Prepared for submission to JINST

Accessing Register Spaces in FPGAs within the ATLAS DAQ Scheme via the SCA eXtension

C. Bakalis[a,b,1] **T. Alexopoulos**[a] **I. Grayzman**[c] **L. Lee Jr.**[d] **L. Levinson**[c] **M. Perganti**[a] **V. Polychronakos**[b]

[a] *National Technical University of Athens, Athens 15773, Greece*
[b] *Brookhaven National Laboratory, Upton NY 11973, USA*
[c] *Weizmann Institute of Science, Rehovot 7610001, Israel*
[d] *Harvard University, Cambridge MA 02138, USA*

E-mail: cbakalis@lbl.gov

Abstract: The foreseen upgrades of the Large Hadron Collider (LHC) are expected to increase the required throughput of the front-end and back-end electronics that support the readout of the LHC detectors. Therefore, the complexity of the electronics systems will be increased as well. An example of this is the electronics system of the New Small Wheel (NSW) upgrade of the ATLAS detector, which will be comprised of a number of Field-Programmable Gate Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs). These ASICs will be configured and monitored by the Slow Control Adapter (SCA), another ASIC designed for this purpose. The Slow Control Adapter eXtension (SCAX) on the other hand, is an FPGA module designed to support FPGA systems that are part of the ATLAS electronics scheme by reading and writing their configuration parameters and status indicators. SCAX emulates both the I^2C interface of the SCA used to access the NSW ASICs, as well as the communication protocol implemented between the SCA and the back-end infrastructure. It thereby enables using the same OPC-UA server and back-end software suite that support the ASICs, to also interface with the FPGAs that are part of the same system. This work describes the context of the SCAX's implementation, alongside architectural considerations of the module, features, and techniques to validate its hardware implementation across a variety of FPGA devices.

Keywords: Data acquisition concepts, Detector control systems, Digital electronic circuits

[1] Corresponding author. Currently with Lawrence Berkeley National Laboratory, Berkeley CA 94720, USA.

Contents

1 Introduction

The New Small Wheel (NSW) [1] upgrade of the ATLAS [2] detector employs a novel Data AcQuisition (DAQ) scheme. Its core is the Front-End Link eXchange (FELIX) [3, 4], a general-purpose data routing device comprised of optical links and an FPGA on a PCIe card hosted by a Linux server. Situated outside the radioactive experimental area, FELIX interfaces with the front-end nodes via optical links, each operating at a data rate of 4.8 Gb/s. These links propagate slow control data alongside trigger signals to the detector electronics, and forward the collision-related hit data to FELIX, which in turn receives and sends the data from and to a commodity network via 25 or 100 Gb/s Ethernet. FELIX will eventually be used by all detector subsystems of ATLAS. The slow control data mediator between FELIX and the front-end devices under configuration or monitoring is the Slow Control Adapter (SCA) [5]. The SCA is a radiation-tolerant ASIC developed at CERN, that interfaces with the back-end slow control software - a dedicated Open Platform Communications - Unified Automation (OPC UA) server [6], via FELIX. The SCA in turn uses several protocols, such as I^2C , to communicate with other front-end ASICs.

The NSW electronics system however (and any ATLAS DAQ system that may utilize the SCA in the future), also includes FPGAs which are usually situated outside the experimental cavern, or in low radiation areas of the detector. The logic in these FPGAs has several configuration parameters that need to be set during run preparation, similarly to the ASICs. To reduce the complexity of configuration software, a uniform scheme for configuring both the ASICs and the FPGAs was desired. Given the available tools that have been described so far, in order to access these parameters, the user would have to deploy an SCA on the same printed circuit board that

bears these FPGAs, connect it to the OPC server, via the back-end interface in the FPGA, and then access the registers in the FPGA via an I^2C interface in its logic that communicates with the SCA. However, this would increase the FPGA board's complexity and cost. Therefore, the approach of deploying an FPGA emulated version of the SCA is followed instead. The Slow Control Adapter eXtension (SCAX) is a custom module deployed within the FPGA itself [7]. Designed purely in VHDL, this FPGA component interfaces with FELIX in the same fashion as the SCA, making it transparent to the OPC server. It is designed to access the User FPGA Logic (UFL) registers of any FPGA that hosts it via sixteen user-defined *Register Files*, which interface with the SCAX's core logic through sixteen distinct *I^2C Channels*. This interface mimics that of the SCA's, which has the same number of I^2C sub-devices, thus further enhancing the transparency to the OPC server[1]. This allows the OPC software suite to access both front-end ASIC and FPGA address spaces within the context of any system that deploys both SCAs and FPGAs. Also, this approach leverages the already-existing physical links between devices, as opposed to other well-established configuration schemes. An example is the IPbus [8], which requires a User Datagram Protocol (UDP) connection between the FPGA and the back-end, alongside dedicated software.

2 Architecture

The SCAX may be used in any FPGA implementation of the ATLAS DAQ electronics that interfaces with FELIX. Therefore, flexibility and adaptability are considered to be of utmost importance. Given that, several design considerations were taken into account in order to facilitate the module's deployment in a pre-existing design. Figure 1 provides an overview block diagram of the SCAX's architecture.

All inbound frames are received by the *Elink2FIFO* block, which decodes and buffers the incoming 8b10b encoded data at a maximum rate of 80, 160 or 320 Mb/s. This module, alongside the *FIFO2Elink* block, originates from the interface logic of the FELIX firmware. The received data are presented to the *Deframer* which registers the frame, checks its integrity via the Frame Check Sequence checker (see *FCS_CHK* in Figure 1). The frame's fields are then forwarded to the *Traffic Handler*; the core logic of the design, with its main purposes being the routing of the inbound frame fields to the corresponding sub-module and reply bus arbitration. For example, one logic block that may receive a frame is the *Controller*, a module loosely based on the corresponding SCA component, but accommodated to the SCAX's needs. Most importantly, all traffic related to the configuration or querying of the UFL register contents are directed to the *I^2C Router*, which in turn determines the *I^2C Channel* that will finally receive the data. Whichever the active sub-module may be, the *Traffic Handler* stands by for the active logic to finish processing and generate a reply, which via the *FIFO2Elink* block will be forwarded to FELIX and to the back-end software[2]. This pipelined design approach allows for operation of the SCAX at relatively high core clock frequencies, which is a highly desirable feature.

[1] Note however that the interface to the FPGA registers themselves is direct, not via an I^2C serial bus.

[2] The logic generates replies with latencies ranging from 700 ns to 2 µs, depending on the operation.

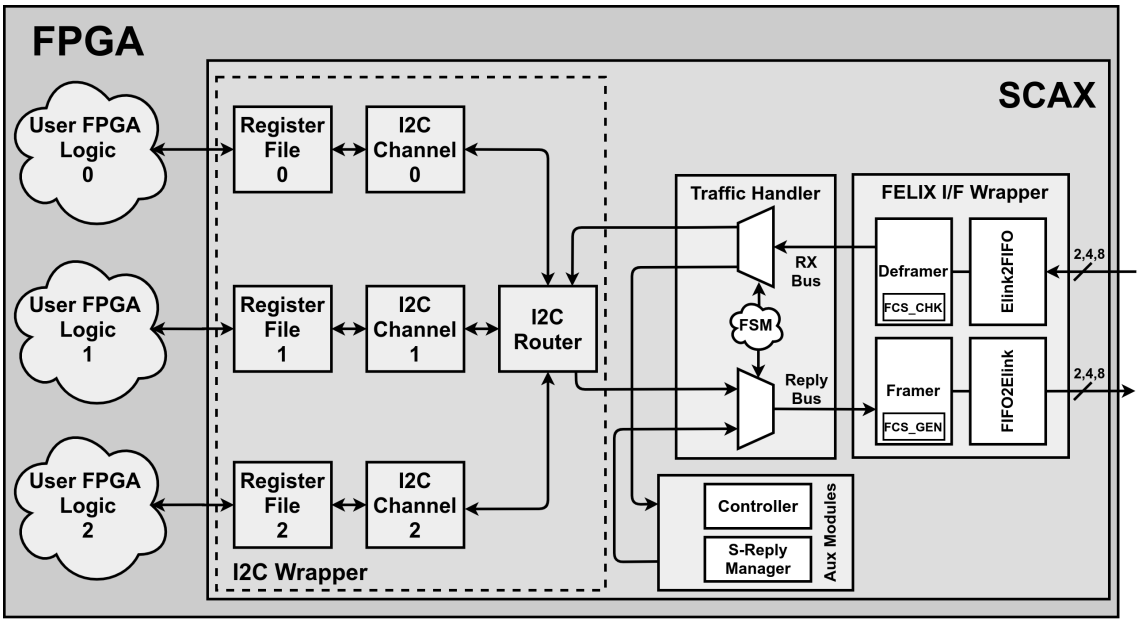

Figure 1. SCAX architecture block diagram. The sub-modules that implement the interface with the back-end can be seen on the right, while the I^{2}C logic alongside the register files that are part of the connection with the surrounding logic, are on the left-hand side of the diagram. The *Traffic Handler* manages the inbound and outbound frame data-flow, and two auxiliary logic components deal with all non-I^{2}C related types of transactions. Note that only three *I^{2}C Register File-Channel* pairs are depicted here, but the SCAX supports up to sixteen independent instances, each connected to a specific user logic component.

2.1 The Register File and its Corresponding Channel

As mentioned above, the SCAX accesses the UFL registers via sixteen possible *I^{2}C Channels*. The back-end software provides a register address and register contents (for write commands) that will eventually reach the corresponding *I^{2}C Channel*, which in turn will interface with the UFL via its associated *Register File*. Even though this operation resembles a standard I^{2}C transaction that is performed by the SCA from the back-end software's perspective, the SCAX does not internally operate an actual I^{2}C bus. The *I^{2}C Channel* receives an address and register contents pair from the software, and uses these to drive the *Register File* when a write operation is desired. For this mode, the Register File essentially acts as a demultiplexer, that switches to the UFL register bound to that address, and allows the associated *I^{2}C Channel* to write the desired value into it. For read operations, the *I^{2}C Channel* receives an address-to-be-read and uses the *Register File*, which for read operations acts as a multiplexer, to access the corresponding register, sample its contents, and forward them to the software. Each *Register File* supports interfacing with up to 1024 32-bit[3] registers. Figure 2 can be used as a reference to the actual interfacing logic.

Although the logic of the *Register File* is simple in nature, its source file changes from implementation to implementation, and may also get quite complex, especially when a significant number of registers are to be accessed. In order to address this, the ability to auto-generate a *Register File* is also provided to potential users of the SCAX (see Section 3).

[3] 32 bits is the maximum allowable register width, but it can also be less than that.

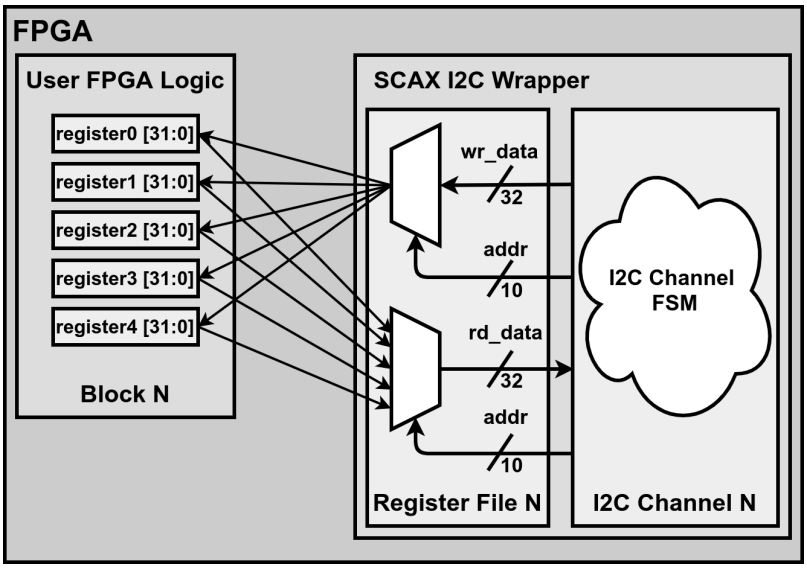

Figure 2. The *Register File* allows the *I²C Channel* to access the desired register by using the register address to switch its multiplexer/demultiplexer. The sub-module allows for interfacing with up to 1024 32-bit registers, while the data are presented to the registers in parallel. The scheme implies that each *Register File* is associated with a specific part of the FPGA's logic.

2.2 RAM and FIFO Extension

In addition to providing the ability to access conventional registers implemented in the FPGA's fabric, the SCAX allows the user to interface with memory elements as well. For instance, the user may associate the write and status ports of a First-In-First-Out (FIFO) instance with some addresses of a *Register File*, thus granting the user writing and reading capabilities into/from the FIFO. Apart from interfacing with a FIFO, the firmware package also provides the means to interface with the FPGA's Random-Access-Memory (RAM) primitives. This can be done via an add-on logic, named *SCAX Memory Controller* (SMC), that connects with both the *Register File* and the memory element in question. The user provides a RAM port address via the *Register File*, and the extension module auto-increments the aforementioned address for each read or write command. This allows the user to access a wide range of the primitive's address space while keeping the amount of transactions with the back-end as low as possible. These two extra features of the SCAX add to its functionalities, but also increase the complexity of the *Register File*, as it should support the interfacing with memory primitives in addition to the FPGA registers. This is one of the other reasons why the concept of the automated *Register File* generation was introduced (see Section 3).

2.3 Clock Domain Crossing Considerations

As mentioned above, the SCAX core clock can have a frequency of up to 320 MHz. This places the module's logic in a single clock domain. However, since a user might also wish to interface with registers situated outside this domain, Clock Domain Crossing (CDC) capabilities were considered when designing the SCAX in order to address this. Each *I²C Channel - Register File* pair may be manually switched by the user to "CDC Mode" prior to deploying the SCAX into their design. This overrides the default clocking of these two modules (which is the SCAX core clock), with the clock

that drives the UFL registers to which the pair corresponds. In essence, this option instantiates two CDC FIFOs in the associated I^{2}C Channel, one for read and one for write transactions. These FIFOs operate in the UFL register clock and the SCAX core clock domains. Apart from that, the CDC flavor of the *I^{2}C Channel* operates in a similar fashion as the original one, with the only difference being that the information is always passed through the corresponding FIFO to ensure data integrity and timing closure.

3 Automated Register File Implementation

As mentioned above, the SCAX design includes a high-level user interface for the definition and implementation of application-specific registers. This system provides developers with the ability to provide a database of desired registers in a CSV form. Entries in this database define properties of the registers – their bit width, read and write ability, multiplicity, and whether they correspond to RAM or FIFO primitives.

Upon update of these CSV databases, a continuous integration (CI) pipeline triggers the construction of automated VHDL implementations of both the package as well as the register interface itself. Additional code is assembled to define XML configurations for the OPC UA server, high-level C++ structures for interfacing with the associated OPC UA client, and LaTeX tables of the register database for automated documentation.

Internally, the registers defined in the CSV are used to build a YAML database. Additional registers are created to support interaction with RAM and FIFO registers, and a dedicated handling of trigger registers is implemented. Individual output formats built from this database are defined in templates using the Jinja2 [9] template engine, passed to the `wuppercodegen` package provided by the FELIX project [10].

The automation in the code generation builds an output for every template and for every CSV database in the repository. In this way, prototyping new register configurations is simple and requires only high-level interactions from the users, and creating additional software interfaces is easily done by adding additional Jinja2 templates to the repository.

4 Development Process

During the development process of the SCAX, several key features had to be tested, in order to ensure its compliance with the requirements that had been set. First of all, the back-end interface with FELIX and the OPC server was verified both at the behavioral simulation level, and in silicon. In addition, since FELIX is foreseen to be used as the universal DAQ agent of the ATLAS experiment, and given the fact that any FPGA that interfaces with FELIX can have its registers accessed by the SCAX, it was considered imperative to ensure that the SCAX can be deployed seamlessly in a variety of FPGA devices. Finally, the SCAX's ability to access UFL registers without any errors was assessed via stress-tests in the actual implementation, which validated both the module's core functionality, and the timing constraints that are introduced later in this Section.

4.1 Pre-Implementation Verification of the OPC Interface

In order to facilitate the SCAX's development and debugging process, a VHDL-based behavioral simulation testbench was used to test both the module's interface with the OPC server, as well as its UFL register connections. Naturally, the same testbench can be used to ease further developments and debugging in the future. Before describing the simulation procedure, a brief description of the back-end interface logic is provided.

During the connection initialization stage of the OPC server with an SCA node, the software suite addresses several of the ASIC's internal registers via FELIX. Also, a subset of these registers are continuously being accessed by the server during runtime, for connection stability checking purposes. The SCA communication protocol dictates that for every query from the supervising software, the addressed node should transmit an associated reply. That is, the SCA produces an outbound packet that indicates which request it is associated with, and whether that request was successful or not. If applicable, it can also yield more information, such as a register's contents. Consequently, the SCAX mimics this behavior, by forming reply packets the same way as the ASIC does, and by implementing the known set of internal status registers that are routinely queried by the back-end software upon initialization of the communication, and during runtime.

For the purpose of validating the aforementioned scheme, the OPC's transactions with the SCA ASIC were recorded using the associated tools provided by FELIX's low-level software infrastructure. The acquired data were used to generate a list of commands and valid responses that was parsed into a dummy OPC server VHDL module, instantiated within the behavioral testbench of the SCAX. The simulation would then assess the responses of the SCAX to the dummy server's queries, and deem if the transactions resemble those between the actual server and the ASIC. Through the same simulation environment, the basic functionalities of the SCAX were also tested, such as accessing of UFL registers.

4.2 Post-Implementation Verification of the OPC Interface

The OPC communication validation was extended to the in-silicon phase of the development as well. Two buffers, one for the inbound, and one for the outbound direction of the back-end interface were deployed, storing the received and generated packets. An Integrated Logic Analyzer (ILA) was then used to inspect the memory element contents. Any disagreement between the data recorded by the SCAX debug buffers and FELIX's traffic inspection tools would indicate an error in the physical layer of the link between the two. Any inability of the server to establish a connection with the SCAX would point to an error in SCAX's logic, or even in its testbench. The buffer-recorded packets, in conjunction with the testbench implementation, aid to the tackling of these issues as well.

4.3 Timing Constraints Considerations

In addition to validating the back-end interface, the in-silicon implementation testing was also used to study the timing closure of the design. First of all, since the first FPGA that will make use of the SCAX is the one hosting the NSW Trigger Processor (TP), the first tests were conducted in a similar device to that of its first host[4]. The Xilinx® XC7VX690T-2FFG1761C FPGA was chosen

[4]The TP's FPGA is the Xilinx® XC7VX690T-2FFG1158C.

for the tests, which was provided by a Xilinx® VC709 evaluation board. The initial studies involved deploying the SCAX in similar register-occupancy conditions as the final NSW TP implementation, but apart from that, different scenarios were explored, where the *Register File* size, the device itself, or even the *Register File - I^2C Channel* address width, would vary. As it will become evident, the introduction of Multi-Cycle Paths (MCPs) across the critical endpoints in the design was mandatory, in order to solidify the module's ability to adapt to different use-cases.

Placement Considerations

In a hypothetical situation where the designers of the FPGA that hosts the SCAX had previously decided to separate different parts of their logic via FPGA floorplanning, the deployment of the SCAX could potentially create timing issues, since each group of the pre-existing FPGA logic would have to be connected to a specific *Register File - I^2C Channel* pair. The same effect could also be caused by the place-and-route procedure of the tool itself, as the automatic physical separation of loosely-related logic primitives is not uncommon, especially if these belong to different clock domains. In order to tackle this, some freedom of movement to the *Register File - I^2C Channel* pair was given, by introducing MCPs on multi-bit buses and pipelines for single-bit control signals on all paths between each *I^2C Channel* and the *I^2C Router*.

Register File Size and Address Width

The NSW TP's register address space is relatively small, when compared to the maximum allowable depth of the *Register File*. It is foreseen that the NSW TP will deploy around five *Register Files* in order to interface with about fifty UFL registers of various widths. However, since the SCAX can be deployed to a design that may require accessing to a larger amount of parameters, and by the time that a fully-implemented *Register File* allows for reading and writing from and into 1024 32-bit registers, this extreme use-case had to be examined as well. Since the *Register File* is a module solely comprised of the FPGA's combinatorial logic elements, as the user adds interfacing with more UFL registers to a *Register File*, the complexity of the combinatorial logic will increase.

Studies showed that in a Xilinx® XC7VX690T-2FFG1158C where the SCAX was running at a core clock of frequency 320 MHz[5], a *Register File* that accessed 256 32-bit registers would cause timing to fail. The amount of negative slack and combinatorial resource utilization would increase with the further addition of registers, which made evident that a *Register File*-related MCP was needed, in order to ease timing closure on the associated nets. Also, in the expected situation where the SCAX would be have to be deployed in an already densely-populated logic, this MCP would affect timing closure in a positive manner.

Finally, the introduction of the Multi-Cycle Path would allow to increase the 10-bit address depth of the *Register File*, in any future implementations. The choice to implement a 10-bit address space depth in each *Register File* stems from the fact that this addressing mode is already being used by the back-end software to interface with the SCA. However, since the ASIC and the software support other modes of I^2C transactions as well, it would be worth to investigate potential modifications in the way the SCAX accesses the UFL registers via its *Register File*, namely by changing its address space depth. For this reason, the possibility to implement a *Register File* with

[5]This is the core clock frequency of the SCAX for the NSW TP.

different address bus widths was explored. The testing procedure involved implementing a *Register File* with address bus widths from 8 to 12 bits, and having it interface with as many 32-bit registers as its depth allowed, in a Xilinx® XC7VX690T-2FFG1761C FPGA. The Look-Up Table (LUT) primitive utilization and amount of negative slack for each address space depth can be inspected in Figure 3. Note that for this testing, no MCP was used, as the effect on the negative slack with respect to the increase in the address bus width had to be studied. Note that all widths cause timing failures, thus further emphasizing the need for an MCP introduction to the *Register File* path.

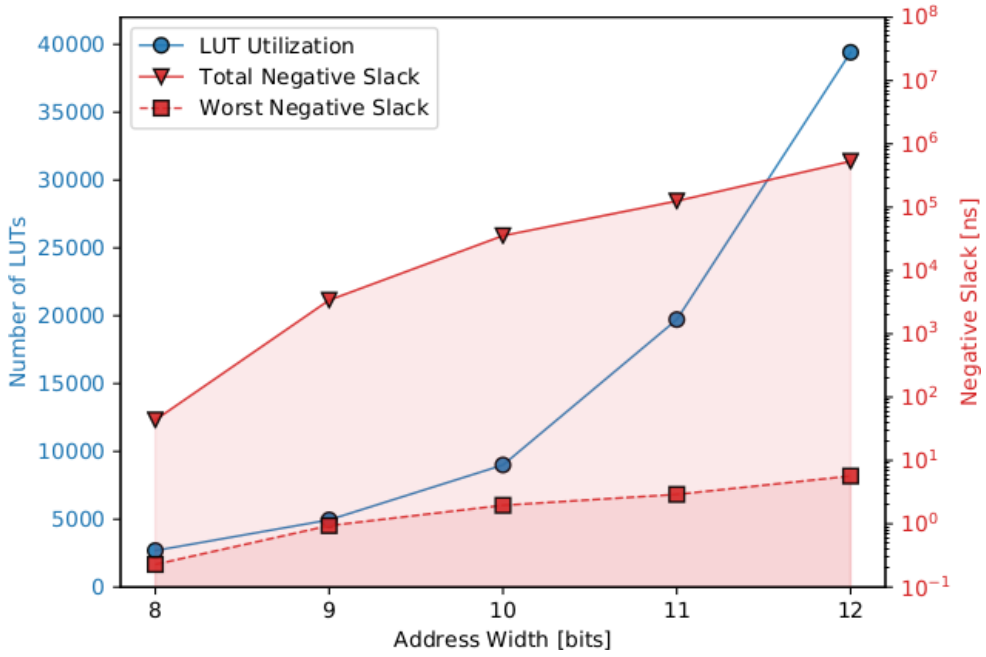

Figure 3. Fully-implemented *Register File* resource utilization and design timing failure quantification for several address bus widths, with a frequency of 320 MHz, in a Xilinx® XC7VX690T-2FFG1761C FPGA. Note that for a 12-bit address *Register File*, the combinatorial resource utilization is about 8 %. For the native 10-bit addressing mode, the *Register File* logic alone utilizes about 10000 LUTs, which still corresponds to 3 % of the resources of the aforementioned package.

The design's default *Register File* MCP length that was finally introduced[6], allows for a *Register File* that is connected with 1024 32-bit (native 10-bit address size) registers under an SCAX core clock of 320 MHz to be deployed successfully in the resourceful Xilinx® XC7VX690T-2FFG1158C. Table 1 lists the implementation results of five Xilinx® FPGA devices, in which the SCAX was successfully deployed under the same conditions, with no timing errors. Note that the SCAX core logic uses a modest 6600 sequential and 3500 combinatorial elements, which correspond to 0.76 % and 0.5 % of the available resources of a Xilinx® XC7VX690T-2FFG1761C FPGA respectively. In Table 1, the majority of the combinatorial logic is used by the *Register File*, due to its size.

[6]The said MCP length is easily configurable by the user, thus further facilitating the accommodation of the configuration block into their design and device.

Table 1. Total SCAX LUT utilization in various Xilinx® FPGA devices, where one fully-populated *Register File* was deployed at a core clock speed of 320 MHz.

Device	Available LUTs	LUTs Utilized	LUT Utilization (%)
XC7A100T-3FGG676	63400	12907	20%
XC7K160T-2FBG484	101400	12903	13%
XC7A200T-3FBG484	134600	12925	9%
XC7VX330T-1FFG1157	204000	12904	6%
XC7VX690T-2FFG1761C	433200	12905	3%

Stress-Testing

In order to adequately test the SCAX's functionalities, an extreme situation was chosen, where the module was deployed in a Xilinx® XC7VX690T-2FFG1761C FPGA. The SCAX had two active *I²C Channels*, connected with two fully-implemented *Register Files* interfacing with 1024 32-bit registers each. The two *I²C Channels* were physically separated to emulate the scenario described earlier in the current Subsection. The layout of the implemented design can be viewed in Figure 4.

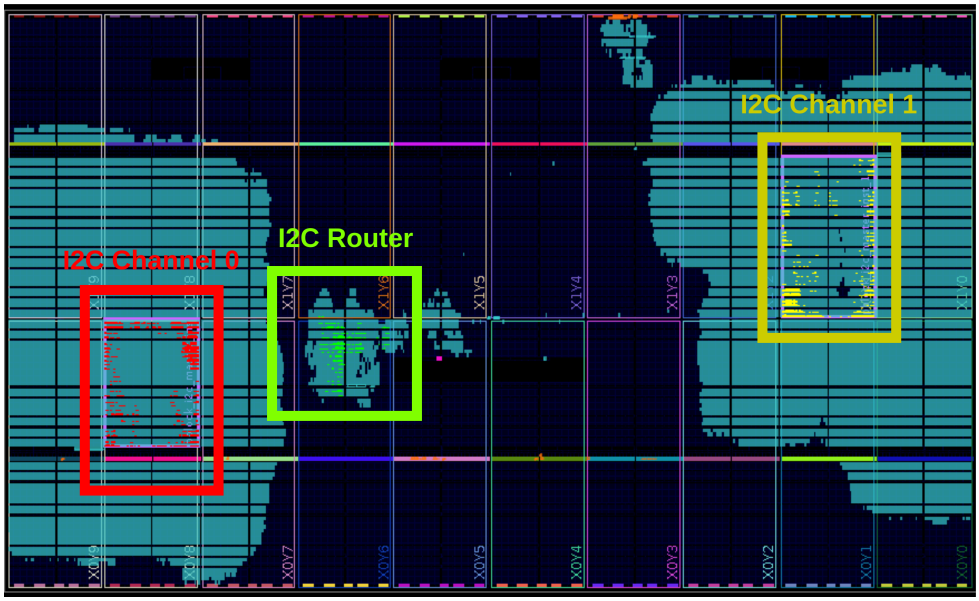

Figure 4. SCAX implementation in the FPGA of a Xilinx® VC709. The *I²C Router* is highlighted in the middle of the Figure, whereas the two *I²C Channels* are in two different areas of the die, to emulate a physical separation that may have been imposed by the designer of the pre-existing logic, or by the implementation tool. Most of the active cells around the two *I²C Channels* correspond to the combinatorial logic of the associated *Register File* block and the UFL registers themselves.

Two tests were performed to verify the hardware implementation: first, an OPC transaction stress-test was used to test the back-end interfacing. During this, 400 million frames were sent to the SCAX at a rate of 11 kreq/s. The server evaluated all SCAX reply frames, and no errors were found. Also, in order to test the MCPs between the *I²C Router* and the two *I²C Channels*, as well as the MCPs between each *I²C Channel* and the UFL via the combinatorial-logic-heavy *Register File*,

a mass read/write test from/into the registers was used. During this procedure, all 1024 registers of a given *Register File* were written into, in random order. This randomization was imperative, as the random switching of the address bus of the module's multiplexer/demultiplexer (see Figure 2) ensured the path would be verified correctly. After writing into all registers, their values were read back, again in nondeterministic order, and were checked offline by a software routine for integrity. The tests showed no errors over several million transactions.

5 Conclusions

The SCAX FPGA module for configuration and monitoring presented here provides easy user access to FPGA registers via FELIX and the SCA OPC server/client software suite. A flexible design, SCAX has been developed in such a way as to be transparent to the surrounding logic of the FPGA in which it is instantiated. All of its functionalities have been tested thoroughly in actual FPGA implementations, and the included software package that allows the user to automatically generate the necessary files that vary depending on the actual application, further enhances the package's ease-of-use. The NSW is the first project that will make use of the SCAX, since it also utilizes the SCA ASIC in its system. The SCAX emulates the SCA's interfacing with the already well-established back-end infrastructure (i.e. FELIX and the OPC UA server), thus making it fully compatible with the pre-existing set of tools that are used to access front-end ASIC address spaces. These facts make the SCAX an ideal solution for configuring and reading-out status registers of FPGA-based systems that use FELIX for their communication with the back-end.

Acknowledgments

This research is co-financed by Greece and the European Union (European Social Fund-ESF) through the Operational Programme "Human Resources Development, Education and Lifelong Learning 2014-2020" in the context of the project "Develop of an advanced system for evaluating of the data readout electronics system in Micromegas detectors for the upgrade of the ATLAS experiment" (MIS 5049537).

References

[1] ATLAS Collaboration , "ATLAS New Small Wheel: Technical Design Report," https://cds.cern.ch/record/1552862, 2013.

[2] ATLAS Collaboration, "The ATLAS Experiment at the CERN Large Hadron Collider," *JINST*, vol. 3, p. S08003, 2008.

[3] J. Anderson, A. Borga, H. Boterenbrood, H. Chen, K. Chen, G. Drake, M. Donszelmann, D. Francis, B. Gorini, D. Guest, F. Lanni, G. L. Miotto, L. Levinson, J. Narevicius, A. Roich, S. Ryu, F. Schreuder, J. Schumacher, W. Vandelli, J. Vermeulen, W. Wu, and J. Zhang, "FELIX: The New Approach for Interfacing to Front-End Electronics for the ATLAS Experiment," in *2016 IEEE-NPSS Real Time Conference (RT)*, 2016, pp. 1–2.

[4] M. Trovato, "FELIX: The New Readout System for the ATLAS Detector," in *2019 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, 2019.

[5] A. Caratelli, S. Bonacini, K. Kloukinas, A. Marchioro, P. Moreira, R. D. Oliveira, and C. Paillard, "The GBT-SCA, a Radiation Tolerant ASIC for Detector Control and Monitoring Applications in HEP Experiments," *Journal of Instrumentation*, vol. 10, no. 03, pp. C03 034–C03 034, mar 2015.

[6] P. Moschovakos, P. Nikiel, S. Schlenker, H. Boterenbrood, and A. Koulouris, "A Software Suite for the Radiation Tolerant Giga-bit Transceiver - Slow Control Adapter," *17th Int. Conf. on Accelerator and Large Experimental Physics Control Systems*, 2019.

[7] C. Bakalis, "SCA eXtension: a Design for FPGA Parameter Configuration within the ATLAS DAQ Scheme," in *2019 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, 2019.

[8] C. G. Larrea, K. Harder, D. Newbold, D. Sankey, A. Rose, A. Thea, and T. Williams, "IPbus: a flexible ethernet-based control system for xTCA hardware," *Journal of Instrumentation*, vol. 10, no. 02, pp. C02 019–C02 019, feb 2015. [Online]. Available: https://doi.org/10.1088/1748-0221/10/02/c02019

[9] "The Jinja2 Website," https://jinja.palletsprojects.com/en/2.10.x/templates/.

[10] M. Donszelmann, J. Valenciano, and J. Schumacher, "The Wupper Code Generator," https://pythonhosted.org/wuppercodegen/.