UNIVERSITY OF THESSALY

Hardware Profiling in a FPGA-based SoC

Author:
Ioannis PARNASSOS

Supervisors:
Dr. Nikolaos BELLAS
Dr. Christos ANTONOPOULOS

A thesis submitted in fulfilment of the requirements for the degree of Diploma of Science in Computer and Communication Engineering

in the

Department of Electrical and Computer Engineering
University of Thessaly

October 14, 2015

UNIVERSITY OF THESSALY

Department of Electrical and Computer Engineering

Hardware Profiling in a FPGA-based SoC

Ανάλυση απόδοσης υλικού για FPGA-based SoC

by

Ioannis Parnassos

Graduate Thesis

for

the degree of

Diploma of Science in Computer and Communication Engineering

Declaration of Authorship

I, Ioannis Parnassos, confirm that this thesis is my own work. All direct or indirect sources used are acknowledged as references. This thesis was not previously presented to another examination board and has not been published.

Dedicated to my family and friends. . .

Abstract

Developing a complete FPGA-based system architecture requires a vast variety of design approaches to be examined and evaluated. Several attempts ostensibly sufficient will not produce the expected outcome in terms of overall system performance. Locating the system's bottleneck cannot be relied entirely on simulation.

The purpose of this Thesis is to fulfill the need of profiling analysis for FPGA-based SoC presenting the development of a hardware design with capabilities similar to software event-based profilers.

RIFFA framework offers a user friendly implementation for communicating data from a host CPU to a FPGA via PCI Express bus and was used as infrastructure. The created design extends RIFFA with profiling mechanisms for monitoring and logging of user created IP cores based on a predefined event set. RIFFA Monitor was tested with already implemented designs and collected data were used for time analysis and event visualization.

During development several tasks were incorporated into RIFFA Monitor and even more are left as future extensions, with the ambition to create a practical and convenient tool for hardware design engineers.

Περίληψη

Κατά την ανάπτυξη ενός συστήματος βασισμένου σε FPGA θα εξεταστούν και θα αξιολογηθούν αρκετές διαφορετικές προσεγγίσεις σχεδιασμού. Πολλές από αυτές μπορεί να δίνουν την εντύπωση ενός ορθά υλοποιημένου συστήματος αλλά δεν θα παράγουν τα αναμενόμενα αποτελέσματα όσον αφορά τη συνολική απόδοση. Ο εντοπισμός συμφόρησης του συστήματος δεν μπορεί να βασιστεί αποκλειστικά στην προσομοίωση.

Ο σκοπός αυτής της διατριβής είναι να καλύψει την ανάγκη για ανάλυση απόδοσης σε FPGA-based SoC παρουσιάζοντας τον σχεδιασμό και την ανάπτυξη υλικού με δυνατότητες αντίστοιχες λογισμικών ανάλυσης απόδοσης βασισμένων σε γεγονότα.

Το RIFFA προσφέρει ένα φιλικό προς το χρηστή πλαίσιο για την επικοινωνία δεδομένων ανάμεσα στο λογισμικό και σε μια FPGA μέσω του διαύλου PCI Express, και χρησιμοποιήθηκε ως υποδομή. Το προαναφερθέν πλαίσιο εμπλουτίστηκε με μηχανισμούς για την παρακολούθηση και καταγραφή στιγμιότυπων λειτουργίας των συστημάτων συμφώνα με ένα σύνολο προκαθορισμένων γεγονότων. Το παραγόμενο υλικό ονόματι RIFFA Monitor χρησιμοποιήθηκε για την παρακολούθηση και αξιολόγηση υλοποιημένων συστημάτων και τα δεδομένα που συλλέχτηκαν χρησιμοποιήθηκαν για την ανάλυση λειτουργιάς και οπτικοποιήθηκαν.

Όσο ο RIFFA Monitor βρισκόταν υπό ανάπτυξη υιοθέτησε μια πληθώρα λειτουργιών, ενώ αρκετές ακόμα δοκιμάζονται για μελλοντικές επεκτάσεις, με τη φιλοδοξία να δημιουργήσουν ένα πρακτικό και βολικό εργαλείο για τους μηχανικούς σχεδίασης υλικού.

Acknowledgements

For the fullfilment of this Thesis, I would like to thank my my professor Dr. Nikolaos Bellas for his advice and guidance and my colleague George Zindros for his support, collaboration and ideas.

Also i would like to thank my family for their support and patience...

Contents

List of Figures

List of Tables

Abbreviations

API	**A**pplication **P**rogramming **I**nteface
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BRAM	**B**lock **R**andom **A**ccess **M**emory
CAD	**C**omputer **A**ided **D**esign
CMT	**C**lock **M**anagment **T**ile
CLB	**C**onfigurable **L**ogic **B**lock
DMA	**D**irect **M**emory **A**ccess
DFF	**D** **F**lip **F**lop
DSP	**D**igital **S**ignal **P**rocessing
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
LUT	**L**ook **U**p **T**able
MMCM	**M**ixed **M**ode **C**lock **M**anagment
MUX	**MU**ultiple**X**er
PCI	**P**eripheral **C**omponent **I**nterconnect
PLL	**P**hase **L**ocked **L**oop
RIFFA	**R**eusable **I**ntegration **F**ramework for **F**PGA **A**ccelerators
RTL	**R**egister **T**ransfer **L**evel

Chapter 1

Introduction

1.1 Describing the Motives

In software engineering, profiling is a form of dynamic program analysis. Information provided can point out which pieces of a program are slower than expected, and might be candidates for rewriting. It can also tell which functions are being called more or less often and can help spotting bugs that had otherwise been unnoticed.

On the other hand when hardware engineers design an FPGA-based SoC they have to rely on simulation for the evaluation and optimization of their work due to the lack of hardware profiling tools. Collecting useful information during SoC run time is either partially supported if a soft processor is implemented, or require manual addition of profiling mechanisms.

The purpose of this Thesis is the development of a hardware design offering similar capabilities to software profiling tools. Hardware Profiler will assist in monitoring, debugging and evaluating FPGA designs, perform time analysis and locate system bottlenecks.

1.2 Thesis Structure

Thesis is divided in three main Chapters, each one of those includes smaller sections and possibly subsections.

Chapter 2 provides background information over the hardware and software used in this project. It begins in section 2.1 with a brief overview over FPGA architecture and operation, and then focuses on the technical characteristics of Virtex 7 VC707 evalution board on which the design was developed. Following in section 2.2 the RIFFA framework is presented and described and in section 2.3 we have a short reference on the development suite.

Chapter 3 analytically describe the RIFFA 2.0 Profiler Core. In the first sections we come across the purpose of the project, a high level view of the architecture, an an introduction to event-based profilers, followed on section 3.4 by a complete analysis of each module. Afterwards the software bindings are provided and expanded. Finally the last section will focus on the development milestones.

Chapter 4 summarizes the work done, results generated and provides ideas for future development.

Chapter 2

Background

2.1 Field Programmable Gate Array - FPGA

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence ”field-programmable”. The FPGA configuration is generally specified using a hardware description language (HDL). As opposed to Application Specific Integrated Circuits (ASICs), where the device is custom built for the particular design, FPGAs can be programmed to the desired application or functionality requirements.

FPGAs contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects that allow the blocks to be ”wired together”, like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

An FPGA can be used to solve any problem which is computable. This is trivially proven by the fact FPGA can be used to implement a soft microprocessor. Their advantage lies in that they are sometimes significantly faster for some applications due to their parallel nature and optimality in terms of the number of gates used for a certain process.

2.1.1 FPGA Architecture

Logic blocks

The most common FPGA architecture among academic and commercial FPGAs consists of an island-style array of logic blocks (called Configurable Logic Block, CLB, or Logic Array Block, LAB, depending on vendor), I/O pads, and routing channel.

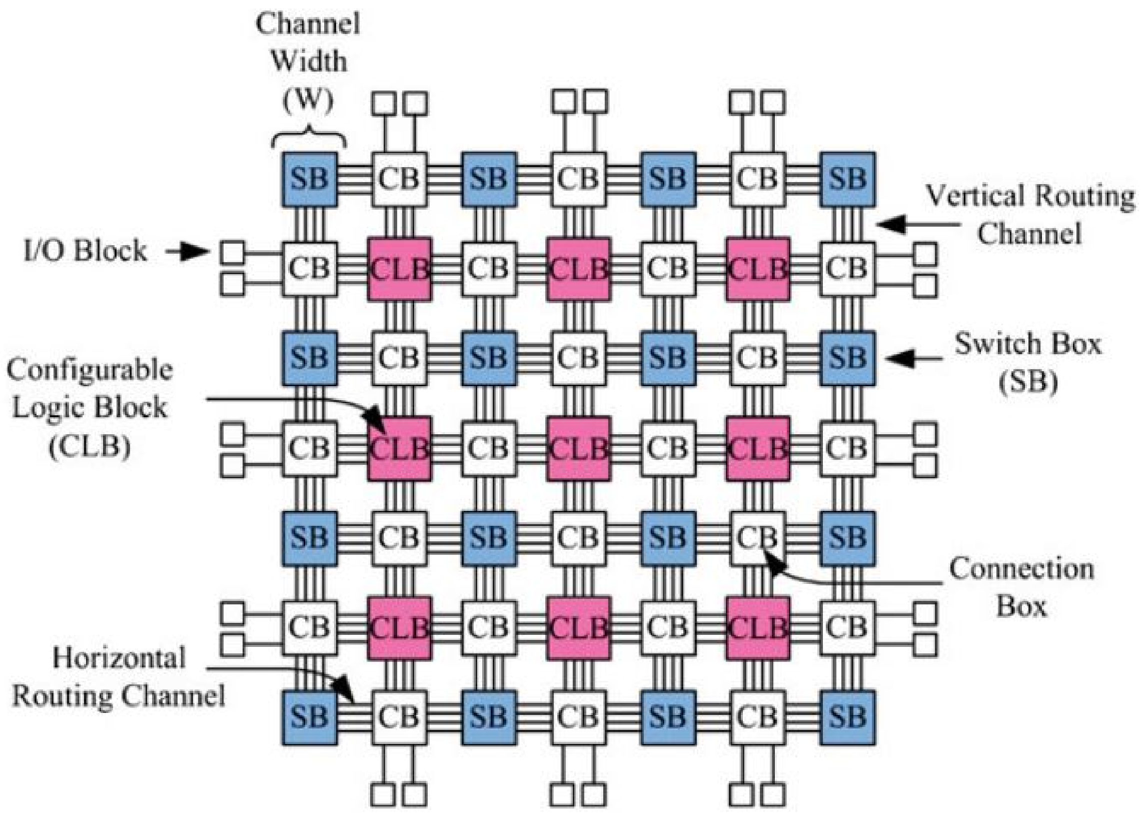


FIGURE 2.1: Overview of Island-Style FPGA architecture

CLB is a the fundamental building block a FPGA and can be configured by the engineer to provide reconfigurable logic gates. A logic block consists of a few logical cells (called ALM, LE, Slice etc.). A typical cell consists of a 4-input LUT, a Full adder (FA) and a D-type flip-flop, as shown in figure 2.1. The LUTs are in this figure split into two 3-input LUTs. In normal mode those are combined into a 4-input LUT through the left mux. In arithmeticmode, their outputs are fed to the FA. The selection of mode is programmed into the middle multiplexer. The output can be either synchronous or asynchronous, depending on the programming of the mux to the right, in the figure example. In recent years, manufacturers have started moving to 6-input LUTs in their high performance parts, claiming increased performance.

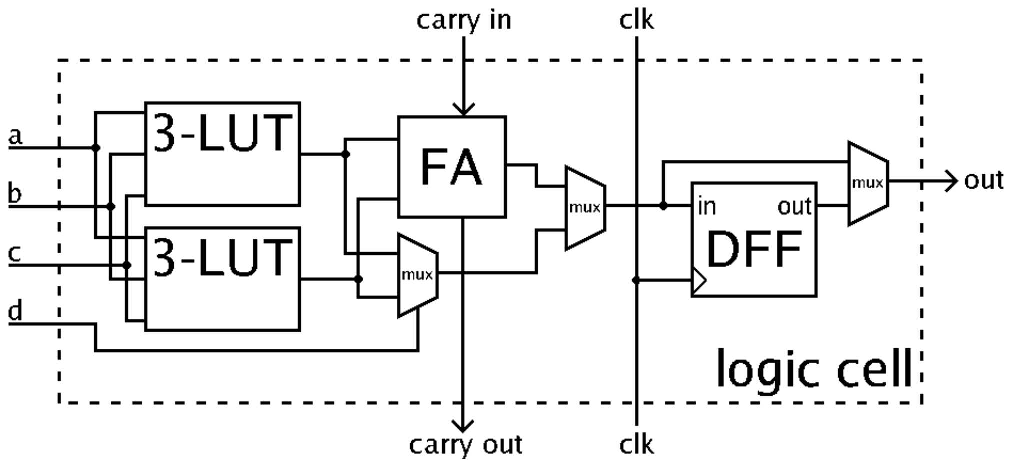


Figure 2.2: Simplified example illustration of a logic cell

Hard blocks

Modern FPGA families expand upon the above capabilities to include higher level functionality fixed into the silicon. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to building them from primitives. Examples of these include multipliers, generic DSP blocks, embedded processors, high speed I/O logic and embedded memories. Higher-end FPGAs can contain high speed multi-gigabit transceivers and hard IP cores such as processor cores, Ethernet MACs, PCI/PCI Express controllers, and external memory controllers. These cores exist alongside the programmable fabric, but they are built out of transistors instead of LUTs so they have ASIC level performance and power consumption while not consuming a significant amount of fabric resources, leaving more of the fabric free for the application-specific logic. The multi-gigabit transceivers also contain high performance analog input and output circuitry along with high-speed serializers and deserializers, components which cannot be built out of LUTs. Higher-level PHY layer functionality such as line coding may or may not be implemented alongside the serializers and deserializers in hard logic, depending on the FPGA.

Routing

An application circuit must be mapped into an FPGA with adequate resources. While the number of CLBs/LABs and I/Os required is easily determined from the design, the number of routing tracks needed may vary considerably even among designs with the same amount of logic. Generally, the FPGA routing is unsegmented. That is, each wiring segment spans only one logic block before it terminates in a switch box. By turning on some of the programmable switches within a switch box, longer paths can be constructed. For higher speed interconnect, some FPGA architectures use longer routing lines that span multiple logic blocks.

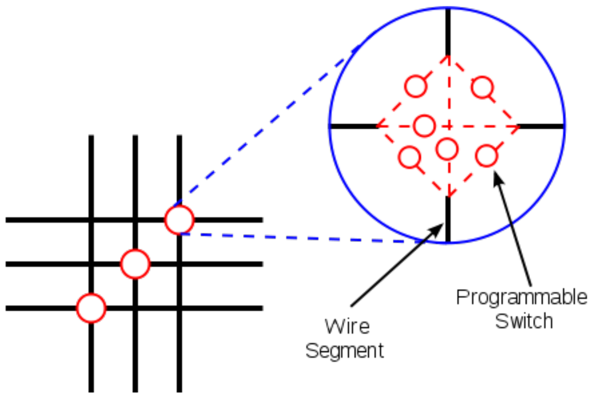


FIGURE 2.3: Switch box Topology

Software Flow

FPGA architectures have been intensely investigated over the past two decades. A major aspect of FPGA architecture research is the development of Computer Aided Design (CAD) tools for mapping applications to FPGAs. It is well established that the quality of an FPGA-based implementation is largely determined by the effectiveness of accompanying suite of CAD tools. Benefits of an otherwise well designed, feature rich FPGA architecture might be impaired if the CAD tools cannot take advantage of the features that the FPGA provides. Thus, CAD algorithm research is essential to the necessary architectural advancement to narrow the performance gaps between FPGAs and other computational devices like ASICs.

The software flow (CAD flow) takes an application design description in a Hardware Description Language (HDL) and converts it to a stream of bits that is eventually programmed on the FPGA. The process of converting a circuit description into a format that can be loaded into an FPGA can be roughly divided into five distinct steps, namely: synthesis, technology mapping, mapping, placement and routing. The final output of FPGA CAD tools is a bitstream that configures the state of the memory bits in an FPGA. The state of these bits determines the logical function that the FPGA implements.

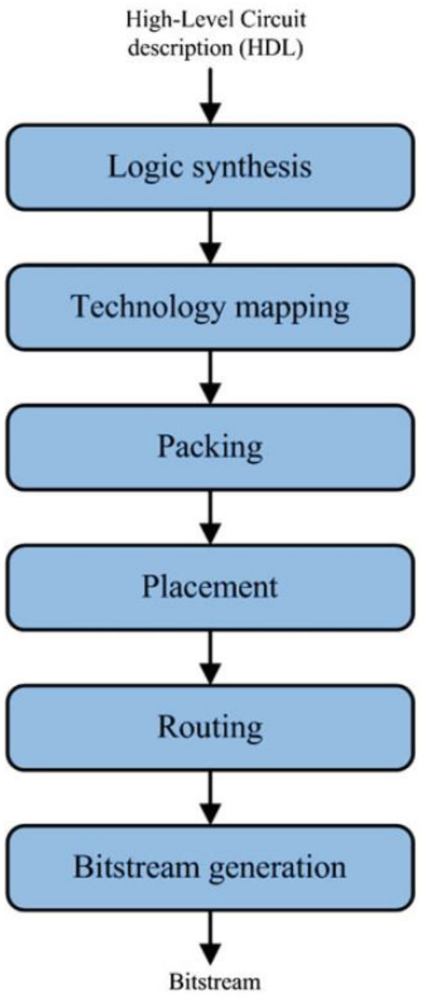


FIGURE 2.4: FPGA Software Flow

2.1.2 Virtex 7^{TM}VC707 Evaluation board

The project was developed on a Virtex-7 VC707 Evaluation board using the XC7VX485T-2FFG1761C FPGA. Virtex is the flagship family of FPGA products developed by Xilinx optimized for highest system performance and capacity. The VC707 board block diagram is shown in Figure 2.6.

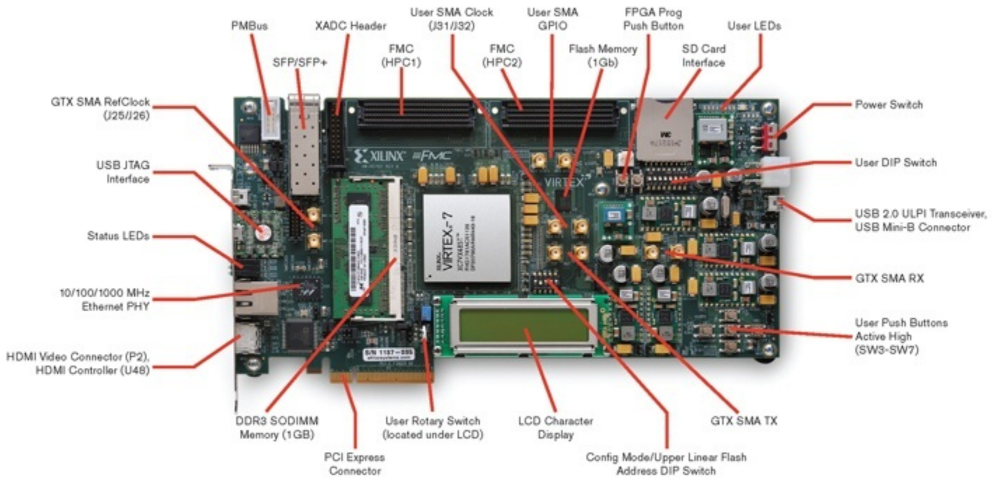


FIGURE 2.5: VC707 Evaluation board

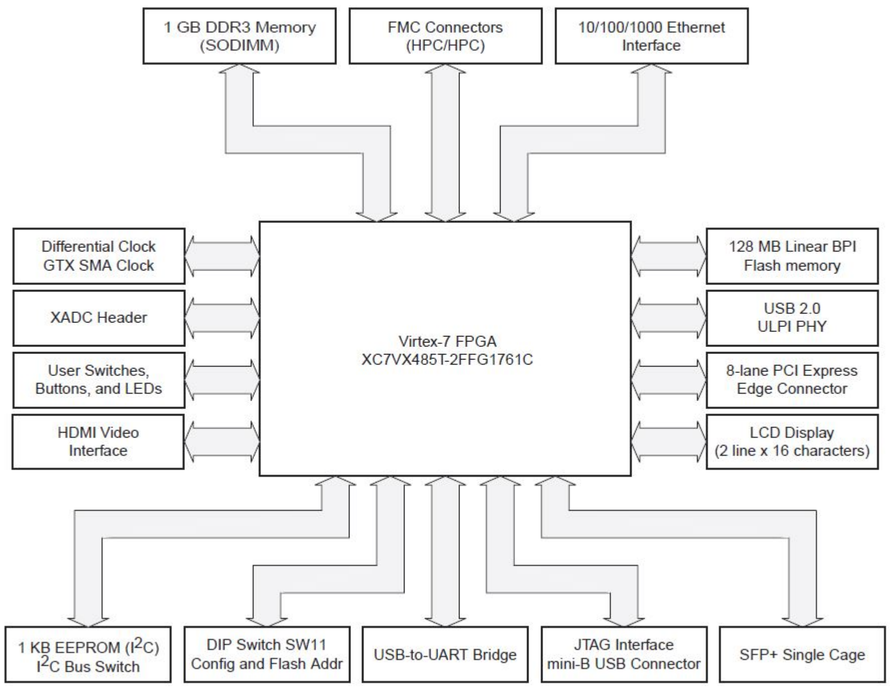
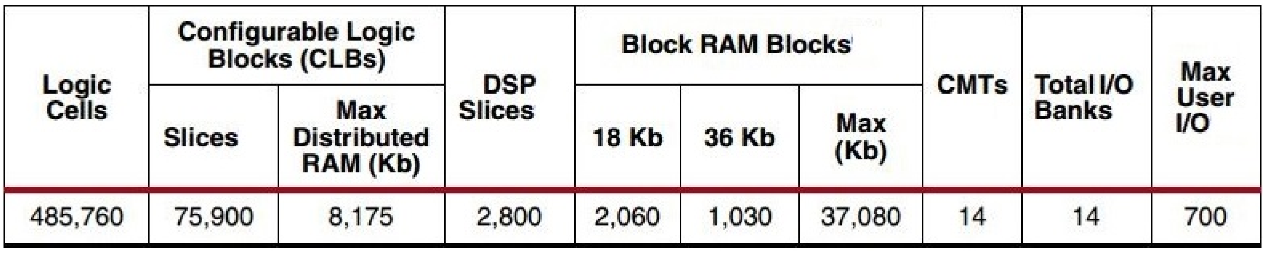

1 GB DDR3 Memory (SODIMM)
FMC Connectors (HPC/HPC)
10/100/1000 Ethernet Interface
Differential Clock GTX SMA Clock
XADC Header
User Switches, Buttons, and LEDs
HDMI Video Interface
Virtex-7 FPGA XC7VX485T-2FFG1761C
128 MB Linear BPI Flash memory
USB 2.0 ULPI PHY
8-lane PCI Express Edge Connector
LCD Display (2 line x 16 characters)
1 KB EEPROM (I2C) I2C Bus Switch
DIP Switch SW11 Config and Flash Addr
USB-to-UART Bridge
JTAG Interface mini-B USB Connector
SFP+ Single Cage

FIGURE 2.6: VC707 board block diagram



Logic Cells	Configurable Logic Blocks (CLBs)		DSP Slices	Block RAM Blocks			CMTs	Total I/O Banks	Max User I/O
	Slices	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)			
485,760	75,900	8,175	2,800	2,060	1,030	37,080	14	14	700

Figure 2.7: XC7VX485T FPGA Feature Summary

7 series FPGA slice contains four LUTs and eight flip-flops; only some slices can use their LUTs as distributed RAM or SRLs. Each DSP slice contains a pre-adder, a 25 x 18 multiplier, an adder, and an accumulator. Block RAMs are fundamentally 36 Kb in size; each block can also be used as two independent 18 Kb blocks. Each CMT contains one MMCM and one PLL.

2.2 Reusable Integration Framework for FPGA Accelerators - RIFFA

RIFFA (Reusable Integration Framework for FPGA Accelerators) is a simple framework for communicating data from a host CPU to a FPGA via a PCI Express bus. The framework requires a PCIe enabled workstation and a FPGA on a board with a PCIe connector. RIFFA supports Windows and Linux, Altera and Xilinx, with bindings for C/C++, Python, MATLAB and Java.

On the software side there are two main functions: data send and data receive. These functions are exposed via user libraries in C/C++, Python, MATLAB, and Java. The driver supports multiple FPGAs (up to 5) per system. The software bindings work on Linux and Windows operating systems. Users can communicate with FPGA IP cores by writing only a few lines of code.

On the hardware side, users access an interface with independent transmit and receive signals. The signals provide transaction handshaking and a first word fall through FIFO interface for reading/writing data. No knowledge of bus addresses, buffer sizes, or PCIe packet formats is required. Simply send data on a FIFO interface and receive data on a FIFO interface. RIFFA does not rely on a PCIe Bridge and therefore is not subject to the limitations of a bridge implementation. Instead, RIFFA works directly with the PCIe Endpoint and can run fast enough to saturate the PCIe link. It communicates data using direct memory access (DMA) transfers and interrupt signaling, achieving high bandwidth over the PCIe link.

For the development of this Thesis RIFFA version 2.0.2 was used as infrastructure. The provided analysis is a replica of the information illustrated in the official RIFFA 2 site.

2.2.1 RIFFA Architecture

Interface has been simplified to expose data as a first word fall through FIFO (valid-data-ready interface). The data is transferred by RIFFA's RX and TX DMA engines using scatter gather address information from the workstation. These engines issue and service PCIe packets to and from the PCIe Endpoint. RIFFA relies on a Vendor PCIe Endpoint core to drive the transceivers. These are lowest-level interface that FPGA vendors provide. The RIFFA interface supports 32-bit, 64-bit and 128-bit widths, depending on the PCIe link configuration. A high level architectural diagram of the RIFFA framework is illustrated in figure 2.8.

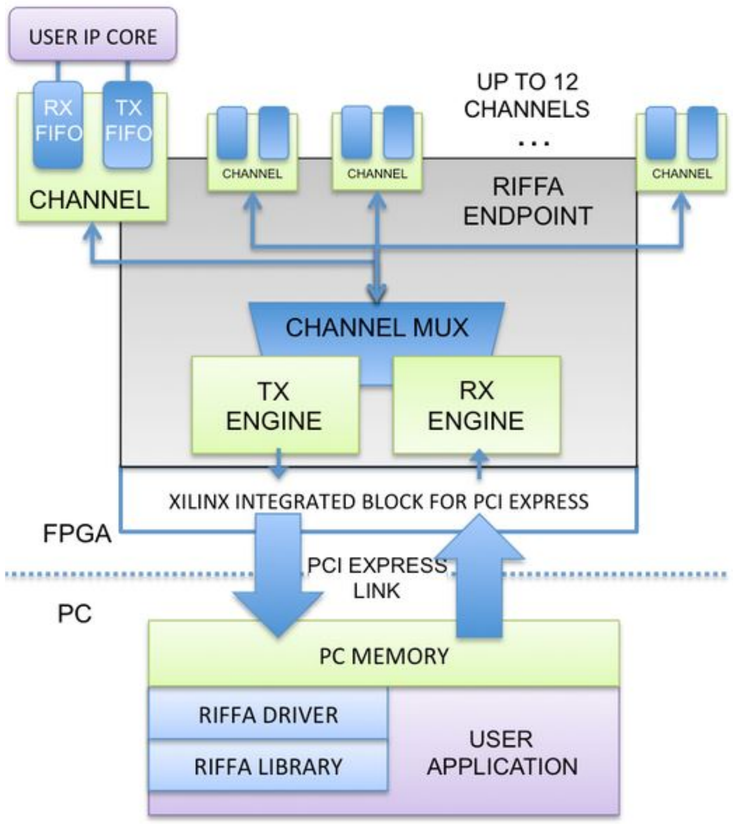


Figure 2.8: RIFFA high level architectural diagram

The upstream transfer is initiated by the FPGA. However, they will not begin until the user application calls the user library function fpga_recv. Upon doing so, the thread enters the kernel driver and begins the pending upstream request. If the upstream request has not yet been received, the thread waits for it to arrive (bounded by the timeout parameter). On the diagram, the user library and device driver are represented by the single node labeled ”RIFFA Library”.

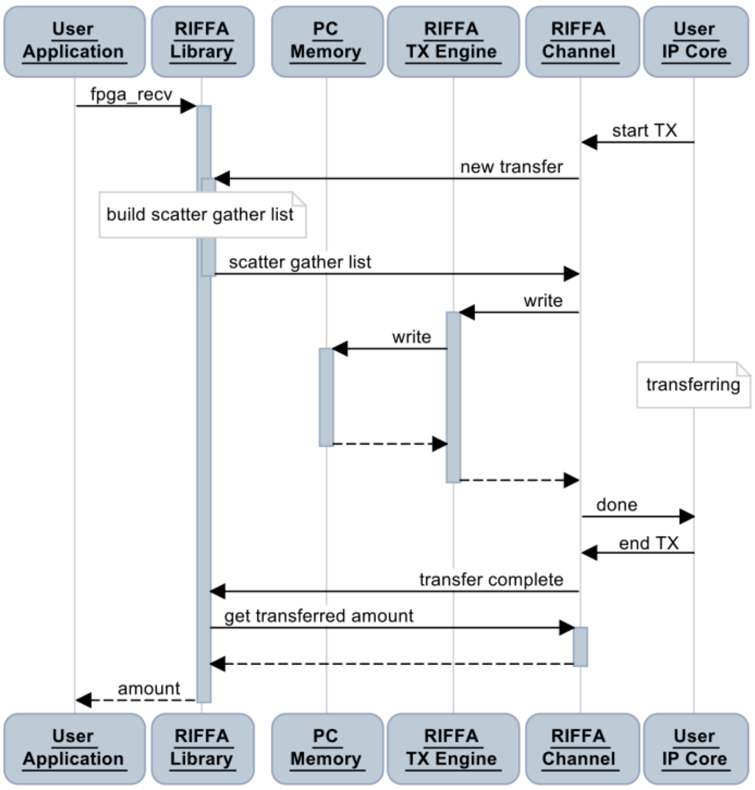


Figure 2.9: Sequence diagram for upstream transfer

Servicing the request involves building a list of scatter gather elements which identify which pages of physical memory correspond to the receptacle byte array. The scatter gather elements are written to a shared buffer. This buffer location and content length are provided to the FPGA. Each page enumerated by the scatter gather list is pinned to memory to avoid costly paging. The FPGA reads the scatter gather data then issues write requests to memory for the upstream data. If more scatter gather elements are needed, the FPGA will request additional elements via interrupt. Otherwise, the kernel driver waits until all the data is written. The FPGA provides this notification, again via an interrupt.

After the upstream transaction is complete, the driver reads the FPGA for a final count of data words written. This is necessary as the scatter gather elements only provide an upper bound on the amount of data that is to be written. This completes the transfer and the function call returns to the application with the final count.

A similar sequence exists for downstream transfers. In this direction, the application initiates the transfer by calling the library function fpga_send The thread enters the kernel driver and writes to the FPGA to initiate the transfer. Again, a scatter gather list is compiled, pages are pinned, and the FPGA reads the scatter gather elements. Each of the elements results in one or more read requests by the FPGA. The read requests are serviced and the kernel driver is notified only when more scatter gather elements are needed or when the transfer has completed.

Upon completion, the driver reads the final count read by the FPGA. In error free operation, this value should always be the length of all the scatter gather elements. The final count is returned to the application.

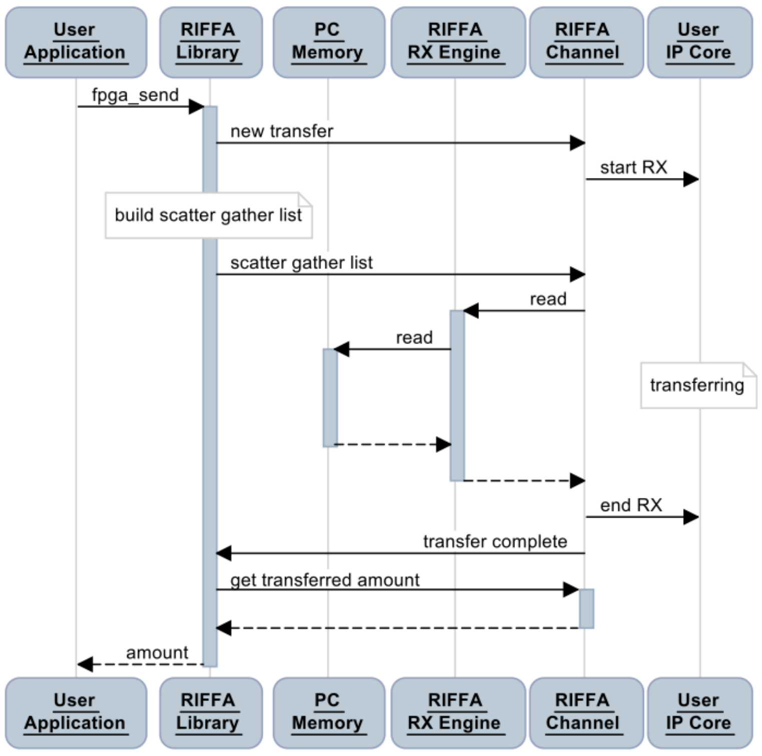
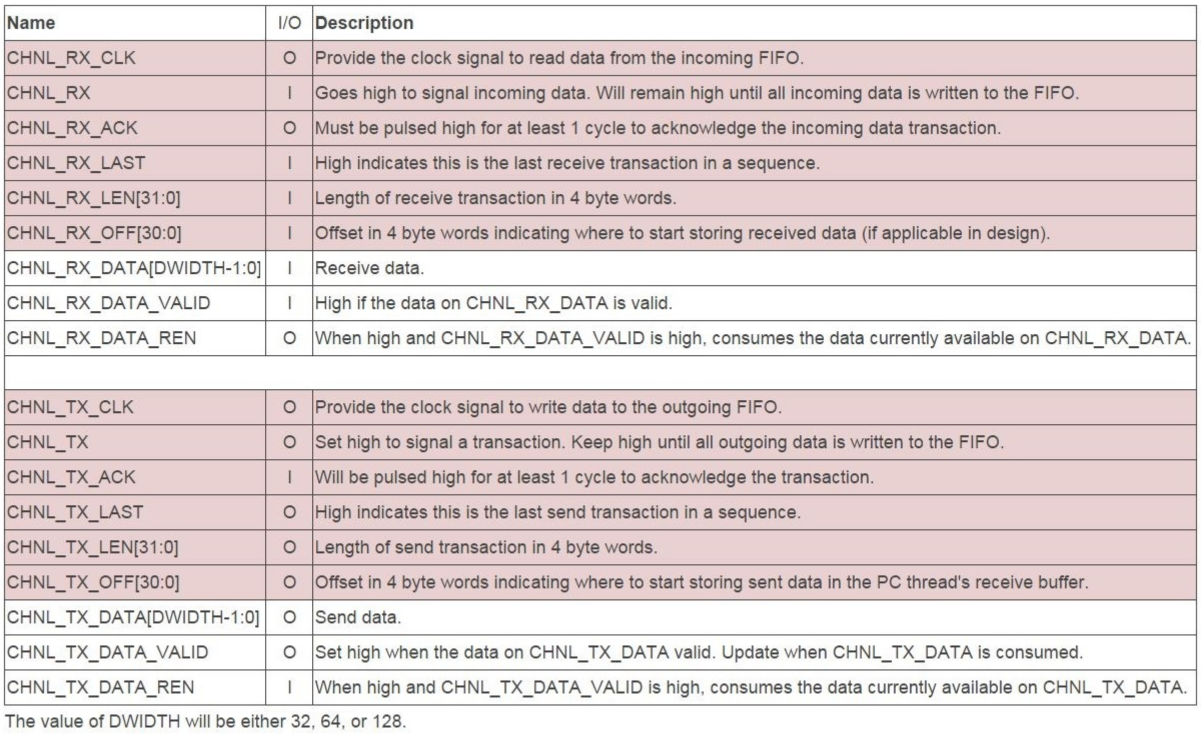


FIGURE 2.10: Sequence diagram for downstream transfer

2.2.2 RIFFA Hardware Interface

A single RIFFA channel has two sets of signals, one for receiving data (RX) and one for sending data (TX). RIFFA has simplified the interface to use a minimal handshake and receive/send data using a FIFO with first word fall through semantics (valid+read interface). The clocks used for receiving and sending can be asynchronous from each other and from the PCIe interface (RIFFA clock). The table below describes the ports. The input/output designations are from your user core's perspective (i.e. the core(s) you write and connect to the RIFFA channel).



Name	I/O	Description
CHNL_RX_CLK	O	Provide the clock signal to read data from the incoming FIFO.
CHNL_RX	I	Goes high to signal incoming data. Will remain high until all incoming data is written to the FIFO.
CHNL_RX_ACK	O	Must be pulsed high for at least 1 cycle to acknowledge the incoming data transaction.
CHNL_RX_LAST	I	High indicates this is the last receive transaction in a sequence.
CHNL_RX_LEN[31:0]	I	Length of receive transaction in 4 byte words.
CHNL_RX_OFF[30:0]	I	Offset in 4 byte words indicating where to start storing received data (if applicable in design).
CHNL_RX_DATA[DWIDTH-1:0]	I	Receive data.
CHNL_RX_DATA_VALID	I	High if the data on CHNL_RX_DATA is valid.
CHNL_RX_DATA_REN	O	When high and CHNL_RX_DATA_VALID is high, consumes the data currently available on CHNL_RX_DATA.
CHNL_TX_CLK	O	Provide the clock signal to write data to the outgoing FIFO.
CHNL_TX	O	Set high to signal a transaction. Keep high until all outgoing data is written to the FIFO.
CHNL_TX_ACK	I	Will be pulsed high for at least 1 cycle to acknowledge the transaction.
CHNL_TX_LAST	O	High indicates this is the last send transaction in a sequence.
CHNL_TX_LEN[31:0]	O	Length of send transaction in 4 byte words.
CHNL_TX_OFF[30:0]	O	Offset in 4 byte words indicating where to start storing sent data in the PC thread's receive buffer.
CHNL_TX_DATA[DWIDTH-1:0]	O	Send data.
CHNL_TX_DATA_VALID	O	Set high when the data on CHNL_TX_DATA valid. Update when CHNL_TX_DATA is consumed.
CHNL_TX_DATA_REN	I	When high and CHNL_TX_DATA_VALID is high, consumes the data currently available on CHNL_TX_DATA.

The value of DWIDTH will be either 32, 64, or 128.

Table 2.1: RX - TX interface Signals

For better understanding of the RX and TX procedures an example of each with their timing diagrams are provided bellow.

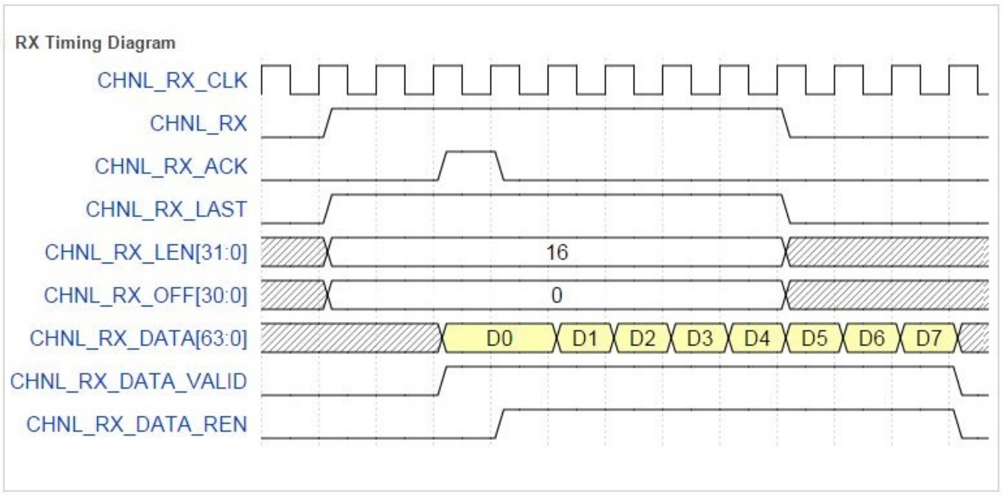


FIGURE 2.11: Timing diagram for receiving data

Figure 2.11 shows the RIFFA channel receiving a data transfer of 16 words (64 bytes). When CHNL_RX is high, CHNL_RX_LAST, CHNL_RX_LEN, and CHNL_RX_OFF will all be valid. In this example, CHNL_RX_LAST is high, indicating to the user core that there are no other transactions following this one and that the user core can start processing the received data as soon as the transaction completes. CHNL_RX_LAST may be set low if multiple transactions will be initiated before the user core should start processing received data. Of course, the user core will always need to read the data as it arrives, even if CHNL_RX_LAST is low.

In the example CHNL_RX_OFF is 0. However, if the PC specified a value for offset when it initiated the send, that value would be present on the CHNL_RX_OFF signal. The 31 least significant bits of the 32 bit integer specified by the PC thread are transmitted (due to packing constraints). The CHNL_RX_OFF signal is meant to be used in situations where data is transferred in multiple sends and the user core needs to know where to write the data (if, for example it is writing to BRAM or DRAM).

The user core must pulse the CHNL_RX_ACK signal high for at least one cycle to acknowledge the receive transaction. The RIFFA channel will not recognize that the transaction has been received until it receives a CHNL_RX_ACK pulse. Note that data on CHNL_RX_DATA may arrive before CHNL_RX_ACK is pulsed, but the FIFO will never overflow.
The combination of CHNL_RX_DATA_VALID high and CHNL_RX_DATA_REN high consumes the data on CHNL_RX_DATA. New data will be provided until the

FIFO is drained. Note that the FIFO may drain completely before all the data has been received. The CHNL_RX signal will remain high until all data for the transaction has been received into the FIFO. Note that CHNL_RX may go low while CHNL_RX_DATA_VALID is still high. That means there is still data in the FIFO to be read by the user core. Attempting to read (asserting CHNL_RX_DATA_REN high) while CHNL_RX_DATA_VALID is low, will have no affect on the FIFO. The user core may want to count the number of words received and compare against the value provided by CHNL_RX_LEN to keep track of how much data is expected.

In the event of a transmission error, the amount of data received may be less than the amount expected (advertised on CHNL_RX_LEN). It is the user core's responsibility to detect this discrepancy if important to the user core.

RIFFA channel's TX interface is nearly symmetric to the receive example. In figure 2.12 RIFFA channel is sending a data transfer of 16 words (64 bytes).

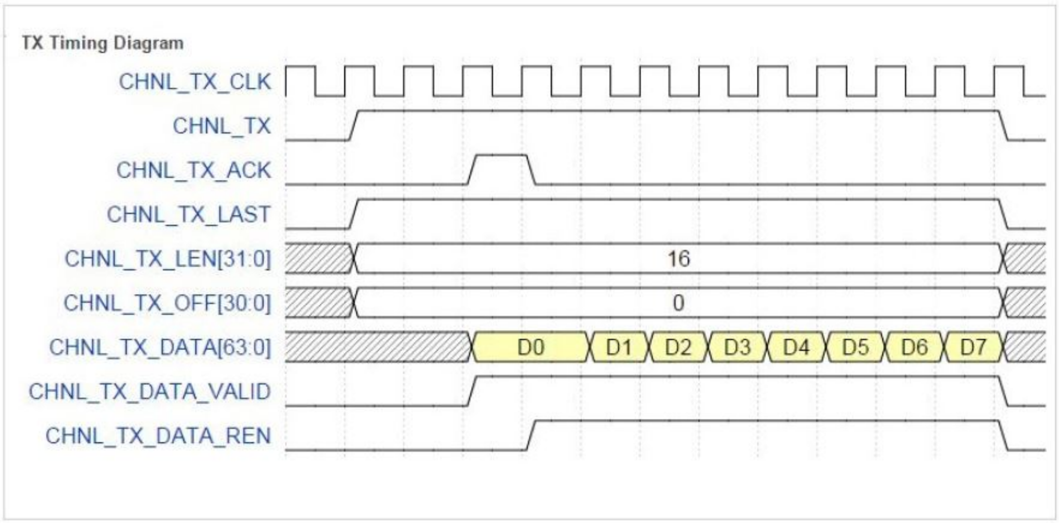


FIGURE 2.12: Timing diagram for transmitting data

The user core sets CHNL_TX high and asserts values for CHNL_TX_LAST, CHNL_TX_LEN, and CHNL_TX_OFF for the duration CHNL_TX is high. CHNL_TX must remain high until all data has been consumed. RIFFA will expect to read CHNL_TX_LEN words from the user core. Any more data provided may be consumed, but will be discarded. The user core can provide less than CHNL_TX_LEN words and drop CHNL_TX at any point. Dropping CHNL_TX indicates the end of the transaction. Whatever data was consumed before CHNL_TX was dropped will be sent and reported as received to the software thread.

As with the receive interface, setting CHNL_TX_LAST high will signal to the PC thread to not wait for additional transactions (after this one). Setting CHNL_TX_OFF will cause the transferred data to be written into the PC thread's buffer starting CHNL_TX_OFF 4 bytes words from the beginning. This can be useful when sending multiple transactions and needing to order them in the PC thread's receive buffer. CHNL_TX_LEN defines the length of the transaction in 4 byte words.

As the CHNL_TX_DATA bus can be 32 bits, 64 bits, or 128 bits wide, it may be that the number of 32 bit words the user core wants to transfer is not an even multiple of the bus width. In this case,CHNL_TX_DATA_VALID must be high on the last cycle CHNL_TX_DATA has at least 1 word to send. The channel will only send as many words as is specified by CHNL_TX_LEN. So any additional data consumed, past the last word, will be discarded. Shortly after CHNL_TX goes high, the RIFFA channel will pulse high the CHNL_TX_ACK and begin to consume the CHNL_TX_DATA bus. The combination of CHNL_TX_DATA_VALID high and CHNL_TX_DATA_REN high will consume the data currently on CHNL_TX_DATA. New data can be consumed every cycle. After all the data is consumed, CHNL_TX can be dropped. Keeping CHNL_TX_DATA_VALID high while CHNL_TX_DATA_REN is low will have no effect.

2.2.3 RIFFA Sorfware API

The software interface is provided by bindings for C/C++. After installation all bindings are available in their respective runtime environments. The API is based on the notion of channels. RIFFA can be configured to support between 1 - 12 independent channels. Each channel connects to an IP core and can be addressed by specifying the channel number from the user application. The channels are independent and thread safe. At most one thread should be used to access a single channel. The C/C++ bindings are used by including the riffa.h header file and linking with the -lriffa library.

API

- **int fpga_list(fpga_info_list * list);**

  Populates the fpga_info_list pointer with all FPGAs registered in the system. See riffa_driver.h for the fpga_info_list definition. Returns 0 on success, a negative value on error.

  list - Pointer to a fpga_info_list struct to populate.

  Returns: 0 on success, a negative value on error.

- **fpga_t * fpga_open(int id);**

  Initializes the FPGA specified by id. On success, returns a pointer to a fpga_t struct. On error, returns NULL. Each FPGA must be opened before any channels can be accessed. Once opened, any number of threads can use the fpga_t struct pointer.

  id - Identifier for the FPGA (in single FPGA installations, this is always 0).

  Returns: A fpga_t struct pointer or NULL.

- **void fpga_close(fpga_t * fpga);**

  Cleans up memory/resources for the FPGA specified by the descriptor.

fpga - Pointer to fpga_t struct.

Returns: Nothing.

- **int fpga_send(fpga_t * fpga, int chnl, void * data, int len, int destoff, int last, long long timeout);**

  fpga - Pointer to fpga_t structure.
  chnl - Channel number over which to communicate.
  data - Pointer to array of data to send.
  len - Length of data to send, in (32 bit) words. Thus a value of 4 means send 16 bytes.
  destoff - Value sent to FPGA core to indicate where to start writing this data. Only the least significant 31 bits are sent (not all 32).
  last - If 1, this transfer is the last in a sequence of transfers. If 0, this transfer is not the last in a sequence of transfers (more transfers to come).
  timeout - Timeout value in ms. If 0, no timeout is specified. Otherwise, the PC will wait up to timeout ms in between PC/FPGA communications.

  Sends len words (4 byte words) from data to FPGA channel chnl using the fpga_t struct. The FPGA channel will be sent len, destoff, and last. The value of destoff is used to support sending data across multiple send transactions. Note that only the low 31 bits of this unsigned int are sent. If last is 1, the channel should interpret the end of this send as the end of a transaction. If last is 0, the channel should wait for additional sends before the end of the transaction. If timeout is non-zero, this call will send data and wait up to timeout ms for the FPGA to respond (between packets) before timing out. If timeout is zero, this call may block indefinitely. Multiple threads sending on the same channel may result in corrupt data or error. This function is thread safe across channels.

  Returns: The number of words sent.

- **int fpga_recv(fpga_t * fpga, int chnl, void * data, int len, long long timeout);**

  fpga - Pointer to fpga_t structure.

  chnl - Channel number over which to communicate.

  data - Pointer to buffer array where received data will be written.

  len - Length of buffer array, in (32 bit) words. Thus a value of 4 means send 16 bytes.

  timeout - Timeout value in ms. If 0, no timeout is specified. Otherwise, the PC will wait up to timeout ms in between PC/FPGA communications.

  Receives data from the FPGA channel chnl to the data pointer, using the fpga_t struct. The FPGA channel can send any amount of data, so the data array should be large enough to accommodate. The len parameter specifies the actual size of the data buffer in words (4 byte words). The FPGA will specify an offset value which will determine where received data will start being written. If the amount of data plus the offset exceed the size of the data array, then the additional data will be discarded. If timeout is non-zero, this call will wait up to timeout ms for the FPGA to respond (between packets) before timing out. If timeout is zero, this call may block indefinitely. Returns the number of words received to the data array.

  Returns: The number of words received to the data array.

- **void fpga_reset(fpga_t * fpga);**

  Resets the state of the FPGA and all transfers across all channels. This is meant to be used as an alternative to rebooting if an error occurs while sending/receiving. Calling this function while other threads are sending or receiving will result in unexpected behavior.

  fpga - Pointer to fpga_t structure.

  Returns: Nothing.

2.3 Vivado Design Suite

This project was desing and implemented on Xilinx's Vivado Design Suite. Vivado is a software suit for synthesis and analysis of HDL designs, superseding Xilinx ISE with additional features for system-on-chip development and high-level synthesis. It includes an in-built logic simulator and high-level synthesis, with a toolchain that converts C code into programmable logic. Vivado HLS was used for the creation of accelarators that were attached to RIFFA channels and monitored.

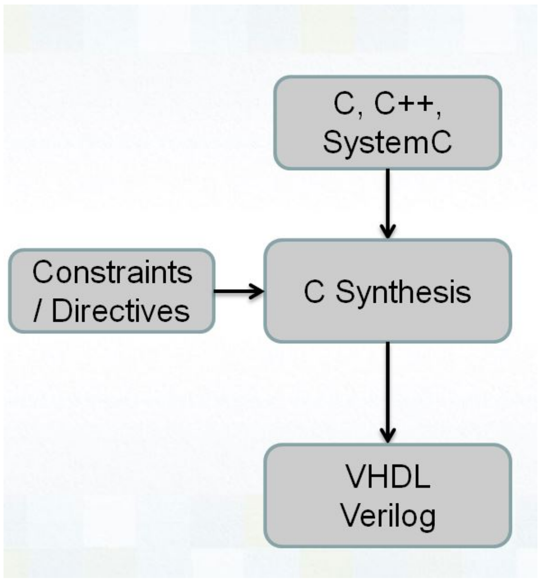


Figure 2.13: Vivado High Level Synthesis

Chapter 3

RIFFA Monitor Core Design & Implementation

This chapter presents the RIFFA 2.0 Monitor Core. The purpose, design and implementation of every module will be described in depth, followed by a brief analysis of the C based API. It concludes with a summary of the different architectural approaches while searching for a viable solution in the transparency and compatibility of the project.

3.1 Purpose & Approach

The purpose of this project is to expand the functionality that RIFFA framework provides with metrics and logging, creating a profiling mechanism for FPGA based SoC. The original concept was that Monitor core would make use of hardware performance counters just like those built into modern microprocessors. Although this was a simple task, the variety of IP cores users can attach to RIFFA channels made it harder to come up with a universal solution. And while the project was already evolving to a viable state the next dilemma emerged. How to perform the extra operations but keep their existence transparent at user level, and at the same time retain compatibility with predated projects.

Every RIFFA project consists of two main parts, the user IP core (accelerator

/ SoC) to be instantiated in riffa_adapter module, and the corresponding software using RIFFA API to access the core. Accelerators should be able to instantiate as is without additional logic or interconnections, besides when users manually demand a specific event to be monitored. At software level the objective was to leave the original API intact. Furthermore all existing RIFFA projects should be able to operate properly on the monitored framework.

Apart from the functionality, compatibility and transparency that is expected the next major factor determining the success of the implementation is the percentage of resource occupation. Monitor had to be designed as a lightweight module, using only the minimum amount of combinational logic possible. It would be meaningless if there was not enough resources left for the actual SoC - accelerator.

3.2 Event-Based Profiler

Monitor started as a single hardware counter measuring the duration of accelerator usage and gradually evolved into a tool capable of monitoring and recording any activity in the user generated modules. That activity will be mentioned as events and consists of RIFFA RX /TX engine usage (channel transactions) and user specified triggers.

Similar to event-based profilers triggering on certain events in the code, RIFFA 2.0 Monitor exhaustively monitors and records every trigger associated with his appointed set of events. User specified triggers are optional and they have to be manual wired to the Monitor. With this simple step a maximum of 16 different events can be monitored and logged simultaneously. For each one Monitor core will count the number of occurrences, measure their duration and log them. The latest version can support a maximum of 16 unique events.

Since RIFFA Monitor is a hardware design event's definition differs from original software profilers. In software engineering an event is a unique trigger like an exception, a function call or a specific mark in the code. On the other hand a hardware event usually has a solid duration. For example a hardware event could be the signal of a sole wire, a specific state on an FSM or even an active transaction of the AXI BUS interface.

3.3 High Level Design

In order to assign a custom IP core to RIFFA channels we have to instantiate it inside riffa_adapter module in user-space provided and we are obliged to use the proper interconnects of RX and TX interfaces. In Figure 2.3 of section 2.2 we show a high level diagram of the original RIFFA architecture.

After a series of trials in different architectural approaches we concluded that the most easy-to-use and compatible with predated projects solution is to instantiate a single Monitor core inside riffa_adapter module, and redirect user space for instantiations one level deeper in module hierarchy. One Monitor core will supervise all channels and monitor all user IP cores. To achieve this profiler core exists between user cores and riffa_adapter and intercepts all TX and RX signals. In figure 3.1 we display the high level architecture of RIFFA with the interfering Monitor module acting as a wrapper for all user cores.

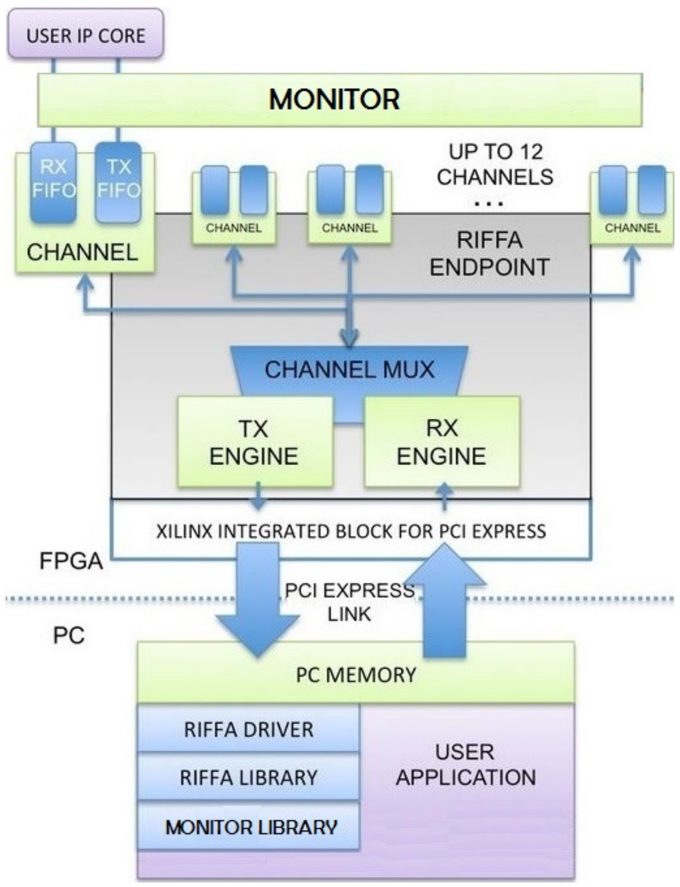


FIGURE 3.1: RIFFA with Monitor high level architecture

3.4 Module Analysis

In this section we will begin to tear down the Monitor core starting from the top module profiler.v. We will see a detailed analysis of the logic behind the FSM controlling all the functionality, the combinational logic intercepting the wiring of TX and RX interface and the level of elasticity we can achieve with various parameters. Monitoring and Logging are handled at a lower hierarchy level offering the ability to instantiate only what is essential over each project for economy in resource occupancy.

3.4.1 Monitor Top Module

Monitor's top module actually implements most of the required logic. It is designed for the 128-bit version of RIFFA 2.0 framework. The clock used is RIFFA's user_clock at 250 Mhz (4ns period). The pre-definition of every register's value through initialization after fpga programming makes it unnecessary to insert a global reset network.

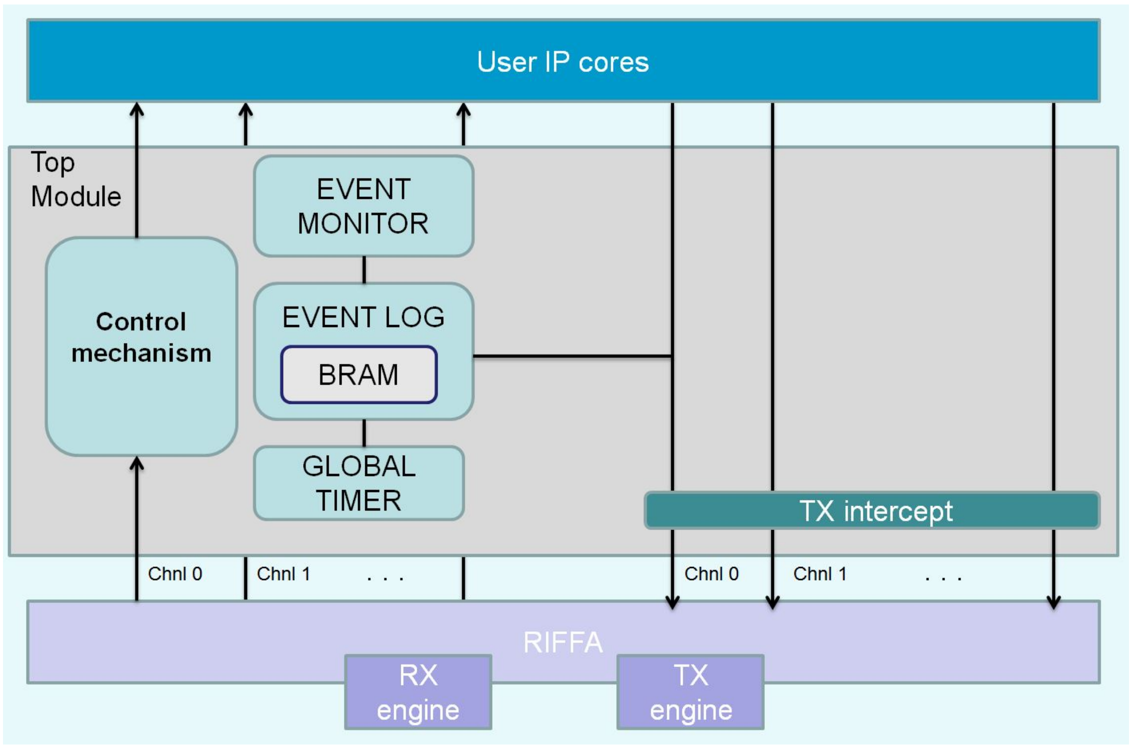


Figure 3.2: RIFFA Monitor block diagram

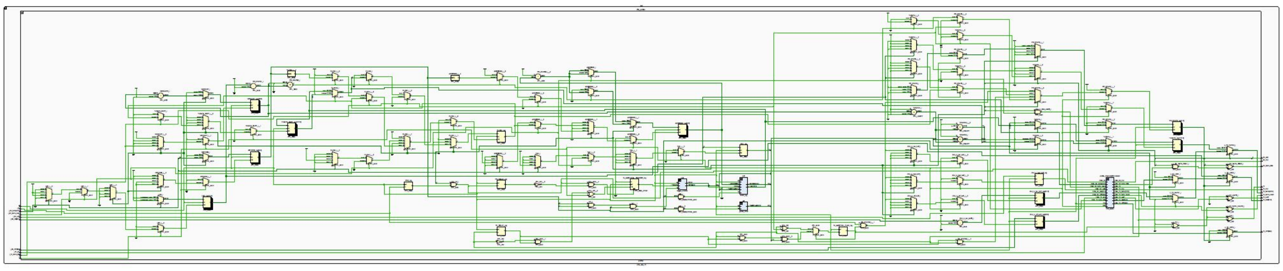


FIGURE 3.3: Monitor's RTL schematic

To the upper RIFFA hierarchy levels Monitor seems just like any other IP core instantiated and connected to RIFFA channels. RIFFA framework offer a maximum of 12 channels. Through them accelerators are able to communicate with software using the PCI express link. Monitor control unit is able to receive orders without binding any channel. It connects to all available channels and intercepts every transaction. So when a user core get instantiated inside the redirected user space neither RIFFA modules nor the IP core should be altered to conform with Monitor's existence. Furthermore If we don't manually communicate with Monitor through his driver, it will silently let user cores to run and will record events in the LOG.

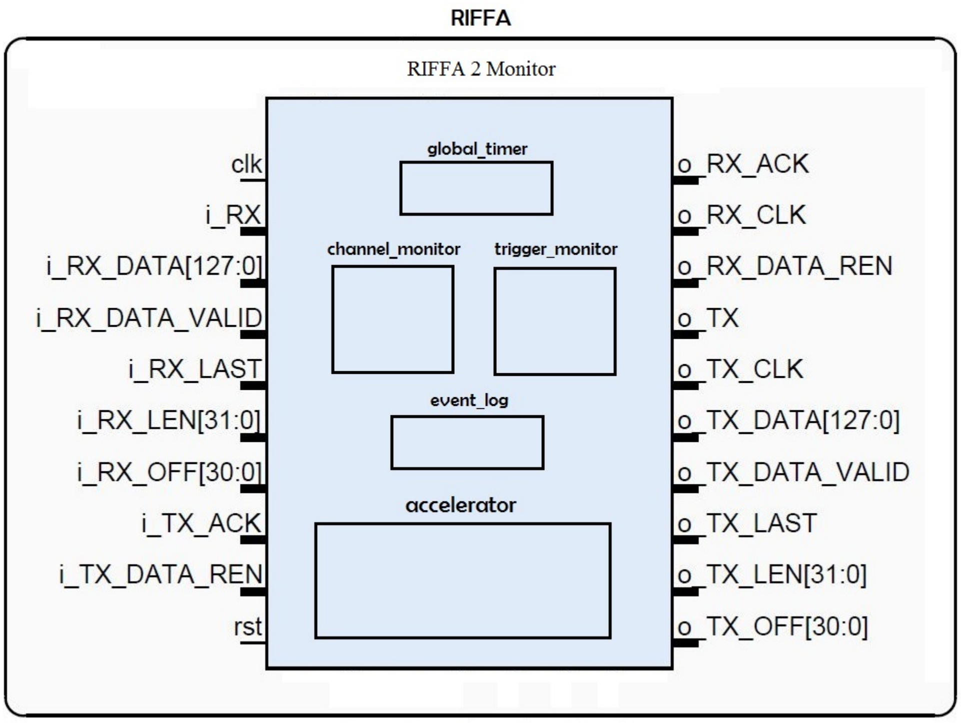


FIGURE 3.4: Monitor's abstract view and I/O

The interception of RX interface gives Monitor the ability to communicate with software without using extra resources and it will be discussed in section 3.4.1.1. On the other hand interception of TX interface gave us the ability to keep transmission active for one extra cycle, providing 128 extra bits of information sent back to the software level. The operation of attaching additional data in accelerator's transmission is called TAIL and will be covered in section 3.4.1.2.

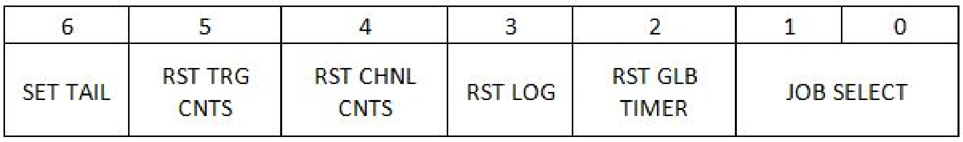
3.4.1.1 Control Mechanism

Operation Code

Monitor must be able to communicate with software without binding any RIFFA channel. To achieve this without altering the original drivers or RIFFA's higher level modules the only way is to share a channel with a user IP/accelerator core. now each time RIFFA has an incoming transaction through this channel the incoming data are aimed at either the Monitor or the accelerator. By adding a header to the data frame we now have an operation code to make this distinction clear. Profiler is sharing channel 0 and will be solely responsible start every RX transaction on this channel. The two LSB act as Job Select and indicate the proper operation.

- FORWARD - Give channel control to user IP core.
- INFO - Transmit the specified information frame.
- FLUSH - Transmit all valid entries in Log's BRAM.
- SET - Reset specific counters and enable or disable tail.

Bits 2 to 6 are read only when operation selected is SET (2 LSB = = 11) , and each on is associated with a unique operation.



6	5	4	3	2	1	0
SET TAIL	RST TRG CNTS	RST CHNL CNTS	RST LOG	RST GLB TIMER	JOB SELECT	

Table 3.1: Monitor's OPCODE bits

Users won't have to worry about sending the correct OPCODE since it is already handled by Monitor's API. On every transaction targeting the Soc - accelerator a header with OPCODE 00 (FORWARD) will be automatically attached.

FSM

Monitor's top module has to perform several actions including event monitoring and logging, and also has to share a channel with a user core in order to receive orders and send back information. Even though profiler receives orders and sends data only through channel 0 to minimize the combinational logic, there still remain considerable amount of tasks to be organized.

To coordinate every operation we need a solid and reliable FSM impervious to poorly designed user cores and their misbehaviors and simultaneously simple, light and with minimum latency. Latest Profiler's version uses a simplified FSM with 4 states.

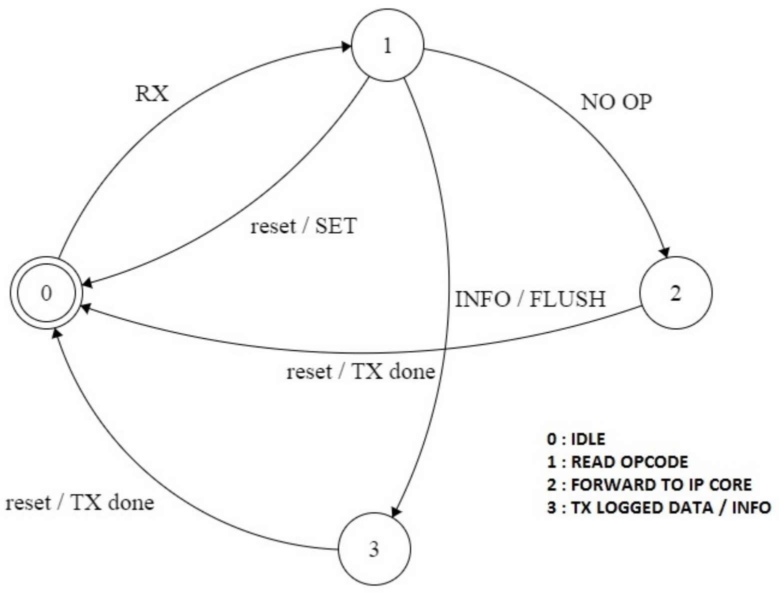


FIGURE 3.5: Monitor's Finite State Machine

00 - IDLE

Idle is Monitor's initial state. It is also the state to return to after riffa reset, giving

Monitor the ability to recover even if user IP cores have undetermined behavior. Monitor will remain in this state until RX signal of channel 0 is high. This indicates either the start of transmission from software to the user core connected on channel 0 or a request to the Monitor. Since we are no longer in idle state control is handed over to state 01.

01 - READ OPCODE

Monitor's driver is responsible to attach the right header on the incoming data. As soon they are valid, Monitor reads the first 128 bits and freezes RX procedure by dropping channel's RX read enable signal. Now depending on the header that is basically an operation code (OPCODE) the proper job will be selected. Even though we don't need all 128 bits, using a full transfer as a header is chosen so that we don't mess with data alignment. A total of 7 bits are used as OPCODE since it was a fair trade between clarification and resource economy. The next state will be determined depending on OPCODES's two LSB and at this point Monitor has to select from four different operations.

1. Do nothing, go to state 10. Transaction was actually targeting accelerator on channel 0.
2. Transmit through channel 0 useful information about channel usage, last measurements, timestamp and parameter values, go to state 11.
3. Transmit through channel 0 all valid entries recorded in the Log, go to state 11.
4. Reset specific counters, activate or deactivate tail, return to idle state 00.

10 - FORWARD TO USER CORE

Incoming Data was sent to accelerator attached on channel 0. The 128-bit header is removed since its already consumed by Monitor at state 01 and the remaining data will be delivered to the accelerator. From this moment on accelerator has complete control over channel 0 when he is ready he will raise the channel's RX read enable flag. A specific event must trigger the moment when Monitor can regain channel management. Using a hard coded trigger would require manual addition on every IP core connected on channel 0, with extra logic and an extra interconnection. As a result efforts for transparency and compatibility would go to waste. Using as consensus the fact that almost every time a RX transaction to the IP core is followed by the immediate response with a TX transaction, Monitor will regain channel 0 control after TX is finished and return to IDLE state. It was achieved with minimal additional logic because all RX and TX signals are already intercepted. One extra cycle of delay before we jump to IDLE is added if tail is enabled.

11 - TRANSMIT (LOG OR INFO)

In this state Monitor will use RIFFA's TX engine to transmit through channel 0. Two distinct operations can be performed, flush the log or return values of specific registers. The first one will return all entries recorded in Log's BRAM. The mechanism that forwards entries from BRAM to TX_data buffer is aware of transmission delays and and uses a secondary register array to avoid data loss. The flush procedure and the structure of the entries will be presented and explained in section 3.3.4.

The second operation has as default configuration the return of a data frame containing parameter values, tail setup, number of user-specified events, size of Log's BRAM, number of valid entries, the current timestamp and recorded values of events usage and duration. The structure of this frame can be efficiently reconfigured to match user's needs.

3.4.1.2 Tail

Appending a number of bits with extra information on accelerator's transactions was in fact the original functionality and source of the whole project. Started as part of accelerator's logic was later removed since all operation requiring major changes to user IP cores were excluded. With the massive intercepting of every signal and complete knowledge of RIFFA's TX engine timing diagram is now possible to attach the extra bits without even interfering with accelerator's TX FSM. Since Monitor is designed for the 128-bit version of RIFFA endpoint, just one extra cycle of active transmission offers 128-bit of information. This was not only straight forward to implement but also sufficient for the amount of data to be attached. As default return values are chosen the current timestamp and the duration of the corresponding accelerator usage. Those values can be changed to match user's needs.

On the software level when the final user want to receive the extra information attached to the original data he should manually increase the number of the received words and adjust the buffer's size accordingly. If he doesn't do that the result is not catastrophic, the tailed data will just be ignored by software. One small detail that deserves some notice is that if word count of outgoing data are not a multiple of 4, the tail is actually more than one 128-bit frame. The missing words of last frame will be filled with zeros so that tail is completely aligned in a single transfer frame.

Once tail is enabled through the correct OPCODE register TAIL value will be set to 1. To determine if TX has finished we need one more register named TAIL FLAG. This flag remains high for all the duration of the TX plus one cycle.

3.4.1.3 Parameters

Monitor core expands the list of parameters that were inherited from riffa_adapter module with seven new that will be presented bellow. RIFFA alone is already resource consuming and if we expect to use it with massive designs at least the profiling part should occupy as less space on FPGA as possible. With enough parameters to specify required functionality and number of events, Monitor core will be generated with only the necessary amount of modules and part of the designed logic.

C_DATA_WIDTH

RX/TX interface data width. This parameter is inherited by RIFFA top module and passed directly to user IP cores. The current Monitor version is designed at 128-bit but with further modification 32-bit and 64-bit support will be achieved.

C_NUM_CHNL

Number of RIFFA channels (1-12). The second parameter coming directly from

RIFFA. Increasing the number of channels does not significantly raise Monitor's resource usage, since the controlling mechanism is only connected to channel 0.

BRAM_SIZE

Log's memory size in words. Entries are 64 bit wide so a total of BRAM_SIZE/2 events can fit in the Log. Default value is 2048 which translates in 128KB, 144KB with parity.(4 x 36KB primitive BRAMs are used)

MONITOR_CHANNELS

If high information of each channel usage will be recorded creating the pseudo-event of a full RX-accelerator usage-TX cycle. That info consists of 64-bit, 16MSB for occurrences number and 48LSB for duration.

LOG_CHANNEL

If high RX and TX of each channel will be recorded in the log , providing a timing analysis for the event mentioned above.

TRIGGERS_NUM

Number of user defined Events (Triggers). Default value is 0 since it is an optional function and users have to manually attach their events them to TRIGGER wire.

LOG_TRIGGER

If high Triggers will be saved as events in the log. Default value is 0 since those triggers are optional events that users want to monitor and they have to manually attach them to TRIGGER wire.

SUM

If high durations (triggers or channel usage) will be accumulated. If low only the last duration will be available.

MERGE_PULSES

If high a total of 16 different events can be monitored (instead of 8). The number of events depends on the size of their ID when saved in the Log. Since we use

64-bit entries with 48-bit timestamps ID consists of the remaining 16 bits. Default option (0) uses 2 bits per event, on for the begging and 1 for termination since it simplifies decoding and visualizing the recored data.

If both LOG_CHANNELS and LOG_TRIGGERS are 0 there is no need to instantiate event_log and its BRAM. Likewise MONITOR_CHANNELS and TRIGGERS_NUM define the number of trigger_monitor modules. Properly setting those parameters can minimize Monitor's size on FPGA.

3.4.2 Global Timer

This minor module has the sole job of counting every cycle since first fpga programming and can only be reseted manually with PROFILER_SET function. It uses 48 bit counter which means: $2^{48} * 4$ ns = 13.0312489 days of nonstop operation. Output of global_timer module is the 48bit timestamp which can be sent directly to the Monitor's driver through TX engine with TAIL and INFO operations or used in the logging procedure.

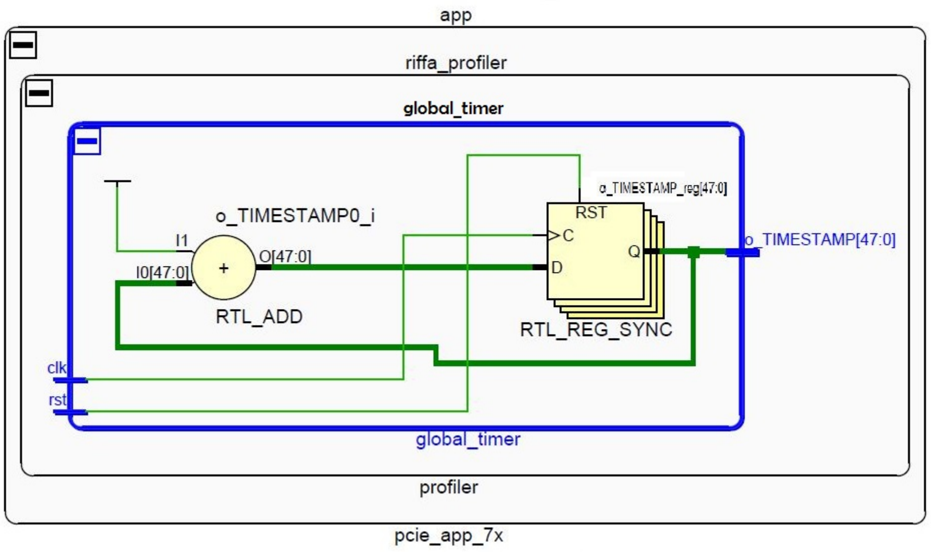


FIGURE 3.6: Global Timer RTL schematic

In order to restart global timer Monitor must be provided with the proper OPCODE, regardless of riffa_reset. An important detail is that timer cannot be restarted if Log is not reseted also. This way we avoid mixing new entries with invalid outdated ones.

3.4.3 Monitor Submodule

Real time monitoring of events is achieved with the assistance of module trigger_monitor. It is responsible for counting event occurrences, measuring their duration, and generating pulses at their beginning and ending. Those pulses will notify Monitor and new entries will be recorded in the Log. For each event monitor module uses a 16-bit counter to counter the occurrences and a 48-bit to measure the last duration or accumulate total duration depending on parameter SUM. All counters combined are propagated with o_INFO output to the parent module(Monitor's top module) and will be used at INFO and TAIL operations.

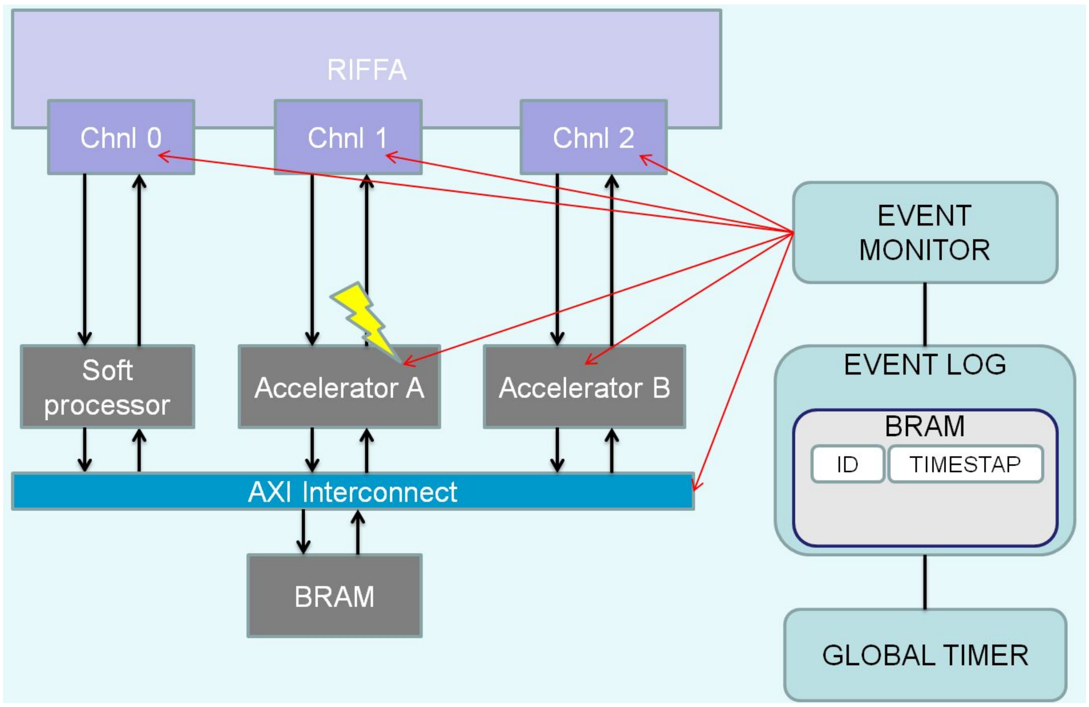


FIGURE 3.7: Event Monitor

Generated pulses START and STOP reach top module through output o_ID. If parameter MERGED is high instead of apointing an exclusive bit to every signal, START[i] and STOP[i] will be compined. This way we can increase the number of monitored events from 8 to 16.

RIFFA Monitor will instantiate up to two versions of this module, one for channels transactions and one for user specified events. Similar to global_timer, reseting the counters is achieved by providing the proper OPCODE.

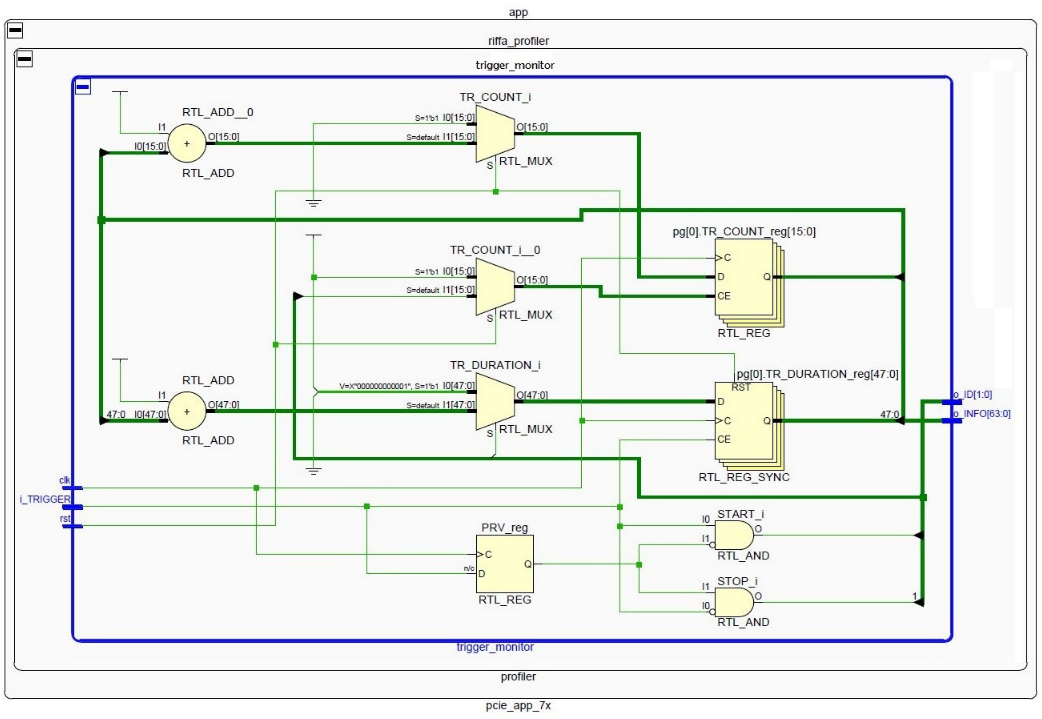


FIGURE 3.8: Monitor Submodule RTL schematic

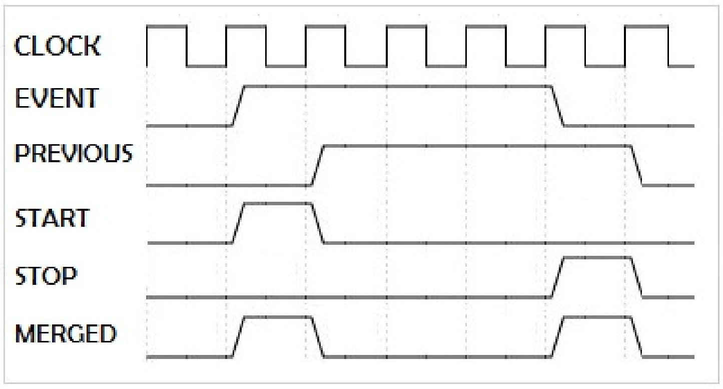


FIGURE 3.9: Monitor module ID generation

3.4.4 Event Log

Module event_log serves as RIFFA Monitor's memory in which the moments that each event begin or terminate are recorded. Each entry consists of 64-bits.

16-bits to classify the trigger and 48-bit as a time stamp. The default number of entries is 2048, chosen to be small related to total resources and on the same time sufficient.

Event_log instantiates and manages a 128KB dual port BRAM primitive module with a 64-bit wide write port (A) and a 128-bit wide read port (B). Only one record can be added per cycle, and it is acceptable since in the 16-bit ID are encoded all possible combinations of the monitored events. When memory is FLUSHed two entries are read per cycle to fill the 128 available bits of the DATA frame. A log(#entries)-bit register is used to keep the current address and in case of memory overflow the logic presented below prevents the out-of-order fetching of valid entries. The module's RTL shcematic is presented in figure 3.6.

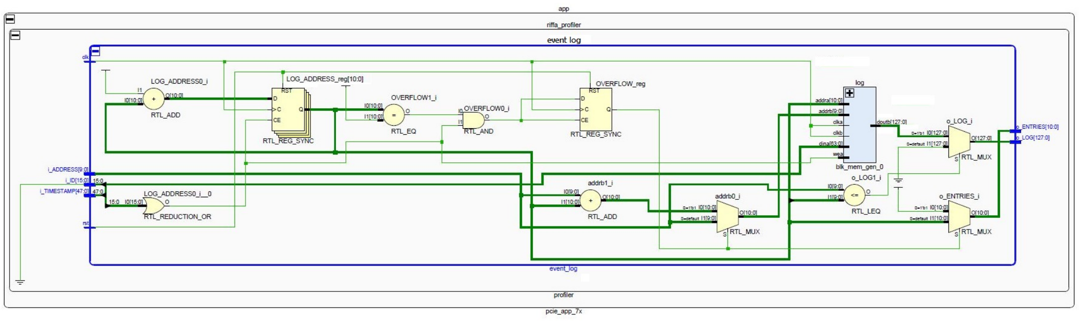


FIGURE 3.10: Event Log Module RTL schematic

BRAM FLUSH

This is the procedure of requesting and downloading all the recorded entries of the Log. While the user cores - accelerators are operating all predefined and manually specified events are recorded in the Log. To export those entries users have to call the function PR_LOG, built in the software interface. OPCODE 0000010 signaling the FLUSH operation will be provided to Monitor's control mechanism. BRAM entries will be streamed through TX engine taking into consideration all possible delays from RIFFA interface. Additional options for displaying and visualization of the downloaded entries are provided and will be presented in software API section.

3.5 Driver

The software api extends the already installed RIFFA driver with additional functions. To use an accelerator attached on the monitored RIFFA framework users have to include in their source code Monitor's library. For fast reconfigurability and following RIFFA's simplified API structure driver is compressed to a single file.

- **int PR_fpga_send(fpga_t * fpga, int chnl, void * data, int len, int destoff, int last, long long timeout)**

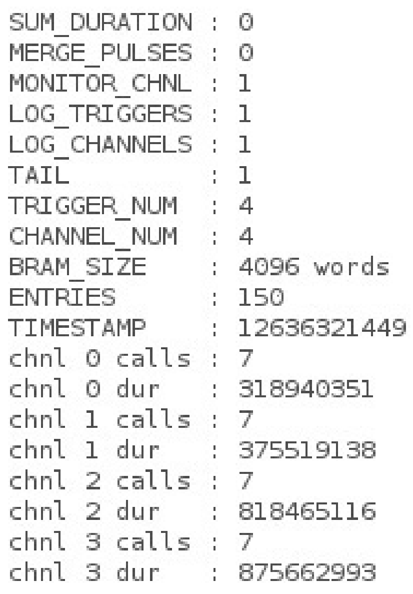
  fpga - Pointer to fpga_t structure.
  chnl - Channel number over which to communicate.
  data - Pointer to array of data to send.
  len - Length of data to send in words.
  destoff - Value sent to FPGA core to indicate where to start writing this data.
  last - If 1, this transfer is the last in a sequence of transfers.
  timeout - Timeout value in ms. If 0, no timeout is specified. Otherwise, the PC will wait up to timeout ms in between PC/FPGA communications.

  All user calls to RIFFA's fpga_send are redirected to PR_fpga_send. If channel selected is 0 a header of 128 zeros will be attached to the send_data buffer. Afterwards the original function fpga_send is called with the same set of parameters. Only len's value and the contents of data will be modified if needed.

- **unsigned long PR_INFO(fpga_t * fpga, int print)**

  fpga - Pointer to fpga_t structure.
  print - Optional display of downloaded information in console.

  The initial version of this function was supposed to return a single Timestamp. While profiler was evolving and enriched with additional metrics and log the core purpose of PR_INFO changed completely several times. Instead of wasting resources on logic for data selection, all available metrics recorded will be downloaded with a single function call. Additional delay of a couple cycles is insignificant in front of the overhead for one transaction. The current timestamp is returned and all additional info are printed in console. As future development a struct can be populated with the received data.



```
SUM_DURATION : 0
MERGE_PULSES : 0
MONITOR_CHNL : 1
LOG_TRIGGERS : 1
LOG_CHANNELS : 1
TAIL         : 1
TRIGGER_NUM  : 4
CHANNEL_NUM  : 4
BRAM_SIZE    : 4096 words
ENTRIES      : 150
TIMESTAMP    : 12636321449
chnl 0 calls : 7
chnl 0 dur   : 318940351
chnl 1 calls : 7
chnl 1 dur   : 375519138
chnl 2 calls : 7
chnl 2 dur   : 818465116
chnl 3 calls : 7
chnl 3 dur   : 875662993
```

FIGURE 3.11: Output of INFO function call

- **void PR_SET(fpga_t * fpga, int TAIL, int RST_TRG_CNTS,**
  int RST_CHNL_CNTS, int RST_LOG,
  int RST_GLTIMER, int print)

fpga - Pointer to fpga_t structure.
TAIL - enable or disable TAIL operation.
RST_TRG_CNTS - reset duration and count of every user specified event.
RST_CHNL_CNTS - reset duration and count of every channel transactions
RST_LOG - reset the log by setting valid entries number to zero. New entries will overwrite outdated ones.
RST_GLTIMER - restart global timer.
print - Optional display of setting in console.

This function provides the proper OPCODE for reseting specific counters and enable or disable TAIL operation. Since all metrics are not connected to global reset network they retain their values after a RIFFA reset call.

- **void PR_LOG(fpga_t * fpga, unsigned short **triggers,**
  long **timestamps, int print, int file,
  int timeline)

fpga - Pointer to fpga_t structure.

triggers - Array of downloaded triggers.

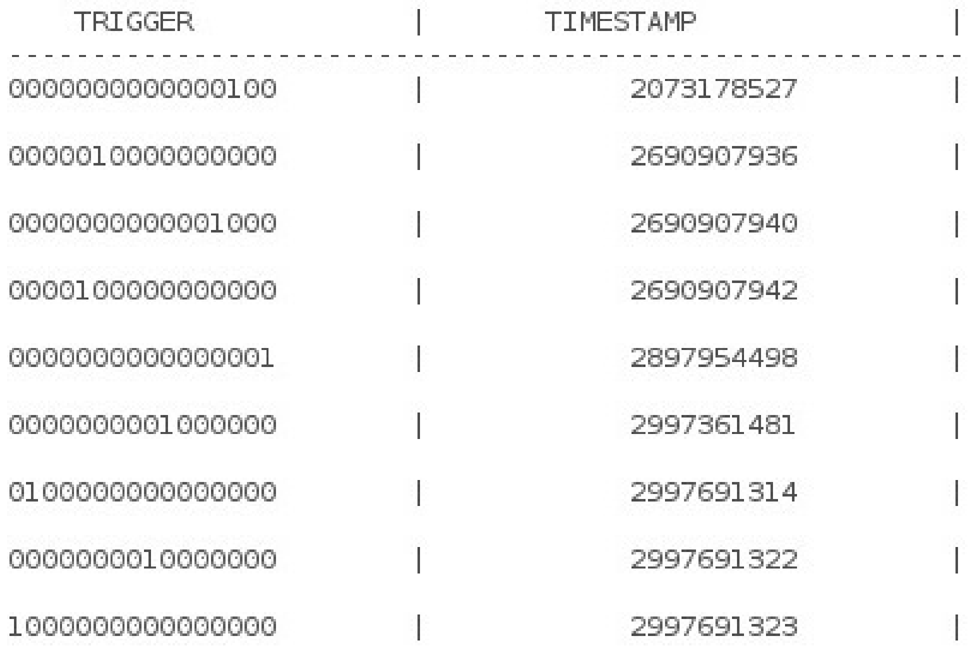
timestamps - Array of downloaded timestamps.

print - Optional display of downloaded entries in console.

file - Optional printing of downloaded entries in file.

timeline - If not 0 a basic visualization of downloaded entries will be printed in file.

The continuation of FLUSH operation on software side is implemented in PR_LOG function. The downloaded entries are displayed, printed in raw and expanded form and visualized in a minimalistic timeline. A console output example can be seen in figure 3.10.



```
   TRIGGER             |      TIMESTAMP          |
---------------------------------------------------
0000000000000100       |        2073178527       |

0000010000000000       |        2690907936       |

0000000000001000       |        2690907940       |

0000100000000000       |        2690907942       |

0000000000000001       |        2897954498       |

0000000001000000       |        2997361481       |

0100000000000000       |        2997691314       |

0000000010000000       |        2997691322       |

1000000000000000       |        2997691323       |
```

Figure 3.12: Output of LOG function call

Each line is a recorded entry in the Log. Trigger column contains the 16-bit IDS which decodes in every combination of events. Reading the ID from right to left every 2 bits correspond to the beginning and termination of the defined events. If timeline parameter is non zero a basic visualization will be print in timeline.txt. The overall period since FPGA programming or Global Timer reset will be split into a number of sections equal to timeline value and will be visualized according to the logged entries.

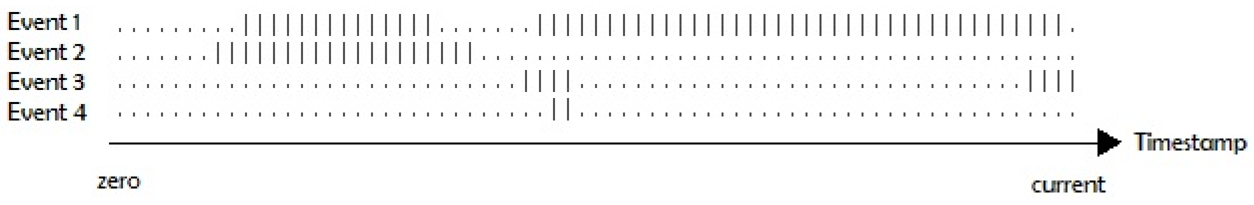


FIGURE 3.13: Basic Vilsualization in the form of a Timeline

3.6 Architectural exploration

1. Before Monitor become a stand-alone module he was implemented as additional lines of code in the accelerator we wanted to monitor. A couple of hardware counters measured execution time and the result was embedded in the returned data stream. The attachment was achieved by the proper alteration of accelerator's TX interface handler. And since user created IP cores don't follow a specific design pattern different logic should be implemented for each one.

2. Next generation was developed as a wrapper module. One accelerator was instantiated in one Monitor wrapper. This was the first step towards a universal design for all accelerators. Again hardware counters were used to measure either total accelerator usage or a manually specified event and results were sent to software level with a mechanism similar to TAIL but with several flaws. Only this function was performed so communication with software level was not required.

3. The next attempt inserted a global timer as extra module, accessed by every Monitor_wrapper and apart from duration a timestamp was attached on the outgoing results. We tried to separate Monitor from accelerator and instantiate them on the same hierarchy level but several compatibility issues arose.

Once again operation selection was hard coded in the design so RX signals were propagated as is.

4. The next implementation was a top profiler module containing Global timer, memory blocks for logging and one profiler wrapper for each channel. Since users should be able to download the recorded data a basic communication with profiler and software was necessary. The fact that every channel had an appointed profiler wrapper made it easier to intercept RX signals, but less user friendly. It was also uncombable with accelerators that use more than one channel.

5. In the next to last design Monitor top module was monitoring and logging every event and was communicating with software through all available channels. Tail operation was fully functional but the design lacked optimizations. Limiting controls to a single channel, parameterizing the module generation, minimizing the FSM and merging similar operation led to the current version of RIFFA Monitor.

Chapter 4

Conclusion

4.1 Project Report

A hardware design for SoC monitoring and logging was successfully developed. After a series of testing and debugging RIFFA Monitor has reached a stable state, user-friendly and resource efficient. It has already been used for evaluation of mathematical accelerators and for better understating of the RIFFA communication engine. Compatibility with legacy RIFFA designs added extra value on deserted projects and will be a useful tool for every future work based on RIFFA infrastructure.

4.2 In the Future

Reaching a stable version was an important milestone but several tasks and ideas are still waiting to be implemented. Extra functionality for sampling will give a more statistical approach though the necessity of such information is still under discussion. Monitoring of standard IPs and AXI bus still requires manual work to be done by user and it could be automated just like RIFFA communication engine. Support for single trigger events without duration could be inserted to avoid wasted entries (2 for 1). Timestamps are chosen to be absolute time in ns for easier understanding of the log but a later version could use time difference and change the balance of ID and TIMESTAMP bit count. RIFFA Monitor at the moment is implemented at 128-bit but 32-bit and 64-bit support is ongoing. Apart from hardware improvements and additions, Monitor's software can be greatly upgraded by adding a graphical user interface. Data interpretation and visualization combined with plenty automatizations will promote RIFFA monitor to a handy and convenient tool for fpga design engineers .

Appendix A

Verilog Source Code

RIFFA Monitor top module

```
`timescale 1ns/1ns

module profiler (clk,o_RX_CLK, i_RX, o_RX_ACK,
  i_RX_LAST, i_RX_LEN, i_RX_OFF, i_RX_DATA,
  i_RX_DATA_VALID, o_RX_DATA_REN, o_TX, i_TX_ACK,
  o_TX_LAST, o_TX_LEN, o_TX_OFF, o_TX_DATA,
  o_TX_DATA_VALID, i_TX_DATA_REN,rst);

parameter C_DATA_WIDTH      = 9'd128;
parameter C_NUM_CHNL        = 4'd1;
parameter BRAM_SIZE         = 16'd4096;
parameter MONITOR_CHANNELS  = 1'b1;
parameter LOG_CHANNELS      = 1'b1;
parameter TRIGGERS_NUM      = 4'd0;
parameter LOG_TRIGGERS      = 1'b0;
parameter SUM               = 1'b0;
parameter MERGE_PULSES      = 1'b0;

input                               clk;
output   [C_NUM_CHNL-1:0]           o_RX_CLK;
input    [C_NUM_CHNL-1:0]           i_RX;
output   [C_NUM_CHNL-1:0]           o_RX_ACK;
input    [C_NUM_CHNL-1:0]           i_RX_LAST;
input   [(C_NUM_CHNL*32)-1:0]           i_RX_LEN;
input   [(C_NUM_CHNL*31)-1:0]           i_RX_OFF;
input   [(C_NUM_CHNL*C_DATA_WIDTH)-1:0] i_RX_DATA;
input   [C_NUM_CHNL-1:0]                i_RX_DATA_VALID;
output  [C_NUM_CHNL-1:0]                o_RX_DATA_REN;

output  [C_NUM_CHNL-1:0]                o_TX_CLK;
output  [C_NUM_CHNL-1:0]                o_TX;
input   [C_NUM_CHNL-1:0]                i_TX_ACK;
output  [C_NUM_CHNL-1:0]                o_TX_LAST;
output  [(C_NUM_CHNL*32)-1:0]           o_TX_LEN;
output  [(C_NUM_CHNL*31)-1:0]           o_TX_OFF;
output  [(C_NUM_CHNL*C_DATA_WIDTH)-1:0] o_TX_DATA;
output  [C_NUM_CHNL-1:0]                o_TX_DATA_VALID;
input   [C_NUM_CHNL-1:0]                i_TX_DATA_REN;
input                                   rst;

reg     DONE                = 1'b0;
reg     RX0                 = 1'b0;
reg     TAIL                = 1'b0;
reg     FLUSH               = 1'b0;
reg     chnl_rx_last_buff   = 1'b0;
reg      [31:0]  chnl_rx_len_buff    = {32{1'b0}};
reg      [30:0]  chnl_rx_off_buff    = {31{1'b0}};
reg      [5:0]   OPCODE              = {6{1'b0}};
reg      [31:0]  LEN                 = {32{1'b0}};
reg      [29:0]  WCOUNT              = {29{1'b0}};
reg      [1:0]   PR_STATE            = {2{1'b0}};
reg      [9:0]   ADDRESS             = {10{1'b0}};
reg      [C_NUM_CHNL-1:0]   TAIL_FLAG  = {C_NUM_CHNL{1'b0}};
reg      [C_NUM_CHNL-1:0]   TX_DELAY   = {C_NUM_CHNL{1'b0}};
reg      [C_DATA_WIDTH-1:0] TxDATA     = {C_DATA_WIDTH{1'b0}};
reg      [C_DATA_WIDTH-1:0] TxDATA_BUFF= {C_DATA_WIDTH{1'b0}};

wire    [C_DATA_WIDTH-1:0]     LOG;
wire    [TRIGGERS_NUM-1:0]     TRIGGER;
wire    [2*TRIGGERS_NUM-1:0]   TR_ID;
wire    [2*C_NUM_CHNL-1:0]     CH_ID;
wire    [15:0]                  ID;
wire    [TRIGGERS_NUM*64-1:0]   TR_INFO;
wire    [C_NUM_CHNL*64-1:0]     CH_INFO;
wire    [47:0]                  TIMESTAMP;
wire    [15:0]                  ENTRIES;

wire                            user_clk;
wire                            riffa_reset;
wire   [C_NUM_CHNL-1:0]     chnl_rx_clk;
wire   [C_NUM_CHNL-1:0]     chnl_rx;
wire   [C_NUM_CHNL-1:0]     chnl_rx_ack;
wire   [C_NUM_CHNL-1:0]             chnl_rx_last;
wire   [(C_NUM_CHNL*32)-1:0]          chnl_rx_len;
wire   [(C_NUM_CHNL*31)-1:0]          chnl_rx_off;
wire   [(C_NUM_CHNL*C_DATA_WIDTH)-1:0]      chnl_rx_data;
wire   [C_NUM_CHNL-1:0]             chnl_rx_data_valid;
wire   [C_NUM_CHNL-1:0]             chnl_rx_data_ren;
wire   [C_NUM_CHNL-1:0]             chnl_tx_clk;
wire   [C_NUM_CHNL-1:0]             chnl_tx;
wire   [C_NUM_CHNL-1:0]             chnl_tx_ack;
wire   [C_NUM_CHNL-1:0]             chnl_tx_last;
wire   [(C_NUM_CHNL*32)-1:0]          chnl_tx_len;
wire   [(C_NUM_CHNL*31)-1:0]          chnl_tx_off;
wire   [(C_NUM_CHNL*C_DATA_WIDTH)-1:0]      chnl_tx_data;
wire   [C_NUM_CHNL-1:0]             chnl_tx_data_valid;
wire   [C_NUM_CHNL-1:0]             chnl_tx_data_ren;

////////////////////////////////////////////////////////////
////  ASSIGN PROFILER OUTPUTS & ACCELERATOR INPUTS   //////
////////////////////////////////////////////////////////////
assign o_RX_CLK         = chnl_rx_clk;
assign o_RX_ACK[0]      = (PR_STATE == 2'd1);
assign o_RX_DATA_REN[0] = (PR_STATE == 2'd1) | ((PR_STATE == 2'd2) &
    chnl_rx_data_ren[0]);
assign o_TX_CLK         = chnl_tx_clk;
assign o_TX[0]          = (PR_STATE == 2'd3) | TAIL_FLAG[0]| chnl_tx[0];
assign o_TX_LAST[0]     = (PR_STATE == 2'd3) | chnl_tx_last[0];
assign o_TX_LEN[31:0]    = (PR_STATE == 2'd3) ? LEN    : (TAIL ? -1 :
    chnl_tx_len[31:0]);
assign o_TX_OFF[30:0]    = (PR_STATE == 2'd3) ? 31'd0 : chnl_tx_off[30:0];
assign o_TX_DATA[C_DATA_WIDTH-1:0]= (PR_STATE == 2'd3) ? TxDATA :
    (chnl_tx_data_valid[0] ? chnl_tx_data[C_DATA_WIDTH-1:0] :
    {TIMESTAMP,16'b0,CH_INFO[47:0]});
assign o_TX_DATA_VALID[0]= (PR_STATE == 2'd3) | TAIL_FLAG[0] |
    chnl_tx_data_valid[0];
assign user_clk         = clk;
assign riffa_reset      = rst;
assign chnl_rx[0]       = (PR_STATE == 2'd2) & RX0;
assign chnl_rx_last[0]  = chnl_rx_last_buff;
assign chnl_rx_len[31:0]= chnl_rx_len_buff;
assign chnl_rx_off[30:0]= chnl_rx_off_buff;
assign chnl_rx_data_valid[0]= (PR_STATE == 2'd2) & i_RX_DATA_VALID[0];
assign chnl_rx_data    = i_RX_DATA;
assign chnl_tx_ack     = i_TX_ACK;
assign chnl_tx_data_ren= i_TX_DATA_REN;
genvar q;
generate
  for (q = 1; q < C_NUM_CHNL; q = q + 1) begin : assign_chnls_1_12
    assign o_RX_ACK[q] = chnl_rx_ack[q];
        assign o_RX_DATA_REN[q] = chnl_rx_data_ren[q];
        assign o_TX[q] = TAIL_FLAG[q] | chnl_tx[q];
        assign o_TX_LAST[q] = chnl_tx_last[q];
        assign o_TX_LEN[32*(q+1)-1:32*q]= TAIL ? -1 :
    chnl_tx_len[32*(q+1)-1:32*q];
        assign o_TX_OFF[31*(q+1)-1:31*q]= chnl_tx_off[31*(q+1)-1:31*q];
        assign o_TX_DATA[C_DATA_WIDTH*(q+1)-1 : C_DATA_WIDTH*q] =
    chnl_tx_data_valid[q] ? chnl_tx_data[C_DATA_WIDTH*(q+1)-1 :
    C_DATA_WIDTH*q] : {TIMESTAMP,16'b0,CH_INFO[64*(q+1)-17:64*q]};
        assign o_TX_DATA_VALID[q]= TAIL_FLAG[q] | chnl_tx_data_valid[q];
        assign chnl_rx[q] = i_RX[q];
        assign chnl_rx_last[q] = i_RX_LAST[q];
        assign chnl_rx_len[32*(q+1)-1:32*q] = i_RX_LEN[32*(q+1)-1:32*q];
        assign chnl_rx_off[31*(q+1)-1:31*q] = i_RX_OFF[31*(q+1)-1:31*q];
        assign chnl_rx_data_valid[q] = i_RX_DATA_VALID[q];
    end

    case ({LOG_TRIGGERS,LOG_CHANNELS})
    2'b00: assign ID = 0;
    2'b01: assign ID = CH_ID;
    2'b10: assign ID = TR_ID;
    2'b11: assign ID = {TR_ID,CH_ID};
    endcase

    for (q = 0; q < C_NUM_CHNL; q = q + 1) begin : fix_tail
        always @(posedge clk)
            TAIL_FLAG[q] <= TAIL_FLAG[q] ? chnl_tx[q] | ~i_TX_DATA_REN[q]
    :  chnl_tx[q] & ~TX_DELAY[q] & TAIL;
    end

endgenerate

always @(posedge clk) {TX_DELAY,RX0}  <= {chnl_tx,i_RX[0]};

// FSM

always @(posedge clk)
if(rst) PR_STATE <= 0;
else
case (PR_STATE)
//IDLE
2'b00:
begin
    {FLUSH,DONE,OPCODE,WCOUNT} <= 0;
    if (i_RX[0] & ~TAIL_FLAG[0])
    begin
        chnl_rx_len_buff    <= i_RX_LEN[31:0] -(C_DATA_WIDTH/32);
        chnl_rx_last_buff   <= i_RX_LAST[0];
        chnl_rx_off_buff    <= i_RX_OFF[30:0];
        PR_STATE            <= 2'b01;
    end
end
//READ OPCODE
2'b01:
```

```
begin
    if (i_RX_DATA_VALID[0])
    begin
        OPCODE          <= i_RX_DATA[5:0];
    case (i_RX_DATA[1:0])
        2'b00: PR_STATE <= 2'b10;//USE ACCELERATOR
    2'b01://RETURN DURATION OR TIMESTAMP
        begin
            LEN             <= 32'd68;
            TxDATA          <=
    {SUM,MERGE_PULSES,MONITOR_CHANNELS,LOG_TRIGGERS,
    LOG_CHANNELS,TAIL,12'b0,TRIGGERS_NUM,12'b0,C_NUM_CHNL,
    BRAM_SIZE,ENTRIES,TIMESTAMP};
            PR_STATE        <= 2'b11;
        end
    2'b10://FLUSH LOG
        begin
            LEN         <= BRAM_SIZE;
            TxDATA      <= LOG;
            FLUSH       <= 1'b1;
            ADDRESS     <= ADDRESS + 1;
        end

    2'b11: //RESET TIMER-COUNTERS, SET TAIL (128-BIT EXTRA INFO AFTER EACH
    TRANSMITION)
        begin
            TAIL <= i_RX_DATA[6];
            PR_STATE <= 2'b00;
        end
    endcase
    end
    else if (FLUSH) {ADDRESS,PR_STATE}  <=  {ADDRESS + 1, 2'b11};
end
//ASSUME ACCELARATOR WORK COMPLETED AFTER TRANSMITION
2'b10:
    if      (TX_DELAY[0] & ~chnl_tx[0]) PR_STATE <= 2'b00;
//DATA TRANSMISSION
2'b11:
```

```
begin
    if (i_TX_DATA_REN[0])
    begin
        WCOUNT      <=  WCOUNT + (C_DATA_WIDTH/32);
        if (WCOUNT >= LEN-4 ) {PR_STATE,FLUSH,ADDRESS}   <= 'b0;
        else if (OPCODE == 2'b01)
        begin
            TxDATA[63:0]    <= CH_INFO>>(WCOUNT<<4);
            TxDATA[127:64]  <= TR_INFO>>(WCOUNT<<4);
        end
        else
        begin
            {TxDATA,FLUSH}  <= FLUSH ? {LOG,1'b1} : {TxDATA_BUFF, 1'b1};
            ADDRESS         <= ADDRESS + 1'b1;
        end
    end
    else if (FLUSH)     {FLUSH,TxDATA_BUFF}     <= {1'b0,LOG};
end

endcase

// INSTANTIATIONS
global_timer gt0(
    .clk(clk),
    .o_TIMESTAMP(TIMESTAMP),
    .rst(&OPCODE[3:0])
    );
generate
if ((LOG_CHANNELS & MONITOR_CHANNELS) | LOG_TRIGGERS)
begin
    event_log ev0(
        .clk(clk),
        .i_ID(ID),
        .i_ADDRESS(ADDRESS),
        .i_TIMESTAMP(TIMESTAMP),
        .o_LOG(LOG),
        .o_ENTRIES(ENTRIES[10:0]),
        .rst(&OPCODE[2:0])
    );
end
if (MONITOR_CHANNELS)
begin
    reg     [C_NUM_CHNL-1:0]    RX_DELAY        = {C_NUM_CHNL{1'b0}};
    reg     [C_NUM_CHNL-1:0]    CHNL_TRIGGER    = {C_NUM_CHNL{1'b0}};
    always @(posedge clk) RX_DELAY    <= chnl_rx;
    for (q = 0; q < C_NUM_CHNL; q = q + 1) begin : trigg
        always @(posedge clk)
            if      (chnl_rx[q] & !RX_DELAY[q])         CHNL_TRIGGER[q] <=
    1'b1;
            else if ((!chnl_tx[q] & TX_DELAY[q])|rst)   CHNL_TRIGGER[q] <=
    1'b0;
    end
    trigger_monitor #(C_NUM_CHNL, SUM, MERGE_PULSES) cm0(
        .clk(clk),
        .i_TRIGGER(CHNL_TRIGGER),
        .o_ID(CH_ID),
        .o_INFO(CH_INFO),
        .rst(OPCODE[0]&OPCODE[1]&OPCODE[5])
    );
end
if (TRIGGERS_NUM)
    begin
    trigger_monitor #(TRIGGERS_NUM, SUM ,MERGE_PULSES) tm0(
        .clk(clk),
        .i_TRIGGER(TRIGGER),
        .o_ID(TR_ID),
        .o_INFO(TR_INFO),
        .rst(OPCODE[0]&OPCODE[1]&OPCODE[4])
    );
end
endgenerate
////////////////////////////////////
// START USER CODE (do edit)
////////////////////////////////////

// CHANNEL TESTER EXAMPLE
```

```
genvar i;
generate
  for (i = 0; i < C_NUM_CHNL; i = i + 1) begin : profile_channels
    chnl_tester #(C_DATA_WIDTH) module1 (
      .CLK(user_clk),
      .RST(riffa_reset),  // riffa_reset includes riffa_endpoint resets
      // Rx interface
      .CHNL_RX_CLK(chnl_rx_clk[i]),
      .CHNL_RX(chnl_rx[i]),
      .CHNL_RX_ACK(chnl_rx_ack[i]),
      .CHNL_RX_LAST(chnl_rx_last[i]),
      .CHNL_RX_LEN(chnl_rx_len[32*i +:32]),
      .CHNL_RX_OFF(chnl_rx_off[31*i +:31]),
      .CHNL_RX_DATA(chnl_rx_data[C_DATA_WIDTH*i +:C_DATA_WIDTH]),
      .CHNL_RX_DATA_VALID(chnl_rx_data_valid[i]),
      .CHNL_RX_DATA_REN(chnl_rx_data_ren[i]),
      // Tx interface
      .CHNL_TX_CLK(chnl_tx_clk[i]),
      .CHNL_TX(chnl_tx[i]),
      .CHNL_TX_ACK(chnl_tx_ack[i]),
      .CHNL_TX_LAST(chnl_tx_last[i]),
      .CHNL_TX_LEN(chnl_tx_len[32*i +:32]),
      .CHNL_TX_OFF(chnl_tx_off[31*i +:31]),
      .CHNL_TX_DATA(chnl_tx_data[C_DATA_WIDTH*i +:C_DATA_WIDTH]),
      .CHNL_TX_DATA_VALID(chnl_tx_data_valid[i]),
      .CHNL_TX_DATA_REN(chnl_tx_data_ren[i])
    );
  end
endgenerate
////////////////////////////////////
// END USER CODE
////////////////////////////////////
endmodule
```

Global Timer

```verilog
`timescale 1ns/1ns

module global_timer(clk, o_TIMESTAMP, rst);

input                   clk;
output reg  [47:0]  o_TIMESTAMP   = {48{1'b0}};
input                   rst;

always @(posedge clk)
  if (rst)     o_TIMESTAMP     <= {48{1'b0}};
  else         o_TIMESTAMP     <= o_TIMESTAMP + 1'b1;

endmodule
```

Event Monitor

```verilog
`timescale 1ns / 1ps

module trigger_monitor(
    clk,
    i_TRIGGER,
    o_ID,
    o_INFO,
    rst
);

parameter       TRIGGERS_NUM  = 4'd1;
parameter       SUM           = 1'b0;
parameter       MERGE_PULSES  = 1'b0;

input                                               clk;
input       [TRIGGERS_NUM-1:0]                      i_TRIGGER;
output      [(2-MERGE_PULSES)*TRIGGERS_NUM-1:0]     o_ID;
output      [64*TRIGGERS_NUM-1:0]                   o_INFO;
input                                               rst;

```

```
reg         [16*TRIGGERS_NUM-1:0]   TR_COUNT;
reg         [48*TRIGGERS_NUM-1:0]   TR_DURATION;
reg         [TRIGGERS_NUM-1:0]      PRV         = {TRIGGERS_NUM{1'b0}};
wire        [TRIGGERS_NUM-1:0]      START;
wire        [TRIGGERS_NUM-1:0]      STOP;


always @(posedge clk)  PRV <= i_TRIGGER;

assign      START   =  i_TRIGGER & ~PRV;
assign      STOP    = ~i_TRIGGER &  PRV;

genvar i;
generate
for (i = 0; i < TRIGGERS_NUM; i = i + 1) begin : pg

  if (MERGE_PULSES)  assign o_ID[i]         =  STOP[i] | START[i] ;
  else               assign o_ID[2*i+1:2*i] = {STOP[i] , START[i]};

  assign o_INFO[64*(i+1)-1:64*i] =
    {TR_COUNT[16*(i+1)-1:16*i],TR_DURATION[48*(i+1)-1:48*i]};

  always @(posedge clk)
  begin
    if (rst)  {TR_COUNT[16*(i+1)-1:16*i],TR_DURATION[48*(i+1)-1:48*i]} <=
    0;
    else
    begin
      if(START[i]) TR_COUNT[16*(i+1)-1:16*i] <= TR_COUNT[16*(i+1)-1:16*i]
    + 1'b1;
      if(i_TRIGGER[i])
      begin
        if(START[i] & !SUM) TR_DURATION[48*(i+1)-1:48*i] <= 48'd1;
```

Event Log

```verilog
`timescale 1ns / 1ps

module event_log(
    clk,
    i_ID,
    i_ADDRESS,
    i_TIMESTAMP,
    o_LOG,
    o_ENTRIES,
    rst
    );

input               clk;
input   [15:0]      i_ID;
input   [9:0]       i_ADDRESS;
input   [47:0]      i_TIMESTAMP;
output  [127:0]     o_LOG;
output  [10:0]      o_ENTRIES;
input               rst;

reg     [10:0]      LOG_ADDRESS     = {11{1'b0}};
reg                 OVERFLOW        = 1'b0;
wire    [127:0]     LOG;

assign o_LOG     = (i_ADDRESS <= LOG_ADDRESS[10:1]) ? LOG : 0;
assign o_ENTRIES =  OVERFLOW ? {11{1'b1}} : LOG_ADDRESS;

always @(posedge clk)
  if       (rst)        LOG_ADDRESS <= {11{1'b0}};
  else if (|i_ID)       LOG_ADDRESS <= LOG_ADDRESS + 1'b1;

always @(posedge clk)
  if       (rst)                                 OVERFLOW <= 1'b0;
  else if ((LOG_ADDRESS=={11{1'b1}})& (|i_ID))   OVERFLOW <= 1'b1;

//  SIMPLE DUAL PORT RAM
//  128 * 1024
blk_mem_gen_0 log(
        .clka(clk),
        .wea(|i_ID),
        .addra(LOG_ADDRESS),
        .dina({i_ID,i_TIMESTAMP}),
        .clkb(clk),
        .addrb(OVERFLOW ? i_ADDRESS + LOG_ADDRESS :i_ADDRESS),
        .doutb(LOG)
   );

endmodule
```

Instantiation

```
profiler #(C_DATA_WIDTH , C_NUM_CHNL) riffa_profiler (
    .clk(user_clk),
    .o_RX_CLK(chnl_rx_clk),
    .i_RX(chnl_rx),
    .o_RX_ACK(chnl_rx_ack),
    .i_RX_LAST(chnl_rx_last),
    .i_RX_LEN(chnl_rx_len),
    .i_RX_OFF(chnl_rx_off),
    .i_RX_DATA(chnl_rx_data),
    .i_RX_DATA_VALID(chnl_rx_data_valid),
    .o_RX_DATA_REN(chnl_rx_data_ren),
    .o_TX_CLK(chnl_tx_clk),
    .o_TX(chnl_tx),
    .i_TX_ACK(chnl_tx_ack),
    .o_TX_LAST(chnl_tx_last),
    .o_TX_LEN(chnl_tx_len),
    .o_TX_OFF(chnl_tx_off),
    .o_TX_DATA(chnl_tx_data),
    .o_TX_DATA_VALID(chnl_tx_data_valid),
    .i_TX_DATA_REN(chnl_tx_data_ren),
    .rst(riffa_reset)
);
```

Appendix B

Software interface - RIFFA Monitor API

```c
#define TIMEOUT 8000
#define BRAM_SIZE 2048


int PR_fpga_send(fpga_t * fpga, int chnl, void * data,
        int len, int destoff, int last,
        long long timeout){
  // If data are sent to channel 0 profiler will use the first 128 bit.
  // To avoid that extra 128'b0 is added as header.
  // Profiler will read OPCODE == 0 and forward data without header to
    accelarator.

  if (chnl) return fpga_send(fpga, chnl, data, len, destoff, last,
    timeout);
  int i, buffer[len+4];
  for(i=0; i<len; i++) buffer[i+4] = ((int *)data)[i];
  buffer[0]= 0;
  return fpga_send(fpga, chnl, buffer, len+4, destoff, last, timeout);
}

int *PR_PRINT_BINARY(size_t const size, void const * const ptr, int print)
  // Helper function. Used for proper display of TRIGGERS
```

```
{
    unsigned char *b = (unsigned char*) ptr;
    unsigned char byte;
    int i, j;

    int *bits   = (int *)calloc(1000,1);
    for (i=size-1;i>=0;i--)
    {
        for (j=7;j>=0;j--)
        {
            byte = b[i] & (1<<j);
            byte >>= j;
            if (print) printf("%u", byte);
      bits[i*8+j] = (int)byte;
        }
    }
    return bits;
}

unsigned long PR_INFO(fpga_t * fpga, int print){
// Returns current Timestamp
// Prints parameter and counter values. (Populate struct incoming)
// Timestamp * clk_period = time since last profiler_reset (or bitstream
    assignment)

  int i, *bits;
  unsigned long buff[34];
  unsigned short entries, bram_s, n_chnl, n_trigg,
           tail, log_chnl, log_trigg, mon_chnl, merge, sum;
  unsigned short * sbuff = (unsigned short *)buff;
  buff[0]=1;

  fpga_send(fpga, 0, &buff, 1, 0, 1, TIMEOUT);
  fpga_recv(fpga, 0, &buff, 68, TIMEOUT);

  bits = PR_PRINT_BINARY(sizeof(short),sbuff+7 ,0);
  entries   = sbuff[3];
  bram_s    = sbuff[4];
  n_chnl     = sbuff[5];
  n_trigg    = sbuff[6];
  sbuff[3]   = 0;

  if(print){
    printf("SUM_DURATION : %d\n", bits[5]);
    printf("MERGE_PULSES : %d\n", bits[4]);
    printf("MONITOR_CHNL : %d\n", bits[3]);
    printf("LOG_TRIGGERS : %d\n", bits[2]);
    printf("LOG_CHANNELS : %d\n", bits[1]);
    printf("TAIL         : %d\n", bits[0]);
    printf("TRIGGER_NUM  : %d\n", n_trigg);
    printf("CHANNEL_NUM  : %d\n", n_chnl);
    printf("BRAM_SIZE    : %d words\n", bram_s);
    printf("ENTRIES      : %d\n", entries);
    printf("TIMESTAMP    : %ld\n", buff[0]);
    for (i=0; i<n_chnl; i++){
      printf("chnl %d calls : %d\n", i, sbuff[11+8*i]);
      sbuff[11+8*i]=0;
      printf("chnl %d dur   : %ld\n", i, buff[2+2*i]);
    }
    for (i=0; i<n_trigg; i++){
      printf("chnl%d calls  : %d\n", i, sbuff[15+8*i]);
      sbuff[15+8*i]=0;
      printf("chnl%d dur    : %ld\n", i, buff[3+2*i]);
    }
  }
  return buff[0];
}

void PR_LOG(fpga_t * fpga, unsigned short **triggers,
      long **timestamps, int print, int file,
      int timeline){
  // Returns an array with all valid entries in BRAM log.
  // Each entry is 64bit (16bit ID , 48bit Timestamp)

  int i, w, finished, active ,start, stop;
  int buffer  = 2;
```

```c
char ch;
long max, step;
unsigned short *ts_buff;

*triggers     = (short *)calloc(BRAM_SIZE,2);
*timestamps   = (long *)calloc(BRAM_SIZE,8);
ts_buff     = (short *)(*timestamps);

fpga_send(fpga, 0, &buffer, 1, 0, 1, TIMEOUT);
printf("rcvd %d\n",fpga_recv(fpga, 0, ts_buff, BRAM_SIZE*2, TIMEOUT));

if (file){
  FILE *raw_log, *log;
  raw_log = fopen("RAW_LOG.txt", "w");
  log = fopen("LOG.txt", "w");

  for (i=0; i<BRAM_SIZE; i++){
    fprintf(raw_log,"%ld\n",(*timestamps)[i]);
    (*triggers)[i]     = ts_buff[4*i+3];
    ts_buff[4*i+3]     = 0;
    fprintf(log,"%d\t%ld\n",(*triggers)[i], (*timestamps)[i]);
  }

  fclose(raw_log);
  fclose(log);
}

if (print){
  printf("\n    TRIGGER\t\t|       TIMESTAMP
  \t\t|\n----------------------------------------------------------\n");
  i=0;
  while((*triggers)[i] != 0) {
    PR_PRINT_BINARY(sizeof((*triggers)[i]), (*triggers)+i, 1);
    printf("\t|  %20ld \t|\n\n",(*timestamps)[i]);
    i++;
  }
}

```

```c

  if (timeline){
    max      = PR_INFO(fpga,0);
    step     = max / timeline;
    FILE *out    = fopen("TIMELINE.txt", "w");
    start      = 0;
    stop     = 1;
    int *bits = NULL;
    for (w=0; w<8; w++){
      finished  = 0;
      active    = 0;
      ch    = '.';
      fprintf(out,"%d: ",w);
      for (i=0; i<=timeline; i++){
        while ((*timestamps)[active] < i*step) {
          if ((*timestamps)[active] == 0) break;
          bits = PR_PRINT_BINARY(sizeof(short), (*triggers)+ active ,0);
          if (bits[start] == 1) {
            ch = '|';
            finished = 0;
          }
          if (bits[stop] == 1) finished = 1;
          active++;
        }
        fprintf(out,"%c",ch);
        if (finished) ch = '.';
      }
      fprintf(out,"\n");
      start = start + 2;
      stop  = stop  + 2;
    }
    fclose(out);
  }
}

void PR_SET(fpga_t * fpga, int TAIL, int RST_TRG_CNTS,
      int RST_CHNL_CNTS, int RST_LOG, int RST_GLTIMER,
      int print){
```

```
  // RESET TIMER-COUNTERS, SET TAIL (128-BIT EXTRA INFO AFTER EACH TRANSMITION)

  int buffer = 3 + (TAIL!=0)*64 + (RST_TRG_CNTS!=0)*32 + (RST_CHNL_CNTS!=0)*16 + (RST_LOG||RST_GLTIMER)*8 + (RST_GLTIMER!=0)*4;
  if (fpga_send(fpga, 0, &buffer, 1, 0, 1, TIMEOUT)>0) {
    if(print){
      if (RST_GLTIMER ) printf("Global Timer Reseted.\n");
      if (RST_LOG||RST_GLTIMER ) printf("Log Reseted\n");
      if (RST_CHNL_CNTS) printf("\nChannel Counters Reseted\nTrigger Counters Reseted\n");
      if (RST_TRG_CNTS) printf("\nChannel Counters Reseted\nTrigger Counters Reseted\n");
      if (TAIL) printf("\nTAIL ON\n\n");
      else printf("\nTAIL OFF\n\n");
    }
  }
  else  printf("FPGA_SENT ERROR\n");
}
```

BIBLIOGRAPHY

(1) **Field-programmable gate array - Wikipedia, the free encyclopedia**
http://en.wikipedia.org/wiki/Field-programmable_gate_array

(2) **FPGA Architectures: An Overview**
http://www.springer.com/cda/content/document/cda_downloaddocument/9781461435938-c2.pdf?SGWID=0-0-45-1333135-p174308376

(3) **Xilinx Virtex-7 FPGA VC707 Evaluation Kit**
http://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug848-VC707-getting-started-guide.pdf

(4) **Virtex 7 Series FPGAs Overview**
http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

(5) **VC707 Evaluation Board User Guide**
http://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf

(6) **RIFFA: A Reusable Integration Framework For FPGA Accelerators**
http://riffa.ucsd.edu/

(7) **Jacobsen, M. and Kastner, R. "RIFFA 2.0: A reusable integration framework for FPGA accelerators."**
https://sites.google.com/a/eng.ucsd.edu/matt-jacobsen/fccm_final.pdf?attredirects=0&d=1

(8) **Vivado Design Suite**
http://www.xilinx.com/products/design-tools/vivado.html

(9) **Get Smart About Reset:Think Local, Not Global**
http://www.xilinx.com/support/documentation/white_papers/wp272.pdf

(10) **Gprof**
https://en.wikipedia.org/wiki/Gprof