Chapter 3
An Open-Source RISC-V Evaluation Platform

Enormous innovations are enabled by today's embedded systems, in particular the *Internet-of-Things* (IoT) applications where 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. On the other hand, 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* [218, 219]. 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) [139] at the abstraction of *Electronic System Level* (ESL) [12]. In industrial practice, the standardized C++-based modeling language SystemC and *Transaction Level Modeling* (TLM) techniques [71, 113] are being heavily used together to create VPs. Depending on the specific use case, advanced state-of-the-art SystemC-based techniques beyond

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

V. Herdt et al., *Enhanced Virtual Prototyping*,
https://doi.org/10.1007/978-3-030-54828-5_3

functional modeling (see e.g., [76, 80, 89, 91, 162, 169, 200]) are to be applied on top of the basic VPs. The much earlier availability and the significantly faster simulation speed in comparison to RTL are among the main benefits of SystemC-based VPs.

First, in Sect. 3.1 we propose a 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 32- and 64-bit RISC-V core supporting the RISC-V standard instruction set with different privilege levels, the RISC-V-specific CLINT, and PLIC interrupt controllers and an essential set of peripherals. Furthermore, we support simulation of (mixed 32- and 64-bit) multi-core platforms, provide SW debug and coverage measurement capabilities, and support the FreeRTOS and Zephyr operating systems. We further demonstrate the extensibility and configurability of our VP by three examples: addition of a sensor peripheral, describing the integration of the GDB SW debug extension, and configuration to match the RISC-V HiFive1 board from SiFive. Our evaluation demonstrates the quality and applicability of our VP to real-world embedded applications and shows the high simulation performance of our VP. Finally, our RISC-V VP is fully open source[1] (MIT license) to stimulate further research and development. The approach has been published in [95, 96, 106].

Then, in Sect. 3.2 we propose an efficient core timing model and integrate it into the RISC-V VP core, to enable fast and accurate performance evaluation for RISC-V-based systems. Our timing model is attached to the core using a set of well-defined interfaces that decouple the functional from the non-functional aspects and enable easy customization of the timing model. This is very important, because the timing strongly depends on the actual system, e.g., the structure of the execution pipeline in the CPU core or the use of caches. Our interface and timing model allows to consider pipelining, branch prediction, and caching effects. This feature set makes our core timing model suitable for a large set of embedded systems. Our timing model essentially works by observing the executed instructions and memory access operations at runtime. A previous static analysis of the binary or access to the source code is not required. As a case study, we provide a timing configuration matching the RISC-V HiFive1 board from SiFive. The HiFive1 core combines a five-level execution pipeline with an instruction cache and a branch prediction unit. Our experiments demonstrate that our approach allows to obtain very accurate performance evaluation results (less than 5% mismatch on average compared against a real HW board) while still retaining a high simulation performance (less than 20% performance overhead on average for the VP simulation). We use the *Embench* benchmark suite for evaluation, which is a modern benchmark suite tailored for embedded systems. Our core timing model including the integration with the open-source RISCV VP is available as open source too in order to further

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

advance the RISC-V ecosystem and stimulate research. The approach has been published in [104].

3.1 RISC-V-Based Virtual Prototype

The RISC-V ecosystem already has various high-speed ISSs such as the reference simulator Spike [197], RISCV-QEMU [185], RV8 [186], or DBT-RISE [42]. They are mainly designed to simulate as fast as possible and 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 [18], 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. Renode [176] is another full-system simulator with RISC-V support. Renode puts a particular focus on modeling and debugging multi-node networks of embedded systems. However, both gem5 and Renode employ a different modeling style and thus hinder the integration of advanced SystemC-based techniques.

FORVIS [58] and GRIFT [67] are Haskell-based implementations that aim to provide an executable formalization of the RISC-V ISA to be used as foundation for several (formal) analysis techniques. SAIL-RISCV [184] aims to be another RISC-V formalization that is implemented in Sail, which is a special language for describing ISAs with support for generation of simulator back-ends (in C and OCaml) as well as theorem-prover definitions.

The project SoCRocket [189] 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.

In the following we present our open-source RISC-V based VP that is implemented in SystemC TLM and attempts to close the gap on virtual prototyping in the RISC-V ecosystem.

3.1.1 RISC-V-Based VP Architecture

The VP is implemented in SystemC and designed as extensible and configurable platform around a RISC-V RV32/64IMAC (multi-)core with a generic bus system employing TLM 2.0 communication and support for the GNU toolchain with SW coverage measurement (GCOV) and debug capabilities (GDB). Overall, the VP consists of around 12,000 lines of C++ code with all extensions. Figure 3.1 shows an overview of the VP architecture. In the following we present more details.

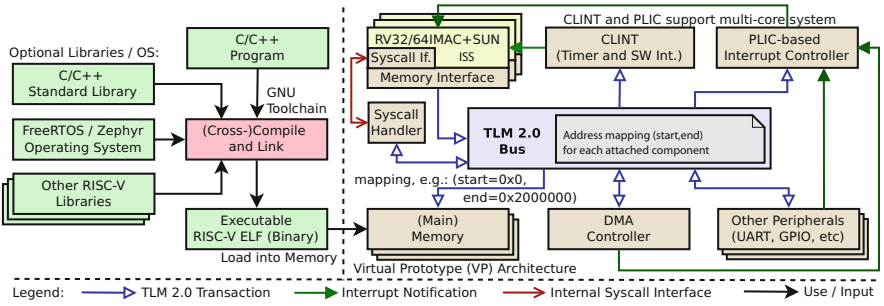

Fig. 3.1 Virtual prototype architecture overview

3.1.1.1 RV32/64 (Multi-)Core

The CPU core loads, decodes, and executes one instruction after another. RISC-V compressed instructions are expanded on the fly into regular instructions in a pre-processing step before being passed to the normal execution unit. We provide a 32-bit and 64-bit core supporting the RISC-V RV32IMAC and RV64IMAC instruction set, respectively. Besides the mandatory Machine mode, each core implements the RISC-V Supervisor and User mode privilege levels and provides support for user mode interrupt and trap handling (N extension). This includes the *Control and Status Register* (CSR) for the corresponding privilege levels (as specified in the RISC-V privileged architecture specification [219]) as well as instructions for interrupt handling (*wfi*, *m/s/uret*) 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.

Multiple RISC-V cores can be integrated to build a multi-core platform. It is also possible to mix 32- and 64-bit cores. The Atomic extension provides instructions to enable synchronization of the cores. Each core is attached to the bus through a memory interface. Essentially, the memory interface translates load/store requests into TLM transactions and ensures that the atomic instructions are handled correctly. We will provide more details on simulation of multi-core platforms in Sect. 3.1.4.

3.1.1.2 TLM 2.0 Bus

The TLM bus is responsible for 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 master and 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.

3.1.1.3 Traps and Interrupts

Both traps and interrupts result in the CPU core performing a context switch to the trap/interrupt handler (based on a SW configurable address stored in the *MTVEC* CSR). Traps are raised to perform a system call or when an execution exception, e.g., invalid memory access, is encountered.

Two sources of interrupts are available: (1) local and (2) external. Essentially, there are two sources of local interrupts: SW as well as timer interrupts generated by the RISC-V-specific CLINT (Core Local INTerruptor). The timer is part of the CLINT, and the interrupt frequency can be configured for each core through memory mapped I/O. CLINT also provides a memory mapped register for each core to trigger a SW interrupt for the corresponding core. External interrupts are all remaining interrupts triggered by the various components in the system. To handle external interrupts, we provide the RISC-V-specific PLIC (Platform Level Interrupt Controller). PLIC will collect and prioritize all external interrupts and then route them to each CPU core one by one. The core that claims the interrupt first will process it. According to the RISC-V specification, external interrupts are processed with higher priority than local interrupts, and SW interrupts are higher prioritized than timer interrupts. We will describe the interrupt handling process in more detail in Sect. 3.1.2.

3.1.1.4 System Calls

The C/C++ library defines a set of system calls as abstraction from the actual execution environment. For example, the *printf* function performs the formatting in platform independent C code and finally invokes the *write* system call with a fixed char array. Typically, an embedded system provides a trap handler that redirects the *write* system call to a UART/terminal component.

We also provide a *SyscallHandler* component to emulate system calls of the C/C++ library by redirecting them to the simulation host system. Our emulation layer for example allows us to open and read/write from/to files of the host system. We use this functionality for example to support SW coverage measurement with GCOV.

The syscall handler can be called in one of two ways: (1) through a trap handler that redirects the system call to the syscall handler from SW using memory mapped I/O (this approach enables a flexible redirection of selected system calls), or (2) directly intercept the system call (i.e., the RISC-V *ECALL* instruction) in the CPU core, instead of jumping to the trap handler. The behavior is configurable per core. We will describe the system call handling process in more detail in Sect. 3.1.2.

3.1.1.5 VP Initialization

The main function in the VP is responsible to instantiate, initialize, and connect all components, i.e., set up the architecture. An ELF loader is provided to parse and load an executable RISC-V ELF file into the memory and set up the program counter in the CPU core accordingly. Finally, the SystemC simulation is started. The ELF file is produced by the GNU toolchain by (cross-)compiling the application program and optionally linking it with the C/C++ standard library or other RISC-V libraries. We also support a bare-metal execution environment without any additional libraries and test our VP to work with the FreeRTOS and Zephyr operating systems.

3.1.1.6 Timing Model

We provide a simple and configurable instruction-based timing model in the core, and by following the TLM 2.0 communication standard, transactions can be annotated with optional timing information to obtain a more accurate timing model of the executed software. We also provide a plugin interface to integrate external timing information into the core, and we are working on integrating more accurate timing models.

3.1.2 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.

3.1.2.1 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 bootstrap code to initialize interrupt handling and describe how interrupts are processed in more detail. Later

```
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. 3.2 Example application running on the VP to demonstrate the hardware/software interaction

in Sect. 3.1.5.1 we present the corresponding SystemC-based sensor implementation in our VP.

Software Side Figure 3.2 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 I/O. 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 Lines 29–30 using memory mapped I/O. The

```
1  .globl _start
2  .globl main
3  .globl level_1_interrupt_handler
4
5  _start:
6  la t0, level_0_interrupt_handler
7  csrw mtvec, t0 # register interrupt/trap handler
8  csrsi mstatus, 0x8 # enable interrupts in general
9  li t1, 0x888
10 csrw mie, t1 # enable external/timer/SW interrupts
11 jal main
12
13 # stop simulation with the *exit* system call
14 li a7, 93 # syscall exit has number 93
15 li a0, 0 # argument to exit
16 ecall # RISC-V system call
17
18 level_0_interrupt_handler:
19 # ... store registers on the stack if necessary ...
20 csrr a0, mcause
21 jal level_1_interrupt_handler
22 # ... re-store registers in case they have been saved ...
23 mret # return from interrupt/trap handler
```

Fig. 3.3 Bare-metal bootstrap code demonstrating interrupt handling

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 (Lines 32–33) before the program terminates. Each iteration starts by waiting for sensor data (Lines 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 that the *wfi* instruction will power down the CPU core until the next interrupt occurs.

Bootstrap Code and Interrupt Handling Figure 3.3 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*, *mstatus*, *mie*, and *mcause* are CSRs that essentially store the interrupt handler address, core status information, enabled interrupts, and interrupt source, respectively. The instructions *csrr* and *csrw* read and write a CSR into and from a normal CPU register, respectively. The instruction *csrsi* sets the bits in the CSR based on the provided immediate value. Before the main function is called (Line 11), the interrupt handler base address (level-0) is stored in *mtvec* (Lines 6–7) and all interrupts are enabled (Lines 8–10). After the

[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.

main function returns, the exit system call is invoked to terminate the VP simulation (Lines 14–16). We provide more details on system calls in the next section.

In general, an interrupt can occur at any time during execution of the application SW. All interrupts propagate to the PLIC (i.e., the RISC-V interrupt controller) first and are prioritized there. The CPU core only receives a notification that some interrupt is pending and needs to be processed. The core will prepare the interrupt by storing the program counter into the *mepc* CSR and setting the *mcause* CSR appropriately (to denote an interrupt in this case). The core then reads the base address from the *mtvec* CSR and sets the program counter to that address, i.e., effectively directly jumping to the level-0 interrupt handler (first instruction at Line 20). The interrupt handler (level-0) first in Line 20 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 [179] stores the first argument of a function call. Then in Line 21 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 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 (Lines 11–13 in Fig. 3.2, this step is ignored if none has been registered for the interrupt number). Finally, the *mret* instruction restores the previous program counter from the *mepc* CSR. Please note that the level-0 handler typically stores and re-stores the register values by pushing and popping them to/from the stack before/after calling the level-1 handler, respectively.

3.1.2.2 Environment Interaction: Syscall Emulation and C/C++ Library

We provide an emulation layer for executing system calls by redirecting them to the host system running the VP simulation. This requires passing arguments from the guest application into the host system and integrating 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).

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. Figure 3.4 shows the relevant part of a stub that is provided in the RISC-V port of the C library. Essentially, the arguments of the system call are stored in the CPU registers *a0–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

```
1  #define SYS_write 64
2
3  ssize_t write(int fd, const void *buf, size_t count) {
4    return syscall(SYS_write, fd, (long)buf, count, 0);
5  }
6
7  long syscall(long n, long _a0, long _a1, long _a2, long _a3) {
8    // store arguments in CPU register and trigger ecall
9    register long a0 asm("a0") = _a0;
10   register long a1 asm("a1") = _a1;
11   register long a2 asm("a2") = _a2;
12   register long a3 asm("a3") = _a3;
13   register long a7 asm("a7") = n;
14
15   // special RISC-V instruction denoting a system call
16   asm volatile ("ecall" : "+r"(a0) : "r"(a1), "r"(a2), "r"(a3), "r"(a7));
17
18   // store potential error code and return result
19   if (a0 < 0) {
20     errno = -a0;
21     return -1;
22   } else {
23     return a0;
24   }
25 }
```

Fig. 3.4 System call handling stub linked with the C library (guest side, executed on the VP host system). This example listing is based on the RISC-V *newlib* port https://github.com/riscv/riscv-newlib

system as shown in Fig. 3.5.[3] In case of the *write* syscall, a pointer argument *buf* is passed. This is a pointer value from the guest system, i.e., 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 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) can be 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

[3]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 the syscall identifier) and then redirect the write to, e.g., a UART/Terminal component.

```
#define SYS_write 64

// execute syscall on the host system (SyscallHandler)
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: {
      if (intercept_syscalls_option) {
        // intercept and redirect syscall to host system
        regs[a0] = execute_syscall(regs[a7], regs[a0], regs[a1], regs[a2], regs[a3]);
      } else {
        // jump to SW trap handler (let SW decide how to handle syscall)
        // SW will either direct the syscall to a peripheral or our SyscallHandler
            component (which is what the core does directly when intercepting
            syscalls)
        raise_trap(EXC_ECALL);
      }
    } break;
    //...
  }
}
```

Fig. 3.5 Concept of system call execution on the VP, either redirects to the host system or takes trap

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.

3.1.3 VP Performance Optimizations

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 sections.

3.1.3.1 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.

3.1.3.2 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 would 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 any 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.

3.1.4 Simulation of Multi-Core Platforms

Our VP provides support for simulation of RISC-V multi-core platforms. We support to instantiate and mix multiple 32- and 64-bit cores. Each core is attached to the bus using a (local TLM) memory interface. Furthermore, each core is assigned a unique identifier, starting with zero, during instantiation. This core identifier is denoted as *hart-id*. Access to the hart-id is provided through the read-only SW accessible *mhartid* CSR. Based on the hart-id, the SW can control the execution for each core. Furthermore, the SW is using atomic instructions for synchronization.

In the following we show an example RISC-V multi-core SW for illustration and provide more details on the implementation of the RISC-V atomic ISA extension.

3.1.4.1 Example Bare-Metal Multi-Core SW

Figure 3.6 shows an example bare-metal SW that demonstrates the multi-core simulation concept from the SW side. For illustration purposes, we assume that the platform consists of two cores. Both cores start at the same time and use the same entry point (Line 5, i.e., the *_start* label). By reading the *mhartid* CSR, the SW obtains the id, either 0 or 1, of the executing core (Line 6). The core id is stored in register a0. Based on the id, the SW control flow is manipulated. Each core is initializing its stack pointer (sp) address to a separate area (Lines 13 and 16). Finally, an external main function is called (Line 20) with the core id passed as first argument (according to the RISC-V calling convention, the first argument is provided in register a0). The code after the main function ensures that only the last core can proceed to the exit system call (Lines 32–34) and stop the simulation. The first core that returns from the main function keeps spinning in the loop at Line 29. This synchronization mechanism is achieved by using an AMO instruction to read and increment a shared counter.

3.1.4.2 Implementation of the Atomic ISA Extension

Figure 3.7 shows the relevant part of the memory interface that shows how support for atomic instructions is implemented. Please note that each core has its own separate memory interface. As already discussed in the preliminaries Sect. 2.2.2, the A instruction set extension provides two types of instructions: (1) AMO and (2) LR/SC. In the following we provide more details on how to implement these instructions and for illustration we will refer to Fig. 3.7.

```
1  .globl _start
2  .globl core_main
3
4  # NOTE: each core will start here with execution
5  _start:
6  csrr a0, mhartid # return a core specific number 0 or 1
7  li t0, 0
8  beq a0, t0, core0
9  li t0, 1
10 beq a0, t0, core1
11 # initialize stack for core 0 and core 1
12 core0:
13 la sp, stack0_end # code executed only by core 0
14 j end
15 core1:
16 la sp, stack1_end # code executed only by core 1
17 end:
18
19 # function argument stored in register a0 (according to RISC-V calling convention)
20 jal core_main
21
22 # wait until all two cores have finished
23 la t0, exit_counter
24 li t1, 1
25 li t2, 1
26 amoadd.w a0, t1, 0(t0) # get current counter value and increase existing value
27 # the first core reaching this point will spin
28 1:
29 blt a0, t2, 1b # jump one label backwards in case a0 < t2
30
31 # stop whole simulation with the *exit* system call
32 li a7, 93 # syscall exit has number 93
33 li a0, 0 # argument to exit
34 ecall # RISC-V system call
35
36 stack0_begin:
37 .zero 32768 # allocate 32768 zero-initialized bytes in memory
38 stack0_end:
39 stack1_begin:
40 .zero 32768
41 stack1_end:
42 exit_counter:
43 .word 0 # allocate 4 zero-initialized bytes in memory
```

Fig. 3.6 Bare-metal bootstrap code for a multi-core simulation with two cores

AMO Instructions To execute an AMO instruction, the core has to perform a load (Line 1) and store (Line 6) operation atomically without intervening memory access operations of other cores. We ensure the atomic execution property by locking the bus during the *amo* instruction. Therefore, a shared lock is acquired by the core's memory interface before the load operation (Line 2) and released again after the store (Line 52) operation. In case the bus is already locked, the lock operation will wait until the lock is released. Before performing a memory access operation, each

core waits until it has obtained access rights (Line 35), i.e., the bus is not locked or is locked by the core itself. This locking scheme also supports DMI operations (Lines 39–45).

Peripherals that have write access to the bus (e.g., the DMA controller) are attached through a 1-to-1 TLM interconnect to the bus in order to ensure that they respect the bus locking. The peripheral interconnect transparently forwards all peripheral write transactions but waits in case the bus is locked by any core. Once the bus lock is released, all waiting (SystemC) processes are notified using a (SystemC) event.

LR/SC Instructions To execute an LR instruction, the core (memory interface) tracks a reservation on the load address and acquires the shared bus lock (Lines 12–13). The lock is kept acquired while *"forward-progress"* (see preliminaries Sect. 2.2.2) is maintained by the core. Essentially, the lock is released in case:

- More than 16 instructions are executed.
- A trap (exception) or interrupt is taken.
- A store is performed by the core holding the lock.

The SC instruction succeeds in case the lock is still acquired (Line 19) and a reservation on the store address exists (Line 20). In any case, the bus lock is released by the memory interface after executing the SC instruction (Line 24 or Line 52).

Similar to the AMO instructions, the shared bus lock ensures that other cores (and peripheral bus masters) do not interfere with the LR/SC instruction sequence execution and hence the LR/SC sequence eventually succeeds when maintaining *"forward-progress."*

3.1.5 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. Furthermore, we provide a 32- and 64-bit core that can also be integrated into a multi-core platform. 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 information to obtain a more accurate timing model of the executed software. Support for additional RISC-V ISA extensions (beyond IMAC) can be added inside the CPU core by extending the decode and execute functions accordingly. Our VP already provides a large feature set with large potential for additional extensions, which makes our VP suitable as foundation for different application areas.

In the following, we demonstrate the extensibility and configurability of our VP by three concrete examples: addition of a sensor peripheral, extension of a GDB-

```
int32_t atomic_load_word(uint64_t addr) {
  bus_lock->lock(get_hart_id());
  return load_word(addr);
}

void atomic_store_word(uint64_t addr, uint32_t value) {
  assert (bus_lock->is_locked(get_hart_id() ));
  store_word(addr, value);
}

int32_t atomic_load_reserved_word(uint64_t addr) {
  bus_lock->lock(get_hart_id());
  lr_addr = addr; // reservation for the load address on the whole memory
  return load_word(addr);
}

bool atomic_store_conditional_word(uint64_t addr, uint32_t value) {
  /* The lock is established by the LR instruction and the lock is kept while "forward-progress" is maintained. */
  if (bus_lock->is_locked(get_hart_id() )) {
    if (addr == lr_addr) {
      store_word(addr, value);
      return true; // SC succeeded
    }
    bus_lock->unlock();
  }
  return false; // SC failed
}
void store_word(uint64_t addr, uint32_t value) {
  store_data(addr, value);
}

template <typename T>
inline void store_data(uint64_t addr, T value) {
  // only proceed if the bus is not locked at all or is locked by this core
  bus_lock->wait_for_access_rights(get_hart_id() );

  // check if this access falls within any DMI range
  bool done = false;
  for (auto &e : dmi_ranges) {
    if (e.contains(addr)) {
      quantum_keeper.inc(e.dmi_access_delay );
      *(e.get_mem_ptr_to_global_addr<T>(addr )) = value;
      done = true;
    }
  }

  // otherwise (no DMI), perform a normal transaction routed through the bus
  if (!done)
    do_transaction(tlm::TLM_WRITE_COMMAND, addr, (uint8_t *)&value, sizeof(T));

  // do nothing in case the bus is not locked by this hart
  bus_lock->unlock(get_hart_id());
}

// ... other load/store functions and load_data similar ...
```

Fig. 3.7 Core memory interface with atomic operation support

based SW debug functionality, and configuration matching the RISC-V HiFive1 board from SiFive.

3.1.5.1 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 Sect. 3.1.2.1. It shows the principles in modeling peripherals and extending our VP as well as demonstrates the TLM communication and basic SystemC-based modeling and synchronization.

```
1  struct SimpleSensor : public sc_core::sc_module {
2    tlm_utils::simple_target_socket<SimpleSensor> tsock;
3
4    std::shared_ptr<InterruptController> IC;
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     // implementation available in Fig. 3.9
36   }
37
38   void run() {
39     while (true) {
40       run_event.notify(sc_core::sc_time(scaler, sc_core::SC_MS));
41       sc_core::wait(run_event); // 40 times per second by default
42
43       // fill with random data
44       for (auto &n : data_frame) {
45         if (filter == 1) {
46           n = rand() % 10 + 48;
47         } else if (filter == 2) {
48           n = rand() % 26 + 65;
49         } else {
50           // fallback for all other filter values: random printable
51           n = rand() % 92 + 32;
52         }
53       }
54
55       IC->trigger_interrupt(irq_number);
56     }
57   }
58 };
```

Fig. 3.8 SystemC-based configurable sensor model that is periodically filled with random data—demonstrates the basic principles in modeling peripherals

The sensor is instantiated in the main function of the VP alongside other components and is attached to the TLM bus.

The sensor implementation is shown in Fig. 3.8. The sensor model has a data frame of 64 bytes that is periodically updated (overwritten with new data, Lines 44–53) 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 value, this thread is periodically unblocked (Line 40) 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

```
59  void transport(
        tlm::tlm_generic_payload &trans,
        sc_core::sc_time &delay) {
60    auto addr = trans.get_address();
61    auto cmd = trans.get_command();
62    auto len = trans.get_data_length();
63    auto ptr = trans.get_data_ptr();
64
65    if (addr >= 0 && addr <= 63) {
66      // access data frame
67      assert (cmd ==
            tlm::TLM_READ_COMMAND);
68      assert ((addr + len) <=
            data_frame.size());
69
70      // return last generated random
            data at requested address
71      memcpy(ptr, &data_frame[addr],
            len);
72    } else {
73      assert (len == 4); // NOTE: only
            allow to read/write whole
            register
74
75      auto it = addr_to_reg.find(addr);
76      assert (it != addr_to_reg.end());
            // access to non-mapped
            address
77
78      // trigger pre read/write actions
79      if ((cmd ==
            tlm::TLM_WRITE_COMMAND) &&
            (addr == SCALER_REG_ADDR)) {
80        uint32_t value =
              *((uint32_t*)ptr);
81        if (value < 1 || value > 100)
82          return; // ignore invalid values
83      }
84
85      // actual read/write
86      if (cmd == tlm::TLM_READ_COMMAND)
            {
87        *((uint32_t *)ptr) = *it->second;
88      } else if (cmd ==
            tlm::TLM_WRITE_COMMAND) {
89        *it->second = *((uint32_t *)ptr);
90      } else {
91        assert (false && "unsupported
              tlm command for sensor
              access");
92      }
93
94      // trigger post read/write actions
95      if ((cmd ==
            tlm::TLM_WRITE_COMMAND) &&
            (addr == SCALER_REG_ADDR)) {
96        run_event.cancel();
97        run_event.notify(sc_core::sc_time(
              scaler, sc_core::SC_MS));
98      }
99    }
100 }
```

Fig. 3.9 The *transport* function for the example in Fig. 3.8

SW. Therefore, the sensor has a reference to the interrupt controller (IC, Line 4) and an interrupt number provided during initialization (Lines 23 and 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 (Lines 60–61), the action is selected. It will either read (part of) the data frame (Line 71) or read/write one of the configuration registers (Lines 86–92). In case of a register access, a pre-read/write validation and post-read/write action can be defined as necessary. In this example, the sensor will ignore invalid scaler values (Lines 79–83) and reset the data generation thread on a scaler update (Lines 95–98). 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.

3.1.5.2 SW Debugging Support Extension

In this section we describe our VP extension to provide SW debug capabilities in combination with (for example) the Eclipse IDE by implementing the GDB *Remote Serial Protocol* (RSP) interface. Debugging for example enables to step through the SW line by line, set (conditional) breakpoints, obtain and even modify variable values, and also display the RISC-V disassembly (with the ability to step through the disassembly). Debugging can also be extremely helpful on the VP level to investigate errors, due to the deterministic and reproducible SW execution on the VP.

Using the GDB RSP interface, our VP acts as server and the GDB[4] 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 denotes a memory read, in this case read 0×4 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 with little endian byte order). 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 local host 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.

Please note that 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

[4] In particular the freely available RISC-V port of the GDB, which knows about the available RISC-V register set, the CSRs, and can provide a disassembly of the RISC-V instruction set.

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 nine can simply return an empty packet and a few more some predefined 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).

3.1.5.3 HiFive1 Board Configuration

As an example, we provide the configuration matching the HiFive1 board from SiFive [108]. The HiFive1 is based around a SoC that integrates a single RISC-V RV32IMAC core with several peripherals and memories. Interrupts are processed by the CLINT and PLIC peripherals following the RISC-V ISA specification. The PLIC supports 53 interrupt sources with 7 priority levels. We configured our PLIC peripheral to match this specification. A (SPI) non-volatile flash memory is provided to store the application code and a small writable memory (DTIM) to hold the application data. The application data is initialized during the boot process by copying relevant data (e.g., initialized global variables) from the flash memory into the DTIM. This initialization code is embedded in the application binary that is placed in the flash memory. GPIOs and an UART are provided for communicating with the environment. We redirect each write access to the UART to stdout. We used our register modeling layer to create these additional peripheral models in SystemC. For the GPIO peripheral, we provide a (re-usable) interface to access an environment model. We provide more details on this interface in the following section.

GPIO Environment Interaction Interface GPIOs are used to connect the embedded system to the outside world. Each GPIO pin can be configured to serve as output or input connection. Input pins can trigger an interrupt when being written.

We integrate a GPIO-(TCP) server to provide access to the GPIO pins in order to attach an environment model. The server runs in a separate system thread and hence needs to be synchronized with the SystemC kernel. Therefore, we use the SystemC *async_request_update* function to register an update function to be executed on the next update cycle of the SystemC thread. The update function triggers a normal SystemC event notification to wake up the SystemC GPIO processing thread.

Figure 3.10 shows an example virtual environment implemented in C++ using the Qt Framework. It shows the HiFive1 board with a seven-segment display and a button attached to the GPIO pins. The display shows the current counter value, and the button is used to control the count mode (switch between increment and decrement). We have also executed the same RISC-V ELF binary on the real HiFive1 board using the same setup as shown in the virtual environment.

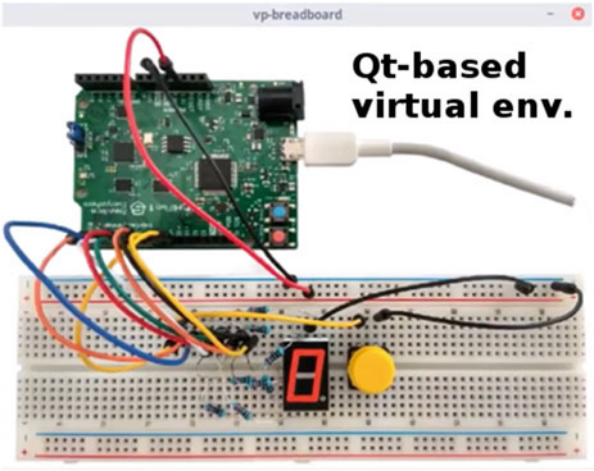

Fig. 3.10 Qt-based virtual environment, showing the HiFive1 board with a seven segment display (output) and a button (input), attached to the VP simulation through a TCP connection

3.1.6 VP Evaluation

In this section we first describe how we have tested our VP to evaluate and ensure the VP quality, then we present results of a performance evaluation.

3.1.6.1 Testing

Testing is very important to ensure that the VP is working correctly. In the following we describe how we have tested our VP. In particular, we used

1. The official RISC-V ISA tests [181], in particular the RV32/64IMAC tests.
2. Use the RISC-V Torture test-case generator [182], also targeting the RV32/64IMAC instruction set.
3. Employ state-of-the-art coverage-guided fuzzing (CGF) techniques for test-case generation. For more information on this approach, please refer to [102].
4. Several example applications, ranging from bare-metal applications to examples using the FreeRTOS [60] and Zephyr [224] operating systems. We observe if the example applications behave as expected.

Please note that the RISC-V ISA tests are directed tests that are handwritten and already encode the expected result inside of the test. Hence, no reference simulation is required for the ISA tests.

Figure 3.11 shows an overview of the Torture and CGF approach. They both work by generating a set of tests, i.e., each testcase is a RISC-V ELF binary, which is then executed one after another on our VP and (one or multiple) reference simulators. We

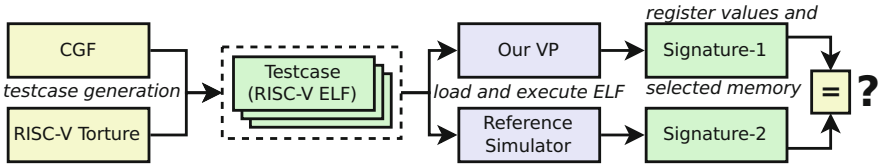

Fig. 3.11 Overview on the RISC-V Torture and CGF approach for VP testing

use the official RISC-V reference simulator Spike [197] in this evaluation. We have extended our VP to dump the execution result, called *signature*, as supported by Spike. The *signature* contains the register values and selected memory contents. After each execution, the dumped signatures are compared for differences. Hence, the simulator is considered as a black box and no intrusive modifications are required for this testing.

The RISC-V ISA tests as well as the Torture and Fuzzer test generation approaches primarily focus on testing the core. The example applications however also tend to analyze larger portion of the whole VP platform. They further integrate the CLINT and PLIC interrupt controller as well as selected peripherals that are required for the particular application. In the following we provide more details on some selected applications that demonstrate the applicability of our VP for real-world embedded applications.

Zephyr Examples The Zephyr OS is designed around a small-footprint kernel primarily targeting (resource constrained) embedded systems. Zephyr supports multiple architectures, including RISC-V. For the RISC-V architecture, support for the HiFive1 board is available. We used the HiFive1 board configuration of our VP to run several Zephyr example applications that extensively use core components of the kernel, including threads, timers, semaphores, and message queues. In addition, we run examples that perform aes128 and sha256 encryption/decryption using the TinyCrypt library.

FreeRTOS Examples Similar to Zephyr, FreeRTOS is also designed around a kernel component, targets embedded systems, and provides support for the RISC-V architecture. We also build several example applications that create multiple threads with different priorities, use queues for data passing, and integrate interrupts. In addition, we created two applications that use the FAT and UDP library extensions of FreeRTOS. The first application formats an SD card by creating a new MBR and writing an FAT partition. The second application sends/receives a set of UDP packets using our ethernet peripheral to/from a remote computer.

3.1.6.2 Performance Evaluation

In our paper [96] we have already demonstrated: (1) that our VP provides more than 1000x faster simulation performance compared to an RTL implementation,

and (2) the effectiveness of our presented VP simulation performance optimization techniques (between 6.1x and 7.8x improvement on the considered benchmark set). This book presents an updated performance evaluation of our VP on a more recent and faster simulation host. We use a set of different applications to demonstrate the execution performance, measured in MIPS (Million Instructions Per Second), of our VP. Furthermore, we provide a performance comparison to other RISC-V simulators, in particular: FORVIS, SAIL (the C simulator back-end in particular), gem5, SPIKE, and QEMU

Experiment Setup and VP Results All experiments are performed on a Fedora 29 Linux system with an Intel Xeon Gold 5122 processor with 3.6 GHz. The memory limit is set to 32 GB and the timeout to 4 h (i.e., 14,400 s). Our VP is compiled with GCC 8.2.1 with -O3 optimization.

Table 3.1 shows the results. The first six columns show the benchmark name (column: *Benchmark*), number of executed instructions (column: *#instr*), *Lines of Code* (LoC) in C (column: *C*) and in assembly (column: *ASM*), the aforementioned MIPS (column: *MIPS*) performance metric, and the simulation time (column: *time*) for our VP. The remaining five columns show the simulation time for the other RISC-V simulators. All simulation times are reported in seconds. M.O. denotes a memory out and N.S. that the benchmark is not supported by the simulator. We consider benchmarks from different application areas:

- *dhrystone* is a synthetic computing benchmark designed to measure the general (integer) execution performance of a CPU. We execute 10,000,000 iterations of the algorithm.
- *qsort* is an efficient sorting algorithm. We use a standard implementation to sort an array with 50,000,000 elements.
- *fibonacci* is a small program implemented in assembler that performs 1,000,000,000 iterations and ignores any integer overflow.
- *zephyr-crypto* uses two threads that communicate through a (single-element) message queue. The first thread encrypts an ~1 MB data set, using the aes128 algorithm of the TinyCrypt library, the second thread decrypts it again.

Table 3.1 Experiment results—all execution times reported in seconds and number of executed instructions (#instr) reported in Billions (B)

		LoC		VP		Other simulators (Time)				
Benchmark	#instr	C	ASM	MIPS	Time	FORVIS	SAIL	gem5	SPIKE	QEMU
dhrystone	4.06 B	362	2 212	42.7	94.86	M.O.	10 986.24	1 170.08	13.90	3.53
qsort	2.93 B	146	2 279	42.8	68.71	M.O.	T.O.	985.99	11.01	3.04
fibonacci	5.99 B	/	17	53.9	111.15	13 954.57	12 830.05	1 447.37	17.19	1.94
zephyr-crypto	2.52 B	86	5 407	46.6	54.11	N.S.	N.S.	N.S.	N.S.	6.50
mc-ivadd	2.39 B	26	1 798	44.4	53.76	N.S.	N.S.	N.S.	48.11	N.S.

MIPS = Millions Instructions Per Second. LoC = Lines of Code in C and assembly (ASM). M.O. = Memory Out (32GB limit). T.O. = Time Out (4h = 14,400 s limit). N.S. = Not Supported

- *mc-ivadd* is a multi-core benchmark that performs a vector addition and stores the result in a new vector. Each core operates on a different part of the vector. The vector size is set to 4,194,304 and the algorithm performs 30 iterations.

We report results for using our VP with an RV32 core (four RV32 cores for the multi-core benchmark). However, similar results are obtained when using an RV64 core(s).

It can be observed that our VP provides a high simulation performance between 42 and 53 MIPS with an average of 46 MIPS on our benchmark set, which demonstrates the applicability of our VP to real-world embedded applications. The pure assembler program (fibonacci) achieves the highest simulation performance, since it does not require to perform memory access operations (besides instruction fetching). It can also be observed that the additional synchronization overhead (in the SystemC simulation) to perform a multi-core simulation has no significant performance impact[5] Please note that, for the multi-core simulation benchmark, we report the total MIPS for all four cores.

Comparison with Other Simulators Compared to the other RISC-V simulators (right side of Table 3.1), our VP shows very reasonable performance results and is located in the front midfield. As expected, our VP is not as fast as the high-speed simulators SPIKE and in particular QEMU. The reason is the performance overhead of the SystemC simulation kernel and the more detailed simulation of our VP.[6] However, our VP is much faster than FORVIS and SAIL as well as gem5, which arguably is the closest to our VP in terms of the intended use cases. Please note that the *zephyr-crypto* and *mc-ivadd* benchmarks are not supported (N.S.) on some simulators due to missing support for the Zephyr operating system and the multi-core RISC-V test environment, respectively. In the following we discuss the results in more detail.

Compared to QEMU, the performance overhead is most strongly pronounced on the *fibonacci* benchmark. The reason is that the benchmark iterates a single basic block and only performs very simple operations (mostly additions) without memory accesses. This has two implications: This single basic block is pre-compiled once into native code using DBT and then reused for all subsequent iterations (hence QEMU executes at near native performance). Furthermore, since only simple operations are used, the simulation overhead induced by SystemC has a comparatively strong influence on the overall performance.

On more complex benchmarks such as *qsort* and *zephyr-crypto*, the overhead to QEMU is less strongly pronounced. These benchmarks have a much more complex control flow and hence require to compile several basic blocks and also use more

[5]Please note that the SystemC kernel is not using a multi-threaded simulation environment but executes one process (i.e., one core) at a time and switches between the processes.

[6]This includes more accurate timing by leveraging SystemC, instruction accurate interrupt handling, and the ability to integrate TLM-2.0 memory transactions. Furthermore, compared to QEMU we currently do not integrate DBT (Dynamic Binary Translation).

complex instructions that take longer time for (native) execution (compared to a simple addition). Furthermore, *zephyr-crypto* performs several simulated context switches between the Zephyr OS and the worker threads, which has additional impact on the simulation performance of QEMU (since it causes an indirect and non-regular transfer between the basic blocks).

Compared to SPIKE, the performance overhead is mostly uniformly distributed (since SPIKE does not use DBT either) on the single core benchmarks (with SPIKE being around 6.2x–6.8x faster than our VP). The performance on the multi-core benchmark *mc-ivadd* however is very similar for both SPIKE and VP. Apparently, SPIKE is very strongly optimized for simulation of single core systems and hence the newly introduced synchronization and context switch overhead is very significant.

Compared to gem5, our VP is significantly faster (between 12.3x to 14.4x). The primary use case of gem5 is also not a pure functional simulation (which is the goal of SPIKE and QEMU) but rather an architectural exploration and analysis. Thus, gem5 provides more detailed processor and memory models with complex interfaces, which in principle can also be extended for accurate extra-functional properties. Furthermore, gem5 is a large (and aims to be a rather generic) platform, which supports different architectures besides RISC-V, which causes additional performance overhead.

Compared to FORVIS and SAIL, our VP is consistently much faster (up to two orders of magnitude). The reason is that these simulators have been designed with a different use case in mind (establishing an executable formal representation of the RISC-V ISA) and hence very fast simulation performance is only a secondary goal. Furthermore, FORVIS runs into memory outs (M.O.), which may indicate a memory leak in the implementation. SAIL has a time out (T.O.) on the *qsort* benchmark.[7]

In summary, the simulation performance very strongly depends on the simulation technique, which in turn depends on the primary use case of the simulator. The goal of our VP is to introduce an (open-source) implementation that leverages SystemC TLM-2.0 into the RISC-V ecosystem to lay the foundation for advanced SystemC-based system-level use cases. Overall, our VP provides a high simulation performance with an average of 46 MIPS on our benchmark set.

3.1.7 Discussion and Future Work

Our RISC-V based VP is implemented in SystemC TLM 2.0 and already provides a significant set of features, which makes our VP suitable as foundation for various application areas, including early SW development and analysis of interactions at

[7] A 10x smaller version of *qsort* was completed successfully within 700.22 s on SAIL, and hence this time out may also indicate a memory-related problem in SAIL (since *qsort* requires a significant amount of memory for the large array to be sorted).

the HW/SW interface of RISC-V-based systems. Recently, we extended our VP to provide support for RISC-V floating-point extensions and integrated an MMU (Memory Management Unit) to support virtual memory and memory protection. This, extensions enabled our VP to boot the Linux OS. Nonetheless, our VP can still be further improved. In the following we sketch different directions that we consider for future work.

One direction is the extension of our VP with new components and integration of additional RISC-V ISA extensions. In particular, dedicated support for custom instruction set extensions is very important. We plan to look into appropriate specification mechanisms and develop suitable interface to facilitate specification, generation, and integration of custom RISC-V extensions.

Performance optimizations are also very important, in particular to run a whole SW stack including the Linux OS. Two techniques seem very promising: (1) Integration of DBT/JIT (Dynamic Binary Translation/Just-In-Time) techniques to avoid the costly interpreter loop whenever possible, e.g., translate and cache RISC-V basic blocks.[8] (2) Use a real thread for each core in a multi-core simulation. This requires dedicated techniques to synchronize with the SystemC simulation.

While we have already performed an extensive testing of our VP, in particular the core, we plan to consider additional verification techniques and also put a stronger emphasis on verification of peripherals and other IP components. In particular, we plan to consider formal verification techniques for SystemC, e.g., [38, 100, 212], and also investigate (UVM-based) constrained random techniques for test-case generation [222].

Finally, we want to provide support for SW debugging of multi-core platforms by extending our implementation of the GDB RSP interface accordingly.

The next section, presents an efficient core timing model, which we have integrated into the RISC-V VP core, to enable fast and accurate performance evaluation for RISC-V-based systems.

3.2 Fast and Accurate Performance Evaluation for RISC-V

As mentioned already, there exist a number of RISC-V simulators such as the reference simulator Spike [197], RISCV-QEMU [185], RV8 [186], DBT-RISE [42], or Renode [176]. These simulators differ in their implementation techniques and their intended use case, which range from mainly pure CPU simulation (RV8, Spike) to full-system simulation (RISCV-QEMU, DBT-RISE) and even support for multi-node networks of embedded systems (Renode). However, they are mainly designed to simulate as fast as possible to enable efficient functional validation

[8] This (dynamic) translation from RISC-V instructions to the simulation host instruction set should also preserve (for example) timing information of the SystemC simulation to avoid losing accuracy in the simulation timing model.

of large and complex systems and predominantly employ DBT (*Dynamic Binary Translation*) techniques, i.e., translate RISC-V instructions on the fly to *x86_64*. This is however a trade-off as accurately modeling timing information becomes much more challenging and thus these simulators do not offer a cycle-accurate performance evaluation.

A full-system simulator that can provide accurate performance evaluation results and recently got RISC-V support is *gem5* [18]. However, *gem5* integrates more detailed functional models (e.g., of the CPU pipeline) instead of abstract timing models that are sufficient for a pure performance evaluation and hence the simulation performance is significantly reduced.

Commercial VP tools, such as Synopsys Virtualizer or Mentor Vista, might also support RISC-V in combination with fast and accurate timing models but their implementation is proprietary.

Many recent state-of-the-art performance evaluation techniques focus on *Source-Level Timing Simulation* (SLTS) to enable a high-speed performance evaluation [26, 145, 165, 198, 215, 216]. SLTS works by instrumenting the application source code with timing information that are typically obtained by a static analysis, and then host compiles the instrumented source code and natively executes the resulting binary without any emulation layer. However, due to the source-level abstraction it is very challenging to provide accurate models for peripherals and consider complex HW/SW interactions such as interrupts accurately.

Other recent approaches leverage DBT-based techniques. The papers [31, 33] discuss a combination of QEMU and the SystemC kernel. However, these methods either only provide rough performance estimates or incur significant synchronization overhead between QEMU and SystemC and lose the determinism of a SystemC simulation. Another DBT-based approach is [24], which integrates a timing model with a DBT-based execution engine to obtain near cycle accurate results. The work by Thach et al. [206] is conceptually similar but uses QEMU as execution back-end. However, neither of these approaches are VP-based or target the RISC-V ISA.

In the following we present an efficient core timing model and its integration into the RISC-V VP core, to enable fast and accurate performance evaluation for RISC-V-based systems.

3.2.1 Background: HiFive1 Board

The HiFive1 board from SiFive has a 32-bit RISC-V core that integrates a 16-KiB 2-way set-associative instruction cache with a branch prediction unit and a single-issue, in-order pipeline with five stages. The pipeline can achieve a peak execution rate of one instruction per cycle. However, multiplication and division as well as load and store instructions require additional cycles to provide the result and hence can stall the pipeline. The branch prediction unit has a 40-entry *Branch Target Buffer* (BTB), a 128-entry *Branch History Table* (BHT), and 2-entry *Return Address Stack* (RAS). We will describe these branch prediction components as part of our branch

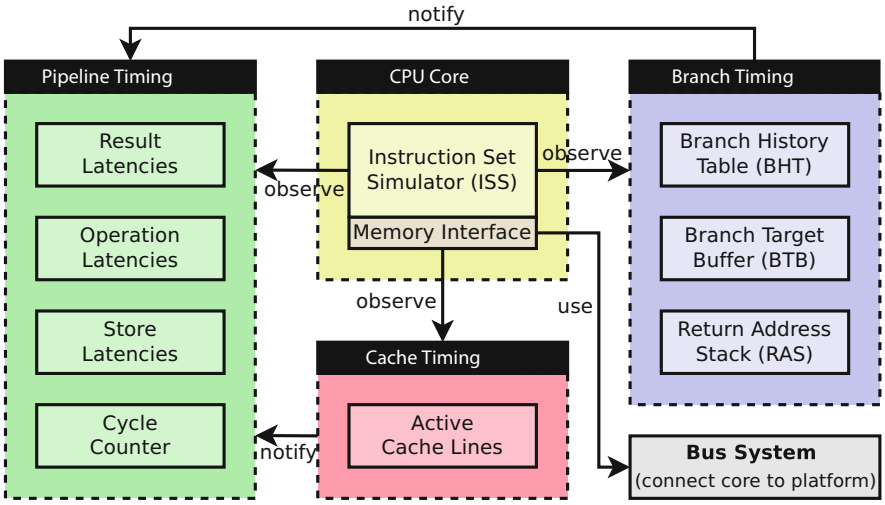

Fig. 3.12 Overview on our core timing model and the integration with the CPU core model

prediction timing model in Sect. 3.2.2.3 in more detail. Correctly predicted branches result in no execution penalty, but mispredicted branches incur a three-cycle penalty. Instructions can be fetched with a one-cycle latency from the cache. A cache miss triggers a burst access to load the corresponding cache line (32-byte). Please note that the HiFive1 board has no designated DRAM but provides a small cache-like data memory that has access rates with fixed latencies (depending on the access size). However, nearby load/store instructions with overlapping addresses result in additional execution penalties. For more technical details, please refer to the official manual [195].

3.2.2 Core Timing Model

In this section we present our core timing model that enables a fast and accurate performance evaluation in combination with VPs. We start with an overview. Then, we present more details on our branch prediction (Sect. 3.2.2.3), pipeline (Sect. 3.2.2.2) and cache (Sect. 3.2.2.4) timing models.

3.2.2.1 Overview

Figure 3.12 shows an overview on our core timing model. Essentially, it consists of three parts: a pipeline timing model (left side Fig. 3.12), a branch (predictor) timing model (right side Fig. 3.12), and a cache timing model (bottom Fig. 3.12). These

timing models allow to consider timing information that depends on pipelining, branch prediction, and caching effects, respectively. Please note that in this work we do not consider effects of out-of-order execution, since this is typically not so common for embedded systems.

All three timing models are attached to the CPU core model, which essentially consists of an ISS and a memory interface. The ISS fetches, decodes, and executes instructions one after another. The memory interface performs all memory access operations for the ISS. In particular it is responsible to perform the actual fetch of the next instruction as well as perform load and store operations. The memory interface typically wraps all operations into TLM bus transactions and forwards them to the bus system, which then routes the transactions to the actual target peripherals (hence connects the ISS with the rest of the VP platform).

We observe all executed instructions in the ISS in order to update our pipeline and branch prediction models accordingly. The branch predictor is only queried and updated on branch and jump instructions. The pipeline needs to be updated for instructions that do not finish within a single execution cycle, as they can introduce latencies in case the next instructions depend on the result of a previous instruction before it is available. In this case the pipeline needs to be *stalled*. Typical instruction that require multiple cycles to completion are multiplication/division as well as load/store instructions and instructions that access special registers. We also observe the memory interface in order to update our cache timing model.

Both the branch prediction timing model and the cache timing model have access to the pipeline timing model. In case of a branch misprediction or cache miss, the pipeline is effectively *stalled*. This will update any pending result latencies in the pipeline and increase the cycle counter accordingly.

Please note that it is not necessary to build timing models that are fully functional but only sufficient to keep track of timing-related information. For example, a timing model for the cache needs only to decide if a cache hit or miss occurs, therefore it is not necessary to keep track of the actual cache contents but only keep track of the currently cached address ranges. Another example is the timing model for the pipeline, which needs only enough information to decide if the pipeline will stall and for how many cycles. Therefore, it is typically not necessary to keep track of all the pipeline stages for each instruction, but only result dependencies between instructions and their computation latencies. By performing these (functional) abstractions in the timing models, the computational complexity is significantly reduced, and hence also the performance overhead minimized for keeping track of the timing information, while a very high accuracy is obtained. In the following we describe our three timing models in more detail.

3.2.2.2 Pipeline Timing Model

The pipeline timing model keeps track of timing relevant execution dependencies between instructions in order to decide when and for how many cycles to stall the pipeline. A stall of the pipeline corresponds to *advancing* the pipeline timing model

and is implemented by simply increasing the (total execution) cycle counter and accordingly decreasing the (local latency) counters of any pending dependencies (for illustration we show an example later in this section). We consider three kinds of dependencies: *result*, *operation*, and *store*.

A result dependency denotes that an instruction N is accessing a register R that a previous instruction P writes to and P takes more than one cycle to compute the result, and hence N has to wait for R. In this case we add a result latency for R to our pipeline model. In case a register is read/written (during instruction execution), which has a result latency X, the pipeline model is stalled for X cycles. For better illustration, Fig. 3.13 shows an example. It shows four instructions, two additions (*add*) and two multiplications (*mul*) where *mul* has a 5-cycle result latency and *add* has no result latency (i.e., the result of *add* is directly available for the next instruction in the pipeline). Figure 3.13 (right side) reports the current cycle counter and result latencies after the instruction has finished. The cycle counter starts with zero, and the result latencies are empty. Both *mul* instructions add a 5-cycle result latency on their destination register. After each instruction, the pipeline is advanced one clock cycle, updating the cycle counter and pending result latencies accordingly. The fourth instruction additionally stalls the pipeline model for 4 cycles due to the pending 4-cycle result latency on register a1. Pending operation and store latencies are handled similarly to result latencies.

An operation dependency denotes that an instruction requires a specific resource and hence reserves it for a specific time T. Other instructions that require the same resource need to wait until the resource is released, i.e., T cycles pass. For example, the RISC-V HiFive1 board has only a single multiplication unit, which has a 5-cycle result latency.

A store dependency denotes that an instruction has performed a memory write access, which is scheduled to be committed after X cycles. Load instructions, which access an overlapping memory address range before X cycles passed, will be delayed for a specific configurable time by stalling the pipeline.

Please note, many instructions—like addition, shift, bit operations, etc.—can be effectively executed in a single cycle on embedded systems with a pipeline. For these common instructions, no updates are necessary in the pipeline timing model, and hence they effectively incur no performance overhead (besides incrementing

Operation	Instr.	Reg. Operands			Cycle Counter	Result Latencies
		RD	RS1	RS2		
1: a0=a1+a2	add	a0	a1	a2	1	[]
2: a1=a0*a1	mul	a1	a0	a1	2	[a1 -> 5]
3: a0=a0+a2	add	a0	a0	a2	3	[a1 -> 4]
4: a2=a1*a2	mul	a2	a1 #	a2	8	[a2 -> 5]

stall for 4 cycles due to the pending result latency on a1

Fig. 3.13 Example to illustrate the pipeline timing model

```
1  struct BranchPredictionTiming {
2    static constexpr uint
         BranchMispredictionPenalty = 3;
3    std::shared_ptr<PipelineTiming>
         pipeline;
4    DefaultRAS RAS; // Return Address
         Stack
5    DefaultBTB BTB; // Branch Target
         Buffer
6    DefaultBHT BHT; // Branch History
         Table
7
8    void jump(uint instr_pc, uint
         target_pc) {
9      if (BTB.find(instr_pc) !=
           target_pc) { // wrong BTB
           prediction
10       BTB.update(instr_pc, target_pc);
11       pipeline->stall(
             BranchMispredictionPenalty);
12     }
13   }
14
15   void call(uint instr_pc, uint
         target_pc, uint link_pc) {
16     jump(instr_pc, target_pc);
17     RAS.push(link);
18   }
19
20   void ret(uint instr_pc, uint
         target_pc) {
21     if (RAS.empty() || (target_pc !=
           RAS.pop()))
22       pipeline->stall(
             BranchMispredictionPenalty);
23   }
24
25   void branch(uint instr_pc, uint
         target_pc, bool branch_taken) {
26     Entry &entry =
           BHT.get_entry(instr_pc);
27     if (entry.predict_branch_taken()){
28       // predict that branch will be
             taken
29       if (branch_taken) {
30         jump(instr_pc, target_pc); //
               correct BHT prediction
31       } else { // wrong BHT prediction
32         pipeline->stall(
               BranchMispredictionPenalty);
33       }
34     } else {
35       // predict that branch will not
             be taken
36       if (branch_taken) { // wrong BHT
             prediction
37         BTB.update(instr_pc,
               target_pc);
38         pipeline->stall(
               BranchMispredictionPenalty);
39       }
40     }
41     bht_entry.update_history(
           branch_taken);
42   }
43 };
```

Fig. 3.14 Branch predictor timing model that provides four interface functions (*jump*, *call*, *ret*, and *branch*) to update the timing model after a jump, function call/return, and branch instruction, respectively

the cycle counter) on our timing model in particular if no active dependencies are pending (who would need to be updated).

3.2.2.3 Branch Prediction Timing Model

Figure 3.14 shows our timing model for our branch predictor. The model is updated on branch (Line 25) and jump (Line 8) instructions. For jump instructions, we also distinguish the special cases of a (function) call (Line 15) and return (Line 20).

In case of a branch, the model has to predict if the branch will be taken and in case of a taken branch the address of the target instruction. For a jump (and its variants: call and return), it has to predict the target of the jump. In case of a misprediction,

an execution time penalty is added by *stalling* the pipeline timing model (which mimics the timing effects of a pipeline flush due to the misprediction).

To perform the predictions, the branch predictor timing model manages three sub-models: BTB (*Branch Target Buffer*), BHT (*Branch History Table*), and RAS (*Return Address Stack*). For the following description, as well as in Fig. 3.14, we use the following naming convention: *instr_pc* refers to the address of the branch/jump instruction, *target_pc* refers to the address of the branch/jump target, and *link_pc* is the address of the instruction following the branch/jump instruction in the code (due to potentially varying instruction lengths, *link_pc* typically cannot be inferred based on *instr_pc*).

The BTB is simply a fixed size mapping from *instr_pc* to (the predicted) *target_pc*. The size and replacement strategy is configurable. By default the lower bits of *instr_pc* are used to index the mapping. The BTB is queried to predict the target of branch and jump instructions (Lines 9 and 30) and updated in case of mispredictions (Lines 10 and 37).

The RAS manages a fixed size stack of return addresses for the last function calls. RAS is updated on every *call* instruction (Line 17) and checked on every *return* instruction (Line 21). In case of a misprediction, an execution latency can be directly incurred (Line 22) or an alternative would be to query the BTB (i.e., call the *jump* function in Line 22 instead). RAS drops old entries in case a new entry is pushed on a full stack (Line 17).

The BHT keeps track of the recent execution history for branch instructions, i.e., whether the branch has been taken or not, to predict the outcome of subsequent branches. The BHT is queried (Line 27) and updated (Line 41) on every branch instruction. In case of a correct BHT prediction, the BTB is queried (Line 30 by calling *jump*) to predict the actual branch target. In case of a BHT misprediction, an execution cycle penalty is incurred by stalling the pipeline timing model (Lines 32 and 38). Our BHT implementation has a fixed size mapping indexed by *instr_pc* (i.e., using the lower bits of the address of the branch instruction). Each mapping entry stores the recent (execution) history and a set of (prediction) counters for every execution history variant. The size of the mapping as well as the length of the execution history and counters is configurable.

As an example, consider a BHT configuration with two history bits and two counter bits for each mapping entry. Two history bits allow to distinguish four different cases (by interpreting a 1 as branch Taken and a 0 as Not taken). Thus, four two-bit counters are provided (one for each possible history case). Each two-bit counter can store four possible values: 0 (strongly Not taken), 1 (Not taken), 2 (Taken), and 3 (strongly Taken). Half of the values result in a branch Taken prediction (2,3) and the other half a branch Not taken prediction (0,1). Thus, Line 27 essentially checks if *counters[history] >= 2* and the update in Line 41 basically increments or decrements the *counters[history]* and shifts the history, depending on the actual observed branch direction, i.e., the *branch_taken* variable. Figure 3.15 shows a concrete example for further illustration. It shows a simple C/C++ loop (left top) with corresponding (RISC-V) assembler code (left bottom) and a table that shows relevant data for the loop branch instruction (right side). It shows the

```
i = 1;
while (i < 10)
  ++i;
```

```
li a0,1
li a1,10
loop:
bge a0,a1,end
addi a0,a0,1
j loop
end:
```

loop-branch BHT entry		Init	Loop Iteration				
			1	2	3	...	10
Prediction		/	T	N	N	N	N
Observation		/	N	N	N	N	T
Counters	NN	1	1	1	0	0	1
	NT	1	1	0	0	0	0
	TN	3	3	3	3	3	3
	TT	3	2	2	2	2	2
History		TT	NT	NN	NN	NN	TN

Fig. 3.15 Example to illustrate the branch timing model

current history and counter values for the BHT entry of the loop branch after each loop iteration (the initial state is chosen arbitrarily in this example) as well as the predicted and really observed branch direction. Based on the BHT entry in iteration N, a prediction is made for iteration N+1. For example, after iteration 1 the history is NT, and hence the NT counter decides the prediction N due to the counter value 1. Based on the observation N in iteration 2, the counter NT is updated accordingly (decremented) and the N is shifted into the history to obtain the BHT entry state after iteration 2.

Besides several configuration options, like the execution history and prediction counter lengths, this branch prediction timing model can also be easily adapted by, e.g., simply removing the RAS in case the embedded system does not cache function call return addresses.

3.2.2.4 Cache Timing Model

The cache timing model is queried on memory access operations to cached regions. Typically, an embedded system provides at least an instruction cache, and hence the timing model will be queried for every instruction fetch. Thus, the timing model should be very efficient in deciding a cache hit or miss to reduce the performance overhead. In case of a cache miss, an execution penalty is added by *stalling* the pipeline timing model (which mimics the timing effects of waiting for a cache line fetch).

We provide a model matching a two-way associative cache with LRU replacement strategy, which is a common choice for embedded systems. However, our cache timing model can be easily generalized to an N-way associative cache with a different replacement strategy. The cache table consists of cache lines, which in turn consist of two entries (for a two-way associative cache). We only keep track of the memory address that the entries represent and not the data, since the actual data is not relevant for our timing model. The number of lines and their size in bytes is

configurable. Each memory access address is translated to an index into the cache table by taking the lower bits of the address. Finally, we compare the memory access address with the address of both cache line entries to decide if the memory access is a cache hit or miss. In case of a miss (none of the entries does match), one of the two entries is updated (i.e., associate the entry with a new memory address range, which matches the memory access address) based on the replacement strategy.

3.2.3 Experiments

We have implemented our proposed core timing model and integrated it into the open-source RISC-V VP (see Sect. 3.1). Furthermore, we configured our core timing model to match the RISC-V HiFive1 board from SiFive (see Sect. 3.2.1). In this section we present experiments that evaluate the accuracy and simulation performance overhead of our core timing model. We obtain accuracy results by comparing against a real RISC-V HiFive1 board from SiFive, and we obtain the performance overhead results by comparing against the RISC-V VP without integrating our core timing model. All experiments are evaluated on a Linux machine with Ubuntu 16.04.1 and an Intel i5-7200U processor with 2.5 GHz (up to 3.1 GHz with Turbo Boost).

For evaluation, we use the *Embench* benchmark suite [50, 51].[9] It is a freely available standard benchmark suite specifically targeting embedded devices. The *Embench* benchmark suite is a collection of real programs instead of synthetic programs like *Dhrystone* or *Coremark* to ensure that more realistic workloads are considered. For example, it contains benchmarks that perform error checking, hashing, encryption and decryption, sorting, matrix multiplication and inversion, JPEG and QR-code as well as regex and state machine operations. The benchmarks have a varying degree of computational, branching, and memory access complexity [15]. Each benchmark starts with a *warm-up* phase that executes the benchmark body a few times to warm up the caches. Then, the real benchmark starts by executing the benchmark body again several more times. The number of iterations is configurable and depends on the CPU frequency. Furthermore, it varies between different benchmarks to ensure a mostly uniform runtime complexity across the benchmark suite. We set the CPU frequency constant to 320 MHz to match the HiFive1 board.

Table 3.2 shows the results. Starting from the left, the columns show the benchmark name (Column 1), the *Lines of Code* (LoC) in C (Column 2) and RISC-V Assembler (Column 3), and the number of executed RISC-V instructions (Column 4, measured by the RISC-V VP). Then, Table 3.2 shows the simulation time in seconds for running the benchmark on the RISC-V VP without (Column 5) and

[9] We omitted three of the nineteen benchmarks from the comparison due to problems in executing them on the real HiFive1 board.

Table 3.2 Experiment results—all simulation time results reported in seconds

	LoC			VP	VP + core timing model		HiFive1 board	Difference	
1: Benchmark	2: C	3: ASM	4: #Instrs.	5: Time	6: Time	7: #Cycles	8: #Cycles	9: #Cycles	10: Time
aha-mont64	306	4887	1,083,316,249	27.46	32.05	1,130,126,565	1,192,518,796	−5.2%	+16.7%
crc32	256	3890	1,093,865,270	30.02	33.51	1,283,897,183	1,284,037,753	−0.0%	+11.6%
edn	440	4655	1,009,188,407	30.81	34.12	1,772,167,527	1,847,419,217	−4.1%	+10.7%
matmult-int	330	4076	1,039,623,303	30.61	33.68	1,761,624,764	1,780,530,752	−1.1%	+10.0%
minver	343	6262	948,801,750	32.90	39.78	1,346,343,318	1,298,778,561	+3.7%	+20.9%
nbody	322	6427	995,233,372	32.25	39.18	1,356,400,421	1,364,440,490	−0.6%	+21.5%
nettle-aes	1173	5831	1,203,436,482	34.82	42.10	1,219,069,707	1,223,022,326	−0.3%	+20.9%
nettle-sha256	503	6252	1,078,331,659	36.65	43.57	1,101,319,048	1,106,544,095	−0.5%	+18.9%
picojpeg	2337	9482	957,883,248	30.23	36.09	1,208,694,004	1,158,481,492	+4.3%	+19.4%
qrduino	1091	7726	1,142,278,239	34.28	38.58	1,399,022,255	1,544,185,605	−9.4%	+12.5%
sglib-combined	2002	6501	906,977,088	29.28	35.69	1,190,579,040	1,209,895,484	−1.6%	+21.9%
slre	661	5518	1,110,788,540	35.71	43.69	1,443,734,072	1,272,198,737	+13.5%	+22.3%
st	271	6643	1,051,545,074	33.60	40.94	1,425,869,796	1,333,455,328	+6.9%	+21.8%
statemate	1456	5866	789,501,877	27.68	38.04	988,087,257	1,077,693,835	−8.3%	+37.4%
ud	250	4774	841,910,324	27.02	30.55	1,241,243,866	1,184,148,661	+4.8%	+13.1%
wikisort	1021	8482	833,317,333	26.65	32.89	1,057,727,371	1,187,236,518	−10.9%	+23.4%

The columns are numbered to enable referencing them

with (Column 6) integrating our core timing model, respectively. The simulation time denotes how long (wall time) it takes to run the benchmark on the VP. Please note that we have configured our core timing model to match the core of the HiFive1 board (see Sect. 3.2.1). The next two columns give the number of execution cycles for the benchmark as reported by the RISC-V VP with integrating our core timing model (Column 7, which provides a fast estimation of the real HiFive1 board) and the real HiFive1 board as comparison (Column 8). RISC-V provides a special register that holds the current number of execution cycles. We query this register before and after running the benchmark to obtain the spent execution cycles. Finally, the last two columns report the difference in the estimated number of execution cycles with the really measured execution cycles (Column 9) and the performance overhead in integrating our core timing model into the RISC-V VP (Column 10). Figure 3.16 presents the accuracy and performance overhead results in graphical form for a better illustration.

It can be observed that our core timing model provides a very accurate estimation of the number of execution cycles that we measured on the real HiFive1 board. For some benchmarks, our core timing model estimation under-approximates (i.e., estimates less cycles), and for other benchmarks over-approximates (i.e., estimates more cycles), the actual number of execution cycles. In summary, we observed a minimum and maximum difference of 0.0% and 13.5%, respectively, with an average of 4.7% on this benchmark set. These results demonstrate that our core timing model can be used for a very accurate timing/performance evaluation and hence enables an efficient VP-based design space exploration. The remaining accuracy gap is due to incomplete specification of the timing-related functionality of the HiFive1 board (e.g., the replacement strategy in the branch prediction buffers). We believe that with a more complete specification we can adapt the configuration of our timing model appropriately to obtain even more accurate results.

Furthermore, please note that our core timing model is not limited to the HiFive1 board but can be configured to match other specifications as well by configuring our pipeline, branch prediction, and caching models appropriately.

Besides high accuracy, our core timing model retains at the same time the high simulation performance of the VP by introducing a reasonably small performance overhead. We observed a minimum and maximum overhead of 10.0% and 37.4%, respectively, with an average of 18.9% on this benchmark set. With our core timing model, the RISC-V VP simulates with an average of 27 MIPS (*Million Instructions Per Second*) compared to 32 MIPS of the original RISC-V VP. The performance overhead depends on the number of data flow dependencies between subsequent instructions as well as number of branching instructions as they update our branch prediction model. These parameters are highly dependent on the actual benchmark. Furthermore, we have an additional (almost constant) time overhead for each instruction fetch, as we have to check (and potentially update) if the fetch results in a cache miss and hence incurs an execution time penalty.

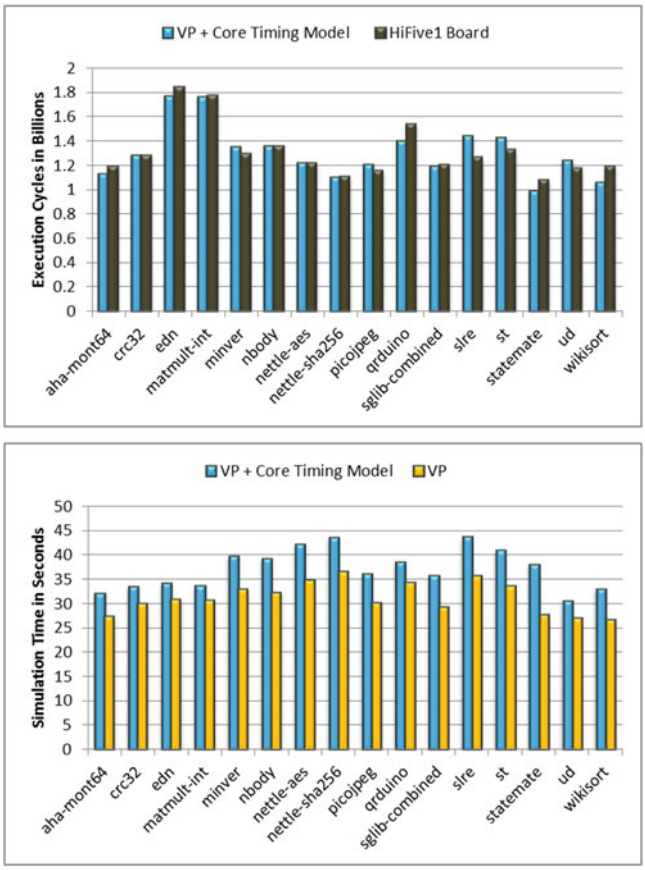

Fig. 3.16 Summary of the results on the accuracy compared to the HiFive1 board (top diagram) and performance overhead compared to the original RISC-V VP (bottom diagram) of our approach (VP + Core Timing Model)

3.2.4 Discussion and Future Work

Our proposed core timing model allows to consider pipelining, branch prediction as well as caching effects and showed very good results on a case study using the RISC-V HiFive1 board. In future work we plan to improve our work in the following directions:

- Investigate further abstractions of our timing models and generation of (e.g., basic block) summaries to simplify and reduce the number of model updates and hence reduce the performance overhead while retaining sufficient accuracy.

- Evaluate integration of DBT into the VP ISS for further performance optimization. This may also require adoptions of the timing model to ensure a smooth integration.
- Add and evaluate configurations matching other RISC-V cores, besides the core of the HiFive1 board.
- Extend the timing model to consider more advanced features like out-of-order execution.

3.3 Summary

We have proposed and implemented a 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 RV32/64 (multi-)core with a generic bus system employing TLM 2.0 communication. In addition to the RISC-V core(s), we provide SW debug and coverage measurement capabilities, a set of essential peripherals, including the RISC-V CLINT and PLIC interrupt controllers, support for several operating systems (recently also including Linux support), and an example configuration matching the HiFive1 board from SiFive. The existing feature set in combination with the extensibility and configurability of our VP makes our VP suitable as foundation for various application areas and system-level use cases, including early SW development and analysis of interactions at the HW/SW interface of RISC-V-based embedded systems. Our evaluation demonstrated the quality and applicability to real-world embedded applications as well as the high simulation performance of our VP. Finally, our RISC-V VP is fully open source to stimulate further research and development of ESL methodologies.

In addition, we presented an efficient core timing model that allows to consider pipelining, branch prediction, and caching effects. We integrated the timing model into our open source RISC-V VP to enable fast and accurate performance evaluation for RISC-V-based systems. As a case study, we provided a timing configuration matching the RISC-V HiFive1 board from SiFive. Our experiments using the *Embench* benchmark suite demonstrate that our approach allows to obtain very accurate performance evaluation results (less than 5% mismatch on average) while still retaining a high simulation performance (less than 20% simulation performance overhead on average). To help in advancing the RISC-V ecosystem and stimulate research, our core timing model, including the integration with the RISCV VP, is available open source as well.