ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUAPL03

SOLVING VENDOR LOCK-IN IN VME SINGLE BOARD COMPUTERS THROUGH OPEN-SOURCING OF THE PCIE-VME64X BRIDGE

Grzegorz Daniluk*, Juan David Gonzalez Cobas, Maciej Suminski, Adam Wujek,
CERN, Geneva, Switzerland
Gunther Gräbner, Michael Miehling, Thomas Schnürer
MEN, Nürnberg, Germany

Abstract

VME is a standard for modular electronics widely used in research institutes. Slave cards in a VME crate are controlled from a VME master, typically part of a Single Board Computer (SBC). The SBC typically runs an operating system and communicates with the VME bus through a PCI or PCIe-to-VME bridge chip. The de-facto standard bridge, TSI148, has recently been discontinued, and therefore the question arises about what bridging solution to use in new commercial SBC designs. This paper describes our effort to solve the VME bridge availability problem. Together with a commercial company, MEN, we have open-sourced their VHDL implementation of the PCIe-VME64x interface. We have created a new commodity which is free to be used in any SBC having an FPGA, thus avoiding vendor lock-in and providing a fertile ground for collaboration among institutes and companies around the VME platform. The article also describes the internals of the MEN PCIe-VME64x HDL core as well as the software package that comes with it.

INTRODUCTION

The VME (Versa Module Europa) modular electronics standard emerged from the VME bus electrical standard and the Eurocard mechanical form factor. The former is a parallel communication bus developed in the 1980's. The latter defines the dimensions of Printed Circuit Boards (PCBs) as well as the mechanics of a rack hosting multiple cards. VME usually comes as a multi-slot chassis with a power supply and a fan tray (Fig. 1). Depending on the needs of a particular application, various types of VME crates are available in the market, starting from the smallest 1U, 2-slots to 9U, 20-slots with a possibility of having Rear Transition Modules (RTMs). These are the boards that are plugged to the slots at the back of a typical VME crate. They don't have direct access to the VME bus in the backplane, but instead connect to the corresponding modules installed in the front slots of a crate. Usually an RTM would be a quite simple board routing signals from cables connected in the back of the crate, to a front VME Slave board. The front module, on the other hand, is a more complex PCB with for example ADCs and an FPGA to process inputs and/or generate outputs going to a controlled system. A typical VME crate is deployed with one or multiple VME Master cards controlling several Slave cards. The Master module is very often a Single Board Computer (SBC). This is in fact a miniaturized PC running some operating system (e.g. Linux) and communicating with higher layers of the control system over an Ethernet link.

Figure 1: VME chassis example.

Despite, the availability of more modern standards, like MicroTCA or PXIe, VME is still a widely used solution for not only control systems in research facilities, but also in industrial and military applications. The reason behind that is all the already existing investment in the technology and the usual long lifetime of modular electronics in the fields mentioned above. People are reluctant to upgrade their critical systems and redesign their application-specific boards, once they have spent a long time to make them robust and reliable. For all these applications also the VME bus performance is very often sufficient, which delays modernization plans even further. Taking the CERN accelerators case as an example, we currently have almost 900 VME crates installed in various operational and lab systems. Out of those, about 50 new crates were installed only in 2016. We still plan to install about 200 new VME crates in various renovations during Long Shutdown 2 in 2019-2020.

VME BUS

VME bus was originally developed in the 1980's for Motorola 68k processors as a multi drop, parallel bus with big endian byte ordering [1]. It is organized as a master-slave architecture, where master devices can transfer data to and from slave cards. In practice, the majority of setups use a single master, multiple slaves configuration. It is a quite slow communication bus for today's standards. For the base specification, it can reach 80MB/s throughput. An absolute

* grzegorz.daniluk@cern.ch

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUAPL03

maximum data rate of 320MB/s can be achieved using the 2eSST extension [2].

VME bus is asynchronous, which means there is no common clock to be used for data transfer among all cards in the crate. Instead, 4-edge handshaking is used to validate data and for flow control. This, together with the fact that the bus lines are shared among all the devices, means that the overall performance of the system is determined by the slowest module. The standard splits the communication lines into four sub-buses:

- Data Transfer Bus - address, data and control lines for data transfers
- Data Transfer Bus Arbitration - used in multi-master setups to arbitrate bus accesses among all the masters.
- Priority Interrupt Bus - 7 prioritized interrupt lines and acknowledge daisy chain
- Utility Bus - reset, 16MHz system clock, power information signals

There are various size of data transfers possible in the VME environment, but in general they can be collected in two main groups: single cycles and block transfers. The type of transfer is determined by address modifier lines `AM[5..0]` - part of the Data Transfer Bus. Although the standard defines 32 address and 32 data lines, not all of them have to be used for a given transfer. The number of valid address bits is also determined by the address modifier with possible options being A16, A24 and A32 (respectively 16, 24 and 32 address lines used). Typically a VME slave would respond to one of the addressing modes. The data bus can also be used partially to read/write 8, 16 or 32 bits in a single cycle (D8, D16, D32 modes).

Figure 2 shows how a single cycle VME access is done between master and slave devices:

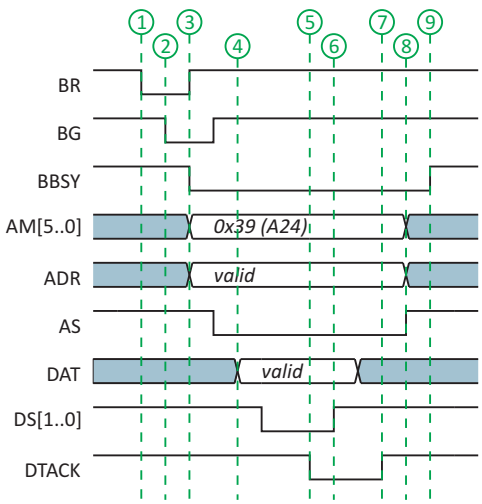

Figure 2: Example single cycle VME access.

1. Master requests the bus access by driving `BR` bus request line.
2. If `BBSY` line is idle, i.e. no other master is currently holding the bus, the arbiter drives the `BG` line (bus granted) to allow the requesting master to take control over the bus.
3. Master sets the transfer type to A24 single cycle with the address modifier lines (`AM[5..0]`); puts the address it wants to access (for A24, only 24 addressing bits are valid); asserts the address strobe (`AS` line).
4. Master puts data on the bus and asserts data strobe (`DS` lines) to validate the data word.
5. Slave reads the data from the VME bus and drives data acknowledge (`DTACK`).
6. Master releases data strobe after receiving acknowledge from the Slave.
7. Slave releases data acknowledge.
8. Master releases address strobe and bus busy lines. The transfer is finished and the bus is released for another operation.

The single cycle accesses are a good way to read/write a few bytes. However, they are not a very efficient way for transferring larger blocks of data. For these, the VME standard defines also block transfers (BLT) and multiplexed block transfers (MBLT). The former is based on the single cycle access mechanism described before. It sends multiple data words one after another using the data strobe and data acknowledge lines for each of them. The difference is that the state of the address lines does not change during the whole transfer. Instead, both master and slave use their internal counters to increment the address for each data word. The multiplexed block transfer brings another performance improvement. It uses address lines together with data lines to transfer 64 bits of data in each cycle.

In all transfer types mentioned till now, the master device is the originator of the read or write cycle. In real-world applications, where the slave could be a multi-channel ADC, it is more efficient if it has a way to indicate when there is some data to be fetched. For this purpose VME bus uses the Priority Interrupt sub-bus. There are 7 interrupt lines `IRQ[7..1]` mapped to 7 interrupt priorities and the interrupt acknowledge `IACK` daisy chain. If any of the VME slaves wants to generate an interrupt, it drives the IRQ line for a desired priority and waits for the `IACK` from the interrupt handler. Each VME board gets `IACK` from the previous board in a VME crate and forwards it to the next one (if it is not waiting for `IACK`). This ensures that only one slave responds to the acknowledge. If there are multiple VME slaves generating interrupts of the same priority, the one closer to slot 1 is served first. The interrupt handler upon detection of any of the `IRQ[7..1]` lines going down, originates a special single cycle access to read the 8 bit IRQ vector. This IRQ vector identifies the slave that has generated the interrupt.

Since all slaves installed in a VME crate are connected to the same VME bus, they need to be uniquely addressed to distinguish transfers designated to each one of them. In the original VME bus standard, every slave card in the system had its own base address and IRQ vector set mechanically

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUAPL03

with on-board jumpers or dip switches. Later, the VME64x extensions added the CR/CSR configuration space with geographic addressing for plug&play functionality [3]. With these improvements, base addresses do not have to be set mechanically for every deployed board. Instead, each slave has a well defined Configuration ROM (CR) and a set of Control and Status Registers (CSR). The address of CR/CSR space for each card can be derived from the slot number where the card was plugged (geographical addressing) and can be accessed by using a special address modifier 0x2F. During initialization, the master relocates each module to the desired base address by writing it to the CR/CSR space.

Other VME extensions also introduce fast data transfer modes like 2eVME and 2eSST [2]. These however are not used in CERN VME systems and are outside the scope of this paper.

PCI-EXPRESS BUS

PCI-Express (PCIe) is a high speed, serial communication bus found in all modern computers. Although the first official standard for PCIe 1.0 was published in 2002, the technology has been improved several times since then. At the time this paper is written, PCIe 4.0 is released and the work has officially started for standardizing PCIe 5.0.

In contrast to the VME bus introduced in the previous section, PCI-Express is based on serial, point-to-point communication. It uses differential pairs for full-duplex data transfer and two of such pairs create what is called a PCIe lane. In the simplest scenario (and the slowest data rate) a PCIe link can have just a single lane (PCIe x1). In that case, for PCIe 1.0 the maximum data rate is 250MB/s. For higher throughputs, multiple lanes can be combined for interleaved data transfer. The most common configurations are: x2, x4, x8 and x16 with 2, 4, 8 and 16 PCIe lanes. Figure 3 shows a typical PCIe system. It consists of several different devices:

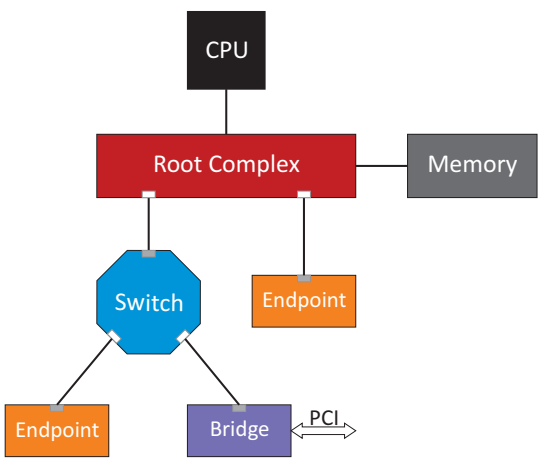

Figure 3: PCI Express topology.

- Root Complex - connects CPU and memory to the PCIe system and may have several ports.
- Endpoint - peripheral device connected to the bus (e.g. network interface, graphics card, etc.)
- Switch - multi-port device that allows connecting several Endpoints to a single port of the Root Complex.
- Bridge - translator used when another bus needs to be connected to the PCIe system (e.g. legacy PCI hardware).

Originally, when PCIe was introduced the PCIe Root Complex was connected to the CPU through the north bridge. Later however, manufacturers started integrating it together with the memory controller onto the processor die.

Similarly to computer networks, PCIe uses packets to send data between the Root Complex and Endpoints. Since there are no extra lines like in parallel buses, also all the control messages and interrupts are sent as packets using those same data lanes. To analyze how these packets are assembled and sent over the fabric, the standard [4] splits the PCIe protocol into logical layers: Transaction Layer, Data Link Layer and Physical Layer.

The Transaction Layer encapsulates all read/write transactions and events into Transaction Layer Packets (TLPs). It also maps each request coming from software to one of the 4 transaction types:

- Memory read/write - transfer to a memory mapped location
- I/O read/write - transfer to an I/O mapped location
- Configuration read/write - transfer to configure a device and read its capabilities
- Messages - special events like legacy interrupts or power-management.

This layer also manages point-to-point flow control by using a credit-based mechanism. Since a PCIe system contains switches and multiple Endpoints may be connected to a single Root Complex port, TLPs have to be addressed to mark their destination. There are several addressing schemes available in PCIe systems. For memory and I/O read/writes, the TLP header has a 32 or 64-bit destination address field. Each PCIe device has a set of Base Address Registers (BARs) to store their base addresses assigned during bus enumeration (using configuration requests). For Configuration requests, the TLP header specifies a set of IDs for the bus, device and function numbers. Messages use a special type of addressing e.g. *to Root Complex* or *broadcast from Root Complex*.

The Data Link Layer manages the PCIe communication link. It prefixes the TLPs sent by the Transaction Layer with a sequence number. It also ensures data integrity by error detection and correction, and manages retransmission when needed.

The Physical Layer is the closest one to the actual wires connecting PCIe devices. Therefore it is responsible for impedance matching, data serialization/deserialization and contains logic responsible for initializing and maintaining the interface. Before TLPs are serialized in the Physical Layer they are first encoded using a 8b/10b or 128b/130b scheme (depending on the PCIe generation). For example

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUAPL03

8b/10b encoding maps every 8-bit word into a 10-bit symbol so that the output data stream is DC balanced (does not contain a DC component). It also ensures there are enough state transitions that the PLL in the Physical Layer of a receiving device is able to recover the clock signal from the received data stream.

PCIe devices can also generate interrupts to the host CPU. These are also packets sent through the PCIe fabric and we call them Message Signaled Interrupts (MSI). MSIs are in fact regular memory write transactions originated by the interrupters to a predefined address. This address is configured by the CPU during device initialization.

WHY ANOTHER BRIDGE?

For many years, computers have been able to use buses like PCI or PCI-Express to communicate with their peripherals. Nowadays every PC-class processor has a built-in PCIe Root Complex with multiple ports. However, there are no processors that would have an integrated VME master interface. In consequence, every VME Single board computer needs to have a PCI or PCIe to VME bridge. Till mid-2015 the IDT TSI148 chip was the de-facto standard bridge used in many VME systems. However, it has been discontinued, therefore the question has arisen about what bridging solution should be used in new commercial SBCs.

For the CERN VME installations we were providing our users with MEN A20 Single Board Computer boards. Since these were also using the obsolete TSI148 chip, we had to find a new solution. We issued a call for tender for supplying new SBCs for the next few years. There were three possible options specified for the PCI/PCIe to VME64x bridging:

- the company should have enough stock of TSI148 chips to be able to produce the number of boards specified in the contract
- use the Tundra Universe II, the predecessor of TSI148
- use FPGA technology - in that case we required the bidders that the complete HDL sources for the FPGA design shall be made available through a GPL3-or-later license.

Besides the first two obvious choices, we knew there were companies with proprietary implementations of VME bridges done in FPGAs. Previous generations of SBC we used at CERN (before TSI148-based boards) had a PowerPC processor with an FPGA attached to it for interfacing to a VME bus. Therefore, with the last option in the call for tender we hoped that at least one of these companies would be ready to open-source their implementation. On the other hand, to keep fair conditions to anyone submitting their offers, we did not give preference to any of these options. The final selection of the company was based on the price offer submitted to the call for tender. In the end, the company with the best pricing offer to be granted the contract was MEN from Nürnberg, Germany. They proposed their brand new A25 card to be a next generation VME Single Board Computer used in CERN installations.

MEN A25 is based on the Intel server class, 4-core XEON processor connected to 8GB DDR3 RAM. To interface to the VME bus an FPGA-based PCIe-to-VME bridge is used. MEN has made their own VHDL implementation using an Intel Cyclone IV FPGA. After the company was granted the contract, we started working with them to test and validate the A25 board for CERN VME installations and to publish their VME bridge implementation. As a result, all the VHDL sources are available under the GPL3-or-later license and the Linux driver package under the GPL2-or-later license in the *pcie-vme-bridge* project page of the Open Hardware Repository [5].

Open sourcing the PCIe-to-VME bridge is a big step not only for CERN, but also for all other places around the world where VME is still in use. First of all, we do not depend any more on a particular vendor discontinuing their VME bridging solution. Even if the FPGA chip that is currently used becomes obsolete, having complete VHDL sources lets us port the bridge to various other FPGA families. Thanks to the fact that the design is open, any institute or company can now not only buy an existing MEN A25 product but also build any other VME Single Board Computer that would use the same VME bridge. Using the same VME bridge means also the same Linux kernel and userspace VME API for all institutes/companies. In the future, this should allow us to collaborate more efficiently in the VME world and have more freedom to share and re-use Linux kernel drivers for the VME Slave boards that we design.

FPGA IMPLEMENTATION

The MEN PCIe-to-VME bridge translates the read and write operations in the PCIe address space to read and write transactions on the VME bus. It acts as a PCIe Endpoint on one side and VME bus Master on the other. The bridge can generate VME single cycles and block transfers. The following access types are currently supported:

- VME single cycles: A16, A24, A32 with any of the D8, D16, D32 data widths
- VME block transfers: A16, A24, A32 with any of the D8, D16, D32 plus the A32D64 multiplexed block transfer (MBLT)

The VME block transfers are executed by a built-in Direct Memory Access (DMA) engine, where the blocks of data are transferred between the system memory and the VME bus, bypassing the CPU. In general this is a faster and more efficient way of exchanging multiple data words, as the CPU is free to continue its normal operation until the DMA engine is done with a programmed task. The bridge supports also some features added in the VME64x extensions. It is able to use the geographical addressing pins and generate a special type of A24 access to read and write the CR/CSR configuration space of VME slaves installed in the same crate. However, none of the fast transfer modes (2eVME, 2eSST) is currently implemented.

The internal HDL architecture of the PCIe-to-VME bridge is presented in figure 4. It is built around the Wishbone (WB)

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUAPL03

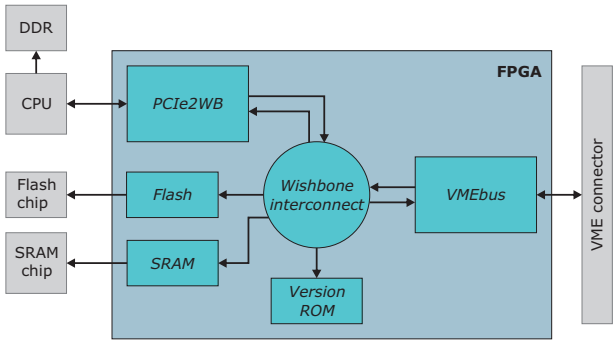

Figure 4: PCIe-to-VME bridge HDL architecture.

Table 1: PCIe Memory Windows

BAR no.	name	offset
BAR0	*Version ROM*	0
BAR0	*Flash*	200
BAR0	*VMEbus* - registers	10000
BAR0	*VMEbus* - A16D16 access	20000
BAR0	*VMEbus* - A16D32 access	30000
BAR1	*SRAM*	0
BAR2	*VMEbus* - A24D16 access	0
BAR2	*VMEbus* - A24D32 access	1000000
BAR3	*VMEbus* - A32 access	0
BAR4	*VMEbus* - CR/CSR access	0

bus, an open-source computer bus often used in FPGA designs. The bridge is split into several VHDL modules communicating together through the central *Wishbone Interconnect* block. Each of these modules has its own function and can either control other modules (is a Wishbone Master), be controlled (is a Wishbone Slave) or have both interfaces:

- *PCIe2WB* - PCI Express x4 version 1.1 Endpoint with both WB Master and WB Slave interface.
- *VMEbus* - VME Master with both WB Master (for DMA transfers) and WB Slave (for single cycle accesses) interface.
- *Flash* - WB Slave module that interfaces the Flash chip outside the FPGA
- *SRAM* - WB Slave module that interfaces the SRAM chip outside the FPGA
- *Version ROM* - WB Slave module with FPGA memory blocks initialized at synthesis time with various information about the firmware (the so called chameleon table).

The *PCIe2WB* module is in fact a wrapper for the Intel auto-generated IP core. This IP core customizes a PCI Express IP block hardened in the Cyclone IV FPGA chip. Currently we use it as a four-lane PCIe version 1.1 interface with vendor Id and device Id specified by MEN. It provides several memory windows assigned to 4 Base Address Registers. These memory windows are then mapped to the wishbone addresses and therefore allow accessing the WB Slave devices as well as generating different VME accesses (see table 1).

The *VMEbus* module can act as both a VME Master and a VME Slave. However, in this paper and for the VME Single Board Computer application, we focus only on its VME Master functionality. It provides several Wishbone address spaces for various types of access (A16, A24, A32, CR/CSR). These WB windows are then directly mapped to the PCIe memory windows, therefore accessing one of the PCIe windows automatically generates a VME bus access of appropriate type and data width.

The behavior of this module depends on the detected VME crate slot where it is installed. If slot 1 is detected, the bridge generates system clock and reset signals to the VME backplane. An arbiter module is also activated in that case, thus the card can arbitrate accesses in multi-master environments by driving the Bus Granted line. The *VMEbus* VHDL module can also read the VME interrupt lines of all 7 levels. The built-in VME interrupt handler can be configured to forward only a subset of the priorities when needed. Apart from the standard cycles the bridge has also an option to perform a read-modify-write cycle. In that case the bus is blocked after the first read until a consecutive write is done to the same address. The module is also equipped with bus debugging components. There are two independent location monitors that can be programmed with a VME address value (one for each). These modules, when armed, constantly monitor the bus and generate interrupts to the CPU when the required address is detected on the VME bus.

The *VMEbus* VHDL module is also equipped with a Direct Memory Access (DMA) engine. It is responsible for performing block transfers and multiplexed block transfers between the VME bus and the system memory without the active control of the CPU during the transfer. The controller can be configured by writing a set of linked buffer descriptors to the SRAM area (maximum 112). Each descriptor specifies the complete information about the transfer like: the source and destination device, size of the transfer, source and destination address, VME modifier and data width (for VME transactions). Additionally one can specify if the DMA should automatically increment the source or destination address for each transferred word in the block, e.g. for transferring data words to SRAM. The source and destination device can be any of of VME bus, PCIe2WB or SRAM modules.

Besides these two main VHDL modules there are also a few smaller ones in the bridge design. The SRAM block is a VHDL core that translates 32-bit data width Wishbone accesses into the 16-bit SRAM interface. The Flash module instantiates the Intel Remote Update IP core which lets us access the on-board Flash memory. It is used to store the FPGA image and program it when the board is powered up. In fact, we store there two images, the fail-safe image (at ad-

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-TUAPL03

dress 0x000000) and the main image (at address 0x200000). This way, the fail-safe image is flashed only once at production time and it has to provide at least the PCIe access to the Flash HDL module. The main image we can re-write over the PCIe interface whenever we want to release a new bridge firmware. When the board is powered on, the FPGA gets configured with the main image. However, if for some reason the main image is corrupted, the FPGA is configured with the fail-safe image.

Finally, the last HDL module present in the PCIe-to-VME bridge design is the *Version ROM*. It is in fact an instantiation of FPGA RAM memory blocks initialized at synthesis time. The ROM content is a binary called the Chameleon table generated from an **.xls* file. This **.xls* contains various information about the FPGA bitstream including: firmware version number, Wishbone addresses and PCIe BAR mappings for each HDL module. The *Version ROM* is read by the software tools and Linux kernel drivers that access the FPGA.

The whole bridging design described briefly in this section occupies about 30% of the Intel Cyclone IV EP4CGX30. This means that even on its current hardware platform (MEN A25 Single Board Computer) there is still plenty of space for new features. To facilitate any future developments, we have published together with MEN also their set of VHDL testbenches for the described bridge design. They use an Intel PCI Express bus functional model (BFM) to transfer test data patterns between the PCIe and VME interfaces of the bridge. The testbench is fully automated and includes both the stimulus generator and the verification module. It runs a set of tests for both the VME Master and Slave functionality where various access types are checked. After that, also the interrupt handler and various possible configurations of the DMA engine are verified.

LINUX KERNEL DRIVERS PACKAGE

MEN provides a set of Linux kernel drivers to access the FPGA bridge from the operating system that can be compiled for both 32 and 64-bit architectures. After the drivers are loaded, they require manual creation of character devices, where each of them represents a different VME access mode: `/dev/vme4l_a16d16`, `/dev/vme4l_a24d32`, `/dev/vme4l_cr_csr`, `/dev/vme4l_a32d32_blt`, etc. Read/write operation on a given device is then mapped to a read/write to the corresponding PCIe memory window, which is translated by the FPGA bridge to a corresponding VME bus access. The driver also supplies a kernel-level interface to enable custom drivers development for VME Slave devices.

The original TSI148 API used at CERN provides dual userspace and kernel level interface, as is the case with the new API developed by MEN. The dual interface makes it possible to write VME device drivers using the kernel level interface or develop userspace applications that take advantage of the character devices and associated I/O control calls. Despite the MEN and CERN interfaces being different, the set of possible operations executed on the VME bus does not change. This means the only adaptation we had to develop as part of the CERN internal support project was a wrapper module which translates the original TSI148 interface calls to the new FPGA-based bridge interface. The final result is a kernel module that is loaded on top of the MEN drivers. The module provides the original TSI148 interface that can be used by any previously developed application, and internally calls functions exported by the A25 driver. Such solution is very convenient for the users, as it allows them to keep running their well-tested software without any modifications. In case of any new FPGA bridge driver releases by MEN, the CERN driver update operation is seamless as long as the FPGA bridge interface remains unchanged.

CONCLUSIONS

The open source, FPGA-based implementation of the PCIe-to-VME bridge solves the bridging problem for VME Single Board Computers forever. Since now, any company can take this design and make a custom SBC around it. For places where VME is still actively used, this bridge provides freedom to change among various Single Board Computers at will without re-writing custom VME drivers and user space applications. With this new PCIe-to-VME bridge, we also aim at stimulating VME collaborations among research institutes as now they can more easily re-use the VME Slave boards together with their custom Linux drivers.

We would like to gather a community among people working with VME electronics to help us improve this newly published bridge.

FUTURE WORK

The FPGA-based PCIe-to-VME bridge described in this article is ready to be used in VME systems. In the process of insourcing the design we have identified some places in the VHDL code that could be improved in the future. One of them would be to clean up the code and use some Wishbone register generator (like *wbgen2* [6]) to provide a convenient and consistent method of accessing fields of the configuration registers. Currently this is done with many hardcoded indexes in various places of the code. We could also foresee adding support for the fast data transfer modes like 2eVME and 2eSST.

REFERENCES

[1] *American National Standard for VME64*, ANSI/VITA Std. 1.0-1994

[2] *2eSST*, ANSI/VITA Std. 1.5-2003

[3] *VME64 Extensions*, ANSI/VITA Std. 1.1-1997

[4] *PCI Express Base Specification*, PCI-SIG Std. Revision 1.1

[5] PCIe-to-VME bridge project page and sources, `https://www.ohwr.org/projects/pcie-vme-bridge`

[6] *wbgen2*: Wishbone slave generator `https://www.ohwr.org/projects/wishbone-gen`