

3-24-2011

The Development of Hardware Multi-core Test-bed on Field Programmable Gate Array

Mohan Shivashanker

Florida International University, mohan.uvce@gmail.com

DOI: 10.25148/etd.FI11050902

Follow this and additional works at: <https://digitalcommons.fiu.edu/etd>

Recommended Citation

Shivashanker, Mohan, "The Development of Hardware Multi-core Test-bed on Field Programmable Gate Array" (2011). *FIU Electronic Theses and Dissertations*. 395.
<https://digitalcommons.fiu.edu/etd/395>

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

THE DEVELOPMENT OF HARDWARE MULTI-CORE TEST-BED ON
FIELD PROGRAMMABLE GATE ARRAY

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

by

Mohan Shivashanker

2011

To: Dean Amir Mirmiran
College of Engineering and Computing

This thesis, written by Mohan Shivashanker and entitled The Development of Hardware Multi-core Test-bed on Field Programmable Gate Array, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

Chen Liu

Jean Andrian

Gang Quan, Major Professor

Date of Defense: March 24, 2011

The thesis of Mohan Shivashanker is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Interim Dean Kevin O'Shea
University Graduate School

Florida International University, 2011

DEDICATION

I dedicate this thesis to my parents. Without their patience, understanding, support, and most of all love, the completion of this work would not have been possible.

ACKNOWLEDGMENT

I feel pleasure and privilege to express my deep sense of gratitude, indebtedness and thankfulness towards my advisor, Dr. Gang Quan, for his guidance, constant supervision and continuous inspiration and support throughout the course of work. His valuable suggestion and critical evaluation have greatly helped me in successful completion of the work. I would like to thank the members of my committee Dr. Chen Liu and Dr. Jean Andrian, for their support and patience. Their gentle but firm direction has been most appreciated. I am also thankful to all those who helped me directly or indirectly in completion of this work.

This thesis is supported in part by NSF under projects CNS-0969013, CNS-0917021 and CNS-1018108.

ABSTRACT OF THE THESIS

THE DEVELOPMENT OF HARDWARE MULTI-CORE TEST-BED ON FIELD
PROGRAMMABLE GATE ARRAY

by

Mohan Shivashanker

Florida International University, 2011

Miami, Florida

Professor Gang Quan, Major Professor

The goal of this project is to develop a flexible multi-core hardware test-bed on field programmable gate array (FPGA) that can be used to effectively validate the theoretical research on multi-core computing, especially for the power/thermal aware computing. Based on a commercial FPGA test platform, i.e. Xilinx Virtex5 XUPV5 LX110T, we develop a homogeneous multi-core test-bed with four software cores, each of which can dynamically adjust its performance using software. We also enhance the operating system support for this test platform with the development of hardware and software primitives that are useful in dealing with inter-process communication, synchronization, and scheduling for processes on multiple cores. An application based on matrix addition and multiplication on multi-core is implemented to validate the applicability of the test bed.

TABLE OF CONTENTS

CHAPTER	PAGE
1. INTRODUCTION	1
2. GENERAL FRAMEWORK.....	8
2.1 The Hardware Architecture.....	8
2.2 The System Software Setup	12
2.3 Hardware/Software Test Platform	13
3. HARDWARE ARCHITECTURE DESIGN	18
3.1 The Design of DFS	19
3.2 The Design of Sharing Memory	34
3.3 The Design of Inter-processor Communication.....	36
4. OPERATING SYSTEM SUPPORT AND SOFTWARE PLATFORM SETTINGS.....	45
4.1 Standalone Board Support Package	45
4.2 Building of Xilkernel Real-time Operating System.....	46
4.3 Scheduling.....	47
5. EXPERIMENTS AND RESULT	50
5.1 Varying the Working Frequency	50
5.2 Debugging.....	52
5.3 Four Microblazes System Using Single PLB Bus	53
5.4 Four Microblazes System Using Different PLB Buses and a Shared Console.....	55
6. CONCLUSIONS AND FUTURE WORK.....	59
LIST OF REFERENCES	60
GLOSSARY	64

LIST OF FIGURES

FIGURE	PAGE
1. Exponential Increase in Power Consumption	2
2. The Hardware Architecture	9
3. Microblaze Core Block Diagram.....	10
4. Clock Control Unit	11
5. Software Design Flow.....	13
6. Front view of Xilinx virtex-5 LX110T FPGA board.....	15
7. Back view of Xilinx virtex-5 LX110T FPGA board	16
8. Dual Microblaze Processor System.....	19
9. Schematic diagram of PLB-to-PLB Bridge	20
10. Homogeneous multi-core architecture with variable working frequency	21
11. Microblaze Cache Setup	22
12. MPMC base Configuration.....	23
13. Address Port configure for MPMC	24
14. MPMC Port and Pipeline configuration	24
15. CacheLink.vhd instantiate two FSL unit	26
16. Cache Link signal declaration in mpmc_xcl_if.v	27
17. Cache link signal assignment and instance in mpmc_xcl_if.v	28
18. Cache link signal and clock replacement of read data interface in mpmc_xcl_if.v.....	29
19. Cache link signal and clock replacement of write data interface in mpmc_xcl_if.v.....	30

20. Sample of FSL signal assignment	31
21. Port signal declaration.....	32
22. Clock Control Unit declaration in file user_logic.vhd.....	32
23. File Structure of Clock Control unit.....	33
24. Clock assignment in top.vhd	33
25. Port declaration in Configuration Unit	34
26. Logic in Configuration Unit.....	34
27. Implicit Multiprocessing.....	35
28. Explicit Multiprocessing.....	35
29. Schematic Representation of XPS Mutex connection	37
30. Schematic Representation of Matrix Addition	38
31. Loading more than one image into FLASH PROM.....	39
32. Define starting address of image in FLASH PROM.....	40
33. Reset other processor before image loading	40
34. How to enter main program after loading	40
35. Boot Program Sample	41
36. Programming FLASH memory.....	43
37. Setting RS232 serial port terminal for output.....	43
38. Overall System View	44
39. Software Platform Settings.....	45
40. Setting Compiler Options	49
41. The Timing Overhead for Varying the Working Frequencies.....	51
42. Block Diagram of Microblaze Debug Module (MDM)	53

43. Output for matrix addition and multiplication example on 4 Microblazes system.....	55
44. For 4 Microblaze system using different PLB buses and a shared console.....	57
45. System architecture view for 4 Microblaze system with single PLB bus.....	58
46. System architecture view for 4 Microblaze system with individual PLB buses connected with PLB-to-PLB Bridge and XPS Mutex	58

CHAPTER 1

INTRODUCTION

The computer industry is switching from the single core based platform to the multi-core. Die yield, limits in instruction level parallelism (ILP) and memory/processor performance gap, coupled with the exponentially increased power consumption and heat dissipation have forced this switching. Increasing the working frequency and building a more complicated single processor is no longer an effective way to improve the computing performance. Multi-core platforms, which facilitate the process or thread level parallelism and can thus work at lower clock rates, can potentially deliver high computing performance without consuming excessive power and producing prohibit heat. As a result, multi-core processor systems have been one of the most popular methods to overcome the complexities for many applications within the corporate, medical, military and other commercial markets requiring high performance and real-time processing power. For example, Cisco today embeds in its routers a network processor with 188 cores implemented in 130 nm technology, which dissipates 35W at a 250MHz clock rate, and produces an aggregate 50 billion instructions per second [1].

Power consumption and thermal management have been two of the most critical issues in developing all but the most trivial computing systems. With the demand for increased performance and decrease in size from mobile electronics to high performing game consoles, there has been an exponential increase in power density and chip temperature, according to SOC consumer stationary power consumption trends shown in Figure1.

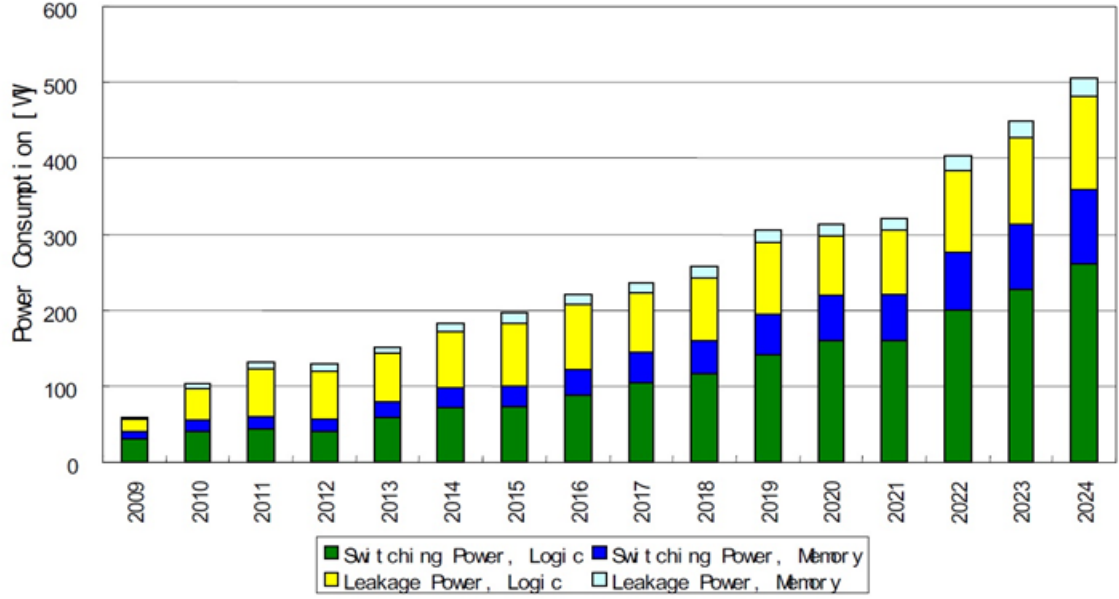


Figure 1: Exponential Increase in Power Consumption

The power consumption has been one of the most critical constraints for multi-core computing. The power consumption in modern processors consists of (i) the dynamic power and (ii) the leakage power. The dynamic power is due to the transistor switching activities during runtime, and used to be the dominant between the two. The dynamic power is given by,

$$P_{dyn} = \alpha CV^2 f$$

where, P_{dyn} is the dynamic power consumed, α is the activity factor, C is the load capacitance, V is the supply voltage and f is the working frequency.

Leakage power is due to the leakage current flowing through the transistor and is given by,

$$P_{leak} = VI_{leak}$$

where, P_{leak} is the leakage power, V is the supply voltage and I_{leak} is the leakage current through the transistor. As shown above, the dynamic power consumption is quadratically dependent on supply voltage. Thus, reducing the supply voltage can dramatically reduce the power consumption for the processor. As a result, many processors are designed to have the Dynamic Voltage and Frequency Scaling (DVFS) capability, i.e. being able to dynamically vary the supply voltage and/or frequency of a microprocessor. DVFS has proven to be one of the most effective ways to reduce the power consumption and also to manage the thermal condition of the processor while meeting the required performance [3, 10].

Besides power consumption, the rapidly elevated temperature in the computing system also raises serious concerns for computing system designers. The rise in chip temperature has significant impact on power consumption, reliability, cooling and packaging costs. In fact, the rapid increase in cooling and packaging costs has the potential to greatly affect the computer industry capability to deploy new systems. There have been extensive theoretical researches conducted on power and thermal aware computing for a multi-core platform [26, 27]. Dynamic power management techniques have already been developed to cover a wide spectrum of system characteristics [8]. It has been shown that, proactive use of software on top of the hardware can balance the overall thermal profile with minimal overhead using the operating system support. There also exists a hardware DVFS technique [7, 9] where they have proposed a novel method of producing high speed variable fractional clock rates. Y Liu et. al. [27], have presented a design time optimization technique for real-time embedded systems that make use of

DVFS to minimize peak temperature. In [26], DVFS technique is used to reduce the overall energy consumption by exploiting both static and dynamic slack times where, voltage and frequency is decreased below the operating level.

While a variety of power/thermal aware techniques have been proposed for a hardware multi-core platform, most of the existing works are validated through software simulations with simplified and idealized theoretical models. The change in computer architecture to multi-core has complicated the use of software simulators as they are difficult to parallelize well with greater number of processors [13]. Even though the SPEC2k benchmark suite and Virtutech's Simics [15] are more robust, they provide limited application programming interface (APIs) and become extremely time consuming as greater number of cores are added to the system. Furthermore, while the software simulators help to simplify the problem, results may also lead to conclusions deviated from what they really are in the practical scenarios, since the practical computing systems need to deal with more complicated scenarios that cannot be accurately modeled in the software simulation. Thus, it becomes necessary to test or verify the theoretical results in a more practical environment.

While a number of commercial platforms based on multi-core architecture are reported in the literature, such as IBM's Cell processor [1], they are not readily available at a reasonable price. Researchers also use current multi-core desktops for the validation purpose [9]. However, such a "test-bed" can only be used to test theoretical research

based on a fixed architecture. In addition, these test beds are also limited by their software supports and cannot be easily updated.

To design a versatile, flexible, and reusable multi-core test bed, we seek to develop such a test bed using the FPGA technology [4]. FPGA is an integrated circuit that can be reconfigured according to the required application by the designer. The configurable logic cell blocks are the basic logic unit in an FPGA even though the features and number of logic cells used vary from device to device. Today's FPGAs are highly scalable, field debuggable, re-configurable, have a lower cost and readily available in the market. They can be re-programmed using the powerful and versatile developing tools commercially available today. It has a much shorter design cycle and requires much less engineering equipment costs compared with those for the design of Application Specific Integrated Circuits (ASIC). The FPGA has thus been widely used in both academy and industry to build test platforms to test design alternatives and validate theoretical research results. For example, recent works on FPGA like Fort et al. [33] employed multi-threading to improve utilization of soft-core processors with little dimensional costs. The FPGA based complete system called Protoflex [15] was designed to provide similar functionality of Simics [14] is the motivation research for us to choose FPGA as our test bed.

Project goals and objectives

The goal of our project is to develop a multi-core hardware test bed on Field Programmable Gate Array (FPGA). Specifically, we want to develop a test platform that can be used effectively to validate the theoretical researches on power/thermal aware

computing. We envisioned our test platform to have the following capabilities: (1) Consisting of 4 or more homogeneous soft-core processors such as Microblaze system [12]; (2) The performance enhancement, i.e., working frequency, of each core can be dynamically varied and evaluated; (3) The multi-core platform can be supported with a real-time operating system for ease of development and testing.

Our Contributions

From our work, we have successfully developed a hardware multi-core test-bed with a real-time operating system booted on each core. The test platform consists of four homogeneous soft-core processors (i.e., Microblazes system [12]). By using a customized clock control unit in the design, we were able to dynamically vary the working frequency i.e., improve performance of each core. As discussed before, it is desirable that both the supply voltage and working frequency of a processor core can be dynamically varied. However, it would be extremely expensive if not totally impossible to change the supply voltage for a soft core on the FPGA chip. Therefore, we change the performance of the processor only by changing its working frequency. We also considered inter-processor communication between the processors to perform synchronization when accessing the shared resource. Finally, an application example of matrix addition is implemented to validate the applicability of the test-bed.

Thesis Organization:

The rest of the thesis is organized as follows. We first present the general framework of our design. In Chapter 3, we discuss the hardware aspect in our development and present

our work on the design and implementation of a multi-core system with dynamically variable frequency based on Virtex5 using FPGA development tools. Chapter 4 presents the operating system boot up and software platform settings. The experiments conducted and results are discussed in Chapter 5. We conclude the thesis in Chapter 6.

CHAPTER 2

GENERAL FRAMEWORK

The goal of this project is to develop a multi-core hardware test-bed on FPGA that can be used effectively to validate the theoretical research on power-aware computing embedded system design. The system is supposed to be flexible and support different applications. The number of processing element can be configured according to our computational needs and the chip capacity. The flexibility of FPGA has helped us in customizing necessary peripheral components.

2.1 The Hardware Architecture

Figure 2 shows the overall hardware architecture of our system. The systems consists of four homogeneous processing core. To support different clock speeds on each of the processing element units during runtime, a configurable clock control unit is developed that can be accessed through software. All four cores are connected using so called the Fast Simplex Link (FSL) [16]. Note that since each processing core can potentially work at different working frequencies, connecting all processing core using bus is not an option in our case. FSL allows asynchronous communication mechanism and is therefore selected to connect the multiple processor cores. In addition, multiple port memory controller (DDR_SDRAM) and inter-processor communication-XPS Mutex hardware IP [17] are also incorporated into our test bed to facilitate the memory sharing and inter processor synchronization and communication. Each processing element consists a Microblaze soft-core processor, a small scratch-pad memory, Local Memory Bus (LMB),

Processor Local Bus (PLB), a customized clock control IP and some specific peripheral components.

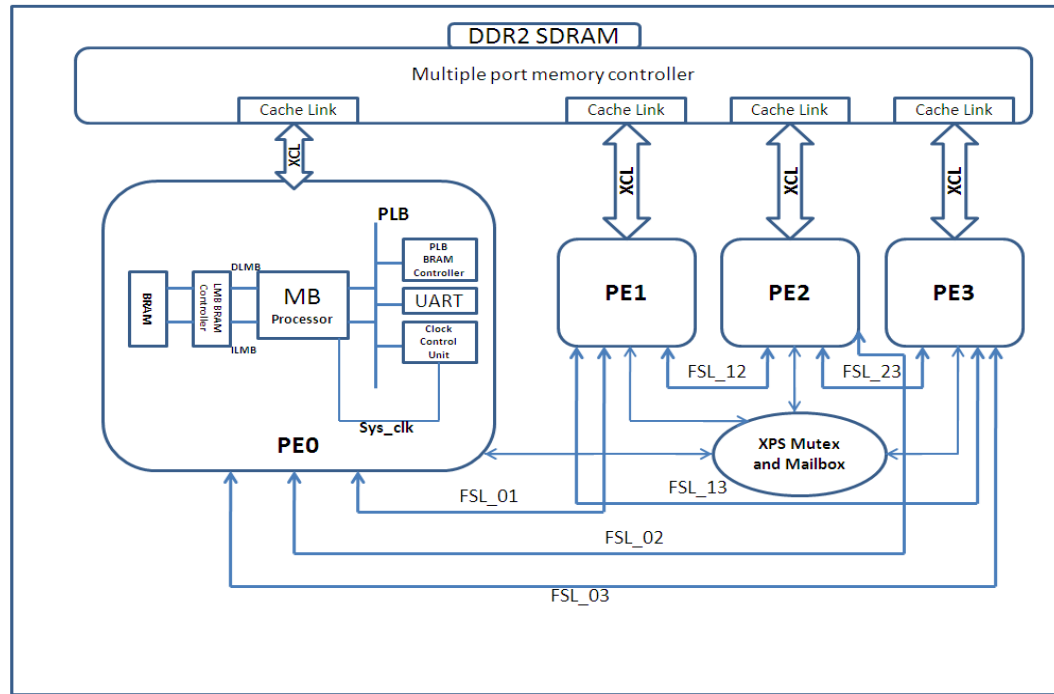


Figure 2: The Hardware Architecture

Microblaze

In our project, a Microblaze core is used as the processor core. The Microblaze soft-core processor is a 32-bit Reduced Instruction Set Computer (RISC) architecture optimized for embedded applications. Microblaze can be user configured like pipeline depth, cache size, embedded peripherals, and bus-interfaces. The Microblaze core block diagram is shown in Figure 3.

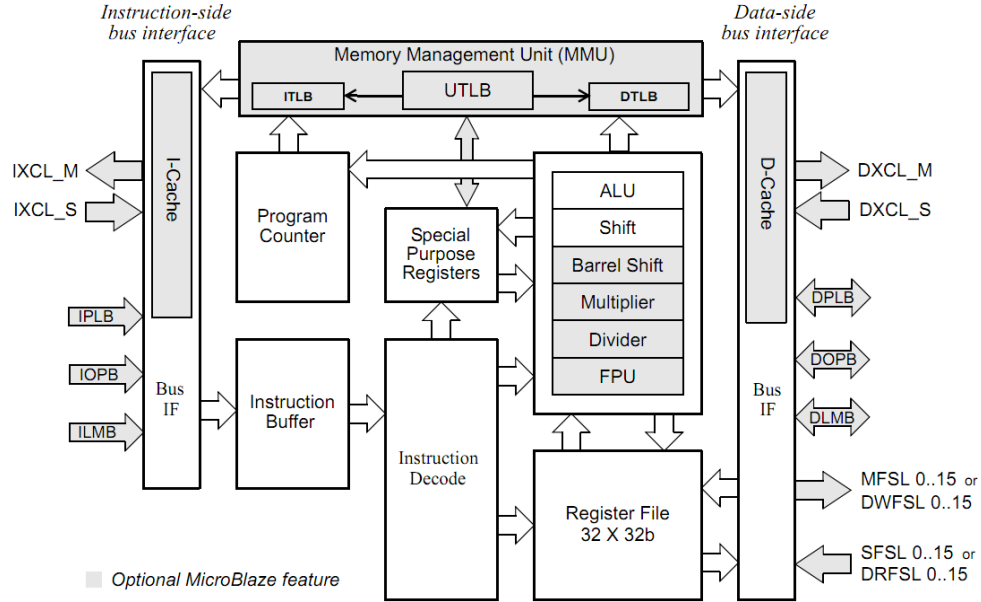


Figure 3: Microblaze Core Block Diagram [12].

The fixed feature set of the processor includes:

- Thirty-two 32-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- 32-bit address bus
- Single issue pipeline

In addition to these fixed features, the Microblaze processor is parameterized to allow selective enabling of additional functionalities like floating point arithmetic, multiplication and division.

Customized Clock Control Unit

To dynamically vary the frequency of the processors, we built a customized IP (as shown in Figure 4) that can control the clock for each processor core at run-time. Each

customized clock control IP consists of a digital clock management (DCM) unit [18] and a configuration logic unit. The Xilinx's DCM is a multi-function clock management unit which supports dynamic configuration of clock frequencies ranging from 33MHz-210MHz. For the Virtex5, the DCM unit includes a Dynamic Reconfigurable Port (DRP), which can be used by FPGA fabric to access the configuration memory within DCM.

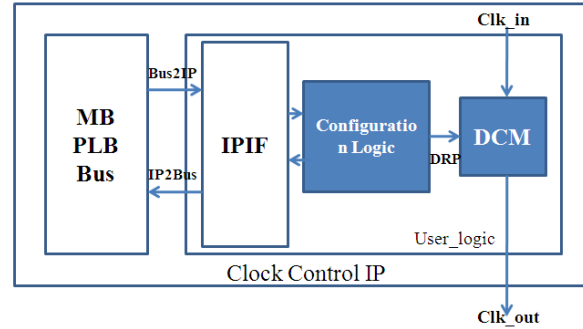


Figure 4: Clock Control Unit

The configuration logic translates the control signal from bus into control data for DCM, which makes the run-time programmable clock possible. In our design, four different clock frequencies, i.e. 40MHz, 50MHz, 66.7MHz and 100MHz, are provided and we use the two least significant bits of the bus to select the desired frequency. The customized clock control IP is connected with the PLB bus via the standard PLB-IPIF interface.

Bus Interfaces

There are a number of choices for Microblaze soft-core processor interconnections. It follows the Harvard architecture with separate paths for data and instruction accesses. Processor Local Bus (PLB) is a fully synchronous bus that provides connection to both on-chip and off-chip peripherals and memory. The Local Memory Bus (LMB) provides single-cycle access to on-chip dual-port Block RAM. The Xilinx Cache Link (XCL)

interface is intended for use with specialized external memory controllers. Memory located outside the cache area is accessed through PLB or LMB. The debug interface is used with the Microblaze Debug Module (MDM) and is controlled through JTAG port by the Xilinx Microprocessor Debugger (XMD).

For our design, even though the traditional bus connection is possible, it is less attractive due to scalability concern. An alternative is to use the logic source in FPGA to create the Network-On-Chip (NOC) infrastructure. For example, Schelle and Grunwald [19] implemented a switching network as interconnection for general purpose processor in a Virtex II-pro device. One major disadvantage of this solution is the large amount of resources it requires. Henceforth, we made use of a convenient point-to-point connection mechanism, i.e., the Fast Simplex Link (FSL) bus, provided by Xilinx. FSL is a FIFO-based connection and can be synchronous or asynchronous. An asynchronised communication scheme is particularly useful in our design with different processors running at different speeds. Each core from Xilinx supports multiple FSL buses. For example, a Microblaze has up to sixteen FSL ports with one master and one slave interface to connect up to sixteen different components for duplex communication, which makes it reasonably easy and effective to build popular multi-core topologies, such as the tree, mesh, or torus structure.

2.2 The System Software Setup

In our project, we made use of software platform settings under XPS GUI to boot the Xilkernel RTOS into the board and configured the operating system and library files

according to our requirements. C code was implemented and compiled through RTOS, as shown in Figure 5 below.

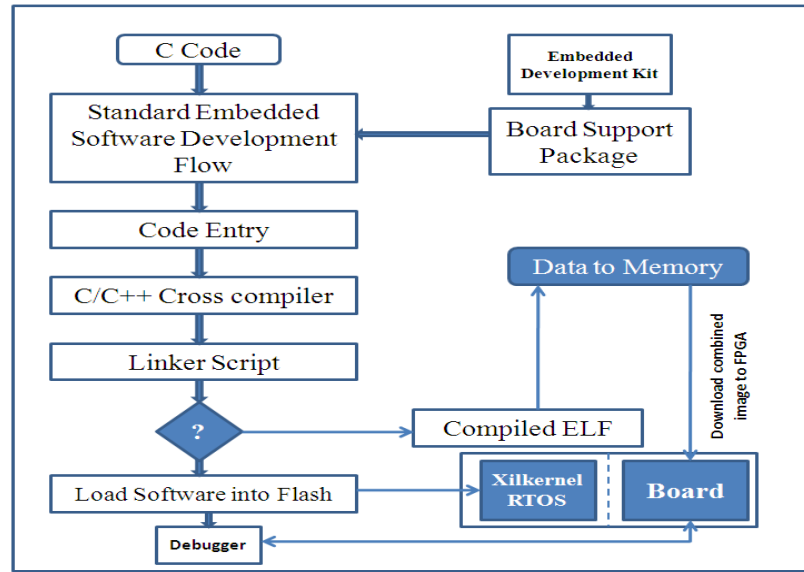


Figure 5: Software Design Flow

2.3 The Hardware/Software Test Platform

In order to develop the embedded system design on the chip, Xilinx has provided the Embedded Development Kit (EDK) suite. We used EDK design suite 10.1.03 version. Using this suite of tools we can incorporate a wide range of Hard and soft-IP cores, such as microprocessors, interconnects, memories, and an assortment of peripherals. The main advantage of EDK suite is that on a single environment we can perform design, simulation, synthesis, and compilation.

In EDK embedded system design, we have two separate steps, hardware and the software design which interact each other. Firstly, we have to develop and design the hardware part where a custom circuitry IP core is developed using a hardware description language

(HDL) like verilog or VHDL. Synthesis tool translates this hardware description language into the low level gate logic. Then, all available IP cores are combined into a single design using the Xilinx platform studio tool. The microprocessor is connected to all the IP cores using the bus architecture, thus allowing the entire design to be controlled using software programs. Secondly, in order to control the microprocessor and all connected peripheral, application software is developed where a system programmer should correctly implement the low level details of interacting with any given peripheral. We can install the embedded operating system on the chip design, which helps in the development of our design.

Xilinx Embedded Development Kit (EDK) is the package for building Microblaze in Xilinx FPGAs. It consists of two separate environments: Xilinx Platform Studio (XPS) and Software Development Kit (SDK). As said earlier, we have used Xilinx virtex-5 LX110T FPGA board for our project. Below are the figures (Figure 6 and 7) showing the front and back view of virtex-5 LX110T FPGA.

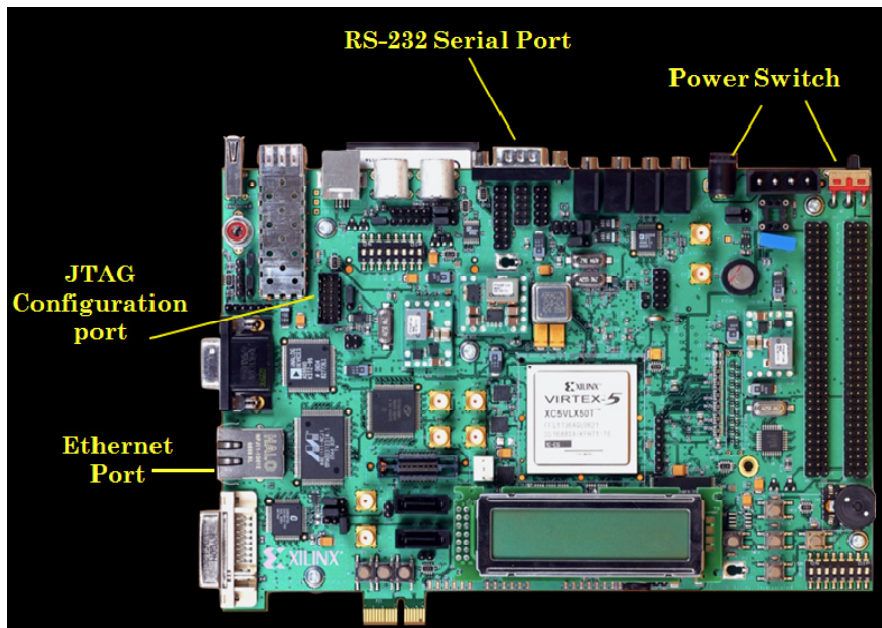


Figure 6: Front view of Xilinx virtex-5 LX110T FPGA board

The main features of this board are:

- Xilinx XCF32P Platform Flash PROMs (32 MB each) for storing large device configurations
- Xilinx System ACE Compact Flash configuration controller
- 64-bit wide 256Mbyte DDR2 small outline DIMM (SODIMM) module compatible with EDK supported IP and software drivers
- On-board 32-bit ZBT synchronous SRAM and Intel P30 Strata Flash
- 10/100/1000 tri-speed Ethernet PHY supporting MII, GMII, RGMII, and SGMII interfaces
- USB host and peripheral controllers
- Programmable system clock generator
- Stereo AC97 codec with line in, line out, headphone, microphone, and SPDIF digital audio jacks

- RS-232 port, 16x2 character LCD, and many other I/O devices and ports.

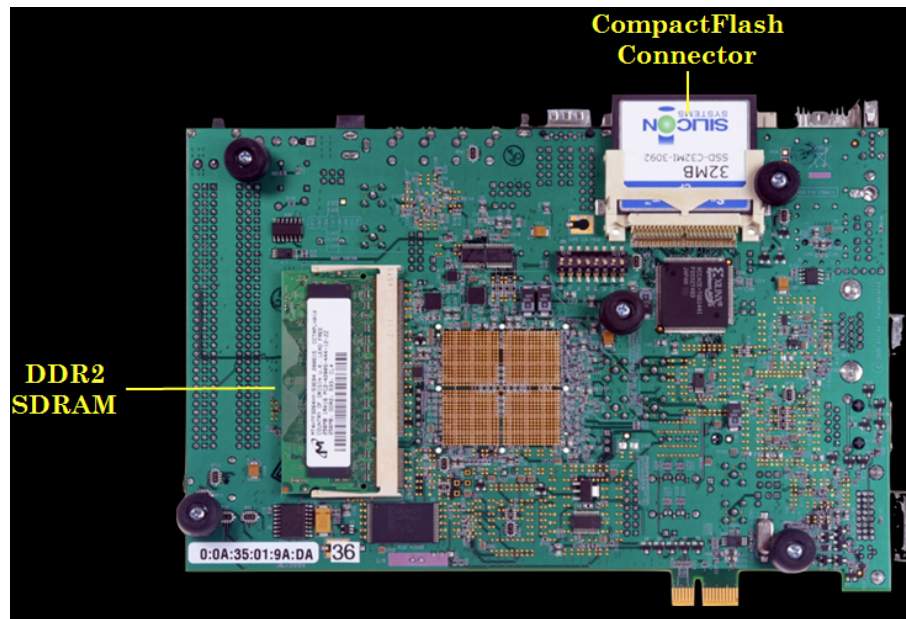


Figure 7: Back view of Xilinx virtex-5 LX110T FPGA board

Once we have completed the embedded design, it is translated into an implementation suitable to the board. Here we have two phases for the implementation process: (a) Synthesis phase and (b) Compilation phase. Under the synthesis phase, a Xilinx synthesis tool will translate the hardware description language into a gate level description. EDK provides Xilinx Synthesis Technology (XST) the following, which happens in the synthesis phase: (1) System-on-Chip elaboration: It translates the HDL into a computer readable format. (2) Soft-IP core synthesis: Converts soft-IP core into a Net list which is a logical circuit description. (3) Physical Mapping: Maps the low-level Net list into a physical description circuit. (4) Net list placement and routing: Physical Net lists are placed into the FPGA and channel is established between the different components for

communication. (5) Bit-stream generation: Physical design is converted into a bit-level description i.e., bit-stream files which can be downloaded into the board for execution. In the compilation phase, the software program design is compiled from the C source and converted into the binary format used by the microprocessor. After the software has been compiled into a binary executable file, the hardware and software are combined into one overall bit-stream. This bit-stream is used to initialize the SRAM with the hardware design, and the chip memories with the software program. Thus, when the system-on-chip is boot-strapped, any microprocessors in the system will execute the software associated with them.

CHAPTER 3

HARDWARE ARCHITECTURE DESIGN

As explained in the previous chapter, we have used FPGA board from Xilinx. The board is Xilinx Virtex5 XUPV5-LX110T evaluation platform [20]. The chip is xc5vlx110t, grade ff1136, speed -1. In order to develop the embedded system design on the chip, Xilinx has provided the Embedded Development Kit (EDK) [21] and an Integrated synthesis environment (ISE) software design suite [23]. Using this suite of tools, we can incorporate a wide range of hard and a soft-IP core, such as microprocessors, interconnects, memories, and an assortment of peripherals. The main advantage of this design suite is that on a single environment, we can perform design, simulation, synthesis, place and routing and finally, download and debug the system.

Xilinx Intellectual Property (IP) is the building block of several Xilinx design platforms. Various IP cores are available to address requirements of FPGA designers in Digital Signal Processing (DSP), embedded and other connectivity application designs. The entire used IPs version is specified in the following sections. Xilinx keep updating IPs regularly and are available on their website but, these new IP versions have no guarantee to work in this project.

We started with a single Microblaze processor design. The local BRAM is connected and used to instantiate the processor with an instruction and data BRAM controllers. The Microblaze is sitting on the PLB. The next processor was added to the system from the IP

catalog and provided with connections of different BRAM and BRAM controllers similar to the first processor. A block diagram of the dual processor system is shown in Figure 8.

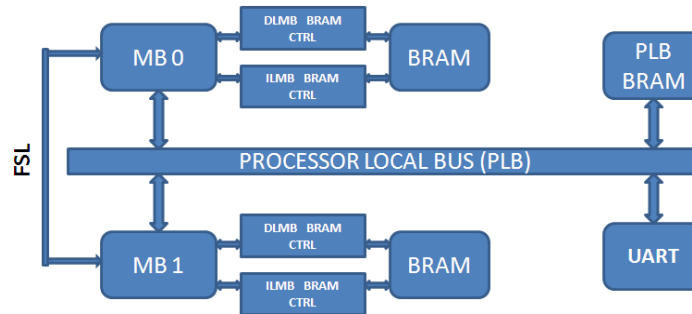


Figure 8: Dual Microblaze Processor System

The PLB BRAM acts as a shared memory for the two processors. Any address on the PLB is accessible by both the Microblazes. Since Microblaze uses memory mapped I/O, both of them can access any resource on the PLB bus. The XPS Mutex can also be used in place of FSL for inter-processor communication between the processors.

After the hardware platform design is complete, FPGA configuration bitstream is generated. Xilinx Platform Studio (XPS) is used to build the net list and bitstream file. Then, the software component is set with a downloadable Executable Linked Format (ELF) file and merged with the hardware bitsream to dynamically download onto the board via JTAG cable connected to the FPGA.

3.1 The Design of DFS

To accommodate more numbers of Microblaze processors and to improve the system performance, we made use of PLB bus for each individual processor and connected them using PLB-to-PLB Bridge. The Xilinx PLB-to-PLB Bridge design allows the user to

tailor the bridge according to a specific application by setting certain parameters to enable/disable features. The PLB-to-PLB Bridge is a slave on the primary and master on the secondary PLB. We can isolate some of the slow PLB peripheral from the primary PLB and improve the system performance. The bridge allows the Microblazes to access the external DDR2SDRAM memory and other peripherals on different buses by mapping to their address space.

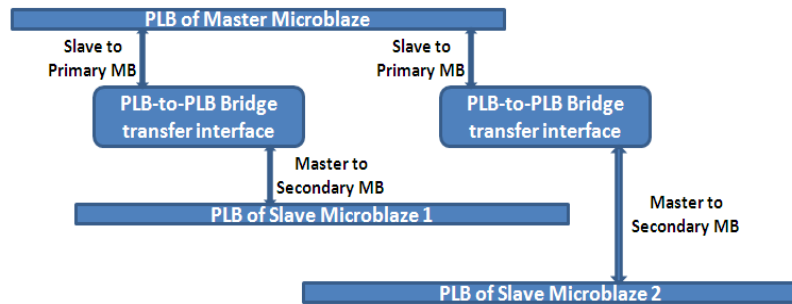


Figure 9: Schematic diagram of PLB-to-PLB Bridge

Further, a hardware test-bed with four Microblaze homogeneous soft-cores on a same chip of FPGA and point-to-point network topology was built (as shown in Figure 10). Each core was made to dynamically change its working frequency using the clock control unit (CCU) based on Xilinx's Digital Clock Manager (DCM) [18]. A clock generator module is used to initiate the DCM, where the clock control unit translates the control into the DCM and generates the desired clock. The configuration and customized units for the experiment to test the multi-core test bed is explained below.

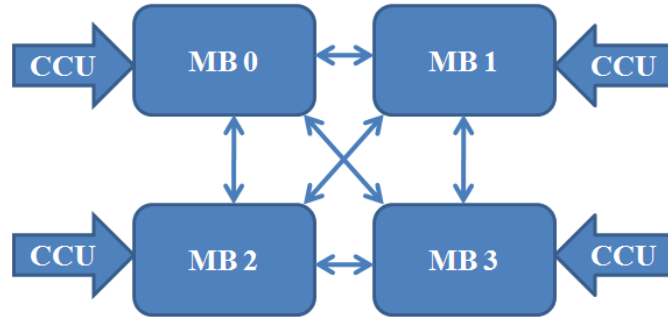


Figure 10: Homogeneous multi-core architecture with variable working frequency

Configuration and Customized Units

Xilinx provides two types of processor units, hard-core PowerPC and soft-core Microblaze. The so-called hard-core unit is basically a silicon unit integrated into FPGA. The advantage of hard-core is the speed, but it lacks the flexibility of a soft-core unit. The soft-core is an integrated logic design based on FPGA fabric, which make Microblaze much more flexible. The Virtex5 board supports only microblaze processor with speed of 125MHz. With soft-core, you can add an arbitrary number of processors, as long as the FPGA chip has enough capacity. Besides, each processing unit could be tailored according to need. For Microblaze, you can specify the number of the pipeline stages, add or delete the float point unit, change the bus interface, make tradeoff between space and throughput, and so on.

The detailed configuration for Microblaze (7.10.d) is explained as follows:

Bus Interface: Setting from BUS Interface Tab. It can also be seen and changed from MHS file. Remember to name the 3 FSL master and slave channels.

Port Interface: Change the clock to the one you are going to drive the processor. By default, it is all sys_clk_s.

Instruction Tab-Optimization (Select implementation to optimize area): This option could make tradeoff between area and processor throughput. Enable this option when you need to reduce the logic used by Microblaze.

Exception: Enable both data and instruction side PLB exceptions, enable illegal instruction exception and enable unaligned data exception

Cache: Due to the current bus connection, you must enable cache to use XCL interface. Enable both Instruction and Data caches. Also enable the option “Use cache links for all D/I cache memory access”. Because we are going to modify the XCL interface in MPMC in order to support different clocks. All the memory access must go through this XCL interface, otherwise system fails. The cache tab configuration is listed as below.

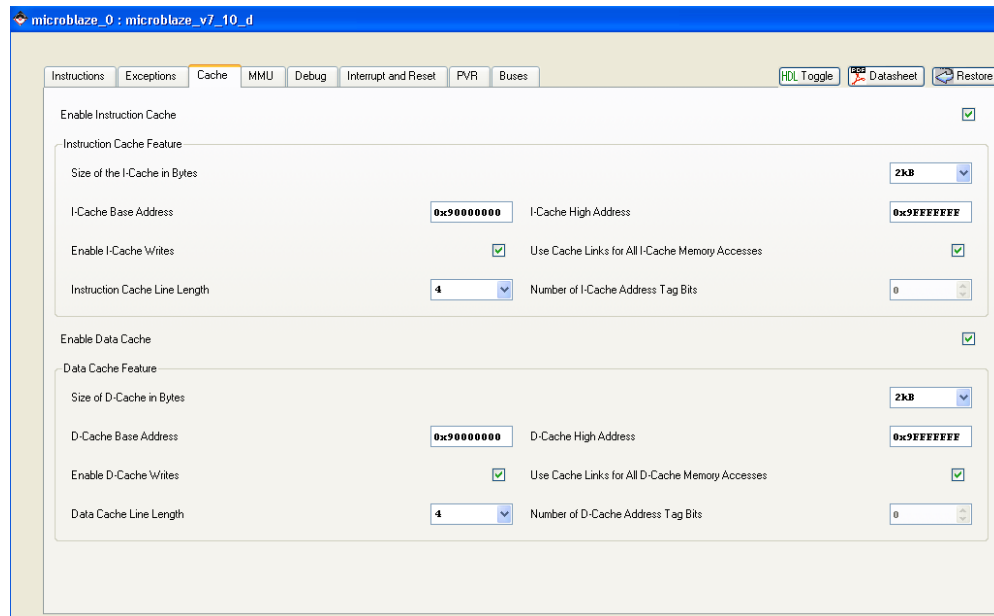


Figure 11: Microblaze Cache Setup

Buses: Choose the appropriate number of FSL for the example project.

Multiple-port Memory Controller (MPMC) 4.03.a version: DDR2_SDRAM

Memory control unit manages the DDR2_SDRAM memory and provides multiple ports access for processors. The basic configuration of MPMC is as follows:

Bus interface: Rename the XCL interface so that each one connected to a Microblaze XCL bus.

Port interface: The basic ports configuration is based on the board setting. Their corresponding user constraints are defined in the system.ucf file. The memory unit needs a 200MHz clock to enable the DDR2. This unit also requires a 100MHz clock with a 90 degree shift, which can be generated by module Clock Generator.

Base Configuration: Set the all the ports interface to be XCL.

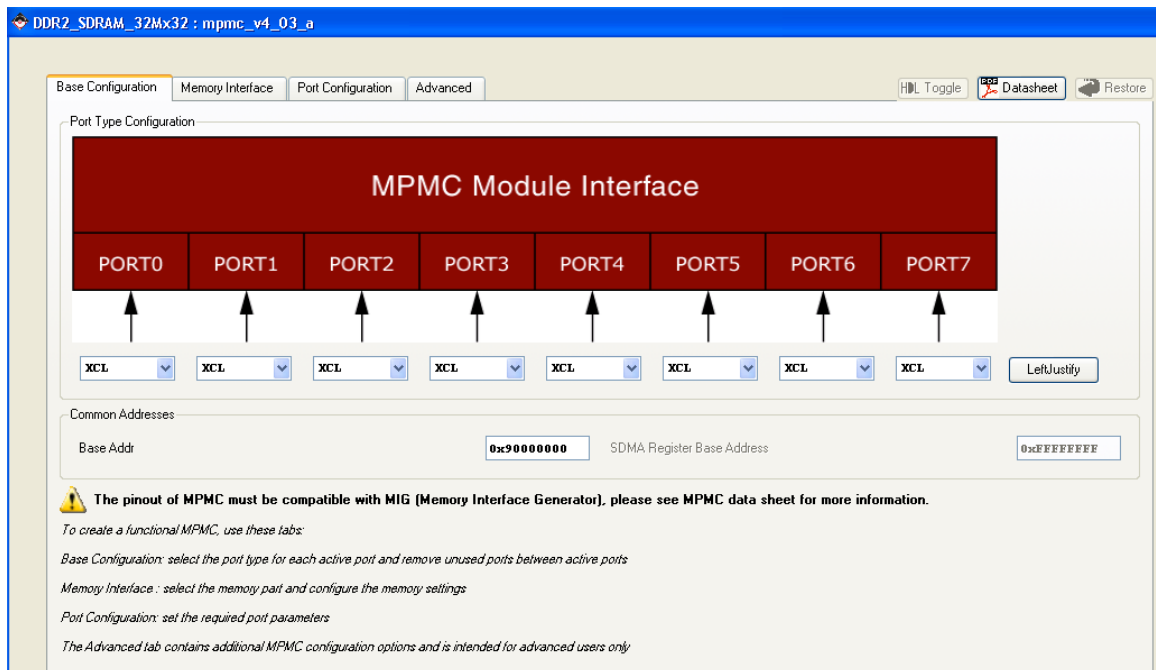


Figure 12: MPMC base Configuration

Memory Interface Tab: Choose the right part number: MT4HTF3264H-53E

Port Configuration Tab: XCL port is left default. Choose common Port Address for easy access.

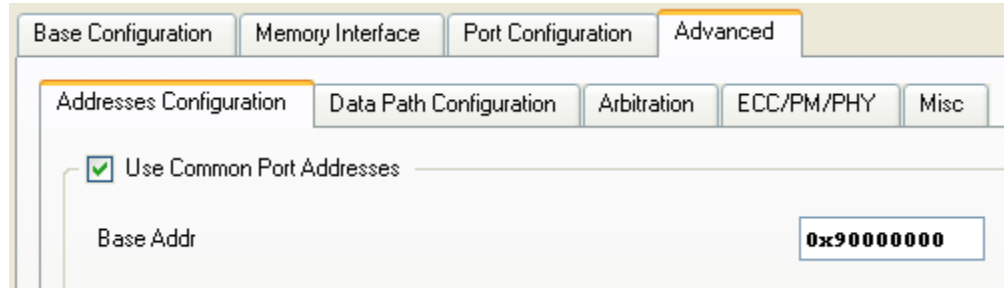


Figure 13: Address Port configure for MPMC

Advanced Tab: Set BRAM as the FIFO configuration for each Port.

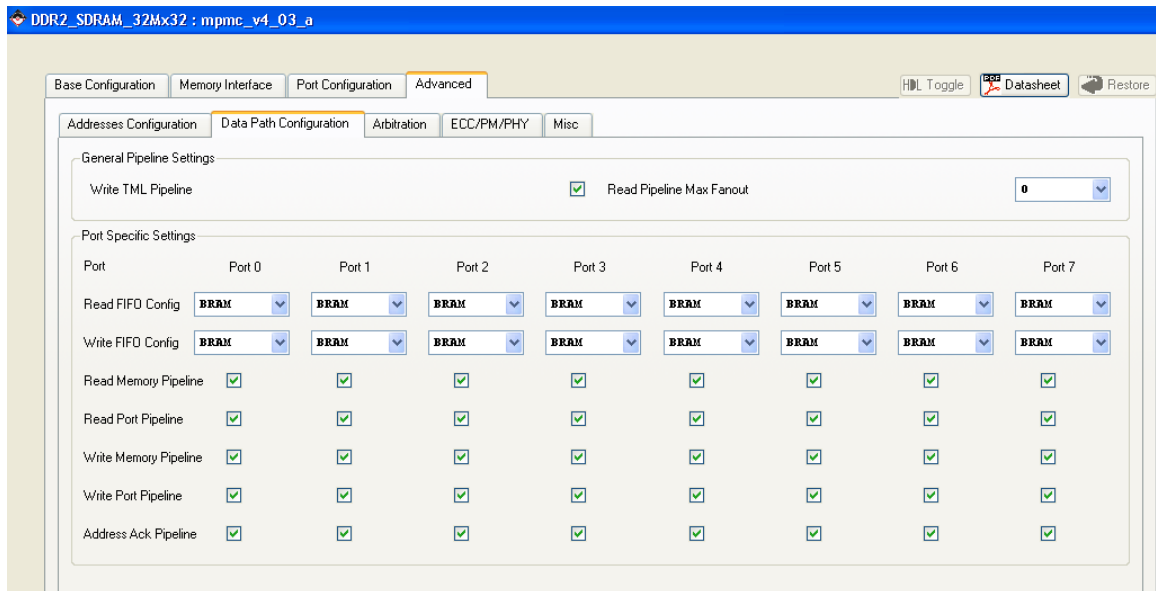


Figure 14: MPMC Port and Pipeline configuration

In vhd, we need to add source files from FSL. The added source files and structure are listed as below:

cachelink.vhd

```
|-----fsl_v20.vhd  
    |-----async_fifo.vhd  
    |----- async_fifo_bram.vhd  
    |-----gen_srlfifo.vhd  
    |-----gen_sync_bram.vhd  
    |-----gen_sync_dpram.vhd  
    |-----sync_fio.vhd
```

As you can see, cachelink.vhd is the top file of the added sources. You can view the source by opening file cachelink.vhd. It can be treated as a wrapper. It did not add any logic into the system. The only function it performs is creating two instance of FSL unit. Because FSL is unidirectional, you need one pair of FSL to communicate in duplex mode.

```

239 fsl_0 : fsl_v20
240 generic map(
241     C_EXT_RESET_HIGH      => 1,--Active HIGH
242     C_ASYNC_CLKS          => 1,--Async
243     C_IMPL_STYLE          => 0,--Using LUT RAM
244     C_USE_CONTROL          => 1,--Enable control bit
245     C_FSL_DWIDTH          => 32,
246     C_FSL_DEPTH           => 16,
247     C_READ_CLOCK_PERIOD   => 10 )
248 port map (
249     -- Clock and reset signals
250     FSL_Clk => FSL_A_clk,
251     SYS_Rst => SYS_A_clk,
252     FSL_Rst => FSL_O_Rst,
253
254     -- FSL master signals
255     FSL_M_Clk      => FSL_A_M_CLK,
256     FSL_M_Data     => FSL_A_M_DATA,
257     FSL_M_Control  => FSL_A_M_Control,
258     FSL_M_Write    => FSL_A_M_Write,
259     FSL_M_Full     => FSL_A_M_Full,
260
261     -- FSL slave signals
262     FSL_S_Clk      => FSL_B_S_Clk,
263     FSL_S_Data     => FSL_B_S_Data,
264     FSL_S_Control  => FSL_B_S_Control,
265     FSL_S_Read     => FSL_B_S_Read,
266     FSL_S_Exists   => FSL_B_S_Exists,
267
268     -- FIFO status signals
269     FSL_Full       => FSL_O_Full,
270     FSL_Has_Data   => FSL_O_Has_Data,
271     FSL_Control_IRQ => FSL_O_Control_IRQ );
272
273
274
275 fsl_1 : fsl_v20
276 generic map(
277     C_EXT_RESET_HIGH      => 1,--Active HIGH
278     C_ASYNC_CLKS          => 1,--Async
279     C_IMPL_STYLE          => 0,--Using LUT RAM
280     C_USE_CONTROL          => 1,--Enable control bit
281     C_FSL_DWIDTH          => 32,
282     C_FSL_DEPTH           => 16,
283     C_READ_CLOCK_PERIOD   => 10 )
284 port map (
285     -- Clock and reset signals
286     FSL_Clk => FSL_B_clk,
287     SYS_Rst => SYS_B_clk,
288     FSL_Rst => FSL_1_Rst,

```

Figure 15: CacheLink.vhd instantiate two FSL unit

Fsl_v20 and others file are copied from Xilinx's fsl_v2.11.a. The entire source files can be found from Xilinx ISE's IP library. For the verilog, all you need to take care is the file mpmc_xcl_if.v (If you want to create your own mpmc_xcl_if.v, be sure to change the file

property to write/read from read only). This file provides the XCL interface to the external memory access. The original interface conforms to Xilinx's FSL bus interface. The inserted cachelink.vhd unit also conforms to FSL bus interface. The only difference lies in the clock because; each side is synchronized to different clock. In simple way, you can deem the cache link as an asynchronous FIFO with XCL interface.

At first, you need to introduce some intermediate signal between the inserted unit and the original FSL interface as shown below.

```

270
271 ////////////////////////////////////////////////////
272 //Declare of cachelink intermediate wires
273 ////////////////////////////////////////////////////
274
275 //Signal for FSL bus internal interface
276 wire      FSL_A_S_Clk;          //
277 wire [0:31] FSL_A_S_Data;       // (0-31)
278 wire      FSL_A_S_Control;     //
279 wire      FSL_A_S_Read;        //
280 wire      FSL_A_S_Exists;      //
281 wire      FSL_A_M_Clk;         //
282 wire [0:31] FSL_A_M_Data;       // (0-31)
283 wire      FSL_A_M_Control;     //
284 wire      FSL_A_M_Write;       //
285 wire      FSL_A_M_Full;        //
286 //Internal FSL signal
287 wire      FSL_int_S_Clk;        //
288 wire [0:31] FSL_int_S_Data;     // (0-31)
289 wire      FSL_int_S_Control;    //
290 wire      FSL_int_S_Read;       //
291 wire      FSL_int_S_Exists;     //
292 wire      FSL_int_M_Clk;        //
293 wire [0:31] FSL_int_M_Data;     // (0-31)
294 wire      FSL_int_M_Control;    //
295 wire      FSL_int_M_Write;      //
296 wire      FSL_int_M_Full;       //
297

```

Figure 16: Cache Link signal declaration in mpmc_xcl_if.v

Second, assign intermediate signals and instantiate the cache link unit.

```

301
302 assign FSL_A_M_Data = FSL_int_S_Data;
303 assign FSL_A_M_Control = FSL_int_S_Control;
304 assign FSL_int_M_Data = FSL_A_S_Data;
305 assign FSL_int_M_Control = FSL_A_S_Control;
306
307 assign FSL_int_M_Write = FSL_A_S_Exists && (!FSL_int_M_Full);
308 assign FSL_A_S_Read = FSL_A_S_Exists && (!FSL_int_M_Full);
309
310 assign FSL_A_M_Write = (! FSL_A_M_Full) && FSL_int_S_Exists ;
311 assign FSL_int_S_Read = (! FSL_A_M_Full) && FSL_int_S_Exists ;
312
313 // assign FSL_A_S_Clk = Clk;
314 // assign FSL_A_M_Clk = Clk;
315 //Change Clk to Clk_MPMC
316 assign FSL_A_S_Clk = Clk_MPMC;
317 assign FSL_A_M_Clk = Clk_MPMC;
318
319 cachelink #(
320 .C_BASEADDR(C_BASEADDR),
321 .C_HIGHADDR(C_HIGHADDR)
322 )
323 xcl_cachelink(
324     //FSL_A_Clk(Clk),           //I
325     //Change Clk to Clk_MPMC
326     .FSL_A_Clk(Clk_MPMC),           //I
327     .FSL_A_Rst(Rst),           //I
328     //FSL_A_S_Clk(Clk),           //I
329     //Change Clk to Clk_MPMC
330     .FSL_A_S_Clk(Clk_MPMC),           //I
331     .FSL_A_S_Data(FSL_A_S_Data),           //O (0-31)
332     .FSL_A_S_Control(FSL_A_S_Control),           //O
333     .FSL_A_S_Read(FSL_A_S_Read),           //I
334     .FSL_A_S_Exists(FSL_A_S_Exists),           //O
335     .FSL_A_M_Clk(FSL_A_S_Clk),           //I
336     .FSL_A_M_Data(FSL_A_M_Data),           //I (0-31)
337     .FSL_A_M_Control(FSL_A_M_Control),           //I
338     .FSL_A_M_Write(FSL_A_M_Write),           //I
339     .FSL_A_M_Full(FSL_A_M_Full),           //O
340
341     .FSL_B_Clk(Read_Data_FSL_S_Clk),           //I
342     .FSL_B_Rst(Rst),           //I
343     .FSL_B_S_Clk(Read_Data_FSL_S_Clk),           //I
344     .FSL_B_S_Data(Read_Data_FSL_S_Data),           //O (0-31)
345     .FSL_B_S_Control(Read_Data_FSL_S_Control),           //O
346     .FSL_B_S_Read(Read_Data_FSL_S_Read),           //I
347     .FSL_B_S_Exists(Read_Data_FSL_S_Exists),           //O

```

Figure 17: Cache link signal assignment and instance in mpmc_xcl_if.v

Third, modify the original read interface. Replace old signals with the new one from cache link. In addition, modify the original write interface. Note the clock signal need to be replaced carefully. Assigning a wrong clock signals will make the system very difficult to debug.

```

360 //////////////////////////////////////////////////
361 // Handle Read Data Path Signals
362 //////////////////////////////////////////////////
363 xcl_read_data #(
364     .C_PI_DATA_WIDTH      (C_PI_DATA_WIDTH),
365     .C_PI_RDWDADDR_WIDTH  (C_PI_RDWDADDR_WIDTH),
366     .C_PI_RDDATA_DELAY    (C_PI_RDDATA_DELAY),
367     .C_LINESIZE            (C_LINESIZE),
368     .C_MEM_SDR_DATA_WIDTH (C_MEM_SDR_DATA_WIDTH),
369     .C_READ_FIFO_PIPE     (P_READ_FIFO_PIPE),
370     .C_RDFIFO_EMPTY_PIPE  (P_RDFIFO_EMPTY_PIPE)
371 )
372 xcl_read_data_0 (
373     // .Clk              (Clk),                // I
374     // Change Clk to Clk_MPMC
375     .Clk(Clk_MPMC),                // I
376     .Clk_MPMC      (Clk_MPMC),      // I
377     .Rst            (Rst),           // I
378     .Clk_PI_Enable  (clk_pi_enable), // I
379     .PI_RdFIFO_Data (PI_RdFIFO_Data), // I [C_PI_DATA_WIDTH-1:0]
380     .PI_RdFIFO_Pop  (PI_RdFIFO_Pop), // O
381     .PI_RdFIFO_RdWdAddr (PI_RdFIFO_RdWdAddr), // I [C_PI_RDWDADDR_WIDTH-1:0]
382     .PI_RdFIFO_Empty (PI_RdFIFO_Empty), // I
383     .PI_RdFIFO_Flush (PI_RdFIFO_Flush), // O
384     .Target_Word     (access_data[26:29]), // I
385     // Comment the old port mapping
386     // .Read_Data_Exists (Read_Data_FSL_S_Exists), // O
387     // .Read_Data_Control (Read_Data_FSL_S_Control), // O
388     // .Read_Data         (Read_Data_FSL_S_Data), // O
389     // .Read_Data_Read    (Read_Data_FSL_S_Read), // I
390     // Start of new mapping
391     .Read_Data_Exists(FSL_int_S_Exists), // O
392     .Read_Data_Control(FSL_int_S_Control), // O
393     .Read_Data(FSL_int_S_Data), // O
394     .Read_Data_Read(FSL_int_S_Read), // I
395     .Read_Start      (read_start), // I
396     .Read_Done        (read_done), // O
397 );
398

```

Figure 18: Cache link signal and clock replacement of read data interface in mpmc_xcl_if.v

```

399 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
400 // Instantiate FSL FIFOs
401 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
402
403 // Access FSL FIFO
404 SRL16E access_fifo[0:32] (
405     //CLK(Clk),
406     //Change Clk to Clk_MPMC
407     .CLK(Clk_MPMC), //I
408     // .CE(Access_FSL_M_Write),
409     // .D({Access_FSL_M_Control, Access_FSL_M_Data}),
410     //Start of change
411     .CE(FSL_int_M_Write),
412     .D({FSL_int_M_Control, FSL_int_M_Data}),
413     //end of change
414     .A0(access_raddr[0]),
415     .A1(access_raddr[1]),
416     .A2(access_raddr[2]),
417     .A3(access_raddr[3]),
418     .Q({access_control_i, access_data_i})
419 );
420
421 // Access FSL read counter
422 mpmc_rdcntr access_raddr_cntr (
423     //rclk(Clk),
424     //Change Clk to Clk_MPMC
425     .rclk(Clk_MPMC), //I
426     .rst(Rst),
427     .ren(access_ren_i),
428     // .wen(Access_FSL_M_Write),
429     //Change
430     .wen(FSL_int_M_Write),
431     .raddr(access_raddr),
432     // .full(Access_FSL_M_Full),
433     //Change
434     .full(FSL_int_M_Full),
435     .exists(access_exists_i)
436 );
437

```

Figure 19: Cache link signal and clock replacement of write data interface in mpmc_xcl_if.v

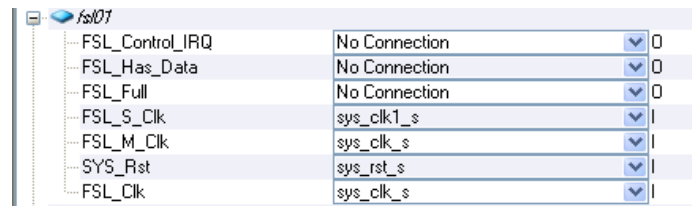
If all these source files are changed for particular experiment, we need to add these source file in the pao files. (The .pao file is introduced in [Platform Specification Format Reference Manual](#), which can be found at http://www.xilinx.com/support/documentation/sw_manuals/edk63i_psf_rm.pdf. Besides, Xilinx's XAPP 967 gives the detail to create an IP, which is available at http://www.xilinx.com/support/documentation/application_notes/xapp967.pdf). The declaration in pao file will help the parser recognize these files and compile them before use.

Fast Simplex Link (FSL)

As explained in the previous chapter, FSL is a fast connection between 2 processors. It is a uni-direction channel, so you should use a pair for duplex communication. It supports asynchronous communication between two clock domains. The base version is 2.10.a. The last version 2.11.a would cause some implementation error, so we stick to the older version. Any later version should be tested before you integrated it into the system.

Base configuration: Set this unit run in asynchronous mode. The depth of FIFO can also be configured from 1 to 32. You can tailor the depth according to communication need.

Port Configuration: In each FSL channel, you should assign a master clock, FSL_M_CLK, to transmitter end and a slave clock, FSL_S_CLK, to the receiver. There is another clock FSL_CLK, which only used in synchronous communication. You can ignore it in this project. There are some other handshake signals in the port section. For their usage, you can refer to FSL datasheet. Here is an example:



fsl07		
FSL_Control_IRQ	No Connection	0
FSL_Has_Data	No Connection	0
FSL_Full	No Connection	0
FSL_S_Clk	sys_clk1_s	1
FSL_M_Clk	sys_clk_s	1
SYS_Rst	sys_rst_s	1
FSL_Clk	sys_clk_s	1

Figure 20: Sample of FSL signal assignment

Clock Control Unit

The clock unit is based on Xilinx's dynamic programmable Digital Clock Manager (DCM). The detail of the clock generation can be found in the DCM datasheet and manual. The data interface is PLB. Clock control unit has a PLB slave interface. The processor can program this unit via PLB bus. The function of this unit is to translate the

control word, and write the control into DCM unit, and generate the desired clock. There is no need to configure this unit. The customization work is given in the following.

There are two version of clk_control. In /ProjectName/pcore, clk_control_v2_01_a and clk_control_v3_01_a. The former one is obsolete unit. We only used the latter unit. The user ports are declared in mpd file, which can be found in /ProjectName/pcore/clk_control_v3_01_a/data.

```

37  ## Ports
38  PORT clk_in = "", DIR = I
39  PORT clk0 = "", DIR = O
40  PORT clk90 = "", DIR = O
41  PORT clk2x = "", DIR = O
42  PORT clk_out = "", DIR = O
43  PORT lock = "", DIR = O
44  PORT mux = "", DIR = O, VEC = [1:0]

```

Figure 21: Port signal declaration

In file user_logic.vhd, the design unit is declared like this.

```

--USER signal declarations added here, as needed for user logic

signal mux_i : std_logic_vector(13 downto 0);
component top is
  Port ( clk_in : in STD_LOGIC;
        --comment clk sam
        --clk_sam : out std_logic;
        clk0 : out std_logic;
        clk90 : out std_logic;
        clk2x : out std_logic;
        clk_out : out std_logic;
        rst : in STD_LOGIC;
        mux : in std_logic_vector(13 downto 0);
        lock : out STD_LOGIC);
end component;

```

Figure 22: Clock Control Unit declaration in file user_logic.vhd

For the design source file, you can see files like this. The clk_control.vhd and user_logic.vhd are generated by system template. The design top unit, top.vhd is instantiated in user_logic.vhd. The top.vhd calls drp_control.vhd for logic control, dcm1.vhd for clock generation. The sample.vhd is for debug purpose.

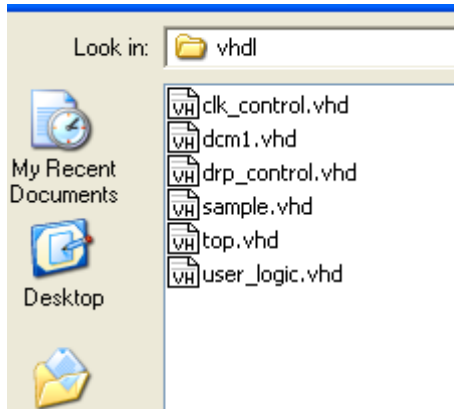


Figure 23: File Structure of Clock Control unit

The decode logic is included in top.vhd and the following code specify the desired clock. You can change it to other clock frequency or you can modify them to support more clock choices.

```

219 process(clkin)
220 begin
221   if(clkin'event and clkin='1') then
222     if mux_reg /= mux then
223       start <= '1';
224       case mux is
225         when "01" => multiply<="00001"; divide<="00100"; --40M
226         when "10" => multiply<="00001"; divide<="00011"; --50M
227         when "11" => multiply<="00001"; divide<="00010"; --66.7M
228         --when "00" => multiply<="00001"; divide<="00101"; --33.3M
229         when others => multiply<="00001"; divide<="00001"; --100M
230       end case;
231     else
232       start <= '0';
233     end if;
234   end if;
235 end process;

```

Figure 24: Clock assignment in top.vhd

Configuration Unit

This is also a simple customized unit to facilitate the multiple processor programs loading. The purpose is to enable the main processor (program loading processor) to reset the other processor when the program loading from flash to DDR2 is done. The added ports are declared at the beginning of user_logic.vhd.

```

98  (
99  -- ADD USER PORTS BELOW THIS LINE -----
100 --USER ports added here
101 -- ADD USER PORTS ABOVE THIS LINE -----
102 mb_reset                : in std_logic;--input MicroBlaze reset
103 mbctrl_rst              : out std_logic;-- output controlled MicroBlaze reset
104 -- DO NOT EDIT BELOW THIS LINE -----

```

Figure 25: Port declaration in Configuration Unit

The basic logic is shown in the following figure. The mbctrl_rst is the output reset signal. This unit controls the reset signal by the data (Bus2IP_Data) from main processor. In this way, the main Microblaze can reset the other processors and release them after program loading.

```

143 begin
144
145 --USER logic implementation added here
146
147 process(Bus2IP_Clk) is
148 begin
149 if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
150 if Bus2IP_Reset = '1' then
151 ctrl1 <= '0';
152 else
153 if Bus2IP_WrCE(0) = '1' then
154 ctrl1 <= Bus2IP_Data(0);
155 else
156 ctrl1 <= ctrl1;
157 end if;
158 end if;
159 end if;
160 end process;
161
162 process(Bus2IP_Clk) is
163 begin
164 if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
165 if ctrl1 = '1' then
166 mbctrl_rst <= '1';
167 else
168 mbctrl_rst <= mb_reset;
169 end if;
170 end if;
171 end process;
172
173

```

Figure 26: Logic in Configuration Unit

3.2 The Design of Sharing Memory

A. Implicit Multiprocessing

In this method, a single copy of operating system runs and controls any number of processors in the system (as shown in Figure 27). Parallelism between the processors is

hidden by an operating system and hardware. The Microblaze processor does not support cache coherency and hence the implementation of cache coherency in software application has a very large impact.

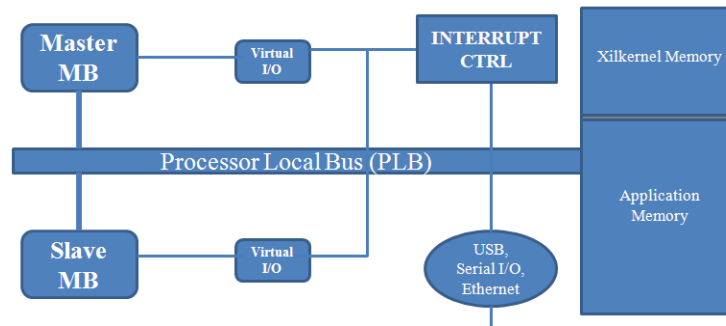


Figure 27: Implicit Multiprocessing

B. Explicit Multiprocessing

Another method of achieving multi-processor systems is to run every Microblaze with its own copy of RTOS. The Microblazes will have their private own address zones within the shared memory and the shared memory region with protocols. Since a different RTOS sits on each of the processor it leads to lot of memory space.

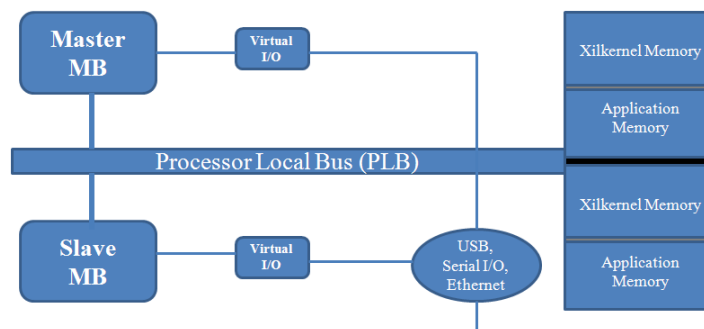


Figure 28: Explicit Multiprocessing

However, for more number of processors using shared BRAM is not supported, so we make use external memory controller MPMC. To enable large test programs and data sizes, we divide the external memory (DDR2_SDRAM) into several memory sections and one shared section. Each private memory section is associated with one processor and can only be accessed by that processor. The shared memory section, on the other hand, can be accessed by all processor cores.

3.3 The Design of Inter-processor Communication

A. XPS Mutex

In multiple processors XPS Mutex helps in synchronization when accessing shared resources. The mutex core has a configurable number of mutexes and writes to lock scheme. The mutex provides a mechanism for mutual exclusion to enable one processor to gain access to the shared resource. The shared resource in our project is the on board RS 232 UART interface where all the processors redirect their STDOUT to this shared console. Without synchronization, the console output would become useless. Hence, each processor locks XPS Mutex core before doing any output and then unlocks when done. The XPS Mutex IP currently available can support up to 8 processors. The connection of XPS Mutex to the PLB of individual processors is shown in figure below. The current XPS Mutex IP supports up to 8 PLB or FSL interfaces.

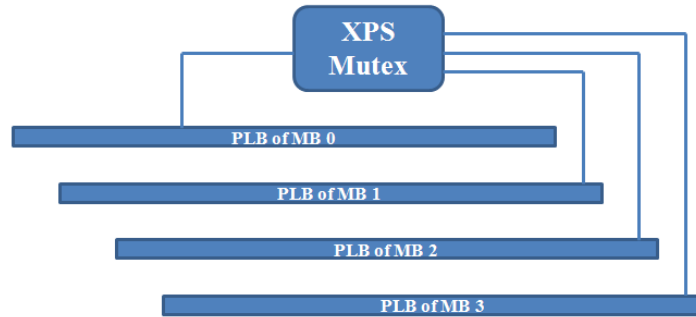


Figure 29: Schematic Representation of XPS Mutex Connection

B. XPS Mailbox

In a multi-processors environment, Processors need to communicate the data among them. The mailbox IP helps passing these simple messages (< a few 100 bytes) from one processor to another in a FIFO fashion. These mailboxes have a bi-directional communication channel and can be connected through PLB or FSL interface (similar to Figure 29). Apart from sending the data between processors, the mailbox can also be used to generate interrupts between the processors. The XPS Mailbox has two interfaces that are used to connect to the rest of the system. Both of these interfaces can be independently configured to use PLB or FSL bus.

The Matrix Addition Example on the system

Initially, the dual Microblaze processor system was tested with matrix addition example with booted real-time operating system, xilkernel. Using inter-processor communication technique, a system with two processes that uses a shared BRAM and an external memory DDR2SDRAM to accommodate the OS xilkernel, is built. The master

Microblaze has the data on which matrix addition is to be performed. It writes the data to the shared BRAM. The slave Microblaze waits and keeps on checking whether a particular bit is set or not flagging that data has been written completely. Then the slave Microblaze starts reading the data. Once completed it calculates and writes the result to a different location on the shared BRAM memory. The master Microblaze then calculates the final result and displays the result on the hyper terminal window (RS232 UART).

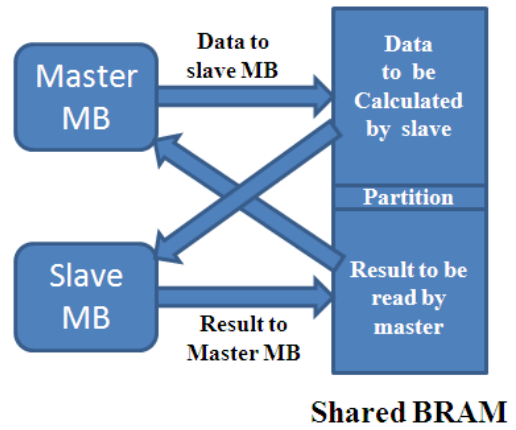


Figure 30: Schematic Representation of Matrix Addition

Output Display RS232UART

The serial output is supposed to print out computation result. The baud rate can be configured in this IP. Due to the limit of computer serial output, the highest baud rate can be set to 112k. In addition, the parity and bit number should be set the same as with the computer serial terminal.

Boot loader

The scratch pad memory within a PE unit is limited. The Xilinx's EDK boots only one processor at a time. To boot all four Microblazes, we need to customize the bootloader and configure other programs like link script. Also, the flash programmer is supposed to program the main program into the flash. Boot loader is the program in charge of program loading from flash memory into DDR2_SDRAM. As we have multiple programs to load, we also need to customize the default bootloader, so that they can load all the images into DDR_SDRAM. After the program loading, it needs to notify the other processor, so that they can jump into the memory. The default boot loader only loads one image. The following code reloads multiple copies of program images from flash:

```
120 //Load the image for MB3
121 print(" Start of Image 3\r\n");
122 flbuf = (uint8_t*)FLASH_IMAGE_BASEADDR3;
123 ret3 = load_exec ();
124 print(" End of Image 3\r\n");
125
126 //Notify the MB3_Boot to jump into the program address
127 //putsfsl(1,2);
128
129 //Load the image for MB2
130 print(" Start of Image 2\r\n");
131 flbuf = (uint8_t*)FLASH_IMAGE_BASEADDR2;
132 ret2 = load_exec ();
133 print(" End of Image 2\r\n");
134 //Notify the MB2_Boot to jump into the program address
135 //putsfsl(1,1);
136
137 //Load the image for MB1
138 print(" Start of Image 1\r\n");
139 flbuf = (uint8_t*)FLASH_IMAGE_BASEADDR1;
140 ret1 = load_exec ();
141 print(" End of Image 1\r\n");
142 //Notify the MB1_Boot to jump into the program address
143 //putsfsl(1,0);
144
145 //Load the image for MBO
146 //Make sure the last image is for MBO
147 print(" Start of Image 0\r\n");
148 flbuf = (uint8_t*)FLASH_IMAGE_BASEADDR;
149 ret = load_exec ();
150 print(" End of Image 0\r\n");
151 //laddr = (uint8_t *)XPAR_DDR_SDRAM_MPMC_BASEADDR ;
152 //(*laddr)(); //start the program
153
```

Figure31: Loading more than one image into FLASH PROM

The starting address of each image can be found in header file blconfig.h.

```

1 #define FLASH_IMAGE_BASEADDR      ( XPAR_FLASH_MEMO_BASEADDR+ 0x90000000) //Program for MB0
2 #define FLASH_IMAGE_BASEADDR1     ( XPAR_FLASH_MEMO_BASEADDR + 0x91000000) //Program for MB1
3 #define FLASH_IMAGE_BASEADDR2     ( XPAR_FLASH_MEMO_BASEADDR + 0x92000000) //Program for MB2
4 #define FLASH_IMAGE_BASEADDR3     ( XPAR_FLASH_MEMO_BASEADDR + 0x93000000) //Program for MB3
5

```

Figure 32: Define starting address of image in FLASH PROM

Before the program loading, reset all other processor via Configuration control unit.

```

109
110
111 //reset other MicroBlazes
112 val = 1 ;
113 shift = val<<31;
114 CONFIG_CTRL_mWriteReg(XPAR_CONFIG_CTRL_O_BASEADDR,0,shift);
115

```

Figure 33: Reset other processor before image loading

After the program loading, reset other processors, and jump into starting address of new program. The jump start address is predefined.

```

154
155
156 //Release other MicroBlazes
157 val = 0 ;
158 shift = val<<31;
159 CONFIG_CTRL_mWriteReg(XPAR_CONFIG_CTRL_O_BASEADDR,0,shift);
160
161 //Jump to the starting address of MB0
162 func_ptr = PROG_START_ADDR;
163 func_ptr();
164
165 /* If we reach here, we are in error */
166

```

Figure 34: How to enter main program after loading

Boot Program

Here we can boot load the program in the processors other than the main processor during the loading period. The main purpose for Boot Program is to enter the main program address. Because Configuration unit already resets other processors during load, Boot Program only need to jump to the correct main program. For MicroBlaze_0, the initialized program is bootlaoder. Because processor '0' is in charge of program loading.

For the other processor, you still need to initialize a program, which can wait for the ready signal from Microblaze_0. When the boot loader finishes the program loading, it will reset other processor, the Boot Program restart and enter the specified address. The address is predefined in the boot program. Define the starting address and jump into the address after release.

```
24
25 #define PROG_START_ADDR XPAR_DDR_SDRAM_MPMC_BASEADDR + 0x01000000 //Starting address of MB1_app
26
27
28 int (*func_ptr) ();
29
30 int main (void) {
31
32     int m;
33
34     func_ptr = PROG_START_ADDR;
35
36     //getfsl(m,1);
37
38     // jump to start execution code at the address
39     // PROG_START_ADDR
40     func_ptr();
41
42 }
```

Figure 35: Boot Program Sample

Linker Script

The linker script is the file which allocates memory space for the program. You need to assign the sections, manage the heap and stack. What's more important, you need to specify the loading address for the other program. By default, loading address is the starting address of the DDR2_SDRAM. For multiple processors program loading, the latter will overlap the previous one if they share the same starting address. So the linker script file must be modified to avoid this overlap.

For other processors, after the basic configuration, you need to modify the original address and the length of the DDR2_SDRAM should be customized. Correct address and length should be calculated as well.

Flash Programmer

Flash programmer can write the main program into FLASH memory, but each program must be written into different address. And there must be enough space between programs to avoid overlap. Please note you need to instantiate debug module and download the bit stream before you use flash programmer. The detail can be found in Xilinx's manual.

The following is the flash programmer sample setting. As you can see, the offset is set to be '0x90000000'. For other processor, you need to set an offset, which is used in the boot loader. This offset should be consistent with your offset setting in header file blconfig.h of boot loader.

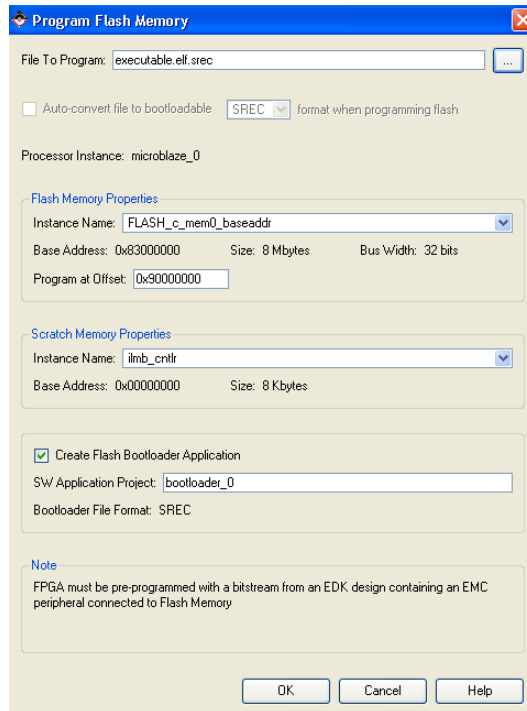


Figure 36: Programming FLASH memory

Setup HyperTerminal:

The hyper terminal connections are set as shown in the figure below. We have to make sure that the RS232 UART and the hyper terminal connections are same.

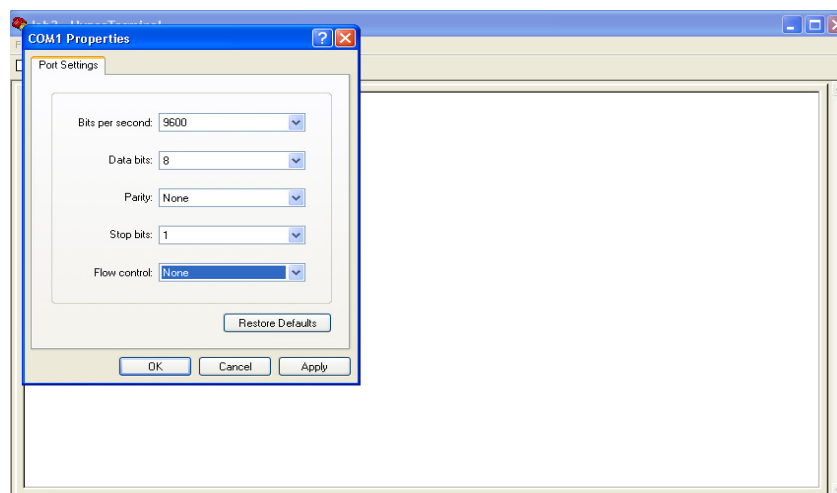


Figure 37: Setting RS232 serial port terminal for output

The overall hardware architecture of our design through Xilinx EDK platform studio is as shown below.

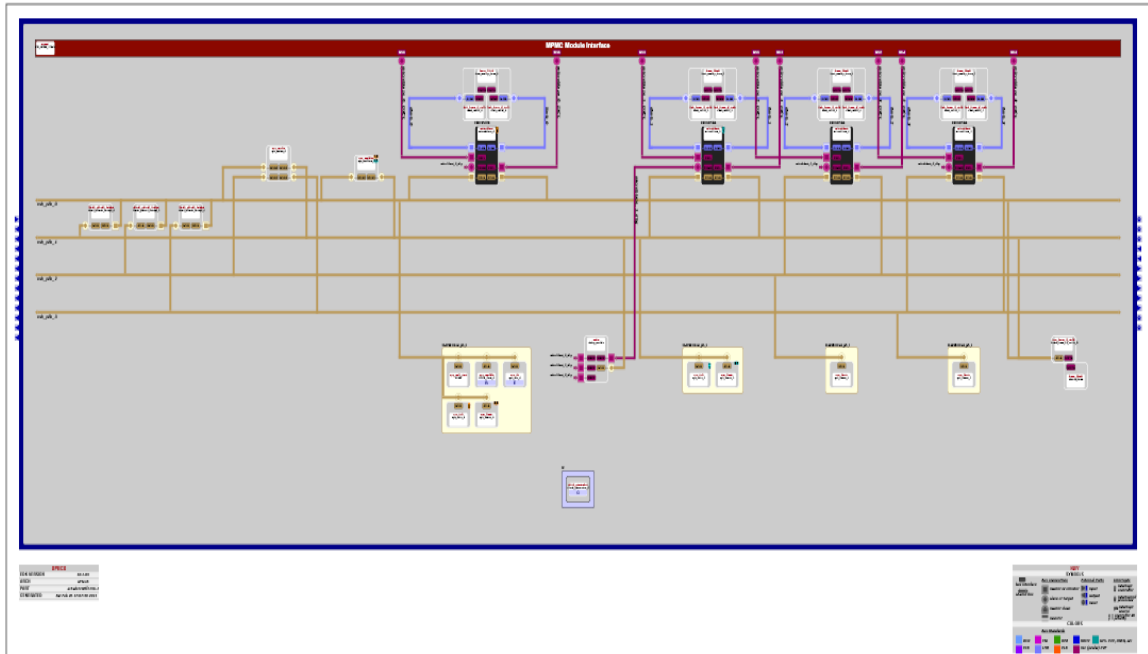


Figure 38: Overall System View.

CHAPTER 4

OPERATING SYSTEM SUPPORT AND SOFTWARE PLATFORM SETTINGS

As explained earlier, we want to boot up our hardware with real-time operating system in order to implement different power/thermal scheduling policies for ease of development and testing. Therefore, under XPS GUI we configured operating system and library as shown in Figure 39 below. The RTOS used in our project is explained in the following section.

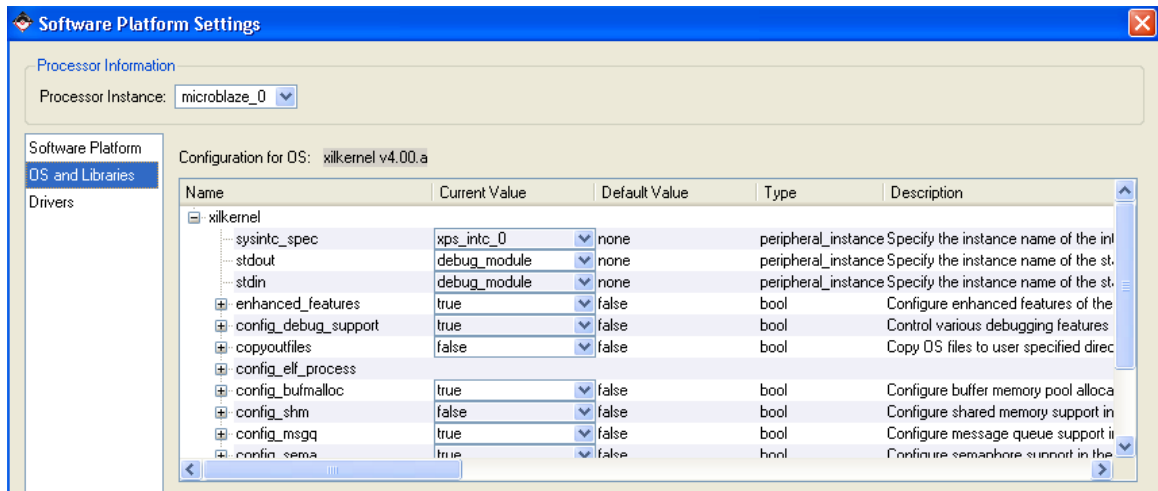


Figure 39: Software Platform Settings

4.1 Standalone Board Support Package

The Board Support Package (BSP) is the lowest layer of software modules used to access processor specific functions. When there is no operating system in our design, the library generator automatically builds the standalone BSP in the project library [22]. The standalone BSP is designed for use when an application accesses board/processor features directly and is below the operating system layer. Initially, the multi-core

Microblaze was tested with the standalone BSP and implemented to perform synchronization when accessing a shared resource.

4.2 Building of Xilkernel Real-time Operating System

Xilkernel is a small, robust and modular kernel that provides scheduling multiple execution contexts. It is a free software library integrated within the Xilinx embedded development kit (EDK) tool [5]. It is a simple embedded processor kernel that can be customized to a large degree for a given system. It supports the core features required in a lightweight embedded kernel, with a POSIX API [22]. Xilkernel IPC services can be used to implement higher level services (such as networking, video, and audio) and subsequently run applications using these services.

The main advantages of using xilkernel are:

- Breaking down tasks as individual applications and implementing them on an operating system
- It enables us to write the code at an abstract level, instead of at a small, micro-controller level
- Highly scalable kernel (inclusion or exclusion of functionality as required) and
- Complete kernel configuration (deployment within minutes from inside of Xilinx platform studio of EDK) [5, 12].
-

Xilkernel includes the following key features:

- A POSIX API targeting embedded kernels
- Core kernel features such as:

- POSIX threads with round-robin or strict priority scheduling
 - POSIX synchronization services - semaphores and mutex locks
 - POSIX IPC services - message queues and shared memory
 - Dynamic buffer pool memory allocation
 - Software timers
 - User level interrupt handling API
- Highly robust kernel, with all system calls protected by parameter validity checks and proper return of POSIX error codes
 - Statically creating threads that startup with the kernel
 - System call interface to the kernel
 - Support for creating processes out of separate executable Executable Link Files (ELF)

4.3 Scheduling

In computer multitasking, multi-processing operating system and real-time operating systems, the key concept is scheduling. Scheduling is the process of deciding how to commit available resources between varieties of available or possible tasks. In xikernel RTOS, we can make use two types of scheduling techniques: Round robin scheduling and Priority based scheduling. However, for our project we made use of only round robin scheduling.

A. Round robin scheduling:

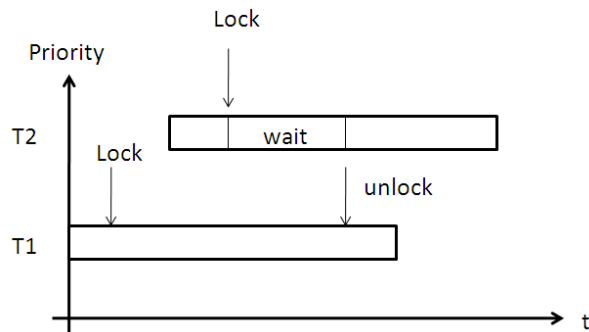
Round robin is one of the simplest scheduling algorithms available for processes or threads in an operating system. They assign time slices to each process or thread in equal

proportion and in circular order. They handle the processes or threads without any priority.

B. Priority based scheduling:

Using priority based scheduling, we can assign higher priority to processes or threads that are more important than other processes. The process or thread with the highest priority will have the control over the CPU when it is available.

In our project, to gain exclusive access to a resource, when multiple threads are created and each of them want to send debug information to the serial I/O interface, we made use of mutexes in real-time concepts. Mutexes are a synchronization mechanism between threads.



Mutexes are a type of synchronization between threads because even though thread 2 has a higher priority it has to wait on an event (unlock) of thread 1. It means that while T1 is using the UART and hence locks the mutex, T2 has to wait to use the UART. Like this we ensure exclusive access to the UART resource.

Once we have finished adding all the required IP cores and other peripherals we have to set the platform for software application. The xilkernel Real-Time Operating System (RTOS) is selected and configured according to the application requirements i.e., mutex enabled and then the board support package is generated using library generator. The library generator uses the configuration information and the hardware design net list to setup complete software application. The xilkernel sources, make files and other scripts are built with conditional code to generate the correct software based on the hardware described in EDK XPS GUI. The compiler options are set to link against the xilkernel RTOS to obtain the ELF file and then, downloaded to the board.

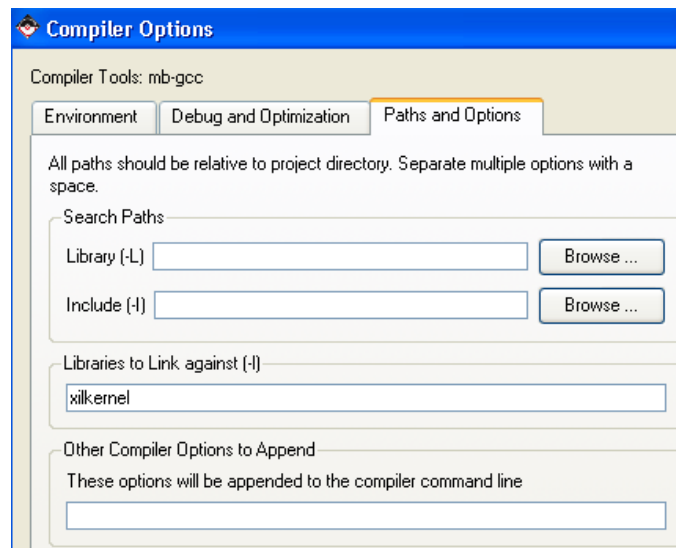


Figure 40: Setting Compiler Options

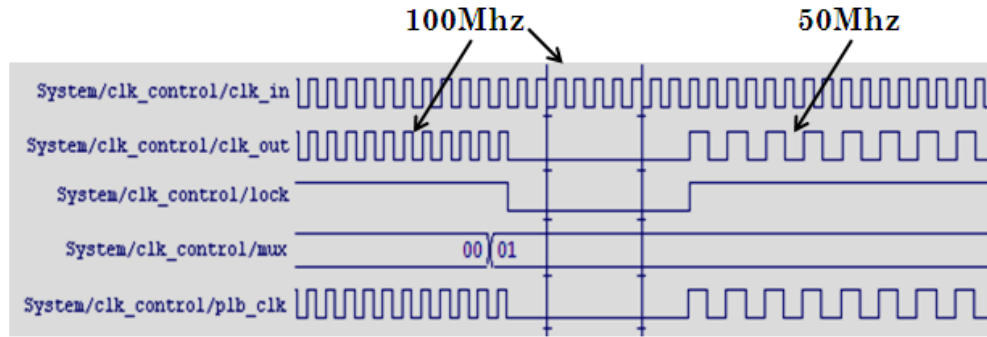
CHAPTER 5

EXPERIMENTS AND RESULT

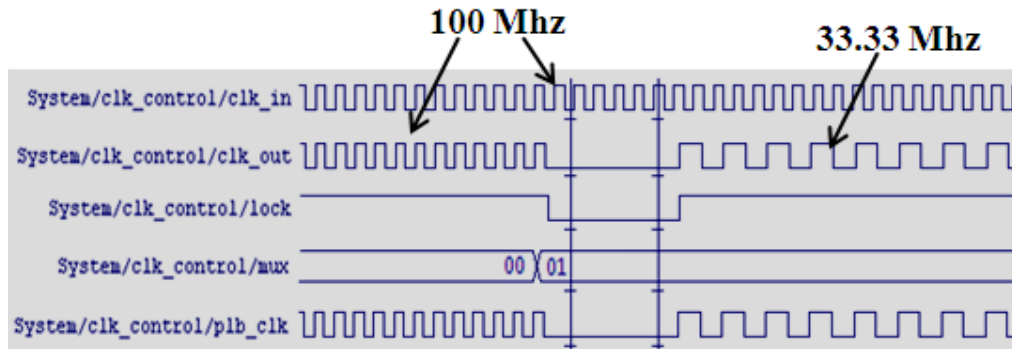
Once the hardware system and all the software platform settings are configured according to the required applications, we can obtain the output on the hyper terminal. In our case, as discussed in the previous chapters we considered the Matrix addition example and the utilization of XPS Mutex hardware IP for multi-processors to achieve synchronization when accessing shared resources.

5.1 Varying the Working Frequency

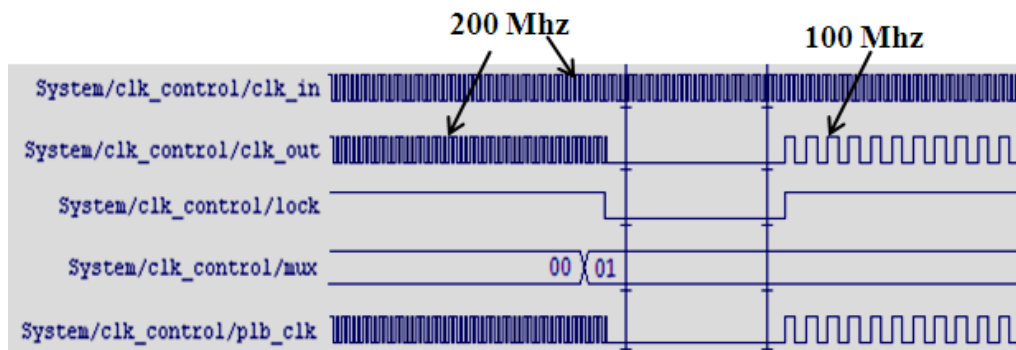
Several interesting parameters were investigated with profiling. This includes the frequency transition and the timing overhead. The frequency switching overhead, i.e., the time required to change the processor frequency from one to another, was measured by inserting Xilinx's ChipScope Integrated Logic Analyzer (ILA) core into the design and sampling the IP internal signals via JTAG connection. The timing diagram is shown in Figure 40. The *clk_in* is the input and *clk_out* is system clock. The *lock* signal indicates the DCM working status: it goes low when reconfiguration starts, and goes high when stable output is available. Therefore, the interval when lock goes low represents the frequency switching overhead. The overhead is quite constant and not much dependent to the starting and ending frequencies.



Case (i) The switching overhead is approximately 9 to 10 cycles.



Case (ii) The switching overhead is approximately 6 to 7 cycles.



Case (iii) The switching overhead is approximately 12 to 14 cycles.

Figure 41: The Timing Overhead for Varying the Working Frequencies

The overall system resource utilization summary table is shown in Table 1. When considering whole design, the utilization rate of slices may exceed 100% because the system only provides estimation based on subsystem utilization.

Device resource	Utilization on FPGA Virtex5 Device: XC5VLX110T-1FF1136		
	Used	Available	Percentage
No. of slice registers	32684	69120	47%
No. of slice LUTs	38543	69120	56%
DCM	4	4	100%
Number of fully used LUT pairs	3	13	23%
No. of BUFG/BUFGCTRLs	3	32	9%
No. of bonded IOBs	28	640	4%

Table 1: Resource Utilization Summary

5.2 Debugging

In order to enable more number of processors to download the ELF file, we have to make use Xilinx Microblaze Debug Module (MDM) in our design. MDM enables JTAG-based debugging of one or more Microblaze processors. The present Xilinx MDM IP of XPS GUI supports up to eight Microblaze processors. They are also helpful in achieving synchronized control for multiple Microblazes used. Xilinx MDM supports a JTAG-based UART with a configurable AXI4-Lite or PLB interface. The main advantage of MDM is connecting to the chipscope Integrated CONTroller (ICON) cores through BSCAN signals.

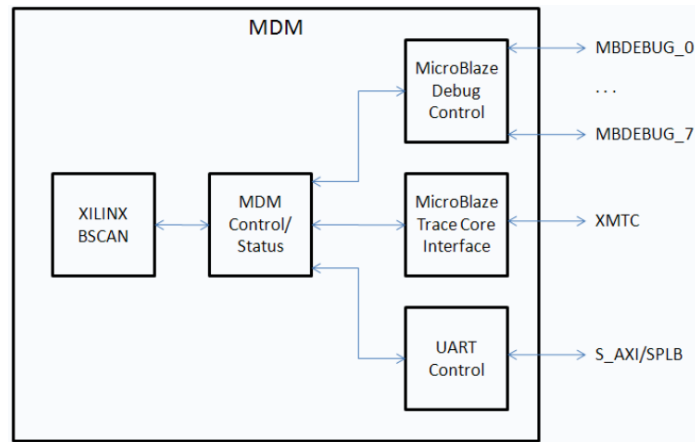
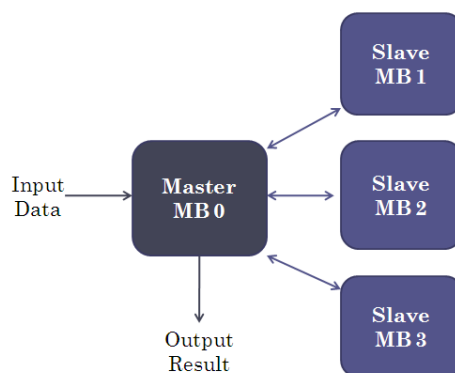


Figure 42: Block Diagram of Microblaze Debug Module (MDM)

From the Xilinx's Microprocessor Debug (XMD) shell we connect the JTAG based UART to MDM, and then download the ELF file to the respective processors to run the system and display the output.

5.3 Four Microblazes system using single PLB bus

For the matrix addition and multiplication example we considered different hardware setups. Firstly, a single PLB bus was shared by all the processors with one master Microblaze and three slave Microblaze processors (as shown in Figure 45). We made use of external memory to write and read data by different processors.



Initially, we considered a simple addition example and later a 3X3 matrix multiplication example to increase the workload of the processors. All the data required for computation of addition or multiplication is entered into master processor then it distributes the data evenly between the three slave processors. The slave processors compute the addition or multiplication and send the result back to master processor. After retrieving the results from the slaves, the master processor computes the total result and displays the output on the hyper terminal (as shown in Figure 43).

```

-----
1  initial = 0;
2  s = initial << 30;
3  CLK_CONTROL_mWriteReg (BASEADDR,0,s);
4  while (1)
5  {
6    getfsl(temp, 0);
7    temp++;
8    putfsl(temp, 1);
9    //Matrix calculations during transition
10   for (i=0; i<3; i++) {
11     for (j=1; j<3; j++) {
12       mat[i][j] = 2*i * temp - j *temp ;
13     }
14   }
15 }
-----

```

Code line1-3 configures the clock control unit. As the MicroBlaze adopts little-endian bus, we should shift the data before write it into registers. Line 3 calls a register write function and write configuration into custom IP.

Line 6-8 is passing the FSL data from one processor to another. The getfsl and putfsl are blocking FSL read and write.

Line 10 -14 is the for loop calculation.

For the simplicity, we keep the processor in the while loop from line 4 to line 15, which is matrix calculation. The code is expected to keep the processor busy.


```

9600 - HyperTerminal
File Edit View Call Transfer Help

-- Entering the main function in the MASTER processor --
These numbers are to be added to each other:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19
The Master processor is distributing data...
(1)The slave1 adds up the following numbers...
0
1
2
3
4
(2)The slave2 adds up the following numbers...
5
6
7
8
9
(3)The slave3 adds up the following numbers...
10
11
12
13
14
(4)The master adds up the following numbers...(local calculations)
15
16
17
18
19
slave1 result = 10
slave2 result = 35
slave3 result = 60
master result = 85
total result = 190

Done :-)
-

```

```

9600 - HyperTerminal
File Edit View Call Transfer Help

-- Entering the main function in the MASTER processor --
These numbers are to be multiplied:
4 8 15 19 24 28 12 7 3 16 6 11 15 18 23 17 4 3 6 8 12 17 19 25 18 11 20 22
15 9 3 5 18 19 27 8 18 21 9 7 12 13 17 11 26 25 11 9 5 10 18 22 28 6

The Master processor is distributing data...
The slave1 multiplies the following numbers....
4 8 15 19 24 28 12 7 3 16 6 11 15 18 23 17 4 3

The slave2 multiplies the following numbers....
6 8 12 17 19 25 18 11 20 22 15 9 3 5 18 19 27 8

The slave3 multiplies the following numbers....
18 21 9 7 12 13 17 11 26 25 11 9 5 10 18 22 28 6

The Master adds up the total result after multiplication from slaves....

slave1 result = 439 228 273 1140 658 845 348 210 302
slave2 result = 384 454 294 906 1025 695 809 865 520
slave3 result = 753 660 594 521 451 357 1052 1025 507
Total result = 1576 1342 1161 2567 2134 1897 2209 2100 1329

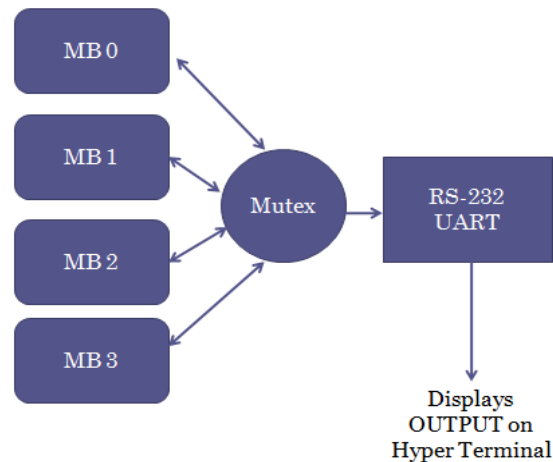
Done

```

Figure 43: Output for matrix addition and Multiplication example on 4 Microblazes system

5.4 Four Microblazes system using different PLB buses and a shared console

Secondly, we made use of PLB buses for each of the processor which are internally connected by PLB-to-PLB Bridge as explained in the previous chapter. XPS Mutex hardware IP was used to attain the synchronization for shared console RS232 UART between the 4 Microblaze processors. We also made use of a XPS Mailbox between the two processors to enable inter-processor communication. Since the mailbox is suited for small sized messages (< a few 100 bytes), we considered FSL bus for a 4 Microblazes system considering fast communication ability in them.



As we can see in the block diagram above, the hardware mutex IP is connected to all the processors through individual PLB buses. Whenever a processor wants to display the output, it locks the mutex thereby no other processor can access the resource for output display. All the processors rendezvous between each other to synchronize their output to RS232 console as shown in Figure 44. You can see the change of the state in the processors when synchronizing or accessing resource one after the other in cyclic manner.

```

107      /* Rendezvous first to enable co-ordinated output */
108      #if XPAR_CPU_ID == 0
109          *sharedstate = 0x3;
110          while (*sharedstate != 0x0 && *sharedstate != 0x1 && *sharedstate != 0x2)
111              ;
112      #else if
113          XPAR_CPU_ID == 1
114          *sharedstate = 0x2;
115          while (*sharedstate != 0x0 && *sharedstate != 0x1 && *sharedstate != 0x3)
116              ;
117      #else if
118          XPAR_CPU_ID == 2
119          *sharedstate = 0x1;
120          while (*sharedstate != 0x0 && *sharedstate != 0x2 && *sharedstate != 0x3)
121              ;
122      #else if
123          | XPAR_CPU_ID == 3
124          while (*sharedstate != 0x1 && *sharedstate != 0x2 && *sharedstate != 0x3)
125              ;
126          printf ("done\r\n");
127          *sharedstate = 0x0;
128      #endif

```

```

9600 - HyperTerminal
File Edit View Call Transfer Help

Multiprocessor Synchronization Demo!
-----
Desc -- CPU0 CPU1 CPU2 and CPU3 will synchronize their output to this
console using XPS Mutex Locks.
-----
SHARED_CONSOLE: CPU0 CPU1 CPU2 and CPU3 rendezvousing...done
CPU(3) -- Changing sharedstate from 0 to 4000.
CPU(2) -- Changing sharedstate from 4000 to 3000.
CPU(1) -- Changing sharedstate from 3000 to 2000.
CPU(0) -- Changing sharedstate from 2000 to 1000.
CPU(3) -- Changing sharedstate from 1000 to 4001.
CPU(2) -- Changing sharedstate from 4001 to 3001.
CPU(1) -- Changing sharedstate from 3001 to 2001.
CPU(0) -- Changing sharedstate from 2001 to 1001.
CPU(3) -- Changing sharedstate from 1001 to 4002.
CPU(2) -- Changing sharedstate from 4002 to 3002.
CPU(1) -- Changing sharedstate from 3002 to 2002.
CPU(0) -- Changing sharedstate from 2002 to 1002.
CPU(3) -- Changing sharedstate from 1002 to 4003.
CPU(2) -- Changing sharedstate from 4003 to 3003.
CPU(1) -- Changing sharedstate from 3003 to 2003.
CPU(0) -- Changing sharedstate from 2003 to 1003.
CPU(3) -- Changing sharedstate from 1003 to 4004.
CPU(2) -- Changing sharedstate from 4004 to 3004.
CPU(1) -- Changing sharedstate from 3004 to 2004.
CPU(0) -- Changing sharedstate from 2004 to 1004.
CPU(3) -- Changing sharedstate from 1004 to 4005.
CPU(2) -- Changing sharedstate from 4005 to 3005.
CPU(1) -- Changing sharedstate from 3005 to 2005.
CPU(0) -- Changing sharedstate from 2005 to 1005.
CPU(3) -- Changing sharedstate from 1005 to 4006.
CPU(2) -- Changing sharedstate from 4006 to 3006.
CPU(1) -- Changing sharedstate from 3006 to 2006.
CPU(0) -- Changing sharedstate from 2006 to 1006.
CPU(3) -- Changing sharedstate from 1006 to 4007.
CPU(2) -- Changing sharedstate from 4007 to 3007.
CPU(1) -- Changing sharedstate from 3007 to 2007.
CPU(0) -- Changing sharedstate from 2007 to 1007.
CPU(3) -- Changing sharedstate from 1007 to 4008.
CPU(2) -- Changing sharedstate from 4008 to 3008.
CPU(1) -- Changing sharedstate from 3008 to 2008.
CPU(0) -- Changing sharedstate from 2008 to 1008.
CPU(3) -- Changing sharedstate from 1008 to 4009.
CPU(2) -- Changing sharedstate from 4009 to 3009.
CPU(1) -- Changing sharedstate from 3009 to 2009.
CPU(0) -- Changing sharedstate from 2009 to 1009.
SHARED_CONSOLE: CPU(3) Ends
SHARED_CONSOLE: CPU(2) Ends
SHARED_CONSOLE: CPU(1) Ends
SHARED_CONSOLE: CPU(0) Ends

```

Figure 44: For 4 Microblazes system using different PLB buses and a shared console

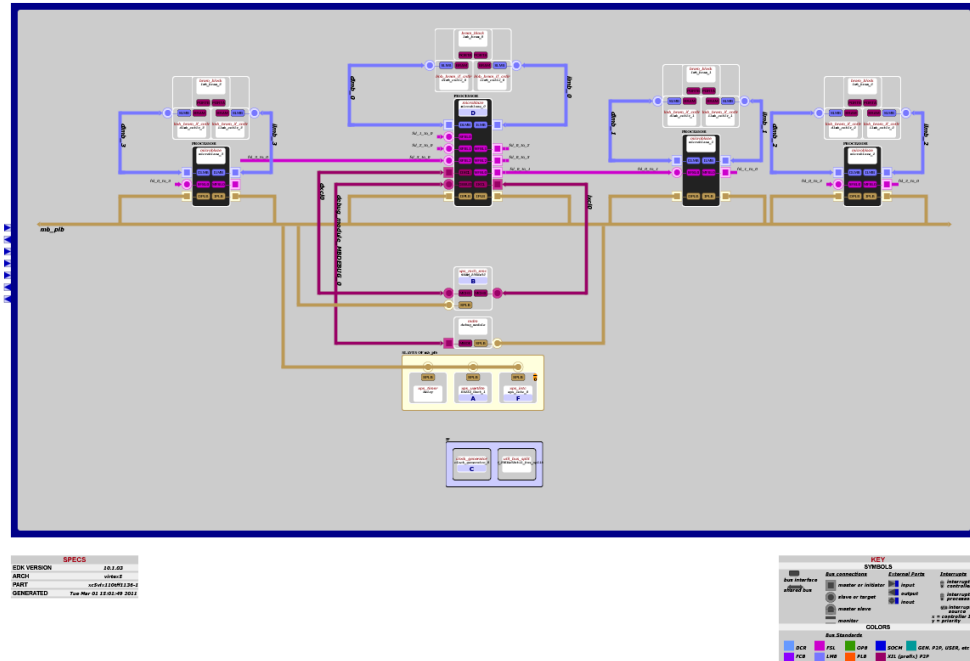


Figure 45: System architecture view for 4 Microblaze system with single PLB bus

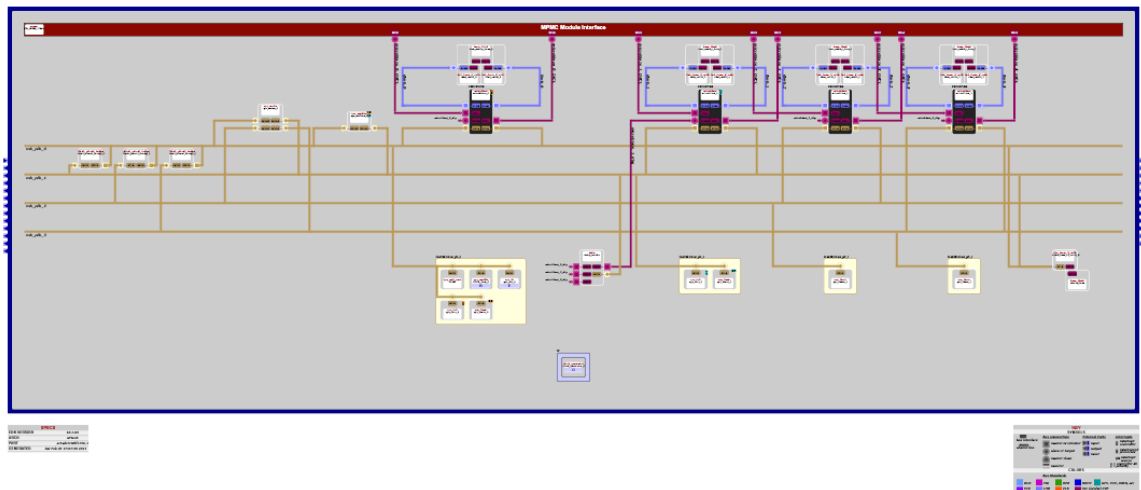


Figure 46: System architecture view for 4 Microblazes system with individual PLB buses connected with PLB-to-PLB Bridge and XPS Mutex.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The adaptivity is critical for multi-core architecture, which has the great potential to meet the increasingly demanding performance requirements but also needs to satisfy the stringent resource constraints. Multi-core processors have become the mainstream computing research. We have developed a flexible, reusable, and versatile multi-core test bed on FPGA that can be used effectively to validate the theoretical research on power/thermal aware computing. We expect that this test bed can lead to new findings and research directions in our power/thermal computing research.

We need to manually partition the codes and map them into processors. This makes the program model very difficult for parallel computing. Further work on parallel programming is expected to improve both the productivity and efficacy of this system. We are watching closely for the DVS features in the new generations FPGA products. While changing the frequencies helps to vary the performance, this has not transformed to its real benefit, i.e., more effective power/energy conservation.

Due to inaccuracy in the simulator and not being able to dynamically vary the voltage in the present FPGAs, we are not able to develop more complicated power/thermal models with scheduling policies. In our design, only a 4-MicroBlaze point-to-point network topology is integrated and no float-point unit is added in the system. It would be interesting to integrate more processors and float-point units. Besides, some connections other than FSL, like network on chip, are also considerations for future work.

LIST OF REFERENCES

- [1] *Krste Asanović, Rastislav Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick*, “The Landscape of Parallel Computing Research: A View from Berkeley”, Dec 2006.
- [2] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Veluswamy, Karthik Sankaranarayanan and David Tarjan, University of Virginia “Temperature-Aware Computer Systems: Opportunities and challenges”.
- [3] Osman S. Unsal, and Israel Koren, “System-Level Power-Aware Design Techniques in Real-Time Systems”, proceedings of the IEEE, Vol. 91, No. 7, July 2003.
- [4] Katarina Paulsson, Michael Hubner, Jurgen Becker, “Dynamic Power Optimization by exploiting self-reconfiguration in Xilinx Spartan 3-based systems”.
- [5] Xilinx platform studio user guide, UG113 (v4.0), February 2005.
- [6] Xilinx Embedded systems tools reference manual, UG111 (v6.0) June 2006.
- [7] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli, “Dynamic frequency scaling with buffer insertion for mixed working loads”. IEEE transactions on computer aided design of integrated circuits and systems, vol.21, No.11, November 2002.
- [8] Bishop Brock and Karthick Rajamani, “Dynamic power management for embedded systems”. Proceedings of IEEE international SoC conference September 2003.
- [9] Kumar, A. Li Shang, Li-Shiuan Peh, Jha, N.K. ”System-Level Dynamic Thermal Management for High-Performance Microprocessors” in proceedings of Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on Volume: 27 pp 96-108, 2008.
- [10] G. Magklis, G. Semeraro, D.H. Albonesi, S.G. Dropsho, S. Dwarkadas, and M.L. Scott, “Dynamic frequency and voltage scaling for a multiple clock-domain microprocessor”. IEEE Micro, vol. 23, no. 6, pp. 62–68, Nov/Dec 2003.
- [11] Vasanth Asokan, “Dual Processor Reference Design Suite” Xilinx (v1.3) October 6, 2008.

- [12] Xilinx Micro-kernel on a MicroBlaze processor using Xilinx EDK tool.
- [13] K. Asanović et al. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.
- [14] M. M. K. Martin et al. Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.
- [15] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner., “Simics: A full system simulation platform,” in *IEEE Computer*, 2002.
- [16] Xilinx LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c) DS449 April 19, 2010.
- [17] Xilinx XPS Mutex (v1.00c) DS631 June 24, 2009.
- [18] Xilinx Digital Clock Manager (DCM) Module DS485 April 24, 2009.
- [19] G.Schelle, D.Grunwald, “Onchip interconnect exploration for multicore processors utilizing FPGAs” *2nd Workshop on Architecture Research using FPGA Platforms*, 2006
- [20] <http://www.xilinx.com/products/devkits/XUPV5-LX110T.htm>
- [21] <http://www.xilinx.com/ise/embedded/edk.htm>
- [22] http://www.xilinx.com/support/documentation/sw_manuals/edk10_oslib_rm.pdf.
Operating System and document collections.
- [23] http://www.xilinx.com/ise/design_tools/
- [24] J.Wawrzynek et al “RAMP: A Research Accelerator for Multiple Processors”
<http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-158.pdf>
- [25] P. Choudhary, D.Marculescu, “Hardware based frequency/voltage control of voltage frequency island systems” *Proc. IEEE/ACM Intl. Conference on Hardware-Software Codesign and System Synthesis (CODES-ISSS)*, Seoul, South Korea, Oct. 2006

- [26] Min Bao, Alexandru Andrei, Petru Eles, Zebo Peng “On-line Thermal Aware Dynamic Voltage Scaling for Energy Optimization with Frequency/Temperature Dependency Consideration” DAC’09, July 26-31, 2009, San Francisco, California, USA
- [27] Yongpan Liu, Huazhong Yang, Robert P. Dick, Hui Wang, Li Shang “Thermal vs Energy Optimization for DVFS-enabled Processors in Embedded Systems” proceedings of the 8th international symposium ISQED’07
- [28] L.Shang, A.Kaviani, K.Bathala “Dynamic power consumption in Virtex II FPGA family” *Proc. ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, 2002, pp. 157 - 164
- [29] W. Eatherton, “The Push of Network Processing to the Top of the Pyramid,” keynote address at *Symposium on Architectures for Networking and Communications Systems*, Oct. 26–28, 2005.
- [30] J.Chen, C.Kuo, “ Energy-Efficient Scheduling for Real-Time systems on Dynamic Voltage Scaling (DVS) Platforms” *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications(RTCSA 2007)* Aug. 2007, pp28-38
- [31] G.Quan, X.Hu, “Minimum energy fixed-priority scheduling for variable voltage processors” *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2002, pp.782-787
- [32] G. Quan and X. Hu, “Energy efficient DVS schedule for fixed-priority real-time systems," *Proc. 38th Design Automation Conference*, 2001
- [33] Blair Fort, Davor Capalija, Zvonko G. Vranesic and Stephen D. Brown, “A Multithreaded Soft Processor for SoPC Area Reduction” 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)
- [34] J.Hennessy and D.Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, Morgan Kauffman, San Francisco, 2007
- [35] Chip Multi Processor Watch http://view.eecs.berkeley.Edu/wiki/Chip_Multi_Processor_Watch
- [36] M.LaPedus, “To save power, embedded tries multicore” *EE times*, Issue 1481, June 25, 2007 pp. 1-6T.

- [37] J.Becker, M. Huebner, and M. Ullmann, “Power estimation and power measurement of Xilinx Virtex FPGAs: Trade-offs and Limitations” *Proc.16th Symposium on Integrated Circuits and Systems Design*, September 2003, pp. 283- 288
- [38] J. Suh, D.I. Kang and S. P. Crago. “Dynamic power management of multiprocessor systems”. In *International Parallel and Distributed Processing Symposium*, pages 97–104, 2002.

GLOSSARY

BRAM	Block Random Access Memory
BSP	Board Support Package
CLB	Configurable Logic Blocks
EDK	Embedded Development Kit
ELF	Executable Linked Format
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
IP	Intellectual Property
ISE	Integrated Synthesis Environment
LMB	Local Memory Bus
MDM	Microprocessor Debug Module
MHS	Microprocessor Hardware Specification
MMU	Memory Management Unit
MSS	Microprocessor Software Specification
NOC	Network on Chip
PLB	Processor Local Bus
RISC	Reduced Instruction Set Computer
RTOS	Real-Time Operating System
XCL	Xilinx Cache Link
XMD	Xilinx Microprocessor Debug