UVM-based Verification of a RISC-V Processor Core Using a Golden Predictor Model and a Configuration Layer

Marcela Zachariasova, Lubos Moravec[1] John Stickley, Hans van der Schoot, Shakeel Jeeawoody[2]

[1]Codasip Ltd., Bozetechova 1/2, 612 00 Brno, Czech Republic, phone +420 541 141 475
{zachariasova, moravec}@codasip.com
[2]Mentor, 8005 SW Boeckman Rd, Wilsonville, OR USA 97070-7777, phone 503-685-7000
{john_stickley, hans_vanderschoot, shakeel_jeeawoody}@mentor.com

***Abstract*: RISC-V is a new ISA (Instruction Set Architecture) that introduces a high level of flexibility into processor architecture design, and enables processor implementations tailored for applications in a variety of domains, from embedded systems, IoT, and high-end mobile phones to warehouse-scale cloud computers. The downside of this extent of flexibility is the verification effort that must be devoted to all variants of the RISC-V cores. In this paper, Codasip and Mentor aim to describe their methodology of effective verification of RISC-V processors, based on a combination of standard techniques, such as UVM and emulation, and new concepts that focus on the specifics of the RISC-V verification, such as configuration layer, golden predictor model, and Mentor Veloce FlexMem memory feature.**

I. INTRODUCTION

RISC-V is a free-to-use and open ISA developed at the University of California, Berkeley, now officially supported by the RISC-V Foundation [1][2]. It was originally designed for research and education, but it is currently being adopted by many commercial implementations similar to ARM cores. The flexibility is reflected in many ISA extensions. In addition to basic Integer ("I"/"E") ISA, many instruction extensions are supported, including multiplication and division extension ("M"), compressed instructions extension ("C"), atomic operations extension ("A"), floating-point extension ("F"), floating-point with double-precision ("D"), floating-point with quad-precision ("Q"), and others. By their combination, more than 100 viable ISAs can be created.

Codasip is a company that delivers RISC-V IP cores, internally named Codix Berkelium (Bk). In contrast to the standard design flow, as defined for example in [3][4], the design flow utilized by Codasip is highly automated, see Fig. 1. Codasip describes processors at a higher abstraction level using an architecture description language called CodAL. Each processor is described by two CodAL models, the instruction-accurate (IA) model, and the cycle-accurate (CA) model. The IA model describes the syntax and semantics of the instructions and their functional behavior without any micro-architectural details. To complement, the CA model describes micro-architectural details such as pipelines, decoding, timing, etc. From these two CodAL models, Codasip tools can automatically generate SDK tools (assembler, linker, C-compiler, simulators, profilers, debuggers) together with RTL and UVM verification environments, as described in [5]. In UVM, the IA model is used as a golden predictor model, and the RTL generated from the CA model is used as the Design Under Test (DUT). Such high level of automation allows for very fast exploration of the design space, producing a unique processor IP with all the software tools in minutes.

This paper aims to demonstrate that the flexibility of RISC-V ISA presents benefits as well as challenges, namely in verification. We will show how to overcome these challenges with a suitable verification strategy, comprising several stages (described in separate sections of the paper):

1. Defining a configuration layer for RISC-V design and verification to check all possible variants.
2. Defining a golden predictor model based on an ISA simulator to decrease RTL-simulation overhead.
3. Utilizing an emulation environment to effectively perform all test suites.

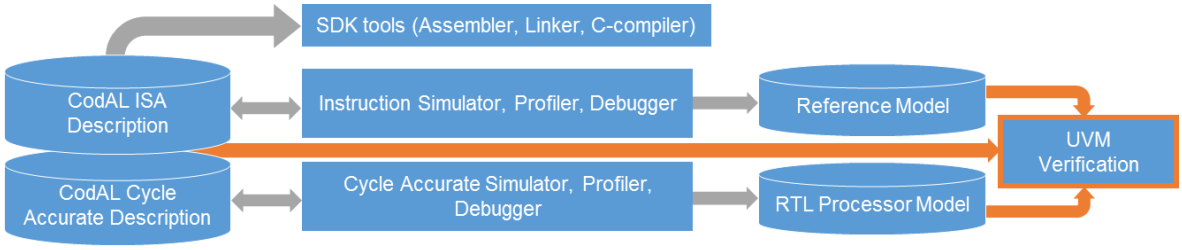

Figure 1: Codasip's flow of generating SDK, RTL, reference models, and UVM verification environment.

II. Defining a Configuration Layer for RISC-V Design and Verification

Considering the possible number of RISC-V core variants, it is not practical to manually implement and maintain all corresponding RTL representations and UVM environments. Automation is advisable, its type depending on the configuration variability of RISC-V cores. This paper will present three options.

For demonstration purposes, we will use two different configurations of the Codasip Berkelium processor:

a) *bk3-32IM-pd*,
b) *bk3-32IMC-pd.*

"I", "M", and "C" stand for standard RISC-V instruction extensions as defined in Section I, "p" signifies enabled hardware parallel multiplier, and "d" enabled JTAG debugging. The difference between the presence and absence of the multiplication and division extension ("M") in the Bk processor configuration practically only consists in the number of supported instructions. From the RTL and UVM point of view, this means that the existing RTL modules are longer, and the instruction decoder is more complicated. However, when the compressed instructions extension ("C") is enabled, many new logic blocks are added to the RTL together with a new dedicated instructions decoder. This requires compiling additional RTL files that will describe these new logic blocks. From the UVM point of view, a new UVM agent for the compressed instructions decoder needs to be compiled and properly connected to the rest of the UVM environment.

1. The first option is to place the configuration layer at the beginning of the automation flow. Codasip does so by inserting the configuration string into the high-level CodAL description; see an example of a GUI configuration entry in Codasip Studio [6] (the license based processor development environment) in Fig. 2. As one can see, the configuration text string consists of three parts divided by dashes. The first part specifies the name of the processor and the number of pipeline stages. The second part contains the used ISA extensions, and the last part specifies optional hardware extensions.

    a. ***bk3-32IM-pd*:** When considering the *bk3-32IM-pd* configuration string in Codasip Studio (Fig. 2), the defined processor model will have three pipeline stages and 32-bit word-width support. It will contain base integer instructions ("I"), instructions for multiplication and division ("M"), and it will have two hardware extensions – parallel multiplier ("p") and JTAG debug interface ("d"). Other settings, such as memory size or caches enabled, can be found in the options table. Once the configuration boxes are completed, the corresponding RTL and UVM code are automatically generated with a single click.

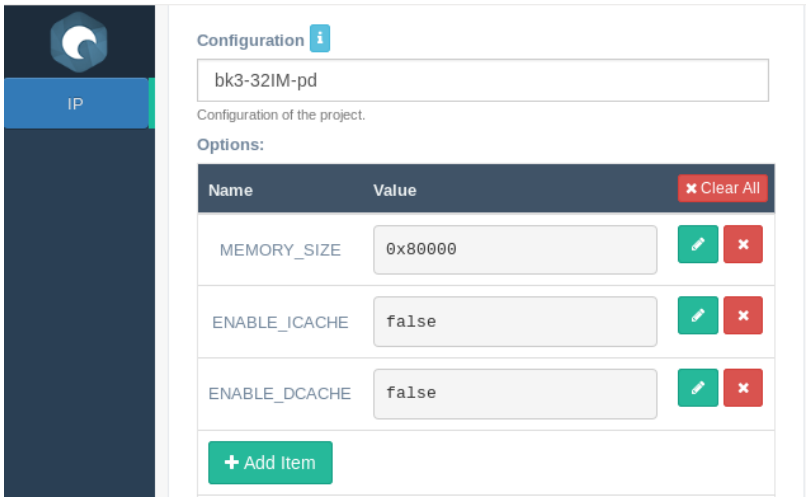

Figure 2: ***bk3-32IM-pd*** configuration in Codasip Studio.

b. ***bk3-32IMC-pd:*** When using the configuration layer in Codasip Studio, no overhead is created by enabling the compressed instructions extension ("C") in the *bk3-32IMC-pd* configuration (Fig. 3). The only action needed is rebuilding of the RTL and UVM environments to reflect the change in the configuration.

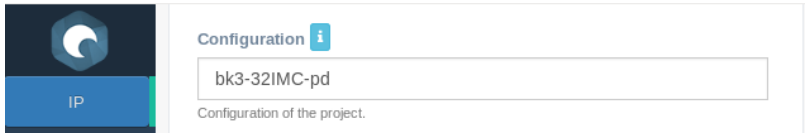

Figure 3: ***bk3-32IMC-pd*** configuration in Codasip Studio.

2. The second option, suitable for manually written RTL and verification environment, is to implement RTL and UVM that can be configured by static compile time constructs such as `ifdef` and related scripts. With this method, only one RTL and one verification environment for multiple RISC-V core variants are needed (one code base for multiple variants).

   a. ***bk3-32IM-pd*:** To compile the source files, we use the compiler define options that are common for RTL and UVM, as seen in Fig. 4. Code snippets in Example 1 show that the configurable extensions are enclosed in their specific define parts. For example, "M" instructions are enclosed in `` `ifdef EXTENSION_M `` in the *decoder.svh* file. Only when this file is compiled with `+define+EXTENSION_M`, the "M" instructions will be recognized by the decoder. The same applies to the part of the example related to coverage. Thus, the principle of this method allows for configuring the UVM environment during compilation before each individual verification run. Furthermore, it makes for easier extending of the UVM environment with new ISA or hardware extensions (when compared to the previous option where the Codasip Studio tool must be updated to accommodate a new configuration).

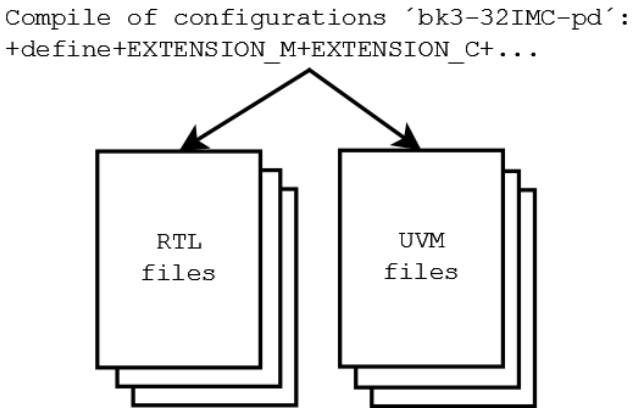

Figure 4: Compilation of all files with defines.

```
/* Decoder description
 * file: codix_berkelium_ca_core_dec_t_decoder.svh
 */

// enumeration code for every decoded instruction
typedef enum {
    add_,
    and_,
    ...
    } m_instruction;

// "M" instructions
`ifdef EXTENSION_M
typedef enum {
    mul_,
    mulh_,
    ...
    } m_instruction_ext_m;
`endif



/* UVM Instructions decoder coverage collection
 * file: decoder_coverage.sv
 */

// Covergroup definition
covergroup FunctionalCoverage( string inst );

    cvp_instructions : coverpoint m_transaction_h.m_instruction {
        illegal_bins unknown_instruction = {UNKNOWN};
        option.comment = "Coverpoint for decoded instructions. Unknown instruction
                          is considered as illegal.";
    }

    `ifdef EXTENSION_M
    cvp_instructions_ext_m : coverpoint m_transaction_h.m_instruction_ext_m {
        illegal_bins unknown_instruction = {UNKNOWN};
        option.comment = "Cover-point for decoded instructions for M extension.
                          Unknown instruction is considered as illegal.";
    }
    `endif
```

Example 1: Code snippets with ifdef parts for ***bk3-32IM-pd*** configuration.

b. ***bk3-32IMC-pd*:** When adding the "C" extension, it is vital to ensure that an additional agent for the compressed instructions decoder will be compiled and connected. As shown in Example 2, `ifdef EXTENSION_C allows compiling the agent package which contains all agent files for the compressed instructions decoder in the *compile.tcl* file, binding the agent to the RTL signals of the processor in the *dut.sv* file and registering it into the UVM configuration database in the UVM environment *env.sv* file.

```
/* Compile script
 * file: compile.tcl
 */

# UVM packages, interfaces and probes compilation
compile_fve_source ${LIBRARY} [list \
...
    [file join agents codix_berkelium_ca_core_dec_t_agent
sv_codix_berkelium_ca_core_dec_t_agent_pkg.sv] \

    # Compilation of compressed instructions decoder
    `ifdef EXTENSION_C
    [file join agents codix_berkelium_ca_core_decompressor_16b32b_t_agent
sv_codix_berkelium_ca_core_decompressor_16b32b_t_agent_pkg.sv] \
    `endif



/* Interconnection of probes and RTL modules
 * file: dut.sv
 */

bind HDL_DUT_U.codix_berkelium.core.dec
icodix_berkelium_ca_core_dec_t_probe probe(
    .ACT(ACT),
    .codasip_param_0(codasip_param_0)
);

// binding of compressed instructions decoder
`ifdef EXTENSION_C
bind HDL_DUT_U.codix_berkelium.core.decompressor_16b32b
icodix_berkelium_ca_core_decompressor_16b32b_t_probe probe(
    .ACT(ACT),
    .codasip_param_0(codasip_param_0)
);
`endif



/* Environment registration to UVM configuration database
 * file: codix_berkelium_ca_env.svh
 */

// main sub-components used to collect instruction coverage
codix_berkelium_ca_core_dec_t_agent m_codix_berkelium_ca_core_dec_t_agent_dut_h;

`ifdef EXTENSION_C
codix_berkelium_ca_core_decompressor_16b32b_t_agent
m_codix_berkelium_ca_core_decompressor_16b32b_t_agent_dut_h;
`endif

uvm_config_db #( codix_berkelium_ca_core_dec_t_agent_config )::set( this,
    "m_codix_berkelium_ca_core_dec_t_agent_dut_h*",
    "codix_berkelium_ca_core_dec_t_agent_config",
     m_cfg_h.m_codix_berkelium_ca_core_dec_t_agent_config_dut_h );

m_codix_berkelium_ca_core_dec_t_agent_dut_h =
codix_berkelium_ca_core_dec_t_agent::type_id::create(
"m_codix_berkelium_ca_core_dec_t_agent_dut_h", this );

`ifdef EXTENSION_C
uvm_config_db #( codix_berkelium_ca_core_decompressor_16b32b_t_agent_config
)::set( this,
    "m_codix_berkelium_ca_core_decompressor_16b32b_t_agent_dut_h*",
    "codix_berkelium_ca_core_decompressor_16b32b_t_agent_config",
     m_cfg_h.m_codix_berkelium_ca_core_decompressor_16b32b_t_agent_config_dut_h );

m_codix_berkelium_ca_core_decompressor_16b32b_t_agent_dut_h =
codix_berkelium_ca_core_decompressor_16b32b_t_agent::type_id::create(
"m_codix_berkelium_ca_core_decompressor_16b32b_t_agent_dut_h", this );
`endif
```

Example 2: Code snippets with ifdef parts for ***bk3-32IMC-pd*** configuration.

3. The third option is to use dynamic runtime configuration approach such as the UVM configuration database *uvm_config_db*, see [7]. This option is similar to using `ifdef` (second option presented in this paper), but it is limited to the UVM environment.

    a. ***bk3-32IM-pd*:** As indicated by the code snippet in Example 3, enabling of the "M" instruction extension is reflected by the `extension_M` parameter which is saved into the UVM configuration database. It is then possible to obtain its value by calling the get function from a specific part of the UVM environment (the coverage file, for example), and then use it for creating new objects.

```
/* Decoder agent configuration
 * file: codix_berkelium_ca_core_dec_t_agent_config.svh
 */

// uvm_config_db configuration and set
function void set_decoder_config_params();
  //set configuration info
  decoder = new();

  decoder.extension_M = 1;
  ...
  uvm_config_db #(decoder_config)::set( this, "*" , " decoder_config" , decoder );
endfunction



/* Decoder agent coverage
 * file: codix_berkelium_ca_core_dec_t_coverage.svh
 */

// Decoder uvm_config_db get and use example
decoder_config dec_config;
if( !uvm_config_db #( decoder_config )::get( this , "" , "decoder_config" ,
dec_config ) ) begin
      `uvm_error(...)
end
...
cvp_instructions : coverpoint m_transaction_h.m_instruction{…}
if( m_config.extension_M ) begin
    cvp_instructions_ext_m : coverpoint m_transaction_h.m_instruction_ext_m {…}
end
```

Example 3: Code snippet with *uvm_config_db* example for ***bk3-32IM-pd*** configuration.

    b. ***bk3-32IMC-pd*:** When we applied the *uvm_config_db* approach to the configuration with the "C" extension enabled, we encountered difficulties with connecting the additional agent and compiling the source files. That tells us that this method is suitable for ISA or hardware extensions that extend the functions of the processor without additional logic files that need to be compiled and connected.

For the comparison of three above-mentioned configuration methods, we summarize the main advantages and disadvantages in Table 1.

TABLE I
ADVANTAGES AND DISADVANTAGES OF THE CONFIGURATION METHODS

	Pros	Cons
Configuration set at higher abstraction level and RTL + UVM are generated	Easy setting of desired configuration. Better readability of generated source files. Very fast.	The generated RTL + UVM are dedicated just to one processor configuration. Price - generator presented in this paper is a paid tool.
Ifdefs for manually written RTL and UVM files	Support of multiple processor configurations.	Worse readability of code in source files.
uvm_config_db for manually written UVM files	Support of multiple processor configurations in UVM source files.	Worse readability of code in source files. Limited only to UVM.

III. Defining a Golden Predictor Model Based on an ISA Simulator

An ISA simulator, or an instruction simulator, is used to execute the instruction stream. High performance is achieved by omitting micro-architectural implementation details. The simulator is usually implemented in C/C++/SystemC, and represents the reference functionality [8].

Codasip generates an ISA simulator from the high-level instruction-accurate CodAL model as part of their automation flow. The ISA simulator is also used as a golden predictor model (called also reference model) in UVM verification, meaning that the RTL processor model (DUT) generated from the high-level cycle-accurate CodAL model is verified against it. There are some obstacles to this approach, such as the asynchronous nature of the C++ ISA model (RTL is cycle-accurate), resolved by memory loaders that can load a program to the program memory consistently in RTL and C++ – after that, both are running on their own speed. In the generated simulation environment, the result comparison is handled by buffering the outputs of the faster component: when data is present in both the golden predictor model FIFO and the DUT FIFO in the Scoreboard, the comparison is performed.

In the Codasip automation flow, assembling the UVM verification environment including the golden predictor model is much faster than in the standard flow, where all components are written manually. Time savings on the verification work are counted in man-months.

IV. Utilizing an Emulation Environment to Effectively Perform Tests

Getting a viable golden predictor model is, however, only the first step. The second important criterion is simulation runtime. Flexibility of the RISC-V ISA allows for implementing tens of viable processor RISC-V micro-architectures. For example, at Codasip we are currently working with 48 variants of RISC-V. Considering that each of these micro-architectures is verified by at least 10,000 programs of around 500 instructions (C-programs, benchmarks, randomly generated assembler programs), the verification runtime is enormous. To handle the effort, we divide the verification runtime into phases. In the first phase, we run suitable program representatives in RTL simulation, benefiting from very good visibility and debug capabilities of the simulator. In the second phase, after debugging, we run the rest of the programs (mainly random ones) in the emulation environment.

The main drawback of RTL simulation is the inability to perform tasks in parallel. A good solution is to use emulation and exploit inherent parallelism of real hardware. Several papers and books have already been published that present the possible runtime improvements, for example [9]; our goal in this paper is to show how verification of the RISC-V processors in particular can benefit from emulation.

Our path of porting a quite complex UVM environment for Berkelium processors to the Veloce emulator [10] was not straightforward; based on our experience, we defined the following recommendations.

1. We started by comparing pure simulation and pure emulation environment runtimes. This means that we measured the time of loading a specific program to the program part of the memory and the runtime of evaluating this program on the RISC-V Berkelium processor in UVM in the Mentor Questa RTL simulator, and the runtime of evaluating the same program on the Berkelium processor located on the emulator. In this case, we used a very simple emulator top-level module which instantiates the Berkelium processor. At the beginning, the program is loaded, and by deactivating the processor's RESET signal it starts processing the program. A simple comparison of this type clearly indicates the maximum emulation performance for a specific DUT, as no software counterparts are slowing it down. In addition, it is also possible to estimate the benefits of using the emulator for a specific project. For example, we estimated that we can achieve at least 100 times verification acceleration when using the emulator.

2. As a second step, we recommend creating two top-levels: the emulator top-level module and the test-bench top-level module. The emulator top-level module instantiates the Berkelium processor, and the test-bench top-level module contains a simple SystemVerilog class with two pipes to start processing programs and detect the end. It also initiates comparison of the content of registers to the reference content, which is for now done by a simple diff, so no golden predictor model is used in this phase. This approach makes it possible to run more programs on the processor and detect first bottlenecks. For example, we found out that processing the programs is very fast, but loading new programs is ineffective and decreases the emulation performance 25 times. To eliminate this problem, we used the Mentor Veloce FlexMem blocks later in the process, as described in the next section of this paper.

3. Finally, we recommend to connect all UVM objects you intend to use into the test-bench top-level module, as they usually add some additional bottlenecks. We connected SystemVerilog programs loader, Codasip C++ ISA simulator as the reference model, one active UVM agent to drive processor's input ports and read the decoded instructions (important for measuring instruction coverage and instruction sequences coverage), and passive UVM agents for reading the transactions on buses, content of architectural registers, and content of memory, as these are compared to the reference results originating from the reference model. The emulation performance decreased so much that we started with profiling in Questa as well as in the emulator. The result was that the Mentor Veloce FlexMem blocks are very effective for loading programs from software to the emulator, however, we had to implement so-called "transactors" [11] for driving processor's input ports from the active UVM agent, for monitoring decoded instructions from the processor, and for detecting the end of the program. We also realized that having the golden predictor model and Scoreboard comparison as part of the software test-bench is not effective, as moving the content of transactions, registers and memories between the emulation top-level and the test-bench top-level negatively impacts the runtime performance. Thus we decided to locate the predictor outside of the test-bench top-level and to leave it up to the emulation top-level to trigger the results comparison by the diff tool when the end of the program is detected. This means that the golden predictor model is running in parallel to the emulation, and we used dumping of DUT as well as reference data resources from both the golden predictor model and from the processor located in the emulator. For illustration of the resultant environment, see Fig. 5.

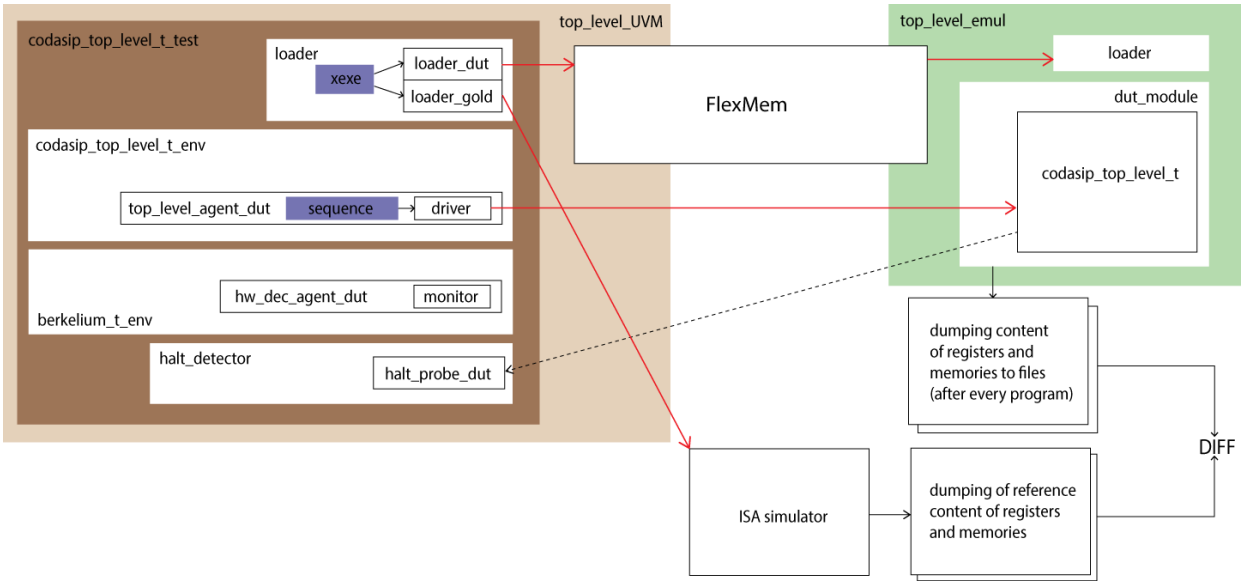

Figure 5: Codasip UVM ported to the Veloce emulator.

We achieved the results shown in Table 2 by employing the three mentioned steps. The current version achieves **75x** acceleration, but we are working on further optimizations including data aggregation on the test-bench and on the emulation side, and measuring instruction coverage on the emulator directly.

TABLE II
EMULATION PERFORMANCE RESULTS

	Average runtime for 1 program in seconds (~100 000 instructions)	Acceleration achieved
Pure simulation-based verification vs. pure emulation-based verification	**128 (simulation)** **1.28 (emulation)**	**100 ×**
Emulation-based verification with simple test-bench and pipes	**32**	**4 ×**
Emulation-based verification with UVM, FlexMem and external ISA simulator	**1.7**	**75 ×**

V. SUMMARY

This paper covered three topics that result from the flexibility of RISC-V IP cores. The first topic described usage of the configuration layer, and was divided into three sub-parts. The first part introduced the automation flow used in the processor development environment called Codasip Studio. This flow enables the user to simply input desired supported configuration details, and the Studio automatically generates all the tools needed for verification and application development.

The second part explained the usage of defines in RTL and UVM files. This part showed that the user can have multiple configurations implemented in one package of source files. This is possible thanks to compiler defines allowing to mark parts of the code that are specific to individual processor extensions.

The third part elaborated on the procedure of using a UVM configuration database. The advantage of this approach is the integrated configuration in UVM itself. As it is restricted to UVM files, RTL needs to only include files for one configuration at a time.

We transformed a pure simulation-based UVM environment into an emulation environment employing the best practices, and measured the acceleration results. In cooperation with Mentor, we identified parts of the processor that required specific treatment to achieve improved emulation performance. Excluding the predictor model from UVM, implementing transactors, and utilizing `diff` comparison outside of the UVM allowed us to decrease the burden of data transfers between software and the emulation hardware. Also, utilizing Mentor Veloce FlexMem approach when loading programs to the program memory or reading data from the memory proved to be much favorable over the standard approach of communicating directly with emulator memory.

REFERENCES

[1] RISC-V Foundation. (2017, July) *RISC-V Specifications* [Online]. Available: https://riscv.org/specifications/
[2] David Patterson and John Hennessy. (2017) *Computer Organization and Design, RISC-V Edition*, Morgan Kaufmann.
[3] John Shen and Mikko Lipasti. (2013) *Modern Processor Design*, Waveland Press.
[4] Jari Nurmi. (2007) *Processor Design*, Springer.
[5] Marcela Zachariášová, Zdeněk Přikryl, et al. (2013) "Automated Functional Verification of ASIPs," in *IFIP Advances in Information and Communication Technology*. Springer Verlag, pp. 128–138.
[6] Codasip Studio [Online]. Available: https://www.codasip.com/
[7] Verification Academy. (2017, July) *UVM Configuration* [Online]. Available: https://verificationacademy.com/cookbook/configuration
[8] Rainer Leupers and Olivier Temam. (2010) *Processor and System-on-Chip Simulation*, Springer
[9] Hans van der Schoot and Ahmed Yehia. (2015) "UVM and Emulation: How to Get Your Ultimate Testbench Acceleration Speed-up", DVCon Europe 2015.
[10] *Veloce emulator* [Online]. Available: https://www.mentor.com/products/fv/emulation-systems/
[11] Janick Bergeron, Alan Hunter, Andy Nightingale, Eduard Černý. (2006) *Verification Methodology Manual for SystemVerilog.* Springer.