



**NANYANG  
TECHNOLOGICAL  
UNIVERSITY**

## **Linux on Next-Generation Hybrid FPGAs**

**Ansari Zain Us Sami Ahmed**

**(G1200904F)**

**SCHOOL OF COMPUTER ENGINEERING**

**A DISSERTATION SUBMITTED IN PARTIAL FULFILMENT OF**

**THE REQUIREMENTS FOR THE DEGREE OF  
MASTER OF SCIENCE IN EMBEDDED SYSTEMS**

**2013**

## Contents

Abstract.....	2
Acknowledgements .....	3
List of Acronyms .....	4
List of Figures.....	7
List of Tables .....	8
Why Use FPGA based Computing platforms .....	10
Challenges faced by FPGA based Computing platforms .....	14
The Emergence of Extensible Programming Platform (EPP) .....	16
Why run an OS on FPGA based Computing Platforms .....	20
Task Binding:.....	23
Scheduling: .....	23
Communication:.....	24
Virtual Memory: .....	24
I/O: .....	25
Synchronization: .....	25
Protection: .....	25
Building the Tool chain .....	27
Building the Boot Loader .....	30
Building the First Stage Boot loader .....	32
Building the BOOT.BIN .....	34
Building the Device Tree.....	36
Building the Linux Kernel.....	37
Running Zynq on QEMU .....	38
Writing Device Drivers for Zynq .....	39
Code Walkthrough Register Access Driver:.....	44
DSP block-based intermediate fabric: .....	51
Code Walkthrough IF Driver:.....	63
Conclusion and Future Work.....	69
References .....	70

# Abstract

This report focuses on the promising role of FPGA based computing platforms for general purpose computation. ASICs are becoming infeasible due to high NRE costs and short life cycles of the products, while general purpose processors do not offer the performance required by modern applications, within tight energy budgets. This report discusses the real ground breaking reason for demanding a paradigm shift in computing and what are the sort of challenges that this kind paradigm shift could lead to. It also discusses why it makes sense to run an operating system on a FPGA, considering complex embedded systems which cannot be managed without the abstractions provided by the OS. The report then presents the Xilinx Zynq architecture which is seen as a promising evolution of FPGA based computing because it bridges the wide gap between hardware and software engineers promising a new breed of Hardware/Software Embedded System designers. Section 2 focuses on system development for a Xilinx Zynq system. It describes how to bring up the Xilinx Zynq system, how to execute the Linux Kernel on it, how to develop applications for a Xilinx Zynq and how to talk to the custom peripherals that are created on the FPGA.

# **Acknowledgements**

I would distinctively like to thank my supervisor Suhaib A Fahmy for helping me a lot at every stage of the project, without his help and directions I would have not been able to complete my project successfully.

Moreover I would also like to thank Abhishek Kumar Jain for his continuous support, effective suggestions, constructive criticism and timely help.

# List of Acronyms

<b>ACP</b>	Accelerator Coherency Port
<b>ADC</b>	Analog to Digital Convertor
<b>AHB</b>	AMBA High Performance Bus
<b>AMBA</b>	Advanced Microcontroller Bus Architecture
<b>APB</b>	Advanced Peripheral Bus
<b>API</b>	Application Processing Interface
<b>APU</b>	Application Processing Unit
<b>ASB</b>	Advanced System Bus
<b>ASIC</b>	Application Specific Integrated Circuit
<b>AXI</b>	Advanced eXtensible Interface
<b>BIN</b>	Binary
<b>BSD</b>	Berkeley Software Distribution
<b>BSP</b>	Board Support Package
<b>CB</b>	Connection Boxes
<b>CDMA</b>	Central Direct Memory Access
<b>CPU</b>	Central Processing Unit
<b>DAC</b>	Digital to Analog Convertor
<b>DMA</b>	Direct Memory Access
<b>DTB</b>	Device Tree Binary
<b>DTS</b>	Device Tree Source
<b>EDK</b>	Embedded Development Kit
<b>ELF</b>	Executable Linkable Format
<b>EMIO</b>	Extended Multiplexed Input Output
<b>EPP</b>	Extensible Programming Platform

<b>ES</b>	Embedded Systems
<b>FIFO</b>	First In First Out
<b>FLOPS</b>	Floating Point Operation per Second
<b>FPGA</b>	Field Programmable Gate Array
<b>FSBL</b>	First Stage Boot Loader
<b>FTP</b>	File Transfer Protocol
<b>GPL</b>	GNU Public License
<b>GPP</b>	General Purpose Processor
<b>GUI</b>	Graphical User Interface
<b>HDL</b>	Hardware Description Language
<b>HP</b>	High Performance
<b>HPC</b>	High Performance Computing
<b>HW</b>	Hardware
<b>ICT</b>	Information and Communication Technology
<b>IDE</b>	Integrated Development Environment
<b>IF</b>	Intermediate Fabric
<b>IOP</b>	Input Output Peripherals
<b>ILP</b>	Instruction Level Parallelism
<b>IP</b>	Intellectual Property
<b>ISE</b>	Integrated Software Environment
<b>MIPS</b>	Million Instructions Per Second
<b>MIPS/W</b>	Million Instructions Per Second / Watt
<b>MMC</b>	Multimedia Card
<b>MMU</b>	Memory Management Unit
<b>NFS</b>	Network File System
<b>NRE</b>	Non Recurring Expense

<b>OCM</b>	On-Chip Memory
<b>OS</b>	Operating System
<b>PL</b>	Programmable Logic
<b>PLL</b>	Phase Lock Loop
<b>PS</b>	Processing System
<b>QEMU</b>	Quick Emulator
<b>RAM</b>	Random Access Memory
<b>RC</b>	Reconfigurable Computing
<b>ROM</b>	Read Only Memory
<b>RPC</b>	Remote Procedure Call
<b>RTL</b>	Register Transfer Level
<b>SCP</b>	Secure Copy
<b>SD</b>	Secure Digital
<b>SDK</b>	Software Development Kit
<b>SoC</b>	System on Chip
<b>SSH</b>	Secure Shell
<b>SW</b>	Software
<b>TRD</b>	Targeted Reference Design
<b>TTM</b>	Time to Market
<b>U-Boot</b>	Universal Boot
<b>VDMA</b>	Video Direct Memory Access
<b>VF</b>	Virtual Fabric
<b>VHDL</b>	<i>VHSIC Hardware Description Language</i>
<b>XPS</b>	Xilinx Platform Studio

# List of Figures

- 1.1** Figure showing the speed ups achieved using FPGAs.
- 1.2** Figure to represent choice factors on different IC technologies.
- 1.3** Figure illustrates the functional blocks present in a Zynq System.
- 1.4** Figure illustrates PL interface to PS memory subsystem using HP Ports.
- 1.5** Figure illustrates the Unix System Structure.
- 1.6** Figure illustrates a typical Unix transition from user mode to Kernel mode for executing a system call.
  
- 2.1** Figure illustrating a very typical cross development platform.
- 2.2** Figure illustrating the SDK Kit.
- 2.3** Figure illustrating the FSBL Build Parameters.
- 2.4** Figure illustrating the FSBL ELF.
- 2.5** Figure illustrating the SDK boot image creation tab.
- 2.6** Figure illustrating the boot image creation procedures and requirements.
- 2.7** Figure illustrating the Register Access application screen dump (Kernel Module Based).
- 2.8** Figure illustrating the Register Access application screen dump (Memory Mapped)
- 2.9** Figure illustrating the IF architecture.
- 2.10** Figure illustrating the 4 tap FIR filter implementation.
- 2.11** Figure illustrating the IF Registers.
- 2.12** Figure illustrating the State machine based context sequencer.
- 2.13** Figure illustrating the IF Access application screen dump (Kernel Module Based) 32 Samples.
- 2.14** Figure illustrating the IF Access application screen dump (Kernel Module Based) 64 Samples.



- 2.15** Figure illustrating the IF Access application screen dump (Kernel Module Based) 128 Samples.
- 2.16** Figure illustrating the IF Access application screen dump (Kernel Module Based) 256 Samples.
- 2.17** Figure illustrating the IF Access application screen dump (Kernel Module Based) 512 Samples.
- 2.18** Figure illustrating the IF Access application screen dump (Memory Mapped) 32 Samples.
- 2.19** Figure illustrating Xilinx Zynq, Linux Development flowchart.

## **List of Tables**

- 1.1** Speed up and Power saving data from software migration to FPGA based implementations.
- 1.2** Table illustrating the features of Zynq Family of Devices PS end.
- 1.3** Table illustrating the features of Zynq Family of Devices PL end.
- 2.1** Table illustrating the Utilities found in the Binutils Package.
- 2.2** Table illustrating the architectures supported by the Linux Kernel.
- 2.3** Table illustrating Xilinx Zynq features supported by QEMU.

## **Section 1**

# **Introduction**

# Why Use FPGA based Computing platforms

Spatial computing platform provides a significant reduction in energy budget and speed up factors of considerably large magnitude in comparison to the standard Von-Neumann architecture [1]. As discussed in [2] if the current computing trends continue the electricity consumed by the ICT industry alone would grow by a factor of 30. Along with this factor, in [3] it has been described that cost of a warehouse scale compute or a data center is now entirely calculated on the power cost and is not based on the hardware cost or the cost to maintain it. In such circumstances it is completely justified to calculate performance not just on the basis of MIPS but MIPS/W.

The premise of saving energy using FPGAs is not new and has been thoroughly investigated in [4, 5]. Almost 15 years ago it was concluded microprocessors using the standard compilation tool chains have been up to five hundred times power hungrier compared to a standard FPGA implementation [5].

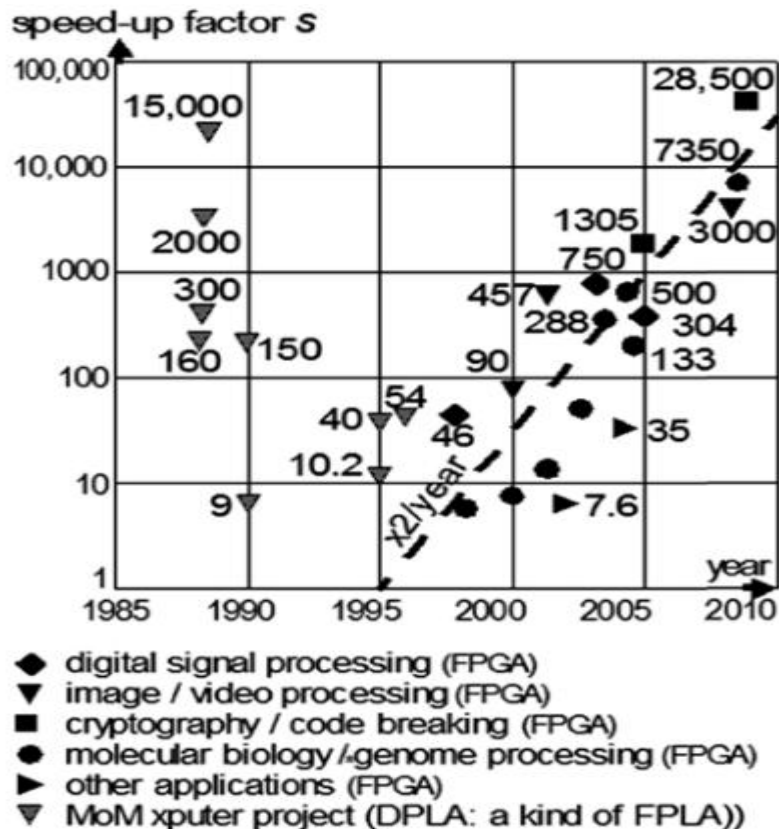


Figure 1.1 [6]

Figure 1.1 illustrates the speed ups achieved using FPGAs over the years compared to the software implementations and these are just a few examples from a very broad range of publications.

SGI Altix 4,700 w. RC 100 RASC vs. Beowulf cluster		Save factor		
	Speed-up factor	Power	Cost	Size
DNA and protein sequencing	8,723	779	22	253
DES braking	28,514	3,439	96	1,116

**Table 1.1 [7]**

Referring to the DES braking, there is a speedup of 28,500 the Power save factor of 3439 has been reported in [7] and cost saving of 96x too. The argument for FPGAs strengthen, and illustrates the fact that no cost intensive hardware is required for scientific computing and FPGAs are not just restricted to embedded systems and are pervasive in several domains. The figures presented in these findings suggest that the era of desktop supercomputing is just around the corner.

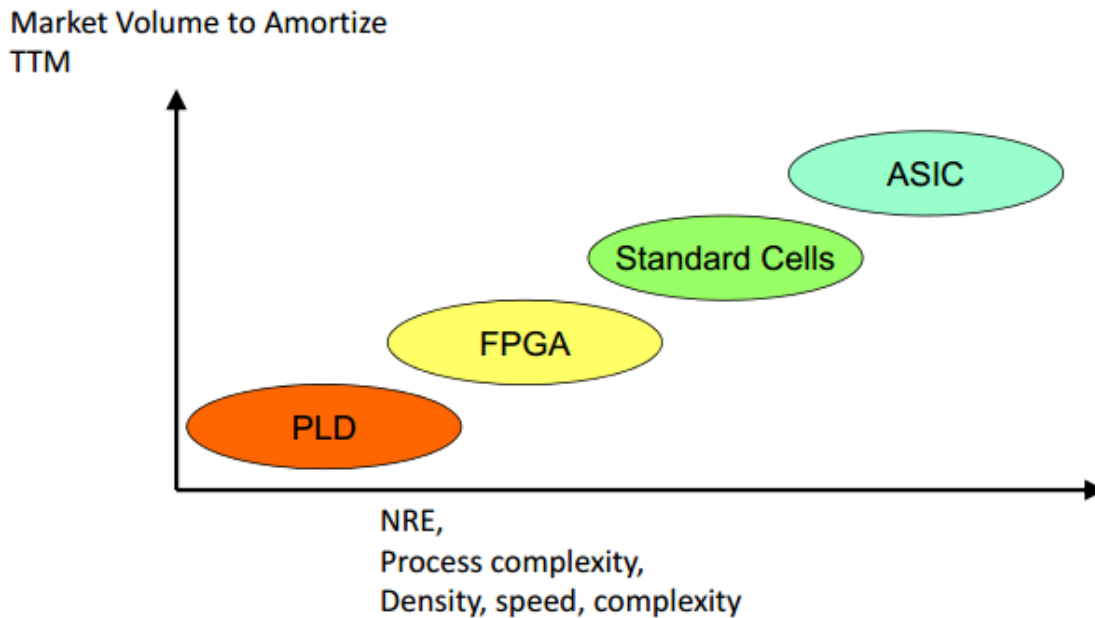
Originally there was a perception that FPGAs are too power hungry, expensive and slow to be used in embedded systems but with the emergence of Low power diverse packages FPGAs are now seen in several embedded systems including but not limited to medical devices, cameras, network switches, routers, military radios, etc.

“FPGAs have become incredibly capable with respect to handling large amounts of logic, memory, digital-signal-processor (DSP), fast I/O, and a plethora of other intellectual property (IP)” [8].

Recently more and more commercial solutions are available which speed up large scale computationally intensive tasks using FPGAs among them the prominent ones include Maxeler [9] and Convey [10].

Apart from all these factors the standard microprocessor based computing is itself faced with acute challenges and is not promising any drastic speed ups. The standards microprocessors face the challenge of power wall, memory wall and ILP wall and the future of computing is towards heterogeneous multi core computing and parallel paradigms. This shift in programming paradigm is massively favorable to FPGA based computing.

There are other business related factors that are now in favor of FPGA based computing and developers would need to keep an eye on when choosing between a Microprocessor, FPGA or an ASIC



**Figure 1.2 [11]**

- 1. Time to Market (TTM):** Since product lifecycles are becoming short, FPGA's biggest strength is the time to market compared to ASIC. The design process of an ASIC involves Design, Verification, Fabrication, Packaging and Device Test. Apart from all this Software Development for an ASIC can only be started when the final product comes in hand. Whereas the FPGA is already fabricated, packaged and tested by the vendor this immediately cuts almost four months from the initial development time giving access to market earlier.
- 2. Cost:** The unit cost of FPGAs is significantly higher than ASICs, but if the demand is in low volumes, this is balanced by the high ASIC NRE costs.
- 3. Development Time:** FPGAs are developed using standard hardware design approach and they are described in Verilog or VHDL. There are commercial high level synthesis tools which convert Imperative languages such as C to HDLs which significantly further reduces the development time.
- 4. Standard Tools:** The vendor tools available with the commercial tool chains provide significant observability and diversity for debugging and application development.

**5. Reprogramability:** ASICs and microprocessors both provide fixed logic whereas the reconfigurability of FPGAs make them highly programmable

To summarize, FPGAs provide high performance and flexibility. They are becoming as design compatible and portable as ASICs and their costs are nearing the microprocessor spectrum. FPGAs give the highest possible performance when the applications they are utilized in are inherently parallel.

# Challenges faced by FPGA based Computing platforms

Due to current semiconductor technology trends, constraints such as area overhead for reconfigurable substrate remain significant because of the significant resources found on chip and increased power sensitivity in many applications. Hence, FPGAs face several acute challenges before they can be considered a mainstream compute platform.

The applications that can benefit the most from FPGA based computing platforms are required to have certain characteristics which have been summarized by DeHon and Hauck in [12] and these can be broadly classified into:

- Parallelism.
- Streaming.
- Heterogeneous Computing requiring adaptation.

The application developers need to focus on application based migrations from legacy serial software which possess these characteristics to Hardware Descriptions in FPGAs. Automated tools need to evolve to facilitate this migration and automatically exploit these capabilities.

FPGAs enable the hardware to adapt with changes in system requirements this is the reason it FPGA based computing is often referred to as Reconfigurable Computing. But the biggest challenge in this domain is the reconfiguration time of hardware. The reconfiguration speed of hardware must be at speed with real time constraints of its applications.

Since FPGAs are inherently redundant they should be able to provide Reliability by introducing capabilities such as fault tolerance, defect analysis and self-healing. The flexibility of the system should be utilized to isolate the bad parts in a system and new parts or modules must take over. However at present such tools are lacking which can provide such flexibility to FPGA based computing platforms.

The synthesis tools currently under use focus on mapping hardware description languages to efficient hardware implementations based on the metrics preferred by the designer such as speed, power consumption and area. But FPGA based computing platform or RC require an end to end tool chain which not only cover synthesis and analysis but also require the sophistication

provided by the tools to develop complex software. Apart this tool chain other valuable tools include profilers, Visualization tools, Load Balancing Tools, custom instruction set generation, runtime resource management tools and functional simulation. Static Analyzers and Dynamic Analyzers for RC are also an open problem which would not only allow optimization at Compiler or design time but also dynamic reconfiguration.

Since RC's main target domain is Embedded Systems which is an extremely demanding industry to satisfy because not only demand real time performance or adaptability but they also challenging requirements such a Safety Criticality and other certain guarantees which are not explicitly available in FPGA based computing platforms.

FPGA based computing platform certainly provide improved performance, better power efficiency but the usability of such systems remains an open question. Several successful technology ventures have proved that the magic is not in technology but the magic is in usability. In most embedded systems projects the software team turns out to be the main driving force for the selection of a certain processing element. With current tool chains and interfaces FPGAs is a no go area for Software Developers because it is simply too complex and risky for them. As argued by Hartenstein in [13], there is indeed a need to train a programmer population who think parallel and also think about locality. But this legacy code is not going away easily and we need to look in the direction of tools to provide Automatic Dynamic Parallelization but also work on providing high level languages closer to Java and C++ to make FPGAs really accessible.



# The Emergence of Extensible Programming Platform (EPP)

**“A Processor with FPGA accelerators” is more attractive than an “FPGA with Processor inside” – R. Hartenstein**

As established by [14] embedded system designers are simply not fond of FPGAs with CPUs inside. FPGA has entirely remained the domain of Hardware Engineers and software engineers have stayed away from FPGAs. For the very reason of involving the software team right from the start, Xilinx has decided to go beyond the FPGA and introduced the Extensible Programming Platform. In 2010 [15] Xilinx pre announced a new platform with Dual Core ARM Cortex A9 Processing System (PS) along with the programmable logic (PL).

This announcement was based on the fact that Xilinx realizes the importance of Software in Computer systems and wants FPGAs to be considered a computing platform by providing a single chip solution. For several years processors have been implemented in FPGAs where standard FPGA based configurations had to be done after which the soft processor could take over. However in this platform FPGA is attached with the a standard processor and these new chips would have a standard SD Boot like the Raspberry pi [16] and boot binary which also contains the bit stream can configure the FPGA. However the FPGA also be used as bare metal using the JTAG mode which has been explained in later chapters.

ARM and Xilinx both have very large ecosystems around them, with the emergence of ARM in mobile devices there has been extensive development around ARM. Now ARM is looking to enter the desktops and Servers market, whereas Xilinx has always been the pioneer of programmable FPGAs and has large number of pre built 3<sup>rd</sup> party IPs. When these two qualities are combined it makes FPGA based computing platforms a lot more realistic. This approach is extremely software centric, it is now completely upon the discretion of the software developer what to keep in the PS and what to push to the PL right at the design time.

This family of Xilinx Devices has been called the Xilinx Zynq, apart from the PS and PL these chips also contain on-chip memory, external memory interfaces, and a rich set of peripheral connectivity interfaces.

Zynq-7000 All Programmable SoC						
	Device Name	Z-7010	Z-7020	Z-7030	Z-7045	Z-7100
	Part Number	XC7Z010	XC7Z020	XC7Z030	XC7Z045	XC7Z100
Processing System	Processor Core	Dual ARM® Cortex™-A9 MPCore™ with CoreSight™				
	Processor Extensions	NEON™ & Single / Double Precision Floating Point for each processor				
	Maximum Frequency	667 MHz (-1); 733 MHz (-2); 800 MHz (-3)			667 MHz (-1); 733 MHz (-2); 1 GHz (-3)	
	L1 Cache	32 KB Instruction, 32 KB Data per processor				
	L2 Cache	512 KB				
	On-Chip Memory	256 KB				
	External Memory Support <sup>(1)</sup>	DDR3, DDR3L, DDR2, LPDDR2				
	External Static Memory Support <sup>(1)</sup>	2x Quad-SPI, NAND, NOR				
	DMA Channels	8 (4 dedicated to Programmable Logic)				
	Peripherals <sup>(1)</sup>	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO				
	Peripherals w/ built-in DMA <sup>(1)</sup>	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO				
	Security <sup>(2)</sup>	RSA Authentication, and AES and SHA 256b Decryption and Authentication for Secure Boot				
Processing System to Programmable Logic Interface Ports (Primary Interfaces & Interrupts Only)		2x AXI 32b Master 2x AXI 32b Slave 4x AXI 64b/32b Memory AXI 64b ACP 16 Interrupts				

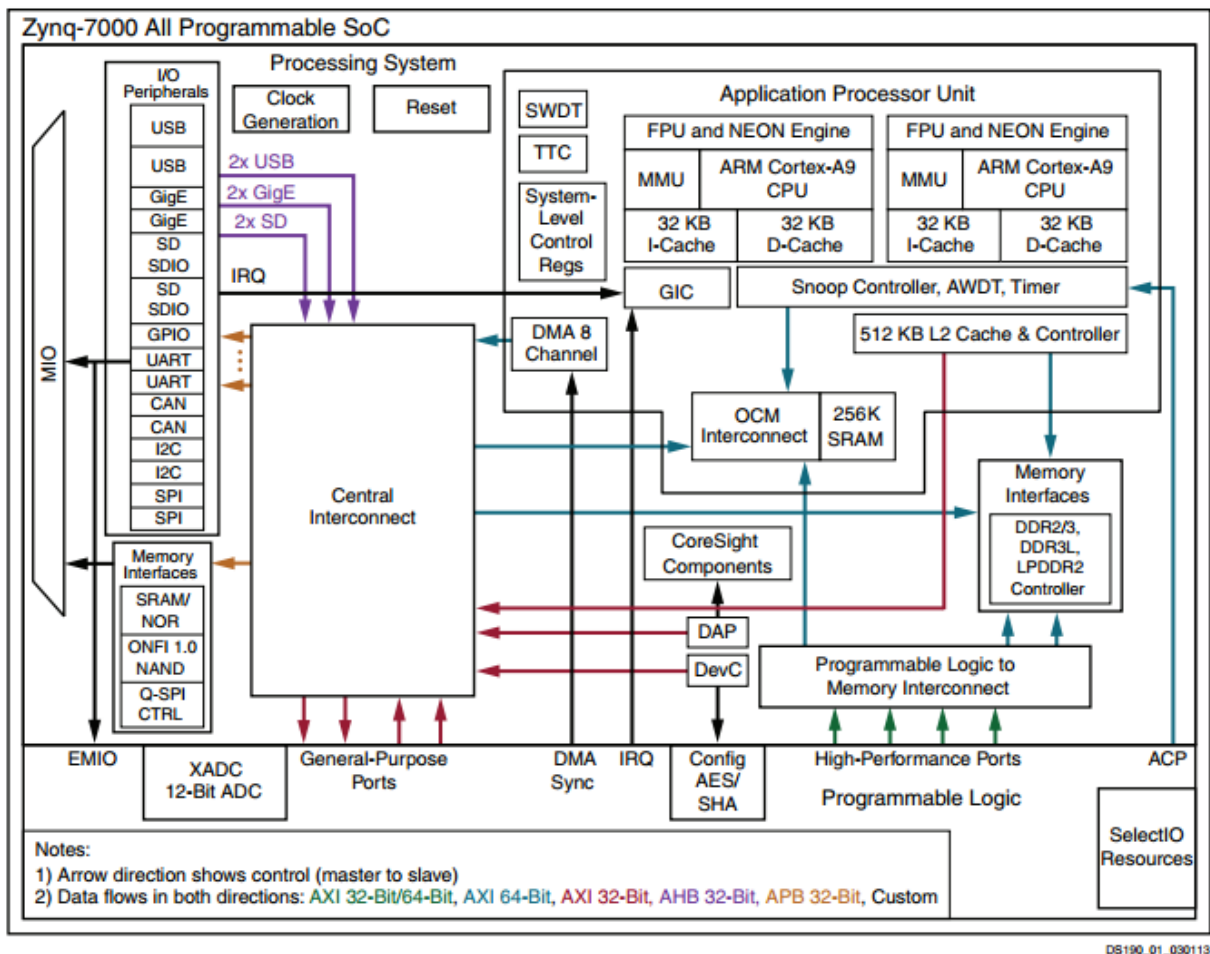
**Table 1.2 [18]**

Table 1.2 illustrates the features of Zynq family of devices on the PS end.

Zynq-7000 All Programmable SoC						
	Device Name	Z-7010	Z-7020	Z-7030	Z-7045	Z-7100
	Part Number	XC7Z010	XC7Z020	XC7Z030	XC7Z045	XC7Z100
Programmable Logic	Xilinx 7 Series Programmable Logic Equivalent	Artix™-7 FPGA	Artix-7 FPGA	Kintex™-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA
	Programmable Logic Cells (Approximate ASIC Gates <sup>(3)</sup> )	28K Logic Cells (~430K)	85K Logic Cells (~1.3M)	125K Logic Cells (~1.9M)	350K Logic Cells (~5.2M)	444K Logic Cells (~6.6M)
	Look-Up Tables (LUTs)	17,600	53,200	78,600	218,600	277,400
	Flip-Flops	35,200	106,400	157,200	437,200	554,800
	Extensible Block RAM (# 36 Kb Blocks)	240 KB (60)	560 KB (140)	1,060 KB (265)	2,180 KB (545)	3,020 KB (755)
	Programmable DSP Slices (18x25 MACCs)	80	220	400	900	2,020
	Peak DSP Performance (Symmetric FIR)	100 GMACs	276 GMACs	593 GMACs	1,334 GMACs	2,622 GMACs
	PCI Express® (Root Complex or Endpoint)	—	—	Gen2 x4	Gen2 x8	Gen2 x8
	Analog Mixed Signal (AMS) / XADC	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs				
	Security <sup>(2)</sup>	AES and SHA 256b for Boot Code and Programmable Logic Configuration, Decryption, and Authentication				

**Table 1.3 [18]**

Table 1.3 illustrates the features of Zynq family of devices on the PL.



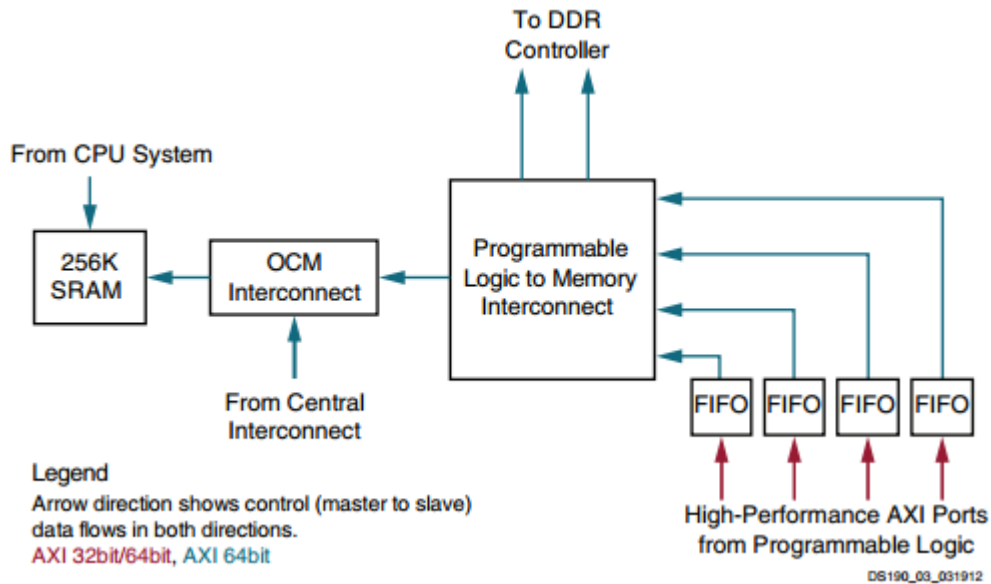
**Figure 1.3 [18]**

Figure 1.3 illustrates the functional blocks present in a Zynq System.

The PS contains the Four Major blocks.

- Application Processing Unit.
- Interconnects.
- Input/Output Peripherals.
- Memory Interfaces.

For more details on these blocks [18] should be referred.



**Figure 1.4 [18]**

Figure 1.4 illustrates PL interface to PS memory subsystem using HP Ports.

Extensive details about the Xilinx Zynq SoC can be found in the Technical Reference Manual [19].

Following the suite Altera the other major manufacturer have stepped up and introduced the SoC FPGA [17] based product line.

With features very similar to Xilinx Zynq the Altera SoC FPGAs promise a peak bandwidth of 125 Gbps between the processors and the FPGA which can be very significant for high performance applications.

Similar to arguments made above Altera [17] promises features such as:

- Low Power Systems.
- Smaller Board Size.
- Lower System Cost.
- Increased Performance.
- Rich ARM ecosystem.

# Why run an OS on FPGA based Computing Platforms

Generally an Operating System is considered the software that turns hardware into something useful. But the main functions it performs include:

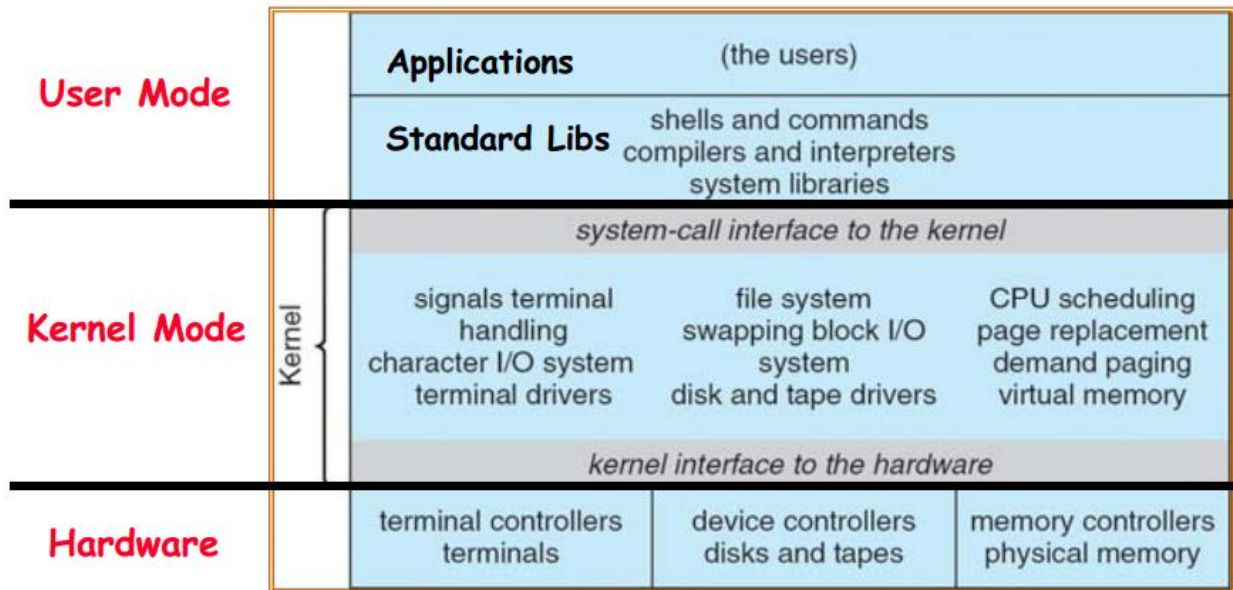
- Multiplex Hardware among multiple applications.
- Abstract Hardware for Programmer's convenience.
- Isolate applications to contain bugs.

These are the main tasks that an operating system performs in all systems these tasks can be further sub divided into.

- Memory Management.
- I/O Management.
- Multitasking/Multiprogramming.
- File System.
- Networking.
- Process Management
- Etc.....

In fact there is no hard and fast rule to what an OS is limited to whatever gets shipped with an operating system is considered a part of the operating system.

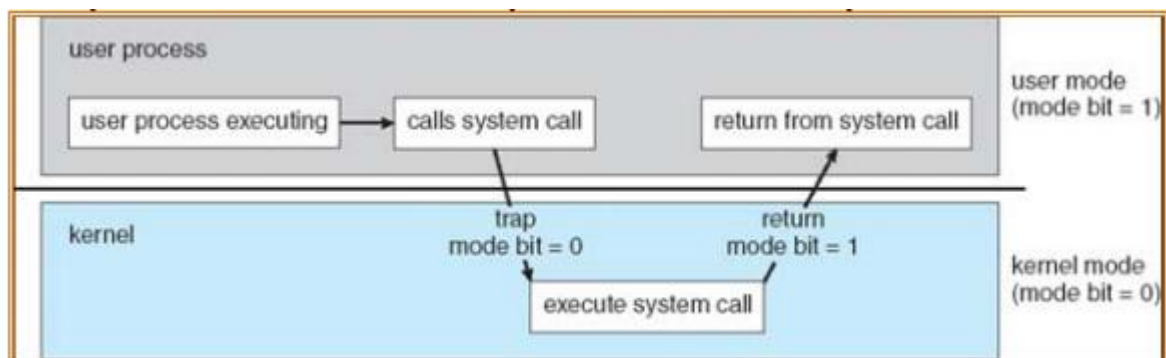
But the definition that satisfies the most is that, the one program the always runs in the operating system is the Kernel and that is what I consider the core OS. Everything else can be categorized as System Software or Application software.



**Figure 1.5 [21]**

Figure 1.5 illustrates the Unix System Structure, all the Unix like system including Linux are based on this structure and this is the classical model illustrated by [Silberschatz, Galvin, Gagne], this figure also represents the differentiation between the User Space, Kernel Space and Hardware.

This typical User Mode and Kernel mode is well known as dual mode operation, normal programs are executed in the user mode but they can transfer to the Kernel Mode to do certain supervisor tasks which cannot be executed in user mode.



**Figure 1.6 [21]**

Figure 1.6 illustrates a typical Unix transition from user mode to Kernel mode for executing a system call, similarly this can occur for interrupts and exceptions.

As it has been pointed out and similarly has been discussed by Tanenbaum in [22] the two most important roles that Operating Systems perform are managing shared resources and simplifying the programming platform with the usage of an abstracted programming model. Both these functionalities are extremely valuable for FPGA based computing platform. As pointed out in [20] an operating system coupled with the compilation toolchain can truly simplify the programming of an FPGA based computing platform. This model would not only define the software tasks that would run on the PS but also define hardware tasks for PL.

The basic functionalities that an OS is expected to perform on an FPGA based Computing platform are.

- Providing methods for communicating and synchronization between PL and PS.
- Abstraction of the resources and configuration on the PL.
- Scheduling of overlapping resources within and across tasks.
- Protection of PL tasks from PS tasks.

Apart from these features it has been pointed out in [23] that it is extremely desirable to have a unified development environment for both hardware and software to maximize functionality and reduce debugging effort. In [25] Rupnow has also emphasized the importance of running an OS to manage multiple compute resources in order to maximize the system performance. Moreover in [26] Khoa et al have written about utilizing Microkernel OS based hypervisors to achieve application isolation and simpler use of Hardware resources. The virtual machine concept provides the developer the illusion of dedicated use of Compute resources [27].

The concept of utilizing an operating system with FPGA based computing platform is not new and has been around since 1996 [24] similarly soft CPUs have been around for a long time and since 2009 Linux Kernel has supported the Xilinx's soft CPU Microblaze out of the box. But even these developments have not been able to support the FPGA computing cause and have been victim of hardware/software dilemma. Since FPGAs are a domain of Hardware engineers they have always been happy to forgo the abstraction provided by OS and have created workarounds by including hardware management operations.

But since the complexity in embedded systems is growing very rapidly these workarounds seem insufficient. One should consider the Cellular phone example which started with providing Calling functionality and other basic functionality which was very achievable without an operating system. Today a Cellular phone not only makes phone calls, but also takes photos, support gaming, provide Internet browsing, run video and audio playback. Not only do Cellular phones perform several functions but also are able to multi task, all due to the abstraction provided by the operating systems. Even in the mobile phone market Unix like operating systems have been clear winners. Clearly Cellular phones have become the “General Purpose” handheld device, although FPGAs are not aiming to become “General Purpose” Computers but clearly FPGAs must be able to adapt to some kind of OS.

**Task Binding:**

Since FPGAs are inherently flexible, they demand much more intensity in terms of shared resource management from the OS. The complexity is in binding tasks to the hardware resources available in terms of Hardware Resource utilization and Performance. Tasks can be implemented by dynamically linking them to pre compiled libraries of hardware [28], these libraries can be implemented as part of the drivers in the OS.

**Scheduling:**

Scheduling is another very important task for OSs in FPGA based platforms, because which resource has to be used when by a task has to be decided by the OS. The simplest reconfiguration scheduling is to run a queue and reconfigure on demand [24]. The application can be profiled and analyzed to form a static schedule for it and based on this analysis hardware tasks can be configured [29, 30 31]. This method can also minimize configuration overhead. Scheduling for performance is a very important factor but scheduling for deadlines is also significantly important. Real time systems and the operating systems they utilize focus significantly on this research are [29, 32]. The scheduling algorithms for real time dead line can be tailored for FPGA based computing platform based on their Hardware capacity, the tasks hardware requirement and the deadline to execute the task [33, 34]. Scheduler must also have the feature of preemption to allocate hardware resources to a task of higher priority [34].



**Communication:**

Communication between tasks is also another very important abstraction provided by the operating systems and achieving this in FPGAs is not very straightforward. Shared memory model is the common communication style used in multi core systems but in FPGA based systems data might have to be copied from a far away memory which can insert significant communication time overhead complicating the shared memory abstraction. Synchronization between shared task is most common source of error in shared memory system and the viability of this model has been questioned in [35, 36] in massively parallel systems.

Since shared memory seems unviable for FPGA based systems the second method used is method calls. Message passing is a technique of method call similar to the remote procedure call [37]. Another method call implementation is MPI [38] and MPI has already been tested with FPGA based computing platforms [39] but MPI is considered heavy weight for FPGA based platforms specially for fine grained tasks. A light weight method call system has been developed Nollet et al as discussed in [40] for reconfigurable FPGA systems. Similarly RPC have also been developed [41] for use between PS and PL.

Method call is a communication mechanism which is very dynamic and the OS has no information about which task is going to talk when. Streams are based on graph structures for task communication and they can be used to transmit data and control information. The process receives data from one or multiple streams and uses these inputs to compute data for one or multiple streams on the output [42, 43].

**Virtual Memory:**

When FPGAs are tightly coupled with a processor with MMU support, the FPGA can share processor's MMU [44]. The processor can now be used by the OS to perform memory accesses to feed data to the FPGA for computation [45]. This model brings good control but reduces the ability of the processor to act as a compute unit as it is kept busy in memory transfers. DMA controllers can be implemented to counter this issue of handling memory transfers.

**I/O:**

Apart from communicating with other tasks the tasks would need to communicate with I/O in the system. Chang discusses the libraries in [46] which can be used to abstract hardware interfaces from the programmer. Apart from these libraries the driver implementation in the OS should be generic enough to deal with changes in the hardware of same family.

**Synchronization:**

FPGA based computations are inherently concurrent where more than one hardware tasks occur in parallel with software tasks, in such scenarios synchronization between tasks is a critical issue. With parallelism the problem of concurrent programming is exacerbated [47]. In such circumstances synchronization abstraction provided by OS is extremely critical for design and performance. The simplest method is the thread style synchronization [48, 49] on hardware tasks in this style of synchronization semaphores are used to control the hardware. The thread style synchronization is difficult to design and debug and with degree of parallelism in FPGA based system implementing this is simply very challenging [35, 36]. In such scenarios we can use implicit synchronization using a scoreboarding technique as described in [50] for the prevention of hazard in aggressive processor pipelines.

**Protection:**

All computing systems need to protect processes or tasks from interfering with each other not only for sake of data correctness but also to prevent malicious task to destroy the good task. When this comes to hardware implementations short circuits is the biggest concern [51, 52], the operating system must be able to find out and deny any such implementations. To provide inter task protection, communication can be restricted to only via the shared virtual memory [45, 53], but this methodology introduces significant amount of latency into the system. However if operating system allocates resources [43] which eliminates user control, this strategy generates an inherent isolation in the programming model.

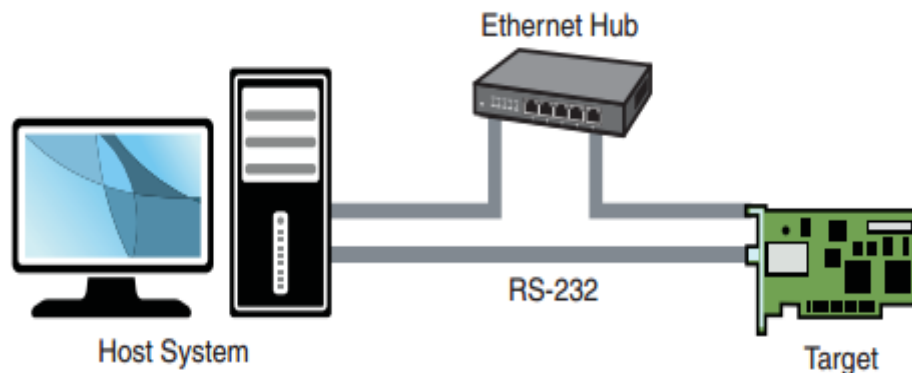
## **Section 2**

# **Running Linux on Xilinx Zynq**

# Building the Tool chain

The first step to running Linux or any other operating system on any embedded system is to set up a cross compiler tool chain for building the Kernel and other system and application software. The word cross implies that the system that compilation takes place on is of one family called the *host* usually x86, whereas the system that compilation is done for is of another family called the *target* which in our case is ARM.

This is done because the target which is usually an embedded system may not have enough memory, disk space to contain the native compilation tool chain. The system might not even have an interface to give commands for running the compilation environment. Another very important reason for doing cross compilation is the compilation speed that is offered by standard x86 systems. This is referred to as a complete tool chain because a compiler alone is of no use. A compiler requires linkers, assemblers and standard libraries to successfully compile a program.



**Figure 2.1 [54]**

Figure 2.1 illustrates a very typical cross development platform where the host is connected to the target using Serial Console and the Ethernet. The Ethernet only becomes useful after some sort of kernel is loaded onto the target until then all commands are passed via the serial console. This configuration can be utilized for both development and debugging the kernel and the application and system software.

The compilation Tool chain for Xilinx Zynq is provided by Mentor Graphics at [59] and called the Sourcery CodeBench for ARM GNU/Linux which provides open source development for C/C++ on x86 host machines.

This tool chain can also be obtained from the Zynq repository.

<https://dl.dropbox.com/u/44609249/xilinx-2011.09-50-arm-xilinx-linux-gnueabi.bin>

After the tool chain has been installed export the environment variables with the following commands.

```
bash> PATH=~/CodeSourcery/Sourcery_CodeBench_Lite_for_Xilinx_GNU_Linux/bin:$PATH  
bash> export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
```

Utility	Use
<i>as</i>	GNU assembler
<i>ld</i>	GNU linker
<i>gasp</i>	GNU assembler pre-processor
<i>ar</i>	Creates and manipulates archive content
<i>nmu</i>	Lists the symbols in an object file
<i>objcopy</i>	Copies and translates object files
<i>objdump</i>	Displays information about the content of object files
<i>ranlib</i>	Generates an index to the content of an archive
<i>readelf</i>	Displays information about an ELF format object file
<i>size</i>	Lists the sizes of sections within an object file
<i>strings</i>	Prints the strings of printable characters in object files
<i>strip</i>	Strips symbols from object files
<i>c++filt</i>	Converts low-level, mangled assembly labels resulting from overloaded C++ functions to their user-level names
<i>addr2line</i>	Converts addresses into line numbers within original source files

**Table 2.1 [56]**

The most common utilities found in a Tool chain are listed in the table above along with their functionalities.

After the compilation has been done on the host, the next phase is getting the executable into the target system. There are several ways to do including.

- Building a Network File System (NFS) and copying the executable on to the target or running it directly from the host memory.
- Copying the executable on the SD card and mounting the desired partition to access the executable.
- The file can also be copied over the network using *SCP* or *FTP*.

Next is the execution phase where the executable can be invoked by a script or directly from the over the serial console or SSH.

The tool chain provided by Xilinx uses standard C library called the *glibc*, however this considered too bloated and has a large foot print for embedded system development and other options available to developers are *uClibc* [60] and *Buildroot* [61]. Both these libraries are highly customizable and are available for ARM processors.

# Building the Boot Loader

The boot loader itself runs for a very negligible time on the system but performs some of the most important tasks to bring up the system these tasks can vary from a General purpose system to an embedded system and there are several variations in Embedded System too but generally an Embedded System boot loader perform the following the following functions.

- Provide initialization code for the board to load which is usually written in native assembly of the operating processor.
- Flushing the processor Cache.
- Load Processor registers with useful values.
- Determine the hardware present in the system.
- Pass the hardware information to kernel.
- Loading the Kernel into the memory.
- Executing the Kernel.
- Loading the init file system.
- Validate the Operating System Image.

The boot loader of choice these days for ARM based embedded systems is U-Boot [62]. Xilinx has also chosen U-Boot as the default boot loader for Zynq. U-Boot is a sensible choice because it is a universal boot loader it is compatible across several different architectures.

There are several points in its favor including:

- Free and Open Source.
- Active Community Support.
- Large Architecture level support including ARM, MIPS, PPC and x-86.
- Lower Development Cost.
- Highly Optimized for Embedded Systems.
- Large Debugging and Development Support.
- Easy to port from one platform to another.

After the compilation tool chain has been built it would be used to compile the boot loader.

Clone the boot loader source code from Xilinx's repository.

```
bash> git clone git://git.xilinx.com/u-boot-xlnx.git  
  
bash> cd u-boot-xlnx
```

Configure the boot loader for a specific Chip which in our case is the Xilinx Zynq ZC7000 series.

```
bash> make zynq_zc70x_config
```

After the configuration has completed build the boot loader using the make file.

```
bash> make
```

This build would produce *u-boot.elf* which would be later used to build the BOOT.BIN in order to boot the Zynq system.



# Building the First Stage Boot loader

The on-chip ROM code loads when the CPU is powered up, this code seeks for FSBL and upon success it loads the FSBL into the memory.

The tasks that FSBL is responsible for include:

- Configuring the bitstream on to the FPGA.
- Load the DDR controller.
- Load and execute U-Boot from SD card into the RAM.
- Initialize the clock Phase Lock Loop.

Beyond FSBL the U-Boot takes over and performs its roles.

To general a new FSBL open the Xilinx SDK.

**File → New → Application Project.**

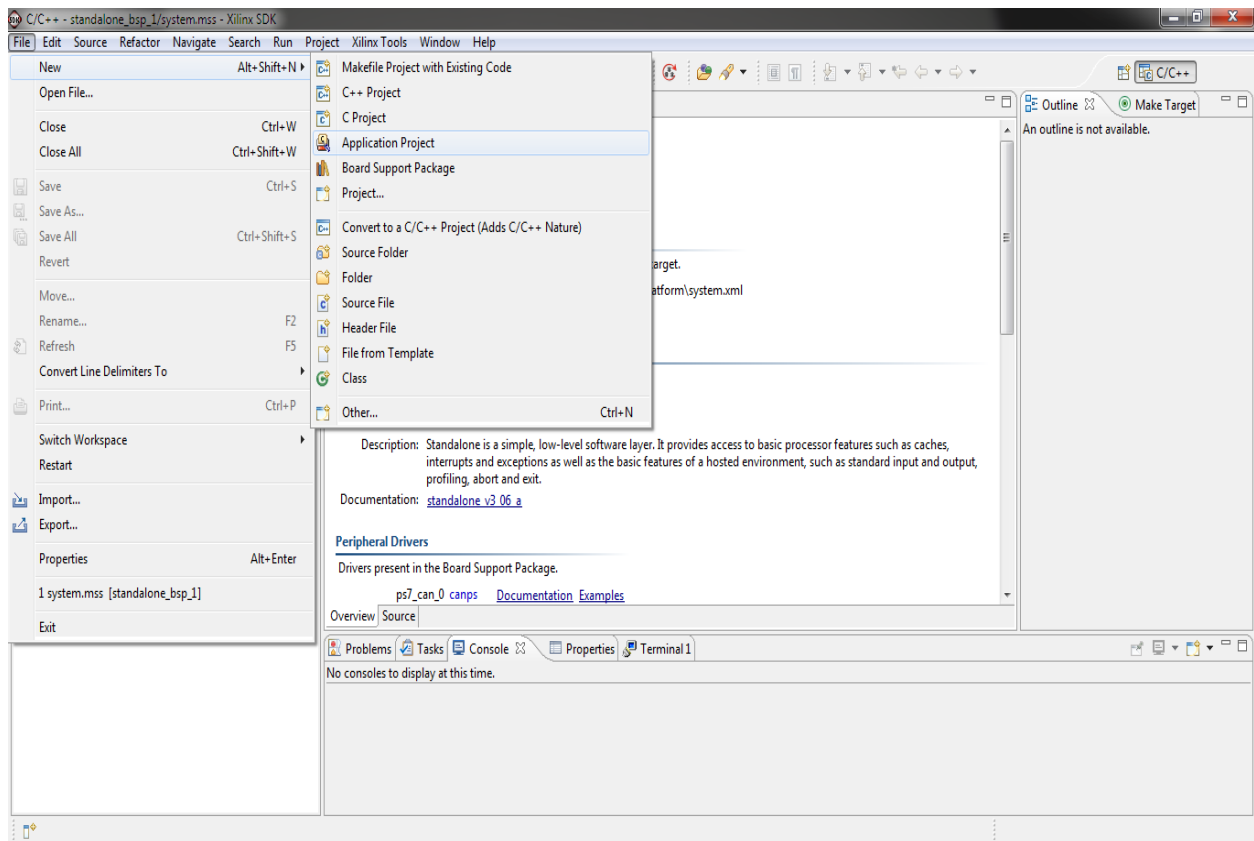
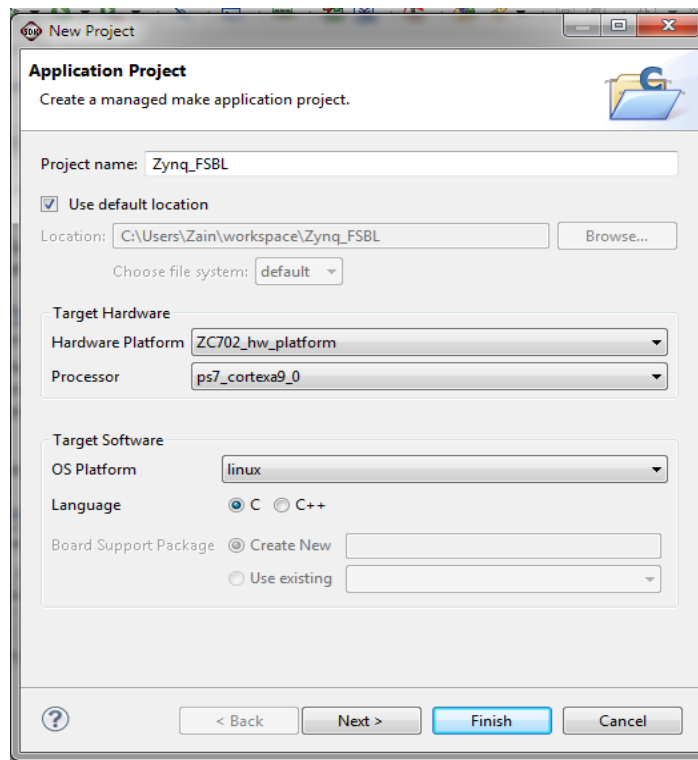
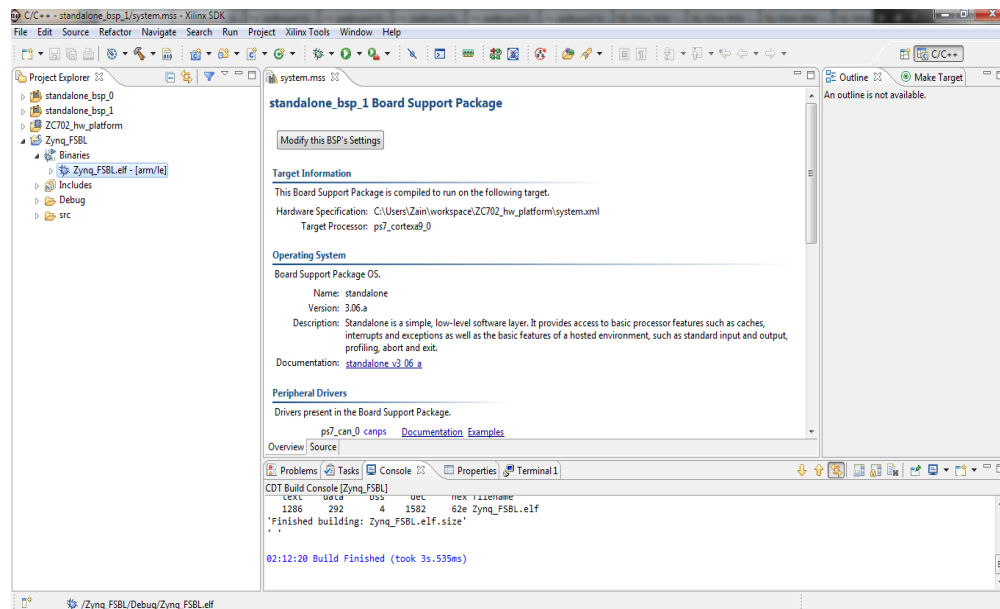


Figure 2.2



**Figure 2.3**

Setup the New project according to the settings above, click Finish.



**Figure 2.4**

Upon Build completion the SDK would generate an ELF which would be later used to generate the BOOT.BIN

# Building the BOOT.BIN

BOOT.BIN is the boot image stored in the SD card it contain the boot loader ELF, the FSBL and the bitstream to configure the FPGA.

To create a bit stream launch the Xilinx SDK.

**Xilinx Tools → Create Boot Image.**

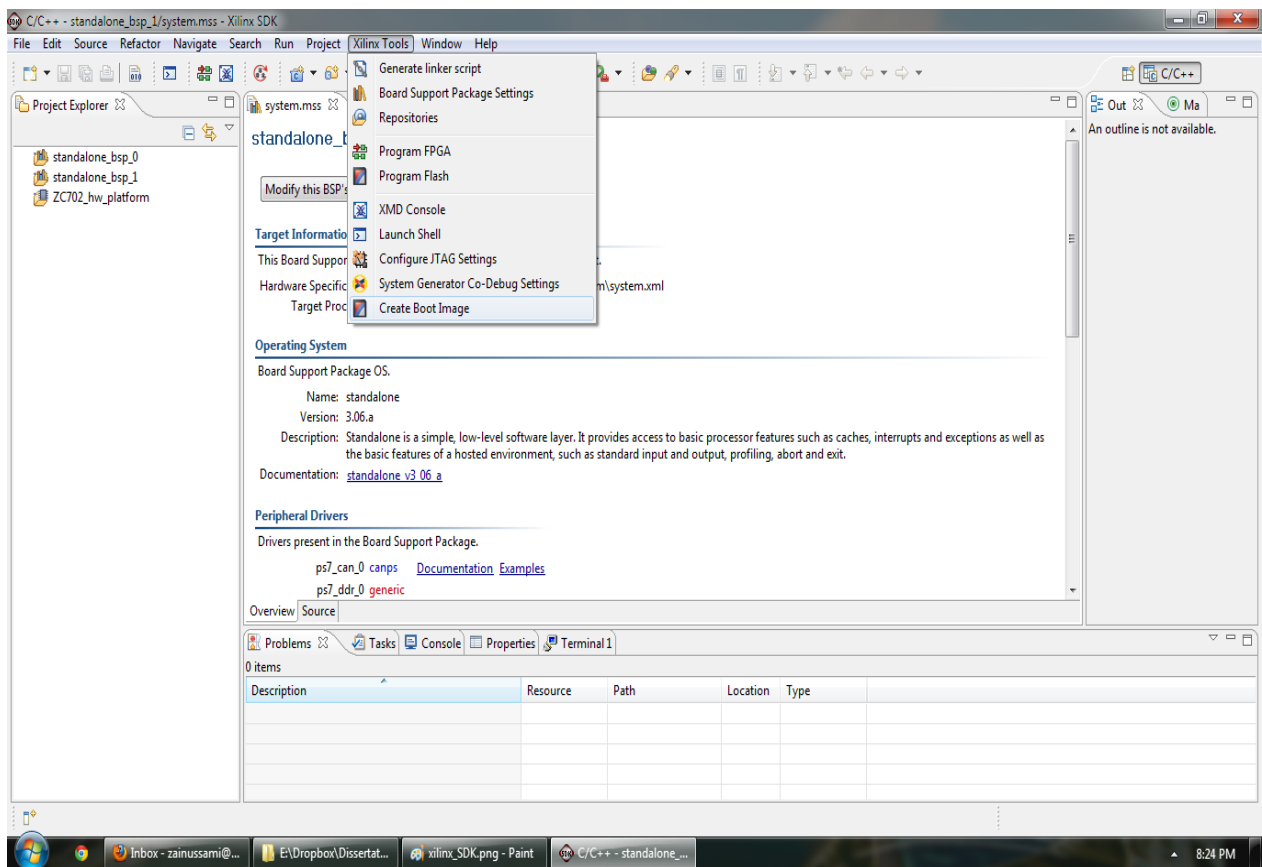
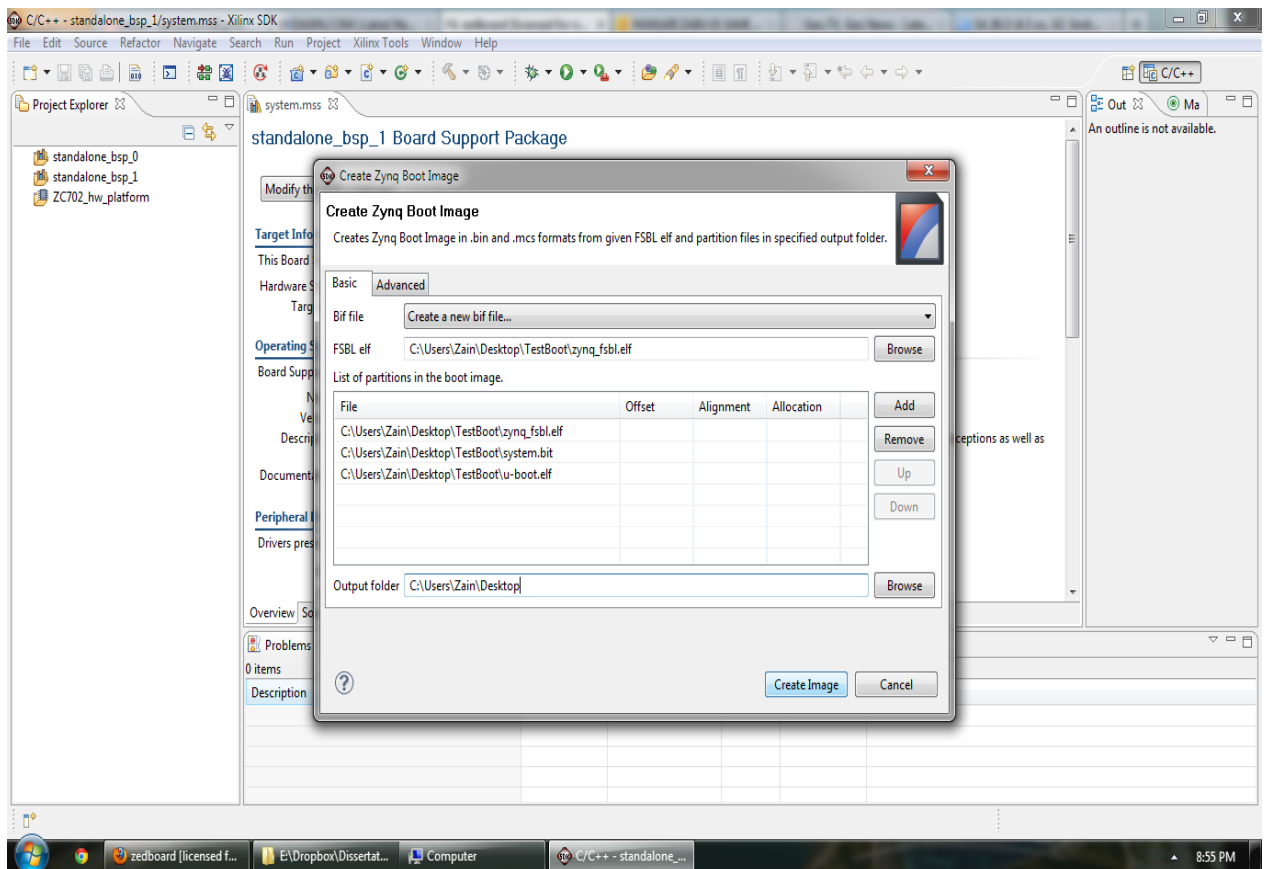


Figure 2.5

Load the files

```
--zynq_fsbl.elf  
  
--system.bit  
  
--u-boot.elf
```



**Figure 2.6**

The particular order of the files has to be maintained for the system to load properly.

# Building the Device Tree

Device tree blob is a database of the hardware connected to a certain board. So when a new peripheral is created into the PL region a new entry needs to be created for it in the device tree.

In this example a FIFO has been attached and its entry in the device tree looks such.

```
fifo_dma1: fifo_dma@7C000000 {  
  
    compatible = "xlnx,fifo-dma";  
  
    reg = <0x7C000000 0x2000>;  
  
    fifo-depth = <4096>;  
  
    dma-channel = <2>;  
  
    burst-length = <4>;  
  
};
```

After the device has been added to the device tree compile the DTS to DTB.

```
bash> cd linux-digilent  
  
bash> scripts /dtc/dtc -I dts -O dtb -o /path/to/devicetree.dtb /path/to/devicetree.dts
```

After compiling the device tree copy binary to SD card.

# Building the Linux Kernel

Next step is to build the kernel for running on the Zynq. It is the most fundamental component in the Linux system.

Processor architecture	System and processor selection option	Kernel architecture name
x86	Processor type and features	i386
ARM	System Type	arm
PPC	Platform support	powerpc (or ppc for older targets)
MIPS	Machine selection/CPU selection	mips
SH	System type	sh
M68k	Platform-dependent support or processor type and features	m68k or m68knommu
AVR32	System Type and features	avr32

**Table 2.2 [56]**

The major architectures supported by the Linux kernel are listed in the table above.

To begin compiling the Kernel, obtain the Linux Kernel Source Tree from Digilent's GIT repository.

```
bash> git clone git://github.com/Digilent/linux-digilent.git  
  
bash> cd linux-digilent
```

Now configure the Kernel to be compiled for Zedboard and Compile it.

```
bash> make ARCH=arm digilent_zed_defconfig  
  
bash> make
```

Upon successful compilation the built kernel image would be formed at location

```
linux-digilent/arch/arm/boot/zImage
```

Copy this image to the SD Card.

# Running Zynq on QEMU

QEMU [65] is an open source emulator which has the ability to execute the code one on architecture on another by using dynamic translation. For slow processors it is able to achieve near native performance on a fast x86 machine.

There is a strong case for using QEMU because all the developers cannot be given the target machine to work on due to cost and availability constraints, whereas x86 machines are readily available to each and every application developer. Apart from these reasons there is a trend of disposing the cross compiler and using the native compilers on QEMU. This gives the best of performance and optimization.

Model Source	Hardware Block
QEMU	Cortex A9
QEMU	NEON
QEMU	ARM MPCore
Xilinx	EMAC
Xilinx	Timer Counter
Xilinx	UART
Xilinx	USB
Xilinx	I2C (with switch and EEPROM)
Xilinx	DMA
Xilinx	SPI (with flash)
Xilinx	NAND
Xilinx	NOR
Xilinx	QSPI (with flash)

**Table 2.3 [66]**

Table 2.3 illustrates the hardware components of a Xilinx Zynq System that have successfully implemented on the QEMU.

To run Xilinx Zynq on QEMU, Download the precompiled QEMU source with Zynq kernel from the link.

[http://zedboard.pbworks.com/w/file/64538134/zynq\\_linux.tar.gz](http://zedboard.pbworks.com/w/file/64538134/zynq_linux.tar.gz)

After downloading the compressed source, uncompress it and run the script.

```
bash> tar xvzf zynq_linux.tar.gz

bash> cd zynq_linux

bash> ./start_qemu.sh
```

# Writing Device Drivers for Zynq

Device driver development becomes the most important segment in a FPGA based system running an operating system like Linux. The peripherals created onto the PL would need an interface to talk to the PS so that data and control information can be exchanged. A device driver is system software directly associated to the hardware at the lowest level

There are three main types of Device Drivers:

- **Character Devices:** These are the drivers that would be most commonly utilized in FPGA based system. These drivers treat devices as files by giving the open, read, write and close functions. They usually allow only sequential access and written and read byte by byte at a time.
- **Block Devices:** In a Linux based system Block devices also behave like character devices with an additional capability of transferring data multiple block sizes. They also allow random access to device and occasionally file systems are mounted using these device drivers.
- **Network Devices:** This kind of device driver does not handle bytes or streams of data but packets of data. These drivers usually utilize the BSD sockets API to communicate with the hardware. All the network interfaces are mapped using these device drivers.

The user space and kernel space has already been explained in an earlier section, so according to that particular model the device drivers or the modules execute in the Kernel Space whereas the application that calls the drivers and send commands and data to it is executed in the user space. The device driver can be made part of the Kernel tree during the Kernel compilation or the device driver can also be loaded into a running kernel as a loadable module.

Developing device drivers through modules is the most convenient method of device driver development. Modules are loadable code which can loaded into the kernel using *insmod* and removed from the kernel using *rmmmod* as and when needed. All device drivers can be implemented as kernel modules but all modules are not necessarily device drivers.



Following is an implementation of Kernel Module Device Driver implemented on Xilinx Zynq for register access peripheral on the PL.

In the Register Access core, three registers are memory mapped in the PL region and they can be accessed from PS.

PS writes to a Register in the PL and as soon as the acknowledgement for write arrives, the core starts counting the number of cycles in the third register. The PS then writes to the second register on the PL and as soon as the acknowledgement for write arrives, the counter stops counting in the third register giving a good estimate of the write delay and register access latency in terms of number of cycles.

Copyright (c) 2013 Ansari Zain Us Sami Ahmed ([s120050@ntu.edu.sg](mailto:s120050@ntu.edu.sg))

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/io.h>

#define SUCCESS 0

#define DEVICE_NAME "/dev/reg_access"

#define REG_ADDRESS 0x7C600000

static int *mmio;
static int major_num;

static int Device_Occupied = 0; //Set to 1 When Device is under Use
```

```

static int device_open(struct inode *inode, struct file *file)
{
    if (Device_Occupied)
        return -EBUSY;

    Device_Occupied++;

    try_module_get(THIS_MODULE);

    return SUCCESS;
}

static int device_release(struct inode *inode, struct file *file)
{
    Device_Occupied--;

    module_put(THIS_MODULE);

    return SUCCESS;
}

static ssize_t device_read (struct file *file, char __user * buffer, size_t length,
loff_t * offset)
{
    return SUCCESS;
}

static ssize_t device_write (struct file *file, char __user * buffer, size_t length,
loff_t * offset)
{
    return SUCCESS;
}

```

```

int device_ioctl(struct file *file, unsigned int ioctl_num, unsigned long ioctl_param)
{

    mmio[1]= (int *) 0x00000001;

    mmio[2]= (int *) 0x00000002;

    printk(KERN_INFO "The Write Latency is %d Cycles\n", *(unsigned int *) mmio);

    return SUCCESS;

}

struct file_operations Fops = {

    .read = device_read,

    .write = device_write,

    .unlocked_ioctl = device_ioctl,

    .open = device_open,

    .release = device_release,

};

```

```

int init_module()
{
    int ret_val;

    major_num = register_chrdev(0,DEVICE_NAME, &Fops); //TO Register the Character
Driver

    //Negative Values Signify the Errors in the Code

    if (major_num < 0)
    {

        printk(KERN_ALERT "%s failed with %d\n","Sorry, registering the character
device ", ret_val);

        return ret_val;

    }

    printk(KERN_INFO "%s The major device number is %d.\n", "Registration was
Successful",major_num);

    printk(KERN_INFO "If you want to talk to the Device Driver, \n");

    printk(KERN_INFO "Than create the following device file by the command. \n");

    printk(KERN_INFO "mknod %s c %d 0\n", DEVICE_NAME, major_num);

    mmio = ioremap(REG_ADDRESS,0x100);

    return 0;
}

void cleanup_module()
{
    int ret;

    //Unregister the Device

    iounmap(mmio);

    unregister_chrdev(major_num,DEVICE_NAME);
}

MODULE_AUTHOR("Zain Ansari");
MODULE_DESCRIPTION("Xilinx Zynq Reg Access Driver");
MODULE_LICENSE("GPL v2");

```

## Code Walkthrough Register Access Driver:

The register access driver works as follows.

```
#define REG_ADDRESS 0x7C600000
```

The REG\_ADDRESS defines the physical base address to be access on the memory mapped PL all registers are implemented with respect to this address.

```
.read = device_read,  
  
.write = device_write,  
  
.unlocked_ioctl = device_ioctl,  
  
.open = device_open,  
  
.release = device_release, #define
```

This device driver supports the file operations mentioned above, the functions read and write have not been utilized but are implemented for complex operations in future.

```
init_module()
```

When the user *insmod* this kernel module into the kernel the above function is called mapping the physical addresses to the virtual addresses and requesting the user to create a device node for the driver to access it from the user mode.

```
cleanup_module()
```

When the user calls *rmmmod* the above function is called causing the physical to virtual mapping being released.

```
static int device_open(struct inode *inode, struct file *file)
```

The above function has been implemented to achieve a lock or a mutual exclusion on the device driver so only one user mode program can access it at a particular instance.

```
static int device_release(struct inode *inode, struct file *file)
```

The above function has been implemented to release the mutual exclusion lock on the device driver so when a user mode application has finished utilizing the driver it is available for other applications.

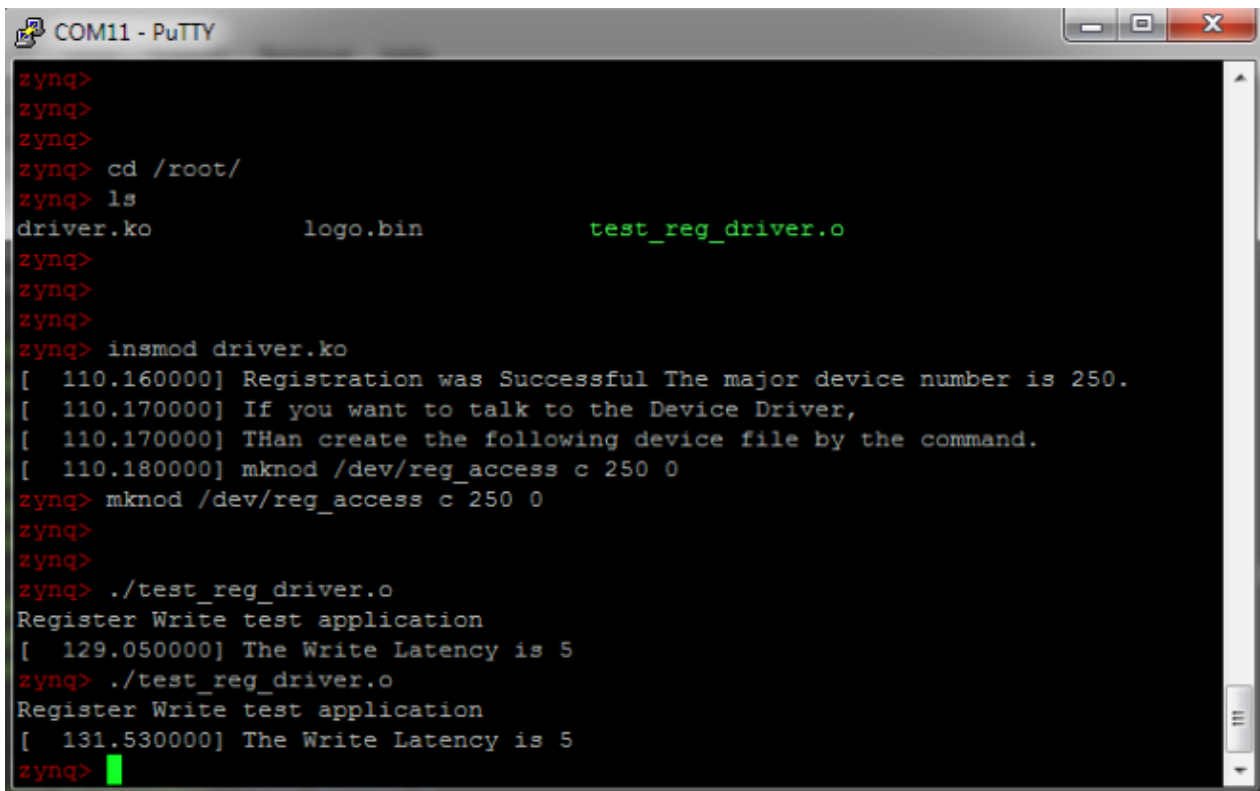
```
int device_ioctl(struct file *file, unsigned int ioctl_num, unsigned long ioctl_param)
```

The above function is the main function of the device driver which is called using:

```
ioctl(file_desc, 1, NULL); // Call The IOCTL in Kernel Module
```

In this function the driver writes two different values to different physical address mapped to the PL as registers. Upon completion of writing to the first register in the PL a performance counter is invoked which counts the number of cycles it takes to complete the second write which presents the latency between the two operations and prints it to the screen.

The Performance counter register is located at the base address whereas register 1 and 2 are 4 and 8 bytes ahead of the base address respectively.



```
COM11 - PuTTY
zynq>
zynq>
zynq>
zynq> cd /root/
zynq> ls
driver.ko          logo.bin          test_reg_driver.o
zynq>
zynq>
zynq>
zynq> insmod driver.ko
[ 110.160000] Registration was Successful The major device number is 250.
[ 110.170000] If you want to talk to the Device Driver,
[ 110.170000] THan create the following device file by the command.
[ 110.180000] mknod /dev/reg_access c 250 0
zynq> mknod /dev/reg_access c 250 0
zynq>
zynq>
zynq> ./test_reg_driver.o
Register Write test application
[ 129.050000] The Write Latency is 5
zynq> ./test_reg_driver.o
Register Write test application
[ 131.530000] The Write Latency is 5
zynq> █
```

**Figure 2.7**

When the driver is loaded into the kernel using *insmod*, the function *init\_module()* executes. Since Xilinx Zynq running is MMU enabled system this function maps the physical address to the virtual address so that the application program can access it.

The driver requests the user to create a node to access the register access driver file from user space.

```
zynq> mknod /dev/reg_access c 250 0
```

Issuing this command on the Xilinx Zynq Linux system creates file node */dev/reg\_access*, *c* represents the type of driver (Character Driver), 250 is the major number of the driver and 0 is the minor number of the driver.

Following is the application code that talks to driver implemented above.

Copyright (c) 2013 Ansari Zain Us Sami Ahmed ([s120050@ntu.edu.sg](mailto:s120050@ntu.edu.sg))

```
#include <stdio.h>

#include <stdlib.h>

int main()
{
    int file_desc;

    file_desc = open(DEVICE_FILE_NAME, O_RDWR | O_SYNC);

    if (file_desc < 0)
    {
        printf("Cannot open the device file: %s\n", DEVICE_FILE_NAME);

        exit(-1);
    }

    printf("Register Write test application \n");

    ioctl(file_desc, 1, NULL); // Call The IOCTL in Kernel Module

    return 0;
}

#include <fcntl.h>

#include <unistd.h>

#include <sys/ioctl.h>

#define DEVICE_FILE_NAME "/dev/reg_access"
```

In this application the node that has been created is accessed in `file_desc` the `ioctl` function invokes `device_ioctl` function in the kernel module.



A Makefile is also required to compile the Kernel Modules so that a loadable .ko module can be created.

```
ARCH := arm

KER_DIR := /home/zain/linux-digilent

obj-m += driver.o

all:

    make ARCH=$(ARCH) -C $(KER_DIR) M=$(PWD) modules

clean:

    make ARCH=$(ARCH) -C $(KER_DIR) M=$(PWD) clean
```

This Makefile tells the tool chain to use the ARM compiler, the location of Kernel headers and the file which contains the Module code.

Upon execution of this make file .ko modules are generated which can be loaded into the Kernel.

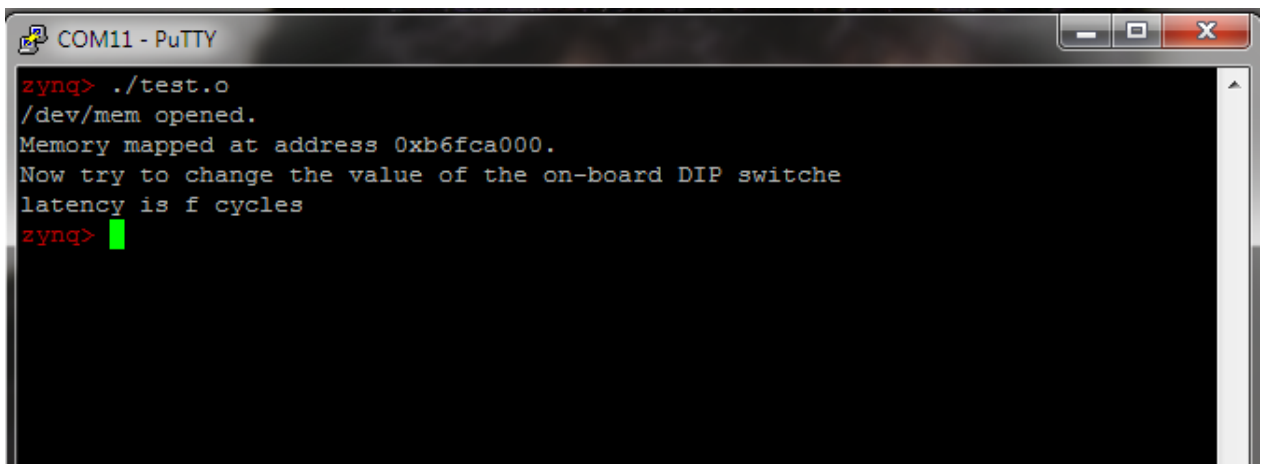
This is not the sole or exclusive method to access the Physical Addresses or peripherals but this is the safest method of doing so. The other common method of writing drivers is writing the complete driver in the user space which has its own pros and cons.

Pros:

- Complete C libraries are available to the user to perform any functions, however when programming in the kernel space a certain set of rules have to be followed with access to limited libraries.
- Easier to compile without access to the Kernel headers.
- Debugging can be relatively simpler compared to Kernel level debugging.
- Source can be close with relative ease.
- User Space memory is swappable.

Cons:

- Interrupts cannot be utilized in the user space.
- I/O and physical address can only be access via mmap on the `/dev/mem` which is only available by root access.
- Higher latency to get data from user space to hardware because of the context switch overhead.
- User space drivers are limited to Character drivers.



```
COM11 - PuTTY
zynq> ./test.o
/dev/mem opened.
Memory mapped at address 0xb6fca000.
Now try to change the value of the on-board DIP switch
latency is f cycles
zynq> █
```

**Figure 2.8**

The figure above shows the run of a user space driver implementation of the same Register Access driver implemented as the Kernel Module. This driver takes *15* cycles to complete the operation whereas the kernel space driver took 5 cycles to complete the same operation.

Copyright (c) 2013 Abhishek Kumar Jain ([abhishek013@ntu.edu.sg](mailto:abhishek013@ntu.edu.sg))

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/mman.h>

#define MAP_SIZE 4096UL
#define MAP_MASK (MAP_SIZE - 1)
```

```

int main(void) {

    int fd;

    int *map_base, *virt_addr;

    off_t target = 0x7C600000;

    if((fd = open("/dev/mem", O_RDWR | O_SYNC)) == -1) {

        printf("/dev/mem could not be opened.\n");

        perror("open");

        exit(1);

    } else {

        printf("/dev/mem opened.\n");

    }

    map_base = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, target &
~MAP_MASK);

    if(map_base == (void *) -1) {

        printf("Memory map failed.\n");

        perror("mmap");

    } else {

        printf("Memory mapped at address %p.\n", map_base);

    }

    virt_addr = map_base + (target & MAP_MASK);

    printf("Now try to change the value of the on-board DIP switch\n");

    virt_addr[1] = (int *)0x00000001;

    virt_addr[2] = (int *)0x00000002;

    printf("latency is %x cycles\n",*(int *)virt_addr);


    if(munmap(map_base, MAP_SIZE) == -1) {

        printf("Memory unmap failed.\n");

    }

    close(fd);

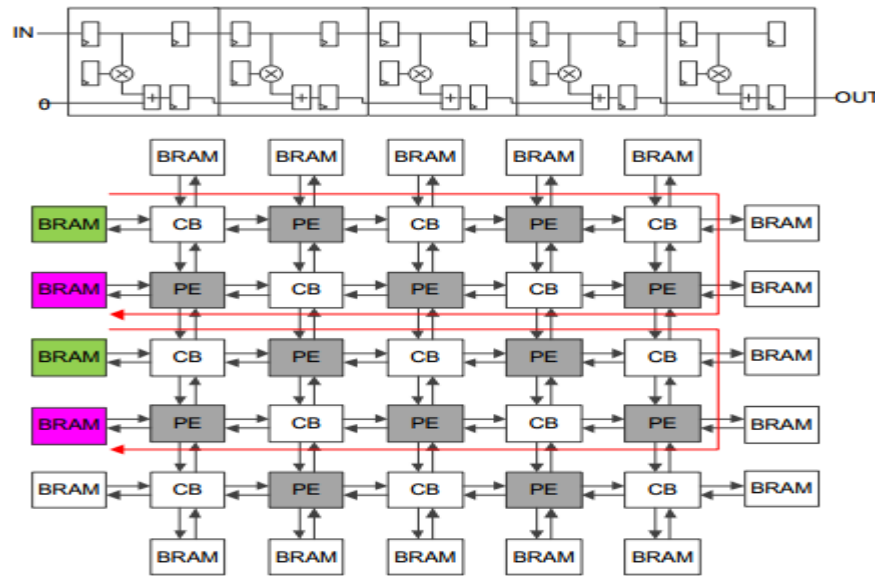
}

```

Even in this User Space driver the physical address that is being accessed is 0x7C600000, it is first converted to a virtual address and all the same functionality is performed on this using */dev/mem* interface.

Similar drivers have also been developed to pass data into a DSP block-based intermediate fabric on the Xilinx Zynq Linux.

### DSP block-based intermediate fabric:

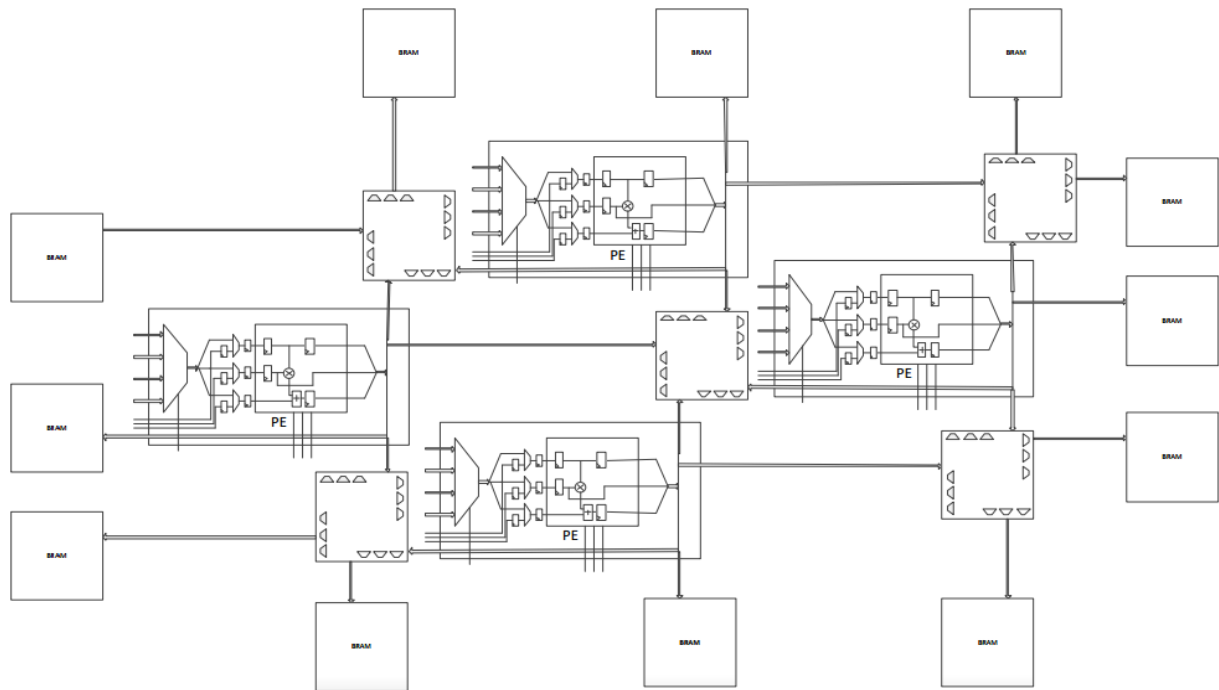


**Figure 2.9[26]**

Figure 2.9 illustrates the implementation of intermediate fabric which consists of 12 DSP slices which are the primary processing elements and 13 connection boxes. The connection boxes are used to provide a programmable interconnection between the processing elements. Each PE is connected to its intermediate neighbors using connection boxes. Dual port Block RAM memory is utilized to transfer data between PS and PL, one port is connected to the IF and second port is connected to the AXI interconnect.

If a simple 4 tap FIR filter is implemented on the intermediate fabric it utilizes 4 PEs and 5 CBs, the input data is transferred into the IF's input BRAM using the AXI interface, the processed data

is stored into the output BRAM which is available to read by the processor. Figure 2.10 illustrates the implementation described above.



**Figure 2.10**

CB Area Overhead: One CB consists of 12 Multiplexers (16-bit 12 to 1). Each Multiplexer is consuming 66 LUTs and 1 slice register. Hence Total LUTs used =  $66 \times 12 = 792$  LUTs

Slice Logic Utilization:

Number of Slice Registers: 12 out of 106400

Number of Slice LUTs: 792 out of 53200

PE Area Overhead: One PE consists of 1 DSP slice and 4 Multiplexers (3 mux (2-to-1) and 1 mux (4-to 1)). Hence Total LUTs used =  $(48 \times 1) + (16 \times 3) = 96$  LUTs.

16-bit 2-to-1 multiplexer : 3

48-bit 4-to-1 multiplexer : 1

Slice Logic Utilization:

Number of Slice Registers: 144 out of 106400

Number of Slice LUTs: 96 out of 53200

Number of Slice DSP Slices: 1 out of 220

Operating frequency of PE = 175.5 MHz

Total area utilization of current IF: 12 PE and 13 CB. Total LUTs used =  $96 \times 12 + 792 \times 13 = 11448$  LUTs

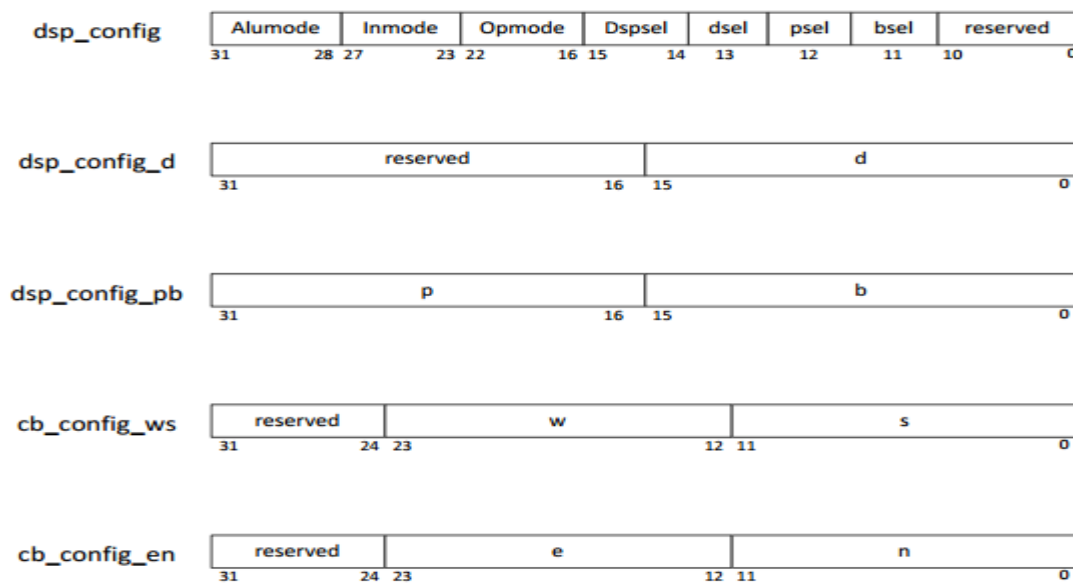
Slice Logic Utilization:

Number of Slice Registers: 156 out of 106400

Number of Slice LUTs: 11448 out of 53200

Number of Slice DSP Slices: 12 out of 220

The IF is configured using the registers specified in the figure 2.11.



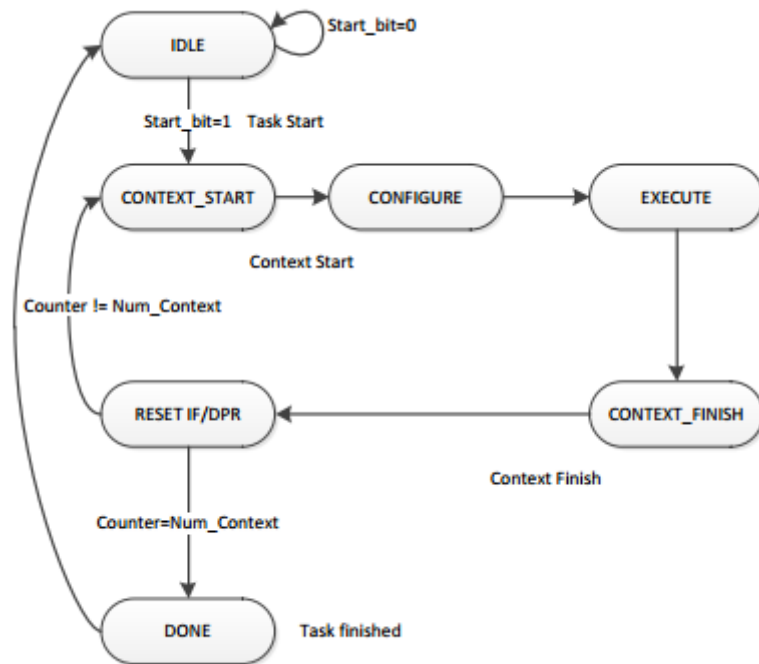
**Figure 2.11**

Three registers `dsp_config`, `dsp_config_d`, `dsp_config_pb` are used to configure the PEs and store the coefficients of the FIR filter.

Two registers `cb_config_ws` and `cb_config_en` are used to configure the routing information on each connection box.

The IF provides a memory mapped interface from the PL to PS a context sequencer is implemented in the PL to control and monitor the execution of hardware tasks on the IF.

Figure 2.12 illustrates the state machine based context sequencer.



**Figure 2.12[26]**

The state machine in figure 2.12 describes all the states from start to finish how a hardware task is executed on IF and is controlled and monitored by the state machine based context sequencer. For complete details on the behavior of context sequencer refer to [26].

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/io.h>

#define SUCCESS 0
#define DEVICE_NAME "/dev/vf_access"
#define MEM_0 0x61C20000
#define MEM_3 0x61C60000

#define NUM_SAMPLES 32
#define dst_mem 0x08000000
#define latency 18
#define start 0x00000001
```



```

static int *mmio; //Map Mem 0 to this Address

static int *mmio1; //Map Mem 3 to this Address

static int major_num;

static int Device_Occupied = 0; //Set to 1 When Device is under Use


static int device_open(struct inode *inode, struct file *file)
{
    if (Device_Occupied)
        return -EBUSY;

    Device_Occupied++;

    try_module_get(THIS_MODULE);

    return SUCCESS;
}


static int device_release(struct inode *inode, struct file *file)
{
    Device_Occupied--;

    module_put(THIS_MODULE);

    return SUCCESS;
}


static ssize_t device_read (struct file *file, char __user * buffer, size_t length,
loff_t * offset)
{
    return SUCCESS;
}


static ssize_t device_write (struct file *file, char __user * buffer, size_t length,
loff_t * offset)
{
    return SUCCESS;
}

```

```

int device_ioctl(struct file *file, unsigned int ioctl_num, unsigned long ioctl_param)
{
    int i;

    int start_cycles=0;

    int end_cycles = 0;

    int *XPAR_VF_0_S_AXI_MEM0_BASEADDR, *XPAR_VF_0_S_AXI_MEM3_BASEADDR;

    int value;

    XPAR_VF_0_S_AXI_MEM0_BASEADDR=mmio;
    XPAR_VF_0_S_AXI_MEM3_BASEADDR=mmio1;


    volatile int *c_base      = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000050;
    volatile int *d_base      = XPAR_VF_0_S_AXI_MEM3_BASEADDR + 0x00000050;
    volatile int *ctrl        = XPAR_VF_0_S_AXI_MEM0_BASEADDR;
    volatile int *stat        = XPAR_VF_0_S_AXI_MEM3_BASEADDR;
    volatile int *count = XPAR_VF_0_S_AXI_MEM3_BASEADDR + 0x0000004F;
    volatile int *count_idle = XPAR_VF_0_S_AXI_MEM3_BASEADDR + 0x0000004E;
    *count=0x00000000;
    *ctrl= 0x00000000;

    volatile int *dsp_config01    = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000001;
    volatile int *dsp_config01_d  = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000002;
    volatile int *dsp_config01_pd = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000003;
    volatile int *dsp_config03    = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000004;
    volatile int *dsp_config03_d  = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000005;
    volatile int *dsp_config03_pd = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000006;
    volatile int *dsp_config10    = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000007;
    volatile int *dsp_config10_d  = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000008;
    volatile int *dsp_config10_pd = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000009;
    volatile int *dsp_config12    = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000000A;
    volatile int *dsp_config12_d  = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000000B;
    volatile int *dsp_config12_pd = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000000C;
    volatile int *dsp_config14    = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000000D;

```

```

volatile int *dsp_config14_d = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000000E;

volatile int *dsp_config14_pd = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000000F;

volatile int *dsp_config21 = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000010;

volatile int *dsp_config21_d = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000011;

volatile int *dsp_config21_pd = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000012;

volatile int *dsp_config23 = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000013;

volatile int *dsp_config23_d = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000014;

volatile int *dsp_config23_pd = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000015;

volatile int *dsp_config30 = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000016;

volatile int *dsp_config30_d = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000017;

volatile int *dsp_config30_pd = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000018;

volatile int *dsp_config32 = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000019;

volatile int *dsp_config32_d = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000001A;

volatile int *dsp_config32_pd = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000001B;

volatile int *dsp_config34 = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000001C;

volatile int *dsp_config34_d = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000001D;

volatile int *dsp_config34_pd = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000001E;

volatile int *dsp_config41 = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000001F;

volatile int *dsp_config41_d = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000020;

volatile int *dsp_config41_pd = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000021;

volatile int *dsp_config43 = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000022;

volatile int *dsp_config43_d = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000023;

volatile int *dsp_config43_pd = XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000024;

volatile int *cb_config00_ws=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000025;

volatile int *cb_config00_en=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000026;

volatile int *cb_config02_ws=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000027;

volatile int *cb_config02_en=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000028;

volatile int *cb_config04_ws=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000029;

volatile int *cb_config04_en=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000002A;

volatile int *cb_config11_ws=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000002B;

volatile int *cb_config11_en=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000002C;

volatile int *cb_config13_ws=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000002D;

volatile int *cb_config13_en=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000002E;

```

```

volatile int *cb_config20_ws=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000002F;
volatile int *cb_config20_en=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000030;
volatile int *cb_config22_ws=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000031;
volatile int *cb_config22_en=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000032;
volatile int *cb_config24_ws=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000033;
volatile int *cb_config24_en=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000034;
volatile int *cb_config31_ws=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000035;
volatile int *cb_config31_en=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000036;
volatile int *cb_config33_ws=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000037;
volatile int *cb_config33_en=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000038;
volatile int *cb_config40_ws=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x00000039;
volatile int *cb_config40_en=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000003A;
volatile int *cb_config42_ws=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000003B;
volatile int *cb_config42_en=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000003C;
volatile int *cb_config44_ws=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000003D;
volatile int *cb_config44_en=XPAR_VF_0_S_AXI_MEM0_BASEADDR + 0x0000003E;

*dsp_config01=0x07b5a000;
*dsp_config01_d=0x00000005;
*dsp_config01_pd=0x00000000;
*dsp_config03=0x07b5a000;
*dsp_config03_d=0x00000007;
*dsp_config03_pd=0x00000000;
*dsp_config10=0x07b52000;
*dsp_config10_d=0x00000007;
*dsp_config10_pd=0x00000000;
*dsp_config12=0x07b52000;
*dsp_config12_d=0x00000005;
*dsp_config12_pd=0x00000000;
*cb_config00_ws=0x00000000;
*cb_config00_en=0x00678000;
*cb_config02_ws=0x00000000;

```

```

*cb_config02_en=0x00678000;

*cb_config11_ws=0x00678000;

*cb_config11_en=0x00000000;

*cb_config13_ws=0x00345000;

*cb_config13_en=0x00000000;

*cb_config20_ws=0x00345000;

*cb_config20_en=0x00000000;


for(i=0;i<NUM_SAMPLES;i++){

    c_base[i]=i+0x01;

}


*ctrl= ( dst_mem | (NUM_SAMPLES<<12) | (latency<<4) | start );

*stat=0x00000000;

while(*stat!=0x1);

for(i=0;i<NUM_SAMPLES;i++){

    value= d_base[i];

}

*ctrl=0x00000000;

for(i=0;i<NUM_SAMPLES;i++){

    c_base[i]=i+0x01;

}

*ctrl= ( dst_mem | (NUM_SAMPLES<<12) | (latency<<4) | start );

*stat=0x00000000;

while(*stat!=0x1);

printk(KERN_INFO "Execution time %d cycles in HW--\n\r", *(int*)count);

for(i=0;i<NUM_SAMPLES;i++){

    value= d_base[i];

}

printk(KERN_INFO "IDLE time %d cycles in HW--\n\r", *(int*)count_idle);

```

```

    int e_base[NUM_SAMPLES];

    for(i=0;i<NUM_SAMPLES;i++){

        e_base[i]=5*((int)c_base[i+3])+7*((int)c_base[i+2])+5*((int)c_base[i+1])+7*((int)
)c_base[i]);

    }

    int Success=0;

    for(i=0;i<NUM_SAMPLES-5;i++){

        if(e_base[i]!=(int)(d_base[i] >>16)){

            Success=0;

        }

        else{

            Success=1;

        }

    }

    if(Success){

        printk(KERN_INFO "Verification Successful.....\n\r");

    }

    else{

        printk(KERN_INFO "Verification failed.....\n\r");

    }

    return SUCCESS;

}

struct file_operations Fops = {

    .read = device_read,

    .write = device_write,

    .unlocked_ioctl = device_ioctl,

    .open = device_open,

    .release = device_release,

};

```

```

int init_module()
{
    int ret_val;

    major_num = register_chrdev(0,DEVICE_NAME, &Fops); //TO Register the Character
Driver

    //Negative Values Signify the Errors in the Code

    if (major_num < 0)
    {

        printk(KERN_ALERT "%s failed with %d\n","Sorry, registering the character
device ", ret_val);

        return ret_val;

    }

    printk(KERN_INFO "%s The major device number is %d.\n", "Registration was
Successful",major_num);

    printk(KERN_INFO "If you want to talk to the Device Driver, \n");

    printk(KERN_INFO "Than create the following device file by the command. \n");

    printk(KERN_INFO "mknod %s c %d 0\n", DEVICE_NAME, major_num);

    mmio = ioremap(MEM_0,0x100);

    mmio1 = ioremap(MEM_3,0x100);

    return 0;
}

void cleanup_module()
{
    int ret;

    //Unregister the Device

    iounmap(mmio);

    unregister_chrdev(major_num,DEVICE_NAME);
}

MODULE_AUTHOR("Zain Ansari");

MODULE_DESCRIPTION("Xilinx Zynq VF Access Driver");

MODULE_LICENSE("GPL v2");

```

### Code Walkthrough IF Driver:

Similar to register access drives the IF access drivers are implemented but they are utilizing the two Block RAMs on the PL one is the source of configuration and data while the other one the sink for the processed data.

```
#define NUM_SAMPLES 32
```

This defines the number of samples or the number of taps to be processed.

```
#define latency 18
```

For different TAP values the useful data comes after a certain latency in a 4 TAP filter the latency is 18 whereas 8 and 12 TAP the latency is 34 and 50 respectively.

```
.read = device_read,  
  
.write = device_write,  
  
.unlocked_ioctl = device_ioctl,  
  
.open = device_open,  
  
.release = device_release, #define
```

This device driver supports the file operations mentioned above, the functions read and write have not been utilized but are implemented for complex operations in future.

```
init_module()
```



When the user *insmod* this kernel module into the kernel the above function is called mapping the physical addresses to the virtual addresses and requesting the user to create a device node for the driver to access it from the user mode.

```
cleanup_module()
```

When the user calls *rmmmod* the above function is called causing the physical to virtual mapping being released.

```
static int device_open(struct inode *inode, struct file *file)
```

The above function has been implemented to achieve a lock or a mutual exclusion on the device driver so only one user mode program can access it at a particular instance.

```
static int device_release(struct inode *inode, struct file *file)
```

The above function has been implemented to release the mutual exclusion lock on the device driver so when a user mode application has finished utilizing the driver it is available for other applications.

```
int device_ioctl(struct file *file, unsigned int ioctl_num, unsigned long ioctl_param)
```

The above function is the main function of the device driver which is called using:

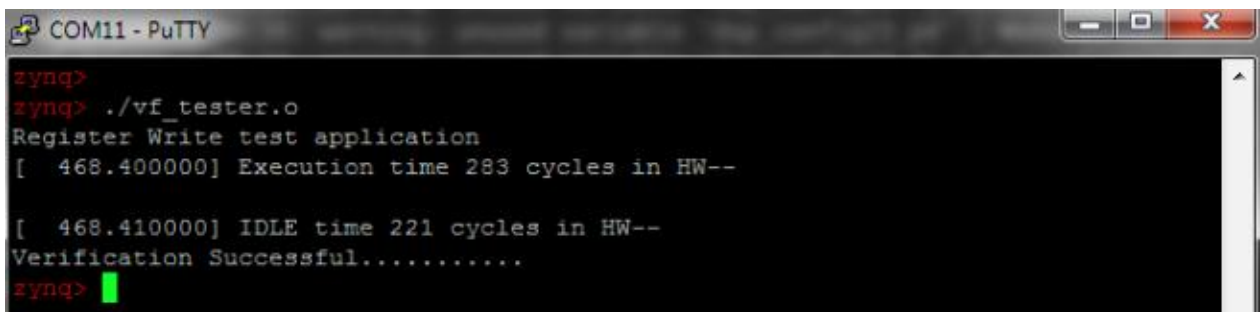
```
ioctl(file_desc, 1, NULL); // Call The IOCTL in Kernel Module
```

This call is made from the application program, it does the following operations.

- Initially it allocates the BRAMs on the virtual addresses to be accessible from the kernel.
- It configures 4 PEs.
- It configures 5 CBs.
- Writes coefficients and data to PEs.
- Computes the execution time in terms of number of cycles.
- Computes IDLE time in terms of number of cycles.
- Verifies the correctness of results.

For Different Value of samples the results have been such with the test application.

### 32 Samples

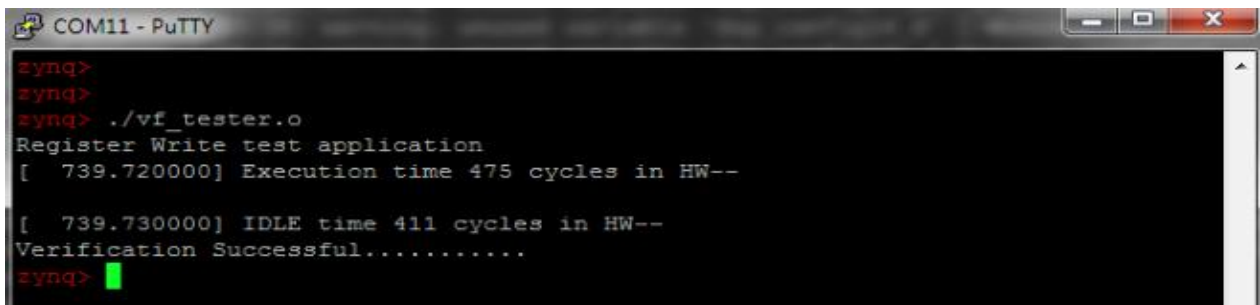


```
COM11 - PuTTY
zynq>
zynq> ./vf_tester.o
Register Write test application
[ 468.400000] Execution time 283 cycles in HW--

[ 468.410000] IDLE time 221 cycles in HW--
Verification Successful.....
zynq>
```

Figure 2.13

### 64 Samples

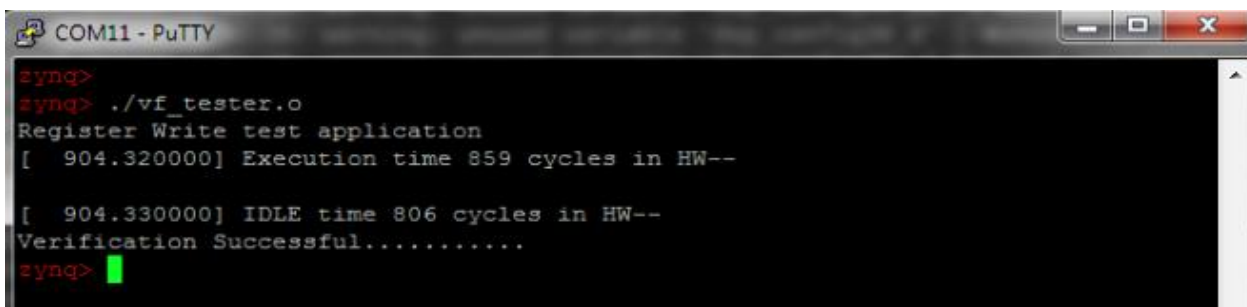


```
COM11 - PuTTY
zynq>
zynq> ./vf_tester.o
Register Write test application
[ 739.720000] Execution time 475 cycles in HW--

[ 739.730000] IDLE time 411 cycles in HW--
Verification Successful.....
zynq>
```

Figure 2.14

## 128 Samples

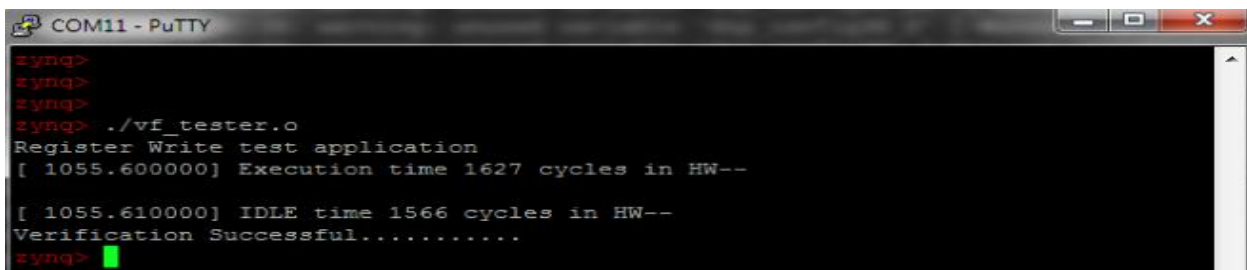


```
COM11 - PuTTY
zynq>
zynq> ./vf_tester.o
Register Write test application
[ 904.320000] Execution time 859 cycles in HW--

[ 904.330000] IDLE time 806 cycles in HW--
Verification Successful.....
zynq> █
```

Figure 2.15

## 256 Samples

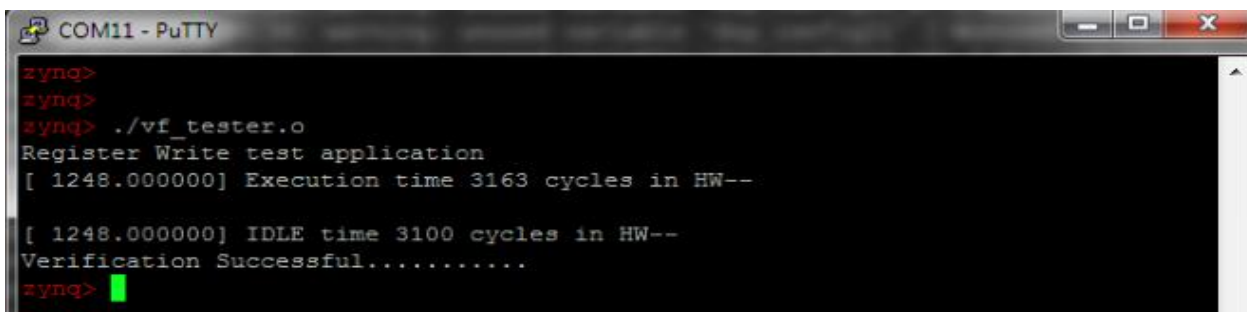


```
COM11 - PuTTY
zynq>
zynq>
zynq>
zynq> ./vf_tester.o
Register Write test application
[ 1055.600000] Execution time 1627 cycles in HW--

[ 1055.610000] IDLE time 1566 cycles in HW--
Verification Successful.....
zynq> █
```

Figure 2.16

## 512 Samples

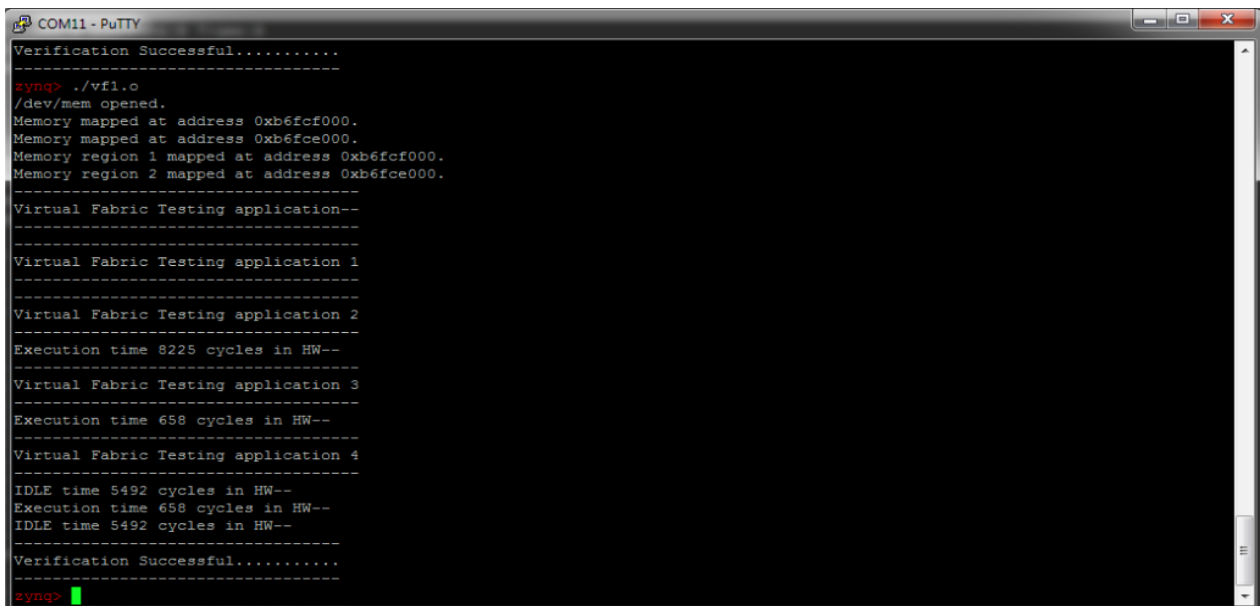


```
COM11 - PuTTY
zynq>
zynq>
zynq> ./vf_tester.o
Register Write test application
[ 1248.000000] Execution time 3163 cycles in HW--

[ 1248.000000] IDLE time 3100 cycles in HW--
Verification Successful.....
zynq> █
```

Figure 2.17

### 32 Samples using User Space Drivers:



```
COM11 - PuTTY
Verification Successful.....
-----
zynq> ./vf1.o
/dev/mem opened.
Memory mapped at address 0xb6fcf000.
Memory mapped at address 0xb6fce000.
Memory region 1 mapped at address 0xb6fcf000.
Memory region 2 mapped at address 0xb6fce000.
-----
Virtual Fabric Testing application--
-----
Virtual Fabric Testing application 1
-----
Virtual Fabric Testing application 2
-----
Execution time 8225 cycles in HW--
-----
Virtual Fabric Testing application 3
-----
Execution time 658 cycles in HW--
-----
Virtual Fabric Testing application 4
-----
IDLE time 5492 cycles in HW--
Execution time 658 cycles in HW--
IDLE time 5492 cycles in HW--
-----
Verification Successful.....
-----
zynq>
```

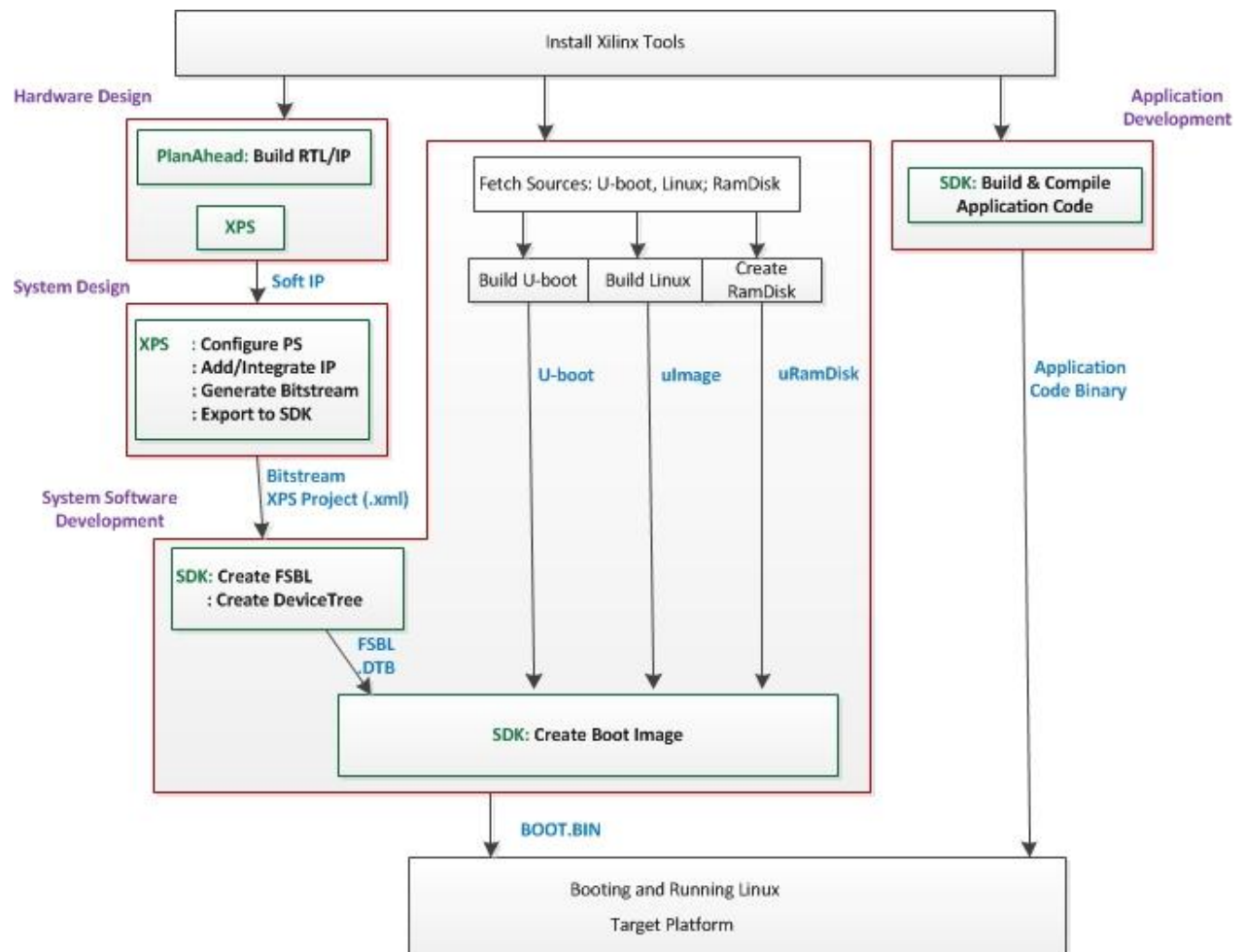
**Figure 2.18**

There are scenarios where large chunks of data need to be transferred between the memory and the PL. In such scenarios DMA is method of preference DMA utilizes interrupts for synchronous and asynchronous transfers. DMA permits the peripherals to transfer data between the system memory and the peripheral memory without the involvement of the processor. The efficiency of DMA is entirely dependent on how the interrupt are dealt with.

The ARM cores built into the Xilinx Zynq has a PL-330 DMA controller built into it. The software driver running on the ARM processor can issue status and control commands to the DMA controller [63].

The Xilinx Zynq Wiki [64] contains the full driver support for the PL-330 controller which can be ported to transfer data to BRAMs or FIFOs on the PL from the system memory. This driver has now been made part of the main line kernel as well and can be run directly to the DMA controller on the Xilinx Zynq.

The following picture sums up the complete Xilinx Zynq development on Linux ecosystem.



**Figure 2.19 [67]**

## Conclusion and Future Work

The points that have been raised in this report create a significant ground for FPGAs to be now seen as a serious computing platform but along with that it also faces massive challenges and some of them are open research problems. This report also focused on a vendor specific platform but since it is a widely available solution it seemed a wise choice, the methodologies and procedures have been described in this report which could enable typical hardware design FPGA engineer to utilize the processor and the operating system to develop some of the most complex embedded systems.

This sort of enabling technology leaves plenty of open research problems for the researchers in High Performance Computing. The author of this report is working on CUDA like library for Hybrid FPGAs using the LLVM compiler infrastructure, utilizing such library selected portions of code could be executed on the FPGA based virtual fabric which could be static or reconfigurable depending on the application or situation it is being utilized in. Apart from this now FPGA based SoC are taking flight and even more high performance chips are entering the market which significantly reduces the challenges faced by the FPGA based computing platforms.

# References

- [1] John von Neumann: First Draft of a Report on the EDVAC; University of Pennsylvania, June 30, 1945.
- [2] G. Fettweis: ICT Energy Consumption - Trends and Challenges; WPMC'08, Lapland Finland, 8 –11 Sep 2008.
- [3] J. Rabaey: Low Power Design Essentials; Springer Verlag, 2009.
- [4] J.Rabaey: Reconfigurable Processing: The Solution to Low-Power Programmable DSP, Proc. ICASSP 1997.
- [5] T. Claasen: High Speed: Not the Only Way to Exploit the Intrinsic Computational Power of Silicon; ISSCC-1999, pp. 22–25, Feb. 1999.
- [6] J.M.P. Cardoso and M. Hübner (eds.), Reconfigurable Computing: From FPGAs to Hardware/Software Codesign, DOI 10.1007/978-1-4614-0061-5\_2, © Springer Science+Business Media, LLC 2011.
- [7] K. Gaj, T. El-Ghazawi: Cryptographic Applications; RSSI Reconfigurable Systems Summer Institute, July 11–13, 2005, Urbana-Champaign, IL, USA  
<http://www.ncsa.uiuc.edu/Conferences/RSSI/presentations.html>
- [8] Garrison Jeff: What! How big did you say that FPGA is? (Team-design for FPGAs) (EETimes Design Article). 27 Sept. 2010
- [9] <http://www.maxeler.com/>
- [10] <http://www.conveycomputer.com/>
- [11] Ciletti MD, “Advanced Digital Design with the Verilog HDL”, Prentice Hall, 2003
- [12] Hauck, S. and DeHon, A. “Reconfigurable Computing: the Theory and Practice of FPGA Based Computation”, Morgan Kaufmann Publishers Inc., 2008.
- [13] R. Hartenstein (keynote): Reconfigurable Computing: boosting Software Education for the Multicore Era; IV Southern Programmable Logic Conference (SPL 2010), Porto Galinhas Beach, Brazil, 24–26 March 2010.
- [14] J. Turley: How Many Times Does CPU Go Into FPGA? Embedded Technology Journal - June 8, 2010.
- [15] Selwood: EPP - A Platform to Bridge a Gap? Emb. Technology J. June 8, 2010
- [16] <http://www.raspberrypi.org/>

- [17] <http://www.altera.com/devices/processor/soc-fpga/overview/proc-soc-fpga.html>
- [18] DS190 – Zynq – 7000 – Overview
- [19] Xilinx Inc. Zynq-7000 All Programmable SoC Technical Reference Manual. Xilinx Inc., 1.4 edition, 2012. Xilinx UG585.
- [20] K. Compton, A. DeHon, "Chapter 11: Operating System Support for Reconfigurable Computing", *Reconfigurable Computing: The Theory and Practice of FPGA-based Computation*, S. Hauck, A. DeHon (editors), Morgan Kaufmann/Elsevier, 2008.
- [21] Silberschatz, Galvin, Gagne: Operating System Concepts, 7th Edition
- [22] A. S. Tanenbaum. Modern Operating Systems, Prentice-Hall, 1992.
- [23] M. Kretz and A. Kugel, Linux on FPGA platforms: control software to connect custom peripherals. ;In Proceedings of SIGBED Review. 2012, 12-16.
- [24] G. Brebner. A virtual hardware operating system for the Xilinx XC6200. International Workshop on Field-Programmable Logic and Applications, 1996.
- [25] K. Rupnow, "Operating system management of reconfigurable hardware computing systems," in Proceedings of the International Conference on Field-Programmable Technology (FPT), 2009, pp. 477–478.
- [26] Microkernel Hypervisor for a Hybrid ARM-FPGA Platform *Khoa D. Pham, Abhishek K. Jain, Jin Cui, Suhaib A. Fahmy and Douglas L. Maskell*
- [27] B. Ylvisaker, B. Van Essen, C. Ebeling. A type architecture for hybrid microparallel computers. IEEE Symposium on Field-Programmable Custom Computing Machines, 2006.
- [28] N. Moore, A. Conti, M. Leeser, L. S. King. Writing portable applications that dynamically bind at run time to reconfigurable hardware. IEEE Symposium on Field-Programmable Custom Computing Machines, 2007.
- [29] Z. Li, K. Compton, S. Hauck. Configuration caching management techniques for reconfigurable computing. IEEE Symposium on FPGAs for Custom Computing Machines, 2000
- [30] R. Maestre, F. J. Kurdahi, M. Fern´andez, R. Hermida, N. Bagherzadeh, H. Singh. A framework for reconfigurable computing: Task scheduling and context management. IEEE Transactions on VLSI 9(6), December 2001.
- [31] Y. Markovskiy, E. Caspi, R. Huang, J. Yeh, M. Chu, J. Wawrzynek, A. DeHon. Analysis of quasi-static scheduling techniques in a virtualized reconfigurable machine. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2002.



- [32] H. Koptez. Real-Time Systems: Design Principles for Distributed Embedded Applications, Kluwer Academic Publishers, 1997.
- [33] C. Steiger, H. Walder, M. Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transactions on Computers* 53(11), 2004.
- [34] H. Walder, M. Platzner. Online scheduling for block-partitioned reconfigurable devices. *Design, Automation and Test in Europe*, 2003.
- [35] E. Lee. The problem with threads. *Computer* 39(5), May 2006.
- [36] S. Singh. Integrating FPGAs in high-performance computing: Programming models for parallel systems—the programmer’s perspective. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2007.
- [37] B. J. Nelson. Remote Procedure Call, Xerox Palo Alto Research Center technical report, 1981.
- [38] M. Snir, W. Gropp. *MPI: The Complete Reference*, 2nd ed., MIT Press, 1998.
- [39] A. Patel, C. A. Madill, M. Saldana, C. Comis, R. Pomes, P. Chow. A scalable FPGA-based multiprocessor. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.
- [40] V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable SoC. *Proceedings of the Reconfigurable Architectures Workshop*, 2003
- [41] M. Budiu, M. Mishra, A. Bharambe, S. C. Goldstein. Peer-to-peer hardware–software interfaces for reconfigurable fabrics. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002.
- [42] M. Butts, A. M. Jones, P. Wasson. A structural object programming model, architecture, chip and tools for reconfigurable computing. *IEEE Symposium on Field Programmable Custom Computing Machines*, 2007.
- [43] A. DeHon, Y. Markovskiy, E. Caspi, M. Chu, R. Huang, S. Perissakis, L. Pozzi, J. Yeh, J. Wawrzynek. Stream computations organized for reconfigurable execution. *Microprocessors and Micro systems* 30, September 2006.
- [44] J. R. Hauser. *Augmenting a Microprocessor with Reconfigurable Hardware*, Ph.D. thesis, University of California, Berkeley, 2000.
- [45] W. Fu, K. Compton. A simulation platform for reconfigurable computing research. *International Conference on Field-Programmable Logic and Applications*, August 2006.
- [46] C. Chang, J. Wawrzynek, R. W. Brodersen. BEE2: A high-end reconfigurable computing system. *IEEE Design and Test of Computers* 22(2), 2005.

- [47] S. Qadeer, D. Wu. KISS: Keep It Simple and Sequential. ACM SIGPLAN Conference on Programming Language Design and Implementation, 2004.
- [48] D. Andrews, D. Niehaus, R. Jidin. Implementing the thread programming model on hybrid FPGA/CPU computational components. Workshop on Embedded Processor Architectures, International Symposium on Computer Architecture, 2004.
- [49] G. Brebner. Multithreading for logic-centric systems. International Conference on Field-Programmable Logic and Applications, 2002.
- [50] J. A. Jacob, P. Chow. Memory interfacing and instruction specification for reconfigurable processors. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 1999.
- [51] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, G. Snider, L. Albertson. Plasma: An FPGA for million gate systems. ACM International Symposium on Field-Programmable Gate Arrays, 1996.
- [52] Xilinx. XC6200 FPGA Advanced Product Specification, June 1996.
- [53] P. Garcia, K. Compton. A reconfigurable hardware interface for a modern computing system. IEEE Symposium on Field-Programmable Custom Computing Machines, 2007.
- [54] Embedded Linux Primer: A Practical Real-World Approach by Christopher Hallinan, Prentice Hall PTR; 2nd edition, 2010.
- [55] Linux Device Drivers 3rd Ed. by Jonathan Corbet, Alessandro Rubini, and Greg Kroah Hartman, Oreilly.
- [56] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, Philippe Gerum: "Building Embedded Linux Systems 2nd edition", Paperback: 462 pages, O'Reilly & Associates; (August 2008); ISBN 10: 0-596-52968-6; ISBN 13: 9780596529680 ISBN 059600222X
- [57] P. Raghavan; Amol Lad, Sriram Neelakandan Embedded Linux System Design and Development, December 21, 2005 by Auerbach Publications.
- [58] Xilinx Inc. Zynq-7000 All Programmable SoC: Concepts, Tools, and Techniques (CTT). Xilinx UG873 V14.5 (March 20, 2013).
- [59] <https://sourcery.mentor.com/sgpp/lite/arm/portal/release1039>
- [60] <http://uclibc.org>
- [61] <http://buildroot.uclibc.org>
- [62] <http://www.denx.de/wiki/U-Boot>

[63] A description of the DMA Controller (DMAC) including the programmers model and instruction set can be found in the DMA-330 Technical Reference Manual, (ARM DDI 0424) available from <http://infocenter.arm.com>.

[64] Zynq Linux pl330 DMA <http://www.wiki.xilinx.com/Zynq+Linux+pl330+DMA>

[65] [http://wiki.qemu.org/Main\\_Page](http://wiki.qemu.org/Main_Page)

[66] <http://www.wiki.xilinx.com/QEMU>

[67] <http://www.wiki.xilinx.com/Getting+Started>