2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

# An Open and Fast Virtual Platform for TriCore™-based SoCs Using QEMU*

Bastian Koppelmann, Bernd Messidat, Markus Becker, Christoph Kuznik
*C-LAB, University of Paderborn, Fuerstenallee 11, Paderborn, Germany*

Wolfgang Mueller, Christoph Scheytt
*Heinz Nixdorf Institute, University of Paderborn, Fuerstenallee 11, Paderborn, Germany*

***Abstract*—In this paper, we present an efficient approach to virtual platform modeling for TriCore™-based SoCs by combining fast and open software emulation with IEEE-1666 Standard SystemC simulation [9]. For evaluation we consider Infineon's recently introduced AURIX™ processor family as a target platform, which utilizes multiple CPU cores operating in lockstep mode, memories, hierarchical buses, and a rich set of peripherals. For SoC prototyping, we integrate the fast and open instruction accurate QEMU software emulator with the TLMu library for SystemC co-verification [2]. This article reports our most recent efforts of the implementation of the TriCore™ instruction set for QEMU. The experimental results demonstrate the functional correctness and performance of our TriCore™ implementation.**

***Keywords*—HW/SW co-verification; software emulation; dynamic binary translation;**

## I. Introduction

Today, early HW/SW co-verifications for embedded systems system level designs are crucial activities with ever growing systems complexity and shortened time-to-market requirements. Best practice gaining productivity is the application of pre-silicon development platforms for virtual prototyping supporting architectural and timing properties at adequate levels of details.

We present a virtual platform approach for TriCore™-based SoCs by combining open and fast software emulation with a discrete event-driven simulation interface for IEEE-1666 Standard SystemC [9]. Our studies and implementation focus on Infineon's recently introduced AURIX™ processor as a target platform (see Figure 1). As such, the platform supports the integration of multiple QEMU cores, with a TLM 2.0-based SystemC bus model by interfaces for co-simulation of AURIX™ peripherals like memories and hierarchical buses. Via TLM 2.0 this approach also supports the integration of a variety of existing TLM or RTL hardware models by means of appropriate transactors.

In competition to QEMU there exist several other commercial and non-commercial tools based on the principles of Just-in-Time (JiT) compilation like OVPSim [10] or Synopsys' CoMET/METeor [12]. Unlike QEMU they either do not support the TriCore™ architecture and/or their source code is either partly or fully not freely available. Thus, they are limited for building an entire open virtual platform. Additionally, there are several approaches combining QEMU and SystemC, such as GreenSoCs' SystemC-QEMU [11] or Rabbits [8]. We have also introduced an

*This work was partly funded by the German Ministry of Education and Research (BMBF) through the projects EffektiV (01S13022) and ARAMiS (01IS11035).

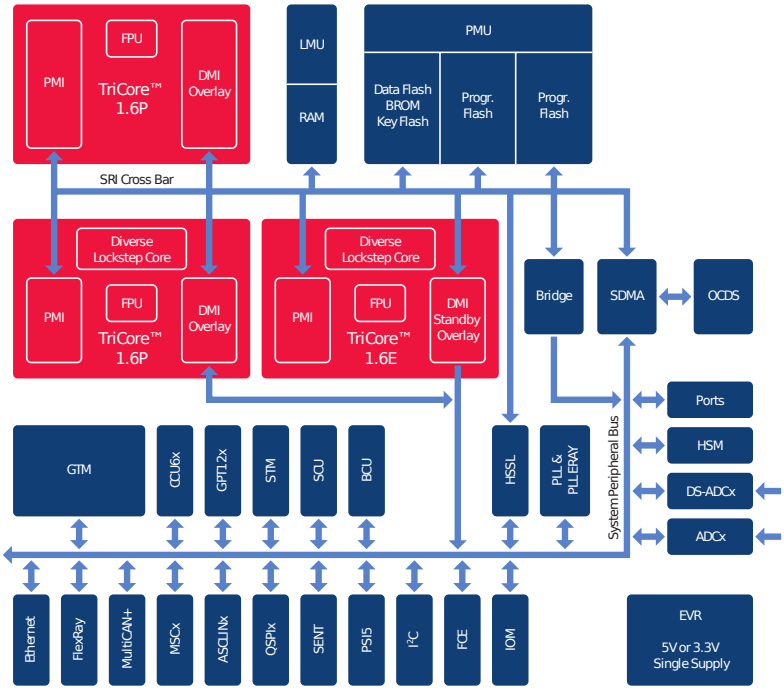


Figure 1: Infineon AURIX™ platform architecture [1].

approach for the integration of QEMU user mode emulation with abstract RTOS (Real-Time Operating System) and HAL (Hardware Abstraction Layer) models in SystemC as an intermediate step towards QEMU full system emulation [4, 6, 5]. However, in general, all existing QEMU/SystemC integrations are very specific/limited and, moreover, they are mostly based on outdated QEMU versions, so that their migration to current versions is either hardly feasible or at least most time consuming. In this article, we present the TLMu [2] library as a general wrapper concept for current QEMU versions. This is based on our most recent QEMU TriCore™-based processor emulation, which supports the vast majority of the TriCore™ instruction set.

The remaining paper is organized as follows. Section II. gives a brief introduction to QEMU, the TriCore™ architecture, and SystemC. Section III. presents details of our QEMU implementation and the TriCore™ instruction set. Section IV. presents details of our combined QEMU/SystemC virtual platform based on TLMu. Section V. shows results from our experiments. Finally, Section VI. concludes with an outlook on open issues.

## II. Fundamentals

### A. *QEMU*

QEMU is an open source software for PC virtualization and processor and systems emulation. It supports huge array of embedded systems instruction set architectures (ISAs) like ARM, PowerPC, SPARC, MIPS, or Microblaze as guests and x86, IA-64, ARM, PowerPC, MIPS or Sparc for emulation host implementations. However, TriCore™ is not supported yet. For emulation QEMU supports two different software emulation modes: user mode and full system mode. The first one allows the emulation of a single user process. System calls of the emulated guest system are not handled by QEMU rather than handed over to the host operating system. Full system emulation on the other hand allows the emulation of a complete system, including peripherals and physical memory with a memory management unit (MMU). For full system mode, all peripheral models have to be implemented by a

QEMU-specific variant of the C language. To emulate a CPU, QEMU applies dynamic binary translation, which is a Just-in-Time compilation technique that allows to efficiently emulate the software binary of the guest system by caching already translated basic blocks. For binary translation and a higher portability, QEMU implements an intermediate layer, the Tiny Code Generator (TCG). The TCG frontend translates guest instructions into an intermediate format (microcode); the TCG backend in turn translates that intermediate format into host instructions. Additionally, the TCG allows the generation of TCG helper functions, which can be created instead of a host instruction. A helper is basically a C function for the support of more complex emulations.

### B. *TriCore™ Architecture*

Infineons' TriCore™ is a 32 bit ISA mainly used for automotive applications. It combines MCU, RISC and DSP features into one instruction set supporting instructions of 16 and 32 bit length. Most instructions can be executed in one clock cycle [3] where all data in memory is organized in little endian byte ordering. TriCore™ supports several special features. A unique feature is the automatic handling of the processor context. On each call instruction a set of registers is automatically saved to a context save area and restored on return. TriCore™ also provides a set of core special function registers (CSFR) that are mapped into the memory address space. Additionally, the TriCore™ interrupt system is built around programmable service request nodes (SRN). Each of those nodes may have up to three service providers, like a CPU or a DMA unit. Each interrupt from a module is connected to a SRN. All these SRNs are again connected over a bus to an interrupt control unit (ICU), which communicates with the service provider.

### C. *SystemC*

IEEE-1666 Standard *SystemC* [9]. is a C++-based library with SystemC language elements and a simulation kernel. SystemC allows HW developers to model the architecture and behavior of HW components as C++ objects taking advantage of C++'s object-oriented features. Meanwhile, many chip manufacturers and IP vendors apply SystemC and provide SystemC models allowing a system designer to quickly compose HW system simulations from proprietary and externally provided IPs for testing and evaluation.

SystemC is designed for integrating HW/SW modules (*SystemC modules*) at arbitray abstraction levels and compile them into a single executable HW/SW co-simulation. The behavior of the modules is defined by concurrent processes (*SystemC threads and methods*) that are synchronized by events. Meanwhile, Transaction-Level Modeling (*TLM-2.0*) is part of the SystemC since IEEE 1666-2011. TLM 2.0 defines coding-styles and interfaces for bus abstractions at different levels of accuracy and timing between modules. For a more detailed introduction to SystemC and TLM 2.0 the reader is referred to the IEEE Standard [9] or to various widely available tutorials.

## III. Modeling TriCore™

In order to implement the TriCore™ ISA for QEMU, we introduce three models: processor state model, instructions set model, and memory model.

### A. *Processor State Model*

We reduced the TriCore™ architecture to the functional interface of the processor. The model contains 32 general purpose registers (GPR), 3 system registers, and 111 core special function registers (CSFR). As we focused on the

implementation of the QEMU full system mode, we implemented emulated memory (see Section C.).

GPRs are divided into 16 data (d0-d15) and 16 address (a0-a15) registers. Several of those registers are used by convention for dedicated tasks, e.g., the return address (a11) or the stack pointer (a10). The system register contains the program status word (PSW), the previous context information and pointer register (PCXI) and the program counter (PC); registers d8-d15, a10-a15, PSW and PCXI form the upper context. Those registers are automatically saved on a call instruction into a context save area (CSA).

We implemented the processor state along the implementation of other QEMU ISAs with a data structure that contains all those registers as a variable. The frequently used GPR and system registers a linked to the TCG as TCG register and thus allow to use those as registers in microcode, which improves the performance when accessing those registers.

The CSFR is a 64KB register set that is mapped into the virtual address space of a process. All GPR and system registers are mapped into the same space. Additionally, it contains register for special purposes, like MMU control or the handling of CSAs. Those registers can only be accessed by two dedicated instructions (Move To/From Core Register - MTCR/MFCR) that access the registers by an offset which is added to a software defined base address. The additional registers are not linked to the TCG; their access is described in the register paragraph of Section B..

### *B. Instruction Set Model*

To implement the TriCore™ ISA for QEMU, we defined five classes: *arithmetic/logic*, *load/store*, *control*, *register*, and *system*. Hereafter, we outline how we implemented the instructions of those classes focusing on special instructions in greater detail. Finally, we close with a short example of the translation of a simple TriCore™ program.

**Arithmetic/Logic:** contains instructions like *add*, *addi* or *and*; they typically use GPRs. Most of them can be implemented by an equivalent microcode operation that accesses the processor context through TCG registers.

**Register:** contains instructions that move data between registers like *mov*. Most of them can be implemented — like arithmetic instructions — by equivalent microcode operations. An exception is the *move to core register*(MTCR) instruction, which moves data from a GPR to a CFSR. CFSR are addressed by a 16 bit offset that is added to the base address of the CFSR address space. We used an offset lookup at translation time to directly generate an access to the CSFR, which are part of the emulated processor state to avoid a slow memory access to the guest memory.

**Load/Store:** contains instructions for moving data between registers and memory. They can be implemented by an equivalent microcode operation for direct addressing. Indirect addressed instructions need an extra add operation to calculate the address. For bit reverse and circular addressing, we implemented a TCG helper that computes the addresses.

**System:** contains instructions to control the processor like instructions for exception handling or disabling/enabling interrupts. Most of them depend on interrupting the execution of a single translated basic block and are, therefore, implemented by a TCG helper function to give the control back to QEMU. A special case is the debug instruction, which is used for hardware debugging on the real chip. We use it in our implementation to quit the emulation process.

Table I: Translation of a TriCore™ basic block.

| TriCore™ assembler (front end) | TCG micro code (intermediate) | x86 assembler (back end) |
|---|---|---|
| `mov d0,5` | `movi_i32 d0,0x5` | `mov 0x5,ebx`<br>`mov ebx,0x70(ebp)` |
| `addi d0,d0,5000` | `movi_i32 tmp0,0x1388`<br>`add_i32 d0,d0,tmp0` | `mov 0x70(ebp),ebx`<br>`add 0x1388,ebx`<br>`mov ebx,0x70(ebp)` |
| `call a000000a` | `movi_i32 tmp0,0xa0000008`<br>`movi_i32 tmp1,0x2`<br>`movi_i32 tmp2,0x6007277c`<br>`call tmp2,0x0,0,env,tmp0,tmp1`<br>`movi_i32 PC,0xa000000a` | `mov ebp,(esp)`<br>`mov 0xa0000008,ebx`<br>`mov ebx,0x4(esp)`<br>`mov 0x2,ebx`<br>`mov ebx,0x8(esp)`<br>`call 0x6007277c`<br>`mov 0xa000000a,ebx`<br>`mov ebx,0x8(ebp)` |
| | `exit_tb 0x0` | `xor eax,eax`<br>`jmp 0x621dfeb4` |

**Control:** contains instructions that conditionally and unconditionally branch the program flow. Those instructions always mark the end of a basic block and thus the end of a binary translation. We distinguish between jump/branch, loop and call/return instructions. Unconditional jump instructions can be implemented by a move microcode operation that moves the target address to the emulated PC register. For conditional branch instructions, we additionally need to evaluate the branch condition. For that case, the TCG allows to define jump markers in microcode, which can be used by jump microcode operations that allow a conditional jump to those markers. We use them to evaluate the condition and generated a move of the target address into the emulated PC register for true conditions. If false, we generate a move that increases the emulated PC register by the instruction length.

The loop instruction is a special case of a conditional branch instruction and is used to efficiently implement loops. It checks if a specified register is zero. If false, it jumps to the target address. If true, it increases the PC by the instruction length. Additionally, it decreases the specified register by one. We implemented it as a branch instruction and added a subi microcode operation that decrements the register.

Special features of the TriCore™ include the call and return instructions. In principle, they work like unconditional jump instructions. Additionally, in the case of the call instruction, it saves the return address and the context of the processor automatically. We implemented the call instruction like an unconditional jump instruction additionally saving the return address by a move microcode operation and using a TCG helper to implement the saving of the processor context. The return instruction is implemented correspondingly.

**Translation Example:** Let us consider the translation of a simple basic block with three TriCore™ instructions (mov, addi, and call) as given in Table I. The columns show the TriCore™ instructions (left), the translated TCG microcode (middle), and the corresponding generated x86 instructions executed by the host (right). In this example, we assume that the basic block has not been translated before and is therefore not cached.

A translation for a basic block always starts with the generation of a prologue and ends with generation of the epilogue. In between, the TriCore™ instructions are loaded and decoded until an instruction of the control class is encountered. Decoding of TriCore™ instructions is done by a routine that determines its type and format through applied bit masks. With this data, we can generate equivalent microcode operations. The simplest case is illustrated by the move instruction, which can be implemented by a mov microcode operation. The addi instruction, in contrast, needs two microcode operations, which are linked through a temporary TCG register (tmp0). In order

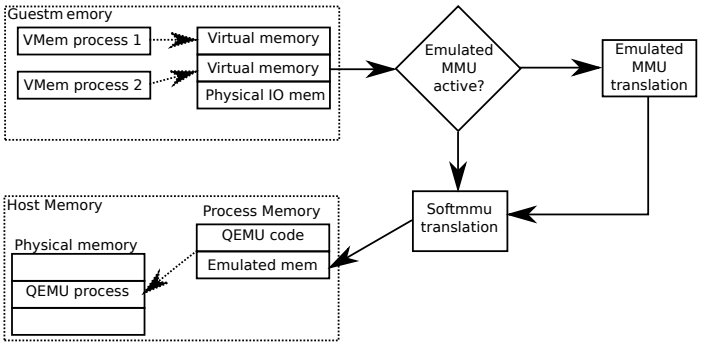


Figure 2: Overview of the QEMU memory emulation subsystem.

to save the context for the call instruction, a call to a TCG helper is implemented. That helper has three parameters: the return address, the processor state, and the instruction length, which are moved into temporary registers (tmp0, ..., tmp2) by a mov microcode operation. Then a call to this helper function is generated. The function itself checks first, if a CSA frame is still available to store the current context. If not, an exception is generated. Otherwise, the address of the CSA frame is converted by the software emulated MMU – the so-called *softmmu* – (see Section C.) in order to copy the register data into the guest memory. Finally, we update the linked lists to handle used and free CSA frames. The final call instruction stops the translation process for the basic block. Additionally, an exit_tb microcode operation needs to be generated to mark the end of the basic block. It is used to return the control to QEMU when the basic block is executed. Finally, x86 code is generated from the TCG microcode by the TCG which is executed on the host.

### C. *Memory Model*

QEMU full system mode emulates the physical memory and peripherals mapped into physical memory. For that QEMU uses a memory management unit that translates guest memory addresses into host memory addresses. They are also cached within a QEMU translated block. It is important to note that the softmmu it different from the MMU of the guest or the host system. The MMUs can co-exist with the softmmu. Figure 2 illustrates this with two executed processes of the guest system. They can be mapped into the virtual memory of the guest system. Those addresses are then translated by the softmmu into the addresses of the QEMU process in the host memory. The host addresses can be virtual addresses of the host system. Whenever a memory access occurs in the emulation, the corresponding address is translated by the softmmu. In order to emulate peripherals, the address space of the device is marked in the softmmu as peripheral space and a function for write and read accesses is defined in QEMU, which controls the access the device. In our approach, those functions are used for defining a device that is mapped into the whole emulated address space for interfacing with the TMLu. As such, any memory access calls the corresponding write or read function that call backs the SystemC simulation, which is outlined in details in the following section.

## IV. QEMU/SystemC Integration

We apply TLMu [2] to combine our QEMU-based TriCore™ emulation with SystemC simulation to a homogeneous virtual prototyping platform for TriCore™-based SoCs. Figure 3 shows an overview of the TLMu architecture. On the right is a SystemC model with several modules (A, B, C) which are connected by a TLM 2.0 bus. The interface to QEMU is wrapped by the TLMu wrapper module, which is connected to the TLM 2.0 bus. The wrapper loads

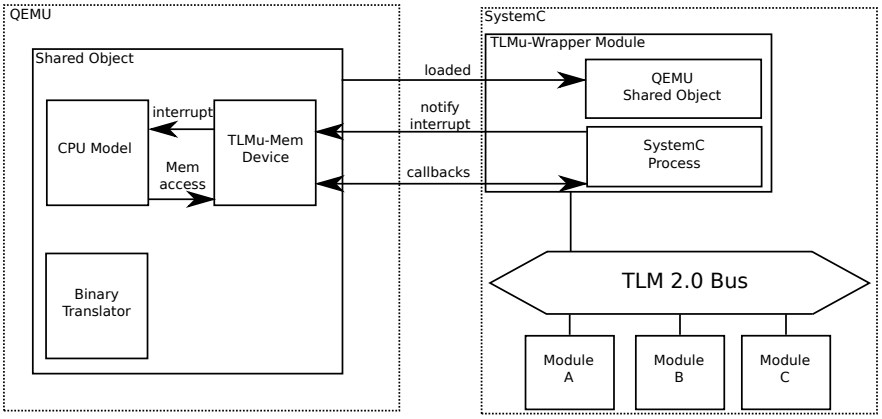


Figure 3: TLMu approach.

QEMU as an shared object and accesses QEMUs' main function by a loaded function pointer from the shared object. The wrapper itself defines callback functions, which are registered at the TLMu memory device and called on every memory access by the QEMU CPU. This allows to switch the control between QEMU and SystemC. Interrupts from any module are sent to the TLMu wrapper module. They are forwarded to the TLMu memory device by a notify function, which prepares QEMU to stop execution of guest code. When the SystemC kernel returns the control to QEMU, it stops the execution of the guest code and starts a method for CPU interrupt handling. This approach allows us to implement service request nodes (SRN) and the corresponding interrupt control unit (ICU) of the TriCore™ architecture either in QEMU as part of the TLMu memory device or by SystemC supporting the communication with the CPU through the wrapper. Note here, that the interrupt handling is not part of TLMu and was implemented by us in the context of our TriCore™ efforts. Note also, that our approach is not limited to one CPU core as the TLMu wrapper module can load and simulate more than one shared object of a QEMU CPU.

## V. Experimental Results

### A. *Functional Correctness*

To evaluate the correctness and performance of the proposed TriCore™ emulator, we compared our QEMU implementation with the TSIM simulator and monitor the state of the emulated CPU before and after the execution of each TriCore™ instruction. We consider our implementation as correct, if the content the TSIM and QEMU registers are equal for each executed TriCore™ instruction. For this comparison, TSIM allows to print every altered register after each executed TriCore™ instruction. For comparison, we extended our QEMU implementation by an equivalent output format. Additionally, we modified the QEMU binary translation by generating a TCG helper after each translated instruction that prints the modified registers. Our test programs finally showed that all QEMU and TSIM processor states are exactly the same.

### B. *Simulation Speed*

For the comparison of the simulation speed between QEMU and TSIM, we implemented two C programs: *Add* and *Fibonacci*. The *Add* program contains on loop of length k and four additions in its body. The *Fibonacci* program recursively computes the nth Fibonacci number. We scaled the execution times by the parameters n and k and measured them with *GNU time*. Figure 4 shows the logarithmically scaled execution times of the test programs on a notebook with an Intel Core2Duo processor where QEMU and TSIM both just use one CPU core of the host. The

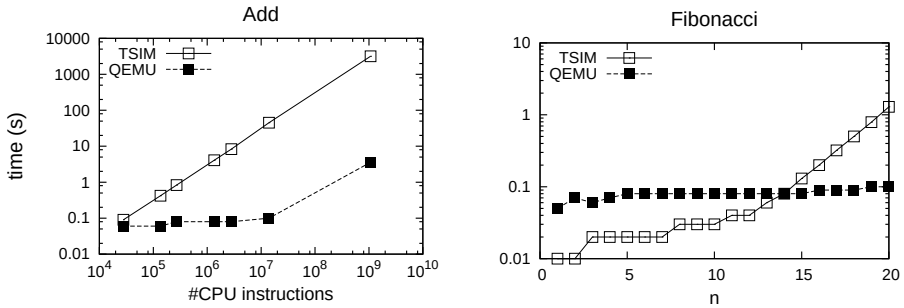


Figure 4: Emulation speed of TSIMs and QEMUs CPU core.

*Fibonacci* program executed faster for $n \leq 14$ on TSIM due to the binary translation overhead for short executions. For $n > 14$ the overhead amortizes and the QEMU execution time just increases very slowly. For long executions both tests showed a proportional difference in speed. We measured a gain of up to 886x on the given hardware.

## VI. Conclusion and Outlook

In this paper we presented our implementation of the TriCore™ ISA for the fast and open QEMU CPU and system emulator. Furthermore, we integrated it into SystemC by means of TLMu for efficient virtual platform modeling. Our first results comparing our implementation with TSIM demonstrate a speed-up of up to 886x with still several potentials for improvements.

Future work will first integrate our TriCore™ implementation into the main QEMU development tree [7]. Thereafter, we will address the implementation of the complete TriCore™ instruction set. Currently, our implementation covers more than 500 TriCore™ instructions where most of the DSP related instructions are not implemented.

## References

[1] Infineon Website. http://www.infineon.com/.

[2] Transaction Level eMulator Website. http://edgarigl.github.io/tlmu/.

[3] TriCore User's Manual V 1.3.8, Januar, 2008.

[4] M. Becker, G. Di Guglielmo, F. Fummi, W. Mueller, G. Pravadelli, and T. Xie. RTOS-Aware Refinement for TLM2.0-based HW/SW Designs. In *DATE '10*, 2010.

[5] M. Becker, U. Kiffmeier, and W. Mueller. HeroeS: Virtual Platform Driven Integration of Heterogeneous Software Components for Multi-Core Real-Time Architectures. In *ISORC 2013*, 2013.

[6] M. Becker, H. Zabel, and W. Mueller. A Mixed Level Simulation Environment for Stepwise RTOS Refinement. In *DIPES'10*, 2010.

[7] QEMU git. http://git.qemu.org/.

[8] M. Gligor, N. Fournel, and F. Pétrot. Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation. In *CODES+ISSS*, 2009.

[9] IEEE. *IEEE Standard SystemC Language Reference Manual – IEEE Std 1666-2005*. IEEE Computer Society, New York, NY, USA, 2006.

[10] Imperas Software. Open Virtual Platforms (OVP). http://www.ovpworld.org/.

[11] Marius Monton. Mixed Simulation Kernels for High Performance Virtual Platforms. 2009.

[12] Synopsys Inc. Synopsys METeor. http://www.synopsys.com/.