Minimally Invasive HW/SW Co-debug Live Visualization on Architecture Level

Pascal Pieper
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
pascal.pieper@dfki.de

Ralf Wimmer
Concept Engineering GmbH
Freiburg im Breisgau, Germany
Albert-Ludwigs-Universität Freiburg
Freiburg im Breisgau, Germany
ralf@concept.de

Gerhard Angst
Concept Engineering GmbH
Freiburg im Breisgau, Germany
gerhard@concept.de

Rolf Drechsler
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
Institute of Computer Science, University of Bremen
Bremen, Germany
drechsle@informatik.uni-bremen.de

ABSTRACT

We present a tool that allows developers to debug hard- and software and their interaction in an early design stage. We combine a SystemC virtual prototype (VP) with an easily configurable and interactive graphical user interface and a standard software debugger. The graphical user interface visualizes the internal state of the hardware. At the same time, the software debugger monitors and allows to manipulate the state of the software. This co-visualization supports design understanding and live debugging of the HW/SW interaction. We demonstrate its usefulness with a case-study where we debug an OLED display driver running on a RISC-V VP.

CCS CONCEPTS

• **Hardware → Simulation and emulation**.

KEYWORDS

RISC V, debugging, virtual prototype, visualization

ACM Reference Format:
Pascal Pieper, Ralf Wimmer, Gerhard Angst, and Rolf Drechsler. 2021. Minimally Invasive HW/SW Co-debug Live Visualization on Architecture Level. In *Proceedings of the Great Lakes Symposium on VLSI 2021 (GLSVLSI '21), June 22–25, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

1 INTRODUCTION

Virtual prototypes (VPs) [5, 6] are an important tool for hardware/-software co-design. A VP is typically a transaction-level model (TLM), written in a high-level language like SystemC [7], which abstracts from implementation details of the hardware. It models the hardware to a level of detail such that it can execute software that is supposed to run later on the developed hardware. This way, VPs allow to write software for a target system before the actual hardware is finalized and produced, resulting in a shorter time-to-market. Additionally, it also enables effective debugging early in the design process, in particular of the often complex interplay between hard- and software.

In this paper, we present an easily configurable graphical debugging tool called RISCview. Its architecture is sketched in Fig. 1. RISCview features a graphical user interface (GUI) that shows abstract views of the (virtual) hardware. They are rendered automatically using the industrial-strength drawing engine Nlview™ [1]. The schematics are annotated with live simulation data, i. e., the current values of signals, busses, and registers while executing software instructions. These annotations can include internal values that are not accessible from the hardware's interface via software instructions, but still important for debugging.

The HW debugging GUI is connected via TCP to the executable consisting of the virtual prototype and user-defined views of the hardware structure. For establishing the connection to the debugging GUI, RISCview provides a visualization interface that is linked into the VP executable. This model/view scheme separates the VP and the displayed information, minimizing the impact of adding our framework to a virtual prototype. It also reduces the interference

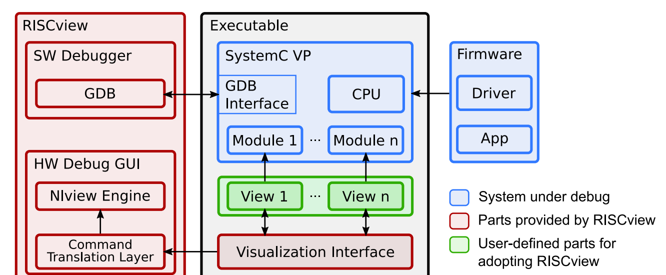

Figure 1: Architecture of RISCview (in red) together with a system under debug (in blue). Highlighted in green are the user-defined parts that are necessary for the adoption.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
GLSVLSI '21, June 22–25, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8393-6/21/06.
https://doi.org/10.1145/XXXXXX.XXXXXX

with other automated testing systems. Additionally, we can flexibly highlight areas of interest by dynamic reconfiguration of the views without modifying the VP.

The VP exhibits a debugging interface such that a standard software debugger like GDB [10] can be used to inspect and manipulate the internal state of the software that is currently executed on the VP. It allows to monitor variables, to set breakpoints etc.

The combination of the HW debugging GUI with a GDB instance for the executed software gives the user deep insights into the interplay between HW and SW. RISCview can be used, e. g., to aid the integration process of new peripherals and matching software drivers into VPs, to visualize the existing architecture at run-time, and to analyze interrupt and timer correlations.

We evaluate our tool by debugging a hardware abstraction layer (HAL) for a newly designed OLED-Screen shield for the RISC-V processor board HiFive1[1]. Our experience shows clearly that RISCview allows to find bugs in the HW/SW interaction more efficiently than the available alternatives.

Related Work. While debugging tools for later design phases exist both on the software side and on the hardware side, a lucid and easy-to-use hardware visualization tool for early virtual prototypes is not available yet. For instance, [8, 13] offer debugging tools for systems on a chip (SoCs) at gate level later in the design process. [13] emulates CPU and IPs by implementing a GDB interface to an FPGA simulator, while [8] proposes a debug controller that can be integrated in SoCs on the final silicon.

Both Rogin et al. [9] and Große et al. [3] propose SystemC IDEs for low-level interactions with a focus on the signal layer. These IDEs are incompatible with transaction-level models and do not offer a live view of the system at run-time.

Since a virtual prototype is a software implementation – in our case using the C++ class library SystemC –, it is also possible to attach a software debugger like GDB [10] directly to the virtual prototype and to step this way through the software model of the hardware logic. Compared to our tool, this approach has the severe drawback that it shows the variables of the SystemC implementation, but not a direct view of the modeled hardware; not to mention of the software that is running on the VP.

In summary, the existing solutions are either too late in the design process, provide no live view, or are not appropriate for debugging the hardware/software interaction.

Main contributions of this paper are:

(1) an implementation-agnostic HW/SW visualization,
(2) early visual debug parallel to existing software tools,
(3) a live view of the system's state during the debugging session,
(4) a case-study showing that our approach is well suited for finding bugs in the interaction of hard- and software.

Organization of the paper. In the following section we introduce the building blocks of our tool and the relevant concepts for our case-study. Then we go into details of RISCview's architecture and implementation in Sect. 3. Section 4 presents our case-study. Finally, we summarize our contributions with an outlook to future work in Sect. 5.

[1] https://www.sifive.com/boards/hifive1

2 PRELIMINARIES

Here we explain the core concepts used in our tool and the case-study.

SystemC. SystemC [7] is a virtual prototyping framework for C++. It offers a class library to model hardware systems with modules and ports in an event-driven simulation kernel. The main benefit of SystemC is the flexible trade-off between timing accuracy and simulation time, operating from abstract transaction-level modeling (TLM) down to the register transfer level (RTL). This support for multiple abstraction levels enables developers to refine the design and even re-use the VP to verify the final hardware [2, 12].

TLM. SystemC TLM is a mechanism to speed up the simulation time for the penalty of reduced timing accuracy (although retaining cycle-accuracy is possible [11]).

RISC-V Instruction Set Architecture. For our case-study we extended the open-source RISC-V VP [4] in its *HiFive* mode. This mode emulates the tinkering board HiFive1 of the company SiFive. The processor board comes with peripherals such as buffered SPI, DMA, and UART, which are all modeled in the virtual prototype. The VP offers two ways to debug the system: A GDB connection to the simulated CPU (software side) and a GDB session over the SystemC executable itself (hardware side). While it is possible to access the hardware IPs through a GDB session, the effort to gain information of interest is disproportionate because one has to access variables through the SystemC kernel with its user-space scheduling. The software GDB module inside the VP, however, is usable as if the RISC-V binary was executed locally.

SPI. In the case-study, we use the Serial Peripheral Interface. This protocol operates on three or four wires for data transmission between a master device and one or multiple slave devices. The bus master starts a transmission by activating the *Chip Select* (CS) line of the target device, starting a clocking signal on the CLK line in sync with its *Master Out/Slave In* (MOSI) line. Eight bits can be transferred for each burst. Depending on the use-case, the *Master In/Slave Out* (MISO) line may be used to transmit data from the slave fully duplex. The master device has to actively poll slaves if no additional bit lines are used.

Nlview. Nlview™ [1] is a commercial state-of-the-art library by Concept Engineering GmbH for creating schematic diagrams for electronic systems at different abstraction levels, ranging from transistor level via gate and RTL-level to system level. It is compatible with different GUI frameworks like Tcl/Tk, Qt, WxWidgets, and HTML5 canvas. Nlview provides APIs in C, Tcl, Java, Perl, and Python. The automatically generated schematic layout can be modified and controlled both by the APIs and by human intervention. Interactive circuit exploration is supported by Nlview's incremental schematic generation technology.

RISCview uses the Nlview Tcl/Tk widget to render views of the hardware modeled in the VP together with simulation data (see Fig. 5 for an example view). Nlview's incremental navigation features thereby allow to interactively explore the hardware views and hide irrelevant parts.

3 IMPLEMENTATION

To extend an existing SystemC VP, the system designer needs to add *views* for every IP module that shall be a part of the visualization. Views are abstract representations of modules containing the relevant information with high control over the module's layout. These representations act independently of the actual SystemC behavior, separating the view from the model as much as possible. The views are automatically collected by the *visualization interface*. The visualization interface translates the instantiated views and their data into a live stream of commands for the debugging GUI via TCP. During the simulation of the SystemC VP, the interface extracts updated information via the registered views asynchronously.

In the GUI, the *command translation layer* receives the commands from the visualization interface and generates appropriate API calls of the visualization engine to render the model in a graphical representation. This additional translation layer offers the flexibility of using different visual styles or levels of detail. Adding other visualization engines requires only implementing a different translation layer. In our case study, we chose the industry-proven Nlview engine [1] that allows creating structure components and connections via Tcl/Tk commands. The combination allows an interactive exploration of the underlying model, offering an auto routing of individual nodes and a partial exploration to limit the view to the relevant parts at run-time.

As already said, there are two GDB interfaces that can be used simultaneously: A GDB session of the simulated CPU (software side) and the SystemC executable itself (hardware side). The RISC-V binary can be loaded with GDB as a remote target to the SystemC VP. The virtual CPU inside the VP then can be halted with breakpoints and the virtual memory can be explored. Additionally, the actual VP including its numerous IP models are written in C/C++ and thus can also be debugged with the native GDB. Due to the visualization interface running in an asynchronous thread, the hardware can be inspected in real-time with both methods.

3.1 Symbols and Connections

A view has to implement at least two functions: `getSymbol()` and `update()` (e. g., see Fig. 2). In `getSymbol()`, the view's layout such as size, shape, location of attribute fields and input/output pins is defined. This function is only called once during instantiation of the views. The actual values for the attributes are generated in the `update()` function, which is periodically called by the *visualization interface* (see Sect. 3.2). It may update the attributes of its instance and the values of all connected pins. To display useful information, the view needs a reference to the module it describes. For convenience, we supplied an auto-generation compiler macro for trivial views (non-templated models and no extra functions) to speed up the design process. How the view accesses its model is up to the designer and available interfaces; directly over class pointers, indirect over function calls, or any other way that C/C++ allows. Lastly, a *Connection* is a meta-element to connect two or more pins and can display relevant data, which can be set by any symbol that has connected pins to it.

```
1  const Symbol GPIOView::getSymbol() {
2    Rect size = default_box;  //100x100 units
3    riscview::Pin bus{"BUS", Direction::INOUT,
4      PinLocation{Orientation::left, Point{0,1*size.y/5}}
5    };
6    riscview::Pin o12{"12", Direction::OUT,
7      PinLocation{Orientation::right,
           Point{size.x,1*size.y/5}}
8    };
9    [...]
10   std::map<std::string, Attribute> attrs {
11     //name, init value, lower left alignment, margin, size
12     {"regs", {"", Locator::ll, {default_attrtextsize,
           size.y-default_attrtextsize},
           default_textsize/3}},
13   };
14   return Symbol("GPIO", {bus, o12, [...]}, size, attrs);
15 };
16 void GPIOView::update() {
17   std::string text = "VAL: " + toBin(model.value, 3);
18   instance.getPin("16")->getConnection()->setText(model.port
         & (1 << 10) ? "1" : "0");
19   instance.setAttribute("regs", text);
20 }
```

Figure 2: Example view building pins and attributes of a general purpose I/O (GPIO) hardware module (cf. the resulting symbol in Fig. 5)

```
1  #define GEN_DEFAULT_VIEW(CLASS)
2  struct CLASS##View : public Viewable {
3    static const Symbol symbol;
4    Instance instance;
5
6    static const Symbol getSymbol();
7
8    CLASS &model;
9    CLASS##View(CLASS &model, string name = #CLASS);
10
11   void update() override;
12 };
```

Figure 3: Trivial class structure of a default view. Trailing slashes are omitted for readability.

3.2 Visualization Interface

The visualization interface provides a library of usable layout objects (e. g., `Symbol`, `Direction`, `Orientation`, . . .), a registration function for all views, and an own update thread. At program startup, the interface tries to connect to the *command server* of the debugging GUI over a TCP connection. If no connection is possible, all further view-related function calls are ignored and the SystemC program continues as normal. Otherwise, the registered layout objects are serialized into individual commands and sent to the command server.

Every module and connection needs to be defined in an elaboration phase. This definition allows setting the size of the module, location and names of input or output pins, and the layout of attributes along with a unique identification (see Fig. 4). These properties cannot be changed after the instantiation to allow the visualization engine to place modules in a space-efficient manner. When the SystemC simulation starts, the update thread starts polling all registered instances periodically and checks for changed attributes. All changed attributes can be updated via their respective identification strings and are then serialized and sent to the command server (see Sect. 3.3).

Note that the update thread is independent of the SystemC simulation, and thus does neither affect nor is affected by the simulation time. Since our implementation of SystemC[2] is single-threaded, the impact of our proposed debugger on the simulation speed can be neglected when run on a multicore system.

```
1   GPIO gpio0("GPIO0", INT_GPIO_BASE);  // SysC HW-Model
2   RV_DEF_AND_ADD(GPIOView, gpio0);     // View
3   SPI spi1("SPI1");
4   RV_DEF_AND_ADD(SPIView, spi1);
5   SS1106 oled([...]);
6   RV_DEF_AND_ADD(SS1106View, oled);
7
8   riscview::Connection gpio_oled_dc("GPIO-OLED-DC");
9   gpio_oled_dc.connect(gpio0_v.instance.getPin("16"));
10  gpio_oled_dc.connect(oled_v.instance.getPin("DC"));
11  riscview.add(gpio_oled_dc);
12  [...]
13
14  if(!riscview.connect()) exit(-1); // connect with GUI
15  std::thread updater([&riscview]{  // start RISCview thread
16    while(true) {
17      ViewableRegistrar::updateAll(riscview);
18    }
19  });
20  sc_core::sc_start();              //start SysC thread
```

Figure 4: Excerpt of an initialization list of HW-modules and their views. RV_DEF_AND_ADD() is a compiler macro that instantiates and registers a view, naming it with the suffix _v.

3.3 Debugging GUI

The debugging GUI is responsible for collecting visualization commands and drawing appropriate structures. For interchangeability of the graphical representation, the visualization commands are based on Tcl/Tk. The GUI opens a server at start-up and listens for incoming commands from the visualization interface. These commands are then translated by the *command translation layer* into API-calls to the graphics engine. The Nlview visualization engine includes a placement algorithm to minimize the needed screen size and concisely routes the connections between the components.

4 CASE STUDY

As a case study, we implemented an OLED display as a HW module into an existing open source RISC-V Virtual Prototype (VP) [4] and wrote a software driver to interface the display. The VP is able to model the SiFive HiFive1 processor board including some of the most used peripherals (i. e., UART, SPI, Timers). The SS1106 OLED display driver is a multi-protocol driver (SPI 3-wire, SPI 4-wire, I^2C and others) supporting monochrome displays with up to 64×132 pixels resolution. We chose the SPI 4-wire connection because it has the fastest net transmission capabilities. To show the real-world comparability, we also designed a PCB (printed circuit board) with an OLED display and seven buttons. The PCB was designed such that it can be stacked on top of the HiFive1 board.

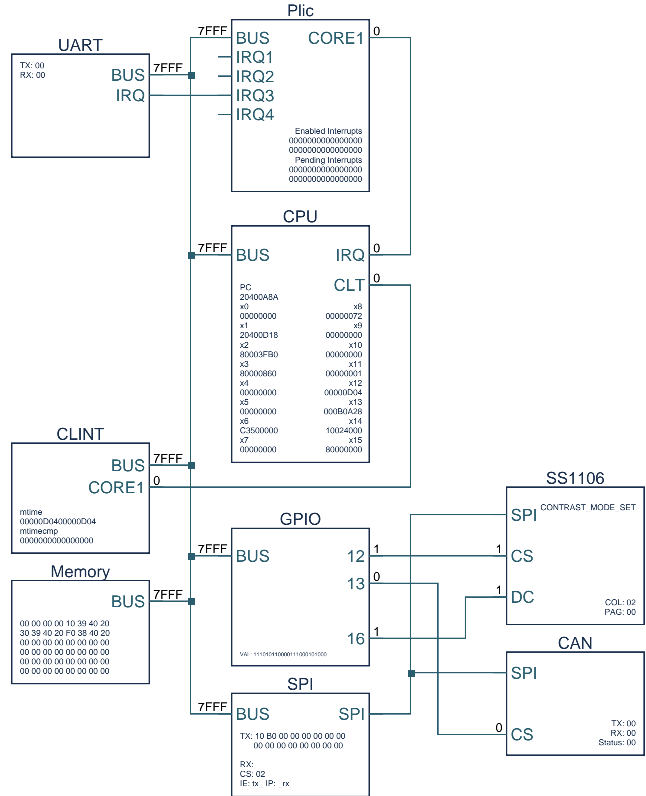

Figure 5: Screenshot of the architecture view in RISCview.

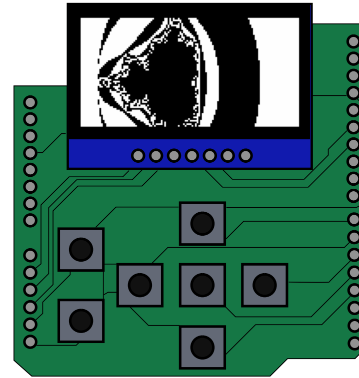

Figure 6: Screenshot of the VP simulation with an active OLED Display running an example program.

4.1 Display HW Model

We implemented a model for the display-driver according to its datasheet[3] and connected it to the HiFive's SPI peripheral (see Sect. 2). The SPI 4-Wire mode requires a differentiation of *command* and *data* bytes via a dedicated pin connected from the GPIO module to the display. Commands may consist of one to three bytes and expect up to two trailing value bytes. For instance, to set the display's

[2] https://www.accellera.org/downloads/standards/systemc

[3] https://www.velleman.eu/downloads/29/infosheets/sh1106_datasheet.pdf

contrast, the command line has to be set low, and the byte 0x81 for *set contrast* along with the value (encoded in one byte) has to be sent over SPI. Issuing multiple data bytes after a PAGE_ADDR command are interpreted as consecutive pixel values, incrementing the internal pixel pointer state. To aid with the design process, we also created a view of display driver showing sent data, the last command, and an excerpt of internal state. The implementation of the view took only 23 lines of code (see Fig. 7).

```
1  GEN_DEFAULT_VIEW(SS106);
2  const Symbol SS1106View::getSymbol() {
3    Rect size = default_box;
4    std::vector<nlv::Pin> pins = {
5      nlv::Pin {"SPI", Direction::INOUT,
6      PinLocation{Orientation::left, Point{0,  size.y/5}}},
7      nlv::Pin { "CS", Direction::IN,
8      PinLocation{Orientation::left, Point{0,  size.y/2}}},
9      nlv::Pin { "DC", Direction::IN,
10     PinLocation{Orientation::left, Point{0,4*size.y/5}}},
11   };
12   std::map<std::string, Attribute> attrs {
13     {"command", {"", Locator::lr,
           {size.x-default_attrtextsize,
           1.5*default_textsize}, default_attrtextsize}},
14     {"regs", {"", Locator::lr,
           {size.x-default_attrtextsize,
           size.y-default_attrtextsize},
           default_attrtextsize}},
15   };
16   return Symbol("SS1106", pins, size, attrs);
17 };
18 void update() {
19   std::string text = "COL: " + toHex(model.state->column)
         +
20   "\nPAG: " + toHex(model.state->page);
21   instance.setAttribute("command", ~model.last_cmd.op);
22   instance.setAttribute("regs", text);
23 };
```

Figure 7: Code to generate a view for the SS1106 Controller (cf. Fig. 5).

4.2 Display SW Driver

The SW driver offers a set of high-order functions like *set pixel at position x* and *draw line from point x to y* and translates them to series of low-level commands for the display. It also manages the values for GPIO-Pins and handles the SPI peripheral interface, both over memory mapped I/O.

4.3 Debugging

During development of the software driver, we noticed undefined behavior of the display during operations with a high pixel-throughput. Sometimes, the display glitched in a way that the image was distorted or showed random artifacts (see Fig. 8).

Our first approach to finding this bug was starting the simulation with a breakpoint on the software side in the display driver routine that handles the SPI transfers. However, this did not yield any results, because the simulation did not show any false behavior as long as the breakpoint was active. Also, printing out the SPI bytes over the serial monitor suppressed the undefined behavior. Our second approach was to set a breakpoint in the display module (hardware side) at the command interpretation state machine. We noticed that the display driver got invalid command bytes that were not implemented in the software driver. Also, the display got too

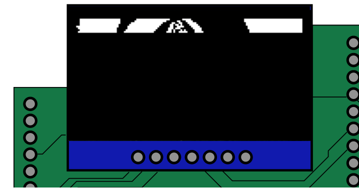

Figure 8: Glitched display showing only a partial image and distorted lines. This simulation behaves exactly like the real HiFive1 board with our custom PCB.

```
1  void mode_data(void) {
2    setPin(OLED_DC, 1);
3  }
4  void mode_cmd(void) {
5    setPin(OLED_DC, 0);
6  }
7  void setContrast(uint8_t contrast) {
8    mode_cmd();
9    spi(0x81);   //Command: next byte is contrast value
10   spi(contrast);
11 }
12 void oled_init() {
13   spi_init();
14   // Initial setup
15   // Enable RESET and D/C Pin
16     GPIO_REG(GPIO_OUTPUT_EN) |= (1 <<
           mapPinToReg(OLED_RES) | 1 <<
           mapPinToReg(OLED_DC));
17   setPin(OLED_DC, 0);
18
19   // RESET
20   setPin(OLED_RES, 0);
21   sleep_u(10);  // at least 10us
22   setPin(OLED_RES, 1);
23   sleep(100);   // at least 100ms
24   // Initialize display to desired operating mode.
25   [...]
26   setChargePumpVoltage(0b10);
27   setContrast(0xff);
28   // Clear screen (overwrite entire memory with zeroes)
29   oled_clear();
30   setDisplayOn(1);
31 }
```

Figure 9: Part of the original display software driver.

many consecutive data bytes, thus writing out of bounds of its page buffer. We paused the execution of the SystemC executable with a breakpoint, halting when the display detected an invalid command. By inspecting the RISCview window (Fig. 8), we could see that the TX-Queue of the SPI module still contained command bytes, but the D/C-line was already set high (*data mode*). In this state, the display's state machine still expected a second command byte for the contrast value (CONTRAST_MODE_SET). This observation led us to the idea that the switch between data and command mode did not wait until the whole SPI transmit queue was emptied. It also explained why a debug print in the software driver suppressed the problem; the time it takes to send text through the comparatively slower UART was enough for the SPI TX queue to run empty.

The fix itself required only a few lines to change: Before switching between data- and command mode, wait for the lower SPI transmit watermark (SPI_IP_TXWM) to indicate an empty transmit queue (see Fig. 11).

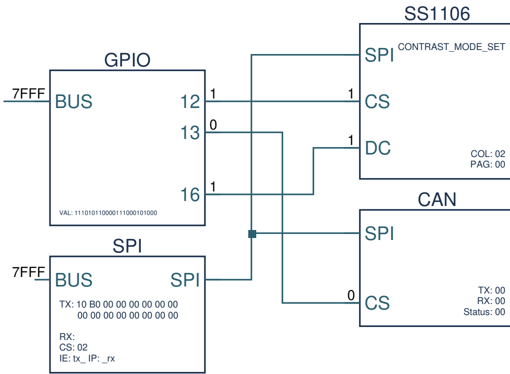

Figure 10: Snapshot of a still command-populated TX queue, although Data/Command line just toggled to data mode. Note the populated TX buffer in the SPI peripheral, where the top left byte is the first to be transmitted. The first two are still commands: `0x10` for the contrast value and `0xB0` for the charge pump voltage. Following bytes are all zeroes to clear the screen. Additional status flags indicate that the RX queue is empty, chip select (CS) is set to device 2 (ss1106), and the TX interrupt is enabled but not pending.

```
1  void spi_complete() {
2    // Wait for interrupt condition.
3    while (!(SPI1_REG(SPI_REG_IP) & SPI_IP_TXWM))
4      asm volatile("nop");
5    // TX-Watermark is set while byte is still in transit
6    // One byte at 8KBit/s is one microsecond
7    sleep_u(1);
8  }
9  void mode_data(void) {
10   // not already in data mode
11   if(!getPin(OLED_DC)) {
12     // wait for SPI to complete before toggling
13     spi_complete();
14     setPin(OLED_DC, 1);
15   }
16 }
17 void mode_cmd(void) {
18   // not already in command mode
19   if(getPin(OLED_DC)) {
20     // wait for SPI to complete before toggling
21     spi_complete();
22     setPin(OLED_DC, 0);
23   }
24 }
```

Figure 11: Fixed part of the software driver.

4.4 Evaluation

If we had used just the normal GDB debugger, the underlying problem would not have been clear. When the program is halted at the memory interface of the display module, the access to the state of the SPI module is hidden behind the stack-frames of the different user-space threads of SystemC. The encountered bug was also noticeable in the real hardware, which shows the accuracy of our case-study.

5 CONCLUSION AND FUTURE WORK

This paper has presented a novel system for hardware/software co-debugging that is applicable in an early stage of the development with a minimal impact on design-time. Using a transaction-level virtual prototype of the hardware, written in SystemC, it provides a live view on the internals of the hardware design, while stepping through the executed software using a state-of-the-art software debugger like GDB. The integration into a project requires little adaptation to the code-base with a flexible view on the hardware. A case study with a modeled OLED-Display operated by a RISC-V processor demonstrated the usefulness of our visualization for finding bugs related to hardware-software interactions.

Our system opens up possible future work, including:

- Combining the system with a dynamic flow analysis framework to visualize security policy violations and data flow in real-time;
- Adding a static code analysis based on re-occurring SystemC class patterns, which would enable automated visualization of modules at the expense of displaying possibly irrelevant information;
- Implementing a hardware-version of the visualization interface to permit hardware debugging with the real hardware in the same style as its VP.

ACKNOWLEDGMENTS

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SATiSFy under contracts no. 16KIS0821K and 16KIS0825.

REFERENCES

[1] Concept Engineering GmbH. 2021. Nlview 7.3.11. https://www.concept.de.

[2] Mehran Goli and Rolf Drechsler. 2019. Scalable Simulation-based Verification of SystemC-based Virtual Prototypes. In *Euromicro Conf. on Digital System Design (DSD)*. IEEE, 522–529. https://doi.org/10.1109/DSD.2019.00081

[3] Daniel Große, Rolf Drechsler, Lothar Linhard, and Gerhard Angst. 2003. Efficient Automatic Visualization of SystemC Designs. In *Forum on Specification and Design Languages (FDL)*. ECSI, 646–658.

[4] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2018. Extensible and Configurable RISC-V based Virtual Prototype. In *Forum on Specification and Design Languages (FDL)*. 5–16. https://doi.org/10.1109/FDL.2018.8524047

[5] 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 (Oct. 2020), 101756. https://doi.org/10.1016/j.sysarc.2020.101756

[6] M. Holzer, B. Knerr, P. Belanović, M. Rupp, and G. Sauzon. 2004. Faster Complex SoC Design by Virtual Prototyping. In *Int'l Conf. on Cybernetics and Information Technologies, Systems and Applications (CITSA)*. 305–309.

[7] IEEE Computer Society. 2015. *IEEE Standard for Standard SystemC Language Reference Manual*. Standard IEEE 1666-2015. https://doi.org/10.1109/IEEESTD.2012.6134619

[8] K. Lee, A. Su, Long-Feng Chen, Jia-Wei Jhou, J. Kuo, and M. Liu. 2011. A software/hardware co-debug platform for multi-core systems. In *IEEE Int'l Conf. on ASIC*. 259–262. https://doi.org/10.1109/ASICON.2011.6157171

[9] Frank Rogin, Christian Genz, Rolf Drechsler, and Steffen Rülke. 2008. An Integrated SystemC Debugging Environment. In *Embedded Systems Specification and Design Languages*. Lecture Notes in Electrical Engineering, Vol. 10. Springer, 59–71.

[10] Richard M. Stallman, Roland Pesch, Stan Shebs, et al. 2020. *Debugging with GDB: The GNU Source-Level Debugger* (10th ed.). GNU. https://sourceware.org/gdb/current/onlinedocs/gdb.pdf

[11] Lukas Steiner, Matthias Jung, Felipe S. Prado, Kirill Bykov, and Norbert Wehn. 2020. DRAMSys4.0: A Fast and Cycle-Accurate SystemC/TLM-Based DRAM Simulator. In *Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Springer, 110–126.

[12] S. Swan. 2006. SystemC transaction level models and RTL verification. In *43rd ACM/IEEE Design Automation Conference*. 90–92.

[13] Rüdiger Willenberg and Paul Chow. 2013. Simulation-based HW/SW co-debugging for field-programmable systems-on-chip. In *Int'l Conf. on Field-Programmable Logic and Applications (FPL)*. IEEE, 1–8.