A Novel μ-Processor Architecture for Concurrent Execution of OS Kernel and User mode Code

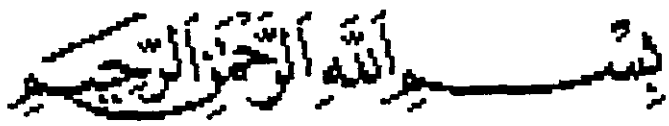
Developed by

Haroon Muneer

Supervised By:

Prof. Dr. Khalid Rashid
Dr. Tauseef Ur. Rehman

International Islamic University Islamabad
Faculty of Applied Sciences
Department of Computer Science
(2005)



**In the name of ALMIGHTY ALLAH,
The most Beneficent, the most
Merciful.**

International Islamic University Islamabad
Faculty of Applied Sciences
Department of Computer Science

08, 10, 2005

Final Approval

It is certified that we have read the thesis, entitled "**A Novel μ-Processor Architecture for Concurrent Execution of OS Kernel and User mode Code**" submitted by **Haroon Muneer,** University Reg. No. **72-CS/MS/02**. It is our judgment that this thesis is of sufficient standard to warrant its acceptance by the International Islamic University, Islamabad, for the Degree of MS Computer Science.

Committee

External Examiner
Mr. Shaftab Ahmed
Senior Faculty Member,
Bahria University Islamabad.

Internal Examiner
Engr. Dr. Syed Afaq Hussain
Head Department of Computer Engineering,
International Islamic University, Islamabad.

Supervisors

Prof. Dr. Khalid Rahsid
Dean Faculty of Applied Sciences,
International Islamic University, Islamabad.

Dr. S. Tauseef-ur-Rehman
Head Department of Telecom Engineering,
International Islamic University, Islamabad.

Dedication

Dedicated to The Holy Prophet Muhammad (SAW) and to my family.

A dissertation submitted to the
Department of Computer Science,
International Islamic University, Islamabad
as a partial fulfillment of the requirements
for the award of MS in
Computer Science

Declaration

I hereby declare that this software, neither as a whole nor as a part thereof has been copied out from any source. It is further declared that I have developed this software entirely on the basis of my personal efforts made under the sincere guidance of our teachers. No portion of the work presented in this report has been submitted in support of any application for any other degree or qualification of this or any other university or institute of learning.

Haroon Muneer
72-CS/MS/02

Acknowledgements

All praise to the Almighty Allah, the most Merciful, the most Gracious, without whose help and blessings, I was unable to complete the project.

Thanks to my Parents who helped me during my most difficult times and it is due to their unexplainable care and love that I am at this position today.

Thanks to my project supervisors ***Prof. Dr. Khalid Rashid and Dr. Tauseef Ur Rehman***, their sincere efforts helped me to complete my project successfully.

I acknowledge teachers and friends for their help in the project.

Haroon Muneer

Project in Brief

Project Title:	A Novel μ-Processor Architecture for Concurrent Execution of OS Kernel and User mode Code
Objective:	To Develop a new Chip Multiprocessor Architecture to execute both OS Kernel and User programs Concurrently.
Undertaken By:	**Haroon Muneer** **72-CS/MS/02**
Supervised By:	**Prof. Dr. Khalid Rahsid** Dean Faculty of Applied Sciences, International Islamic University, Islamabad **Dr. S. Tauseef-ur-Rehman** Head Department of Telecom Engineering, International Islamic University, Islamabad
Software Used:	Xilinx ISE Foundation 6.3i and ModelSim XE 5.8g
Hardware Used:	Xilinx Spartan IIE 300 mounted on the Burched FPGA prototyping Kit
System Used:	Pentium® III
Operating System Used:	Microsoft® Windows® XP Professional
Date Started:	15th March, 2004
Date Completed:	1st January, 2005

Abstract

One of the advances that will be enabled by system-on-chip (SOC) technology is the single-chip multiprocessor. As VLSI technology improves to allow us to fabricate hundreds of millions of transistors on a single chip, it is also possible to put a complete multiprocessor, including both CPUs and memory, on a single chip. The advent of single-chip multiprocessors will require us to rethink multiprocessor architectures to fit the advantages and constraints of VLSI implementation. This thesis is based on research in this very direction. We propose a new single chip multiprocessor architecture that concentrates specifically on improving operating system performance. This goal is achieved by implementing two CPUs on a single chip, configured as a master and a slave. Only the master processor will run the OS kernel, and only the slave processor will run the user code.

Table of Contents.

Abbreviations

ASIC	Application Specific Integrated Circuit
CMP	Chip Multiprocessor
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HLL	Higher Level Language
ILP	Instruction Level Parallelism
IPC	Inter Process Communication
MT	Multi Threading
NUMA	Non Uniform Memory Access
RTL	Register Transfer Level
SMP	Symmetric Multiprocessing
SoC	System on a Chip
TLB	Translation Look Aside Buffer
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VITAL	VHDL Initiative Toward ASIC Libraries
VLSI	Very Large Scale Integration

1. Introduction

For CPU design, a major movement in the 1980s was towards RISC instruction sets which made it simpler and more efficient to design CPU cores. Although, not all the RISC designs were that simple. During the 1990s the main focus was on boosting instruction-level parallelism (ILP) and clock rates of single-core processors. As a result, processors have become far more complex. However, there are various reasons why performance doesn't scale well with these techniques. Recent years have witnessed a shift of focus to exploiting thread-level parallelism (TLP) with techniques like CMP and MT. CMP/MT scales the performance of multi-threaded applications (or multiple running processes/programs) through the integration of multiple cores onto a single silicon die and the execution of instruction streams from multiple hardware threads on each core. Implementations of these technologies exist in the market today. Figure 1 shows the architectural differences of a single-core processor and a dual-core dual-thread CMP/MT processor.

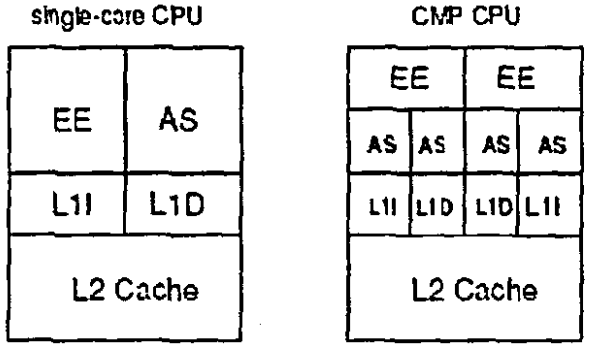


Figure 1.1

1.1 OS Kernels

The part of OS critical to its correct operation execute in supervisor mode, while other software, such as generic system software, and all applications programs execute in user mode. The part of the system software executing in the supervisor mode is called the kernel of the operating system. The kernel operates as trusted software, meaning that when it is designed and implemented, it is intended to implement protection mechanisms that cannot be covertly changed through the actions of untrusted software executing in the user mode.

1.1.1 Monolithic Kernels

The older monolithic kernels are written as a mixture of everything the OS needs, without much of an organization. The monolithic kernel offers everything the OS needs: processes, memory management, multiprogramming, interprocess communication (IPC), device access, file systems, network protocols.

1.1.2 Microkernel

This method structures the operating system by removing all nonessential components from the kernel, and implementing them as system and user level programs. The result is a smaller kernel. Microkernel typically provides minimal process and memory management, in addition to a communication facility. The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in the user space.

1.1.3 ExoKernel

Traditional operating systems limit the performance, flexibility, and functionality of applications by fixing the interface and implementation of operating system abstractions such as interprocess communication and virtual memory. The *ExoKernel* operating system architecture addresses this problem by providing application-level management of physical resources. In the ExoKernel architecture, a small kernel securely exports all hardware resources through a low level interface to untrusted library operating systems. Library operating systems use this interface to implement system objects and policies. This separation of resource protection from management allows application-specific customization of traditional operating system abstractions by extending, specializing, or even replacing libraries.

1.2 Parallelism

Traditionally computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as a sequence of instructions. Processors execute the programs by executing the machine instructions in a sequence and one at a time. This view of computer has never been entirely true. At the micro-operation level, multiple control signals are generated at the same time. Instruction pipelining, at least to the extent of overlapping fetch and execute operation, has been

around for a long time. This approach is taken further with superscalar organization, which exploits instruction level parallelism.

1.2.1 Instruction Level Parallelism

Since 1985, all processors use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called instruction level parallelism (ILP) because the instructions can be executed in parallel. There are two basic approaches. Dynamic, hardware intensive approaches, and Static, compiler intensive approaches.

1.2.2 Thread Level Parallelism

Thread level parallelism allows multiple threads to share the functional units of a single processor in an overlapping faction. To permit this sharing, the processor must duplicate the independent state of each thread. For example, a separate copy of register file, a separate PC and a separate page table are required for each thread. The memory itself can be shared through the virtual memory mechanisms, which already support multiprogramming.

1.2.3 Processor Level Parallelism

The demand for ever faster computers seems to be insatiable. Instruction-level parallelism helps a little, but pipelining and superscalar operations rarely win more than a factor of five or ten. To get gains of 50, 100 or even more, the only way is to design computers with multiple CPUs. There are quiet a few approaches of parallel organization. For example: Symmetric Multiprocessors (SMPs), Cluster computers, and Non Uniform Memory Access computers (NUMA).

1.2.4 Chip Multiprocessors

Chip Multiprocessors (CMPs) use relatively simple single-thread processor cores to exploit only moderate amounts of parallelism within any one thread, while executing multiple threads in parallel across multiple processor cores. If an application cannot be effectively decomposed into threads, CMPs will be underutilized.

1.3 Literature Review

Bloch [Bloch 1959] and Bucholtz [Bucholtz 1962] describe a four stage pipeline and its engineering tradeoffs, including the use of ALU bypass. Kunkel and Smith

[Kunkel & Smith 1986] evaluate the impact of pipeline overhead and dependencies on the choice of optimal pipeline depth. Smith and Pleszkun [Smith & Pleszkun 1988] evaluate a variety of techniques for preserving precise exceptions. Weiss and smith [Weiss & smith 1984] evaluate a variety of hardware pipeline scheduling techniques and instruction issue techniques. Killian [Killian 1991] and Heinrich [Heinrich 1993] describe MIPS R4000 which was one of the first deeply pipelined microprocessors.

J Smith [Smith 1981] described a 2-bit branch prediction technique. Yeh and Patt [Yeh & Patt 1992, 1993] described multilevel predictors. Kaeli and Emma [Kaeli & Emma 1991] described return address prediction. The 2-bit Branch prediction improves implementation of branch predicition in super scalar processors.

Sohi [Sohi 1990] describes renaming and dynamic scheduling. Smith, Johnson and Horowitz [Smith, Johnson & Horowitz 1989] described the use of speculation a technique in multiple issue microprocessors. Dynamic scheduling and register renaming improves performance of heavy superscalar designs.

Agarwal et al. [Agarwal 1993] describes SPARCLE a block multithreaded processor. Laudon, Gupta and Horrowitz [Gupta & Horrowitz 1994] advocate fine grained multithreading. Yamamoto et al [Yamamoto 1994] proposed a design using dynamic scheduling to mix instructions from different threads. Tullsen et al [Tullsen 1996] addresses questions about the challenges of scheduling ILP versus TLP. Lo et al. [Lo 1997] gives an extensive discussion of SMT concept. Lo at al [Lo 1998] evaluated database performance on an SMT processor.

In 2000, IBM announced the first commercial chip with two general purpose processors on a single die, the Power4 processor. Each Power4 contains two Power3 microprocessors, a shared secondary cache and a chip to chip communication system.

In 1997 [Lance Hammond, Basem A. Nayfeh, Kunle Olukotun 1997] provided concrete evidence of the performance improvement possibilities using Single-Chip Multiprocessors.

[Jan Gray 2000] proposed on using FPGA based prototyping systems for teaching Micro Processor and Integrated Systems Design. Jan Gray. [Jan Gray 2001] proposed a simple RISC CPU and System-on-a-Chip on a single FPGA.

Hsiao-Ping Juan, Nancy D. Holmes Smita Bakshi, Daniel D. Gajski [Hsiao-Ping Juan, Nancy D. Holmes Smita Bakshi, Daniel D. Gajski 1992] proposed on Top Down

Modeling of RISC Processors in VHDL. [Takayuki Morimoto, Kazushi Saito, Hiroshi Nakamura, Taisuke Boku, Kisaburo Nakazawa] proposes a new hardware description language AIDL for Advance Processor Design. Makiko ITOH [Makiko ITOH 2000] proposed Synthesizable HDL Generation for Pipelined Processors from A Micro-Operation Description.

Prof. Lizy Kurian John [Lizy Kurian John 2002] provided research on Hardware Performance Evaluation: Techniques, Tools and Benchmarks.

Paul Kohout [Paul Kohout 2002] proposed on providing hardware support for real-time operating Systems.

Jochen Liedtke [Jochen Liedtke 1993] Discusses in detail about improving IPC by appropriate μ-Kernel Design

[Jochen Liedtke [Jochen Liedtke 1995] provides a detailed discussion and example implementation of high performance second generation μ-Kernel operating systems.

Jochen Liedtke, Hermann H¨artig, Michael Hohmuth, Sebastian Sch¨onberg, Jean Wolter [Liedtke, H¨artig, Hohmuth, Sch¨onberg, Wolter 1997] researched on the Performance of *μ*-Kernel-Based Systems.

Jochen Liedtke, Andreas Haeberlen, Yoonho Park, Lars Reuther, Volkmar Uhlig [Jochen Liedtke, Andreas Haeberlen, Yoonho Park, Lars Reuther, Volkmar Uhlig 2000] provide a detailed analysis and techniques for efficient Stub-Code for high performance μ -kernel based operating systems.

Jochen Liedtke [Jochen Liedtke 2001] discusses and evaluates the high performance *L4*Ka μ-kernel. *L4*Ka μ-kernel is completely written in assembly language and provides higher performance then the traditional 2^{nd} generation μ-kernel.

Benjamin Gamsa, Orran Krieger, Eric W. Parsons, Michael Stumm [Benjamin Gamsa, Orran Krieger, Eric W. Parsons, Michael Stumm 1995] discusses the Performance Issues in Multiprocessor Operating Systems.

Dawson R. Engler [1998] provides detailed description of ExoKernel operating systems.

Joshua A. Redstone, Susan J. Eggers and Henry M. Levy [Joshua A. Redstone, Susan J. Eggers & Henry M. Levy 2000] provided an Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture.

1.4 Academic CMPs

1.4.1 The Jamaica Project

The Jamaica project is investigating the design of chip multi-processors (CMPs) and their accompanying parallel software environments. CMP architectures have been widely accepted by many processor chip manufacturers as a solution to the design problems accompanying the scale-up to "billion transistor" chips. The rationale for this choice is that designing logic to interconnect multiple cores based on existing designs is *enormously* simpler than trying to build one core which will use all the available silicon.

Despite the widespread convergence on CMP as a promising design strategy, there are many issues yet to be resolved in the design of both the hardware and its accompanying software environment. In particular:

- The hardware must be able to efficiently support an operating system which can distribute execution of application code to all the available cores. Distribution and synchronization costs must be low and it must be easy to detect the presence of idle cores.
- The operating system must, in turn, rely upon advanced compiler technology to automate, as far as possible, this distribution of work. Most applications cannot feasibly be designed (or rewritten) to allow for all possible CMP configurations, coping with anything from, say, 2 to 64 cores.
- A dynamic parallelizing compiler is essential if the distribution problem is to be addressed. Both application and operating system code need to be optimized appropriate to the CMP configuration found at runtime or even recompiled on the fly using feedback directed recompilation.

All these elements of the CMP platform need to be designed together if the promise of CMP architectures is to be fulfilled. The Jamaica project is organized into three collaborating strands reflecting the interplay between computer architecture, compiler and operating system.

Advances in silicon technology have provided designers with more on-chip resources. However, this poses greater design problems in using the available silicon. The Jamaica project is focusing on the design of chip multi-processors. Benchmarks run on cycle accurate simulators allow development of prototypes and determination of optimal

configurations without incurring long development cycles. The project also carries out co-design of architectural features and compiler optimizations.

1.4.2 The Hydra Project

Hydra is a new microarchitecture that combines shared-cache multiprocessor architectures, innovative synchronization mechanisms, advanced integrated circuit technology and parallelizing compiler technology to produce breakthroughs in microprocessor cost/performance and parallel processor programmability. In Hydra, four high performance processors are integrated on a single die. Hydra represents a new way to build microprocessors that will demonstrate that it is possible for a multiprocessor to achieve better performance and better cost/performance than wide superscalar mircroarchitecture on sequential applications. Hydra will use a single chip shared cache architecture to fundamentally improve the communication bandwidth and latency between multiple processors. The shared-cache architecture takes advantage of the on-chip bandwidth to provide an order of magnitude improvement in interprocessor communication and synchronization latency compared to current-bus based multiprocessor implementations. This will improve parallel processing efficiency to the point that it is feasible to exploit fine-grained parallelism in sequential programs with a multiprocessor.

The shared-cache architecture and the support for specialized synchronization makes Hydra an ideal target for emerging parallelizing compiler technology. Most of this technology has focused on parallelizing applications into large grains so they will work efficiently on current multiprocessors. Hydra will have the ability to exploit fine grained parallelism and so will allow development of parallelizing compiler technology that is capable of extracting this sort of parallelism.

1.5 Commercial CMPs

1.5.1 IBM POWER4

This processor is meant for the maximum performance, for hi-end server and supercomputer market, designed for 32-processor SMP systems. Development of high-performance communication means for processors and memory was given much attention. The POWER4 has a high fault-tolerance: critical fails do not make the system hang; instead, interrupts are generated and processed by the system. The POWER4 was

developed for an efficient operation of commercial (server), scientific and technical applications. IBM Power/Power PC processors were divided into server and scientific ones - POWER and RS64. The POWER4 suits a wide range of hi-end applications and uses all topical performance boosting ways (within the PowerPC instruction set). We won't find there truncated caches and lacking FUs.

The POWER4 houses 2 processors each having an L1 cache for data and instructions. The die has a single L2 cache of 1450 KBytes controlled by 3 separate controllers connected to the cores via a CIU (Core Interface Unit). The controllers work independently and can process 32 bytes per clock. Each processor uses two separate 256-bit buses to connect the CIU for data fetching and data loading, as well as a separate 64-bit bus to save the results; the L2 cache has a bandwidth of 100 GBytes/s. The L2 cache's system looks well balanced and very powerful. Each processor has a special unit to support noncachable operation. The L3 controller and the memory's one are located on die as well. For connection with the L3 cache working at 1/3 of the processor's speed and with the memory there are two 128-bit buses operating at 1/3 of the processor's frequency.

1.5.2 Sun MAJC

With two identical and independent but cooperative processor cores, the MAJC-5200 is one of the first microprocessors to implement chip multiprocessing (CMP), though Sun prefers to classify the chip as a multiprocessor system on a chip (MPSOC). It will offer a relatively high clock rate (500 MHz), eight powerful function units, a unique geometry decompression engine, and copious amounts of off-chip data bandwidth. Future MAJC processors could incorporate hundreds of cores on the same die.

1.6 Microprocessor Design

20–50-MHz FPGA CPUs are perfect for many embedded applications. They can support custom instructions and function units, and can be reconfigured to enhance system-on-chip (SoC) development, testing, debugging, and tuning. FPGA systems offer high integration, short time-to-market, low NRE costs, and easy field updates of entire systems.

FPGA CPUs may also provide new answers to old problems. Consider one system, during self-test, its FPGA is configured as a CPU and it runs the tests. Later the

FPGA is reconfigured for normal operation as a hardwired signal processing datapath. The ephemeral CPU is free and saves money by eliminating test interfaces.

In the past, field programmable gate arrays (FPGAs) have been used to absorb glue logic, perform signal processing, and even to prototype system-on-chip (SoC) ASICs. Now with the advent of large, fast, cheap FPGAs, it is practical and cost-effective to skip the ASIC and ship volume embedded systems in a single FPGA plus off-chip RAM and ROM -- the FPGA implements all of the system logic including a processor core. A soft CPU core enables custom instructions and function units, and can be reconfigured to enhance SoC development, debugging, testing, and tuning. And if you control your own "cores" intellectual property (IP), you will be less at the mercy of the production and end-of-life decisions of chip vendors, and can ride programmable logic price and size improvement curves.

Processor and SoC design is not rocket science, and is no longer the exclusive realm of elite designers in large companies. FPGAs are now large and fast enough for many embedded systems, with soft CPU core speeds in the 33-100 MHz range. HDL synthesis tools and FPGA place-and-route tools are now fast and inexpensive, and open source software tools help to bridge the compiler chasm.

1.6.1 Overview of a Microprocessor

The Von Neumann model of a computer, pictured in Figure 1, consists of four main components: the input, the output, the memory, and the microprocessor.

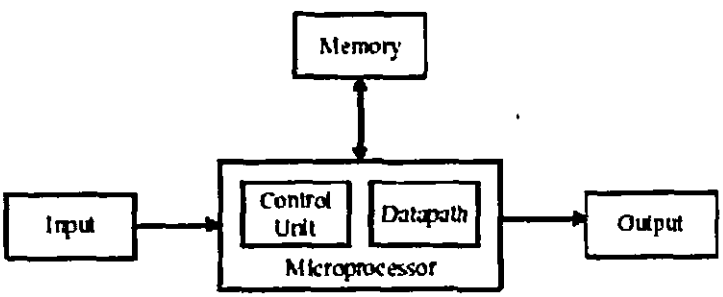


Figure 1.2 Von Neumann model of a computer.

The keyboard and mouse are examples of input devices. The CRT (cathode ray tube) and speakers are examples of output devices. The different types of memory, cache, read-only memory (ROM), random-access memory (RAM), and the disk drive are all considered as part of the memory box in the model. The focus in these seminars is on the

design of the digital circuitry of the microprocessor, the memory, and other supporting logical circuits, and their implementation on FPGAs.

The circuit for the microprocessor can be divided into two parts: the **datapath** and the **control unit** as shown in Figure 1. Figure 2 shows the details inside the control unit and the datapath. The datapath is responsible for the actual execution of all operations performed by the microprocessor, such as the addition inside the arithmetic logic unit (ALU). The datapath also includes the registers for the temporary storage of your data. The functional units inside the datapath (ALU, shifter, counter, etc.) and the registers are connected together with multiplexers and buses to form one unit, the datapath.

Even though the datapath is capable of performing all the operations of the microprocessor, it cannot, however, do it on its own. In order for the datapath to execute the operations automatically, the control unit is required. The control unit, also known as the controller, controls the operations of the datapath, and therefore, the operations of the entire microprocessor.

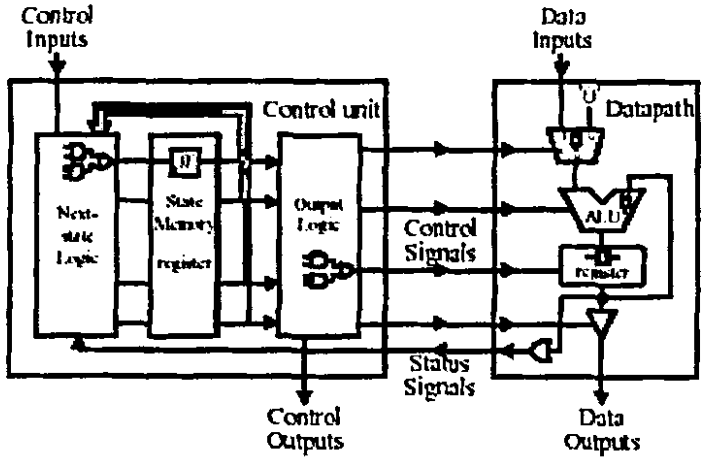


Figure 1.3 Internal parts of a microprocessor

The controller is a ***finite state machine*** (FSM) because it is a machine that executes by going from one state to another and the fact that there are only a finite number of states for the machine to go to. The controller is made up of three parts: the **next-state logic**, the **state memory**, and the **output logic**. The purpose of the state memory is to remember the current state that the FSM is in. The next-state logic is the circuit for determining what the next state should be for the machine. And the output logic is the circuit for generating the actual control signals for controlling the datapath.

1.7 Field programmable gate arrays

Field programmable gate arrays (FPGAs) are digital *integrated circuits (ICs)* that contain configurable (programmable) blocks of logic along with configurable interconnects between these blocks. Design engineers can configure (program) such devices to perform a tremendous variety of tasks. Depending on the way in which they are implemented, some FPGAs may only be programmed a single time, while others may be reprogrammed over and over again. The "field programmable" portion of the FPGA's name refers to the fact that its programming takes place "in the field" (as opposed to devices whose internal functionality is hardwired). This may mean that FPGAs are configured in the laboratory, or it may refer to modifying the function of a device resident in an electronic system that has already been deployed in the outside world. If a device is capable of being programmed while remaining resident in a higher-level system, it is referred to as being *in-system programmable*.

1.8 A Xilinx logic cell

The core building block in a modern FPGA from Xilinx is called a *logic cell (LC)*. Among other things, an LC comprises a 4-input LUT (which can also act as a 16 × 1 RAM or a 16-bit shift register), a multiplexer, and a register.

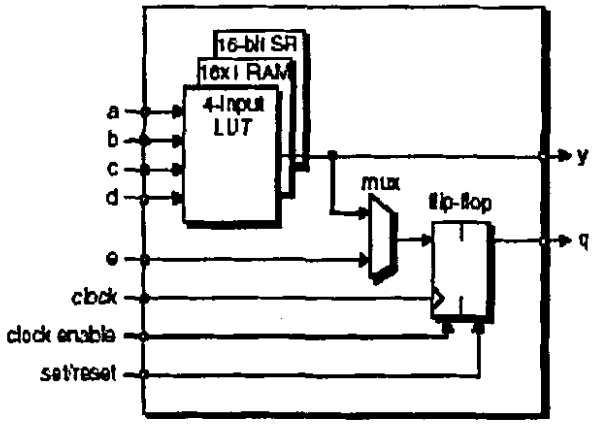


Figure 1.4 Architecture of a XILINX LC

In addition to the LUT, MUX, and register, the LC also contains a smattering of other elements, including some special fast carry logic for use in arithmetic operations.

1.8.1 Slicing and dicing

The next step up the hierarchy is what Xilinx calls a *slice*. A slice contains two logic cells as shown below.

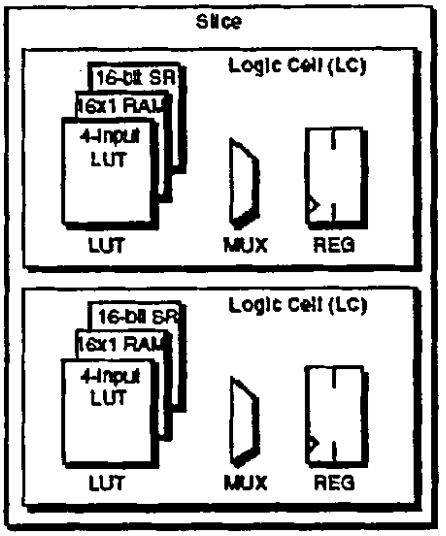


Figure 1.5 A slice containing two logic cells.

1.8.2 CLBs

And moving one more level up the hierarchy, we come to what Xilinx calls a *configurable logic block (CLB)*. Using CLBs as an example, some Xilinx FPGAs have two slices in each CLB, while others have four. A CLB equates to a single logic block in our original visualization of "islands" of programmable logic in a "sea" of programmable interconnect. There is also some fast programmable interconnect within the CLB. This interconnect is used to connect neighboring slices.

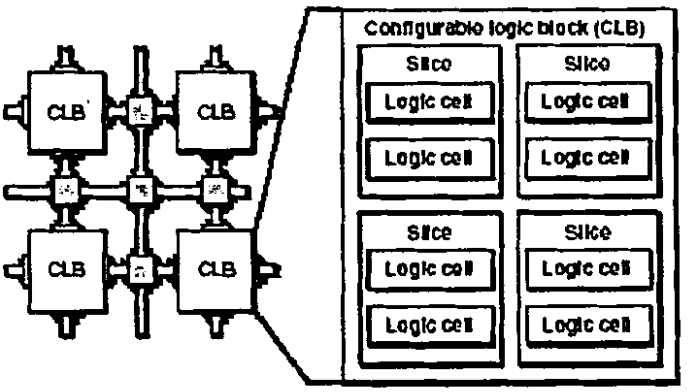


Figure 1.6 A CLB containing four slices (the number of slices depends on the FPGA family).

The reason for having this type of logic-block hierarchy, LC→ Slice (with two LCs)→ CLB (with four slices), is that it is complemented by an equivalent hierarchy in the interconnect. Thus, there is fast interconnect between the LCs in a slice, then slightly slower interconnect between slices in a CLB, followed by the interconnect between CLBs. The idea is to achieve the optimum trade-off between making it easy to connect things together without incurring excessive interconnect-related delays.

1.8.3 Distributed RAMs and shift registers

We previously noted that each 4-bit LUT can be used as a 16 × 1 RAM. Assuming the four-slices-per-CLB configuration all of the LUTs within a CLB can be configured together to implement the following:

1. Single-port 16 × 8 bit RAM
2. Single-port 32 × 4 bit RAM
3. Single-port 64 × 2 bit RAM
4. Single-port 128 × 1 bit RAM
5. Dual-port 16 × 4 bit RAM
6. Dual-port 32 × 2 bit RAM
7. Dual-port 64 × 1 bit RAM

Alternatively, each 4-bit LUT can be used as a 16-bit shift register. In this case, there are special dedicated connections between the logic cells within a slice and between the slices themselves that allow the last bit of one shift register to be connected to the first bit of another without using the ordinary LUT output. This allows the LUTs within a single CLB to be configured together to implement a shift register containing up to 128 bits..

1.8.4 Embedded RAMs

A lot of applications require the use of memory, so FPGAs now include relatively large chunks of embedded RAM called *e-RAM* or *block RAM*. Depending on the architecture of the component, these blocks might be positioned around the periphery of the device, scattered across the face of the chip in relative isolation, or organized in columns, as shown in Figure below.

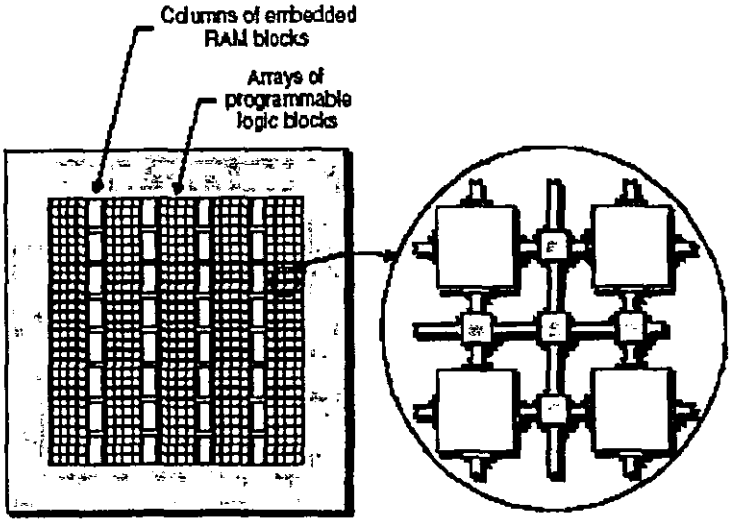


Figure 1.7 Bird's-eye view of chip with columns of embedded RAM blocks.

Depending on the device, such a RAM might be able to hold anywhere from a few thousand to tens of thousands of bits. Furthermore, a device might contain anywhere from tens to hundreds of these RAM blocks, thereby providing a total storage capacity of a few hundred thousand bits all the way up to several million bits. Each block of RAM can be used independently, or multiple blocks can be combined together to implement larger blocks. These blocks can be used for a variety of purposes, such as implementing standard single- or dual-port RAMs, *first-in first-out (FIFO)* functions, state machines, and so forth.

1.9 HDL Based Design Flow for FPGAs

The idea behind a hardware description language is, perhaps not surprisingly, that you can use it to describe hardware. In a wider context, the term *hardware* is used to refer to any of the physical portions of an electronics system, including the ICs, printed circuit boards, cabinets, cables, and even the nuts and bolts holding the system together. In the context of an

HDL, however, "hardware" refers only to the electronic portions (components and wires) of ICs and printed circuit boards. In the early days of electronics, almost anyone who created an EDA tool created his or her own HDL to go with it. Some of these were analog HDLs in that they were intended to rep resent circuits in the analog domain, while others were focused on representing digital functionality. Here, we are interested in HDLs only in the context of designing digital ICs in the form of FPGAs.

1.9.1 Different levels of abstraction

The functionality of a digital circuit can be represented at different levels of abstraction and that different HDLs support these levels of abstraction to a greater or lesser extent.

The lowest level of abstraction for a digital HDL would be the *switch level*, which refers to the ability to describe the circuit as a netlist of transistor switches. A slightly higher level of abstraction would be the *gate level*, which refers to the ability to describe the circuit as a netlist of primitive logic gates and functions. Both switch-level and gate-level netlists may be classed as *structural* representations. It should be noted, however, that "structural" can have different connotations because it may also be used to refer to a hierarchical block-level netlist in which each block may have its contents specified using any of the levels of abstraction.

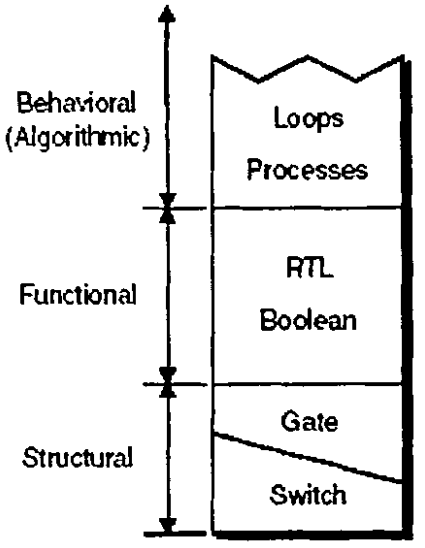


Figure 1.8 Different levels of abstraction.

The next level of HDL sophistication is the ability to support *functional* representations, which covers a range of constructs. At the lower end is the capability to describe a function using Boolean equations.

The functional level of abstraction also encompasses *register transfer level (RTL)* representations. The term *RTL* covers a multitude of manifestations, but the easiest way to understand the underlying concept is to consider a design formed from a collection of registers linked by combinational logic.

The highest level of abstraction sported by traditional HDLs is known as *behavioral*, which refers to the ability to describe the behavior of a circuit using abstract constructs like loops and processes. This also encompasses using algorithmic elements like adders and multipliers in equations.

1.9.2 A simple HDL-based FPGA flow

HDL-based flows featuring logic synthesis technology became fully available in the FPGA world in the very early 1990s. The key feature of HDL-based design flows is their use of *logic synthesis* technology, which began to appear around the mid-1980s. These tools can accept an RTL representation of a design along with a set of timing constraints. In this case, the timing constraints are presented in a side-file containing statements along the lines of "the maximum delay from input X to output Y should be no greater than N nanoseconds". The logic synthesis application automatically converts the RTL representation into a mixture of registers and Boolean equations, performs a variety of minimizations and optimizations (including optimizing for area and timing), and then generates a gate-level netlist that can (or at least, should) meet the original timing constraints.

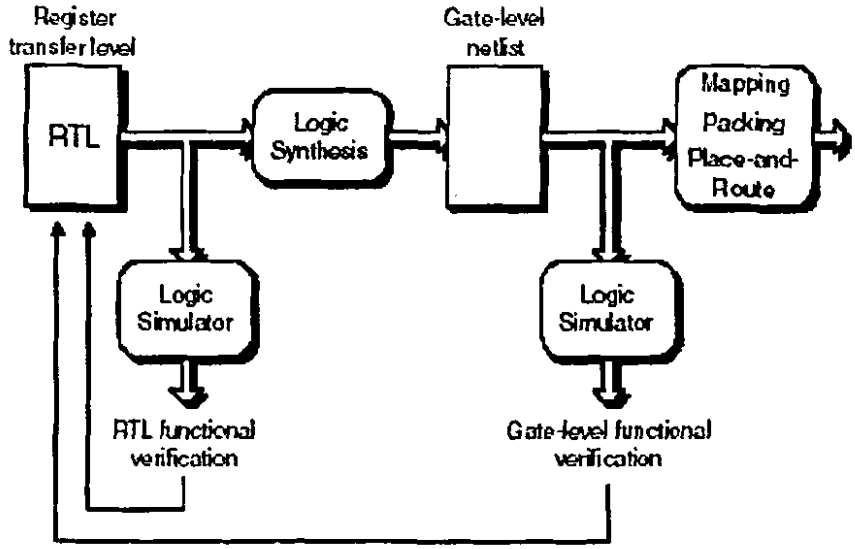


Figure 1.9 Simple HDL-based FPGA flow.

There are a number of advantages to this type of flow. First of all, the productivity of the design engineers rises dramatically because it is a lot easier to specify, understand, discuss, and debug the required functionality of the design at the RTL level of abstraction as opposed to working with reams of gate-level schematics. Also, logic simulators can run designs described in RTL much more quickly than their gate-level counterparts.

Once the synthesis tool have generated a gatelevel netlist, the gate-level netlist can be simulated to ensure its functional validity, and it can also be used to perform timing analysis based on estimated values for tracks and other circuit elements. The netlist can then be used to drive the FPGA's mapping, packing, and place-and-route software, following which a more accurate timing report can be generated using real-world (physical) values.

1.9.3 VHDL and VITAL

In 1980, the U.S. *Department of Defense (DOD)* launched the *very high speed integrated circuit (VHSIC)* program, whose primary objective was to advance the state of the art in digital IC technology. Under VHSIC, a project to develop a new hardware description language called *VHSIC HDL* (or *VHDL* for short) was launched in 1981. The first official release of VHDL occurred in 1985. DOD donated all rights to the VHDL language definition to the IEEE in 1986. After making some modifications to address a few known problems, VHDL was released as official standard IEEE 1076 in 1987. The language was further extended in a 1993 release and again in 1999 giving VHDL-2001.

As a language, VHDL is very strong at the functional (Boolean equation and RTL) and behavioral (algorithmic) levels of abstraction, and it also supports some system-level design constructs. However, VHDL is a little weak when it comes to the structural (switch and gate) level of abstraction, especially with regard to its delay modeling capability. It quickly became apparent that VHDL had insufficient timing accuracy to be used as a sign-off simulator. For this reason, the VITAL initiative was launched at the *Design Automation Conference (DAC)* in 1992. *VHDL Initiative toward ASIC Libraries (VITAL)* was an effort to enhance VHDL's abilities for modeling timing in ASIC and FPGA design environments. The end result encompassed both a library of ASIC/FPGA primitive functions and an associated method for back-annotating delay information into these library models, where this delay mechanism was based on the same underlying tabular format used by Verilog.

1.10 FPAG Prototyping Kit

For this project BurchED FPGA boards and accessories have been selected because of their flexibility and large capacity FPGA. The BurchED system consists of following components.

1.10.1 B5-X300 FPGA Board

- 300K gate Xilinx SpartanIIE device
- Access to all FPGA user I/Os
- Works with the Xilinx ISE design software
- Complete stand-alone system, including programming cable
- JTAG and serial mode configuration of the FPGA
- 1 to 100MHz header-programmable oscillator onboard

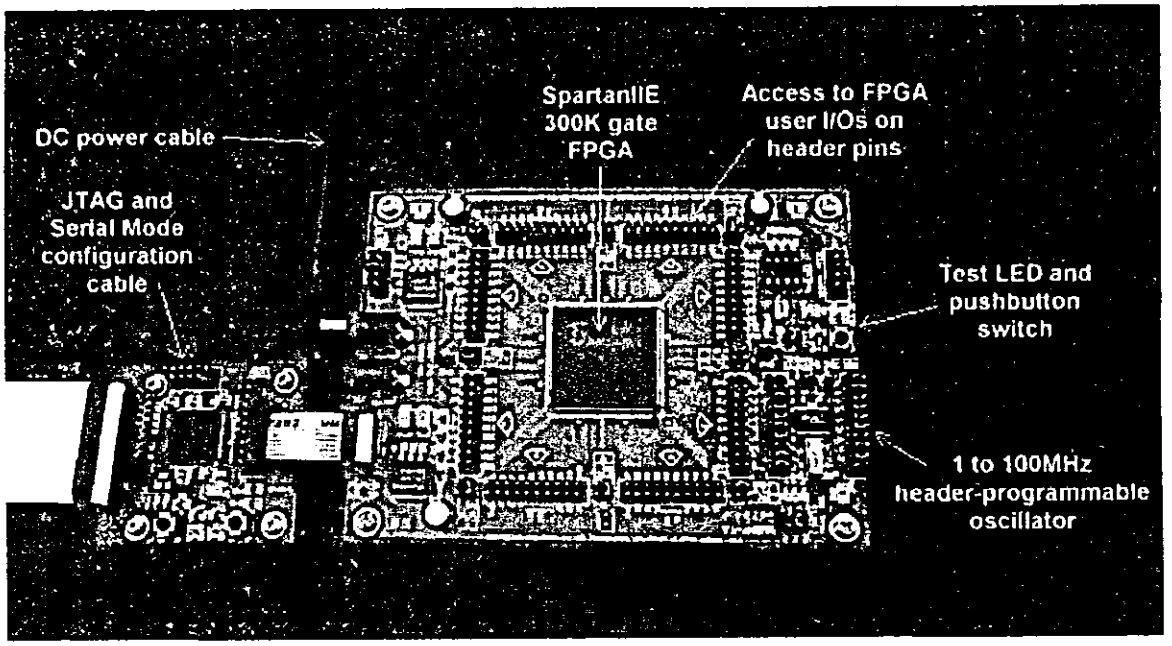


Figure 1.10: The B5-X300, B5-X-Advanced-Download-Cable is included.

1.10.2 B5-SRAM plug-on module

- 2 MBits of very fast 15ns static RAM
- Accessible as 128K x 16, or 256K x 8
- Large storage, external to the FPGA, for data, code, images

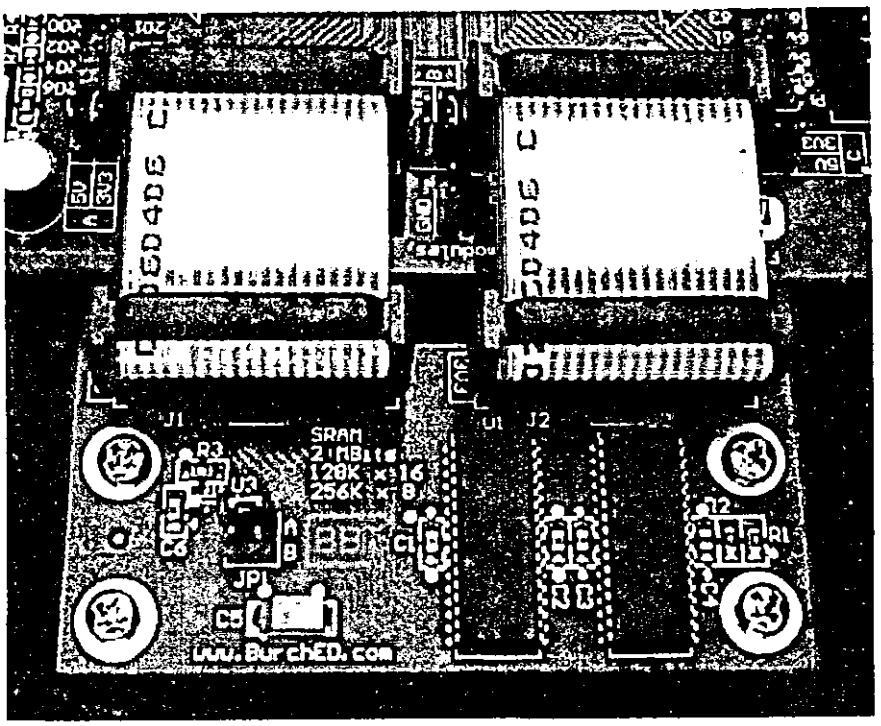


Figure 1.11: B5-SRAM

1.10.3 B5-Peripheral-Connectors plug-on module

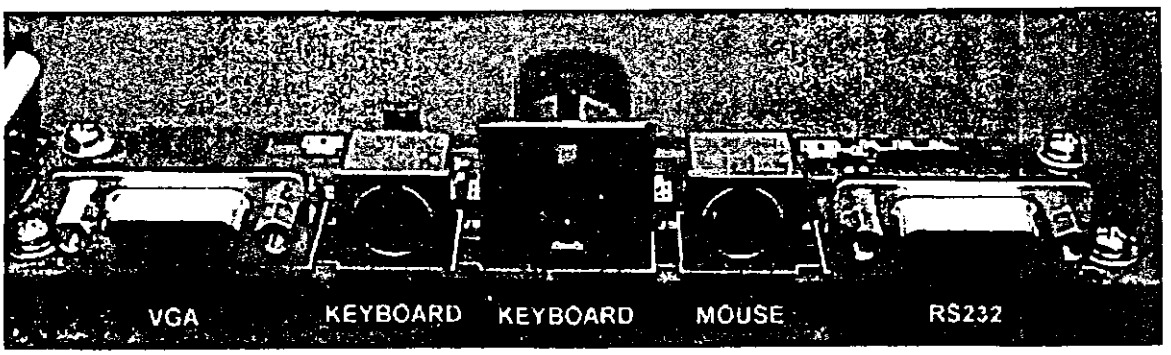


Figure 1.12: B5-Peripheral-Connectors

- Connect standard PC peripherals to FPGA
- VGA output, with 4 level resistor DAC on each of R, G and B
- Keyboard connectors - PS2 or 5-pin DIN
- Mouse connector - PS2
- RS232 level converter onboard
- DB9 RS232 connector - serial communications to a PC
- Piezo buzzer for "system beep", audible diagnostics and testing

1.10.4 B5-X-Flash-Config plug-on module

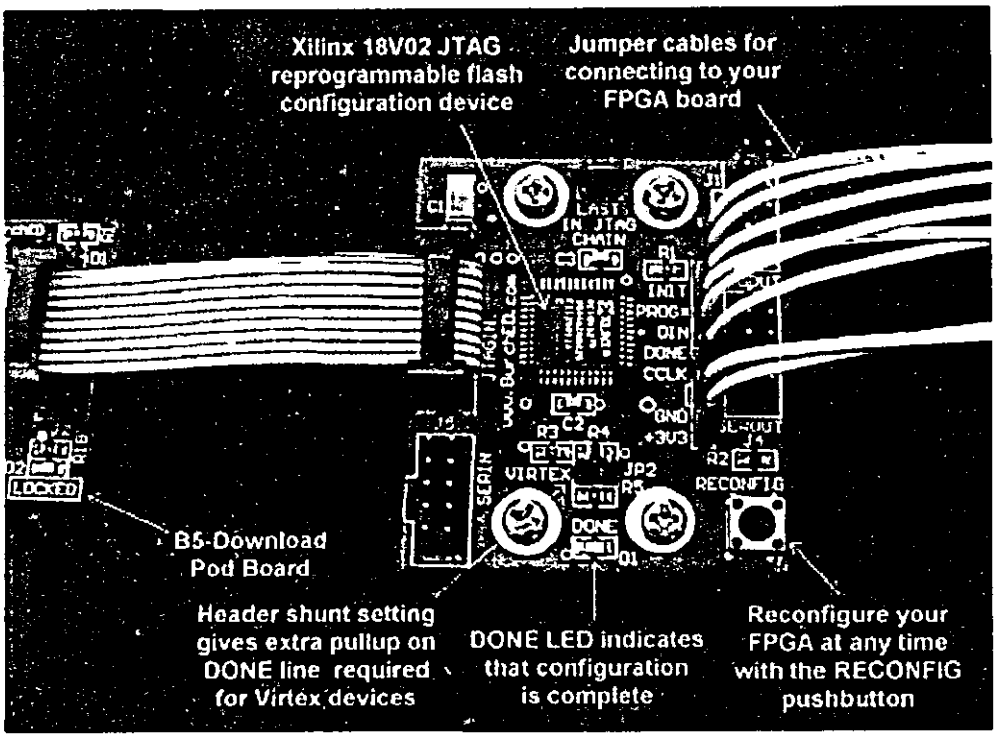


Figure 1.13: B5-X-Flash-Config

- Flash PROM automatically configures the FPGA on power up
- "Instant-on" configuration in less than 1 second
- JTAG reprogrammable
- Xilinx 18V02 flash configuration PROM onboard

2. Problem Definition

The basic problem is OS performance. In This section we present the problem and its proposed solution.

2.1 Operating Systems

As modern applications become increasingly dependent on multimedia, graphics, and data movement, they are spending an increasing fraction of their execution time in the operating system (OS) kernel, an area of the system almost completely ignored by traditional performance enhancement research. As an illustrative example, consider what must undoubtedly be today's leading server application: the web server. Web servers have been shown to spend over 85% of their CPU cycles running operating system code; in contrast, the near-ubiquitous SPEC benchmarks execute less than 9% of their instructions in the OS kernel. *[Aaron B. Brown 1997]*

For server-based environments, the operating system is a crucial component of the workload. Previous research suggests that database systems spend 30 to 40 percent of their execution time in the kernel, and measurements show that the Apache Web server spends over 75% of its time in the kernel. Operating systems are also known to be more demanding on the processor than typical user code. *[Joshua A. Redstone, Susan J. Eggers & Henry M. Levy 2000]*

Multi-server and component-based operating systems are promising architectural approaches for handling the ever increasing complexity of operating systems. Components or servers (and clients) communicate with each other through cross-domain method invocations. Such interface method invocations, if crossing protection boundaries, are typically implemented through the inter-process communication (IPC) mechanisms offered by a microkernel. Therefore, component interaction in such systems has to be highly efficient. Thus inter-process communication (IPC) by message passing is one of the central paradigms of most μ-kernel based and other client/server architectures. It helps to increase modularity, flexibility, security and scalability, and the key for distributed systems and applications. IPC has to be fast and effective, otherwise programmers will not be able to use remote procedure calls RPC, multithreading and multitasking adequately. Thus IPC performance is vital for modern operating systems, especially μ-kernel based ones. Surprisingly, most μ-kernels exhibit poor IPC performance. Since

context and user/kernel mode switches are central to IPC operation, reducing them is a critical factor in IPC performance improvement *[Jochen Liedtke 1993]*.

Also the following set of problems is caused by the operating systems.

i. Operating systems are huge programs that can overwhelm the cache and TLB due to code and data size, thereby causing severe performance penalty for user programs.

ii. Operating systems may impact branch prediction performance, because of frequent branches and infrequent loops.

iii. OS execution is often brief and intermittent, invoked by interrupts, exceptions, or system calls, and can cause the replacement of useful cache, TLB and branch prediction state for little or no benefit.

iv. The OS may perform spin-waiting, explicit cache/TLB invalidation, and other operations not common in user-mode code, affecting user code.

v. In current modularized kernels, every kernel invocation causes context switch, and in case of μ-kernels every call means a couple of context switches, thus wasting a considerable time in just switching processes.

vi. IPC-performance problems result from 64 bit architectures with large number of registers and register stack engines. In short, the large number of registers contributes to a potentially massive context (more than 2KB) to be stored on each thread context switch. This added context switch overhead may prove fatal to microkernel systems. A combined hardware/software solution is therefore required to reduce the amount of information stored *[Jochen Liedtke 2001]*.

vii. Overall, operating system code causes poor instruction throughput on a superscalar microprocessor.

For these reasons, ignoring the operating system (as is typically done in architectural & system simulations) may result in a misleading characterization of system-level performance. Even for applications that are not OS-intensive, the performance impact of the OS may be disproportionately large compared to the number of instructions the OS executes.

To overcome these problems many techniques have been used, but each has its disadvantages.

i. Monolithic kernels cause most of the above cited disadvantages, except internal OS function calls do not cause any context switches and System calls only cause a single context switch, thus making kernel calling and return fast.

ii. To avoid many of these problems μ-kernels have been designed, but they waste too much time in message passing thus giving poor performance.

iii. ExoKernels try to make OS extendable and try to reduce many known performance penalties.

A μ-kernel can provide higher layers with a minimal set of appropriate abstractions that are flexible enough to allow implementation of arbitrary operating systems and allow exploitation of a wide range of hardware. Choosing the right abstractions is crucial for both flexibility and performance. Some existing μ-kernels chose inappropriate abstractions, or too many or specialized and inflexible ones. Similar to optimizing code generators, μ-kernels must be constructed per processor and are inherently not portable. Basic implementation decisions, most algorithms and data structures inside μ-kernel are processor dependent. Their design must be guided by performance prediction and analysis. Besides inappropriate basic abstractions, the most frequent mistakes come from insufficient understanding of the combined hardware-software system or inefficient implementation *[Jochen Liedtke 1995].*

2.2 Microprocessor Architecture

Internally microprocessors have limited support for operating systems besides the features that are critical for current protected virtual memory based operating systems, like μ-kernel based operating systems. As we have seen that modern applications are spending an increasing fraction of their execution time in the operating system (OS) kernel. Techniques are required at the micro-architecture level to specifically improve OS performance.

Performance improvements at the micro-architecture level are only due to superscalar architecture, speculative execution, speculative loading, branch prediction, Simultaneous Multithreading, and Explicitly Parallel Instruction Computing etc. All these techniques generally improve performance of executing code but are not intentionally designed to improve OS performance.

At the multi-processor level performance improvement is due to SMP, NUMA or clustering. In each of these techniques the processing nodes are either running a copy of the kernel or the whole OS. None of these are aimed at improving OS performance directly. Rather OS problems mentioned at section 2.1 appear at each node and further set of problems appear due to multiple copies of the OS running simultaneously.

Integrated circuit processing technology offers increasing integration density, which fuels microprocessor performance growth. It is becoming possible to integrate a billion transistors on a reasonably sized silicon chip. At this integration level, it is necessary to find parallelism to effectively utilize the transistors. Currently, processor designs dynamically extract parallelism with these transistors by executing many instructions within a single, sequential program in parallel. Future performance improvements will require processors to be enlarged to execute more instructions per clock cycle. However, reliance on a single thread of control limits the parallelism available for many applications, and the cost of extracting parallelism from a single thread is becoming prohibitive. This cost manifests itself in numerous ways, including increased die area and longer design and verification times. In general, we see diminishing returns when trying to extract parallelism from a single thread.

2.3 Proposed System

Amdahl's law tells us that if we want modern applications to run quickly, the operating system must run quickly as well. Since traditional performance models essentially ignore the operating system and modern OS-dependent applications, a need has arisen for new tools and methodologies that direct their attention at the performance of the OS kernel. *[Aaron B. Brown 1997]*

The demand for ever faster computer systems seems to be insatiable. Instruction-level parallelism helps a little, but pipelining and superscalar operations rarely win more than a factor of five or ten. To get gains of 50, 100 or even more, the only way is to design computers with multiple CPUs. Thus high level of gain is only promised by parallelism at the processor level. Traditionally the processor level parallelism has used discrete processors. Making one processor master and run the OS is attractive as it solves most of the previously cited problems, but is prone to the bus latencies and hence poor performance.

As mentioned earlier OS kernel workload has significantly increased, especially server based applications are putting heavy loads on the kernel. What must be realised is that we have a huge potential for performance improvement. If some how the kernel runs on an independent processor and the user code runs on another, without any bus latencies, this master-slave processor architecture can improve performance significantly. This will also open doors for future operating system improvements due to available processor power at the disposal of the OS.

Researchers have proposed two microarchitectures that exploit multiple threads of control: simultaneous multithreading (SMT) and chip multiprocessors (CMP). From a purely architectural point of view, the SMT processor's flexibility makes it superior. However, the need to limit the effects of interconnect delays, which are becoming much slower than transistor gate delays, will also drive the billion-transistor chip design. Interconnect delays will force the microarchitecture to be partitioned into small, localized processing elements. For this reason, the CMP is much more promising because it is already partitioned into individual processing cores. Because these cores are relatively simple, they are amenable to speed optimization and can be designed relatively easily *[Lance Hammond].*

Programmers must find thread level parallelism in order to maximize CMP performance. With current trends in parallelizing compilers, multithreaded operating systems, and awareness of programmers about how to program parallel computers, this problem should prove less daunting in future. Additionally, having all of the CPUs on a single chip allows designers to exploit thread-level parallelism even when threads communicate frequently. This has been a limiting factor on today's multichip multiprocessors, preventing some parallel programs from attaining speedups. The low communication latencies inherent in single-chip microarchitecture allow speedup to occur across a wide range of parallelism *[Lance Hammond].*

Therefore a new microprocessor architecture has been proposed that is specifically designed to improve OS performance significantly as described below.

2.5 Design of Proposed Microprocessor.

In current multiprocessor architectures multiple independent microprocessors form a system, and communicate with each other over a system buss. Each runs a copy of the OS kernel, or, as in asymmetric multiprocessing, a single microprocessor runs the OS

and acts as a master and controls the remaining microprocessors. This technique has an inherent disadvantage of OS overload. Also the techniques used in SMP or NUMA are indispensable and must be used in multiprocessor architectures, neither has specific support for OS kernels. We will use a different technique, a modified form of chip multiprocessors (CMP). The new microprocessor architecture consists of two tightly coupled microprocessors. Both will be able to communicate with each other directly and will be fabricated as a single chip in the same package.

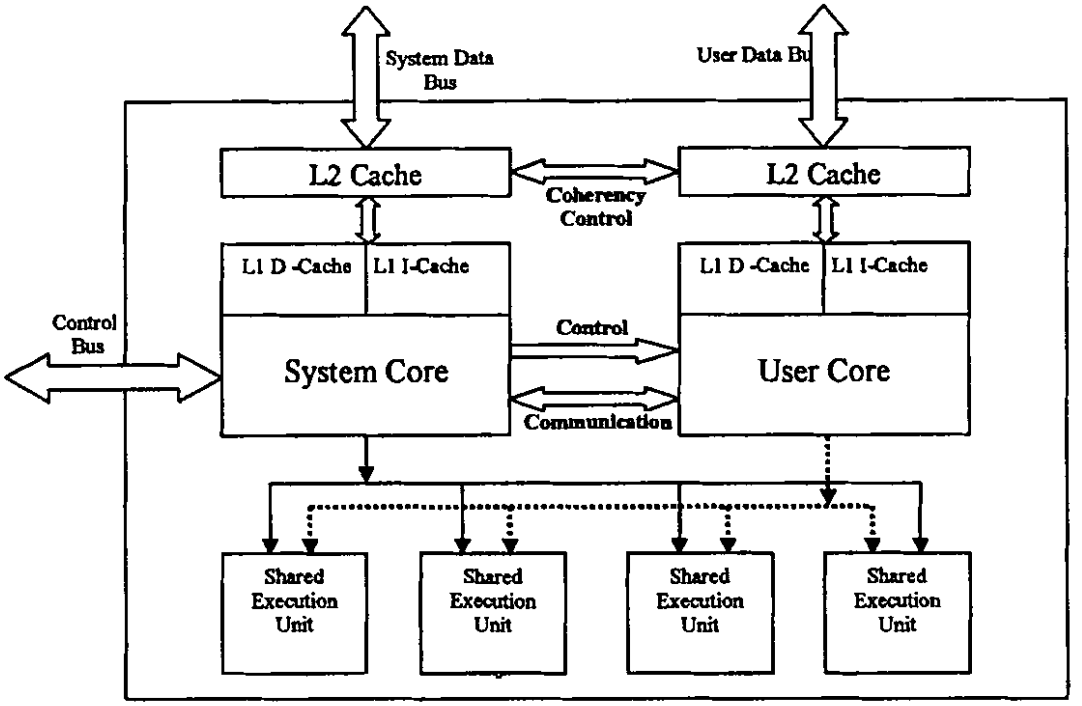


Figure 2.1 Proposed μ-Processor Architecture

One of the microprocessors will be the master processor and will implement privileged instruction as well as rest of the instruction set. Operating system alone will run on this microprocessor. The second microprocessor will only implement the non-privileged instructions. Complex processor execution units like floating point units and vector units will be shared among both processors to avoid over complex design and waste of resources as these are seldom used by the OS kernels. Proposed design is envisioned to remove earlier mentioned OS problems as follows.

1. *Operating systems are huge programs that can overwhelm the cache and TLB due to code and data size, thereby causing severe performance penalty for User programs.*

   The OS core has its own caches and TLB thus not affecting the user programs. Also the system core caches will have OS instructions and data structures in them

at all time thus removing cache contention faced by kernel in conventional CPUs.

2. *Operating systems may impact branch prediction performance, because of frequent branches and infrequent loops.*

   Since the system core will have its own prediction logic, thus the user branch prediction will not be effected. Also this issue will be handled in more detail during OS design.

3. *OS execution is often brief and intermittent, invoked by interrupts, exceptions, or system calls, and can cause the replacement of useful cache, TLB and branch prediction state for little or no benefit.*

   OS code will permanently reside on the system core thus avoiding the above to a certain extent.

4. *The OS may perform spin-waiting, explicit cache/TLB invalidation, and other operations not common in user-mode code, again effecting user code.*

   OS code will permanently reside on the system core only thus again avoiding the above for the user core.

5. *In current modularized kernels, every kernel invocation causes context switch, and in case of μ-kernels every call means a couple of context switches, thus wasting a considerable time in just switching processes.*

   The user and kernel code will be able to communicate directly by cross function calls that will have no context switch latency. Therefore context and memory space switches will be minimized by this architecture. This is one of the biggest advantage of this design.

6. *IPC-performance problems result from 64 bit architectures with there large number of registers and register stack engines. In short, the large number of registers contributes to a potentially massive context (more than 2KB) to be stored on each thread context switch. This added context switch overhead may prove fatal to microkernel systems. A combined hardware/software solution is therefore required to reduce the amount of information stored. [Jochen Liedtke 2001]*

   Since number of context and memory space switches will be dramatically reduced by this architecture, therefore the scale of this particular problem will be reduced.

But for the problem itself a few techniques are being developed including lazy context switching, such techniques will be explored in the proposed architecture.

7. *Overall, operating system code causes poor instruction throughput on a superscalar microprocessor.*

The system core will be designed specifically with OS code in mind. Thus, trying to avoid the previously mentioned poor OS performance. Also this issue will be handled in more detail in OS design.

3. Instruction Set Architecture

Much of the computer's architecture/organization is hidden from a HLL programmer. In the abstract sense, the programmer should not care about the underlying architecture. The instruction set is the boundary where the computer designer and the computer programmer can view the same machine.

The operation of CPU is determined by the instructions it execute, referred to as machine instructions or computer instructions. The collection of different instructions that the CPU executes is referred to as the CPU's instruction set. Each instruction must contain the information required by the CPU for execution. Figure 3.1 Shows the steps involved in instruction execution and.

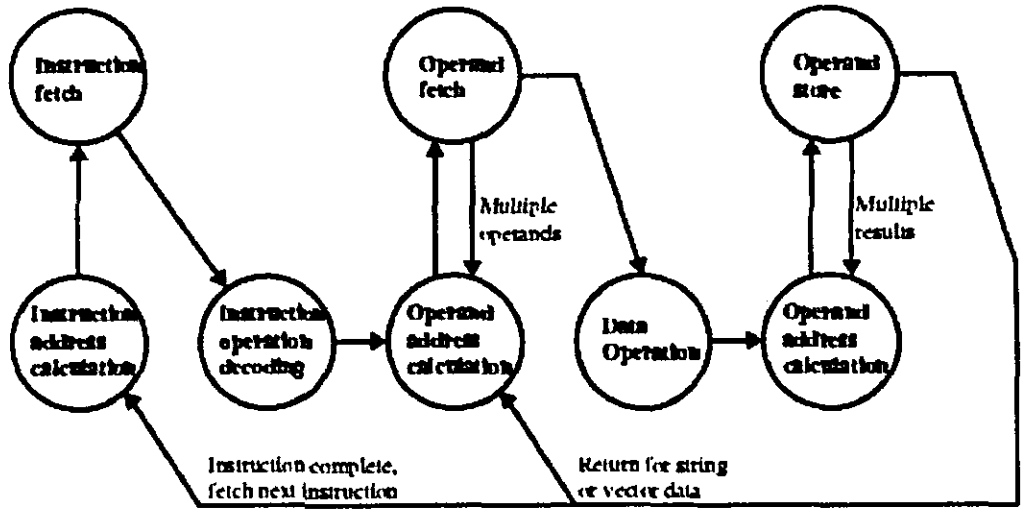


Figure 3.1 Instruction Cycle State Diagram [1]

The elements of machine instructions are as follows:

TH-4481

1	**Opcode:**	Opcode Specifies the operation to be performed. The OpCode is specified as a binary code.
2	**Function:**	Function code specifies what particular operation is to be performed or OpCodes specifying a range of operations.
3	**Source Operand:**	The Source Operand or Rs specifies the register that is to be used in Input/Output Operations.
4	**Source Operand 1:**	The Source Operand 1 or Rs1 specifies the register that is to be used as first operand in Arithmetic Operations.
5	**Source Operand 2:**	The Source Operand 2 or Rs2 specifies the register that is to be used as second operand in Arithmetic Operations.
6	**Destination Operand:**	The Source Operand 1 or Rs1 specifies the register that is to be used as first operand in Arithmetic Operations.

7	**Immediate-16:**	Immediate-16 Specifies 16 bit Immediate data.
8	**Immediate-20:**	Immediate-20 Specifies 20 bit Immediate data.
9	**Direction:**	Specifies the Jump direction.
9	**Port Address:**	Specifies the Port Address for I/O operations.

3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. MC-CPU processor is a "little endian" machine; this means the bytes of a word are numbered starting from the least significant byte. Figure 3.2 illustrates these conventions.

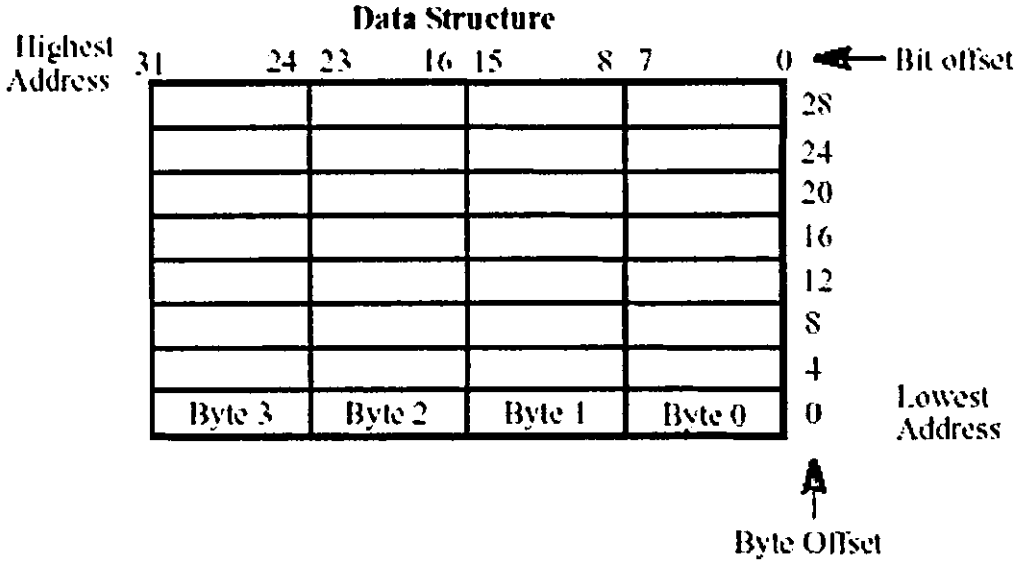


Figure 3.2 Bit and Byte Order

3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.

- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

3.3 Registers

There are 32 general purpose 32 Bit registers Ro to R31. A PC, Address Register, Stack Pointer and Flag Register. Also there are 32 special purpose 32 bit registers S0 to S31.

3.4 Instructions Implementation

The important design phase when designing a processor is to decide which instructions to include in the instruction set. The first step in deciding about instructions is the types of instruction formats. Following Instructions formats have been defined. All of the instructions have a 6-bit opcode which is used to determine the type of instruction to be executed. Each of the register specifications in all of the instructions is 5-bits wide, this means that the register file has 32 registers in it.

3.4.1 R-Type (Register to register)

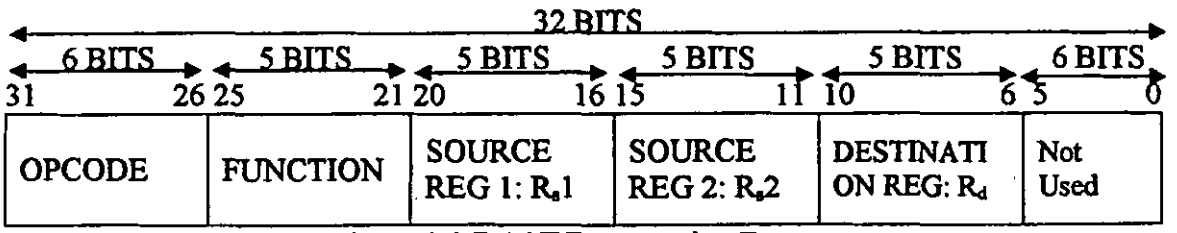


Figure 3.3 R-TYPE Instruction Format

In an R-type instruction the first 6-bit specification is the Instruction opcode. The following 5-bits specify function. These function bits specify what the *actual* instruction that will be performed is. This means for a single R-type opcode there can be up to 32 different instructions, as shown in figure 3.3.

In an R-type instruction the first 5-bit register specification is the source operand register 1 or Rs1; the following 5-bit register specifies second source operand R_s2. The third 5-bit register specifies the destination register Rd. The final 6-bits from bit 5 downto bit 0 in the instruction are not used.

3.4.2 I-Type (Register & Immediate Value)

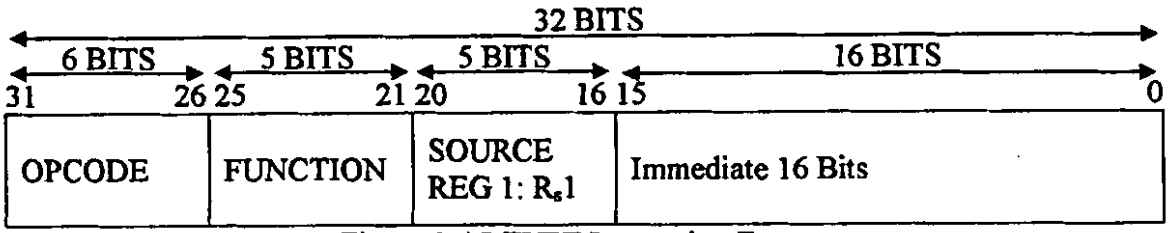


Figure 3.4 I-TYPE Instruction Format

In an I-type instruction the first 6-bit specification is the Instruction opcode. The following 5-bits specify function. These function bits specify the *actual* instruction that will be performed. This means for a single I-type opcode there can be up to 32 different instructions. In an I-type instruction only 5-bit register specification is the source operand register 1 or R_s1, which is also the destination register. The third 16-bit value is the immediate value, as shown in figure 3.4.

3.4.3 J-Type (JUMP USING PC +/- IMMEDIATE VALUE)

In a J-type instruction first 6-bit specification is the Instruction opcode. The following 5-bits specify function. These function bits specify the *actual* instruction that will be performed. This means for a single J-type opcode there can be up to 32 different instructions. In a J-type instruction the 1 bit field specification is the direction of jump i.e. forward or backwards. The third 20-bit value is the immediate value, as shown in figure 3.5.

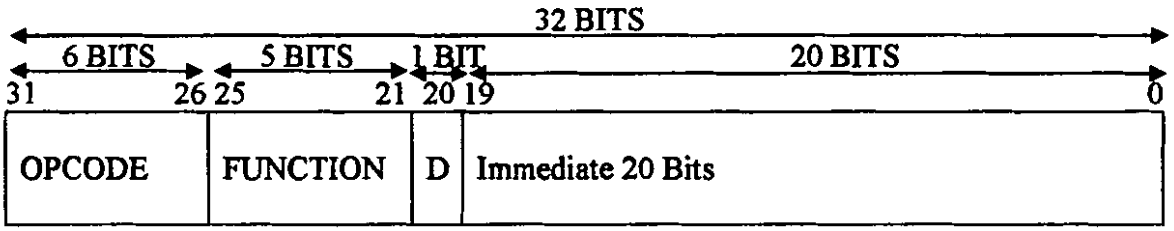


Figure 3.5 J-TYPE Instruction Format

3.4.4 N-Type (NO Operand)

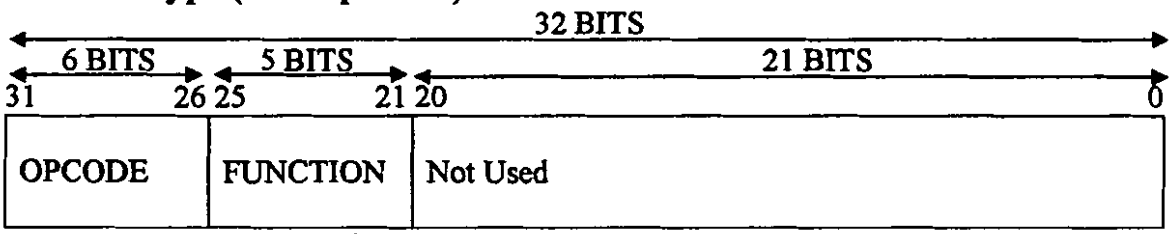


Figure 3.6 N-TYPE Instruction Format

In an N-type instruction the first 6-bit specification is the Instruction opcode. The following 5-bits specify function. These function bits specify the *actual* instruction that will be performed. This means for a single I-type opcode there can be up to 32 different instructions. In an N-type instruction the remaining 21 bits are not used, as shown in figure 3.6.

3.4.5 IO-Type

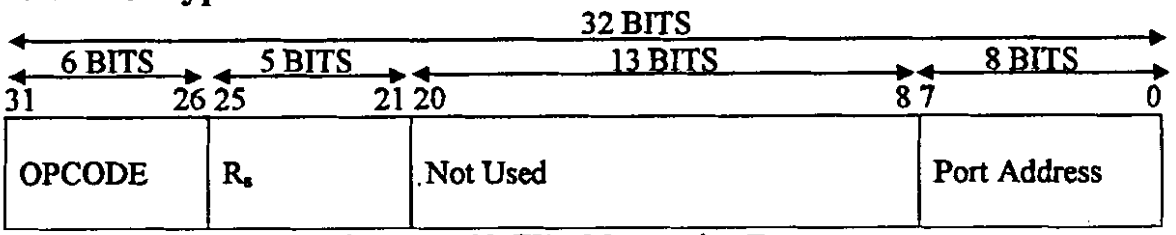


Figure 3.7 IO-TYPE Instruction Format

In an IO-type instruction the first 6-bit specification is the Instruction opcode. The following 5-bits specify the register. The next 13-bits are not used. The final 8 bits specify the Device and Port address, as shown in figure 3.7.

3.4.6 OI-Type

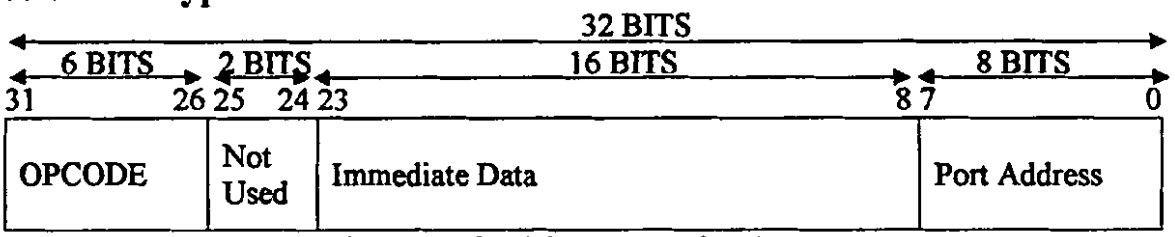


Figure 3.8 OI-TYPE Instruction Format

In an OI-type instruction the first 6-bit specification is the Instruction opcode. The following 2-bits are not used. The next 16-bits specify the immediate data to be sent to the output port. The final 8 bits specify the Device and Port address, as shown in figure 3.8.

3.5 Instruction OpCode Map

Instruction	OpCode	Function	Operands	Test Flag Position
No Operation				
NOP	000000	00000	—	0
Load to Rd From Memory Location Pointed By Rs				
LOAD	000001	00000	R_s1, R_d	1

Load from Memory Location Immediately following this Instruction in Memory				
LOADI	000010	00000	R_s1	2
Load to Register Using PC + Imm Data				
LOADIA	000011	00000	R_s1, Imm Address Value	3
Store Rd to Memory Location Pointed By R_s				
STORE	000100	00000	R_s1, R_d	4
Store Register Using PC + Imm Data				
STOREIA	000101	00000	R_s1, Imm Address Value	5
Move from R_s to R_d				
MOVE	000110	00000	R_s1, R_d	6
Move Immediate Data to Register				
MOVEI	000111	00000	R_s1, Imm Value	7
ALU Single Operand Instructions ALUS				
NOT	001000	00011	R_s1	8
IINC	001000	00111	R_s1	8
DEC	001000	01000	R_s1	8
ZERO	001000	01001	R_s1	8
ALU Double Operand Instructions ALUD				
AND	001001	00001	R_s1, R_s2, R_d	9
OR	001001	00010	R_s1, R_s2, R_d	9
XOR	001001	00100	R_s1, R_s2, R_d	9
ADD	001001	00101	R_s1, R_s2, R_d	9
SUB	001001	00110	R_s1, R_s2, R_d	9
ALU Double Operand With Immediate Data Instructions ALUDI				
ANDI	001010	00001	R_s1, Imm Data	10
ORI	001010	00010	R_s1, Imm Data	10
XORI	001010	00100	R_s1, Imm Data	10
ADDI	001010	00101	R_s1, Imm Data	10
SUBI	001010	00110	R_s1, Imm Data	10
MULTIPLY and DEVIDE Instructions MULDIV				
MUL	001011	00001	R_s1,R_s2,R_d	11

DIV	001011	00010	R_s1,R_s2,R_d	11
MULI	001011	00011	R_s1, Imm Data	11
DIVI	001011	00100	R_s1, Imm Data	11
Shift and Rotate Instructions 1-Bit SR				
SHL	001100	00001	R_s1	12
SHR	001100	00010	R_s1	12
ROTL	001100	00011	R_s1	12
ROTR	001100	00100	R_s1	12
Register Comparison Instructions RC				
EQ	001101	00001	R_s1, R_s2	13
NEQ	001101	00010	R_s1, R_s2	13
GT	001101	00011	R_s1, R_s2	13
GTE	001101	00100	R_s1, R_s2	13
LT	001101	00101	R_s1, R_s2	13
LTE	001101	00110	R_s1, R_s2	13
Register Comparison With Immediate Data Instructions RCI				
EQI	001110	00001	R_s1, Imm Data	14
NEQI	001110	00010	R_s1, Imm Data	14
GTI	001110	00011	R_s1, Imm Data	14
GTEI	001110	00100	R_s1, Imm Data	14
LTI	001110	00101	R_s1, Imm Data	14
LTEI	001110	00110	R_s1, Imm Data	14
Unconditional Branch to Memory Address Contained at the Next Memory Location				
BRANCHI	001111	00000	---	15
Conditional Branches to Memory Address Contained at the Next Memory Location BRANCHCI				
BRANCHEQI	010000	00001	R_s1, R_s2	16
BRANCHNEQI	010000	00010	R_s1, R_s2	16
BRANCHGTI	010000	00011	R_s1, R_s2	16
BRANCHGTEI	010000	00100	R_s1, R_s2	16
BRANCHLTI	010000	00101	R_s1, R_s2	16
BRANCHLTEI	010000	00110	R_s1, R_s2	16
Branch Unconditionally to PC + Immediate Data				
BRANCH	010001	00000	Imm Data	17
Branch Conditionally to PC + Immediate Data (BASED ON FLAG REGISTER) BRANCHC				
BRANCHEQ	010010	00001	Imm Data	18

BRANCHNEQ	010010	00010	Imm Data	18
BRANCHGT	010010	00011	Imm Data	18
BRANCHGTE	010010	00100	Imm Data	18
BRANCHLT	010010	00101	Imm Data	18
BRANCHLTE	010010	00110	Imm Data	18
BRANCHZ	010010	00111	Imm Data	18
Input Data from Input Port to Register				
INPUT	010011	00000	R_d	19
Output Register Data to Output Port				
OUTPUT	010100	00000	R_s	20
Output Immediate Data to Output Port				
OUTPUTI	010101	00000	Imm Value	21
Push Register Data on Top of Stack				
PUSH	010110	00000	R_s1	22
PUSHS	010110	00001	R_s1	22
POP Data to Register from Top of Stack				
POP	010111	00000	R_d	23
POPS	010111	00001	R_d	23
Function Call				
CALL	011000	00000	Imm Address Value	24
CALLS	011000	00001	Imm Address Value	24
Return from Function Call				
RETURNC	011001	00000	---	25
RETURNCS	011001	00001	---	25
Return From Interrupt				
RETURNI	011010	00000	---	26
Enable or Disable Interrupts EDI				
EI	011011	00001	---	27
DI	011011	00010	---	27
Trap to OS Kernel				
TRAP	011100	00000	Imm Address Value	28
Wait for Interrupt				
WAITI	011101	00000	---	29

Wait for Time (Given)				
WAITT	011110	00000	Imm Time Value	30
Halt Execution				
HALT	011111	00000	---	31

Figure 3.9 Instruction OP-Code Map

3.6 Instruction Implementation Detail

The following sections explain the implementation details of each instruction.

3.6.1 NOP

Instruction type:	N-Type
Opcode:	000000
Function Code:	000000
Operation:	No Operation

Figure 3.10 NOP Instruction

Description:

NOP performs no operation. It is used to consume a complete Fetch, Decode, and Execute Cycle. Used normally by compilers to align load stores. And eliminate pipeline hazards.

3.6.2 Load

Instruction type:	R-Type
Opcode:	000001
Function Code:	000000
Source Register 1:	Memory Address
Destination Register:	Destination for read data
Operation:	$R_d \leftarrow MEM[R_s1]$

Figure 3.11 LOAD Instruction

Description:

Load uses register indirect memory addressing to load a 32 bit word from memory. Memory address is provided in the Source register 1. Read data is placed in the destination register

3.6.3 LoadI

Instruction type:	I-Type

Opcode:	000010
Function Code:	000000
Destination Register:	Destination for read data
Operation:	$R_s1 \leftarrow$ MEM [PC+1]

Figure 3.12 LOADI Instruction

Description:

LoadI uses address contained at the next memory location pointed to by Program Counter + 1 to load a 32 bit word from memory. Read data is placed in the Destination Register.

3.6.4 LoadIA

Instruction type:	I-Type
Opcode:	000011
Function Code:	000000
Source Register 1:	Destination for read data
Immediate-16	Memory address
Operation:	$R_s1 \leftarrow$ MEM [Immediate-16]

Figure 3.13 LOADIA Instruction

Description:

LoadI uses address contained inside the instruction as 16 bit value to load a 32 bit word from memory. Memory address is padded to make it 32 bits. Read data is placed in the Source Register 1.

3.6.5 Store

Instruction type:	R-Type
Opcode:	000100
Function Code:	000000
Source Register 1:	Source Register Containing Data
Destination Register:	Destination Memory Address
Operation:	MEM $[R_d] \leftarrow R_s1$

Figure 3.14 STORE Instruction

Description:

Store uses register indirect memory addressing to store a 32 bit word to memory. Memory address is provided in the destination register. Data to be stored is provided in Source Register 1.

3.6.6 StoreIA

Instruction type:	I-Type
Opcode:	000101
Function Code:	000000
Source Register 1:	Source Register Containing Data
Immediate-16	Memory address
Operation:	MEM [Immediate-16] ← R_s1

Figure 3.15 STOREIA Instruction

Description:

StoreIA uses address contained inside the instruction as 16 bit value to store 32 bit word to memory. Memory address is padded to make it 32 bits. Data to be stored is provided in Source Register 1.

3.6.7 Move

Instruction type:	R-Type
Opcode:	000110
Function Code:	000000
Source Register 1:	Source Register Containing Data
Destination Register:	Destination Register for data
Operation:	R_d ← R_s1

Figure 3.16 MOVE Instruction

Description:

Move instruction is used to move data between two registers.

3.6.8 MoveI

Instruction type:	I-Type
Opcode:	000111
Function Code:	000000
Source Register 1:	Destination Register for data
Immediate-16	Immediate Data
Operation:	R_s1 ← Immediate-16

Figure 3.17 MOVEI Instruction

Description:

Move instruction is used to move immediate data to a register. Immediate Data is padded to make it 32 bits.

3.6.9 ALUS

Instruction type:	R-Type
Opcode:	001000
Function Code:	NOT : 00011 INC : 01111 DEC : 01000 ZERO : 01001
Source Register 1:	Register to perform operation on
Operation:	NOT : $R_s1 \leftarrow$ NOT R_s1 INC : $R_s1 \leftarrow$ INC R_s1 DEC : $R_s1 \leftarrow$ DEC R_s1 ZERO : $R_s1 \leftarrow$ ZERO R_s1

Figure 3.18 ALUS Instruction

Description:

ALUS or ALU Single Operand instructions use R_s1 as the source and destination register. 4 possible operations are specified by the function code.

3.6.10 ALUD

Instruction type:	R-Type
Opcode:	001001
Function Code:	AND : 00001 OR : 00010 XOR : 00100 ADD : 00101 SUB : 00110
Source Register 1:	Source Operand 1
Source Register 2:	Source Operand 1
Destination Register :	Destination for result
Operation:	AND : $R_d \leftarrow R_s1$ AND R_s2 OR : $R_d \leftarrow R_s1$ OR R_s2 XOR : $R_d \leftarrow R_s1$ XOR R_s2 ADD : $R_d \leftarrow R_s1$ ADD R_s2 SUB : $R_d \leftarrow R_s1$ SUB R_s2

Figure 3.19 ALUD Instruction

Description:

ALUD or ALU Double Operand instructions use R_s1 and R_s2 as the source Operands and R_d as destination register. 5 possible operations are specified by the function code.

3.6.11 ALUDI

Instruction type:	I-Type
Opcode:	001010
Function Code:	AND : 00001 OR : 00010 XOR : 00100 ADD : 00101 SUB : 00110
Source Register 1:	Source Operand 1
Immediate-16	Immediate Data
Operation:	AND : $R_s1 \leftarrow R_s1$ AND Imm Data OR : $R_s1 \leftarrow R_s1$ OR Imm Data XOR : $R_s1 \leftarrow R_s1$ XOR Imm Data ADD : $R_s1 \leftarrow R_s1$ ADD Imm Data SUB : $R_s1 \leftarrow R_s1$ SUB Imm Data

Figure 3.20 ALUDI Instruction

Description:

ALUDI or ALU Double Operand instructions with Immediate data use R_s1 as the source and destination register. Immediate data provides the second operand. 5 possible operations are specified by the function code.

3.6.12 MULDIV

Instruction type:	R-Type / I-TYPE
Opcode:	001011
Function Code:	MUL : 00001 DIV : 00010 MULI : 00011 DIVI : 00100
Source Register 1:	Source Operand 1
Source Register 2:	Source Operand 1
Destination Register :	Destination for result
Immediate-16	Immediate Data
Operation:	MUL : $R_d \leftarrow R_s1$ MUL Rs2 DIV : $R_d \leftarrow R_s1$ DIV Rs2 MULI : $R_s1 \leftarrow R_s1$ MUL Imm Data DIVI : $R_s1 \leftarrow R_s1$ DIV Imm Data

Figure 3.21 MULDIV Instruction

Description:

MULDIV OR Multiply Divide instructions use R_s1 and R_s2 as the source Operands and R_d as destination register OR R_s1 as the source and destination

register and Immediate data as second operand. 4 possible operations are specified by the function code.

3.6.13 SR

Instruction type:	R-Type
Opcode:	001100
Function Code:	SHL : 00001 SHR : 00010 ROTL : 00011 ROTR : 00100
Source Register 1:	Register to perform operation on
Operation:	SHL : $R_s1 \leftarrow$ SHL R_s1 SHR : $R_s1 \leftarrow$ SHR R_s1 ROTL : $R_s1 \leftarrow$ ROTL R_s1 ROTR : $R_s1 \leftarrow$ ROTR R_s1

Figure 3.22 SR Instruction

Description:

SR or Shift Rotate Single Operand instructions use R_s1 as the source and destination register. 4 possible operations are specified by the function code.

3.6.14 RC

Instruction type:	R-Type
Opcode:	001101
Function Code:	EQ : 00001 NEQ : 00010 GT : 00011 GTE : 00100 LT : 00101 LTE : 00110
Source Register 1:	Source Operand 1
Source Register 2:	Source Operand 1
Flag Register	Destination for Boolean result
Operation:	EQ : FL-EQ $\leftarrow$ 1 IF $R_s1 = R_s2$ NEQ : FL-NEQ $\leftarrow$ 1 IF $R_s1 /= R_s2$ GT : FL-GT $\leftarrow$ 1 IF $R_s1 > R_s2$ GTE : FL-GTE $\leftarrow$ 1 IF $R_s1 >= R_s2$ LT : FL-LT $\leftarrow$ 1 IF $R_s1 < R_s2$ LTE : FL-LTE $\leftarrow$ 1 IF $R_s1 <= R_s2$

Figure 3.23 RC Instruction

Description:

RC or Register Compare Double Operand instructions use R_s1 and R_s2 as the source Operands and Flag Register as destination register. 6 possible operations are specified by the function code.

3.6.15 RCI

Instruction type:	I-Type
Opcode:	001110
Function Code:	EQ : 00001 NEQ : 00010 GT : 00011 GTE : 00100 LT : 00101 LTE : 00110
Source Register 1:	Source Operand 1
Immediate-16	Immediate Data
Operation:	EQ : FL-EQ ← 1 IF R_s1 = Imm Data NEQ : FL-NEQ ← 1 IF R_s1 /= Imm Data GT : FL-GT ← 1 IF R_s1 > Imm Data GTE : FL-GTE ← 1 IF R_s1 >= Imm Data LT : FL-LT ← 1 IF R_s1 < Imm Data LTE : FL-LTE ← 1 IF R_s1<= Imm Data

Figure 3.24 RCI Instruction

Description:

RC or Register Compare Single Operand instructions use R_s1 and Immediate Data as the source Operands and Flag Register as destination register. 6 possible operations are specified by the function code.

3.6.16 BranchI

Instruction type:	N-Type
Opcode:	001111
Function Code:	000000
Operation:	PC ← MEM [PC+1]

Figure 3.25 BRANCHI Instruction

Description:

Unconditional Branch to the memory address contained at the next memory location pointed by PC+1.

3.6.17 BranchCI

Instruction type:	R-Type
Opcode:	010000
Function Code:	BranchEQI : 00001 BranchNEQI : 00010 BranchGTI : 00011 BranchGTEI : 00100 BranchLTI : 00101 BranchLTEI : 00110
Source Register 1:	Source Operand 1
Source Register 2:	Source Operand 1
Flag Register	Destination for Boolean result
Operation:	BranchEQI : PC ← MEM [PC+1] IF $R_s1 = R_s2$ BranchNEQI : PC ← MEM [PC+1] IF $R_s1 /= R_s2$ BranchGTI : PC ← MEM [PC+1] IF $R_s1 > R_s2$ BranchGTEI : PC ← MEM [PC+1] IF $R_s1 >= R_s2$ BranchLTI : PC ← MEM [PC+1] IF $R_s1 < R_s2$ BranchLTEI : PC ← MEM [PC+1] IF $R_s1 <= R_s2$

Figure 3.26 BRANCHCI Instruction

Description:

Conditional Branch to the memory address contained at the next memory location pointed by PC+1. BranchCI Double Operand instructions use R_s1 and R_s2 as the source Operands and branches on the base of comparison result. 6 possible operations are specified by the function code.

3.6.18 Branch

Instruction type:	J-Type
Opcode:	010001
Direction D	Jump Direction
Immediate-20	20 Bit Immediate Address value
Function Code:	000000
Operation:	PC ← MEM [Imm Address]

Figure 3.27 BRANCH Instruction

Description:

Unconditional Branch to the Immediate memory address.

3.6.19 BranchC

Instruction type:	J-Type
Opcode:	010010
Function Code:	BranchEQI : 00001 BranchNEQI : 00010 BranchGTI : 00011 BranchGTEI : 00100 BranchLTI : 00101 BranchLTEI : 00110 BranchZ : 00111
Immediate-20	20 Bit Immediate Address value
Direction D	Jump Direction
Flag Register	Used for Jump Conditions
Operation:	BranchEQ : PC ← MEM PC+1] IF FL-EQ = 1 BranchNEQ : PC ← MEM PC+1] IF FL-NEQ =1 BranchGT : PC ← MEM PC+1] IF FL-GT = 1 BranchGTE : PC ← MEM PC+1] IF FL-GTE = 1 BranchLT : PC ← MEM PC+1] IF FL-LT = 1 BranchLTE : PC ← MEM PC+1] IF FL-LTE = 1

Figure 3.28 BRANCHC Instruction

Description:

Conditional Branch to the immediate memory address. BranchC uses the flag register to make conditional jumps. Therefore a register compare or ALU operation should have been performed before this instruction. 6 possible operations are specified by the function code.

3.6.20 Input

Instruction type:	IO-Type
Opcode:	010011
Source Operand:	Destination Register for Input Data
Port Address:	8 bit port address
Operation:	R_s ← INPUT[Port Address]

Figure 3.29 INPUT Instruction

Port Address Format

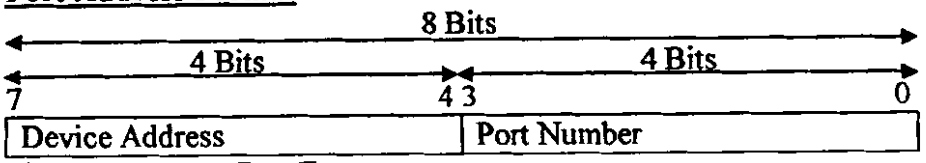


Figure 3.30 Input Port Format

Total Devices: 16
Port per Device: 16

Description:

Reads Data from the input port and places it into the Source register R_s.

3.6.21 Output

Instruction type:	IO-Type
Opcode:	010100
Source Operand:	Source Register for output Data
Port Address:	8 bit port address
Operation:	OUTPUT[Port Address] ← R_s

Figure 3.32 OUTPUT Instruction

Port Address Format

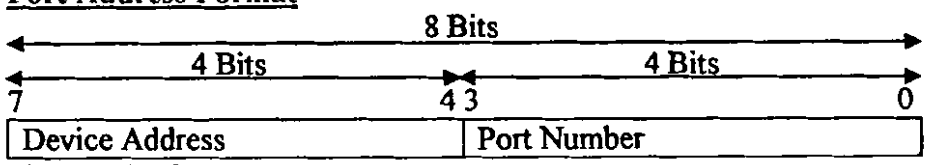


Figure 3.32 Output Port Format

Total Devices: 16
Port per Device: 16

Description:

Outputs the data contained in the Source register R_s to the output port.

3.6.22 OutputI

1	**Instruction type:**	OI-Type
2	**Opcode:**	010101
3	**Immediate-16:**	Output Data
4	**Port Address:**	8 bit port address
5	**Operation:**	OUTPUT[Port Address] ← Imm Data

Figure 3.33 OUTPUTI Instruction

Port Address Format

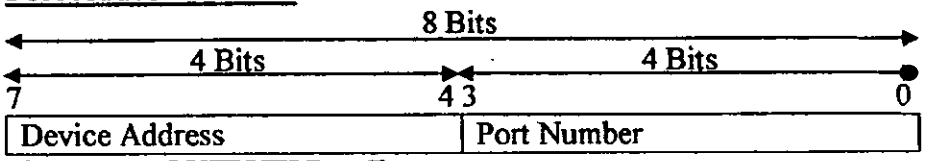


Figure 3.34 OUTPUTI Port Format

Total Devices: 16
Port per Device: 16

Description:

Outputs the Immediate data to the output port.

3.6.23 PUSH

Instruction type:	R-Type
Opcode:	010110
Function Code:	PUSH : 00000 PUSHS : 00001
Source Register 1:	Register to push on top of stack
Operation:	PUSH : StackPointer++ Stack[StackPointer] ← R_s1 PUSHS : SStackPointer++ SStack[SStackPointer] ← R_s1

Figure 3.35 PUSH Instruction

Description:

Pushes the source register on top of the stack or remote stack.

3.6.24 POP

Instruction type:	R-Type
Opcode:	010111
Function Code:	POP : 00000 POPS : 00001
Destination Register:	Destination Register for popped Data
Operation:	POP : R_d ← Stack[StackPointer] StackPointer-- POPS : R_d ← SStack[SStackPointer] SStackPointer--

Figure 3.36 POP Instruction

Description:

Pops data from top of stack or shared stack to the destination register.

3.6.25 CALL

Instruction type:	I-Type
Opcode:	011000
Function Code:	CALL : 00000 CALLS : 00001
Immediate-16:	Function address
Operation:	CALL : StackPointer++ Stack[StackPointer] ← PC PC ← Imm Data CALLS : CallCode ← Imm Data Assert CALLS

Figure 3.37 CALL Instruction

Description:

Calls function on the same processor or the remote processor.

3.6.26 RETURNC

Instruction type:	N-Type
Opcode:	011001
Function Code:	RETURNC : 00000 RETURNCS : 00001
Operation:	RETURNC : PC ← Stack[StackPointer] StackPointer-- RETURNCS : Assert RETURNS

Figure 3.38 RETURNC Instruction

Description:

Returns from the function call on the same or remote processor.

3.6.27 RETURNI

Instruction type:	N-Type
Opcode:	011010
Function Code:	00000
Operation:	RETURNI : PC ← Stack[StackPointer] StackPointer--

Figure 3.39 RETRUNI Instruction

Description:

Returns from the interrupt handler routine.

3.6.28 EDI

Instruction type:	N-Type
Opcode:	011011
Function Code:	EI : 00001 DI : 00001
Operation:	EI : FL-I ← 0 DI : FL-I ← 1

Figure 3.40 EDI Instruction

Description:

Enable or disable interrupts.

3.6.29 TRAP

Instruction type:	N-Type
Opcode:	011100
Function Code:	00000
Immediate-16:	Service Requested
Operation:	CALL : StackPointer++ Stack[StackPointer] ← PC PC ← S1

Figure 3.41 TRAP Instruction

Description:

Traps to the kernel entry point.

3.6.30 WAITI

Instruction type:	I-Type
Opcode:	011101
Function Code:	00000
Immediate-16:	Interrupt Number
Operation:	CALL : Wait until INT = 1 AND INT_CODE = Interrupt Num

Figure 3.42 WATI Instruction

Description:

Waits for specified interrupt.

3.6.31 HALT

1	**Instruction type:**	N-Type
2	**Opcode:**	011111
3	**Function Code:**	00000
4	**Operation:**	Halts the CPU

Figure 3.43 HALT Instruction

Description:

CPU stops executing instructions.

4. System Design

Design of a microprocessor is a complex task. On top of that, in order to test a microprocessor a complete working set of peripherals is required. In this chapter, design of the microprocessor and its associated peripherals is explained.

4.1 MultiCore Processor Architecture

The MC-CPU implements the Instruction Set Architecture defined in the previous chapter. The Master processor implements all the privileged instruction as well as rest of the instruction set. Caches and Shared execution units are not implemented in this version. The Slave processor only implements the non-privileged instructions. Figure 4.1 gives the block diagram of MC-CPU.

4.1.1 MC-CPU Architecture

MC_CPU consists of Master CPU (CPU-1), Slave CPU (CPU-2) and Shared Stack as shown in the figure 4.1. Each of these functional units and their operations are explained individually in the following sections.

4.1.1.1 CPU1

CPU1 is the master CPU. It implements all the privilege instructions. Only the master processor can access I/O devices. Interrupt handling is performed only by the master processor. Shared stack is also controlled by master processor. The master processor can control the behavior of the slave processor by the means of INTS interrupts. Slave processor implements special interrupt handlers for INTS rather than for the normal interrupts.

4.1.1.2 CPU2

CPU2 is the slave CPU. It does not implement the privilege instructions. The slave processor can not physically access I/O devices. System level interrupt handling is not performed by the Slave processor as it does not have an INT line. Shared stack is accessed by the slave processor when ever the master processor grants it access. Slave processor implements special interrupt handlers for INTS rather than for the normal interrupts. When ever the master processor asserts INTS, the slave processor immediately

jumps to the particular interrupt based on INTSCODE. The INTSCODE is an 8-bit value indicating particular interrupt.

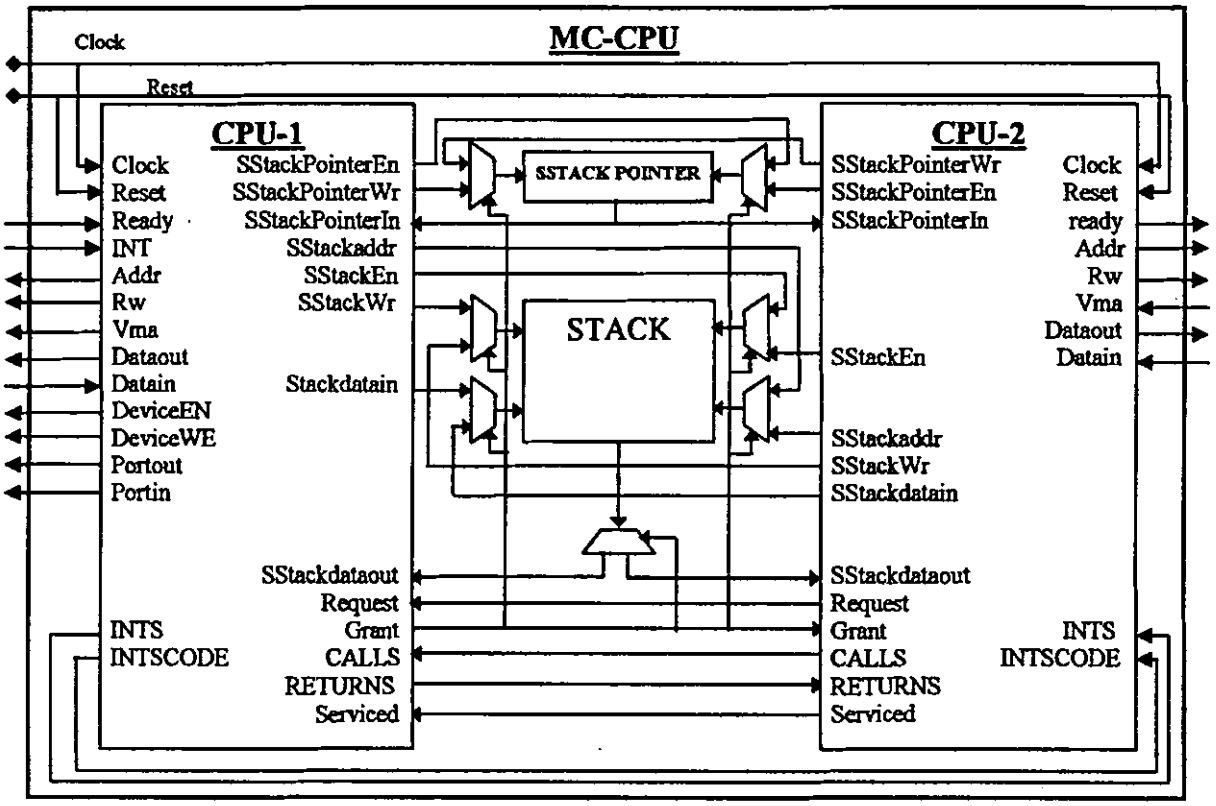


Figure 4.1 MC-CPU

4.1.1.3 Shared Stack

Shared Stack consists of the stack memory and the shared stack pointer. The shared stack pointer is a 32 bit register. Its output value is constantly supplied to both processors. It can only be modified by one processor at any given time. Shared stack memory consists of single port 1024 bit distributed RAM, arranged as 32 X 32 bits. Only one processor can push or pop from the shared stack at any given time.

4.1.1.4 Shared Stack Operation

Shared stack operates in the following manner:

i. Master processor has initial control of the shared stack

ii. Master processor can modify the shared stack pointer any time; only exception is when it has granted control of the shared stack to the slave processor

iii. Master processor can push or pop values from the shared stack any time; except when it has granted control of the shared stack to the slave processor

iv. Slave processor can not directly access the shared stack

v. Slave processor must assert the REQUEST Signal to gain access to the shared stack

vi. Whenever REQUEST is asserted by the Slave processor, the master processor can grant or disallow access to the shared stack

vii. Access is disallowed only when master processor is modifying or accessing the shared stack itself

viii. Slave processor is blocked or in a wait state during this period

ix. When the master processor is not accessing the shared stack and the Slave processor requests for it, request is granted by asserting the GRANT signal

x. When the GRANT signal is asserted Slave processor gets access to the shared stack

xi. Slave processor can now modify both shared stack pointer and shared stack

xii. After the slave processor has modified the stack it deasserts the GRANT signal to indicate that the shared stack is now free

xiii. When the GRANT signal is deasserted the master processor deasserts the GRANT signal and takes the control of shared stack back

4.1.1.5 Remote Call

All the communication between the master and slave processor is based on remote calls. In fact these are not remote calls in the classic sense rather these are traps to the OS kernel running on the master processor. Only the slave processor can trap to the master processor by asserting CALLS signal. The remote calls work in the following manner.

i. When ever the user code running on the Slave processor needs some operating system service it must invoke a remote call

ii. Remote call is invoked by the slave processor by asserting the CALLS signal

iii. Before asserting the CALLS signal slave processor must request access to the shared stack, and at least place the 32 bit service code on top of the shared stack. It can also place any parameters on the stack if there is any

iv. After placing the service code and/or any parameters on the shared stack, the Slave processor asserts the CALLS signal

v. On receiving the CALLS signal Master processor invokes the remote call handler

vi. Remote call handler checks for user access rights and proper parameters and then calls the appropriate OS function. This is a normal function call

vii. Normally no context switch takes place during this whole procedure

viii. After servicing the call and placing return values onto the shared stack the Master processor asserts the RETURNS signal

ix. On receiving the RETURNS signal, Slave processor request for the shared stack, gets the return values, and deasserts the CALLS signal

4.1.1.6 Remote Interrupt

The Master processor controls the slave processor by using Remote Interrupts. Only the Master processor can raise remote interrupts and only the slave processor serves remote interrupts. Interrupt vector table and interrupt service routines for the remote interrupts are placed in the Slave processor's memory space by the Master processor. These interrupts can range from memory management to context switching to process cleanup. Remote interrupts work in the following manner.

i. Operating System running on the Master processor can raise remote interrupts

ii. A remote interrupt is raised by asserting the INTS signal

iii. Interrupt type is indicated by INTSCODE

iv. Upon receiving an INTS the Slave processor immediately jumps to the appropriate handler based on INTSCODE

v. After servicing the INTS the slave processor assert the SERVICED signal

vi. Upon receiving the SERVICED signal the Operating System on the Master processor considers the work done and deasserts the INTS signal

4.1.2 Base CPU Architecture

In this section we describe the base CPU architecture and its implementation. The base CPU consists of the following main units.

4.1.2.1 Internal TriState DataBus

The internal TriState DataBus is the main internal processor bus. All the datapath components are connected to this bus. All datapath components have enable signals. When the enable signal is low the component is disabled and has high impedance. Only one component is enabled and driving the bus. Multiple components can act as destination at any given time.

4.1.2.2 Registers

The CPU has a set of registers for normal operation. Since the design is based on the principles of RISC, a large number of registers are provided. All registers are 32 bit wide. There are 32 general purpose 32 Bit registers Ro to R31m, a 32 bit PC, Address Register, Stack Pointer and Flag Register. Also there are 32 special purpose 32 bit registers S0 to S31.

4.1.2.3 ALU

The CPU has a single 32 bit ALU that performs all arithmetic and logic operations except multiplication and division. The ALU performs the following operation on 32 bit operands:

i. PASS
ii. AND
iii. OR
iv. NOT
v. XOR
vi. ADD
vii. SUBTRACT
viii. INCREMENT
ix. DECREMENT
x. ZERO
xi. ONE

4.1.2.4 Multiplier/Divider

The multiply/divide unit provides the CPU with hardware multiplication capability. Full 32 bit X 32 bit multiplication and division are implemented.

4.1.2.5 Comparator

The comparator unit provides the CPU with comparison operation. Only scalar data comparisons are implemented. Following operations are available.

i. >
ii. >=
iii. <
iv. <=
v. =
vi. !=

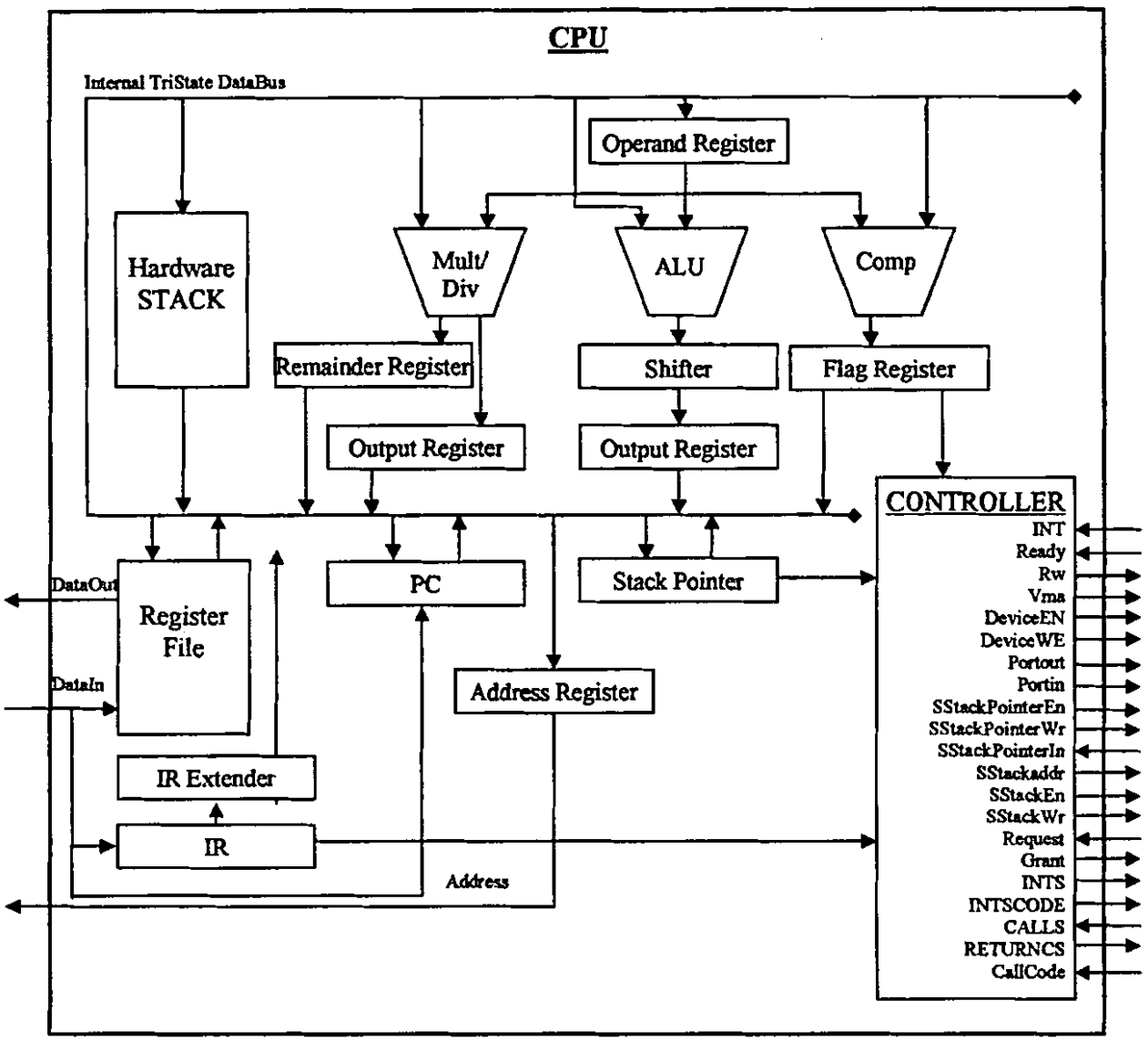


Figure 4.3 CPU

4.1.2.6 Shifter

The shifter unit provides the CPU with shift and rotate operation. Only scalar data shift and rotates are implemented. Full 32 bit shifts and rotates are available. The following operations are available.

i. shl
ii. shr
iii. rotl
iv. rotr

4.1.2.7 IR extender

IR extender is used for the extraction of immediate data and address values from the instructions.

4.1.2.8 Hardware Stack

The hardware stack allows for the storage of procedure return addresses and parameters. It is implemented using single port distributed RAM. It is arranged as 64 X 32 bits. Data can be pushed or popped from the stack.

4.1.2.9 Register File

The register file provides for the main program variable storage. It is a 32 bit dual port register file. Data read from the memory is brought straight to the register file. Data stored to the memory is taken from the register file. Hence the register file is the main component in implementing the load store architecture of the CPU.

4.1.2.10 Controller

Controller is the brain of the CPU. It is implemented as a Finite State machine. All instruction decoding and sequencing takes place inside the controller.

4.2 System Architecture

The complete system is composed of multiple independent units that work together to form a complete working system as shown below in the system level block diagram. The whole system is implemented as a SoC on a single FPGA. The complete system utilizes approximately 95% of a Spartan-IIE (XC2S300pq208-e). The complete system consists of the following units.

i. MC_CPU

MC_CPU architecture and working was explained in detail previously

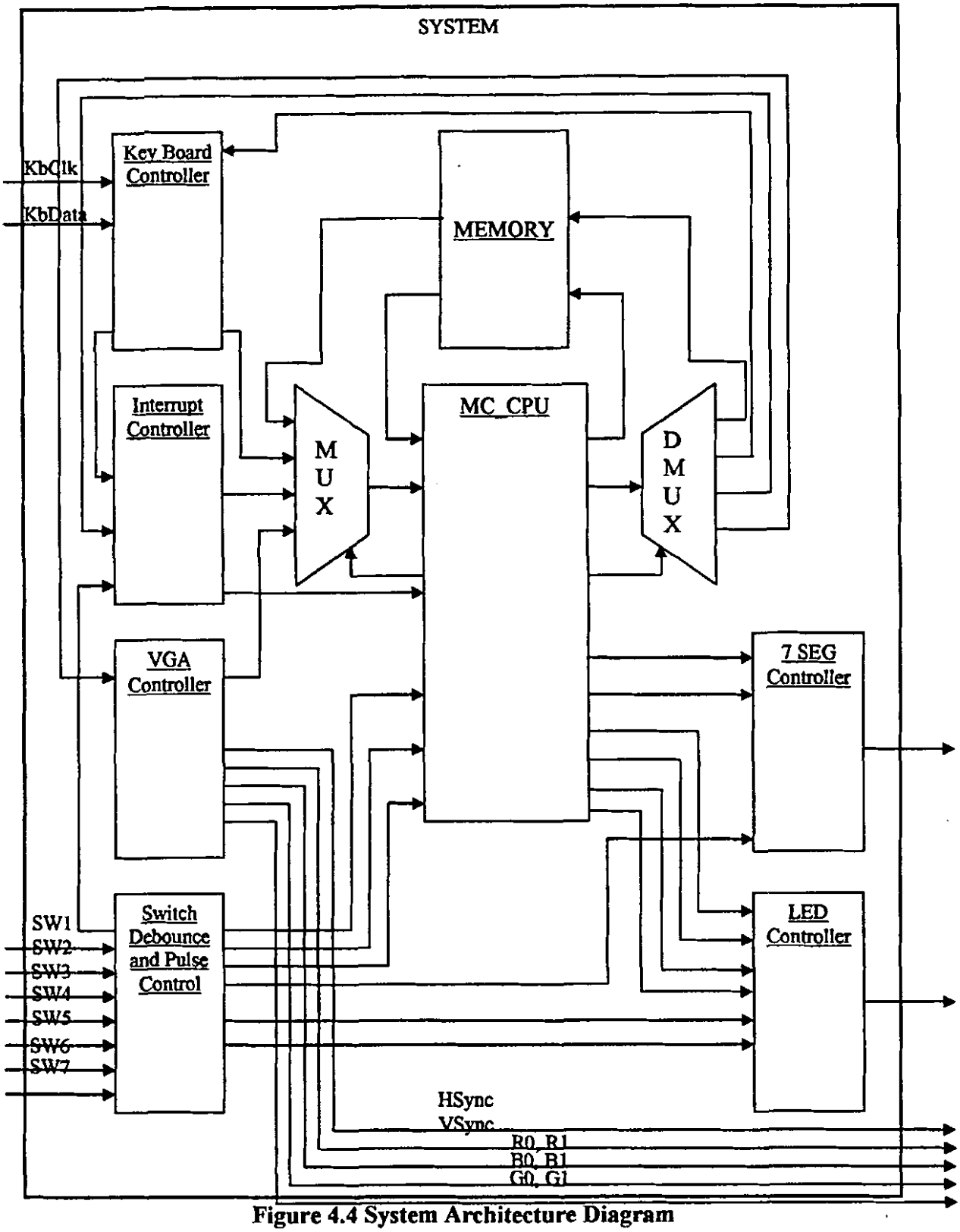


Figure 4.4 System Architecture Diagram

ii. **MC_CPU**

MC_CPU architecture and working was explained in detail previously

iii. **Memory**

The memory is based on dual port RAM and is arranged as 512 X 32

iv. **MC_CPU**

MC_CPU architecture and working was explained in detail previously

v. **Memory**

The memory is based on dual port RAM and is arranged as 512 X 32

vi. **MUX**

MUX is the input multiplexer and is controlled by MC_CPU

vii. **DMUX**

MUX is the output multiplexer and is controlled by MC_CPU.

viii. **Keyboard Controller**

Keyboard controller is used for interfacing to the standard PS2 style keyboard

ix. **VGA Controller**

VGA controller is used to control the standard VGA display at 640 pixels X 480 pixels, 60Hz refresh.

x. **Interrupt Controller**

Provides priority based interrupt handling. 8 interrupts are provided. Each is individually maskable

xi. **Switch Debounce and Pulse Control**

Switch Debounce and pulse control provides debouncing of external switch input. Also it provides one-shot capability

xii. **7-Seg Controller**

7-Seg Controller Controls the two 7-segment displays on the FPGA prototyping kit. Also it switches between PC and Address register

xiii. LED Controller

LED Controller Controls the two LED displays on the FPGA prototyping kit. It switches between INSTRUCTION TYPES or INSTRUCTION REGISTER or ALU OUTPUT or STACK POINTER

4.2.1 Memory

The memory of the system is built onto the FPGA chip. It is arranged as 512 X 32 bits. It is composed of using 4 dual port Xilinx 4096 bit block RAMs. The memory provides two independent read/write ports. One port is used by master CPU while the other is used by the slave CPU.

4.2.2 MUX

MUX is the input multiplexer and is controlled by MC_CPU. It currently provides 4 input ports but can be extended to support 16 input ports. Each port can be upto a maximum of 32 bits wide. The first input port is used by the master CPU memory.

4.2.3 DMUX

DMUX is the output multiplexer and is controlled by MC_CPU. It currently provides 4 output ports but can be extended to support 16 output ports. Each port can be upto a maximum of 32 bits wide. The first output port is used by the master CPU memory.

4.2.4 KEYBOARD Controller

The communication between the keyboard and the controller uses two signals, *KeyboardClock* and *KeyboardData*. When there is no activity, that is, when there is no key press on the keyboard, both *KeyboardClock* and *KeyboardData* are at 1. When a key is pressed (or released), the keyboard sends a unique code for that key to the controller serially over the *KeyboardData* line. The serial data on the *KeyboardData* line is synchronized between the keyboard and the controller by clock pulses that the keyboard sends over the *KeyboardClock* line.

The data for each key that is sent over the *KeyboardData* line consists of eleven bits. These eleven bits are: a 0 for the start bit, 8 data bits for the key code starting with the least significant bit to the most significant bit, an odd parity bit, and lastly a 1 for a

stop bit. Figure 4.6 lists some of the key codes generated by the keyboard when the corresponding key is pressed. When a key is released, a different code is generated. The odd parity bit is set such that the total number of 1 bits in the eight data bits plus the parity bit is an odd number.

Figure 4.5 shows a sample timing diagram for the data transmission of the key code 4E (hex) or 01001110 (binary). Starting from the inactive state where both the *KeyboardData* and *KeyboardClock* lines are at 1, the transmission begins by setting the *KeyboardData* line low for the start bit. The keyboard then sends out the data and parity bit on the *KeyboardData* line at a rate of one bit per clock cycle on the *KeyboardClock* line. The clock pulses on the *KeyboardClock* line are generated by the keyboard. The parity bit for the key code 4E is 1, since the eight data bits consist of an even number of 1 bits, therefore, to make the parity odd, the parity bit must be 1.

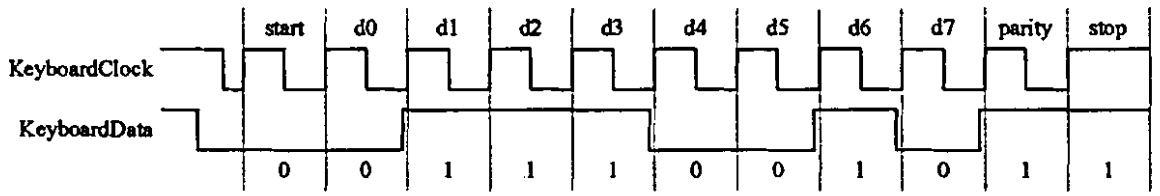


Figure 4.5 Sample timing diagram for the data transmission of the key code 4E

Key	Key Code
1	16
2	1E
3	26
4	25
5	2E
6	36
7	3D
8	3E

Key	Key Code
A	1C
B	32
C	21
D	23
E	24
F	2B
G	34
H	33

Key	Key Code
K	42
L	4B
M	3A
N	31
O	44
P	4D
Q	15
R	2B

Key	Key Code
U	3C
V	2A
W	1D
X	22
Y	35
Z	1A
Esc	76
BS	66

9	46
0	45

I	43
J	3B

S	1B
T	2C

CR	5A
Ctrl	14

Figure 4.6 A partial list of key codes generated by the keyboard

The state diagram for keyboard controller is derived by following the timing diagram shown in fig 4.5. In each of the eight data states, d0, d1, ..., d7, we will get one corresponding data bit from the *KeyboardData* input line. For example, suppose we use an 8-bit register named *keycode* for storing the eight data bits. Then in state d0, we will assign *KeyboardData* to *keycode*(0), in state d1, we will assign *KeyboardData* to *keycode*(1), and so on for all eight data bits. This is possible because the transition of the FSM from one state to another is synchronized by the keyboard clock signal *KeyboardClock*. For simplicity, we will not check for the start bit, parity, and stop bits.

This controller circuit actually does not control the keyboard because it does not generate control signals for the operation of the keyboard. Instead, it receives the serial data signals from the keyboard, and packaged it into data bytes. The output of this controller is simply the data bytes, which represent the key code of the keys being pressed on the keyboard. In state d0, the bit on the *KeyboardData* line is loaded into bit 0 of the *Keycode* register; in state d1, the bit on the *KeyboardData* line is loaded into bit 1 of the *Keycode* register; and so on. Each bit of the *Keycode* register must, therefore, be able to load in the *KeyboardData* independently, and each load enable line is asserted by the corresponding state encoding

4.2.5 VGA Controller

The monitor screen for a standard VGA format contains 640 columns by 480 rows of picture elements called pixels. An image is displayed on the screen by turning on or off individual pixels. The monitor continuously scans through the entire screen turning on or off one pixel at a time at a very fast speed. The scan starts from row 0, column 0 at the top left corner, and moves to the right until it reaches the last column in the row. When the scan reaches the end of a row, it continues at the beginning of the next row. When the scan reaches the last pixel at the bottom right corner of the screen, it goes back to the top left corner of the screen, and repeats the scanning process again. In order to reduce flicker on the screen, the entire screen must be scanned 60 times per second or higher. During the horizontal and the vertical retraces, all the pixels are turned off.

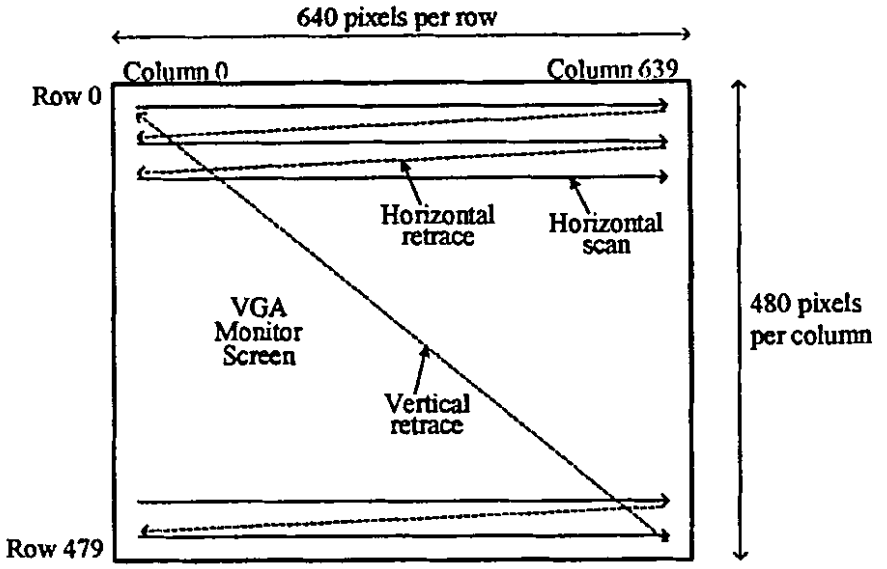


Figure 4.7 VGA monitor with 640 columns × 480 rows. Scan starts from row 0, column 0, and moves to the right and down until row 479, column 639.

The VGA monitor is controlled by five signals: red, green, blue, horizontal synchronization, and vertical synchronization. The three color signals, referred to collectively as the RGB signal, are used to control the color of a pixel at a location on the screen. These three color signals on the FPGA board are connected such that they can individually be either turned on or off, hence each pixel can display only one of eight colors. In order to produce more colors, each analog color signal must be supplied with a voltage between 0.7 to 1.0 volts for varying the intensities of the colors. The horizontal and vertical synchronization signals are used to control the timing of the scan rate. The horizontal synchronization signal determines the time to scan a row, while the vertical synchronization signal determines the time to scan the entire screen. By manipulating these five signals, images are formed on the monitor screen. Each analog color input can be set to one of four levels by two digital outputs using a simple two-bit digital-to analog converter. The four possible levels on each analog input are combined by the monitor to create a pixel with one of 4 X 4 X 4 = 64 different colors. The six digital control lines let us select from a palette of 64 colors.

The horizontal and vertical synchronization signals timing diagram is shown in Figure 4.7. When inactive, both synchronization signals are at a 1. The start of a row scan begins with the horizontal sync signal going low for 3.77 μsec as shown by region B in Figure 5.10. This is followed by a 1.79 μsec high on the signal as shown by region C. Next, the data for the three color signals are sent, one pixel at a time, for the 640 columns

as shown in region D for 25.42 μsec. Finally, after the last column pixel, there is another 0.79 μsec of inactivity on the RGB signal lines as shown in region E before the horizontal sync signal goes low again for the next row scan. The total time to complete one row scan is 31.77μsec.

The timing for the vertical synchronization signal is analogous to the horizontal sync signal. The 64μsec active low vertical sync signal resets the scan to the top left corner of the screen as shown in region P, followed by a 1020 μsec high on the signal as shown by region Q. Next, there are 480 row scans of 31.77μsec each, giving a total of 15250μsec as shown in region R. Finally, after the last row scan, there is another 450μsec as shown in region S before the vertical sync signal goes low again to start another complete screen scan starting at the top left corner. The total time to complete one complete scan of the screen is 16784μsec.

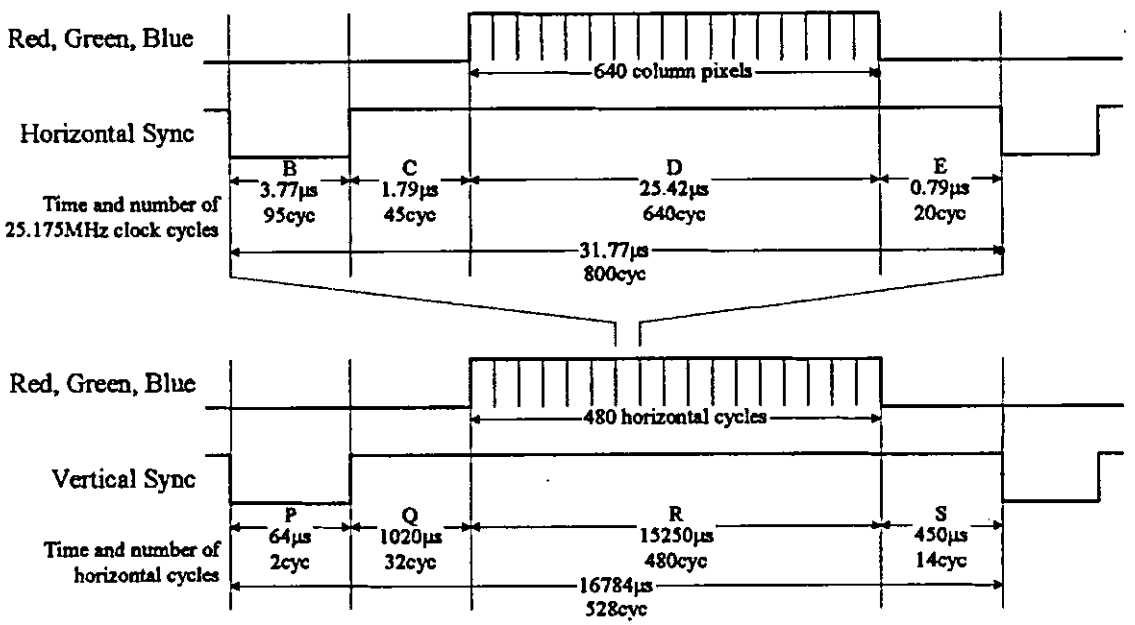


Figure 4.9 Horizontal and vertical synchronization signals timing diagram.

In order to get the monitor to operate properly, we simply have to get the horizontal and vertical synchronization signals timing correct, and then send out the RGB data for each pixel at the right column and row position. It turns out that it is fairly simple to get the correct timing for the two synchronization signals. The built-in clock crystal on the UP2 board runs at a speed of 25.175MHz, which gives a period of $1 / 25.175 \times 10^6$ which is about 0.0397μsec per clock cycle. For region B in the horizontal synchronization

signal, we need 3.77μsec, which is approximately 3.77 / 0.0397 = 95 clock cycles. For region C, we need 1.79μsec, which is approximately 45 clock cycles. Similarly, we need 640 clock cycles for region D for the 640 columns of pixels, and 20 clock cycles for region E. The total number of clock cycles needed for each row scan is, therefore, 800 clock cycles.

To get this timing correct, we can design FSM with 800 states running at a clock speed of 25.175MHz. For the first 95 states, we will output a 0 for the horizontal sync signal *H_Sync*. For the next 45+640+20=705 states, we will output a 1 for *H_Sync*. The problem with this, however, is that it is difficult to manually derive the circuit for an 800 state FSM. A simple solution around this difficulty is to use just two states; one for when *H_Sync* is 0 in region B, and one for when it is 1 in regions C, D, and E. We will then use a counter that runs at the same clock speed as the FSM to keep count of how many times we have been in a state. For the first state, we will stay there for 95 counts before going to the next state, and for the second state, we will stay there for 705 counts before going back to the first state. In the first state, we will output a 0 for *H_Sync*, and in the second state, we will output 1 for *H_Sync*.

The vertical sync timing is analogous to the horizontal sync timing, so we can do the same thing using a second counter, and a second FSM. This second vertical FSM is identical to the horizontal FSM. The only difference is in the timing. Looking at the times for each region in the vertical synchronization signal in Figure 6.10, we see that 64 μsec for region P is approximately 2 times the total horizontal scan time of 31.77 μsec each. 1020 μsec for region Q is approximately 32 horizontal scan time (1020 / 31.77 ≈ 32). For region R, it is 480 horizontal cycles, and for region S, it is approximately 14 horizontal cycles. Hence, the clock for both the vertical counter and the vertical FSM can be derived from the horizontal counter. The vertical clock ticks once for every 800 counts of the horizontal clock.

We will need to use two instances of this FSM circuit; one for the horizontal FSM, and one for the vertical FSM. The clock for the horizontal FSM is the 25MHz clock, while the clock for the vertical FSM is derived from the roll over signal from the horizontal counter. The four status signals for the four counter conditions are generated from two counters: a horizontal counter, and a vertical counter.

To display something on the screen, we simply have to check the current column and row that the scan is at, and then assert the RGB signal if we want the pixel at that location to be turned on. For example, if we simply assert the red signal continuously, all the pixels will be red, and we will see the entire screen being red. On the other hand, if we just want the first row of pixels to be red, then we need to assert the red signal only when the counter *Row* = 0. To get a red border around the screen, we would assert the red signal when *Row* = 0, or *Row* = 639, or *Column* = 0, or *Column* = 479. Figure 4.10 shows the circuit to draw a red border around the entire screen using the VGA controller circuit from.

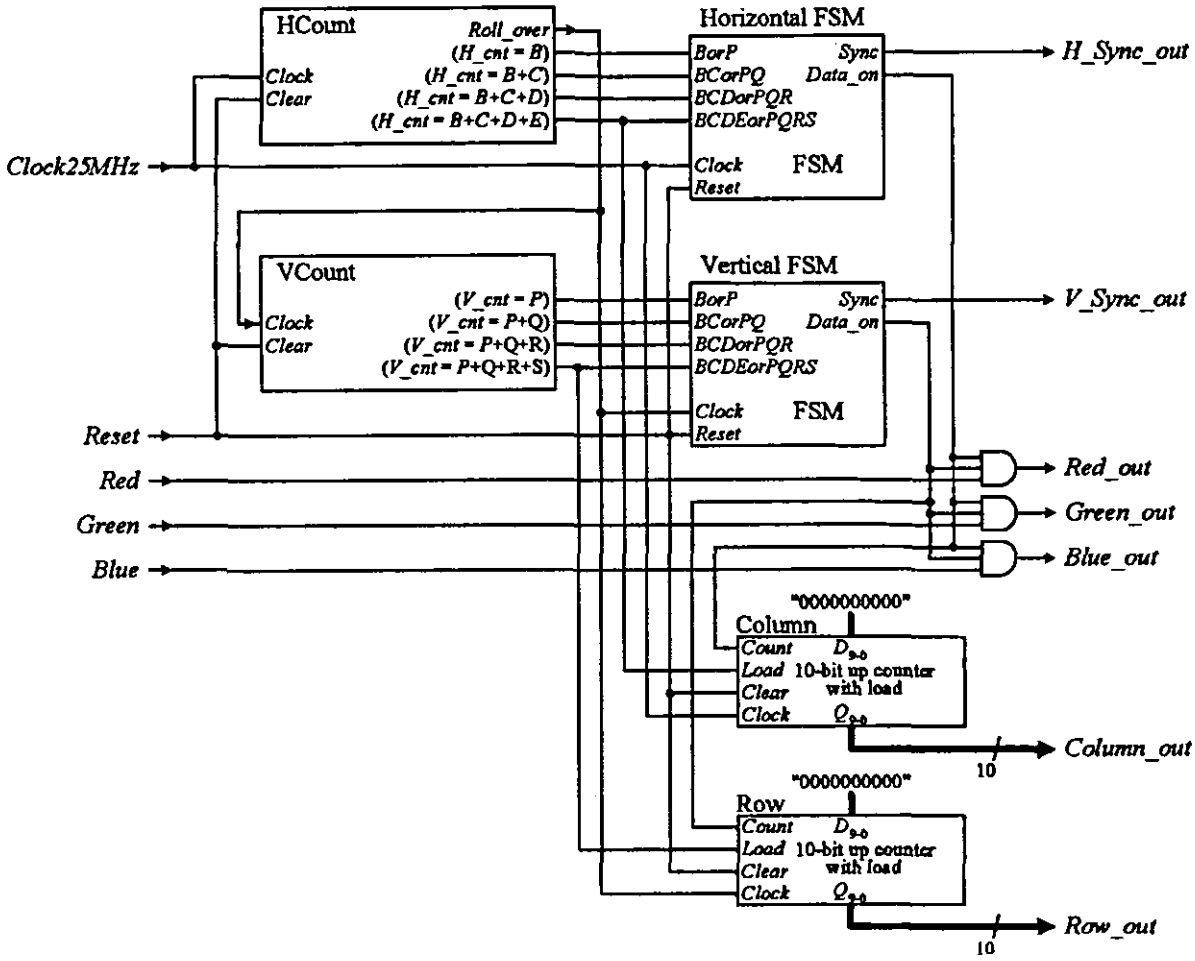


Figure 4.10 Complete circuit for the VGA controller.

4.2.6 Interrupt Controller

The interrupt controller provides priority based interrupt handling. 8 interrupts are provided. Each is individually maskable. The interrupt controller is programmable and can be read or written using the input and output multiplexers. Each interrupt can be masked by setting the corresponding bit in the mask register. When ever there is an interrupt the interrupt controller asserts the CPU INT pin. The CPU reads the interrupt register to know which device has generated the interrupt.

4.2.7 Switch Debounce and Pulse Control

Switch Debounce and pulse control provides debouncing of external switch inputs. It also provides one-shot capability for single stepping the MC_CPU. The basic component in this controller is the 25MHz to 100KHz clock divider. Every input to the device is debounced first. Two of the outputs are also one pulsed. In the case of one pulse the system clock of 25MHz is used. No matter how long the button is pressed the one pulse will only generate one pulse of the same time period as of the system clock.

4.2.8 7-Seg Controller

7-Seg Controller Controls the two 7-segment displays on the FPGA prototyping kit. Also it switches between PC and Address register. Each 7-segment displays in turn have two displays, giving a total of four displays. The basic building blocks of the controller are the Hex to 7-segment decoders. Four of which are used. The implementation schematic of interrupt controller is shown in figure 5.14.

4.2.9 LED Controller

LED Controller Controls the two LED displays on the FPGA prototyping kit. Also it switches between INSTRUCTION TYPES or NSTRUCTION REGISTER or ALU OUTPUT or STACK POINTER. The LED controller is basically a big multiplexer that multiplexes between its four inputs based on two switch inputs.

5. Conclusion

The whole system is implemented as a SoC on a single FPGA. The complete system utilizes approximately 95% of a Spartan-IIE (XC2S300pq208-e).

5.1 Hardware Arrangement

The complete system was mounted on a chipboard base in the following arrangement. It was interfaced with the computer using the parallel port. A test program, given in Appendix A, was used to test proper operation of the system.

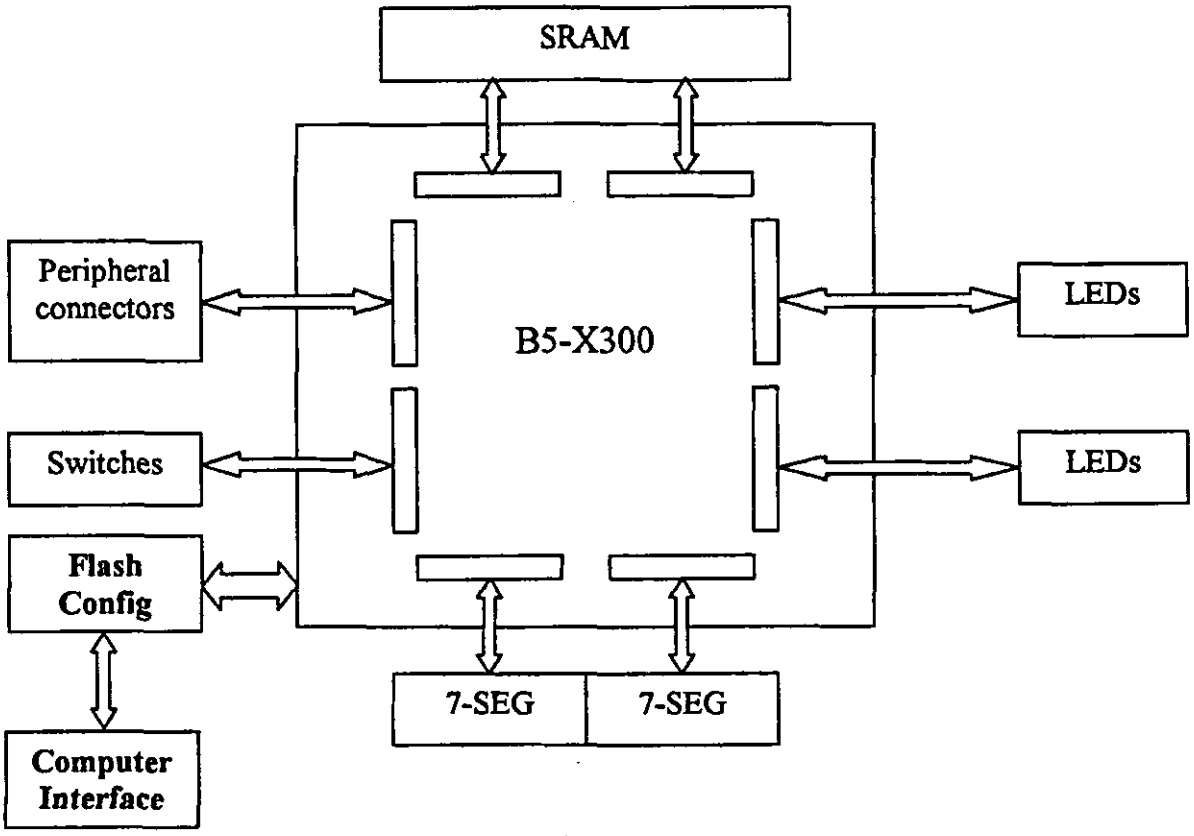


Figure 5.1 Hardware Arrangement

5.2 Result

As a result of this research project we have been able to verify the benefits of the MultiCore design. Specifically, a marked reduction in the context switch penalty. Since, the code running on the master processor is never preempted, it is able to service user requests more efficiently.

This design has allowed us to compare it against the current SMP and CMP systems. When compared to the SMP systems the outcome is very clear that the main bottle neck is the Interprocessor communication buss. In case Of MC-CPU there is no such external Interprocessor communication buss and hence such latencies are avoided altogether.

When compared to the other CMP processors the MC-CPU architecture does not employ any packet protocol for communication among the two processors. This improves the inter processor communication capability significantly. The downside is that it require extensive hardware support.

The direct measure of MC-CPU performance comes from comparing a piece of code that calls OS services, first on the Master CPU and then on the slave CPU. When the code is run on the master processor, the system behaves just like a normal single processor system. At every system call performed by the user routine there is a context switch and the OS is switch back. The OS performance the necessary operation and then preempts itself while making then user program active.

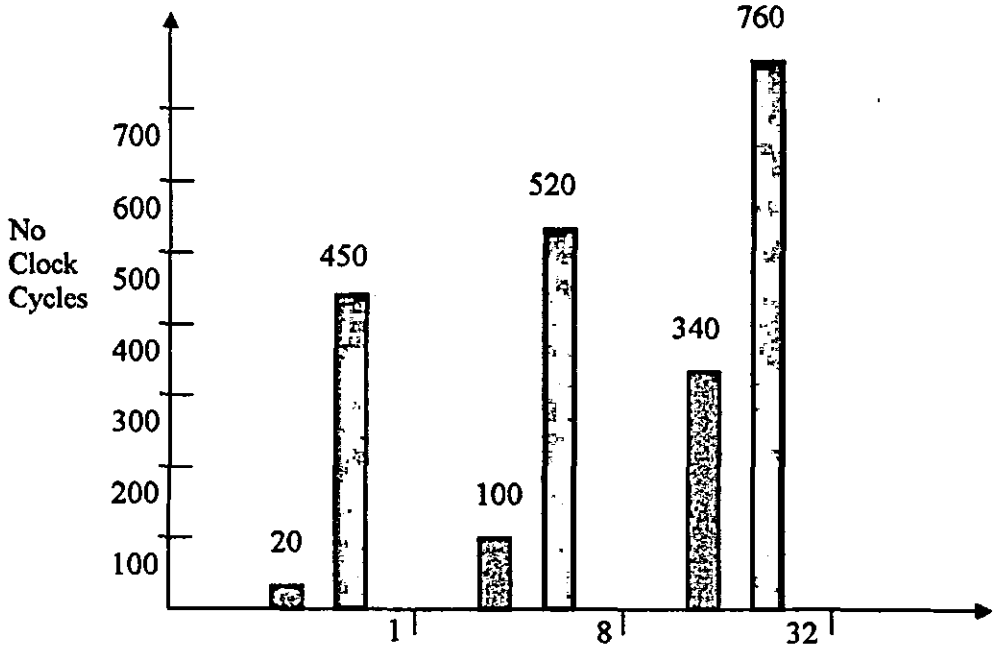


Fig 5.2 No of parameters

In a context switch 43 registers are saved and then 43 registers are restored. Saving a single register to memory takes 10 clock cycles. Saving 43 registers requires 430 clocks. In all a single context switch takes about 450 clock cycles on the master processor. This is for an OS service that only requires the service code and no parameters.

On the MC-CPU, when the user code is run on the slave processor and it performs a system call then there is no need for a context switch to occur since the OS is running on separate processor. A remote trap that only passes the service code to the master processor requires only 20 cycles

It can be easily seen from figure 5.2 that a single context switch requires at least 500 clock cycles whereas a remote trap only requires 10 clock cycles. Thus it has been shown that incorporating features at the microarchitecture level can improve IPC performances significantly. Improvements in IPC performance improve OS performance significantly.

References and Bibliography

1. The Designer's Guide to VHDL, 2nd Edition 2002, Peter J. Ashenden
2. VHDL Programming by Example, Fourth Edition 2002, Douglas L. Perry.
3. Microprocessor Design Principles and Practices With VHDL, 2004, Enoch O. Hwang
4. Computer Architecture A Quantitative Approach, Third Edition 2003, John L. Hennessy & David A. Patterson.
5. Computer Organization and Architecture, 6th Edition 2003, William Stallings.
6. DEDICATED DIGITAL PROCESSORS Methods in Hardware/Software System Design, 2004, F. Mayer-Lindenberg
7. Operating Systems Concepts, Sixth Edition2003, Silberschatz Galvin Gagne
8. Operating Systems, Forth Edition2001, William Stallings
9. [Aaron B. Brown 1997] A Decompositional Approach to Computer System Performance Evaluation, Center for Research in Computing Technology Harvard University Cambridge, Massachusetts.
10. [Benjamin Gamsa, Orran Krieger, Eric W. Parsons, Michael Stumm 1995] Performance Issues of Multiprocessor Operating Systems.
11. [Chu Xia & Josep Torrellas 1999] Comprehensive Hardware and Software Support for Operating Systems to Exploit MP Memory Hierarchies.
12. [Dawson R. Engler 1998] The ExoKernel operating systems architecture.
13. [Hsiao-Ping Juan, Nancy D. Holmes Smita Bakshi, Daniel D. Gajski 1992] Top Down Modeling of RISC Processors in VHDL.
14. [Jan Gray 2001] Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip.
15. [Jan Gray 2000] Hands-on Computer Architecture - Teaching Processor and Integrated Systems Design with FPGAs.
16. [Jochen Liedtke 1993] Improving IPC by Kernel Design
17. [Jochen Liedtke 1995] On u-Kernel Construction

18. [Jochen Liedtke , Hermann H¨artig, Michael Hohmuth, Sebastian Sch¨onberg, Jean Wolter 1997] The Performance of μ-Kernel-Based Systems.

19. [Jochen Liedtke, Andreas Haeberlen, Yoonho Park, Lars Reuther, Volkmar Uhlig 2000] Stub-Code Performance is Becoming Important

20. [Jochen Liedtke 2001] J. Liedtke, U. Dannowski, K. Elphinstone, G. Liefländer, E. Skoglund, V. Uhlig, C. Ceelen, A. Haeberlen, and M. Völp. **The L4Ka Vision.**

21. [Joshua A. Redstone, Susan J. Eggers & Henry M. Levy 2000] Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture.

22. [Kunkel & Smith 1986] Optimal pipelining in supercomputers.

23. [Lance Hammond, Basem A. Nayfeh, Kunle Olukotun 1997] A Single-Chip Multiprocessor.

24. [Makiko ITOH 2000] Synthesizable HDL Generation for Pipelined Processors from A Micro-Operation Description.

25. [Paul Kohout 2002] Hardware Support for Real-Time Operating Systems

26. [Prof. Lizy Kurian John 2002] Performance Evaluation: Techniques, Tools and Benchmarks.

27. [Takayuki Morimoto, Kazushi Saito, Hiroshi Nakamura, Taisuke Boku, Kisaburo Nakazawa] Advance Processor Design Using Hardware Description Language AIDL.

28. [Tao Li, Lizy Kurian John, Anand Sivasubramaniam, N. Vijaykrishnan, Juan Rubio 2002] Understanding and Improving Operating System Effects in Control Flow Prediction.

29. [Todd Austin, Eric Larson, Dan Ernst 2002] SimpleScalar: An Infrastructure for Computer System Modeling.

30. [Bloch 1959], [Bucholtz 1962], [Kunkel & Smith 1986], [Smith & Pleszkun 1988], [Weiss & smith 1984], [Killian 1991], [Heinrich 1993], [Smith 1981], [Yeh & Patt 1992, 1993], [Kaeli & Emma 1991], [Sohi 1990], [Smith, Johnson & Horowitz 1989], [Agarwal 1993], [Gupta & Horrowitz 1994], [Yamamoto 1994], [Tullsen 1996], [Lo 1997], [Lo 1998]. **References From Computer Architecture A Quantitative Approach, Third Edition 2003, John L. Hennessy & David A. Patterson.**

Appendix-A Test Code

-----JUMP TO PROGRAM CODE

B"010001_00000_0_00000000000000000111",

-----INTERRUPT HANDLER CALLS

B"011000_00000_0_00000000000011100101",

B"011010_00000_0_00000000000000000000",

X"00000000",--03 RESERVED

X"00000000",--04 RESERVED

X"00000000",--05 RESERVED

X"00000000",--06 RESERVED

-----LOAD ALL DATA

B"000111_00000_00000_10111100_01001000",--07 MOVEI TO R0
B"000111_00000_00001_10111100_01000001",--08 MOVEI TO R1
B"000111_00000_00010_10111100_01010010",--09 MOVEI TO R2
B"000111_00000_00011_10111100_01001111",--0A MOVEI TO R3
B"000111_00000_00100_10111100_01001111",--0B MOVEI TO R4
B"000111_00000_00101_10111100_01001110",--0C MOVEI TO R5
B"000111_00000_00110_10111100_00000000",--0D MOVEI TO R6
B"000111_00000_00111_10111100_01001101",--0E MOVEI TO R7

B"000111_00000_01000_10111100_01010111",--0F MOVEI TO R8
B"000111_00000_01001_10111100_01001111",--10 MOVEI TO R9
B"000111_00000_01010_10111100_01010010",--11 MOVEI TO R10
B"000111_00000_01011_10111100_01001011",--12 MOVEI TO R11
B"000111_00000_01100_10111100_01001001",--13 MOVEI TO R12
B"000111_00000_01101_10111100_01001110",--14 MOVEI TO R13
B"000111_00000_01110_10111100_01000111",--15 MOVEI TO R14
B"000111_00000_01111_10111100_01011000",--16 MOVEI TO R15

---ROUTINE 1

B"011101_00000000000000000000000000",

--DATA BUS

B"010101_00_000_01011_00_100000_0001_0001",

B"010101_00_10111100_00110001_0001_0000",

B"010101_00_000_01011_00_100001_0001_0001",

B"010101_00_10111100_01000011_0001_0000",

B"010101_00_000_01011_00_100010_0001_0001",

B"010101_00_10111100_00110000_0001_0000",

B"010101_00_000_01011_00_100011_0001_0001",

B"010101_00_10111100_01000001_0001_0000",

B"010101_00_000_01011_00_100100_0001_0001",

B"010101_00_10111100_00110101_0001_0000",

B"010101_00_000_01011_00_100101_0001_0001",

B"010101_00_10111100_00110111_0001_0000",

B"010101_00_000_01011_00_100110_0001_0001",

B"010101_00_10111100_00111001_0001_0000",

B"010101_00_000_01011_00_100111_0001_0001",

B"010101_00_10111100_00110000_0001_0000",

--ADDRESS BUS

B"010101_00_000_01101_00_100100_0001_0001",

B"010101_00_10111100_00110000_0001_0000",

B"010101_00_000_01101_00_100101_0001_0001",

B"010101_00_10111100_00110001_0001_0000",

B"010101_00_000_01101_00_100110_0001_0001",

```
        B"010101_00_10111100_00110000_0001_0000",
        B"010101_00_000_01101_00_100111_0001_0001",
        B"010101_00_10111100_01000110_0001_0000",
--CALL
-----------------------------------------------
        B"010101_00_000_10001_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",
--RETURN
-----------------------------------------------
        B"010101_00_000_10010_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",
--REQUEST
-----------------------------------------------
        B"010101_00_000_10011_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",
--GRANT
-----------------------------------------------
        B"010101_00_000_10100_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",
--CALLR
-----------------------------------------------
        B"010101_00_000_10101_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",
--RETURNR
-----------------------------------------------
        B"010101_00_000_10110_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",

--INT
```

```
-------------------------------------------------

        B"010101_00_000_10111_00_100000_0001_0001",

        B"010101_00_10111100_01011110_0001_0000",

    --RETURNI
    -------------------------------------------------

        B"010101_00_000_11000_00_100000_0001_0001",

        B"010101_00_10111100_01011110_0001_0000",

        B"011101_00000000000000000000000000",


-------------------------------------------------
---ROUTINE 2
-------------------------------------------------

B"011101_00000000000000000000000000",

--DATA BUS
-------------------------------------------

    B"010101_00_000_01011_00_100000_0001_0001",

    B"010101_00_10111100_00110001_0001_0000",

    B"010101_00_000_01011_00_100001_0001_0001",

    B"010101_00_10111100_01000011_0001_0000",

    B"010101_00_000_01011_00_100010_0001_0001",

    B"010101_00_10111100_00110000_0001_0000",

    B"010101_00_000_01011_00_100011_0001_0001",

    B"010101_00_10111100_01000001_0001_0000",

    B"010101_00_000_01011_00_100100_0001_0001",

    B"010101_00_10111100_00110101_0001_0000",

    B"010101_00_000_01011_00_100101_0001_0001",

    B"010101_00_10111100_00110111_0001_0000",

    B"010101_00_000_01011_00_100110_0001_0001",
```

```
B"010101_00_10111100_00111001_0001_0000",
B"010101_00_000_01011_00_100111_0001_0001",
B"010101_00_10111100_00110000_0001_0000",
--ADDRESS BUS
-----------------------------------------------
        B"010101_00_000_01101_00_100100_0001_0001",
        B"010101_00_10111100_00110000_0001_0000",
        B"010101_00_000_01101_00_100101_0001_0001",
        B"010101_00_10111100_00110001_0001_0000",
        B"010101_00_000_01101_00_100110_0001_0001",
        B"010101_00_10111100_00110000_0001_0000",
        B"010101_00_000_01101_00_100111_0001_0001",
        B"010101_00_10111100_01000110_0001_0000",
--CALL
-----------------------------------------------
        B"010101_00_000_10001_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",
--RETURN
-----------------------------------------------
        B"010101_00_000_10010_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",
--REQUEST
-----------------------------------------------
        B"010101_00_000_10011_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",
--GRANT
-----------------------------------------------
```

```
        B"010101_00_000_10100_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",
--CALLR
-----------------------------------------
        B"010101_00_000_10101_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",
--RETURNR
-----------------------------------------
        B"010101_00_000_10110_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",
--INT
-----------------------------------------
        B"010101_00_000_10111_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",
--RETURNI
-----------------------------------------
        B"010101_00_000_11000_00_100000_0001_0001",
        B"010101_00_10111100_01011110_0001_0000",
        B"011101_00000000000000000000000000",
```

Appendix-B Glossary of Terms

Address bus—A unidirectional set of signals used by a processor (or similar device) to point to memory locations in which it is interested.

Analog—A continuous value that most closely resembles the real world and can be as precise as the measuring technique allows.

Asynchronous—A signal whose data is acknowledged or acted upon immediately and does not depend on a clock signal.

Binary digit—A numeral in the binary scale of notation. A binary digit (typically abbreviated to "bit") can adopt one of two values: 0 or 1.

Binary encoding—A form of state assignment for state machines that requires the minimum number of state variables.

Binary logic—Digital logic gates based on two distinct voltage levels. The two voltages are used to represent the binary values 0 and 1 along with their logical equivalents False and True.

Bit—Abbreviation *of binary digit*. A binary digit can adopt one of two values: 0 or 1.

Boolean algebra—A mathematical way of representing logical expressions.

Bus—A set of signals performing a common function and carrying similar data. Typically represented using vector notation: for example, an 8-bit database might be named data[7:0].

Byte—A group of eight binary digits, or bits.

Cache memory—A small, high-speed memory (usually implemented in SRAM) used to buffer the central processing unit from any slower, lower-cost memory devices such as DRAM. The high-speed cache memory is used to store the active instructions and datal associated with a program, while the bulk of the instructions and data resides in the slower memory.

Chip—Popular name for an *integrated circuit (IC)*.

Circuit board—The generic name for a wide variety of interconnection techniques, which include rigid, flexible, and rigid-flex boards in single-sided, double-sided, multilayer, and discrete wired configurations.

CLB (configurable logic block)—The Xilinx term for the next logical partition/entity above a slice. Some Xilinx FPGAs have two slices in each CLB, while others have four.

CMOS (complementary metal oxide semiconductor)—Logic gates constructed using a mixture of NMOS and PMOS transistors connected together in a complementary manner.

Combinational logic—A digital logic function formed from a collection of primitive logic gates (AND, OR, NAND, NOR, etc.), where any output values from the function are directly related to the current combination of values on its inputs. That is, any changes to the signals being applied to the inputs to the function will immediately start to propagate (ripple) through the gates forming the function until their effects appear at the outputs from the function. Some folks prefer to say "combinatorial logic."

CPLD (complex PLD)—A device that contains a number of SPLD (typically PAL) functions sharing a common programmable interconnection matrix.

CPU (central processing unit)—The brain of a computer where all of the decision making and number crunching are performed.

CRC (cyclic redundancy check)—A calculation used to detect errors in data communications, typically performed using a *linear feedback shift register (LFSR)*. Similar calculations may be used for a variety of other purposes such as data compression.

Data bus—A bidirectional set of signals used by a computer to convey information from a memory location to the central processing unit and vice versa. More generally, a set of signals used to convey data between digital functions.
Data-path function—A well-defined function such as an adder, counter, or multiplier used to process digital data.

Digital—A value represented as being in one of a finite number of discrete states called *quanta*. The accuracy of a digital value is dependent on the number of quanta used to represent it.

Digital circuit—A collection of logic gates used to process or generate digital signals.

Diode—A two-terminal device that conducts electricity in only one direction; in the other direction it behaves like an open switch. These days the term *diode* is almost invariably taken to refer to a semiconductor device, although alternative implementations such as vacuum tubes are available.

DSP (digital signal processing)—The branch of electronics concerned with the representation and manipulation of signals in digital form. This form of processing includes compression, decompression, modulation, error correction, filtering, and otherwise manipulating audio (voice, music, etc.), video, image, and other such data for such applications like telecommunications, radar, and image processing (including medical imaging).

Edge sensitive—An input to a logic function that only affects the function when it transitions from one logic value to another.

EEPROM or E2PROM (electrically erasable programmable read-only memory)—A memory *integrated circuit (IC)* whose contents can be electrically programmed by the

designer. Additionally, the contents can be electrically erased, allowing the device to be reprogrammed.

EPROM (erasable programmable read-only memory)—A memory *integrated circuit (IC)* whose contents can be electrically programmed by the designer. Additionally, the contents can be erased by exposing the die to *ultraviolet (UV)* light through a quartz window mounted in the top of the component's package.

FIFO (first in first out)—A special memory device or function in which data is read out in the same order that it was written in.

Firmware—Refers to programs or sequences of instructions that are loaded into nonvolatile memory devices.

FLASH memory—An evolutionary technology that combines the best features of the EPROM and E2PROM technologies. The name FLASH is derived from the technology's fast reprogramming time compared to EPROM.

FPGA (field-programmable gate array)—A type of digital *integrated circuit (IC)* that contains configurable (programmable) blocks of logic along with configurable interconnect between these blocks. Such a device can be configured (programmed) by design engineers to perform a tremendous variety of different tasks.

FSM (finite state machine)—The actual implementation (in hardware or software) of a function that can be considered to consist of a finite set of states through which it sequences.

Giga—Unit qualifier (symbol = G) representing one thousand million, or 109. For example, 3 GHz stands for 3 × 109 hertz.

Glue logic—The relatively small amounts of simple logic that are used to connect ("glue") together—and interface between—larger logical blocks, functions, or devices.

Hardware—Generally understood to refer to any of the physical portions constituting an electronic system, including components, circuit boards, power supplies, cabinets, and monitors.

HDL (hardware description language)—Today's digital *integrated circuits (ICs)* can end up containing hundreds of millions of logic gates, and it simply isn't possible to capture and manage designs of this complexity at the schematic (circuit-diagram) level. Thus, as opposed to using schematics, the functionality of a high-end IC is now captured in textual form using an HDL. Popular HDLs are Verilog, SystemVerilog, VHDL, and SystemC.

High-impedance state—The state associated with a signal that is not currently being driven by anything. A highimpedance state is typically indicated by means of the "Z" character.

Hz (hertz)—Unit of frequency. One hertz equals one cycle, or one oscillation, per second.

IC (integrated circuit)—A device in which components such as resistors, diodes, and transistors are formed on the surface of a single piece of semiconducting material.

ICR (in-circuit reconfigurable)—An SRAM-based or similar component that can be dynamically reprogrammed on the fly while remaining resident in the system.

IP (intellectual property)—When a team of electronics engineers is tasked with designing a complex *integrated circuit (IC)*, rather than reinvent the wheel, they may decide to purchase the plans for one or more functional blocks that have already been created by someone else. The plans for these functional blocks are known as intellectual property, or IP. IP blocks can range all the way up to sophisticated communications functions and microprocessors. The more complex functions, like microprocessors, may be referred to as "cores."

ISP (in-system programmable)—An E2-based, FLASH-based, SRAM-based, or similar *integrated circuit (IC)* that can be reprogrammed while remaining resident on the circuit board.

Kilo—Unit qualifier (symbol = K) representing one thousand, or 103. For example, 3 KHz stands for 3 × 103 hertz.

LC (logic cell)—The core building block in a modern FPGA from Xilinx is called a *logic cell (LC)*. Among other things, an LC comprises a 4-input LUT, a multiplexer, and a register.

LE (logic element)—The core building block in a modern FPGA from Altera is called a *logic element (LE)*. Among other things, an LE comprises a 4-input LUT, a multiplexer and a register.

Logic function—A mathematical function that performs a digital operation on digital data and returns a digital value.

Logic gate—The physical implementation of a simple or primitive logic function.

Logic synthesis—A process in which a program is used to automatically convert a high-level textual representation of a design (specified using a *hardware description language (HDL)* at the *register transfer level (RTL)* of abstraction) into equivalent registers and Boolean equations. The synthesis tool automatically performs simplifications and minimizations and eventually outputs a gate-level netlist.

LSB—(1) (least-significant bit) The binary digit, or bit, in a binary number that represents the least-significant value (typically the right-hand bit). **(2) (least-significant byte)**—The byte in a multibyte word that represents the least-significant values (typically the right-hand byte).

LUT (lookup table)—There are two fundamental incarnations of the programmable logic blocks used to form the medium-grained architectures featured in FPGAs: MUX (multiplexer) based and LUT (lookup table) based. In the case of a LUT, a group of input signals is used as an index (pointer) into a lookup table.

Macroarchitecture definition—A design commences with an original concept, whose high-level definition is determined by system architects and system designers. It is at this stage that *macroarchitecture* decisions are made, such as partitioning the design into hardware and software components, selecting a particular microprocessor core and bus structure, and so forth. The resulting specification is then handed over to the hardware design engineers, who commence their portion of the development process by performing *microarchitecture definition* tasks.

Mega—Unit qualifier (symbol = M) representing one million, or 106. For example, 3 MHz stands for 3 × 106 hertz.

Memory cell—A unit of memory used to store a single binary digit, or bit, of data.

Memory word—A number of memory cells logically and physically grouped together. All the cells in a word are typically written to, or read from, at the same time.

Micro—Unit qualifier (symbol = μ) representing one millionth, or 10–6. For example, 3 μS stands for 3 × 10–6 seconds.

Milli—Unit qualifier (symbol = m) representing one thousandth, or 10–3. For example, 3 mS stands for 3 × 10–3 seconds.

Moore's law—In 1965, Gordon Moore (who was to cofound Intel Corporation in 1968) noted that new generations of memory devices were released approximately every 18 months and that each new generation of devices contained roughly twice the capacity of its predecessor. This observation subsequently became known as *Moore's Law*, and it has been applied to a wide variety of electronics trends.

MOSFET (metal-oxide semiconductor field-effect transistor)—A family of transistors.

MSB—(1) (most-significant bit) The binary digit, or bit, in a binary number that represents the most-significant value (typically the left-hand bit). **(2) (most-significant byte)** The byte in a multibyte word that represents the mostsignificant values (typically the left-hand byte).

Multiplexer (digital)—A logic function that uses a binary value, or address, to select between a number of inputs and conveys the data from the selected input to the output.

Nano—Unit qualifier (symbol = n) representing one thousandth of one millionth, or 10–9. For example, 3 nS stands for 3 × 10–9 seconds.

Noise—The miscellaneous rubbish that gets added to an electronic signal on its journey through a circuit. Noise can be caused by capacitive or inductive coupling or by externally generated electromagnetic interference.

Nonvolatile—A memory device that does not lose its data when power is removed from the system.

Nybble—A group of four binary digits, or bits.

One-hot encoding—A form of state assignment for state machines in which each state is represented by an individual state variable, and only one such variable may be "on/active" ("hot") at any particular time.

Operating system—The collective name for the set of master programs that control the core operation and the baselevel user interface of a computer.

OTP (one-time programmable)—A programmable device, such as an SPLD, CPLD, or FPGA, that can be configured (programmed) only a single time.

PAL (programmable array logic)5—A programmable logic device in which the AND array is programmable, but the OR array is predefined.

PCB (printed circuit board)—A type of circuit board that has conducting tracks superimposed, or "printed," on one or both sides and may also contain internal signal layers and power and ground planes.

Peta—Unit qualifier (symbol = P) representing one thousand million million, or 1015. For example, 3 PHz stands for 3 × 1015 hertz.

Pico—Unit qualifier (symbol = p) representing one millionth of one millionth, or 10–12. For example, 3 pS stands for 3 × 10–12 seconds.

PLA (programmable logic array)—The most user configurable of the traditional programmable logic devices because both the AND and OR arrays are programmable.

PLD (programmable logic device)—An *integrated circuit (IC)* whose internal architecture is predetermined by the manufacturer, but which is created in such a way that it can be configured (programmed) by engineers in the field to perform a variety of different functions. For the purpose of this book, the term PLD is assumed to encompass both *simple PLDs (SPLDs)* and *complex PLDs (CPLDs)*. In comparison to an FPGA, these devices contain a relatively limited number of logic gates, and the functions they can be used to implement are much smaller and simpler.

Primitives—Simple logic functions such as BUF, NOT, AND, NAND, OR, NOR, XOR, and XNOR. These may also be referred to as *primitive logic gates*.

PROM (programmable read-only memory)—A programmable logic device in which the OR array is programmable, but the AND array is predefined. Usually considered to be a memory device whose contents can be electrically programmed (once) by the designer.

RAM (random-access memory)—A data-storage device from which data can be read out and into which new data can be written. Unless otherwise indicated, the term *RAM* is typically taken to refer to a semiconductor device in the form of an *integrated circuit (IC)*.

ROM (read-only memory)—A data storage device from which data can be read out, but into which new data cannot be written. Unless otherwise indicated, the term *ROM* is

typically taken to refer to a semiconductor device in the form of an *integrated circuit (IC)*.

RTL (register transfer level)—A *hardware description language (HDL)* is a special language that is used to capture (describe) the functionality of an electronic circuit. In the case of an HDL intended to represent digital circuits, such a language may be used to describe the functionality of the circuit at a variety of different levels of abstraction. The simplest level of abstraction is that of a gate-level netlist, in which the functionality of the digital circuit is described as a collection of primitive logic gates (AND, OR, NAND, NOR, etc.) and the connections between them. A more sophisticated (higher) level of abstraction is referred to as *register transfer level (RTL)*. In this case, the circuit is described as a collection of storage elements (registers), Boolean equations, control logic such as *if-then-else* statements, and complex sequences of events (e.g., "If the clock signal goes from 0 to 1, then load register A with the contents of register B plus register C"). The most popular languages used for capturing designs in RTL are VHDL and Verilog (with SystemVerilog starting to gain a larger following).

Sequential logic—A digital function whose output values depend not only on its current input values, but also on previous input values. That is, the output value depends on a "sequence" of input values.

Silicon chip—Although a variety of semiconductor materials are available, the most commonly used is silicon, and *integrated circuits (ICs)* are popularly known as "silicon chips," or simply "chips."

Slice—The Xilinx term for an intermediate logical partition/entity between a *logic cell (LC)* and a *configurable logic block (CLB)*. Why "slice"? Well, they had to call it something, and—whichever way you look at it—the term *slice* is "something." At the time of this writing, a slice contains two LCs.

SoC (system on chip)—As a general rule of thumb, a SoC is considered to refer to an *integrated circuit (IC)* that contains both hardware and embedded software elements. In the not-so-distant past, an electronic system was typically composed of a number of ICs, each with its own particular function (say a microprocessor, a communications function, some memory devices, etc.). For many of today's high-end applications, however, all of these functions may be combined on a single device, such as an ASIC or FPGA, which may therefore be referred to as a *system on chip*.

Software—Refers to programs, or sequences of instructions, that are executed by *hardware*.

SPLD (simple PLD)—Originally all PLDs contained a modest number of equivalent logic gates and were fairly simple. These devices include PALs, PLAs, PROMs, and GALs. As more *complex PLDs (CPLDs)* arrived on the scene, however, it became common to refer to their simpler cousins as *simple PLDs (SPLDs)*.

SRAM (static RAM)—A memory device in which the core of each cell is formed from four or six transistors configured as a latch or a flip-flop. The term *static* is used because, once a value has been loaded into an SRAM cell, it will remain unchanged until it is explicitly altered or until power is removed from the device.

State diagram—A graphical representation of the operation of a *state machine*.

State variable—One of a set of registers whose values represent the current state occupied by a state machine.

Synchronous—(1) A signal whose data is not acknowledged or acted upon until the next active edge of a clock signal. (2) A system whose operation is synchronized by a clock signal.

Toggle—Refers to the contents or outputs of a logic function switching to the inverse of their previous logic values.

Tri-state function—A function whose output can adopt three states: 0, 1, and *Z* (high impedance). The function does not drive any value in the *Z* state and, when in this state, the function may be considered to be disconnected from the rest of the circuit.

Truth table—A convenient way to represent the operation of a digital circuit as columns of input values and their corresponding output responses.

µC (microcontroller)—A microprocessor augmented with special-purpose inputs, outputs, and control logic like counter timers.

µP (microprocessor)—A general-purpose computer implemented on a single *integrated circuit (IC)* (or sometimes on a group of related chips called a *chipset*).

Verilog—A *hardware description language (HDL)* that was originally proprietary, but which has evolved into an open standard under the auspices of the IEEE.

VHDL—A *hardware description language (HDL)* that came out of the American *Department of Defense (DoD)* and has evolved into an open standard. VHDL is an acronym for *VHSIC HDL* (where VHSIC is itself an acronym for "very high-speed integrated circuit").

VITAL—The VHDL language is great at modeling digital circuits at a high level of abstraction, but it has insufficient timing accuracy to be used in sign-off simulation. For this reason, the VITAL initiative was launched at the *Design Automation Conference (DAC)* in 1992. Standing for *VHDL Initiative toward ASIC Libraries*, VITAL was an effort to enhance VHDL's abilities for modeling timing in ASIC and FPGA design environments. The end result encompassed both a library of ASIC/FPGA primitive functions and an associated method for back-annotating delay information into these library models.

Volatile—Refers to a memory device that loses any data it contains when power is removed from the system, for example, random-access memory in the form of SRAM or DRAM.

Word—A group of signals or logic functions performing a common task and carrying or storing similar data; for example, a value on a computer's data bus can be referred to as a "data word" or "a word of data."

Appendix-C User Manual

The programming and implementation of the system has been done using Xilinx ISE Foundation 6.3i set of tools. These tools come with extensive user manuals which can be consulted if desired.

The complete System is implemented in hardware. Following control inputs are used for the control of hardware.

SWITCH CONTROL

SW1 : SYSTEM RESET
SW4 : TEST INTERRUPT
SW8 : DISBALE SINGLE STEP
SW13 : SWITCH CPU1 OR CPU2
SW16 : ADDRESS REG / PC SELECT

SW8	SW9	EFFECT
0	0	INST TYPE
0	1	IR
1	0	ALU OUT
1	1	STACKPOINTER

Research Publication

Information Technology Journal 5 (1): 192-197, 2006
ISSN 1812-5638

SPE Architecture for Concurrent Execution OS Kernel and User Code

[1]Haroon Muneer and [2]Khalid Rashid
[1]Department of Computer Sciences, International Islamic University, Islamabad, Pakistan
[2]Faculty of Applied Sciences, International Islamic University, Islamabad, Pakistan

Abstract: Operating system performance is not improving at the same rate as the speed of the execution hardware. As a consequence Operating systems are not keeping up with the demands placed on them. Computational speed up due to the increase in processor clock frequency is reaching its limits as well. Chip Multiprocessors are now being investigated to harness the silicon resources now available due to process improvements in C hip manufacturing. This research presents the study into a specialized Chip Multiprocessor for Simultaneous execution of OS kernel and user Code. SPE or Simultaneous Process Execution Architecture allows for continuous execution of OS kernel and user processes.

Key words: OS, kernel, CMP, FPGA

INTRODUCTION

As modern applications become increasingly dependent on multimedia, graphics and data movement, they are spending an increasing fraction of their execution time in the operating system kernel. Web servers have been shown to spend over 85% of their CPU cycles running operating system code[1]. For server-based environments, the operating system is a crucial component of the workload.

Multi-server and component-based operating systems are promising architectural approaches for handling the ever increasing complexity of operating systems. Components or servers (and clients) communicate with each other through cross-domain method invocations. Such interface method invocations, if crossing protection boundaries, are typically implemented through the Inter-process Communication (IPC) mechanisms offered by a microkernel. Therefore, component interaction in such systems has to be highly efficient.

Thus Inter-process Communication (IPC) by message passing is one of the central paradigms of most u-kernel based and other client/server architectures. IPC performance is vital for modern operating systems; especially u-kernel based ones. Since context and user/kernel mode switches are central to IPC operation, reducing them is a critical factor in IPC performance improvement.

A need exists to design a microprocessor that helps to improve OS performance. Specifically by improving IPC performance alone, OS performance can be improved significantly. User/kernel mode switches or context switches are a key operation in IPC. By reducing or by eliminating this context switch IPC performance can be improved significantly.

The SPE architecture platform combines several aspects of existing processor systems. The actual execution of instructions on specialized hardware originates from the earliest co-processor concept in Intel's 8086/8087 processor system [2]. The use of a configurable set of co-processors is a small step towards increasing performance by adding more specialized hardware and has been applied for numerous processor systems. In these architecture platforms, there is one master processor coordinating the activities performed by the co-processors. An example is the TriMedia architecture platform[3] originally developed Philips, which includes a single (VLIW) master processor exploiting instruction level parallelism.

SoC architecture platforms that allow true parallel execution of tasks on a number of independent master processors are also referred to as single-chip multi-processors[4]. The main design issues for such systems emerge from the necessity of communicating information between tasks running on different processors.

There are many academic and commercial CMP in existence today. Specifically the architecture discussed by Theelen and Verschueren[4] is an excellent design example. The MiP architecture platform[5] exploits parallelism at the task level.

For more efficient utilization the offered processing power could also be obtained through a higher integration of software. Although, this approach can reduce overhead and thus increase performance, it may restrict the adaptability for a wide range of products.

Corresponding Author: Haroon Muneer, Department of Computer Sciences, International Islamic University, Islamabad, Pakistan

Contacting slave processors for performing application dedicated operations requires fast on-chip interconnects. On-chip interconnects have become a very important design issue for many SoCs. The challenge is to reduce latencies for exchanging information between units that are located relatively far apart. The on-chip interconnects of the SPE architecture platform can not be compared to packet-based routing devices: because the onchip interconnect is shared stack based and allows direct function calls or OS trap invocations on the connected processors.

Efficiently accessing (off chip) memory has also been a design issue for many years. Similar to other processor systems, the SPE architecture platform uses caches to absorb memory access latencies. Main difference with other multi-processor systems is the absence of data memory for the individual master processors. With one shared memory, much programming flexibility is offered to the user.

The SPE architecture platform exploits parallelism at the task level by incorporating an independent master processor and a number of slave processors. Running multiple tasks in parallel requires sophisticated facilities for inter task communication. The architecture platform considered by Theelen and Verschueren[4] prescribes the use of so-called wrapper units to allow communication between processors. In SPE architecture platform, like describe by Theelen and Verschueren[4], communication between tasks is enabled through the use of communication resources offered by an OS kernel implemented on the master processor.

The main contribution of this study lies in the overall concept of the SPE architecture platform and more specific in the integration of a Master processor specifically designed for OS functionality.

Operating system problems: A u-kernel can provide higher layers with a minimal set of appropriate abstractions that are flexible enough to allow implementation of arbitrary operating systems and allow exploitation of a wide range of hardware.

Similar to optimizing code generators, u-kernels must be constructed per processor and are inherently not portable. Basic implementation decisions, most algorithms and data structures inside a u-kernel are processor dependent. Their design must be guided by performance prediction and analysis. Besides inappropriate basic abstractions, the most frequent mistakes come from insufficient understanding of the combined hardware-software system or inefficient implementation.

For these reasons Operating Systems have been known to cause the following set of problems:

- Operating systems are huge programs that can overwhelm the cache and TLB due to code and data size, thereby causing severe performance penalty for User programs.
- Operating systems may impact branch prediction performance, because of frequent branches and infrequent loops.
- OS execution is often brief and intermittent, invoked by interrupts, exceptions, or system calls and can cause the replacement of useful cache, TLB and branch prediction state for little or no benefit.
- The OS may perform spin-waiting, explicit cache/TLB invalidation and other operations not common in user-mode code, again effecting user code.
- In current modularized kernels, every kernel invocation causes context switch and in case of μ-kernels every call means multiple context switches, thus wasting a considerable time in switching processes.
- IPC-performance problems result from 64 bit architectures with there large number of registers and register stack engines. The large number of registers contributes to a potentially massive context (more than 2 KB) to be stored on each thread context switch[7].
- Overall, operating system code causes poor instruction throughput on a superscalar microprocessor.

To overcome these problems many techniques have been used, but each had its disadvantages. Amdahl's law tells us that if we want modern applications to run quickly, the operating system must run quickly as well. Since traditional performance models essentially ignore the operating system and modern OS-dependent applications, a need has arisen for new designs and methodologies that direct their attention at the performance of the OS kernel[1].

As mentioned earlier OS kernel workload has significantly increased, especially server based applications are putting heavy loads on the kernel. What must be realised is that we have a huge potential for performance improvement. If some how the kernel runs on an independent processor and the user code runs on another, without any bus latencies, this master-slave processor architecture can improve performance significantly. Therefore we have designed a new CMP architecture that is specifically designed to overcome OS problems.

Microprocessor architecture: Internally microprocessors have limited support for operating systems besides the features that are critical for current protected virtual memory based operating systems, like μ-kernels base

operating systems. As we have seen that modern applications are spending an increasing fraction of their execution time in the Operating System (OS) kernel.

At the multi-processor level performance improvements are due to SMP, NUMA or clustering. In each of these techniques the processing nodes are either running a copy of the kernel or the whole OS. None of these are aimed at improving OS performance directly. Rather the earlier mentioned OS problems appear at each node.

Integrated circuit processing technology offers increasing integration density, which fuels microprocessor performance growth. It is becoming possible to integrate a billion transistors on a reasonably sized silicon chip. At this integration level, it is necessary to find parallelism to effectively utilize the transistors. Currently, processor designs dynamically extract parallelism with these transistors by executing many instructions within a single, sequential program in parallel.

However, reliance on a single thread of control limits the parallelism available for many applications and the cost of extracting parallelism from a single thread is becoming prohibitive[8].

The demand for ever faster computer systems seems to be insatiable. Instruction-level parallelism helps a little, but pipelining and superscalar operations rarely win more than a factor of five or ten. To get gains of 50, 100 or even more, the only way is to design computers with multiple CPUs. Thus high level of gain is only promised by parallelism at the processor level. Traditionally the processor level parallelism has used discrete processors. Making one processor master and run the OS is attractive as it solves most of the previously cited problems, but is prone to the bus latencies and hence poor performance.

Researchers have proposed two microarchitectures that exploit multiple threads of control: Simultaneous Multithreading (SMT) and Chip Multiprocessors (CMP). From a purely architectural point of view, the SMT processor's flexibility makes it superior. However, the need to limit the effects of interconnect delays, which are becoming much slower than transistor gate delays, will also drive the billion-transistor chip design. Interconnect delays will force the microarchitecture to be partitioned into small, localized processing elements. For this reason, the CMP is much more promising because it is already partitioned into individual processing cores[9].

Programmers must find thread level parallelism in order to maximize CMP performance. With current trends in parallelizing compilers, multithreaded operating systems and awareness of programmers about how to program parallel computers, this problem should prove less daunting in future. Additionally, having all of the CPUs on a single chip allows designers to exploit thread-level parallelism even when threads communicate frequently.

SPE architecture: In designing our CMP we have used a modified form of Chip Multiprocessors (CMP). The new microprocessor architecture consists of two tightly coupled microprocessors. Both are able to communicate with each other directly and are implemented as a single unit on a single FPGA.

One of the microprocessors is the master processor and implements privileged instruction as well as rest of the instruction set. Operating system alone runs on this microprocessor. The second microprocessor only implements the non-privileged instructions. Complex processor execution units like floating point units and vector units are shared among both processors to avoid complex design and wastage of physical resources.

Only a single slave processor and no complex execution units as well as no caches were implemented to simplify the research effort.

MC-CPU instruction set architecture: MC-CPU instruction set was deigned from ground up to accommodate the new features of this architecture. It is a 32 bit RISC ISA. The SPE implements the MC-CPU Instruction Set Architecture. SPE consists of Master CPU (CPU-1), Slave CPU (CPU-2) and Shared Stack as shown in the Fig. 1. Each of these functional units and their operations are explained individually in the following subsections.

CPU 1: CPU 1 is the master CPU. It implements all the privilege instructions. Only the master processor can access I/O devices. Interrupt handling is performed only by the master processor. Shared stack is also controlled by master processor. The master processor can control the behavior of the slave processor by the means of INTS interrupts. Slave processor implements special interrupt handlers for INTS rather than for the normal interrupts.

CPU 2: CPU 2 is the slave CPU. It does not implement the privilege instructions. The slave processor can not physically access I/O devices. System level interrupt handling is not performed by the Slave processor as it does not have an INT line. Shared stack is accessed by the slave processor when the master processor grants it access. Slave processor implements special interrupt handlers for INTS rather than for the normal interrupts. When the master processor asserts INTS, the slave processor immediately jumps to the particular interrupt based on INTSCODE.

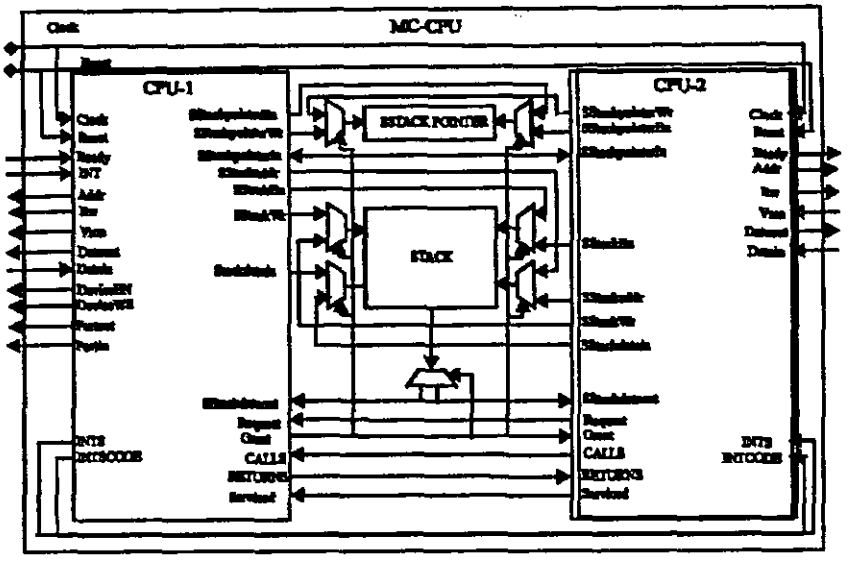


Fig. 1: SPE processor

Shared stack: Shared Stack consists of the stack memory and the shared stack pointer. The shared stack pointer is a 32 bit register. Its output value is constantly supplied to both processors. It can only be modified by one processor at any given time. Shared stack memory consists of single port 1024 bit RAM, arranged as 32 X 32 bits. Only one processor can push or pop from the shared stack at any given time. Shared stack operates in the following manner:

- Master processor has initial control of the shared stack
- Master processor can modify the shared stack pointer any time; only exception is when it has granted control of the shared stack to the slave processor
- Master processor can push or pop values from the shared stack any time; except when it has granted control of the shared stack to the slave processor
- Slave processor can not directly access the shared stack
- Slave processor must assert the REQUEST Signal to gain access to the shared stack
- Whenever REQUEST is asserted by the Slave processor, the master processor can grant or disallow access to the shared stack
- Access is disallowed only when master processor is modifying or accessing the shared stack itself
- Slave processor is blocked or in a wait state during this period
- When the master processor is not accessing the shared stack and the Slave processor requests for it, request is granted by asserting the GRANT signal
- When the GRANT signal is asserted, Slave processor gets access to the shared stack
- Slave processor can now modify both shared stack pointer and shared stack
- After the slave processor has modified the stack it deasserts the REQUEST signal to indicate that the shared stack is now free
- When the GRANT signal is deasserted the master processor deasserts the GRANT signal and takes the control of shared stack back

Remote call: All the communication between the master and slave processor is based on remote calls. In fact these are not remote calls in the classic sense rather these are traps to the OS kernel running on the master processor. Only the slave processor can trap to the master processor by asserting CALLS signal. The remote calls work in the following manner.

- When the user code running on the Slave processor needs some operating system service it must invoke a remote call
- Remote call is invoked by the slave processor by asserting the CALLS signal
- Before asserting the CALLS signal slave processor must request access to the shared stack and at least place the 32 bit service code on top of the shared stack. It can also place any parameters on the stack if there is any
- After placing the service code and/or any parameters on the shared stack, the Slave processor asserts the CALLS signal

- On receiving the CALLS signal Master processor invokes the remote call handler
- Remote call handler checks for user access rights and proper parameters and then calls the appropriate OS function. This is a normal function call
- Normally no context switch takes place during this whole procedure
- After servicing the call and placing return values onto the shared stack the Master processor asserts the RETURNS signal
- On receiving the RETURNS signal, Slave processor request for the shared stack, gets the return values and deasserts the CALLS signal

Remote interrupt: The Master processor controls the slave processor by using Remote Interrupts. Only the Master processor can raise remote interrupts and only the slave processor serves remote interrupts. Interrupt factor table and interrupt service routines for the remote interrupts are placed in the Slave processor's memory space by the Master processor. These interrupts can range from memory management to context switching to process cleanup. Remote interrupts work in the following manner.

- Operating System running on the Master processor can raise remote interrupts
- A remote interrupt is raised by asserting the INTS signal
- Interrupt type is indicated by INTSCODE
- Upon receiving an INTS the Slave processor immediately jumps to the appropriate handler based on INTSCODE
- After servicing the INTS the slave processor assert the SERVICED signal
- Upon receiving the SERVICED signal the Operating System on the Master processor considers the work done and deasserts the INTS signal

Experimental setup: In order to test the SPE processor it was necessary to have a complete computer system with all microprocessor support devices designed and implemented. So, memory, Input and Output multiplexers, a VGA controller, a Keyboard controller and an interrupt controller were also designed and implemented along with the SPE processor.

The whole system (Fig. 2) is implemented as a SoC on a single FPGA. The complete system utilizes approximately 95% of a Spartan-IIE.

The SPE processor achieved a clock speed of 25 MHZ. The system was mounted on a system board based on the arrangement shown in Fig. 3. It was interfaced with the computer using the parallel port. A test program was used to test proper operation of the system.

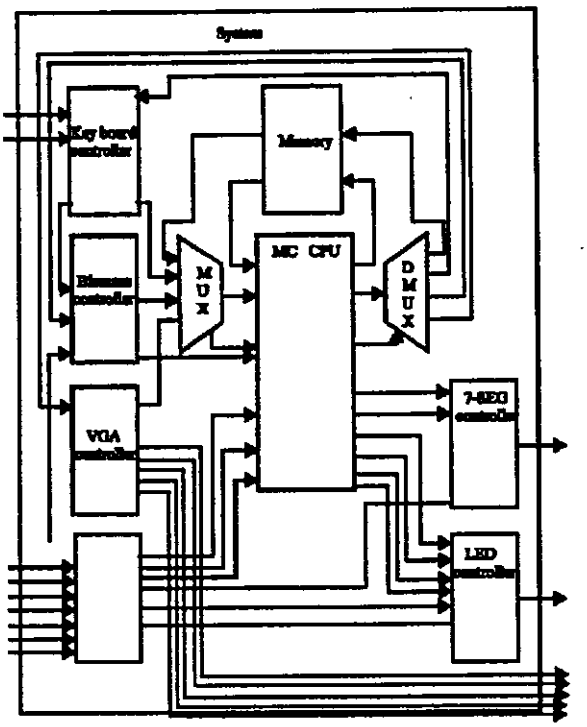


Fig. 2: System architecture diagram

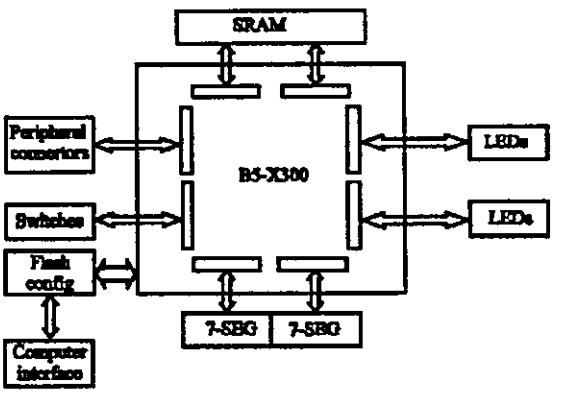


Fig. 3: Hardware arrangement

RESULTS AND DISCUSSION

As a result of this research project we have been able to verify the benefits of the MultiCore design. Specifically, a marked reduction in the context switch penalty. Since, the code running on the master processor is never preempted; it is able to service user requests more efficiently and quickly.

When compared to the SMP systems the outcome is very clear, the main bottle neck is the Interprocessor communication buss. In case of SPE there is no such

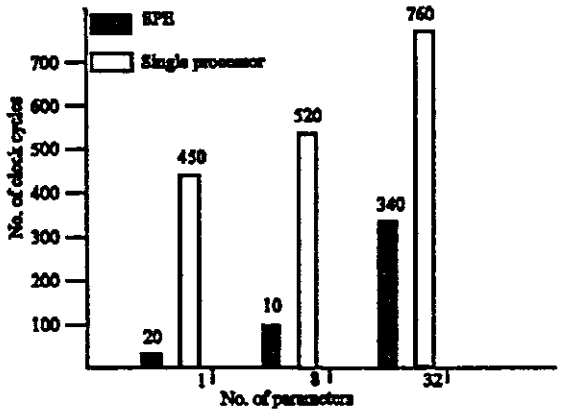


Fig. 4: No. of parameters versus clock cycles in a context switch

external Interprocessor communication buss and hence such latencies are avoided altogether.

When compared to the other CMP processors the SPE architecture does not employ any packet protocol for communication among the two processors. This improves the inter processor communication capability significantly. The downside is that it requires extensive hardware support.

The direct measure of SPE processor performance comes from comparing a piece of code that calls OS services, first on the Master CPU and then on the slave CPU.

When the code is run on the master processor, the system behaves just like a normal single processor system. At every system call performed by the user routine there is a context switch and the OS is switch back. The OS performance the necessary operation and then preempts itself while making the user program active.

In a context switch 43 registers are saved to memory. Saving a single register to memory takes 10 clock cycles. Saving 43 registers requires 430 clocks. In all a single context switch takes about 450 clock cycles on the master processor. This is for an OS service that only requires the service code and no parameters.

On the SPE processor, when the user code is run on the slave processor and it performs a system call then there is no need for a context switch to occur since the OS is running on a separate processor. A remote trap that only passes the service code to the master processor requires only 20 cycles.

It can be easily seen from Fig. 4 that a single context switch requires at least 500 clock cycles whereas a remote trap only requires 10 clock cycles. Thus it can be safely concluded that incorporating features at the microarchitecture level can improve IPC performances significantly. Improvements in IPC performance improve OS performance significantly.

FUTURE WORK

As next to investigating possible extensions, we are currently developing compiler tools capable of handling the offered flexibility. The ultimate goal would be to develop tools that enable fast software compilation by mapping specific performance requirements of an application into partitioned code. On of the code partitions will run on the master processor as a service for the bulk of code running on the slave processors.

Our future research will also concentrate on a method for analyzing which configuration of master processor and Slave processors will meet the requirements for a specific application in an optimal way.

REFERENCES

1. Brown, A.B., 1997. A Decompositional Approach to Computer System Performance Evaluation. Center for Research in Computing Technology Harvard University Cambridge, Massachusetts.
2. Intel Corp., 1993. Microprocessors Volume II, pp: 7-1-7-30 and 7-90-7-111.
3. Ratham, S. and G. Slavenburg, 1996. An architectual overview of the programmable multimedia processor TM-1. In: Proceedings of COMPCON 96, Santa Clara, California, USA, 25– 28 Febraury, IEEE Computer Society, Los Alamitos, California, USA., pp: 319-326.
4. Theelen, B.D. and A.C. Verschuere, 2003. Architecture design of a scalable single-chip multi-processor. J. Sys. Architecture, Special Issue on Systems and Verification, 49: 619-639.
5. Theelen, B.D., A.C. Verschueren, V.V. Suarez, M.P.J. Stevens andA. Nunez, 2003. A scalable single-chip multi-processor architecture with on-chip RTOS kernel. J. Sys. Architecture, pp: 619-639.
6. Bergamaschi, R., I. Bolsens, R. Gupta, R. Harr and A. Jerraya *et al.*, 2001. Are Single-chip Multiprocessors in Reach? In: Wolf W, K. Roy Eds.), IEEE Design and Test of Computers, IEEE Computer Society, Los Alamitos, California, USA, 18: 82-89.
7. Liedtke, J., U. Dannowski, K. Elphinstone, G. Liefländer and E. Skoglund *et al.*, 2001. The L4Ka Vision (White paper).
8. Lance, H., B.A. Nayfeh and K. Olukotun, 1997. A Single-Chip Multiprocessor. In: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp: 2-11.