RISC-V Timing-Instructions for Open Time-Triggered Architectures

Nithin Ravani Nanjundaswamy
German Aerospace Center
Oldenburg, Germany
nithin.ravaninanjundaswamy@dlr.de
0009-0008-4739-2378

Gregor Nitsche
German Aerospace Center
Oldenburg, Germany
gregor.nitsche@dlr.de
0000-0002-5232-0976

Frank Poppen
*German Aerospace Center**
Oldenburg, Germany
frank.poppen@nxp.com
*now: NXP Semiconductors Germany GmbH, Hamburg

Kim Grüttner
German Aerospace Center
Oldenburg, Germany
kim.gruettner@dlr.de
0000-0002-4988-3858

***Abstract*—Time-triggered architectures (TTAs) were a key enabler for time-predictable software execution and, thus, for cyber-physical and embedded systems with real-time requirements. Controlling software-execution by the means of timer-controlled interrupts and a predetermined schedule, TTAs are a common standard to ensure timing in safety-critical systems. Now, with the emerge of the openly available RISC-V architectures and the use of its instruction-set extension allows to easily provide softcore-processors with an application-specific instruction-set configuration. To support the realtime-capability of such RISC-V based, application-specific instruction-set processors (ASIPs), the presented approach provides timing-instructions as a RISC-V instruction-set extension to measure and control the software execution-time at the hardware-level.**

***Index Terms*—Time-triggered Architecture, RISC-V ISAX, Temporal Behavior, Run-Time Monitoring**

I. Introduction

The complexity of CPS is rising continuously to provide the necessary functionalities and thus requires high processing efficiency along with real-time computing capabilities. Yet, these systems should be cheaper for commercial viability. RISC-V is a free and open-source instruction set architecture (ISA) which is highly configurable with its extensible ISA. The fact that the RISC-V is open source combined with its enormous flexibility makes it highly suitable to build efficient application specific processors for CPS [6]. A time-triggered architecture (TTA) provides a computer architecture for distributed real-time systems in safety-critical applications. TTA in CPS aims to ensure predictable and deterministic behavior of the system based on global time reference. This global time base of known precision at each computation-node, provides a computing infrastructure for the design and implementation of reliable distributed real-time systems, allowing to assure synchronized temporal behavior of their software. Having a computation-node to be a centralized, RISC-V based CPS, the extendability of the RISC-V instruction set architecture (ISA) can be utilized to ensure highly precise timing-measurement and -control with minimal overhead. Enabling time-accurate, low-cost, RISC-V-based TTA systems with extendable instruction set, the timing-extensions are further investigated in this work.

This work was partially supported by the German Federal Ministry of Education and Research (BMBF) in the projects Scale4Edge under Grant 16ME0130 and VE-VIDES under Grant 16ME0247.

For CPS and RTS(real-time systems), the correctness of the system not only depends on the logical correctness of the results but also on time at which these results are produced. For instance, in an adaptive cruise control system (ACC), if a vehicle is detected ahead on the road, ACC should adjust the speed accordingly within stipulated amount of time. Failing to respond within the given deadline will lead to catastrophic outcome. Thus, the temporal behavior of a software is as important as its logical behavior. Verifying correct temporal behavior and ensuring it at run-time becomes challenging as the complexity of the system increases. A prime reason for this is precise run-time measurement and -control of software execution time spans all abstraction layers in computing, including appropriate modeling- and programming-languages, memory hierarchy, pipelining techniques, bus architectures, memory management and task scheduling. The majority of software solutions to temporal requirements rely on programmable timers and interrupt service routines (ISRs) and thus require CPU time and other resources. This leads to additional overhead to the overall system which is difficult to predict and a drawback in resource-constrained systems.

The open-source and extendable ISA of RISC-V has enabled a new era of innovation in processor customization, performance and power optimization. It gives the flexibility to develop a tailor-made processor as per the application requirements. In this work (Section II), the RISC-V ISA is extended by adding new instructions to obtain high-precision cycle-accurate temporal behavior of real-time systems with low overhead. The presented work supports controlling the execution time of a software, unlike the RISC-V trace spec [8], which is primarily focused on software-based monitoring and offline-analysis of execution-times. The work proposes a programming model supported by new custom instructions that measure and control the execution time of real-time software. The work builds on the concepts proposed by Libbla [2] which is a C++ based software, originally developed for ARM core. Libbla provides timing annotations, Estimated Execution Time (EET) and Forced Execution Time (FET), to measure and control temporal behavior of RTS. In Section III, the proposed custom instruction-based timing extensions, which are developed from Libbla timing extensions, are then evaluated against Libbla as well as against a purified Libbla software solution utilizing a

2325-6664/23/$31.00 ©2023 European Union

DOI 10.1109/DSN-W58399.2023.00058

traditional timer-interrupt. Focusing on the resulting instruction density and the hardware area overhead, the timing instructions are compared against the two software solutions. Finally, Section IV concludes and gives an outlook to future work.

II. RISC-V TEMPORAL ISA EXTENSIONS

The functionality of a software is implemented using an ISA provided by the processor. But, neither the software nor the ISA usually have a timing-control or timing-measure role to deliver the result in a reliable amount of time. In fact, if the timing properties are to be guaranteed, with a fixed and predictable delay in computations, designers must reach beneath the abstraction layers which makes the system complex and over-designed [1]. Typically processors implement hardware structures for time measurement (cycle counter, instruction counter) and in a certain sense also for influencing execution times (interrupts for timer/counter, watchdog). The software solutions including Libbla [2] helps to measure and monitor the timing behavior of software-blocks. However, in this approach, the software solution itself becomes a part of the software to be monitored. Thus, these solutions will have it's own influence onto the temporal behavior of the software to be monitored.

The RISC-V ISA standard offers the possibility to extend its instruction set with user-specified instructions. For this work, we have considered a Scala-based project -Murax Soc with a VexRiscV core [3]. The VexRiscV core is built upon the 32-bit ISA of RISC-V and is described using the SpinalHDL library which is also written in Scala. With its plugin-architecture, VexRiscV provides an easy to use and possibility to extend RISC-V core with custom instructions. The project is set-up on an ARTY A7–100 development-board, equipped with Artix-7 FPGA (Field Programmable Gate Array).

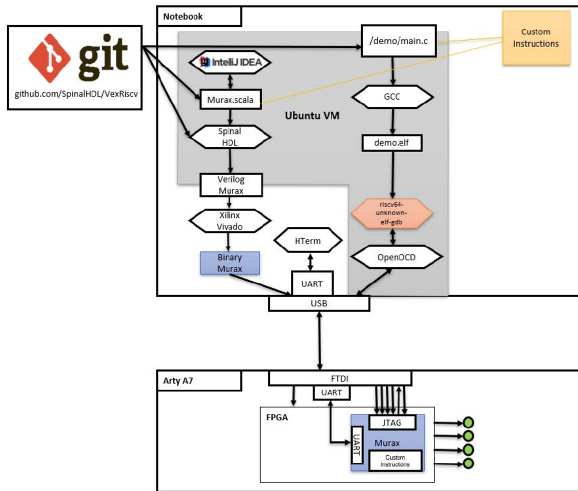

Fig. 1. Implementation flow for Murax SoC and embedded software

Using the plugin-architecture of VexRiscV, the timing instructions are implemented as plugins. The overall implementation-flow of Murax SoC and its embedded software with custom instructions is shown in Fig. 1. The instructions are implemented in Scala and then connected to the Murax SoC through ports. Compilation of Scala project with custom instructions yield RTL files (either VHDL or Verilog), based on which the design is synthesized and the bit stream is generated to program the FPGA.

TABLE I
TIMING INSTRUCTIONS ENCODING

Instruction	funct7	rs2	rs1	funct3	rd	opcode
Measure	0000000	00000	00000	000	00000	0001011
Smonit	0000000	00000	00001	010	00000	0001011
Emonit	0000000	00011	00001	011	00000	0001011

The rightmost part of Fig. 1 represents the flow to compile and execute an embedded firmware which uses the custom instructions that are added to Murax SoC. To this firmware the custom instructions are added as inline assembly. This way, the compiler need not to be modified to detect and compile custom instructions. The GCC compiler compiles the firmware having custom instructions and generates the binary file which is then loaded to the program memory of the Murax through OpenOCD. The results of the firmware can be observed through UART or through LEDs present on the ARTY board. By using this implementation flow, three new instructions – Measure, Smonit and Emonit – are added to VexRiscV to measure and control the temporal behavior of a software. All three instructions are 32-bit, register-type instructions. The binary encoding of these three instructions is given in the Table I. Further details of these instructions are discussed in the following subsections.

A. Timing measure instruction

The measure instruction allows for profiling the temporal software behavior by measuring the execution time of different code-blocks during the design and prototyping phase of software development. It is implemented by using a hardware counter (cycle counter) register which counts the clock cycles of the core. The cycle counter is a 64-bit register and thus can count up-to 2^{64} clock cycles. To measure the execution time of a block of code, the measure instruction has to be executed twice. When the first measure instruction is executed, the current cycle counter value is retained in a register. When the second measure instruction is executed, the difference between the current cycle counter value and the previously retained value is calculated to obtain the exact execution time. In summary, the measure instruction gives the number of clock cycles that have passed since the previous measure instruction was executed.

Fig. 2 shows a sample-code block using the measure-instruction at the beginning and end of the block. To provide the calculated clock cycles to the user, a dedicated UART is implemented inside the measure plugin. This UART is not accessible through software rather its completely controlled by the hardware to reduce the overhead incurred by the measure instruction. The execution time of the code block is determined

```
1  Measure;  //First measure call
2      Sensor signal fusion to detect lead vehicle();
3      Calculate current speed of host and lead car();
4      Calculate distance between host and lead car();
5      Determine cruise mode and desired speed();
6  Measure;  //Second measure call
```

Fig. 2. Sample pseudo code block using measure instruction

```
1  Smonit(2, 'Time_budget_A');  //Start monitoring
2      Sensor signal fusion to detect lead vehicle();
3      Calculate current speed of host and lead car();
4      Calculate distance between host and lead car();
5      Determine cruise mode and desired speed();
6  Emonit(2);                   //End monitoring
```

Fig. 3. Sample pseudo code block using Smonit & Emonit instructions

using the clock cycle information and the processor's clock frequency.

B. Timing control instructions

The timing control-instructions ensure that the software meets a specified timing requirement. After profiling the software using the measure-instruction, the software developer is aware of the time-critical code-blocks and their corresponding execution time. The next step is to ensure that the software always meets this execution time requirement during run-time. The timing control-instructions monitor the software at run-time to verify the timing behavior against its specification. In other words, it provides the functionality of software run-time monitoring. To accomplish this, two new instructions are added to VexRiscV ISA which are Smonit and Emonit.

With the execution time information derived from the measure-block in Fig. 2, the measure-block is transformed into a run-time monitoring block using Smonit and Emonit instructions as shown in Fig. 3. The Smonit instruction is used at the beginning of the code block to start time-monitoring and it requires two parameters - block id: an integer value which uniquely identifies each code block; and deadline: the timing specification to be satisfied by the code-block. Conversely, the Emonit instruction is used at the end of the code-block to indicate the end of monitoring, and requires only the block-id parameter to identify the termination of different blocks. When Smonit instruction is executed, the deadline specification parameter is added to the current cycle counter value and stored in a temporary register (referred as deadline register). When the Emonit instruction is executed at the end of the monitoring block, the current cycle counter value is compared with the deadline register. If the current cycle counter value is less than the deadline register value, then the execution pipeline is halted until the cycle counter reaches the deadline register value. With this approach, the control-instructions make sure that the software code-block always takes the specified time before returning and thus controls the software latency. Inversely, if cycle counter has surpassed the deadline register value, this denotes that the code-block has exceeded the deadline, violating the timing requirements. In this case, a hardware exception is raised and the program flow goes into an exception handler. In the exception handler user-required action will be performed to address the timing violation. New exception code '24', which is reserved for custom use, is written to the mcause-register of RISC-V. This exception code can be read by the software to identify the timing violation and to react to the exception.

Real time embedded systems are typically time-triggered in nature and follow a cyclic execution. This is illustrated in 4 using an ACC system. In such systems, in addition to the overall system's deadline, each subsystem will also have a deadline to meet such that the overall system deadline is greater or equal to the sum of the individual subsystem deadlines.

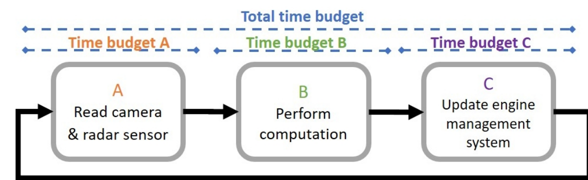

Fig. 4. Time triggered ACC system with cyclic execution

The control instructions must support such systems to monitor the system as a whole and also its subsystems. For this reason, the timing-control instructions provide the feature of blocks nesting as illustrated in Fig. 5 (left). The outermost block is the parent block which monitors the system as a whole, and the inner blocks, also referred as child-blocks, provide the monitoring of the subsystems. Thus, with the help of control instructions nesting, it is possible to guarantee the requirements of both systems and their subsystems.

```
While(true){
Smonit(1, 'Total_Time_budget');
     Smonit(2, 'Time_budget_A');
          Read Camera and Radar sensor signals();
     Emonit(2);
     Smonit(3,'Time_budget_B');
          Sensor signal fusion to detect lead vehicle();
          Calculate current speed of host and lead car();
          Calculate distance between host and lead car();
          Determine cruise mode and desired speed();
     Emonit(3);
     Smonit(4, 'Time_budget_C');
          Update engine management system based on cruise mode();
     Emonit(4);
Emonit(1);
}
```

Fig. 5. Illustration of nesting timing control instructions for the ACC system

III. Evaluation and Results

Predominantly, to measure and control timing requirements of a software, timer/interrupt based software solutions are used. In this section, to evaluate the newly implemented RISC-V ISA

timing extensions, two software based solutions, relying on programmable timer/interrupts, are considered and compared in terms of assembler code overhead and temporal overhead.

The first related software solution is Libbla [2], which is a C++ library for measuring and controlling software execution time, using code-block annotations EET (Estimated Execution Time) and FET (Forced Execution Time) to support time-measurement in the profiling-phase, and thus, to specify the timing-behavior at run-time. EET-blocks provide similar functionality as that of the presented measure-instruction, i.e., to measure the execution time of the code block. FET-blocks provide similar functionality as that of the presented control-instructions, i.e., enforcing the specified timing-behavior of the code-block. The usage of EET and FET block annotations is illustrated in Fig. 6. Originally developed for ARM cores it has been migrated to RISC-V as a part of this work.

```
1 EET{ ← measure annotation                    1 FET('Time_budget_A'){←control annotation
2   Read Camera and Radar sensor signals();    2 Read Camera and Radar sensor signals();
3     }                                        3 }
```

Fig. 6. Libbla-based EET & FET timing annotations of presudo code-blocks

Since original Libbla code is not optimized for performance, but comes with additional features to specify time more conveniently and to allow for further annotations, like Budget Execution Time (BET) blocks, the original Libbla code comes with additional software and timing overhead. Hence, to evaluate the presented measure-instruction in comparison to a more optimized software counterpart, a purified, C-based solution, based on RISC-V performance counters, is considered as the second software solution. The pseudo-code of the C-based solution is given in Fig. 7. Timestamps are obtained at the beginning and end of the code blocks using rdcycle and rdcycleh [4] performance counters. The difference between the time stamps gives the execution time for the code block. Here, 64-bit counters are used since the measure-instruction does also use a 64-bit register as cycle counter.

```
// Read both RiscV performance registers
      asm volatile ("rdcycle & rdcycleh %0" : "=r" (cycle & cycleh));
// combine both 32 Bit values to 64 Bit
     cycle_val= ((cycleh << 32) + cycle);
//software code block to be measured
    asm volatile ("rdcycle & rdcycleh %0" : "=r" (cycle & cycleh));
    cycle_val1= ((cycleh << 32) + cycle);
//Subtract timestamps before & after the code block
     cycle_count = cycle_val - cycle_val;
```

Fig. 7. Pseudo code for C based solution

Table II provides the comparison which is obtained by measuring execution time for the same software code-block with all three approaches. Both in terms of assembler code overhead and temporal overhead, RISC-V timing extensions have lowest overhead, adding 7 lines of assembler code and 20ns temporal overhead to the software block being measured, while original Libbla EET annotation has highest overhead of 1169 lines and 3.002ms. This is explainable as Libbla uses Boost and other external libraries.

TABLE II
CODE OVERHEAD FOR TIMING INSTRUCTIONS

Measurement Block	Code-Overhead	Temporal Overhead
ISA Extension	7	20 ns
Libbla Annotation	1169	3.002 ms
Purified C solution	78	1.89µs

Certainly, the benefit of lower overhead comes at the expense of additional hardware resources. Table III gives details of hardware overhead incurred in terms of Look-Up Tables (LUTs) and Registers for the implementation of the presented timing instructions, to extend the RISC-V ISA.

TABLE III
HARDWARE UTILIZATION FOR TIMING INSTRUCTIONS

Instructions	LUTs	Registers
Measure	189	200
Control	276	225

IV. CONCLUSION AND OUTLOOK

The presented work provides hardware infrastructure in the form RISC-V ISA extension to precisely measure and control the temporal behavior of CPS and RTS. The hardware implementation provides the advantage of cycle accurate timing with lowest run-time overhead on the system unlike comparable software solutions. That way, the control instructions efficiently provide the fundamentals to build time triggered architectures with RISC-V cores. Furthermore, the hardware implementation is less vulnerable to malicious attacks than software, which is more susceptible to them. Hence, the approach approach is attractive to the development of safe and secure CPS and RTS. In the future, the appoach will be extended to automatically deduce code blocks with timing control instructions from the software verification phase such that manual effort will be reduced or removed. For that purpose, tools like the MULTIC-Tooling [7] will be reviewed for integrating timing instructions.

REFERENCES

[1] I. Liu, Isaac Suyu, “Precision timed machines” University of California, Berkeley, 2012.

[2] F. Bruns, I. Yarza, P. Ittershagen and K. Grüttner, “Time Measurement and Control Blocks for Bare-Metal C++ Applications”ACM Trans. Embed. Comput. Syst. 20, 4, Article 34, May 2021.

[3] C. Papon,” , ”VexRiscv git repository,” [Online]. Available: https://github.com/SpinalHDL/VexRiscv. [Accessed 20 October 2022].

[4] ”Instruction Set Manual, Volume I: RISC-V User-Level ISA”.[Online] https://five-embeddev.com/riscv-isa-manual/latest/counters.html.

[5] Kopetz, Hermann and Bauer, Günther, “The time-triggered architecture” Proceedings of the IEEE 91, no. 1 (2003): 112-126.

[6] I. Liu, B. Lickly, H. D. Patel and E. A. Lee, ”Poster Abstract : Timing Instructions - ISA Extensions for Timing Guarantees,” Real-Time Embedded Technology and Applications Symposium (RTAS), 2009.

[7] Forschungsvereinigung Automobiltechnik e.V. “MULTIC-Tooling“ https://www.offis.de/offis/publikation/multic-tooling.html.

[8] Gajinder Panesar, Iain Robertson “Efficient Trace for RISC-V“ https://github.com/riscv-non-isa/riscv-trace-spec/blob/main/riscv-trace-spec.pdf.