Chapter 3

RISC-V Virtual Prototype for GD32V

In this chapter, we present our RISC-V VP for the GD32VF103VBT6 MCU. Our VP configuration is based on and extends the RISC-V VP introduced in [20, 21]. In particular, we have designed our GD32V VP configuration similar to the *SiFive HiFive1 Rev B* platform configuration presented in [21]. We refer to these publications for an in-depth look into the base RISC-V VP. In this thesis, we cover only some parts of the base VP and focus on the GD32V specifics.

Our VP plays an important role in our MT approach presented in the next chapter. The advantages of the VP for our testing framework are discussed in Section 5.3. Here, we first present an overview of the architecture and then discuss the individual components we have added in the subsequent sections.

3.1 Architecture Overview

The GD32V VP is implemented in C++ using SystemC and TLM 2.0. Figure 3.1 provides an overview of the architecture and is an extension of Figure 1 from [21]. The left side represents the input for the VP. The RISC-V GNU toolchain cross-compiles a C or C++ program and optionally links it with the Nuclei SDK and the C/C++ standard library producing an executable RISC-V binary the *Executable and Linkable Format* (ELF). This process is completely independent of the VP, which means that the exact same firmware also runs on the physical GD32V board. To run it on the VP, the binary needs to be loaded into the VP's memory. The architecture of the actual VP is shown in the middle section of the figure.

The centerpiece is a TLM 2.0 bus that connects all the components. Each one is attached to the bus at a specific address range and except for the ISS all components are configured as targets. The ISS is attached to the bus as an initiator via a memory interface. Additionally, the ISS has direct access to the *Enhanced Core Local Interrupt Controller* (ECLIC). This controller is responsible for managing all interrupts. It receives interrupt notifications from the *TIMER* module and from the

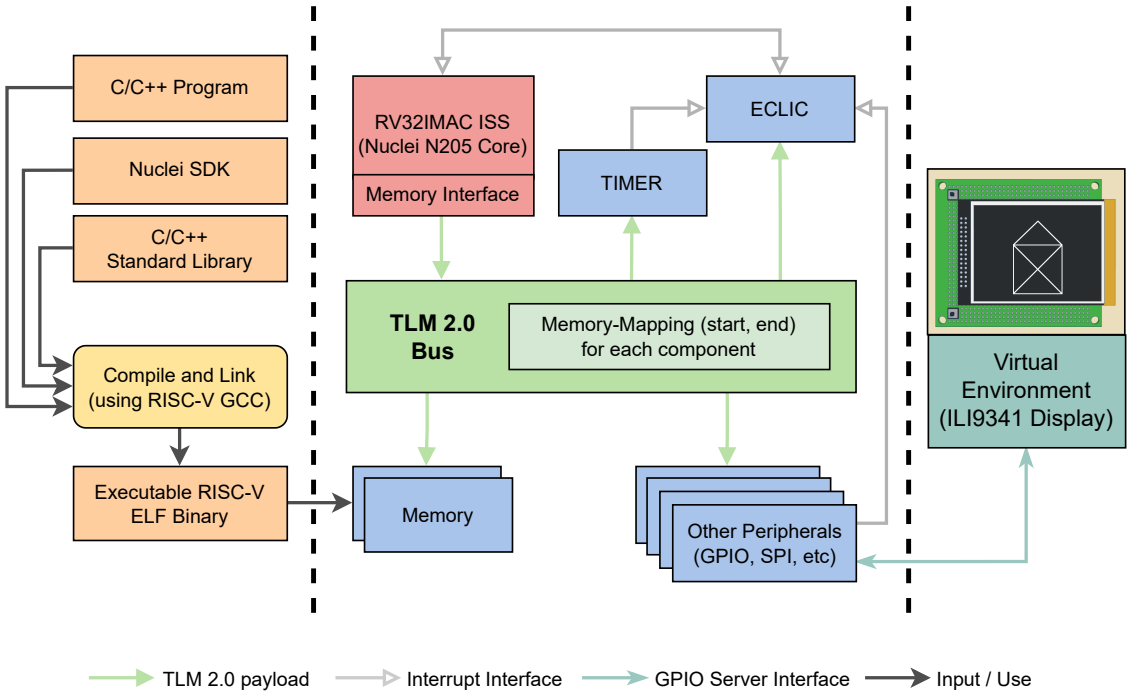


Figure 3.1: Overview of the RISC-V GD32V VP

peripherals. The GPIO peripheral provides a server that enables interaction with the environment. For our setup, we implemented such a virtual environment (shown on the right side of the figure) based on the ILI9341 display.

Overall, the figure — when read from left to right — also represents the general workflow with the VP: We start by developing firmware (left side), which we then execute on the VP (middle), resulting in optional output in the virtual environment (right side).

The individual components of our VP architecture, with the exception of the ISS, are implemented as SystemC modules. In the main file, all components are initiated and their sockets are bound to the bus. Additionally, the ELF file is parsed and loaded into the VP's memory. Finally, the SystemC simulation is started. A detailed description of each component is given in the following sections.

3.2 TLM 2.0 Bus

On the GD32VF103VBT6 MCU, the different components are interconnected via multiple buses. For our VP, we chose to simplify the architecture without sacrificing functionality. Specifically, all components in the VP are connected by a single TLM bus. This bus is responsible for routing transactions between modules. Each module

is assigned a specific non-overlapping address range according to the memory map defined in the GD32VF103VBT6 user manual. The bus routes a transaction as follows: First, the address of the transaction is matched against the defined address ranges, then a translation from the global to the local address range is performed, and finally, the transaction is forwarded to the appropriate target with the local address. Example 3.1 demonstrates this process.

Example 3.1 *The GD32V MCU offers three SPIs: SPI0, SPI1, and SPI2. Consider the SPI0 peripheral which is mapped to the address range (start=`0x40013000`, end=`0x400133FF`). When routing a transaction with the (global) address `0x4001300C`, the bus will translate this to the local address of the SPI0 by subtracting the start address. Therefore, the bus will route the transaction to the SPI0 with the (local) address `0x0C`.*

In total, the bus connects one initiator (the memory interface of the ISS) and 16 targets (including the memory, the interrupt controller, and multiple peripherals). In the main file, the target socket field of each component is bound to an initiator socket of the bus. For the memory interface of the ISS, the connection is reversed, it has an initiator socket field and is bound to a target socket on the bus. For all sockets, we use the blocking transport interface to transfer TLM transactions. We used the existing implementation of [21] for the bus without any changes.

3.3 Instruction Set Simulator for the Nuclei Core

The heart of the VP is the ISS, which simulates the Nuclei N205 core. The main task of the ISS is to load, decode, and execute instructions. It is also responsible for processing interrupts and handling access to the *Control and Status Registers* (CSRs). The ISS itself is not implemented as a SystemC module and is therefore connected to the bus via a memory interface that processes the load and store operations, translates them into transactions, and dispatches them to the bus. The interface actually combines two separate interfaces: A special instruction memory interface that initiates the transactions for loading the instructions from the flash memory and a general data memory interface that handles the transactions for loading and storing data.

The Nuclei N205 core implements the RISC-V ISA and therefore also follows and is compatible with the RISC-V standard definition of the CSRs. However, since the standard only allocates a part of the available address space, vendors can add custom CSRs in unused addresses. This is the case for the Nuclei N205 core. It implements several custom CSRs, the majority of which are related to the interrupt

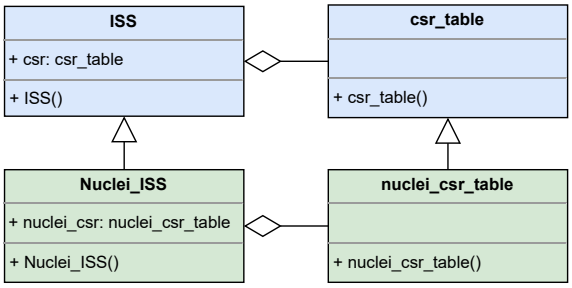


Figure 3.2: ISS Architecture Overview

controller (c.f. next section). Another addition of the Nuclei core is the definition of four sub-modes for the privileged mode to distinguish the exact machine mode status.

To account for these customizations, we had to adapt the existing ISS, which is vendor agnostic and therefore only implements the standard ISA. Figure 3.2 provides a conceptual overview of the architecture, with each box representing a separate C++ class. At the top is the base `ISS` class which has a member of type `csr_table`. The `csr_table` class provides the CSR definitions and handles read and write access to them. The `Nuclei_ISS` class is derived from the `ISS` class and overrides some methods of the base class. It has a member of type `nuclei_csr_table` which is derived from the base `csr_table` and accommodates the Nuclei specific CSRs.

3.4 Interrupt Handling

The Nuclei N205 core uses an ECLIC for handling external interrupts. This controller is based on and extends the proposal for a RISC-V *Core-Local Interrupt Controller* (CLIC) [36]. Additionally, the Nuclei core employs a TIMER unit for handling internal interrupts.

Internal interrupts can be divided into timer and software interrupts. Both are generated by the TIMER unit. For this purpose, the unit provides among others the memory-mapped registers `mtime`, `mtimecmp`, and `msip`. The `mtime` register reflects the system clock, for which the MCU uses an oscillator as a source. In our implementation, we use the SystemC simulation time instead. The `mtimecmp` register can be set by the software and as soon as the value of `mtime` is greater than or equal to the value of `mtimecmp`, a timer interrupt is generated. Software interrupts are generated when the `msip` register is set to 1. As Figure 3.1 indicates, all internal interrupts are forwarded to the ECLIC. In addition to the interrupts from the TIMER unit, the ECLIC unit also receives external interrupts generated by the other components of the MCU.

To allow the various components to send an interrupt signal to the ECLIC, we define an interface called `nuclei_interrupt_gateway` which specifies two methods, namely `void gateway_trigger_interrupt(uint32_t irq_id)` and `void gateway_clear_interrupt(uint32_t irq_id)`. This interface serves as an abstraction for the ECLIC unit. Each component that sends interrupt signals contains a file of type `nuclei_interrupt_gateway` that points to the ECLIC unit. To send an interrupt the component calls the `gateway_trigger_interrupt` method on the pointer with the interrupt ID as an argument.

The ECLIC implements the `nuclei_interrupt_gateway` interface. The controller is responsible for collecting and prioritizing interrupts. To facilitate prioritization, each interrupt is assigned a level and a priority in addition to the ID. We have modeled prioritization in our VP by storing all pending interrupts in a priority queue. When a VP component calls the `gateway_trigger_interrupt` method with the interrupt ID as an argument, the interrupt is added to the priority queue using the custom comparator shown in Listing 3.1. Essentially, the interrupt that first returns a higher value when checking level, priority, and ID in that particular order will be queued first.

```
1 class InterruptComparator {
2    public:
3     inline bool operator()(const Interrupt& a, const Interrupt& b) const {
4         return (b.level > a.level) ||
5                 (b.level == a.level && b.priority > a.priority) ||
6                 (b.level == a.level && b.priority == a.priority &&
7                     b.id > a.id);
8     }
9 };
```

Listing 3.1: Comparator for Interrupts

An interrupt is handled by the Nuclei ISS as follows: After each instruction, the ISS checks if an interrupt is pending. If so, the first interrupt in the priority queue is processed. If the interrupt is configured as *vectored*, the ISS jumps directly to the interrupt handler method. For *non-vectored* interrupts, the ISS jumps to a common base address. There the context, i.e., several CSRs and general purpose registers are stored on the stack before jumping to the interrupt handler method. Upon return from the handler method, the context is restored.

When multiple consecutive interrupts occur, saving and restoring the context introduces unnecessary overhead. Therefore, the Nuclei core supports interrupt tail-chaining, which works as follows: Suppose the ISS is processing an interrupt, and a second interrupt with a lower level than the current one occurs. When the ISS re-

turns from the handler method of the first interrupt, it does not restore the context. Instead, it jumps directly to the handler method of the second interrupt. Only after returning from this method is the context restored. This back-to-back interrupt handling is implemented by the custom `jalmnxti` CSR.

The Nuclei core also supports interrupt preemption, i.e., the nesting of multiple interrupts. If a new higher-level interrupt occurs while an interrupt is being handled, the ISS stops handling the current interrupt and handles the new interrupt. Once the new interrupt is fully processed, the ISS returns to the previous interrupt. This preemption feature allows not only two but multiple levels of nesting.

Since the existing ISS relies on a combination of a Platform-Level Interrupt Controller (PLIC) [37] and a Core Local Interruptor (CLINT), we had to make some modifications to allow handling interrupts with the ECLIC.

3.5 Peripherals

The GD32VF103VBT6 MCU includes several memory-mapped peripherals. Most of them are assigned 1KB of address space. However, we implemented only the peripherals necessary to run basic firmware and the TFT_eSPI library, since the primary purpose of our VP was to facilitate our MT approach. With the exception of the *Reset and Clock Unit* (RCU) and the *Universal Synchronous/Asynchronous Receiver/Transmitter* (USART), all implemented peripherals are connected to the GPIO complex as illustrated by Figure 3.3. To simplify the figure we have omitted the TLM 2.0 bus, to which each peripheral is attached separately (c.f. Figure 3.1). In the following, we describe the connections in detail and for each peripheral provide some implementation details as well as a brief description based on the GD32V MCU manual.

GPIO

The GPIO peripheral consists of five ports with 16 pins each, for a total of 80 pins. A single GPIO SystemC module models one port, therefore, in the main file, we instantiate five GPIO instances. The GPIO pins allow the MCU to interact with the environment by configuring a pin either as input, output, or alternate function. The alternate function configuration allows pins to be used for other peripherals (e.g., *Serial Peripheral Interface* (SPI) or *External Memory Controller* (EXMC)). In order for the VP to be able to interact with a virtual environment, the GPIO module integrates the GPIO server introduced in [21]. For detailed information on the server, we refer to [25]. Here, we only explain the basic concept: The server allows a client to connect to it using a TCP connection. Depending on the pin configuration, the client

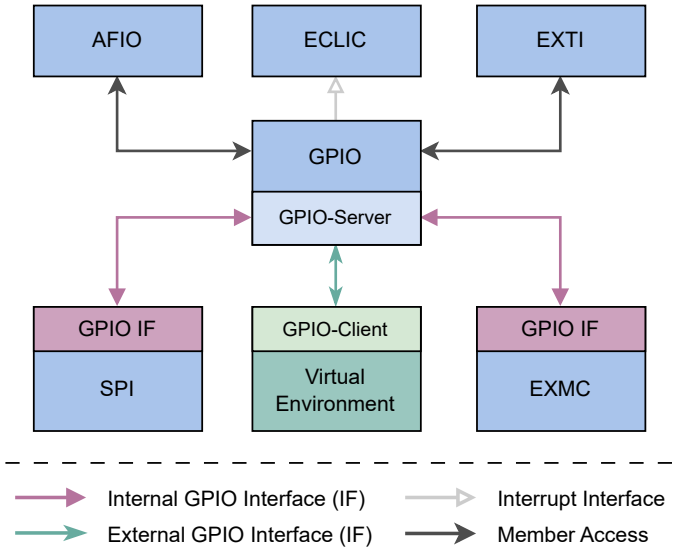


Figure 3.3: GD32V Peripherals

can then read or write individual pins. Note that each GPIO instance integrates its own server instance, so there are a total of five servers running within the VP.

In addition, the server provides an interface for the SPI peripheral to send and receive data frames. Similar to the SPI module, the EXMC peripheral also uses GPIO pins and therefore requires an interface in the GPIO server. Since this was not provided by the existing implementation, we adapt the server in this respect. More details about the SPI and EXMC peripherals are provided below.

For pins configured as input, we can enable interrupts. When the client changes the state of a pin for which interrupts are enabled, the server notifies the GPIO module, which sends an interrupt signal to the ECLIC. The GPIO peripheral also relies on the *External Interrupt/Event* (EXTI) and *Alternate Function Input/Output* (AFIO) peripherals to handle interrupts. These are described below.

EXTI

The EXTI controller is used to generate interrupts for the GPIO pins. It consists of 19 independent edge detectors, 16 of which are used for GPIO (the remaining three are reserved for the Low Voltage Detector (LVD), the Real-Time Clock (RTC), and USB Wakeup). Each input line combines five pins, one from each of the five GPIO ports. For example, line0 combines the first pin of each GPIO port. Selecting which of the five pins to use is done by configuring the registers in the AFIO peripheral (see below). The EXTI controller has several memory-mapped registers that allow configuring each interrupt line, e.g., enabling and disabling a line or setting the type of edge detection (falling, rising, or both).

The implementation of the EXTI controller in our VP differs slightly from the real hardware, but the functionality is not affected. In particular, the EXTI controller is still an independent SystemC module with its own registers, but the interrupt generation is integrated into the GPIO module. The GPIO module can access the registers of the EXTI controller via a pointer. When the server notifies the GPIO module of a pin state change, the GPIO module checks the corresponding EXTI controller and AFIO registers and triggers an interrupt notification to the ECLIC accordingly.

AFIO

The AFIO peripheral is tightly coupled to the GPIO peripheral. The GPIO pins of the GD32VF103VBT6 MCU are pin-shared, i.e., they can be used as alternative functional pins for other peripherals. These alternative functions can be enabled by setting the corresponding AFIO configuration registers. Additionally, the AFIO peripheral provides source selection registers to select the input for a given EXTI line. In other words, which GPIO pin of an EXTI line should be used for interrupts.

We implemented the AFIO peripheral similar to the EXTI controller, i.e., an independent SystemC module with its own registers. In the GPIO module, we used a pointer to access the registers of the AFIO peripheral. As explained in the EXTI section, when the GPIO server registers a change of state of a pin, we check whether interrupts are enabled for the corresponding EXTI line and whether the respective pin is set as the source for this line. If this is the case an interrupt notification is sent to the ECLIC.

SPI

An important peripheral is the SPI which allows synchronous serial communication with external devices. The SPI module of the GD32VF103VBT6 MCU contains a separate transmit and receive buffer each 16 bits wide and supports a data frame size of 8 or 16 bits. By writing to the SPI's data transfer register we can store data in the transmit buffer and by reading the register we can retrieve data from the receiver buffer.

In our VP implementation, writing to the data transfer register does not store the data in a buffer, but transmits it directly using the GPIO server. To this end, the GPIO module provides an interface for the SPI to interact with a virtual external device. A data byte is transferred by calling the transfer method of this interface with the byte as an argument. The transfer method returns the response of the external device which is then stored in the receive buffer. This buffer is implemented using

a queue. Since transmitting is done directly we omitted the transmit buffer in our implementation.

EXMC

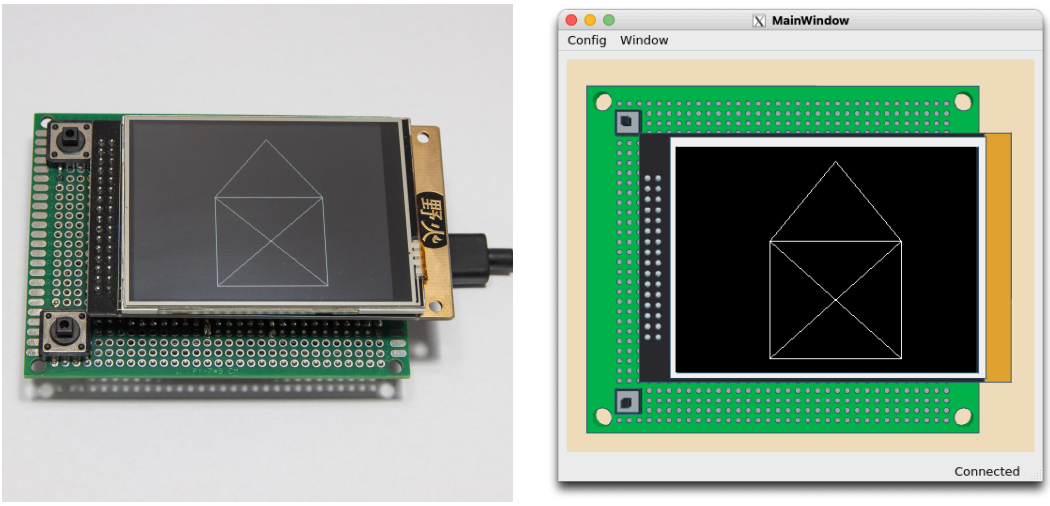
The EXMC (often also referred to as Flexible Static Memory Controller (FSMC)) allows the MCU to access external memory by translating the bus transactions into the appropriate external device protocol. In contrast to the other peripherals, this module is connected to the bus at two address regions. The first one (`0xA0000000` to `0xA0000FFF`) is used for the configuration registers of the EXMC. The second one (`0x60000000` to `0x6FFFFFFF`) is used for accessing the external memory.

To implement the EXMC, we used a SystemC module with two separate target sockets, one for each address range, and implemented a separate transport method for each socket. The transport method for the first address range writes and reads the configuration registers, and the method for the second range reads from and writes to the external memory. Since any external memory, as the name implies, is not part of the VP but rather part of the virtual environment, we again used the GPIO server for communication. However, the existing server implementation does not provide an interface for the EXMC peripheral, so we have added one similar to the SPI interface.

The EXMC peripheral plays a vital role in our MT approach. We used it to connect the graphical RAM of the ILI9341 display to the GD32VF103VBT6 MCU. This allowed us to communicate with the display using the faster 16-bit parallel interface instead of the slower SPI.

RCU

The functionality of the RCU is twofold: (1) it allows resetting parts of the system or the entire system, and (2) it provides ways to manage different clocks for the system. The latter is essential for running firmware on the MCU since the RCU contains a control register that indicates when the oscillators — used for generating the clock signals — are stable. The Nuclei SDK is configured in such a way that the `main` function of a firmware is not called until the corresponding bits in the control register are set to 1, indicating stable oscillators. Instead of oscillators, the VP uses the SystemC simulation time as a clock signal, which naturally does not suffer from frequency variations. Therefore, in our RCU implementation, the corresponding bits in the control register are hardcoded to simulate always stable oscillators.



(a) Real (b) Virtual

Figure 3.4: ILI9341 Display Environment

USART

The USART in the GD32VF103VBT6 MCU is very versatile and offers a wide range of functions. However, since this peripheral was not used in our MT approach we only implemented write access to the data register for debugging purposes. In particular, we redirected data written to the data register to the standard output of the host system.

3.6 Virtual Environment

In the previous section, we described the GPIO server integrated in the GPIO module which allows connecting a virtual environment. For this project, we created such an environment for the ILI9341 display. The physical display (attached to a custom shield for the GD32V board) is shown in Figure 3.4a and our corresponding virtual model can be seen in Figure 3.4b. Both displays show the ”Haus vom Nikolaus” example from the introduction, and since our VP is binary compatible with the GD32VF103VBT6 MCU, we were able to generate both images using the exact same firmware.

The virtual environment is implemented in C++ using the Qt framework. It integrates five GPIO clients, each of which connects to one of the five GPIO servers in the VP. Similar to the server, we adapted the existing client implementation to support communication via the EXMC peripheral. For the graphical RAM of the display, we used a variable of type `QImage`. Read and write access to the graphi-

cal RAM via the EXMC interface is translated to reading from and writing to the `QImage` variable.

In addition to the commands necessary for drawing on the display, our implementation of the display also includes a special command (`0xFF`) for taking screenshots. It can be sent like any other display command using the TFT_eSPI's `wirtecommand` method. When the virtual environment receives this command, the current content of the `QImage` variable is stored on the host system as an RGBA image in PNG file format. The screenshot command allows the resulting file to be numbered. Specifically, the file name consists of the prefix "screenshot_" followed by a number sent with the command.

The display we used in our project includes a touch panel that is controlled by the XPT2046 touch controller. Although this functionality is not required for our MT approach, we still support it in the virtual environment to get a more accurate display model. The touch controller uses the SPI to transmit coordinates. However, these coordinates are not calibrated, i.e., they do not align with the coordinate system which is used for drawing on the display. For example, when touching the physical display at position (10,20) the touch controller transmits the position (405,583). Therefore, when the firmware receives coordinates from the controller, it must first calibrate them.

In our virtual environment, the coordinates of a mouse click can be accessed using a `QMouseEvent` object provided by the Qt framework. However, these coordinates are already calibrated. For example, if we click at the position (10,20) on our virtual display, the `QMouseEvent` object returns that exact position. So to be binary compatible with the physical device, we need to decalibrate these coordinates before sending them to the VP.

List of Acronyms

AFIO Alternate Function Input/Output

CLIC Core-Local Interrupt Controller

CSR Control and Status Register

ECLIC Enhanced Core Local Interrupt Controller

ELF Executable and Linkable Format

EXMC External Memory Controller

EXTI External Interrupt/Event

GPIO General Purpose Input/Output

ISA Instruction Set Architecture

ISS Instruction Set Simulator

MCU Microcontroller Unit

MR Metamorphic Relation

MTC Metamorphic Test Case

MT Metamorphic Testing

RCU Reset and Clock Unit

RTL Register Transfer Level

SPI Serial Peripheral Interface

SUT System under Test

TLM Transaction Level Modelling

USART Universal Synchronous/Asynchronous Receiver/Transmitter

VP Virtual Prototype

Bibliography

[1] Elaine J. Weyuker. 1982. On Testing Non-Testable Programs. *The Computer Journal*, 25, 4, (November 1982), 465–470. ISSN: 0010-4620. DOI: 10.1093/comjnl/25.4.465.

[2] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41, 5, 507–525. DOI: 10.1109/TSE.2014.2372785.

[3] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering*, 42, 9, 805–824. DOI: 10.1109/TSE.2016.2532875.

[4] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.*, 51, 1, Article 4, (January 2018), 27 pages. ISSN: 0360-0300. DOI: 10.1145/3143561.

[5] Tsong Yueh Chen, Shing-Chi Cheung, and Siu-Ming Yiu. 1998. Metamorphic Testing: A New Approach for Generating Next Test Cases. Technical report HKUST-CS98-01. Department of Computer Science, The Hong Kong University of Science and Technology.

[6] Wing Kwong Chan, Shing Chi Cheung, and Karl RPH Leung. 2007. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research (IJWSR)*, 4, 2, 61–81.

[7] Chang-ai Sun, Guan Wang, Baohong Mu, Huai Liu, ZhaoShun Wang, and T.Y. Chen. 2011. Metamorphic Testing for Web Services: Framework and a Case Study. In *2011 IEEE International Conference on Web Services*, pp. 283–290. DOI: 10.1109/ICWS.2011.65.

[8] Zhi Quan Zhou, Shaowen Xiang, and Tsong Yueh Chen. 2016. Metamorphic Testing for Software Quality Assessment: A Study of Search Engines. *IEEE Transactions on Software Engineering*, 42, 3, 264–284. DOI: 10.1109/TSE.2015.2478001.

[9] Christian Murphy, M. S. Raunak, Andrew King, Sanjian Chen, Christopher Imbriano, Gail Kaiser, Insup Lee, Oleg Sokolsky, Lori Clarke, and Leon Osterweil. 2011. On Effective Testing of Health Care Simulation Software. In *Proceedings of the 3rd Workshop on Software Engineering in Health Care (SEHC '11)*. Association for Computing Machinery, Waikiki, Honolulu, HI, USA, pp. 40–47. ISBN: 9781450305853. DOI: 10.1145/1987993.1988003.

[10] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, and Shengqiong Wang. 2009. Conformance Testing of Network Simulators Based on Metamorphic Testing Technique. In *Formal Techniques for Distributed Systems*. David Lee, Antónia Lopes, and Arnd Poetzsch-Heffter, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 243–248. ISBN: 978-3-642-02138-1.

[11] Christian Murphy, Gail E Kaiser, and Lifeng Hu. 2008. Properties of machine learning applications for use in metamorphic testing. DOI: 10.7916/D8XK8PFD.

[12] Xiaoyuan Xie, Joshua W.K. Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84, 4, 544–558. The Ninth International Conference on Quality Software. ISSN: 0164-1212. DOI: 10.1016/j.jss.2010.11.920.

[13] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. *SIGPLAN Not.*, 49, 6, (June 2014), 216–226. ISSN: 0362-1340. DOI: 10.1145/2666356.2594334.

[14] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. 2010. An Automatic Testing Approach for Compiler Based on Metamorphic Testing Technique. In *2010 Asia Pacific Software Engineering Conference*, pp. 270–279. DOI: 10.1109/APSEC.2010.39.

[15] Andrew Waterman and Krste Asanovic. 2019. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*. EECS Department, University of California. Berkeley.

[16] Andrew Waterman and Krste Asanovic. 2021. *The RISC-V instruction set manual, volume II: Privileged architecture*. EECS Department, University of California. Berkeley.

[17] Tom De Schutter. 2014. *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press.

[18] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer.

[19] 2012. IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, 1–638. DOI: 10.1109/IEEESTD.2012.6134619.

[20] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2018. Extensible and Configurable RISC-V Based Virtual Prototype. In *2018 Forum on Specification & Design Languages (FDL)*, pp. 5–16. DOI: 10.1109/FDL.2018.8524047.

[21] Vladimir Herdt, Daniel Große, Pascal Pieper, and Rolf Drechsler. 2020. RISC-V based virtual prototype: An extensible and configurable platform for the system-level. *Journal of Systems Architecture*, 109, 101756. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2020.101756.

[22] Anton Shilov. 2019. Western Digital Rolls-Out Two New SweRV RISC-V Cores For Microcontrollers. Retrieved 03/19/2023 from https://www.anandtech.com/show/15231/western-digital-rollsout-two-new-swerv-riscv-cores.

[23] Dave Kleidermacher, Jesse Seed, Brandon Barbello, and Stephan Somogyi. 2021. Pixel 6: Setting a new standard for mobile security. Retrieved 03/19/2023 from https://security.googleblog.com/2021/10/pixel-6-setting-new-standard-for-mobile.html.

[24] John Aynsley. 2009. *OSCI TLM-2.0 language reference manual*. Open SystemC Initiative.

[25] Pascal Pieper, Vladimir Herdt, and Rolf Drechsler. 2022. Advanced Embedded System Modeling and Simulation in an Open Source RISC-V Virtual Prototype. *Journal of Low Power Electronics and Applications*, 12, 4. ISSN: 2079-9268. DOI: 10.3390/jlpea12040052.

[26] John Ahlgren, Maria Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Erik Meijer, Silvia Sapora, and Justin Spahr-Summers. 2021. Testing Web Enabled Simulation at Scale Using Metamorphic Testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 140–149. DOI: 10.1109/ICSE-SEIP52600.2021.00023.

[27] Darryl C. Jarman, Zhi Quan Zhou, and Tsong Yueh Chen. 2017. Metamorphic Testing for Adobe Data Analytics Software. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, pp. 21–27. DOI: 10.1109/MET.2017.1.

[28] T.H. Tse and S.S. Yau. 2004. Testing context-sensitive middleware-based software applications. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.* 458–466 vol.1. DOI: 10.1109/CMPSAC.2004.1342879.

[29] Fei-Ching Kuo, Tsong Yueh Chen, and Wing K. Tam. 2011. Testing embedded software by metamorphic testing: A wireless metering system case study. In *2011 IEEE 36th Conference on Local Computer Networks*, pp. 291–294. DOI: 10.1109/LCN.2011.6115306.

[30] Mingyue Jiang, Tsong Yueh Chen, Fei-Ching Kuo, and Zuohua Ding. 2013. Testing Central Processing Unit scheduling algorithms using Metamorphic Testing. In *2013 IEEE 4th International Conference on Software Engineering and Service Science*, pp. 530–536. DOI: 10.1109/ICSESS.2013.6615365.

[31] Johannes Mayer and Ralph Guderlei. 2006. On Random Testing of Image Processing Applications. In *2006 Sixth International Conference on Quality Software (QSIC'06)*, pp. 85–92. DOI: 10.1109/QSIC.2006.45.

[32] W. K. Chan, Jeffrey C. F. Ho, and T. H. Tse. 2010. Finding failures from passed test cases: improving the pattern classification approach to the testing of mesh simplification programs. *Software Testing, Verification and Reliability*, 20, 2, 89–120. DOI: https://doi.org/10.1002/stvr.408.

[33] Fei-Ching Kuo, Shuang Liu, and T. Y. Chen. 2011. Testing a Binary Space Partitioning Algorithm with Metamorphic Testing. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (*SAC '11*). Association for Computing Machinery, TaiChung, Taiwan, pp. 1482–1489. ISBN: 9781450301138. DOI: 10.1145/1982185.1982502.

[34] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.*, 1, OOPSLA, Article 93, (October 2017), 29 pages. DOI: 10.1145/3133917.

[35] Frank Riese, Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2021. Metamorphic Testing for Processor Verification: A RISC-V Case Study at the Instruction Level. In *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 1–6. DOI: 10.1109/VLSI-SoC53125.2021.9606997.

[36] Core-Local Interrupt Controller (CLIC) RISC-V Privileged Architecture Extensions. Retrieved 03/12/2023 from https://github.com/riscv/riscv-fast-interrupt/blob/master/clic.pdf.

[37] RISC-V Platform-Level Interrupt Controller Specification. Retrieved 03/12/2023 from https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic-1.0.0.pdf.

[38] Huai Liu, Xuan Liu, and Tsong Yueh Chen. 2012. A New Method for Constructing Metamorphic Relations. In *2012 12th International Conference on Quality Software*, pp. 59–68. DOI: 10.1109/QSIC.2012.10.