MICROBLAZE-BASED COPROCESSOR FOR DATA STREAM MANAGEMENT SYSTEMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Tareq S. Alqaisi

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

December 2017

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

To my daughter, for her never-ending inspiration.

ACKNOWLEDGMENTS

I would first like to express my sincerest gratitude to my adviser Dr. John J. Lee for his patience, guidance, and support during this journey.

I would also like to thank Sherrie Tucker for her patience, help, and encouragement.

Lastly, I must thank Dr. Pranav Vaidya for all his support, technical guidance, and advice. Thank you for taking time out of your busy life to help me finish this study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Alqaisi, Tareq S. MSECE, Purdue University, December 2017. MicroBlaze-based Coprocessor for Data Stream Management Systems. Major Professor: John J. Lee.

Data network's speed and availability are increasing at a record rate. More and more devices are now able to connect to the Internet and stream data. Processing this ever-growing amount of data in real time continues to be a challenge.

Multiple studies have been conducted to address the growing demands for real-time processing and analysis of continuous data streams. Developed in a previous work, Symbiote Coprocessor Unit (SCU) is a hardware accelerator capable of providing up to 150× speedup over traditional data stream processors in the field of data stream management systems.

However, SCU implementation is very complex, fixed, and uses an outdated host interface, which limits future improvements.

In this study, we present a new SCU architecture that is based on a Xilinx MicroBlaze configurable microcontroller. The proposed architecture reduces complexity, allows future implementations of new algorithms in a relatively short amount of time while maintaining the SCU's high performance. It also has an industry standard PCIe interface. Finally, it uses a standard AMBA AXI4 bus interconnect, which enables easier integration of new hardware components.

The new architecture is implemented using a Xilinx VC709 development board. Our experimental results have shown a minimal loss of performance as compared to the original SCU while providing a flexible and simple design.

1. INTRODUCTION

With the development of the Internet of Things (IoT) and the wide availability of wireless networks, more devices are able to continuously stream data. This increased the need for an efficient high throughput data-intensive computing systems that are able to process data streams in real time.

Data Stream Management Systems (DSMSes) are designed to process potentially unbounded, dynamic data streams with limited resources such as main memory. Data in the streams are volatile and sequential with high update rate that can also arrive at variable pattern.

DSMSes such as Aurora [1], Gigascope [2], Nile [3], and Symbiote Coprocessor Unit (SCU) [4] have been developed to address the increasing needs for data stream processing. Most of these systems are software-based running on the host computer. In contrast, SCU is a hybrid system that utilizes a hardware coprocessor to accelerate data stream processing.

1.1 Related Work

Symbiote Coprocessor Unit (SCU) [4] is an FPGA-based coprocessor designed to accelerate data stream applications. Study results show a speedup in the range 12.3$\times$ to 150$\times$ over software-based DSMSes.

SCU contains a Single Instruction Multiple Data (SIMD) Very-Large Instruction Word (VLIW) execution engine that exploits the data level parallelism and instruction level parallelism in DSMS operators. The execution engine is coupled with a high-bandwidth multi-banked streaming memory system that is capable of writing or reading multiple words per clock cycle.

SCU contains a very complex state machine called Stream Controller (SC). The SC acts as a proxy controller for the host processor. It configures, runs, and monitors the SIMD-VLIW execution engine.

The current implementation of the SCU has three limitations. First, it is very complex, which limits future improvements. Second, it uses the outdated Hypter-Transport [5] protocol to interface with a host processor; at the time this study is being conducted no new FPGAs support HyperTransport. Finally, it uses multiple proprietary bus interconnects, which limits expansion of hardware.

1.2 Motivation

To overcome the limitations mentioned in Section 1.1, this study updates the SCU architecture as follows:

- To reduce complexity of the SCU, the proposed architecture replaces the Stream Controller with a microcontroller, namely MicroBlaze [6]. The use of a micro-controller allows a software implementation of the state machine to replace the current fixed, complex hardware-based state machine. Furthermore, Software developers can implement new algorithms, or add new features easily without hardware modifications.

- To increase the compatibility of the SCU, the proposed architecture replaces the outdated HyperTransport protocol with industry standard GEN 3 high speed PCIe bus. PCIe is widely used and can be integrated in any system.

- To allow future hardware expansion of the SCU, the proposed architecture uses industry standard AMBA AXI4 [7] bus interconnect instead of the multiple proprietary bus interconnects used in the current architecture.

One might concern that replacing the hardware implementation of the control finite state machine with a software implementation using a microcontroller would significantly degrade performance. This is fortunately not the case as SC is not in the critical path of the SCU's data processing.

At the time this study is being conducted, the MicroBlaze has not been used to control data stream coprocessors.

1.3 Thesis Outline

In order to better explain the proposed architecture, Chapter 2 introduces the current SCU architecture. Chapter 3 provides a brief introduction to MicroBlaze system, PCIe and AXI, and then describes in detail the proposed hardware architecture. Chapter 4 covers the software architecture and implementation of the software-based control finite state machine. Chapter 5 explains the experimental setup and analyzes the performance of the proposed architecture. Finally, Chapter 6 concludes this thesis and suggests ideas for future work.

2. SYMBIOTE COPROCESSOR UNIT

2.1 Introduction

Symbiote Coprocessor Unit (SCU) [4] is an FPGA-based coprocessor unit designed to accelerate data streaming applications with semi-persistent queries. SCU at its core has a Single Instruction Multiple Data (SIMD) Very-Large Instruction Word (VLIW) execution engine, which exploits Data Level Parallelism (DLP) and Instruction Level Parallelism (ILP) in DSMS operators.

In addition, the SCU has its own toolchain that consists of a procedural query language called SymQL [4] and its dedicated compiler. SCU DSMS queries are written using SymQL, and then compiled to produce kernel binaries optimized to run on the SIMD-VLIW engine. The overall SCU architecture and its components relevant to this study are introduced in the following sections.

2.2 Symbiote Coprocessor Unit Architecture

The SCU is designed as a coprocessor unit. The overall SCU architecture is shown in Figure 2.1. One of its important components is the HyperTransport Interface (HTI), which connects the SCU to the host processor. HTI exposes several components of the SCU, namely the command FIFO, response FIFO, Write Stripe Pipe (WSP), and Read Stripe Pipe (RSP) onto the address space of the host processor.

The Stream Controller (SC) acts as proxy controller for the host processor, receives commands through the command FIFO, and replies via the response FIFO. The host processor programs the Instruction Memory (IMEM) with SymQL queries by sending a specific command and the binaries to SC.

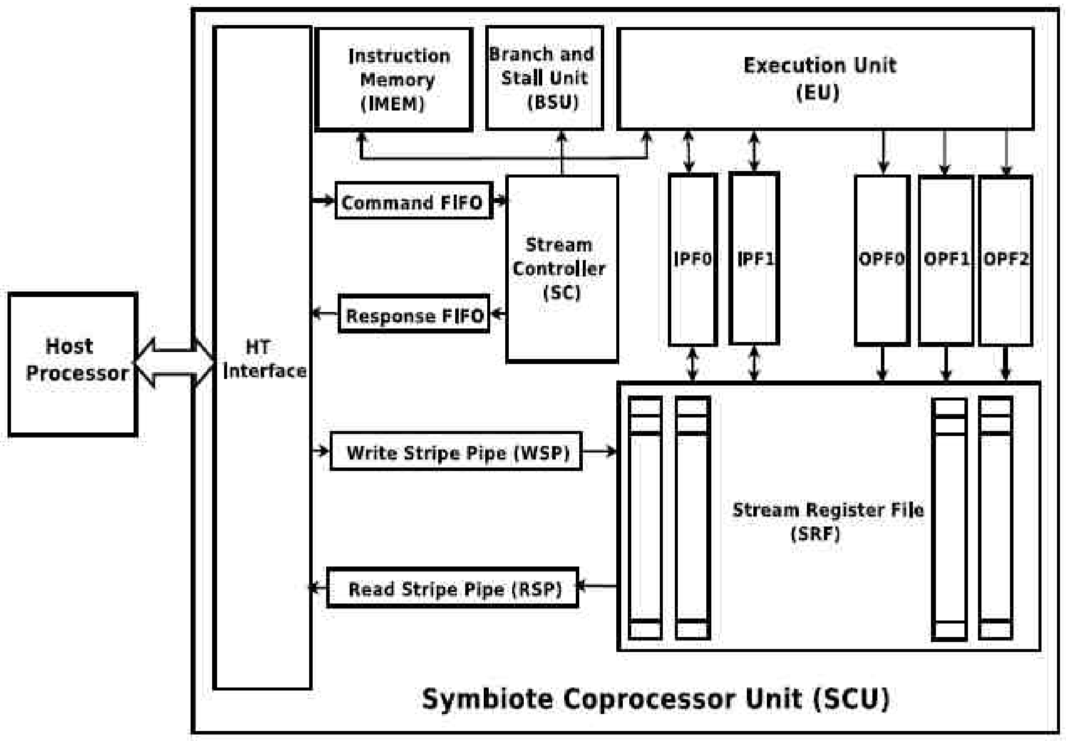


Fig. 2.1.: Symbiote Coprocessor Unit architecture (Courtesy of Vaidya [4]).

The SCU contains a 320-bit wide VLIW Execution Engine (EU). EU executes kernel binaries on input streams and produces output streams that can be read by the host processor. The EU consists of nine functional units, namely a store unit, a load unit, a permute unit, two ALUs, two multipliers, and finally two dividers. Each of the functional units is SIMD-capable and operates on eight 32-bit values in parallel.

To store input and output streams, the SCU contains a 512KB multi-banked memory called Stream Register File (SRF). The SRF is bandwidth optimized with eight memory banks to allow the host processor or the execution engine to read and write eight-word vector simultaneously.

To service multiple clients, the SRF implements read/write access control so as to ensure only one client can access the SRF. Fair round-robin arbiters, namely Read Arbiter (RA) and Write Arbiter (WA), guarantee one client access to SRF.

The main job of Write Stripe Pipe (WSP) is to convert the sequentially stored stream tuples in the host processor to the eight-bank striped physical layout of the SRF, which allows the Execution Unit (EU) to access them in parallel. Read Stripe Pipe (RSP) on the other hand restores the layout of the tuples in the SRF back to sequential host processor layout.

In order for the kernel to run, the input tuples are pre-fetched from the SRF by two input FIFOs. Similarly, output tuples from the EU are written in predetermined order via three output FIFOs.

The rest of this section introduces SCU's most important components to the proposed architecture. Section 2.2.1 explains the architecture of the HyperTransport Interface, being the main data distribution path in the SCU. Section 2.2.2 introduces the SCU's Stream Controller (SC), the main study topic of this thesis. SC is a complex finite state machine responsible for configuring, executing, and monitoring the VLIW Execution Engine. Section 2.2.3 covers Command and Response FIFOs, which are used by the host processor to communicate with the SC. Finally, Section 2.2.4 provides a brief introduction to Stream Register File (SRF), the main storage for input and output streams.

2.2.1 HyperTransport Interface (HTI)

The HyperTransport Interface is the central communication component responsible for connecting SCU's modules to the host processor. It converts the signal and protocol mappings between HyperTransport signals and SCU's HTI interconnects as shown in Figure 2.2.

HTI consists of four main components, namely the HT Read Engine (HT_RDE), HT Write Engine (HT_WRE), HTI Read Interconnect (HTI_RDX), and lastly HTI Write Interconnect (HTI_WRX). HT_RDE and HT_WRE main responsibilities are to manage and control the communication, arbitration, and ordering of the host processor read and write transactions to the SCU components.

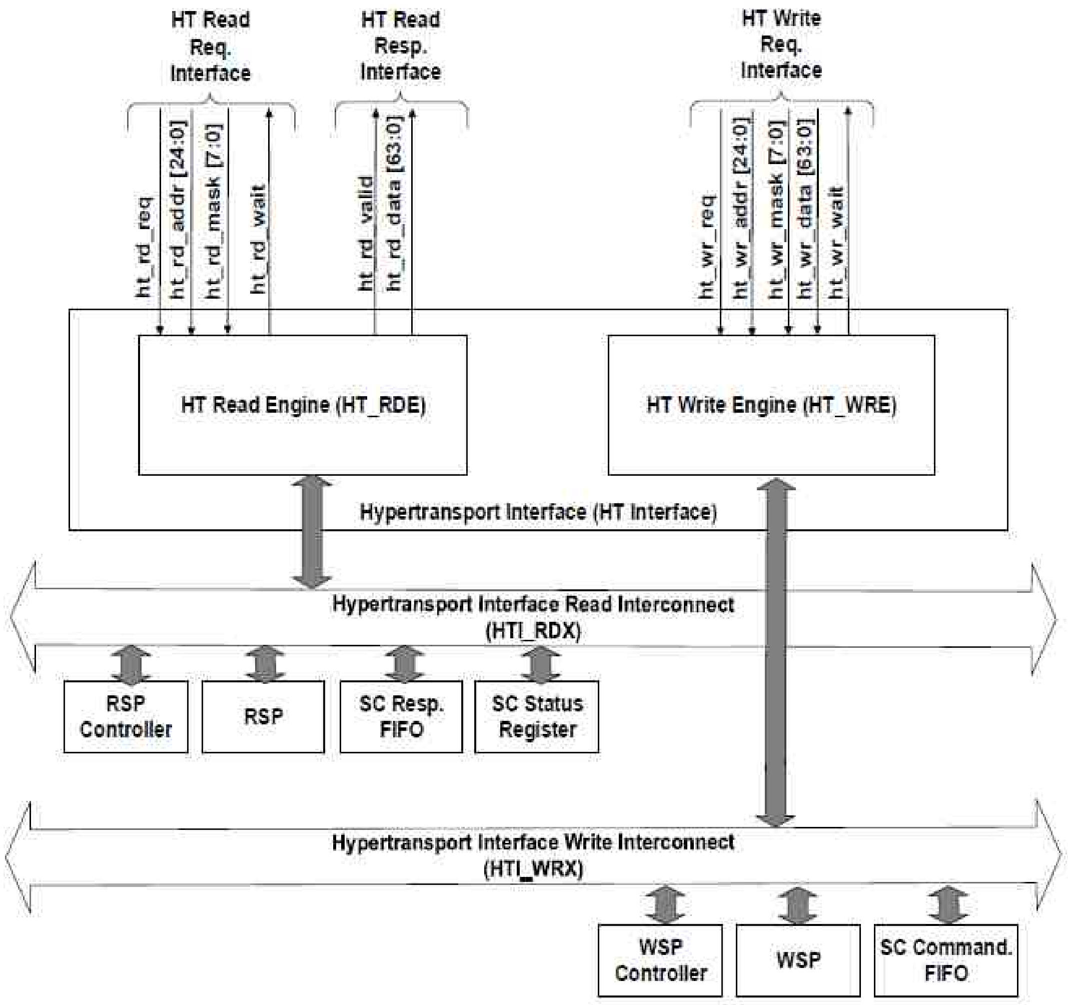


Fig. 2.2.: HyperTranpsort Interface (HTI) components (Courtesy of Vaidya [4]).

Figure 2.2 also shows the SCU blocks exposed over HT address space. HTI_RDX exposes read only blocks of the SCU, in specific, Read Strip Pipe (RSP), Read Stripe Pipe Controller (RSP_CTRL), Stream Controller Response FIFO, and SC Status Register. Additionally, HTI_WRX exposes write only blocks, namely Stream Controller Command FIFO, Write Strip Pipe (WSP), and finally WSP controller to the host.

HT_RDE's internal components are shown in Figure 2.3. Host processor requests are enqueued in Read Request Queue. Once the HT Read Request Dispatcher detects there is a request in the queue, it dequeues and broadcasts the request to its clients via HTI_RDX. At the same time, the request is saved in the Response Ordering FIFO to match it later with its response. Once the targeted client responds to the request, the response is enqueued in the Response FIFO and sent back to the host processor.

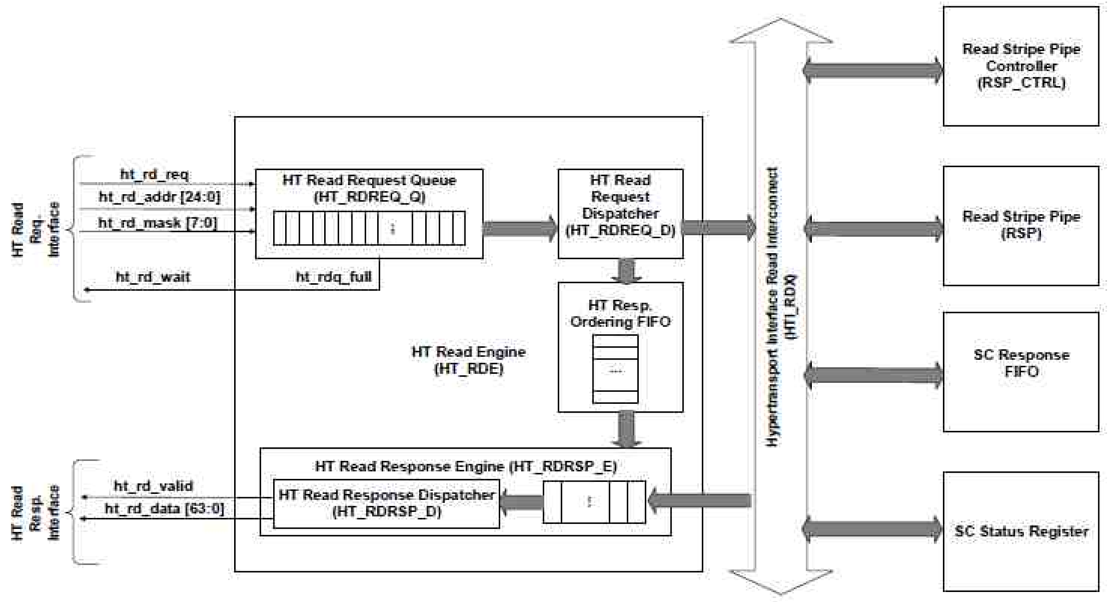

HT Read Req. Interface
ht_rd_req
ht_rd_addr [24:0]
ht_rd_mask [7:0]
ht_rd_wait
ht_rdq_full
HT Read Request Queue (HT_RDREQ_Q)
HT Read Request Dispatcher (HT_RDREQ_D)
HT Resp. Ordering FIFO
HT Read Engine (HT_RDE)
HT Read Response Engine (HT_RDRSP_E)
HT Read Response Dispatcher (HT_RDRSP_D)
HT Read Resp. Interface
ht_rd_valid
ht_rd_data [63:0]
Hypertransport Interface Read Interconnect (HTI_RDX)
Read Stripe Pipe Controller (RSP_CTRL)
Read Stripe Pipe (RSP)
SC Response FIFO
SC Status Register

Fig. 2.3.: HyperTransport Read Engine (HTRDE) architecture (Courtesy of Vaidya [4]).

HTI_RDX communicates with its client using a three phase protocol shown in Figure 2.4. The HT request is broadcast over three signals, htrdx_req_valid, htrdx_req_addr and htrdx_req_mask. If a client accepts the request, it indicates via client_htrdx _can_accept, and also deasserts client_htrdx_can_handle signal. HT_RDREQ_D will only dequeue and dispatch the request if the client indicates that it can accept and also handle the request. Once the client completes the request, it asserts client_htrdx_rsp_rdy line; at this point the HT_RDRESP_E reads the response data via client_htrdx_rsp_data and dispatches it to the host processor. Finally to finish the transaction, HT_RDRSP_E sends a handshake signal htrdx_client_rsp_ack.

Figure 2.5 shows the internal components of the HT Write Engine. When the HT Write Request Dispatcher (HT_WRREQ_D) detects a request in the HT Write Re-

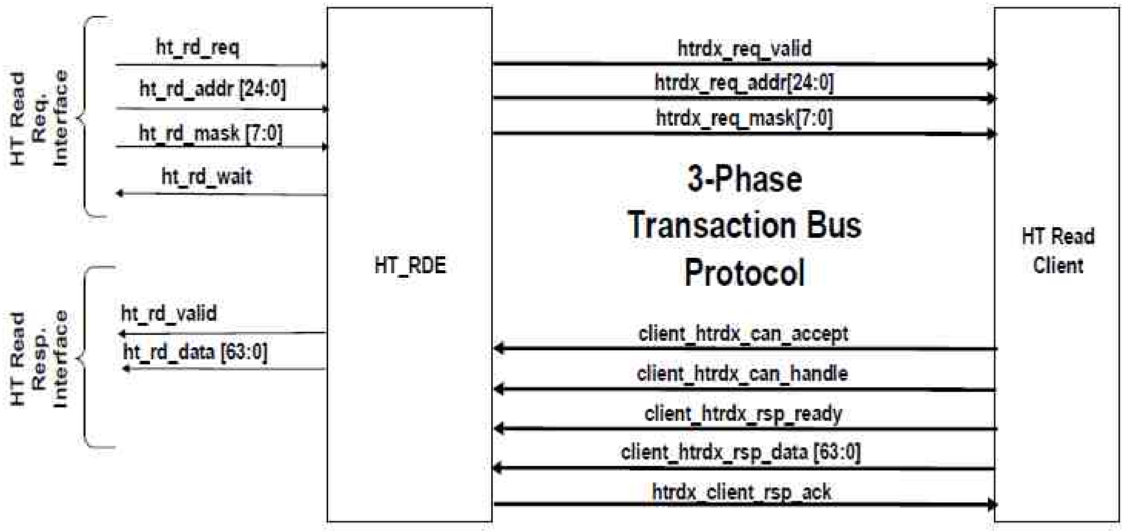


Fig. 2.4.: HyperTransport Interface Read Interconnect Protocol (Courtesy of Vaidya [4]).

quest Queue (HT_WRREQ_Q), it broadcasts the dequeued request to its clients over four signals, namely htwrx_req_valid, htwrx_req_addr, htwrx_req_mask, and finally htwrx_req_data. The clients are continuously monitoring these signals; once a request is addressed to a specific client, the client responds by setting client_htwrx_can_accept signal. Clients can indicate to the HT_WRREQ_D their unavailability by asserting their dedicated client_htwrx_can_handle signal. Similar to the HT_RDE, the request is only dequeued and sent to the client once the client can accept and handle the request. Handshake signal is not required to write request in contrast with read ones. Figure 2.6 shows HyperTransport Write Interconnect signals.

2.2.2 Stream Controller

In order for the host processor to control the SCU, a Stream Controller (SC) was designed to act as a proxy. The host processor sends commands to the SC via the HyperTransport Interface, which then decodes and executes these commands. Executing kernels in the SCU requires important information about the kernel binaries,

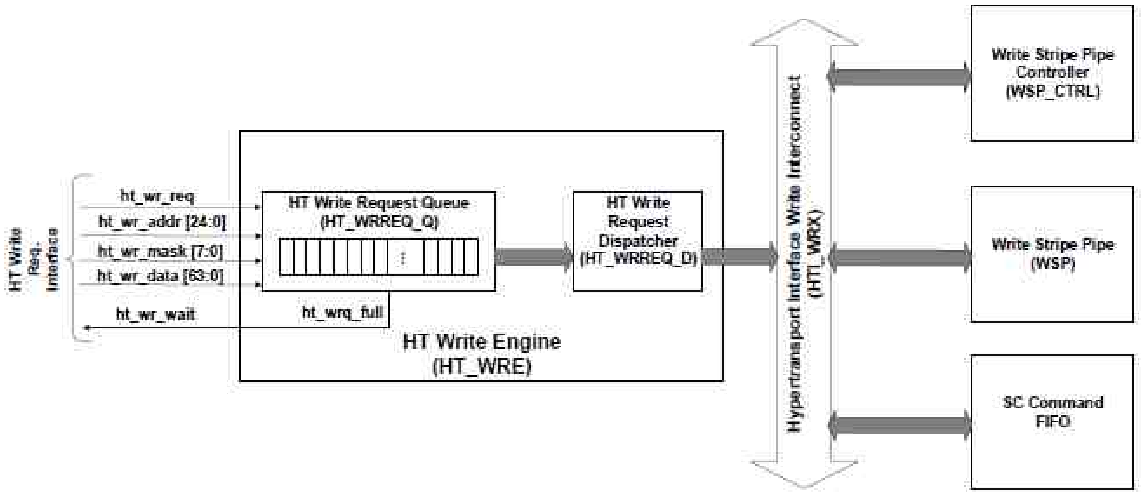

HT Write Req. Interface
ht_wr_req
ht_wr_addr [24:0]
ht_wr_mask [7:0]
ht_wr_data [63:0]
ht_wr_wait
HT Write Request Queue (HT_WRREQ_Q)
ht_wrq_full
HT Write Request Dispatcher (HT_WRREQ_D)
HT Write Engine (HT_WRE)
Hypertransport Interface Write Interconnect (HTI_WRX)
Write Stripe Pipe Controller (WSP_CTRL)
Write Stripe Pipe (WSP)
SC Command FIFO

Fig. 2.5.: HyperTransport Write Engine (HTWRE) architecture (Courtesy of Vaidya [4]).

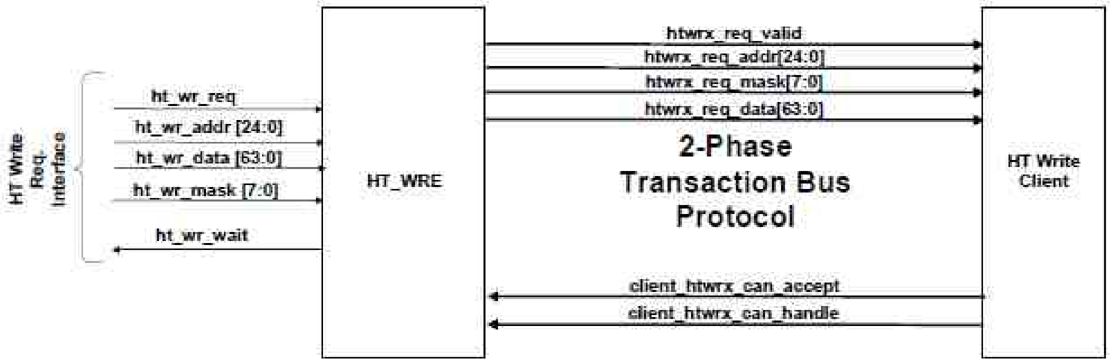
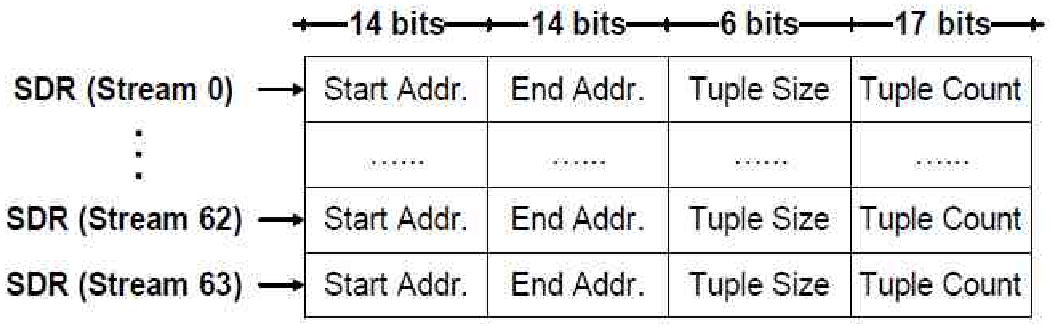


Fig. 2.6.: HyperTransport Interface Write Interconnect Protocol (Courtesy of Vaidya [4]).

the stream description, and SymQL compiler optimization. The SC stores these pieces of information in the following register files:

1. Stream Descriptor File (SDR): a 64×51-bit register file the SC uses to store information about data streams. Data streams are stored in the SRF, and to be able to run kernels on these streams, the Execution Engine needs to know their start addresses in the SRF and their sizes. Each record in the SDR contains

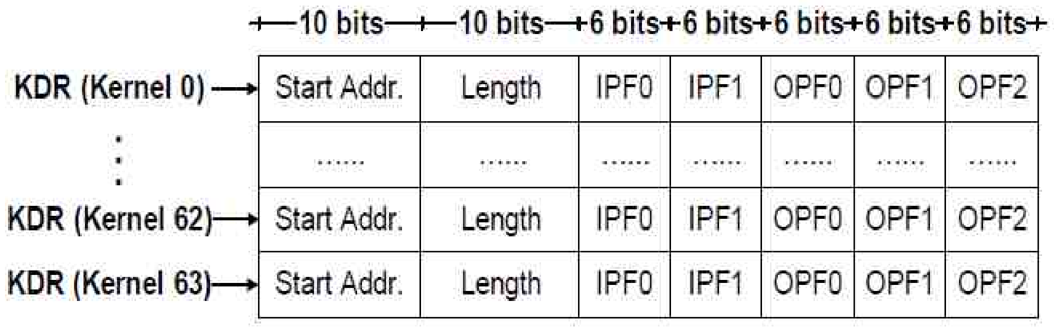
the start address, the end address, tuple size, and finally tuple count for each stream. SDR layout is shown in Figure 2.7.



	14 bits	14 bits	6 bits	17 bits
SDR (Stream 0) →	Start Addr.	End Addr.	Tuple Size	Tuple Count
⋮	......	......	......	......
SDR (Stream 62) →	Start Addr.	End Addr.	Tuple Size	Tuple Count
SDR (Stream 63) →	Start Addr.	End Addr.	Tuple Size	Tuple Count

Fig. 2.7.: Stream Descriptor File (Courtesy of Vaidya [4]).

2. Kernel Descriptor File (KDR): a 64×50-bit register file the SC uses to store information about the kernel binaries stored in the Instruction Memory (IMEM). Each entry in the KDR consists of the start address of the kernel in IMEM, the kernel size, and five fields used to index the SDR to gather information about the input and output streams in SRF. For example, if the query involves processing of one input stream and two output streams, the Execution Engine needs to know where the input stream in the SRF to read from and where to store the results into. Figure 2.8 shows the layout of the KDR.



	10 bits	10 bits	6 bits	6 bits	6 bits	6 bits	6 bits
KDR (Kernel 0) →	Start Addr.	Length	IPF0	IPF1	OPF0	OPF1	OPF2
⋮	......	......	......	......	......	......	......
KDR (Kernel 62) →	Start Addr.	Length	IPF0	IPF1	OPF0	OPF1	OPF2
KDR (Kernel 63) →	Start Addr.	Length	IPF0	IPF1	OPF0	OPF1	OPF2

Fig. 2.8.: Kernel Descriptor File (Courtesy of Vaidya [4]).

3. Offset Register File (ORF): a 64×5×64-bit register file that contains information generated by the SymQL compiler to optimize the execution of the kernels. The SC writes these values to OFFSET_MEM register of the input and output FIFOs before executing the kernel. Figure 2.9 shows the layout of the ORF.

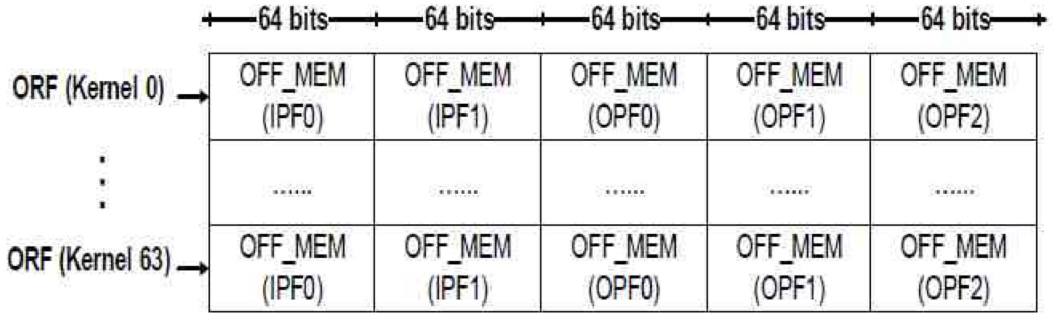


Fig. 2.9.: Offset Register File (Courtesy of Vaidya [4]).

When the host processor commands the SCU to execute a query on a specific stream, it first configures the SCU by writing the KDR register, then copies the kernel binaries to the IMEM, and finally configures the ORF entries for that specific KDR. This process is not time critical and needs to be done once per kernel. When a data stream arrives to the host processor, the host processor first divides the streams into windows, and then sends the SDRs specific for each window to the SC. The host processor transfers the data stream of each window to the SRF, and commands the SCU to run the kernel. Once the results are ready, the host processor reads the data back.

To allow the SC to respond to the host processor's commands while a kernel is running, the SC contains a dedicated component named SC Kernel Run Monitor (SC_KRM) designed to configure the EU and monitor the progress of the kernel execution. Once the SC_KRM detects the completion of the kernel, it interrupts the SCU and reports information related to the output stream(s).

As mentioned earlier, the SC interacts with multiple components of the SCU; as a result, a communication interface named SC Transaction Interconnect (SCTX) is implemented. When the SC (and the SC_KRM) communicates with its clients, it asserts sc_client_txvalid signal specific to that client; the SC then sends a transaction ID over sc_client_txid bus, and the command via sc_client_txcmd. If there is any required data, it sends them over sc_client_txdata.

After the client finishes processing the command, it acknowledges the transaction by asserting client_sc_resp_ack signal; the transaction ID is also returned along with any optional response data over client_sc_resp_txid and client_sc_resp_txdata buses. Figure 2.10 shows the SCTX signals.

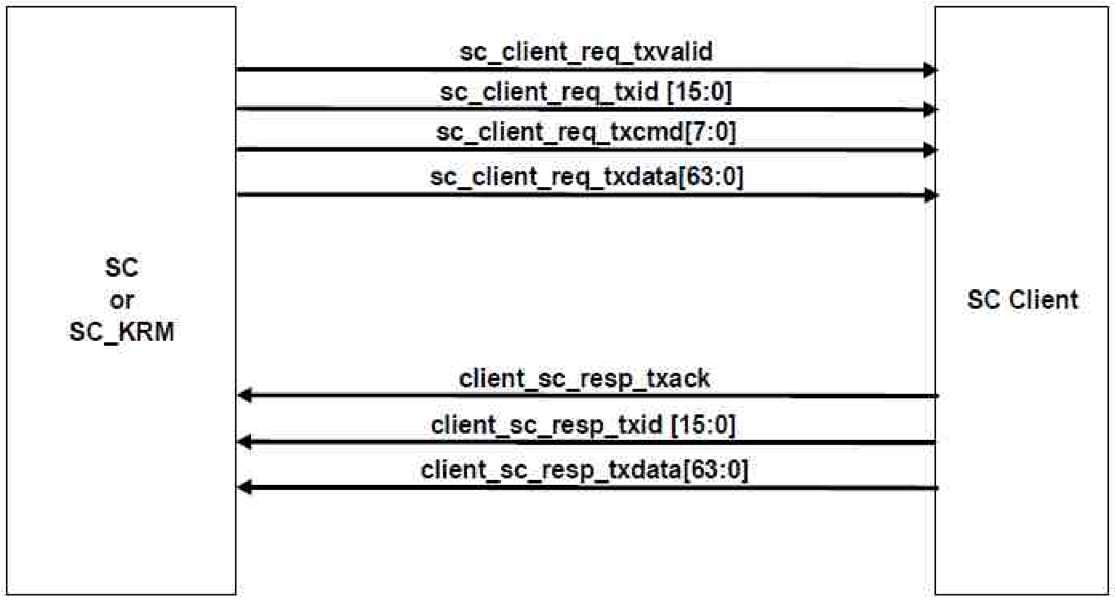

sc_client_req_txvalid
sc_client_req_txid [15:0]
sc_client_req_txcmd[7:0]
sc_client_req_txdata[63:0]
SC
or
SC_KRM
SC Client
client_sc_resp_txack
client_sc_resp_txid [15:0]
client_sc_resp_txdata[63:0]

Fig. 2.10.: Stream Controller Transaction interface (SCTX) signals (Courtesy of Vaidya [4]).

2.2.3 Command and Response FIFOs

Command/Response FIFOs act as staging units for communication between the host processor and the SCU. Requests from the host processor over HT are routed to the Command FIFO where they are staged until the Stream Controller is ready to handle these requests. Once the SC has a response, the SC enqueues it in the

Response FIFO to be read back by the host processor via HT_RDE. Figure 2.11 shows the interface signals between SC and Command/Response FIFOs.

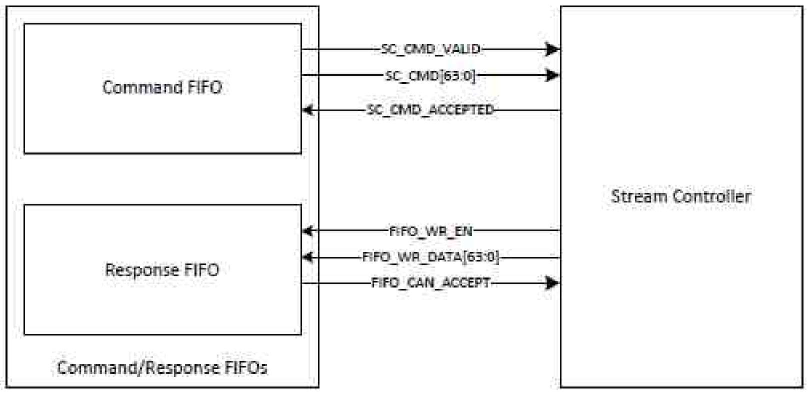


Fig. 2.11.: Interface signals between SC and Command/Response FIFOs

When a request from the host processor is queued in the Command FIFO, the SC is notified via SC_CMD_VALID signal. If the SC can accept the data, it asserts SC_CMD_ACCEPTED, and the Command FIFO dequeues and sends the data over SC_CMD lines.

Once the response is ready, the SC checks if FIFO_CAN_ACCEPT signal is asserted, which indicates the Response FIFO is not full; then the SC copies the response data to FIFO_WR_DATA and asserts FIFO_WR_EN signal.

2.2.4 Stream Register File

The Stream Register File (SRF) is a multi-client memory system used by the SCU to store input and output streams. Since the SRF is in the time critical data path, it is optimized to allow double word reads and writes in one clock cycle by the host processor.

The SRF consists of eight dual-port memory banks for a total of 512KB of storage. Each bank contains two 32-bit internal banks, namely Even-Bank and Odd-Bank.

This architecture allows for a dword data to be written or read in one clock cycle by the host processor or the EU. Figure 2.12 shows a top-level SRF architecture.

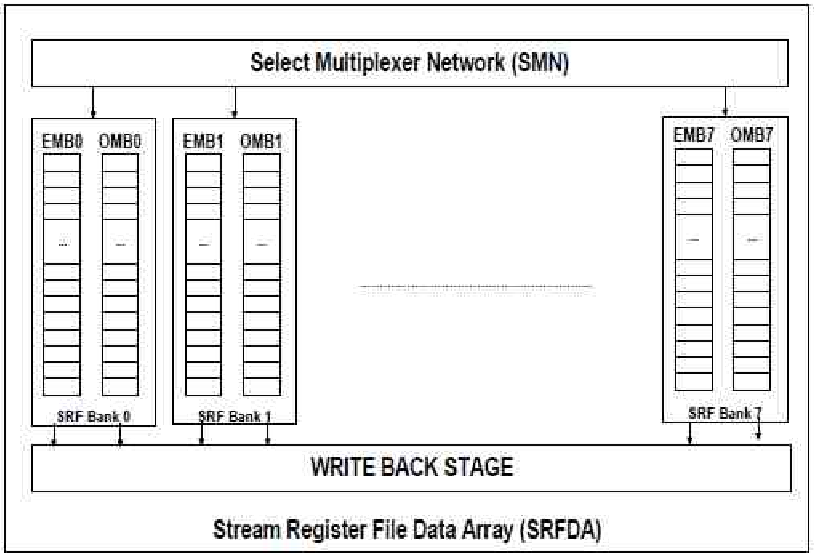

Select Multiplexer Network (SMN)
EMB0
OMB0
EMB1
OMB1
EMB7
OMB7
SRF Bank 0
SRF Bank 1
SRF Bank 7
WRITE BACK STAGE
Stream Register File Data Array (SRFDA)

Fig. 2.12.: Stream Register File (SRF) top-level architecture (Courtesy of Vaidya [4]).

The SCU has a SIMD execution unit that can execute each instruction on eight words per clock cycle; to take advantage of this capability, the stream tuples are striped across the eight SRF banks as shown figure 2.13.

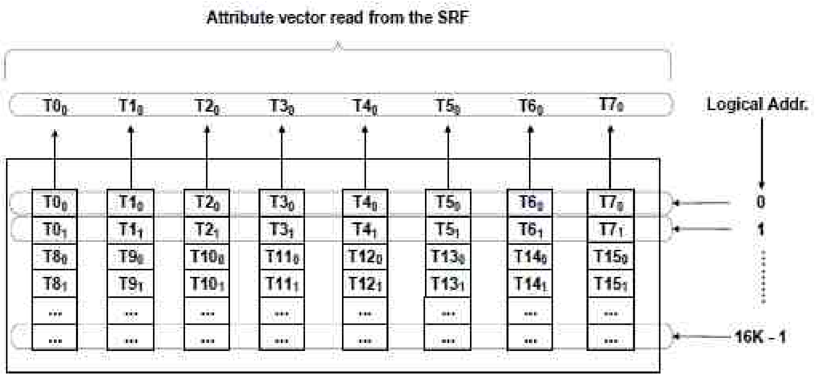


Fig. 2.13.: Organization of tuples in SRF (Courtesy of Vaidya [4]).

2.3 SymQL Query Language

To take advantage of the SIMD-VLIW execution engine in the SCU, a dedicated C-like procedural query language called SymQL [4] was developed. SymQL, like other DSMSs query languages, supports the concept of schema, tuples, and input/output streams.

Figure 2.1 shows a sample SymQL query for a filter operator. The keyword *stream* is used to declare the schema of the data stream just like the keyword *struct* used to define a data structure in C.

Query operators can be declared using the keyword *kernel*, and input and output streams can be passed to the operator as arguments with defined keywords, *input* and *output*, as shown on line 7 of Listing 2.1.

Listing 2.1: Filter Operator code in SymQL

```
1  stream STR_TYPE{
2   int   sensor_id;
3   int   tank_id;
4   float pressure;
5  };
6
7  kernel filter_kernel (input <STR_TYPE> ip, output <STR_TYPE> op)
8  {
9   while (!eos (ip, op))
10  {
11   if (ip.pressure >= 100)
12   {
13    op.sensor_id   = ip.sensor_id;
14    op.tank_id     = ip.tank_id;
15    op.pressure    = ip.pressure;
16   }
17
18  }
19 };
```

3. HARDWARE ARCHITECTURE

3.1 Introduction

In Chapter 2, we introduced the architecture of the Symbiote Coprocessing Unit (SCU) and briefly described the components related to this study. The main job of the SCU is to accelerate processing of data streams using its optimized hardware execution engine. As a result, the SCU is complex, device specific, and heavily dependent on the host processor. The major goal of the SCU redesign in this study is to reduce complexity, increase flexibility, and move some of the intelligence from the host processor to the proposed Stream Controller (SC) replacement while maintaining the SCU's high performance.

Due to the nature of the data streams, the schema of the tuples is relatively persistent; only the data is changing rapidly. As a result, the data flow through the SCU can be divided into two categories, time critical and non-time critical. Time critical execution path consists of components that process the continuously changing data in the streams, and thus, performance of these components must be optimized to meet hard deadlines. On the other hand, the non-critical configuration path consists of components used to configure the SCU. The configuration process does not happen frequently, but only when the schema of tuples change; as a result, high performance is not required for components in the configuration path. Figure 3.1 shows SCU's Configuration and Execution paths.

To maintain the current performance of the SCU, the proposed architecture implements modifications in the configuration path, while the execution path remains unchanged from the previous architecture.

In the previous design, the host processor interacts with the SCU over Hyper-Transport (HT) interface, which limits the SCU usage to systems that only support

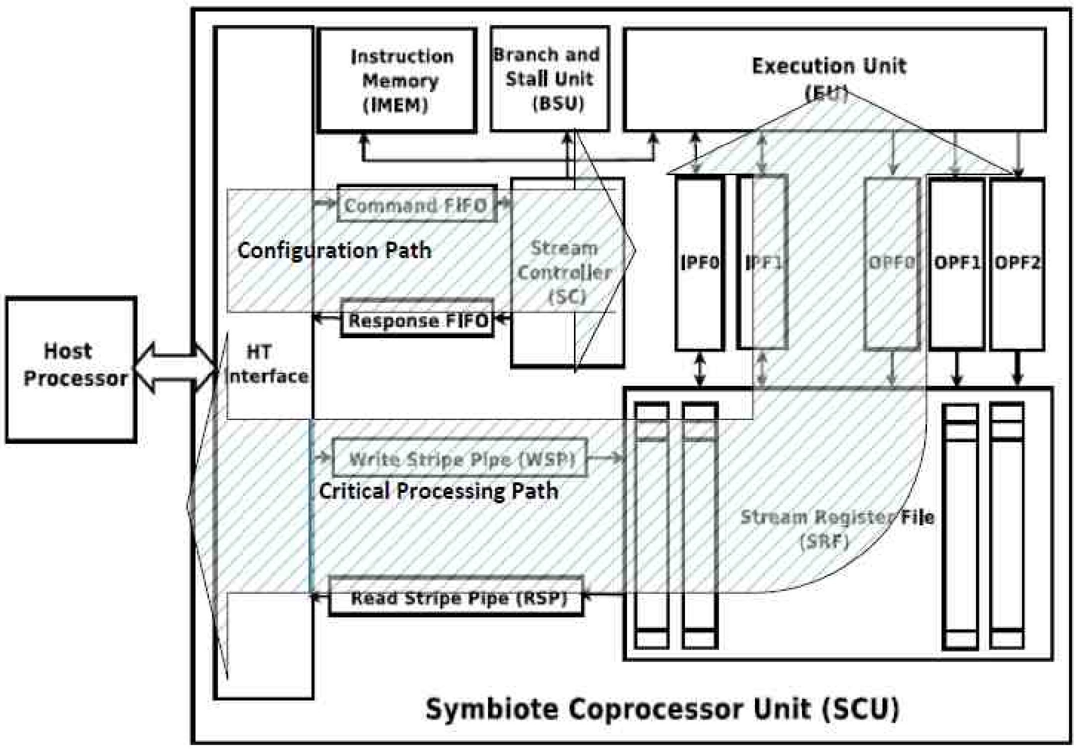


Fig. 3.1.: SCU's Execution and Configuration paths

this protocol. To overcome this limitation, the proposed architecture replaces the HT with high performance PCI Express bus.

At configuration time, the host processor programs the SCU with DSMS queries. Queries consist of one or multiple operators connected to provide the desired output. Some of these operators require preprocessing of the data in the streams, for example, aggregation operates on sorted data. Thus, the host processor sorts the data, then stores them in the SCU's SRF for processing. In some applications, reducing the data handling by the host processor, providing flexibility on the data stream sources and destinations, and offloading the data stream preprocessing can be beneficial. To provide these benefits, the proposed architecture replaces the hardware implementation of the SC controller with a microprocessor, which allows code developers to add or improve features easily.

Figure 3.2 shows the new architecture for coprocessing unit. It contains three major components, namely Stream Management Unit (SMU), Host Processor Interface (HPI), and finally a Stream Processing Unit (SPU).

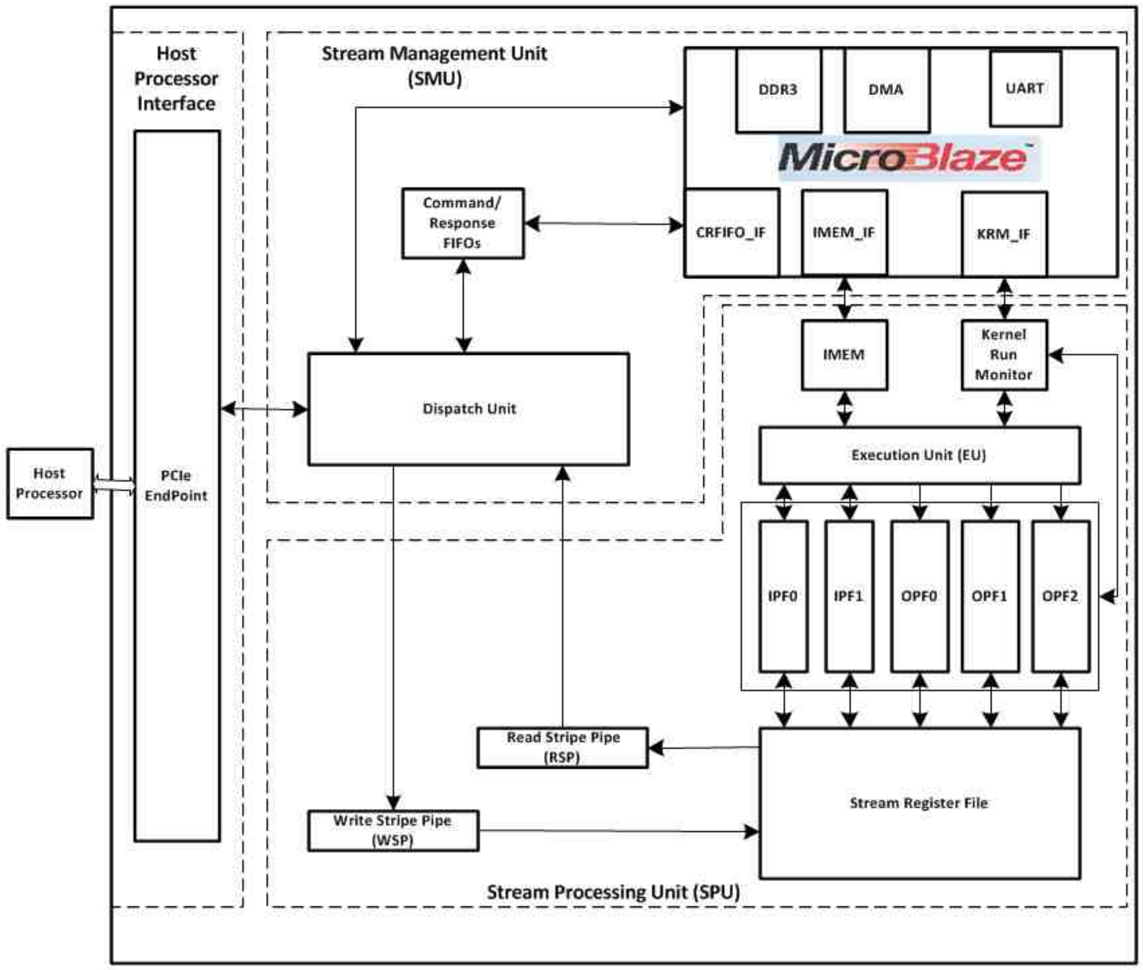


Fig. 3.2.: MicroBlaze-based coprocessing unit overall architecture

The SMU replaces the Stream Controller (SC) in the SCU architecture. It consists of a MicroBlaze controller, Advanced eXtensible Interconnect (AXI) compatible peripherals, and a Dispatch Unit.

To maintain the high performance of the SCU, the SPU uses the same SCU components identified in the time critical path shown in Figure 3.1, namely Write Stripe

Pipe, Read Stripe Pipe, Stream Register File, Input and Output FIFOs, Execution Engine, Kernel Run Monitor, and lastly the Instruction Memory.

Host Processor Interface standardizes the communication between the host processor and the co-processor, which removes the dependency on less popular protocol or architecture. PCIe is used in this implementation as an interface; however, the HPI is designed to work with other protocols.

This chapter first provides a brief introduction to MicroBlaze, AXI and PCIe in Section 3.1. Section 3.2 introduces SMU and its components. Finally, Section 3.3 examines HPI and the PCIe implementation. SPU components were introduced in Chapter 2.

3.1.1 Peripheral Component Interconnect Express (PCIe)

PCIe is a serial bidirectional computer expansion standard introduced by Intel in 2004, and represents a shift from PCI and PCIX parallel bus model. PCIe is based on point-to-point topology, with dedicated physical connection between two PCIe devices called Link. Each link contains one or more differential send and receive signal pairs, each of which is called Lane. Third generation (Gen3) of PCIe is capable of handling up to 2 GB/s per lane, and can support up to 32 lanes per link.

Figure 3.3 represents a simplified PCIe topology. At the top is the CPU, which interfaces with PCIe structure via the Root Complex. PCIe devices are connected to the Root Complex and can be one of three types, namely Switches, Bridges, and Endpoints. Switches and bridges provide expansion capabilities either by allowing more PCIe devices to connect to the Root Complex, or by providing interfaces to other buses. Endpoint devices act as initiators or completers of transactions on the PCIe bus.

PCIe devices contain multiple layers of abstraction as illustrated in Figure 3.4. Device core layer implements the core functionality of the device. It provides requests and responses to the Transaction Layer whose main responsibility is to create

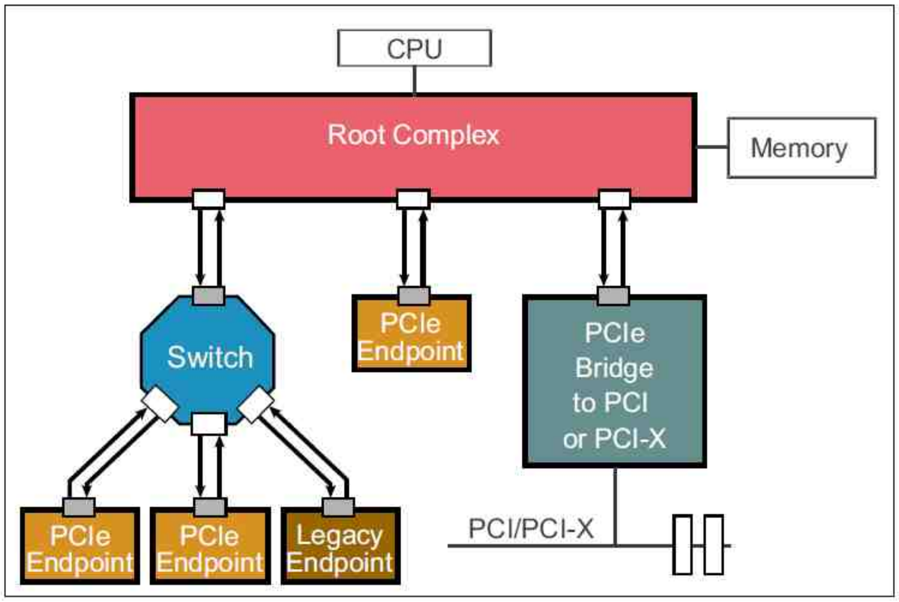


Fig. 3.3.: Example PCIe topology (Courtesy of Jackson and Budruk [8]).

Transaction Layer Packets (TLPs) on the transmit side and decodes TLPs on the receive side. Data Link Layer creates and decodes Data Link Layer Packets (DLLPs). It is also responsible for Link error detection and correction. Physical Layer is responsible for creation and decoding of Ordered-Set packets, processing all three types of packets (TLPs, DLLPs, and Ordered-Sets), and finally transmitting or receiving packets over the physical lines.

PCIe standard supports 256 buses, each bus of which can contain up to 32 devices, and each device contains up to eight functions. In order for CPU to discover all PCIe devices, it reads a defined set of registers in every PCIe device called Configuration Space. Figure 3.5 shows Configuration Space for endpoints and bridges [8].

PCIe devices expose their internal memory space via six registers called Base Address Registers (BARs) as shown in Figure 3.5. Each BAR contains information regarding its type and the size of memory it implements. The host processor reads

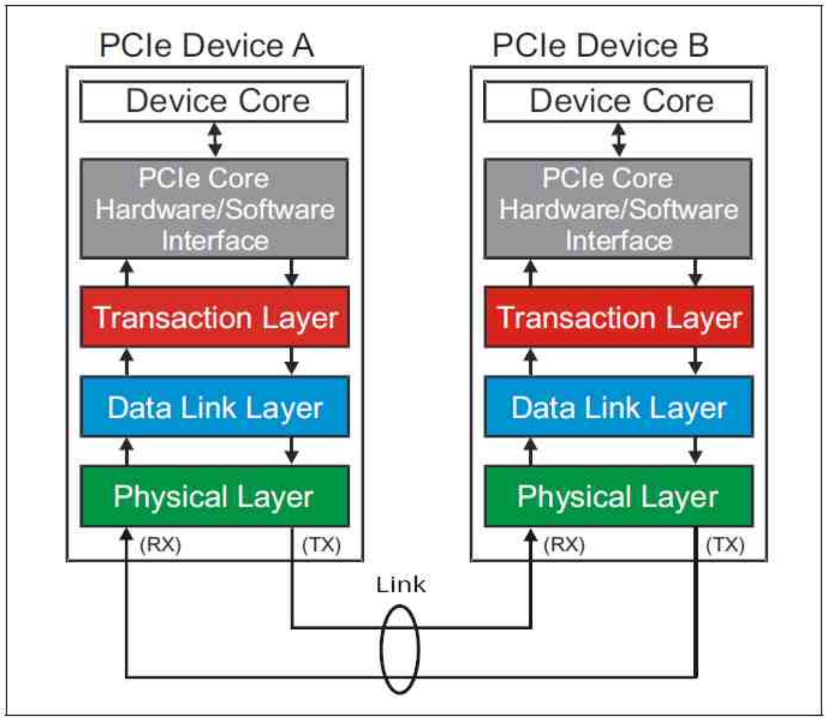


Fig. 3.4.: PCIe device layers (Courtesy of Jackson and Budruk [8]).

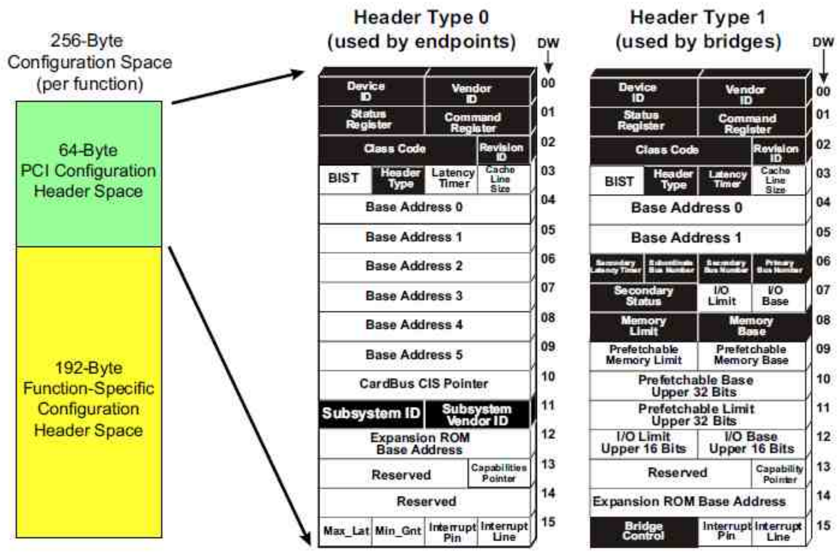


Fig. 3.5.: PCIe configuration space (Courtesy of Jackson and Budruk [8]).

the BAR configuration and assigns an address from its address space. Once each BAR has a designated address, all read and write operations issued to that address are routed to the PCIe device.

3.1.2 MicroBlaze

Xilinx MicroBlaze [6] is a 32-bit Reduced Instruction Set Computer (RISC) soft-core processor designed for implementation in FPGAs. MicroBlaze is highly configurable, allowing designers to select specific features required by their design such as memory management unit, peripherals, and bus architecture. The soft-core can be optimized for size or for performance by changing the number of pipeline stages from three to five. The overall MicroBlaze architecture is shown in Figure 3.6.

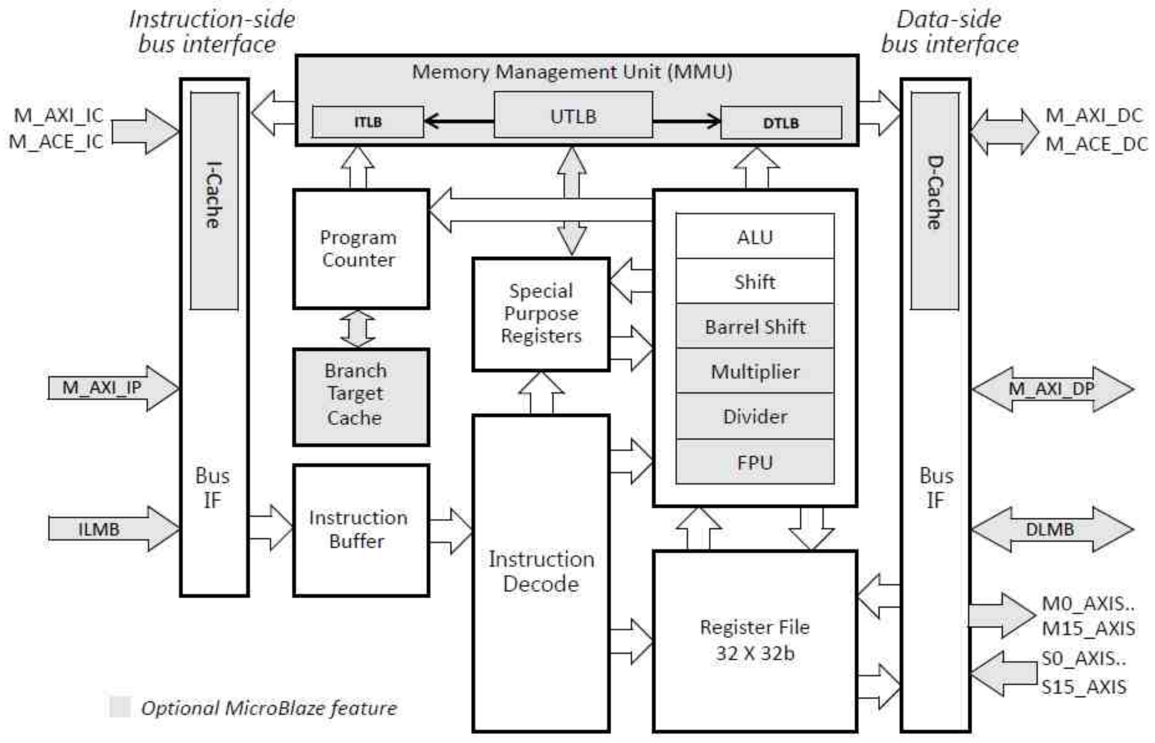


Fig. 3.6.: MicroBlaze architecture (Courtesy of XILINX [6]).

3.1.3 Advanced eXtensible Interface (AXI)

Advanced eXtensible Interconnect is an open-standard on-chip bus architecture first introduced by ARM in 2003 as part of ARM Advanced Microcontroller Bus Architecture (AMBA) family. AMBA AXI4 was introduced in 2010 and is widely used today in microcontrollers, FPGAs, and System-on-Chip (SoC) designs.

AXI4 defines three types of interfaces, which address different design needs:

- AXI4 is used for memory-mapped high performance designs.
- AXI4-Lite is used for low-throughput memory-mapped communication.
- AXI4-Stream is used for high-speed, high-throughput streaming data.

Memory-mapped AXI can be described as an interface between an AXI master and an AXI slave components. In order for multiple masters and slaves to connect, a structure called Interconnect block is used. The Interconnect block contains an AXI master and AXI slave interfaces and acts as a transaction's router between multiple AXI masters and slaves. Figure 3.7 shows multiple AXI masters and slaves connected via an AXI interconnect.

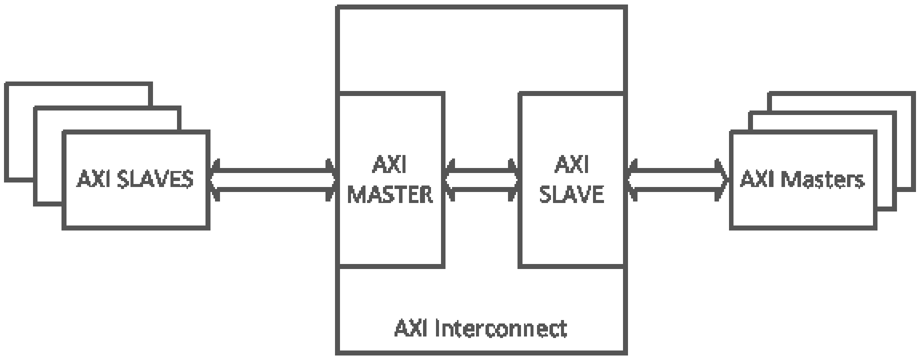


Fig. 3.7.: AXI interconnect

The interface between the AXI4 (or AXI4-Lite) master and slave consists of five different channels, namely Read and Write Address Channels, Read and Write Data

Channels, and finally the Write Response Channel. To be able to support simultaneous bidirectional data transfer, AXI4 (and AXI4-Lite) provides individual data and address channels. Figure 3.8 and Figure 3.9 show AXI4 Read and Write channel architectures, respectively.

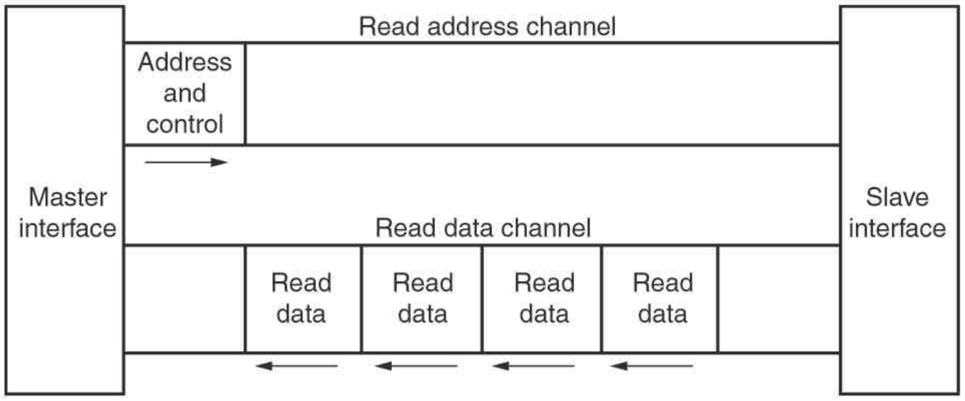


Fig. 3.8.: AXI read channel (Courtesy of XILINX [9]).

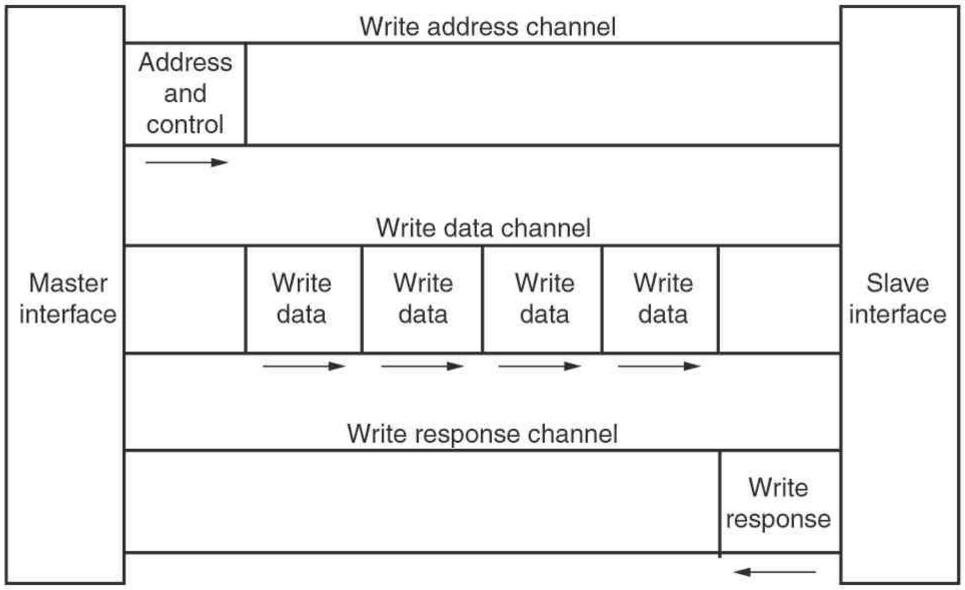


Fig. 3.9.: AXI write channel (Courtesy of XILINX [9]).

AXI4 and AXI4-Lite interfaces are used as the main interconnects between peripherals and MicroBlaze in the SMU design described in Section 3.2.

3.2 Stream Management Unit (SMU)

3.2.1 MicroBlaze Subsystem

At the core of the SMU lies the MicroBlaze, a 32-bit RISC microcontroller that runs at 125MHz with 32KB instruction and data caches, 1MB of local memory, and can access up to 4GB of DDR3 RAM. Furthermore, the MicroBlaze subsystem contains standard and application specific AXI4 memory mapped peripherals, namely a Direct Memory Access (DMA), Interrupt Controller, A Universal Asynchronous Reciever/Transmitter (UART), Kernel Run Monitor Interface (KRM_IF), Command/Response FIFO Interface (CRFIFO_IF), Dispatch Unit (DU), and an Instruction Memory Interface (IMEM_IF). Figure 3.10 shows the MicroBlaze subsystem components.

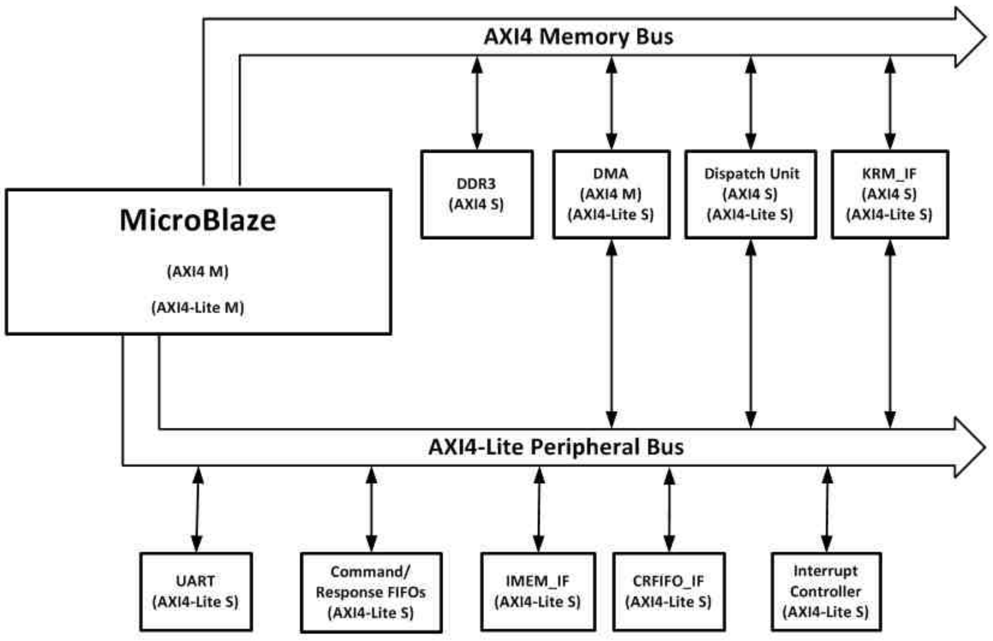

AXI4 Memory Bus
DDR3 (AXI4 S)
DMA (AXI4 M) (AXI4-Lite S)
Dispatch Unit (AXI4 S) (AXI4-Lite S)
KRM_IF (AXI4 S) (AXI4-Lite S)
MicroBlaze
(AXI4 M)
(AXI4-Lite M)
AXI4-Lite Peripheral Bus
UART (AXI4-Lite S)
Command/ Response FIFOs (AXI4-Lite S)
IMEM_IF (AXI4-Lite S)
CRFIFO_IF (AXI4-Lite S)
Interrupt Controller (AXI4-Lite S)

Fig. 3.10.: MicroBlaze subsystem components

The MicroBlaze accesses the peripherals and memories using two AXI4 buses, namely AXI4-lite for configurations and AXI4 for high throughput memory access.

As mentioned in Section 3.1.3, AXI4 is defined as a point to point interface between an AXI4 master and an AXI4 slave. To connect multiple masters and slaves, an AXI4 interconnect is used. The MicroBlaze subsystem contains two Xilinx AXI4 Interconnects [10], namely Peripherals Interconnect and Memory Interconnect.

The Peripherals Interconnect, connects the MicroBlaze's AX4-lite bus to the following peripherals for configuration:

1. Xilinx Central Direct Memory Access (CDMA) [11]: it allows the Microblaze to configure DMA transfers, enable or disable interrupts, and provides transfer status.

2. Xilinx Interrupt controller [12]: it allows the MicroBlaze to manage system interrupts.

3. Xilinx UART Lite [13]: it provides debugging and testing capabilities.

4. Kernel Run Monitor Interface: it allows the MicroBlaze to run and monitor the Execution Engine in the SPU.

5. Instruction Memory Interface: it provides the MicroBlaze with read and write access to kernel binaries stored in SPU's Instruction Memory.

6. Command/Response FIFO Interface: it allows the MicroBlaze to communicate with the host processor.

7. Dispatch Unit: it allows the MicroBlaze to configure the sources and destinations of the data streams.

The AXI4 Memory bus has two masters, namely the MicroBlaze and the DMA. They are connected to the following memories:

1. Local DDR3 RAM via Xilinx Memory Interface Generator [14].

2. KRM_IF 4KB local memory. Described in more detail in Section 3.2.2.

3. Stream Register File 512KB memory, via DU. Described in more detail in Section 3.2.5

Figure 3.11 shows the MicroBlaze subsystem implementation using Xilinx Vivado [15].

The following sections describe the application specific peripherals designed in this study. Section 3.2.2 covers the Kernel Run Monitor Interface, Section 3.2.3 describes the Instruction Memory Interface, Section 3.2.4 covers Command/Response FIFO Interface, and lastly Section 3.2.5 describes the Dispatch Unit.

3.2.2 Kernel Run Monitor Interface (KRM_IF)

KRM_IF is an AXI4 peripheral that allows the MicroBlaze to trigger execution of kernels via Kernel Run Monitor in the SPU. Furthermore, KRM_IF contains a 4KB memory space to store Kernel Descriptor File (KDR), Stream Descriptor File (SDR), and Offset Register File (ORF). Figure 3.12 shows the overall architecture of the KRM_IF.

The MicroBlaze interfaces with the KRM_IF using two AXI4 buses, namely, AXI4-lite and AXI4. The MicroBlaze uses AXI4-lite to trigger execution of specific kernel, and to read the KRM_IF status via a set of 32-bit memory mapped registers as shown in Table 3.1. To access KRM_IF's internal memory, the MicroBlaze uses the high throughput AXI4 bus. Figure 3.13 illustrates the KRM_IF internal memory map.

The host processor triggers the execution of a specific kernel by sending Run Kernel command to the SMU along with the kernel ID. The SMU must decode and execute this command as fast as possible to maintain the overall high performance of the coprocessor unit.

The KRM_IF architecture provides the MircoBlaze with fast access to SDR, KDR, and ORF at configuration time. Furthermore, due to the low latency access to con-

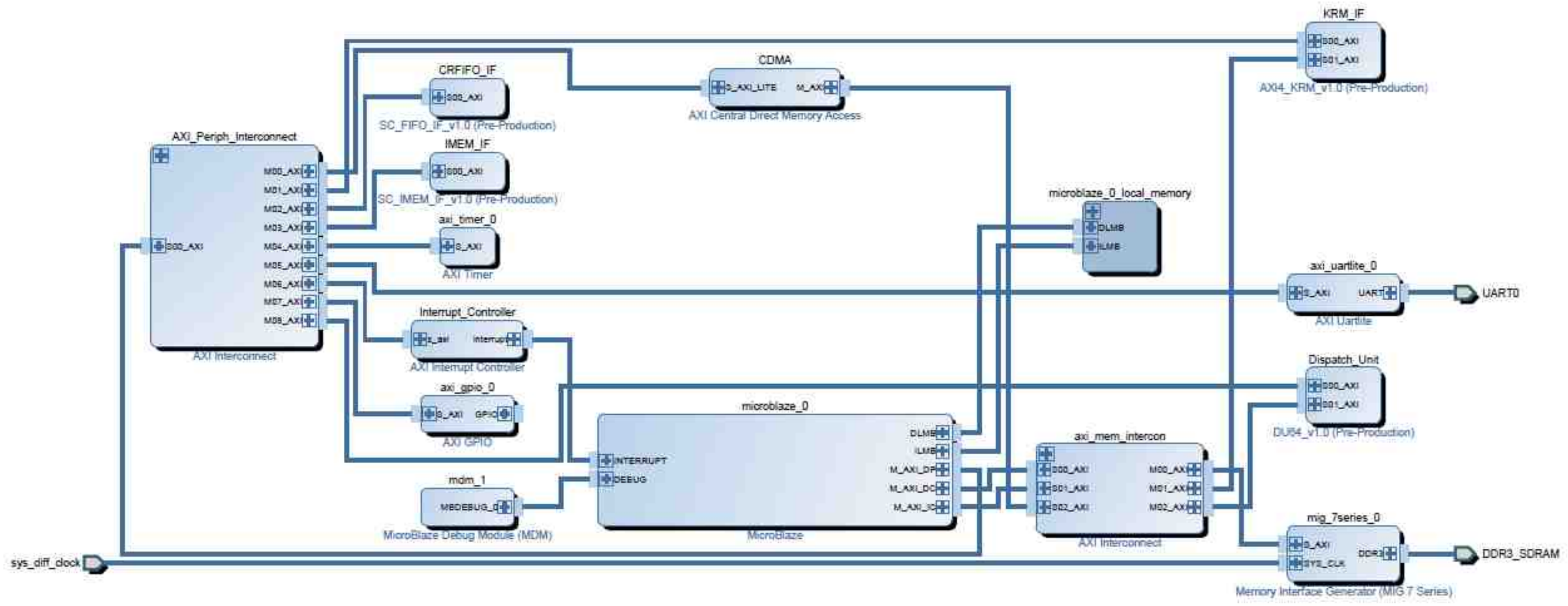


Fig. 3.11.: MicroBlaze subsystem implementation

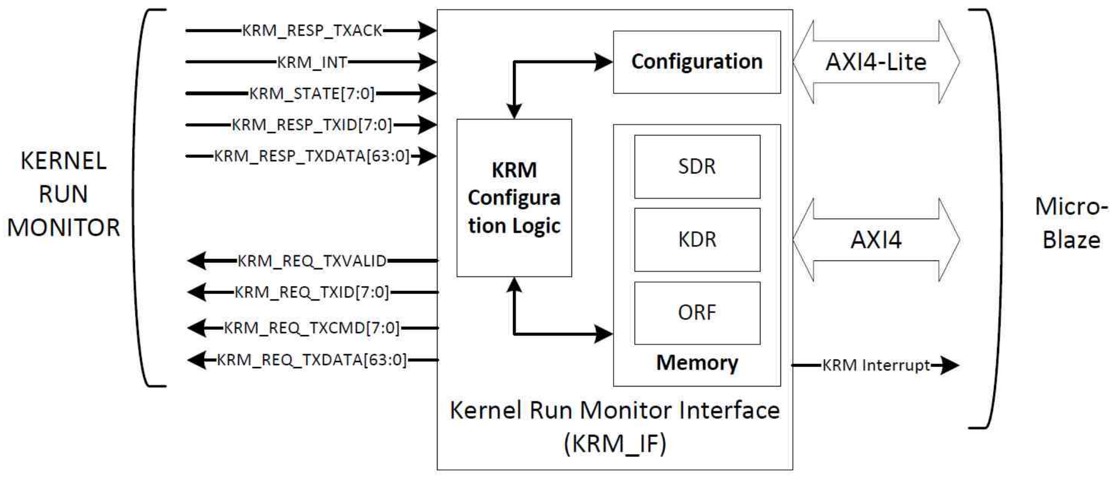


Fig. 3.12.: Kernel Run Monitor Interface (KRM_IF) architecture

Table 3.1.: KRM_IF registers

OFFSET	**Register**	**Description**
00h	KRM_CONF	KRM_IF configuration register
04h	KRM_STATE	KRM State
08h	KRM_EXC_L	Lower 32 bits of Kernel Execution Time
0Ch	KRM_EXC_H	Upper 32 bits of Kernel Execution Time

figuration data stored in its local memory, KRM_IF minimizes the time required to configure the SPU's Kernel Run Monitor (KRM) at execution time.

To execute a specific kernel, the MicroBlaze writes the Kernel ID to KRM_CONF register, enables the interrupt, and finally triggers the kernel execution by asserting the Run_Kernel bit. Table 3.2 illustrates KRM_CONF register.

KRM_IF contains a simple finite state machine to configure the KRM and trigger kernel execution as shown in Figure 3.14. When the KRM_IF receives the run kernel signal along with the kernel ID, it performs the following tasks:

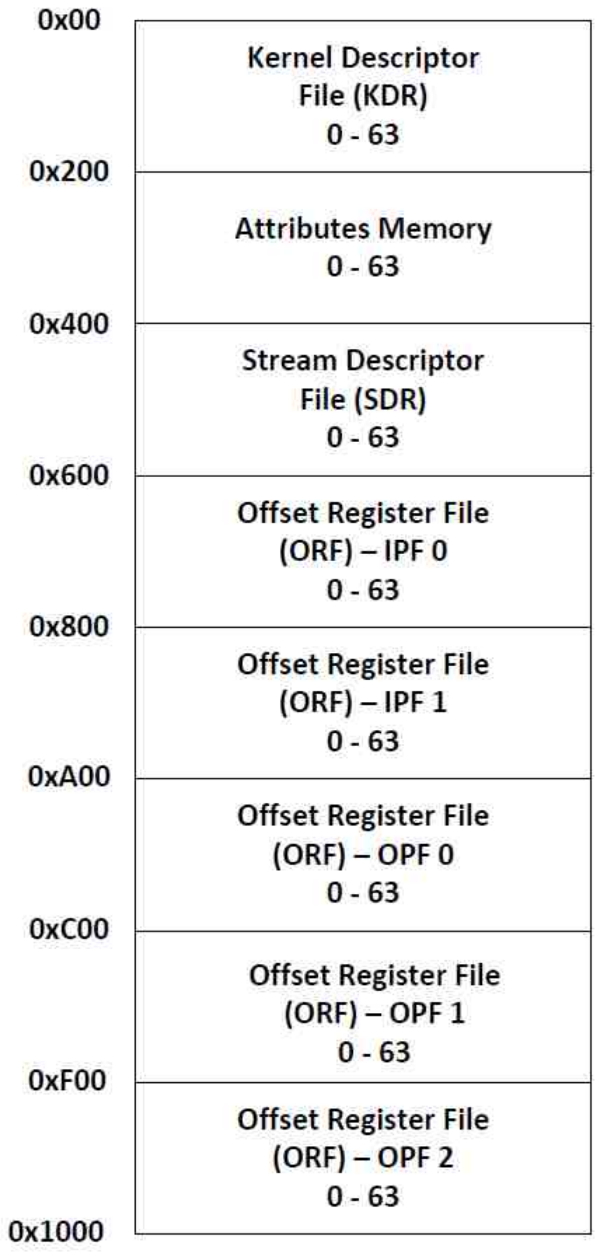

0x00
Kernel Descriptor File (KDR) 0 - 63
0x200
Attributes Memory 0 - 63
0x400
Stream Descriptor File (SDR) 0 - 63
0x600
Offset Register File (ORF) – IPF 0 0 - 63
0x800
Offset Register File (ORF) – IPF 1 0 - 63
0xA00
Offset Register File (ORF) – OPF 0 0 - 63
0xC00
Offset Register File (ORF) – OPF 1 0 - 63
0xF00
Offset Register File (ORF) – OPF 2 0 - 63
0x1000

Fig. 3.13.: Kernel Run Monitor IF (KRM_IF) internal memory map

1. Indexes local memory for the corresponding KDR and sends it to the KRM.

2. Indexes local memory for kernel specific attribute and sends it to the KRM.

Table 3.2.: Kernel Run Monitor Interface Configuration (KRM_CONF) register description

Bit	**Field**	**Value**	**Description**
0-5	ID	0 - 0x3F	Kernel ID
6	Run_Kernel	1	Write: Trigger Kernel Execution
		0	Write: Write 0 to this bit has no effect
7	INT_EN	0	Read: Interrupt is disabled
			Write: Write 0 to this bit has no effect
		1	Read: Interrupt is enabled
			Write: Enable Interrupt

3. Indexes local memory for kernel specific SDRs, and sends these SDRs to the KRM.
4. Indexes local memory for kernel specific ORFs, and sends them to the KRM.
5. Triggers the KRM to execute the specified kernel.

When the EU finishes execution of the kernel, the KRM_IF performs the following tasks:

1. Read kernel execution time from KRM.
2. Reads output stream's SDRs and updates local memory.
3. Releases the Execution Engine.

3.2.3 Instruction Memory Interface (IMEM_IF)

Instruction Memory Interface (IMEM_IF) provides the MicroBlaze with read and write access to kernel binaries stored in IMEM. IMEM_IF maps SCU specific Stream

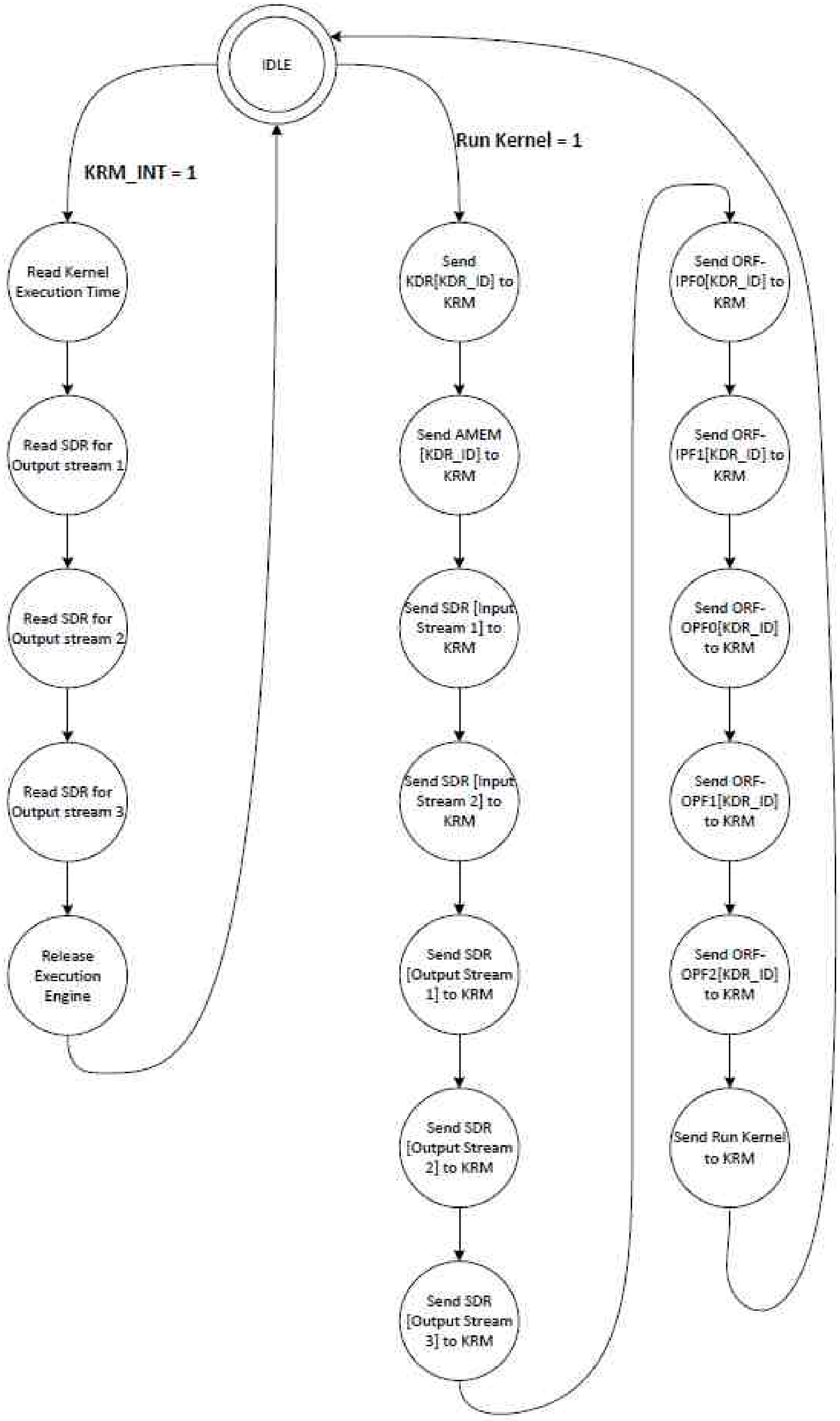

IDLE
KRM_INT = 1
Run Kernel = 1
Read Kernel Execution Time
Read SDR for Output stream 1
Read SDR for Output stream 2
Read SDR for Output stream 3
Release Execution Engine
Send KDR[KDR_ID] to KRM
Send AMEM [KDR_ID] to KRM
Send SDR [Input Stream 1] to KRM
Send SDR [Input Stream 2] to KRM
Send SDR [Output Stream 1] to KRM
Send SDR [Output Stream 2] to KRM
Send SDR [Output Stream 3] to KRM
Send ORF-IPF0[KDR_ID] to KRM
Send ORF-IPF1[KDR_ID] to KRM
Send ORF-OPF0[KDR_ID] to KRM
Send ORF-OPF1[KDR_ID] to KRM
Send ORF-OPF2[KDR_ID] to KRM
Send Run Kernel to KRM

Fig. 3.14.: Kernel Run Monitor (KRM_IF) finite state machine

Controller Interconnect (SCTX) used by IMEM to industry standard AXI4-Lite interconnect via a set of 32-bit memory-mapped registers. Figure 3.15 illustrates IMEM_IF input and output signals and Table 3.3 shows IMEM_IF registers.

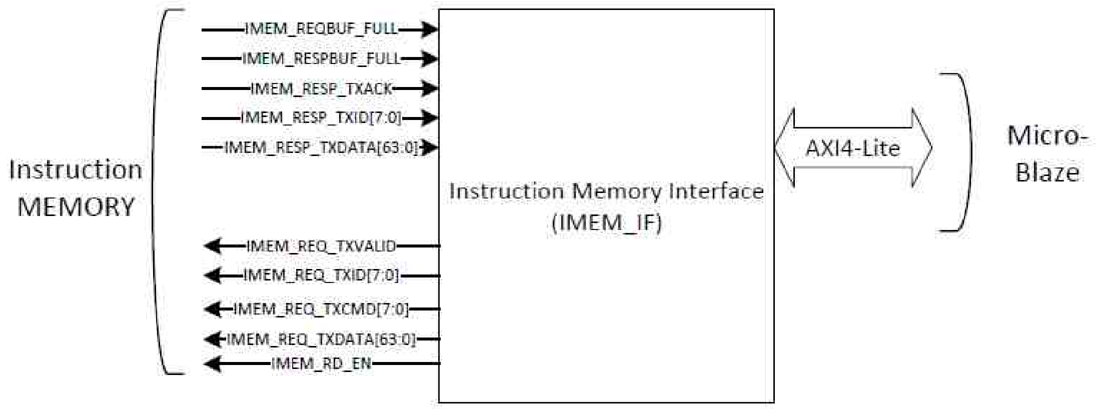


Fig. 3.15.: Instruction Memory Interface (IMEM_IF) input and output signals

Table 3.3.: IMEM_IF registers

OFFSET	**Register**	**Description**
00h	IMEM_RESP_TXDATA_L	Lower 32 bits of IMEM_RESP_TXDATA[63:0]
04h	IMEM_RESP_TXDATA_H	Upper 32 bits of IMEM_RESP_TXDATA[63:0]
08h	IMEM_REQ_TXDATA_L	Lower 32 bits of IMEM_REQ_TXDATA[63:0]
0Ch	IMEM_REQ_TXDATA_H	Upper 32 bits of IMEM_REQ_TXDATA[63:0]
10h	IMEM_REQ_CNTRL	Instruction Memory Request Control
14h	IMEM_RESP_STAT	Instruction Memory Response Status

KRM_IF allows the MicroBlaze to send SCTX transactions to IMEM. Each request transaction contains 64-bit data, transaction ID, and a command. The MicroBlaze uses registers IMEM_REQ_TXDATA_L and IMEM_REQ_TXDATA_H for transaction data, and IMEM_REQ_CNTRL for transaction ID and command. Figure 3.16 and Table 3.4 illustrate the IMEM_REQ_CNTRL register.

To send a request transaction to IMEM, the MicroBlaze sets bit TXVALID in IMEM_REQ_CNTRL and monitors bit TXACK in IMEM_RESP_STAT register, this bit indicates when the IMEM_IF receives a response from IMEM. Each response contains the same transaction ID sent with the request, and may contain response data.

The MicroBlaze accesses the response data by first setting RD_EN bit in register IMEM_REQ_CNTRL, and then reading a 64-bit response data via registers IMEM_RESP_TXDATA_L and IMEM_RESP_TXDATA_H.

IMEM contains two internal buffers, namely the Request Buffer and the Response Buffer. KRM_IF exposes the status of these buffers to the MicroBlaze. Software running on the MicroBlaze must ensure the Request Buffer is not full before sending new request, and must monitor the Response Buffer status to know when response data is available. Figure 3.17 and Table 3.5 illustrate IMEM_RESP_STAT register.

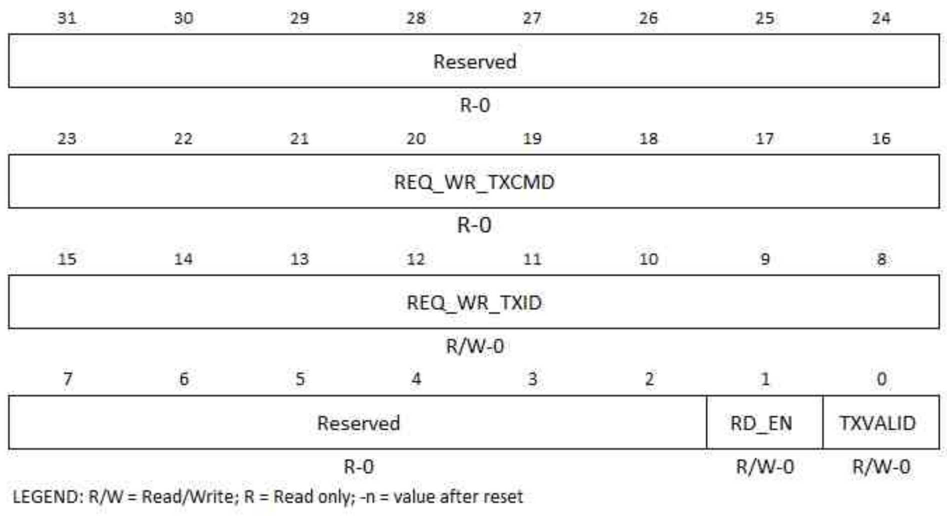


Fig. 3.16.: Instruction Memory Request Control (IMEM_REQ_CNTRL) register [offset = 10h]

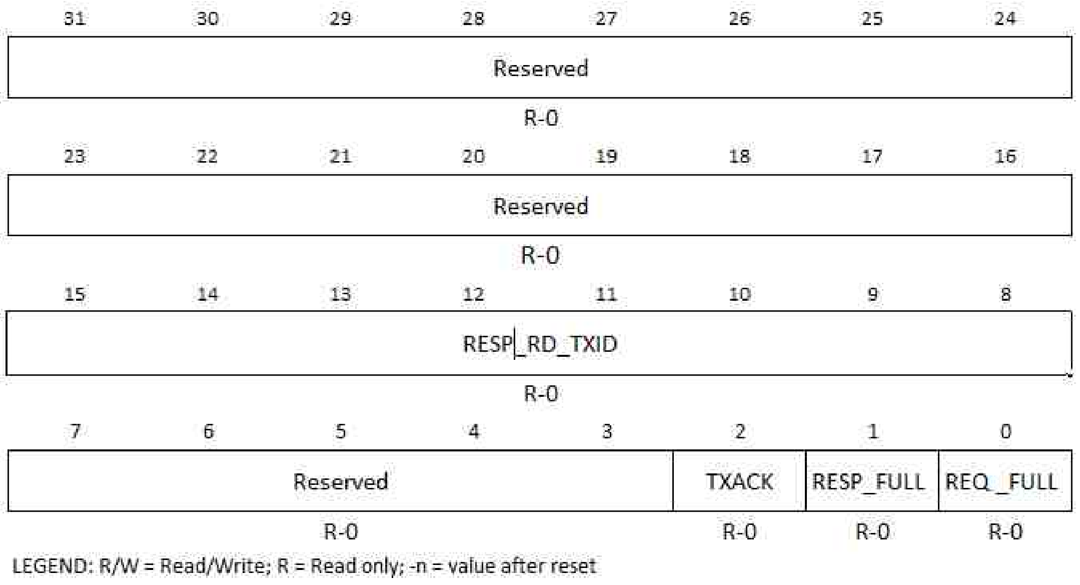
Table 3.4.: Instruction memory Request Control (IMEM_REQ_CNTRL) register field description

Bit	Field	Value	Description
0	TXVALID	0	Read: No IMEM request active
			Write: Write 0 to this bit has no effect
		1	Read: IMEM request active
			Write: Send request to IMEM
1	RD_EN	0	Read: No IMEM read active
			Write: Write 0 to this bit has no effect
		1	Read: IMEM read request active
			Write: Read DWORD from IMEM
8 - 15	REQ_WR_TXID	0 - 0xFF	Read: Current or last transaction ID
			Write: Transaction ID for current request
16 - 23	REQ_TX_CMD	0 - 0xFF	Read: Current or last request command
			Write: Command for current request

IMEM_IF contains logic to assert IMEM_REQ_TXVALID and IMEM_RD_EN signals for exactly one clock cycle to ensure synchronization and prevent multiple reads or writes of the same data.

3.2.4 Command - Response FIFO Interface (CRFIFO_IF)

Command/Response FIFO Interface (CRFIFO_IF) is a memory-mapped AXI4-lite peripheral that provides the MicroBlaze with access to commands and data queued in the Command/Response FIFOs via a set of 32-bit registers. Figure 3.18 shows the block diagram of the CRFIFO_IF and Table 3.6 shows CRFIFO_IF registers.



31	30	29	28	27	26	25	24
Reserved							
R-0							

23	22	21	20	19	18	17	16
Reserved							
R-0							

15	14	13	12	11	10	9	8
RESP_RD_TXID							
R-0							

7	6	5	4	3	2	1	0
Reserved					TXACK	RESP_FULL	REQ_FULL
R-0					R-0	R-0	R-0

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

Fig. 3.17.: Instruction Memory Response Status (IMEM_RESP_STAT) register [offset = 14h]

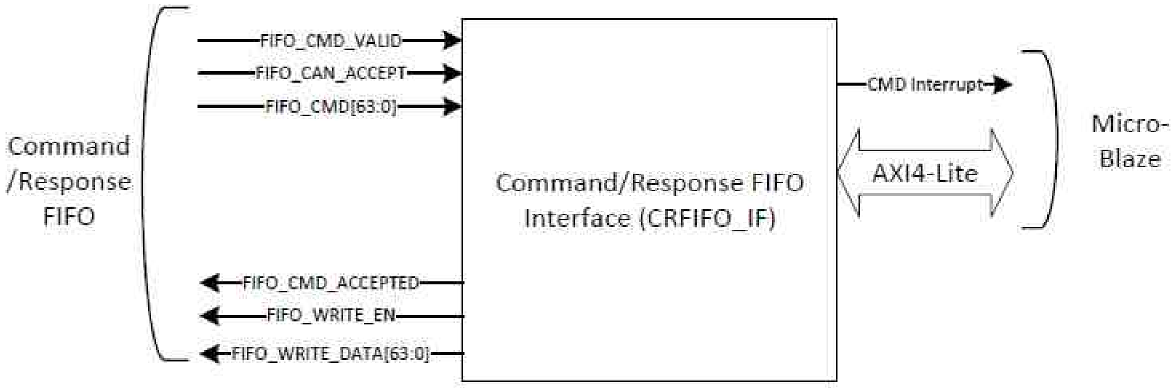


Fig. 3.18.: Command/Response FIFO Interface (CRFIFO_IF) input/output signals

CRFIFO_IF maps the commands sent by the host processor to two 32-bit read only registers. It maps the lower 32 bits of FIFO_CMD [63:0] to FIFO_CMD_L while the upper 32 bits to FIFO_CMD_H. The MicroBlaze can send response to the host processor by writing the response data to FIFO_WR_DATA_L FIFO_WR_DATA_H registers, and CRFIFO_IF then combines the registers to produce FIFO_WRITE_DATA.

Table 3.5.: Instruction memory Response Status (IMEM_RESP_STAT) register field description

Bit	**Field**	**Value**	**Description**
0	REQ_FULL	0	Read: Request buffer is not full Write: Not allowed
		1	Read: Request buffer is full Write: Not allowed
1	RESP_FULL	0	Read: Response buffer is not full Write: Not allowed
		1	Read: Response buffer is full Write: Not allowed
2	TXACK	0	Read: Request not accepted by IMEM Write: Not allowed
		1	Read: Request accepted by IMEM Write: Not allowed
8 - 15	REQ_TXID	0 - 0xFF	Read: Last accepted request ID Write: Not allowed

Furthermore, CRFIFO_IF maps configuration and status signals, namely RD_FIFO, FIFO_WRITE_EN, and FIFO_CAN_ACPT, to the FIFO_CNFG_STAT register as shown in Figure 3.19 and Table 3.7.

When new data arrives in the Command FIFO, CRFIFO_IF interrupts the MicroBlaze by asserting CMD_Interrupt signal. The MicroBlaze reads FIFO_CMD_L and FIFO_CMD_H, and then asserts CMD_ACCEPTED bit in FIFO_CNFG_STAT register to allow for next command to be dequeued.

Table 3.6.: CRFIFO_IF registers

OFFSET	Register	Description
00h	FIFO_CMD_L	Lower 32 bits of FIFO_CMD[63:0]
04h	FIFO_CMD_H	Upper 32 bits of FIFO_CMD[63:0]
08h	FIFO_WR_DATA_L	Lower 32 bits of FIFO_WRITE_DATA [63:0]
0Ch	FIFO_WR_DATA_H	Upper 32 bits of FIFO_WRITE_DATA [63:0]
10h	FIFO_CNFG_STAT	FIFO Configuration and Status

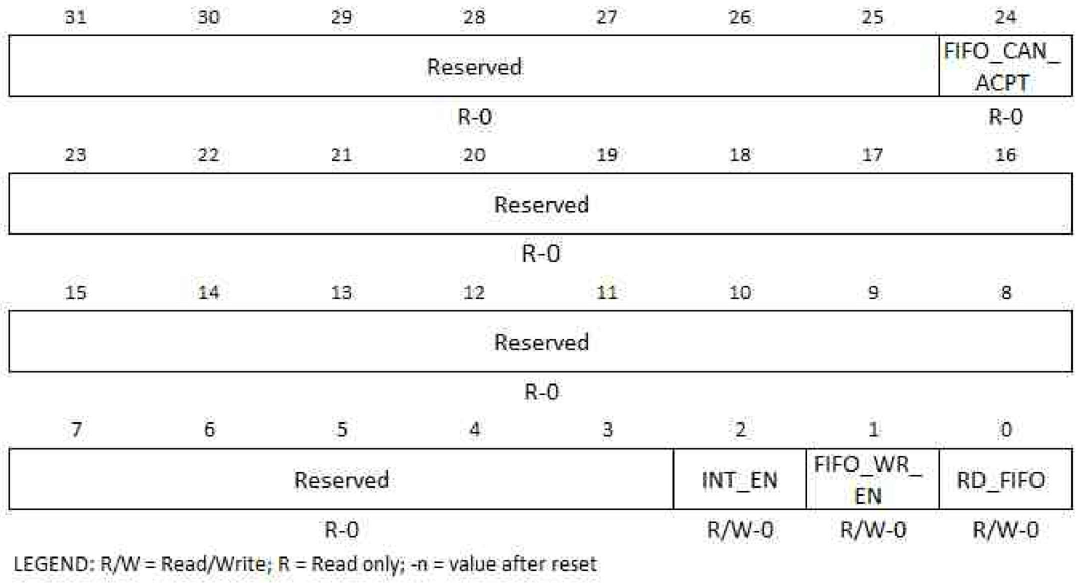


Fig. 3.19.: FIFO_CNFG_STAT [offset = 10h] register

Software running on the MicroBlaze must first check if the response FIFO can accept response data, then copies the response data to FIFO_WR_DATA_L and FIFO_WR_DATA_H registers, and finally asserts FIFO_WR_EN bit.

CRFIFO_IF asserts FIFO_CMD_ACCEPTED and FIFO_WRITE_EN signals for exactly one clock cycle to ensure synchronization and prevent FIFO overrun.

Table 3.7.: FIFO Configuration and Status (FIFO_CNFG_STAT) register field description

Bit	Field	Value	Description
0	RD_FIFO	0	Read: Command FIFO can accept request
			Write: Write 0 to this bit has no effect
		1	Read: Command FIFO is busy
			Write: Read next DWord from Command FIFO
1	FIFO_WR_EN	0	Read: No write request issued to Response FIFO
			Write: Write 0 to this bit has no effect
		1	Read: Response FIFO is busy
			Write: Write DWord to Response FIFO
2	INT_EN	0	Read: Interrupt is disabled
			Write: Write 0 to this bit has no effect
		1	Read: Interrupt is enabled
			Write: Enable Interrupt
24	FIFO_CAN_ACPT	0	Read: Response FIFO is full
			Write: Not allowed
		1	Read: Response FIFO can accept write requests
			Write: Not allowed

3.2.5 Dispatch Unit (DU)

Figure 3.20 illustrates the Dispatch Unit (DU) architecture. The (DU) serves two purposes. First, it provides the Host Processor Interface (HPI) with standardized interconnect. Second, it exposes the SPU address space to the MicroBlaze. The DU consists of four major components, namely HTRDE, HTWRE, Stream Router (SR), and AXI4 Transaction Converter (AXITC). HTRDE and HTWRE provide read and write accesses to SPU internal memory space. These are the same components used in the previous SCU architecture and covered in Chapter 2. The SR allows the MicroBlaze to choose the source of the input streams, either the host processor or the local RAM memory. Similarly, the MicroBlaze can configure the destination of the output streams to be the host processor or the RAM. The AXITC converts AXI4 read and write transactions to HTRDE and HTWRE specific transactions, respectively.

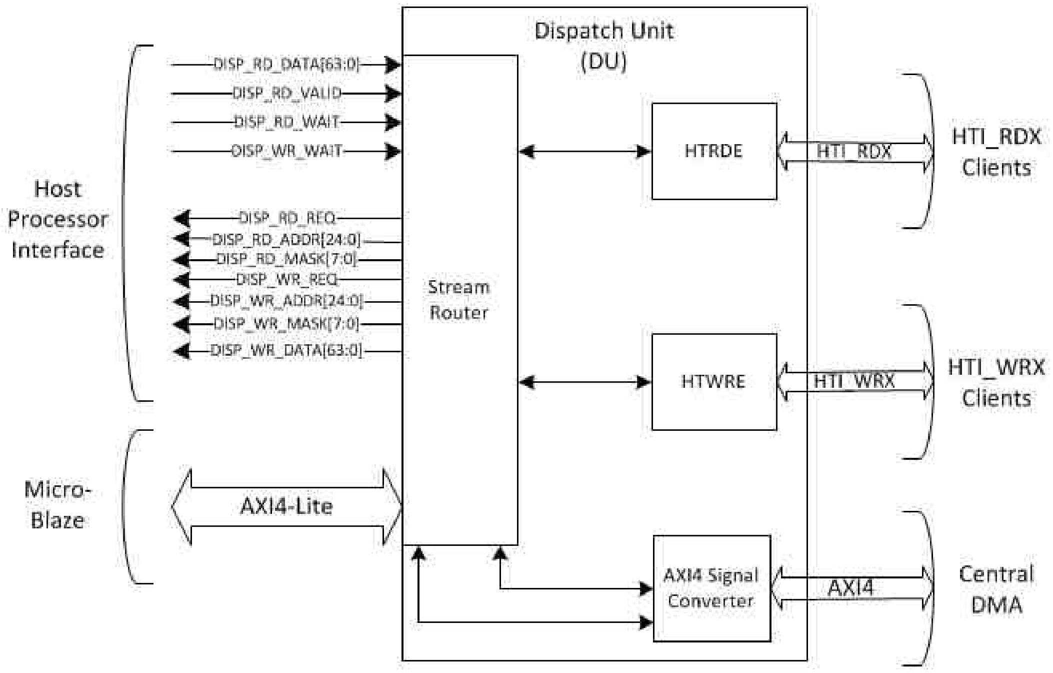


Fig. 3.20.: Dispatch Unit (DU) architecture.

The SR contains one 32-bit memory-mapped register called Dispatch Unit Configuration (DU_CONF) register. The MicroBlaze accesses this register over AXI4-lite interface, and uses it to configure the DU. Table 3.8 illustrates the DU_CONF register.

If the application requires the source of input streams and destination of output streams to be the host processor, the DU does not need to be configured by the MicroBlaze as this is the default DU settings. The host processor can access HT_RDX and HT_WRX clients directly.

The DU provides a high throughput AXI4 interface which allows fast and efficient data transfer from and to the local RAM. AXI4 bus is capable of burst transaction [7],

Table 3.8.: Dispatch Unit Configuration (DU_CONF) register description

Bit	Field	Value	Description
0-7	WSP_SIZE	0 - 0xFF	Write Strip Size in number of attributes
8-15	RSP_SIZE	0 - 0xFF	Read Strip Size in number of attributes
16	CONF_WSP	0	Read: WSP ready for configuration
			Write: Write 0 to this bit has no effect
		1	Read: Configuration of WSP is in progress
			Write: Configure WSP
17	CONF_RSP	0	Read: RSP ready for configuration
			Write: Write 0 to this bit has no effect
		1	Read: Configuration of RSP is in progress
			Write: Configure RSP
18	IN_SRC	0	Read: Input stream source is host processor
			Write: Sets input stream source to host processor
		1	Read: Input stream source is DMA
			Write: Sets input stream source to DMA
19	OUT_DEST	0	Read: Output stream destination is host processor
			Write: Sets Output stream destination to host processor
		1	Read: Output stream destination is DMA
			Write: Sets output stream destination to DMA

where each transaction contains up to 256, 64-bit data payload. Since the MicroBlaze has a 32-bit data bus, reading or writing a Dword requires two transfers. To overcome this limitation, the DU is connected to a DMA with 64-bit data bus instead.

The DU architecture exposes only the SRF to the DMA. The MicroBlaze must first configure the WSP in case of write to SRF, or the RSP in case of read from SRF before initiating a DMA transfer. To configure the WSP, the MicroBlaze writes the stream's tuple size (number of attributes per tuple) to WSP_SIZE in DU_CONF register. Similarly, The MicroBlaze configures the RSP by writing the stream's tuple size to RSP_SIZE bits.

To better explain the architecture of the DU, we provide the following two examples.

Example 1 *Output stream destination is UART*

Suppose we have an application where the host processor provides two input streams, and requires the output stream to be sent over a UART to an external display. The MicroBlaze must first set the source of input streams to the host processor, and the destination of the output stream to its local DDR3 RAM. After the kernel execution is completed, the MicroBlaze reads the SDR for the output stream. The SDR contains the stream start address in SRF, the tuple size, and the tuple count. The MicroBlaze uses the information provided by the SDR to configure the RSP with the tuple size, and the DMA with source address in the SRF, destination address in the RAM, and the number of bytes to transfer. Once the DMA finishes the transfer, the MicroBlaze sends the output stream from its local RAM to the external display over UART.

Example 2 *Input stream source is Ethernet*

Suppose the source of input stream is the MicroBlazes ethernet port and the destination of the output stream is the host processor. First, the MicroBlaze sets the source of input streams to Local RAM and the destination of output streams to the host processor. As the ethernet port copies new data streams to the local RAM, the MicroBlaze divides the data streams in RAM into windows. After that, the MicroBlaze configures the WSP with the number of attributes in each tuple, the DMA with start address in RAM, destination address in SRF, and the byte count. The DMA transfers the input stream tuples to SRF and notifies the MicroBlaze when the transfer is finished. Finally, The MicroBlaze triggers the kernel execution and notifies the host processor when the output stream is ready to be read from SRF via PCIe bus. The hardware architecture of this example is covered in Section 6.2.1

3.2.6 Host Processor Interface (HPI)

Figure 3.21 illustrates the architecture of the HPI. It contains two components, namely Dispatch Unit Interface (DU_IF), and Xilinx PCIe Integrated Block [16].

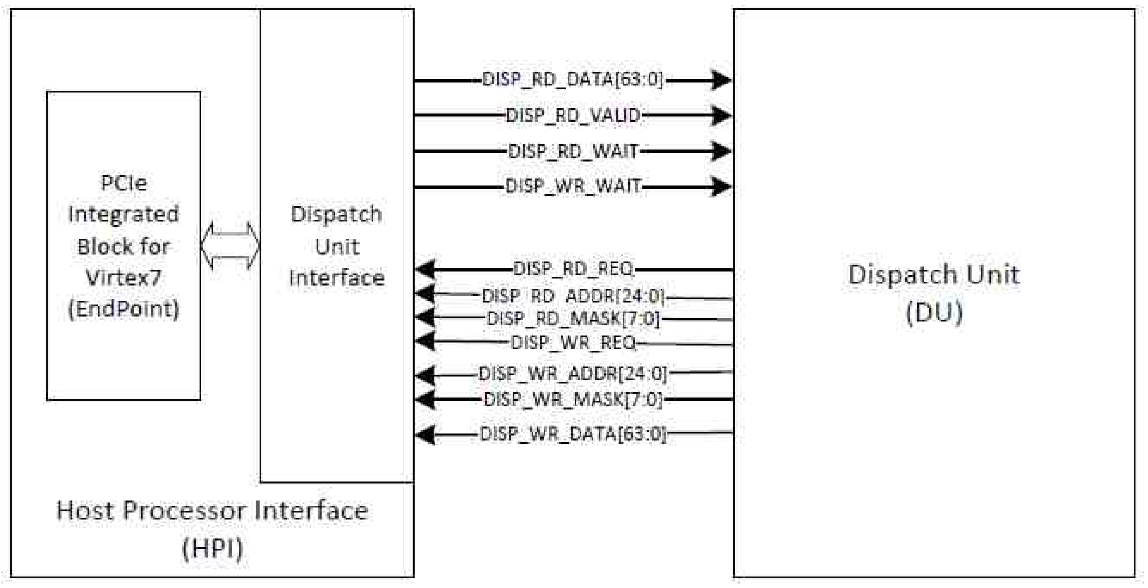


Fig. 3.21.: Host Processor Interface (HPI) architecture

To eliminate dependency on the host interface protocol, the DU_IF maps the PCIe signals to Dispatch Unit compatible interface. As a result, future host interfaces can be easily integrated without modifying internal components of the MicroBlaze-based coprocessor.

The Third Generation (Gen3) of PCIe supports bit rate of 8GT/s using 128b/130b encoding [8]. The total bidirectional bandwidth is 2GB/s per lane. PCIe can support 32 lanes per link for a total of 64GB/s.

Due to SCU's internal propagation delay, the max frequency it can operate at is 125MHz [4]. To stay compatible with the SCU's data width of 64-bit, Gen3 PCIe can have one lane at that frequency.

As mentioned in Section 3.1.1, PCIe devices expose their internal memory space via six 32-bit registers called Base Address Registers (BARs). Each BAR can ad-

Table 3.9.: PCIe Endpoint BAR configuration

BAR	**Addressable Space**	**Usage**
0,1	16KB	Command/Response FIFO
2,3	512KB	Stream Register File
4,5	512KB	Reserved for second Stream Register File

dress up to 4GB of memory space. To be able to support 64-bit addressing used by most current computers, two BARs can be combined. Table 3.9 illustrates the PCIe endpoint BAR configuration for this study.

4. SOFTWARE ARCHITECTURE

In Chapter 3 we described the hardware architecture of the MicroBlaze-based data stream coprocessor. One of the study objectives is to reduce the complexity of the finite state machine that runs the SCU by using software implementation instead of hardware.

The coprocessor is designed to accelerate processing of data streams by offloading the data and computation intensive tasks from the main CPU. To utilize the coprocessor, the CPU must first configure it. After configuration is finished, the host processor loads the coprocessor with the input stream(s), triggers query execution and finally reads back the output stream(s) from the coprocessor. Configuration only happens when the query to be executed or the description of the streams is changed. Figure 4.1 illustrates stream processing sequence.

Configuring the coprocessor consists of providing the Stream Management Unit (SMU) information about the data streams both input and output, also information describing the kernel to be executed on these streams. For each stream (input or output) the SMU needs to know the location of the data in the Stream Register File (SRF), the schema of the tuples, and the size of the stream. In order for the SMU to execute kernels, it first needs to know the kernel configuration, which consists of the kernel ID, location of the kernel binaries in the instruction memory, size of the kernel, and finally the kernel's input and output streams.

Once the kernel information is provided to the SMU, the kernel binaries can be copied to the instruction memory. Kernel binaries consist of instructions, constants used in the query, and compiler optimizations specific to the kernel. Figure 4.2 illustrates SMU configuration sequence.

The coprocessor's internal stream storage is exposed to the host processor over PCIe. The host processor can read and write the SRF without interaction with the

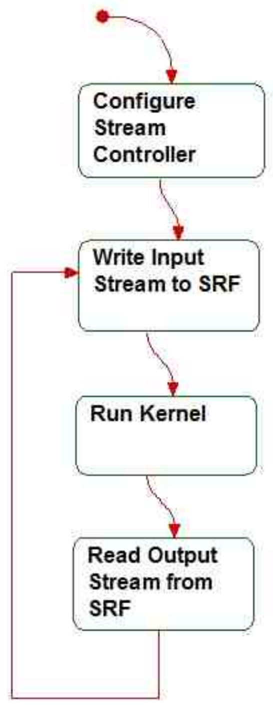

Configure Stream Controller
Write Input Stream to SRF
Run Kernel
Read Output Stream from SRF

Fig. 4.1.: Stream processing sequence

coprocessor's SMU. Once stream data are copied, the host processor triggers execution of a kernel.

The host processor triggers kernel execution by sending a run kernel command containing kernel ID. Once the SMU receives the command, it configures the Execution Engine (EU) via Kernel Run Monitor Interface (KRM_IF). Kernel information stored in the KRM_IF is copied to the Kernel Run Monitor (KRM) along with input and output stream's configurations related to the identified kernel. Finally, SMU issues a run kernel command to KRM and waits for execution to be completed. After kernel execution is finished, the host processor is notified and results are read back.

Figure 4.3 illustrates the proposed software architecture to configure and run the coprocessor. It consists of a main software component called Stream Manager (SM), peripheral drivers, namely Kernel Run Monitor driver, Instruction Memory driver, Command/Response FIFO driver, and Dispatch Unit driver. These software drivers provide the SM with access to hardware components in the Stream Processing Unit

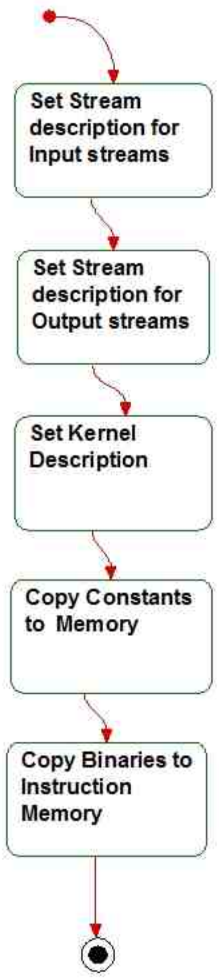

Set Stream description for Input streams
Set Stream description for Output streams
Set Kernel Description
Copy Constants to Memory
Copy Binaries to Instruction Memory

Fig. 4.2.: SMU configuration sequence

(SPU). The architecture also contains an Interrupt Controller to handle interrupts and a Register Access component with a UART driver for software debugging and testing.

The software execution is driven by two interrupts, specifically KRM Interrupt (KRM_Int) and Command/Response FIFO's Interrupt (CRFIFOs_Int). KRM_Int interrupts the SM after the EU finishes kernel; KRM_INT is configured as Fast Interrupt and the highest priority. CRFIFOs_Int notifies the SM when new data is available from the host processor in the Command/Response FIFOs.

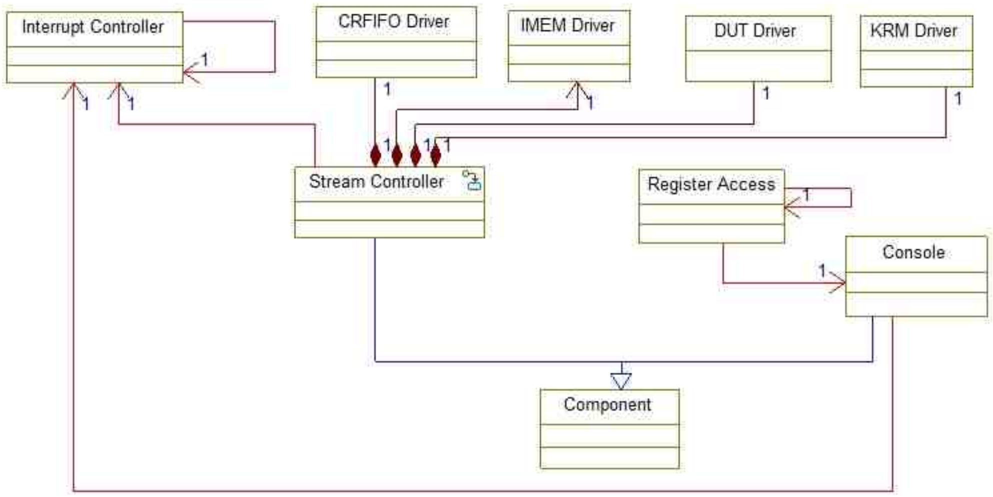


Fig. 4.3.: SMU software architecture

In this chapter, we first describe the software drivers in Section 4.1, and then Section 4.2 describes the SM and its state machine.

4.1 Software Drivers

4.1.1 Kernel Run Monitor Software Driver

KRM software driver allows the SM to configure the KRM hardware component via KRM_IF peripheral. It provides the SM with five functions, namely Enable_Interrupt, Disable_Interrupt, Read_Kernel_Execution_Time, Get_KRM_State, and finally Run_Kernel. Enabling and disabling the KRM_Int is accomplished by asserting or resetting the INT_EN bit in KRM_CONF register. The Get_KRM_State method reads the KRM_STATE bits in KRM_STATE register and returns it to the SM.

The main function of the KRM driver is to trigger kernel execution in the EU. Run_Kernel function sets the kernel ID and asserts the Run_Kernel bit in KRM_CONF register. Once execution is completed, the Read_Kernel_Execution_Time function combines KRM_EXC_L and KRM_EXC_H registers and returns execution time in clock cycles.

Figure 4.4 shows the flowchart Run_Kernel function.

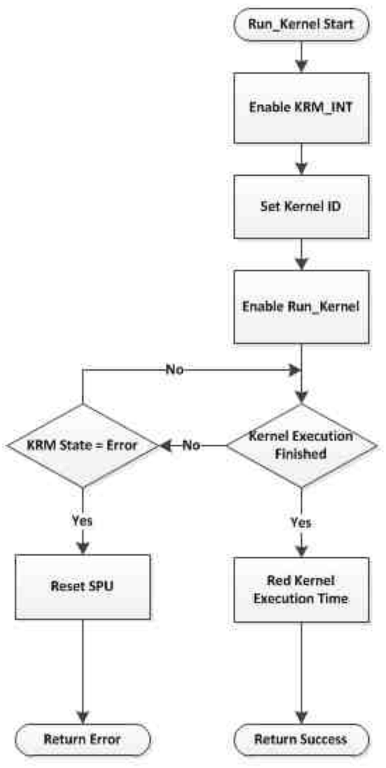


Fig. 4.4.: KRM Run_Kernel function flowchart

4.1.2 Instruction Memory Interface Software Driver

IMEM interface software driver provides the SM with methods to read and write binaries to the Instruction Memory via IMEM_IF peripheral. It abstracts reading and writing to both instruction and constant memory arrays, also, handling the status signals. Figure 4.5 illustrates the flowchart for writing to IMEM. The IMEM driver first configures the IMEM with the number of instructions and the start address; it then starts copying the binaries to IMEM. The same procedure also applies to writing constants.

The IMEM driver reads kernel binaries by first configuring the IMEM for read operation with start address and number of instructions. It then checks if the IMEM

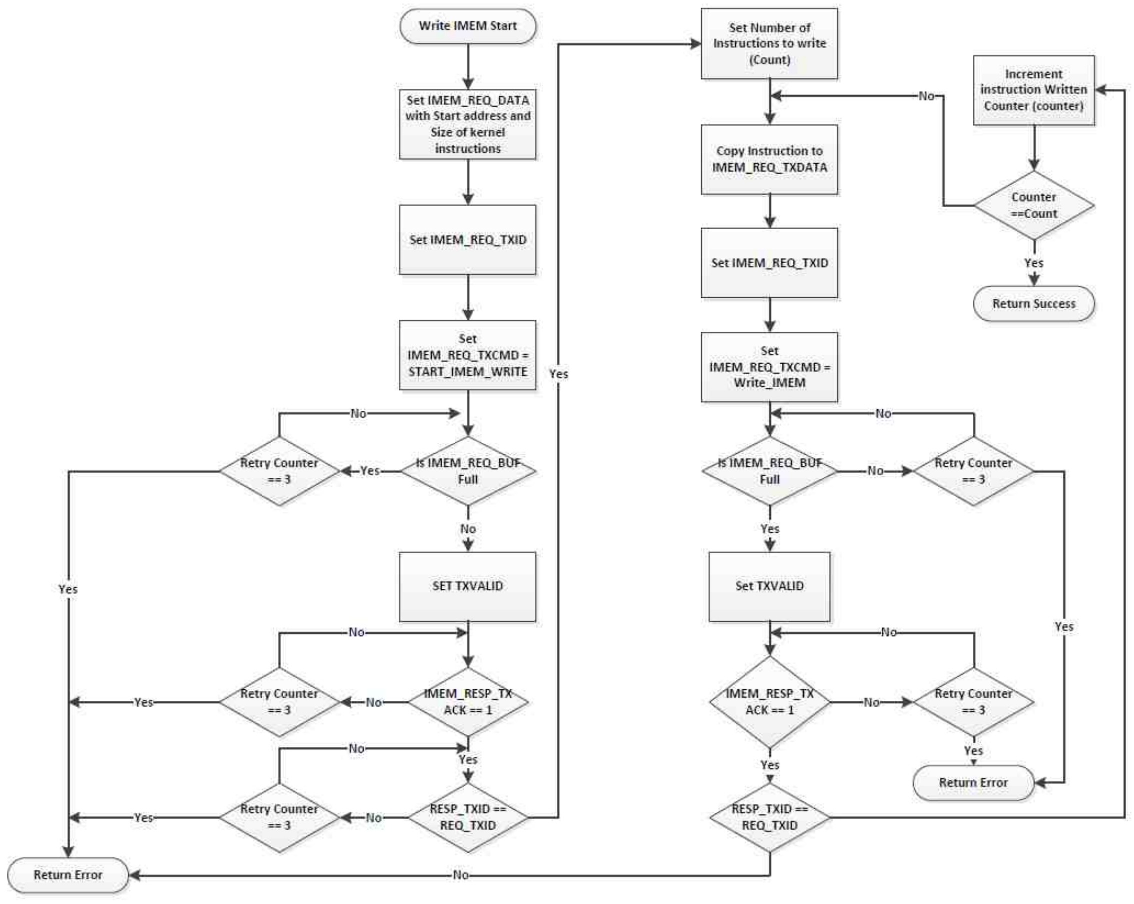

Write IMEM Start
Set IMEM_REQ_DATA with Start address and Size of kernel instructions
Set IMEM_REQ_TXID
Set IMEM_REQ_TXCMD = START_IMEM_WRITE
Is IMEM_REQ_BUF Full
Retry Counter == 3
SET TXVALID
IMEM_RESP_TX ACK == 1
RESP_TXID == REQ_TXID
Return Error
Set Number of Instructions to write (Count)
Copy Instruction to IMEM_REQ_TXDATA
Set IMEM_REQ_TXID
Set IMEM_REQ_TXCMD = Write_IMEM
Set TXVALID
Increment instruction Written Counter (counter)
Counter ==Count
Return Success
Yes
No

Fig. 4.5.: Write IMEM flowchart

response buffers are not empty, and finally setting the read enable bit. Reading from IMEM is implemented for debugging purposes only.

4.1.3 Command/Response FIFO Software Driver

The SM communicates with the host processor via CRFIFO_IF. CRFIFO software driver provides SM with methods to receive host processor commands, and to reply with data if requested.

CRFIFO driver interrupts (if enabled) the SM when new commands or data are received by the Command FIFO. Once the SM reads the command, the driver sets the read FIFO (RD_FIFO) bit in FIFO_CONFIG_STAT register to de-queue the next set of data if available.

When the SM sends data to the host processor, the CRFIFO driver first checks if the Response FIFO is not full, and then copies the data and asserts the write enable signal. Figure 4.6 illustrates the read and write operations of the CRFIFO driver.

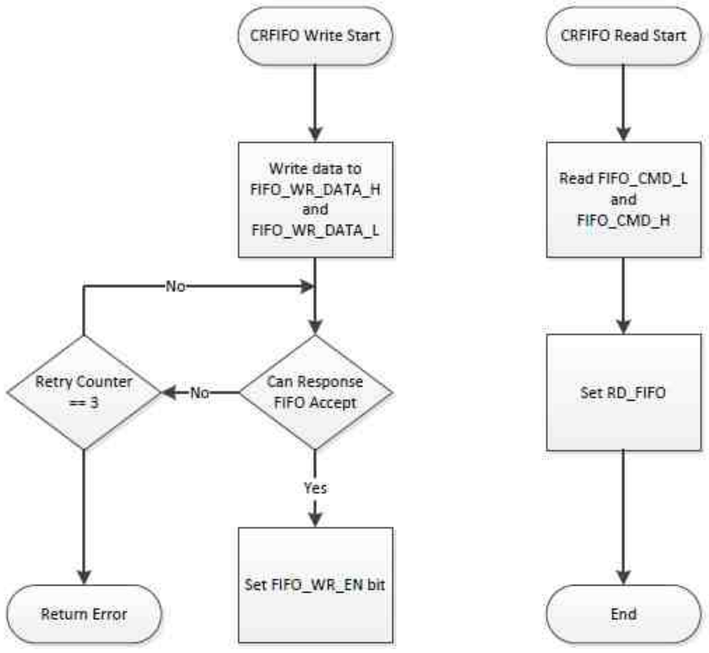


Fig. 4.6.: CRFIFO read and write flowchart

4.1.4 Dispatch Unit Software Driver

Dispatch Unit (DU) software driver exposes the SPU's internal memory to the SM. Data can be read and written by the SM through two methods, namely WriteMemory and ReadMemory. Furthermore, the driver allows SM to control the source of the input streams and the destination of the output streams.

Since the DU is connected to the DMA, the DU driver configures the DMA before moving data between the SRF and the local RAM. Figure 4.7 illustrates the flowchart for DU's driver WriteMemory method. SM provides the source address in RAM, the destination address in SRF, the tuple size, and the number of tuples in the stream. WriteMemory first sets the input stream source in the DU_CONF register to DMA, writes the tuple size to WSP_SIZE bits and asserts bit CONF_WSP bit. Finally, WriteMemory function configures the DMA and initiates the transfer.

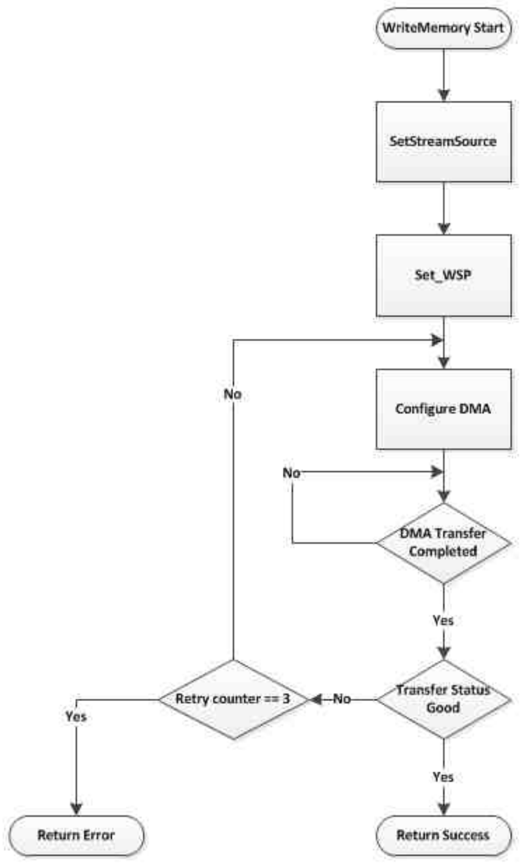


Fig. 4.7.: DU WriteMemory flowchart

ReadingMemory sets the output stream destination to DMA, and then configures the RSP by setting the RSP_SIZE and asserting CONF_RSP bit. Finally, it configures the DMA with source address in SRF, destination address in RAM, and transfer size in bytes. ReadMemory method is illustrated in Figure 4.8.

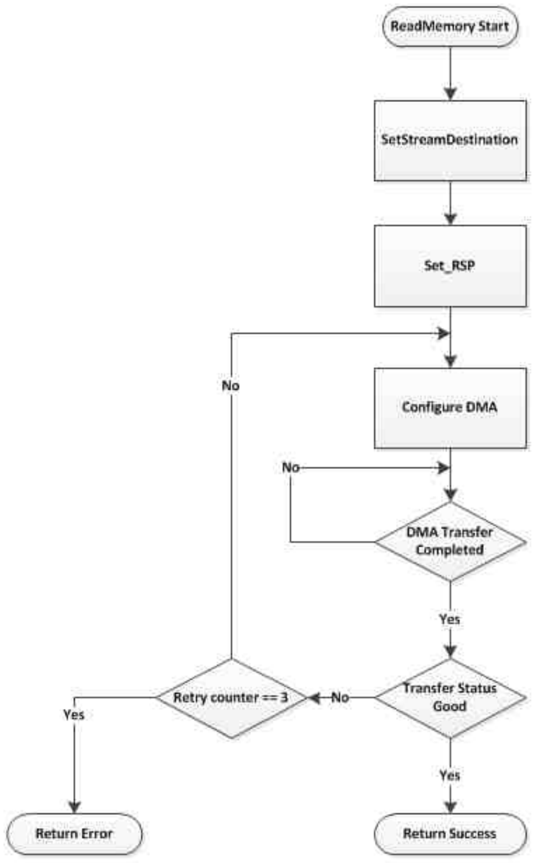


Fig. 4.8.: DU ReadMemory flowchart

DU driver allows the SM to choose the source of input streams and the destination of output streams by calling SetStreamSource and SetStreamDestination methods, respectively. SetStreamSource sets the source of input streams to the DMA by asserting IN_SRC bit in DU_CONF register. SetStreamDestination sets the destination of the output streams to the DMA by asserting OUT_SRC bit in DU_CONF register. Input stream's source, and output stream's destination are set to the host processor by default.

4.2 Stream Manager

Stream Manager is the main software component; it contains the state machine that controls the coprocessor. Furthermore, SM stores information related to streams and kernels, namely SRD, KDR, and ORF. These registers are logically stored in software arrays, but the physical memory is located in the KRM_IF.

Figure 4.9 illustrates SM's state machine. At start up, the SM waits in idle state until a new command is received from the host processor. The command is then decoded and SM moves to the corresponding state. The commands supported by SM are as follows:

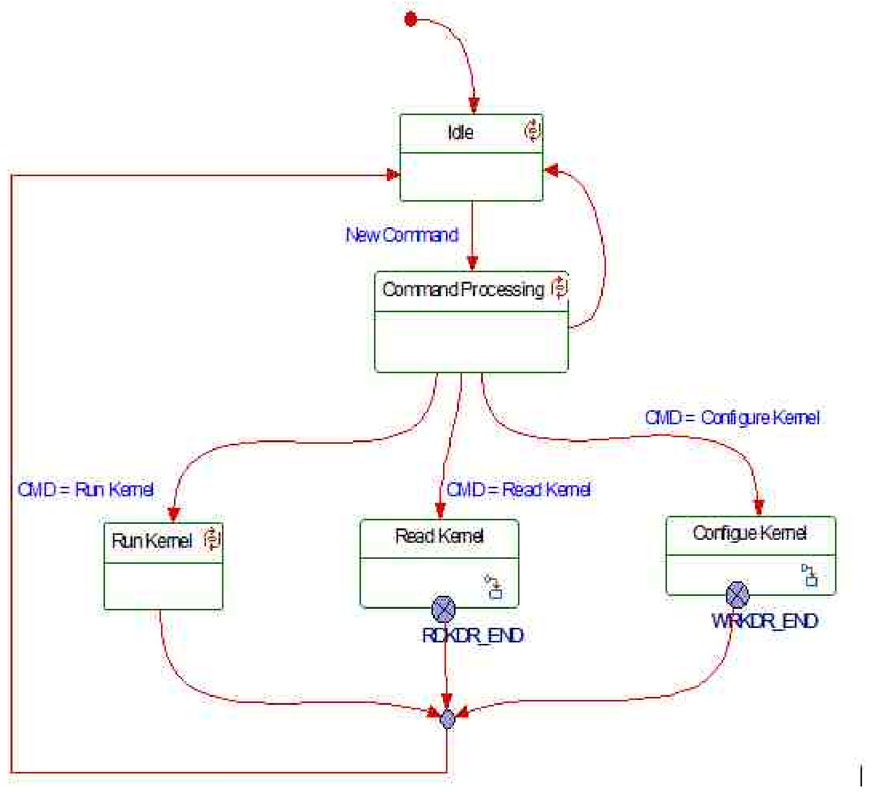


Fig. 4.9.: Stream Manager (SM) state machine

1. Write Stream Descriptor File (SDR): it contains information related to input and output streams such as, stream ID, stream start address in SRF, tuple count, and tuple size. SM stores SDRs in a 64×64-bit array.

2. Read Stream Descriptor File: SM decodes the stream ID, and then fetches the SDR value from SDR's array.

3. Configure Kernel: it contains Kernel Descriptor File (KDR), kernel constants, kernel instructions, and compiler optimizations. KDRs are stored in a 64×64-bit array.

4. Read Kernel: SM decodes the kernel ID, and then fetches the corresponding KDR from KDR's array, finally reads binaries from IMEM and sends them back to the host processor.

5. Run Kernel: SM triggers the execution of the kernel by the EU via KRM_IF.

In the processing command state, the 64-bit command from the host processor is decoded. Read and write SDR commands are processed immediately. The other commands are processed in their corresponding states. Configure kernel state contains the state machine illustrated in Figure 4.10.

Read kernel state consists of reading kernel binaries and constants from IMEM. The offset buffers are read from offset arrays, and finally the data is sent to the host processor via the Response FIFO. Read kernel command is used for debugging purposes only.

Run kernel state consists of triggering the KRM_IF to configure and run a specific kernel. When the kernel execution is finished, an interrupt is generated and the SM acknowledges and releases the KRM.

One of the SM's main responsibilities is to manage information related to streams and kernels, for example KDR and SDR. The SM stores such information in software arrays, for easy access and management. However, these arrays must be stored in KRM_IF physical memory to correctly configure the EU. Software application must ensure the correct placement of KDR, SDR, and ORF registers in KRM_IF memory. The following example illustrates a sample placement of these registers.

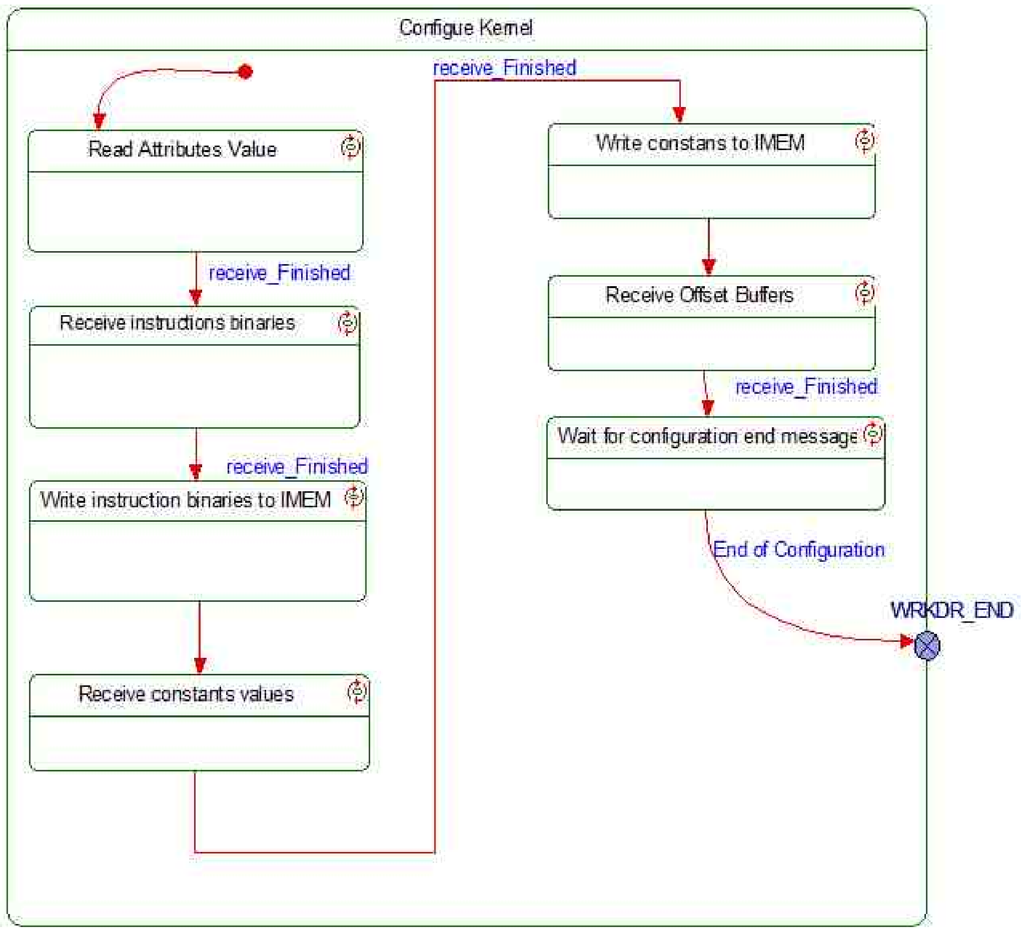


Fig. 4.10.: Configure Kernel state machine

Example 3 *SDR, KDR, and ORF placement in KRM_IF local memory*

Suppose the KRM_IF memory is allocated from 0xC0000000 to 0xC0000FFF in the MicroBlaze 4GB address space. The software application must ensure the SDR, KDR, and ORFs are stored in the KRM_IF address range by modifying the linker script file. The KRM_IF memory must be defined first; then a memory section is declared. Listing 4.1 illustrates defining memory and declaring sections in MicroBlaze linker script file.

Listing 4.1: Defining memory and sections in MicroBlaze linker script file

```
/* Define Memories in the system */

```

```
MEMORY
{
  mig_7series_0 : ORIGIN = 0x80000000, LENGTH = 0x40000000
 KRM_IF : ORIGIN = 0xC0000000, LENGTH = 0x00001000
}

/* Declaring Memory Section */

.Sec1 : {
   *(.Sec1)
  }> KRM_IF
```

After the KRM_IF memory is defined and new section is declared in the linker script file, the software application must explicitly place SDR, KDR, ORF arrays in the defined section. Listing 4.2 shows how to place arrays in KRM_IF memory.

Listing 4.2: Placing arrays in KRM_IF local memory

```
// Assigning arrays to KRM_IF local Memory
volatile u64 mau64_KDRMEM[64] __attribute__ ((section(".Sec1")));
volatile u64 mau64_AttrMEM[64] __attribute__ ((section(".Sec1")));
volatile u64 mau64_SDRMEM[64] __attribute__ ((section(".Sec1")));
volatile u64 mau64_IPF0_offMEM[64] __attribute__ ((section(".Sec1")));
volatile u64 mau64_IPF1_offMEM[64] __attribute__ ((section(".Sec1")));
volatile u64 mau64_OPF0_offMEM[64] __attribute__ ((section(".Sec1")));
volatile u64 mau64_OPF1_offMEM[64] __attribute__ ((section(".Sec1")));
volatile u64 mau64_OPF2_offMEM[64] __attribute__ ((section(".Sec")));
```

5. EXPERIMENTATION AND RESULTS

To compare the performance of the MicroBlaze-based coprocessor against the Symbiote Coprocessor Unit (SCU), the MicroBlaze-based hardware architecture is synthesized and implemented on Xilinx Virtex-7 XC7VX690T FPGA [17] using Xilinx VC709 [18] Evaluation Board. Table 5.1 shows the resource utilization percentage of the MicroBlaze-based coprocessor indicating that the XC7VX690T FPGA has enough resources for future work.

Table 5.1.: Synthesis results

Resource	**% Utilization**
Slice Registers	15.00
Slice LUT	32.42
Memory LUT	8.56
Block RAM	18.10
DSP48	2.83

In this chapter we first describe the testbench setup used to evaluate the proposed design, and then we compare the performance of MicroBlaze-based coprocessor against the SCU.

5.1 Testbench Setup

Figure 5.1 shows the block diagram of Xilinx VC709 evaluation board used for the experiments. The VC709 evaluation board contains a Virtex-7 XC7VX690T with

52,920 Kb Block RAM and 433,200 LUTs. The Virtex-7 is connected to the test computer through a PCI Express edge connector. Test results and debug information are sent to the test computer via the USB_to_UART bridge.

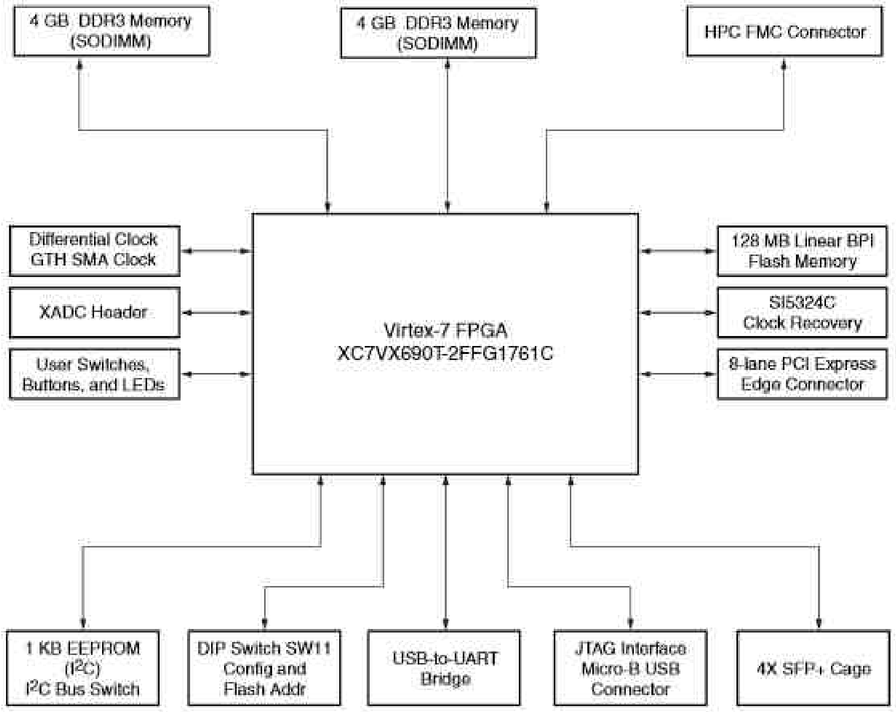

4 GB DDR3 Memory (SODIMM)
4 GB DDR3 Memory (SODIMM)
HPC FMC Connector
Differential Clock GTH SMA Clock
XADC Header
User Switches, Buttons, and LEDs
Virtex-7 FPGA XC7VX690T-2FFG1761C
128 MB Linear BPI Flash Memory
SI5324C Clock Recovery
8-lane PCI Express Edge Connector
1 KB EEPROM (I2C) I2C Bus Switch
DIP Switch SW11 Config and Flash Addr
USB-to-UART Bridge
JTAG Interface Micro-B USB Connector
4X SFP+ Cage

Fig. 5.1.: Xilinx VC709 evaluation board block diagram (Courtesy of XILINX [18]).

The test computer runs test software that configures the Stream Management Unit (SMU), writes input streams to Stream Register File (SRF), triggers kernel execution, and finally reads the output streams from SRF.

To measure kernel execution times accurately, an AXI4 hardware timer is added to the MicroBlaze-based coprocessor. The timer starts counting when the SMU receives Run Kernel command and stops when KRM interrupt is handled and the Execution Engine is free to process the next stream. The SMU transmits the measured execution times to the test computer using its USB port. Figure 5.2 shows the testbench setup.

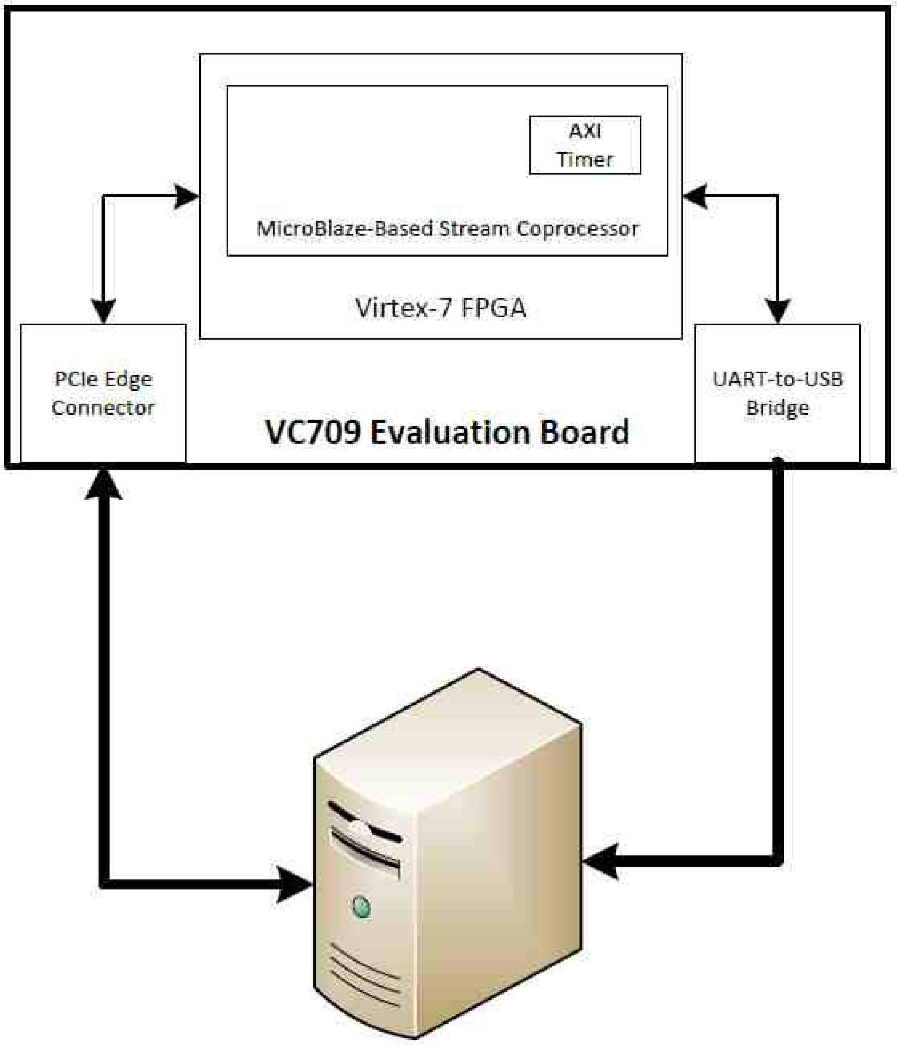


Fig. 5.2.: Testbench setup

5.2 Performance Comparison

5.2.1 Kernel Execution Performance

To compare the performance of the MicroBlaze-based coprocessor against the SCU, three queries, namely $map(n)$, $filter(n)$, and $aggregate(n)$ were executed; the execution time measured by the hardware timer is collected and compared to the reported [4] execution time of the SCU on the same queries. Table 5.2 and Table 5.3 show comparison of execution time (in milliseconds) for the three queries on two different window sizes.

Table 5.2.: Kernel execution time for various operators on 4096 tuples window size

Query	**SCU** (msec)	**MicroBlaze-based Coprocessor** (msec)
map(4096)	0.258	0.267
filter(4096)	0.531	0.540
aggregate(4096)	0.258	0.267

Table 5.3.: Kernel execution time for various operators on 16384 tuples window size

Query	**SCU** (msec)	**MicroBlaze-based Coprocessor** (msec)
map(16384)	1.013	1.022
filter(16384)	2.082	2.091
aggregate(16384)	0.841	0.850

The MicoBlaze-based coprocessor takes on average 9 microseconds longer than the SCU to execute the queries independent of the query, or the window sizes. The delay is due to the handling of KRM interrupt by the MicroBlaze.

5.2.2 PCIe Bandwidth

To measure the PCIe bandwidth, the test software writes 512KB of data to the SRF and reads them back. The time is measured, and the derived average bandwidth in this particular system setup is 796 MB/s.

6. SUMMARY

6.1 Thesis Conclusions

This study has proposed and implemented three modifications to the Symbiote Coprocessor Unit (SCU). First, the hardware implementation of the Stream Controller was replaced with a soft-core microcontroller, namely MicroBlaze. Second, HyperTransport host processor interface was replaced with high speed Gen 3 PCI Express interface. Finally, industry standard AXI4 interconnect replaced multiple proprietary interfaces used in the SCU.

The following provides a list of contributions brought by the aforementioned modifications.

1. The coprocessor is controlled by a software finite state machine instead of the previous complex hardware-based implementation. In general, software is easier to implement, debug, and test.

2. The highly configurable MicroBlaze provides the system with flexibility and intelligence. New software algorithms and control logic can be implemented in relatively short amount of time.

3. Industry standard AXI4 interconnect provides easier integration of new and existing AXI4 compatible peripherals.

4. High speed industry standard PCIe allows wider selection of host processors. PCIe is used in most modern computer systems.

5. The proposed architecture presents a software and hardware platform for development of future data stream processing applications.

The main target of this study is to provide a simple, flexible, future ready architecture while maintaining the SCU's high performance. Experimental results show the software-based control state machine adds a negligible 9 microseconds overhead to the overall execution time achieved by the SCU.

6.2 Future Work

6.2.1 Stand-alone Stream Processing Unit

In some applications, such as process control, it is desired to use a smaller, cheaper, stand-alone stream processing unit. Figure 6.1 shows an architecture of such a unit.

For configuration, the unit implements a web server where the user can design new queries, select which query to execute, and configure the IP addresses for input stream source and output stream destination. The queries are compiled by the MicroBlaze and stored in the SPU's Instruction Memory (IMEM).

Once the unit is configured, the Ethernet port copies input streams to its local RAM where the MicroBlaze divides these streams into windows and performs any preprocessing on the data. After that, the DMA copies the windows to the SRF. Finally, the MicroBlaze triggers kernel execution in the SPU.

Once the SPU executes the kernel, the MicroBlaze transfers the output stream from SRF to the desired destination on the network via its Gigabit Ethernet port.

The MicroBlaze-based coprocessor hardware architecture proposed in this study can be easily modified by removing the PCIe interface and adding an existing IP for the Gigabit Ethernet port.

New software application needs to be developed on top of the software architecture introduced in this study.

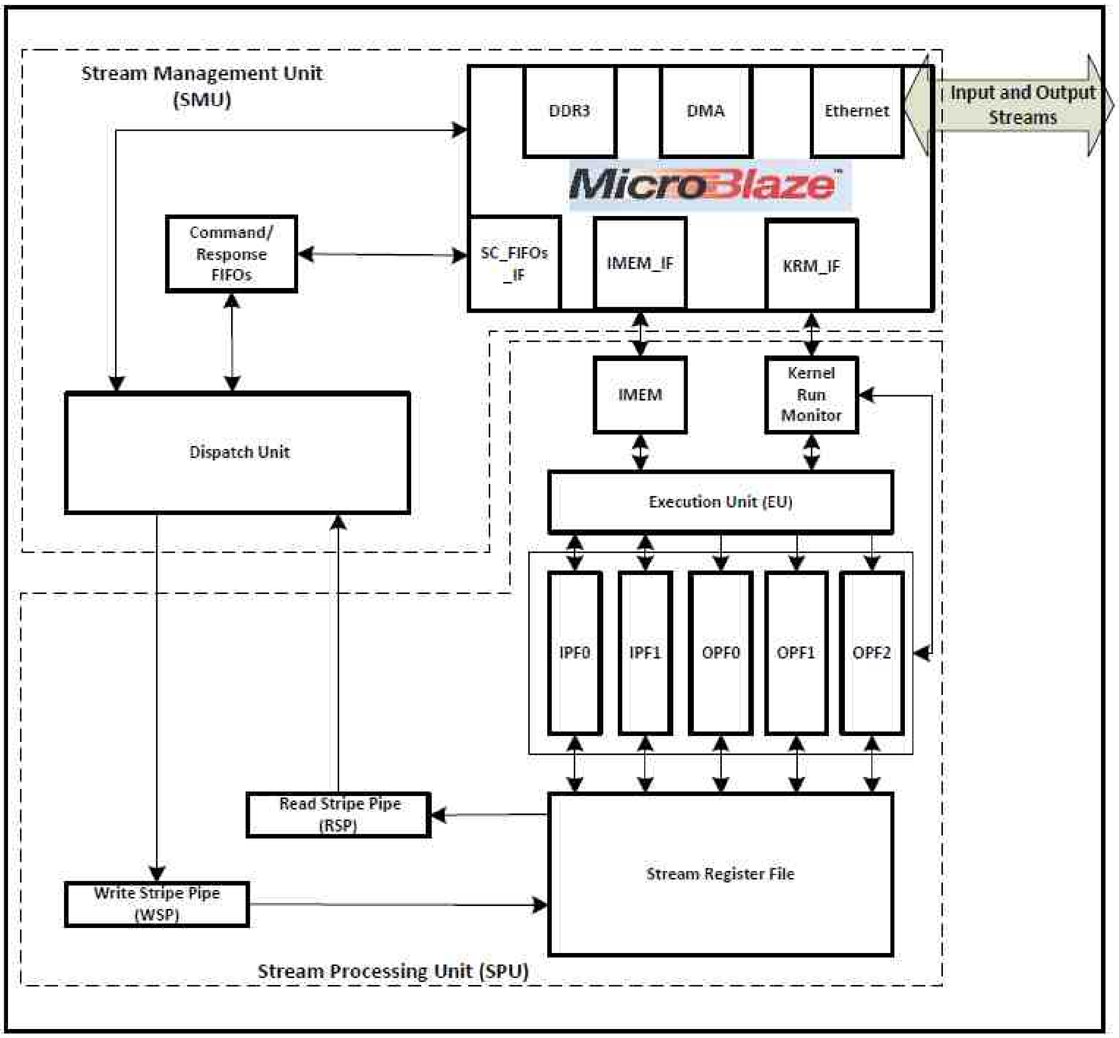


Fig. 6.1.: Stand-alone Stream Processing Unit

6.2.2 Distributed Stream Processing System

Current host processors have a limited number of PCIe slots, which limits the number of Stream Coprocessor units it can utilize. To overcome this limitation, the host processor can be connected to a network of Stand-alone Stream Processing Units through its Gigabit Ethernet port. Figure 6.2 illustrates a network-based Distributed Stream Processing System.

The host processor configures each unit, sends input streams for processing, triggers the kernel execution, and finally reads the output streams.

To implement such a system, only software modifications are needed to the Stand-alone Stream Processing Unit while the hardware architecture remains the same as described in 6.2.1.

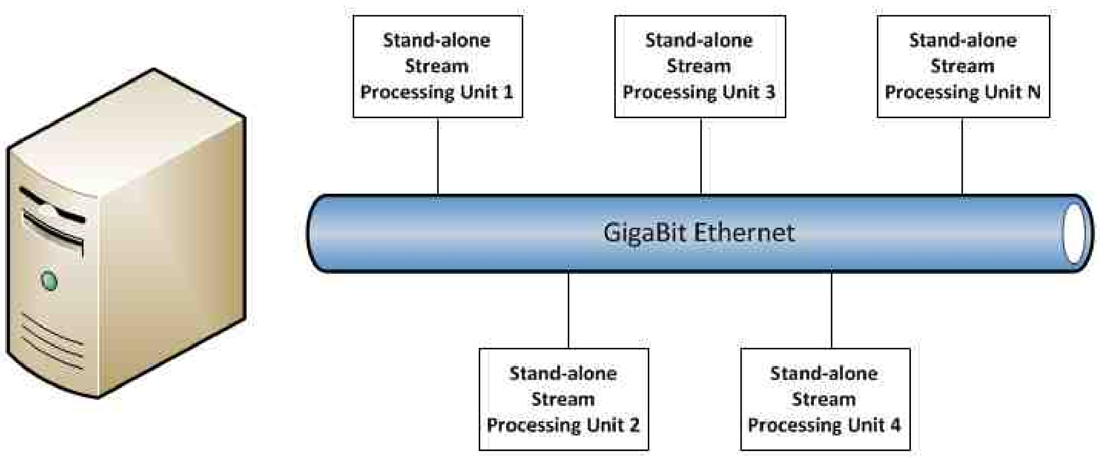

Stand-alone Stream Processing Unit 1
Stand-alone Stream Processing Unit 3
Stand-alone Stream Processing Unit N
GigaBit Ethernet
Stand-alone Stream Processing Unit 2
Stand-alone Stream Processing Unit 4

Fig. 6.2.: Distributed stream processing system

6.2.3 Multiple Stream Processing Units

To increase the throughput of the MicroBlaze-based coprocessor, a second Stream Processing Unit (SPU) can be added to the design. Figure 6.3 shows such an architecture with two SPUs.

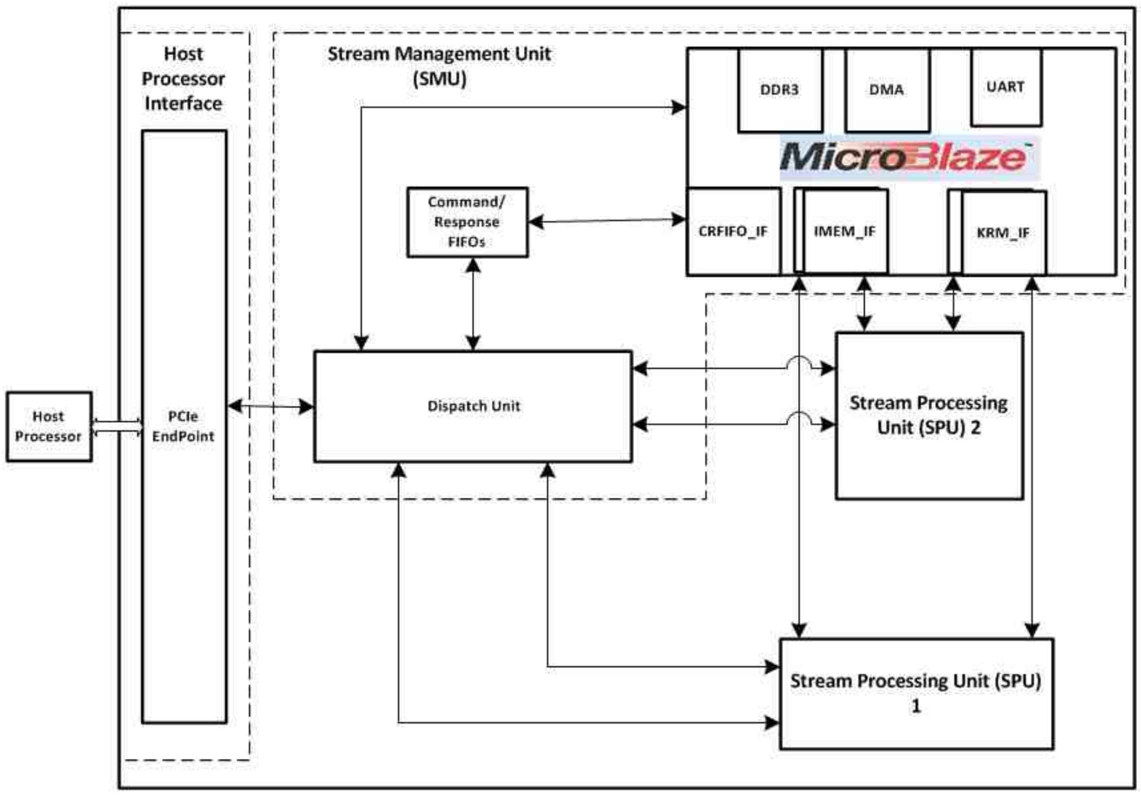


Fig. 6.3.: MicroBlaze-based coprocessor with two SPUs

REFERENCES

REFERENCES

[1] D. Abadi et al., "Aurora: A Data Stream Management System," in *Proc. of ACM SIGMOD*, p. 666, 2003.

[2] C. Cranor, T. Johnson, O. Spatascheck, "Gigascope: A stream database for network applications," in *Proc. of ACM SIGMOD*, pp. 647–651, 2003.

[3] M. A. Hammad et al., "Nile: a query processing engine for data streams," in *Proc. of ICDE*, p. 851, IEEE Computer Society, 2004.

[4] P. S. Vaidya, *Hardware-software co-designed data stream management systems.* PhD thesis, Purdue University, West Lafayette, IN, USA, December 2015.

[5] H. Holden, J. Trodden, D. Anderson, "HyperTransport 3.1 Interconnect Technology." https://www.mindshare.com/files/ebooks/HyperTransport%203.1%20Interconnect%20Technology.pdf, September 2008. Accessed: December 2017.

[6] Xilinx Inc, "MicroBlaze Processor Reference Guide." https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug984-vivado-microblaze-ref.pdf, April 2016. Accessed: November 2017.

[7] Arm Limited, "AMBA AXI and ACE Protocol Specification." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022d/index.html, October 2011. Accessed: October 2017.

[8] M. Jackson and R. Budruk, *PCI Express Technology.* MindShare, Inc., 2012.

[9] Xilinx Inc, "AXI Reference Guide." https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf, March 2011. Accessed: April 2017.

[10] Xilinx Inc, "AXI Reference Guide." https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf, March 2011. Accessed: April 2017.

[11] Xilinx Inc, "AXI Central Direct Memory Access v4.1." https://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf, April 2017. Accessed: November 2017.

[12] Xilinx Inc, "AXI Interrupt Controller (INTC) v4.1." https://www.xilinx.com/support/documentation/ip_documentation/axi_intc/v4_1/pg099-axi-intc.pdf, October 2017. Accessed: November 2017.

[13] Xilinx Inc, "AXI UART Lite v2.0." https://www.xilinx.com/support/documentation/ip_documentation/axi_uartlite/v2_0/pg142-axi-uartlite.pdf, April 2017. Accessed: December 2017.

[14] Xilinx Inc, “Memory Interface Solutions.” https://www.xilinx.com/support/documentation/ip_documentation/ug086.pdf, September 2010. Accessed: August 2017.

[15] Xilinx Inc, “Vivado Design Suite User Guide.” https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug910-vivado-getting-started.pdf, October 2017. Accessed: December 2017.

[16] Xilinx Inc, “Virtex-7 FPGA Gen3 Integrated Block for PCI Express v4.3.” https://www.xilinx.com/support/documentation/ip_documentation/pcie3_7x/v4_3/pg023_v7_pcie_gen3.pdf, October 2017. Accessed: December 2017.

[17] Xilinx Inc, “7 Series FPGAs Data Sheet:Overview.” https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf, August 2017. Accessed: December 2017.

[18] Xilinx Inc, “VC709 Evaluation Board for the Virtex-7 FPGA.” https://www.xilinx.com/support/documentation/boards_and_kits/vc709/ug887-vc709-eval-board-v7-fpga.pdf, August 2016. Accessed: June 2017.