MicroRV32: An Open Source RISC-V Cross-Level Platform for Education and Research

Sallar Ahmadi-Pour
Institute of Computer Science,
University of Bremen
Bremen, Germany
sallar@uni-bremen.de

Vladimir Herdt
Institute of Computer Science,
University of Bremen
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
vherdt@uni-bremen.de

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

ABSTRACT

In this paper we propose µRV32 (MicroRV32) an open source RISC-V platform for education and research. µRV32 integrates several peripherals alongside a 32 bit RISC-V core interconnected with a generic bus system. It supports bare-metal applications as well as the FreeRTOS operating system. Beside an RTL implementation in the modern SpinalHDL language (µRV32 RTL) we also provide a corresponding binary compatible Virtual Prototype (VP) that is implemented in standard compliant SystemC TLM (µRV32 VP). In combination the VP and RTL descriptions pave the way for advanced cross-level methodologies in the RISC-V context. Moreover, based on a readily available open source tool flow, µRV32 RTL can be exported into a Verilog description and simulated with the Verilator tool or synthesized onto an FPGA. The tool flow is very accessible and fully supported under Linux. As part of our experiments we provide a set of ready to use application benchmarks and report execution performance results of µRV32 at the RTL, VP and FPGA level together with a proof-of-concept FPGA synthesis statistic.

KEYWORDS

RISC-V, RTL, FPGA, Virtual Prototype, Open Source

ACM Reference Format:
Sallar Ahmadi-Pour, Vladimir Herdt, and Rolf Drechsler. 2021. MicroRV32: An Open Source RISC-V Cross-Level Platform for Education and Research. In *Design Automation for CPS and IoT (Destion '21), May 18, 2021, Nashville, TN, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3445034.3460508

1 INTRODUCTION

RISC-V [21, 22] is a modern *Instruction Set Architecture* (ISA) with enormous potential in particular for embedded systems used in several application areas such as IoT or edge computing. A key factor for the success story of RISC-V is the free and open nature of the ISA. Moreover, RISC-V is designed in a very modular and extensible way which makes it possible to build highly application specific solutions. Naturally, RISC-V has been strongly adopted by the industry and also in the academic community.

Meanwhile, RISC-V offers a very comprehensive but still growing ecosystem with a plethora of tools, simulators and *Register Transfer Level* (RTL) implementations, both commercial as well as open source. Recently, *Virtual Prototypes* (VPs) emerged in the RISC-V ecosystem to support the design flow for embedded systems. A VP is essentially an abstract model of the entire *Hardware* (HW) platform and predominantly created in SystemC using the *Transaction Level Modeling* (TLM) style [1, 9]. VPs are an industry proven solution to enable early SW development as well as other system-level use-cases and thus complement a RTL implementation [10, 15, 16, 19]. We believe that the availability of a modern, accessible and FPGA friendly RISC-V RTL implementation together with a corresponding VP configuration would be very beneficial for the academic community to stimulate further research and for educational purposes. Such a VP/RTL combination provides a strong foundation for advanced cross-level methodologies.

Therefore, in this paper we propose µRV32 (MicroRV32) an open source RISC-V platform for education and research. µRV32 integrates several peripherals alongside a 32 bit RISC-V core interconnected with a generic bus system. The core supports the base integer instruction set and provides trap and interrupt handling facilities. This allows µRV32 to run bare-metal applications as well as operating systems tailored for the embedded domain such as FreeRTOS. µRV32 is available as RTL description (µRV32 RTL) and implemented in the modern Scala-based SpinalHDL language. Based on a readily available open source tool flow, µRV32 RTL can be exported into a Verilog description and simulated with the Verilator tool or synthesized onto an FPGA. The tool flow is very accessible and fully supported under Linux. In addition to µ32RV RTL we also provide a corresponding VP configuration, called µRV32 VP, which is implemented in standard compliant SystemC TLM and binary compatible with µRV32 RTL. We built µRV32 VP on top of the open source RISC-V VP [18] available at GitHub [5]. The VP enables early and fast *Software* (SW) simulations while the RTL description enables cycle-accurate simulations. In combination the VP and RTL descriptions pave the way for advanced cross-level methodologies. As part of our experiments we provide a set of ready to use application benchmarks and report execution performance results of µRV32 at the RTL, VP and FPGA level together with a proof-of-concept FPGA synthesis statistic. Visit http://systemc-verification.org/risc-v for a GitHub link to obtain µRV32 RTL/VP together with the benchmarks as well as information on our most recent RISC-V related approaches.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Destion '21, May 18, 2021, Nashville, TN, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8316-5/21/05...$15.00
https://doi.org/10.1145/3445034.3460508

2 RELATED WORK

RISC-V already comes with an extensive ecosystem that includes several simulators as well as RTL implementations that can also be used for FPGA prototyping purposes.

With respect to simulators, they are predominantly designed to enable high-speed simulations such as SPIKE [7] or QEMU [6]. Another direction are full platform simulators such as gem5 [4]. Recently, also VP-based solutions that leverage SystemC TLM such as RISC-V VP [5] or DBT-Rise [3] have been introduced into the ecosystem to lay the foundation for advanced SystemC-based system level use-cases for RISC-V. In this work we built upon RISC-V VP to design our µRV32 VP to complement our cross-level platform.

Similarly, there exist various RTL implementations for RISC-V ready to use on FPGAs. Many of the cores rely on commercial FPGA tool flows which makes them not very accessible. Thus, in the following we exemplarily review cores that also rely on open source tools but follow different goals than our µRV32 cross-level platform. For example, the PicoRV32/PicoSoC [13] is a Verilog HDL implementation of the RISC-V ISA which is optimized for size. An examplatory *System-on-Chip* (SoC) with a small amount of periphrals and firmware is available. However, while Verilog provides very good tool support, it is missing many features of the modern emerging *Hardware Description Languages* (HDLs) such as SpinalHDL [11] or Chisel [2]. RocketChip [8] is a RISC-V SoC generator that leverages the Chisel HDL to provide a highly configurable general purpose solution. VexRiscV [12] is a SpinalHDL-based implementation of the RISC-V ISA. VexRiscV makes use of a SW oriented approach by leveraging SpinalHDL to offer a broad range of parametric and customizable RISC-V platforms. Thus, making VexRiscV a powerful family of RISC-V implementations that can even support a Linux operating system. However, the complexity of VexRiscV and RocketChip makes them significantly less accessible.

Moreover, with µRV32 we propose a combined RTL and VP-based implementation which provides the foundation for advanced cross-level methodologies tailored for RISC-V.

3 PRELIMINARIES

In this section we provide relevant background information on the RISC-V ISA (Section 3.1), the open source RISC-V VP which we used as foundation to build µRV32 VP (Section 3.2), and the open source tool flow that covers the VP, RTL and FPGA level (Section 3.3).

3.1 RISC-V ISA

RISC-V is an open, free and modular *Instruction Set Architecture* (ISA). For this work we consider the RISC-V base RV32I ISA. It provides a set of basic mandatory instructions that cover arithmetic, branch and jump, as well as load and store instructions. RV32I defines 32 general purpose registers x0 to x31 (with x0 being hardwired to zero) with 32 bit width each. The RISC-V ISA also defines *Control and Status Registers* (CSRs) which are special purpose registers for extended HW/SW interactions such as trap handling and interrupt processing capabilities. For example, the MTVEC CSR stores the trap handler address which is configured by the SW and used by the HW. For a comprehensive description of the RISC-V instruction set please refer to the official specifications. You can find more information on the instruction set specifications in volume 1 [21] and details on the privileged architecture which in particular covers CSRs and their behavior in volume 2 [22].

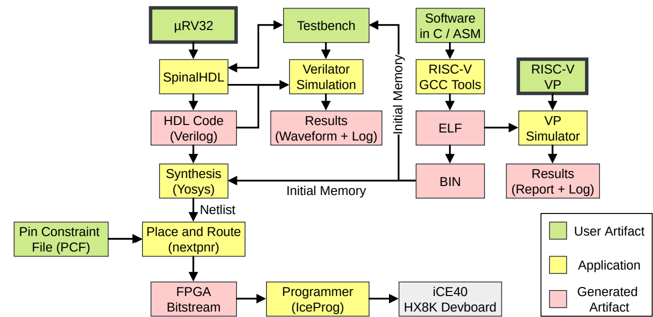

Figure 1: Overview of the open source HW/SW co-simulation and co-design tool flow

3.2 RISC-V VP

The RISC-V VP is an open source VP tailored for RISC-V and implemented in SystemC TLM. It is designed as a configurable and extensible platform around a generic TLM bus system. The VP supports ELF loading (as generated by the GCC or LLVM toolchain) and provides coverage tracking (via GCOV) and debugging (via GDB) support of the SW applications running on the VP. Through these characteristics the VP enables fast SW development iterations. The TLM-based description also allows for quick explorations of new extensions of the ISA or HW platform. For the µRV32 cross-level platform we added a configuration to the VP which represent µRV32 RTL. The RISC-V VP has already been used in several research studies that cover modeling, verification and simulation aspects e.g. [17, 18, 20].

The RISC-V VP also supports the *Direct Memory Interface* (DMI) and *Time Quantum* (TQ) optimizations commonly used to speed-up SystemC TLM simulations. Essentially, DMI boosts the performance of memory access operations by using a direct memory pointer instead of TLM transcations to access the main memory, while TQ avoids costly context switches in the *Instruction Set Simulator* (ISS)[1] by postponing synchronizations with the SystemC simulation kernel.

3.3 Open Source Cross-Level Toolflow

Fig. 1 shows the co-design and co-simulation toolflow. The diagram contains user artifcats (green), applications (yellow) and generated artifacts (red). The user artifacts can be divided into three parts:

1. The SpinalHDL HW description and the respective testbench and *Pin Constraint File* (PCF) for the use of the HW description on the FPGA
2. The RISC-V VP used for TLM simulations
3. The RISC-V SW written in C and assembly

The HW description available in SpinalHDL can be exported to a Verilog description and then used in two ways: 1) simulated with SpinalHDL and Verilator according to the testbench[2], or 2)

[1]The ISS is essentially an abstract model of the CPU core and thus fetches, decodes and executes intructions one after another.
[2]SpinalHDL provides a foreign function interface to interact from the SpinalHDL testbench with the C++ HW description generated by Verilator.

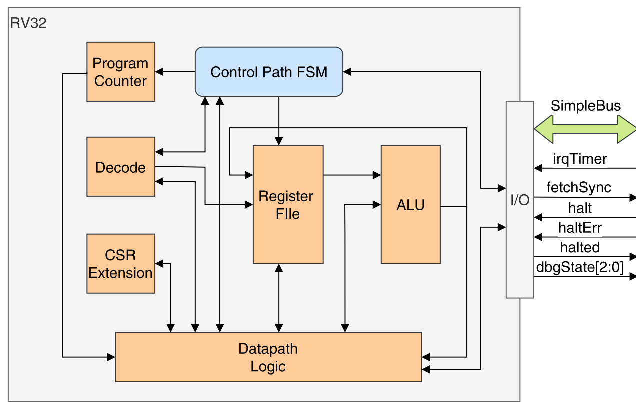

Figure 2: Architecture of the RISC-V core of the µRV32 SoC

synthesized to an FPGA. Therefore, a set of open source tools is provided in particular *Yosys* to generate a netlist, *nextpnr* to generate the FPGA bitstream, and *IceProg* to program the FPGA. These tools are part of the open source toolchain *IceStorm* [14] which supports Lattice Semiconductor iCE40 FPGAs.

The embedded RISC-V SW is compiled using the normal GCC toolchain into an ELF file. The ELF file can either be loaded on the µRV32 VP or transformed by a simple script into a raw binary file that can be loaded into the RTL simulation or the FPGA memory. The µRV32 VP enables co-design and co-simulation workflow with µRV32 RTL. New additions like HW extensions or ISA extensions can be planned and prototyped on the VP. The VP enables quick design space exploration with viable prototypes for the SW development. When an extension (either a platform or ISA extension) is planned out, the VP can serve as an executable specification for the use in the RTL description. Thus, allowing iterations of HW/SW development and debugging to be shorter. The refinement of the SW is not dependent on the RTL description and thus can take place in parallel with the RTL refinement. After the extension is realized in RTL it can be validated on an FPGA.

The complete tool flow covering the VP, RTL and FPGA level is available open source and fully supported under Linux, making it very accessible for education and research.

4 MICRORV32

In this section we present implementation details on µRV32. It is a modern, accessible and FPGA friendly RISC-V platform designed around a 32 bit RISC-V core. It consists of two parts:

(1) µRV32 RTL: a modern, accessible and FPGA friendly RISC-V RTL implementation in SpinalHDL designed to be used with the open source FPGA toolchain IceStorm[14] (cf. Section 3.3).
(2) µRV32 VP: a corresponding binary compatible RISC-V VP (cf. Section 3.2) configuration representing µRV32 RTL at a high level of abstraction.

In combination µRV32 provides a strong foundation for investigating advanced cross-level methodologies and design flow techniques. µRV32 RTL/VP and the complete tool flow is available open source and fully supported under Linux, thus making it very accessible for education and research.

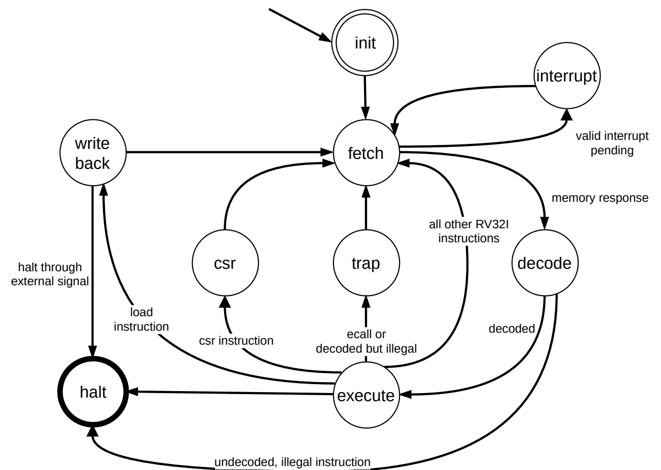

Figure 3: Finite State Machine of the Control Path

Fig. 4 shows an overview of the SoC platform architecture with its components and the inputs/outputs. In the following, we present more details on the overall platform, in particular on the RISC-V core (Section 4.1), the memory bus interface (Section 4.2) and the peripherals of the SoC platform (Section 4.3). In our description we focus on the RTL part of the platform.

4.1 RISC-V Core

The core implements the RV32I instruction set together with the CSR instruction set extension of the RISC-V ISA. Fig. 2 shows the multiple components of the core. Through the CSR extension the core supports SW traps and environment calls and interrupts. Therefore, a set of elementary CSR registers is implemented. The data path contains the necessary operations to execute all instructions. The control path is a finite state machine which follows a lightweight fetch-decode-execute-writeback scheme. The main components are the control path, the instruction decoder, the datapath, an arithmetic logic unit, a program counter, a register file and the CSR extension. Inputs and outputs of the core consist of the memory bus interface, the timer interrupt input, the halt signal inputs to transition the core into the defined halt state and additional signals used for debugging. Fig. 3 shows the finite state machine of the control path. In the `init` state all registers are initialized to their reset values. This includes the program counter, which is set to the starting address of the main memory. This state is suceeded by the `fetch` state. If an interrupt is pending, the FSM transitions into the `interrupt` state. The interrupt enables are saved, the interrupts are disabled, the `mcause` register gets set and the program counter is set to the trap handler address. The program counter, at which the interrupt occurs, is saved for restoration after interrupt handling. After the `interrupt` state the program execution of the traphandler follows the regular instruction execution. On return from the trap handler the instruction preempted by the interrupt will be executed starting from the `fetch` state. The current program counter value is used as the address to read the next instruction from memory. A fetched instruction is decoded in the `decode` state. If the fetched instruction is invalid, the next state is `halt`, otherwise the decoded instruction is executed in the `execute` state. On

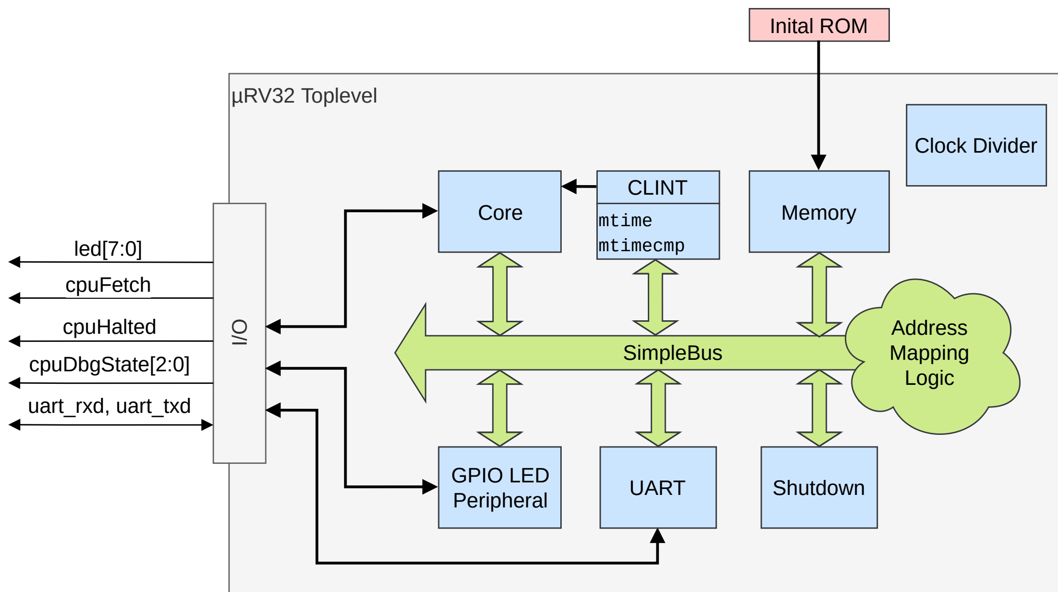

Figure 4: Block Diagram of the top level architecture of the µRV32 SoC platform

```
1  case class SimpleBus(dataWidth:Int, addressWidth:Int)
     extends Bundle with IMasterSlave {
2    val SBaddress = UInt(addressWidth bits)
3    val SBvalid = Bool
4    val SBwdata = Bits(dataWidth bits)
5    val SBwrite = Bool
6    val SBsize = UInt(4 bits)
7    val SBready = Bool
8    val SBrdata = Bits(dataWidth bits)
9
10   override def asMaster(): Unit = {
11       out(SBvalid, SBaddress, SBwdata, SBwrite, SBsize)
12       in(SBready, SBrdata)
13   }
14 }
```

Listing 1: Memory Bus Interface definition in SpinalHDL

a load or store instruction, the state machine will transition to the `writeback` state. A CSR instruction causes the FSM to transition into the `csr` state. This additional state is used to the access the CSR logic. An instruction with the `SYSTEM` opcode or an otherwise undefined instruction that passed the `decode` state causes a transition to the `trap` state. The states `writeback`, `csr` and `trap` are succeeded by the state `fetch`. Any other instruction will cause a transition from the `execute` state back to the `fetch` state.

4.2 Memory Bus Interface

The core interacts with peripherals and its environment through an interface defined by an address, a command and data. For this purpose a lightweight bus interface with a valid-ready handshake is chosen. It is used to interconnect the core and the surrounding peripherals. In this context the core is the bus master while peripherals and other modules act as bus slaves. With the core as the only bus master in place there is no need to consider bus master arbitration. Listing 1 shows the memory bus interface definition in SpinalHDL. The bus master asserts the `valid` signal to notify the bus slaves on a valid command and payload on the bus. On the top level module the address space is mapped onto the peripherals. The transaction then is routed to the respective peripheral. For

```
1  // ...
2  // Instantiate components of SoC
3  val cpu = new RV32Core()
4  val ram = new Memory(Bits(32 bits),4104,initHexfile)
5  val gpio_led = new GPIOLED()
6  val shutdown_periph = new Shutdown()
7  val uartPeriph = new SBUart()
8  val rvCLIC = new RVCLIC()
9  // ...
10 // Interconnect components via memory bus interface
11 cpu.io.sb <> ram.io.sb
12 cpu.io.sb <> gpio_led.io.sb
13 cpu.io.sb <> shutdown_periph.io.sb
14 cpu.io.sb <> uartPeriph.io.sb
15 cpu.io.sb <> rvCLIC.io.sb
16 // ...
```

Listing 2: Interconnecting µRV32 SoC components in SpinalHDL

lightweight design it is defined that peripherals respond one clock cycle after the transaction request. A peripheral should finish its tasks within one clock cycle, otherwise the CPU is stalled. Additionally the identification of incorrectly behaving peripherals becomes easier. Listing 2 shows how the components are interconnected and wired in SpinalHDL. This feature of SpinalHDL allows for less errors in the interconnection of modules.

4.3 SoC Platform

The core is embedded in a SoC platform composed around the SimpleBus. Fig. 4 shows the top-level view of the µRV32 architecture. The memory and the peripherals are mapped on the top level of the SoC, in the address mapping logic. First, the core-local interrupt controller (CLINT) peripheral provides the timer interrupts based on the 64 bit registers `mtime` and `mtimecmp`. `mtime` is a read-only register that increments with the platforms clock frequency. If the value of `mtime` is greater or equal than `mtimecmp` then the timer interrupt is triggered until cleared by the core. In the Shutdown peripheral a defined transition into a halting state can be triggered for the core. The halting state will end program execution and halts the

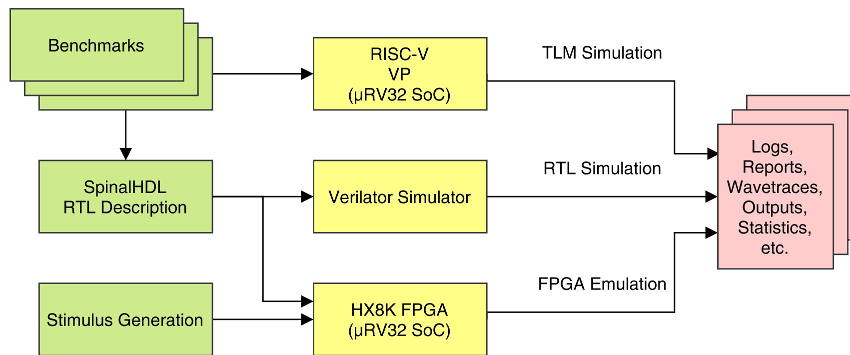

Figure 5: Setup to compare the VP, RTL simulation and FPGA-based emulation

platform in the defined state until reset. In the Memory peripheral the instruction memory and the data memory are contained. The LED peripheral is used to map up to eight LEDs of a development board into the address space of the SoC platform. The *Universal Asynchronous Receive and Transmit* (UART) peripheral provides the platform with external communication abilities. UART is commonly used as serial communication between platforms and devices. For the SW running on the SoC platform, the print statements can be redirected to the UART peripheral.

The SoC platform memory is initialized with a SW program binary at synthesis time.

5 EXPERIMENTS

In this section we present a comparison of the performance of our µRV32 with the RISC-V VP. Our comparison includes the execution of µRV32 in an RTL simulator (SpinalHDL with Verilator) and on an FPGA development board. These modes of execution are compared to the µRV32 VP. The FPGA-based emulation runs on a Lattice Semiconductor HX8K Development Board with the board frequency of 12 MHz. The VP and the RTL simulation are executed on an Intel i7 10510U CPU @ 1.80 GHz on Ubuntu 20.04 LTS. Additionally, we collect the statistics on the FPGA design process, that is the time for synthesis, place & route, the area utilization on the targeted FPGA and the maximum clock frequency at which the design can be used.

In the following, we provide the evaluation setup (Section 5.1), the obtained results (Section 5.2) and the FPGA synthesis statistics (Section 5.3).

5.1 Performance Evaluation Setup

Fig. 5 shows the setup of the experiments. The benchmarks are used to initialize the SoC as well as the VP. For the VP the runtime gets traced with the `time` command from Linux. On the RTL simulation the testbench is instrumented to output the real processing time needed for the simulation. The RTL simulation is executed through the SpinalHDL Verilator backend. At the level of the FPGA-based emulation, the time start of the execution (falling edge of the reset signal) and the end of execution (rising edge of the halt signal) are measured with a logic analyzer. For the experiments we use five benchmarks:

(1) *Fibonacci* calculates the numbers of the Fibonacci sequence up to a defined sequence length of 6000 numbers.
(2) *Greatest Common Divisor (GCD)* calculates the greates common divisor $gcd(a, b)$ of two numbers a and b (for our experiments we calculate $gcd(50000, 1)$)
(3) *Bubblesort* sorts an array with 300 elements.
(4) *FreeRTOS-queues* is a FreeRTOS example of two senders putting data into a queue and a receiver pulling the data from the queue. The example is setup to terminate after 10 iterations.
(5) *FreeRTOS-tasks* is a FreeRTOS example of two scheduled tasks sending data via the UART interface. The example is setup to terminate after 20 iterations.

5.2 Performance Evaluation Results

Table 1 shows the results of the benchmarks. The table is divided double columns into three parts: The left part shows each executed benchmark. The middle part shows the number of instructions executed (column: #instr-exec.) for a benchmark, the lines of code in C (column: LoC C) of the application and the lines of code for the assembly file (column: LoC ASM) respectively. The right part shows the execution times of each benchmark in seconds. The fourth column (column: FPGA) shows the execution time of the FPGA emulation of the µRV32 platform. The character '-' denotes that the benchmark could not have been executed on the FPGA due to the SW binary being too large for the memory on the development board. The third column (column: RTL-Sim) shows the execution time of the RTL simulation with SpinalHDL and Verilator. In the sixth and seventh column the execution times of the VP simulation are shown without optimizations (column: VP normal) and with optimizations (column: VP opt), namely DMI and TQ (cf. Section 3.2).

From Table 1 it can be observed that the VP simulation and the FPGA emulation is significantly faster than the RTL simulation. When comparing the FPGA emulation with the RTL simulation a factor of improvement between ×134 and ×217 is observed. Comparing the VP simulations with the RTL simulation the unoptimized VP is ×25 to ×96 faster the optimized VP is ×75 to ×325. The unoptimized VP shows slightly bigger execution times than the FPGA-based emulation (between ×1.5 to ×2.3). In all cases the optimized VP simulation has the fastest execution time.

Table 1: Results of the benchmark experiments

Benchmark	#instr-exec.	LoC in C	LoC in ASM	FPGA	RTL-Sim.	VP normal	VP opt
Fibonacci	240,118	24	122	0.09 s	12.07 s	0.16 s	0.08 s
GCD	500,075	31	105	0.19 s	32.24 s	0.43 s	0.14 s
Bubblesort	1,518,041	45	194	0.36 s	78.18 s	0.81 s	0.24 s
FreeRTOS-queues	416,897	220	11,048	-	31.67 s	0.64 s	0.22 s
FreeRTOS-tasks	3,078,543	93	10,988	-	40.55 s	1.63 s	0.54 s

Table 2: FPGA timing and area statistics, Synthesis, Place & Route statistics

Description	Value
f_{max}	28.61 MHz
Logic Cells	4297 / 7680 (55%)
BRAM Cells	25 / 32 (78%)
IO Cells	33 / 256 (12%)
Synthesis time	11.6 s
Place & Route time	18.96 s

5.3 FPGA Synthesis Statistics

Table 2 shows the various statistics of the FPGA design flow. The µRV32 SoC can be operated at a maximum clock frequency f_{max} of 28.61 MHz with a device utilization of 55%. The use of 78% of BRAM Cells of the FPGA vary with the program used to initialize the memory. To synthesize the design into a netlist Yosys took 11.6 s. The place & route of the netlist took NextPNR 18.96 s. This sums to circa 30 seconds for the FPGA toolchain to generate a bitstream that can be configured onto the FPGA.

6 DISCUSSION AND FUTURE WORK

The combination of RTL and VP implementation provided by µRV32 delivers a strong foundation for investigating advanced cross-level methodologies and design flow techniques. While the VP allows for fast and early SW development and HW prototyping through its TLM simulations, the RTL simulation provides cycle-accurate results and FPGA realization. At the same time, µRV32 is very accessible for education and research as the platform and complete tool flow is available open source and fully supported under Linux. To further extend and boost the capabilities of this cross-level platform we plan to:

- Investigate cross-level methodologies between the VP and the RTL descriptions for verification, simulation and modeling purposes. One direction is the integration of RTL peripherals into the SystemC TLM simulation using the C++ RTL models obtained through the Verilator tool to selectively obtain fast and cycle-accurate simulation results.
- Extend the µRV32 SoC platform to integrate additional peripherals at the RTL/VP level and extend the core to include support for more standard RISC-V instruction set extensions. Further, investigate a Domain Specific Language (DSL) to integrate custom instruction set extensions at the RTL and VP level.
- Consider formal methods and comprehensive simulation-based techniques to validate the platform and in particular the RISC-V core. A cross-level co-simulation setting in combination with advanced test generation techniques such as fuzzing and constrained random approaches seem very promising to pursue this direction.

ACKNOWLEDGMENTS

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127 and within the project VerSys under contract no. 01IW19001.

REFERENCES

[1] 2011. *IEEE Standard SystemC Language Reference Manual.*
[2] 2021. Chisel HDL. https://www.chisel-lang.org/. Accessed on 2021-02-20.
[3] 2021. DBT-RISE-RISCV. https://github.com/Minres/DBT-RISE-RISCV.
[4] 2021. gem5. https://www.gem5.org/.
[5] 2021. RISC-V Virtual Prototype. https://github.com/agra-uni-bremen/riscv-vp.
[6] 2021. RISCV-QEMU. https://github.com/riscv/riscv-qemu.
[7] 2021. Spike. https://github.com/riscv/riscv-isa-sim.
[8] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html
[9] J. Aynsley. 2009. *OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL.* Open SystemC Initiative (OSCI).
[10] Amir Charif, Gabriel Busnot, Rania Mameesh, Tanguy Sassolas, and Nicolas Ventroux. 2019. Fast Virtual Prototyping for Embedded Computing Systems Design and Exploration. In *RAPIDO Workshop.* 3:1–3:8.
[11] C.Papon. 2021. SpinalHDL. https://github.com/SpinalHDL/SpinalHDL. Accessed on 2021-02-20.
[12] C.Papon. 2021. VexRiscV. https://github.com/SpinalHDL/VexRiscv. Accessed on 2021-02-20.
[13] C.Wolf. 2021. PicoRV32, PicoSoC. https://github.com/cliffordwolf/picorv32. Accessed on 2021-02-20.
[14] C.Wolf and M.Lasser. 2021. Project IceStorm. http://www.clifford.at/icestorm/. Accessed on 2021-02-20.
[15] Tom De Schutter. 2014. *Better Software. Faster!: Best Practices in Virtual Prototyping.* Synopsys Press.
[16] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies.* Springer.
[17] Vladimir Herdt, Daniel Große, Eyck Jentzsch, and Rolf Drechsler. 2020. Efficient Cross-Level Testing for Processor Verification: A RISC-V Case-Study. In *Forum on Specification and Design Languages.*
[18] Vladimir Herdt, Daniel Große, Pascal Pieper, and Rolf Drechsler. 2020. RISC-V based Virtual Prototype: An Extensible and Configurable Platform for the System-level. *Journal of Systems Architecture - Embedded Software Design* (2020).
[19] G. Onnebrink, R. Leupers, G. Ascheid, and S. Schürmans. 2016. Black box ESL power estimation for loosely-timed TLM models. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS).* 366–371. https://doi.org/10.1109/SAMOS.2016.7818374
[20] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. 2021. An Effective Methodology for Integrating Concolic Testing with SystemC-based Virtual Prototypes. In *Design, Automation and Test in Europe.*
[21] Andrew Waterman and Krste Asanović (Eds.). 2019. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA.*
[22] Andrew Waterman and Krste Asanović (Eds.). 2019. *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture.*