RHEALSTONE BENCHMARKING OF FREERTOS AND THE XILINX ZYNQ EXTENSIBLE PROCESSING PLATFORM

A Thesis

Submitted to

the Temple University Graduate Board

In Partial Fulfillment

of the Requirement for the Degree

MASTER OF SCIENCE

in ELECTRICAL ENGINEERING

by
Timothy J. Boger
May, 2013

ABSTRACT

Embedded system designers require deterministic, real-time operating system (RTOS) support for the commonly available processing hardware. The Xilinx Zynq Extensible Processing Platform (EPP) offers software, hardware, and input/output (I/O) programmability on a single chip. The Xilinx Zynq EPP features a Dual ARM Cortex-A9 MPCore, Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface 4 (AXI4) interconnect, and Xilinx Kintex-7 series Programmable Logic (PL) which provide the requisite capabilities for the increasing demands of embedded processing applications. The AMBA AXI4 interconnect provides high speed point to point interconnections between the ARM processor cores and the Field Programmable Gate Array (FPGA) structure allowing for rapid data transmission to optimize system performance. The incorporation of an RTOS ensures predictable execution times of applications. Benchmarks, such as the Rhealstone, were developed to provide designers with a method of evaluating and comparing these multitasking RTOSs running on various hardware platforms. This thesis research performs Rhealstone benchmarking and evaluates the AMBA AXI4 interconnect performance while executing FreeRTOS on the ARM core of the Zynq EPP device.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

ACP	Accelerator Coherence Port
AHB	AMBA High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
AMP	Asymmetric Multi-Processing
APB	Advanced Peripheral Bus
API	Application Programming Interface
APSL	Advanced Processor Systems Laboratory
ARM	Advanced RISC Machine
ASB	Advanced System Bus
ATB	Advanced Trace Bus
AXI	Advanced eXtensible Interface
BIF	Boot Image File
BSB	Base System Builder
BSP	Board Support Package
CPU	Central Processing Unit
DMAC	Direct Memory Access Controller
EPP	Extensible Processing Platform
EDK	Embedded Development Kit
ELF	Executable and Linkable Format
EMIO	Extended Multiplexed I/O
FPGA	Field-Programmable Gate Array
FSBL	First Stage Boot Loader
GDB	GNU Debugger
GPIO	General Purpose I/O
HPS	Hard Processor System
HDL	Hardware Descriptive Language
I/O	Input/Output
IP	Intellectual Property
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
ISE	Integrated Software Environment
JTAG	Joint Test Action Group
MHS	Microprocessor Hardware Specification
MIO	Multiplexed I/O
MMU	Memory Management Unit
MPC	Message Passing Coprocessor
OCM	On-Chip Memory
OS	Operating System
PL	Programmable Logic
PLB	Processor Local Bus
Pmod	Peripheral Module
PPC	Power PC
PS	Programming System
QEMU	Quick EMUlator
QSPI	Queued Serial Peripheral Interface
RTL	Register Transfer Level
RISC	Reduce Instruction Set Computer
RTOS	Real-Time Operating System
SCDL	System Chip Design Laboratory
SDK	Software Development Kit
SIMD	Single Instruction-Multiple Data
SMP	Symmetric Multiprocessing
SCU	Snoop Control Unit
SoC	System-On-a-Chip
TCL	Tool Command Language
TDP	Targeted Design Platforms
UCF	User Constraints File
UART	Universal Asynchronous Rx/Tx
XMD	Xilinx Microprocessor Debugger
XML	eXtensible Markup Language
XMP	Xilinx Microprocessor Project
XPS	Xilinx Platform Studio
ZED	Zynq Evaluation & Development

CHAPTER 1

INTRODUCTION

1.1 Motivation

The goal of this thesis research is to provide performance benchmarks for the Xilinx Zynq-7000 Extensible Processing Platform (EPP) and to provide a premise for future embedded design. The Xilinx Zynq EPP is capable of running Asymmetric Multiprocessing (AMP) of a Real-Time Operating System (RTOS) called FreeRTOS. [1] The Dual ARM Cortex A-9 MPCore processor is provided with various features including a primary Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface 4 (AXI4) 64-bit interconnect that can be used with various soft-core and hard-core peripherals and the 28nm Programmable Logic (PL) of the Xilinx Kintex-7 series Field Programmable Gate Array (FPGA). Embedded system designers require these benchmarks in order to evaluate and design an efficient Processing System (PS).

Multicore processor architectures have the potential to provide increased performance and power efficiency, but at the cost of programming complexity. [2] The complexity involved has been the hindrance in the widespread adoption of multicore architecture. Multicore systems can be implemented in Symmetric Multiprocessing (SMP) or AMP modes. These modes refer to how the Operating System (OS) kernel will run on a system that has more than one Central Processing Unit (CPU) or core.

A kernel is the underlying main component of the majority of computer OS. It bridges the gap between the hardware and the application executing on the PS. The kernel's responsibilities include managing the communication between hardware and software components to allocate the system's resources accordingly through system calls and inter-process communication. The hardware components it manages include the processor and I/O devices. [3] There are various types of kernel structures including monolithic, modular, extensible and layered. Figure 1 depicts these structures.

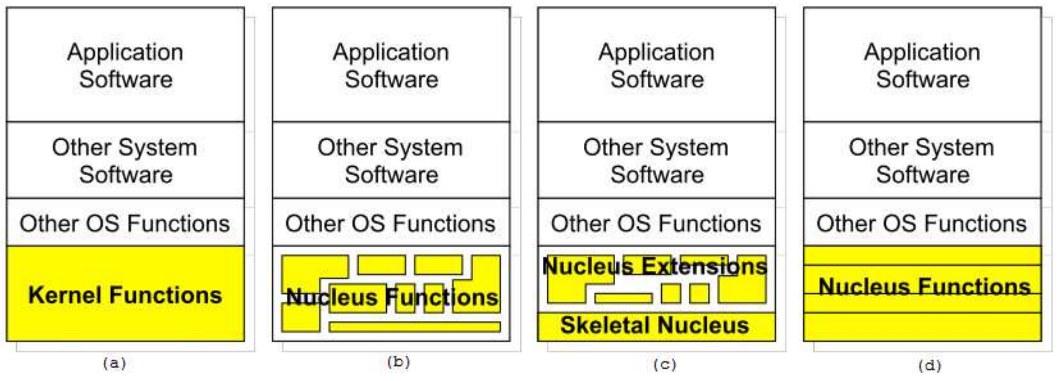

Figure 1: Kernel Layout: (a) Monolithic (b) Modular (c) Extensible (d) Layered [4]

Monolithic kernels are the most primitive with the OS code executing in the same address space. This direct intercommunication is highly efficient and increases performance, but makes it difficult to manage and maintain. Modular kernels allow for better overall functionality with ease of management due to its modular nature, but lacks performance. Layered kernels are used to divide components into manageable layers, but have degraded performance when communication between multiple levels is required. Extensible kernels, also known as microkernels, execute services in user space as servers to improve modularity and maintainability while also having a lower level skeletal nucleus that controls basic process synchronization. [4]

With SMP, the kernel itself can run on any processor and can run simultaneously on multiple processors. SMP handles programs using multiple processors sharing a common OS. There is a single copy of the operating system that supervises all of the processors and shares everything symmetrically among them. The processors share memory and a bus as shown in Figure 2. [5]

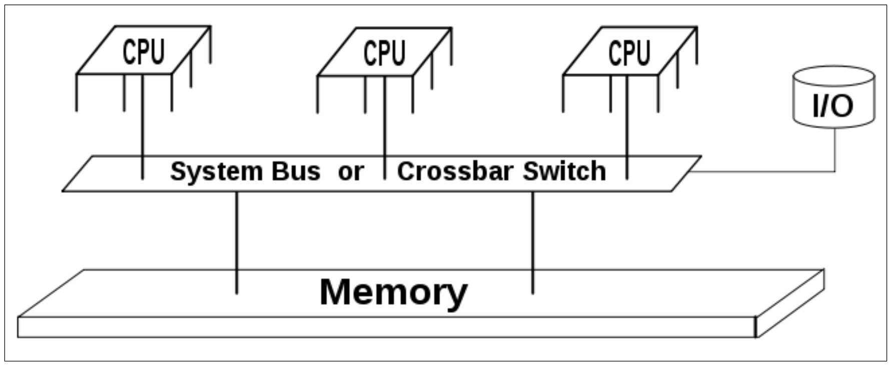

Figure 2: Typical Symmetric Multiprocessing System Layout [6]

AMP is the employ of more than one CPU with a specified role with each kernel running exclusively. This means each processor shares the same physical memory, but have independently running OS on each core. With this method, processes can run on either processor. Figure 3 shows a typical AMP system layout. [7]

Figure 3: Typical Asymmetric Multiprocessing System Layout [8]

There are heterogeneous and homogeneous processor systems. A heterogeneous system is the use of different processor cores with one for general-purpose work and the other for such things as DSP. The advantage of this approach is the ability to match processor cores with features that are appropriate for on-chip tasks applications. Tailoring a processor core to a specific task forces the processor to limit its number of abilities to no more than are required by removing unneeded features from each processor. The heterogeneous design needs a different software-development tool set, which would include a compiler, assembler, debugger, instruction-set simulator, and OS for each of the different processor cores used in the system design. [9] Heterogeneous processors generally do not use SMP due to the processors not being capable of executing identical instructions from the same copy in memory.

Homogeneous processors, such as Xilinx's Zynq EPP with two ARM cores, run the same code from a single copy in memory using SMP. These processors can also be used with AMP creating a more independent processor. In this case each processor can run different code from its personal local memory. AMP with the Xilinx Zynq EPP, for example, could utilize a RTOS on one processor while running Linux on the other or a RTOS could be resident on both cores. [1]

Multicore processing can be somewhat complex and intimidating, so it is important to have an OS developed that offers ease of use independent of the system configuration. OSs can provide autonomy to process load balancing and handling which alleviates concern about how the processors are explicitly handling the workload. Some OS are designed to automatically run processes on any available processor to provide transparent mapping of multithreading on a multicore architecture. [2] Multithreading is the ability of efficiently executing multiple threads running on a single core by utilizing thread-level and instruction-level parallelism. Multiprocessing involve the use of multiple complete CPUs in a single system. These complimentary systems can sometimes be combined in systems with multiple multithreading cores. [10]

FPGAs are used to develop soft-core processors that are used for various embedded applications. The use of FPGAs as soft-core processors such as the 32-bit Xilinx MicroBlaze have some utility, but have various complications including synthesizing the various interconnects on the programmable fabric. The challenge of using FPGAs with embedded CPUs lies in the communication between the processor and PL. Using a processing platform, a processor centric design, compared to an FPGA has various advantages. In the most recent version of the FPGA architecture by Xilinx, the Vertex FX FPGA series, Power PC (PPC) cores are used as hard

Intellectual Property (IP). [11] IP is an algorithm or function that is provided to designers through licensing from software developers. These predefined functions are intended to save time by prebuilt solution for such things as processors and bus interfaces. [15]

A FPGA centric design means that the FPGA is the master and PPC is the slave. Development requires the configuration of the FPGA in order to use the CPU cores and cannot boot independently of the FPGA fabric. On the other hand, the Zynq EPP includes a Dual ARM Cortex-A9 as hard IP. This means the ARM PS is the master and the FPGA is the slave. Additionally, the CPU can boot without powering or configuring the FPGA. [11]

The Zynq EPP utilizes the AMBA AXI4 interconnects in its System-on-a-Chip (SoC) design. An embedded designer needs to understand how to utilize the hardware they are given. The bus interconnect, in any system, is the communication link between hardware. To understand how to utilize the AMBA AXI4 interconnect, we can look at previous bus architectures and methods used to handle multiprocessing systems. The Multibus is an asynchronous bus standard developed by Intel in 1974. The Multibus was designed to be robust and became a widely used industry standard in the 1980s with systems still currently operational. [12]

The Intel MULTIBUS II was designed to address the multiprocessing problem caused by the increased demands for processing power. The MULTIBUSS II was designed to improve system performance and reduce the complexity of multiprocessing systems. It introduced the mechanism of message passing to improve the performance of a system and in doing so simplified multiprocessing system implementations. The mechanism that supports this message passing is the Message Passing Coprocessor (MPC).

There are various ways to implement a multiprocessing system. Traditionally, processors can share data using the bus and a common memory area. This memory is either available globally or dual-ported into the local memory of one processor. [12] Another method for data sharing is to have a host CPU and a disk controlling with communication done through message passing. The use of the MPC in this manner is demonstrated in Figure 4.

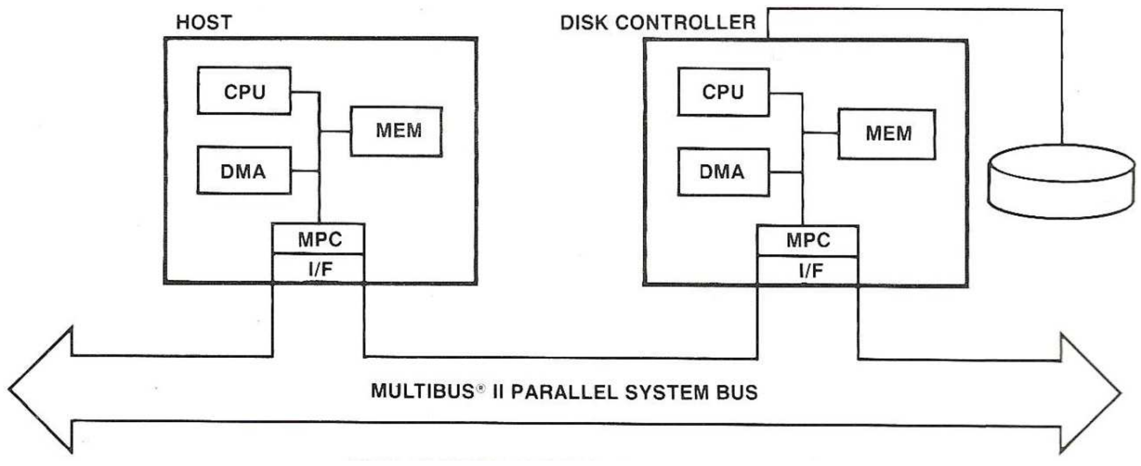

Figure 4: Message Passing using the MULTIBUS II [13]

Depending on how this message passing is implemented, the bus can become a bottleneck. However, efficiently and effectively getting data quickly into the local memory of the second CPU can achieve performance improvements. The MULTIBUS II supported all of these methods for communication.

The design objectives behind using the AMBA for SoC designs is to improve processor independence by encouraging modular system design, the development of reusable libraries for peripherals and system IP and on-chip communication that minimizes silicon infrastructure while maintaining low power and high performance. [14] These IP blocks address the various needs of embedded designers with pre-designed cores that can be implemented on Xilinx FPGA devices. [15] For example, there are IP blocks designed for Xilinx Targeted Design Platforms (TDP) provided by Xilinx and its Alliance Program Members. TDPs are development kits released with boards, Integrated Software Environments (ISE) Design Suite tools, IP cores, reference designs, and designer support for initial application development. [16]

The System Chip Design Laboratory (SCDL) is a research facility of the Department of Electrical and Computer Engineering at Temple University’s College of Engineering. SCDL was started in 1999 and is a descendant of the Advanced Processor Systems Laboratory (APSL) established in 1987. The laboratory worked with the Multibus II multiprocessor computer system which utilized the Intel iRMX III real-time multitasking operating system. The SCDL pursues innovative investigations in the SoC design methodology utilizing hard processor IP cores, configurable SoC and soft core architectures on FPGAs, on-chip busing arbitration architectures, and heterogeneous multiple processor RTOS. [38]

1.2 Research Objectives

The objective of this thesis is to develop embedded operating system support for the Xilinx Zynq EPP with multitasking FreeRTOS. Doing so will develop an understanding of effectively implementing FreeRTOS on the platform and to produce benchmark results that can be used evaluate the AMBA AXI4 interconnect performance. The Rhealstone real-time benchmark will be used to perform this benchmarking. This will provide embedded designers with a platform for further implementation on the Zynq EPP. This work will be added to the Xilinx and Zynq Evaluation & Development (Zed) board websites as a resource to inform and strengthen the Zynq community.

1.3 Organization of the Thesis

The thesis is organized as follows. A background is given in order to lay the foundation for this work. The ARM Architecture is discussed, followed by an outline of the ARM Cortex A9 architecture. AMBA is described to develop an understanding of its associations with the Dual ARM processor. The Zynq EPP architecture is discussed along with Zynq EPP platform which is utilized on both the Xilinx Zynq Evaluation Board and the Zed Board. A brief discussion about RTOSs followed by a discussion of implementing FreeRTOS is provided. The Rhealstone

Benchmark's use and implementation is reviewed and its application to the work in this thesis is discussed. The Design Tools used to develop on the Zynq EPP is review and key software and hardware design elements are discussed. The thesis results are discussed and concluded. Finally, the framework for future work with the Zynq EPP is discussed.

CHAPTER 2

BACKGROUND

2.1 ARM Architecture

2.1.1 Introduction

ARM is known for its high performance for low price and low power consumption. The reduced instruction set computer (RISC) instruction set architecture (ISA) of the ARM is not designed to produce the most powerful processor, but to create a processor capable of powering the latest technologies at a price that could be used in low-cost processing systems. The advantages of RISC stemmed from the concept that performance could be improved through smaller chip sizes with shorter signal paths implying shorter instruction cycles which results in a faster processor. A smaller die size is a result of the RISC chip being simpler and therefore requiring fewer transistors to implement the smaller instruction set. RISC was intended to shorten the design process through smaller chips with fewer instructions making the design less complicated and ultimately taking less time to complete and debug. [17]

The history of the ARM resides in the United Kingdom with Acorn Computers Ltd. ARM was established in Cambridge, originally know as Acorn RISC Machine, and developed its first ARM chip between 1983-1985. The company became popular when Acorn's British Broadcasting Corporation (BBC) Microcomputer which was widely used in UK classrooms during the 1980's. In 1985, the ARM1 was released and focused on improved instruction sets in order to improve and maximize performance of the systems using it. [17]

The Archimedes home computer launched in 1987 was the first commercial product using the ARM. It utilized the ARM2 8 MHz processor and was the first RISC processor available in a low-cost PC. The intent of these first two processors was to offer quality performance in a low-cost system. Since Intel and Motorola-based computers competed on the market with their high-end personal and workstation computer systems the ARM based systems were overshadowed. [17]

The release of the ARM3 in 1989 was designed to improve the performance of the ARM by including a 4 Kbyte on-chip data and instruction cache. This 25 MHz processor could run at a higher clock rate due to the denser fabrication of the chip compared to its predecessors and inherently improving the overall performance while using the same support chips and low cost memory as the ARM2. In 1990 the ARM2aS, a static version of the processor, added low power consumption to the list of ARM feature which opened ARM to the personal hand-held and communications devices market. Though this specific processor only reached prototyping stage of mobile devices, it sparked greater interested in RISC and the ARM family. [17]

With financial growth of Acorn and the increasing demand for RISC processors, an agreement was made between Acorn, VLSI Technology and Apple. This resulted in the foundation of ARM Ltd and the name change to the Advanced RISC Machine (ARM). ARM Ltd licenses its designs to chip foundries for royalties rather than establishing its own fabrication facilities. VLSI Technology, who had built all previous ARM chips, was the first licensee. ARM Ltd's first development after the ARM3, was the ARM6 which included full 32-bit addressing. This was designed to meet the requests of its new partner, Apple. [17] ARM, since then, has continued its growth in various avenues including its ARM Cortex A9 being used on the Zynq EPP.

2.1.2 ARM Cortex A9

ARM Cortex™-A9 processor is available as either a single core or configurable multicore processor with either synthesizable or hard-macro implementations. The ARM Cortex-A9 processor is available as a single core or MPCore model with up to four cores. MPCore is an integrated SMP or AMP with multiple processors in a single device. The Cortex-A9 processor is a power efficient, high performance option for a cost-sensitive system with power or thermal constraints. Full virtual memory capabilities are provided by the L1 cache and implemented byt the ARMv7-A architecture. It can execute 32-bit ARM instructions as well as 16-bit and 32-bit Thumb instructions and 8-bit Java bytecodes. [18] Figure 5 shows the Cortex-A9 Dual MP Core Architecture.

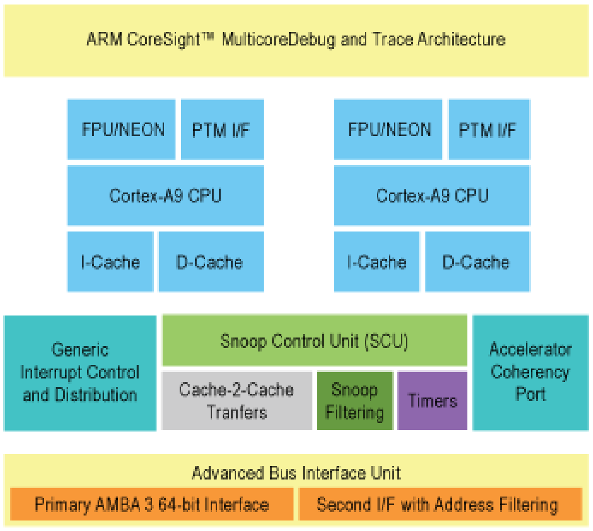
ARM CoreSight™ MulticoreDebug and Trace Architecture
FPU/NEON
PTM I/F
Cortex-A9 CPU
I-Cache
D-Cache
FPU/NEON
PTM I/F
Cortex-A9 CPU
I-Cache
D-Cache
Generic Interrupt Control and Distribution
Snoop Control Unit (SCU)
Cache-2-Cache Tranfers
Snoop Filtering
Timers
Accelerator Coherency Port
Advanced Bus Interface Unit
Primary AMBA 3 64-bit Interface
Second I/F with Address Filtering

Figure 5: Cortex-A9 Dual MP Core Architecture [19]

The processor was designed with high efficiency in mind with dual-issue superscalar, out-of-order, and a speculating dynamic length pipeline. The Cortex-A9 architecture supports 16, 32 or 64KB configurations of four way associative L1 caches and an optional L2 cache controller up to 8MB. The Cortex-A9 has physical IP available for designers. The processor comes with the ARM Development Suite 5 tools and CoreSight Debug & Trace IP. CoreSight is an on-chip debug and real-time trace kit for SoC designs utilizing ARM processors to optimize debugging the system. [20]

The Cortex A-9 MPCore with 2 cores integrated as hard IP component on the Zynq EPP is a 800-MHz dual-core processor that supports both SMP and AMP. Each processor core has a dual-issue superscalar pipeline, the NEON processing engine, a single- and double-precision floating-point unit (FPU), and 32-KB instruction and 32-KB data cache with cache coherence. The ARM Cortex-A series processors utilize NEON technology which is a 128-bit Single Instruction Multiple Data (SIMD) engine used to process multimedia formats. [21] SIMD is an extension to the architecture of the ARM providing operation extensions for registers and floating-point. [22]

The Cortex-A Series also includes a Memory Management Unit (MMU), a Snoop Control Unit (SCU), shared 8-way 512-KB associative L2 cache, generic interrupt controller, Direct Memory Access Controller (DMAC), and a 32-bit general purpose timer on the chip. [19] The ARM Cortex-A9 processor, when combined with embedded peripherals, interfaces, and on-chip Memory (OCM), create a Hard Processor System (HPS). Connecting the HPS and FPGA of the SoC with a high-bandwidth on-chip backbone provides large bandwidth for sharing data between the ARM processor and hardware accelerators within the FPGA fabric.

2.2 AMBA Bus

The ARM Cortex A9 AMBA 3 located on the chip is the backbone for communication within the SoC. The AMBA has been widely used as an on-chip bus architecture in many SoC designs. The AMBA has since exceeded its initial design potential and has gone beyond the use in microcontroller devices. The AMBA 1 consists of the Advanced Peripheral Bus (APB) and Advanced System Bus (ASB). In the second generation, AMBA 2, ARM added a single clock-edge protocol called AMBA High-performance Bus (AHB). AMBA 3, the third generation AMBA, reached higher performance interconnects by adding the Advanced eXtensible Interface (AXI). It also included the Advanced Trace Bus (ATB) which was designed to work with the CoreSight on-chip trace and debug tools. [24]

The AMBA 3 specifications replaced AMBA 2, but AMBA 2 peripherals can still be used on AMBA 3 based systems. The protocol specification of the AMBA family is an ARM open standard for on-chip buses and provides solutions to SoC interconnections and functional block management for embedded design with multiple processors and multiple peripherals. [23] The AMBA 3's interface protocol specification encompasses all the required on-chip data traffic requirements. These requirements include high data throughput from data intensive processes, low bandwidth communication with low power and gate count, and on-chip testing and debugging. [24]

AMBA 3's AXI is an interface and a protocol, but is not a bus. There is no bus arbitration because it is utilizes point to point connections. [53] AXI provides support for data traffic throughput with five unidirectional channels and out-of-order data transaction capabilities. This allows for high speed operations through the pipelined interconnections, simultaneous reading and writing transactions, and efficient high latency peripheral support and bridging between frequencies for power management. The AHB interface enables high efficiency interconnects between single frequency subsystems of simpler peripherals when the AXI is not need. The structure of the AHB is a fixed pipelined and an unidirectional channel allows for back compatibility with AMBA 2 peripherals. [24]

APB provides low bandwidth transaction support to access necessary configuration registers in peripherals as well as data traffic in peripherals with low bandwidth. This interface is highly compact and low power isolates data traffic from the AHB and AXI high performance interconnects. ATB adds a data trace interface for data diagnostics in a trace system. This provides debugging capabilities due to the trace components and bus sitting in parallel with interconnects and peripherals. [24]

2.3 Zynq Extensible Processing Platform

2.3.1 Introduction

The Zynq EPP 7000 family of devices combine the hardware programmability of an FPGA and the software programmability of a processor. The overview of the hardware is depicted in Figure

6. The Zynq EPP platform's PS includes the Dual ARM Cortex-A9 MPCore that utilizes 32kB instruction and data L1 cache per core, shared 512kB L2 cache, FPU and NEON media engine. The memory interfaces include 256kB OCM in addition to NAND Flash and NOR Flash Memory Controller which includes DDR2, LPDDR2, and DDR3. Peripherals include Queued Serial Peripheral Interface (QSPI), USB2.0, GbE, CAN, SDIO, Universal Asynchronous Receiver and Transmitter (UART), SPI, I2C, General Purpose I/O (GPIO), 12bit 1 Mbps ADC, AES and SHA-256. [25]

There are four available models of the Zynq EPP designed for various applications. The available FPGA types for each of the model types include the Artix-7 for Z-7010 and Z-7020 and the Kintex-7 for Z-730 and Z-7045. The FPGA sizes vary and include logic cells that range from 30k-350k, block RAM ranging from 240kB-2,180kB, DSP Slices from 80-900, and user I/Os of 150-400. The Kintex-7 devices also have eight PCI Express2 and 12.5 Gbps Transceivers. Quick EMUlator (QEMU), a virtual platform, is used for the model of the processing subsystem. [25]

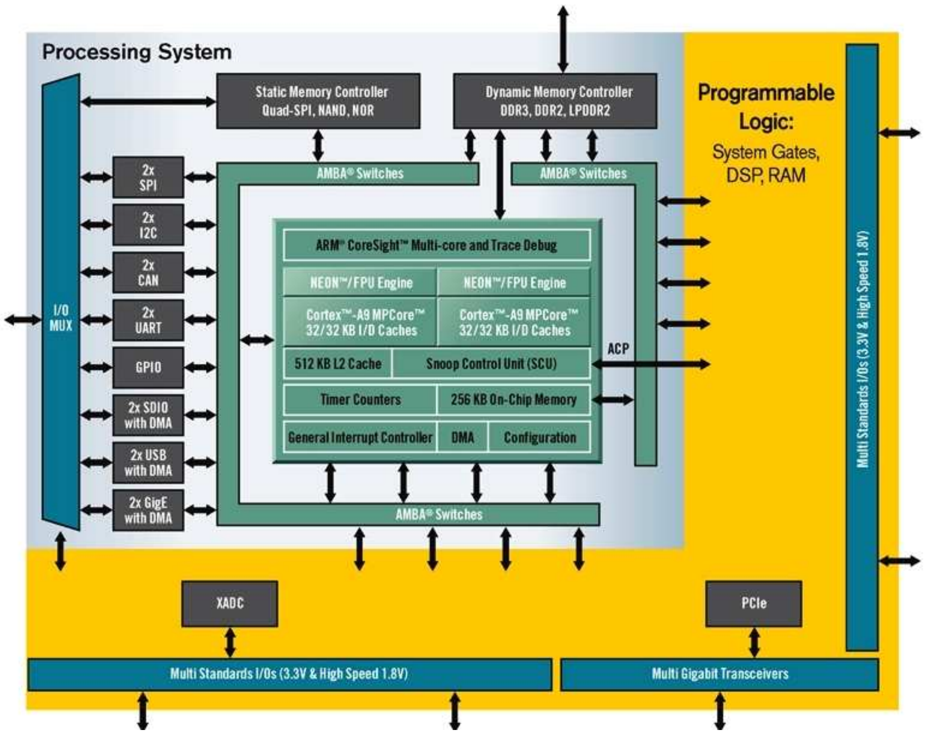

Figure 6: Xilinx Zynq-7000 Extensible Processing Platform Architecture [26]

Xilinx implemented AMBA on the Zynq as two switch matrices. The AMBA AXI interconnect exists in two areas on the Zynq EPP. One is grouped around the DRAM controller and the other is used for general peripherals. There are two switches on the peripheral side. The first has one connection for a hard static memory controller, eight hard I/O controller blocks, five connections for the CPU cluster, and four stubs that end at the programmable fabric. The second has five connections ending in the fabric, two connections for the hard DRAM controller, and two CPU ports. [27]

One of these five ports supports the Accelerator Coherence Port (ACP). This port provides the ability for the accelerator to snoop the processor cluster's caches, but not the cluster's OCM so a CPU task could leave a control and data block in cache. From here, an accelerator in the programmable fabric can read the block directly from cache and therefore avoiding a write-back to DRAM. This protocol is not symmetric and therefore the accelerators are not fully coherent. This is because the CPU reads and writes do not snoop memory in the fabric. The AMBA I/O ports, the DRAM controller accessible AXI ports, and ACP provide the Zynq EPP with a range of programmable fabric structure design possibilities. The current available hardware platforms include the Xilinx Zynq-7000 ZC702 Evaluation Kit, the Xilinx Zyqn-7000 EPP Video Kit, and the Zynq-7000 EPP ZedBoard. [27]

2.3.2 Xilinx Zynq-7000 Evaluation Kit

The Xilinx Zynq-7000 ZC702 Evaluation Kit is a kit from Xilinx that includes a silicon board with the Zynq EPP, development tools, IP, and a variety of reference designs. An image of the board is shown in Figure 7. It provides abundant I/O expandability for embedded designers to develop upon. It is also backed with OS support and by the ARM community. The kit is provided with the XC7Z020-1CLG484CES device Zynq chip, design suites, various cables for scoping the board, 8 GB SD card that contains the provided Linux startup kernel, and documentation with step-by-step guides. It also contains all schematics and PCB files and design examples. [28]

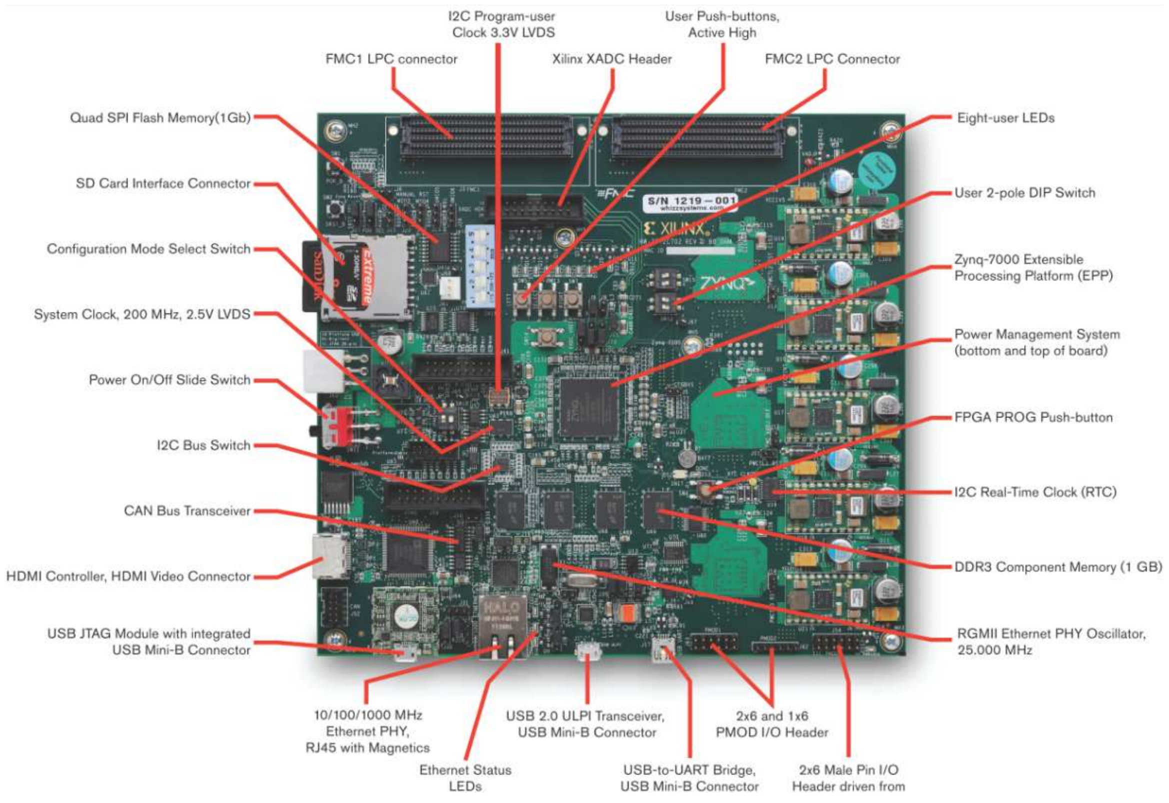
I2C Program-user Clock 3.3V LVDS
User Push-buttons, Active High
FMC1 LPC connector
Xilinx XADC Header
FMC2 LPC Connector
Quad SPI Flash Memory(1Gb)
Eight-user LEDs
SD Card Interface Connector
User 2-pole DIP Switch
Configuration Mode Select Switch
Zynq-7000 Extensible Processing Platform (EPP)
System Clock, 200 MHz, 2.5V LVDS
Power Management System (bottom and top of board)
Power On/Off Slide Switch
FPGA PROG Push-button
I2C Bus Switch
I2C Real-Time Clock (RTC)
CAN Bus Transceiver
DDR3 Component Memory (1 GB)
HDMI Controller, HDMI Video Connector
USB JTAG Module with integrated USB Mini-B Connector
RGMII Ethernet PHY Oscillator, 25.000 MHz
10/100/1000 MHz Ethernet PHY, RJ45 with Magnetics
USB 2.0 ULPI Transceiver, USB Mini-B Connector
2x6 and 1x6 PMOD I/O Header
Ethernet Status LEDs
USB-to-UART Bridge, USB Mini-B Connector
2x6 Male Pin I/O Header driven from

Figure 7: Zynq 7000 Evaluation Kit [28]

2.3.3 ZedBoard

The ZedBoard is a community driven approach of the Zynq EPP by Silica and Digilent. An image of the board is shown in Figure 8. The concept behind the board is to be designed in an open source community manner. The board contains various peripherals with extension options that include a FPGA Mezzanine Card and peripheral modules using the Peripheral Module (Pmod) connector to connect components such as an ADC, DAC, Sensors, Switches, Displays, RF, WiFi, Bluetooth, or Storage. The ZebBoard website, ZedBoard.org, is where all the collaboration material is maintained. [29]

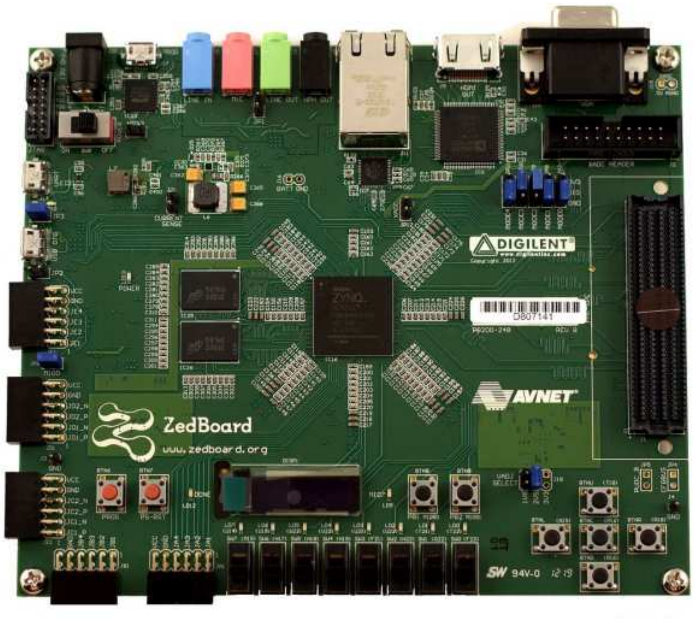

Figure 8: ZedBoard [30]

2.4 Real-Time Operating Systems

An OS is an abstraction of hardware in a system that provides an interface for servicing applications. The OS replaces the direct interface to hardware with program functionalities a user of the system wants or needs. It supports the basic functions of a computer system and makes the system easier to maintain, faster, and easier to write applications. When designing an OS various parameters are considered including performance, resources management, security, marketability, and failure tolerance. It is responsible for managing hardware and software resources. Hardware resources include processors, memory, and I/O devices. Software resources include programs and data files.

An OS is comprised of layers that create an environment that hides and simplifies the underlying hardware by providing sets of commands to meet the user's needs. Though the structure of the OS kernel can vary, they all attempt to provide the user with a platform in which to utilize the system hardware. Many OSs make multiple programs and processes appear to run at the same time through multitasking. However, a processor can only handle one thread of execution at a time. A scheduler is used to manage the processes executing on the processor and to create the illusion of simultaneous execution through a process call time slicing and rapidly switching between program threads. [31]

The type of OS can be defined by its scheduler and how it decides which process threads to run and for how long. A multi user OS, like Unix, will ensure processing time is shared equally between users. A desktop type OS, like Windows, has a scheduler that ensures that the system remains responsive to users when needed. A RTOS's scheduler is designed to provide predictable execution patterns to systems that have real time requirements. Embedded systems often have these demands and means the system must respond to a given event within a strictly defined deadline. This means that the OS's scheduler must be deterministic in order to predict the real time requirements of the system. [32]

FreeRTOS uses a traditional real time scheduler by allowing the user to assign a priority to each thread to determine execution. This scheduler, based on the priority, knows which thread of execution to run next. FreeRTOS is a versatile class of RTOS designed for many applications including being implemented on small microcontrollers. FreeRTOS is designed for systems that do not require a full RTOS implementation, many times in the design of embedded applications, or do not have the ability to run a full RTOS. FreeRTOS only provides the core real time scheduling functionality, inter-task communication, and timing and synchronization primitives and would more accurately be referred to as a real time kernel. If additional functionality is required, they can be included as add-on components. [33]

2.5 FreeRTOS

FreeRTOS is a RTOS from Real Time Engineers Ltd written in C and, as of October 2011, supports 31 processor architectures. FreeRTOS is a lightweight real-time kernel designed for small embedded systems that require deterministic and real-time responsiveness to system events.

Lightweight means it is a less complex OS with a basic instruction set designed to be faster and not as heavily resource dependent. Key features of FreeRTOS include an Application Programming Interface (API), message passing, binary and counting semaphores, mutual exclusion with priority inheritance, pre-emptive scheduling, co-operative scheduling, and round robin with time slicing. Round robin is a simple scheduling algorithm for process time slicing in which each process is assigned equal portions of execution time and in circular order. [33]

With the growing complexity in embedded design due to the availability of more memory and various communication peripherals, there is an inherent increase in software complexity. The inherent benefit of using an OS kernel is clear. FreeRTOS is free and is released to its users as open source. FreeRTOS implements its open source by releasing moderated versions instead of pure open source. This ensures that only software originated by FreeRTOS is used in the official release. There are, however, community contributed files that are separate and available as open source. FreeRTOS's license model is designed around the idea that code on the application side that uses FreeRTOS remains closed, while code that modifies or extends the kernel itself is open source. [34]

FreeRTOS supports several Xilinx products including Microblaze, PowerPC, and the Zynq. Microblaze, which is a 32-bit soft processor core port, runs on various Xilinx FPGA's including the Spartan-6 and Virtex5. PowerPC 405/440 are configurable processor cores that run on Virtex4 and Virtex5 FPGA's repetitively. The initial release of the FreeRTOS is available for the Zynq in October 2011. The original port was for the Xilinx Zynq EPP and was developed to run on the Zynq 7000 EPP based ZC702 board and implemented on version 14.1 of the Xilinx ISE Design Suite. As Xilinx releases newer versions of their design suites, the ports are updated and released accordingly. [35]

2.6 Rhealstone: Real-Time Benchmark

The Rhealstone Benchmark was proposed by Rabindra P. Kar and Kent Porter in 1989 and was designed to be a metric for comparing the performance of real-time multitasking systems independent of any features found in any CPU, bus architecture, or a specific OS or kernel. [36] At that time, there was the Whetstones and Dhrystones that benchmarked code generated by compilers and the throughput of hardware platforms, but no equivalent measurement for real-time systems. Rhealstone was a proposed standard for objectively measuring real-time performance and summarizing the components of performance.

The Rhealstone metric mainly helps embedded developers select real-time systems appropriate for a specific application. It should be noted that an encompassing real-time solution would consist of the system, the application software, and external devices, so Rhealstones doesn't measure the quality of the complete solution, but instead a measurement targeted specifically toward a multitasking solution. The scope of Rhealstones is with complex systems running five to thirty concurrent processes. It will be adopted to be used with a multitasking AMP system.

The Rhealstone takes into account that all real-time applications are unique. One system may be highly interrupt-driven while another relies heavily on message-passing among tasks or another that fights for resources. The Rhealstone figure is a sum obtained from six categories of activity most crucial to the performance of real-time systems. The categories include task switching, preemption, interrupt latency time, semaphore shuffling, deadlock breaking, and intertask message latency time. It uses coefficients that the system designer assigns weight to each Rhealstone component based on relative importance.

2.6.1 Task Switching Time

Task switching time is the average time the system takes to switch between two independent and active, not suspended or sleeping, tasks of equal priority. Task switching is synchronous and nonpreemptive and is an important measure of any multitasking system. This metric is influenced by the host CPU's architecture, instruction set, and features and is designed to assess the compactness of task control data structures and the efficiency with which the executive manipulates the data structures in saving and restoring contexts. Task switching time, additionally, measures the executive's list management capabilities. [36] A demonstration of this performance parameter is shown in Figure 9.

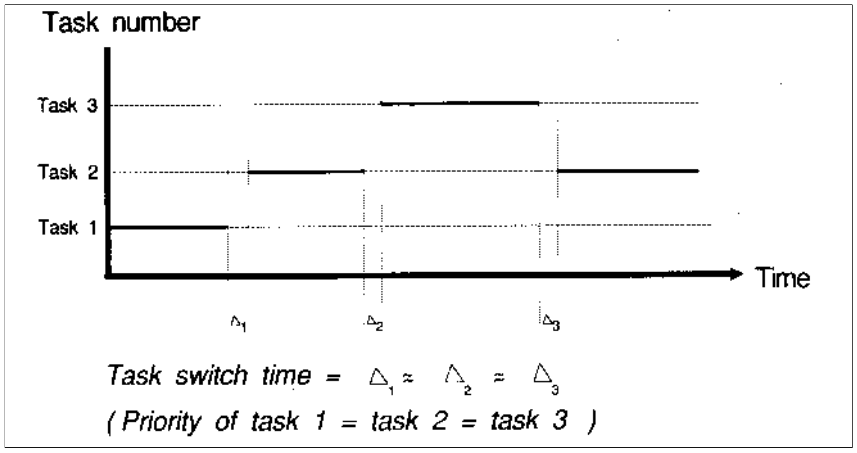
Task number
Task 3
Task 2
Task 1
Time
Δ_1
Δ_2
Δ_3
Task switch time = $\Delta_1 \approx \Delta_2 \approx \Delta_3$
(Priority of task 1 = task 2 = task 3)

Figure 9: Rhealstone Benchmarking: Task-Switching Time [37]

2.6.2 Preemption Time

Preemption time is the average time it takes a higher-priority task to take control of the system from a running task of lower priority and usually occurs when the higher-priority task moves from an idle to a ready state in response to some external event. In other words, it is the average time the executive takes to recognize an external event and switch control of the system from a running task of lower priority to an idle task of higher priority. A demonstration of preemption time is shown in Figure 10. Preemption and interrupt latency, which is discussed next, can be considered most significant real-time performance parameter since multitasking systems assign task priorities and even dynamically through applications. [36]

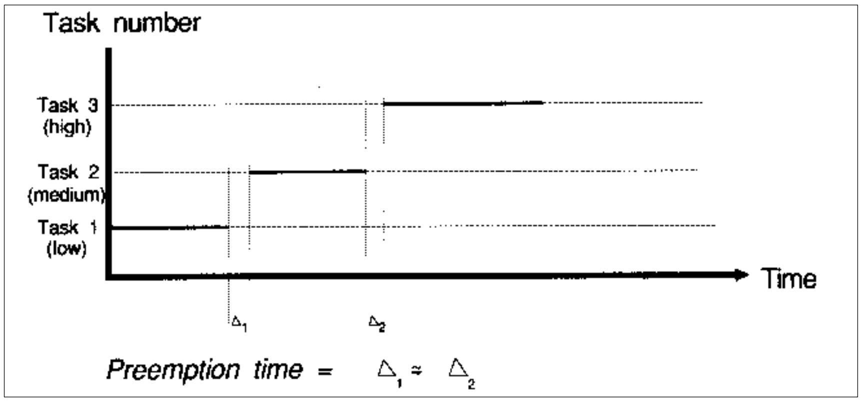

Figure 10: Rhealstone Benchmarking: Preemption Time [37]

2.6.3 Interrupt Latency

Interrupt latency, shown in Figure 11, is the time between the CPU's receipt of an interrupt request and the execution of the first instruction in the interrupt service routine. Its reflected by the delay introduced by an executive and the processor and not delays occurring on the bus or interfaces to external devices. [36]

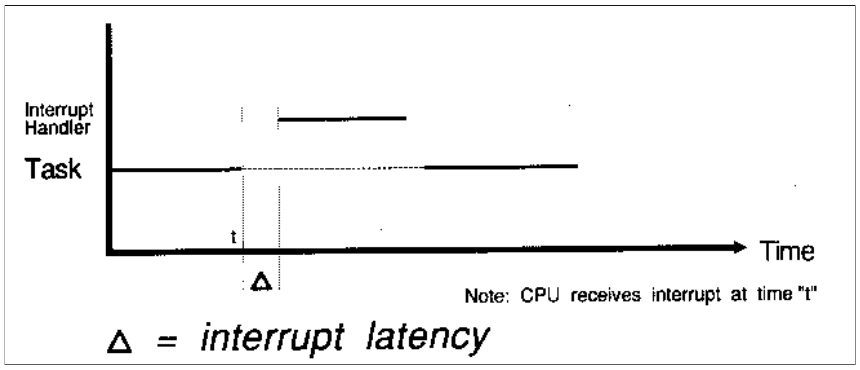

Figure 11: Rhealstone Benchmarking: Interrupt Latency [37]

2.6.4 Semaphore Shuffling Time

Semaphore shuffling time is the delay between a task's release of a semaphore and the activation of another task blocked on the "wait semaphore" primitive. When implementing this, at least three tasks with different priorities should be active and no other tasks should be scheduled in between. The semaphore shuffling time measures the overhead associated with mutual exclusion. This occurs when multiple tasks compete for the same resources. Semaphore based mutual exclusion provides a way of ensuring that a nonshareable resource only serves one master at a time. [36] Semaphore shuffling time is shown in Figure 12.

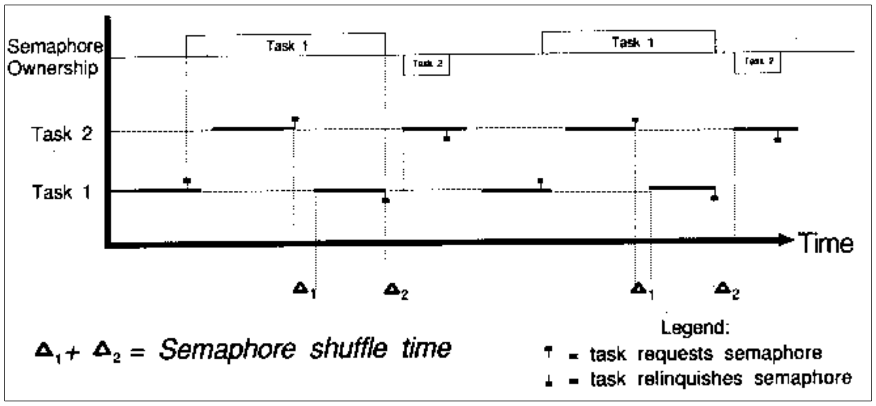

Figure 12: Rhealstone Benchmarking: Semaphore-Shuffle Time [37]

2.6.5 Deadlock Breaking Time

Deadlock breaking occurs when a higher-priority task preempts a lower-priority task that holds a resource needed by the higher-priority task and the metric measures the average time it takes the executive to resolve this conflict. Deadlocks are a common multitasking problem and are sometime not handled effectively. This can be solved by temporarily raising the priority of the running task above that of the interrupting task until the needed resource is released by the lower-priority task. The temporary priority is then lowered so the new task can run. Deadlock breaking, shown in Figure 13, is the sum of times required to resolve an ownership dispute between a low-priority task holding a resource and a higher-priority task that needs it. [36]

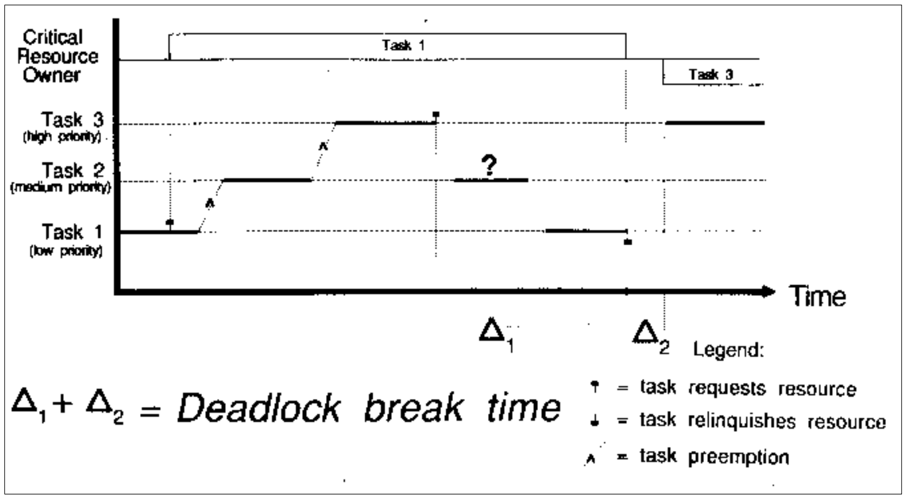

Figure 13: Rhealstone Benchmarking: Deadlock-Break Time [37]

2.6.6 Intertask Messaging Latency

Intertask message latency, demonstrated in Figure 14, is the delay within the executive when a nonzero-length data message is sent from one task to another. In order to measure it properly, the sending task should stop executing immediately after sending the message and the receiving task should be suspended while waiting for it.

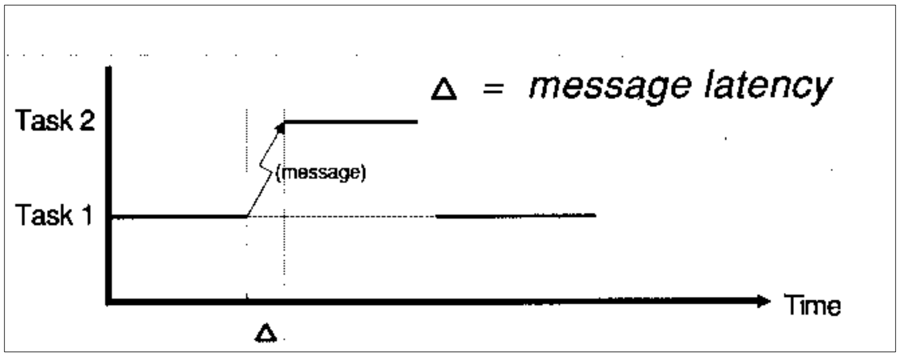

Figure 14: Rhealstone Benchmarking: Intertask Message Latency [37]

The intertask message-passing link must be established at run time and if multiple messages are sent on the same link, the receiving task gets a chance to read an old message before the sending task can overwrite it with a new one. This can be handled with various mechanisms such as pipes, queues, and stream files which are usually provided by multitasking executives for intertask data communication. [36]

2.6.7 Calculating the Rhealstone Performance Number

The measurement of the six performance categories provide embedded designers with a well rounded analysis of the system performance. Rhealstone also makes it easy to compare systems by generating a single real-time value. All of the benchmarks must be first represented in seconds (t1-t6). Then, added together and the average of them found. The number is then inverted to get the Rhealstone performance number that is represented with the units Rhealstones/second as shown in Equation 2.1. [37]

$$Rhealstone\ Performance\ Number = \left(\frac{(t_1+t_2+t_3+t_4+t_5+t_6)}{6}\right)^{-1}_{Rhealstones/Sec} \qquad (2.1)$$

The above performance number is a method to compare systems on a general level by considering all the parameters to be equally occurring. If an embedded designer needs to evaluate a system based on a specific category, for example, an application that is heavily interrupt-driven a weight can be chosen before calculating the performance number. This method is referred to as “application specific Rhealstone” and is shown as Equation 2.2. [37]

$$\begin{matrix} Application\ Specific\ Rhealstone \\ Performance\ Number \end{matrix} = \left(\frac{(n_1t_1+n_2t_2+n_3t_3+n_4t_4+n_5t_5+n_6t_6)}{n_1+n_2+n_3+n_4+n_5+n_6}\right)^{-1}_{Rhealstones/Sec} \qquad (2.2)$$

Nonnegative real coefficients (n1-n6) for each category are set based on occurrence within the application. If interrupts occur 5 times more than task switching, its coefficient should be 5 times larger. Similarly, if a category does not happen at all, the coefficient is set to zero. For example, if there is no inter-task message passing performed by the application, its coefficient should be set to zero. The application specific Rhealstone Performance Number is then again calculated by inverting the average. [37]

CHAPTER 3

DESIGN TOOLS

3.1 Introduction

Complex embedded systems require powerful and well developed design tools. With an embedded system such as the Zynq EPP, embedded engineers are faced with complex design projects that have both hardware and software design problems. Using an FPGA in the design makes the system even more complicated and combining each individually designed subsystem into one complete system is again a difficult task. With the Zynq EPP and the addition of the ARM dual core as Hard IP, Xilinx has developed a set of design tools that manage this complexity and help make the design process as simple as possible. The broad array of development system tools provided by Xilinx is collectively called the ISE Design Suite. The Xilinx ISE Design Suite 14 is the current version used for designing on the Zynq-7000 All Programmable SoC platform. [39]

3.2 Xilinx ISE 14

The Xilinx ISE Design Suite is the current development tool set used to design every aspect of the Zynq-7000 All Programmable SoC. There are currently three editions, the Logic, Embedded, and DSP, of the ISE Design Suite and are all included as part of the System edition. [40] The Xilinx ISE Design Suite 14.2 Embedded Edition was used for development on the Xilinx ZC702 Rev C Evaluation Board for this thesis. Xilinx Vivado is the next generation of this design suite and will be replacing the Xilinx ISE for future Xilinx products. The first generation of Vivado did not support the Zynq EPP, but will support it in the future.

The Embedded Edition of the ISE Design Suite includes the PlanAhead design analysis tool, ChipScope Pro, and the Embedded Development Kit (EDK). The EDK consists of the Xilinx

Platform Studio (XPS) and the Software Development Kit (SDK). [39] Aside from the ISE, Tera Term was also used for the design process. Figure 15 depicts the block diagram for the software packages within the ISE Design Suite and how they interact with each other. PlanAhead is the initial development tool for starting an embedded design. Planahead works with the EDK to design the hardware and software system.

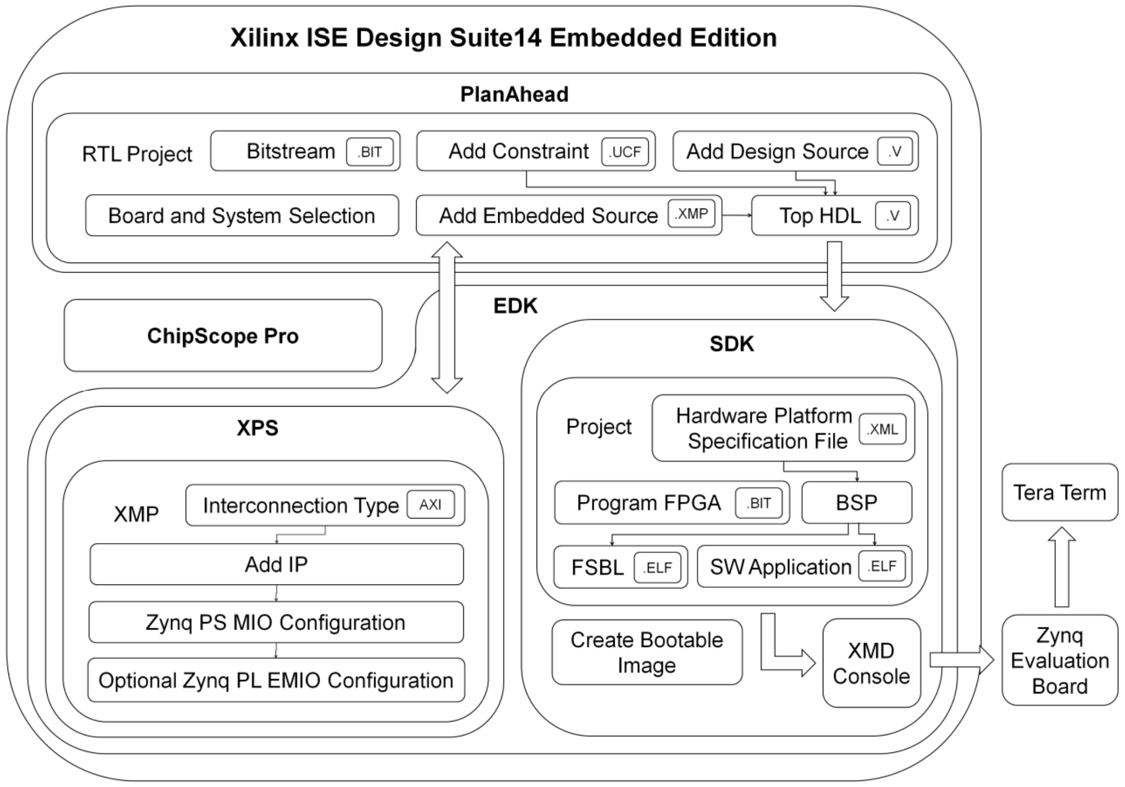

Figure 15: Design Tools Block Diagram – Xilinx ISE 14

3.3 PlanAhead

The PlanAhead design and analysis tool is used to add various hardware sources and manage the link between the hardware and software design aspects of the project. It helps with FPGA I/O assignments and advanced FPGA layout planning to optimize the connectivity between the PCB and FGPA. [40] PlanAhead allows the embedded designer to create a project with an embedded processor system as the top level and works with the EDK to design the embedded system. The hardware system is created using XPS and imported back into the PlanAhead project. The

PlanAhead project is then exported to the SDK to develop software for the hardware design that was just created.

When PlanAhead executes, it allows the embedded designer to create a new project or open an existing one. A Register Transfer Level (RTL) project is created to begin the design in PlanAhead. The RTL Project allows the embedded designer to add sources, generate IP, and run an RTL analysis. The designer starts by setting up the type of hardware board the project is being design for. For this thesis, the Zynq ZC702 Evaluation Board was selected. PlanAhead is then used to import various sources into the project. It can add constraints such as a User Constraint Files (UCF) which specifies how the logical design constraints are implemented on the target device [54], add design sources such as the HDL Verilog, or an Embedded Source for setting up the PS peripherals and various other settings. PlanAhead also generates the bitstream's bit file for programming the PL in the SDK. [51] Figure 16 shows how the project files of PlanAhead interact with the rest of the ISE Design tools.

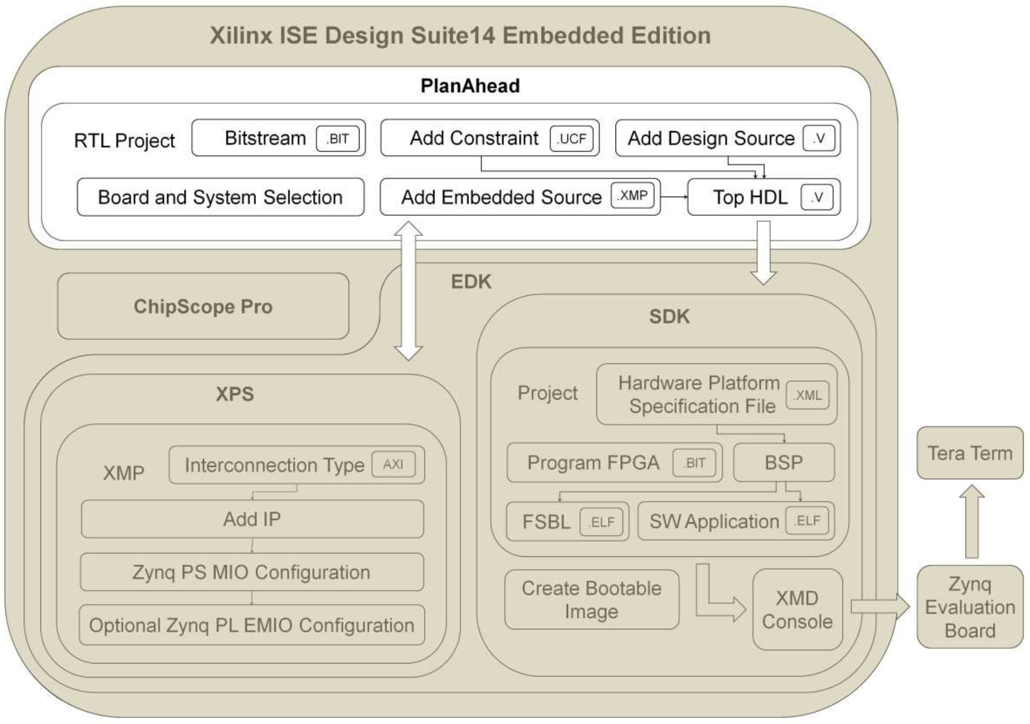

Figure 16: Design Tools Block Diagram – PlanAhead

When an embedded source is added to the project, it recognizes that an embedded processor system was created and starts XPS to setup the added source. When the designer is finished with XPS, the design is updated in the PlanAhead tool. From here the embedded processor system can be created as the top level of the system by creating a Top HDL with Verilog. The entire project is then exported to the SDK. [39]

3.4 Embedded Design Kit

The EDK is used to design a complete embedded processor system for implementation on a Xilinx hardware device. It assists designers in hardware and software application design, debugging, and execution. The design can be run on the destination boards for verification of a working design. The EDK includes hardware IP, drivers and libraries, and GNU compiler and debugger for C/C++ software development for the ARM Cortex-A9MP processors in the Zynq PS. It also provides documentation and sample tutorial projects for understanding the basics. [39] The tool kits included in the EDK are the XPS and SDK. Within these two kits are various tools including the Base System Builder (BSB) Wizard, Xilinx Microprocessor Debugger (XMD) and GNU Software Debugging Tools, Simulation Model Generation Tool (SimGen), Create and Import Peripheral Wizard, GNU Software Development Tools, Library Generation Tool (LibGen), Bitstream Initializer (BitInit), and the Hardware Platform Generation Tool (PlatGen). [40]

3.5 Xilinx Platform Studio

XPS provides a development environment for designing the embedded PS’s hardware. XPS is primarily used for setting up the processor, peripheral, and interconnection configurations for the embedded processor hardware system. It’s designed to make it easy to add desired IPs and create port connections for components like the clock and reset. The XPS project can be designed from the ground up using a blank project or the BSB wizard can be used to add default peripherals to the fabric and to automatically select a default configuration for the PS I/O interface. After the BSB is used, the Zynq EPP PS block diagram is displayed in XPS. This allows the designer to click on any of the configurable green blocks and make configuration changes. The configuration process of XPS is shown in Figure 17.

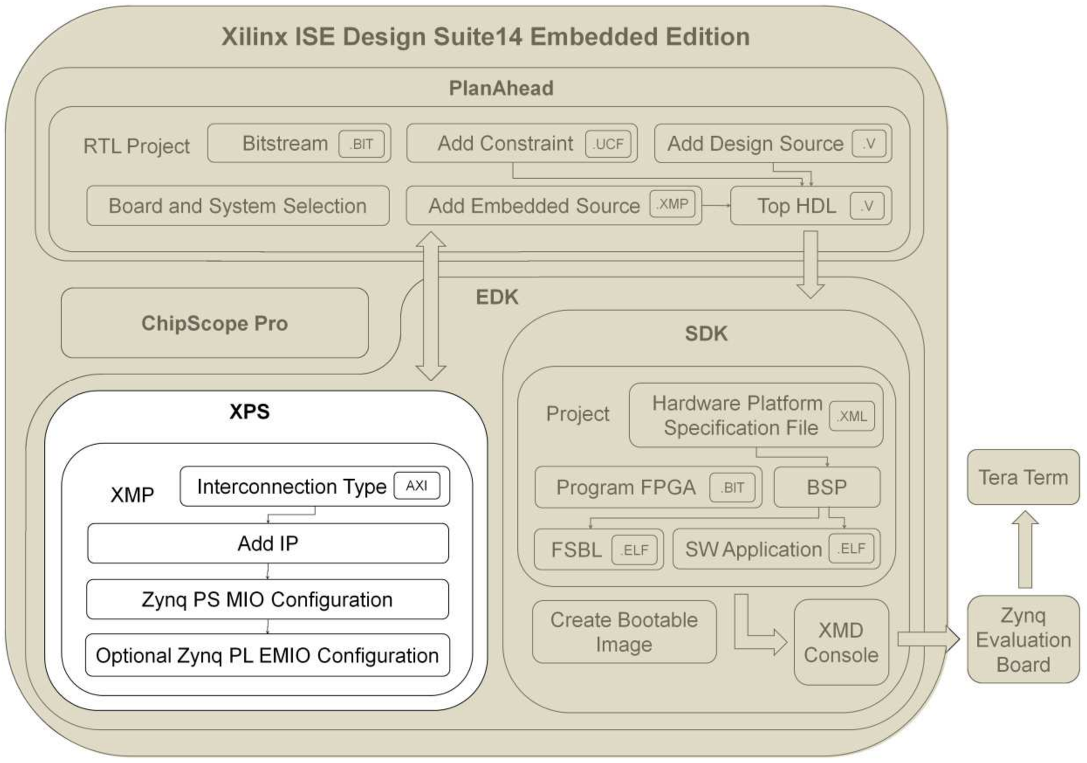

Figure 17: Design Tools Block Diagram – XPS

XPS creates hardware platform information in the Xilinx Microprocessor Project (XMP) file format. [51] This file includes information about the PS configurations including GPIO such as MIO and Extended MIO (EMIO), and adds IP and information about configuring the PL in PlanAhead. Closing XPS will update the currently open PlanAhead session.

3.5.1 Base System Builder Wizard

The BSB wizard is part of the XPS and prompts the designer to choose whether they want assistants in setting up the basic configurations of PS. The BSB helps create a working embedded design for the evaluation board quickly by setting up basic features and common functionality automatically. After setting up the basics of the system, XPS and other ISE software tools can be used to perform system customization. The first aspect of the design the BSB wizard sets up is what type of interface is going to be used whether it be AXI or Processor Local Bus (PLB) which is an old interface standard used by Xilinx. The BSB then needs to know what type of board the system is being design for. Fortunately, this information was imported from PlanAhead and if it wasn't the correct board setup can be selected. [39] The BSB closes and now allows the designer to customize the existing design.

3.5.2 AXI Interconnection

The AXI bus interface IP cores started being used by Xilinx with their Spartan-6 and Virtex-6 hardware devices. An AXI system interface comes with standard Xilinx IP and tool flows and will be the standard interface used for all current and future versions of Xilinx products. The PLB system is a legacy bus standard used by Xilinx FPGA families up to the Spartan 6 and Virtex 6 and is not supporting newer FPGA families. This means it is not suggested to start new projects with PLB if they will be used on new Xilinx platforms. [43] The AXI specification is in charge of providing a framework for defining protocols for moving data between IP. It does this using a defined signaling standard. The AXI standard is responsible for making sure that IP can exchange data is moved across a system properly. [42] The AXI and other IP can be added to the PS design to create a custom embedded system.

3.5.3 Hardware Platform Configuration

The Zynq's PS can be configured in various ways. When the BSB is finished setting up the basic system, the designer is provided with the Zynq EPP processing platform configuration tab shown in Figure 18. This tab allows the designer to configure I/O peripherals, clocks, memory, and other aspects of the PS. The green blocks are customizable portions of the PS.

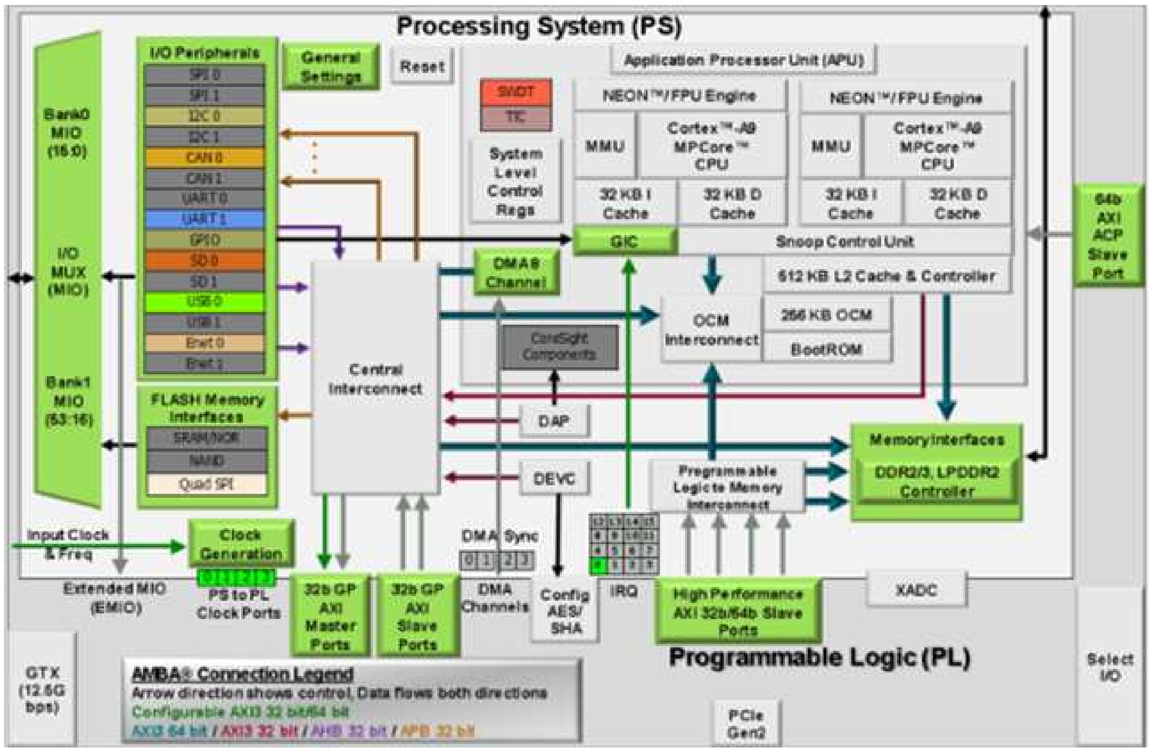

Figure 18: Processing Platform Peripheral Configuration – XPS

Various IP can be added to the PS using the bus interface tab. Once the peripherals are added to the system, the ports tab is used to setup the I/O peripherals and clocks. There are 54 MIO that can be used by the PS. If more I/O is required or the designer wants to utilize the PL, the I/O can be setup as EMIO for use by FPGA fabric. [53] Once the PS is configured, XPS is exited and the design is updated in PlanAhead and ready to be exported to the SDK.

3.6 Software Development Kit

The SDK is used for developing the software design for the embedded project. The SDK is used for C/C++ embedded software application creation and verification of software application projects and was built on the Eclipse open-source standard framework. [40] The SDK provides tools for software project management and gives access to the GNU toolchain for code compilation and debugging. It can be used to run applications on the target hardware board and create bootable images. The FPGA fabric can also be programmed when needed. Figure 19 shows the relationship between the files within the SDK.

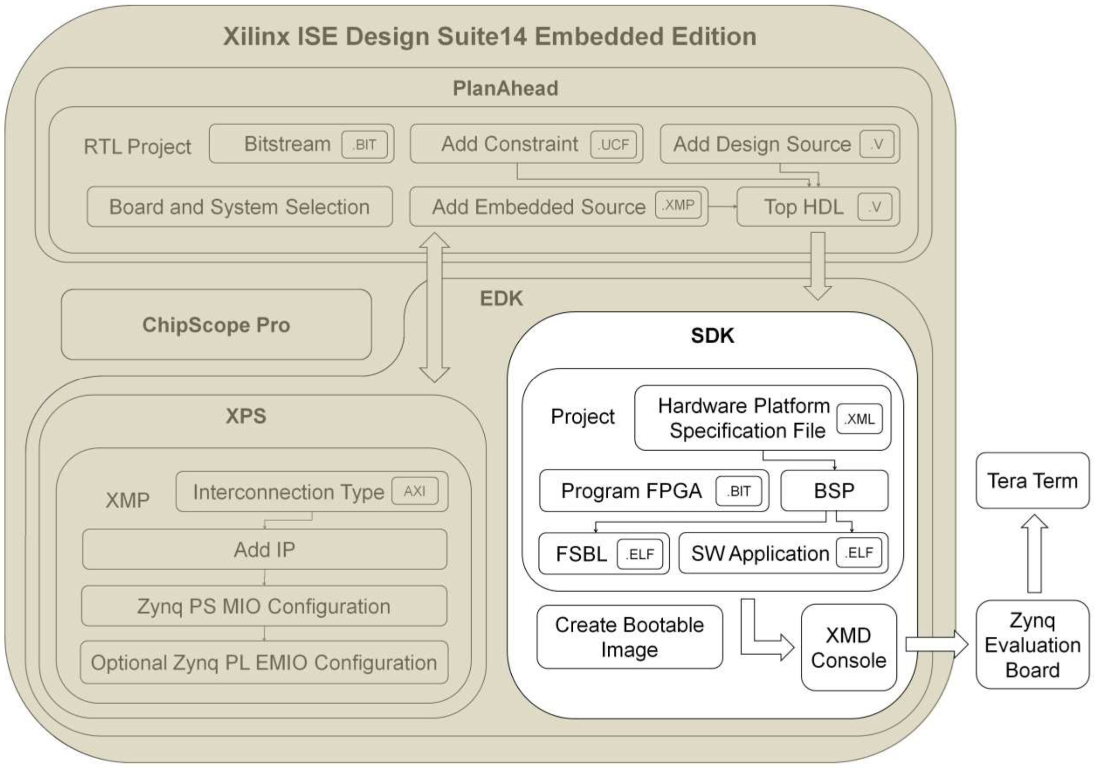

Figure 19 Design Tools Block Diagram – SDK

The PlanAhead design tool exports the hardware platform specification files from XPS to the SDK. These files include the XML, Microprocessor Hardware Specification (MHS), and the ps7_init.c, ps7_init.h, ps7_init.tcl, and ps7_init.html files. The XML file is the main file used for setting up the First Stage Boot Loader (FSBL) and Board Support Package (BSP). The MHS file contains information about the interconnects between the PS and PL. Ps7_init.c, ps7_init.h, ps7_init.tcl, and ps7_init.html files are internal configuration files containing information on the Zynq EPP peripheral configurations. The ps7_init.c and ps7_init.h files contain settings for DDR, clocks, plls, and MIOs to initialize the Zynq EPP PS. The SDK uses these specified settings so that applications can be run on top of the PS. It should be noted that here are some settings of the PS that are fixed for the ZC702 evaluation board and cannot be changed. [44]

3.6.1 Board Support Package

A BSP is created using the SDK from the files imported from PlanAhead. The BSP is the support code for a board or hardware platform which helps with initialization during power up as well as provides support for software applications to run on top of. The BSP is usually specific to the OS and one is needed for each of the cores of the processor. [39] It is a collection of libraries and support drivers that form the application's lowest layer of the software stack. A BSP must be created before a designer can create or use a software application by linking against it or running on top of the software platform. It does this by using the API that the BSP provides. [47] Multiple BSPs can be used in the same SDK workspace.

3.6.2 Xilinx C Project

The SDK allows for application development of C/C++ programs. For this thesis, the C programming language was used. The C program can be compiled with the SDK and an Executable and Linakable Format (ELF) file is generated. This file is used to execute on the processor. The SDK provides a basic Hello world example to understand the basic of programming the PS. The ELF file is also used in creating a bootable image for running on the hardware device. All application development for this thesis was done in C.

3.6.3 First Stage Boot Loader

The FSBL starts after the device boots and is loaded into the OCM. It is responsible for initializing the PS configuration exported from XPS. The FSBL always runs on CPU0 and is the first software application that is executed. It is used to initialize peripherals, programming the PL, load a second stage bootloader, or load the application ELF file. The version of FSBL included in the ISE Design Suite does not support multiple data or ELF file. This is because the FSBL searches for a bit file. If a bit file is found, the FSBL writes it to the PL. The FSBL then loads one application ELF file into memory and executes it. If AMP is desired, the FSBL must be modified so it continues to search for files. [52] The FSBL's ELF file can be stitched with the bitstream to create a Boot Image File (BIF) using the Bootgen application. The create boot image wizard in SDK creates a bootable image that can be flashed to the board. [51]

3.6.4 Program FPGA

When the FPGA fabric and peripherals are utilized on the Zynq evaluation board, a Bitstream BIT file is generated in PlanAhead using the bitstream generator. The bitstream is used to configure the custom design logic in the PL by downloading the system.bit file to the FPGA within the SDK. When only the PS is required, the Bitstream is not needed and can be omitted. [44] The FPGA must be programmed anytime EMIO is used. An example would be when using the Pmod2 connector on the Zynq Evaluation Board.

3.6.5 XMD Console

The XMD console is useful for running and debugging an embedded design application. It can be used for debugging and verifying the system for the Dual ARM Cortex-A9 MPCore processor running on the hardware board and is accessed from the XPS or SDK. The hardware board is debugged using a cycle-accurate Instruction Set Simulator (ISS). XMD provides a Tool Command Language (TCL) interface that is used for command line control and debugging of the target board. Additionally, it can be used to test a complete system by running verification test scripts.

Debugging control of the target board in XMD can be done from the supported GNU Debugger (GDB) remote TCP or JTAG. XMD is used to download the FSBL to the evaluation board and the application's ELF file. The "connect arm hw" command allows the SDK to connect to the ARM processor on the hardware board. The ELF file can be downloaded to the processor using the "dow" command. It can be ran and stop using the commands "con" and "stop" respectively. When downloading a different ELF file, use "rst –processor" to reset the processor. [44]

3.7 ChipScope Pro

ChipScope Pro is useful for on-chip debugging of FPGA designs and assists with in-circuit verification. ChipScope Pro’s tools and IP cores provide embedded designers with a practical ways to test FPGA devices. These tools integrate measurement hardware components with Xilinx target boards for testing. The components communicate with the tools and provide the embedded designer with logic analyzing capabilities. The ChipScope Pro Serial I/O Toolkit, for example, explores and debugs high-speed serial transceiver I/O designs on FPGAs. The Internal Bit Error Ratio Tester core and associated software provides and perform bit error ratio analysis on high-speed serial transceivers channels implemented on the FPGA. [50]

3.8 Tera Term

A serial communication utility is needed to transmit and receive information of the ZC702 Evaluation Board. The SDK has a built in serial terminal utility available to the embedded design. This utility functions well, but there are also various other terminal utilities that designers tend to prefer. Tera Term is a free open-source terminal emulator and was used for the embedded designs for this thesis. [41] The terminal is connected from the Host PC to the UART port of ZC702 Evaluation Board using a USB Type-A to USB Mini-B cable. The standard configuration used for Zynq PS was a Baud rate of 115200, 8 bits, no parity, a stop equal to1 bit and no flow control. [39]

CHAPTER 4

ZYNQ EPP OPERATING SYSTEMS

4.1 Introduction

Selecting the optimal OS for embedded applications is key in designing the system. It important to understand the system's design requirements when choosing the OS as it will affect how applications can be developed and ran. There are a variety of OS able run on the Zc702 Evaluation Board. There are three platforms this thesis is concerned which include Bare-Metal, Xilinx's Linux kernel, and FreeRTOS. The Standalone "Bare-metal" software system provides low level control that is included with the Xilinx ISE Design Suite.

Though Bare-Metal provides low level control, it is not technically an OS, but for all intended purposes it still can run on one or both of the ARM cores and process much like any other OS. A bootable image of Xilinx's Linux kernel comes prepackaged with the evaluation kit and is discussed briefly. Finally, FreeRTOS is a well known free RTOS that provides constantly updated ports that run on the Zynq EPP. Since the Zynq-7000 SoC has a dual ARM processor, a decision must be made when utilizing both cores on whether to use SMP or AMP and which OS(s) will be used for each of the cores. AMP with Bare-Metal on one core and Linux [46] or FreeRTOS on the other or FreeRTOS on both cores are a few examples. [51] This thesis focuses specifically on multitasking FreeRTOS on a single core, but will discuss the other available OS for context.

4.2 Bare-Metal

Bare-Metal is a simple, low-level software layer included in the Xilinx SDK. It provides processor features including caches, interrupts, and exceptions in a single threaded manner. The OS provides basic I/O, profiling, abort, and exit features. A basic C program application can be run on top of the Bare-Metal OS. [39] Bare-metal is used on a software system that typically does not require many features that are normally provided by an actual OS. There are trade-offs between having a simple software system over an OS. An OS requires some processor throughput and tends to be less deterministic than that of a simple software system, but the simple system might not be able to handle the overhead or lack determinism. In today's embedded processing design, processing speeds allow an OS to run with negligible overhead though some system designers avoid an OS due to their complexity. [51]

4.3 Linux

As an addition to the Bare-metal OS, Xilinx provides software design tools for the development of Linux applications. The Zynq EPP evaluation board comes with a pre-installed Linux kernel that is monitored by Xilinx and is specifically designed to run on the Zynq EPP. Additionally, there are various vendors that provide Linux distributions. Linux is a popular OS among the Zynq community. Many embedded designers use Linux because it is regarded as a protected full-featured OS that takes advantage of the MMU in the processor and provides SMP capabilities to utilize multiple processors. Xilinx provides drivers for the peripherals in the PS and additional drivers can be added for custom logic in the PL.

Linux can boot in multiple ways including from a boot image into flash during power up or resetting the board, downloading and running the FSBL which is followed by U-Boot and then the Linux Kernel, or using U-Boot to load and run images. U-Boot is an open source bootloader used by Xilinx and the Linux community. Linux isn't a RTOS, but does have some real-time characteristics. [51] Designers that require a RTOS will find FreeRTOS to be an applicable solution.

4.3 FreeRTOS

The FreeRTOS port for the Zynq EPP is available from the FreeRTOS website. It is based on version 7.0.2 of FreeRTOS and should be noted that it is not supported by Xilinx Technical Support. It was tested to run with the default Zynq ZC702 system, a CPU frequency of 667 MHz, and in JTAG boot mode. It utilizes SCUTIMER, which runs at half the CPU frequency, for generating tick interrupts. The UART is used for displaying messages on a console terminal such as Tera Term. The FreeRTOS port extends the Bare-Metal's Standalone BSP to recognize and include FreeRTOS source files. Some demo applications are included with the port including applications for printing Hello World to the terminal as well as blinking LEDs using semaphores and mutexs. This port utilizes all the standard FreeRTOS functions available and was used as the basis for all of the work in this thesis. [56]

CHAPTER 5

UTILIZING THE ZYNQ EPP HARDWARE

5.1 Introduction

It is important to understand how to develop an embedded system with the Zynq EPP and the various options it makes available to the designer. The evaluation board has several boot options and can boot from a bootable image on an SD card, boot in Quad SPI mode, or with JTAG using a Xilinx Platform Cable. Additional IP in not required to utilize the Zynq PS, but if peripherals that used the PL are, it can be attached by adding IPs in the fabric. This PS + PL combination allows an embedded designer to achieve complex, but efficient designs of a single SoC. Additional hardware components can be attached to the hardware board including a Pmod connection. [51]

5.2 Booting

The Zynq EPP can be configured to boot in secure mode using static memories only, which is JTAG disabled, or in non-secure mode using static memories or JTAG. JTAG mode is primarily used for development and debugging. Other booting options include NAND, parallel NOR, Serial NOR, also known as Quad-SPI, or SD flash memory. There are three boot stages the Zynq can go through. Stage-0 boot know as BootROM, followed by the FSBL, and then optionally a Second Stage Bootloader. [51] The JTAG boot mode was used for the entirety of this thesis. In order to use JTAG for programming and debugging, the board either needs a Xilinx Platform Cable or a Digilent Cable. This thesis used the Xilinx Platform Cable II. If the designer decides to boot from SD, an 8 GB SD card is included to store bootable images for the evaluation to boot in SD mode.

5.3 Pmod Connection

All of the projects of this thesis targeted the Zynq ZC702 Rev C evaluation board. The evaluation board requires several additional hardware components to function. The board gets its power from an AC power adapter that provides 12 VDC. The board communicates with the host pc using a USB Type-A to USB Mini-B cable. Pmod connectors were used to attach an external Pmod module. When performing benchmarking, all signals were sent through the Pmod2 port of the Zynq EPP evaluation board and measured. In order to do this, a Digilent 6-pin Test Point header Pmod module was used that provides connections for probing. [49]

The signals sent to the Pmod2 port were measured using a DigiView Tech Tool Logic Analyzer model DV1-100. It is a 100 MHz, 18 Channel, analyzer that connects to the host terminal through a USB 1.0 to USB 2.0 cable. [55] Signals are monitored using the provided software tool from DigiView. [48] All benchmarking data was recorded using this tool.

CHAPTER 6

RESULTS

6.1 Introduction

Implementing FreeRTOS on the Zynq EPP required an in-depth understanding of the design tools and hardware. The work in this thesis discusses the timeline of development on the Zynq evaluation board. Understanding the basic development tools and hardware began by following the basic tutorials provided with the evaluation kit. [39] It reviews the software design tool basics and implements the infamous “Hello World” program that prints the message to a terminal. There is a strong Zynq EPP community being develop, more specifically for the Zedboard, which is an available resource for beginning designers. Designer blogs, including the Zynq Geek blog, have been supported by Zedboard.org and have their own spot on the community website. [45]

6.2 Bare-Metal - Single Core

Bare-Metal is included with the Xilinx ISE Design Suite and is supported by several tutorials from Xilinx. It provides a basis for understanding basic C program development on the Zynq EPP. It works with the default hardware setup and is strongly supported by Xilinx. Implementing “Hello World” is the start for embedded designers. From here, designers can begin to talk to various built in peripherals including switches and LEDs. The first step is to design with only the PS and using AXI GPIO MIO. This does not require the designer to program the FPGA fabric. This allows the designer to control LEDs and communicate with the UART, and various other AXI interconnects.

Once an understanding is of the PS has been developed, the EMIO can be used. This AXI interconnect utilize the PL and requires the FPGA to be programmed at the most basic level. The PL at the most basic level acts like a wire and passes signals. This allows for the use of GPIO such as the PMO2 connector on the evaluation board. The PL can also eventually be used in more advanced ways than simply passing signals, but will not be addressed in this thesis. A basic example of this is controlling the LEDs using a PWM signal which is sent to the PL as a duty cycle from the PS. [53]

6.3 FreeRTOS - Multitasking Single Core

A strong basic understanding of the Zynq EPP is needed to implement more advanced designs. The concepts utilized with Bare-Metal carry over for work with FreeRTOS. The port provided for FreeRTOS to run on the Zynq EPP provides basic instructions to implement the OS on a single core of the hardware. [56] It contains basic example applications including printing “Hello World” with tasks and blinking LEDs with semaphores and mutexes. With these basic examples and FreeRTOS manuals, [57] more advanced applications can be developed. Similarly, as with Bare-Metal, C programs can be developed to utilize the AXI GPIO MIO interconnects. Again, this just requires just the PS and no PL needs to be programmed.

The PL and the EMIO can be utilized by programming the FPGA fabric. Again, it can used as a basic wire or can eventually be programmed for more advanced system development. There is currently a known problem with FreeRTOS where if the PL is programmed, there are problems with libraries in the SDK and the designer must manually modify them. [56] The Pmod2 was used to perform the benchmarking and required programming the PL due to it being an EMIO interconnect on the evaluation board.

6.4 Benchmarking

The benchmarking for FreeRTOS followed the source code from the Rhealstone Benchmarking done for the iRMX RTOS with minor modifications to work with the new OS. [37] Each benchmark starts by outputting a HIGH single to the Pmod2 port and is measured with the DigiView Logic Analyzer. When the benchmark is finished, it sets the Pmod2 signal low. The time the signal remained HIGH was used as the total execution time of each benchmark. The DigiView Logic Analyzer has a resolution of 10 nsec so the benchmarks utilize sample interpolation to produce a finer measurement. [55] As stated previously, the CPU operates at 633 MHz for FreeRTOS, which results in a period of 1.6 nsec. Each of the benchmarks perform are discussed in detail and the source code is provided in the Appendices.

6.4.1 FreeRTOS Task-Switching Time

The Task-Switching benchmarking sets up two tasks with equal priority. The tasks switch back and forth between the processor and repeats for 50000 iterations. To first determine the time it takes to perform the for loop "work", the benchmark just measures the loops performing no work and not task switching. This is shown in Code Listing 1.

```
for (count1 = 0; count1 < MAX_LOOPS_SERIAL; count1++)
  {
        // Do Nothing
  }
 for (count2 = 0; count2 < MAX_LOOPS_SERIAL; count2++)
  {
        // Do Nothing
  }
```

Code Listing 1: Benchmark without Task-Switching Time

The MAX_LOOPS_SERIAL is the total number of iterations for the benchmark. This code does not actually create tasks and simply determines the execution time of the portions of the code that are not part of the Task-Switching measurement. The execution of this code segment is recorded with the Digiview software and the second portion of the code runs. The second portion utilizes two tasks. This time the two tasks perform task switching for the desired number of iterations. This is shown in Code Listing 2.

The prvFirst and prvSecond are the two tasks respectively and each perform their own "work" loops for the number of iteration specified by MAX_LOOPS_TASK_SWITCHING which for this benchmark was 500,000. The Task-Switching is performed by the taskYIELD(); command.

When the tasks are finished, the vTaskDelete(); is used to delete the task and the xHandle variable points to the desired task. In this case each task deletes itself.

```
static void prvFirst( void *pvParameters ) //Task 1
{
	for( ;; )
	{
		for (count1 = 0; count1 < MAX_LOOPS_TASK_SWITCHING; count1++)
		{
			taskYIELD();
		}
		vTaskDelete(xHandleFirst);  // Delete Task 1
	}
}

static void prvSecond( void *pvParameters ) //Task 2
{
	for( ;; )
	{
		for (count2 = 0; count2 < MAX_LOOPS_TASK_SWITCHING; count2++)
		{
			taskYIELD();
		}
		vTaskDelete(xHandleSecond);  // Delete Task 2
	}
}
```

Code Listing 2: Task-Switching Time

The Task-Switching value is important for the other benchmarks. Since most of the other benchmarks require Task-Switching as part of the other benchmarks, the value calculated in this section can be used to negate the extra time measured for the other benchmarks inflated by Task-Switching.

6.4.2 FreeRTOS Preemption Time

The Preemption benchmarks works by creating two tasks. Task 2 has a higher priority and delays for one tick interrupt. While it’s sleeping, Task 1 runs. Task 1 gets preempted when Task 2 wakes and Task 2 runs again, but immediate delays. This repeats for 15000 iterations. The benchmark first accounts for the processing time required by the for loops that do “work” as shown in Code Listing 3.

```
for (count1 = 0; count1 < MAX_LOOPS; count1++)
  {
          for (i = 0; i < ONE_TICK_AVERAGE; i++)
                {
                     // Do Nothing
                }
  }
 for (count2 = 0; count2 < MAX_LOOPS; count2++)
  {
          // Do Nothing
  }
```

Code Listing 3: Benchmark Time without Preemption

MAX_LOOPS is equal to the benchmark iteration number. ONE_TICK_AVERAGE is the average amount of for loops that can be performed during one tick. Once the time for the for loops are determined, the two tasks are created and the benchmark measures the preemption time. Task 2 runs and immediately sleeps and Task 1 does "work" until it gets preempted by Task 2. This is demonstrated in Code Listing 4.

```
static void prvFirst( void *pvParameters ) //Task 1
{
        for( ;; )
        {
                for (count1 = 0; count1 < MAX_LOOPS; count1++)
                {
                        for (i = 0; i < ONE_TICK; i++)
                        {
                            // Do Nothing
                        }
                }
                vTaskDelete(xHandleFirst);  // Delete Task 1
        }
}

static void prvSecond( void *pvParameters ) //Task 2
{
        for( ;; )
        {
                for (count2 = 0; count2 < MAX_LOOPS; count2++)
                {
                        i = ONE_TICK; // Reset i because i never reaches ONE_TICK
                        vTaskDelay(1); // Delay a single tick
                }
                vTaskDelete(xHandleSecond);  // Delete Task 2
        }
}
```

Code Listing 4: Preemption Time

ONE_TICK is a slightly higher number than the number of for loops that can be performed during one tick. The vTaskDelay(); accepts the number of ticks the specific task should delay. While Task 2 delays, Task 1 can run but only until Task to wakes up from its delay. The two measurement times are subtracted from each other to determine the Preemption and Task switching time. The time determined for Task-Switching, which was determined from the first benchmark, is subtracted from the Preemption benchmark time to calculate the Preemption time itself.

6.4.3 FreeRTOS Semaphore Shuffle Time

This Semaphore Shuffle Time benchmark creates 2 tasks and a binary semaphore. Each task only has 2 capabilities. They can either take or give the semaphore and yield after either action. Task 1 will start by taking the semaphore and then yield. Task 2 runs and also attempts to take the semaphore. It blocks because it cannot, and waits for the semaphore to be available. Task 1 runs and releases the semaphore and yields again. Task 2 now sees that Task 1 has released it and takes the semaphore and then yields. Task 1 now attempts to take the semaphore, cant because Task 2 has it, and therefore it blocks and waits for it to be available. Task 2 runs, releases the semaphore, and yields. The process repeats for the specified number of iterations. Code Listing 5 shows Task 1 and Code Listing 6 shows Task 2.

The benchmark needs to first be ran without the semaphore and then ran with it. The two execution times are subtracted from each other to determine the Semaphore Shuffle Time. Task 1 and Task 2, when the sem_exe is set to zero, the semaphore is not used and the benchmark is ran to determine the execution time of each loop and task-switches. When sem_exe is set to one, the benchmark utilizes the semaphore.

```
static void prvFirst( void *pvParameters ) //Task 1
{
        for( ;; )
        {
                for (count1 = 0; count1 < MAX_LOOPS; count1++)
                {
                        if (sem_exe == 1)
                        {
                                xSemaphoreTake(xSemaphore, portMAX_DELAY);
                        }
                        taskYIELD();

                        if (sem_exe == 1)
                        {
                                xSemaphoreGive(xSemaphore);
                        }
                        taskYIELD();
                }
                vTaskDelete(xHandleFirst); //Delete Task 1
        }
}
```

Code Listing 5: Semaphore Shuffle Task 1

The xSemaphoreTake(); command allows the task to take the semaphore if available and the xSemaphoreGive(); allows the task to give the semaphore. The xSemaphore is the handle for the binary semaphore that was created while portMAX_DELAY forces the task to wait indefinitely until the semaphore is available. After the tasks have run for the desired number of iterations, they delete themselves.

```
static void prvSecond( void *pvParameters ) //Task 2
{
        for( ;; )
        {
                for (count2 = 0; count2 < MAX_LOOPS; count2++)
                {
                        if (sem_exe == 1)
                        {
                                xSemaphoreTake(xSemaphore, portMAX_DELAY);
                        }
                        taskYIELD();

                        if (sem_exe == 1)
                        {
                                xSemaphoreGive(xSemaphore);
                        }
                        taskYIELD();
                }
                vTaskDelete(xHandleSecond); //Delete Task 2
        }
}
```

Code Listing 6: Semaphore Shuffle Task 2

6.4.4 FreeRTOS Deadlock Breaking Time

The Deadlock Breaking Time benchmark creates 3 tasks each with a higher priority than the next. Task 3 has the highest priority, Task 2 has a medium priority and Task 1 has the lowest priority. Task 1 takes the mutex and gets preempted by Task 2. Task 2 runs for a little and gets preempted by Task 3. Task 3 requests the mutex and a deadlock occurs because Task 1 has it. Task 3 blocks due to the dead-lock allowing Task 2 to run. Task 2 finishes and delays letting Task 1 run allowing the it to release the mutex. It then gets preempted immediately by Task 3 which takes the mutex and then releases it immediately. This benchmark repeats for the desired iterations.

This benchmark measures the dead-lock resolution time. By this, we mean that the time is inflated by the time of 2 preemptions, and several task-switches that is caused by the dead-lock. Tasks 1, 2, and 3 are shown in the Code Listings 7, 8, and 9 respectively.

```
static void prvFirst( void *pvParameters ) //Task 1
{
        for( ;; )
        {
                if (count1 == MAX_LOOPS)
                {
                        vTaskDelete(xHandleFirst); //Delete Task 1
                }
                xSemaphoreTake(xMutex, portMAX_DELAY); //Take control

                for (i = 0; i < ONE_TICK; i++)  //delay loop
                {
                        //Do Nothing
                }
                xSemaphoreGive(xMutex); //Release control
                count1++;
        }
}
```

Code Listing 7: Dead-Lock Breaking Task 1

The xMutex is the handle for the mutex.

```
static void prvSecond( void *pvParameters ) //Task 2
{
        for( ;; )
        {
                 for( ;; )
                 {
                  if (count2 == MAX_LOOPS)
                  {
                          vTaskDelete(xHandleSecond); //Delete Task 2
                  }

                 for (j = 0; j < ONE_TICK/4; j++)  //Delay loop
                       {
                               //Do Nothing
                       }
                       vTaskDelay(1); //Delay a single tick
                  count2++;
                 }
        }
}
```

Code Listing 8: Dead-Lock Breaking Task 2

The ONE_TICK variable is again the number of for loops that can be performed for one tick. ONE_TICK/4 is used because we only want the medium priority to run for a small amount of time just to take up some of the tick before Task 1 runs. This will make sure that Task 1 will be preempted. It will also make sure that there is an intermediate task between Task 1 and 3 in order to cause the dead-lock.

```
static void prvThird( void *pvParameters )
{
        for( ;; )
        {
                if (count3 == MAX_LOOPS)
                {
                          vTaskDelete(xHandleThird);  //Delete Task 3
                }
                vTaskDelay(1); //Delay a single tick
                i = ONE_TICK; //Reset Task 1

                if (dead_brk == 1)
                {
                        xSemaphoreTake(xMutex, portMAX_DELAY);  //Take control
                        xSemaphoreGive(xMutex);  //Release control
                }
                count3++;
        }
}
```

Code Listing 9: Dead-Lock Breaking Task 3

The benchmark is ran twice by running the code with and without the dead-lock occurring. The dead_brk variable when set to zero prevents the dead-lock from occurring. The benchmark is measured with this setup and then dead_brk is set to 1. This causes the dead-lock to occur and is measured again. The two measurements subtracted from each other produce the dead-lock resolution time.

6.4.5 FreeRTOS Intertask Messaging Latency

The Intetask Messaging Latency benchmark works is by creating two tasks. Task 2 receives messages while Task 1 sends them. Task 2 has a higher priority and attempts to receive and when it does not receive a message it blocks. This allows Task 1 to run, send a message, and then get preempted by Task 2 who receives the message. Task 2 will attempt to receive another message and then again blocks. It repeats for the specified number of iterations. The measured time is the time it takes to send a message, task switch, receive the message, block due to an empty Queue and then task switch back to Task 1. Again the benchmark must first determine the time it takes to perform the for loops and extra code as shown in Code Listing 10.

```
for (count1 = 0; count1 < MAX_LOOPS; count1++)
  {
            // Do Nothing
  }
for (count2 = 0; count2 < MAX_LOOPS; count2++)
  {
            // Do Nothing
  }
```

Code Listing 10: Benchmark without Intertask Messaging

The benchmark then runs with the two task sending and receiving messages. This is shown in Code Listing 11.

```
static void prvFirst( void *pvParameters )
{
          for( ;; )
          {
                    for (count1 = 0; count1 < MAX_LOOPS; count1++)
                    {
                              if (xQueueSendToBack(xQueue, msg_buf, portMAX_DELAY)!=pdPASS)
                              {
                                        // Nothing could be sent because blocking timer expired
                              }
                    }
                    vTaskDelete(xHandleFirst);  // Delete Task 1
          }
}

static void prvSecond( void *pvParameters )
{
          for( ;; )
          {
                    for (count2 = 0; count2 < MAX_LOOPS; count2++)
                    {
                              if (xQueueReceive(xQueue, recv_buf, portMAX_DELAY)!= pdPASS)
                              {
                                        // Nothing Received because blocking timer expired
                              }
                    }
                              vTaskDelete(xHandleSecond);  // Delete Task 2
          }
}
```

Code Listing 11: Intertask Message Latency

The xQueueReceive(); and xQueueSend(); commands are used to send and receive to the queue that was created. The xQueue is the handle for the queue, msg_buf variable holds the message to be sent, and the recv_buf is the variable that holds the message. The two times are subtracted from each other to provide the Intertask Message Latency and task switching time. The previously calculated task-switching time is then subtracted to get the Intertask Message Latency by itself.

6.4.6 FreeRTOS Rhealstone Benchmark

The FreeRTOS Rhealstone Benchmarks with calculated and the statistics of each are presented in Table 1. The Interrupt Latency benchmark was not included due to difficulties of implementation. The table presents the average time for each parameter, a maximum and minimum value from running each benchmark five times of each set of iterations specified in the Appendices, and the variance of each.

Table 1: FreeRTOS Rhealstone Benchmarks

Rhealstone Benchmarks	**Average Time**	**Minimum**	**Maximum**	**Variance**
Task-Switching Time	230.26 nsec	230.26 nsec	230.26 nsec	.00027 nsec
Preemption Time	11.348 µsec	11.346 µsec	11.352 µsec	5.4858 nsec
Semaphore Shuffle Time	321.85 nsec	321.75 nsec	322.63 nsec	.87514 nsec
Deadlock Breaking Time	24.041 µsec	19.499 µsec	29.315 µsec	9.8150 usec
Intertask Message Latency	1.5564 µsec	1.5559 µsec	1.5571 µsec	1.2327 nsec

The Rhealstone Benchmark can be calculated from these values using equation 2.1 and 2.2 with the exception of the Interrupt Latency measurement.

CHAPTER 7

CONCLUSION

The completed work in this thesis includes 5 of the Rhealstone benchmarks for the Zynq EPP Evaluation Board running FreeRTOS. These benchmarks provide a basis for embedded designers to understand and compare FreeRTOS's performance on the Zynq EPP ARM core. This thesis provides a starting point for more advance application development with FreeRTOS by providing thoroughly commented and detailed code. It provides information on starting a new project with the Zynq EPP and compiled a plethora of resources that may help further the development. The thesis began by developing a history to understand why the Zynq EPP was designed and utilizes the hardware that it does. This provides context on why it is such an important piece of hardware in today's engineering world. The basics of the design tools were discussed in a manner that helps designers understand their overall importance and roles quickly and effectively. With an understanding of the tools, the Zynq EPP could be used to develop application upon with ease. The performance of these applications are important to benchmark in order understand the hardware's capabilities including its strengths and weaknesses. The thesis provides a stepping stone for future Zynq EPP development.

CHAPTER 8

FUTURE WORK

As more embedded designers work with the Zynq EPP, the community will grow. A greater understanding of how the hardware can be utilized will become more readily available. With a stronger understanding of the hardware platform, more resources and support will be available for designers to reference. The future goals of this research are to implement AMP starting with Bare-Metal running on both cores. Next, would be to have FreeRTOS running on both cores or FreeRTOS on one core and Bare-Metal on another. Benchmarking with this type of PS would continue in order to provide embedded designers with an even stronger understanding of the system’s capabilities. Additional goals would be to extend more work to the PL. Once a solid foundation is laid for the Zynq EPP, it will become the training tool for instructing future embedded engineers.

REFERENCES

[1] (Oct. 2012). Zynq-7000 SoC Operating Systems, [Online] Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/operating-systems/index.htm

[2] R. Wilson. "Multicore RTOS can be AMP or SMP for ARM processors". Available: http://www.electronicsweekly.com/Articles

[3] W. Wulf. "HYDRA: The kernel of a multiprocessor operating system," CACM, vol. 17, pp. 337-345, June 1974.
Available: http://www.cs.princeton.edu/~rywang/02f518/papers/hydra.pdf.

[4] E. Kwatny. "Chapter 1: Introduction to Operating Systems". PowerPoint. Fall 2012.

[5] (Oct. 2012). Linux and symmetric multiprocessing, [Online] Available: http://www.ibm.com/developerworks/library/l-linux-smp/

[6] (Oct. 2012). SMP, [Online] Available: http://en.wikipedia.org/wiki/File:Shared_memory.svg

[7] Y. Wiseman. "ASOSI: Asymmetric Operating System Infrastructure". Available: http://u.cs.biu.ac.il/~wiseman/pdccs2008.pdf

[8] (Oct. 2012). AMP, [Online] Available: http://en.wikipedia.org/wiki/File:Asmp_2.gif

[9] (Oct. 2012). Heterogeneous- and Homogeneous-Processor System-Design Approaches; [Online] Available: http://www.globalspec.com/reference/60873/203279/1-10-heterogeneous-and-homogeneous- processor-system-design-approaches

[10] Intel. "Hyper-Threading Technology Technical User’s Guide". Intel Corporation. Jan, 2003.

[11] N. Dankert. "The Xilinx Zynq-7000 Extensible Processing Platform". Technische Univwesität Braunschweig. Mar, 2012.

[12] Intel. "Introduction to the System 310 Microcomputer". Intel Corporation. 1989.

[13] MULTIBUS II Technical Series, "Message Passing in the MULTIBUS II Architecture". Bill Clemow, 1993, Addison-Wesley.

[14] (Oct. 2012). AMBA Open Specifications, [Online] Available: http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[15] (Oct. 2012). IP, [Online] Available: http://www.xilinx.com/products/intellectual-property/index.htm

[16] (Oct. 2012). TDP, [Online] Available: http://www.xilinx.com/products/targeted_design_platforms.htm

[17] C. Atack and A. Someren, "The ARM RISC Chip: A Programmer's Guide". Addison-Wesley. 1993.

[18] (Oct. 2012). Cortex-A9 Processor, [Online]. Available: http://arm.com

[19] (Oct. 2012). Dual-Core ARM Cortex-A9 MPCore Processor, [Online]. Available: http://www.altera.com/devices/processor/arm/cortex-a9/m-arm-cortex-a9.html

[20] (Oct. 2012). CoreSight, [Online]. Available: http://infocenter.arm.com

[21] (Oct. 2012). SIMD Architectures, [Online]. Available: http://arstechnica.com/features/2000/03/simd/

[22] (Oct. 2012). ARM architecture, [Online]. Available: http://infocenter.arm.com

[23] (Oct. 2012). AMBA Specification Documentation, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html

[24] (Oct. 2012). AMBA, [Online]. Available: http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[25] Xilinx. "Zynq-7000 EPP Overview". Advanced Product Specifications. Aug, 2012.

[26] (Oct. 2012). AMBA, [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/index.htm

[27] (Oct. 2012). Exploring the Xilinx Zynq - Software Platform or Very Complex FPGA, [Online]. Available: http://www.embedded.com/electronics-blogs/other/4219403/Exploring-the-Xilinx-Zynq--software-platform--or-very-complex-FPGA-

[28] Xilinx. "Zynq-7000 Product Brief". Xilinx Inc. June 11, 2012.

[29] (Oct. 2012). ZedBoard.org, [Online]. Available: http://zedboard.org/

[30] Xilinx. "Zynq Evaluation and Development Hardware User's Guide". Xilinx Inc. Aug, 2012.

[31] "Operating Systems", Deitel, Deitel and Choffnes, 3rd edition 2004, Pearson Education, pp. 358-359.

[32] (Oct. 2012). What is an RTOS, [Online]. Available: http://www.freertos.org/.

[33] (Oct. 2012). FreeRTOS is Everywhere, [Online]. Available: http://www.freertos.org/.

[34] R. Barry. Buy or roll your own OS? Neither with FreeRTOS. April 5, 2010. Available:

http://www.embedded.com/electronics-blogs.

[35] (Oct. 2012). FreeRTOS Ports, [Online]. Available: http://www.freertos.org/.

[36] R. Kar. and K. Porter. "Rhealstone: A Real-Time Benchmarking Proposal". Dr. Dobb's Journal. 1989. Available: http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles /DDJ/1989/8902/8902a/8902a.htm

[37] R. Kar.. "Implementing the Rhealstone Real-Time Benchmark". 1990. Available: http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1990/9004/90 04d/9004d.htm

[38] (Oct. 2012). Temple University System Chip Design Lab, [Online]. Available: http://www.temple.edu/scdc/

[39] Xilinx. “Zynq-7000 All Programmable Soc: Concepts, Tools, and Techniques”. UG873. Xilinx Inc. July 2, 2012.

[40] (April. 2013). Xilinx ISE Design Suite Overview. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_overview.htm

[41] (April. 2013). Tera Term. [Online]. Available: http://ttssh2.sourceforge.jp/

[42] Xilinx. “AXI Reference Guide”. UG 761. Xilinx Inc. March 7, 2011.

[43] Xilinx. “PLBV46 Interface Simplifications”. SP026. Xilinx Inc. October 11, 2007.

[44] Xilinx. “Embedded System Tools Reference Guide”. UG111. Xilinx Inc. July 25, 2012.

[45] (April. 2013). ZynqGeek. [Online]. Available: Zedboard.org/ZynqGeek

[46] Xilinx. “Simple AMP Running Linux and Bare-Metal”. XAPP1078. Xilinx Inc. February 13, 2013.

[47] (April. 2013). Xilinx SDK Overview [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/SDK_Doc/concepts /sdk_c_bsp_internal.htm .

[48] Tech Tools. DigiView. “DigiView User’s Guide”. Tech Tools. 2012.

[49] (April. 2013). Digilent Inc. Pmod 6 Pin Test Module. [Online]. Available: http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,401,549&Prod=PMOD-TPH

[50] Xilinx. “ChipScope Pro Software and Cores” UG029. Xilinx Inc. July 25, 2012.

[51] Xilinx. “Zynq-7000 All Programmable SoC Software Developers Guide”. UG821. Xilinx Inc. July 2, 2012.

[52] Xilinx. “Simple AMP Linux and Bare-metal”. XAPP 1078. Xilinx Inc. April 3, 2013.

[53] (April. 2013). Avnet Speedway Tutorials. [Online] Available: www.Zedboard.org/trainings-and-videos

[54] (April. 2013). Xilinx Glossary. [Online] Available: www.xilinx.com/company/terms

[55] (April. 2013). Tech Tools Digi View. [Online] Available: http://www.tech-tools.com/index.html

[56] FreeRTOS. “FreeRTOS Port for Xilinx Zynq Devices” FreeRTOS Ltd. February 12, 2013.

[57] FreeRTOS. “The FreeRTOS Reference Manual” FreeRTOS Ltd. 2013.

APPENDIX A

TASK-SWITCHING CODE

```
/*------------------------------------------------------------
Author: Timothy J Boger
Date: 4/29/13

Task Switching Benchmark
OS:FreeRTOS
Platform: ZC702 Evaluation Board
References: - “FreeRTOS Port for Xilinx Zynq Devices” FreeRTOS Ltd. February 12, 2013.
            - R. Kar.. "Implementing the Rhealstone Real-Time Benchmark". 1990.
            - Cory Nakaji. "MIO, EMIO and AXI GPIO LEDS for ZC702". 2013.
/*------------------------------------------------------------*/
// Includes
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include "xil_printf.h"
#include "stdio.h"
#include "xparameters.h"
#include "xgpio.h"
#include "xgpiops.h"

//***************************
//AXI Variables
static XGpioPs emio_pmod2;

#define EMIO_54        54
#define EMIO_55        55
#define EMIO_56        56
#define EMIO_57        57

//***************************
//Benchmark Variables
#define MAX_LOOPS_SERIAL 500000  //Max loops for simulation
#define MAX_LOOPS_TASK_SWITCHING 499999 //Accounting for extra Task3 switching

unsigned long  count1 = 0, count2 = 0;

//*******************************************************
// Priorities at which the tasks are created

#define mainFIRST_TASK_PRIORITY          ( tskIDLE_PRIORITY + 2 )
#define mainSECOND_TASK_PRIORITY         ( tskIDLE_PRIORITY + 2 )
#define mainTHIRD_TASK_PRIORITY          ( tskIDLE_PRIORITY + 3 )
```

```
//******************************************************
//Associate Functions with Tasks
static void prvFirst( void *pvParameters );
static void prvSecond( void *pvParameters );
static void prvThird( void *pvParameters );

//******************************************************
//Task Handle
xTaskHandle xHandleFirst;
xTaskHandle xHandleSecond;
xTaskHandle xHandleThird;

//******************************************************
//Main
int main( void )
{
        prvInitializeExceptions();

        //******************************************************
        //AXI Setup

                XGpioPs_Config *ConfigPtrPS;

                ConfigPtrPS = XGpioPs_LookupConfig(0);
                XGpioPs_CfgInitialize(&emio_pmod2, ConfigPtrPS,
                                                ConfigPtrPS->BaseAddr);

                //******************************************************
                //Setup PMOD 2 pins
            XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_54, 1);
            XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_54, 1);
            XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_55, 1);
            XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_55, 1);
            XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_56, 1);
            XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_56, 1);
            XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_57, 1);
            XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_57, 1);

                //******************************************************
                //Setup PMOD 2 outputs to zero
            XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0);
            XGpioPs_WritePin(&emio_pmod2, EMIO_55, 0x0);
            XGpioPs_WritePin(&emio_pmod2, EMIO_56, 0x0);
            XGpioPs_WritePin(&emio_pmod2, EMIO_57, 0x0);

        //******************************************************
        //Start Benchmark
        xil_printf("Start of Task Switching Benchmark\n\r");
        xil_printf("Each task runs %D times\r\n", MAX_LOOPS_SERIAL);

        /***************************************************************
```

```
 Serial Non_Switching Measurement

 Measure execution time of task1 and task2 when they are executed
 serially (without task switching).

 Measure the time between the High and Low GPIO output
/*****************************************************************/

xil_printf("Start Serial Non_Switching Measurement\r\n");
XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x1); //Set GPIO HIGH
  for (count1 = 0; count1 < MAX_LOOPS_SERIAL; count1++)
    {
        //Do Nothing
    }
  for (count2 = 0; count2 < MAX_LOOPS_SERIAL; count2++)
    {
        // Do Nothing
    }

XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0); //Set GPIO LOW

xil_printf("Serial Non_Switching Measurement Done\r\n");

/*****************************************************************
 Task Switching Measurement

 Create three tasks. Task 1 and Task 2 will perform the task switching.
 Task 3 controls the start and finish of the program and sets the GPIO pin

 Measure the time between the High and Low GPIO output
*****************************************************************/

xil_printf("Start Task Switching Measurement\r\n");

//Create three tasks
xTaskCreate( prvFirst, ( signed char * ) "F",
                    configMINIMAL_STACK_SIZE, NULL,
                    mainFIRST_TASK_PRIORITY, &xHandleFirst  );
xTaskCreate( prvSecond, ( signed char * ) "S",
                    configMINIMAL_STACK_SIZE, NULL,
                    mainSECOND_TASK_PRIORITY, &xHandleSecond  );
xTaskCreate( prvThird, ( signed char * ) "T",
                    configMINIMAL_STACK_SIZE, NULL,
                    mainTHIRD_TASK_PRIORITY, &xHandleThird );

vTaskStartScheduler();

/* If all is well, the scheduler will now be running, and the following line
will never be reached.  If the following line does execute, then there was
insufficient FreeRTOS heap memory available for the idle and/or timer tasks
to be created.  See the memory management section on the FreeRTOS web site
```

```
        for more details. */
        for( ;; );
}
//*********************************************************************
//Task 3
static void prvThird( void *pvParameters )
{
        for( ;; )
        {
                //Runs First due to having highest priority
                                XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x1); //Set GPIO
HIGH

                                vTaskPrioritySet(xHandleThird, tskIDLE_PRIORITY + 1);
//reduce priority below Task 1 and 2

//-------------------------  Task will yield here. Returns when Task 1 and 2 delete themselves

                                //xil_printf("LOW\r\n");
                                XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0); //Set GPIO
LOW

                                xil_printf("Task Switching Measurement Done\r\n");
                                vTaskDelete(xHandleThird);  //Delete Task 3
        }
}

//*********************************************************************
//Task 1
static void prvFirst( void *pvParameters )
{
        for( ;; )
        {
                        for (count1 = 0; count1 < MAX_LOOPS_TASK_SWITCHING;
count1++)
                        {
                                taskYIELD();
                        }
                        vTaskDelete(xHandleFirst); //Delete Task 1

        }
}

//*********************************************************************
//Task 2
static void prvSecond( void *pvParameters )
{
        for( ;; )
        {
                        for (count2 = 0; count2 < MAX_LOOPS_TASK_SWITCHING;
count2++)
```

```
                    {
                            taskYIELD();
                    }
                    vTaskDelete(xHandleSecond); //Delete Task 2
        }
}

//*******************************************************************
void vApplicationMallocFailedHook( void )
{
        /* vApplicationMallocFailedHook() will only be called if
        configUSE_MALLOC_FAILED_HOOK is set to 1 in FreeRTOSConfig.h.  It is a hook
        function that will get called if a call to pvPortMalloc() fails.
        pvPortMalloc() is called internally by the kernel whenever a task, queue or
        semaphore is created.  It is also called by various parts of the demo
        application.  If heap_1.c or heap_2.c are used, then the size of the heap
        available to pvPortMalloc() is defined by configTOTAL_HEAP_SIZE in
        FreeRTOSConfig.h, and the xPortGetFreeHeapSize() API function can be used
        to query the size of free heap space that remains (although it does not
        provide information on how the remaining heap might be fragmented). */
        taskDISABLE_INTERRUPTS();
        for( ;; );
}

//*******************************************************************
void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed char *pcTaskName )
{
        ( void ) pcTaskName;
        ( void ) pxTask;

        /* vApplicationStackOverflowHook() will only be called if
        configCHECK_FOR_STACK_OVERFLOW is set to either 1 or 2.  The handle and name
        of the offending task will be passed into the hook function via its
        parameters.  However, when a stack has overflowed, it is possible that the
        parameters will have been corrupted, in which case the pxCurrentTCB variable
        can be inspected directly. */
        taskDISABLE_INTERRUPTS();
        for( ;; );
}

//*******************************************************************
void vApplicationSetupHardware( void )
{
        /* Do nothing */
}
```

APPENDIX B

PREEMPTION TIME CODE

```
/*--------------------------------------------------------
Author: Timothy J Boger
Date: 4/29/13

Preemption Time Benchmark
OS:FreeRTOS
Platform: ZC702 Evaluation Board
References: -  "FreeRTOS Port for Xilinx Zynq Devices" FreeRTOS Ltd. February 12, 2013.
            -  R. Kar.. "Implementing the Rhealstone Real-Time Benchmark". 1990.
              -  Cory Nakaji. "MIO, EMIO and AXI GPIO LEDS for ZC702". 2013.
/*-------------------------------------------------------*/
// Includes
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include "xil_printf.h"
#include "stdio.h"
#include "xparameters.h"
#include "xgpio.h"
#include "xgpiops.h"

//**************************
//AXI Variables
static XGpioPs emio_pmod2;

#define EMIO_54        54
#define EMIO_55        55
#define EMIO_56        56
#define EMIO_57        57

//**************************
//Benchmark Variables

#define MAX_LOOPS 15000 //Max loops for simulation
#define ONE_TICK 480000 //Number dependent on CPU. Must be longer than sleep period.
                                          //The amount of for loop iterations per one interrupt tick
#define ONE_TICK_AVERAGE 475620
unsigned long  count1, count2, i;

//*****************************************************
// Priorities at which the tasks are created

#define mainFIRST_TASK_PRIORITY          ( tskIDLE_PRIORITY + 2 )
#define mainSECOND_TASK_PRIORITY         ( tskIDLE_PRIORITY + 3 )
```

```
#define mainTHIRD_TASK_PRIORITY          ( tskIDLE_PRIORITY + 4 )

//*******************************************************
//Associate Functions with Tasks

static void prvFirst( void *pvParameters );
static void prvSecond( void *pvParameters );
static void prvThird( void *pvParameters );

//*******************************************************
//Task and Queue Handles

xTaskHandle xHandleFirst;
xTaskHandle xHandleSecond;
xTaskHandle xHandleThird;

//*******************************************************
//Main

int main( void )
{
        prvInitializeExceptions();

        //*******************************************************
        //AXI Setup

                XGpioPs_Config *ConfigPtrPS;

                ConfigPtrPS = XGpioPs_LookupConfig(0);
                XGpioPs_CfgInitialize(&emio_pmod2, ConfigPtrPS,
                                                ConfigPtrPS->BaseAddr);

                //*******************************************************
                //Setup PMOD 2 pins
             XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_54, 1);
             XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_54, 1);
             XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_55, 1);
             XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_55, 1);
             XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_56, 1);
             XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_56, 1);
             XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_57, 1);
             XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_57, 1);

                //*******************************************************
                //Setup PMOD 2 outputs to zero
             XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0);
             XGpioPs_WritePin(&emio_pmod2, EMIO_55, 0x0);
             XGpioPs_WritePin(&emio_pmod2, EMIO_56, 0x0);
             XGpioPs_WritePin(&emio_pmod2, EMIO_57, 0x0);

        //*******************************************************
```

```
//Start Benchmark

xil_printf("Start of Preemption Time Benchmark\n\r");
xil_printf("Each task runs %D times\r\n", MAX_LOOPS);

/************************************************************************
 Serial Execution Measurement Without Task Switching or Preemption

 Measure execution time of task1 and task2 when they are executed
 serially (without messages).

 Measure the time between the High and Low GPIO output
/**********************************************************************/

XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x1); //Set GPIO HIGH
xil_printf("Start Serial Execution Without Task Switching or Preemption\r\n");

  for (count1 = 0; count1 < MAX_LOOPS; count1++)
    {
            for (i = 0; i < ONE_TICK_AVERAGE; i++)
                      {
                        //Do Nothing
                      }
    }
  for (count2 = 0; count2 < MAX_LOOPS; count2++)
    {
            i = ONE_TICK; //reset i because i never reaches ONE_TICK
    }

XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0); //Set GPIO LOW

xil_printf("Serial Execution Without Task Switching or Preemption Done\r\n");

/************************************************************************
 Task Switching and Preemption Time Measurement

 Create three tasks. Task 1 and Task 2 will perform the Task Switching and Preemption.
 Task 1 does busy work and gets preempted by Task 2.
 Task 2 has a higher priority than Task 1. Task 1 only runs when Task 2 yields.
 Task 3 controls the start and finish of the program and sets the GPIO pin

 Measure the time between the High and Low GPIO output

************************************************************************/
xil_printf("Start Task Switching and Preemption Time Measurement\r\n");

//Create three tasks
xTaskCreate( prvFirst, ( signed char * ) "F",
                      configMINIMAL_STACK_SIZE, NULL,
                      mainFIRST_TASK_PRIORITY, &xHandleFirst  );
xTaskCreate( prvSecond, ( signed char * ) "S",
                    configMINIMAL_STACK_SIZE, NULL,
                    mainSECOND_TASK_PRIORITY, &xHandleSecond  );
        xTaskCreate( prvThird, ( signed char * ) "T",
                    configMINIMAL_STACK_SIZE, NULL,
                    mainTHIRD_TASK_PRIORITY, &xHandleThird );

        vTaskStartScheduler();

        /* If all is well, the scheduler will now be running, and the following line
        will never be reached.  If the following line does execute, then there was
        insufficient FreeRTOS heap memory available for the idle and/or timer tasks
        to be created.  See the memory management section on the FreeRTOS web site
        for more details. */
        for( ;; );
}
//*******************************************************************
//Task 3

static void prvThird( void *pvParameters )
{
        for( ;; )
        {
                //Runs First due to having highest priority
                        XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x1); //Set GPIO HIGH

                        vTaskPrioritySet(xHandleThird, tskIDLE_PRIORITY + 1); //reduce
priority below Task 1 and 2

//-------------------------  Task will yield here. Returns when Task 1 and 2 delete themselves

                        XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0); //Set GPIO LOW

                        xil_printf("Task Switching and Preemption Time Measurement
Done\r\n");

                        vTaskDelete(xHandleThird);  //Delete Task 3

        }
}

//*******************************************************************
//Task 1 - Lower Priority, Gets Preempted

static void prvFirst( void *pvParameters )
{
        for( ;; )
        {
                for (count1 = 0; count1 < MAX_LOOPS; count1++)
                {
                        for (i = 0; i < ONE_TICK; i++)
                        {
```

```
                //Do Nothing
            }
        }
        vTaskDelete(xHandleFirst); //Delete Task 1
    }
}

//*********************************************************************
//Task 2 - Higher Priority, Preempts

static void prvSecond( void *pvParameters )
{
    for( ;; )
    {
        for (count2 = 0; count2 < MAX_LOOPS; count2++)
        {
            //xil_printf("i value: = %D \r\n", i); //Used to determine
AVERAGE_ONE_TICK
            i = ONE_TICK; //reset i because i never reaches ONE_TICK
            vTaskDelay(1); //Delay a single tick
        }
        vTaskDelete(xHandleSecond); //Delete Task 2
    }
}

//*********************************************************************
void vApplicationMallocFailedHook( void )
{
    /* vApplicationMallocFailedHook() will only be called if
    configUSE_MALLOC_FAILED_HOOK is set to 1 in FreeRTOSConfig.h.  It is a hook
    function that will get called if a call to pvPortMalloc() fails.
    pvPortMalloc() is called internally by the kernel whenever a task, queue or
    semaphore is created.  It is also called by various parts of the demo
    application.  If heap_1.c or heap_2.c are used, then the size of the heap
    available to pvPortMalloc() is defined by configTOTAL_HEAP_SIZE in
    FreeRTOSConfig.h, and the xPortGetFreeHeapSize() API function can be used
    to query the size of free heap space that remains (although it does not
    provide information on how the remaining heap might be fragmented). */
    taskDISABLE_INTERRUPTS();
    for( ;; );
}

//*********************************************************************
void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed char *pcTaskName )
{
    ( void ) pcTaskName;
    ( void ) pxTask;

    /* vApplicationStackOverflowHook() will only be called if
    configCHECK_FOR_STACK_OVERFLOW is set to either 1 or 2.  The handle and name
    of the offending task will be passed into the hook function via its
```

```
	parameters.  However, when a stack has overflowed, it is possible that the
	parameters will have been corrupted, in which case the pxCurrentTCB variable
	can be inspected directly. */
	taskDISABLE_INTERRUPTS();
	for( ;; );
}

//*******************************************************************
void vApplicationSetupHardware( void )
{
	/* Do nothing */
}
```

APPENDIX C

INTERTASK MESSAGE LATENCY CODE

```
/*------------------------------------------------------------
Author: Timothy J Boger
Date: 4/29/13

Inter-Task Message Latency Benchmark
OS:FreeRTOS
Platform: ZC702 Evaluation Board
References: -  "FreeRTOS Port for Xilinx Zynq Devices" FreeRTOS Ltd. February 12, 2013.
               -  R. Kar.. "Implementing the Rhealstone Real-Time Benchmark". 1990.
                 -  Cory Nakaji. "MIO, EMIO and AXI GPIO LEDS for ZC702". 2013.
/*------------------------------------------------------------*/
// Includes
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include "xil_printf.h"
#include "stdio.h"
#include "xparameters.h"
#include "xgpio.h"
#include "xgpiops.h"

//**************************
//AXI Variables
static XGpioPs emio_pmod2;

#define EMIO_54         54
#define EMIO_55         55
#define EMIO_56         56
#define EMIO_57         57

//**************************
//Benchmark Variables
#define MAX_LOOPS 1000000  //Max loops for simulation

char msg_buf[10] = "MESSAGE", recv_buf[10];

#define Queue_Length 10
#define Queue_Item_Size sizeof(msg_buf)

unsigned long  count1, count2;

//******************************************************
// Priorities at which the tasks are created
```

```
#define mainFIRST_TASK_PRIORITY           ( tskIDLE_PRIORITY + 2 )
#define mainSECOND_TASK_PRIORITY          ( tskIDLE_PRIORITY + 3 )
#define mainTHIRD_TASK_PRIORITY           ( tskIDLE_PRIORITY + 4 )

//*****************************************************
//Associate Functions with Tasks

static void prvFirst( void *pvParameters );
static void prvSecond( void *pvParameters );
static void prvThird( void *pvParameters );

//*****************************************************
//Task and Queue Handles

xTaskHandle xHandleFirst;
xTaskHandle xHandleSecond;
xTaskHandle xHandleThird;

xQueueHandle xQueue;

//*****************************************************
//Main

int main( void )
{
        prvInitializeExceptions();

        //*****************************************************
        //AXI Setup

                XGpioPs_Config *ConfigPtrPS;

                ConfigPtrPS = XGpioPs_LookupConfig(0);
                XGpioPs_CfgInitialize(&emio_pmod2, ConfigPtrPS,
                                        ConfigPtrPS->BaseAddr);

                //*****************************************************
                //Setup PMOD 2 pins
             XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_54, 1);
             XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_54, 1);
             XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_55, 1);
             XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_55, 1);
             XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_56, 1);
             XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_56, 1);
             XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_57, 1);
             XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_57, 1);

                //*****************************************************
                //Setup PMOD 2 outputs to zero
             XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0);
             XGpioPs_WritePin(&emio_pmod2, EMIO_55, 0x0);
             XGpioPs_WritePin(&emio_pmod2, EMIO_56, 0x0);
             XGpioPs_WritePin(&emio_pmod2, EMIO_57, 0x0);

        //*******************************************************
        //Start Benchmark

        xil_printf("Start of InterTask Message Latency Benchmark\n\r");
        xil_printf("Each task runs %D times\r\n", MAX_LOOPS);

        // Create Message Queue

        xQueue = xQueueCreate(Queue_Length, Queue_Item_Size);

        if(xQueue == NULL)
        {
          //The queue could not be created
          xil_printf("Queue Create Error\n\r");
        }

        /**********************************************************************
         Serial Execution Measurement Without Messages

         Measure execution time of task1 and task2 when they are executed
         serially (without messages).

         Measure the time between the High and Low GPIO output
        **********************************************************************/

        XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x1); //Set GPIO HIGH
        xil_printf("Start Serial Execution Measurement Without Messages\r\n");

         for (count1 = 0; count1 < MAX_LOOPS; count1++)
           {
             //Do Nothing
           }
         for (count2 = 0; count2 < MAX_LOOPS; count2++)
           {
             // Do Nothing
           }

        XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0); //Set GPIO LOW

        xil_printf("Serial Execution Measurement Without Messages Done\r\n");

        /**********************************************************************
         Inter-Task Message Latency Measurement

         Create three tasks. Task 1 and Task 2 will perform the Messaging.
         Task 1 sends messages, Task 2 receives them.
         Task 2 has a higher priority than Task 1 to make sure it receives messages immediately
         Task 3 controls the start and finish of the program and sets the GPIO pin
```

```
         Measure the time between the High and Low GPIO output
        ***********************************************************************/

        xil_printf("Start Inter-Task Message Latency Measurement\r\n");

        //Create three tasks
        xTaskCreate( prvFirst, ( signed char * ) "F",
                        configMINIMAL_STACK_SIZE, NULL,
                        mainFIRST_TASK_PRIORITY, &xHandleFirst  );
        xTaskCreate( prvSecond, ( signed char * ) "S",
                        configMINIMAL_STACK_SIZE, NULL,
                        mainSECOND_TASK_PRIORITY, &xHandleSecond  );
        xTaskCreate( prvThird, ( signed char * ) "T",
                        configMINIMAL_STACK_SIZE, NULL,
                        mainTHIRD_TASK_PRIORITY, &xHandleThird );

        vTaskStartScheduler();

        /* If all is well, the scheduler will now be running, and the following line
        will never be reached.  If the following line does execute, then there was
        insufficient FreeRTOS heap memory available for the idle and/or timer tasks
        to be created.  See the memory management section on the FreeRTOS web site
        for more details. */
        for( ;; );
}
//*********************************************************************
//Task 3

static void prvThird( void *pvParameters )
{
        for( ;; )
        {
                //Runs First due to having highest priority
                        XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x1); //Set GPIO HIGH

                        vTaskPrioritySet(xHandleThird, tskIDLE_PRIORITY + 1); //reduce
priority below Task 1 and 2

//-------------------------  Task will yield here. Returns when Task 1 and 2 delete themselves

                        XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0); //Set GPIO LOW

                        xil_printf("Inter-Task Message Latency Measurement Done\r\n");

                        vQueueDelete(xQueue);//Delete Queue

                        vTaskDelete(xHandleThird);  //Delete Task 3

        }
}
```

```
//*********************************************************************
//Task 1 - Sends Messages
static void prvFirst( void *pvParameters )
{
	for( ;; )
	{
		for (count1 = 0; count1 < MAX_LOOPS; count1++)
		{
			if(xQueueSendToBack(xQueue, msg_buf,
portMAX_DELAY)!=pdPASS)
			{
				//Nothing could be sent blocking timer expired
				xil_printf("Sent Blocking Timer Ran Out \r\n");
			}
		}
		vTaskDelete(xHandleFirst); //Delete Task 1
	}
}

//*********************************************************************
//Task 2
static void prvSecond( void *pvParameters )
{
	for( ;; )
	{
		for (count2 = 0; count2 < MAX_LOOPS; count2++)
		{
			if(xQueueReceive(xQueue, recv_buf, portMAX_DELAY)!= pdPASS)
			{
				//Nothing Received because blocking timer expired
				xil_printf("Receive Blocking Timer Ran Out \r\n");
			}
		}
		vTaskDelete(xHandleSecond); //Delete Task 2
	}
}

//*********************************************************************
void vApplicationMallocFailedHook( void )
{
	/* vApplicationMallocFailedHook() will only be called if
	configUSE_MALLOC_FAILED_HOOK is set to 1 in FreeRTOSConfig.h.  It is a hook
	function that will get called if a call to pvPortMalloc() fails.
	pvPortMalloc() is called internally by the kernel whenever a task, queue or
	semaphore is created.  It is also called by various parts of the demo
	application.  If heap_1.c or heap_2.c are used, then the size of the heap
	available to pvPortMalloc() is defined by configTOTAL_HEAP_SIZE in
	FreeRTOSConfig.h, and the xPortGetFreeHeapSize() API function can be used
	to query the size of free heap space that remains (although it does not
	provide information on how the remaining heap might be fragmented). */
	taskDISABLE_INTERRUPTS();
	for( ;; );
}

//*****************************************************************
void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed char *pcTaskName )
{
	( void ) pcTaskName;
	( void ) pxTask;

	/* vApplicationStackOverflowHook() will only be called if
	configCHECK_FOR_STACK_OVERFLOW is set to either 1 or 2.  The handle and name
	of the offending task will be passed into the hook function via its
	parameters.  However, when a stack has overflowed, it is possible that the
	parameters will have been corrupted, in which case the pxCurrentTCB variable
	can be inspected directly. */
	taskDISABLE_INTERRUPTS();
	for( ;; );
}

//*****************************************************************
void vApplicationSetupHardware( void )
{
	/* Do nothing */
}
```

APPENDIX D

DEADLOCK-BREAK TIME CODE

```
/*----------------------------------------------------------
Author: Timothy J Boger
Date: 4/29/13

Deadlock Break-Time Benchmark
OS:FreeRTOS
Platform: ZC702 Evaluation Board
References: -  “FreeRTOS Port for Xilinx Zynq Devices” FreeRTOS Ltd. February 12, 2013.
               -  R. Kar.. "Implementing the Rhealstone Real-Time Benchmark". 1990.
               -  Cory Nakaji. "MIO, EMIO and AXI GPIO LEDS for ZC702". 2013.
/*----------------------------------------------------------*/
// Includes
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include "xil_printf.h"
#include "stdio.h"
#include "xparameters.h"
#include "xgpio.h"
#include "xgpiops.h"
#include "semphr.h"

//**************************
//AXI Variables
static XGpioPs emio_pmod2;

#define EMIO_54         54
#define EMIO_55         55
#define EMIO_56         56
#define EMIO_57         57

//**************************
//Benchmark Variables

#define MAX_LOOPS 10000  //Max loops for simulation  10000
#define ONE_TICK 480000 //Number dependent on CPU. Must be longer than sleep period.
                              //The amount of for loop iterations per one interrupt tick
#define ONE_TICK_AVERAGE 475620

unsigned long  count1 = 0, count2 = 0, count3 = 0;
unsigned long  i, j;
unsigned long  dead_brk;  // 1= Yes  0 = No

//*****************************************************
```

```
// Priorities at which the tasks are created

#define mainFIRST_TASK_PRIORITY          ( tskIDLE_PRIORITY + 2 )
#define mainSECOND_TASK_PRIORITY         ( tskIDLE_PRIORITY + 3 )
#define mainTHIRD_TASK_PRIORITY                    ( tskIDLE_PRIORITY + 4 )
#define mainFOURTH_TASK_PRIORITY         ( tskIDLE_PRIORITY + 5 )

//*******************************************************
//Associate Functions with Tasks
static void prvFirst( void *pvParameters );
static void prvSecond( void *pvParameters );
static void prvThird( void *pvParameters );
static void prvFourth( void *pvParameters );

//*******************************************************
//Task Handle
xTaskHandle xHandleFirst;
xTaskHandle xHandleSecond;
xTaskHandle xHandleThird;
xTaskHandle xHandleFourth;

xSemaphoreHandle xMutex;

//*******************************************************
//Main
int main( void )
{
        prvInitializeExceptions();

        //*****************************************************
        //AXI Setup

                XGpioPs_Config *ConfigPtrPS;

                ConfigPtrPS = XGpioPs_LookupConfig(0);
                XGpioPs_CfgInitialize(&emio_pmod2, ConfigPtrPS,
                                        ConfigPtrPS->BaseAddr);

                //*****************************************************
                //Setup PMOD 2 pins
            XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_54, 1);
            XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_54, 1);
            XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_55, 1);
            XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_55, 1);
            XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_56, 1);
            XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_56, 1);
            XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_57, 1);
            XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_57, 1);

                //*****************************************************
                //Setup PMOD 2 outputs to zero
        XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0);
        XGpioPs_WritePin(&emio_pmod2, EMIO_55, 0x0);
        XGpioPs_WritePin(&emio_pmod2, EMIO_56, 0x0);
        XGpioPs_WritePin(&emio_pmod2, EMIO_57, 0x0);

//******************************************************
//Start Benchmark

xil_printf("Start of Deadlock Break-Time Benchmark\n\r");
xil_printf("Each task runs %D times\r\n", MAX_LOOPS);

/*****************************************************************
 Execution Time Measurement Without Deadlocks

 Create four tasks.
 Task 1 Lowest Priority
 Task 2 Medium Priority. Only uses CPU time and sleeps periodically.
 Task 3 Highest Priority. Potential deadlock when it tries to gain control
            of the "region" resource, because low-priority task holds region mostly.

 Task 4 controls the start and finish of the program and sets the GPIO pin

 Note: when dead_brk = 0;
/*****************************************************************
 Deadlock Resolution Measurement

 Create four tasks.
 Task 1 Lowest Priority
 Task 2 Medium Priority. Only uses CPU time and sleeps periodically.
 Task 3 Highest Priority. Potential deadlock when it tries to gain control
            of the "region" resource, because low-priority task holds region mostly.

 Task 4 controls the start and finish of the program and sets the GPIO pin

 Measure the time between the High and Low GPIO output

 Note: when dead_brk = 1;
/*******************************************************************/
//SET DESIRED BENHCMARK VALUE HERE:
 dead_brk = 1; //Run tasks with/without deadlocking  0 = without, 1 = with
 count1 = count2 = count3 = 0;  //Initialize counts

 //Create Semaphore
 xMutex = xSemaphoreCreateMutex();

if (dead_brk == 0)
{
            xil_printf("Start Execution Time Measurement Without Deadlocks\r\n");
}
else
{
		xil_printf("Start Deadlock Resolution Measurement\r\n");
	}

	//Create four tasks
	xTaskCreate( prvFirst, ( signed char * ) "FI",
				configMINIMAL_STACK_SIZE, NULL,
				mainFIRST_TASK_PRIORITY, &xHandleFirst  );
	xTaskCreate( prvSecond, ( signed char * ) "S",
				configMINIMAL_STACK_SIZE, NULL,
				mainSECOND_TASK_PRIORITY, &xHandleSecond  );
	xTaskCreate( prvThird, ( signed char * ) "T",
				configMINIMAL_STACK_SIZE, NULL,
				mainTHIRD_TASK_PRIORITY, &xHandleThird );
	xTaskCreate( prvFourth, ( signed char * ) "FO",
				configMINIMAL_STACK_SIZE, NULL,
				mainFOURTH_TASK_PRIORITY, &xHandleFourth );

	vTaskStartScheduler();

	/* If all is well, the scheduler will now be running, and the following line
	will never be reached.  If the following line does execute, then there was
	insufficient FreeRTOS heap memory available for the idle and/or timer tasks
	to be created.  See the memory management section on the FreeRTOS web site
	for more details. */

	for( ;; );
}
//*****************************************************************
//Task 4
static void prvFourth( void *pvParameters )
{
	for( ;; )
	{
		//Runs First due to having highest priority
				XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x1); //Set GPIO
HIGH

				vTaskPrioritySet(xHandleFourth, tskIDLE_PRIORITY + 1);
//reduce priority below Task 1 and 2

//-------------------------  Task will yield here. Returns when Task 1, 2, and 3 delete themselves

				XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0); //Set GPIO
LOW

				xil_printf("Measurement Done\r\n");
				vTaskDelete(xHandleFourth);  //Delete Task 4
	}
}
//*****************************************************************
//Task 1
```

```
// Lower Priority task.
static void prvFirst( void *pvParameters )
{
        for( ;; )
        {
                if (count1 == MAX_LOOPS)
                {
                        vTaskDelete(xHandleFirst); //Delete Task 1
                }
                xSemaphoreTake(xMutex, portMAX_DELAY); //Take control

                for (i = 0; i < ONE_TICK; i++)  //delay loop
                {
                        //Do Nothing
                }
                xSemaphoreGive(xMutex); //Release control
                count1++;
        }
}
//*********************************************************************
//Task 2
// Medium priority task. Only uses CPU time and sleep periodically.
static void prvSecond( void *pvParameters )
{
        for( ;; )
        {
                 for( ;; )
                 {
                  if (count2 == MAX_LOOPS)
                  {
                          vTaskDelete(xHandleSecond); //Delete Task 2
                  }

                 for (j = 0; j < ONE_TICK/4; j++)            //delay loop
                        {
                                //Do Nothing
                        }
                        vTaskDelay(1); //Delay a single tick
                  count2++;
                 }
        }
}
//*********************************************************************
//Task 3
// High priority task. Potential deadlock when it tries to gain control
// of the "region" resource, because low-priority task holds region mostly.
static void prvThird( void *pvParameters )
{
        for( ;; )
        {
                if (count3 == MAX_LOOPS)
                {
                        vTaskDelete(xHandleThird);  //Delete Task 3
                }
                vTaskDelay(1); //Delay a single tick
                i = ONE_TICK; //Reset Task 1

                if (dead_brk == 1)
                {
                        xSemaphoreTake(xMutex, portMAX_DELAY); //Take control
                        xSemaphoreGive(xMutex); //Release control
                }
                count3++;
        }
}
//****************************************************************
void vApplicationMallocFailedHook( void )
{
        /* vApplicationMallocFailedHook() will only be called if
        configUSE_MALLOC_FAILED_HOOK is set to 1 in FreeRTOSConfig.h.  It is a hook
        function that will get called if a call to pvPortMalloc() fails.
        pvPortMalloc() is called internally by the kernel whenever a task, queue or
        semaphore is created.  It is also called by various parts of the demo
        application.  If heap_1.c or heap_2.c are used, then the size of the heap
        available to pvPortMalloc() is defined by configTOTAL_HEAP_SIZE in
        FreeRTOSConfig.h, and the xPortGetFreeHeapSize() API function can be used
        to query the size of free heap space that remains (although it does not
        provide information on how the remaining heap might be fragmented). */
        taskDISABLE_INTERRUPTS();
        for( ;; );
}
//****************************************************************
void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed char *pcTaskName )
{
        ( void ) pcTaskName;
        ( void ) pxTask;

        /* vApplicationStackOverflowHook() will only be called if
        configCHECK_FOR_STACK_OVERFLOW is set to either 1 or 2.  The handle and name
        of the offending task will be passed into the hook function via its
        parameters.  However, when a stack has overflowed, it is possible that the
        parameters will have been corrupted, in which case the pxCurrentTCB variable
        can be inspected directly. */
        taskDISABLE_INTERRUPTS();
        for( ;; );
}
//****************************************************************
void vApplicationSetupHardware( void )
{
        /* Do nothing */
}
```

APPENDIX E

SEMAPHORE SHUFFLE TIME CODE

```
/*--------------------------------------------------------
Author: Timothy J Boger
Date: 4/29/13

Semaphore Shuffle Benchmark
OS:FreeRTOS
Platform: ZC702 Evaluation Board
References: -  “FreeRTOS Port for Xilinx Zynq Devices” FreeRTOS Ltd. February 12, 2013.
            -  R. Kar.. "Implementing the Rhealstone Real-Time Benchmark". 1990.
            -  Cory Nakaji. "MIO, EMIO and AXI GPIO LEDS for ZC702". 2013.
/*--------------------------------------------------------*/
// Includes
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include "xil_printf.h"
#include "stdio.h"
#include "xparameters.h"
#include "xgpio.h"
#include "xgpiops.h"
#include "semphr.h"

//**************************
//AXI Variables
static XGpioPs emio_pmod2;

#define EMIO_54       54
#define EMIO_55       55
#define EMIO_56       56
#define EMIO_57       57

//**************************
//Benchmark Variables

#define MAX_LOOPS 100000  //Max loops for simulation 100000

unsigned long  count1 = 0, count2 = 0;
unsigned long  sem_exe;  // 1= Yes  0 = No

//*****************************************************
// Priorities at which the tasks are created

#define mainFIRST_TASK_PRIORITY          ( tskIDLE_PRIORITY + 2 )
#define mainSECOND_TASK_PRIORITY         ( tskIDLE_PRIORITY + 2 )
```

```
#define mainTHIRD_TASK_PRIORITY          ( tskIDLE_PRIORITY + 3 )

//*****************************************************
//Associate Functions with Tasks
static void prvFirst( void *pvParameters );
static void prvSecond( void *pvParameters );
static void prvThird( void *pvParameters );

//*****************************************************
//Task Handle
xTaskHandle xHandleFirst;
xTaskHandle xHandleSecond;
xTaskHandle xHandleThird;

xSemaphoreHandle xSemaphore;

//*****************************************************
//Main
int main( void )
{
        prvInitializeExceptions();

        //*****************************************************
        //AXI Setup

                XGpioPs_Config *ConfigPtrPS;

                ConfigPtrPS = XGpioPs_LookupConfig(0);
                XGpioPs_CfgInitialize(&emio_pmod2, ConfigPtrPS,
                                        ConfigPtrPS->BaseAddr);

                //*****************************************************
                //Setup PMOD 2 pins
             XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_54, 1);
             XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_54, 1);
             XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_55, 1);
             XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_55, 1);
             XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_56, 1);
             XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_56, 1);
             XGpioPs_SetDirectionPin(&emio_pmod2, EMIO_57, 1);
             XGpioPs_SetOutputEnablePin(&emio_pmod2, EMIO_57, 1);

                //*****************************************************
                //Setup PMOD 2 outputs to zero
             XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0);
             XGpioPs_WritePin(&emio_pmod2, EMIO_55, 0x0);
             XGpioPs_WritePin(&emio_pmod2, EMIO_56, 0x0);
             XGpioPs_WritePin(&emio_pmod2, EMIO_57, 0x0);

        //*****************************************************
        //Start Benchmark
```

```
xil_printf("Start Semaphore Shuffle Benchmark\n\r");
xil_printf("Each task runs %D times\r\n", MAX_LOOPS);

/***************************************************************
 Task Execution Time Without Semaphore Shuffling Measurement

 Create three tasks. Task 1 and Task 2 will perform the Task Execution.


 Task 3 controls the start and finish of the program and sets the GPIO pin

 Measure the time between the High and Low GPIO output

 Note: when sem_exe = 0;

/***************************************************************
 Semaphore Shuffling Measurement

 Create three tasks. Task 1 and Task 2 will perform Semaphore Shuffling.
 Time it takes a Task to acquire a semaphore that is owned by another equal priority task.

 Task 3 controls the start and finish of the program and sets the GPIO pin

 Measure the time between the High and Low GPIO output

 Note: when sem_exe = 1;

/******************************************************************/
//SET DESIRED BENHCMARK VALUE HERE:
 sem_exe = 1; //Run tasks with/without semaphore shuffling  0 = without, 1 = with

if (sem_exe == 0)
{
        xil_printf("Start Measurement without Semaphore Shuffling \r\n");
}
else
{
        xil_printf("Start Task Semaphore Shuffling Measurement\r\n");
        //Create Semaphore
        vSemaphoreCreateBinary(xSemaphore);
}

//Create three tasks
xTaskCreate( prvFirst, ( signed char * ) "F",
                configMINIMAL_STACK_SIZE, NULL,
                mainFIRST_TASK_PRIORITY, &xHandleFirst  );
xTaskCreate( prvSecond, ( signed char * ) "S",
                configMINIMAL_STACK_SIZE, NULL,
                mainSECOND_TASK_PRIORITY, &xHandleSecond  );
xTaskCreate( prvThird, ( signed char * ) "T",
					configMINIMAL_STACK_SIZE, NULL,
					mainTHIRD_TASK_PRIORITY, &xHandleThird );

	vTaskStartScheduler();

	/* If all is well, the scheduler will now be running, and the following line
	will never be reached.  If the following line does execute, then there was
	insufficient FreeRTOS heap memory available for the idle and/or timer tasks
	to be created.  See the memory management section on the FreeRTOS web site
	for more details. */

	for( ;; );
}
//*****************************************************************
//Task 3
static void prvThird( void *pvParameters )
{
	for( ;; )
	{
		//Runs First due to having highest priority
				XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x1); //Set GPIO
HIGH

				vTaskPrioritySet(xHandleThird, tskIDLE_PRIORITY + 1);
//reduce priority below Task 1 and 2

//--------------------------  Task will yield here. Returns when Task 1 and 2 delete themselves

				XGpioPs_WritePin(&emio_pmod2, EMIO_54, 0x0); //Set GPIO
LOW

				xil_printf("Measurement Done\r\n");
				vTaskDelete(xHandleThird);  //Delete Task 3
	}
}

//*****************************************************************
//Task 1
static void prvFirst( void *pvParameters )
{
	for( ;; )
	{
		for (count1 = 0; count1 < MAX_LOOPS; count1++)
		{
			if (sem_exe == 1)
			{
				xSemaphoreTake(xSemaphore, portMAX_DELAY);
			}
			taskYIELD();

			if (sem_exe == 1)
```

```
				{
					xSemaphoreGive(xSemaphore);
				}
				taskYIELD();
			}
			vTaskDelete(xHandleFirst); //Delete Task 1
		}
}
//*******************************************************************
//Task 2
static void prvSecond( void *pvParameters )
{
		for( ;; )
		{
			for (count2 = 0; count2 < MAX_LOOPS; count2++)
			{
				if (sem_exe == 1)
				{
					xSemaphoreTake(xSemaphore, portMAX_DELAY);
				}
				taskYIELD();

				if (sem_exe == 1)
				{
					xSemaphoreGive(xSemaphore);
				}
				taskYIELD();
			}
			vTaskDelete(xHandleSecond); //Delete Task 2
		}
}

//*******************************************************************
void vApplicationMallocFailedHook( void )
{
		/* vApplicationMallocFailedHook() will only be called if
		configUSE_MALLOC_FAILED_HOOK is set to 1 in FreeRTOSConfig.h.  It is a hook
		function that will get called if a call to pvPortMalloc() fails.
		pvPortMalloc() is called internally by the kernel whenever a task, queue or
		semaphore is created.  It is also called by various parts of the demo
		application.  If heap_1.c or heap_2.c are used, then the size of the heap
		available to pvPortMalloc() is defined by configTOTAL_HEAP_SIZE in
		FreeRTOSConfig.h, and the xPortGetFreeHeapSize() API function can be used
		to query the size of free heap space that remains (although it does not
		provide information on how the remaining heap might be fragmented). */
		taskDISABLE_INTERRUPTS();
		for( ;; );
}

//*******************************************************************
void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed char *pcTaskName )
```

```
{
	( void ) pcTaskName;
	( void ) pxTask;

	/* vApplicationStackOverflowHook() will only be called if
	configCHECK_FOR_STACK_OVERFLOW is set to either 1 or 2.  The handle and name
	of the offending task will be passed into the hook function via its
	parameters.  However, when a stack has overflowed, it is possible that the
	parameters will have been corrupted, in which case the pxCurrentTCB variable
	can be inspected directly. */
	taskDISABLE_INTERRUPTS();
	for( ;; );
}

//*****************************************************************
void vApplicationSetupHardware( void )
{
	/* Do nothing */
}
```