

Each PU runs a monitor program which receives task IDs form the CP and executes the corresponding task functions. Task input and output registers are communicated via designated input and output memories of each PU (small internal FPGA memory).

CHAPTER - 4

CONTROL PROCESSOR MICROARCHITECTURE

4.1 DESIGN METHODOLOGY

The MLCA Control Processor and a modern high-performance processor share similar functionalities such as breaking of false dependencies and out-of-order execution. However, the differences between the architectural models of the MLCA and a modern high-performance processor dictate significant differences in our design approach if we wish to arrive at a complexity-efficient implementation of the CP, i.e. achieve a high operating frequency and consume minimal chip resources.

For these reasons a novel design of these functionalities is delivered. To aid the design decision, a design methodology is adapted.

This methodology for deriving a digital logic implementation of the required functionalities encompasses three steps, each of which comprises a set of choices. The criteria employed to evaluate the final set of choices were complexity and scalability. The complexity of a unit is defined by two metrics: the maximum operating frequency and resource usage. Lower complexity is better, that is, higher frequency and less resources consumed.

The scalability is the ability of a unit to expand in a chosen dimension with a minimal increase in complexity. To determine the complexity and scalability, FPGA mapping of the design is examined. In the first step, the particular microarchitectural technique is choosed such that it provides a certain subset of the required functionalities.

The second step considers the structural design choices of each microarchitectural technique in terms of temporal and spatial parallelism necessary to meet throughput requirements. In the third taken into consideration is the logic-implementation-specific choices which are driven by the constraints of the target FPGA platform.

It is important to note that choices in all three steps are interdependent. For instance, an initially appealing microarchitectural technique might require logic design choices that lead towards a prohibitively expensive implementation. In that case, the choice of the microarchitectural technique has to be reexamined.

In case there exist feasible implementations of the selected microarchitectural technique, then the required performance can be achieved by balancing the trade-off between throughput and operating frequency. A higher throughput necessitates a higher design complexity, on the other hand to achieve a high operating frequency the design complexity should be kept low.

Since the resulting maximum frequency is highly influenced by the CAD tools which automatically place and route the circuit, an assumption about the maximum frequency of our final CP design to make the analysis is introduced.

Hence, the throughput and latency analysis using the execution times of tasks running on PUs expressed in terms of PU clock cycles is performed. In other words, we initially assume that the frequency of the CP matches that of the processor; i.e. one PU cycle is equal to one CP cycle.

Once design is completed, synthesize and evaluation of the performance is carried. measured by overall application speedup. If the performance is not satisfactory, it is due to two possible reasons. It may be that the assumed maximum frequency is not attained, and the throughput is inadequate. Subsequently, either or both factors are optimized.

The process is iterated until satisfactory performance is achieved. Then the area, second complexity metric is measured as the total count of each type of FPGA resource in a synthesized design. Hence, a separate count of different resource types is maintained. For example, total number of logic cells and memory bits used by the synthesized design are separately counted.

4.2 OVERVIEW OF THE MICROARCHITECTURE

The main functionalities provided by the CP microarchitecture are parallel and out-of-order execution of tasks. The two microarchitectural techniques that enable these functionalities are *register renaming* and *dynamic task scheduling*.

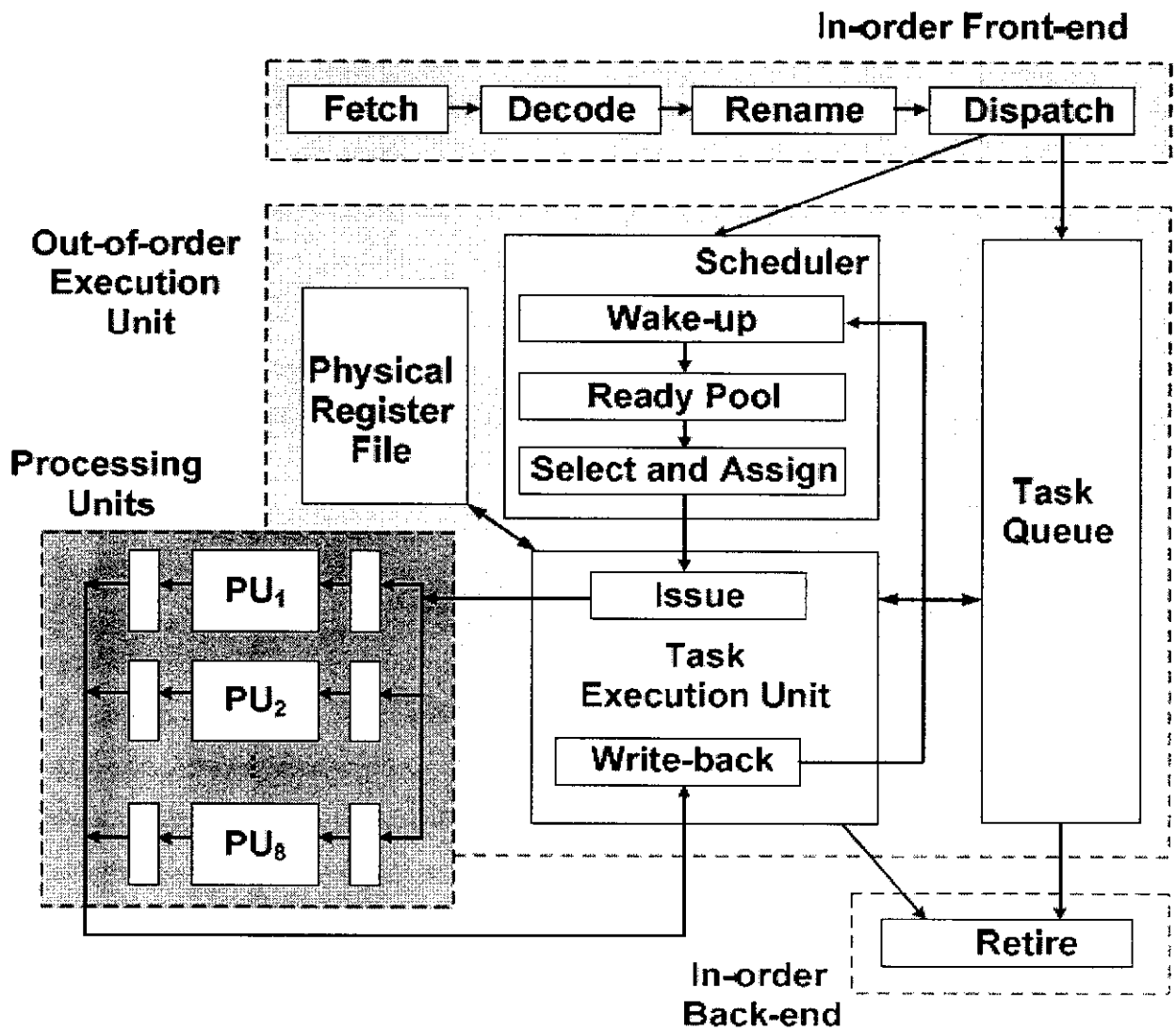


Figure 4.1 Main physical units of the CP

Figure 4.1 depicts the conceptual stages of the CP. The stages are designed by following the pattern common to the pipeline of contemporary high-performance microprocessors. It consists of an in-order front end, an out-of-order execution core, and an in-order back end.

The front end is responsible for bringing task instructions (TIs) into the execution core and it also employs a register renaming mechanism. In the front end, a TI is fetched, decoded, and its inputs and outputs are renamed. The front end operates in an in-order fashion, since the inherent requirement of the fetch and rename mechanisms is an input stream of TIs in their original in-program-order.

The last step of the front end is the dispatch stage which inserts TIs into the execution core, after allocating required resources for each TI. The out-of-order execution core is the central part of the CP. Upon arriving into the execution core, TIs waits until their inputs become ready. Once this condition is met for a task, the wake-up stage updates the state of the task, and it enters the pool of tasks ready for execution.

Next, the execution core selects a ready task from the pool and assigns it to an available PU. In the issue stage the task is sent to the PU. The execute stage represents the actual execution of the task function by the PU. In the writeback stage the arguments written by the task are forwarded to the earlier stages for the wakeup of waiting tasks. The execute stage and the writeback stage can be overlapped since tasks produce multiple arguments during their execution.

The back end comprises only the retire stage which removes (i.e. retires) completed tasks from the execution core. A task is removed from the core only if all tasks earlier to it in program order have finished execution. At this point the resources that the task has occupied in the core are released. The in-program-order completion is intended to support precise interrupts and recovery from exceptions and miss peculation.

Although the CP is based on established principles commonly used for extracting parallelism at the instruction granularity, there are key differences. Task instructions have many inputs and outputs and tasks execute for many cycles as opposed to just few cycles, as is the case with instructions. This leads to different constraints and also to opportunities in the implementation of the MLCA.

A key challenge in the CP design is determining the required throughput of the CP. The two critical components are the issue unit and the write-back unit. Thus, the two questions arise are: at which rate should the CP be able to issue tasks to PUs and at which rate should the CP be able to process outputs received from PUs. In both cases the rates are influenced by the variability of key architectural parameters, including task granularity, number of PUs, number of inputs/outputs, etc.

FPGA memory blocks have only two ports, which dictate careful design and distribution of memory-based structures. Throughout the design, the choice of multiported memory structures is avoided. The CP's memory structures across multiple memory blocks are distributed in order to obtain more throughputs to those structures.

It is possible that for larger MLCA systems with 16 or more PUs, several units of the CP might become a bottleneck. The memory structures in those CP's units will require more than two ports, hence they will have to be replicated or further distributed in order to provide more ports.

4.2.1 CP PIPELINE OVERVIEW

The CP's front end is responsible for bringing TIs into the OoOE unit and it also employs a register renaming mechanism. A TI is fetched, decoded, and its inputs and outputs are renamed in an in-order fashion. The last stage of the front end is the dispatch unit which inserts TIs into the OoOE unit, and allocates required resources for each TI in the OoOE unit.

The OoOE unit is the central part of the CP. Its main components are the task queue, the dynamic scheduler and the task execution unit. Upon arriving into the OoOE unit, TIs are

placed into the task queue and also into the wake-up unit of the dynamic scheduler. TIs wait in the wake-up unit until their inputs become ready. The task queue holds information about all TIs in flight. It provides the functionality similar to that of the reorder buffer in a superscalar processor.

TIs are considered to be in-flight after they have been inserted into the task queue by the dispatch unit. A TI ceases to be in-flight once it is retired by the retire unit. Once all inputs of a TI are available, the corresponding task becomes ready for execution. The wake-up unit then updates the state of the TI and forwards it to the pool of TIs ready for execution. The select and assign unit selects one of the ready TIs from the pool and assigns it to one of the PUs that are free. The issue unit sends the TI to the assigned PU2, where the corresponding task is executed.

The writeback unit collects the registers written by tasks and forwards them to the wakeup unit which processes the registers and determines if any of the waiting TIs have become ready. The task execution and the writeback stages can be overlapped since tasks produce multiple outputs during their execution.

The back-end comprises only the retire unit which removes completed TIs from the task queue and other CP structures. A TI that has finished its execution is removed from the CP only if all tasks earlier to its corresponding task in program order have also finished executing. At this point the resources that the TI has occupied in the CP are released. The in-program order completion is intended to support precise interrupts and recovery from exceptions and mispeculation.

4.3 SOFTWARE-MANAGED COHERENCE

There is no hardware support for cache-coherence. A software-managed mechanism is employed such that it flushes shared data from a cache before it is used by dependent tasks. There is no flushing needed for simple scalar dependencies (e.g., integers), which are handled by the URF. However, tasks can also share complex data structures (such as buffers) and synchronize through URF registers that hold pointers to those data structures. This data must be flushed.

Two methods of flushing shared data can be possible. The false sharing can be prevented by ensuring that individual memory regions are aligned to cache lines. Our flushing mechanism increases execution time in 3 ways. First, it adds execution time to flush dirty lines. Second, it introduces extra memory traffic that can lead to contention among PUs and further prolong task execution time. Third, when FullFlush is used, the cache lines corresponding to PU's stack region are also flushed and must be reloaded when a new task is started.

4.4 MEMORY SUBSYSTEM DESIGN

The caches are direct-mapped and write-back. However, even in the face of these limitations, the design space for the memory system is large, and it is not feasible to explore it all. There are many design options: i) capacity of I/D caches, ii) line size for D cache, iii) burst modes for I/D caches, iv) arbitration priorities, v) off-chip memory bus width.

4.5 OTHER MICROARCHITECTURAL COMPONENTS

In this section the remaining microarchitectural components of the CP is described.

4.5.1 THE EXECUTE UNIT

The execute unit of the CP consists of the issue unit and the write back unit. Conceptually it consists of PUs as well, since they carry out the actual execution of the tasks. However given the well-defined interface between the CP and the PUs (described further below), the design of the PUs is completely decoupled and independent from the CP design.

4.5.2 THE DECODE, RENAME AND DISPATCH UNITS

The decode, rename and dispatch units constitute major part of the CP's front-end. The Task Instruction Decoder (TID) of the decode unit receives task instruction words from the fetch unit. The TID then splits each task instruction word into a task header and individual input and output arguments. The task header is not modified and it is directly sent to the task header dispatch queue which resides within the dispatch unit.

On the other hand, input and output arguments have to be renamed prior to being dispatched. Both inputs and outputs are placed into the register rename queue. Each entry in the queue holds information for a single register. For a given task, all of its inputs are inserted first into the register rename queue, following all of its outputs. The implementation is straightforward, since within a single instruction word, all inputs are listed before the outputs After the TID decodes all words of a TI, it restarts the decode process for the next TI. First, the task header is extracted and the process continues with inputs and subsequently.

CHAPTER – 5

CACHE MEMORY

A **CPU cache** is a cache used by the central processing unit of a computer to reduce the average time to access memory. The cache is a smaller, faster memory which stores copies of the data from the most frequently used main memory locations. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory.

When the processor needs to read from or write to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory. Most modern desktop and server CPUs have at least three independent caches: an **instruction cache** to speed up executable instruction fetch, a **data cache** to speed up data fetch and store, and a translation look aside buffer used to speed up virtual-to-physical address translation for both executable instructions and data.

When the processor needs to read or write a location in main memory, it first checks whether that memory location is in the cache. This is accomplished by comparing the address of the memory location to all tags in the cache that might contain that address. If the processor finds that the memory location is in the cache, we say that a *cache hit* has occurred; otherwise, we speak of a *cache miss*. In the case of a cache hit, the processor immediately reads or writes the data in the cache line. The proportion of accesses that result in a cache hit is known as the *hit rate*, and is a measure of the effectiveness of the cache for a given program or algorithm.

In the case of a miss, the cache allocates a new entry, which comprises the tag just missed and a copy of the data. The reference can then be applied to the new entry just as in the case of a hit. Read misses delay execution because they require data to be transferred from a much slower memory than the cache itself.

Write misses may occur without such penalty since the data can be copied in the background. Instruction caches are similar to data caches but the CPU only performs read accesses (instruction fetch) to the instruction cache. Instruction and data caches can be separated for higher performance with Harvard CPUs but they can also be combined to reduce the hardware overhead.

CHAPTER- 6

SIMULATION RESULTS

6.1 DESIGN SOFTWARE

The implementations have been carried out using the software, ModSim and Xilinx ISE 7.1i. The hardware language used is the Very High Speed Integrated Circuit Hardware Description Language (VHDL). VHDL is a widely used language for register transfer level description of hardware. It is used for design entry, compile and simulation of digital systems. Modelsim is a simulation tool for programming {VLSI} {ASIC}s, {FPGA}s, {CPLD}s, and {SoC}s. Modelsim provides a comprehensive simulation and debug environment for complex ASIC and FPGA designs. Support is provided for multiple languages including Verilog, SystemVerilog, VHDL and SystemC.

6.2 SIMULATION RESULT OF MLCA

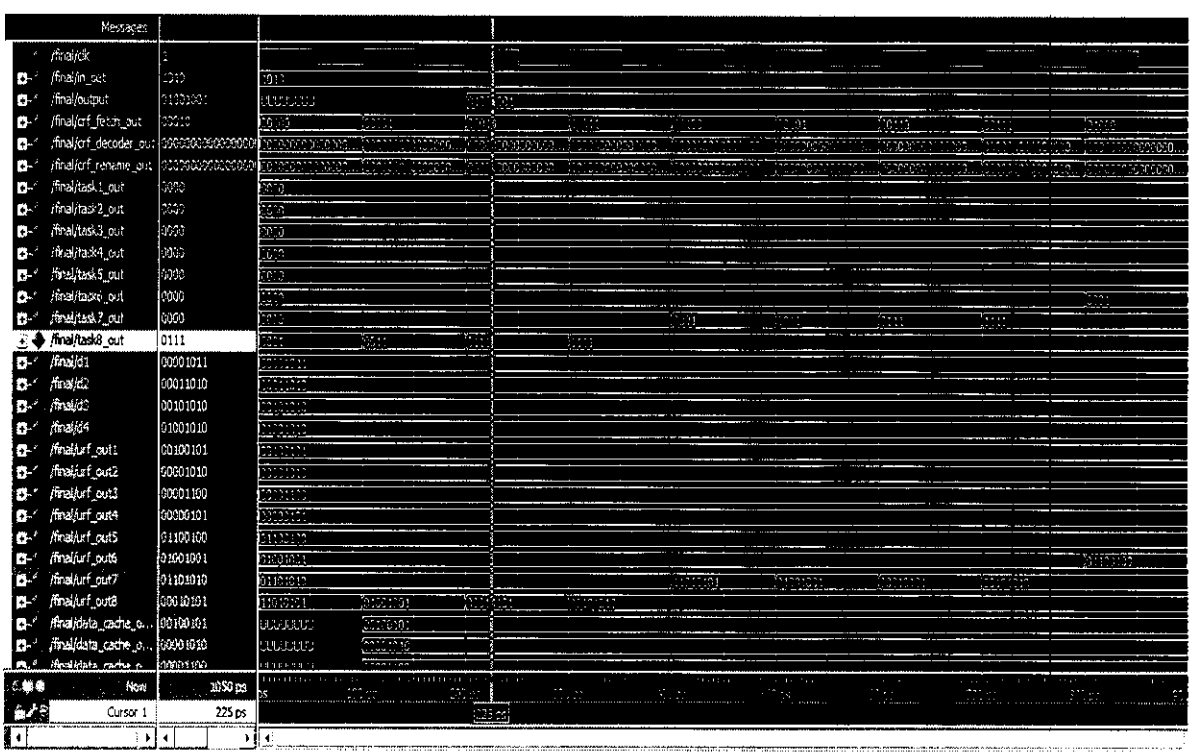


Figure 6.1 Simulation Result Obtained For MLCA

6.3 SYNTHESIS REPORT FOR MLCA

Design summary:

Selected Device: Xilinx FPGA- xc3s500e

Logic utilization	Used	Available	Utilization
No. of slices	1999	4656	42%
No. of sliced flipflops	1784	9312	19%
No. of 4 input LUTs	3406	9312	36%
No. Of bonded IOBs	13	190	6%
No. of BRAMs	3	20	15%
No. of GCLKs	1	24	4%

Figure 6.2 Synthesis report for MLCA

CHAPTER – 7

CONCLUSION AND FUTURE WORK

In this chapter, the main conclusion of this project is summarized and future work directions are described.

7.1 CONCLUSION

In this, design and implementation of the MLCA microarchitecture is considered, in particular CP is focused. The goal of the implementation was twofold. First, a full microarchitecture of the CP is designed for realistic performance experiments. At the same time, high-performance in a complexity-efficient manner is achieved, that is, minimal chip resources are consumed. The FPGAs is the target hardware technology. The advantages of the FPGAs such as fast design, cost allowed us to explore many microarchitectural alternatives. Moreover, the FPGA platform is treated not only as a vehicle for microarchitectural exploration, but also as the target deployment platform. This approach introduced the CP's efficiency as the second design constraint, in addition to the CP's performance which was the primary constraint.

Subsequently, the entire microarchitecture of the CP is designed. On a high-level it resembles the superscalar pipeline including the fetch, decode, rename and dispatch units as parts of the in-order front-end. The retire unit comprises the back-end of the pipeline. The CP's microarchitecture is implemented using System Verilog HDL. The CP is integrated into a shared-memory multiprocessor system based on 8 PUs. Finally the MLCA is synthesized on FPGA platform.

7.2 FUTURE WORK

The goal of future work is to investigate the scalability of the CP performance in order to support MLCA systems with large number of PUs. The first approach to improve the overall CP performance is by improving the maximum operating frequency of the CP. The second possibility for improving the CP performance is by increasing the throughput of individual units which are identified as bottlenecks.

In addition to investigating the scalability of the CP performance for a large number of PUs, the scalability of CP's complexity have to be explored. The first question is whether there would be a significant operating frequency drop if the number of PUs and other main parameters of the CP are increased. Similarly, the second question is how does the relative resource usage between the CP and the PU change as the entire system and the CP parameters grow.

REFERENCES

1. Matosevic, T. Abdelrahman, F. Karim, and A. Mellan, "Power Optimization for the MLCA Using DVS", in *Proc. of SCOPES 2005.*
2. K. Stewart and T. Abdelrahman, "Automatic Task Generation for the Multi-Level Computing Architecture", in *Proc. of PDCS, P*age no. 250-259, *2007.*
3. U. Aydonat, "Compiler support for a system-on-chip multimedia architecture", Master's thesis, U. of Toronto, 2005.
4. S. Singh and D. Greaves, "Kiwi: Synthesis of FPGA circuits from parallel programs", in *Proc. of FCCM*, Page no. 3-12, *2008.*
5. Z. Guo, W. Najjar, and B. Buyukkurt, "Efficient hardware code generation for FPGAs", *ACM TACO*, vol. 5, issue 1, 2008.
6. P. Yiannacouras, G. Steffan, and J. Rose, "Improving Memory System Performance for Soft Vector Processors", in *Proc. of WoSPS 2008.*
7. Ahmed M. Abdelkhalek and Tarek S. Abdelrahman, "Locality Management Using Multiple SPMs on the Multi-Level Computing Architecture", in Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia, pages 67–72, October, 2006.
8. Tarek Abdelrahman, Ahmed Abdelkhalek, Utku Aydonat, Davor Capalija, David Han, Ivan Matosevic, Kirk Stewart, Faraydon Karim, and Alain Mellan "The MLCA: A Solution Paradigm for Parallel Programmable SOCs", in IEEE North-East Work shop on Circuits and Systems (NEWCAS), Page no. 253, 2006.
10. Utku Aydonat and Tarek S. Abdelrahman, "Parallelization of Multimedia Applications on the Multi-Level Computing Architecture" in Parallel and Distributed Computing and Systems (PDCS), pages 438–447, 2006.

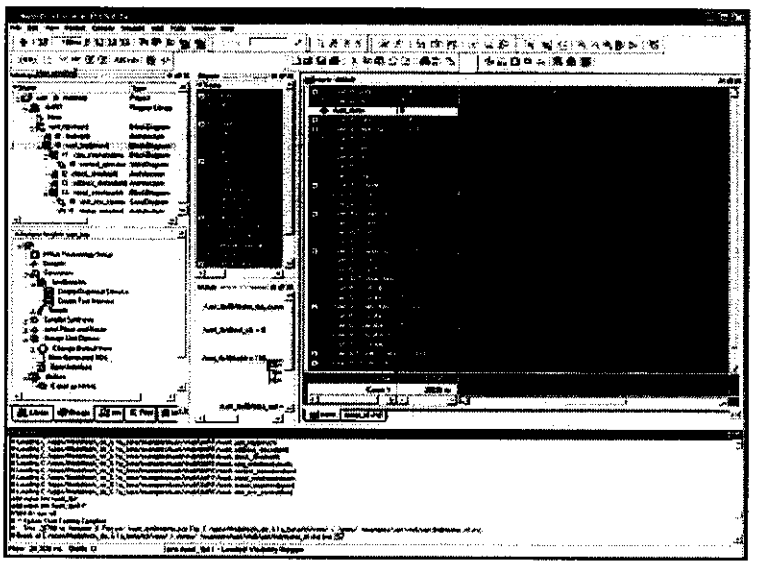
11. Taqi N. Buti, Robert G. McDonald, Zakaria Khwaja, Asit Ambekar, Hung Q. Le,William E. Burky, and Bert Williams, "Organization and implementation of the register-renaming mapper for out-of-order IBM POWER4 processors", IBM Journal of Research and Development, Page no.167–188, 2005.
12. J. Tripp, P. Jackson, and B. Hutchings, "Sea cucumber: A synthesizing compiler for FPGAs," in *Proc. of FPL,* Page no. 51-72, *2002.*
14. Y. Leow, C. Ng, and W. Wong, "Generating hardware from OpenMP programs", in *Proc. of FPT*, Page no. 73-80, *2006.*
15. U. Bondhugula, J. Ramanujam, and P. Sadayappan, "Automatic mapping of nested loops to FPGAs", in *Proc. of PPoPP*, Page no. 101-111, *2007.*
17. D. Unnikrishnan, J. Zhao, and R. Tessier, "Application-specific customization and scalability of soft multiprocessors", in *Proc. of FCCM 2009.*
18. F. Plavec, Z. Vranesic, and S. Brown, "Towards compilation of streaming programs into FPGA hardware", in *Proc. of FDL 2008.*
19. M. Labrecque, P. Yiannacouras, and G. Steffan, "Scaling soft processor systems", in *Proc. of FCCM,* Page no. 195-205, *2008.*
20. P. Yiannacouras, G. Steffan, and J. Rose, "Improving Memory System Performance for Soft Vector Processors", in *Proc. of WoSPS 2008.*

APPENDIX I

Design Creation to Realization

ModelSim Designer

D A T A S H E E T



ModelSim Designer combines easy to use, flexible creation with a powerful verification and debug environment.

The Easy to Use Solution for the FPGA Designer

ModelSim® Designer is a Windows®-based design environment for FPGAs. It provides an easy to use, advanced-feature tool at an entry-level price. The complete process of creation, management, simulation, and implementation are controlled from a single user interface, facilitating the design and verification flow and providing significant productivity gains.

ModelSim Designer combines the industry-leading capabilities of the ModelSim simulator with a built-in design creation engine. To support synthesis and place-and-route, ModelSim Designer is plug-in ready for the synthesis and place-and-route tools of your choice. Connection of these additional tools is easy, giving FPGA designers control of design creation, simulation, synthesis, and place-and route from a single cockpit.

A flexible methodology conforms to existing design flows. Designers can freely mix text entry of VHDL and Verilog code with graphical entry using block and state diagram editors. A single design unit can be represented in multiple views, and the management of compilation and simulation at all levels of abstraction is a single click away.

Major product features:

- Project manager, version control, and source code templates and wizards
- Block and state diagram editors
- Intuitive graphical waveform editor
- Automated testbench creation
- Text to graphic rendering facilitates analysis
- HTML Documentation option
- VHDL, Verilog, and mixed-language simulation
- Optimized native compiled architecture
- Single Kernel Simulator technology
- Advanced debug including Signal Spy™
- Memory window
- Easy-to-use GUI with Tcl interface
- Support for all major synthesis tools and the Actel, Altera, Lattice, and Xilinx place-and-route flows
- Built-in FPGA vendor library compiler
- Full support for Verilog 2001
- Windows platform support

Options: *SWIFT Interface support, graphics-based Dataflow window, Waveform Compare, integrated code coverage, and Profiler (for more details on options, please see the ModelSim SE datasheet)*

www.model.com/modelsimdesigner

Mentor Graphics

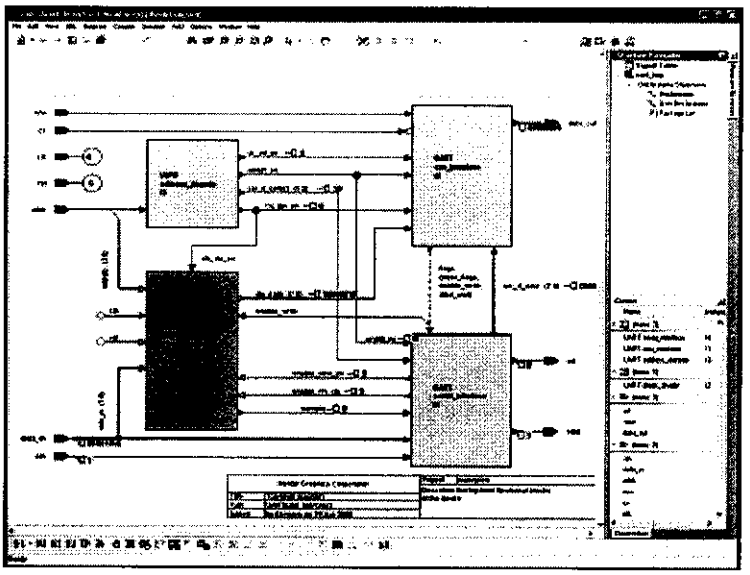
Implementation tasks are achieved through tight integration with the most popular FPGA synthesis and place-and-route tools in the industry. ModelSim Designer can either run these tools directly or externally, via scripts. Either way, the design data is maintained and used in a common and consistent way.

Project Manager

The flexible and powerful project manager feature allows easy navigation through a design in order to understand design content and execute all necessary tasks. The project manager provides easy access to all design data for each individual design unit, throughout the design process. Synthesis results, place-and-route results, back annotated netlists, and SDF files are associated with the relevant design unit. Project archiving is a few simple clicks away, allowing complete version control management of designs. The project manager also stores user-generated design documents, such as Word, PowerPoint, and PDFs.

Templates and Wizards

Creation wizards walk users through the creation of VHDL and Verilog design units, using either text or graphics. In the case of the graphical editor, HDL code is generated from the graphical diagrams created. For text-based design, VHDL and Verilog templates and wizards help engineers quickly develop HDL code without having to remember the exact language syntax. The wizards show how to create parameterizable logic blocks, testbench stimuli, and design objects. Novice and advanced HDL developers both benefit from timesaving shortcuts.



The Block Diagram editor is a convenient way to visually partition your design and have the HDL code automatically generated.

Intuitive Graphical Editors

ModelSim Designer includes block diagram and state machine editors that ensure a consistent coding style, facilitating design reuse and maintenance. These editors use an intuitive, graphical methodology that flattens the learning curve. This shortens the time to productivity for designers who are migrating to HDL methodologies or changing their primary design language.

Designers can automatically view or render diagrams from HDL code in block diagrams or state machines. When the code changes, the diagram can be updated instantly with its accuracy ensured. This helps designers understand legacy designs and aids in the debugging of current designs.

Automated Testbench Creation

ModelSim Designer offers an automated mechanism for testbench generation. The testbench wizard generates VHDL or Verilog code through a graphical waveform editor, with output in either HDL or a TCL script. Users can manually define signals in the waveform editor or use the built-in wizard to define the waveforms. Either way, it is intuitive and easy to use and saves considerable time.

Active Design Visualization Enhances Simulation Debugging

During live simulation, design analysis capabilities are enhanced through graphical design views. From any diagram window, simulations can be fully executed and controlled. Enhanced debugging features include graphical breakpoints, signal probing, graphics-to-text-source cross-highlighting, animation, and cause analysis. The ability to overlay live simulation results in a graphical context speeds up the debug process by allowing faster problem discovery and shorter design iterations.

HTML Documentation

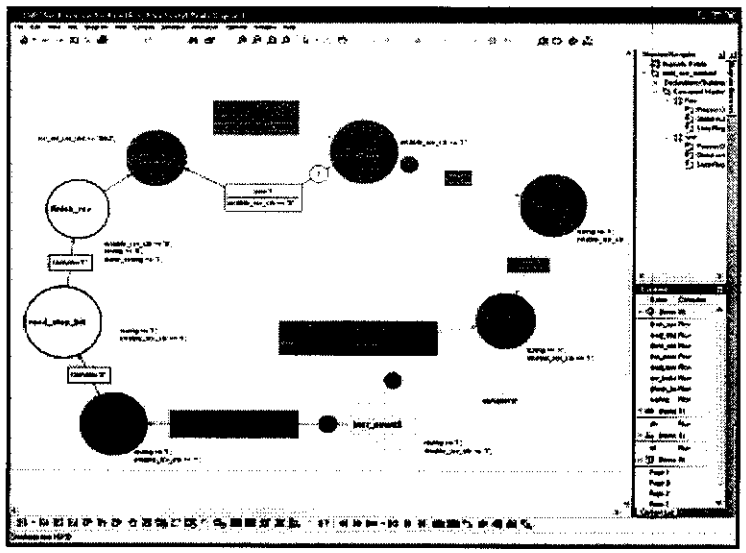
The HTML (HyperText Markup Language) Documentation option allows the user to output the complete design in HTML format. This allows the graphics, HDL, associative design files, and design hierarchy to be shared with anybody that uses an HTML browser. Design hierarchy and details can be viewed and navigated in any HTML browser, aiding communication and documentation.

Memory Window

The memory window enables flexible viewing and switching of memory locations. VHDL and Verilog memories are auto-extracted in the GUI, delivering powerful search, fill, load, and save functionality. The memory window allows pre-loading of memories, thus saving the time-consuming step of initializing sections of the simulation to load memories. All functions are available via the command line, making them available for scripting.

Intelligent GUI

An intelligently engineered GUI makes efficient use of desktop real estate. ModelSim Designer's intuitive layout of graphical elements (windows, toolbars, menus, etc.) makes it easy to step through the design flow. There are also wizards in place which help set up the design environment and make going through the design process seamless and efficient.



The State Diagram Editor uses interactive animation to track the simulation through the state machine and to check that all states were exercised.

Synthesis and Place-and-Route Integration

The industry's popular FPGA synthesis tools, such as Mentor Graphics Precision-RTL and most FPGA vendor synthesis tools, can be integrated into ModelSim Designer with push button convenience. Actel Designer and Libero, Altera Quartus, Lattice ispLEVEL, and Xilinx ISE place-and-route software are similarly integrated. Place-and-route results, together with SDF information, are automatically managed by ModelSim Designer after the process is complete, making them ready for post-place-and-route, gate-level simulation.

FPGA Vendor Library Compiler

ModelSim Designer provides an intuitive mechanism to compile the necessary vendor libraries for post-place-and-route simulation. The compiler detects which FPGA vendor tools have been installed and compiles the necessary libraries as soon as the tool is launched, taking care of this step once and for all.

Visit our web site at www.model.com/modelsimdesigner for more information.

APPENDIX II

Spartan-3E FPGA Family: Introduction and Ordering Information

DS312-1 (v3.8) August 26, 2009

Product Specification

Introduction

The Spartan®-3E family of Field-Programmable Gate Arrays (FPGAs) is specifically designed to meet the needs of high volume, cost-sensitive consumer electronic applications. The five-member family offers densities ranging from 100,000 to 1.6 million system gates, as shown in Table 1.

The Spartan-3E family builds on the success of the earlier Spartan-3 family by increasing the amount of logic per I/O, significantly reducing the cost per logic cell. New features improve system performance and reduce the cost of configuration. These Spartan-3E FPGA enhancements, combined with advanced 90 nm process technology, deliver more functionality and bandwidth per dollar than was previously possible, setting new standards in the programmable logic industry.

Because of their exceptionally low cost, Spartan-3E FPGAs are ideally suited to a wide range of consumer electronics applications, including broadband access, home networking, display/projection, and digital television equipment.

The Spartan-3E family is a superior alternative to mask programmed ASICs. FPGAs avoid the high initial cost, the lengthy development cycles, and the inherent inflexibility of conventional ASICs. Also, FPGA programmability permits design upgrades in the field with no hardware replacement necessary, an impossibility with ASICs.

Features

- Very low cost, high-performance logic solution for high-volume, consumer-oriented applications
- Proven advanced 90-nanometer process technology
- Multi-voltage, multi-standard SelectIO™ interface pins
  - Up to 376 I/O pins or 156 differential signal pairs
  - LVCMOS, LVTTL, HSTL, and SSTL single-ended signal standards
  - 3.3V, 2.5V, 1.8V, 1.5V, and 1.2V signaling
  - 622+ Mb/s data transfer rate per I/O
  - True LVDS, RSDS, mini-LVDS, differential HSTL/SSTL differential I/O
  - Enhanced Double Data Rate (DDR) support
  - DDR SDRAM support up to 333 Mb/s
- Abundant, flexible logic resources
  - Densities up to 33,192 logic cells, including optional shift register or distributed RAM support
  - Efficient wide multiplexers, wide logic
  - Fast look-ahead carry logic
  - Enhanced 18 x 18 multipliers with optional pipeline
  - IEEE 1149.1/1532 JTAG programming/debug port
- Hierarchical SelectRAM™ memory architecture
  - Up to 648 Kbits of fast block RAM
  - Up to 231 Kbits of efficient distributed RAM
- Up to eight Digital Clock Managers (DCMs)
  - Clock skew elimination (delay locked loop)
  - Frequency synthesis, multiplication, division
  - High-resolution phase shifting
  - Wide frequency range (5 MHz to over 300 MHz)
- Eight global clocks plus eight additional clocks per each half of device, plus abundant low-skew routing
- Configuration interface to industry-standard PROMs
  - Low-cost, space-saving SPI serial Flash PROM
  - x8 or x8/x16 parallel NOR Flash PROM
  - Low-cost Xilinx® Platform Flash with JTAG
- Complete Xilinx ISE® and WebPACK™ software
- MicroBlaze™ and PicoBlaze™ embedded processor cores
- Fully compliant 32-/64-bit 33 MHz PCI support (66 MHz in some devices)
- Low-cost QFP and BGA packaging options
  - Common footprints support easy density migration
  - Pb-free packaging options
- XA Automotive version available

Table 1: **Summary of Spartan-3E FPGA Attributes**

Device	System Gates	Equivalent Logic Cells	CLB Array (One CLB = Four Slices)				Distributed RAM bits(1)	Block RAM bits(1)	Dedicated Multipliers	DCMs	Maximum User I/O	Maximum Differential I/O Pairs
			Rows	Columns	Total CLBs	Total Slices						
XC3S100E	100K	2,160	22	16	240	960	15K	72K	4	2	108	40
XC3S250E	250K	5,508	34	26	612	2,448	38K	216K	12	4	172	68
XC3S500E	500K	10,476	46	34	1,164	4,656	73K	360K	20	4	232	92
XC3S1200E	1200K	19,512	60	46	2,168	8,672	136K	504K	28	8	304	124
XC3S1600E	1600K	33,192	76	58	3,688	14,752	231K	648K	36	8	376	156

Notes:

1. By convention, one Kb is equivalent to 1,024 bits.

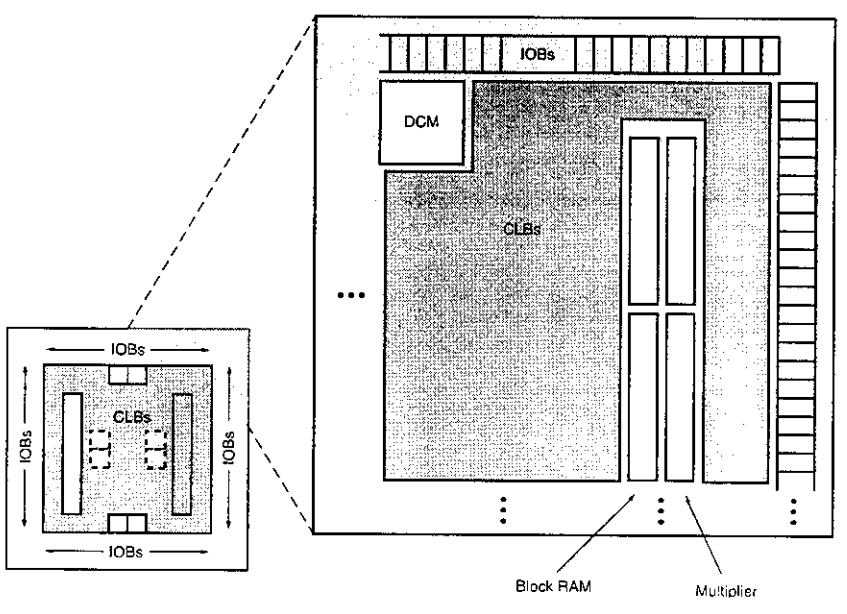
Architectural Overview

The Spartan-3E family architecture consists of five fundamental programmable functional elements:

- **Configurable Logic Blocks (CLBs)** contain flexible Look-Up Tables (LUTs) that implement logic plus storage elements used as flip-flops or latches. CLBs perform a wide variety of logical functions as well as store data.
- **Input/Output Blocks (IOBs)** control the flow of data between the I/O pins and the internal logic of the device. Each IOB supports bidirectional data flow plus 3-state operation. Supports a variety of signal standards, including four high-performance differential standards. Double Data-Rate (DDR) registers are included.
- **Block RAM** provides 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.
- **Digital Clock Manager (DCM) Blocks** provide self-calibrating, fully digital solutions for distributing, delaying, multiplying, dividing, and phase-shifting clock signals.

These elements are organized as shown in Figure 1. A ring of IOBs surrounds a regular array of CLBs. Each device has two columns of block RAM except for the XC3S100E, which has one column. Each RAM column consists of several 18-Kbit RAM blocks. Each block RAM is associated with a dedicated multiplier. The DCMs are positioned in the center with two at the top and two at the bottom of the device. The XC3S100E has only one DCM at the top and bottom, while the XC3S1200E and XC3S1600E add two DCMs in the middle of the left and right sides.

The Spartan-3E family features a rich network of traces that interconnect all five functional elements, transmitting signals among them. Each functional element has an associated switch matrix that permits multiple connections to the routing.



Notes:

1. The XC3S1200E and XC3S1600E have two additional DCMs on both the left and right sides as indicated by the dashed lines. The XC3S100E has only one DCM at the top and one at the bottom.

Figure 1: **Spartan-3E Family Architecture**

Configuration

Spartan-3E FPGAs are programmed by loading configuration data into robust, reprogrammable, static CMOS configuration latches (CCLs) that collectively control all functional elements and routing resources. The FPGA's configuration data is stored externally in a PROM or some other non-volatile medium, either on or off the board. After applying power, the configuration data is written to the FPGA using any of seven different modes:

- Master Serial from a Xilinx Platform Flash PROM
- Serial Peripheral Interface (SPI) from an industry-standard SPI serial Flash
- Byte Peripheral Interface (BPI) Up or Down from an industry-standard x8 or x8/x16 parallel NOR Flash
- Slave Serial, typically downloaded from a processor
- Slave Parallel, typically downloaded from a processor
- Boundary Scan (JTAG), typically downloaded from a processor or system tester.

Furthermore, Spartan-3E FPGAs support MultiBoot configuration, allowing two or more FPGA configuration bitstreams to be stored in a single parallel NOR Flash. The FPGA application controls which configuration to load next and when to load it.

I/O Capabilities

The Spartan-3E FPGA SelectIO interface supports many popular single-ended and differential standards. Table 2 shows the number of user I/Os as well as the number of differential I/O pairs available for each device/package combination.

Spartan-3E FPGAs support the following single-ended standards:

- 3.3V low-voltage TTL (LVTTL)
- Low-voltage CMOS (LVCMOS) at 3.3V, 2.5V, 1.8V, 1.5V, or 1.2V
- 3V PCI at 33 MHz, and in some devices, 66 MHz
- HSTL I and III at 1.8V, commonly used in memory applications
- SSTL I at 1.8V and 2.5V, commonly used for memory applications

Spartan-3E FPGAs support the following differential standards:

- LVDS
- Bus LVDS
- mini-LVDS
- RSDS
- Differential HSTL (1.8V, Types I and III)
- Differential SSTL (2.5V and 1.8V, Type I)
- 2.5V LVPECL inputs

Table 2: **Available User I/Os and Differential (Diff) I/O Pairs**

Package	VQ100 VQG100		CP132 CPG132		TQ144 TQG144		PQ208 PQG208		FT256 FTG256		FG320 FGG320		FG400 FGG400		FG484 FGG484	
Size (mm)	16 x 16		8 x 8		22 x 22		28 x 28		17 x 17		19 x 19		21 x 21		23 x 23	
Device	User	Diff	User	Diff	User	Diff	User	Diff	User	Diff	User	Diff	User	Diff	User	Diff
XC3S100E	**66** *(7)*	**30** *(2)*	**83** *(11)*	**35** *(2)*	**108** *(28)*	**40** *(4)*	-	-	-	-	-	-	-	-	-	-
XC3S250E	**66** *(7)*	**30** *(2)*	**92** *(7)*	**41** *(2)*	**108** *(28)*	**40** *(4)*	**158** *(32)*	**65** *(5)*	**172** *(40)*	**68** *(8)*	-	-	-	-	-	-
XC3S500E	**66**[3] *(7)*	**30** *(2)*	**92** *(7)*	**41** *(2)*	-	-	**158** *(32)*	**65** *(5)*	**190** *(41)*	**77** *(8)*	**232** *(56)*	**92** *(12)*	-	-	-	-
XC3S1200E	-	-	-	-	-	-	-	-	**190** *(40)*	**77** *(8)*	**250** *(56)*	**99** *(12)*	**304** *(72)*	**124** *(20)*	-	-
XC3S1600E	-	-	-	-	-	-	-	-	-	-	**250** *(56)*	**99** *(12)*	**304** *(72)*	**124** *(20)*	**376** *(82)*	**156** *(21)*

Notes:

1. All Spartan-3E devices provided in the same package are pin-compatible as further described in Module 4: **Pinout Descriptions**.
2. The number shown in **bold** indicates the maximum number of I/O and input-only pins. The number shown in (*italics*) indicates the number of input-only pins.
3. The XC3S500E is available in the VQG100 Pb-free package and not the standard VQ100. The VQG100 and VQ100 pin-outs are identical and general references to the VQ100 will apply to the XC3S500E.

Design Documentation Available

The functionality of the Spartan®-3E FPGA family is now described and updated in the following documents. The topics covered in each guide are listed below.

- **UG331: Spartan-3 Generation FPGA User Guide**
  http://www.xilinx.com/support/documentation/user_guides/ug331.pdf
  - Clocking Resources
  - Digital Clock Managers (DCMs)
  - Block RAM
  - Configurable Logic Blocks (CLBs)
    - Distributed RAM
    - SRL16 Shift Registers
    - Carry and Arithmetic Logic
  - I/O Resources
  - Embedded Multiplier Blocks
  - Programmable Interconnect
  - ISE® Design Tools
  - IP Cores
  - Embedded Processing and Control Solutions
  - Pin Types and Package Overview
  - Package Drawings
  - Powering FPGAs
  - Power Management
- **UG332: Spartan-3 Generation Configuration User Guide**
  http://www.xilinx.com/support/documentation/user_guides/ug332.pdf
  - Configuration Overview
    - Configuration Pins and Behavior
    - Bitstream Sizes
  - Detailed Descriptions by Mode
    - Master Serial Mode using Xilinx® Platform Flash PROM
    - Master SPI Mode using Commodity SPI Serial Flash PROM
    - Master BPI Mode using Commodity Parallel NOR Flash PROM
    - Slave Parallel (SelectMAP) using a Processor
    - Slave Serial using a Processor
    - JTAG Mode
  - ISE iMPACT Programming Examples
  - MultiBoot Reconfiguration

For specific hardware examples, please see the Spartan-3E Starter Kit board web page, which has links to various design examples and the user guide.

- **Spartan-3E Starter Kit Board Page**
  http://www.xilinx.com/s3estarter
- **UG230: Spartan-3E Starter Kit User Guide**
  http://www.xilinx.com/support/documentation/userguides/ug230.pdf

Create a Xilinx MySupport user account and sign up to receive automatic E-mail notification whenever this data sheet or the associated user guides are updated.

- **Sign Up for Alerts on Xilinx MySupport**
  http://www.xilinx.com/support/answers/19380.htm

Introduction

As described in **Architectural Overview**, the Spartan™-3E FPGA architecture consists of five fundamental functional elements:

- **Input/Output Blocks (IOBs)**
- **Configurable Logic Block (CLB) and Slice Resources**
- **Block RAM**
- **Dedicated Multipliers**
- **Digital Clock Managers (DCMs)**

The following sections provide detailed information on each of these functions. In addition, this section also describes the following functions:

- **Clocking Infrastructure**
- **Interconnect**
- **Configuration**
- **Powering Spartan-3E FPGAs**

Input/Output Blocks (IOBs)

For additional information, refer to the "*Using I/O Resources*" chapter in UG331.

IOB Overview

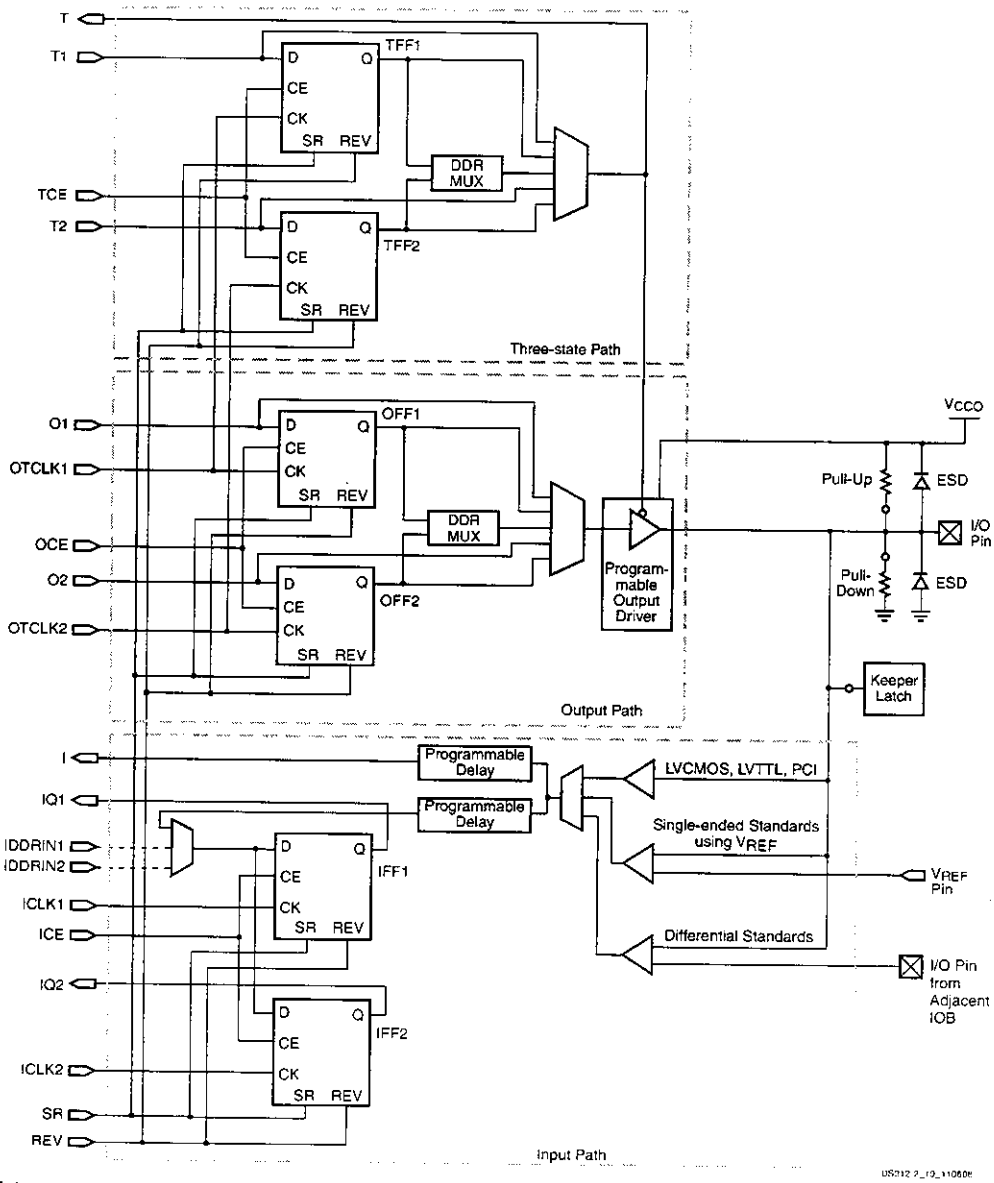
The Input/Output Block (IOB) provides a programmable, unidirectional or bidirectional interface between a package pin and the FPGA's internal logic. The IOB is similar to that of the Spartan-3 family with the following differences:

- Input-only blocks are added
- Programmable input delays are added to all blocks
- DDR flip-flops can be shared between adjacent IOBs

The unidirectional input-only block has a subset of the full IOB capabilities. Thus there are no connections or logic for an output path. The following paragraphs assume that any reference to output functionality does not apply to the input-only blocks. The number of input-only blocks varies with device size, but is never more than 25% of the total IOB count.

Figure 5, page 11 is a simplified diagram of the IOB's internal structure. There are three main signal paths within the IOB: the output path, input path, and 3-state path. Each path has its own pair of storage elements that can act as either registers or latches. For more information, see **Storage Element Functions**. The three main signal paths are as follows:

- The input path carries data from the pad, which is bonded to a package pin, through an optional programmable delay element directly to the I line. After the delay element, there are alternate routes through a pair of storage elements to the IQ1 and IQ2 lines. The IOB outputs I, IQ1, and IQ2 lead to the FPGA's internal logic. The delay element can be set to ensure a hold time of zero (see **Input Delay Functions**).
- The output path, starting with the O1 and O2 lines, carries data from the FPGA's internal logic through a multiplexer and then a three-state driver to the IOB pad. In addition to this direct path, the multiplexer provides the option to insert a pair of storage elements.
- The 3-state path determines when the output driver is high impedance. The T1 and T2 lines carry data from the FPGA's internal logic through a multiplexer to the output driver. In addition to this direct path, the multiplexer provides the option to insert a pair of storage elements.
- All signal paths entering the IOB, including those associated with the storage elements, have an inverter option. Any inverter placed on these paths is automatically absorbed into the IOB.



Notes:

1. All IOB control and output path signals have an inverting polarity option wihtin the IOB.
2. IDDRIN1/IDDRIN2 signals shown with dashed lines connect to the adjacent IOB in a differential pair only, not to the FPGA fabric.

Figure 5: **Simplified IOB Diagram**