EVALUATING HARDWARE/SOFTWARE PARTITIONING AND AN EMBEDDED LINUX PORT OF THE VIRTEX-II PRO DEVELOPMENT SYSTEM

By

HSIANG-LING JAMIE LIN

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2006

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of HSIANG-LING JAMIE LIN find it satisfactory and recommend that it be accepted.

____
Chair

____

____

ACKNOWLEDGMENT

I would like to express my gratitude to those who have helped me academically and personally in the completion of the thesis. Many thanks to my advisor, Dr. Jabulani Nyathi, for his assistance on my research and efforts in searching for support. I have learned a great deal of knowledge from him. He has been very patient and kind with a paranoid student like myself. I acknowledge Clint Cole and Digilent Inc., for all the help during the years I have been in Pullman and the sponsorship on my research.

Thank you Hui-ling Lin and Chuan Shiu, for all personal favors over the years and for offering me a great place to stay when I need a break from school. I would like to acknowledge Lee Verberne, for his inspiration in the technical field. Thank you Min Lin, for being a good friend. I wouldn't have gone abroad for college at the first place without meeting an adventure pal like you. Thank you Judi and Gary Wutzke, for the love and support as a family. Thank you my friends, especially Charles Chen, Wendy Chao, Priscilla Septiany, and Yuki Makino, for the great friendship and encouragement.

Last, I would like to acknowledge myself. Without my persistence in all difficulties I have faced, I wouldn't have gone this far.

EVALUATING HARDWARE/SOFTWARE PARTITIONING

AND AN EMBEDDED LINUX PORT OF THE VIRTEX-II

PRO DEVELOPMENT SYSTEM

Abstract

by Hsiang-ling Jamie Lin, M.S.
Washington State University
May 2006

Chair: Jabulani Nyathi

The embedded system application space is growing at a fast pace and has a very wide range that encompasses minute sensor nodes through large FPGA based systems with multiple embedded processors within a single chip. Regardless of the application type and size; testing, monitoring and debugging of these systems remain central to their success as solutions to today's problems. The Virtex-II Pro development system offered by Xilinx is an embedded development environment that has benefited greatly from the system on chip design approach. It is a programmable system with two embedded IBM Power PCs and an FPGA all of which are connected via IBM's core-connect bus. This makes the system suitable for emulating applications in actual hardware while offering at speed testing. This thesis examines several embedded systems design considerations such as the hardware/software partitioning and the timeliness of event handling. The objective is to provide a stable development environment that exploits the hardware features of the board to allow for ease of use particularly in the educational sector. Digital adaptive filtering is considered to demonstrate the benefits and flexibility offered by this development system. Significant performance gains are recorded with a well-partitioned

finite impulse response filter showing that the software-based filter is outperformed by 72%.

Another aspect of this research is to port an embedded operating system to manage the hardware and offer design flexibility. The embedded Linux kernel has been considered as the suitable real-time operating system (RTOS) and the first challenge is to ensure that the embedded cores are simultaneously visible to the operating system and user under shared memory system environment. This approach has been chosen with the view that tasks executing on any of the processors will for the most part be required to work towards a common goal. The shared memory approach has not been a success due to the cache coherence issues, however, sample device drivers under the Linux kernel have been written and the kernel successfully ported to run on a single processor.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS iii

ABSTRACT iv

LIST OF TABLES ix

LISTOF FIGURES x

CHAPTER

1. INTRODUCTION 1

1.1 The Xilinx Virtex-II Pro Platform FPGAs 1

1.2 The Xilinx EDK 3

1.3 Research Brief 6

1.4 Conclusion 7

2. BACKGROUND INFORMATION 8

2.1 The Virtex-II Pro Development System 8

2.2 Embedded Applications Design 10

2.2.1 Design Specification and Components 10

2.2.2 Implementation 12

2.3 Conclusion 17

3. DISTRIBUTED SYSTEM V.S. SHARED MEMORY SYSTEM 19

3.1 Parallel Computing 19

3.2 Using Dual PowerPC Cores 22

3.2.1 A Simple Shared-Memory System 22

3.3 The Virtex-II Pro Distributed System 28

3.4 Conclusion 28

4. EMBEDDED LINUX ON THE VIRTEX-II PRO SYSTEM 29

4.1 Background 29

4.2 Embedded Linux on the Virtex-II Pro Development System 29

4.3 Requirements for the Linux Port 31

4.4 Porting Linux 34

4.5 Comments on Symmetric Multiprocessing 42

4.6 Linux Device Drivers 42

4.6.1 Loadable Kernel Module 43

4.6.2 Loadable Kernel Module with User Programs 44

4.7 Conclusion 46

5. HARDWARE SOFTWARE PARTITIONING 47

5.1 Background 47

5.1.1 Digital Signal Processing 48

5.2 FIR Filter Design 48

5.2.1 FIR Filter Specifications 52

5.2.2 FIR Filter Plots 53

5.2.3 Software Based Audio Filtering Design 56

5.2.4 Hardware Software Partitioning Audio Filtering Design 56

5.3 System Performance 57

5.4 Adaptive Filtering 59

5.5 Conclusion 60

6. RESEARCH CONTRIBUTIONS AND FUTURE WORK 62

BIBLIOGRAPHY 63

APPENDIX

A. NETWORKING APPLICATION 66

B. DUAL-CORE DESIGN 75

C. LOADABLE MODULE 77

D. ENHANCED LOADABLE MODULE 80

E. SOFT AUDIO FILTERING APPLICATION 84

F. HARD/SOFT AUDIO FILTERING APPLICATION 87

LIST OF TABLES

Page

1. Table 5.1: FIR Filter specifications 53
2. Table 5.2: Execution time measurements for the band-pass filter 58
3. Table 5.3: Execution time measurements for the low-pass filter..................................58
4. Table 5.4: Execution time measurements for the high-pass filter 58
5. Table 5.5: Execution time measurements for the AC97 codec read/write 59
6. Table 5.6: Voice filter specification..60

LIST OF FIGURES

Page

1. Figure 1.1: System evolution [1] 1
2. Figure 1.2: The Virtex-II Pro development system 3
3. Figure 1.3: The Xilinx EDK tool chain [1] 4
4. Figure 2.1: System block diagram 14
5. Figure 2.2: Peripherals in the system 14
6. Figure 2.3: Bus connections 15
7. Figure 2.4: Sample outputs 16
8. Figure 3.1: Shared memory system 20
9. Figure 3.2: Distributed memory system 21
10. Figure 3.3: Hybrid distributed-shared memory system 21
11. Figure 3.4: Shared memory system block diagram 23
12. Figure 3.5: Bus connections for the shared memory system 24
13. Figure 3.6: BRAM ports connected to the controller ports 24
14. Figure 3.7: Address Map 25
15. Figure 3.8: Ports for PPC1 26
16. Figure 3.9: Added ports for PLB and BRAM controller connected to PPC1 26
17. Figure 3.10: Sample Outputs 27
18. Figure 4.1: Loadable module output 44
19. Figure 4.2: Enhanced loadable module output 45
20. Figure 5.1: Sample plot #1 of the low-pass filter 54
21. Figure 5.2: Sample plot #2 of the low-pass filter 54
22. Figure 5.3: Sample plot #1 of the high-pass filter 55

23. Figure 5.4: Sample plot #2 of the high-pass filter 55

24. Figure 5.5: System flow diagram for the software based audio filtering design 56

25. Figure 5.6: System flow diagram for the hardware/software audio filtering design 57

26. Figure 5.7: Input/output plot for the adaptive filter 60

Dedication

This thesis is dedicated to my father, who has passed.

He would be so proud of my achievement in education like he had been.

This thesis is dedicated to my family, especially my mother.

They have supported my stubborn and determined mind with love,

no matter how unreasonable I am sometimes.

CHAPTER ONE

INTRODUCTION

1.1 The Xilinx Virtex-II Pro Platform FPGAs

Before we study embedded systems, it is useful to look at the system evolution. The following figure shows the development of computer systems through the time.

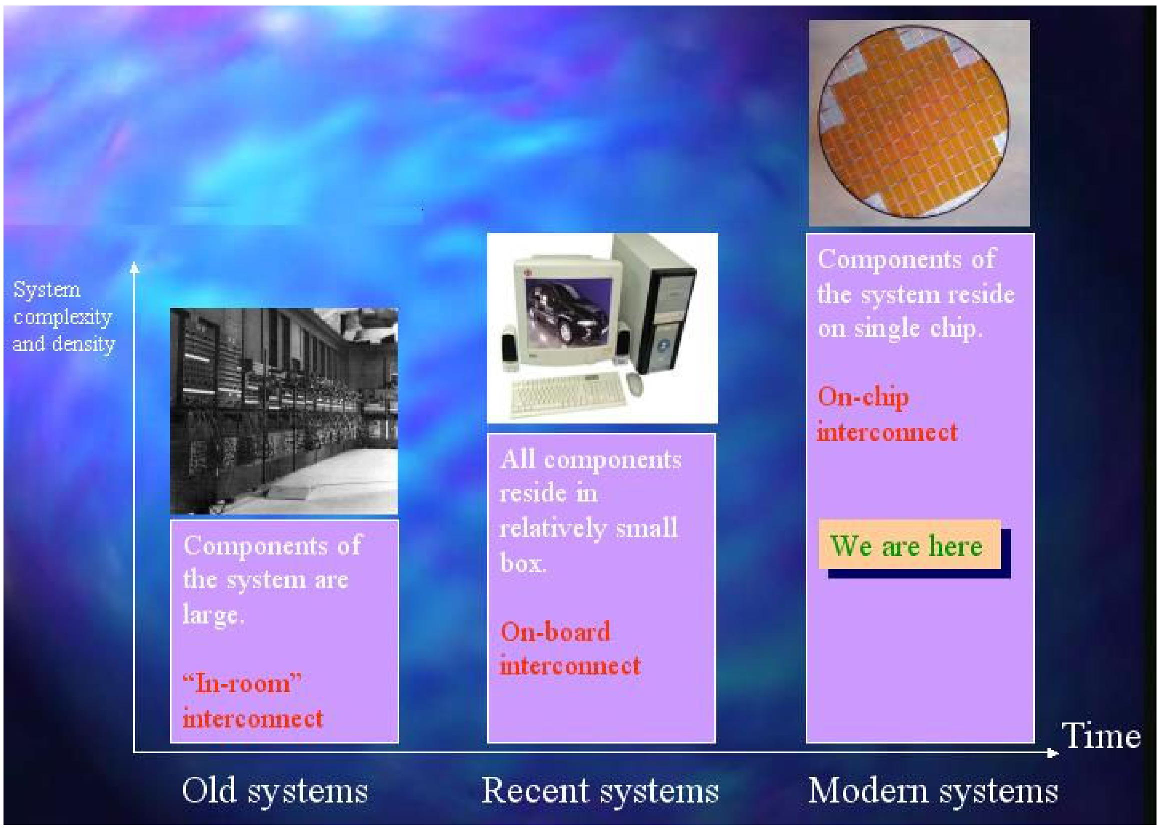


Figure 1.1: System evolution [1]

From a system that occupied an entire room to a handheld computer that has the size of a pencil box, System-on-a-Chip (SoC) has become an important role in the system development. The embedded computing filed is growing fast, and new technologies continue to be developed. The Xilinx Virtex-II Pro FPGA is such an example. It has taken system-on-a-chip to the next level: system on a programmable chip.

Field Programmable Gate Array (FPGA) and microprocessors accomplish different tasks. FPGAs are configurable and re-programmable digital logic devices, and programming code is usually written in Hardware Description Languages (HDL). Microprocessors execute predefined commands, and do not have much flexibility. Engineers usually write programs for microprocessors in a language such as C. As applications become more complex, use of one or the other becomes insufficient. Traditionally, engineers program an FPGA and a microprocessor individually. If there is a need for the two to communicate, a link can be established by manually setting it up using expansion connectors or other techniques. The Virtex-II Pro development system (Figure 1.2) integrates two technologies. It has two hard PowerPC405 processors, one soft MicroBlaze processor, and an FPGA on a single chip. With such a powerful chip, design of embedded applications becomes more flexible and efficient. The communication latency between a processor and hardware Intellectual Property (IP) is reduced, because of direct connections between them and they even share memory. The FPGA and processors co-existence feature allows us to perform hardware software partitioning. We can also take the advantage of dual cores to design a parallel computing system. Other benefits of this integration include ease of testing, monitoring and debugging both hardware and software components of the system on a chip. The development system also comes with expansion connectors. Commonly used devices such as the USB port, serial port, audio, video, and so on are already built on this board.

However, if a project requires some external device, it can be attached through these expansion connectors.

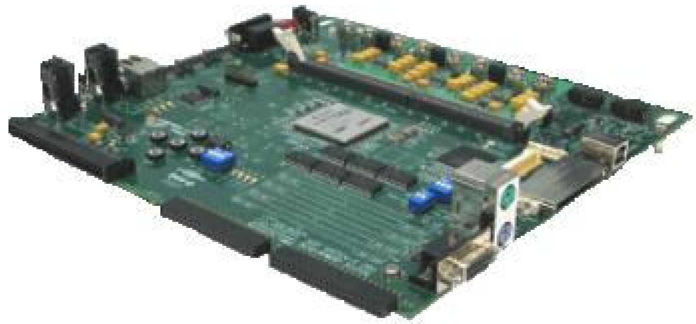


Figure 1.2: The Virtex-II Pro development system

1.2 The Xilinx EDK

The Xilinx Embedded Development Kit (EDK) contains Embedded System Tools (EST), documentation, and Hardware IPs for the Xilinx embedded processors and peripherals [2]. Every embedded system design using the Xilinx EDK is divided into two parts: hardware design and software design. The Xilinx Platform Studio (XPS) provides an Integrated Development Environment (IDE) that combines hardware and software designs in one interface. The hardware specification and corresponding libraries can be generated based on user selections. We can access hardware components in a processor with appropriate drivers. We can also design custom IP cores for specific embedded system requirements. Once the hardware design is implemented and software programs are compiled, they can be combined into a bitstream and downloaded to the target system. Below is a detailed block diagram that shows the system design flow using the Xilinx EDK.

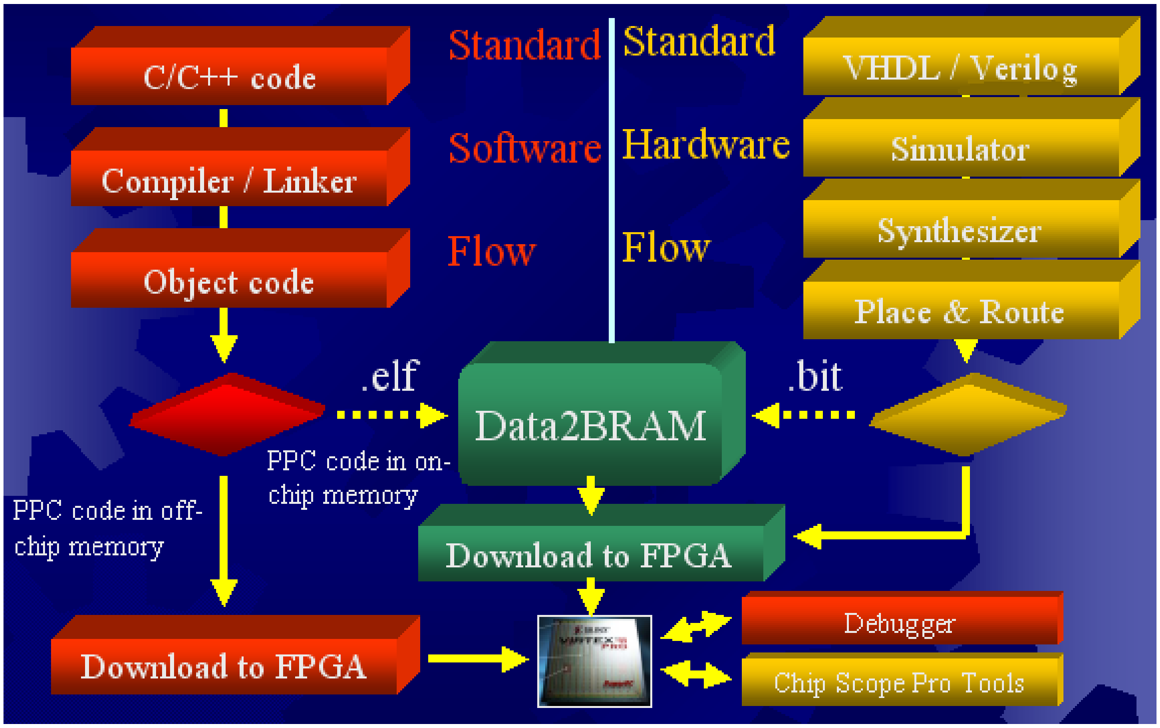


Figure 1.3: The Xilinx EDK tool chain [1]

At the end of hardware design, we will have a downloadable bitstream. At the end of software design, we will have an executable binary for the software program. If the software program uses on-chip block RAMs, the object code (.elf) can be combined with the hardware bitstream (.bit) to form a new downloadable bitstream using the DATA2BRAM utility. DATA2BRAM takes the .bit file as an input, and adds new block RAM contents if there is any. This eliminates the need to re-implement the entire system after modifications have been made to the software. To debug a software program, we can use the software debugger that comes with the Xilinx EDK. Hardware debugging requires Xilinx ChipScope, if no other device such as an oscilloscope or a logic analyzer is available. The following lists the detailed steps for designing an embedded application using the Xilinx EDK [3]:

Creating an Embedded Hardware System

This first step is to create a hardware platform, which contains processor information, buses, and peripheral devices attached to the processors. This is done in the Base System Builder (BSB) wizard. The Platform Generator (PlatGen) takes the hardware specification file, and generates netlists for the system.

Creating Software for the Embedded System

XPS allows us to add software applications to available processors. The applications can be compiled with GCC like compilers.

Software Libraries

Library Generator (LibGen) is used to generate software libraries. The output of LibGen is based on the hardware platform and user selected internal libraries.

System Implementation

There are two ways to implement an embedded system design using the Xilinx EDK: Xflow and ISE Integration. Xflow implements a design directly in XPS. The main advantage of using Xflow is that we can have the entire design done in one GUI. However, Xflow does not give us direct controls of synthesis and implementation options. On the other hand, ISE and XPS integration allows us to control implementation and synthesis for the design, and add additional logic to the FPGA. The only drawback is that we have to work with two different software interfaces.

Initialize the System and Download to the Board

The last step involves updating the hardware bitstream, and downloading the most recent bitstream to the board. The Virtex-II Pro development system configures the board through JTAG. It automatically scans the JTAG chain and downloads the bitstream it finds.

1.3 Research Brief

The Virtex-II Pro development system is ideal for research and embedded system class projects. Students can experiment embedded applications on different devices on the board. By working with such a complex system, students can gain valuable experience in the embedded system field. A lot of the potential for this board remains unexplored. For example, running applications in embedded Linux on the board has been mentioned, but no work has been presented. Therefore, it is also an excellent choice for system-on-a-chip research projects.

The focus of the research has been to investigate potential educational experiments, ranging from an embedded operating system port to performance benefits offered by the integration of the embedded cores and the FPGA. Of particular interest is the investigation of the benefits and efficiency of system on a programmable chip, particularly the hardware software partitioning. With the Virtex-II Pro FPGA, we will be able to split tasks for computation efficiency. For example, the hardware multipliers can be used to improve performance of multiplication, which is slow if done in a processor. We can easily reduce the workload for processors by having hardware handle certain tasks, if such hardware has better performance over processors for these tasks. Since each processor has its own memory, we can have the processors perform independent tasks or work together towards a common goal. This research examines the above aspects of the Virtex-II Pro development system. The research also presents the pros and cons of having and not having an embedded operating system on this particular platform.

1.4 Conclusion

This chapter presents the research brief, and introduces the Xilinx Virtex-II Pro Platform FPGAs and EDK, which will be used for the research. System on a programmable chip takes the design of embedded applications to a new level. This research will explore the potentials and capabilities of the Virtex-II Pro Platform FPGAs.

In Chapter 2 we present information on the development system architecture, highlighting primarily features of interest. Chapter 3 presents parallel computing basics and how to build a dual-core system, while in Chapter 4 the Linux port along with host development system requirements are presented. The hardware and software partitioning approach is discussed in Chapter 5, and an elaborate digital signal processing example is also described and implemented in this chapter. Chapter 6 provides some concluding remarks and some directions on future work.

CHAPTER TWO

BACKGROUND INFORMATION

2.1 The Virtex-II Pro Development System

The Virtex-II Pro development system has just been introduced, and this chapter will present a deep look at its capability. The following is a list of important features of the development system [4]:

- Virtex-II Pro XC2VP30 FPGA with 30,816 Logic Cells, 136 18-bit multipliers, 2,448Kb of block RAM, one MicroBlaze Soft Processor, and two PowerPC405 Hard Processors
- DDR SDRAM DIMM that can accept up to 2-Gbyte of RAM
- 10/100 Ethernet port
- USB2 port
- Compact Flash card slot
- XSGA Video port
- Audio Codec
- SATA (Serial Advanced Technology Attachment), PS/2, and RS-232 ports
- High and Low Speed expansion connectors with a large collection of available expansion boards
- System ACE™ controller and Type II CompactFlash™ connector for FPGA configuration and data storage

Traditional embedded applications are controlled by micro-controllers. The micro-controller collects data, does computations, and then transfers data to some display if any. What tasks a micro-controller must perform depends on the application, but the micro-controller unit (MCU) has to do all the computational work. With the introduction of Virtex-II Pro Platform FPGAs, embedded application designs have moved to a new level of flexibility and efficiency. The IBM PowerPC405 core is a 32-bit RISC processor. It implements the 5-stage data path pipeline, and has 32 32-bit general-purpose registers and 16KB instruction and data caches. PowerPC405 processors have dedicated Harvard architecture controllers to interface instruction and data On-Chip-Memory (OCM). OCM is used as additional memory to the instruction and data caches, and provides memory-access performance same as a cache hit. The PowerPC405 is an implementation of the PowerPC embedded environment architecture. It provides high performance at low power consumption for embedded applications [5]. MicroBlaze is a soft processor. It is implemented using general logic primitives instead of a dedicating block in the FPGA. The MicroBlaze soft core allows a user to control the cache sizes and execution units [6]. It does not implement a Memory Management Unit (MMU), and so only operating systems lacking of MMU such as uClinux can be ported. This research focuses on the PowerPC processors. Further details of using the soft core are beyond the scope of the topic and will be omitted.

Evidently, with such a powerful FPGA, complex arithmetic and logic operations can be done in the hardware efficiently. The processors can handle software tasks as needed. The FPGA and processor co-design capability allows for easily solving complex engineering problems in a timely manner and within such a small system.

The bus architecture used on the Virtex-II Pro development system is the IBM CoreConnect standard. The architecture allows engineers to assemble custom SoC designs on the cores that support CoreConnect specifications. Processor Local Bus (PLB), On-chip Peripheral Bus (OPB), and Device Control Register (DCR) bus are included in the CoreConnect standard [7]. The PLB is fully synchronous and supports up to 16 master and 16 slave high bandwidth devices. The OPB is fully synchronous and supports up to 16 master and an unlimited number of slave lower bandwidth devices. The DCR bus provides processor blocks a mechanism to control peripheral devices on the FPGA. The bus architecture reduces the time and costs for SoC designs.

2.2 Embedded Applications Design

The Xilinx Platform Studio enables us to design both hardware and software specifications in one interface. There are many built-in peripherals we can choose from. We can also design custom peripherals. The development interface allows us to view system block diagrams, bus connections, address maps, and other design related components. To experiment with the Xilinx EDK in building embedded applications, a networking embedded application has been implemented. This design combines important features that a designer can use in Platform Studio.

2.2.1 Design Specification and Components

The design shows how to use UART, onboard general-purpose input/output (GPIO) registers, and Ethernet. A multiplier peripheral is added to show how a custom IP can be designed and imported to a project. The application does the following: When the server starts to run, the application accepts a user input from a web browser. The input value should be a hexadecimal number between 0 and F. The server processes this data, and the value is displayed to the 4-bit LEDs and in the browser window. The multiplier takes the user input and multiplies it by 4 in the FPGA. The processor then reads the product from the FPGA and displays it to a terminal. The terminal shows the current input value, product, and web connection information. The exercise shows the benefits of system on a programmable chip enabling a user to interface the embedded core with the FPGA work that would otherwise require complex communication between the FPGA and the development host. The required components for this design are listed below:

PLB Ethernet

Ethernet is considered as a high bandwidth IP, so it is attached to the processor as a PLB device. Xilnet is one of the Xilinx EDK built-in libraries. It provides functions for networking. For example, socket(), bind(), receive(), send(), etc. Xilinx has customized the standard networking functions to adapt their devices, so the usage might be slightly different. Before using any of the functions, one should consult the library specifications. To use Xilnet functions, we need to associate the Ethernet_Mac device to the library, and run the Library Generator (LibGen) to generate corresponding libraries. To enable Xilnet, we simply select the option in the Software Platform Settings window.

OPB LEDs

The LEDs are useful for displaying outputs. In this experiment, it is used to display the user input from a web browser.

OPB RS232 UART

We will need a way to check if the application works correctly. The UART is used as the standard output for viewing the current information in the running application.

PLB BRAM Controller: 64KB

64KB of BRAM is used to store the application.

OPB Multiplier

This is a custom IP that multiplies two 32-bit numbers. The peripheral is used to show how a custom IP can be designed and imported to the embedded application. It takes a user input from a web browser, and multiplies the input by 4. The product is read by the processor to display in a terminal.

2.2.2 Implementation

Based on the components of the list above, a base system consisting of the following components has been built:

- PowerPC
- Jtag PPC
- 4-Bit LEDs
- Ethernet_MAC controller: default setting
- RS232 Uart: baudrate at 115200
- PLB BRAM Controller: 64KB

A multiplier custom IP is built using the Create/Import Peripheral Wizard. The IP has three registers: Reg0 and Reg1 for inputs, and Reg2 for the product. The read and write processes are modified to meet the multiplier requirements. Also, a process is added for the multiplication function. The following shows the code for these processes in the user_logic.vhd file:

```
MUL_PROC : process( Bus2IP_Clk ) is
begin
  if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
    if Bus2IP_Reset = '1' then
```

```
      slv_reg2 <= (others => '0');
    else
      slv_reg2 <= slv_reg0 * slv_reg1;
    end if;
  end if;
end process MUL_PROC;

SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
begin

  if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
    if Bus2IP_Reset = '1' then
      slv_reg0 <= (others => '0');
      slv_reg1 <= (others => '0');
    else
      case slv_reg_write_select is
        when "100" => slv_reg0 <= Bus2IP_Data(0 to C_DWIDTH-1);
        when "010" => slv_reg1 <= Bus2IP_Data(0 to C_DWIDTH-1);
        when others => null;
      end case;
    end if;
  end if;
end process SLAVE_REG_WRITE_PROC;

SLAVE_REG_READ_PROC : process( slv_reg_read_select, slv_reg0, slv_reg1, slv_reg2 ) is
begin

  case slv_reg_read_select is
    when "100" => slv_ip2bus_data <= slv_reg0;
    when "010" => slv_ip2bus_data <= slv_reg1;
    when "001" => slv_ip2bus_data <= slv_reg2;
    when others => slv_ip2bus_data <= (others => '0');
  end case;

end process SLAVE_REG_READ_PROC;
```

When a read is issued in the processor for reg2, the product of reg0 and reg1 will be sent from the FPGA. The following shows the system block diagram:

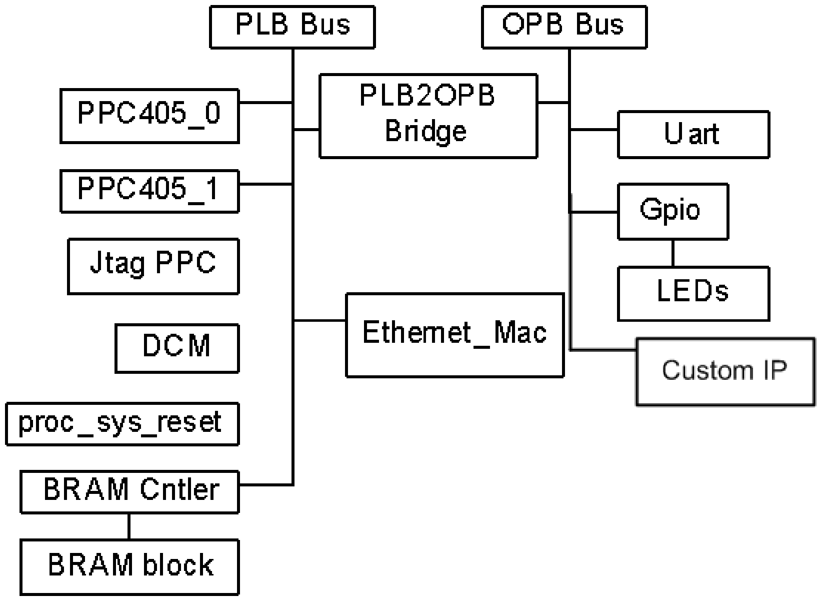


Figure 2.1: System block diagram

After adding the custom IP, the following peripherals and bus connections exist:

Peripherals | Bus Connections | Addresses | Ports | Parameters

Cells with white backgrounds can be edited. To delete peripherals, choose one or more rows and click Delete.

Peripheral	HW Ver	Instance
ppc405	2.00.c	ppc405_0
ppc405	2.00.c	ppc405_1
jtagppc_cntlr	2.00.a	jtagppc_0
proc_sys_reset	1.00.a	reset_block
plb2opb_bridge	1.01.a	plb2opb
opb_uartlite	1.00.b	RS232_Uart_1
plb_ethernet	1.01.a	Ethernet_MAC
opb_gpio	3.01.b	LEDs_4Bit
plb_bram_if_cntlr	1.00.b	plb_bram_if_cntlr_1
bram_block	1.00.a	plb_bram_if_cntlr_1_bram
dcm_module	1.00.a	dcm_0
multiplier	1.00.a	multiplier_0

Figure 2.2: Peripherals in the system

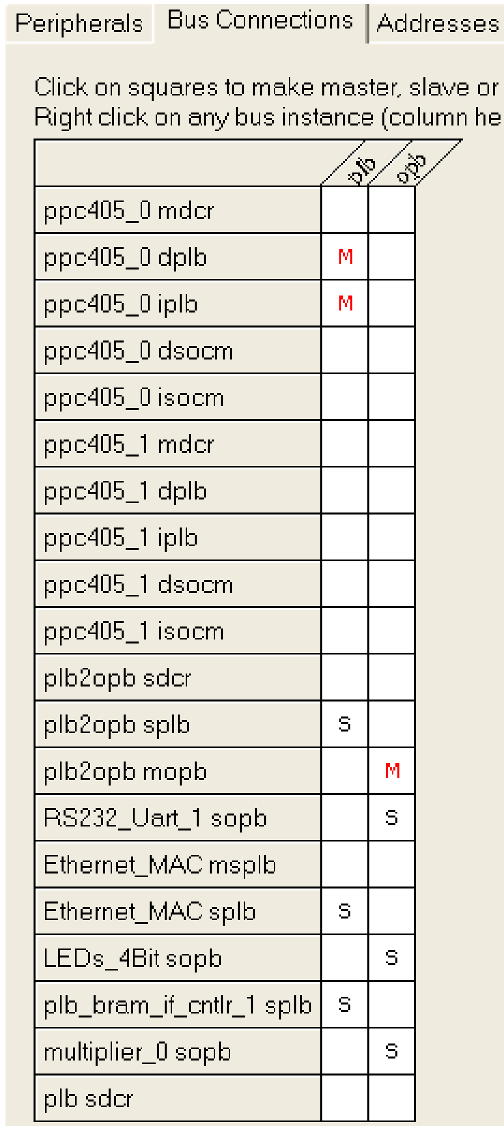


Figure 2.3: Bus connections

The software application code can be found in Appendix A. The following is a screenshot of sample outputs:

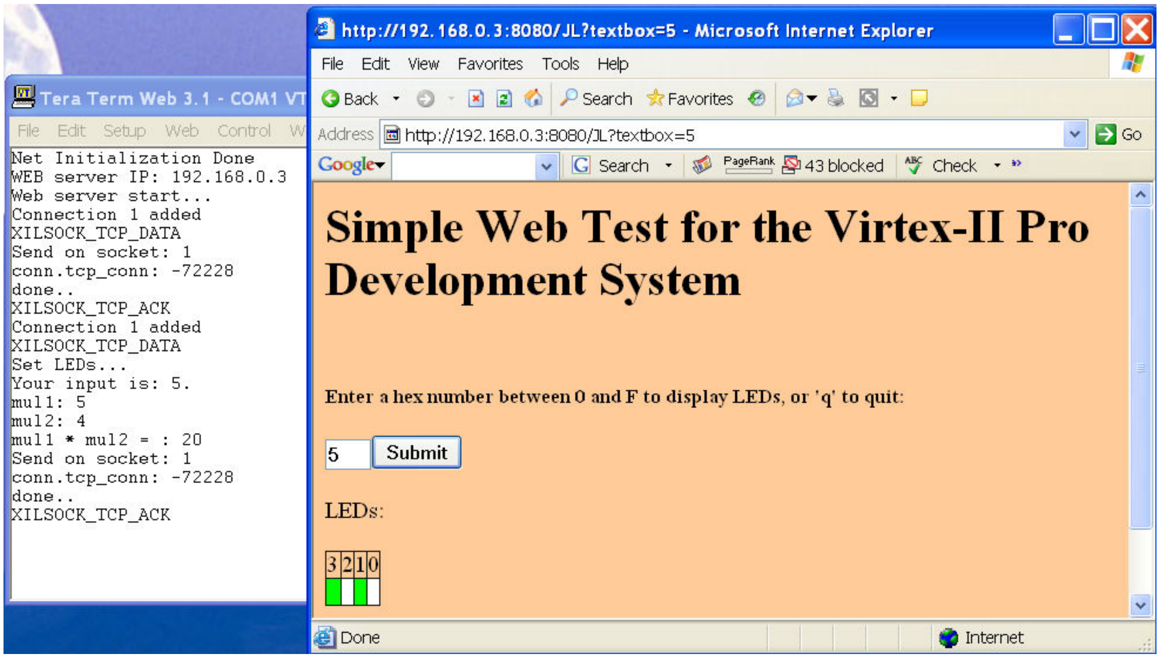


Figure 2.4: Sample outputs

Output Log:

```
Net Initialization Done
WEB server IP: 192.168.0.3
Web server start...
Connection 1 added
XILSOCK_TCP_DATA
Send on socket: 1
conn.tcp_conn: -72228
done..
XILSOCK_TCP_ACK
Connection 1 added
XILSOCK_TCP_DATA
Set LEDs...
Your input is: 5.
mul1: 5
mul2: 4
mul1 * mul2 = : 20
Send on socket: 1
conn.tcp_conn: -72228
done..
XILSOCK_TCP_ACK
Connection 1 added
XILSOCK_TCP_DATA
Set LEDs...
Your input is: 9.
mul1: 9
mul2: 4
mul1 * mul2 = : 36
Send on socket: 1
```

```
conn.tcp_conn: -72228
done..
XILSOCK_TCP_ACK
Connection 1 added
XILSOCK_TCP_DATA
Set LEDs...
Your input is: 12.
mul1: 12
mul2: 4
mul1 * mul2 = : 48
Send on socket: 1
conn.tcp_conn: -72228
done..
XILSOCK_TCP_ACK
Connection 1 added
XILSOCK_TCP_DATA
Set LEDs...
Your input is: 0.
mul1: 0
mul2: 4
mul1 * mul2 = : 0
Send on socket: 1
conn.tcp_conn: -72228
done..
XILSOCK_TCP_ACK
Connection 1 added
XILSOCK_TCP_DATA
Set LEDs...
Your input is: 3.
mul1: 3
mul2: 4
mul1 * mul2 = : 12
Send on socket: 1
conn.tcp_conn: -72228
done..
XILSOCK_TCP_ACK
```

2.3 Conclusion

This chapter provides background information of the Virtex-II Pro development system. The sample design demonstrates how an embedded application can be implemented using the Xilinx EDK. This design can be served as a laboratory experiment, since its complexity would give students a chance to learn the Xilinx EDK and design a small yet complex embedded application that uses one of the two embedded cores and the FPGA. The Xilinx EDK and the Virtex-II Pro development system together provide engineers with a flexible embedded application development environment. We can attach new HDL components to meet design requirements. This feature allows us to easily add and remove devices in an operating system, which will be shown in Chapter 4.

CHAPTER THREE

DISTRIBUTED SYSTEM V.S. SHARED MEMORY SYSTEM

3.1 Parallel Computing

Traditionally, a computer solves problems by executing a series of instructions in a processor, and only one instruction can be executed at a time. Parallel computing in short refers to more than one processors running simultaneously to solve one problem. It is a method to speed up computation. We can have all processors run the same instructions. We can also split a task into smaller sub-tasks, and assign each processor some sub-tasks. These processors are running in parallel to solve the computational problem. Parallel computing is an ideal solution for systems that have interrelated events happening at the same time, require complex numerical simulations, or process a large amount of data. Examples of parallel computing systems include weather patterns, automobile assembly line, manufacturing processes, web search engines, corporation managements, etc. Memory architectures for parallel computing can be classified into three categories [8]:

Shared Memory

Processors share the same memory resources. This is often seen in a computer with multiple processors. Changes made by a processor in the shared memory region must be visible to all other processors. Because we have two processors in the Virtex-II Pro FPGA, and we are free to use available memory on the board, a shared memory system can be built in this particular platform.

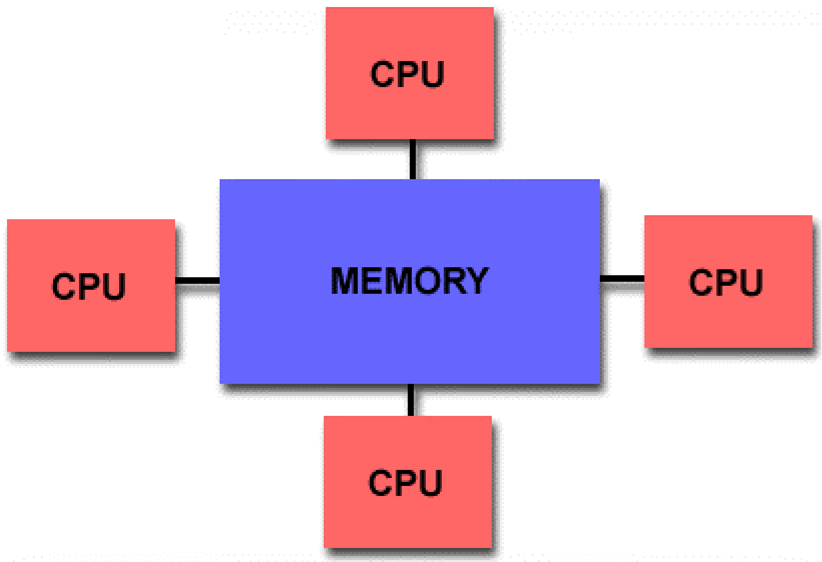


Figure 3.1: Shared memory system

Distributed Memory

Multiple processors are running in a network, but appear to a user as a single system. Each processor has its own local memory and operates independently. Changes in the local memory do not have effects in the memory of other processors. If a processor needs to access data in other processors, a communication link must be established. An example of distributed system would be an automatic banking system. Any machine in the system must be informed when a transaction occurs to an account. The system looks like one computer to a user, but in fact many machines are running together to maintain bank accounts. It is possible to establish a distributed system using the Virtex-II Pro development systems. Each processor can perform independent work, and share resources through a communication protocol. Because we have Ethernet available on the board, networking can be used for inter-communication.

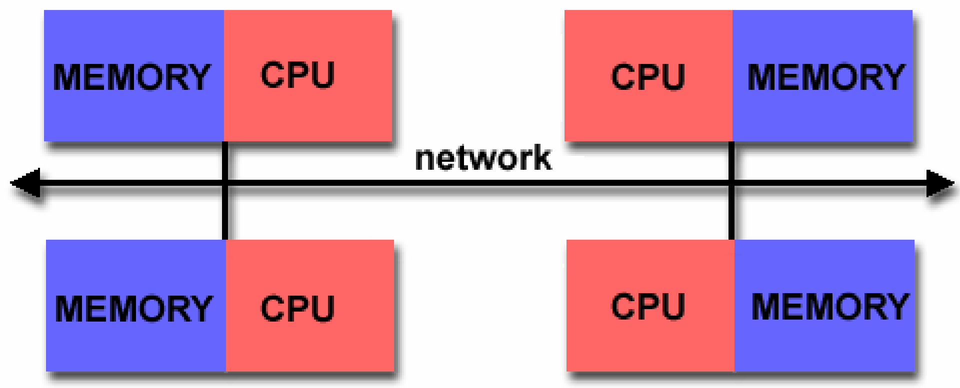


Figure 3.2: Distributed memory system

Hybrid Distributed-Shared Memory

This architecture combines the above two. This type of architecture can be seen in a network of Symmetric Multiprocessing (SMP) machines. Each SMP machine has shared memory, usually the cache areas. To communicate with other SMP machines, the distributed memory architecture is used. For the purpose of this research, the hybrid distributed-shared memory system will not be further discussed.

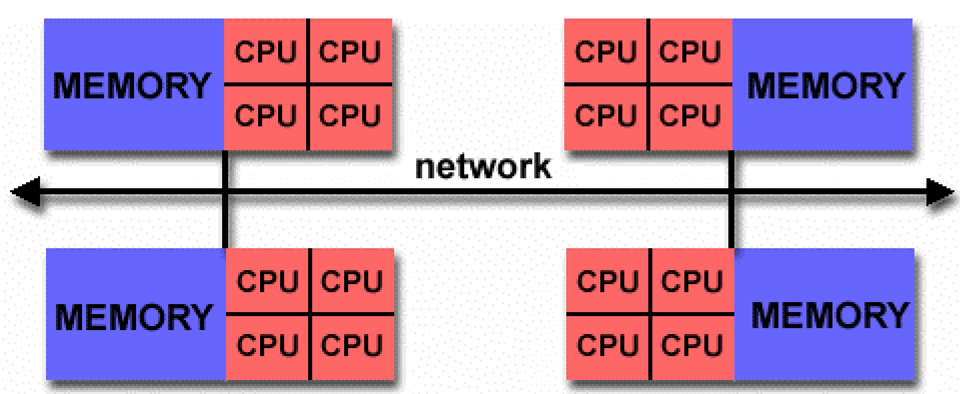


Figure 3.3: Hybrid distributed-shared memory system

3.2 Using Dual PowerPC Core

Since the Virtex-II Pro FPGA has two PowerPC processors, we will want to be able to build a shared memory system. Because the Virtex-II Pro FPGA uses the IBM CoreConnect bus architecture, building a shared memory system for PowerPC405 cores can be done in Platform Studio.

3.2.1 A Simple Shared-Memory System

To demonstrate how a shared memory system can be built on the Virtex-II Pro development system, the following design has been implemented. Since the standalone operating system does not provide any synchronization mechanism and the hardware does not implement cache coherence, an engineer has to handle related issues in the software program. The focus here will be on how to build a shared memory system and how to access shared data.

Shared Memory System Design

Each PowerPC processor has its own block RAM (BRAM). There is also a block of shared memory for the two processors. Shared data is stored in the shared BRAM region. PPC0 monitors the status of switches, and PPC1 controls the status of LEDs. PPC0 receives a user input from the switches, and PPC1 displays the corresponding value on the LEDs. A UART is attached to PPC0 to display current shared data. PPC0 also displays a counter variable changed by PPC1. The following shows the block diagram of the shared memory system.

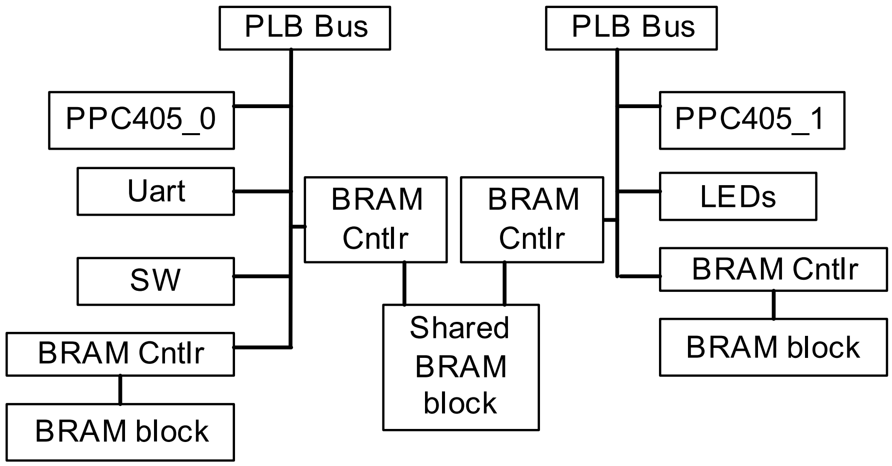


Figure 3.4: Shared memory system block diagram

Shared Memory Implementation

Because the Xilinx EDK does not particularly support dual core designs, there is a need to manually set up the system. The following figures show the bus connections and BRAM port connections:

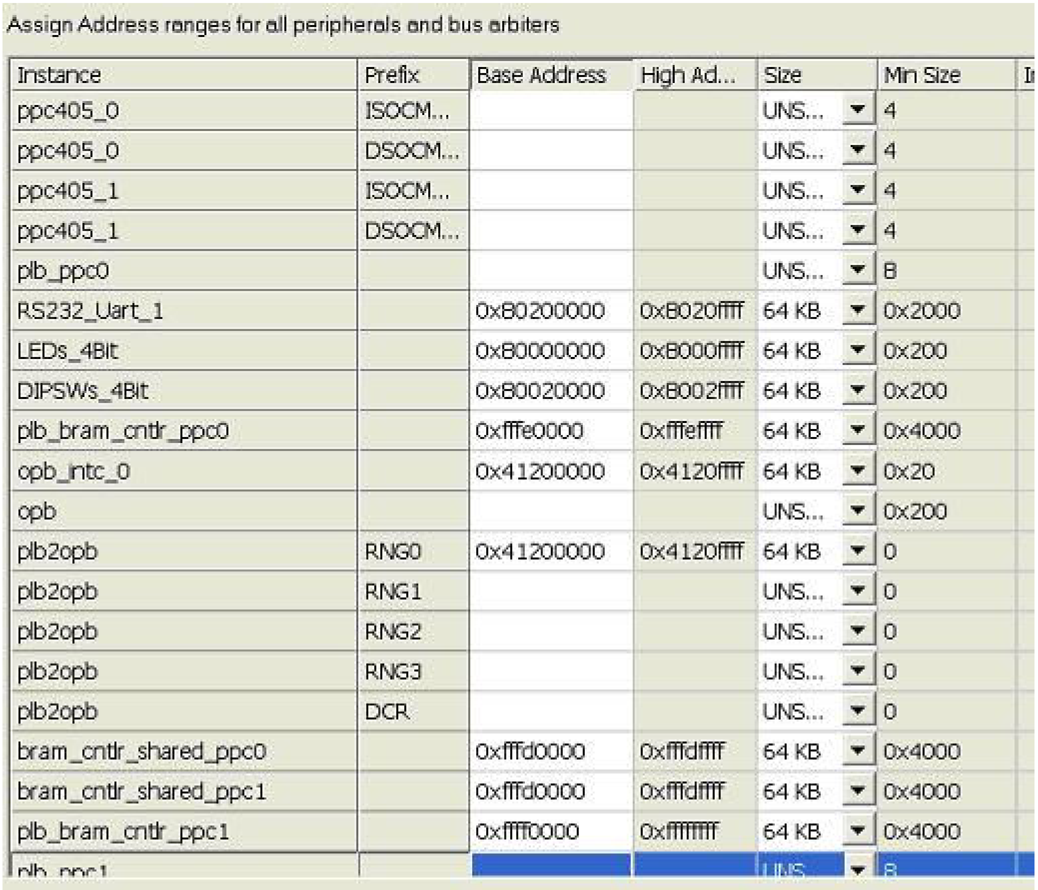
	plb_ppc0	opb	plb_ppc1
ppc405_0 mdcr			
ppc405_0 dplb	M		
ppc405_0 iplb	M		
ppc405_0 dsocm			
ppc405_0 isocm			
ppc405_1 mdcr			
ppc405_1 dplb			M
ppc405_1 iplb			M
ppc405_1 dsocm			
ppc405_1 isocm			
RS232_Uart_1 splb	S		
LEDs_4Bit splb			S
DIPSWs_4Bit splb	S		
plb_bram_cntlr_ppc0 splb	S		
opb_intc_0 sopb		S	
plb2opb sdcr			
plb2opb splb	S		
plb2opb mopb		M	
bram_cntlr_shared_ppc0 splb	S		
bram_cntlr_shared_ppc1 splb			S
plb_bram_cntlr_ppc1 splb			S
plb_ppc0 sdcr			
plb_ppc1 sdcr			

Figure 3.5: Bus connections for the shared memory system

Cntlr Port	BRAM Port	Connector
plb_bram_cntlr_ppc0 porta	bram_ppc0 PORTA	plb_bram_ppc0_porta
bram_cntlr_shared_ppc0 porta	bram_shared PORTA	bram_shared_ppc0_port
bram_cntlr_shared_ppc1 porta	bram_shared PORTB	bram_shared_ppc1_port
plb_bram_cntlr_ppc1 porta	bram_ppc1 PORTA	plb_bram_ppc1_porta

Figure 3.6: BRAM ports connected to the controller ports

As shown in the figure, both UART and switches are connected to PPC0 as PLB slave devices, and LEDs are connected to PPC1 as a PLB slave device. Each processor has two BRAM controllers: one for the local BRAM, and the other for the shared BRAM. The screenshot shown below displays the address map of the shared memory system:


Assign Address ranges for all peripherals and bus arbiters

Instance	Prefix	Base Address	High Ad...	Size	Min Size	I
ppc405_0	ISOCM...			UNS...	4	
ppc405_0	DSOCM...			UNS...	4	
ppc405_1	ISOCM...			UNS...	4	
ppc405_1	DSOCM...			UNS...	4	
plb_ppc0				UNS...	8	
RS232_Uart_1		0x80200000	0x8020ffff	64 KB	0x2000	
LEDs_4Bit		0x80000000	0x8000ffff	64 KB	0x200	
DIPSWs_4Bit		0x80020000	0x8002ffff	64 KB	0x200	
plb_bram_cntlr_ppc0		0xfffe0000	0xfffeffff	64 KB	0x4000	
opb_intc_0		0x41200000	0x4120ffff	64 KB	0x20	
opb				UNS...	0x200	
plb2opb	RNG0	0x41200000	0x4120ffff	64 KB	0	
plb2opb	RNG1			UNS...	0	
plb2opb	RNG2			UNS...	0	
plb2opb	RNG3			UNS...	0	
plb2opb	DCR			UNS...	0	
bram_cntlr_shared_ppc0		0xfffd0000	0xfffdffff	64 KB	0x4000	
bram_cntlr_shared_ppc1		0xfffd0000	0xfffdffff	64 KB	0x4000	
plb_bram_cntlr_ppc1		0xffff0000	0xffffffff	64 KB	0x4000	
plb_ppc1				UNS...	8	

Figure 3.7: Address Map

Internal ports identical to PPC0 for PPC1 need to be added. Clock and reset ports also need to be added for PLB and BRAM controller connected to PPC1. The following figures show the newly added internal port connections. The prefix “ppc_1_” for C405RSTCHIPRESETREQ, C405RSTCORERESETREQ, and C405RSTSYSRESETREQ has been added, because the Xilinx EDK does not allow multiple drivers on these ports. Parameters for each core need to match one another. This is the last step in building a dual-core system.

ppc405_1	CPMC405CLOCK	proc_clk_s	I	CLK
ppc405_1	PLBCLK	sys_clk_s	I	CLK
ppc405_1	C405RSTCHIPRESETREQ	ppc1_C405RSTCHIPRESETREQ	O	
ppc405_1	C405RSTCORERESETREQ	ppc_1C405RSTCORERESETREQ	O	
ppc405_1	C405RSTSYSRESETREQ	ppc1_C405RSTSYSRESETREQ	O	
ppc405_1	RSTC405RESETCHIP	RSTC405RESETCHIP	I	
ppc405_1	RSTC405RESETCORE	RSTC405RESETCORE	I	
ppc405_1	RSTC405RESETSYS	RSTC405RESETSYS	I	
ppc405_1	EICC405EXTINPUTIRQ	EICC405EXTINPUTIRQ	I	INTE...

Figure 3.8: Ports for PPC1

plb_bram_cntlr_ppc1	plb_clk	sys_clk_s	I	CLK
plb_ppc0	SYS_Rst	sys_bus_reset	I	
plb_ppc0	PLB_Clk	sys_clk_s	I	CLK
opb	SYS_Rst	sys_bus_reset	I	
opb	OPB_Clk	sys_clk_s	I	CLK
plb_ppc1	PLB_Clk	sys_clk_s	I	CLK
plb_ppc1	SYS_Rst	sys_bus_reset	I	

Figure 3.9: Added ports for PLB and BRAM controller connected to PPC1

Once the hardware system is built, the system can be tested with a sample application. The software code for PPC0 reads the switch input from a user and stores the value into a shared memory location. PPC1 reads the value from the shared memory location and displays it to LEDs. PPC1 also modifies a counter variable in another shared memory location and lets PPC0 display the updates in the terminal. The source code can be found in Appendix B. The following figure is a screenshot of sample outputs:

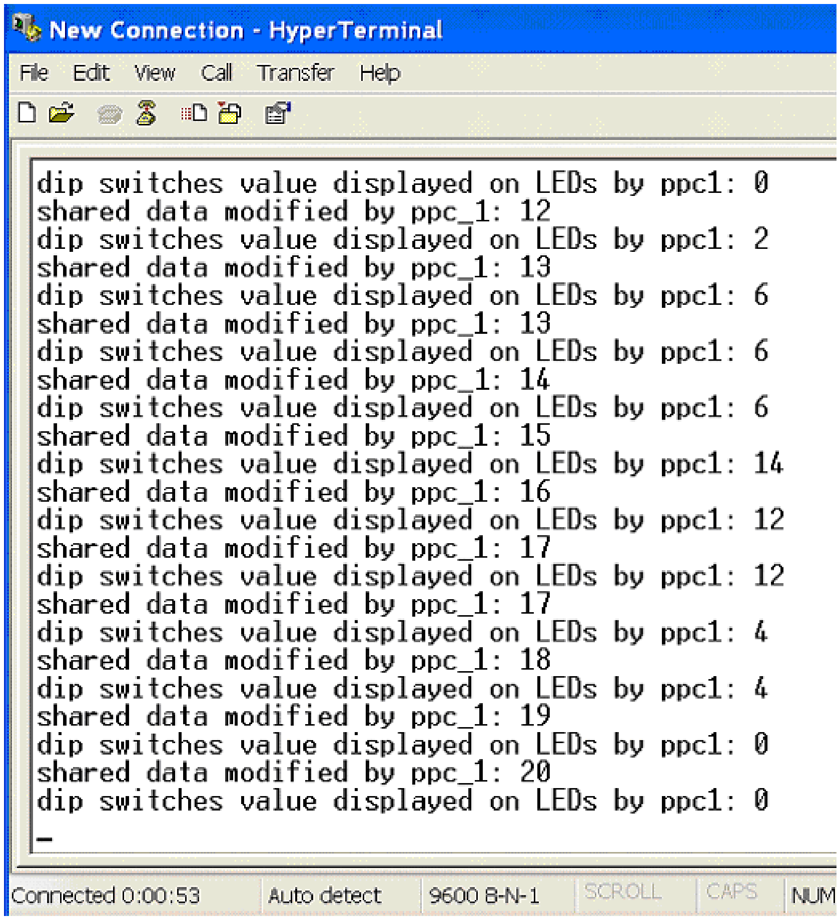


Figure 3.10: Sample Outputs

The above design shows how a shared memory system can be implemented in the Virtex-II Pro development system. There are other ways to design a dual-core system. However, the above experiment shows the fundamental ideas. One important note regarding dual core designs on the PowerPC cores: Because the PowerPC405 cores do not implement cache coherence, it is not feasible to build a symmetric multiprocessing system under Linux. This will be further discussed in the embedded Linux port section. At the time this experiment was done, Xilinx did not offer support of any kind on using both PowerPC cores in the Virtex-II Pro FPGA. The current solution is to build one manually in a standalone operating system as described above.

3.3 The Virtex-II Pro Distributed System

As mentioned before, we can use Ethernet to set up a Virtex-II Pro distributed system. Using the shared memory architecture in an operating system for the PowerPC405 cores is not feasible at this time. As a result, the development platform is not a good choice to build shared memory applications under an embedded operating system. But it is an excellent solution as a distributed system. We can have each board complete tasks using the FPGA and processors co-design feature. The data can be shared among all computing components within the distributed system. There will be a lot of factors to consider before we can design an efficient and fault tolerant distributed system. The topic will not be further discussed here, as it is not within the scope of this research, and worse more hardware to do distributed system experiments is not available.

3.4 Conclusion

This chapter explores the capability of parallel computing on the Virtex-II Pro development system. As a result, we are able to build a parallel computing system in the standalone environment, even though the Xilinx EDK does not specifically offer this feature. Because the onboard PowerPC processors do not implement cache coherence, applications that can be built on the development system are limited. The chapter is also part of the preliminary work of the embedded Linux port, as the possibility of extending the Linux port to the second PowerPC processor will be investigated.

CHAPTER FOUR

EMBEDDED LINUX ON THE VIRTEX-II PRO SYSTEM

4.1 Background

Most embedded applications require the system to handle multiple concurrent processes. An embedded system without an operating system does not offer this capability. For example, if we want a process that gathers information in a field and another process monitoring errors, we will need to combine them into one application. A kernel offers services such as schedulers, synchronization mechanisms, and threads, which give programmers great flexibility in designing complex embedded applications [9]. With the services an operating system provides, we can have many independent processes running in a system. Inter-process communication can be done using shared memory, synchronization, remote procedure calls, or message passing, depending on the operating system design and the programmer's decision. Operating systems on embedded systems are usually designed with the real-time response feature. Most embedded applications have time constraints, so embedded applications within a real-time operating system (RTOS) gives us a more reliable system. Given the advantages of having an operating system, we explore the possibilities of installing one on the Virtex-II Pro development system, preferable a real-time operating system.

4.2 Embedded Linux on the Virtex-II Pro Development System

The Xilinx EDK provides several options: Xilkernel, VxWorks, and MontaVista Linux. Xilkernel is a robust and modular kernel that is highly integrated with the Platform Studio framework [3]. It comes with a licensed EDK. It provides a Portable

Operating System for Unix (POSIX) interface to the kernel, and supports core features for a real-time embedded kernel such as schedulers, synchronization services, and inter-process communication services. The main disadvantage of Xilkernel is that it is specific to certain Xilinx FPGAs and can only be invoked in the Platform Studio. It does not provide the flexibility as a general purpose embedded operating system. MontaVista Linux and VxWorks both require licensed development packages. These commercial products provide ready-to-use cross compiling tools, and custom Unix-based operating systems. As researchers, we will want to be able to use open sources for obvious reasons.

Linux has been a popular choice in the embedded system world for several reasons:

- Open Source and Royalty Free: Anyone can obtain the source and make modifications to fit specific design requirements.
- Small in Size: The kernel image of an embedded Linux is usually about 2MB to 8MB.
- Stable and Well-Supported Operating System: Linux has over ten years of development history, and has been used in many hi-tech products. It supports most processor architectures and devices.

In order to find out whether embedded Linux is suitable for the Virtex-II Pro development system, more factors need to be considered and examined.

As we all know, Linux supports most desktop computers. However, every device is unique to a particular embedded system. Therefore, an embedded Linux system normally requires custom drivers support. Montavista has partnered with Xilinx, and has developed commercial embedded Linux kits for some Xilinx platforms. Although the complete kit requires licensing, the device drivers are under General Public License (GPL). Even though these drivers are generic and might not work on different Xilinx platforms, we can modify the source code to adapt devices on the Virtex-II Pro development system. This advantage enables us to bring up the peripherals on the board without spending time developing drivers. We can also add the Board Support Package (BSP) for the Virtex-II Pro development system to the Linux kernel, which contains development platform specific device drivers

The Virtex-II Pro FPGA allows custom processor peripherals. We can easily access these custom peripherals in the standalone operating system. However, with the loadable module feature of Linux, we can decide whether to build the drivers in the kernel or to load these modules dynamically [9]. This feature allows us to load a device at run-time, which we will need in some embedded systems. For example, we might want the system to reload a backup device when the current device fails at any time. Embedded system tasks are critical, and this feature brings us a more stable system.

Embedded Linux is configured to fit small systems. It can be viewed as a stripped version of standard Linux. In order to handle critical tasks, a real-time scheduler and preemptive kernel are added to embedded Linux. We can write embedded applications using built-in threads packages once we have the embedded Linux running. Based on all the above factors, embedded Linux is ideal for the Virtex-II Pro development system.

4.3 Requirements for the Linux Port

Port of Linux to an embedded system is not as simple as the desktop installation. Embedded systems have limited resources, and usually do not have enough space to run a compiler. Even if we have sufficient disk space to hold a compiler, we will have poor performance out of the slow compiler. For this reason, most people choose to compile source code in a host machine, and then download binaries to the target system. If the host computer is not a PowerPC machine, a cross compiler needs to be built. Because embedded platforms are hardware specific, we will need a suitable kernel. At the time the research is conducted, several companies have developed Virtex-II Pro FPGA compatible kernels. Among all, only MontaVista has released an open source kernel (2.4.x) with the support. The newest kernel at this time is 2.6, but the support for this specific FPGA is removed from 2.5 and above. Therefore, 2.4.x seems to be the only open source that can be used for the port. Other than the kernel and compiler, we will also need a root filesystem and BSP. The following lists the requirements for the Linux port:

Linux Kernel

MontaVista offers an open Linux kernel 2.4.x that supports ML300 boards. The ML300 boards use the Virtex-II Pro FPGA series same as our development system. Even though the kernel is specific to the ML300 board, its support for the Virtex-II Pro FPGAs can be used on our development system.

Board Support Package (BSP)

The BSP allows us to access devices specific to this platform. Because MontaVista supports Linux, we can use their BSP that comes with the Xilinx EDK.

Crosstool

The target build is a PowerPC processor. Since the development host machine available is not a PowerPC machine, a cross compiler is needed to compile source code to PowerPC machine code. It is a cumbersome process to build a cross compiler. We will need to build GNU Binutils, which contains a linker, assembler, and other binary tools. We then install Glibc, the GNU C library. The last step is to build the Gcc compiler. It takes a lot of time and patience to have everything correctly configured and properly installed. Fortunately, Dan Kegel [10] has developed Crosstool, which provides a set of useful shell scripts that does all the work for us. The scripts will download tools for you if they are not found in the host machine. It takes about two hours on a Pentium 4 machine to build a cross compiler using Crosstool.

Compatible GCC and Glibc

Crosstool needs to be built with certain Gcc and Glibc combinations. If the Gcc and Glibc combination on the host machine is not supported, they need to be updated.

Root Filesystem

We will need a root filesystem in Linux to do meaningful work. The filesystem contains startup files, utilities, and file systems. BusyBox combines tiny versions of many common Unix utilities into a single small executable [11], and will be used to create the root filesystem.

Base System

The base system contains hardware specifications for the embedded system.

System Advanced Configuration Environment (ACE) File

System ACE files provide an easy way to program the FPGA. The system ACE controller allows us to boot from a CompactFlash.

CompactFlash Card

Both the kernel and the root filesystem will be stored in the CompactFlash card.

4.4 Porting Linux

Thanks to the pioneer work done at BYU [12] and by Wolfgang Klingauf [13], good documentation on porting Linux for some Xilinx platforms can be found across the Internet. This section shows the steps for the Linux port.

Hardware and Software Specifications

Hardware:

- Sony VAIO with a Pentium 4 processor for Debian
- Pentium III 1.0 GHz for Windows XP
- Xilinx Virtex-II Pro Development System
- Compact Flash Memory Card (512MB) and Reader
- Kingston 256MB Memory
- Serial Cable for Standard Input/Output
- USB Cable

Software:

- Xilinx EDK7.1.2 SP2
- RedHat 9 Shrike & Debian 3.1 Sarge (either one can be used)
- Crosstool 0.38
- BusyBox 1.1.0
- TeraTerm Pro 3.1

Building A Cross Compiler

A cross compiler is built on a Pentium 4 machine using Crosstool.

Obtaining the Linux Source

The kernel source for this research is downloaded from MontaVista and has a version number of 2.4.25.

Configuring the Kernel

For this Linux port, the kernel configuration contains the following settings:

Code Maturity Level Options

Prompt for development and/or incomplete drivers

Loadable Module Support

Enable loadable module support

Platform Support

40x Processor Type

Xilinx-ML300 Machine Type

Math emulation

<UART0> TTYS0 device and default console

UART0

General Setup

Networking support

Sysctl support

System V IPC

Default bootloader kernel arguments

"console=ttyS0,9600 root=/dev/xsysace/disc0/part3 rw"

Memory Technology Devices (MTD)

Memory Technology Device (MTD) Support

MTD partitioning support

RedBoot partition table parsing

Direct char device access to MTD devices

Caching block device access to MTD devices

RAM/ROM flash chip device drivers

Detect flash chips by Common Flash Interface (CFI) probe

Support for AMD/Fujitsu flash chips

Block Devices

Xilinx on-chip System ACE

Loopback device support

Network block device support

RAM disk support

(4096) Default RAM disk size

Initial RAM disk (initrd) support

Networking Options

Socket Filtering

Unix domain sockets

TCP/IP networking

IP: multicasting

IP: kernel level autoconfiguration

IP: DHCP support

IP: TCP syncookie support (disabled per default)

Network Device Support

Network device support

Ethernet (10 or 100Mbit)

Xilinx on-chip ethernet

Character devices

Standard/generic (8250/16550 and compatible UARTs) serial support

Support for console on serial port

Unix98 PTY support

File systems

Journaling Flash File System v2 (JFFS2) support

JFFS2 debugging verbosity (0=quiet, 2=noisy)

Virtual memory file system support (former shm fs)

/proc file system support

/dev file system support (EXPERIMENTAL)

Automatically mount at boot

/dev/pts file system for Unix98 PTY

Second extended fs support

Native Language Support

Default NLS Option: "iso8859-1"

Kernel hacking

Kernel debugging

Include BDI-2000 user context switcher

Add any additional compile options

Additional compile arguments: "-g -ggdb"

(0) Kernel messages buffer length shift (0=default)

The above kernel services are chosen to fit this particular system. This configuration contains a minimal working kernel. The kernel can always be re-compiled with more options, if more services are required.

Building a Base System

A base system that has the following components has been built using the Xilinx Platform Studio.

- PowerPC at 300MHz
- RS232_Uart_1: Peripheral OPB UART 16550, Configure as UART 16550, and use Interrupt
- Ethernet_MAC: Peripheral OPB ETHERNET, No DMA, and use Interrupt
- SysACE_CompactFlash: Peripheral OPB SYSACE, and use Interrupt
- DDR_256MB_32MX64_rank1_row13_col10_cl2_5: Peripheral PLB DDR, and use Interrupt
- PLB BRAM IF CNTLR: Memory Size 128KB

After downloading a bitstream to an FPGA, the processor comes out of the reset state and starts executing. If no application is initialized, the processor might execute random code and get in some state that it cannot be brought out of with a soft reset [3]. XPS provides a bootloop program that keeps the processor in a defined state until an application is ready to run. We will need a bootloop to keep the PowerPC processor defined prior the kernel startup. To use the bootloop, set ppc405_0 bootloop to initialize BRAMs, and create a downloadable bitstream for the system.

Generating BSP

The following parameters are set in the Software Platform Settings window.

- linux_mvl31 version 1.0.1a as the operating system for ppc405_0.
- MEM_SIZE: 0x10000000
- PLB_CLOCK_FREQUENCY: 100000000
- TARGET_DIR: ‘C:/XUPV2P/temp’
- connected_peripherals: RS232_Uart_1, Ethernet_MAC, SysACE_CompactFlash, opb_intc_0

BSPs are generated by Libgen. Once Libgen is done, BSPs can be found in the target directory.

Compiling the Kernel

Once the kernel is compiled with no errors, the kernel image zImage.elf is stored in the arch/ppc/boot/images directory.

Creating an ACE file

The genace.tcl file contains several board configurations. The parameters for the Virtex-II Pro development system need to be manually added. ACE files are generated by issuing proper commands in the Xilinx Cygwin shell.

Creating a Root File System

There are different ways to locate a root filesystem. The CompactFlash card can be partitioned to store the root filesystem. This way, the CompactFlash card can simply be inserted and have the hardware do the rest.

Wolfgang Klingauf provides a very useful script to make a root filesystem using BusyBox. BusyBox combines tiny versions of many common Unix utilities into a single small executable. A root filesystem is made with proper options using Klingauf's scripts for this Linux port.

Moving Everything to the CompactFlash

The CompactFlash needs to be divided into three partitions. The following shows the partition table:

Partition 1: FAT16 (6)

Partition 2: Linux Swap Partition (82)

Partition 3: Linux (83)

Partition 3 is where the root filesystem is stored. The FAT partition needs a particular format under Windows [4]. The ACE file is copied to this partition.

Running Linux

The Linux port is now completed. The following is the log of Linux running on the Virtex-II Pro development system:

```
loaded at:     00400000 004A01E4
board data at: 0049D13C 0049D154
relocated to:  00405634 0040564C
zimage at:     00405B39 0049C3B3
avail ram:     004A1000 10000000

Linux/PPC load: console=ttyS0,9600 root=/dev/xsysace/disc0/part3 rw
Uncompressing Linux...done.
Now booting the kernel
Linux version 2.4.25 (root@jabu-01) (gcc version 3.4.1) #45 Thu Feb 23
10:50:40
PST 2006
Xilinx Virtex-II Pro port (C) 2002 MontaVista Software, Inc.
(source@mvista.com)
On node 0 totalpages: 65536
zone(0): 65536 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: console=ttyS0,9600 root=/dev/xsysace/disc0/part3
rw
Xilinx INTC #0 at 0x41200000 mapped to 0xFDFFE000
Calibrating delay loop... 299.82 BogoMIPS
Memory: 257620k available (1068k kernel code, 308k data, 60k init, 0k
highmem)
Dentry cache hash table entries: 32768 (order: 6, 262144 bytes)
```

```
Inode cache hash table entries: 16384 (order: 5, 131072 bytes)
Mount cache hash table entries: 512 (order: 0, 4096 bytes)
Buffer cache hash table entries: 16384 (order: 4, 65536 bytes)
Page-cache hash table entries: 65536 (order: 6, 262144 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Initializing RT netlink socket
Starting kswapd
devfs: v1.12c (20020818) Richard Gooch (rgooch@atnf.csiro.au)
devfs: boot_options: 0x1
JFFS2 version 2.2. (C) 2001-2003 Red Hat, Inc.
pty: 256 Unix98 ptys configured
Serial driver version 5.05c (2001-07-08) with no serial options enabled
ttyS00 at 0xfdfff003 (irq = 29) is a 16550A
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
loop: loaded (max 8 devices)
Partition check:
 xsysacea: p1 p2 p3
System ACE at 0x41800000 mapped to 0xD1000000, irq=30, 500976KB
eth0: using fifo mode.
eth0: No PHY detected.  Assuming a PHY at address 0.
eth0: Xilinx EMAC #0 at 0x40C00000 mapped to 0xD1013000, irq=31
eth0: id 2.0h; block id 7, type 1
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP, IGMP
IP: routing cache hash table of 2048 buckets, 16Kbytes
TCP: Hash tables configured (established 16384 bind 32768)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
EXT2-fs warning: mounting unchecked fs, running e2fsck is recommended
VFS: Mounted root (ext2 filesystem).
Mounted devfs on /dev
Freeing unused kernel memory: 60k init

Welcome to ML300 powerpc linux 2.4.21, E.I.S. edition

Starting system...
mounting /proc: done.
Mounting '/' read-write: done.
brining up loopback interface: done.
Mounting /tmp: done.
Starting syslogd: done.
Starting klogd: done.
Starting inetd: done.
System started.
ML300 powerpc linux 2.4.21-pre7 E.I.S. edition
(none) login: root


Welcome to the ML300, EIS edition

Be careful, it's blue.



BusyBox v1.1.0 (2006.01.17-20:03+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.
```

```
# cd /
# ls
a.out         dev           lib           modules0201  root          usr
bin           etc           linuxrc       opt          sbin          var
boot          home          mnt           proc         tmp
#
```

4.5 Comments on Symmetric Multiprocessing

Once Linux is ported to the board, the next thing we want to do is to enable Symmetric Multiprocessing (SMP). SMP allows multiple processors to complete their own tasks simultaneously. It uses one operating system and shares common resources among processors. Unfortunately, due to the hardware architecture of PowerPC405 cores, SMP is not feasible. The PowerPC405 cores do not define the size, structure, replacement algorithm, or mechanism used for maintaining cache coherency [4], which is what we need in a multiprocessing environment. It is possible to implement cache coherence in software. However, it is an expensive process, as the software program will have to track all memory accesses in all processors. This is an in-deterministic process and will sacrifice timeliness for hard real-time applications. Cache coherence is rarely done in software. As for the PowerPC405 cores, no one has enabled SMP at this point. One way to enable both cores is to have a copy of operating system running in each processor. Mind, a Belgian company, has managed to do this in Linux for the Virtex-II Pro FPGA. This approach is not studied any further in this research, because it is not considered as a shared memory multiprocessor system.

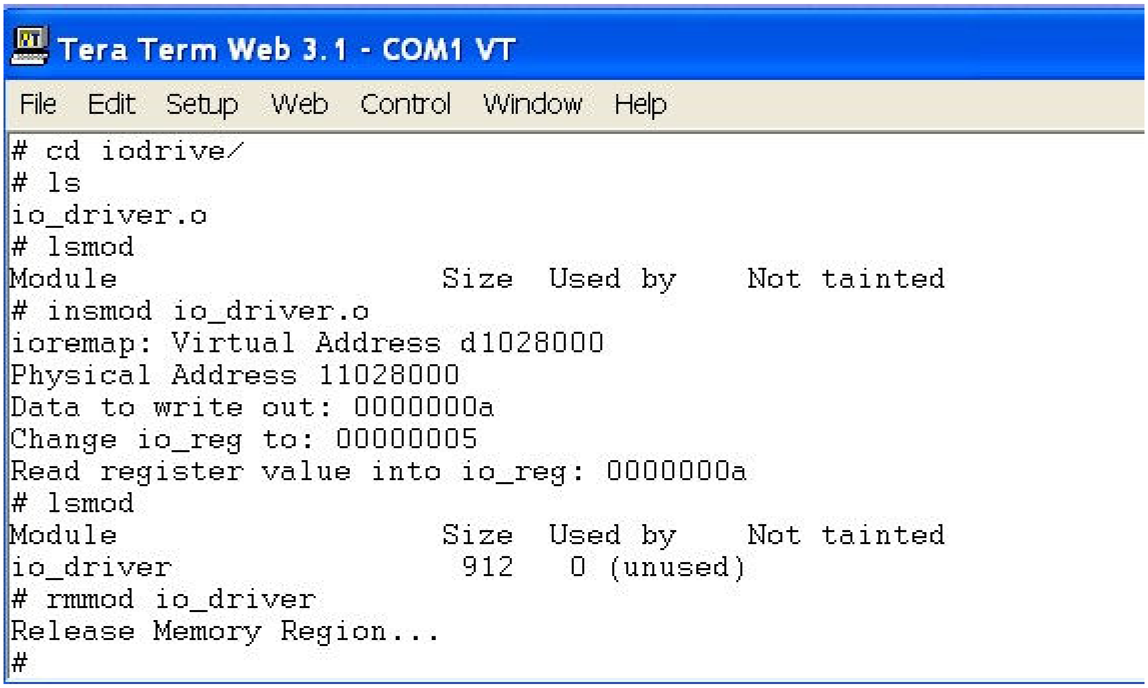
4.6 Linux Device Drivers

Once we have an operating system running on the board, we will want to run applications in the OS. Most embedded applications require interactions with hardware devices. Because MontaVista only offers a limited number of generic drivers for the

Virtex-II Pro platforms, we will need to write custom device drivers for our designs. Unix classifies devices into three types [14]: character module, block module, and network module. There are two ways to build a Linux device driver namely: (i) build in as part of the kernel source, and (ii) create a loadable module. The former requires kernel re-compilation each time you modify the driver source. The latter provides more flexibility as you load the module while the kernel is running. This section provides two sample driver designs for research completeness. A character device can be accessed as a file, which is ideal for implementing device IOs. The first module does the work in kernel, without any user application support. The second module gets loaded into the kernel, and a user application interfaces the module to perform desired IOs.

4.6.1 Loadable Kernel Module

This module reads and writes a register in the FPGA. A custom IP with one software accessible register is created. The user logic VHDL code is generated by EDK, which performs read and write operations. In the kernel module code, a write is issued and then a read. After the data is written out, the data value is changed to some other number. This ensures the data read in later is correct. The code is compiled using the Makefile. The output binary (.o) is loaded into the kernel with the command "insmod io_driver.o". The following is a screenshot of the output:




```
# cd iodrive/
# ls
io_driver.o
# lsmod
Module                  Size  Used by    Not tainted
# insmod io_driver.o
ioremap: Virtual Address d1028000
Physical Address 11028000
Data to write out: 0000000a
Change io_reg to: 00000005
Read register value into io_reg: 0000000a
# lsmod
Module                  Size  Used by    Not tainted
io_driver                912   0 (unused)
# rmmod io_driver
Release Memory Region...
#
```

Figure 4.1: Loadable module output

The source code for the driver can be found in Appendix C.

4.6.2 Loadable Kernel Module with User Programs

The above module works well for the system, however, we will eventually want to access hardware from a user program. We can make drivers to support user programs. One way to enter the kernel mode from user mode in Linux is through system calls. Two kernel routines copy_from_user() and copy_to_user() are needed for data transfer from/to the user space. This design performs the same function as above, except that we can now modify the value of the register in a user application.

In this design, we can access the hardware same as the way we access a file in a user program, meaning we can use open(), close(), read(), and write() function calls. This module works as a loadable module and as a built-in module.

After the module is loaded successfully, a special file that has the same name as the file in the user program needs to be created. In this case, such a file is created with the command "mknod /dev/xgpio c 10 23", where "c" indicates it is a character device, 10 is the major number for misc devices, and 23 is the minor number assigned to the device. The permission of the file also needs to be changed: "chmod 666 /dev/xgpio". The program can be tested by executing the binary. The following is a screenshot of the output:

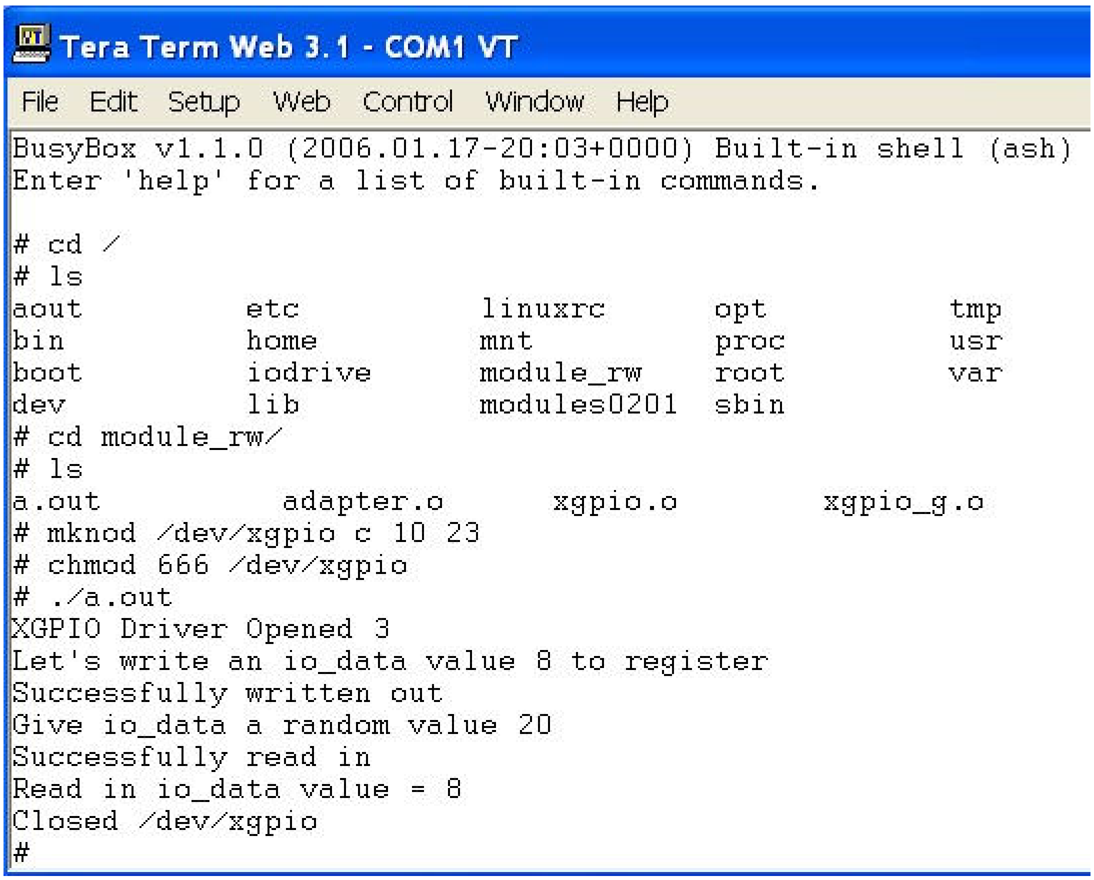


Figure 4.2: Enhanced loadable module output

The source code for the driver can be found in Appendix D.

4.7 Conclusion

An embedded system with an operating system offers more flexibility to embedded applications. An embedded Linux operating system has been ported to one PowerPC processor on the Virtex-II Pro development system successfully. The Xilinx EDK allows us to add and remove built-in and custom IP cores. We can now modify devices within the operating system by loading/un-loading proper device drivers. This feature is particularly useful when we want to load a device at run time, which is required for some embedded applications. Although Symmetric Multiprocessing can not be achieved on this development platform, the research effort provides the insight, which suggests an alternative approach or a new hardware development.

CHAPTER FIVE

HARDWARE SOFTWARE PARTITIONING

5.1 Background

The FPGA and embedded processors co-existence feature of the Virtex-II Pro FPGA enables us to design embedded applications that take the advantage of hardware software partitioning. The problem of hardware software partitioning has seen over a decade of activity and some of the notable work includes that presented in [15, 16]. This problem has been made easy by the development of logic systems with embedded cores. It is not our desire to design algorithms to exploit these benefits. In simple terms hardware software partition involves being able to identify an application's components that can be better performed in hardware and those that can be better performed in software, and dividing them to compute in their respective units (Hardware or Software). In this study no algorithm has been used to identify the respective tasks since efforts are focused on presenting a diverse design environment that offers computational efficiency along with the flexibility of a real-time kernel. A simple digital Finite Impulse Response (FIR) filter has been used to demonstrate the hardware software co-design as well as the computational efficiency gained. Partitioning of this FIR filter is done manually and is not as cumbersome to manage.

As mentioned already, we can move tasks that can be done efficiently in the hardware (FPGA) to obtain better performance of an application. An example of such would be to move the multiply-and-accumulate part of the FIR filter to the FPGA for fast computation, as this takes much longer in the PowerPC processors. The memory management and switching of coefficients will be left as software tasks. This chapter shows how hardware software partitioning can be done in the Virtex-II Pro development system, as well as the performance comparison between a case with and one without hardware software partitioning systems and its significance. An audio filtering application will be designed, and the finite impulse response filter is chosen for its compute intensive feature.

5.1.1 Digital Signal Processing

Digital signal processing provides probably some of the most compute intensive applications in engineering. The Virtex-II Pro development system is not intended for digital signal processing, but it has such powerful computational capabilities and digital signal processing applications have computational needs that can be met using this system. In order to experiment hardware and software partitioning, real-time processing and other related aspects of real-time systems design; adaptive filtering has been considered. Finite impulse response filters have been used to demonstrate these important embedded systems design issues on the Virtex-II Pro development system. In the following a brief on two types of digital filters is presented.

5.2 FIR Filter Design

Finite impulse response (FIR) filters are one of two primary types of digital filters used in Digital Signal Processing (DSP) applications the other type being infinite impulse response (IIR). IIR filters use feedback and each type of filter has advantages and disadvantages [17, 18]. Overall, though, the advantages of FIR filters outweigh the disadvantages as a result they are used much more than IIR filters. The FIR filters offer the following advantages:

- They can easily be designed to have a linear phase, in other words they delay the input signal, but do not distort its phase.
- They are simple to implement and on most DSP microprocessors the computations can be done by looping a single instruction.
- They are easy to manipulate allowing decimation (reducing the sampling rate), interpolation (increasing the sampling rate), or both.
- Whether decimating or interpolating, the use of FIR filters allows some of the calculations to be omitted, thus providing an important computational efficiency. Coefficient symmetry also saves memory space.
- They have desirable numeric properties. In practice, all DSP filters must be implemented using "finite-precision" arithmetic, that is, a limited number of bits. The use of finite-precision arithmetic in IIR filters can cause significant problems due to the use of feedback, but FIR filters have no feedback, so they can usually be implemented using fewer bits, and the designer has fewer practical problems to solve related to non-ideal arithmetic.
- They can be implemented using fractional arithmetic. Unlike IIR filters, it is always possible to implement a FIR filter using coefficients with magnitude of less than 1.0. (The overall gain of the FIR filter can be adjusted at its output, if desired).

Despite the highlighted advantages, FIR filters sometimes have the disadvantage that they require more memory and/or calculation to achieve a given filter response characteristic. In addition, certain responses are not practical to implement with FIR filters. Some of the most important FIR filter parameters and characteristics include:

- *Impulse Response* – this is a set of FIR coefficients. If an impulse input to an FIR filter with the impulse consisting of a “1” sample followed by many “0” samples, the output of the filter will be the set of coefficients, as the “1” sample moves past each coefficient in turn to form the output.
- *Tap* - A FIR "tap" is simply a coefficient/delay pair. The number of FIR taps is an indication of (i) the amount of memory required to implement the filter, (ii) the number of calculations required, and (iii) the amount of "filtering" that should be done; in effect, more taps mean more stop-band attenuation, less ripple, narrower filters)
- *Multiply-Accumulate (MAC)* - In a FIR context, a MAC is the operation of multiplying a coefficient by the corresponding delayed data sample and accumulating the result. FIR filters usually require one MAC per tap.
- *Transition Band* - The band of frequencies between pass-band and stop-band edges. The narrower the transition band, the more taps are required to implement the filter.
- *Delay Line* - The set of memory elements that implement the Z^{-1} delay elements of the FIR calculation.

In this study there is no concentration on digital filter design expertise and the background presented is deemed sufficient to enable for the software, hardware and hardware and software implementations of the development system. Adaptive filtering is also considered to enable for an evaluation of real-time responsiveness of the system.

FIR filters are widely used in digital signal processing. The linear phase property of FIR filters gives a fixed amount of delay and provides no delay distortion. The symmetry of coefficients saves memory space for storage. The design of an FIR filter involves the following steps:

- Filter specification
- Coefficients calculation
- Implementation

In this study, a basic FIR filter algorithm is used. The filter stores an input, calculates the output, and shifts the delay line. The output is described by:

$$\mathbf{y(n)=h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + ... h(N-1)x(n-N-1)}$$

Where h(i) represents the coefficients, and x(n), x(n-1)……x(n-N-1) being the inputs. Since this experiment focuses on the hardware software partitioning capability of the system, the design of FIR filters is done using available tools. The source code is derived from dspguru.com [15], and modifications have been made to meet design requirements. The coefficients are generated by the online FIR Filter Designer Pro software, which is written by Vijaya Chandran Ramasami [16]. The Hamming Window method is used for all FIR filter designs in this experiment.

The National Semiconductor LM4550 AC97 audio codec on the Virtex-II Pro development system is fully supported by the Xilinx EDK. It is paired with a stereo power amplifier made by Texas Instrument. The LM4550 uses 18-bit Sigma-Delta A/Ds and D/As, providing 90 dB of dynamic range. The implementation on this board allows for full-duplex stereo A/D and D/A with one stereo input and two mono inputs, each of which has separate gain, attenuation, and mute control. The mono inputs are a microphone input with 2.2V bias and a beep tone input from the FPGA [3]. In this experiment, the microphone input will be read by a PowerPC processor, the voice data will be filtered and stored in the SDRAM, and the resulting signals will then be output to speakers. The FIR filter is used and all three types of filters, namely low-pass filter, high-pass filter, and band-pass filter, will be examined though some might not have any effect on the signals of interest. In this experiment we have recorded a human voice (speech signal) while there is “music” in the background and we aim to filter the human voice and play back just the music. A high pass filter is therefore ideal in this case especially given that the frequency range of the human voice would be at low frequencies and those of music a bit higher. A more complex problem would involve separating the mixed signals into the speech and the music signals. The blind source separation of real world signals is examined in [19].

A control experiment involves recording the mixed signals, performing no filtering and then playing back. This enables us to determine that the filters are at least functioning as expected.

5.2.1 FIR Filter Specifications

The following table shows the specifications for all three types of FIR filters used in the experiment. The values displayed in Table 5.1 are based primarily on the fact that it is a human voice that will be processed and these frequency ranges are within the suggested range. Also the sampling rate has been influenced by the fact that we wanted to keep values within the range of frequencies the human ear can capture.

Table 5.1: FIR Filter specifications

	Low-pass	High-pass	Band-pass
Pass-band Frequency1	10,000Hz	10,000Hz	500Hz
Stop-band Frequency1	12,000Hz	9,500Hz	0
Pass-band Frequency2			19,000Hz
Stop-band Frequency2			21,000Hz
Sampling Frequency	44,100Hz	44,100Hz	44,100Hz
Pass-band Ripple	0.1	0.1	0.1
Stop-band Attenuation	30dB	30dB	40dB

Design restrictions:

- Unsigned integers are used for input samples, coefficients, and outputs. Because PowerPC processors do not have a floating-point unit, and any floating number computation is done using software emulation, use of floating numbers in the system adds significant delay and is not ideal for embedded applications.
- The filters experimented with are 30-tap filters and according to [20] FIR filters commonly require anything from 10 to 256 taps.

5.2.2 FIR Filter Plots

The following shows sample input/output plots for a low-pass filter and a high-pass filter based on the designs using software. For these plots, floating points are used for output accuracy.

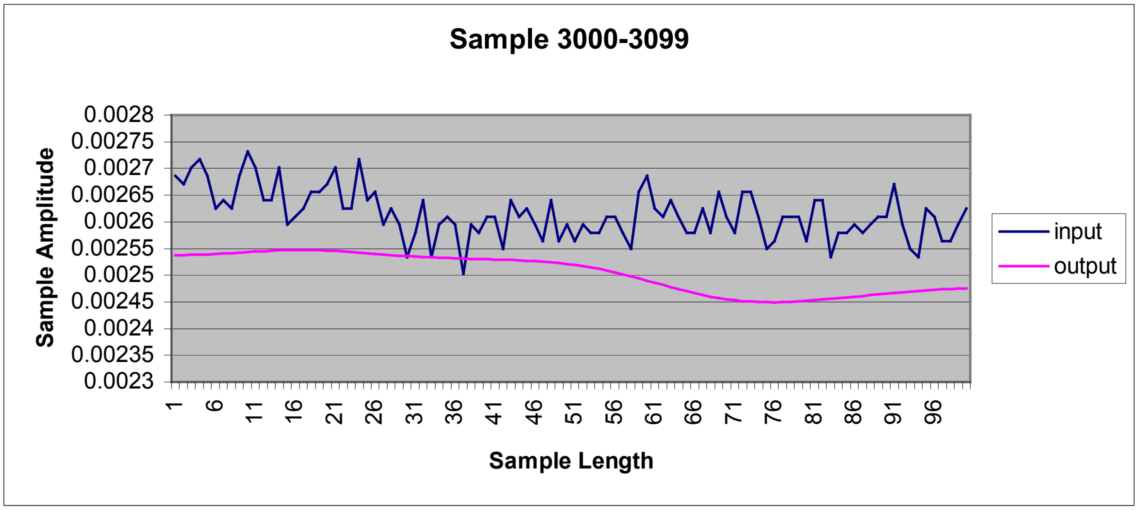


Figure 5.1: Sample plot #1 of the low-pass filter

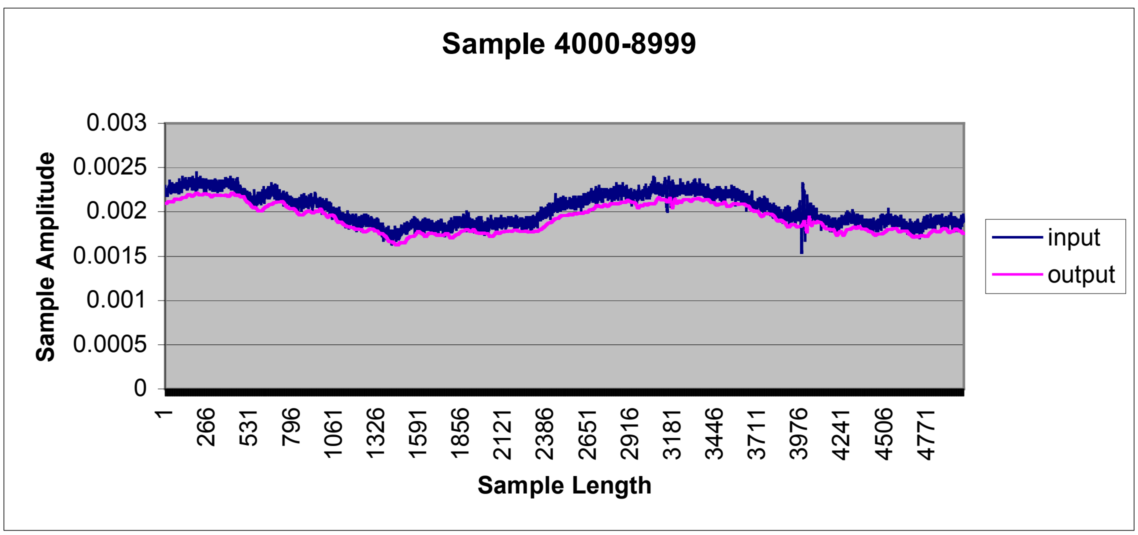


Figure 5.2: Sample plot #2 of the low-pass filter

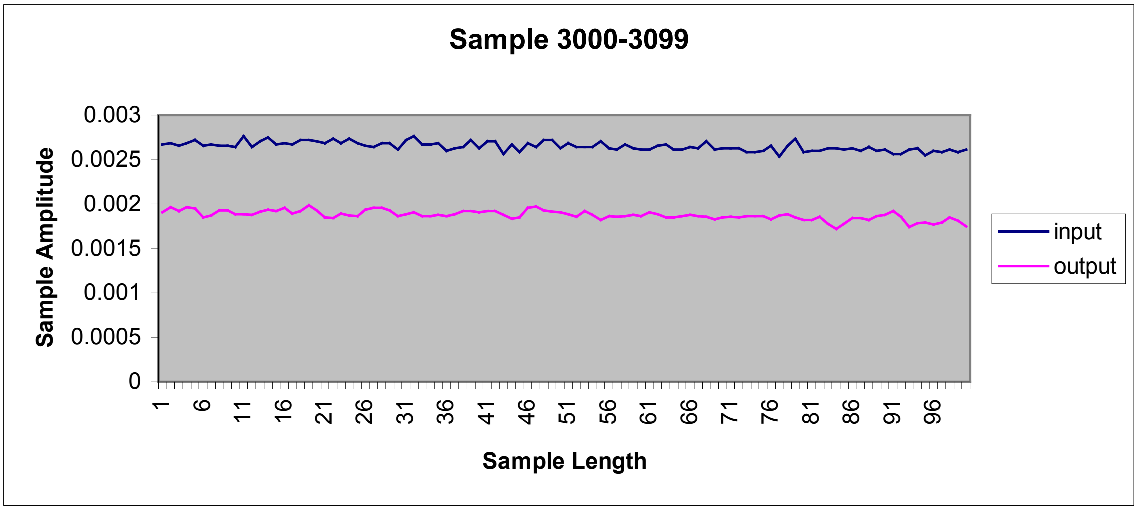


Figure 5.3: Sample plot #1 of the high-pass filter

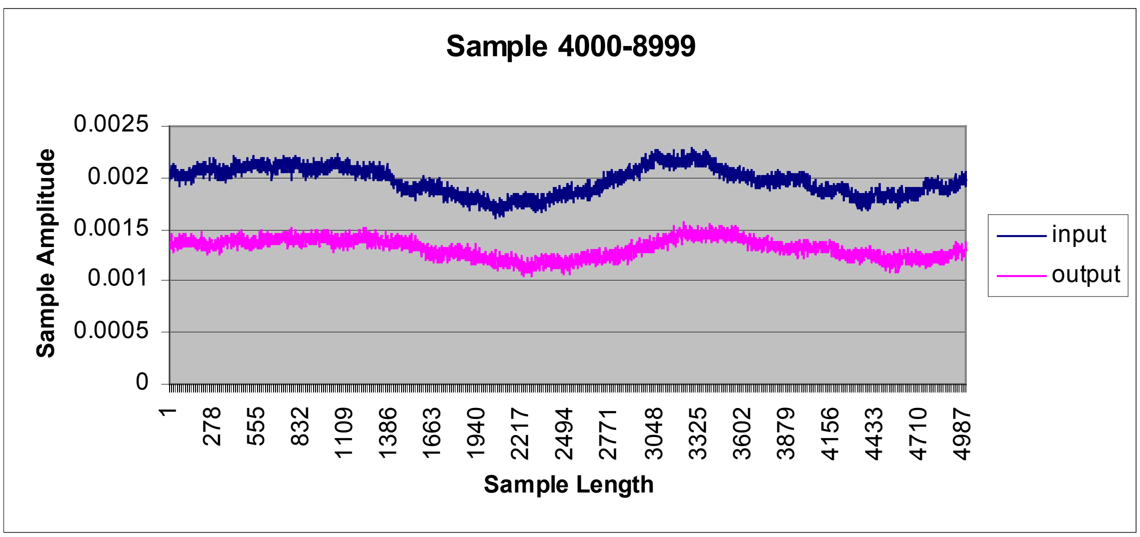


Figure 5.4: Sample plot #2 of the high-pass filter

Though the accuracy of these filters is not the primary focus of this design, it can be seen from the traces that the output signals are less noisy, further more the audio signals played back do confirm that the lower frequencies are filtered out. Of significant importance are the performance gains arrived at through the partitioning of hardware and software tasks of the filter reported in the subsequent subsections.

5.2.3 Software Based Audio Filtering Design

The following block diagram shows the system flow:

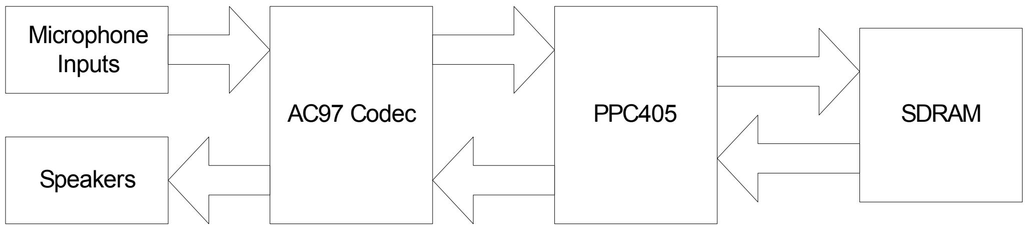


Figure 5.5: System flow diagram for the software based audio filtering design

The microphone inputs are converted to digital signals through the AC97 codec. The PowerPC processor does the filtering work, and sends the outputs to the AC97 codec for playback through speakers. Both inputs and outputs are stored in the SDRAM, so that UART can retrieve data.

The source code is a modified and simplified version of the code obtained from Dspguru.com. The audio data is filtered in the processor before it is output to the speakers. A sample code of the audio filtering design can be found in Appendix E.

5.2.4 Hardware Software Partitioning Audio Filtering Design

The audio input and filtered output are stored in SDRAM. The FIR filter is created as a custom OPB core that attaches to the PowerPC processor. The multiply-and-accumulate operations are now done in the FPGA. The PowerPC processor reads samples from the audio input, sends them to the FIR filter core, reads the filtered outputs from the FPGA, stores them into SDRAM, and plays the results to the speakers. The system flow diagram is as follows:

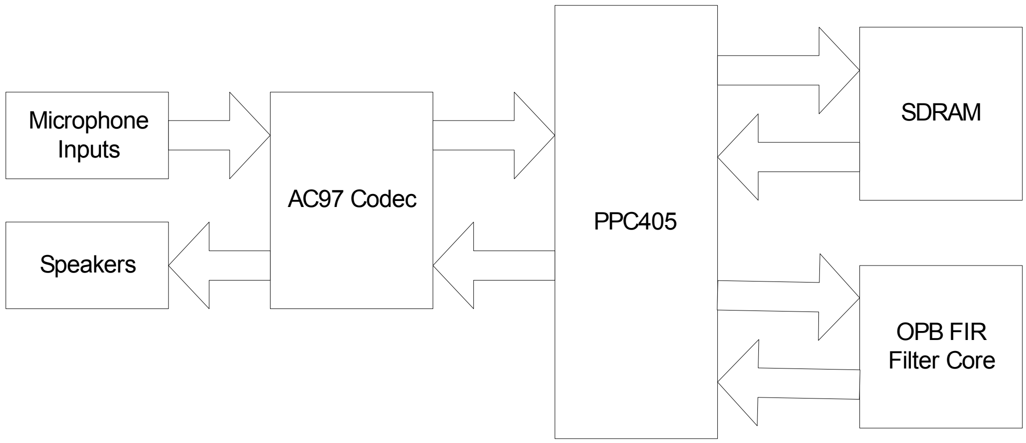


Figure 5.6: System flow diagram for the hardware/software audio filtering design

The design is different from the previous one by filtering audio signals in the FPGA. The program structure is the same as the software based design. However, instead of having the PowerPC core compute outputs, this task is now done in the FPGA for fast computation. A sample of the source code can be found in Appendix F.

5.3 System Performance

Different execution time measurements are taken from the running system.

- Total Samples: The time it takes to read 200,000 samples from the AC97 codec
- Filtering Time: The time it takes for an input signal to be filtered
- Playback Time: The time it takes to play all filtered signals to speakers
- Total: Non real-time Audio signal processing time, which is the summation of the above three.
- RT Filtering: Real-time filtering. The program takes an input, filters the signal, and plays it back, and it loops 200,000 times to process all signals.

The following tables show the execution time in clock cycles for different systems.

Table 5.2: Execution time measurements for the band-pass filter

	Soft FIR Filter		Hard FIR Filter	
	Cached	Non-cached	Cached	Non-cached
Total Samples	1249439427	1249465080	1249513259	1249439571
Filtering Time	144793993	1459799709	40133456	74800047
Playback Time	1245970725	1245315936	1246234493	1244900220
Total	2638278583	3856175346	2535881168	2574206502
RT Filtering	1249514438	1722997476	1249513946	1249440441

Table 5.3: Execution time measurements for the low-pass filter

	Soft FIR Filter		Hard FIR Filter	
	Cached	Non-cached	Cached	Non-cached
Total Samples	1249513271	1249464546	1249513440	1249440486
Filtering Time	144793921	1391400264	40133420	76937316
Playback Time	1247438840	1248730965	1249729913	1244882928
Total	2638278639	3886181352	2535881032	2567831730
RT Filtering	1249514396	1722997098	1249513955	1249440411

Table 5.4: Execution time measurements for the high-pass filter

	Soft FIR Filter		Hard FIR Filter	
	Cached	Non-cached	Cached	Non-cached
Total Samples	1249512893	1249464306	1249512835	1249446984
Filtering Time	144793930	1391400234	40133366	69046302
Playback Time	1247438018	1248730311	1249728731	1246374300
Total	2638278283	3886180797	2535880670	2574206007
RT Filtering	1249514300	1722990300	1249513802	1249440315

If we look at the filtering time for both cached systems, the execution time is improved by:

$$1 - \frac{HFT}{SFT} \approx 72\%$$

Where HFT is the cached hard FIR filter filtering time, and SFT is the cached soft FIR filter filtering time. For the real-time cached systems, they have similar execution time, which is also close to the execution time for total samples and playback time. The AC97 codec is full-duplex, so the execution time is similar for read, write, and read/write. The main reason we do not see significant performance improvement in real-time systems is because of the delay for audio compression/de-compression in the AC97 codec. This is proven in Table 5.5. As we can see, more delay is added as the sample size increases. One comment on the real-time systems: The sound quality is better and quieter when the filtering is done in hardware. Software filtering slows down the process, and adds noise that is obvious to human ears.

Table 5.5: Execution time measurements for the AC97 codec read/write

Samples	Execution time (clock cycles)
1	1160
10	9983
100	170228
1,000	5795024
10,000	62043653
100,000	624529317

5.4 Adaptive Filtering

Applications such as voice cancellation and unknown system identification require the use of adaptive filters. Adaptive filtering reacts to run-time events, which is considered as real-time responsiveness. Design of an adaptive filter requires proper algorithms and specifications. The filter coefficients for an adaptive filter are generated at run-time in a DSP processor. For the completeness of this experiment in observing real-time responsiveness of the systems, an emulation of adaptive voice filter has been designed. The aim is to take advantage of the configurable logic to enable for run-time updating of the coefficients. The filter specification is as follows:

Table 5.6: Voice filter specification

	Low-pass	High-pass
Pass-band Frequency1	1,000Hz	3,000Hz
Stop-band Frequency1	2,000Hz	2,000Hz
Sampling Frequency	44,100Hz	44,100Hz
Pass-band Ripple	0.1	0.1
Stop-band Attenuation	30dB	30dB

The first 10,000 audio signals are filtered by the low-pass filter. The rest of signals are filtered by the high-pass filter. The filter should react to the event change at the 10,001th input signal. Figure 5.7 shows the plot of the real-time adaptive filtering system responsiveness.

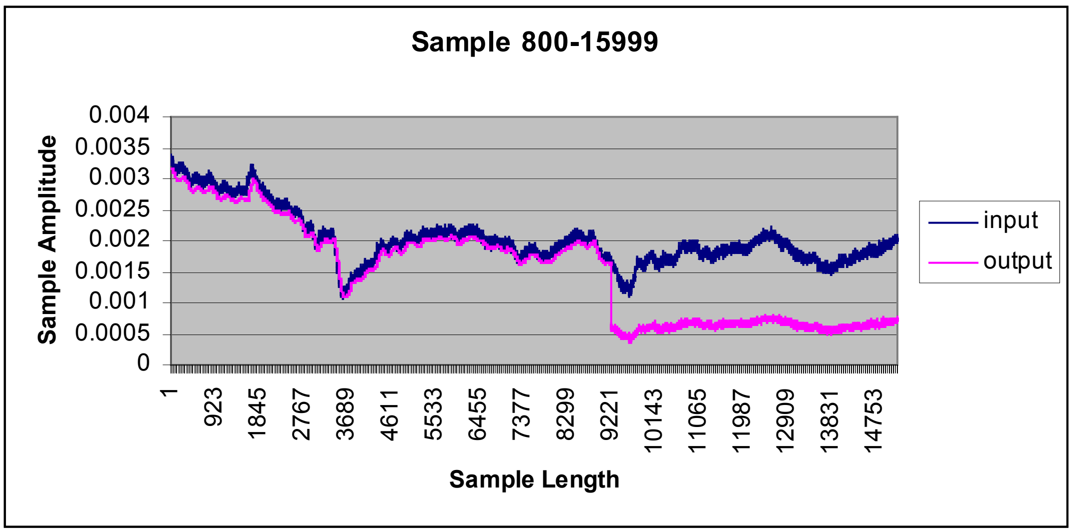


Figure 5.7: Input/output plot for the adaptive filter

5.5 Conclusion

From the experiment results, we can see that the filtering time is improved significantly by using the hardware instead of software. The real-time responsiveness of the system is verified in the adaptive filter design. The overall execution time for the audio filtering application running in real-time is similar for both systems. This is due to the significant signal compression/de-compression delay in the AC97 codec.

Traditionally, embedded applications are designed solely in a micro-controller. For complex applications, we usually need to make a trade-off between accuracy and performance. We can either choose to have accurate outputs in a slow system, or to lose some accuracy for achieving the real-time aspect. With the Virtex-II Pro FPGA, we can now achieve better performance by using the hardware for certain tasks, and still be able to maintain the accuracy of outputs. The idea of hardware/software partitioning is new, and this trend will continue to be researched in the embedded systems field. Hardware and software partitioning is promoted by the emergency of system on chip and the resulting programmable logic systems such as Virtex-II Pro. Current research focuses on developing algorithms to automate the partitioning. In this thesis partitioning of hardware and software is performed manually.

CHAPTER SIX

RESEARCH CONTRIBUTIONS AND FUTURE WORK

The Virtex-II Pro development system is a complex embedded system, and many of its capabilities remain unexplored. With two PowerPC processors, parallel computing is certainly an area one would like to explore. An embedded operating system offers more flexibility and capability to an embedded application. Porting an operating system to the Virtex-II Pro FPGA is definitely of interest. The development system has powerful components, in particular, the system on a programmable chip. As researchers, we would like to find out how to efficiently use the development system to produce high performance embedded applications. In this research project, I have investigated all of the above aspects, and provided insightful information. The pioneer work I have done has this far received encouraging feedback from graduate students and professors across the nation and beyond who have found my research helpful. This research project has a significant contribution in institutions, and has been used in embedded system courses in at least two universities: Washington State University and Rochester Institute of Technology. As more people use this type of development system, the research work will continue to have an impact.

Future work of this research includes exploring possibilities of converting high level programs to HDL in order to reduce the time in designing a hardware/software partitioning system, designing algorithms to automate hardware/software partitioning, and experimenting with monitoring of embedded applications by using some of the features to record runtime events of note for offline replay when required.

BIBLIOGRAPHOY

[1] J. Lillie, "2000 Virtex-II Pro based SoPC design", Rochester Institute of Technology, pp5, pp38.

[2] Xilinx Inc, "Getting Started with EDK", Xilinx Inc, September 2, 2003.

[3] Xilinx Inc, "Platform Studio User Guide", Xilinx Inc, February 15, 2005.

[4] Xilinx Inc, "Xilinx University Program Virtex-II Pro Development System Hardware Reference Manual", Xilinx Inc, March 8, 2005.

[5] Xilinx Inc, "PowerPC Processor Reference Guide", Xilinx Inc, September 2, 2003, pp. 32-33.

[6] Xilinx Inc, "MicroBlaze Processor Reference Guide", Xilinx Inc, October 5, 2005, pp. 11-40.

[7] IBM Corp, "CoreConnect Bus Architecture", IBM Corp, http://www-03.ibm.com/chips/products/coreconnect/

[8] B. Barney, "Introduction to Parallel Computing", Livermore Computing, http://www.llnl.gov/computing/tutorials/parallel_comp/

[9] M. Saini, "Unleash Your Creativity with Embedded Linux on Virtex-II Pro FPGAs", Xilinx, Inc, 2004.

[10] D. Kegel, "Crosstool", Google, http://kegel.com/crosstool/#download

[11] E. Anderson, "BusyBox", http://www.busybox.net/downloads/BusyBox.html

[12] B. Nelson and B. Baillio, "Configuring and Installing Linux on the Xilinx FPGA Boards", Brigham Young University, http://splish.ee.byu.edu/projects/LinuxFPGA/configuring.htm

[13] W. Klingauf, "Virtex2Pro & Linux", http://www.klingauf.de/v2p/index.phtml

[14] A. Rubini, and J. Corbet, "Linux Device Drivers, Second Edition", O'Reilly, June 2001, http://www.xml.com/ldd/chapter/book/

[15] D. E. Thomas, J. K Adams and H. Schmit, "A Model and Methodology for Hardware Software Codesign," *IEEE Design and Test for Computers*, Vol. 10, No 3, September 1993, pp. 5-15

[16] M. Baleani, A. Ferrari, A. Sangiovanni-Vincentelli, and C. Turchetti, "HW/SW Codesign of an Engine Management System," *Proceedings Europe Conference and Exhibition 2000 Design Automation and Test*, March 2000, pp.263-267

[17] "DSPGuru", Dspguru.com

[18] "FIR Filter Designer Pro", http://www.ittc.ku.edu/~rvc/projects/firfilter/normal/

[19] T-W Lee, A. J. Bell, and R. Orglmeister, "Blind Source Separation of Real World Signals" *International Conference on Neural Networks*, Vol. 4, June 9-12, 1997, pp. 2129-2134.

[20] T. Rissa, R. Uusikartano and J. Niittylahti, "Adaptive FIR Filter Architectures for Run-Time Reconfigurable FPGAs," *Proceedings of the IEEE 2002 Conference on Field-Programmable Technology*, December 16-18, 2002, pp. 52-59.

APPENDIX

A. NETWORKING APPLICATION

```
/*****************************************************************
 * Description: This program is used to demonstrate networking
 * and custom IP core design on the Virtex-II Pro development
 * system. The program interacts with a web browser and displays
 * a user input between 0-F to the 4-bit LEDs. The input value
 * is multiplied by 4 in the FPGA. The product will be read by
 * the processor and displayed to the terminal.
 *
 * Author: Jamie Lin
 * Date Created: 02/24/2006
 *****************************************************************/
#include <string.h>
#include <net/xilsock.h>
#include <xgpio.h>
#include <xemac_l.h>
#include "xparameters.h"
#include "multiplier.h"
#include "xutil.h"

#define SERVER_PORT 8080
#define MAXWEBCONNS 1
#define MAXPENDING 4

unsigned char hw_addr[]="00:11:22:33:44:55";  // hardcoded mac address
for the board
Xuint8 ip_addr[16]="192.168.0.3";    // this ip address can be changed
XGpio led;
int conns[MAXWEBCONNS]; // this is used to record web connections


/** The following defines html files required for the application **/
unsigned char *index_html = "<html><head></head><body
bgcolor=\"#FFCC99\"><h1>Simple Web Test for the Virtex-II Pro
Development System</h1><br><br><h5>Enter a hex number between 0 and F
to display LEDs, or 'q' to quit:</h5><form action=\"JL\"
method=\"get\"><input type=\"text\" size=\"1\" name=\"textbox\"><input
type=\"submit\" value=\"Submit\"><br></form>";
unsigned char *led_html = "<p>LEDs:</p><table border=\"1\"
cellpadding=\"0\" cellspacing=\"0\" style=\"border-collapse: collapse\"
bordercolor=\"#111111\" width=\"7%\" id=\"AutoNumber1\"><tr><td
width=\"25%\">3</td><td width=\"25%\">2</td><td width=\"25%\">1</td><td
width=\"25%\">0</td></tr><tr>";
unsigned char *led_on_html = "<td width=\"25%\"
bgcolor=\"#00FF00\"> </td>";
unsigned char *led_off_html = "<td width=\"25%\"
bgcolor=\"#FFFFFF\"> </td>";
unsigned char *end_html = "</tr></table><p>LED bits are inverted.
<br>LED 3 is the LSB and LED 0 is the MSB</p></body></html>";
unsigned char *http_hdr = "HTTP/1.1 200 OK\n\rContent-type:
text/html\n\rConnection: close\n\r\r\n";

/****************************************************/
 * Initialize network interface
 *
```

```
 * @return
 *   nothing
 *******************************************************/
void init_net()
{

      // set up MAC.
      // The MAC address will be used as the source ethernet address
      // in all the ethernet frames
      xilnet_eth_init_hw_addr(hw_addr);

      // initialize hardware address table. This function must
      // be called before using other functions of LibXilNet.
      xilnet_eth_init_hw_addr_tbl();

      // set up address
      xilnet_ip_init(ip_addr);

      // Initialize the MAC OPB base address (MAC driver in net/mac.c)
      xilnet_mac_init(XPAR_ETHERNET_MAC_BASEADDR);

      // set the station address of the EMAC device
      XEmac_mSetMacAddress(XPAR_ETHERNET_MAC_BASEADDR,mb_hw_addr);

      // enable the transmitter and receiver.
      // preserve the contents of the control register.
      XEmac_mEnable(XPAR_ETHERNET_MAC_BASEADDR);

      // reset MII compliant PHY.
      XEmac_mPhyReset(XPAR_ETHERNET_MAC_BASEADDR);


      xil_printf("Net Initialization Done\n\r");

      // Print IP address
      xil_printf("WEB server IP: %s\n\r",ip_addr);
}

 /*************************************************
 * Initialize the socket interface
 *
 * @param
 *   struct sockaddr_in: socket structure
 *
 * @return
 *   int : Status
 *         s : socket number
 *         -1: error
 *************************************************/
int init_socket(struct sockaddr_in *addr)
{
      int s;

      // get a socket descriptor
      if((s = xilsock_socket(AF_INET,SOCK_STREAM,AF_INET)) == -1)
      {
            xil_printf("socket error\n\r");
```

```
            return -1;
      }

      // set up structure for the internet socket
      addr->sin_family = AF_INET; //Internet address family
      addr->sin_port = SERVER_PORT;
      //accept any incoming interface

      addr->sin_addr.s_addr = INADDR_ANY;
```

//bind socket given the descriptor s to the ip address/port

//number pair given in structure pointed to by addr.

```
      if(xilsock_bind(s,(struct sockaddr *)addr,sizeof(struct
                  sockaddr)) == -1)
      {
            xil_printf("bind error\n\r");
            return -1;
      }

      //listen to a max of 5 connections
      if(!(xilsock_listen(s,MAXPENDING)))
      {
            printf("listen error\n\r");
            return -1;
      }

      xil_printf("Web server start...\n\r");
      return s;
}

/*********************************************
 * This function converts the input char to a
 * decimal number
 *
 * @param
 *      c: character to be converted
 *
 * @return
 *   char : decimal number
 *
 *********************************************/
char get_number(char c)
{
      char data;

        if (c >= '0' && c <= '9' )
            data = c - 48 ;
        else if ( c >= 'a' && c <= 'f' )
                  data = c - 87;
            else if (c >= 'A' && c <= 'F')
                  data = c - 55;

      return data;
}
```

```
/*********************************************
 * This routine is used to convert a decimal
 * number into binary representation
 *
 * @param
 *      number: decimal number
 *          bit[]: binary bits
 *
 * @return
 *   none
 *
 *********************************************/
void decimal2binary(int number, int bit[])
{
      int i,j,k,n;

      n = number;
      j = n/2;
      k = n%2;
      for(i=0; i<4; i++)
      {
            bit[i] = k;
            k = j % 2;
            j /= 2;
      }
}


/*********************************************/
/*
 * Handle a client connection
 *
 * @param
 *   int n: Connection number
 *
 * @return
 *   int : Status
 *         -1 : Terminate WEB server
 *         0  : Normal
 ********************************************/
int handle_client(int n)
{

    int  i,num;
    char *tok;
      int led_bit[4];
      int mul1, mul2, product;
      char led_data;    // user input data

    // Obtain a pointer to the location in the "send frame" where the
    // data is supposed to start. The pointer needs to
    // skip over all of the header information.
    unsigned char *sndptr = (unsigned char*)(sendbuf +
                                          LINK_HDR_LEN +
                                          IP_HDR_LEN*4 +
                                          (TCP_HDR_LEN*4));
```

```
unsigned int http_hdr_len = strlen(http_hdr);
  unsigned int index_len = strlen(index_html);
  unsigned int led_begin_len = strlen(led_html);
  unsigned int end_len = strlen(end_html);
  unsigned int led_len;

// Obtain a pointer to the receive buffer.
// Global array of sockets (xilsock.h)
unsigned char *receive_buffer =
              xilsock_sockets[conns[n]].recvbuf.buf;

  int led_html_size = strlen(led_on_html) * 4;
  unsigned char led_status_html[led_html_size];

// nothing to do
if (!receive_buffer)
    return 0;

//Ack for Data Sent
if (xilsock_status_flag & XILSOCK_TCP_ACK)
  {
    xil_printf("XILSOCK_TCP_ACK\r\n");
    xilsock_close(conns[n]);
        conns[n] = -1;  // free the connection in the array
}

//GET Request
else if (xilsock_status_flag & XILSOCK_TCP_DATA)
  {
    xil_printf("XILSOCK_TCP_DATA\r\n");

    // Find the first space in the receive buffer
    tok = strtok(receive_buffer, " ");
    // Find the scond space in the receive buffer
    tok = strtok('\0', " ");
    // Increment the pointer.
    tok ++;

    // See the submit button is pushed
    if (tok[0] == 'J' && tok[1] == 'L')
        {
              // quit the application
              if(tok[11] == 'q' || tok[11] == 'Q')
                    return -1;

              xil_printf("Set LEDs...\r\n");

              // tok[11] is where user input hex number
              // will be stored in the http header file
              led_data = get_number(tok[11]);

              printf("Your input is: %d.\n\r",led_data);
              XGpio_DiscreteWrite(&led,1,15-led_data);


              // the multiplier takes the user input data for LEDs,
```

```
            // and multiply it by 4 in the FPGA. The processor
            // reads the product off the register and displays it
            // on the terminal

            // write the led value to register0

            MULTIPLIER_mWriteReg(XPAR_MULTIPLIER_0_BASEADDR,0,
                                                    led_data);

            // confirm the register value by reading it back
            mul1 = MULTIPLIER_mReadReg
                        (XPAR_MULTIPLIER_0_BASEADDR,0);
            xil_printf("mul1: %d\n\r",mul1);

            // write a 4 to register1
            MULTIPLIER_mWriteReg
                  (XPAR_MULTIPLIER_0_BASEADDR,0x4,4);

            // confirm the register value by reading it back
            mul2 = MULTIPLIER_mReadReg
                        (XPAR_MULTIPLIER_0_BASEADDR,0x4);
            xil_printf("mul2: %d\n\r",mul2);
            sleep(1);

            // read the value
            product = MULTIPLIER_mReadReg
                  (XPAR_MULTIPLIER_0_BASEADDR,0x8);
            xil_printf("mul1 * mul2 = : %d\n\r",product);

}

      // send the same LEDs value back to the browser
      decimal2binary(led_data, led_bit);

      // reset buffer for led html table
      memset(led_status_html,0,led_html_size);

      // compose the led_status_html for all the 4 LEDs
      if(led_bit[0] == 0)
            strcpy(led_status_html,led_off_html);
      else
            strcpy(led_status_html,led_on_html);

      for(i=1; i<4; i++)
      {
            if(led_bit[i] == 0)
                  strcat(led_status_html, led_off_html);
            else
                  strcat(led_status_html, led_on_html);
      }

      led_len = strlen(led_status_html);

      /*********************************************************/

// part of stdlib. Fills the 'sendbuf' buffer with all zeros
memset(sendbuf, 0, LINK_FRAME_LEN);
```

```
        // Set the HTTP 1.1 header
        memcpy(sndptr, http_hdr, http_hdr_len);

        // compose the html file
      memcpy((sndptr+http_hdr_len),index_html,index_len);
      memcpy((sndptr+http_hdr_len+index_len),led_html, led_begin_len);
      memcpy((sndptr+http_hdr_len+index_len+led_begin_len),
                  led_status_html, led_len);
      memcpy((sndptr+http_hdr_len+index_len+led_begin_len+led_len),
                  end_html, end_len);

        xil_printf("Send on socket: %d\r\n", conns[n]);
        num = xilsock_send(conns[n], sendbuf,
            http_hdr_len+index_len+led_begin_len+led_len+end_len);
        memset(sendbuf, 0, LINK_FRAME_LEN);
        xil_printf("done..\n\r");

        return 0;
    }

}

/****************************************
 * Add a web connection
 *
 * @param: socket number
 *
 * @return:
 *          success: array subscript
 *          error: -1
 ***************************************/
int add_connection(int s)
{
    int i;

    // search for a free connection
    for ( i = 0; i < MAXWEBCONNS; i++)
      {
        if (conns[i] == -1)
        {
                  conns[i] = s;
            xil_printf("Connection %d added\n\r", conns[i]);
            return i;
            }
      }

    xil_printf("Can't add a new connection\n\r");
    return -1;
}


/*****************************************/
 * Process all web connections
 * @param
 *   None
 *
```

```
 * @return
 *   int : Status
 *         -1 : Terminate WEB server
 *         other: Normal
 *
 * @note
 *   None
 *
 ******************************************/
int process_connections()
{
   int i, result;


   for (i = 0; i < MAXWEBCONNS; i++)
       result = handle_client(i);

   return result;
}


int main()
{
      int i;
    int s;
    int client_sock;
    struct sockaddr_in addr;

    // Initialize network device
    init_net();

      // Create the socket
    if((s = init_socket(&addr)) == -1)
      {
        xil_printf("socket() error \n\r");
        exit(1);
    }

      // initialize LEDs
      XGpio_Initialize(&led,XPAR_LEDS_4BIT_DEVICE_ID);
      XGpio_SetDataDirection(&led,1,0);   // set to 0 as outputs
      XGpio_DiscreteWrite(&led,1,0xf);    // turn off all LEDs

      // initialize web connection array
      for(i=0; i<MAXWEBCONNS; i++)
            conns[i] = -1;

      for (;;)
      {

        int addr_len = 0;

        addr_len = sizeof(struct sockaddr);

        // Accept a new connection. Sets the global
        // xilsock_status_flag.
        client_sock = xilsock_accept(s, (struct sockaddr *)&addr,
```

```
                        &addr_len);

        // A new connection was found. Add it to the array
        if (xilsock_status_flag & XILSOCK_NEW_CONN)
            add_connection(client_sock);

        // Process all existing connections
        if (process_connections()==-1)
            {
            print("WEB server terminated\r\n");
            return;
        }

    }
    return;
}
```

B. DUAL-CORE DESIGN

```
/*****************************************************************
 * PPC0 TASK CODE
 *
 * Description: This program reads the status of switches, and
 * saves the value to a shared memory location. The value is printed
 * to the terminal. The program also prints the value of a counter
 * that's modified by ppc1 to the terminal.
 *
 * Author: Jamie Lin
 * Date Created: 11/07/2005
 * Revision History:
 *****************************************************************/
#include "xparameters.h"
#include "xuartns550_l.h"
#include "xutil.h"
#include "xgpio.h"

int main (void)
{
        volatile int *shared0, *shared1;
        XGpio sw;

        // initialize dip switches
        XGpio_Initialize(&sw, XPAR_DIPSWS_4BIT_DEVICE_ID);

        // set as inputs
        XGpio_SetDataDirection(&sw,1,0xffffffff);

        // two locations are used, one for status of switches
         // and the other for an integer value
        shared0 = XPAR_BRAM_CNTLR_SHARED_PPC0_BASEADDR;
        shared1 = XPAR_BRAM_CNTLR_SHARED_PPC0_BASEADDR + 32;


        // Initialize RS232_Uart_1 - Set baudrate and number
        // of stop bits
        XUartNs550_SetBaud(XPAR_RS232_UART_1_BASEADDR,
                               XPAR_XUARTNS550_CLOCK_HZ, 9600);
        XUartNs550_mSetLineControlReg(XPAR_RS232_UART_1_BASEADDR,
                               XUN_LCR_8_DATA_BITS);

        print("-- Entering main() --\r\n");

        while(1)
        {
                // read the status of dip switches and save it into
                // a shared memory location
                *shared1 = XGpio_DiscreteRead(&sw,1);
                xil_printf("shared data modified by ppc_0:
                               %d\n\r",*shared0);
                xil_printf("dip switches value displayed on LEDs by
                               ppc1: %d\n\r",*shared1);
                sleep(1);
        }
```

```
   print("-- Exiting main() --\r\n");
   return 0;
}



/*********************************************************************
 * PPC1 TASK CODE
 *
 * Description: This program increments a counter in a shared memory
 * location. It also reads the value of switches monitored by ppc0,
 * and displays the value to LEDs.
 *
 * Author: Jamie Lin
 * Date Created: 11/07/2005
 * Revision History:
 *********************************************************************/
#include "xparameters.h"
#include "xutil.h"
#include "xgpio.h"

int main()
{
        volatile int *shared0, *shared1;
        XGpio led;
        int led_val;
        int i=0; // used to modify shared data

        // initialize leds
        XGpio_Initialize(&led, XPAR_LEDS_4BIT_DEVICE_ID);

        // set as outputs
        XGpio_SetDataDirection(&led,1,0);

        // two locations are used, one for status of switches and the
        // other for an integer value
        shared0 = XPAR_BRAM_CNTLR_SHARED_PPC1_BASEADDR;
        shared1 = XPAR_BRAM_CNTLR_SHARED_PPC1_BASEADDR + 32;

        while(1)
        {
                // write the value of dip switches from ppc0 to leds in
                // ppc1
                XGpio_DiscreteWrite(&led,1, *shared1);

                // increment the value of shared integer data, and print
                //it out to terminal in ppc0
                *shared0 = i++;
                sleep(1);
        }

    return 0;
}
```

C. LOADABLE MODULE

Driver Code:

```
/************************************************************
 * Sample loadable kernel module for IO operations
 *
 * Description: This loadable kernel module performs read/write
 *                             tests on a register.
 *
 * Author: Jamie Lin
 *
 * Date Created: 01/30/2006
 * Revision History:
 *
 *
 ************************************************************/
#include <linux/init.h>    // to use module_init and module_exit
#include <linux/module.h>  // macros for modules
#include <linux/kernel.h>
#include <linux/ioport.h>
#include <linux/errno.h>
#include <asm/io.h>

MODULE_AUTHOR("Jamie Lin <jamiehl@mail.wsu.edu>") ;
MODULE_DESCRIPTION("Generic Module for FPGA cores") ;
MODULE_SUPPORTED_DEVICE("Custom IP on the XUPV2P Board") ;
MODULE_LICENSE("GPL") ;

EXPORT_NO_SYMBOLS ;

static unsigned int    io_reg = 0xa;          // test data
static unsigned int    *virtual_base = 0;     // remapped address
static unsigned long   mem_addr = 0x7d600000;  // IP base address
static unsigned long   mem_size = 0x10000;    // 64KB

MODULE_PARM(mem_addr,"i") ;
MODULE_PARM_DESC(mem_addr,"base address of I/O memory for the custom IP
core") ;
MODULE_PARM(mem_size,"i") ;
MODULE_PARM_DESC(mem_size,"size of I/O memory segment for the custom IP
core") ;


int io_driver_init(void)
{
        int i;

        if(check_mem_region(mem_addr,mem_size))
        {
            printk("XGPIO: memory already in use\n");
            return -EBUSY;
        }

        // request memory for the device
```

```
        request_mem_region(mem_addr,mem_size,"xgpio");

        // remap
        virtual_base = ioremap_nocache(mem_addr,mem_size);
        printk("ioremap: Virtual Address %08x\n",(unsigned
                       int)virtual_base);
        printk("Physical Address %08x\n",(unsigned
                       int)virt_to_phys(virtual_base));

        if( virtual_base==0 )
        {
            printk("ioremap failed\n");
            return -EBUSY ;
        }
        else
        {
                    printk("Data to write out: %08x\n",io_reg);
                    writel(io_reg,virtual_base);
                    wmb() ;
                    for( i=0 ; i<10000 ; i++);

                    io_reg = 0x5;  // give it some other value
                    printk("Change io_reg to: %08x\n",io_reg);
                    barrier();
                    io_reg = readl(virtual_base);
                    rmb() ;
                    printk("Read register value into io_reg:
                               %08x\n",io_reg) ;
                    return 0; // indicate a success
        }
}

void io_driver_exit(void)
{
        printk("Release Memory Region...\n") ;
        iounmap(virtual_base) ;
        release_mem_region(mem_addr,mem_size) ;
}

module_init(io_driver_init);
module_exit(io_driver_exit);
```

Driver Makefile:

```
KERNELDIR=/home/jamie/tempkernel/linuxppc_2_4_devel_cp1

include $(KERNELDIR)/.config

CC = powerpc-405-linux-gnu-gcc
LD = powerpc-405-linux-gnu-ld
CFLAGS = -D__KERNEL__ -DMODULE -I$(KERNELDIR)/include \
    -I$(KERNELDIR)/arch/ppc \
    -O -Wall
```

```
ifdef CONFIG_SMP
  CFLAGS += -D__SMP__ -DSMP
endif

io_driver.o: io_driver.c

skull.o: skull_init.o skull_clean.o
	$(LD) -r $^ -o $@


clean:
	rm -f io_driver.o
```

D. ENHANCED LOADABLE MODULE

Driver Source Code:

```
/******************************************************************
 * Linux Device Driver for GPIO on the XUPV2P board
 *
 * Description: The driver performs IO operations on a 32-bit
 *              register
 *
 * Author: Jamie Lin
 *
 * Date Created: 02/01/2006
 *
 * Revision History:
 *
 ******************************************************************/
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/miscdevice.h>
#include <asm/io.h>
#include <asm/uaccess.h>

#include "xparameters.h"

#define XGPIO_MINOR 23

struct xgpio_ioctl_data
{
              int data;
                  /**can add more members later**/
};


MODULE_AUTHOR("Jamie Lin");
MODULE_DESCRIPTION("GPIO driver for XUPV2P");
MODULE_LICENSE("GPL");

static u32 remapped_addr;
const static long remap_size = XPAR_GPIO_0_HIGHADDR -
XPAR_GPIO_0_BASEADDR + 1;

static int
xgpio_open(struct inode *inode, struct file *file)
{
      MOD_INC_USE_COUNT;
      return 0;
}

static int
xgpio_release(struct inode *inode, struct file *file)
{
      MOD_DEC_USE_COUNT;
```

```
	return 0;
}

static int
xgpio_read(struct file *file, struct xgpio_ioctl_data *io_data)
{
	struct xgpio_ioctl_data ioctl_data;

	ioctl_data.data = readl(remapped_addr);
	rmb();

	/* copy data to the user space */
	if (copy_to_user((struct xgpio_ioctl_data *)io_data,
			&ioctl_data, sizeof (ioctl_data)))
	{
		return -EFAULT;
	}
	return 0;
}

static int
xgpio_write(struct file *file, struct xgpio_ioctl_data *io_data)
{
	struct xgpio_ioctl_data ioctl_data;

	/* copy input from the user space */
	if (copy_from_user(&ioctl_data, io_data, sizeof (*io_data)))
		return -EFAULT;

	writel(ioctl_data.data,remapped_addr);
	wmb();
	return 0;
}

static struct file_operations xfops = {
	owner:THIS_MODULE,
	open:xgpio_open,
	release:xgpio_release,
	write: xgpio_write,
	read: xgpio_read
};
/*
 * We get to all of the GPIOs through one minor number.  Here's the
 * miscdevice that gets registered for that minor number.
 */
static struct miscdevice miscdev = {
	minor:XGPIO_MINOR,
	name:"xgpio",
	fops:&xfops
};

static int xgpio_init(void)
{
	int rtn;

	/* remap address */
	remapped_addr = (u32)ioremap(XPAR_GPIO_0_BASEADDR,remap_size);
```

```
	/* Register the driver with misc and report success. */
	rtn = misc_register(&miscdev);
	if (rtn)
	{
		printk(KERN_ERR "%s: Could not register
driver.\n",miscdev.name);
		return rtn;
	}
	return 0;
}

static void xgpio_cleanup(void)
{
	iounmap(remapped_addr);
	misc_deregister(&miscdev);
}

EXPORT_NO_SYMBOLS;

module_init(xgpio_init);
module_exit(xgpio_cleanup);
```

Driver Test Code:

```
/*********************************************************
 * This program tests the custom GPIO driver
 *
 * Description: The program writes a value to the
 *              register, and then reads it back. Two
 *              values are compared to make sure
 *              it works correctly.
 *
 * Author: Jamie Lin
 *
 * Date Created: 02/01/2006
 *
 * Revision History:
 *
 *********************************************************/
#include <stdio.h>
#include <fcntl.h>

#include "adapter.h"

int main()
{
	int io_test, i;
	int handle;
	struct xgpio_ioctl_data io_data;

	/* Opening */
	handle = open("/dev/xgpio", O_RDWR);
	if(handle > 0)
```

```
		printf("XGPIO Driver Opened %d\n",handle);
	else
	{
		printf("Error opening GPIO\n");
		exit(1);
	}

	/* Write a value to the register */
	io_data.data = 8;
	printf("Let's write an io_data value %d to register\n",
			io_data.data);

	io_test = write(handle, &io_data);
	if(!io_test)
		printf("Successfully written out\n");
	else
		printf("Failed to write out\n");

	/* Change the value in io_data to make sure read in value is
		correct */
	io_data.data = 20;
	printf("Give io_data a random value %d\n",io_data.data);

	/* Read the value back */
	io_test = read(handle, &io_data);
	if(!io_test)
		printf("Successfully read in\n");
	else
		printf("Failed to read\n");

	printf("Read in io_data value = %d\n", io_data.data);

	/* Closing */
	if(close(handle))
		printf("Couldn't close /dev/xgpio\n");
	else
		printf("Closed /dev/xgpio\n");

	return 0;
}
```

E. SOFT AUDIO FILTERING APPLICATION

```
/*******************************************************************
 * Soft FIR Filter
 *
 * Lowpass Filter:
 * PassBand Frequency 1 = 10000Hz
 * StopBand Frequency 1 = 12000Hz
 * Sampling Frequency set at 44100Hz
 * PassBand Riple = 0.1
 * StopBand Attenuation 30dB
 *
 * Description: The program receives data from the mic input, filters
 * the sounds, and plays them back to the speakers.
 *
 * Thank you for the FIR Basic C code from dspguru.org.
 *
 * Author: Jamie Lin
 * Data Created: 03/30/2006

*******************************************************************/
#include "xparameters.h"
#include "xuartns550_l.h"
#include <stdio.h>
#include "xac97_l.h"
#include "xcache_l.h"

#define SAMPLE Xuint32
#define NTAPS 30
#define SAMPLE_SIZE 200000


/*************************************************************
 * fir_basic: Does the basic FIR algorithm: store input sample,
 * calculate output sample, move delay line
 *************************************************************/
SAMPLE fir_basic(SAMPLE input)
{
   static const int h[NTAPS/2] = {2, 1, -3, -2,
                              4, 5, -6, -9, 7, 17,
                              -9, -34, 9, 110, 164};

    static SAMPLE z[2 * NTAPS];
    int ii, j, mid;
    SAMPLE accum;

    /* store input at the beginning of the delay line */
    z[0] = input;

    /* calc FIR */
    accum = 0;

    j=NTAPS/2 - 1;
    mid = NTAPS/2;
    for (ii = 0; ii < NTAPS; ii++)
        {
```

```
                if(ii < mid)
                accum += ((SAMPLE)h[ii]) * z[ii];
                else
                {
                        accum += ((SAMPLE)h[j]) * z[ii];
                        j--;
                }
    }

    /* shift delay line */
    for (ii = NTAPS - 2; ii >= 0; ii--)
        {
        z[ii + 1] = z[ii];
    }

    return accum;
}

void SoftFIR()
{
  int i, sampleCnt=0, j=0;
  SAMPLE sample[SAMPLE_SIZE], output[SAMPLE_SIZE];

    printf("Initializing audio chip...\n");
    XAC97_InitAudio(XPAR_AUDIO_CODEC_BASEADDR, 0);
    XAC97_EnableInput(XPAR_AUDIO_CODEC_BASEADDR, AC97_MIC_INPUT);

    printf("MIC Recording...\n");
    while(sampleCnt < SAMPLE_SIZE)
    {
        sample[sampleCnt] = XAC97_ReadFifo(XPAR_AUDIO_CODEC_BASEADDR);
        sampleCnt++;
    }
    printf("Recording done. Total samples = %d\n", sampleCnt);

    //printf("\nFIR Filtering...\n");
    for (i = 0; i < SAMPLE_SIZE; i++)
        output[i] = fir_basic(sample[i]);

    //printf("Playing outputs...\n");
        for(i=0; i< SAMPLE_SIZE; i++)
                XAC97_WriteFifo(XPAR_AUDIO_CODEC_BASEADDR, output[i]);

    // reset AC97
    XAC97_SoftReset(XPAR_AUDIO_CODEC_BASEADDR);
    printf("Done...\n");
}


int main()
{

   /* Initialize RS232_Uart_1 - Set baudrate and number of stop bits */
   XUartNs550_SetBaud(XPAR_RS232_UART_1_BASEADDR,
                          XPAR_XUARTNS550_CLOCK_HZ, 115200);
   XUartNs550_mSetLineControlReg(XPAR_RS232_UART_1_BASEADDR,
                                          XUN_LCR_8_DATA_BITS);
```

```
   printf("\n\n------------------------\n");
   printf("Soft FIR Filter\n");
   printf("------------------------\n");

   printf("Enabling Caches...\n");
   XCache_EnableICache(0x70000001);
   XCache_EnableDCache(0x70000001);

   XAC97_HardReset(XPAR_AUDIO_CODEC_BASEADDR);

   // sampling rate: 44100Hz
   XIo_Out32(AC97_PCM_DAC_Rate, 0xAC44);
   XIo_Out32(AC97_PCM_ADC_Rate, 0xAC44);

   SoftFIR();

   return 0;
}
```

F. HARD/SOFT AUDIO FILTERING APPLICATION

SOFTWARE CODE

```
/******************************************************************
 * Hard FIR Filter
 *
 * Lowpass Filter:
 * PassBand Frequency 1 = 10000Hz
 * StopBand Frequency 1 = 12000Hz
 * Sampling Frequency set at 44100Hz
 * PassBand Riple = 0.1
 * StopBand Attenuation 30
 *
 * Description: The program receives data from the mic input, filters
 * the sounds, and plays them back to the speakers. The filtering is
 * done in the FPGA.
 *
 * Thank you for the FIR Basic C code from dspguru.org.
 *
 * Author: Jamie Lin
 * Data Created: 03/30/2006
 ******************************************************************/
#include "xparameters.h"
#include "xuartns550_l.h"
#include <stdio.h>
#include "xac97_l.h"
#include "lowpass_fir.h"
#include "xcache_l.h"

#define SAMPLE Xuint32
#define SAMPLE_SIZE 200000

void HardFIR()
{
    int i, sampleCnt=0, j=0;
    SAMPLE sample[SAMPLE_SIZE], output[SAMPLE_SIZE];

    printf("Initializing audio chip...\n");
    XAC97_InitAudio(XPAR_AUDIO_CODEC_BASEADDR, 0);
    XAC97_EnableInput(XPAR_AUDIO_CODEC_BASEADDR, AC97_MIC_INPUT);

    printf("MIC Recording...\n");
    while(sampleCnt < SAMPLE_SIZE)
    {
      sample[sampleCnt] = XAC97_ReadFifo(XPAR_AUDIO_CODEC_BASEADDR);
      XAC97_WriteFifo(XPAR_AUDIO_CODEC_BASEADDR, sample[sampleCnt]);
      sampleCnt++;
    }

    printf("Recording done. Total samples = %d\n", sampleCnt);
    printf("\nFiltering...\n");

    for (i = 0; i < SAMPLE_SIZE; i++)
    {
```

```
        LOWPASS_FIR_mWriteReg(XPAR_LOWPASS_FIR_0_BASEADDR, 0, sample[i]);
        output[i] = LOWPASS_FIR_mReadReg(XPAR_LOWPASS_FIR_0_BASEADDR,
                                                                  0x4);
        }

        printf("Playing outputs...\n");
        for(i=0; i< SAMPLE_SIZE; i++)
              XAC97_WriteFifo(XPAR_AUDIO_CODEC_BASEADDR, output[i]);

        // reset AC97
        XAC97_SoftReset(XPAR_AUDIO_CODEC_BASEADDR);
        printf("Done...\n");
}

int main()
{

   /* Initialize RS232_Uart_1 - Set baudrate and number of stop bits */
   XUartNs550_SetBaud(XPAR_RS232_UART_1_BASEADDR,
                            XPAR_XUARTNS550_CLOCK_HZ, 115200);
   XUartNs550_mSetLineControlReg(XPAR_RS232_UART_1_BASEADDR,
                            XUN_LCR_8_DATA_BITS);

   printf("\n\n------------------------\n");
   printf("Hard FIR Filter\n");
   printf("------------------------\n");

   printf("Enabling Caches...\n");
   XCache_EnableICache(0x70000001);
   XCache_EnableDCache(0x70000001);

   XAC97_HardReset(XPAR_AUDIO_CODEC_BASEADDR);

   // sampling rate: 44100Hz
   XIo_Out32(AC97_PCM_DAC_Rate, 0xAC44);
   XIo_Out32(AC97_PCM_ADC_Rate, 0xAC44);

   HardFIR();

return 0;
}
```

HARDWARE CODE

```
architecture IMP of user_logic is

  --USER signal declarations added here, as needed for user logic
  signal data_in             : std_logic_vector(0 to 31);
  signal accum_out           : std_logic_vector(0 to 63);
  constant NTAPS             : integer := 29;
  signal ii                  : integer := 0;
  ------------------------------------------
  -- Signals for user logic slave model s/w accessible register example
  ------------------------------------------
  signal slv_reg0                      : std_logic_vector(0 to
```

```
                                                     C_DWIDTH-1);
  signal slv_reg1                       : std_logic_vector(0 to
                                                    C_DWIDTH-1);
  signal slv_reg_write_select           : std_logic_vector(0 to 1);
  signal slv_reg_read_select            : std_logic_vector(0 to 1);
  signal slv_ip2bus_data                : std_logic_vector(0 to
                                                    C_DWIDTH-1);
  signal slv_read_ack                   : std_logic;
  signal slv_write_ack                  : std_logic;

begin

  --USER logic implementation added here

  process(Bus2IP_Clk, Bus2IP_WrCE)
      subtype DWORD is std_logic_vector(0 to 31);
      TYPE  filter_coefficients_type is ARRAY(0 TO NTAPS) OF DWORD;
      TYPE  storage_type is ARRAY(0 TO 2*NTAPS) OF    DWORD;
      VARIABLE h : filter_coefficients_type;
      VARIABLE z : storage_type;
      VARIABLE previous_result : std_logic_vector(0 to 63) :=
"0000000000000000000000000000000000000000000000000000000000000000";
      VARIABLE accum : std_logic_vector(0 to 63) :=
"0000000000000000000000000000000000000000000000000000000000000000";
      begin

            if(Bus2IP_Clk'event and Bus2IP_Clk='1') then
               if Bus2IP_WrCE = "10" then
                  --store input at the beginning of the delay line--
                  z(0) := Bus2IP_Data(0 to 31);

                  h(0):= "00000000000000000000000000000010"; --2
                  h(1):= "00000000000000000000000000000001"; --1
                  h(2):= "11111111111111111111111111111101"; --(-3)
                  h(3):= "11111111111111111111111111111110"; --(-2)
                  h(4):= "00000000000000000000000000000100"; --4
                  h(5):= "00000000000000000000000000000101"; --5
                  h(6):= "11111111111111111111111111111010"; --(-6)
                  h(7):= "11111111111111111111111111110111"; --(-9)
                  h(8):= "00000000000000000000000000000111"; --7
                  h(9):= "00000000000000000000000000010001"; --17
                  h(10):= "11111111111111111111111111110111"; --(-9)
                  h(11):= "11111111111111111111111111011110"; --(-34)
                  h(12):= "00000000000000000000000000001001"; --9
                  h(13):= "00000000000000000000000001101110"; --110
                  h(14):= "00000000000000000000000010100100"; --164
                  h(15):= "00000000000000000000000010100100"; --164
                  h(16):= "00000000000000000000000001101110"; --110
                  h(17):= "00000000000000000000000000001001"; --9
                  h(18):= "11111111111111111111111111011110"; --(-34)
                  h(19):= "11111111111111111111111111110111"; --(-9)
                  h(20):= "00000000000000000000000000010001"; --17
                  h(21):= "00000000000000000000000000000111"; --7
                  h(22):= "11111111111111111111111111110111"; --(-9)
                  h(23):= "11111111111111111111111111111010"; --(-6)
                  h(24):= "00000000000000000000000000000101"; --5
                  h(25):= "00000000000000000000000000000100"; --4
```

```
                h(26):= "11111111111111111111111111111110"; --(-2)
                h(27):= "11111111111111111111111111111101"; --(-3)
                h(28):= "00000000000000000000000000000001"; --1
                h(29):= "00000000000000000000000000000010"; --2


                accum :=
"0000000000000000000000000000000000000000000000000000000000000000";

                for ii in 0 to NTAPS loop
                      previous_result := h(ii)*z(ii);
                      accum := previous_result + accum;
                end loop;

                --shift delay line--
                for ii in NTAPS-1 downto 0 loop
                      z(ii+1) := z(ii);
                end loop;

                accum_out <= accum;
             end if;
          end if;

      end process;

  slv_reg_write_select <= Bus2IP_WrCE(0 to 1);
  slv_reg_read_select  <= Bus2IP_RdCE(0 to 1);
  slv_write_ack        <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1);
  slv_read_ack         <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1);

  -- implement slave model register read mux
  SLAVE_REG_READ_PROC : process( slv_reg_read_select, accum_out ) is
  begin

    case slv_reg_read_select is
      when "10" => slv_ip2bus_data <= accum_out(0 to 31);
      when "01" => slv_ip2bus_data <= accum_out(32 to 63);
      when others => slv_ip2bus_data <= (others => '0');
    end case;

  end process SLAVE_REG_READ_PROC;
```