NANYANG TECHNOLOGICAL UNIVERSITY

MEMORY SUB-SYSTEM AND OS SUPPORT FOR COMMUNICATION ABSTRACTION ON XILINX-ZYNQ

by

SIVASANKARAN SARANYA
(G1301605G)

A Dissertation Submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Embedded Systems

Supervised by

Assoc. Prof. Douglas L. Maskell

June 2014

Contents

1 Introduction **2**

1.1 Motivation . . . 2

1.2 Contribution . . . 3

1.3 Organization . . . 3

2 Background **4**

2.1 Hardware Acceleration . . . 4

2.1.1 Task Acceleration . . . 6

2.1.2 Application Acceleration . . . 6

2.2 Embedded Reconfigurable Platforms . . . 7

2.2.1 Advantages . . . 8

2.2.2 Challenges . . . 8

2.2.3 Performance Metrics . . . 8

2.2.4 Example: Xilinx Zynq . . . 9

2.2.4.1 PS and PL . . . 10

2.2.4.2 PS-PL Communication ports . . . 12

2.2.4.3 HW debugging using Chipscope . . . 16

2.3 Overlay Architectures . . . 16

2.4 Memory Sub-system and Communication abstraction . . . 19

2.5 OS support . . . 19

2.5.1 Xillinux . . . 20

2.5.2 Xillybus . . . 20

3 Literature Survey **23**

3.1 Introduction 23

3.2 Prior efforts for the memory-subsystem 23

3.3 Prior efforts for the OS support 26

3.4 Summary 30

4 Memory Sub-system **31**

4.1 Introduction 31

4.2 Xillybus System Architecture 33

4.2.1 Hardware Infrastructure 33

4.2.2 Software Infrastructure 35

4.3 Custom core Architecture 35

4.3.1 Hardware Infrastructure 35

4.3.2 Software Infrastructure 35

4.4 Summary 36

5 Characterization **37**

5.1 GP ports 40

5.1.1 PIO based 41

5.1.2 PS-DMA based 41

5.1.3 Characterization 41

5.2 HP ports 45

5.2.1 Single CDMA IP Core and Memory Peripheral 47

5.2.2 Increasing operating frequency to 150 MHz 48

5.2.3 Increasing channel width to 64 bit 49

5.2.4 Final design for maximum bandwidth 50

5.3 ACP Port 51

5.4 Summary 52

6 Experiments **54**

6.1 Evaluation of Xillybus core 54

6.1.1 Loopback Experiment 55

6.1.2 Hardware accelerator Experiment . . . 56
6.2 Performance estimation of custom core . . . 61
6.2.1 PIO Xillinux experiment . . . 61
6.2.2 PS-DMA Xillinux experiment . . . 62
6.2.3 Conclusion . . . 64

7 Conclusions and Future Work 65
7.1 Conclusions . . . 65
7.2 Future work . . . 66

Bibliography 67

List of Figures

2.1 Typical DSP Algorithm Implemenation 4
2.2 Usage of Hardware Accelerators 5
2.3 Comparison of FPGA and ASIC based Accelerators[1] 6
2.4 Usage of Co-Processor for hardware Acceleration 7
2.5 Zynq Use Cases Diagram 9
2.6 Processing System (PS) Block Diagram[2] 11
2.7 HP port Connectivity Diagram[3] 14
2.8 ACP port Connectivity Diagram[3] 15
2.9 Overlay architecture and Application mapping 18
2.10 Xillybus Block Diagram[4] 21

3.1 GARP Diagram[5] 24
3.2 CoRAM Diagram[6] 25
3.3 PyCoRAM Diagram[6] 26
3.4 SIRC Diagram[7] 28
3.5 RIFFA 1 Diagram[8] 29
3.6 RIFFA 2 Diagram[9] 29

4.1 Proposed Design 32
4.2 Integration of Xillybus core 34
4.3 Xillybus infrastructure for Zedboard 34

5.1 AXI4-Lite IPIF Block Diagram. 38
5.2 AXI4 IPIF Block Diagram. 38

5.3 AXI4-Lite based Register Peripheral. 39
5.4 AXI4 based Memory Peripheral. 40
5.5 PS DMA Connectivity Diagram. 42
5.6 PIO Transactions. 42
5.7 latency in ns 43
5.8 Time in us for data transactions 44
5.9 Throughput comparison between PIO and PS-DMA method 45
5.10 PL DMA Connectivity Diagram. 46
5.11 PL-DMA method 47
5.12 PL-DMA method at 150 MHz 48
5.13 PL-DMA method at 64-bit 49
5.14 PL-DMA method at 64-bit and 150 MHz 50
5.15 Throughput for CDMA transactions through ACP 52

6.1 Connectivity diagram for Loopback experiment 55
6.2 Throughput for Xillybus FIR transactions 56
6.3 Connectivity diagram for Accelerator experiment 57
6.4 Throughput for Xillybus FIR transactions 59
6.5 Throughput for Xillybus FIR transactions 60

List of Tables

5.1 Experiment results for PS-DMA based transactions 44
5.2 Resource usage of memory peripheral 44
5.3 Experiment results for CDMA based transactions 47
5.4 Experiment results for CDMA based transactions at 150 MHz 48
5.5 Experiment results for CDMA based transactions for 64-bit data channel 49
5.6 Experiment results for CDMA with 64-bit data channel at 150 MHz . 50
5.7 Experiment results for CDMA based transactions through ACP . . . 51

6.1 Experiment results for Xillybus loopback transactions 56
6.2 Experiment results for Xillybus with accelerator for small sample size 59
6.3 Experiment results for Xillybus with accelerator for large sample size 60
6.4 PS to PL Communication latency for Xillinux using PIO 62
6.5 Experiment results for Xillinux using PIO 62

Abbreviations

ACP Accelerator Coherency Port

ADC Analog-to-Digital Converter

API Application Processing Interface

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

BRAM Block random access memory

CDMA Central Direct Memory Access

DDR Double data rate

DSP Digital signal processor

DTB Device Tree Binary

DTS Device Tree Source

EDK Embedded Development Kit

FIFO First In First Out

FPGA Field Programmable gate array

GP General purpose

GPP General Purpose Processor

HDL Hardware description language

HP High Performance

ISE Integrated Software Environment

JTAG Joint Test Action Group

LUT	lookup table
MIO	Multiplexed Input Output
OCM	On-Chip Memory
OS	Operating system
PE	Processing Element
PIO	Programmable Input Output
PL	Programmable Logic
PS	Processing system
QOS	Quality of service
SCU	Snoop Control unit
SoC	System on Chip
XPS	Xilinx Platform Studio

Abstract

In the quest for hardware acceleration of applications, embedded reconfigurable platforms have emerged with significant potential for addressing the demand for performance at low power consumption. Research efforts have shown strength of these platforms in a wide range of application domains where compute intensive tasks gets executed efficiently on reconfigurable fabric. Despite these advantages, these platforms have not yet been ready for mainstream computing due to the lack of abstractions. Overlay architectures, such as intermediate fabrics, have been evolved as a computation abstraction to provide application portability and scalability. Integration of overlay architecture with a memory subsystem and with an embedded processor is crucial to enable management and sharing of limited overlay resources. This report presents a memory subsystem around streaming overlay architectures and OS support for the communication abstraction in an embedded reconfigurable platform, the Xilinx Zynq. We provide support of available hard IP blocks such as embedded ARM Cortex-A9 processor cores and AXI interfaces for streaming overlay architectures. We first present experiments to quantify the communication overhead and to characterize the AXI interfaces using bare metal SW applications. For AXI general purpose (GP) interfaces, we observe a maximum throughput of 640 Mb/s and 200 Mb/s per interface, with and without using PS-DMA, respectively. For AXI high performance (HP) interfaces and ACP interfaces, we observe a maximum throughput of 8.4 Gb/s per interface while theoretical maximum throughput is 9.6 Gb/s. After that, in order to study the overheads associated with OS abstraction, we conducted experiments using Xillybus system and observed high throughput (up to 80 MS/s) for large data streaming (above 4K samples) but low throughput (2-650 KS/s) for small data streaming (below 4K samples) through overlay because of high latency associated with the system. We then integrated our own developed custom core and estimated a sustained high throughput (3 MS/s) which can be further improved by using device drivers for DMA controllers for small data streaming (below 4K samples).

Acknowledgment

I would like to express my deepest gratitude to my supervisor Assoc Prof Dr Douglas Leslie Maskell for guiding me at every stage of this project. I would have not been able to complete my project successfully without his support and encouragement.

I wish to thank Abhishek Kumar Jain for his professional guidance and continuous support. I am deeply grateful to him for the technical discussions that helped me sort out the details of my work. I am also thankful to him for carefully reading and commenting on countless revisions of this report.

Thanks to Mr. Jeremiah Chua in Centre for High Performance Embedded Systems (CHiPES) for his technical support and the facilities.

Chapter 1

Introduction

1.1 Motivation

Hardware acceleration is an emerging domain in embedded systems which has attained prominence due to the emergence of embedded reconfigurable platforms. These platforms have added advantages of low power consumption, isolated execution of tasks and scalability. However, lack of suitable abstractions prevents the use of these platforms commercially. Poor design productivity has been a key limiting factor, restricting their effective use to experts in hardware design [10]. Abstractions can be employed at various levels, from computation, interfaces, to management and programming. Overlay architectures, such as intermediate fabrics[11], have evolved as a computation abstraction to provide application portability and scalability. Integration of overlay architecture with a memory subsystem and an embedded processor is crucial to enable management and sharing of limited overlay resources. Since memory sub-system and communication interfaces affect the performance of the system heavily, these components need to be studied and designed very carefully. In our work, we aim to study communication overheads and keep them as small as possible while employing abstractions. Objective is to develop and maximize performance of abstract memory subsystem around overlay architecture, supported by OS abstraction.

1.2 Contribution

This report focuses on building a memory subsystem around an overlay architecture and providing OS support for the communication abstraction in an embedded reconfigurable platform, the Xilinx Zynq[3]. Zedboard[12] have been used for all the experiments. It is a complete development kit for designers interested in exploring Zynq based system designs. The board contains all the necessary interfaces and supporting functions to enable a wide range of applications. The different methods to interface the processor, main memory and a streaming overlay architecture are explored in order to identify the method with maximum bandwidth and minimum communication latency. We first present experiments to quantify the communication overhead and to characterize the AXI interfaces of Zynq platform using bare metal SW applications. After that, in order to study the overheads associated with OS abstraction, we conducted experiments using Xillybus system and observed high throughput for large data streaming but low throughput for small data streaming through overlay because of high latency associated with the system. We then inserted our own developed custom core to support high throughput and low latency for small data streaming through overlay.

1.3 Organization

The remainder of the report is organized as follows: Chapter 2 presents background information. Chapter 3 studies current state of the art techniques in memory subsystems and their OS support. In chapter 4, we present and describe Xillybus system architecture and our custom core and its features. In chapter 5, we present experiments conducted to characterize ZynQ communication ports using bare metal SW applications. Chapter 6, presents the experiments conducted using Xillybus system and performance estimation of our custom core under the control of Linux. We conclude in chapter 7 and discuss future work.

Chapter 2

Background

2.1 Hardware Acceleration

The method of using specialized and scalable computational hardware to execute some compute intensive functions faster than a General Purpose Processor (GPP) is generally known as hardware acceleration. Hardware acceleration is one of the methods of improving performance of sequential GPPs by placing one or more specialized functional units alongside a GPP. These dedicated functional units, also known as co-processors, can provide dramatic acceleration[13]. Examples include video decoding acceleration , functionality implemented as an embedded video decoder in a smart phone.

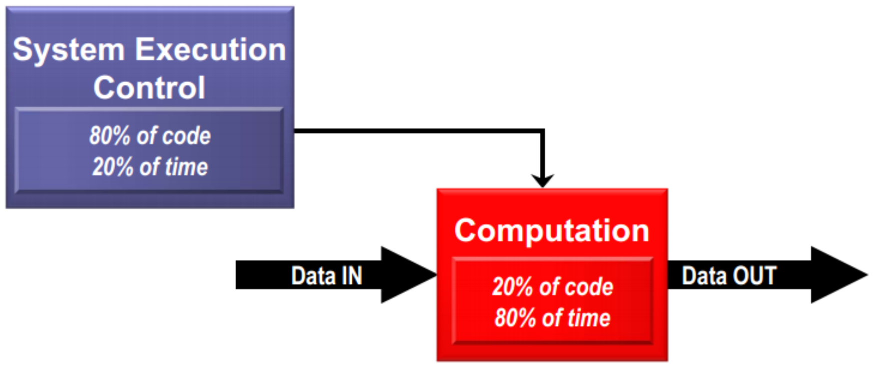


Figure 2.1: Typical DSP Algorithm Implemenation

In a typical Digital signal processor (DSP) application, 20% of the program code consumes 80% of the application execution time. Fig. 2.1 illustrates this concept. This requires time-consuming, difficult-to-maintain assembly coding to increase system performance. Hence usage of specialized hardware accelerators is becoming significantly important in accelerating signal processing applications. These accelerators are normally deployed as an Application Specific Integrated Circuit (ASIC) block alongside a GPP such as ARM processor or a digital signal processor. Fig.2.2 illustrates this technique. This limits the flexibility and increases time to market since developing an ASIC is still a complex and time consuming process. On the other hand, FPGAs are becoming popular for rapid-prototyping of hardware accelerators at the cost of some overheads. Fig.2.3 shows the differences between FPGA and ASIC based accelerator design.

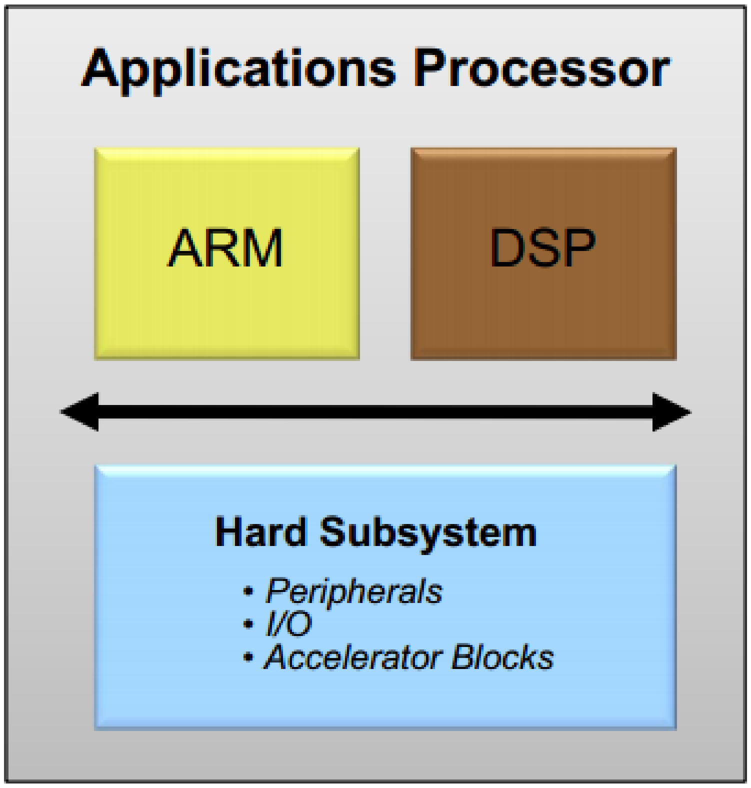
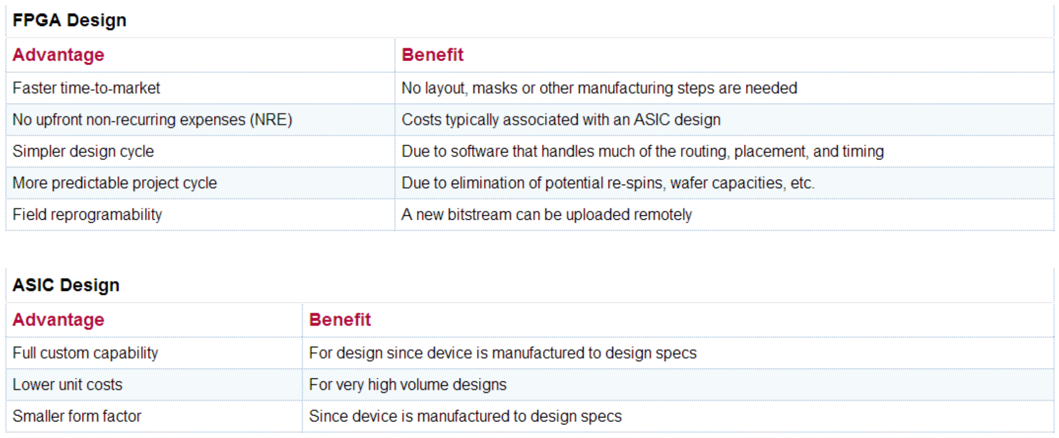


Figure 2.2: Usage of Hardware Accelerators



FPGA Design

Advantage	Benefit
Faster time-to-market	No layout, masks or other manufacturing steps are needed
No upfront non-recurring expenses (NRE)	Costs typically associated with an ASIC design
Simpler design cycle	Due to software that handles much of the routing, placement, and timing
More predictable project cycle	Due to elimination of potential re-spins, wafer capacities, etc.
Field reprogramability	A new bitstream can be uploaded remotely

ASIC Design

Advantage	Benefit
Full custom capability	For design since device is manufactured to design specs
Lower unit costs	For very high volume designs
Smaller form factor	Since device is manufactured to design specs

Figure 2.3: Comparison of FPGA and ASIC based Accelerators[1]

2.1.1 Task Acceleration

Task acceleration normally refers to the development of a specialized functional unit (co-processor) to accelerate certain task (compute-intensive) of an application as shown in Fig.2.4. For example, a typical software defined radio application requires execution of several DSP tasks such as FIR filtering, FFT etc. Hardware description languages, such as Verilog and VHDL, are normally used in order to implement a co-processor on Field Programmable gate array (FPGA) for such tasks. It requires significant design effort and hardware expertise. As high level synthesis tools are normally used to synthesize C/C++ code [*high level language (HLL) description*] to generate hardware description of tasks, these tools are becoming significantly important[14]. It has been shown in literature that these accelerators can provide several orders of performance improvement depending on the implicit parallelism in the task.

2.1.2 Application Acceleration

Despite having the implementation of a hardware accelerator (co-processor) and its performance gain, there is no guarantee that it will surely provide reduction in application execution time[15]. This depends heavily on how the accelerator is connected to the overall system, communication mechanism between accelerator and the system,

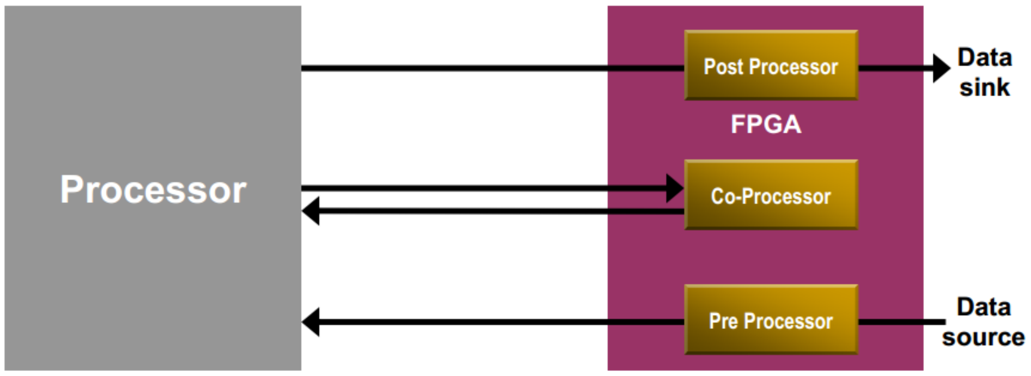


Figure 2.4: Usage of Co-Processor for hardware Acceleration

communication bandwidth and latencies etc. It is also not straight forward to plug the accelerator into the system. FPGA vendors nowadays provide functionality in their tools to automatically connect an accelerator to the system via standard interfaces and interconnects such as AXI, PLB etc. In order to improve communication bandwidth between system and the accelerator, vendors started embedding hard processor cores and necessary infrastructure in the latest generation of platforms. In the next section, we describe these platforms in detail.

2.2 Embedded Reconfigurable Platforms

Embedded reconfigurable platforms couple one or more processors with a shared fabric of reconfigurable hardware, such as a field programmable gate array (FPGA). Some examples are: Programmable System on Chip (PSoC) from Cypress, SmartFusion SoC FPGAs from Microsemi, Zynq-7000 All Programmable SoC from Xilinx etc. Evolution of these platforms has dramatically changed the process of designing custom programmable system on chips. Some of the key advantages over other available platforms includes re-programmability, lower power consumption than multicore processors and GPUs, real-time execution capability , and most importantly, high spatial parallelism which can be used to significantly accelerate complex algorithms [16][17].

2.2.1 Advantages

Presence of both processor and reconfigurable fabric in a platform seems beneficial due to several reasons. This approach is usually processor centric for easy control and adaptability. Streaming data-flow and control intensive tasks can be executed on such a platform efficiently. These platforms can take the advantage of thread level parallelism. Reconfigurable fabric provides application specific acceleration and power of reconfigurability. Also reconfigurable designs are more energy and power efficient, smaller and more flexible when compared to SW implementation, which makes it more suitable for embedded systems. These platforms also result in ease of design and faster time-to-market. Solutions like ASIC have high performance benefits but falls back in flexibility metric. On the other hand, solutions like embedded reconfigurable platforms are highly flexible.

2.2.2 Challenges

Following are the challenges in the implementation of application on these platforms:

- Efficient communication interface between the reconfigurable fabric, processor and memory and its management.
- Reconfiguration management and minimization of reconfiguration time.
- Operating system support for providing certain services such as scheduling, synchronization and communication.
- Programming model and tools for mapping applications onto heterogeneous resources of the platform.

2.2.3 Performance Metrics

Following are the desired performance metrics:

- Short communication latency and high bandwidth to allow fast data communication.
- Short reconfiguration time to allow fast task switching.
- Less overheads associated with automated synthesis of the hardware.

2.2.4 Example: Xilinx Zynq

Both major FPGA vendors, Xilinx and Altera, have recently introduced reconfigurable platforms consisting of high performance processors coupled with programmable logic. These platforms partition the hardware into a processor system (PS), containing one or more processors along with peripherals, bus and memory interfaces, and other infrastructure, and the programmable logic (PL) where custom hardware can be implemented. The two parts are coupled together with high throughput interconnect to maximize communication bandwidth. We focus on the Xilinx-Zynq[3].

The Xilinx-Zynq consists of a dual-core ARM Cortex A9 processor platform equipped with a double-precision floating point unit (FPU), commonly used peripherals and reconfigurable fabric. The programmable logic (PL) is connected to the Processing System (PS) through high-speed, low-latency AXI interfaces. Presence of ARM processor enables development of both Bare-metal applications and Linux based applications. It exhibits improved performance compared to a two chip solution, which is limited by IO bandwidth and loose coupling. Zynq follows a processor centric approach by allowing the processors to boot first.

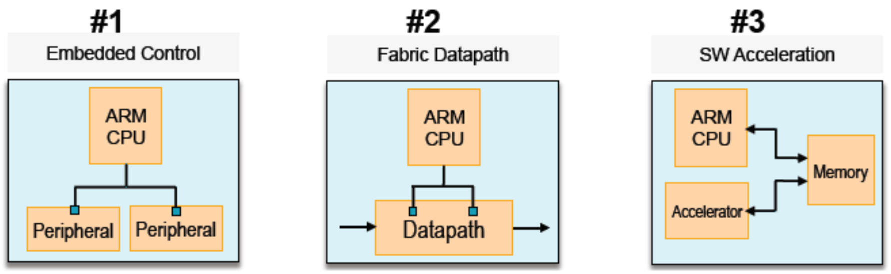


Figure 2.5: Zynq Use Cases Diagram

Fig. 2.5 shows some use cases for Zynq:

- Embedded control Used to access and control peripheral configuration registers
- Fabric Data path - Used to access data path configuration registers and data path memory.
- Software acceleration - Used to move data between Hardware and software domains.

2.2.4.1 PS and PL

The Processing system (PS) contains a dual-core ARM Cortex-A9 processor which can boot independently unlike other cores such as PowerPC. The PS also consists of a double-precision floating point unit, commonly used peripherals, a dedicated hard DMA controller (PL330), On-Chip Memory (OCM) and external memory interfaces. Each processor is a low-power and high-performance core that contains 32 KB Level-1 separate caches for instruction and data. L1 cache is 32 kB and 4-way set associative present for each core whereas L2 cache is 512 kB and 8-way set-associative which is shared. The components of PS are:

- Two ARM Cortex-A9 cores which can be configured as a single processor, asymmetric or symmetric multiprocessor.
- The Snoop Control Unit (SCU) and the Accelerator Coherency Port(ACP) for cache coherence
- Dual ported 256 KB On-Chip Memory (OCM) with parity
- A set of IO peripherals
- Controller for Double data rate (DDR) memory and interrupt.
- DMA and Timers

The components in the PS are connected though AXI high performance datapath switches as shown in Fig. 2.6. They are of two types: OCM interconnect and Central interconnect. The features of PS are:

- Supports upto 1 GHz operation
- It supports vector processing through NEON extensions
- Out of order execution is facilitated by eight stage instruction pipeline.
- Multi issue processor support up to 4 instructions
- It can deliver 2.5 DMIPs/MHz
- Little Endian processor.
- Speculative execution enabled by register renaming.
- Support for single/double precision floating point operations

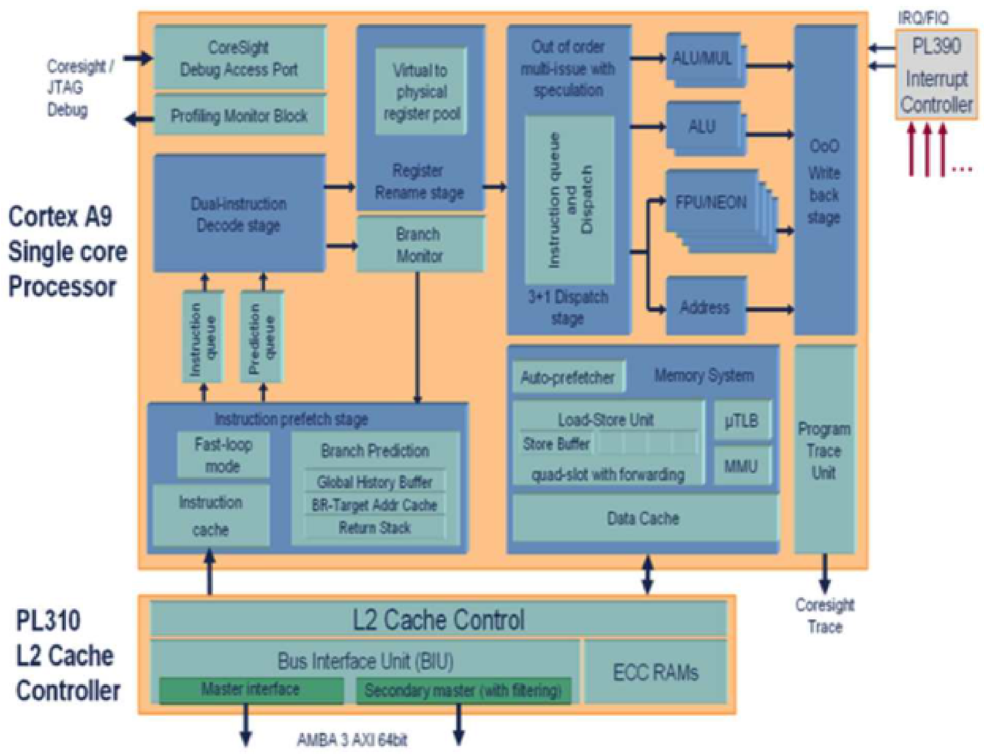


Figure 2.6: Processing System (PS) Block Diagram[2]

The Programmable Logic (PL) consists of Artix 7 FPGA fabric. The Artix family of FPGAs is focused on power and cost optimization. It has 6 input LUTs and 36kb Block RAMs which can be configured as two 18 kb blocks. The advantages of PL are increased system performance and reduced power consumption. The predictable latency feature of PL is useful for real time applications. Power management can be achieved by powering down the PL as it is on different power domain than the PS. The components of PL are:

- 13,300 Logic slices
- 53,200 Six input LUTs
- 140 Dual port 36kb Block RAM
- 220 DSP slices
- Dual Analog-to-Digital Converter (ADC) blocks with 12-bit and 1 MSPS rate
- 4 Select I/O banks

The PL can be configured during boot process or at a later time. Also complete and partial dynamic reconfiguration are supported. There are four stages in booting of FPGA - start-up *(occurs after power is stable)*, initialize *(SRAM cells in the PL are cleared)*, configure *(programming using bitstream)* and enable *(enabling PS control)*. The features of PL are-

- Memory capability within the lookup table (LUT)
- High speed and low jitter clocks
- Synchronous memory with programmable data width

2.2.4.2 PS-PL Communication ports

The Xilinx-Zynq contains several AXI based interfaces between PS and PL. The interfaces are of two types: functional and configuration interfaces.

Functional interfaces:

- AXI interconnect
- Extended Multiplexed Input Output (MIO) interfaces
- Interrupts
- DMA flow control
- Clocks
- Debug interfaces

Configuration interface

- PCAP
- Status of Configuration
- SEU
- Signaling Program, Done and Init

The interconnection between PS and PL provided by the Advanced eXtensible Interface (AXI) bus facilitates any logic implemented in PL to be addressable by PS thereby acting as a memory-mapped peripheral. AXI [18] is an asynchronous interface with independent read and write channels. This interface provides low latency as well as processor DMA access to the peripherals. Each interface consists of multiple AXI channels, enabling high throughput data transfer between the PS and the PL. AXI has four different unidirectional channels for enhanced parallelism: write, read, address and write response channels. This multichannel implementation takes advantage of the fact that in burst transfer address information required is less than data. The AXI interfaces to the fabric include:

- AXI_GP - Two master and two slave interfaces
- AXI_HP - Four slave interfaces with direct access to DDR and OCM
- AXI_ACP - One slave interface for coherent memory access

General purpose (GP) port

Four 32-bit interfaces, two master and two slave interfaces. The ID width of master port is 12 and for the slave port it is 6. As there are no FIFOs present, GP exhibits constrained performance. Most of the PS peripherals and memory can be accessed by the two slave GP ports.

High Performance (HP) port

Four 32/64 bit slave ports whose read and write command issuing capabilities are separate. 1 KB data FIFOs are present to smoothen out long latency read and write transfers. The caches are software managed. The PL is connected to memory using 3 output First In First Out (FIFO) controllers: two for DDR and one for OCM. PL ports have Quality of service (QOS) signaling and write access is provided with programmable release threshold. There is a PS-PL asynchronous domain crossing feature for clock signals. 32 bit aligned transfers have dynamic upsizing to 64 bits and unaligned transfers have automatic expansion. Upsizing is more bandwidth efficient when compared to expansion. HP port provides many functions to establish priority and manage queues. These functions are flexible and available to both PL and PS in the form of signals and registers respectively. QOS related functions can be used to assign priority. For continuous data read by the logic in the reconfigurable fabric, the read FIFOs must be filled fully with the required data before first pop; the FIFO level information is useful in this case. If multi threaded read operations are issued, the logic should be capable of handling out of order reads. Diagram of HP connectivity is shown in Fig. 2.7.

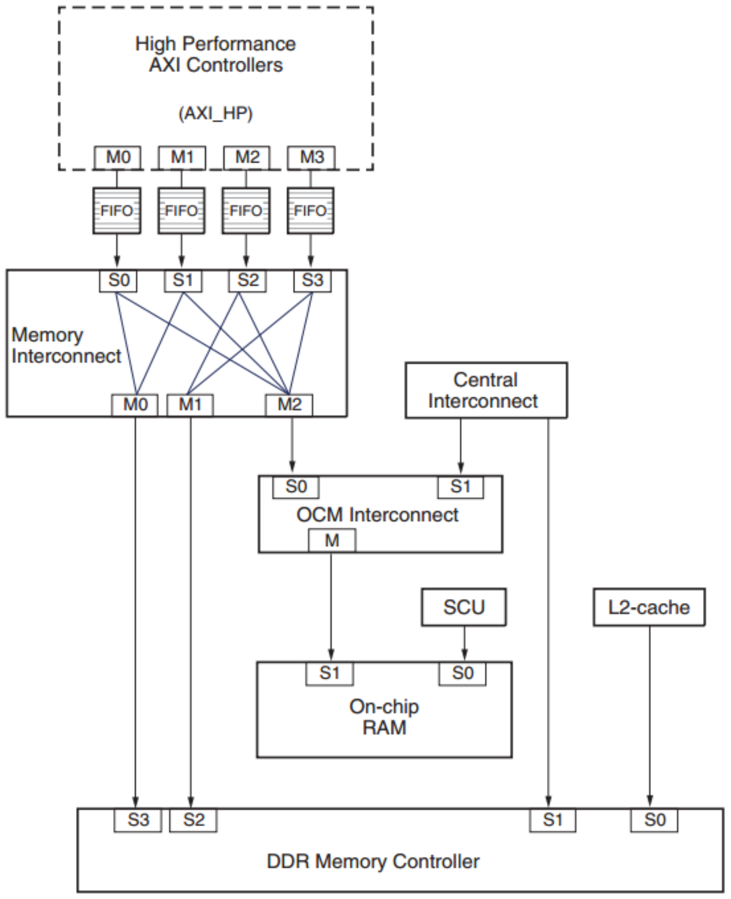


Figure 2.7: HP port Connectivity Diagram[3]

Accelerator Coherency Port (ACP) port

One 64 bit slave port which aids in asynchronous cache coherent transactions between PS and PL. One can access L1 or L2 cache from the PL and this can be used to share data between PS and PL with low latency. The ACP port must be utilized carefully as improper use may pollute the cache thereby hindering other applications. The AXI HP port directly accesses memory (bypassing the cache) whereas the AXI ACP works with the cache for better writing speeds. It is connected to the Snoop Control unit (SCU),which monitors cache traffic and maintains coherency between the L1 caches. It also performs arbitration between the two cores for L2 access and also manages ACP access. Hence processor is freed from cache flush and invalidate operations. Also there is reduction in external memory flushes due to no wasted cache flushing and CPU cache read for shared data. There are two modes of ACP access: coherent and non coherent. In coherent read, ACP interacts with the SCU to check for L1 hit/miss, L2 hit/miss and then main memory. In coherent write, coherency is enforced before writing to the main memory. In non-coherent mode, read/write operation directly interacts with the main memory. Diagram of ACP connectivity is shown in Fig. 2.8.

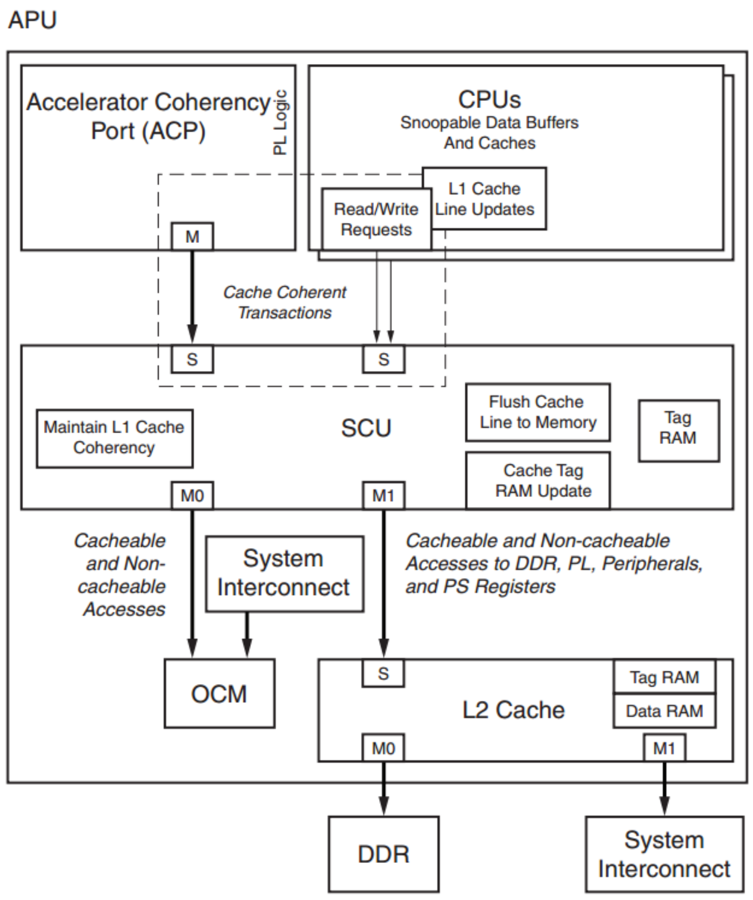

APU
Accelerator Coherency Port (ACP)
PL Logic
CPUs
Snoopable Data Buffers And Caches
L1 Cache Line Updates
Read/Write Requests
M
Cache Coherent Transactions
S
S
Maintain L1 Cache Coherency
SCU
Flush Cache Line to Memory
Tag RAM
Cache Tag RAM Update
M0
M1
Cacheable and Non-cacheable Accesses
System Interconnect
Cacheable and Non-cacheable Accesses to DDR, PL, Peripherals, and PS Registers
OCM
S
L2 Cache
Tag RAM
Data RAM
M0
M1
DDR
System Interconnect

Figure 2.8: ACP port Connectivity Diagram[3]

2.2.4.3 HW debugging using Chipscope

Chipscope pro tool performs in-circuit verification by simulating the hardware and probing the results[19]. Independence in testing is achieved by placing Chipscope cores in the FPGA and connecting them to the required nodes. It acts like a logic analyzer by inserting buffers to monitor the signals selected. Verification of design's functionality in an iterative fashion is accomplished by breaking down the problem into parts. BRAM is used for storing data and trigger while logic is used for comparison.

There are two types of Chipscope cores- capture and control. AXI monitor is used as a capture core which triggers and stores the data. Internal nodes and signals can be accessed. ICON control core is used to interface capture cores to the Joint Test Action Group (JTAG) chain. It is used for timing analysis and complex debugging of AXI based systems.

Functions: The Chipscope Pro tool can be utilized for three purposes-

- Verification: to determine compliance to specification
- Debugging: finding and correcting discrepancy
- Data capture: collection of board data for simulation

Advantages: As it is placed inside the hardware, it is more accurate than software simulation. Also this die level functional verification eliminates delay which features when debug is done using oscilloscopes and logic analyzers. It reduces the time required for debug and verification and removes external debug solution's variation. Chipscope is an part of Xilinx FPGA design flow. Using it there is reduction of 25 % in overall design time and 50% in on-chip verification and debug time .

Limitations: The number of samples captured is limited in this method and also addition of Chipscope core involves additional place and route

2.3 Overlay Architectures

In order to support wide spread usage of embedded reconfigurable platforms, overlay architectures, such as intermediate fabrics, have been evolved as a computation abstraction. These architectures provide application portability and scalability. An Overlay is an array of reconfigurable PEs, implemented on top of a Commercial Off-The-Shelf (COTS) reconfigurable fabric. The PEs are interconnected using programmable interconnect (PI) and the functions of the Processing Element (PE) and the PI are controlled by configuration data.

Several researchers have recently proposed these overlay architectures, including ZIPPY[20], QUKU[21], VDR[22], ZUMA[23], CARBON[24] and IF[25]. These solutions are getting popular in research community because FPGAs are still dealing with the following issues:

- Programmability and compilation time: Application description for FPGAs is usually done on a lower abstraction level (in order to get most efficient implementation). Although the use of high level synthesis tools tried to raise the level of programming abstraction, the lengthy low level compilation times are much higher which results in slower design cycles.
- Reconfiguration latency: HW Task switching in FPGAs is too slow (order of milliseconds) because of large configuration data associated with the fine grained mapping of applications on FPGA. Implementing dynamic reconfiguration on FPGAs is technically complex and require low-level details.

Furthermore, high level application developers often lack low level hardware design experience, therefore undermining the usefulness of FPGA as accelerators. Thus, there is a growing need to make FPGAs easier and more accessible to application developers, who are accustomed to higher-level software abstractions and fast development cycles. The key attraction of Overlays is their near software like programmability, compilation and engineering efficiency. Overlays can easily be a part of a programmable System on Chip (SoC) where there is a need of fast compilation and fast reconfiguration capabilities. System designs can incorporate these Overlays at platform level to make use of reconfigurable hardware as application accelerators. The architecture of Overlays can be reconfigured at runtime to allow customizable PEs as per the application needs using PR capability of FPGAs. As both Altera and Xilinx started embedding coarse grained hard DSP blocks within fine grained reconfigurable fabric, researchers also started to explore these blocks as dynamically reconfigurable coarse grained processing elements.

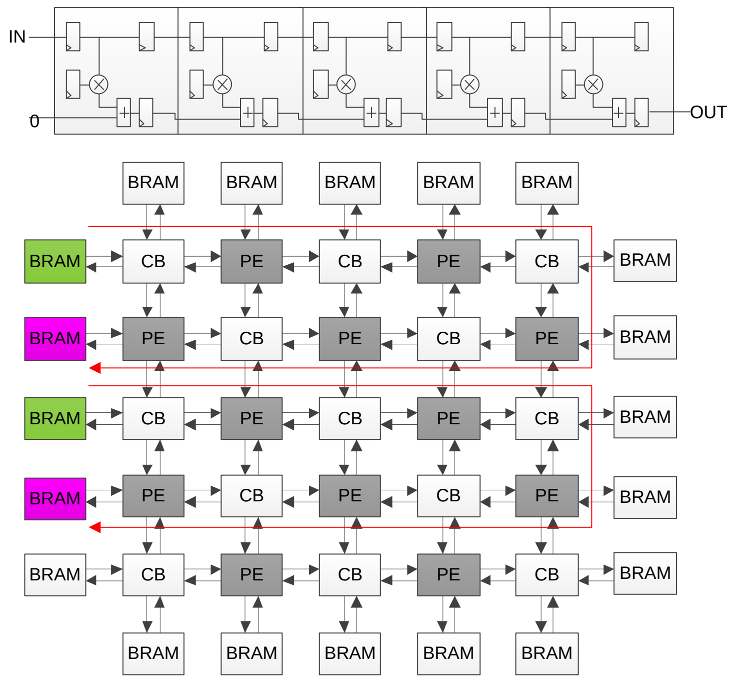


Figure 2.9: Overlay architecture and Application mapping

Integration of overlay architecture with a memory subsystem and with an embedded processor is crucial to enable sharing of limited overlay resources. This report presents a memory subsystem around streaming overlay architectures and operating system (OS) support for the communication abstraction in an embedded reconfigurable platform, the Xilinx Zynq. Fig. 2.9 shows a simple streaming overlay architecture and the mapping of an application (5-tap FIR filter) on it[26].

2.4 Memory Sub-system and Communication abstraction

Communication abstraction is a method to represent communication interfaces and memory sub-system at a logical view by providing an interface similar to SW Application Processing Interface (API). This logical view basically decouples the functionality of communication interfaces from their actual implementation. Communication abstraction might be required by system developers to ensure standards-compliance, handle the multitude of communication protocols, and reduce developer effort.

The key attraction of communication abstraction techniques is their capability to seamlessly access memory sub-system and abstraction of physical interfaces. An abstracted memory sub-system should be an integral part of programmable SoC where there is a need for easy and isolated memory transactions[27]. System designs can incorporate these abstractions at platform level to make use of reconfigurable memory as local memory space for application accelerators. As both Altera and Xilinx started embedding hard memory blocks (Block random access memory (BRAM)) within fine grained reconfigurable fabric, researchers also started to explore these blocks as distributed storage elements.

In this report we present the proposed memory subsystem by exploring how BRAMs can be used as local and distributed memory space to provide seamless access of data to highly efficient overlay architectures. We also explore the ways to provide OS support for the memory sub-system and communication abstraction. Different techniques have been explored in literature to develop such an abstracted memory sub-system, including Garp[5], CoRAM[6] and PyCoRAM[28]. These are explained in detail in Chapter 3

2.5 OS support

OS management abstraction for reconfigurable hardware helps in coordinating multiple HW tasks and managing shared reconfigurable resources among tasks[29]. Presence of OS such as Linux provides certain services like scheduling, communication and synchronization at the cost of some performance overhead compared to bare metal based applications. Bare metal SW applications provide better performance compared to Linux based applications but require additional effort in handling scheduling, communication and synchronization. Achieving maximum performance in the presence of OS management abstraction is a challenge. SIRC[7], RIFFA[8][9] and Xillybus[4] are some examples of OS support for reconfigurable hardware. These are explained in detail in Chapter 3. In the next section, we give brief description of Xillinux and Xillybus.

2.5.1 Xillinux

Xillinux[30] is a software + FPGA code kit based on Ubuntu 12.04 LTS OS for Zynq. It differs from other Linux distributions in the fact that it contains some hardware logic. The PL can be configured and managed by the software. Also PL events can be observed and synchronized. Xillybus solution is used for PS-PL interaction through FIFO queues. Attaching monitor, keyboard and mouse to the Zedboard with Xillinux enables it to act as a graphical desktop with SD card as hard disk. For Bare-metal like interface, Xillybus-Lite can be used. FPGA code kit in Xillinux consists of Xillybus pipes which simplifies IO operations to simple file read/write.

Setting up Xillinux is simple and requires no knowledge of Linux kernel, drivers and FPGA. Setting it up on the Zedboard requires two components: Zedboard boot partition kit and SD card image of the boot up file system. As Xillinux is a development platform with ready to use custom logic environment, its initial boot up is lengthy. For booting, the following components are required: FAT32 file system boot partition consisting of boot loaders, PL configuration bit-stream and Linux kernel binaries; and Linux mount ext4 root file system. The image downloaded is all encompassing except for files from the bundle and bit file.

2.5.2 Xillybus

Xillybus is a DMA based data transfer mechanism between the ARM and FPGA[4]. The presence of DMA buffer is transparent to both the processor and reconfigurable logic. The interface interacts using FIFOs and file I/O operations. Xillybus provides abstraction to FPGA logic in the form of FIFOs. Fig. 2.10 illustrates this concept. The Reconfigurable fabric is connected using customizable FIFOs with empty and full signals to facilitate easy data transfer. The logic needs to read/write from/to the FIFOs.The presence of data on the FIFO alerts the IP core to map the same to processor user space. The Xillybus IP core provides platform dependent clock for the design. It accepts only one common clock for both read and write operations.

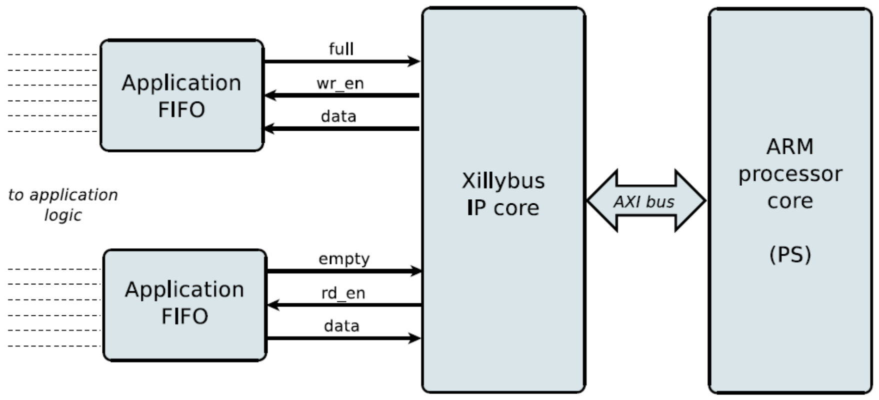


Figure 2.10: Xillybus Block Diagram[4]

The FPGA demo bundle for Zynq is used. The Demo bundle contains Integrated Software Environment (ISE) and Xilinx Platform Studio (XPS) project, boot.bin and device tree files. Initially the net list is generated using XPS project file. The FIFO IP cores provided are regenerated in ISE to obtain all the Hardware description language (HDL) files. The bit stream is generated for top file to be downloaded to the FPGA. Demo bundle contains a default set of device nodes, the number of nodes can be increased or customized as per design requirements. There are two type of nodes available: stream and memory mapped. The default design consists of FIFOs connected in a loop back manner for testing. The application logic can be inserted in between to meet the requirements. Device nodes can be customized and added as per the design. The default design consists of one GP port to transfer configuration information, HP port for VGA and ACP port for Xillybus IP. The Xillybus implementation consists of host program which runs on the processor, synthesized function which runs on the reconfigurable fabric and wrapper function which acts as an interface between processor and reconfigurable fabric. The synthesized function isn't called directly but data is transferred using API calls. In order to exploit the parallelism present in hardware, one thread sends data from the processor as soon as it is available and another thread collects the data back.

Chapter 3

Literature Survey

3.1 Introduction

In this chapter, we first give a detailed description of various techniques of developing memory sub-system proposed in literature for solving the problem of communication abstraction. Finally, we present prior efforts in providing Operating system (OS) support for communication abstraction.

3.2 Prior efforts for the memory-subsystem

As we have already mentioned in section 2.2 that modern embedded reconfigurable platforms have a large number of embedded memories to avoid communication bottlenecks. This created opportunities to exploit high speed interfaces and raises questions of how to organize, manage, and exploit these embedded reconfigurable memories[31]. To address possible bottleneck problems, particularly in providing high bandwidth transfers between the CPU and the reconfigurable fabric, it has been proposed to more tightly integrate the processor and the reconfigurable fabric. A number of tightly coupled architectures have resulted [5, 32], including vendor specific systems with integrated hard processors, as presented here:

GARP

In GARP the reconfigurable unit is attached to the MIPS processor in the form of a co-processor.[5]. Garp's reconfigurable array can transfer data to and from main memory via memory buses. It has five 32-bit buses , one for address and the other four for data. Array can directly access the data contained in cache via memory queues and can perform random read and writes. Three Memory queues have been used to support memory mapped to stream transactions between memory and reconfigurable data path. Array can access data from all three queues in parallel at every clock cycle and hence achieves a high bandwidth and low latency data access path. Some key features are run time reconfigurability, reduced configuration size and reconfiguration time. Instead of storing results in array, results return to memory because of no local memory support in array datapath. The architecture of GARP is shown in Figure 3.1.

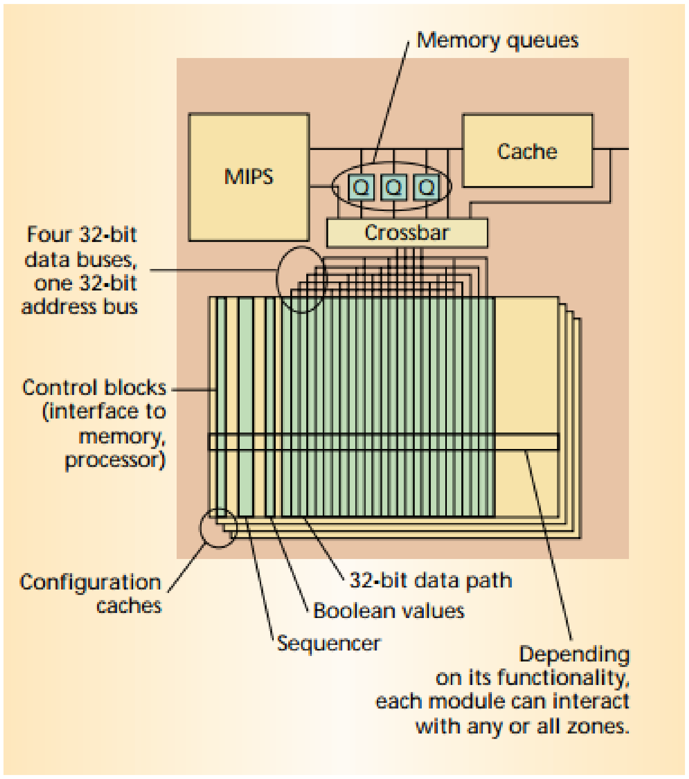

Memory queues
MIPS
Cache
Q
Q
Q
Crossbar
Four 32-bit data buses, one 32-bit address bus
Control blocks (interface to memory, processor)
Configuration caches
32-bit data path
Boolean values
Sequencer
Depending on its functionality, each module can interact with any or all zones.

Figure 3.1: GARP Diagram[5]

CoRAM

CoRAM was proposed as a data-transport mechanism using a shared and scalable memory architecture[6]. It provides interface to off-chip memory using on-chip interconnect generated by CONNECT NOC generator[33]. Communication between off-chip memory and on-chip memory is abstracted and can be controlled using software threads. CoRAM is a collection of specialized distributed SRAMs that are used to store application data which in turn paves way for application interaction with the external environment. Whenever data is ready, a control thread *(state machine)* is used to inform the user logic. Access to the data in specific CoRAMs can be accomplished through SRAM interfaces which are locally addressed. Control thread can be expressed in C *(high-level language)* which can be translated to a state machine. This itself can provide high level of management abstraction or else a multi-threaded processing core can be used for compiling and execution. In short control thread can be soft or hard. Some key features are application independent memory interface to the hardware accelerators, abstracted communication between accelerators and external memory. Currently CoRAM does not provide any OS support such as Linux. The architecture of CoRAM is shown in Figure 3.2.

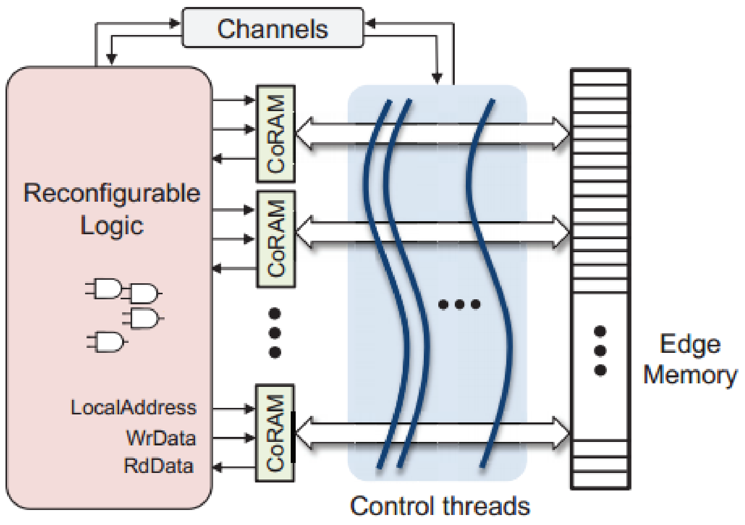


Figure 3.2: CoRAM Diagram[6]

PyCoRAM
PyCoRAM [28] is a python based automation infrastructure of CoRAM project with slight modifications. For example, instead of making use of CONNECT NOC generator, it uses AMBA AXI-4 infrastructure and provides support for AXI based IPs. It replaces the C with Python scripting language to represent a soft control thread. Currently, It does not provide system support for controlling hardware accelerators. The architecture of PyCoRAM is shown in Fig. 3.3.

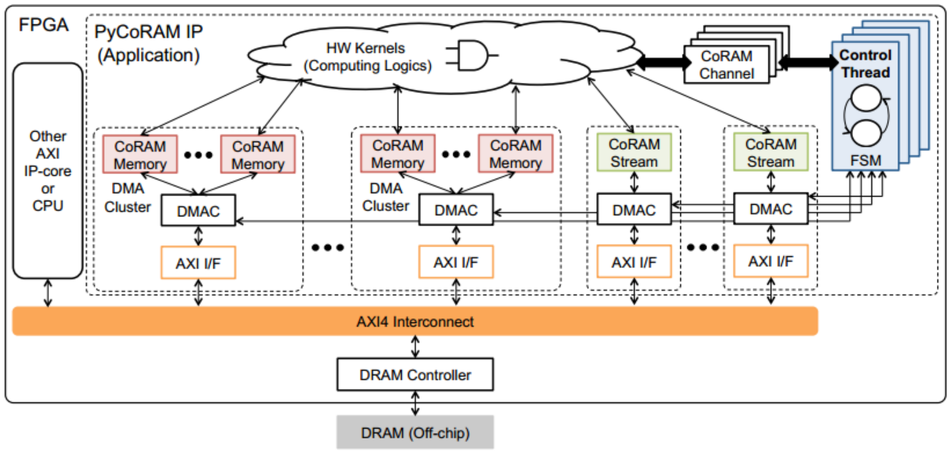


Figure 3.3: PyCoRAM Diagram[6]

3.3 Prior efforts for the OS support

SIRC[7] and RIFFA[8] are two main open source frameworks to abstract communication interfaces by providing OS support. Xillybus[4] is Xillinux based commercial framework which is designed to abstract the interfaces: the PCIe interface (in a typical x86 based system) and the AXI interface (in an ARM based system), as the underlying transport mechanism by providing Standard FIFOs as interfaces to the application logic.

Basically OS support provides task, memory and I/O management in a system. OS management abstraction for reconfigurable computing helps in coordinating multiple HW tasks, managing shared reconfigurable resources among tasks and providing methods for communication and synchronization among HW and SW tasks. Scheduling is one of the most important functionality encapsulated in an OS, because which resource has to be used when by a task has to be decided by the OS. Communication between tasks is another important abstraction provided by the OS. Shared memory model is the common communication style used in multi-core systems but in reconfigurable computing systems several models have been proposed for effective communication such as Message passing interfaces (MPI) and Remote procedure calls (RPC). Streams are based on graph structures for task communication and they can be used to transmit data and control information which is very suitable for reconfigurable computing systems. Virtual memory is another important abstraction provided by the OS. When reconfigurable hardware is tightly coupled with a processor with memory management unit (MMU) support, reconfigurable hardware can share processor's MMU. The processor can now be used by the OS to perform memory accesses and then it can feed data to reconfigurable HW for computation. 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 transactions. DMA controllers are normally used in such cases to counter this issue of handling memory transactions. Synchronization using threads is an important functionality encapsulated in the OS. Reconfigurable HW 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. The simplest method is thread style synchronization on hardware tasks.

SIRC

SIRC [7] was proposed as an open source OS based abstract interface (a software API and hardware interface) for communication between PC and FPGA. It consists of a software side C++ interface which communicates and synchronizes with hardware side HDL interface. The API provides isolation of implementation from user code. It works on basic principle of HW-SW communication which works as follows:

First SW sends data to local buffer in FPGA and triggers the user logic to get this data from local buffer and put it in another local buffer after processing. After finishing its processing user logic will notify the SW to get the processed data from local buffers. SIRC uses Ethernet to communicate between a Windows based workstation and Xilinx FPGA board. User does not need the knowledge of communication protocol, OS or proprietary drivers. The architecture of SIRC is shown in Figure 3.4.

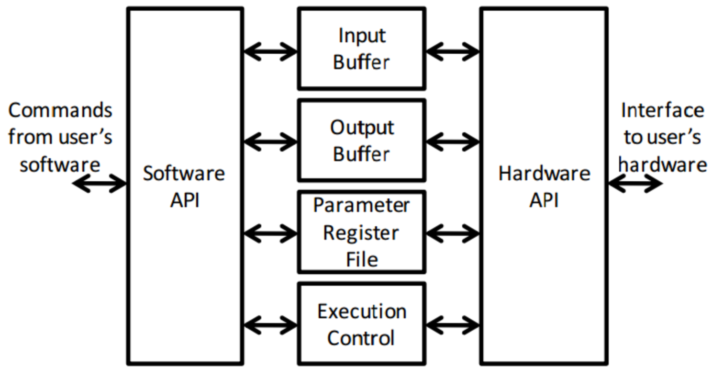


Figure 3.4: SIRC Diagram[7]

RIFFA

RIFFA 1 [8] was proposed as an open source OS based reusable framework to integrate the FPGA IP cores to workstation software using PCIe interface. The requirements of this framework include a workstation and FPGA board. PCIe bus must be enabled in the former and latter should have a PCIe peripheral. The FPGA is unaware of timing and protocol details. It uses interrupt based DMA method of transfer. On the SW side, it provides a device driver in Linux for PCIe and SW libraries; and on the FPGA side, PCIe endpoint obtains PLB requests from requests coming from the PC via address translation. PC to FPGA communication bandwidth is very low (25 MB/s) due to the use of a PLB to PCIe bridge. RIFFA 2 [9] was proposed to address bandwidth bottleneck but support for DDR access to FPGA was not provided. Currently, RIFFA only supports PCIe interface hence it is not possible to use it on Zynq for AXI based PS-PL communication. The architecture of RIFFA 1 is shown in Fig. 3.5 and the architecture of RIFFA 2 is shown in Fig. 3.5.

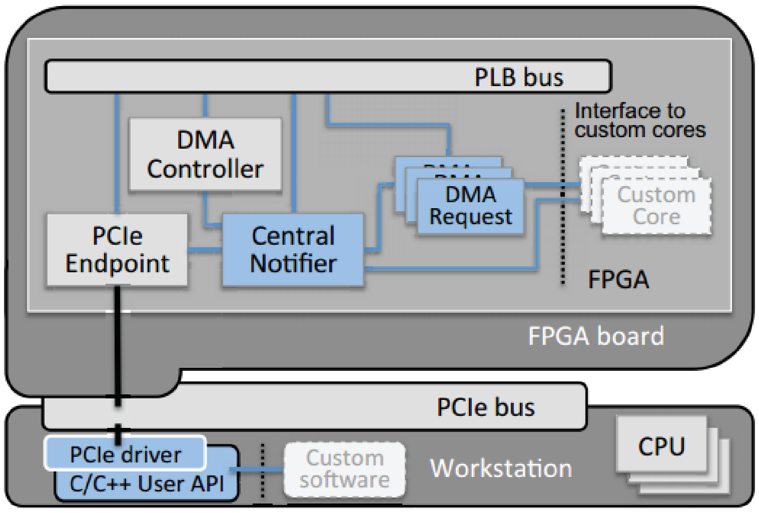


Figure 3.5: RIFFA 1 Diagram[8]

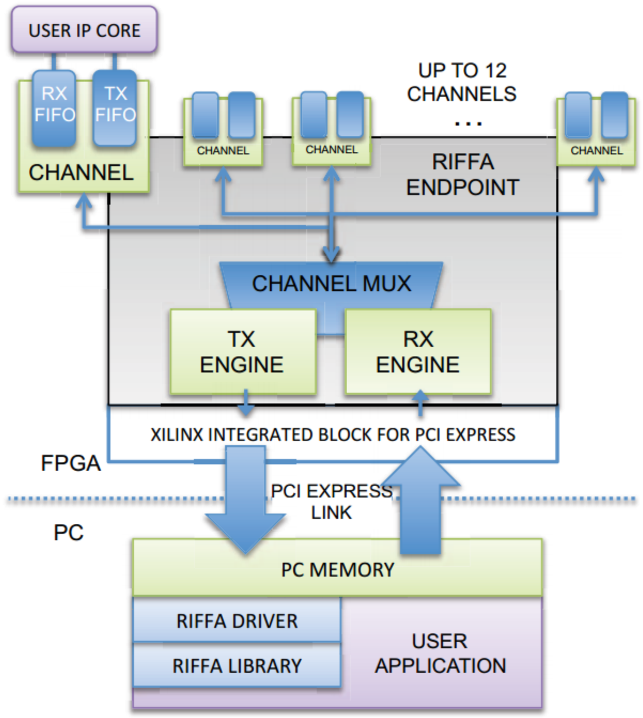


Figure 3.6: RIFFA 2 Diagram[9]

Xillybus
Xillybus [4] is a portable, DMA-based data transfer mechanism between the FPGA and Linux OS running on ARM processor. It is designed to work with two interfaces: the PCIe interface (in a typical x86 based system) and the AXI interface (in an ARM based system), as the underlying transport mechanism. Standard FIFOs are provided as interfaces to the application logic on the FPGA. Each FIFO stream is mapped to a device file by a universal Xillybus driver. The drawback in this method is that communication latency is very high.

3.4 Summary

This section discusses prior efforts in the field of communication abstraction of memory sub-system and its OS support with emphasis on the important features of each project. Literature related to the categories of memory sub-system and its OS support for communication abstraction have been reviewed in order to present an overall view of techniques that have been implemented. GARP, CoRAM and PyCoRAM have been analyzed under the memory subsystem category. It is clear from the analysis that though GARP and CoRAM, both provide the required MM-stream interfaces, absence of OS support and use of custom bus interface makes them unsuitable to use as an abstracted system. Lack of OS support and hardware accelerator control in PyCoRAM undermines its use too. SIRC, RIFFA and Xillybus have been studied under the OS support category. SIRC and RIFFA use bus interfaces which are incompatible to Zynq platform hence can not be used to support Zynq based memory sub-system and overlay architectures. Xillybus technique is the one best suited not only for the platform but also for the memory sub-system management requirements of this project. Currently available solutions related to OS support for reconfigurable hardware do not support run time reconfiguration (RTR)[34]. We integrated support for RTR in our proposed design by using the concept of dynamically reconfigurable overlay architectures.

Chapter 4

Memory Sub-system

4.1 Introduction

In this chapter, we give the detailed description of our memory sub-system. The idea is to have seamless data transactions among overlay architectures, DDR memory and system components. We designed a memory sub-system as shown in Fig. 4.1. The design consists of two methods to stream data through overlay.

- Method-1: For large amount of data streaming (more than 1M samples). Xillybus core is used to stream data from external memory to overlay and vice versa.
- Method-2: For small amount of data streaming (less than 1M samples). Our custom core is used to stream data from local memory to overlay and vice versa. Data movement between external and local memory has been abstracted.

Our custom core consists of a slave interface, which is used to communicate data between local and external memory (using a memory-mapped memory peripheral). In order to evaluate the efficiency of the proposed subsystem, we first evaluated the efficiency of PS-PL communication ports using bare metal SW applications. We present experiments to quantify the communication overhead and to estimate the performance of our custom core. Our custom core consists of register and memory peripherals and their controllers. Detailed description and results of the experiments is given in the next chapter.

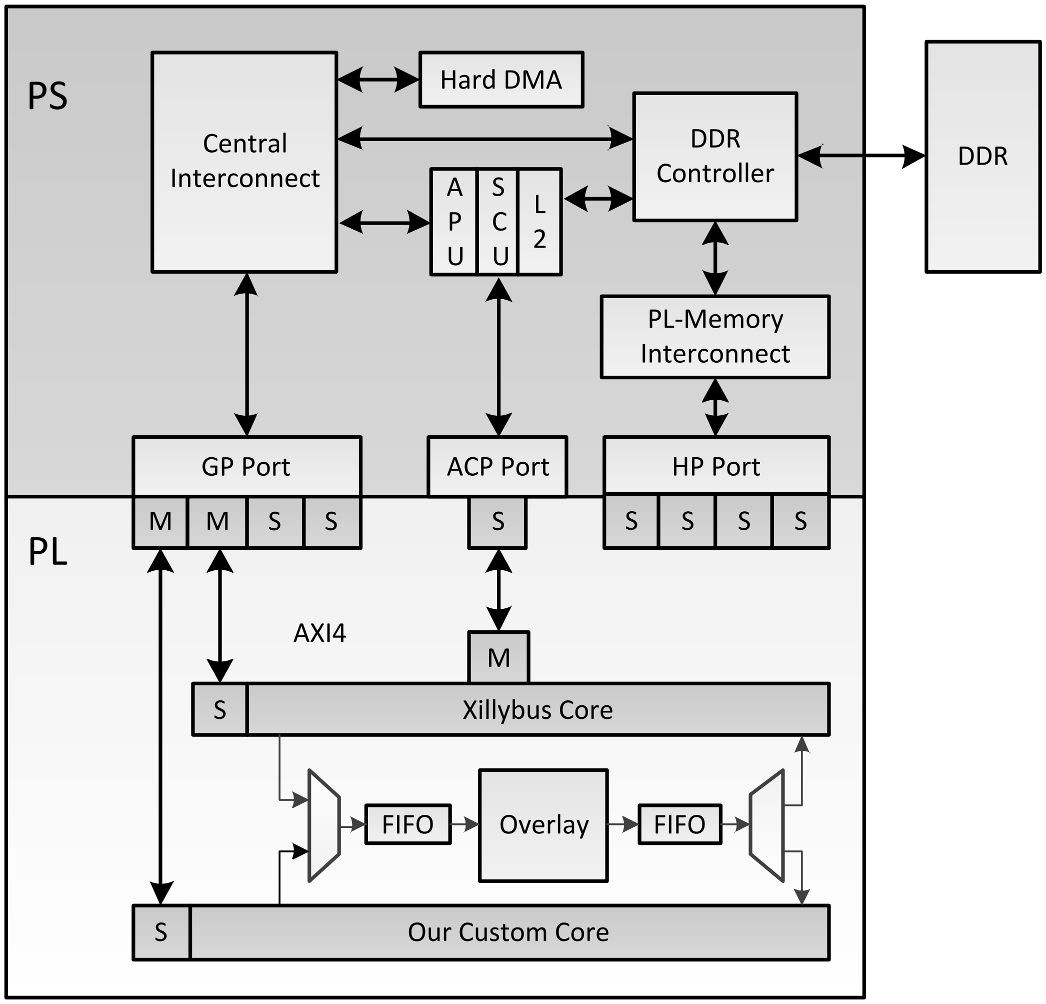


Figure 4.1: Proposed Design

After that, in order to study the overheads associated with OS abstraction, we conducted experiments using Xillybus system and observed that it works quite well for large amount of data streaming (more that 1M samples). However for small amount of data streaming, its performance is poor. We then inserted our custom core into the system and estimated the efficiency of the same. Detailed description and results of the experiments is given in the Chapter 6.

In the next sections, we first give a detailed description of Xillybus system architecture. After that we present the architecture of our custom core.

4.2 Xillybus System Architecture

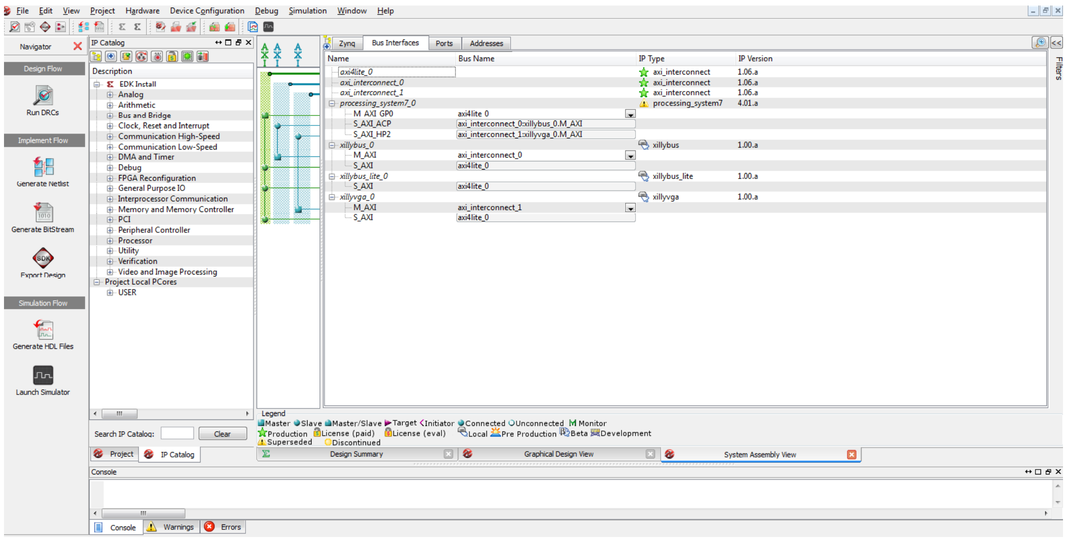
Xillybus provides both SW and HW infrastructure to be used under control of Linux. The Xillybus IP core is connected to the ACP port to provide increased data transfer speed. The ACP port is used in coherent mode to achieve cache coherency. Any PL logic connected to the Xillybus communicates with the PS by using FIFOs. Support for display is provided by the means of a VGA core attached to the HP port. Both the cores are controlled by the processor via GP port.

4.2.1 Hardware Infrastructure

The initial Demo bundle contains FIFOs connected in a loopback fashion thereby causing the Xillybus core to act both as a source and sink. The logic is inserted in this loopback path and its round trip time is measured from the processor to determine its throughput. High level wrapper file present in the Xillybus bundle is used to integrate our custom logic. The logic thus added communicates with the processor using FIFOs. These FIFOs get initialized as dev nodes in the Linux kernel thereby facilitating easy access to them in the form of file IO operations. Read/Write operations are done by using low level driver code which are present in the form of APIs ensuring abstraction. Fig. 4.2 shows the integration of xillybus core in the system.

Xillybus core provides data stream to input FIFO which connects to a hardware accelerator as shown in Fig. 4.3. Accelerator provides the processed data to output FIFO which connects again to xillybus core. The following steps are involved in addition of custom logic (hardware accelerator) to the Xillybus:

The FPGA demo bundle for Zynq is used,which contains ISE and XPS project, boot.bin and device tree files. Initially the net list is generated using XPS project file. The FIFO IP cores provided are regenerated in ISE to obtain all the HDL files. The custom logic can be described in a high level language like C/C++ which can be synthesized using Vivado HLS tool to obtain its HDL description. This logic is instantiated as a component within the Xillybus provided wrapper file, xillydemo.v. The logic is inserted in the path by breaking the loopback connection between the



Name	Bus Name	IP Type	IP Version
axi4lite_0		axi_interconnect	1.06.a
axi_interconnect_0		axi_interconnect	1.06.a
axi_interconnect_1		axi_interconnect	1.06.a
processing_system7_0		processing_system7	4.01.a
M_AXI_GP0	axi4lite_0		
S_AXI_ACP	axi_interconnect_0:xillybus_0.M_AXI		
S_AXI_HP2	axi_interconnect_1:xillyvga_0.M_AXI		
xillybus_0		xillybus	1.00.a
M_AXI	axi_interconnect_0		
S_AXI	axi4lite_0		
xillybus_lite_0		xillybus_lite	1.00.a
S_AXI	axi4lite_0		
xillyvga_0		xillyvga	1.00.a
M_AXI	axi_interconnect_1		
S_AXI	axi4lite_0		

Legend
Master Slave Master/Slave Target Initiator Connected Unconnected Monitor
Production License (paid) License (eval) Local Pre Production Beta Development
Superseded Discontinued

Figure 4.2: Integration of Xillybus core

FIFOs.

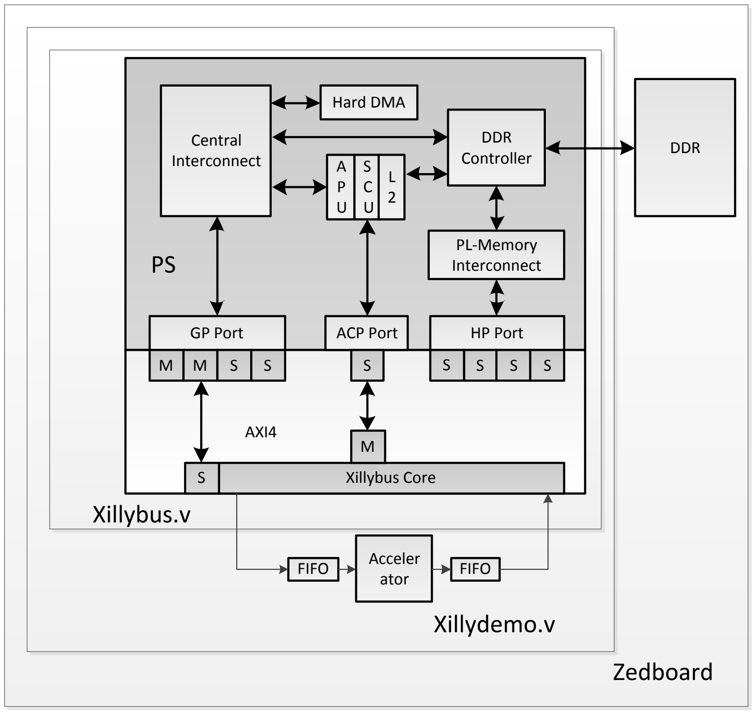


Figure 4.3: Xillybus infrastructure for Zedboard

4.2.2 Software Infrastructure

A part of the application resides in the PL in the form of a hardware accelerator. Xillybus provides SW infrastructure for users to make use of accelerator in an abstracted manner. The ISE tool is used to generate the bit file of the design which needs to be ported onto the SD card for FPGA configuration. Once configured, the application can communicate with the logic by reading and writing to the dev nodes using APIs. Write and Read functions are provided to communicate via xillybus core.

4.3 Custom core Architecture

We developed a custom core which also provides both SW and HW infrastructure to be used under control of Linux. It can be used when small amount of data streaming (less than 1M samples) is needed through the accelerator.

4.3.1 Hardware Infrastructure

Currently our custom core uses GP port to communicate with the processing system. Fig.4.1 shows the integration of our custom core in the system. Our custom core consists of local memories and their controllers. Our custom core first accepts data from external memory and then streams the data to the input FIFO which connects to a hardware accelerator. We implemented a stream mux which can select the input stream either from our custom core or from xillybus core. Similarly we also implemented a stream demux which can select the destination of the output stream (either our custom core or xillybus core).

4.3.2 Software Infrastructure

A part of the application resides in the PL in the form of a hardware accelerator. Our custom core provides SW infrastructure for users to make use of accelerator in an abstracted manner. Write and Read functions are provided to communicate via our custom core.

4.4 Summary

In this chapter, we presented xillybus system architecture and our custom core which we integrated into xillybus system. In the next chapter, we present characterization of communication interfaces and efficiency of our memory sub-system.

Chapter 5

Characterization

In this chapter, we present characterization of PS-PL communication interfaces. The Xilinx-Zynq contains several AXI based interfaces to the programmable logic (PL). Each interface consists of multiple AXI channels, enabling high throughput data transfer between the PS and the PL, thereby eliminating common performance bottlenecks for control, data, I/O, and memory. The AXI interfaces to the fabric include:

- AXI_GP – two 32-bit master and two 32-bit AXI general purpose (GP) slave interfaces
- AXI_HP – four 64-bit/32-bit configurable, buffered AXI high performance (HP) slave interfaces with direct access to DDR and on chip memory
- AXI_ACP – One 64-bit AXI accelerator coherency port (ACP) slave interface for coherent access to CPU memory

We have used Embedded Development Kit (EDK) for the system design. XPS has been used to automatically generate custom AXI based peripherals. A peripheral connects to AXI interconnect through corresponding AXI IP interface (IPIF) modules, which provides a quick way to implement interface between AXI interconnect and the user logic. A peripheral can have either a slave interface or a master interface. Slave interface typically required by most peripherals for operations like logic control, status report etc. Block diagrams of AXI4-Lite and AXI IP interfaces are shown in and Fig. 5.1 and Fig. 5.2 respectively.

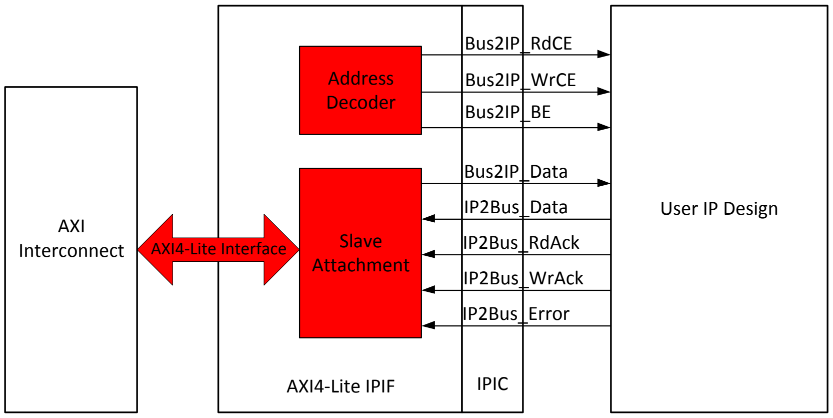


Figure 5.1: AXI4-Lite IPIF Block Diagram.

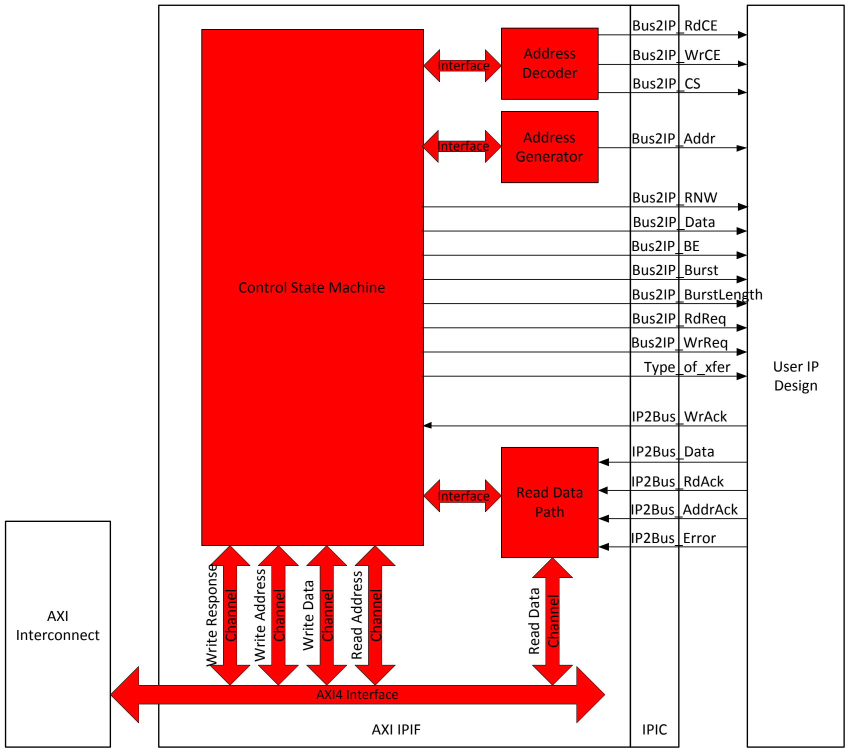


Figure 5.2: AXI4 IPIF Block Diagram.

Master interface is typically required by complex peripherals like DMA controller for commanding data transfers between regions. XPS also provides a BFM simulation platform so that user can verify the functionality of the generated peripheral. We generated following custom peripherals in the PL using XPS base system builder (BSB) in order to communicate with the PS:

- AXI-lite based Register peripheral – user specific SW accessible registers (upto 32) interface for operations like logic control, status report etc. Fig. 5.3 shows one example of the register peripheral having 4 registers.
- AXI based Memory peripheral – user specific memory regions (upto 8) to provide local storage of data in PL. It supports burst transfer by default. This feature provides higher data transfer rates when using DMA controller for transactions. Fig. 5.4 shows one example of the memory peripheral having 4 block RAMs.

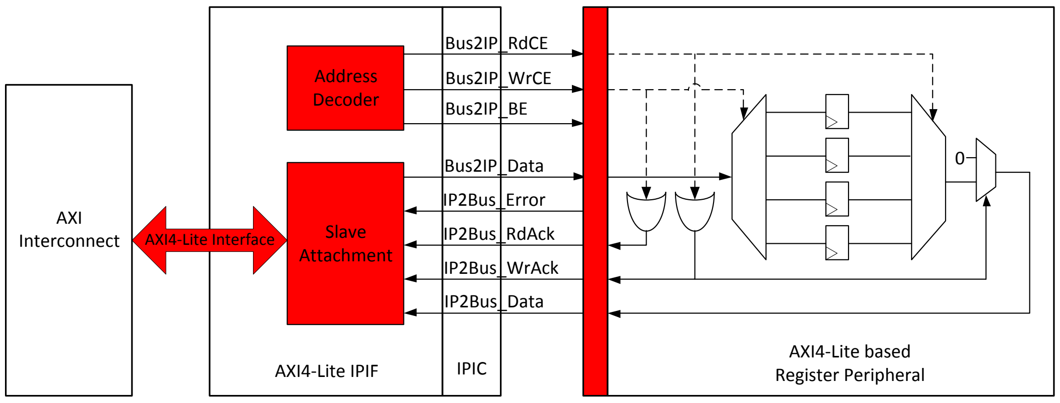


Figure 5.3: AXI4-Lite based Register Peripheral.

We evaluated the performance of all three ports ie. *GP,HP and ACP* by analyzing the communication between above mentioned peripherals and PS as discussed in further sections. These are then used along with the Linux kernel to develop our custom application.

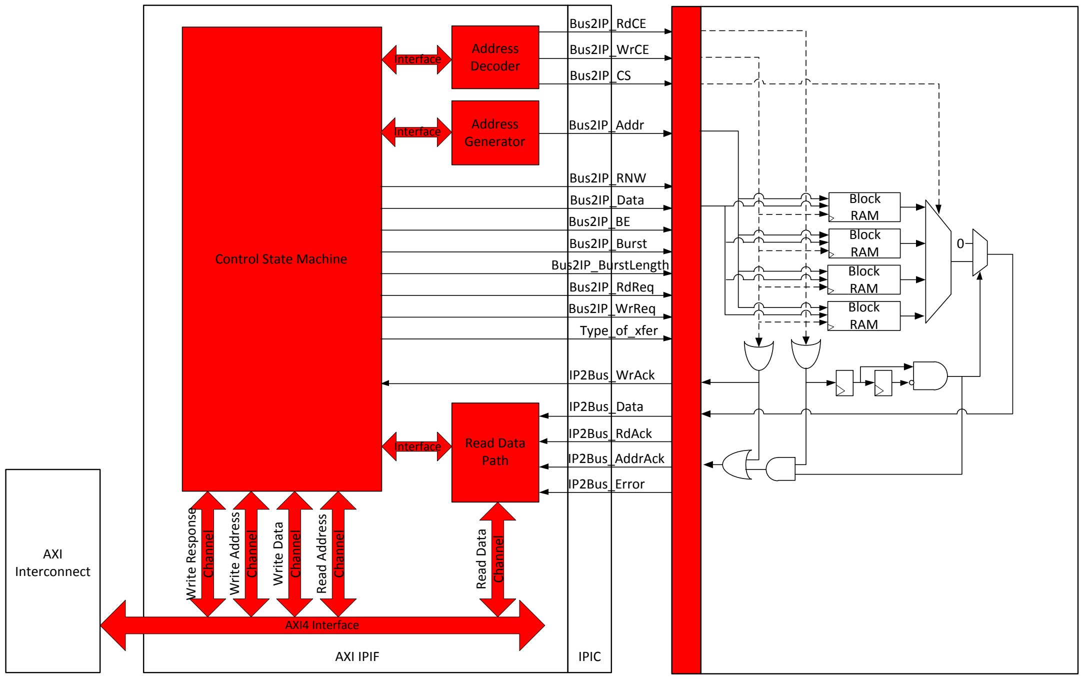


Figure 5.4: AXI4 based Memory Peripheral.

5.1 GP ports

AXI_GP interfaces are connected directly to the ports of master interconnect and slave interconnect, without any additional FIFO buffering, unlike the AXI_HP interfaces which has elaborate FIFO buffering to increase performance and throughput. In order to make use of GP master port, there are two communication mechanisms:

- PIO based
- PS-DMA based

In both of these methods, GP master port can be used to communicate with a slave peripheral. Slave can either be a register peripheral or a memory peripheral. In the next sections, we first describe these two methods in detail and then present the characterization of GP ports.

5.1.1 PIO based

One method of HW-SW communication in Zynq is to use Programmed IO (PIO) transfers from PS to PL. It requires no extra resources via GP master port. The CPU controls the data movement between main memory and the PL and hence latency between two transactions is quite high (150 ns) which corresponds to a bandwidth of approximately 25 MB/s (for data channel width = 4 byte). Thus, PIO is not suitable for large data transactions but is suitable for small data transaction (less than 1KB) and for controlling user registers like control and status register.

5.1.2 PS-DMA based

It is a method of transferring data via GP master port without processor intervention. PS-DMA controller takes a chunk of data from main memory and sends it to PL. The processor is free during this transfer and interrupted by the DMA controller at the end of data transfer. This hard DMA controller is able to perform MM-MM burst transactions and does not require any FPGA resources.

This method is suitable for applications with moderate bandwidth requirement (upto 80 MB/s) but normally faces the problem of large setup time overhead. A Multichannel First-In-First-Out (MFIFO) data buffer is used to store read/write data during DMA transfer. Theoretical maximum performance is 600MB/s. Xilinx provides bare metal SW drivers for PS DMA. The connectivity diagram is shown in Figure 5.5.

5.1.3 Characterization

We first connected the slave interface of the register peripheral to the GP port master interface via AXI lite interconnect. This scenario is similar to the one in which a processor communicates with general purpose I/O (GPIO). We did an experiment to find out the exact number of clock cycles to write 4 bytes of information to a register residing in PL. We set clock frequency to 100 MHz. We repetitively write 4 byte of data to the same register address. When measuring using Chipscope, latency between two transactions is obtained as 15 clock cycles as shown in Fig. 5.6.

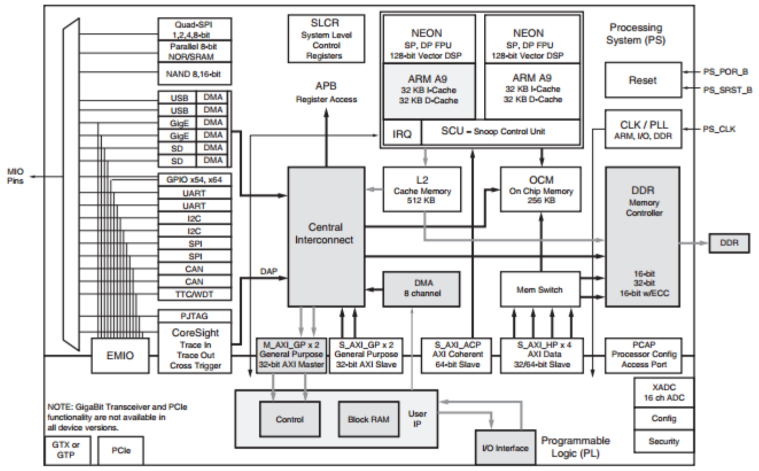

Quad-SPI 1,2,4,8-bit
Parallel 8-bit NOR/SRAM
NAND 8,16-bit
USB DMA
USB DMA
GigE DMA
GigE DMA
SD DMA
SD DMA
GPIO x54, x64
UART
UART
I2C
I2C
SPI
SPI
CAN
CAN
TTC/WDT
PJTAG
CoreSight Trace In Trace Out Cross Trigger
EMIO
MIO Pins
SLCR System Level Control Registers
APB Register Access
NEON SP, DP FPU 128-bit Vector DSP
ARM A9 32 KB I-Cache 32 KB D-Cache
IRQ
SCU – Snoop Control Unit
Processing System (PS)
Reset
CLK / PLL ARM, I/O, DDR
PS_POR_B
PS_SRST_B
PS_CLK
Central Interconnect
L2 Cache Memory 512 KB
OCM On Chip Memory 256 KB
DDR Memory Controller 16-bit 32-bit 16-bit w/ECC
DDR
DAP
DMA 8 channel
Mem Switch
M_AXI_GP x 2 General Purpose 32-bit AXI Master
S_AXI_GP x 2 General Purpose 32-bit AXI Slave
S_AXI_ACP AXI Coherent 64-bit Slave
S_AXI_HP x 4 AXI Data 32/64-bit Slave
PCAP Processor Config Access Port
XADC 16 ch ADC
Config
Security
NOTE: GigaBit Transceiver and PCIe functionality are not available in all device versions.
GTX or GTP
PCIe
Control
Block RAM
User IP
I/O Interface
Programmable Logic (PL)

Figure 5.5: PS DMA Connectivity Diagram.

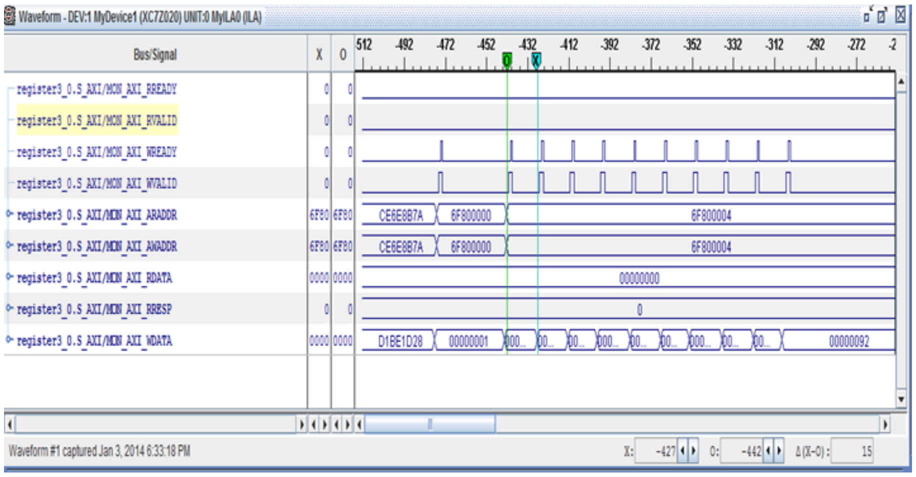

Waveform - DEV:1 MyDevice1 (XC7Z020) UNIT:0 MyILA0 (ILA)
Bus/Signal
X
O
512 -492 -472 -452 -432 -412 -392 -372 -352 -332 -312 -292 -272 -2
register3_0.S_AXI/MON_AXI_BREADY
register3_0.S_AXI/MON_AXI_RVALID
register3_0.S_AXI/MON_AXI_WREADY
register3_0.S_AXI/MON_AXI_WVALID
register3_0.S_AXI/MON_AXI_ARADDR
register3_0.S_AXI/MON_AXI_AWADDR
register3_0.S_AXI/MON_AXI_RDATA
register3_0.S_AXI/MON_AXI_BRESP
register3_0.S_AXI/MON_AXI_WDATA
CE6E8B7A
6F800000
6F800004
00000000
D1BE1D28
00000001
00000092
Waveform #1 captured Jan 3, 2014 6:33:18 PM
X: -427
O: -442
Δ(X-O): 15

Figure 5.6: PIO Transactions.

This experiment involved determining the latency in two consecutive write operations using PIO method. We did another experiment to see if it is possible to further reduce the latency below 150 ns by increasing the operating frequency. We found the results as shown in Fig. 5.7. Latency got reduced from 150 ns to 104 ns, as we increased the frequency from 100 MHz to 250 MHz.

We conducted another experiment to evaluate the bandwidth of GP port while both methods, PIO and PS-DMA based, were used for data transactions (sample size

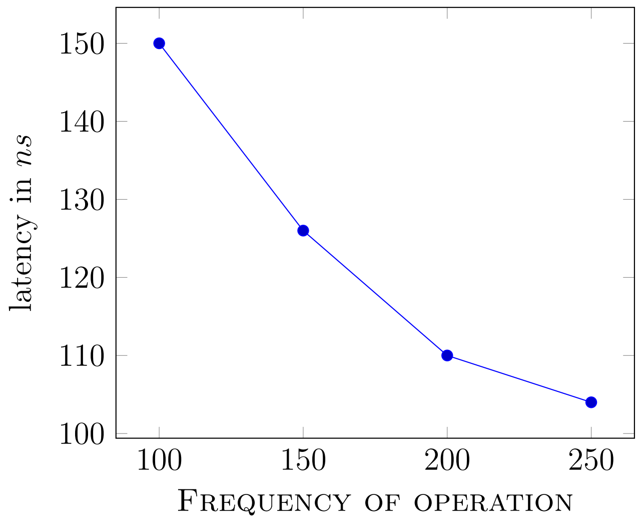


Figure 5.7: latency in ns

= 4 byte) between PL memory region and main memory. We set clock frequency to 100 MHz and connected the slave interface of the memory peripheral to the GP port master interface via AXI interconnect. A bare metal SW application was used for data transactions between 32 KB memory (implemented using 8 BRAMs) and the main memory. Results in Fig. 5.8 shows that PIO method takes less time compared to PS-DMA method for data size less than 1 KB. Hence, PIO method provide a better option for transferring small chunk of data like overlay configuration (which is generally less than 1KB) to the PL.

Table 5.1 and Fig. 5.9 shows maximum performance obtained using PS-DMA method as 80MB/s. The transfer time between PS and MFIFO is the performance bottleneck.

As we already mentioned that XPS provides an automated method to generate custom peripheral, containing slave or master interface, which can be either register peripheral or a memory peripheral. We have generated a custom memory peripheral using XPS and found that it does not make used of BRAM to implement memory. In order to make use of BRAM as a hard IP block available in Zynq device, we modified the RTL of the custom IP core and instantiated BRAM as hard IP block. Table 5.2 shows the resource usage of the vendor-generated and modified custom peripheral.

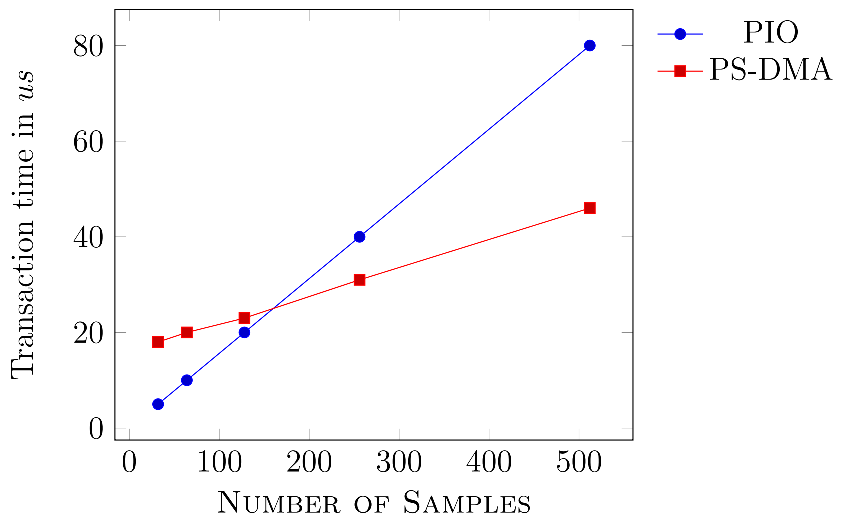


Figure 5.8: Time in us for data transactions

Table 5.1: Experiment results for PS-DMA based transactions

No. of Samples	Time taken in us	Throughput in MB/s
32	18.4	6.9
64	20.0	12.7
128	23.5	21.7
256	31.3	32.6
512	46.7	43.8
1024	76.6	53.4
2048	113.6	72.1
4096	210.6	77.7
8192	403.5	81.2

Table 5.2: Resource usage of memory peripheral

Resource Type	No. of resources in original peripheral	No. of resources in customized peripheral
LUT	8726	372
FF	16575	179
LUTRAM	33	33
BRAM	0	2

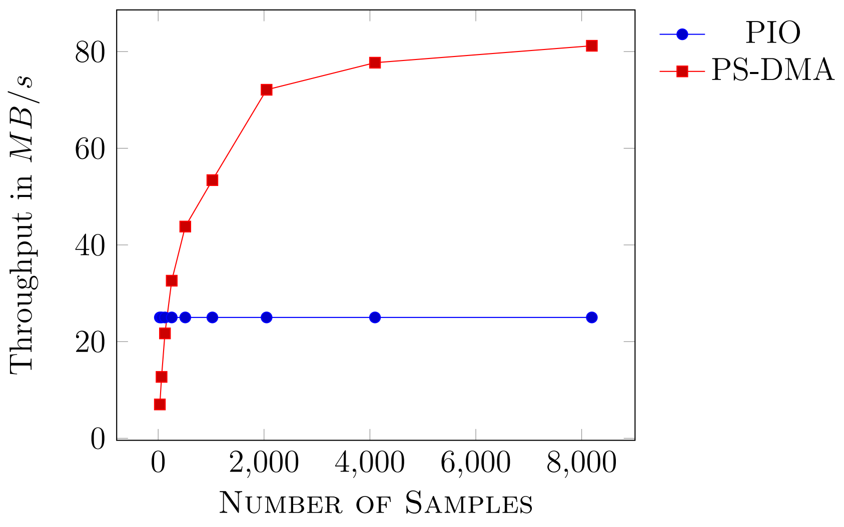


Figure 5.9: Throughput comparison between PIO and PS-DMA method

5.2 HP ports

In previous section, we have seen that GP master ports can be used effectively for controlling SW accessible register peripheral and also for the applications requiring moderate communication bandwidth (less than 80 MB/s). High performance (HP) slave ports can be used for applications requiring high communication bandwidth (more than or equal to 80 MB/s) at the cost of some extra area overhead. There are four HP interfaces, each including two FIFO buffers for read and write traffic. These interfaces are also referenced as AFI(AXI FIFO interface), to emphasize their buffering capabilities. The PL to memory interconnect routes the HP ports to the DDR memory ports or the OCM. In order to obtain maximum performance when using only two of the four HP ports, either the odd or the even numbered ports are used.

PS-DMA can not be used for data transfer through HP ports. Only a Soft-DMA implemented in the FPGA fabric can be used to transfer data through HP ports. Soft DMA uses an FPGA soft IP core to control the data movement between the memory and PL. Two variations are possible: memory mapped to memory mapped (MM-MM DMA) transactions and memory mapped to streaming (MM-S DMA) transactions. For soft DMA transfers the CPU is free during the transactions and can be interrupted by the DMA controller IP core at the end of data transfer. We have used the AXI Central DMA (CDMA) Controller IP core from Xilinx as a soft DMA for MM-MM transactions between DDR and memory peripheral. The DDR memory acts as the source and the memory in PL acts as destination. This connectivity diagram is shown in Figure 5.10.

Interrupt mode of data transfer is used which gets triggered either on completion of data transfer or occurrence of error. Data transfer steps are as follows:

First the source memory is filled with data and initial configuration settings for DMA and Interrupt controller are loaded. The CDMA register is loaded with source and destination address along with length of transfer. The transfer begins and when interrupt occurs the DMA transfer state is checked. CDMA controller uses a master interface to transfer data between two slave interfaces. In this experiment, this master interface is connected to the slave interface of the memory peripheral and also to one HP slave port. The CDMA slave is connected to GP port for configuration and status information. The maximum burst length is 256, which means that with a burst size of 4 and 8 bytes, we can transfer 1K and 2K bytes, respectively in a single burst. Huge FIFOs are used to facilitate large data transfer without stalling the PL. PS is responsible for the clock management. Xilinx also provides bare metal SW drivers to use with their CDMA IP core.

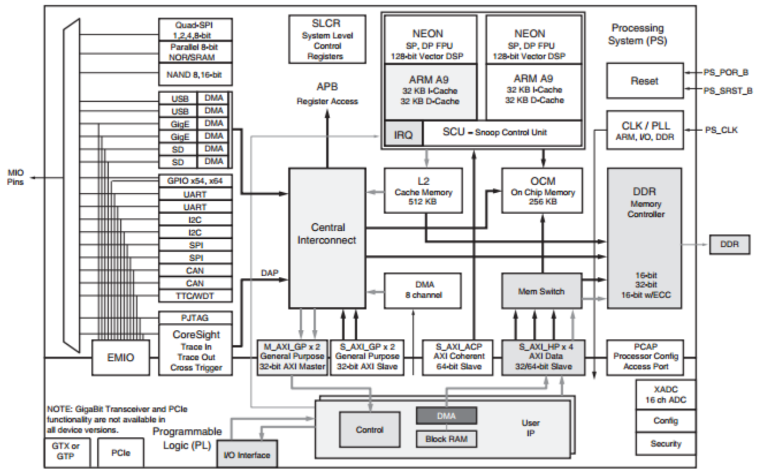

Quad-SPI 1,2,4,8-bit
Parallel 8-bit NOR/SRAM
NAND 8,16-bit
USB DMA
USB DMA
GigE DMA
GigE DMA
SD DMA
SD DMA
GPIO x54, x64
UART
UART
I2C
I2C
SPI
SPI
CAN
CAN
TTC/WDT
PJTAG
CoreSight Trace In Trace Out Cross Trigger
EMIO
MIO Pins
SLCR System Level Control Registers
APB Register Access
NEON SP, DP FPU 128-bit Vector DSP
ARM A9 32 KB I-Cache 32 KB D-Cache
IRQ
SCU – Snoop Control Unit
Processing System (PS)
Reset
PS_POR_B
PS_SRST_B
CLK / PLL ARM, I/O, DDR
PS_CLK
Central Interconnect
L2 Cache Memory 512 KB
OCM On Chip Memory 256 KB
DDR Memory Controller
16-bit 32-bit 16-bit w/ECC
DDR
DAP
DMA 8 channel
Mem Switch
M_AXI_GP x 2 General Purpose 32-bit AXI Master
S_AXI_GP x 2 General Purpose 32-bit AXI Slave
S_AXI_ACP AXI Coherent 64-bit Slave
S_AXI_HP x 4 AXI Data 32/64-bit Slave
PCAP Processor Config Access Port
XADC 16 ch ADC
Config
Security
NOTE: GigaBit Transceiver and PCIe functionality are not available in all device versions.
Programmable Logic (PL)
GTX or GTP
PCIe
I/O Interface
Control
DMA
Block RAM
User IP

Figure 5.10: PL DMA Connectivity Diagram.

5.2.1 Single CDMA IP Core and Memory Peripheral

Experiments are done to obtain throughput values for data transfer using Central Direct Memory Access (CDMA) soft IP core in the PL. In order to ensure cache coherency, SW application must manage caches which means explicit flushing and invalidation of caches needs to be performed. It takes many CPU cycles and affects application execution time as shown in Table 5.3.

Table 5.3: Experiment results for CDMA based transactions

No. of Samples	src-flush time in us	Transfer time in us	dst-flush time in us	Total time in us
32	1.3	3.6	0.9	5.93
64	1.9	4.2	1.5	7.81
128	3.3	5.5	2.7	11.61
256	5.8	8.1	5.0	19.01
512	11.4	10.7	9.7	31.81
1024	21.8	15.8	19.0	56.69
2048	43.6	26.2	37.6	107.64
4096	84.4	47	74.9	206.42
8192	169.7	88.5	149.5	407.72

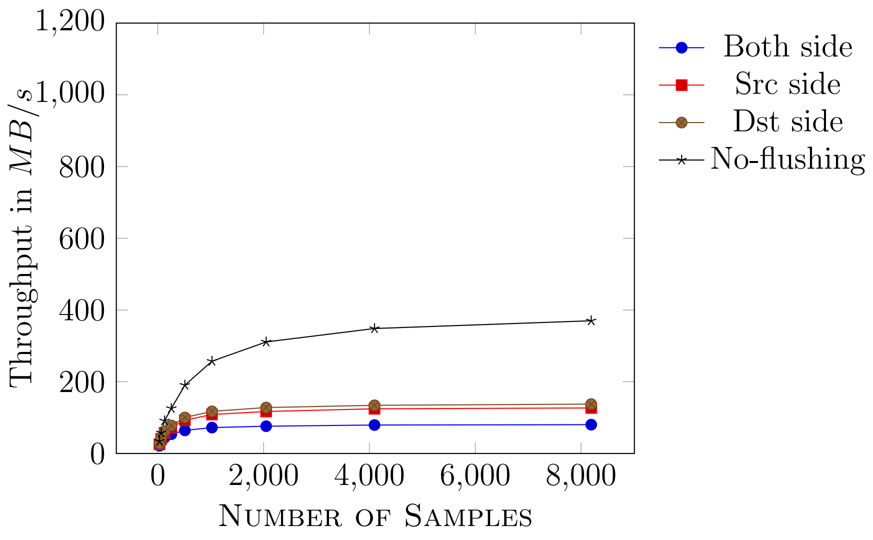


Figure 5.11: PL-DMA method

Maximum throughput obtained for data transfer is 80, 137, 126 and 370 MB/s for the cases of both side, source side, destination side and no flushing respectively.

5.2.2 Increasing operating frequency to 150 MHz

The operating frequency can be increased up to 150 MHz. The experiment presented in previous section is repeated for this increased frequency. Experimental results are shown below in the table 5.4.

Table 5.4: Experiment results for CDMA based transactions at 150 MHz

No. of Samples	src-flush time in us	Transfer time in us	dst-flush time in us	Total time in us
32	1.3	3.2	0.9	5.5
64	1.9	3.7	1.5	7.1
128	3.3	4.7	2.7	10.6
256	5.8	6.4	5.0	17.3
512	11.4	8.2	9.7	29.3
1024	21.8	11.7	19.0	52.6
2048	43.6	19.0	37.6	99.5
4096	84.4	33.3	74.9	193.6
8192	169.7	62.0	149.5	380.0

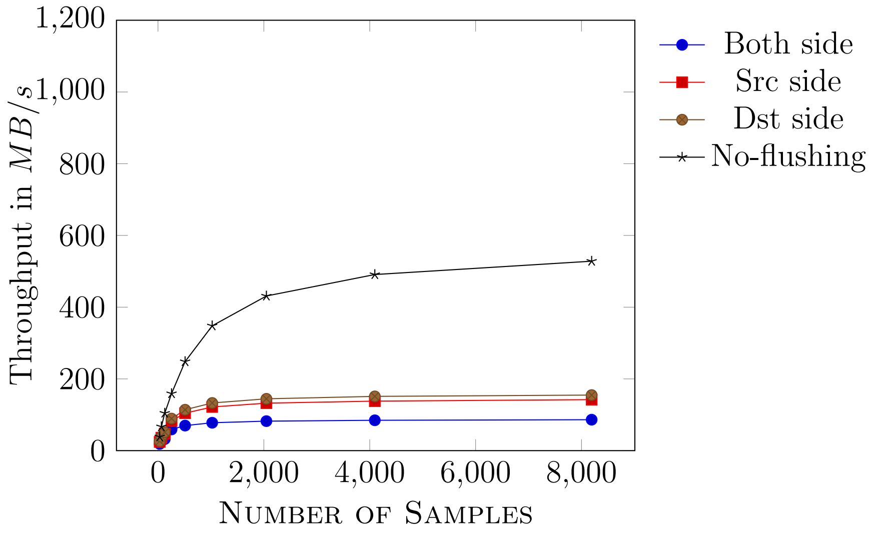


Figure 5.12: PL-DMA method at 150 MHz

It is clear from the Fig. 5.12 that there is no benefit of increasing operating frequency in cases where any kind of flushing is required. Maximum throughput obtained for data transfer is 86, 142, 154 and 528 MB/s for the case of both side, source side, destination side and no flushing respectively.

5.2.3 Increasing channel width to 64 bit

The HP port which is used in CDMA method supports up to 64 bit data transfer. The experiment is repeated with increase in size. The Experimental results are shown below in the table 5.5.

Table 5.5: Experiment results for CDMA based transactions for 64-bit data channel

No. of Samples	src-flush time in us	Transfer time in us	dst-flush time in us	Total time in us
32	1.9	4.2	1.5	7.6
64	3.2	4.8	2.7	10.8
128	5.8	6.1	5.0	17.0
256	11.1	8.7	9.7	29.5
512	21.5	11.4	19.0	52.0
1024	42.5	16.4	37.6	96.5
2048	84.4	26.6	74.9	185.9
4096	168.2	47.1	149.5	364.8
8192	335.8	88.1	298.6	722.6

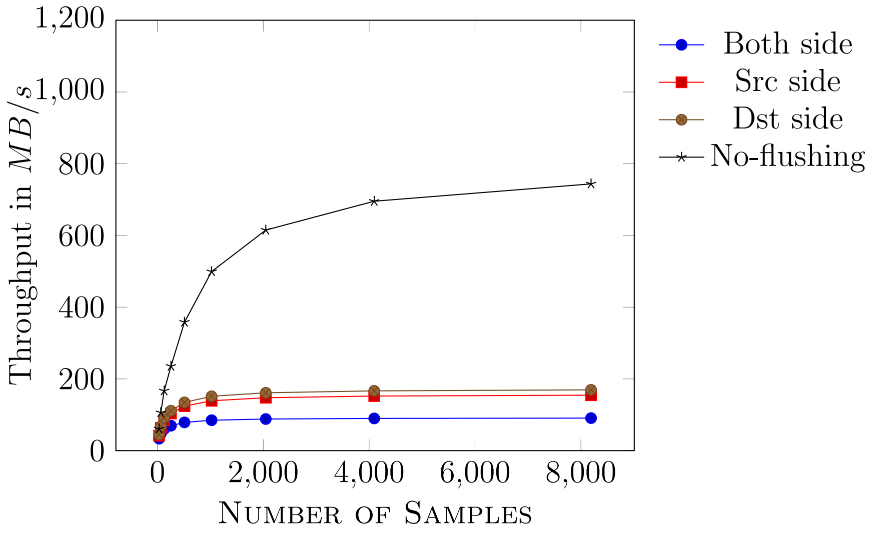


Figure 5.13: PL-DMA method at 64-bit

It is clear from the Fig. 5.13 that there is no benefit of increasing channel width, when flushing is required. Maximum throughput obtained for data transfer is 90, 154, 169 and 743 MB/s for the cases of both side, source side, destination side and no flushing respectively.

5.2.4 Final design for maximum bandwidth

The experiment is repeated with 64 bit data channel width as well as 150 MHz clock frequency. The Experimental results are shown below in the table 5.6.

Table 5.6: Experiment results for CDMA with 64-bit data channel at 150 MHz

No. of Samples	src-flush time in us	Transfer time in us	dst-flush time in us	Total time in us
32	1.9	3.32	1.5	6.8
64	3.2	3.84	2.7	9.8
128	5.8	4.9	5.0	15.8
256	11.1	6.53	9.7	27.3
512	21.5	8.25	19.0	48.8
1024	42.5	11.87	37.6	92.0
2048	84.4	19.06	74.9	178.4
4096	168.2	33.4	149.5	351.1
8192	335.8	62.3	298.6	696.8

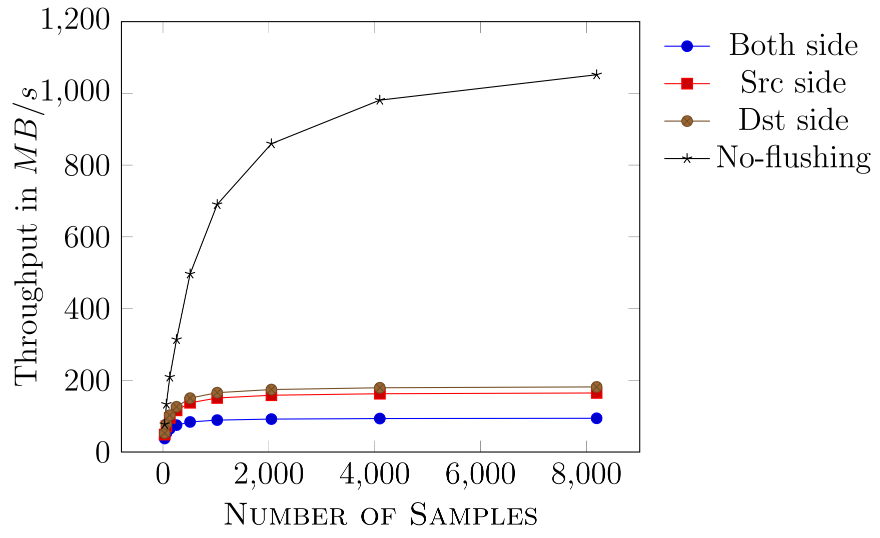


Figure 5.14: PL-DMA method at 64-bit and 150 MHz

It is clear from the Fig. 5.14 that there is no benefit of increasing both data width and operating frequency in cases where any kind of flushing is required. Maximum throughput obtained for data transfer is 94, 164, 181 and 1051 MB/s for the case of both side, source side, destination side and no flushing respectively.

5.3 ACP Port

One high performance 64 bits wide low-latency cache-coherent slave port. The port can access both L2 cache and on-chip memory. ACP port is connected to the Snoop control unit due to which caches do not need to be flushed and invalidated. SCU is responsible for cache coherency and management between L1 and L2 caches. Duplicated 4-way associative tag RAMs act as a local directory. It also manages arbitration, communication, cache and system memory transfers for the processor and ACP. During coherent write request the L1 cache is checked for address, if present, data is invalidated. The cache miss or invalidated cache then results in request to L2 cache or main memory. Therefore it requires no explicit cache invalidation unlike HP port. During coherent read, L1 cache is checked if there is a cache miss the L2 and main memory are checked hierarchically. Hence it has fast write but slow read when compared to HP port.

Table 5.7: Experiment results for CDMA based transactions through ACP

No. of	Time taken in us		Throughput in MB/s	
Samples	100 MHz	150 MHz	100 MHz	150 MHz
32	2.3	2.3	111.3	111.3
64	2.97	2.66	172.39	192.48
128	4.19	3.45	244.39	296.81
256	6.73	4.78	304.30	428.45
512	9.46	7.44	432.98	550.54
1024	14.58	10.93	561.86	749.50
2048	25.13	18.39	651.96	890.92
4096	46.6	32.82	703.17	998.42
8192	88.33	62.21	741.94	1053.46

CDMA experiment is conducted using the ACP port. As ACP is connected to the Snoop control unit, no explicit flushing is required. Therefore the throughput obtained is equal to transfer time without flushing of 64 bit HP port.

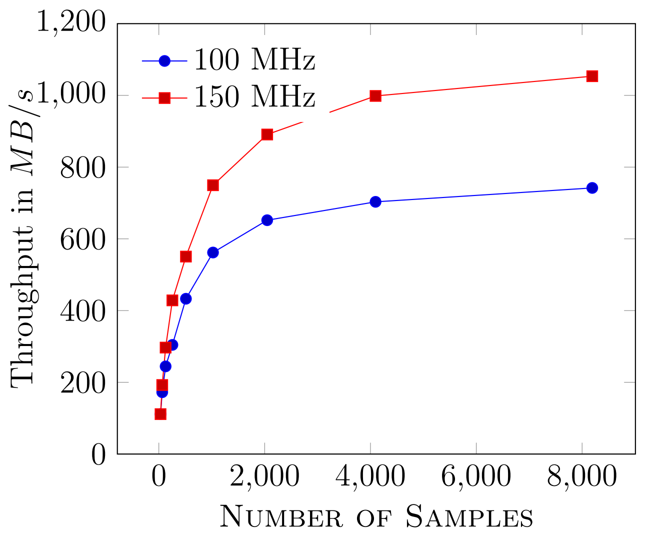


Figure 5.15: Throughput for CDMA transactions through ACP

5.4 Summary

Following is the summary of this chapter:

- Achievable bandwidth of GP Port using PIO method is 25 MB/s (200Mb/s) which is suitable for small size data transactions (less than 1KB) and for controlling user registers.
- Achievable bandwidth of GP port using PS-DMA method is approximately 80 MB/s (640 Mb/s) which is suitable for moderate bandwidth requirement (up to 80 MB/s).
- PIO and PS-DMA based methods don't require any FPGA hardware resources for data movement and hence consume less power.
- High performance (HP) ports can be used for applications requiring high bandwidth (more than 80 MB/s) at the cost of some area overhead (consumed by DMA controller).
- Theoretical bandwidth of HP port is 1200 MB/s (9.6 Gb/s) when using 64-bit channel and 150 MHz frequency. We achieved a bandwidth of 1050 MB/s (8.4 Gb/s) considering that pre-processing and post-processing of data is not required (which means cache coherency is not required).

- Cache coherency is an issue while using HP ports and SW application must have to manage caches by doing explicit flushing and invalidation.
- If cache coherency is required in the application, then we can only achieve a throughput of 95, 165, 180 MB/s for the cases of both side, source side and destination side flushing respectively.
- Upto four DMA controllers can be connected to HP ports and can provide a cumulative bandwidth of four times the above mentioned values. It means we can achieve maximum bandwidth of 4200 MB/s (33.6 Gb/s) using all HP ports simultaneously.
- HW support for cache flushing and invalidation have been provided in Zynq device in form of snoop control unit (SCP) and ACP port is directly connected to SCU due to which it provides best performance (1050 MB/s) even in the case when pre-processing and post-processing of data is required.

Chapter 6

Experiments

In this chapter, we present experiments related to hardware acceleration using streaming IP cores by making use of Linux OS infrastructure. Streaming IP core can either be an overlay architecture or a specialized hardware accelerator such as FIR filter. OS based experiments have been conducted under control of Xillinux running on Zedboard. This infrastructure contains communication abstraction in the form of Xillybus package. In order to study the overheads associated with OS abstraction, we conducted experiments using Xillybus system. We then inserted our custom core into the system and estimated the efficiency of the same.

6.1 Evaluation of Xillybus core

The Xillybus core uses the DMA soft IP core in order to perform transactions between the PS and the PL transparently. The Xillybus IP is unaware of the user logic characteristics like expected data rate, read/write occurrence and frequency. Except for the FIFO full and empty states, it has no knowledge of the FIFO state also. The Xillybus stream is similar to a TCP/IP stream which is efficient for high data rate or infrequently sent single byte transfers. DMA buffers are filled after which data is sent and acknowledged to provide an illusion of continuous data streaming. The application software is responsible for signaling the partially filled DMA buffers to push or pull the data for reduced latency. The signaling parameter is passed as the length parameter to the device driver. Its value controls the CPU load and hence is optimally chosen so as to make efficient use of the DMA buffer as well as to achieve the required performance. If the requested length of transfer cannot be completed, sleep mode is reached where the transfer operation occurs after 10 ms irrespective of presence or absence of data of required length.

We performed two main experiments: one in which communication happens in a loop-back fashion and another in which data passes through hardware accelerator (streaming FIR filter).

6.1.1 Loopback Experiment

The Xillybus demo bundle is used for this experiment with a loopback connectivity between the read and write FIFOs. Fig.6.1 shows this connection. The bit file generated for this design is downloaded on to the FPGA. The application is loaded to the Zedboard and native compilation is done on the target board itself. Data is written and read back from the processor and the time taken for the same is calculated as shown in Table 6.1. The throughput values obtained for this experiment are illustrated in Fig.6.2.

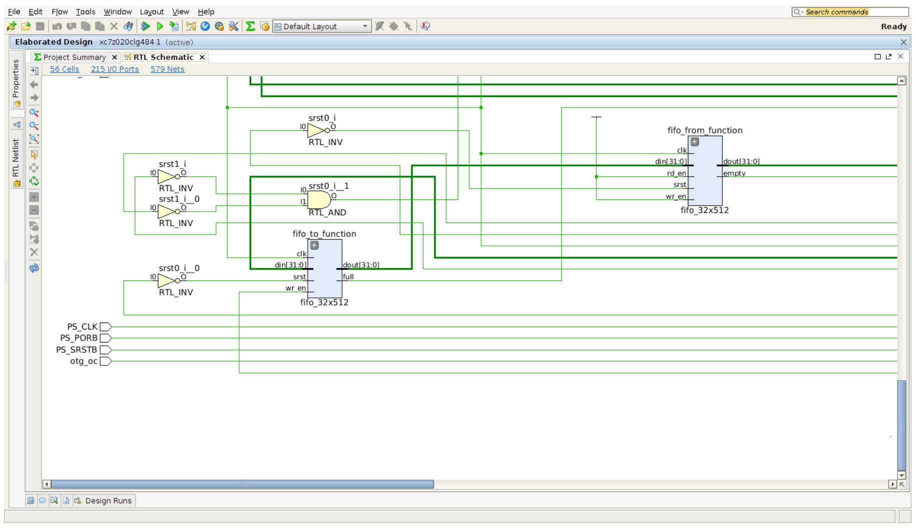


Figure 6.1: Connectivity diagram for Loopback experiment

Table 6.1: Experiment results for Xillybus loopback transactions

No. of Samples	Round trip Time in us	Throughput in KS/s
16	6175	2.59
32	6193	5.17
64	6304	10.15
128	6359	20.13
256	6286	40.73
512	6138	83.41
1K	6322	161.97
2K	6267	326.79
4K	6212	659.37

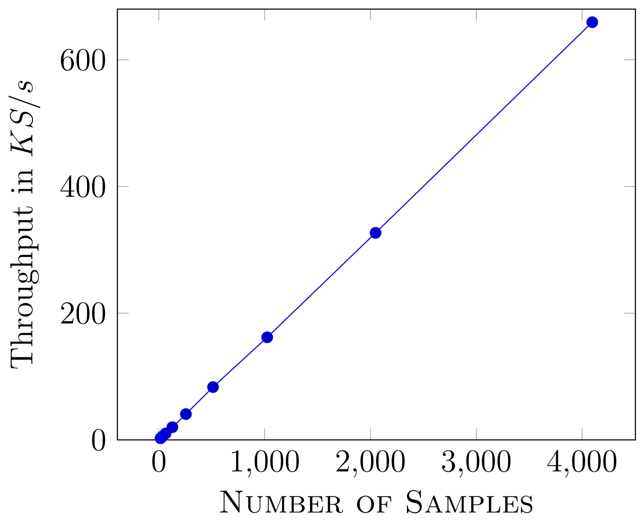


Figure 6.2: Throughput for Xillybus FIR transactions

6.1.2 Hardware accelerator Experiment

This experiment had been conducted after inserting hardware accelerator inside the Xillybus infrastructure as shown in Fig.6.3. The hardware accelerator used is a FIR filter. The round trip throughput is calculated by placing the FIR logic between the source and destination FIFOs. The FIR logic is obtained by synthesis of code in high level language *[C]* using Vivado-HLS tool. The filter uses streaming logic to process the input data. The input data and partial sums produced propagate through the filter to produce output. A 13 tap filter is used with random filter coefficients. HLS code is shown in code listing 6.1.

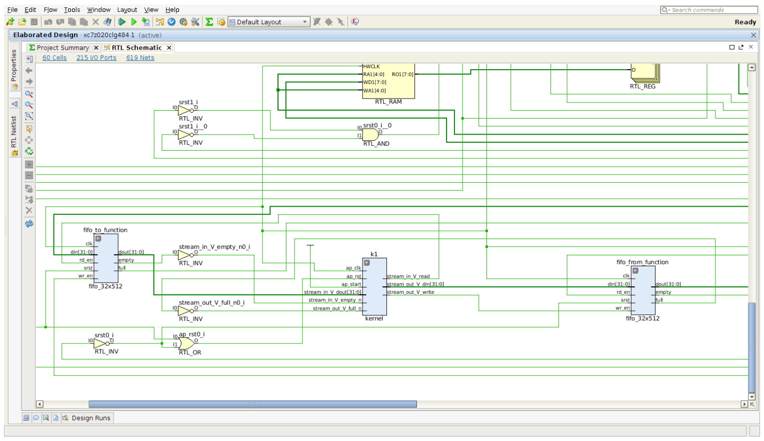


Figure 6.3: Connectivity diagram for Accelerator experiment

Listing 6.1: HLS code for FIR filter

```
#include "kernel.h"

using namespace hls;

void kernel(stream<uint32_t> &stream_in,stream<uint32_t> &stream_out)
{
        static int ind[]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
        int outd[13];
        int sum ;
        int temp[130];
        int i;
        int j=0;
        int p=0;
        int k[]={12,10,23,21,32,54,34,2,4,6,63,76,47};
        int Max=0;
        int N = 13;
        int x;
        int m;
        int acc;

        if (!stream_in.empty())
        {
                ind[0] = stream_in.read();
                sum =0;
                for(i=0;i<=12;i++)
                {
                        sum  += k[i] *  ind[i];
                }
                outd[0]=sum;
                if (!stream_out.full())
                {
                        stream_out.write(outd[0]);
                }
                for(p=12;p>=1;p--)
                {
                        ind[p]= ind[p-1];
                }
        }
}
```

The application code for FIR filter is shown in code listing 6.1.

Listing 6.2: Application code for FIR filter

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <pthread.h>
#include <stdint.h>
#include <sys/time.h>

        int fdr32 = 0;
        int fdw32 = 0;
        int N = 0;
        int i;
        int *array_input;
        int *array_hardware;
        struct timeval tv1, tv2;
        ssize_t t1,t2;

int main(int argc, char *argv[])
{

        if (argc!=4)
        {
                fprintf(stderr, "Usage:␣%d␣devfile\n", argc);
                exit(1);
        }
        fdr32 = open(argv[1], O_RDONLY);
        fdw32 = open(argv[2], O_WRONLY);
        N = atoi(argv[3]);
        if (fdr32 < 0 || fdw32 < 0)
        {
                perror("Failed␣to␣open␣devfiles");
                exit(1);
        }

        //allocate memory
        array_input = (int*) malloc(N*sizeof(int));
        array_hardware = (int*) malloc(N*sizeof(int));


        // generate inputs and prepare outputs
        for(i=0; i<N; i++)
        {
                array_input[i] = i;
                array_hardware[i] = 0;
        }
        gettimeofday(&tv1, NULL);
        t1=write(fdw32, array_input, sizeof(int)*N);
        t2= read(fdr32,array_hardware, sizeof(int)*N);
        gettimeofday(&tv2, NULL);
        printf("Execution␣time␣%f␣us␣\n\r",
                                (double)(tv2.tv_usec - tv1.tv_usec));

        return 0;
}
```

The latency and throughput values obtained by varying the data size are shown in Table 6.2. It is observed that the round trip time values are almost constant for sample size less than 4K. This behavior can be attributed to the use of ACP port for data transfer. As caches are used in this method, a constant behavior is seen for sequential access of data whose length is shorter than the cache size.

Table 6.2: Experiment results for Xillybus with accelerator for small sample size

No. of Samples	Round trip Time in *us*	Throughput in KS/s
16	6194	2.58
32	7023	4.56
64	6322	10.12
128	6304	20.30
256	6322	40.49
512	6303	81.23
1K	6175	165.83
2K	6248	327.78
4K	6212	659.37

The throughput plot for this experiment is given in Fig.6.4.

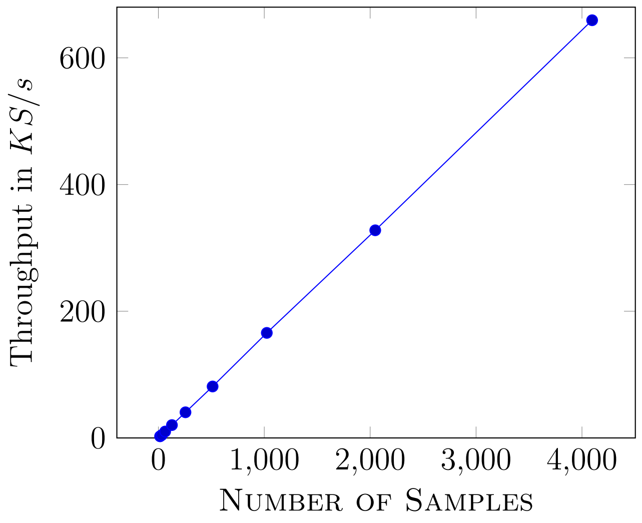


Figure 6.4: Throughput for Xillybus FIR transactions

As the sample size is increased beyond 4K samples, it is seen that the round trip time increases. This occurs due to the cache size limit which results in cache miss for large data sizes. This cache miss causes extra latency to obtain data from the main memory. The experimental values obtained for large data sizes are presented in Table 6.3 and Fig.6.5.

Table 6.3: Experiment results for Xillybus with accelerator for large sample size

No. of Samples	Round trip Time in us	Throughput in KS/s
8K	15962	513.22
16K	13953	1174.23
32K	14137	2317.89
64K	19169	3418.85
128K	14672	8933.48
256K	21842	12001.83
512K	26542	19753.15
1M	29454	35600.46
2M	33436	62721.38
4M	58522	71670.55
8M	105856	79245.47

The throughput plot for this experiments are given in Fig.6.5.

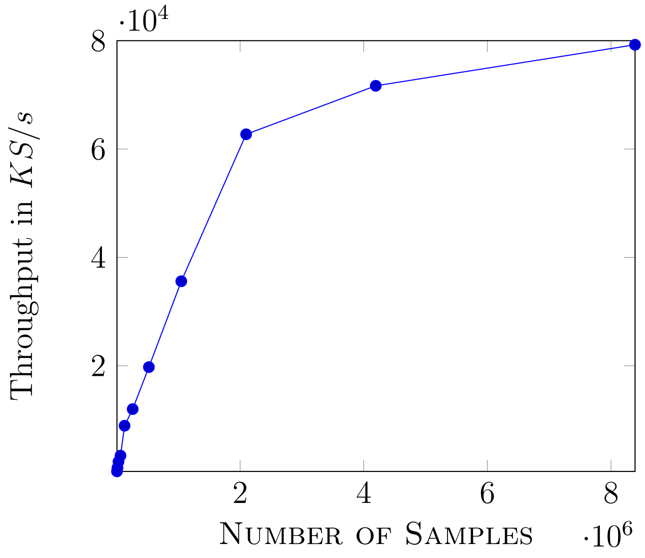


Figure 6.5: Throughput for Xillybus FIR transactions

6.2 Performance estimation of custom core

In previous section, we observed that inclusion of Xillybus core is beneficial for large data streaming. However in the case of small data streaming, latency associated with the core undermines its usage. In this section, we present performance estimation of our own developed custom core (for small data streaming) which we integrated in the system as described in chapter 4. There are three data transfer mechanisms which can be used in such a scenario.

- PIO method
- PS DMA method
- PL DMA method

The details of each method for the Bare-metal usage is presented in Chapter5. The Bare-metal technique provides high performance but absence of OS support makes its usage difficult. Adapting them to the OS at the cost of low overhead is attempted in the experiments described in next couple of sections.

6.2.1 PIO Xillinux experiment

This experiment is conducted on Xillinux platform using our custom core. Data is transferred from the PS to the custom core (by doing memory mapping) in order to access the peripheral through the dev/mem port. The mmap() function is utilized in order to map the peripherals physical address space to virtual address space. From the experiments it is seen that it takes 15 cycles (clock frequency 100 MHz) for the custom core to receive the one sample of data sent from the PS. It corresponds to a latency of 150 ns per transaction which is similar to what we observed in case of bare metal experiments. Table6.4 shows the time taken for data relay.

Round trip time in this case depends not only on the communication latency but also computation latency which in turn is dependent on the size of data transferred. In Xillybus the transfer time was independent of the data size due to the use of hard FIFOs to buffer the read and write operations. This characteristic is illustrated clearly in the Table 6.5. The round trip throughput saturates at 3 MS/s.

Table 6.4: PS to PL Communication latency for Xillinux using PIO

No. of Samples	Time in *us*
16	3
32	5
64	9
128	20
256	39
512	81
1K	159
2K	321
4K	640

Table 6.5: Experiment results for Xillinux using PIO

No. of Samples	Round trip Time in *us*	Throughput in MS/s
16	6.16	2.59
32	10.32	3.1
64	18.64	3.43
128	41.28	3.1
256	80.56	3.06
512	167.12	3.06
1K	328.24	3.11
2K	662.48	3.09
4K	1320.96	3.10

6.2.2 PS-DMA Xillinux experiment

The Programmable Input Output (PIO) experiment conducted in the previous section shows behavior similar to the Baremetal case. This paves way for adapting the experiment to accommodate PS-DMA based transfers in order to improve its performance. But DMA transfer cannot be accomplished by memory mapping technique due to the absence of interrupts in the aforementioned method. This can be overcome by creating custom device driver for PS-DMA.

In order to implement a custom device driver, the device tree needs to be modified. This is done in order to ensure that the kernel detects the hardware and loads the driver. The driver should contain information such as physical address, interrupt details and application specific details. The steps involved in setting up device tree is as follows:

The device tree element name needs to be defined and also its address which can be obtained from XPS. If interrupts are required, it needs to be declared with allocation of address. Custom parameter details can also be added. The Device Tree Source (DTS) file of Xillinux is obtained by executing the following command in the user/kernel/(uname r) directory of the root file system (RFS).

```
scripts/dtc/dtc -I fs -O dts -o  /effective.dts /proc/devicetree/
```

After adding the data pertaining to the new driver, the DTS file is converted into its binary format [Device Tree Binary (DTB)] by executing this command in the same directory.

```
scripts/dtc/dtc -I dts -O dtb -o /path/to/devicetree.dtb /path/to/devicetree.dts
```

The dtb along with the bitfile, boot.bin and image file is then loaded to the SD card. The custom driver file is converted to kernel object file (.ko) and the module is linked to the running kernel using insmod command. The dev node for the driver is created in order to provide user space entry point with parameters for type of node and major number and minor number. Node of character type is used in which data is written serially byte by byte.

```
mknod dma-fifo c 60 0
```

The data is then transferred using the following command

```
dd if=/dev/urandom bs=1024 count=1 of=/dev/mem
```

This experiment can be repeated by increasing the sample size by varying the *"bs"* parameter to characterize its behavior. The driver provided by Xilinx does not support our custom core. In order to provide support of PS-DMA for our custom core, we are in the process of developing a device driver.

6.2.3 Conclusion

From the experiments conducted on Xillinux it is clear that PIO shows similar behavior both in Xillinux as well as Baremetal case. This happens due to the use of memory mapping technique in which the kernel memory is directly accessed. The major drawback in use of this technique is the exposure of kernel and also the absence of interrupts which makes it unsuitable for DMA transfer. PS-DMA experiment conducted by creation of custom device drivers can be explored to obtain the dual benefits of performance and OS abstraction.

Chapter 7

Conclusions and Future Work

This chapter concludes and summarizes this report. Furthermore, in this chapter we discuss future research directions in detail.

7.1 Conclusions

This report proposed an OS supported approach for efficient communication abstraction by using a memory subsystem around an overlay architecture. This work included developing an understanding of embedded reconfigurable platforms, hardware acceleration concept, terminologies and techniques, HW-SW communication mechanisms and characterization of communication interfaces. Experiments were designed to test the performance of HW-SW communication interfaces of a commercial reconfigurable platform, the Xilinx-Zynq. A memory sub-system was developed by inserting a custom core into Xillybus infrastructure and performance estimation was presented of the same. Before we could begin developing memory sub-system, an in-depth knowledge of the current trends and previous efforts in this field were studied to compare and contrast their features. Experiments were conducted to characterize communication interfaces on the Zynq platform and results were presented in chapter 5. A proposal of the memory subsystem was presented in chapter 4 which was evaluated furthermore by the experiments presented in 6. In order to provide OS support for the communication abstraction, we evaluated Xillybus infrastructure and observed it efficiency for large data streaming (above 4K samples) but inefficiency for small data streaming (below 4K samples). We then inserted our own developed custom core to support high throughput (3 MS/s) for small data streaming (below 4K samples) using PIO method. We estimated the performance of our custom core and showed that it works better compared to xillybus core in case of small data streaming. Furthermore, the approach presented in this report facilitates high level application developers to use programmable system on chips for hardware acceleration without need for hardware know-how.

7.2 Future work

The throughput can be further improved by using device drivers for DMA controllers for small data streaming (below 4K samples). Adapting PIO based experiment to transfer data using PL330 DMA controller would increase the throughput of transfer as seen in the bare-metal case. We expect a throughput of 10 MS/s using PL330 DMA controller. Moreover, it would be beneficial to provide multiple interfaces from our custom core to overlay since overlay can have multiple input and output interfaces. It will allow multiple tasks to run in parallel and each task would be able to request its own interface. It would also be beneficial to explore the feasibility of integration of the infrastructure (proposed in this report) with other operating systems such as an RTOS or a Hypervisor. Finally, with these initiatives we hope to reduce HW-SW communication bottleneck and provide a uniform communication abstraction while executing tasks on overlay architectures.

Bibliography

[1] Xilinx Ltd. FPGA vs. ASIC Design Advantages. http://www.xilinx.com/fpga/asic.htm.

[2] ARM Ltd. The ARM Cortex-A9 Processors. http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf.

[3] Xilinx Ltd. Zynq-7000 technical reference manual. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, 2013.

[4] Xillybus Ltd. Xillybus: IP Core Product Brief. http://xillybus.com/downloads/xillybus_product_brief.pdf.

[5] T.J. Callahan, J.R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, April 2000.

[6] Eric S. Chung, James C. Hoe, and Ken Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Proceedings of the International Symposium on Field programmable gate arrays (FPGA)*, pages 97–106, 2011.

[7] K. Eguro. SIRC: an extensible reconfigurable computing communication API. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 135–138, 2010.

[8] M. Jacobsen, Y. Freund, and R. Kastner. RIFFA: a reusable integration framework for FPGA accelerators. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 216–219, May 2012.

[9] M. Jacobsen and R. Kastner. RIFFA 2.0: A reusable integration framework for FPGA accelerators. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, September 2013.

[10] D. Thomas, J. Coutinho, and W. Luk. Reconfigurable computing: Productivity and performance. In *Asilomar Conference on Signals, Systems and Computers*, pages 685–689, 2009.

[11] G. Stitt and J. Coole. Intermediate fabrics: Virtual architectures for near-instant FPGA compilation. *IEEE Embedded Systems Letters*, 3(3):81–84, September 2011.

[12] ZedBoard Project. Zynq Evalualtion & Development Board. `http://zedboard.org/content/overview`.

[13] O.T. Albaharna, P. Y K Cheung, and T.J. Clarke. On the viability of FPGA-based integrated coprocessors. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 206–215, 1996.

[14] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: high-level synthesis for FPGA-based Processor/Accelerator systems. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 33–36, 2011.

[15] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden. Programming models for hybrid FPGA-cpu computational components: a missing link. *IEEE Micro*, 24(4):42–53, 2004.

[16] Grant Martin, Brian Bailey, and Andrew Piziali. *ESL design and verification: a prescription for electronic system level methodology.* Morgan Kaufmann, 2010.

[17] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[18] Xilinx Ltd. AXI Reference Guide. `http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf`.

[19] Xilinx Ltd. ChipScope Pro Software and Cores. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/chipscope_pro_sw_cores_ug029.pdf`.

[20] Rolf Enzler, Christian Plessl, and Marco Platzner. Virtualizing hardware with multi-context reconfigurable arrays. In *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL*, pages 151–160, 2003.

[21] Neil W. Bergmann, Sunil K. Shukla, and Jrgen Becker. QUKU: a dual-layer reconfigurable architecture. *ACM Transactions on Embedded Computing Systems (TECS)*, 12:63:1–63:26, March 2013.

[22] Davor Capalija and Tarek S. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2013.

[23] A. Brant and G.G.F. Lemieux. ZUMA: an open FPGA overlay architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96, 2012.

[24] Alexander Dunlop Brant. Coarse and fine grain programmable overlay architectures for fpgas. Master's thesis, 2013.

[25] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 13–22, October 2010.

[26] Khoa Dang Pham, Abhishek Kumar Jain, Jin Cui, Suhaib A Fahmy, and Douglas L Maskell. Microkernel hypervisor for a hybrid ARM-FPGA platform. In

Proceedings of the International Conference on Application-Specific Systems, Architecture Processors (ASAP), 2013.

[27] A. Agne, M. Platzner, and E. Lubbers. Memory virtualization for multithreaded reconfigurable hardware. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 185–188, September 2011.

[28] Shinya Takamaeda-Yamazaki, Kenji Kise, and James C. Hoe. Pycoram: Yet another implementation of coram memory architecture for modern fpga-based computing. In *Workshop on the Intersections of Computer Architecture and Recon?gurable Logic (CARL)*, 2013.

[29] Aws Ismail. *Operating system abstractions of hardware accelerators on field-programmable gate arrays*. Thesis, August 2011.

[30] Xillybus Ltd. Getting started with Xillinux for Zynq-7000 EPP. `http://xillybus.com/downloads/doc/xillybus_getting_started_zynq.pdf`.

[31] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceedings of the International Symposium on Field programmable gate arrays (FPGA)*, pages 25–28, 2011.

[32] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 225–235, 2000.

[33] Michael K. Papamichael and James C. Hoe. CONNECT: re-examining conventional wisdom for designing nocs in the context of FPGAs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 37–46, 2012.

[34] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl. ReconOSAn operating system approach for reconfigurable computing. *IEEE Micro*, 2013.