NANYANG TECHNOLOGICAL UNIVERSITY

FPGA OVERLAY ARCHITECTURES ON THE XILINX ZYNQ AS PROGRAMMABLE ACCELERATORS

by

SHEN YUE
(G1402150A)

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

July 2015

Contents

List of Figures

List of Tables

Abbreviations

API	Application Processing Interface
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
COTS	Commercial Off-the-shelf
DFG	Data Flow Graph
DMA	Direct Memory Access
FPGA	Field Programmable Gate Array
FU	Functional Unit
GPP	General Purpose Processor
HDL	Hardware description language
HLS	High Level Synthesis
IF	Intermediate Fabric
II	Initiation Interval
ISE	Integrated Software Environment
PL	Programmable Logic
PS	Processing system
RTL	Register Transfer Level
XPS	Xilinx Platform Studio

Abstract

Emerging reconfigurable platforms tightly couple capable processors with high performance reconfigurable fabrics. This coupling promises to move the focus of reconfigurable computing systems from static accelerators to a more software oriented view, where reconfiguration is a key enabler for exploiting available hardware resources. This requires a revised look at how to use reconfigurable hardware within a software-centric processor-based system. Recently, coarse grained overlay architectures have been shown to be effective when paired with general purpose processors as this allows the hardware fabric to be viewed as a software-managed hardware task, enabling more shared use, offering software-like programmability, fast compilation, application portability and improved design productivity. These architectures enable general purpose hardware accelerators, allowing hardware design at a higher level of abstraction, but at the cost of area and performance overheads. This report examines the DySER overlay architecture as a hardware accelerator paired with a general purpose processor in the Xilinx Zynq. We evaluate the DySER architecture mapped on the Xilinx Zynq and show that it suffers from a significant area and performance overhead. We then propose an improved functional unit architecture using the flexibility of the DSP48E1 primitive which results in a 2.5$\times$ frequency improvement and 25% area reduction compared to the original functional unit architecture. We demonstrate that this improvement results in the routing architecture becoming the bottleneck in performance. We also develop an overlay architecture using a linear array of time-multiplexed functional units which can be used to host arbitrary size Data Flow Graph (DFG) of a compute kernel considering N$\geq$D holds true, where N is the number of functional units in the overlay and D is the depth of the DFG. Finally, we present experiments to evaluate the performance of HLS generated fully parallel and pipelined RTL implementations and the ARM processor for the execution of a set of compute kernels.

Chapter 1

Introduction

1.1 Motivation

While the performance benefits of reconfigurable computing over processor based systems have been well established [9, 10, 11], such platforms have not seen wide use beyond specialist application domains such as digital signal processing and communications. Poor design productivity has been a key limiting factor, restricting their effective use to experts in hardware design [12]. This requires a revised look at how to use Field Programmable Gate Array (FPGA) fabrics within a software-centric processor-based system effectively. Coarse grained overlay architectures [13, 14, 4, 5, 1, 15, 6, 16] have been shown to be effective when paired with general purpose processors [2, 14] as this allows the hardware fabric to be viewed as a software-managed hardware task, enabling more shared use, offering software-like programmability, fast compilation, application portability and improved design productivity. Overlay architectures consist of a regular arrangement of coarse grained routing and compute resources. The key attraction of overlay architectures is software-like programmability through mapping from high-level descriptions, application portability across devices, design reuse, fast compilation by avoiding the complex FPGA implementation flow, and hence, improved design productivity. Although research in the area of overlay architectures has increased over the last decade, the field is still in its infancy with only relatively few overlay architectures demonstrated in prototype form [14, 17, 5]. One such example is the DySER architecture targeted to the Xilinx Virtex-5 FPGA [18]. Area and performance overheads have, however, prevented the realistic use of DySER in practical FPGA-based systems. One of the reasons for this poor performance is that overlays are typically designed without serious consideration of the underlying FPGA architecture. In our work, we aim to examine the DySER overlay architecture as a hardware accelerator paired with a general purpose processor in the Xilinx Zynq. We also aim to develop an overlay architecture using a linear array of time-multiplexed functional units which can be used to host arbitrary size DFG of a compute kernel.

1.2 Contribution

Embedded hard macros, such as DSP blocks, have been added to FPGAs in recent years. Many existing overlay architectures [14, 17, 5, 18] do not specifically use these macros, except insofar as they are inferred by the synthesis tools. In this report, we first evaluate the DySER architecture mapped on the Xilinx Zynq and show that it suffers from a significant area and performance overhead. We then propose an improved functional unit architecture using the flexibility of the DSP48E1 primitive which results in a 2.5× frequency improvement and 25% area reduction compared to the original functional unit architecture. We demonstrate how adopting the Xilinx DSP48E1 primitive in the Functional Unit (FU) of the DySER architecture improves both performance and area, which results in the routing architecture becoming the bottleneck in performance. Another contribution of this report is the implementation of an overlay architecture based on time-multiplexed functional units. The main contributions can be summarized as follows:

- RTL implementation of a functional unit (compatible with the DySER architecture) using the DSP48E1 primitive, which can operate at near theoretical maximum frequency.
- RTL implementation of a time-multiplexed FU using the DSP48E1 primitive and an implementation of a linear array of these units as an overlay.

1.3 Organization

The remainder of the report is organized as follows: Chapter 2 presents background information on computer kernels, overlay architectures and software-hardware communication. Chapter 3 studies current state of the art in high performance overlay architectures. In chapter 4, we present the analysis of DySER architecture, functional unit, implementation on the Xilinx Zynq and a modified functional unit architecture using the flexible DSP48E1 primitive. In chapter 5, we present a time multiplexed functional unit and a linear array of these units as an Overlay architecture. Chapter 6, presents experiments to evaluate the performance of HLS generated RTL implementations and the ARM processor for the execution of a set of compute kernels. We conclude in chapter 7 and discuss future work.

Chapter 2

Background

2.1 Execution of Compute Kernels

We refer to compute kernels as computationally intensive part of an application which can be offloaded to an accelerator for fast execution. A General Purpose Processor (GPP) can be used for the execution of compute kernels by describing their functionality using a high level language. Since a GPP executes the kernel operations sequentially, the execution time of the kernels increases on increasing the complexity of the kernel. With the advancements in technology, parallel processing architectures such as multi-cores CPUs and DSPs, GPUs, Massively parallel processor arrays, FPGA based accelerators are gaining popularity for accelerated execution of kernels. Application Specific Integrated Circuit (ASIC) have been usually used as accelerator, but due to its limits of flexibility and long time to market, FPGAs are becoming popular for rapid-prototyping of accelerators. FPGA devices can be used for implementing kernels as high performance fully parallel and pipelined designs [19]. For more than a decade, researchers have shown that FPGAs can accelerate a wide range of software, in some cases by several orders of magnitude compared to state-of-the-art general purpose processors [20, 21].

To use an FPGA for accelerating compute kernels, designers typically start by manually converting the compute kernel into an fully pipelined datapath specified using Hardware description language (HDL). A fully pipelined datapath on FPGA results in maximum performance by producing output data at every clock cycle. However this performance comes at the cost of designer effort. Therefore, High Level Synthesis (HLS) has been proposed as a way of addressing the limited design productivity and manpower capabilities associated with hardware design [22, 23]. Advancements in HLS tools have helped raise the level of programming abstraction from Register Transfer Level (RTL) to high level languages, such as C or C++. Even as HLS tools improve in efficiency, prohibitive compilation times (specifically the place and route times in the backend flow) still limit productivity and mainstream adoption [12]. Hence, there is a growing need to make FPGAs more accessible to application developers who are accustomed to software API abstractions and fast development cycles [24].

Coarse grained configurable overlay architectures have been proposed as a method to overcome some of these issues [13, 14, 4, 5, 1, 16]. Overlays can be used for reducing the prohibitive compilation time required to map an application to the conventional fine-grained FPGA fabric. Overlays have also been shown to be effective when paired with general purpose processors [2, 14] as this allows the hardware fabric to be viewed as a software-managed hardware task, enabling more shared use. We describe FPGA Overlay architectures in the next section.

2.2 FPGA Overlay Architectures

Overlay architectures consist of a regular arrangement of coarse grained routing and compute resources. The key attraction of overlay architectures is software-like programmability through mapping from high level descriptions, application portability across devices, design reuse, fast compilation by avoiding the complex FPGA implementation flow, and hence, improved design productivity. Accelerators can be described at a higher level of abstraction and compiling it for overlays is several orders of magnitude faster than for the fine grained FPGAs. The overlay overcomes the need for a full cycle through the vendor implementation tools, instead presenting a much simpler problem of programming an interconnected array of processing elements. The possible configuration space and reconfiguration data size is much smaller than for direct FPGA implementation of kernels because of the coarser granularity of the overlay. An overlay provides a leaner mechanism for hardware task management at runtime as there is no need to prepare distinct bitstreams in advance using vendor-specific compilation (synthesis, map, place and route) tools. Instead, the behaviour of the overlay can be modified using software defined overlay configurations.

Despite having the implementation of the overlay architecture and its performance gain, there is no guarantee that it will surely provide reduction in kernel execution time. It depends heavily on how the overlay is interfaced to the host processor, communication mechanism between overlay, host processor and the external memory, communication bandwidth and latencies etc. Researchers have shown the effective use of coarse grained overlay architectures by pairing them with host processors as a coprocessor [25, 2] or as a part of the processor's pipeline [18]. Fig. 2.1 shows the integration of DySER [18, 26] overlay into the pipeline of a processor.

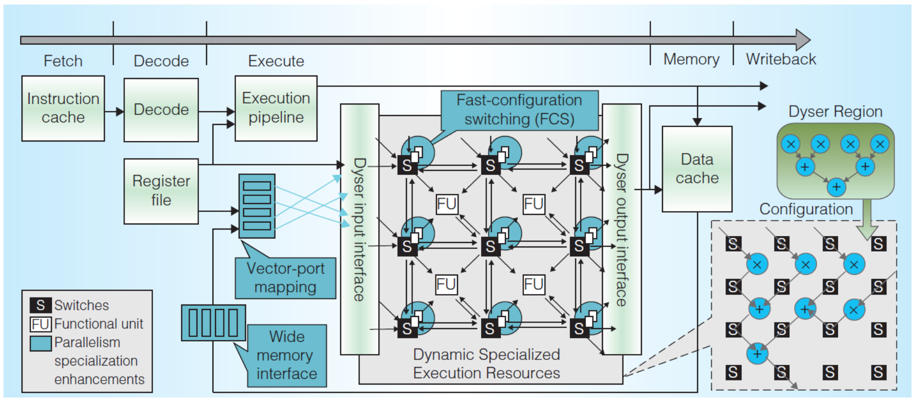


Figure 2.1: DySER Interfacing with Host Processor [1]

Integrating an overlay within a processor pipeline can provide huge performance and energy efficiency at the expense of complete redesign of processor micro-architecture. Another possible approach is to interface the overlay (as a co-processor) with the host processor via standard communication interfaces. To address possible bottleneck problems, particularly in providing high bandwidth transfers between the host procesor and the co-processor implemented on the FPGA fabric [27], it has been proposed to more tightly integrate the processor and the FPGA fabric. A number of tightly coupled architectures have resulted [28, 29], including vendor specific systems with integrated hard processors. One example of pairing the overlay (Intermediate Fabric (IF) Overlay [4]) with a high performance ARM processor via an Advanced eXtensible Interface (AXI) interface in a commercial computing platform (the Xilinx Zynq[30]) is shown in Fig. 2.2. Zynq platform partition the hardware into a Processing 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 describe Zynq platform in the next section.

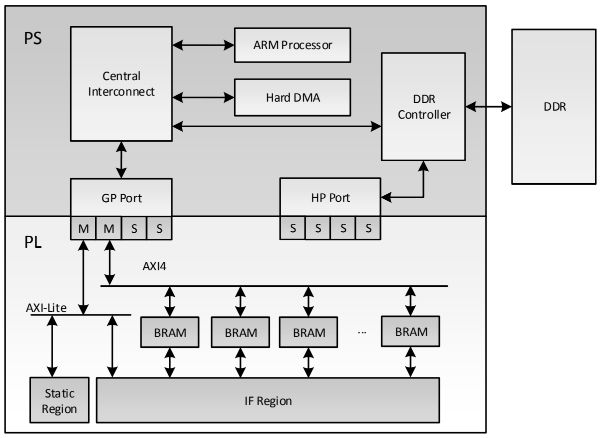


Figure 2.2: Intermediate Fabric (IF) Interfacing with Host Processor [2]

2.3 Zynq as a hybrid computing platform

Both major FPGA vendors have recently introduced hybrid platforms consisting of high performance processors coupled with programmable logic, aimed at use in systems-on-chip. These architectures 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 maximise bandwidth. In this report, we focus on the

Xilinx Zynq-7000, the block diagram is shown in Fig 2.3.

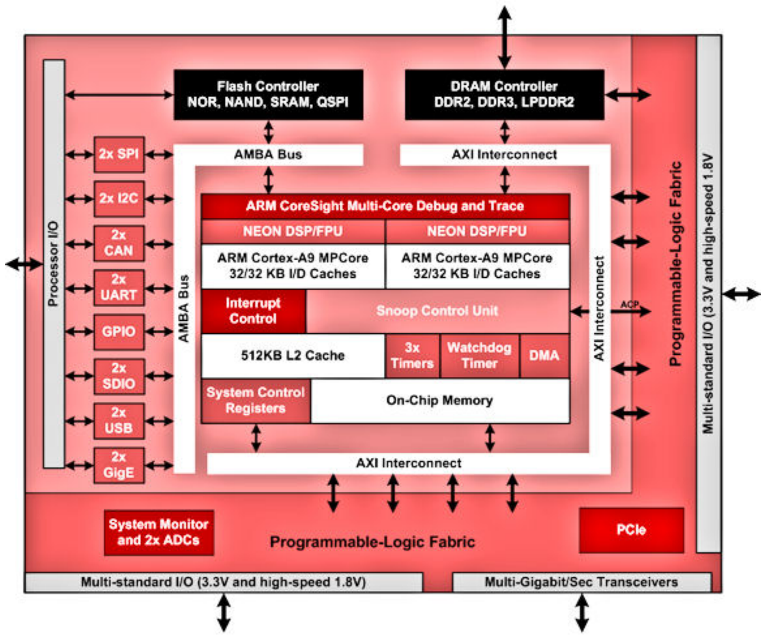


Figure 2.3: Block Diagram of the Hybrid Platform.

The Zynq-7000 contains a dual-core ARM Cortex A9 processor equipped with a double-precision floating point unit, commonly used peripherals, a dedicated hard Direct Memory Access (DMA) controller (PS-DMA) and General interrupt controller (GIC), L1 and L2 cache, on chip memory (OCM) and external memory interfaces. It also 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_ACP – One 64-bit AXI accelerator coherency port (ACP) slave interface for coherent access to CPU memory
- AXI_HP – four 64-bit/32-bit configurable, buffered AXI high performance (HP) slave interfaces with direct access to DDR memory and OCM
- AXI_GP – two 32-bit master and two 32-bit AXI general purpose (GP) slave interfaces

2.4 Communication abstraction using Xillybus

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. The key attraction of communication abstraction techniques is their capability to seamlessly access accelerator for data processing and abstraction of physical interfaces. System designs can incorporate these abstractions at platform level to make use of high speed communication interfaces for communicating with streaming accelerators.

Xillybus provides communication abstraction in which DMA based data transfer mechanism is used between the ARM and FPGA[3]. 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. 2.4 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.

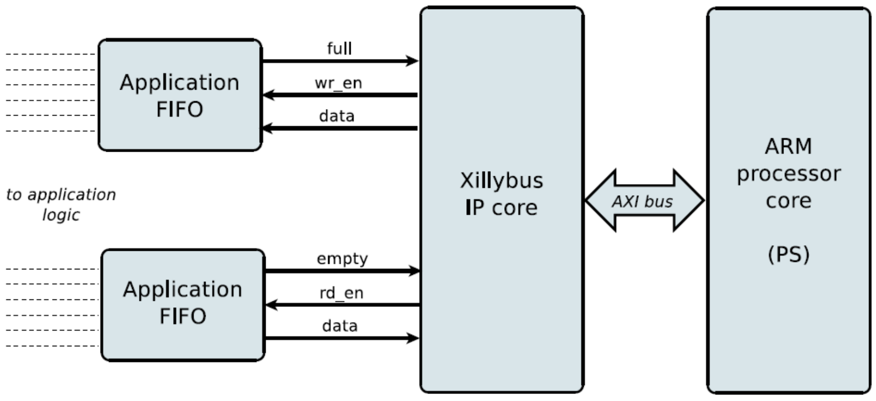


Figure 2.4: Xillybus Block Diagram[3]

Chapter 3

Literature Survey on FPGA Overlays

In the area of coarse grain overlay architectures, the compute the routing logic can either perform the same operation over the time by spatially configuring them, or can loop over a short list of instructions for time-multiplexed execution of kernel operations. In spatially configured functional units based overlays, the compute logic of the overlay are unchanged while a compute kernel is executing while in time-multiplexed functional unit based overlays, the compute logic of the overlay change on a cycle by cycle basis while a compute kernel is executing [16, 31, 32].

3.1 Spatially configured FU based Overlays

Overlays based on Spatially configured functional units normally have a single instruction register within each FU. This type of overlay fits well in a scenario where performance in terms of throughput is a primary objective given the rich logic resources. With the exponential increase of logic density on FPGA devices, it is now possible to accommodate a massive number of FUs on an FPGA which allows to map all of the operations in a compute kernel spatially on the array of FUs. The throughput under this mapping would be one kernel iteration per cycle since the Initiation Interval (II) would be one.

An overlay architecture, referred to as an intermediate fabric (IF) [4], [17] was proposed to support near-instantaneous placement and routing. Standard VPR [33] algorithms were used for placement and routing of compute kernels. It consists of 192 heterogeneous functional units comprising 64 multipliers, 64 subtracters, 63 adders, one square root unit, and five delay elements with a 16-bit datapath and supported the fully parallel, pipelined implementation of compute kernels.

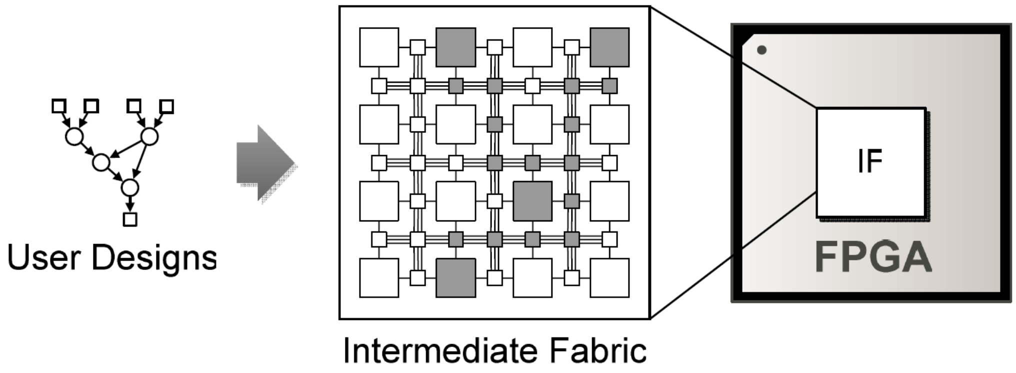


Figure 3.1: Intermediate Fabrics as Island-style Overlay [4].

Unlike a physical device, whose architecture must support many applications, IFs have been specialized for particular domains or even individual applications. Such specialization hides the complexity of fine-grained Commercial Off-the-shelf (COTS) devices, thus enabling fast place and route (700x speedup over vendor tools) at the cost of significant area (34% - 44%) and performance (7%) overhead when implemented on an Altera Stratix III FPGA [17]. However, the IF only achieved an F_{max} of 125 MHz resulting in low throughput for the application benchmarks tested. It consists of 192 heterogeneous functional units comprising 64 multipliers, 64 subtracters, 63 adders, one square root unit, and five delay elements with a 16-bit datapath and supported the fully parallel, pipelined implementation of compute kernels. Area overhead comes into picture mainly because of virtual interconnect logic which comprised of multiplexers based routing. This overhead was reduced by 48% - 54% by reducing flexibility of routing in [34], while improving speed by 24% with a modest routability overhead of 16%. Based on the above mentioned work on IFs, an end-to-end tool flow was presented for FPGA-accelerated scientific computing [35].

Mesh of FU based Overlay was proposed to execute a given DFG by mapping the graph nodes to the FUs and by configuring the routing logic to establish inter-FU connections [5]. Multiple instances of the DFGs are then executed in a pipelined fashion on the overlay to achieve high performance. In addition to integer arithmetic, overlay also used floating point processing elements. It consisted of a 24×16 overlay with a nearest-neighbor-connected mesh of 214 routing cells and 170 heterogeneous functional units (FU) comprising 51 multipliers, 103 adders and 16 shift units. When implemented on an Altera Stratix IV FPGA, the overlay consumed 75% of the total device ALMs, with the routing network consuming 90% of the ALM resource used. An F_{max} of 355 MHz and a peak throughput of 60 GOPS was reported. A placer and router was also developed by customizing VPlace [36] and PathFinder [37], respectively.

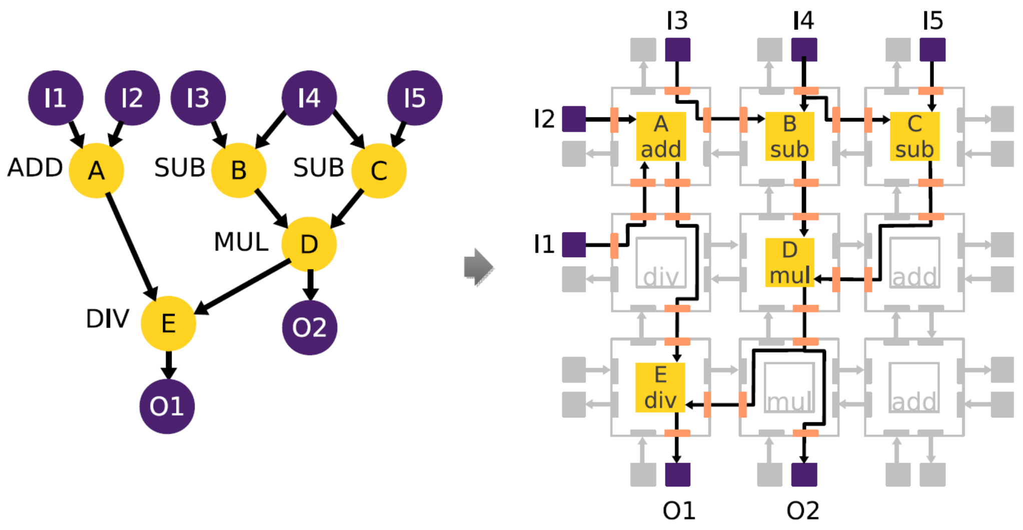


Figure 3.2: Nearest-neighbor connected Mesh of Functional units [5].

Key features are high speed of overlay, Mesh of FUs, elastic pipelines for latency balancing and synchronization, runtime compilation, data driven pipeline units, dynamic and distributed control, No Fmax drop on scaling the overlay size, DFG relocation within VDR, data-driven execution (dynamic triggering of FU on the availability of input data).

DySER [1, 26] was proposed as a coarse grained overlay architecture for improving the performance of general purpose processors. It was originally designed as a heterogeneous array of 64 functional units interconnected with a circuit-switched mesh network and implemented on ASIC. The DySER architecture was then improved and prototyped, along with the OpenSPARC T1 RTL, on a Xilinx XC5VLX110T FPGA [18]. However, due to excessive LUT consumption, it was only possible to fit a 2x2 32-bit DySER, a 4x4 8-bit DySER or an 8x8 2-bit DySER on the FPGA.

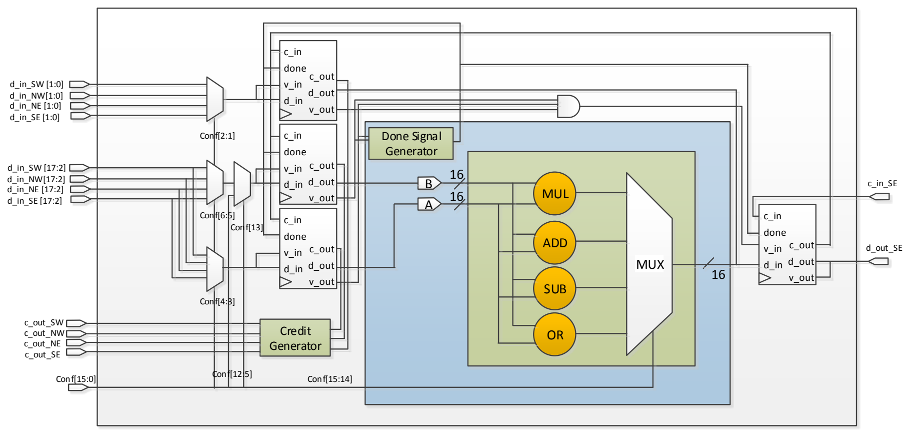
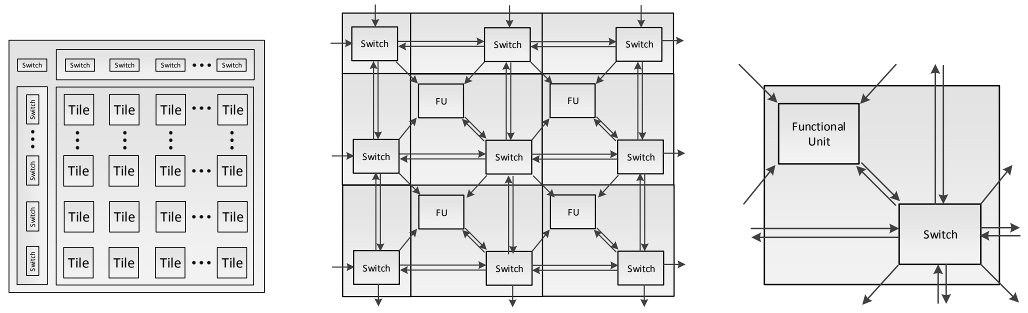


Figure 3.3: DySER functional unit [6].



(a) DySER block diagram. (b) Architecture of a 2×2 DySER. (c) Tile architecture.

Figure 3.4: DySER architecture as Island-style overlay.

3.2 Time-multiplexed FU based Overlays

These overlays ought to have a set of instruction registers within each functional unit (FU) where each FU behaves like a conventional processor core. Unlike the overlay discussed in previous section, these overlays focus on the saving of hardware resources, especially for the kernels having large amount of operations. The basic idea is to time-multiplex the FUs among multiple operations. A single core soft processor like iDEA [8] can be considered a form of an overlay architecture, comprising only one time-multiplexed FU. There are many soft processor designs for FPGAs including RISC processors and vector processors [38]. The array of the FUs are normally interconnected via a nearest neighbor (NN) style programmable interconnect which allows FUs to communicate only to neighbor FUs. One such example is CARBON [31] which supports 256 instructions per functional unit and achieves a frequency of 90 MHz when implemented on Altera Stratix III. One major benefit of using these type of overlays is that well established algorithms and tools can be used for application mapping. The algorithms commonly used are List scheduling [39], Force Directed Scheduling [40] and Modulo Scheduling [41] to schedule the operations of a compute kernel on an array of FUs. Each FU uses the instruction memory to store the assigned multiple instructions and switches during execution.

To address the productivity problem and explore high frequency on FPGA, CGRA architecture overlaid on top of COTS FPGA devices was proposed [16]. Authors also proposed HLS methodology that utilizes SCGRA as an intermediate compilation step. They proposed that new SCGRA design and generation of corresponding bitstream is needed per application domain basis using vendor tools. After that, proposed HLS tool can schedule DFGs on SCGRA and generate instructions, per application basis, which can be merged with SCGRA bitstream to generate final downloadable bitstream for the target FPGA. Authors have designed configurable processing element (CPE) in such a way that it can fetch configuration, from instruction ROM, every cycle. Configuration defines the behavior of the CPE for a particular clock cycle. In comparison with the conventional HLS methodology, the proposed overlay shows 10-100$\times$ reduction in compilation time and achieves up to 21$\times$ speedup in application run time.

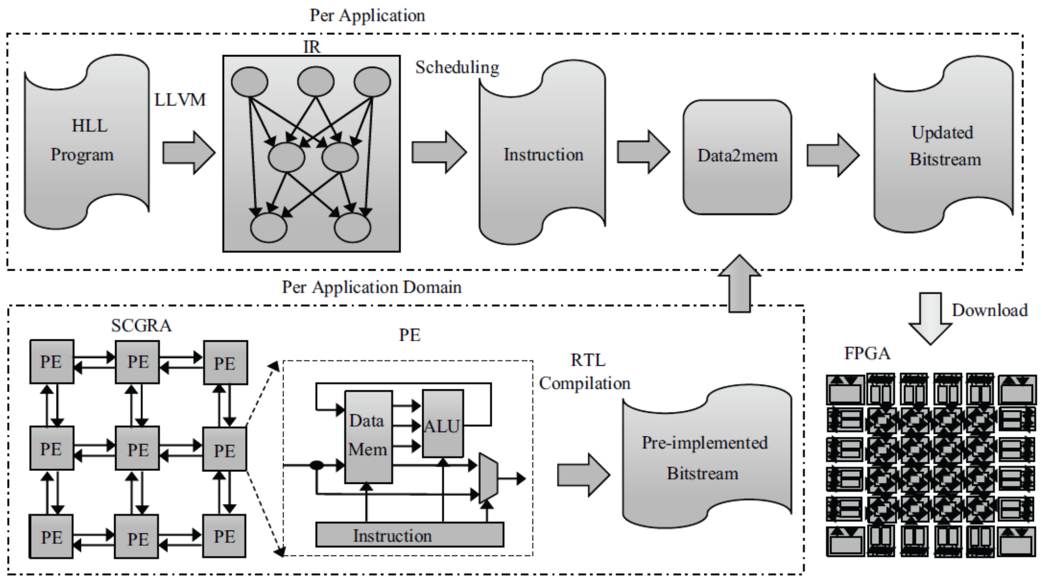


Figure 3.5: Soft CGRA as an FPGA overlay architecture.

To improve the design productivity, another overlay referred as TILT [42] was presented in comparison with Altera OpenCL HLS. In TILT overlay, an array of FUs fetch the data from banked multi-ported memory via the read and write crossbars. Proposed overlay, with an operating frequency of 200 MHz, is able to achieve higher computational throughput by instantiating multiple copies of the TILT core.

Another overlay referred as reMORPH [32], takes advantage of coarse grained modules (CGRMs) such as DSP48E blocks and BRAMs by forming the computing tiles using a mesh-like structure. The DSP block acts as an efficient ALU, which allows the computing tile to achieve a frequency of up to 400 MHz. Similar to SCGRA based HLS design, the bottleneck in reMORPH is the number of BRAMs available on the device to store the list of instructions for each FU.

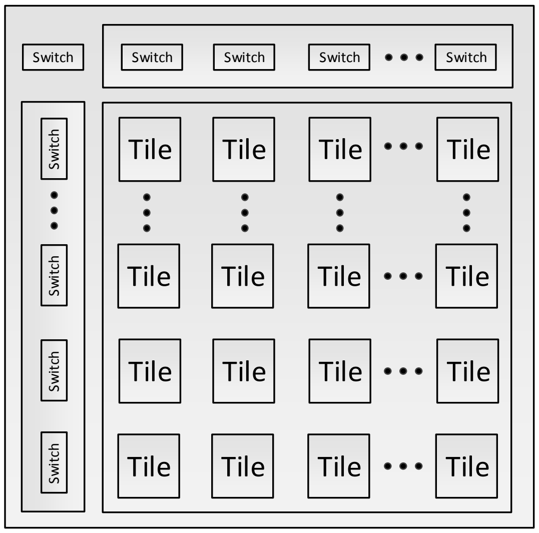
Chapter 4

Analysis of DySER as an Overlay Architecture on Zynq

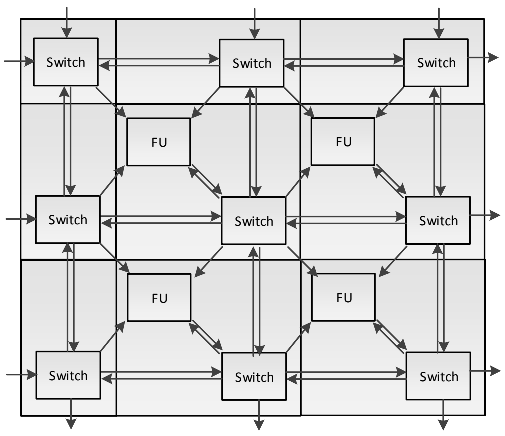
As discussed in the previous chapter, DySER architecture, although relatively efficient from an application mapping perspective, suffered because it was implemented without much consideration for the underlying FPGA architecture. Considering the presence of hard macro blocks, and previous work that has demonstrated how these can be used for general processing at near to their theoretical limits [15, 8], we propose enhancing DySER by using the DSP48E1 found in all modern Xilinx FPGAs to take on most functions of the FU.

4.1 The DySER Architecture

The DySER architecture consists of two blocks, the tile fabric and the edge fabric, where each tile in the tile fabric instantiates a switch and a functional unit (FU), while the edge fabric only instantiates a switch, forming the boundary at the top and left of the tile fabric. The resulting architecture contains I/O ports around the periphery of the fabric, which are connected to FIFOs. A simple 2×2 DySER overlay, consists of four tile instances and five switch instances along the North and West boundaries, resulting in 4 FUs and 9 switches, as shown in Fig. 4.1. Extrapolating this to an $N \times N$ DySER architecture results in N^2 FUs and $(N+1)^2$ switches.



(a) DySER block diagram.



(b) Architecture of a 2×2 DySER.

Figure 4.1: DySER architecture.

4.1.1 DySER Switch

The switches allow datapaths to be dynamically specialized. They form a circuit-switched network that creates paths from inputs to the functional units, between functional units, and from functional units to outputs. Switches in DySER have 5 inputs (4 from neighbour switches and 1 from the functional unit at the North-West direction) and 8 outputs (to all 8 directions). Hence, switches require a 5:1 multiplexer and a state machine for synchronization at each output.

4.1.2 DySER Functional Unit

The functional unit (FU) provides resources for the mathematical and logical operations, and synchronization logic. It receives its input values from the four neighbouring switches and outputs its result to the switch in the south-east direction. The FU consists of programmable computation logic and a state machine as synchronization logic at each input and output of the computation logic. The state machine implements a credit-based flow-control protocol to enable receiving of inputs asynchronously at arbitrary times from the FIFO interfaces.

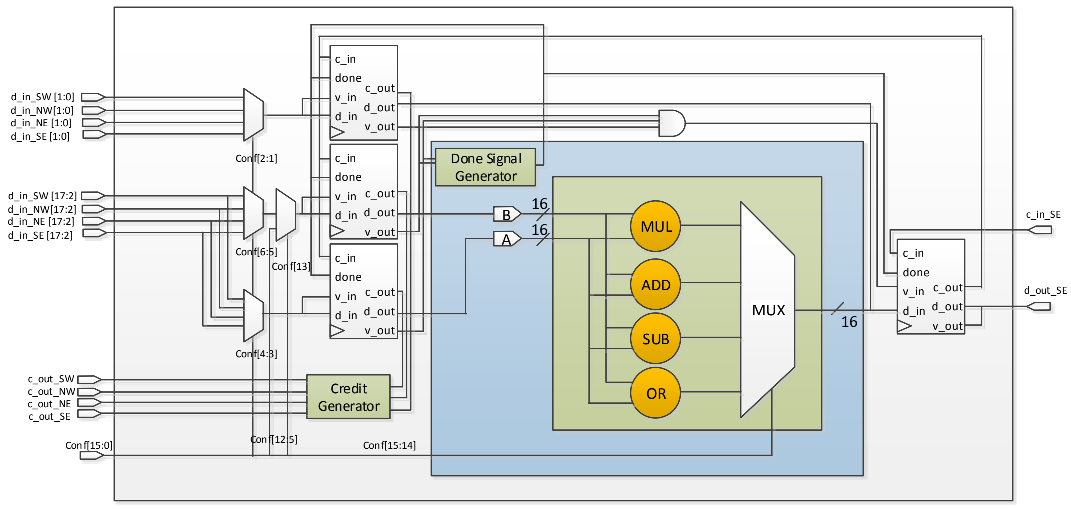


Figure 4.2: Functional unit architecture.

The operators in the FU can be selected according to application requirements. We choose four operators: Add, Sub, Mul and OR in the FU, as shown in Fig. 4.2, to map the benchmarks from [43]. The benchmark characteristics are shown in Table 4.1. Benchmarks, where a small code region dominates the runtime, and where computation can easily be scheduled, are taken from [43]. These benchmarks mimic the workloads of the PARBOIL suite.

Table 4.1: Benchmark Characteristics

No.	Benchmark	Add	Sub	Mul	OR	Total
1.	fft	3	3	4		10
2.	kmeans	7	8	8		23
3.	mm	7		8		15
4.	mri-q	3		6	1	10
5.	spmv	6		8		14
6.	stencil	10	2	2		14
7.	conv	8		8		16
8.	radar	6		2		8

The original DySER FU was implemented using Xilinx ISE 14.6 targeting a Xilinx Zynq XC7Z020. The FU consumes 49 Slices (148 LUTs, 66 FFs) and 1 DSP48E1 block, with a critical path of 6.7 ns. Hence the maximum operating frequency of the FU is 150 MHz. Fig. 4.3 shows the physical mapping of the FU to the FPGA fabric.

While synthesizing, the tool infers a DSP block for multiplication. The remainder of the operations and the multiplexer in the compute logic are mapped to 17 Slices (57 LUTs). State machines and input selection multiplexers are mapped to 32 Slices (91 LUTs and 66 FFs). After integrating the FU into the DySER tile and implementing it on the FPGA fabric, we found that the critical path in the DySER Tile is the same as the critical path of the FU (6.7 ns), and hence the FU limits the performance of the DySER tile.

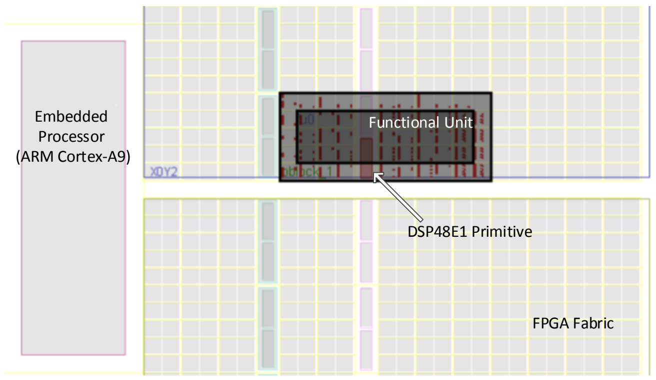


Figure 4.3: Physical mapping of functional unit on FPGA.

4.2 DSP Block Based DySER

Building on the advantages of hard DSP macros for implementing high speed processing elements, we examine the use of the Xilinx DSP48E1 primitive as a programmable FU in DySER targeting data-parallel compute kernels. Despite the fact that the original FU uses a DSP block for multiplication, it does not fully exploit the performance advantage of the DSP block. Since the DSP48E1 can be dynamically configured and used for operations required by the FU, we show that an area and performance efficient FU can be built by making use of DSP block as an ALU, instead of just as a multiplier, and enabling the internal pipeline registers of the DSP block.

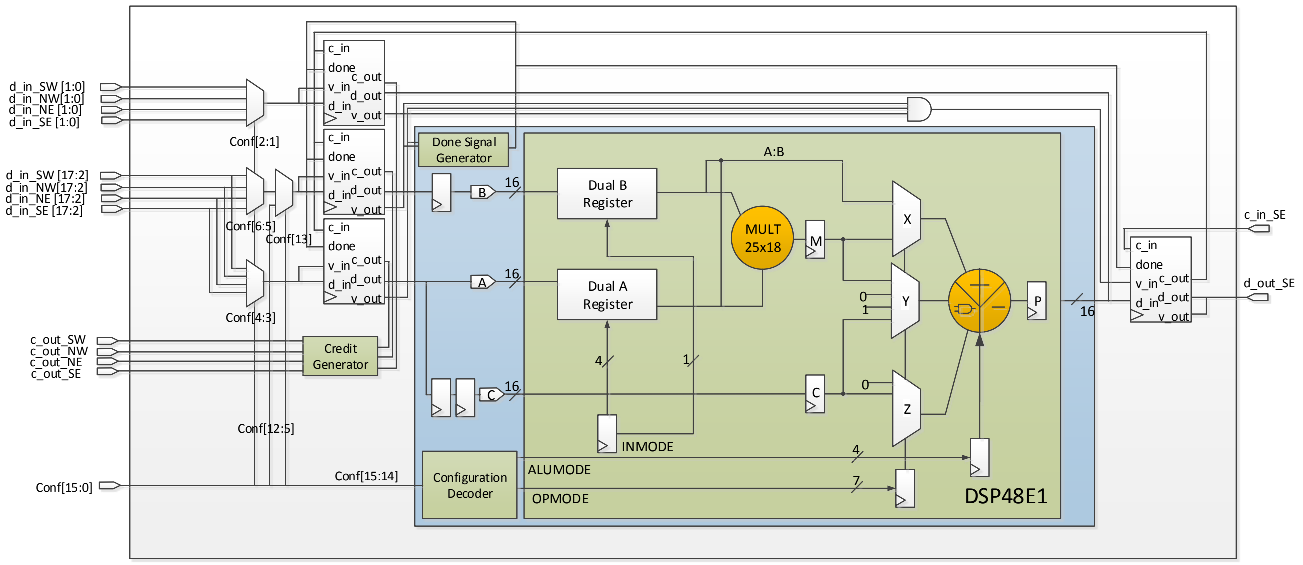


Figure 4.4: DSP48E1 based functional unit architecture.

4.2.1 DSP48E1 Based Functional Unit

We use the DSP48E1 primitive, as shown in Fig. 4.4, to implement computation logic in the modified functional unit. The DSP48E1 primitive has a pre-adder, a multiplier, an ALU, four input ports for data, and one output port P, as shown in Fig. 4.4, and can be configured to support various operations such as multiply, add, sub, bitwise OR, etc. These functions are determined by a set of dynamic control inputs that are wired to configuration registers. The DSP48E1 primitive is directly instantiated providing total control of the configuration of the primitive. This allows us to maximize the compute kernel throughput and achieve a high FU frequency by operating the DSP48E1 at its maximum frequency.

We enable all of the pipeline stages of the DSP48E1 primitive. The redesign of the DySER functional unit replaces the original compute unit (CU), shown in Fig. 4.2, with the fully pipelined DSP48E1 primitive, along with modifications to the done signal generation logic and configuration decoding logic, as shown in Fig. 4.4. The two inputs from the FU (to the CU) are connected to the three ports of the DSP48E1 primitive, as shown in Fig. 4.4. The FU configuration register includes 2 bits for operation selection with the other 14 bits for constant and input multiplexers.

Additionally, we require three 16-bit registers at the DSP input ports (as shown in Fig. 4.2), consuming 48 FFs to balance the internal pipeline stages of the DSP block. Table 4.2 shows the DSP48E1 configuration settings required for each operation. Inmode remains same for all of the operations and hence we hard-code it to 00000.

Table 4.2: DSP48E1 configuration for each operation

Operation	ALUMODE	OPMODE	INMODE
ADD	0000	011 0011	00000
SUB	0011	011 0011	00000
MUL	0000	000 0101	00000
OR	1100	011 1011	00000

4.2.2 Analysis of Performance Improvement

We analyze the performance improvement of the FU in terms of frequency and resource usage. The DSP48E1 based FU consumes 37 Slices (116 LUTs, 117 FFs) (25% less than the original FU) and 1 DSP block. Apart from obvious area savings, the strategy of using a fully pipelined DSP block as the computational part of the FU also improves overall timing performance. The FU has a critical path of just 2.7 ns, resulting in a maximum frequency of 370 MHz, which is 2.5× that of the original FU. Fig. 4.5 shows the physical mapping of functional unit onto the FPGA fabric.

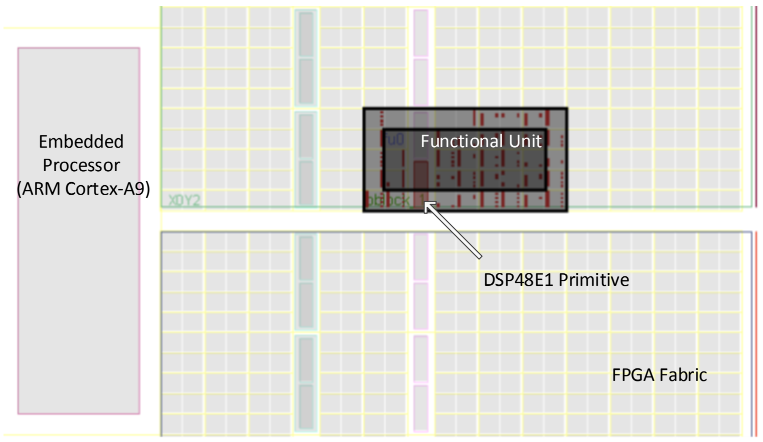


Figure 4.5: Physical mapping of enhanced functional unit.

Since a hard primitive is used for the implementation of CU operations, only minimal additional circuitry is implemented in the logic fabric which consists of configuration decoding logic, three 16-bit balancing registers and done signal generation logic. All of this additional circuitry is mapped to 10 Slices (25 LUTs and 51 FFs). State machines and input selection multiplexers are mapped to 27 Slices (91 LUTs and 66 FFs).

By integrating the enhanced FU into the DySER tile and implementing it on the FPGA fabric, we found that the critical path of the switch, which is 5.3 ns, now limits the performance of the DySER tile. Fig. 4.6 shows the physical mapping of the DySER Tile to the FPGA fabric. It is clear that the major area overhead in DySER is due to significant resources consumed in the switch implementation. The switch consumes 251 Slices (995 LUTs and 325 FFs) and hence the whole tile consumes 288 Slices (1118 LUTs and 447 FFs). The largest source of area overhead comes from the multiplexing logic in the switch which can be minimized by using techniques mentioned in [34, 44].

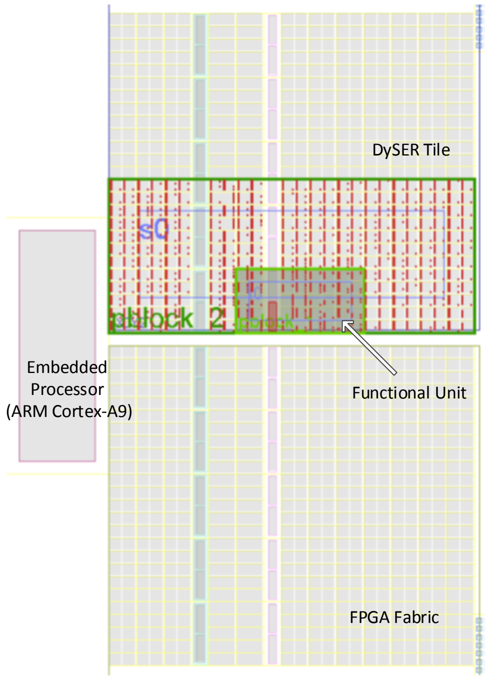


Figure 4.6: Physical mapping of the DySER Tile on FPGA.

4.3 Kernel Mapping on DySER

As discussed previously, an $N \times N$ DySER overlay incorporates N^2 Tiles in the tile fabric and $2N+1$ switches in the edge fabric. Hence, theoretically a 6×6 DySER overlay is the largest that can fit on the Zynq-7020. Fig. 4.7 shows the mapping of kernels on DySER, previously described in [7]. A fixed configuration 5×5 FU array can be used to implement all of the compute kernels without flexible routing. This consumes 5.5% LUTs, 2.7% FFs, 6.9% Slices and 11.4% DSP blocks, while a fully functional 5×5 DySER overlay consumes 63.7% LUTs, 12.6% FFs, 92.4% Slices and 11.4% DSP blocks.

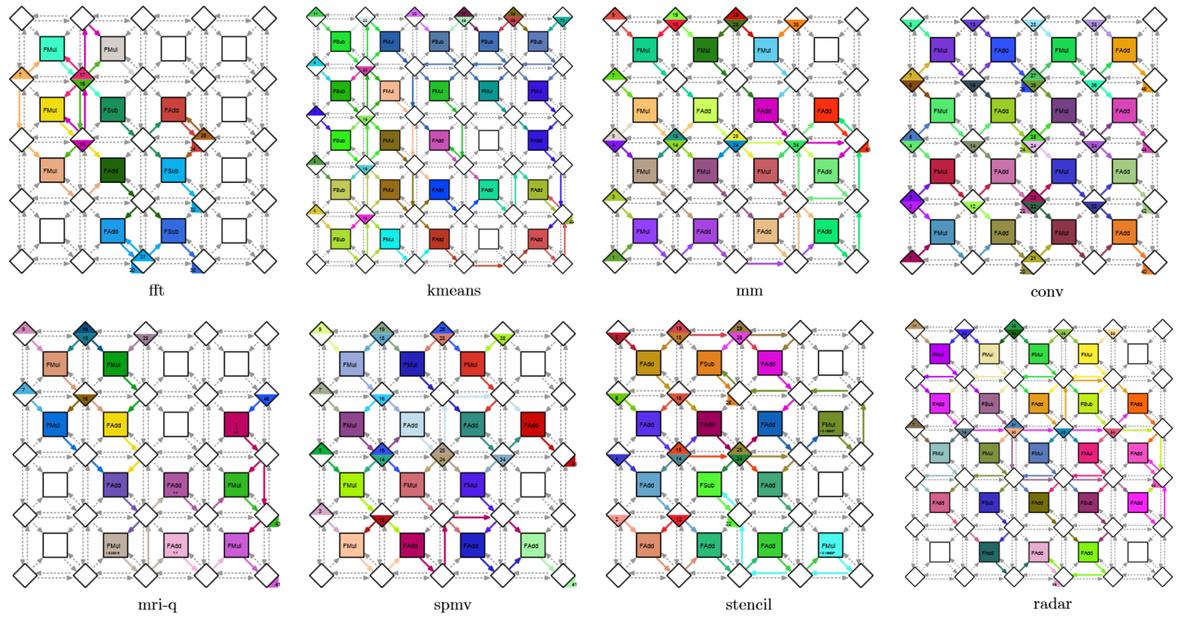


Figure 4.7: Mapping of Kernels on DySER Architecture.[7]

We assess the overhead of the programmability in a similar way to [5]. The programmability overhead is the ratio of the DySER overlay resources to those of the fixed configuration array of FUs that comprise it. Hence, a 5×5 DySER overlay can be used to implement all of the compute kernels with a programmability overhead of 11× more LUTs, 5× more FFs, and 13× more Slices.

4.4 Summary

We have presented an enhancement to the DySER coarse-grained overlay that uses the Xilinx DSP48E1 primitive to implement most of the functional unit, improving area and performance. We show an improvement of 2.5× in frequency and a reduction of 25% in area compared to the original functional unit design. We have shown that a more architecture-oriented approach to designing the FU enables it to be small and fast and exposes the significant overhead of the flexible routing. As a result the routing for the coarse grained array becomes the limiting factor. An area and performance efficient interconnect architecture is necessary for improving the performance of the overlays. Another way to build area efficient overlay is to use a linear array of time-multiplexed functional units. In the next chapter, we present an overlay designed using a linear array of time-multiplexed functional units which can be used to host arbitrary size DFG of a compute kernel considering $N \geq D$ holds true, where N is the number of functional units in the overlay and D is the depth of the DFG. The proposed overlay would enable us to explore resource sharing for larger computer kernels.

Chapter 5

Overlay based on Time-multiplexed FU

In the previous chapter, We have discussed DySER overlay and shown that a more architecture-oriented approach to designing the FU enables it to be small and fast and exposes the significant overhead of the flexible routing. Most of the existed overlays (including DySER) are spatially configured, having fixed configuration for a kernel during execution. In other words, each functional unit has fixed operation during execution and the required number of functional units depends on the number of operations in the kernel. One way to build area efficient overlay is to time-multiplex the functional units among kernel operations. In order to take advantage of this approach, we design a new overlay architecture using a linear array of time-multiplexed functional units which can be used to host arbitrary size DFG of a compute kernel considering N$\geq$D holds true, where N is the number of functional units in the overlay and D is the depth of the DFG. We execute kernel operations in a stage of ASAP scheduled DFG on one time-multiplexed FU. Hence, each stage gets mapped to one FU and after performing all operations in the stage on the FU, the processed data gets transferred to the next FU via FIFO. Compared to a spatially configured overlay where II is generally one, in the proposed overlay, DFG width determines the II for the DFG execution. The proposed overlay would enable us to explore resource sharing for larger computer kernels at the cost of reduced II.

5.1 Time-multiplexed FU based on DSP Block

We first present the time multiplexed functional unit to execute the kernel operations sequentially by exploiting cycle by cycle reconfiguration capability of the DSP block. The design of the FU is inspired by iDEA[8] processor as shown in Fig. 5.1. The processor has a basic, yet comprehensive enough, instruction set for general purpose applications. By limiting the addition of hardware modules such as branch prediction, control complexity was minimized in iDEA processor. iDEA processor runs at about double the frequency of MicroBlaze while occupying around half the area.

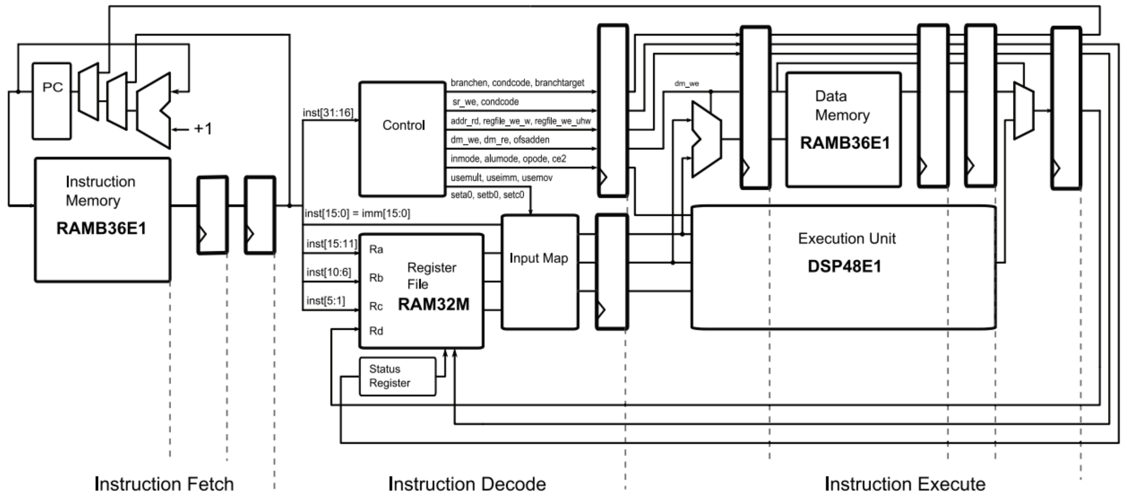


Figure 5.1: iDEA processor block diagram.[8]

There are, in total, 6 stages in the processor execution pipeline with a latency of 1-clock cycle per stage. The full 3-stage pipeline is enabled for the DSP48E1 primitive which is used as a high speed execution engine. The DSP48E1 primitive has a pre-adder, a multiplier, an ALU, four input ports for data, and one output port P, as shown in Fig. 5.2, and can be configured to support various operations such as multiply, add, sub, bitwise OR, etc. These functions are determined by a set of dynamic control inputs that are wired to configuration registers.

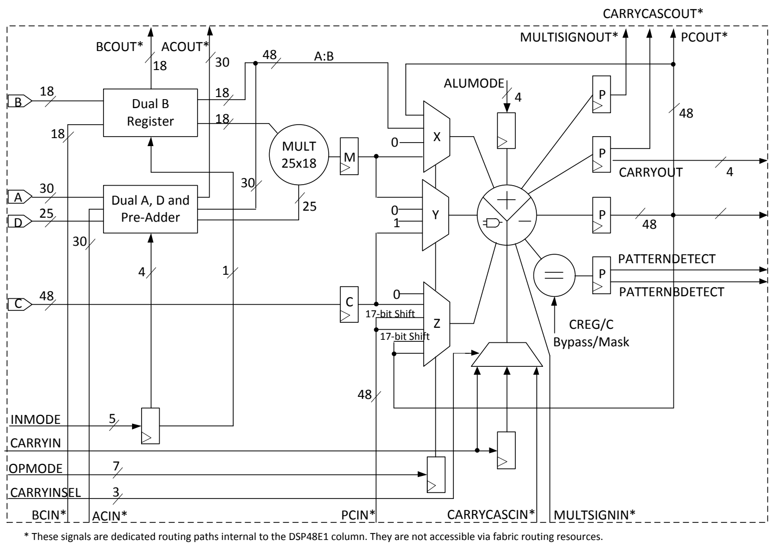


Figure 5.2: DSP48E1 architecture.

5.1.1 Architecture of Proposed Functional Unit

The main components of the FU are DSP block, a small instruction memory, input map logic for DSP block and a register file as shown in Figure 5.3. As the FU needs to work as a data flow processing element, data memory is not required in the functional unit. According to the figure, the 24-bit instruction memory is divided into 4 sections: operation code, destination address, source address of first operand and source address of second operand. In the operation code, the most significant bit (MSB) determines whether the output of FU or the immediate data should be stored into the register file, while the following 5 bits are used to define up to 32 different computational operations. Currently, there are 6 operations in total, such as ADD, SUB, MUL, ADDI, SUBI, and MULI. The regular inputs of DSP48E1 are coming from the register file and the immediate input is coming from the instruction memory directly. Since there is only one FU to process all the operations of the kernel, instructions need to be executed one by one and the result of each step should be restored into the register file. As we enable all the built-in registers inside the DSP48E1, it takes 3 cycles to finish each instruction.

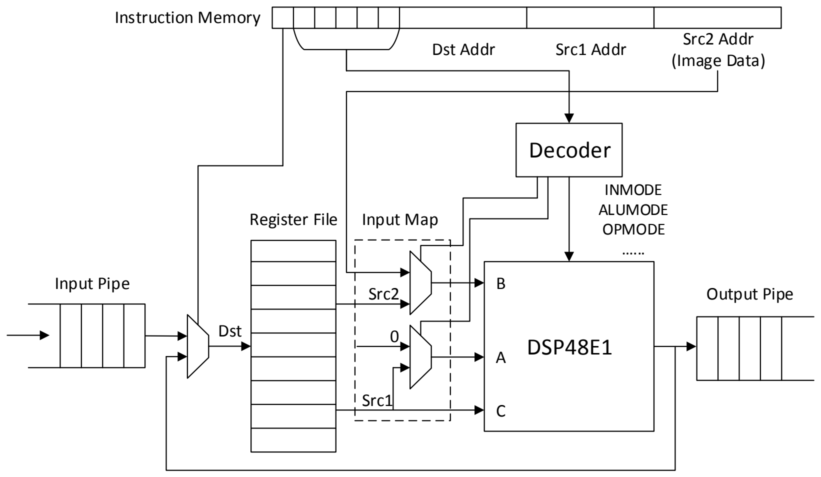


Figure 5.3: Time-multiplexed FU Block Diagram.

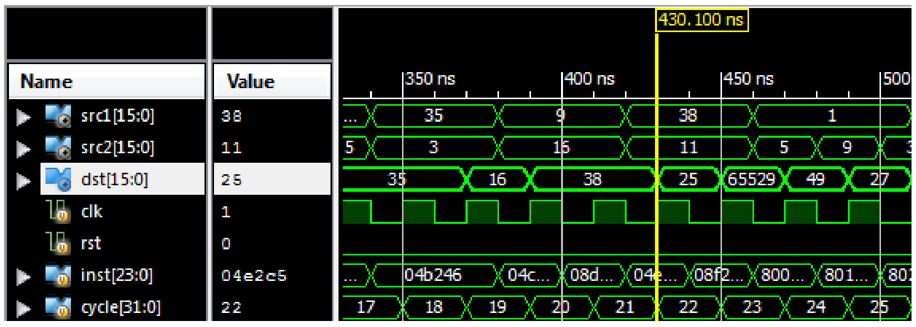
5.1.2 Kernel Execution

To verify whether the design of the FU works properly and test its performance, we write the test bench for each kernel, and specially show the instructions of kernel FFT in Figure 5.4.

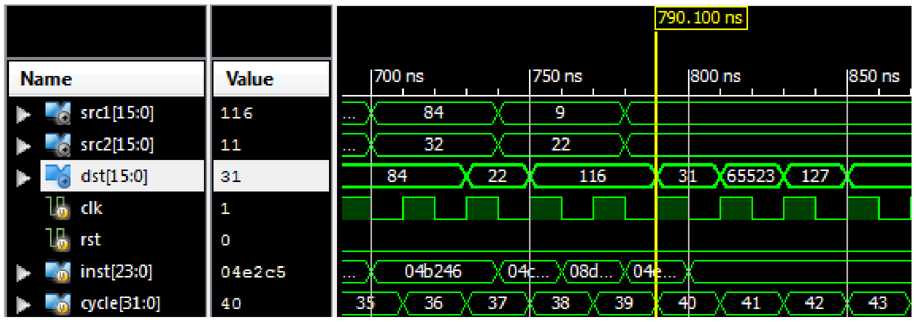
```
//----------------------------- FFT ------------------------------//
#20;  inst    = 24'b100000_000000_000000_000001; //Ldi R0, 1
#20;  inst    = 24'b100000_000001_000000_000011; //Ldi R1, 3
#20;  inst    = 24'b100000_000010_000000_000101; //Ldi R2, 5
#20;  inst    = 24'b100000_000011_000000_000111; //Ldi R3, 7
#20;  inst    = 24'b100000_000100_000000_001001; //Ldi R4, 9
#20;  inst    = 24'b100000_000101_000000_001011; //Ldi R5, 11
#20;  inst    = 24'b000011_000110_000001_000000; //Mul R6, R1, R0
#20;  inst    = 24'b000011_000111_000010_000000; //Mul R7, R2, R0
#20;  inst    = 24'b000011_001000_000011_000001; //Mul R8, R3, R1
#20;  inst    = 24'b000011_001001_000011_000010; //Mul R9, R3, R2
#20;  //inst    = 24'b000000_000000_000000_000000; //NOP
#20;  inst    = 24'b000010_001010_001000_000111; //Sub R10, R8, R7
#20;  inst    = 24'b000001_001011_001001_000110; //Add R11, R9, R6
#20;  //inst    = 24'b000000_000000_000000_000000; //NOP
#20;  inst    = 24'b000001_001100_000100_001010; //Add R12, R4, R10
#20;  inst    = 24'b000010_001101_000100_001010; //Sub R13, R4, R10
#20;  inst    = 24'b000001_001110_001011_000101; //Add R14, R11, R5
#20;  inst    = 24'b000010_001111_001011_000101; //Sub R15, R11, R5
```

Figure 5.4: Snapshot of Test Bench for Kernel FFT

As shown in Figure 5.4, there are 18 instructions to be executed in this kernel, including 2 extra NOPs. These NOPs are added due to the fact the an instruction takes 3 cycles to finish. If the previous processing is still running, i.e., the required register is not ready for current instruction. It should keep waiting until there are valid data in the registers. While the system can only process instructions in series, it is time consuming to use only one FU to implement the kernel.



(a)



(b)

Figure 5.5: Simulation Results of kernel execution on FU

Simulation results for this particular kernel is displayed in Figure 5.5. Assuming that the host is providing data to the input pipe at every clock cycle, ready signal is used to control the read enable signal of the input pipe. Therefore, different sets of inputs are hold in the FIFO when the functional unit is occupied with the processing of the previous set of input, and the next set of input is allowed to enter the functional unit when it is in idle state. As we can see from the simulation results, result of the kernel for two consecutive input sets is generated correctly at clock cycle 22 and 40, respectively. It means the II of the kernel execution is 18, which limits the throughput. Hence we design a linear array of these time-multiplexed functional units which is discussed in the next section.

5.2 Linear Array of Functional Units as Overlay

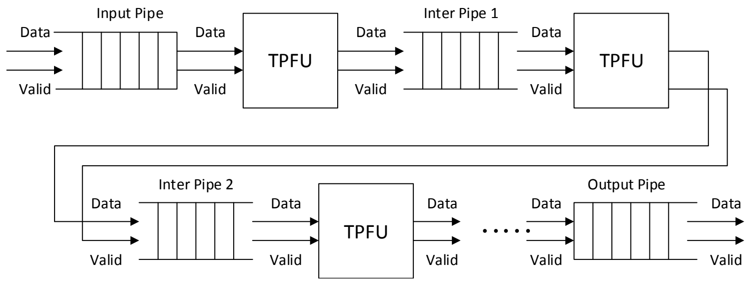


Figure 5.6: Linear Array of time-multiplexed FUs

5.2.1 Architecture

Although a single time-multiplexed functional unit works properly with the appropriate handshake between input pipe and computational logic, its performance is limited by the serial processing and extra inserted NOPs. To resolve this issue, we design a linear array of these time-multiplexed functional units (as shown in Figure 5.6) where each FU can process one complete stage of a ASAP scheduled DFG of the kernel.

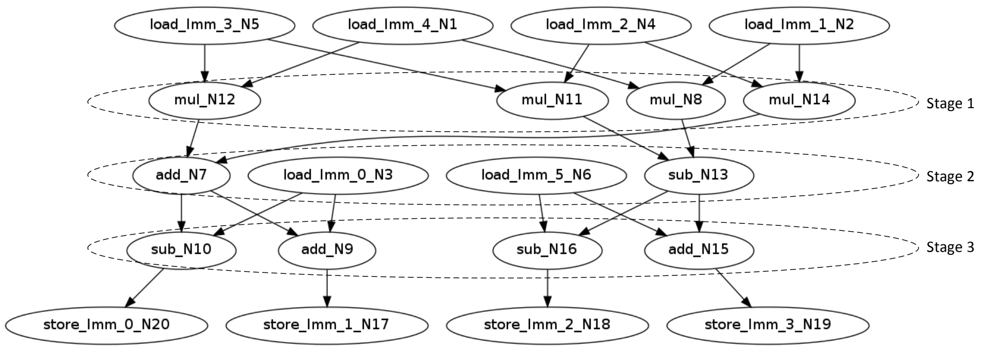
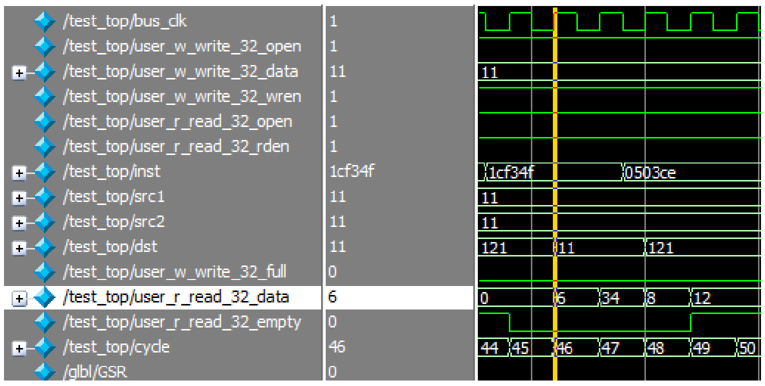
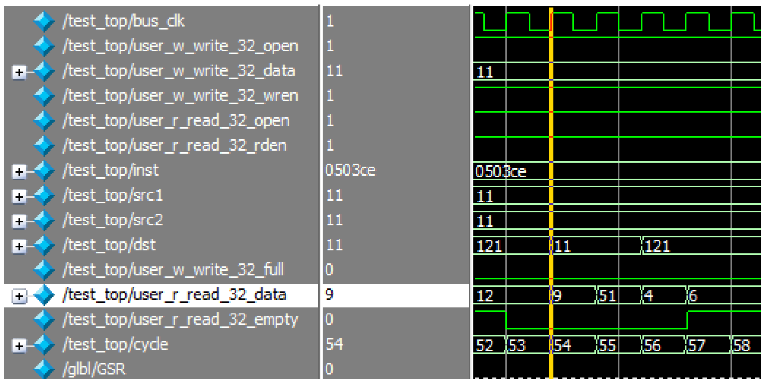


Figure 5.7: Nodes Clustering for Kernel FFT

In this design, one FU is responsible for executing one complete stage instead of one node of the data flow graph (DFG). linear array of these FUs can be used to host arbitrary size DFG of a compute kernel considering N≥D holds true, where N is the number of functional units and D is the depth of the DFG. Compared to a spatially configured overlay where II is generally one, in the proposed overlay, DFG width determines the II for the DFG execution. Hence the performance is no longer limited by the No. of operations, but it is limited by the DFG width. In other words, the required number of functional units and the II for a kernel can be determined by the depth and width of the ASAP scheduled DFG, respectively. As shown in Figure 5.7, stage 1 clusters all the operations in N12, N11, N8, and N14 which represent multiplying N5 with N1, multiplying N5 with N4, multiplying N1 with N2, and multiplying N4 with N2.


```
/test_top/bus_clk                1
/test_top/user_w_write_32_open   1
/test_top/user_w_write_32_data   11
/test_top/user_w_write_32_wren   1
/test_top/user_r_read_32_open    1
/test_top/user_r_read_32_rden    1
/test_top/inst                   1cf34f
/test_top/src1                   11
/test_top/src2                   11
/test_top/dst                    11
/test_top/user_w_write_32_full   0
/test_top/user_r_read_32_data    6
/test_top/user_r_read_32_empty   0
/test_top/cycle                  46
/glbl/GSR                        0
```

(a)


```
/test_top/bus_clk                1
/test_top/user_w_write_32_open   1
/test_top/user_w_write_32_data   11
/test_top/user_w_write_32_wren   1
/test_top/user_r_read_32_open    1
/test_top/user_r_read_32_rden    1
/test_top/inst                   0503ce
/test_top/src1                   11
/test_top/src2                   11
/test_top/dst                    11
/test_top/user_w_write_32_full   0
/test_top/user_r_read_32_data    9
/test_top/user_r_read_32_empty   0
/test_top/cycle                  54
/glbl/GSR                        0
```

(b)

Figure 5.8: Simulation Results of kernel execution on linear array

5.2.2 Kernel Execution

Take kernel FFT for example, three FUs are required to execute different instructions in parallel according to its DFG, and the simulation results generated by ModelSim are displayed in Figure 5.8. The four outputs of this kernel are generated one by one from clock cycle 46, while next series of outputs are valid from clock cycle 54, which means the II is only 8 in this case. Noticed that if this kernel is implemented by the single FU, the No. of instructions would be 18 in total, which indicates a much higher value of initial interval.

5.3 Summary

In this chapter, a novel overlay architecture based on linear array of time-multiplexed functional units is proposed for kernel execution. Compared to spatially configures overlays, proposed overlay is much more flexible as the FUs can execute one complete stage of a ASAP scheduled DFG. Thus, the required number of FUs is reduced at the cost of processing throughput since spatially configured (SC) overlays can provide II of one. We compare II of kernel execution for different architecture as shown in Table 5.1.

Table 5.1: II comparison for different architectures

Kernels	II for single FU	II for proposed overlay	II for SC overlay
fft	18	8	1
mm	39	17	1
radar	19	11	1
spmv	31	17	1

Chapter 6

Experiments

6.1 Introduction

In this chapter, we present experiments for interfacing accelerators (HLS generated RTL of kernel or overlay) within Zynq using Xillybus and evaluating the performance of HLS generated fully parallel and pipelined RTL implementations, ARM processor and the proposed overlay for a set of compute kernels. The goal is to measure the performance gap between the accelerator and the ARM processor.

We first describe the infrastructure of Xillybus followed by the characterization. For characterization purpose, we use a loopback application which writes an array of data to Xillybus pipe and reads it back. Considering that the operating frequency is 100 MHz, the theoretical bound of write and read bandwidth is 100 MS/s (Million samples per second). Hence the theoretical bound on round trip bandwidth would be 50 MS/s. Xillybus allows to generate multiple input/output streaming interfaces sharing the bandwidth of ACP interface (configured as 32-bit interface running at 100 MHz) on the Zynq platform. We verified it by creating two interfaces for write and read (using Xillybus IP core factory) and measured the round trip time. We observed that the ACP interface on the Zynq platform can be time-multiplexed among multiple streaming interfaces, since there is only one ACP interface available. Hence the compute kernels requiring multiple input and output interfaces have to share the ACP interface in a time-multiplexed fashion which can limit the performance of the accelerator. A fully pipelined accelerator (running at 100 MHz) having N input interfaces can process 100 M-Samples per second, considering no time sharing of I/O interfaces. In this chapter, we generate RTL implementation of compute kernels (having multiple I/O interfaces) using Vivado HLS, integrate the kernels within Zynq using Xillybus and measured the overall performance of the kernel execution on the platform. We compare the performance of HLS generated fully parallel and pipelined accelerators with ARM processor for a set of kernels.

6.2 Xillybus

There are various solutions for interfacing accelerators with a host processor, such as some PCle based solutions (DyRACT FPGA) and some AXI based solutions (Xillybus). DyRACT is a DMA based streaming architecture which enable high throughput data transfer between host and accelerator. By combining high data throughput and fast reconfiguration, it becomes feasible to implement software applications with dynamically reconfigurable hardware accelerators. However, DyRACT currently supports PCIe based platforms.

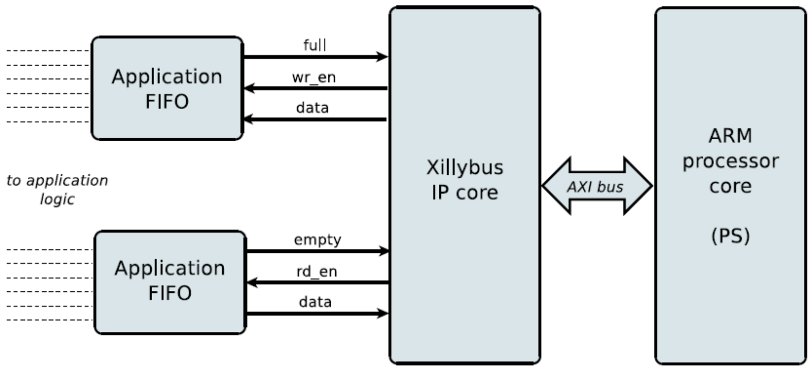


Figure 6.1: Xillybus Simplified Block Diagram

Xillybus is a portable, simple to use DMA based data transfer solution which provides a simple abstraction of communication interfaces on Zynq. It offers a convenient way for data acquisition or accelerator interfacing. In the demo bundle, Xillybus IP core and drivers are offered, so that no more than basic programming skills and logic design skills are needed to complete the integration of an accelerator within platform. It can be used within Linux or Windows. In our case, all the experiments with Xillybus are running in Linux environment.

As shown in Figure 6.1, our application logic (e.g. HLS generated kernel or an Overlay) is connected to the Xillybus IP core through customizable standard FIFOs with empty/full signals and write/read enable signals to facilitate the so-called easy data transport. The application logic grab inputs through the input FIFOs which are connected with Xillybus interface when the FIFO is not empty, after processing then fill the result data into the output FIFOs. Then Xillybus IP core will automatically detect that data is available for transmission in the FIFOs other end. Soon, the data in the FIFO can be sent to the host, making it visible by the host program.

On the host side, each FIFO stream is mapped to a pipe-like device file by a common Xillybus driver. When executing the device file (we called pipe), data flows smoothly between the file handler opened by the host program and the FIFOs. The Xillybus IP core provides platform dependent clock for the design. Xillybus is very simple-to-use as if there is a requirement of modifying accelerator connections or the accelerator itself, bitstream can be replaced just by uploading the bitstream on a specific file location in root file system and rebooting the system. Then all the connections and the accelerator can be changed immediately after reboot. There is also an online interface provided by the Xillybus team called IP Core Factory which can be used to customize and download user-defined Xillybus IP based on specific requirements. A virtually arbitrary number of pipes, direction, data width and expected bandwidth can be defined, having attributes best meeting the needs of a specific application.

The FPGA demo bundle for Zynq (shown in Fig. 6.2) is used as a base for all of our experiments. 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 required netlists. The bit stream is generated from top level HDL file and then downloaded to the FPGA. Demo bundle contains a default set of device nodes (single in and single out). The number of nodes can be increased or

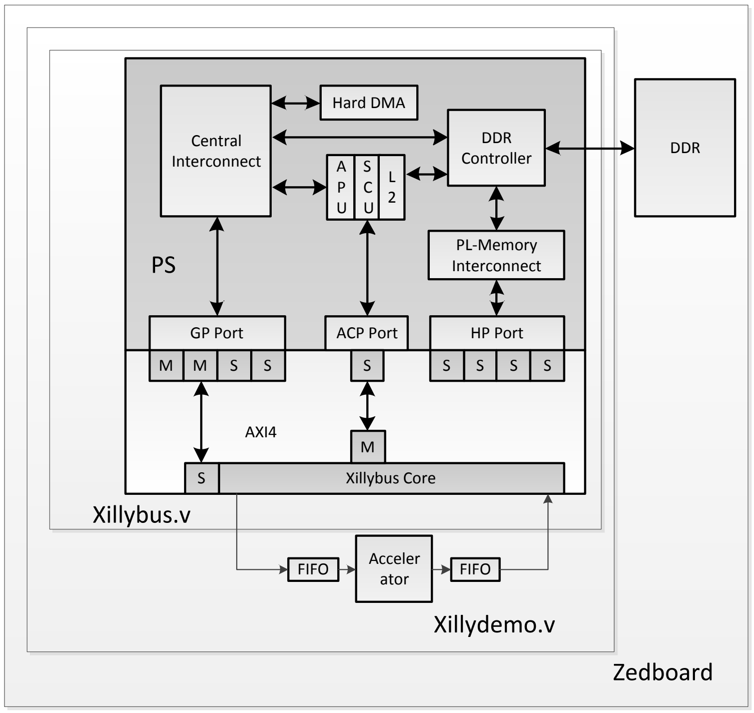


Figure 6.2: Xillybus infrastructure for Zedboard

customized as per design requirements through IP Core Factory. 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 HP port for VGA and ACP port for Xillybus IP. The Xillybus implementation consists of a 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 is abstracted from the user and data is passed to and from it using API. In order to exploit the parallelism present in hardware, one thread can send data from the processor as soon as it is available and another thread can collect the data back.

6.3 Xillybus Characterization

6.3.1 Single pipe loopback

First experiment we did for Xillybus characterization is the single pipe loopback. This is provided in the Xillybus initial demo buddle for Zynq, therefore we can directly use the original file to generate the bit stream and download to the FPGA. The initial Demo bundle contains FIFOs connected in a loopback fashion thereby causing the Xillybus core to act both as a source and sink. As shown in Figure 6.3, the input (user_w_write) and output (user_r_read) pipes connect to the same FIFO to form a loopback circuit.

```
// 32-bit loopback
fifo_32x512 fifo_32
(
   .clk(bus_clk),
   .srst(!user_w_write_32_open && !user_r_read_32_open),
   .din(user_w_write_32_data),
   .wr_en(user_w_write_32_wren),
   .rd_en(user_r_read_32_rden),
   .dout(user_r_read_32_data),
   .full(user_w_write_32_full),
   .empty(user_r_read_32_empty)
)

assign user_r_read_32_eof = 0;
```

Figure 6.3: Xillybus 32-bit loopback FIFO connection

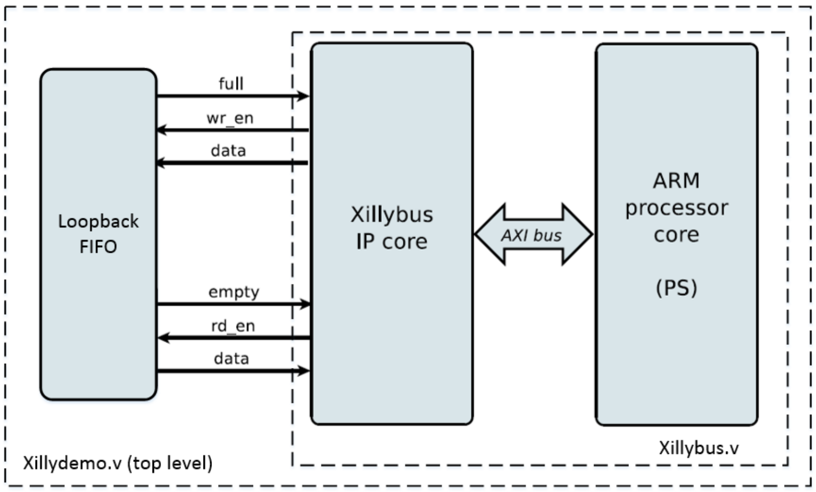


Figure 6.4: Xillybus initial demo bundle block diagram

Figure 6.4 shows the structure of the Demo Bundle. Xillybus.v build up the interface between Xillybus IP core and ARM processor core by using AXI bus. And the top level Xillydemo.v wrapper file present in the Xillybus bundle is used to integrate our application logic. In this specific case, there is no application logic involved, data from Xillybus IP core will just loopback through a FIFO. After the bit stream is download and configured on the FPGA, the system now is ready to go. We need a host program to do the round trip time measurement. The single thread code is shown in Figure 6.5.

For the single thread program, if large amount of data is required for transmission, the reading operation will wait until all the data is written. Therefore, there is a drawback of the long communication latency. In order to explore the best use of Xillybus IP core, we use multithreading to improve its performance by implementing parallelism. In order to take full advantage of the capabilities provided by threads, we are using Pthreads (POSIX threads) which is a light weight, efficient communications and data exchange standard. The multithread code is shown in Figure 6.6.

Table 6.1: Single Pipe Loopback results

	Single-threading		Multi-threading	
No. Samples	Round Trip Time(us)	Troughput (KS/s)	Round Trip Time(us)	Troughput (KS/s)
256	129	1984.5	811	315.7
512	129	3969.0	811	631.3
1K	129	7938.0	811	1262.6
2K	147	13932.0	848	2415.1
4K	166	24674.7	848	4830.2
8K	258	31751.9	903	9072.0
16K	406	40354.7	1014	16157.8
32K	756	43343.9	1235	26532.8
64K	1567	41822.6	1880	34859.6
128K	2820	46479.4	3207	40870.6
256K	5714	45877.5	5861	44726.8
512K	11336	46249.8	10912	48046.9
1M	22727	46137.9	15907	65919.2

From the results listed in table 6.1, it is clear that for small amount of data streaming, the Xillybus performance is poor which means that communication overhead is high. But for large amount of data, its performance is close to theoretical limit (50 MS/s). There is a possibility of writing and reading the pipe at the same time by

```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 tstart, tend;
ssize_t t1,t2, temp;
int main(int argc, char *argv[]) {
fdr32 = open("/dev/xillybus_read_32", O_RDONLY);
fdw32 = open("/dev/xillybus_write_32", O_WRONLY);
N = atoi(argv[1]);
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;
}
//Measure the excution time
gettimeofday(&tstart, NULL);
t1=write(fdw32, array_input, sizeof(int)*N);
temp = write(fdw32, NULL, 0);
t2= read(fdr32,array_hardware, sizeof(int)*N);
gettimeofday(&tend, NULL);
printf("Execution time is %f us\n\r", (double)1000000*(tend.tv_sec-tstart.tv_sec)+(tend.
     tv_usec-tstart.tv_usec));
return 0;
}
```

Figure 6.5: Xillybus Single threading Code Example

using separate threads in software for write and read. In this case, the theoretical bound on round trip bandwidth would be 100 MS/s. From the highlight red results, multi-threading shows its benefit only when the data size is bigger than 512K because of the thread creation time overhead.

```c
//(include the same head files)
define NTH 2
int fdw32 = 0;
int fdr32 = 0;
int N = 0;
int i;
int *array_input;
int *array_hardware;
struct timeval tstart,tend,tv1, tv2;
ssize_t w1,w2,e1,e2,r1,r2;
void *sample_1(void *arg);
void *sample_2(void *arg);
void *sample_1(void *arg) {
w1 = write(fdw32, array_input, sizeof(int)*N);
e1 = write(fdw32, NULL, 0);
pthread_exit(NULL);
}
void *sample_2(void *arg) {
r1 = read(fdr32, array_hardware, sizeof(int)*N);
pthread_exit(NULL);
}
int main(int argc, char *argv[]) {
fdw32 = open("/dev/xillybus_write_32", O_WRONLY);
fdr32 = open("/dev/xillybus_read_32", O_RDONLY);
N = atoi(argv[1]);
if (fdw32 < 0 || fdr32 < 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;
}
pthread_t tid[NTH];
int loop = 0;
int value[NTH] = {1,2};
/** Creation of threads*/
pthread_create(&tid[0], NULL, &sample_1, &value[0]);
pthread_create(&tid[1], NULL, &sample_2, &value[1]);

/** Synch of threads in order to exit normally*/
gettimeofday(&tstart, NULL);
for(loop=0; loop<NTH; loop++) {
pthread_join(tid[loop],NULL);
}
gettimeofday(&tend, NULL);
printf("%f\n\r", (double)1000000*(tend.tv_sec-tstart.tv_sec)+(tend.tv_usec-tstart.tv_usec));
return EXIT_SUCCESS;
}
```

Figure 6.6: Xillybus Muilti threading Code Example

6.3.2 Double Pipe Loopback

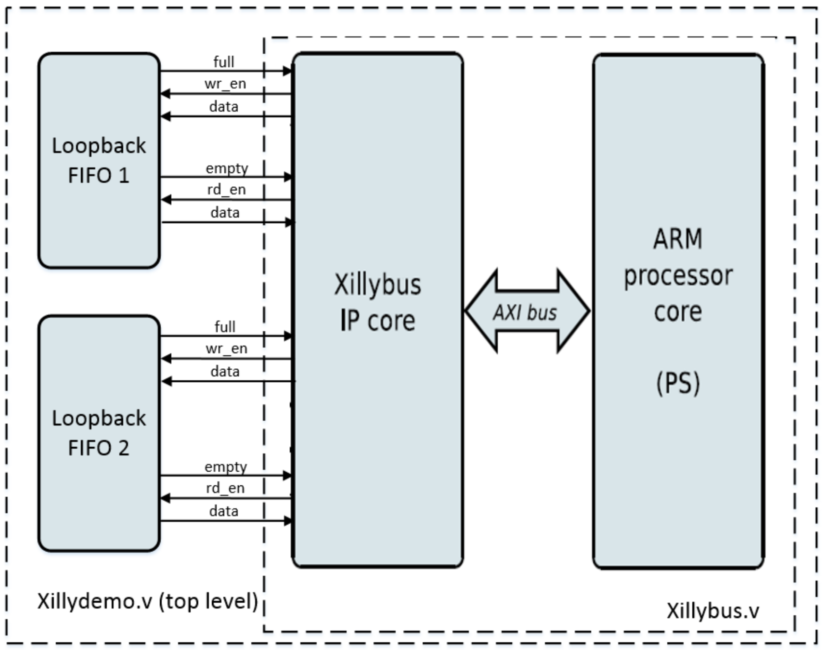


Figure 6.7: Xillybus Double Pipe Loopback block diagram

As mentioned earlier, Xillybus allows to generate multiple input/output streaming interfaces sharing the bandwidth of ACP interface (configured as 32-bit interface running at 100 MHz) on the Zynq platform. We verified it by creating two interfaces for write and read (using Xillybus IP core factory) and measured the round trip time in order to see the impact of writing/reading to multiple pipes. We performed this experiment due to the reason that most of the kernels require multiple input and output interfaces. The block diagram of double pipe loopback is shown in Figure 6.7. Based on the results listed in Table 6.2, we can see that the bandwidth of the ACP interface on the Zynq platform can be shared in a time-multiplexed manner among multiple streaming interfaces. Multi-threading can achieve better performance when data size is bigger than 128K. The main observation from this experiment is that the compute kernels requiring multiple input and output interfaces have to share the bandwidth of the ACP interface in a time-multiplexed fashion which would limit the performance of the accelerator.

Table 6.2: Double Pipe Loopback results

No. Samples	Single-threading		Multi-threading	
	Round Trip Time(us)	Troughput (KS/s)	Round Trip Time(us)	Troughput (KS/s)
256	203	2522.2	811	631.3
512	203	5044.3	811	1262.6
1K	203	10088.7	848	2415.1
2K	240	17066.7	848	4830.2
4K	295	27769.5	977	8384.9
8K	424	38641.5	1106	14813.7
16K	737	44461.3	1327	24693.3
32K	1530	42834.0	1880	34859.6
64K	2728	48046.9	2986	43895.5
128K	5714	45877.5	5456	48046.9
256K	11354	46176.5	10838	48375.0
512K	22893	45803.3	20736	50567.9
1M	45435	46157.2	33343	62896.3

6.4 Xillybus for interfacing HLS generated kernels

In order to integrate an accelerator in the Xillybus system, we need to customize specific Xillybus IP core online according to the accelerator requirement in terms of number of I/O interfaces, bit-width etc. After that we need to insert our logic in the top level module with appropriate connections.

Xillybus core provides stream data to input FIFO which connects to a hardware accelerator (”Application logic” in the diagram) as shown in Figure 6.8. Accelerator provides the processed data to output FIFO which connects again to Xillybus core. The following steps are involved in addition of your custom application logic (e.g. hardware accelerator) to the Xillybus: The FPGA demo bundle for Zynq is used as a base. To customize your own Xillybus IP core online at Xillybus IP Core Factory with specific No. of pipes, direction, data width and expected bandwidth. Then change the initial Xillybus IP core with your custom core. Insert the application logic at the top level module and place the input/output FIFOs between the application logic and Xillybus IP core. After these modification, then just repeat the very same thing like loopback experiment to generate the bit stream and download to FPGA. The application logic can be described in a high level language like C/C++ which can be synthesized by using Vivado HLS tool to obtain its HDL description. The RTL code is highly optimized and fully pipelined. This application logic is instantiated as a component within the Xillybus provided top level wrapper file, xillydemo.v. The logic is inserted in the path by breaking the loopback connection between the FIFOs.

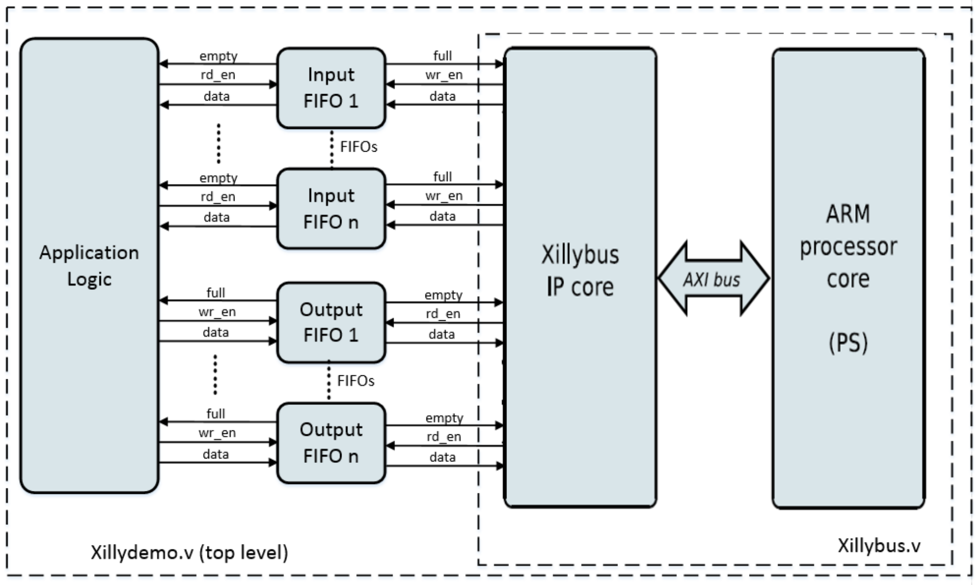
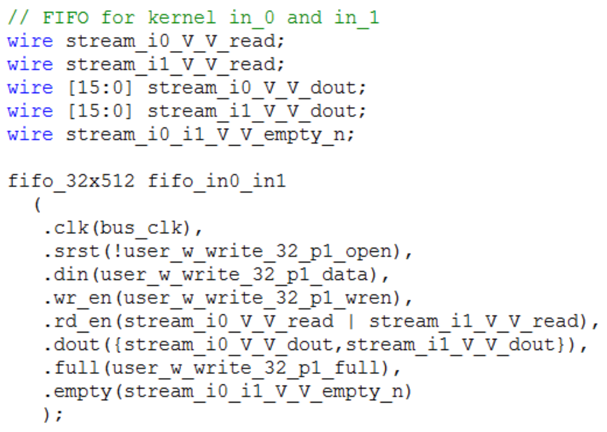


Figure 6.8: Multiple I/O Accelerator integrated in Xillybus system

Table 6.3: No. of I/O for all the benchmarks

benchmarks	conv	fft	kmeans	mm	mriq	radar	spmv	stencil
No. of inputs	24	6	16	16	11	10	16	15
No. of outputs	8	4	1	1	2	2	2	2
Data Width	16	16	16	16	16	16	16	16

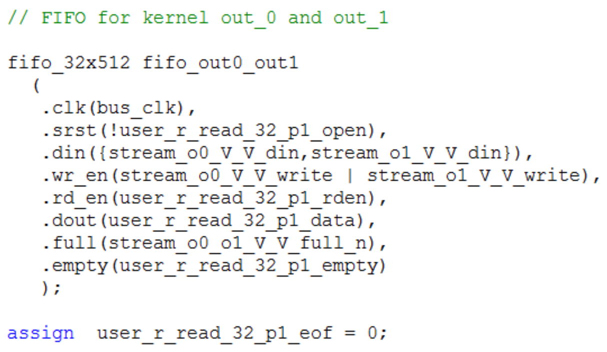
The table 6.3 shows the number of input/outputs for all the benchmarks. We first customize the Xillyus IP core with 16-bit data width pipes which match the data width of the benchmarks input and output. This kind of integration is very straight-forward as each input/output correspond to one Xillybus pipe. For example, for FFT, it needs 6 16-bit input pipes and 4 16-bit output pipes. But we found there is a limit on the number of Xillybus pipes, the 16-bit pipe solution only works for kernels having less I/O ports, such as FFT, mriq and radar. So we moved to another solution which use 32-bit Xillybus pipes and also 32-bit FIFOs in between (as shown in figure 6.9 and 6.10), where each 32-bit pipe can carry two 16-bit input data. After the FIFO, the data bundle will branch to each input port of the accelerator. Similarly, two 16-bit output processed by the accelerator will be combine as a 32-bit word and sent to the output FIFO, then finally to the Xillybus IP core.



```
// FIFO for kernel in_0 and in_1
wire stream_i0_V_V_read;
wire stream_i1_V_V_read;
wire [15:0] stream_i0_V_V_dout;
wire [15:0] stream_i1_V_V_dout;
wire stream_i0_i1_V_V_empty_n;

fifo_32x512 fifo_in0_in1
  (
   .clk(bus_clk),
   .srst(!user_w_write_32_p1_open),
   .din(user_w_write_32_p1_data),
   .wr_en(user_w_write_32_p1_wren),
   .rd_en(stream_i0_V_V_read | stream_i1_V_V_read),
   .dout({stream_i0_V_V_dout,stream_i1_V_V_dout}),
   .full(user_w_write_32_p1_full),
   .empty(stream_i0_i1_V_V_empty_n)
   );
```

Figure 6.9: 32-bit input pipe connection



```
// FIFO for kernel out_0 and out_1

fifo_32x512 fifo_out0_out1
  (
   .clk(bus_clk),
   .srst(!user_r_read_32_p1_open),
   .din({stream_o0_V_V_din,stream_o1_V_V_din}),
   .wr_en(stream_o0_V_V_write | stream_o1_V_V_write),
   .rd_en(user_r_read_32_p1_rden),
   .dout(user_r_read_32_p1_data),
   .full(stream_o0_o1_V_V_full_n),
   .empty(user_r_read_32_p1_empty)
   );

assign  user_r_read_32_p1_eof = 0;
```

Figure 6.10: 32-bit output pipe connection

For this method, we can save the number of Xillybus pipes by half and improve the PS-PL communication efficiency as within the same amount of time, the transferred data is doubled. The test results are shown in the table 6.4 and 6.5.

6.5 Performance evaluation

A fully pipelined accelerator (running at 100 MHz) having N input interfaces can process 100 M-Samples per second, considering no time sharing of I/O interfaces. Since we are using Xillybus and sharing the bandwidth of ACP interface among kernel I/O interfaces, we expect significantly reduced performance from the accelerator.

Table 6.4: Round Trip Time in us For Kernels intergrated with Xillybus 1

No. Samples	fft(3w2r)		kmeans(8w1r)		mm(8w1r)		mriq(6w1r)	
	Single-thread	Multi-thread	Single-thread	Multi-thread	Single-thread	Multi-thread	Single-thread	Multi-thread
256	258	940	424	1161	424	1161	332	977
512	258	940	424	1137	442	1143	350	977
1K	276	940	479	1161	479	1161	387	977
2K	313	977	627	1106	627	1124	516	977
4K	442	1032	922	1106	922	1198	737	1069
8K	682	1069	1567	1493	1567	1493	1235	1327
16K	1235	1153	2857	2304	2857	2304	2212	1788

Table 6.5: Round Trip Time in us For Kernels intergrated with Xillybus 2

No. Samples	radar(5w1r)		spmv(8w1r)		stencil(8w1r)	
	Single-thread	Multi-thread	Single-thread	Multi-thread	Single-thread	Multi-thread
256	295	811	424	1161	424	1143
512	295	811	442	1161	442	1180
1K	350	1006	479	1161	461	1161
2K	442	1132	627	1161	627	1161
4K	627	1235	903	1198	922	1198
8K	1032	1327	1567	1272	1567	1493
16K	1880	1696	2894	2304	2857	2304

Table 6.6: Results processed by ARM processor on Linux

No. Samples	Round Trip Time in us						
	fft	kmeans	mm	mriq	radar	spmv	stencil
1K	74	166	129	74	129	74	74
2K	148	332	277	166	258	148	129
4K	276	645	553	332	516	276	258
8K	553	1272	1124	663	1032	571	516
16K	1088	2543	2212	1346	2119	1143	1032

The results are shown in the table 6.4 and 6.5. The performance is not satisfactory due to the large overhead of communication, especially for large amount of data. For the data size of 16K, multi-threading slightly improves the performance but for small data size, multi-threading results in worst performance. We also evaluate the performance of ARM processer for the kernels as shown in table 6.6. Fig. 6.11 shows the linear increase in execution time (fft and kmeans executing on ARM processor) as we increase the number of samples. Lower bound on execution time is shown for a fully pipelined accelerator interfaced with the host processor (considering zero communication overhead) running at 100 MHz. Fig. 6.11 also shows the execution time (fft and kmeans executing on FPGA fabric and interfaced within Zynq using Xillybus) as we increase the number of samples. It is clear from the graph that the achieved performance is same for ARM processor and accelerator due to the communication overheads involved in the Xillybus infrastructure. In fact, for small number of samples, processor can perform the operations faster than the Xillybus interfaced accelerator. In summary, Xillybus does not provide a good solution for accelerating kernels requiring multiple I/O interfaces. The performance gap can be reduced by developing high speed communication infrastructure on Zynq platform. Then only it will make sense to integrate proposed overlay within Zynq.

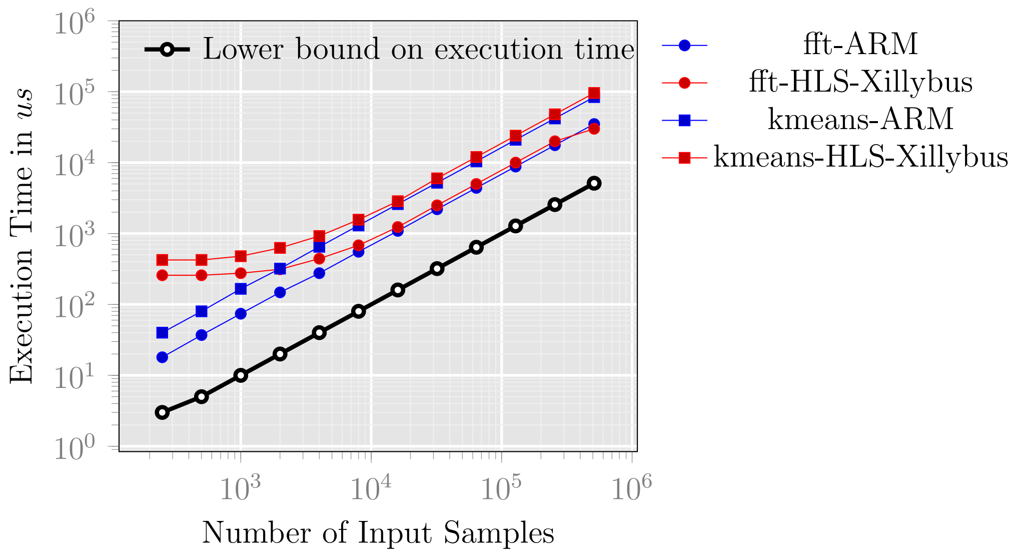


Figure 6.11: Execution time analysis

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 presented an analysis of DySER architecture as an overlay on the Xilinx Zynq and proposed an area efficient overlay based on linear array of time-multiplexed functional units. Use of Xillybus infrastructure provided the base for integrating accelerators within the Zynq Platform. This work included developing an understanding of hardware acceleration concept, overlay architectures and software-hardware communication on a hybrid computing platform (the Xilinx Zynq). Experiments were designed to analyze the performance of DSP block based functional unit for DySER, verify the functionality of time-multiplexed functional unit and linear array of these units, characterize the Xillybus infrastructure, evaluate the performance of HLS generated RTL implementations and the ARM processor for the execution a set of compute kernels. A set of RTL implementations were developed for the DSP based spatially configured functional unit (compatible with DySER), time-multiplexed functional unit and also for the linear array of functional units. RTL implementations of kernels were also generated using HLS tool which were then connected within Zynq via Xillybus. Before we could begin implementing overlay based on time-multiplexed functional units, an in-depth knowledge of the current trends and previous efforts in the field of overlay architectures were studied to compare and contrast their features.

An analysis of DySER Architecture as an Overlay on the Xilinx Zynq was presented in chapter 4. An enhancement to the DySER coarse-grained overlay was presented that uses the Xilinx DSP48E1 primitive to implement most of the functional unit, improving area and performance. We show an improvement of 2.5× in frequency and a reduction of 25% in area compared to the original functional unit design. We have shown that a more architecture-oriented approach to designing the FU enables it to be small and fast and exposes the significant overhead of the flexible routing. The key finding from this chapter is that an area and performance efficient interconnect architecture is necessary for improving the performance of the overlays. One approach to improve the area efficiency was shown in chapter 5 by proposing an overlay based on linear array of time-multiplexed functional units by exploiting cycle by cycle reconfiguration capability of the DSP block. The performance of HLS generated RTL implementations and the ARM processor was evaluated for the execution of a set of compute kernels and results were presented in chapter 6. Furthermore, the approach presented in this report facilitates high level application developers to use area-efficient overlays for hardware acceleration of compute kernels.

7.2 Future work

Some of the main future research directions are routing network designs for area overhead reduction, automated mapping of kernels on time-multiplexed functional unit based overlay, integration of overlay with ARM processor using high performance communication interfaces. We describe these directions in detail as follows:

- **Area efficient routing network architectures for DySER overlay**: The area overheads of the DySER can be reduced by carefully designing the routing network architecture. Some of the possible choices are multistage switching networks, hierarchical routing network and omega network etc.
- **Automated mapping of kernels on time-multiplexed functional unit based overlay**: To generate instruction for each FU in the overlay architecture

in the current version, the user has to manually write the instructions for each FU based on the kernel. A mapping tool can be developed for automated mapping of kernels on time-multiplexed functional unit based overlay.

- **Integration of accelerators with ARM processor using high performance communication interfaces**: Integration of accelerators (HLS generated kernels or overlay) with a host processor using high performance communication interfaces is crucial for overall application acceleration.

Finally, with these initiatives we hope to develop area efficient overlay architectures based on time-multiplexed functional units where compute kernels can be compiled at runtime within a platform in which overlay can be interfaced to the host processor using high performance communication interfaces.

Bibliography

[1] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, 2012.

[2] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, and D. L. Maskell. Virtualized execution and management of hardware tasks on a hybrid ARM-FPGA platform. *Journal of Signal Processing Systems*, 77(1–2):61–76, Oct. 2014.

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

[4] 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.

[5] 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.

[6] A. K. Jain, X. Li, S. A. Fahmy, and D. L. Maskell. Adapting the DySER architecture with DSP blocks as an Overlay for the Xilinx Zynq. In *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*, 2015.

[7] Zachary Marzec. Detailed performance evaluation of data-parallel workloads on the dyser prototype system. `http://research.cs.wisc.edu/vertical/papers/thesis/marzec-report.pdf`.

[8] Hui Yan Cheah, Suhaib A. Fahmy, and Douglas L. Maskell. iDEA: A DSP block based FPGA soft processor. In *Proceedings of the International Conference on Field Programmable Technology (FPT)*, pages 151–158, 2012.

[9] Russell Tessier, Kenneth Pocek, and Andre DeHon. Reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):332–354, 2015.

[10] Andre DeHon. Fundamental underpinnings of reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):355–378, 2015.

[11] Stephen M Trimberger. Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology. *Proceedings of the IEEE*, 103(3):318–331, 2015.

[12] Greg Stitt. Are field-programmable gate arrays ready for the mainstream? *IEEE Micro*, 31(6):58–63, 2011.

[13] C. Plessl and M. Platzner. Zippy - a coarse-grained reconfigurable array with support for hardware virtualization. In *Proceedings of the International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, pages 213–218, 2005.

[14] 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.

[15] A. K. Jain, S. A. Fahmy, and D. L. Maskell. Efficient overlay architecture based on DSP blocks. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2015.

[16] Cheng Liu, C.L. Yu, and H.K.-H. So. A soft coarse-grained reconfigurable array based high-level synthesis methodology: Promoting design productivity and exploring extreme FPGA frequency. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 228–228, 2013.

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

[18] Jesse Benson, Ryan Cofell, Chris Frericks, Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. Design, integration and implementation of the dyser hardware accelerator into opensparc. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2012.

[19] A. George, H. Lam, and G. Stitt. Novo-g: At the forefront of scalable reconfigurable supercomputing. *Computing in Science Engineering*, 13(1):82–86, 2011.

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

[21] 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.

[22] 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.

[23] Yun Liang, Kyle Rupnow, Yinan Li, and et. al. High-level synthesis: productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*, 2012(649057):1–14, January 2012.

[24] David Bacon, Rodric Rabbah, and Sunil Shukla. FPGA programming for the masses. *Queue*, 11(2):40:40–40:52, February 2013.

[25] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. A fully pipelined and dynamically composable architecture of cgra. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 9–16, 2014.

[26] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 503–514, 2011.

[27] P.J. Bakkes, J.J. Du Plessis, and B.L. Hutchings. Mixing fixed and reconfigurable logic for array processing. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 118–125, 1996.

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

[29] 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.

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

[31] Alexander Brant. Coarse and fine grain programmable overlay architectures for FPGAs. Master's thesis, University of British Columbia, 2013.

[32] K. Paul, C. Dash, and M.S. Moghaddam. reMORPH: a runtime reconfigurable architecture. In *Euromicro Conference on Digital System Design*, 2012.

[33] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for fpga research. In *Field-Programmable Logic and Applications*, pages 213–222, 1997.

[34] Aaron Landy and Greg Stitt. A low-overhead interconnect architecture for virtual reconfigurable fabrics. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 111–120, 2012.

[35] G. Stitt, A. George, H. Lam, C. Reardon, M. Smith, B. Holland, V. Aggarwal, Gongyu Wang, J. Coole, and S. Koehler. An end-to-end tool flow for FPGA-Accelerated scientific computing. *IEEE Design and Test of Computers*, 28(4):68–77, August 2011.

[36] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Timing-driven placement for fpgas. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 203–213, 2000.

[37] Larry McMurchie and Carl Ebeling. Pathfinder: a negotiation-based performance-driven router for fpgas. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 1995.

[38] Christopher H. Chou, Aaron Severance, Alex D. Brant, Zhiduo Liu, Saurabh Sant, and Guy G.F. Lemieux. VEGAS: soft vector processor with scratchpad memory. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 15–24. ACM, 2011.

[39] Colin Yu Lin, Ngai Wong, and H Kwok-Hay So. Operation scheduling for fpga-based reconfigurable computers. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 481–484, 2009.

[40] Alexander Fell, Zoltán Endre Rákossy, and Anupam Chattopadhyay. Force-directed scheduling for data flow graph mapping on coarse-grained reconfigurable architectures. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2014.

[41] Liang Chen and Tulika Mitra. Graph minor approach for application mapping on cgras. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):21, 2014.

[42] Rafat Rashid, J Gregory Steffan, and Vaughn Betz. Comparing performance, productivity and scalability of the tilt overlay processor to opencl hls. In *Proceedings of the International Conference on Field Programmable Technology (FPT)*, pages 20–27, 2014.

[43] Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki, Zachary Marzec, Preeti Agarwal, Chris Frericks, Ryan Cofell, Jesse Benson, and Karthikeyan Sankaralingam. Performance evaluation of a DySER FPGA prototype system spanning the compiler, microarchitecture, and hardware implementation. *Energy (mJ)*, 5(10):15, 2015.

[44] Karel Heyse, Tom Davidson, Elias Vansteenkiste, Karel Bruneel, and Dirk Stroobandt. Efficient implementation of virtual coarse grained reconfigurable arrays on FPGAS. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2013.