Extensible and Configurable RISC-V based Virtual Prototype⋆

Vladimir Herdt[1] Daniel Große[1,2] Hoang M. Le[1] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{vherdt,grosse,hle,drechsle}@informatik.uni-bremen.de

Abstract—**Internet-of-Things (IoT) opens a new world of possibilities for both personal and industrial applications. At the heart of an IoT device, the processor is the core component. Hence, as an open and free instruction set architecture RISC-V is gaining huge popularity for IoT. A large ecosystem is available around RISC-V, including various RTL implementations at one end and high-speed instruction set simulators (ISSs) at the other end. These ISSs facilitate functional verification of RTL implementations as well as early SW development to some extent. However, being designed predominantly for speed, they can hardly be extended to support further system-level use cases such as design space exploration, power/timing/performance validation or analysis of complex HW/SW interactions.**

In this paper, we propose and implement the first RISC-V based Virtual Prototype (VP) with the goal of filling this gap. We provide a RISC-V RV32IM core, a PLIC-based interrupt controller and an essential set of peripherals together with SW debug capabilities. The VP is designed as extensible and configurable platform with a generic bus system and implemented in standard-compliant SystemC and TLM-2.0. The latter point is very important, since it allows to leverage cutting-edge SystemC-based modeling techniques needed for the mentioned use cases. Our VP allows a significantly faster simulation compared to RTL, while being more accurate than existing ISSs. Finally, our RISC-V VP is fully open source to help expanding the RISC-V ecosystem and stimulating further research and development.

I. Introduction

Enormous innovations are enabled by the *Internet-of-Things* (IoT) since every device is connected to the Internet. Forecasts see additional economic impact resulting from Industrial IoT. In the last years the complexity of IoT devices has been increasing steadily with various conflicting requirements. On the one hand IoT devices need to provide smart functions with a high performance including real-time computing capabilities, connectivity and remote access as well as safety, security and high reliability. At the same time they have to be cheap, work efficiently with an extremely small amount of memory and limited resources and should further consume only a minimal amount of power to ensure a very long lifetime.

To meet the requirements of a specific IoT system, a crucial component is the processor. Stimulated from the enormous momentum of open source software, a counterpart on the hardware side recently emerged: *RISC-V* [1], [2]. RISC-V is an open-source *Instruction Set Architecture* (ISA) which is license-free and royalty-free. The ISA standard is maintained by the non-profit RISC-V foundation and is appropriate for all levels of computing systems, i.e. from micro-controllers to supercomputers. The RISC-V ecosystem is rapidly growing, ranging from HW, e.g. various HW implementations (free as well as commercial) to high-speed *Instruction Set Simulators* (ISSs) These ISSs facilitate functional verification of RTL implementations as well as early SW development to some extent. However, being designed predominantly for speed, they can hardly be extended to support further system-level use cases such as design space exploration, power/timing/performance validation or analysis of complex HW/SW interactions.

A major industry-proven approach to deal with these use cases in earlier phases of the design flow is to employ *Virtual Prototypes* (VPs) [3] at the abstraction of *Electronic System Level* (ESL) [4]. In industrial practice, the standardized C++-based modeling language SystemC and *Transaction Level Modeling* (TLM) techniques [5], [6] are being heavily used together to create VPs. Depending on the specific use case, advanced state-of-the-art SystemC-based techniques beyond functional modeling (see e.g. [7], [8], [9], [10], [11]) are to be applied on top of the basic VPs. The much earlier availability as well as the significantly faster simulation speed in comparison to RTL are among the main benefits of SystemC-based VPs.

In this paper, we propose and implement the first RISC-V based VP to further expand and bring the benefits of VPs to the RISC-V ecosystem. With the goal of filling the mentioned gap in supporting further system-level use cases, SystemC is necessarily the language of choice. The VP is therefore implemented in standard-compliant SystemC and TLM-2.0 and designed as extensible and configurable platform with a generic bus system. We provide a RISC-V RV32IM core and a PLIC-based interrupt controller with an essential set of peripherals. We demonstrate the extensibility of our VP by two examples: addition of a sensor peripheral and extension by GDB debug functionality from the application SW perspective. In the experimental evaluation we show the high simulation performance of our VP based on several optimizations. Our RISC-V VP is fully open source[1] to stimulate further research and development.

Related Work: As mentioned earlier, the RISC-V ecosystem already has various high-speed ISSs such as the reference simulator Spike [12], RISCV-QEMU [13], or RV8 [14]. They are mainly designed to simulate as fast as possible and

⋆ This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project CONFIRM under contract no. 16ES0565 and by the University of Bremens Central Research Development Fund and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

[1] Available at https://github.com/agra-uni-bremen/riscv-vp, for more information and updates also visit www.systemc-verification.org/riscv-vp

predominantly employ dynamic binary translation (to x86_64) techniques. This is however a trade-off as accurately modeling power or timing information for instructions becomes much more challenging. The full-system simulator gem5 [15], at the time of writing also has initial support for RISC-V. gem5 provides more detailed models of processors and memories and can in principle also be extended for accurate modeling of extra-functional properties. However, it employs a different modeling style and thus hinders the integration of advanced SystemC-based techniques. The project SoCRocket [16] that develops an open-source SystemC-based VP for the SPARC V8 architecture, can be considered comparable to our effort. Finally, commercial VP tools such as Synopsys Virtualizer or Mentor Vista might also support RISC-V but their implementation is proprietary.

II. Preliminaries

A. RISC-V

RISC-V is an open and free instruction set architecture (ISA). The ISA consists of a mandatory base integer instruction set and various optional extensions. The integer set is available in three different configurations with 32, 64 and 128 bit width registers, respectively: RV32I, RV64I and RV128I. Additionally, the RV32E configuration, which is essentially a lightweight RV32I with a reduced number of registers, is available and intended for (very) small embedded devices. Extensions are denoted with a single letter, e.g. M (integer multiplication and division), A (atomic instructions), C (compressed instructions), etc. A comprehensive description of the RISC-V instruction set is available in the specification [1].

The second volume of the RISC-V ISA specification defines a privileged architecture description [2]. It defines *control and status registers* (CSRs), which are registers serving a special purpose. For example the *misa* (Machine ISA) register is a read-only CSR that contains information about the supported ISA. Another example is the *mtvec* (Machine Trap-Vector Base-Address) CSR that stores the address of the trap/interrupt handler. The privileged architecture description provides a small set of instructions for interrupt handling (*wfi*, *mret*) and interacting with the system environment (*ecall*, *ebreak*).

B. SystemC and TLM

SystemC is a C++ class library that includes an event-driven simulation kernel. The structure of a SystemC design is described with ports and modules, whereas the behavior is described in processes which are triggered by events. The execution of a process is non-preemptive, i.e. the kernel receives the control back if the process has finished its execution or suspends itself by calling *wait()*. SystemC provides three types of processes with SC_THREAD being the most general type, i.e. the other two can be modeled by using SC_THREAD. For event-based synchronization, SystemC offers many variants of *wait()* and *notify()* such as *wait(time)*, *wait(event)*, *event.notify(delay)*, *event.notify()*, etc.

Communication between modules is implemented through (TLM) transactions. A transaction object essentially consists of the command (e.g. read/write), the start address, the data length, and the data pointer. It allows to implement various memory access operations. Optionally, a transaction can be associated with a delay (modeled as *sc_time* data structure), which denotes the execution time of the transaction and allows to obtain a more accurate overall simulation time estimation.

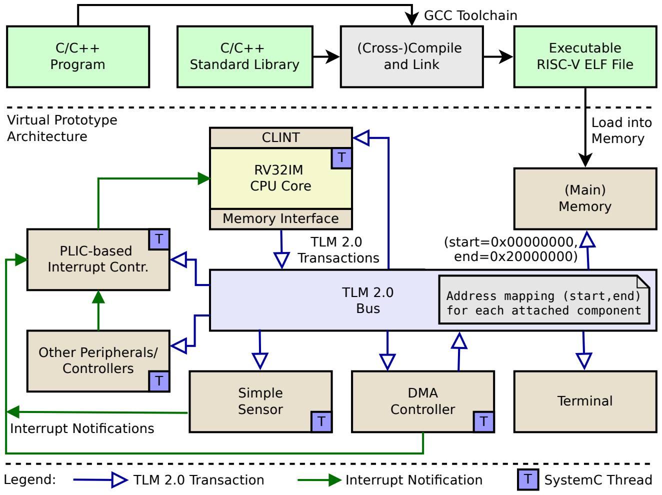


Fig. 1. Virtual prototype architecture overview

Fig. 6 shows a basic sensor model implementation in SystemC that communicates through TLM transactions (the *transport* method) to demonstrate the modeling principles. We will describe the example in more detail later in Section VI-A.

III. RISC-V based VP Architecture

The VP is implemented in SystemC and designed as extensible and configurable platform around a RISC-V RV32IM CPU core with a generic bus system employing TLM 2.0 communication and support for the GCC toolchain - including coverage tracking with GCOV and debugging with GDB, of the software applications executed on our VP. Overall, the VP consists of around 3000 lines of C++ code with all extensions. Fig. 1 shows an overview of the VP architecture. In the following we present more details.

A. Core

The CPU core loads, decodes and executes one instruction after another. We provide support for the RISC-V RV32IM instruction set in the CPU core and target the current version of the RISC-V machine level ISA as defined in the RISC-V privileged architecture specification [2]. This includes the machine level *control and status register* (CSRs) as well as instructions for interrupt handling (*wfi*, *mret*) and environment interaction (*ecall*, *ebreak*). We will provide more details on the implementation of interrupt handling and system calls (environment interaction) in the following sections.

B. Bus

The TLM bus is responsible on routing transactions from an initiator, i.e. (bus) master, to a target. Therefore, all target components are attached to the TLM bus at specific non-overlapping address ranges. The bus will match the transaction address with the address ranges and dispatch the transaction accordingly to the matching target. Please note, in this process the bus performs a *global-to-local* address translation in the transaction. For example, assume that a sensor component is mapped to the address range (start=0x50000000, end=0x50001000) and the transaction address is 0x50000010, then the bus will route the transaction to the sensor and change the transaction address to 0x00000010 before passing it on to the sensor. Thus the sensor works on local address ranges. The TLM bus supports multiple masters initiating transactions. Currently, the CPU core as well as the DMA controller are configured as bus masters. Please note, that a single component can be both a master and a target, as for example the DMA controller receives transactions initiated by the CPU core to configure the source and destination address ranges and also initiates transactions by itself to perform the memory access operations without the CPU core.

C. Interrupts

Two sources of interrupts are available: 1) local and 2) external. Essentially, there are two sources of local interrupts: traps/exceptions encountered during software execution as well as timer interrupts generated by a special built-in timer component. The timer is part of the *core local interrupt controller* (CLINT) and can be configured through memory mapped IO. Local interrupts are processed with higher priority than external interrupts. External interrupts are all remaining interrupts triggered by the various components in the system. To handle external interrupts, we provide a PLIC-based interrupt controller (IC), based on the description from [2]. The IC will collect and prioritize all external interrupts and then route them to the CPU core one by one. We will describe the interrupt handling process in more details later.

D. VP Initialization

The main function in the VP is responsible to instantiate, initialize and connect all components, i.e. setup the architecture. An ELF loader is provided to parse and load an executable RISC-V ELF file into the memory and setup the program counter in the CPU core accordingly. Finally, the SystemC simulation is started. The ELF file is produced by the GCC toolchain by (cross-)compiling the application program and optionally linking it with the C/C++ standard library (we also support a bare-metal execution environment without C/C++ library).

IV. VP Interaction with SW and Environment

In this section we present more details on the HW/SW interaction, in particular on interrupt handling, and environment interaction via system calls in our VP.

A. Interrupt Handling and HW/SW Interaction

In the following we present an example application that periodically accesses a sensor to demonstrate the interaction between hardware (VP-side) and software with a particular focus on interrupt handling. We first describe the software application running on the VP and then present a minimal assembler boostrap code to initialize interrupt handling and describe how interrupts are processed in more details. Later in Section VI-A we present the corresponding SystemC-based sensor implementation in our VP.

```
 1 #include "stdint.h"
 2 #include "irq.h"
 3
 4 static volatile char * const TERMINAL_ADDR = (char *
        const)0x20000000;
 5 static volatile char * const SENSOR_INPUT_ADDR = (char *
        const)0x50000000;
 6 static volatile uint32_t * const SENSOR_SCALER_REG_ADDR =
        (uint32_t * const)0x50000080;
 7 static volatile uint32_t * const SENSOR_FILTER_REG_ADDR =
        (uint32_t * const)0x50000084;
 8
 9 _Bool has_sensor_data = 0;
10
11 void sensor_irq_handler() {
12   has_sensor_data = 1;
13 }
14
15 void dump_sensor_data() {
16   while (!has_sensor_data) {
17     asm volatile ("wfi");
18   }
19   has_sensor_data = 0;
20
21   for (int i=0; i<64; ++i) {
22     *TERMINAL_ADDR = *(SENSOR_INPUT_ADDR + i);
23   }
24 }
25
26 int main() {
27   register_interrupt_handler(2, sensor_irq_handler);
28
29   *SENSOR_SCALER_REG_ADDR = 5;
30   *SENSOR_FILTER_REG_ADDR = 2;
31
32   for (int i=0; i<3; ++i)
33     dump_sensor_data();
34
35   return 0;
36 }
```

Fig. 2. Example application running on the VP to demonstrate the hardware/software interaction

```
.globl _start
.globl main
.globl level_1_interrupt_handler

_start:
la t0, level_0_interrupt_handler
csrw mtvec, t0
li t1, 0x888
csrw mie, t1
jal main

loop:
j loop

level_0_interrupt_handler:
csrr a0, mcause
jal level_1_interrupt_handler
mret
```

Fig. 3. Bare-metal bootstrap code demonstrating interrupt handling

1) Software Side: Fig. 2 shows an example application that reads data from a sensor and copies the data to a terminal component. The sensor and terminal are accessed through memory mapped (MM) IO. Their addresses are defined at the top of the program. They need to match with the configuration in the VP. The sensor periodically triggers an interrupt, denoting that new data is available. The main function starts by registering an interrupt handler for the sensor interrupt (Line 27). Again, the interrupt number specified in SW has to match the configuration in the VP. Next, the sensor is configured in Line 29-30 using MMIO. The scaler denotes how fast sensor data is generated and the filter setting what kind of post-processing is performed on the data. Finally, the copy process is iterated for three times (Line 32-33) before the program terminates. Each iteration starts by waiting for sensor data (Line 16-18). The global boolean flag *has_sensor_data* is used for synchronization. It is set in the interrupt handler (Line 12) and unset again immediately after the waiting loop (Line 19). Please note, the *wfi* instruction will power down the CPU core until the next interrupt occurs.

2) Bootstrap Code and Interrupt Handling: Fig. 3 shows the essential parts of a bare-metal boostrap code, which is written in assembler and linked with the application code, to handle interrupts[2]. The *_start* label is the entry point of the whole program. The registers *mtvec*, *mie* and *mcause* are CSRs that essentially store the interrupt handler address, enabled interrupts and interrupt source, respectively. The instructions *csrr* and *csrw* read and write a CSR into and from a normal CPU register, respectively. Before the main function is called (Line 10), the interrupt handler base address (level-0) is stored in *mtvec* (Line 6-7) and all interrupts are enabled (Line 8-9). After the main function returns, the VP simulation terminates, because a loop is detected which does not contain any further instructions (Line 13).

In general, an interrupt can occur at any time during execution of the application SW. All interrupts propagate to the interrupt controller (IC) first and are prioritized there. The CPU core only receives a notification that some interrupt is pending and needs to be processed. The CPU will first store the execution context, i.e. program counter and register values, and then read the base address from the *mtvec* CSR and set the program counter to that address, i.e. effectively directly jumping to the level-0 interrupt handler (first instruction at Line 16). The interrupt handler (level-0) first in Line 16 reads the reason (i.e. local or external interrupt) for the interrupt into the *a0* CPU register, which according to the RISC-V calling convention [17] stores the first argument of a function call. Then in Line 17 an interrupt handler implemented in C is called (level-1, not shown in this example). Essentially, this level-1 handler deals with a local timer interrupt by resetting the timer and with an external interrupt by asking the IC for the actual interrupt number with the currently highest priority (through a memory mapped register access) and then calls the application provided interrupt handler function (Line 11-13 in Fig. 2, this step is ignored if none has been registered for the interrupt number). Finally, the *mret* instruction restores the previously stored execution context. Please note, that storing and re-storing the register values can also be implemented in SW, by pushing and popping them to/from the stack before/after calling the level-1 handler, respectively.

B. Environment Interaction: Syscalls and C/C++ Library

System calls (syscalls) are executed by redirecting them to the host system running the VP simulation. This requires to pass arguments from the guest application into the host system and integrate the return values back into the guest application (i.e. memory of the VP). Implementing syscalls enables support for the C/C++ standard library. Furthermore, we can directly use GCOV to track the coverage of the applications simulated on our VP (the GCOV instrumentation requires syscall support to open and write to files).

```
#define SYS_write 64

ssize_t write(int fd, const void *buf, size_t count) {
  return syscall(SYS_write, fd, (long)buf, count, 0);
}

long syscall(long n, long _a0, long _a1, long _a2, long _a3) {
  // store arguments in CPU register and trigger ecall
  register long a0 asm("a0") = _a0;
  register long a1 asm("a1") = _a1;
  register long a2 asm("a2") = _a2;
  register long a3 asm("a3") = _a3;
  register long a7 asm("a7") = n;

  // special instruction causing a jump to the syscall handler
  asm volatile ("ecall" : "+r"(a0) : "r"(a1), "r"(a2), "r"(a3),
        "r"(a7));

  // store potential error code and return result
  if (a0 < 0) {
    errno = -a0;
    return -1;
  } else {
    return a0;
  }
}
```

Fig. 4. System call handling stub linked with the C library (guest side, executed on the VP host system)

```
#define SYS_write 64

// execute syscall on the host system
ssize_t sys_write(int fd, const void *buf, size_t count) {
    const void *p = (const void *)guest_to_host_pointer(buf);
    return write(fd, p, count);
}

long execute_syscall(long n, long _a0, long _a1, long _a2, long
      _a3) {
    switch (n) {
        case SYS_write:
            return sys_write(_a0, (const void *)_a1, _a2);
        //...
    }
}

// function inside the CPU core
void execute_step() {
  auto instr = mem_if->load_instr(program_counter);
  auto op = decode(instr);

  switch (op) {
    case Opcode::ECALL: {
      regs[a0] = execute_syscall(regs[a7], regs[a0], regs[a1],
            regs[a2], regs[a3]);
    } break;
    //...
  }
}
```

Fig. 5. System call execution on the VP by redirecting to the host system

For example consider the *printf* function provided by the C standard library. Most of its functionality is implemented as portable C code independent of the execution environment. Essentially, the *printf* function will apply all formatting rules and create a simple char buffer, which is then passed to the *write* system call. At this point interaction with the execution environment is required. Fig. 4 shows the relevant part of a stub that is provided in the RISC-V port of the C library[3]. Essentially, the arguments of the system call are stored in the CPU registers *a0* to *a3* and the syscall number in *a7*. Then the *ecall* instruction is executed. The VP simulator will detect the *ecall* instruction and directly execute the syscall on the host system as shown in Fig. 5[4]. In case of the *write* syscall a

[2] Support for integration with the C/C++ library is also available, e.g. by executing the instructions at the beginning of the main function or integrating them directly into the *crt0.S* file, which is the entry point of the C library and similarly to the bare-metal code also calls the main function after performing some basic initialization tasks.

[3] Example based on the *newlib* port https://github.com/riscv/riscv-newlib.

[4] It is also possible to execute a trap handler, similar to the interrupt handler described in the previous section (e.g. essentially, jump to the level-0 interrupt handler with the *mcause* CSR being set to a syscall number), and then redirect the write to e.g. a terminal component.

pointer argument *buf* is passed. This is a pointer value from the guest system, i.a. an index in the VP byte memory array *mem*, and has to be translated to a host memory pointer in order to execute the *write* syscall on the host system. Therefore, the guest_to_host_pointer function (Line 5) adds the the base address of the VP byte memory array, i.e. *mem + buf*. The result of the syscall is stored in the *a0* register and passed back to the C library. We have implemented other syscalls in a similar way to the *write* syscall.

In general the guest and host system have a different architecture with different word sizes, e.g. in our case the guest system (which is simulated in the VP) is a 32 bit and the host system (which runs the VP) is a 64 bit system. Therefore, one has to be careful when data is passed between the guest and the host. Primitive types, e.g. int and bool, can be passed directly from the guest to the host, because our host system running the VP uses data types with equal or larger sizes, thus no information is lost when passing the arguments. When passing values back from the host a check can be performed, if necessary, to ensure that no relevant information is truncated, e.g. due to casting a 64 bit value into a 32 bit one. Pointer arguments need to be translated to host addresses, as described above, before accessing them on the host system. A write access is thus directly propagated back to the guest application. Structs can be accessed and copied recursively, considering the rules for accessing primitive and pointer types.

V. VP Performance Optimzations

In this section we discuss two performance optimizations for our VP that result in significant simulation speed-ups. The first optimization is a direct memory interface to fetch instructions and perform load/store operations from/to the (main) memory more efficiently. The second is a temporal decoupling technique with local time quantums to reduce the number of costly context switches, especially, in the CPU core simulation. We describe both techniques in the following.

A. Direct Memory Interface

The CPU core translates every load and store operation into a transaction which is routed through the bus to the target. Most of the time the main memory is the target of the access. Always accessing the memory through a bus transaction can be very costly. Even more so, because fetching the next instruction requires to load it from the memory too. Thus, at least one bus transaction is executed for every instruction. To optimize the access of the main memory and in particular instruction fetching, we provide two proxy classes with a direct memory interface. The direct memory interface stores the address offset where the memory is mapped in the overall address space as well as the size and pointer to the start of the memory. We have a proxy class for fetching instructions and one to access the memory in general, i.e. to perform load/store byte/half/word instructions. With the proxy classes enabled, the CPU core will first query the proxy class. It will match in case the main memory is accessed (for the instruction proxy class we only allow to fetch instructions from main memory) and otherwise convert the access into a transaction and normally route it through the bus.

B. Local Time Quantums

A SystemC-based simulation is orchestrated by the SystemC simulation kernel that switches execution between the various threads. While this is not a performance problem for most components, since they become runnable on very specific events, context switching can become a major bottleneck in simulating the CPU core. The reason is that a direct implementation will perform a context switch after executing every instruction, because simulation time has passed and the SystemC kernel needs to check for other runnable threads to perform synchronization. However, most of the time no other thread becomes runnable and the CPU thread is resumed again. Even if some other thread would become runnable it is still fine to keep running the CPU thread for some time (ahead of the global simulation time of the system). For example, even if the sensor thread would be runnable and trigger an interrupt once executed, delaying the sensor thread execution for a small amount of time and keeping the CPU thread running should not have influence on the functional behavior of the system. In general the software does have no knowledge of the exact timing behavior and thus is written in such a way, e.g. by employing locks and flags, to always wait for certain conditions.

VI. VP Extension and Configuration

Our VP is designed as a configurable and in particular extensible platform. It is very easy to add additional components (i.e. peripherals/controllers including bus masters) and attach them to the bus system at a new address range, or change the address mapping of the existing components. This allows for an easy (re-)configuration of the VP. By following the TLM 2.0 communication standard, transactions can be annotated with optional timing informations to obtain a more accurate timing model of the executed software. Support for additional RISC-V ISA extensions (beyond I and M) can be added inside the CPU core by extending the decode and execute functions accordingly. In general the compact implementation size (around 3000 lines of C++ code with all extensions) makes the VP very manageable and thus suitable as foundation for different application areas. In the following, we demonstrate the extensibility of our VP by two concrete examples: addition of a sensor peripheral and extension by GDB debug functionality from the application SW perspective.

A. Extending the VP with a Sensor Peripheral

This section presents the SystemC-based implementation of the VP sensor peripheral, which is used by the SW example presented in Section IV-A. It shows the principles on modeling peripherals and extending our VP as well as demonstrates the TLM communication and basic SystemC-based modeling and synchronization. The sensor is instantiated in the main function of the VP alongside the other components and attached to the TLM bus.

The sensor implementation is shown in Fig. 6. The sensor model has a data frame of 64 bytes that is periodically updated (overwritten with new data, Line 83-92) and two 32 bit configuration registers *scaler* and *filter*. The update happens in the run thread (the run function is registered as SystemC thread inside the constructor in Line 26). Based on the scaler register

```
1  struct SimpleSensor : public sc_core::sc_module {
2    tlm_utils::simple_target_socket<SimpleSensor> tsock;
3
4    interrupt_controller *ic = 0;
5    uint32_t irq_number = 0;
6    sc_core::sc_event run_event;
7
8    // memory mapped data frame
9    std::array<uint8_t, 64> data_frame;
10
11   // memory mapped configuration registers
12   uint32_t scaler = 25;
13   uint32_t filter = 0;
14   std::unordered_map<uint64_t, uint32_t *> addr_to_reg;
15
16   enum {
17     SCALER_REG_ADDR = 0x80,
18     FILTER_REG_ADDR = 0x84,
19   };
20
21   SC_HAS_PROCESS(SimpleSensor);
22
23   SimpleSensor(sc_core::sc_module_name, uint32_t irq_number)
24       : irq_number(irq_number) {
25     tsock.register_b_transport(this, &SimpleSensor::transport);
26     SC_THREAD(run);
27
28     addr_to_reg = {
29       {SCALER_REG_ADDR, &scaler},
30       {FILTER_REG_ADDR, &filter},
31     };
32   }
33
34   void transport(tlm::tlm_generic_payload &trans,
          sc_core::sc_time &delay) {
35     auto addr = trans.get_address();
36     auto cmd  = trans.get_command();
37     auto len  = trans.get_data_length();
38     auto ptr  = trans.get_data_ptr();
39
40     if (addr >= 0 && addr <= 63) {
41       // access data frame
42       assert (cmd == tlm::TLM_READ_COMMAND);
43       assert ((addr + len) <= data_frame.size());
44
45       // return last generated random data at requested address
46       memcpy(ptr, &data_frame[addr], len);
47     } else {
48       assert (len == 4);  // NOTE: only allow to read/write
              whole register
49
50       auto it = addr_to_reg.find(addr);
51       assert (it != addr_to_reg.end()); // access to non-mapped
              address
52
53       // trigger pre read/write actions
54       if ((cmd == tlm::TLM_WRITE_COMMAND) && (addr ==
              SCALER_REG_ADDR)) {
55         uint32_t value = *((uint32_t *)ptr);
56         if (value < 1 || value > 100)
57           return; // ignore invalid values
58       }
59
60       // actual read/write
61       if (cmd == tlm::TLM_READ_COMMAND) {
62         *((uint32_t *)ptr) = *it->second;
63       } else if (cmd == tlm::TLM_WRITE_COMMAND) {
64         *it->second = *((uint32_t *)ptr);
65       } else {
66         assert (false && "unsupported tlm command for sensor
                access");
67       }
68
69       // trigger post read/write actions
70       if ((cmd == tlm::TLM_WRITE_COMMAND) && (addr ==
              SCALER_REG_ADDR)) {
71         run_event.cancel();
72         run_event.notify(sc_core::sc_time(scaler,
                sc_core::SC_MS));
73       }
74     }
75   }
76
77   void run() {
78     while (true) {
79       run_event.notify(sc_core::sc_time(scaler,
              sc_core::SC_MS));
80       sc_core::wait(run_event);  // 40 times per second by
              default
81
82       // fill with random data
83       for (auto &n : data_frame) {
84         if (filter == 1) {
85           n = rand() % 10 + 48;
86         } else if (filter == 2) {
87           n = rand() % 26 + 65;
88         } else {
89           // fallback for all other filter values: random
                printable
90           n = rand() % 92 + 32;
91         }
92       }
93
94       ic->trigger_interrupt(irq_number);
95     }
96   }
97 };
```

Fig. 6. SystemC-based configurable sensor model that is periodically filled with random data - demonstrates the basic principles on modeling peripherals.

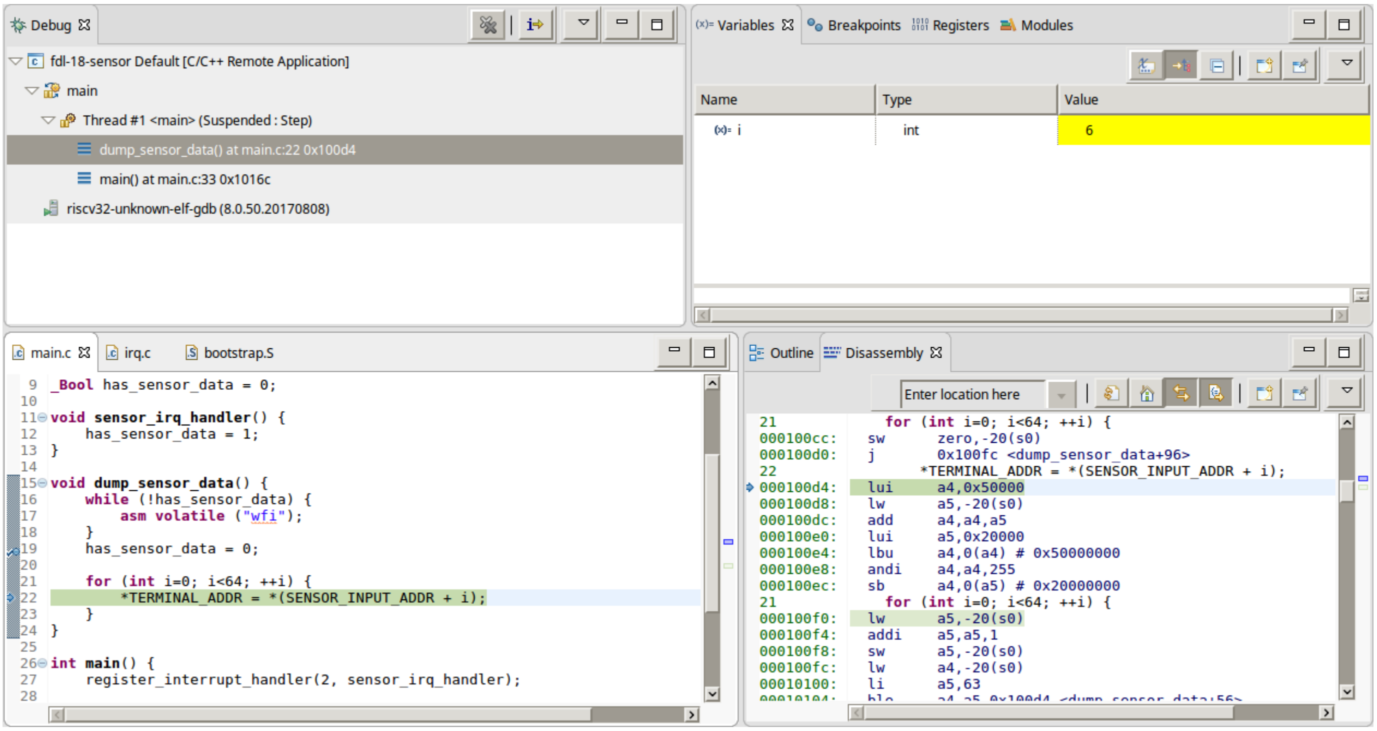
value this thread is periodically unblocked (Line 79) by calling the notify function on the internal SystemC synchronization event. Thus, *scaler* defines the speed at which new sensor data is generated. The *filter* register allows to select some kind of post-processing on the data. After every update an interrupt is triggered, which will propagate through the interrupt controller to the CPU core up to the interrupt handler in the application SW. Therefore, the sensor has a reference to the interrupt controller (IC, Line 4) and an interrupt number provided during initialization (Line 23 and Line 24).

Access to the data frame and configuration registers is provided through TLM transactions. These transactions are routed by the bus to the transport function (Line 34). The routing happens as follows: 1) The sensor has a TLM target socket field, which is bound in the main function (i.e. VP simulation entry point) to an initiator socket of the TLM bus. 2) The transport function is bound as destination for the target socket in the constructor (Line 25).

Based on the address and operation mode, as stored in the generic payload (Line 35-36), the action is selected. It will either read (part of) the data frame (Line 46) or read/write one of the configuration registers (Line 61-67). In case of a register access a pre-read/write validation and post-read/write action can defined as necessary. In this example, the sensor will ignore invalid scaler values (Line 54-58) and reset the data generation thread on a scaler update (Line 70-73). Please note, that the transaction object (generic payload) is passed by reference and provides a pointer to the data, thus a write access is propagated back to the initiator of the transaction. Optionally, an additional delay can be added to the sc_time delay parameter (also passed by reference) for a more accurate timing model.

B. Debugging Support Extension

We have implemented the GDB RSP (Remote Serial Protocol) interface to provide direct debugging support of applications running on our VP with the GDB debugger (in particular the freely available RISC-V port of the GDB, which knows about the available register set and the CSRs). Our VP acts as server and the GDB as client. They communicate through a TCP connection and send text based messages. A message is either a packet or a notification (a simple single char '+') that a packet has been successfully processed. Each packet starts with a '$' char and ends with a '#' char followed by a two digit hex checksum (the sum over the content chars modulo 256). For example the packet `$m111c4,4#f7` has the content m111c4,4 and checksum f7. The m command



```
fdl-18-sensor Default [C/C++ Remote Application]
  main
    Thread #1 <main> (Suspended : Step)
      dump_sensor_data() at main.c:22 0x100d4
      main() at main.c:33 0x1016c
  riscv32-unknown-elf-gdb (8.0.50.20170808)
```

Name	Type	Value
i	int	6

```c
_Bool has_sensor_data = 0;

void sensor_irq_handler() {
    has_sensor_data = 1;
}

void dump_sensor_data() {
    while (!has_sensor_data) {
        asm volatile ("wfi");
    }
    has_sensor_data = 0;

    for (int i=0; i<64; ++i) {
        *TERMINAL_ADDR = *(SENSOR_INPUT_ADDR + i);
    }
}

int main() {
    register_interrupt_handler(2, sensor_irq_handler);

```

```
21              for (int i=0; i<64; ++i) {
000100cc:   sw      zero,-20(s0)
000100d0:   j       0x100fc <dump_sensor_data+96>
22                  *TERMINAL_ADDR = *(SENSOR_INPUT_ADDR + i);
000100d4:   lui     a4,0x50000
000100d8:   lw      a5,-20(s0)
000100dc:   add     a4,a4,a5
000100e0:   lui     a5,0x20000
000100e4:   lbu     a4,0(a4) # 0x50000000
000100e8:   andi    a4,a4,255
000100ec:   sb      a4,0(a5) # 0x20000000
21              for (int i=0; i<64; ++i) {
000100f0:   lw      a5,-20(s0)
000100f4:   addi    a5,a5,1
000100f8:   sw      a5,-20(s0)
000100fc:   lw      a4,-20(s0)
00010100:   li      a5,63
00010104:   ble     a4,a5,0x100d4 <dump_sensor_data+56>
```

Fig. 7. Debugging the sensor application with Eclipse (screenshot showing relevant part of the debug view inside the Eclipse IDE)

denotes a memory read, in this case read 0x4 bytes starting from address 0x111c4. Our server might then for example return `+$05000000#85`, i.e. acknowledge the packet and return the value 5 (two chars per byte). To handle the packet processing and TCP communication we added a gdb-stub component to our VP. The whole debugging extension is only about 500 additional lines of C++ code most of them to implement the gdb-stub. On the VP side, only the CPU core has been modified to lift the SystemC thread into the gdb-stub, to allow the CPU to interrupt and exit the execution loop in case of a breakpoint and thus effectively transfer execution control to the gdb-stub.

Debugging works as follows: Start our VP in *debug-mode* (command line argument), this will transfer control to the gdb-stub implementing the RSP interface, waiting for a connection from the GDB debugger. In another terminal start the GDB debugger. Load the same executable ELF file into the GDB (command ”file main-elf”) as in our VP. Connect to the TCP server of the VP (command ”target remote :5005”, i.e. to connect to localhost using port 5005). Now the GDB debugger can be used as usual to set breakpoints, continue and step through the execution. It is also possible to directly use a visual debugging interface, e.g. *ddd* or *gdb-dashboard* or even the *Eclipse* IDE. Fig. 7 shows a screenshot of debugging the sensor application in Eclipse.

Please note, the ELF file contains information about the addresses and sizes of the various variables in memory. Thus, a print(x) command with an int variable x is already translated into a memory read command (e.g. m11080,4). Therefore, on the server side, i.e. our VP, an extensive parsing of ELF files is not necessary to add comprehensive debugging support. In total we have only implemented 24 different commands of which 9 can simply return an empty packet and a few more some pre-defined answer. Relevant packets are for example: read a register (p), read all registers (g), read memory range (m), set/remove breakpoint (Z0/z0), step (s) and continue (c).

VII. Experiments

In this section we present a performance comparison of our RISC-V based VP implementation with the RISC-V based *PULPino* platform (RTL implementation). For this comparison, the *PULPino* platform is simulated in a commercial RTL simulator. The *PULPino* platform is configured to use the *RISCY* core, which, similar to our core also supports the RV32IM instruction set. We also demonstrate the effectiveness of our presented VP simulation performance optimization techniques. All experiments are performed on a Linux system with an AMD Opteron 2220 SE processor with 2.8 GHz and 32 GB RAM.

For the evaluation we use the following benchmarks from the RV8 benchmark set: 1) *primes* computes prime numbers up to a limit of 33,333,333; 2) *qsort* sorts an array with 50 million elements; 3) *sha512* applies the sha512 cryptographic hash function on a 60 MB data set (1 million iterations). The RV8 benchmark set contains some additional benchmarks, which we have omitted from the comparison due to problems on executing them on the PULPino platform in the commercial RTL simulator. In addition to the RV8 benchmarks, we have added a bubblesort (sorting 50,000 elements) and a recursive mergesort (sorting 1 million elements) benchmark to the comparison. Due to timeouts (set to 4 hours, denoted T.O.) in the RTL simulation we also added down-scaled versions of the benchmarks (/s appended) to the comparison: primes/s has a limit of 33,333; qsort/s sorts 5,000 elements; sha512

TABLE I

Experiment results - all execution times are reported in seconds, timeout (T.O.) set to 4 hours (14400 seconds)

Benchmark	#instr-exec.	LOC	PULPino	Our RISC-V VP					
			RTL Sim.	basic	+i_dmi	+d_dmi	+q10	+q100	+q1000
bubblesort/s	2022052	20	787.49	0.97	0.71	0.58	0.43	0.41	0.40
mergesort/s	297226	41	56.70	0.39	0.35	0.35	0.32	0.29	0.29
primes/s	4341572	24	823.11	1.73	1.01	0.96	0.59	0.49	0.48
qsort/s	290765	146	64.50	0.40	0.35	0.34	0.31	0.30	0.27
sha512/s	8120416	154	1307.23	3.23	1.87	1.57	0.90	0.71	0.68
bubblesort	200197558	20	T.O.	69.21	44.16	30.71	16.31	11.97	11.35
mergesort	535918604	41	T.O.	197.32	107.89	86.48	41.17	27.77	26.15
primes	7114988801	24	T.O.	2400.34	1214.71	1089.36	542.46	387.09	374.33
qsort	3061611834	146	T.O.	1204.98	698.50	510.70	262.93	162.73	154.93
sha512	8071548963	154	T.O.	2773.6	1556.02	1302.75	616.1	432.52	406.34

operates on a 0.6 MB data set (1,000 iterations); mergesort/s and bubblesort/s sort 1,000 elements, respectively.

Table I shows the results of the experiments. The table is divided in two halfves: the upper half shows the down-scaled benchmark versions, while the lower half shows the longer running versions. All execution times are reported in seconds. The first column shows the benchmark name. The second and third columns show the number of executed instructions (measured on our VP) and LOC of the benchmark, respectively. The fourth column (PULPino) shows the execution time for running the benchmark on the PULPino platform (RTL implementation) in the commercial RTL simulator. The remaining columns show the execution time for running the benchmark on our VP with various optimization techniques enabled: no optimization (column: basic), with an instruction proxy using direct memory interface (dmi) for instruction fetching (column: +i_dmi), additionally with a data access proxy using dmi for loading/storing data from/to the (main) memory (column: +d_dmi), additionally with a local time quantum of 10 (column: +q10), 100 (column: +q100) and 1000 (column: +q1000) instruction cycles, respectively.

It can be observed that every optimization technique significantly improves the simulation performance on our VP. We observed a factor of improvement between 6.1x and 7.8x on this benchmark set with all optimization techniques enabled on the longer running benchmarks. Increasing the time quantum further beyond 1000 instruction cycles has only a minor effect on the simulation performance, because the impact of the SystemC thread context switch becomes marginal on the overall execution time. It can be observed that our VP is multiple orders of magnitude faster than the RTL simulation, especially, with optimizations enabled. Our VP executes up to 20 million instructions per second on the longer running benchmarks (between 17.6 and 20.5 millions, depending on the benchmark) on our evaluation system (AMD 2.8 GHz).

VIII. Conclusion

In this paper, we have proposed and implemented the first RISC-V based VP to further expand the RISC-V ecosystem. The VP has been implemented in SystemC and designed as extensible and configurable platform around a RISC-V RV32IM core with a generic bus system employing TLM 2.0 communication. Our VP is very compact, with around 3000 lines of C++ code including all extensions, making it very manageable and thus suitable as foundation for various application areas, including early SW development and analysis of interactions at the HW/SW interface of RISC-V based systems. Finally, our RISC-V VP is fully open source to stimulate further research and development of ESL methodologies.

For future work we consider two different directions: 1) further extend our VP, and 2) verify our VP using verification techniques for SystemC, e.g. [18].

References

[1] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: User-Level ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.

[2] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.

[3] R. Leupers, F. Schirrmeister, G. Martin, T. Kogel, R. Plyaskin, A. Herkersdorf, and M. Vaupel, "Virtual platforms: Breaking new grounds," in *DATE*, 2012, pp. 685–690.

[4] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.

[5] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.

[6] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[7] M. Streubhr, R. Rosales, R. Hasholzner, C. Haubelt, and J. Teich, "ESL power and performance estimation for heterogeneous mpsocs using SystemC," in *FDL*, Sept 2011, pp. 1–8.

[8] K. Grüttner, R. Görgen, S. Schreiner, F. Herrera, P. Peñil, J. Medina, E. Villar, G. Palermo, W. Fornaciari, C. Brandolese, D. Gadioli, E. Vitali, D. Zoni, S. Bocchio, L. Ceva, P. Azzoni, M. Poncino, S. Vinco, E. Macii, S. Cusenza, J. Favaro, R. Valencia, I. Sander, K. Rosvall, N. Khalilzad, and D. Quaglia, "CONTREX: Design of embedded mixed-criticality CONTRol systems under consideration of extra-functional properties," *Microprocessors and Microsystems*, vol. 51, pp. 39 – 55, 2017.

[9] G. Onnebrink, R. Leupers, G. Ascheid, and S. Schürmans, "Black box ESL power estimation for loosely-timed TLM models," in *SAMOS*, July 2016, pp. 366–371.

[10] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "On the application of formal fault localization to automated RTL-to-TLM fault correspondence analysis for fast and accurate VP-based error effect simulation - a case study," in *FDL*, 2016, pp. 1–8.

[11] ——, "Towards early validation of firmware-based power management using virtual prototypes: A constrained random approach," in *FDL*, 2017, pp. 1–8.

[12] "Spike," https://github.com/riscv/riscv-isa-sim, accessed: 2018-05-13.

[13] "RISCV-QEMU," https://github.com/riscv/riscv-qemu, accessed: 2018-05-13.

[14] "RV8," https://rv8.io, accessed: 2018-05-13.

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[16] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, "Socrocket - A virtual platform for the European Space Agency's SoC development," in *ReCoSoC*, 2014, pp. 1–7.

[17] "Calling convention," https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf, accessed: 2018-05-13.

[18] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Verifying SystemC using intermediate verification language and stateful symbolic simulation," *TCAD*, 2018.