embeddedworld2019
Exhibition&Conference
... it's a smarter world

Methodology for Implementation of Custom Instructions in the RISC-V Architecture

Lee Moore, Simon Davidmann and Larry Lapides
Imperas Software Ltd.
Oxford, United Kingdom
larryl@imperas.com

Carl Shaw
Cerberus Security Labs
Bristol, United Kingdom
carl.shaw@cerb-labs.com

***Abstract*—One of the key advantages of the new RISC-V Instruction Set Architecture (ISA) is that SoC designers are able to add custom features to the ISA to support their specific applications. There are some risks to doing this, both business and technical, plus there is a need to be able to analyze and optimize the customizations.**

One approach to customization of RISC-V cores is to use correct-by-construction tools to generate both the compliant and custom pieces of the processor. A second approach is to implement the custom features directly in RTL.

With both approaches to implementation, there is still the need for both compliance testing and analytical feedback to enable optimization of the customizations.

This paper discusses the alternatives for implementation, and describes an instruction accurate virtual platform methodology for compliance testing and architecture exploration. In this methodology, there is an existing parameterized model of the RISC-V ISA specification, and the custom features are added in an external library. This has the advantage of providing a well-verified compliant model, while at the same time enabling the use of the software debug, analysis and test tools in the virtual platform environment.

A case study involving the addition of custom security functionality to a 32-bit RISC-V core is presented, including the compliance testing, memory analysis, function and instruction profiling including timing estimation.

Keywords*—*RISC-V, processor, custom instructions, virtual platforms, processor models, simulation

I. Introduction

The new RISC-V Instruction Set Architecture (ISA) [1] has received a growing amount of attention recently. This may be due to the open nature of the ISA, so that processor developers can build their own processor implementations and add custom features. In addition RISC-V has enabled creative business models that includes academic projects, open source cores and a growing IP ecosystem of commercial vendors. The RISC-V ISA is modular, in that developers can choose to implement only the base integer instructions, or any of the instructions subsets, such as Multiply (M), Atomic (A), Floating Point (F), Double Precision Floating Point (D) and Compressed (C). In addition to these subsets, additional subsets are being defined by working groups in the RISC-V community, such as for vector instructions and bit manipulation instructions. Also, developers can choose to only implement the base machine privilege level, or also implement supervisor and user privilege modes.

While other ISAs used in embedded systems are set, the RISC-V ISA has the advantage of providing processor developers with the ability to add custom features – instructions, registers, … – to their implementations of the RISC-V architecture. This ability to customize means that RISC-V cores can be adapted to fit the functional requirements of a SoC. RISC-V cores are also being looked at for IoT applications at the low performance/power end for edge applications, and at AI and machine learning applications at the high end, again taking advantage of the ability to customize the cores.

However, with customization comes risks. Some of the risks are of more a business flavor, while some are more technical in nature. For example, customization means having to also customize the tool chain. Customization means having to optimize the complete processor implementation, and the extra effort of verifying the customizations.

This paper starts by discussing the alternative approaches to the implementation of customizations to the RISC-V cores. Architecture analysis and optimization methodology is presented, finishing with a case study on the addition of custom security features to a 32-bit RISC-V core.

II. Implementing Custom RISC-V Features

A. Implementation Methodology Overview

Most developers regard the addition of custom features as a hardware design task, similar to adding custom components to the SoC as RTL. Taking this path of adding the custom features to the RTL first is not recommended, however, it is instructive in the context of RISC-V to understand how this could be done. Note that while features such as additional system registers can be added to the processor, in this paper the focus is on custom instructions.

The RISC-V processor specification has several deliberately defined decode spaces, for example custom0, custom1 etc. into which new custom instructions can be added.

B. *Roll-Your-Own, Use Open Source RTL, or Processor IP Vendors*

The RISC-V ISA is open, so that anyone can implement that ISA in RTL themselves, the "roll-your-own" approach. Alternatively, there are a number of open source RTL implementations that can be used [2], or one could license an implementation from a growing list of RISC-V processor IP vendor [3]. Factors that should be taken into account in this decision include the experience and expertise of in-house engineering resources with respect to processor design and verification, the degree of compliance to the RISC-V specification required for the specific use case, the ease of adding custom features in the various environments.

C. *Adding Custom Features*

In the case of roll-your-own or using open source implementations, adding the custom features is a matter of modifying the RTL to add in the custom feature. This feature then needs to be thoroughly verified, both for its own functionality as well as to ensure that the addition of this feature did not break any other piece of the processor.

For most of the commercially available RISC-V processor IP, the vendors provide an environment or tool for adding custom instructions. For many of these, the tool offering includes automatically generating many of the ancillary "ecosystem" tools and models, such as the compiler, debugger and simulation models.

D. *Compliance*

For most ISAs compliance to the ISA specification is a given. Since all the SoC designers license the RTL from a single source, of course the RTL complies with the ISA specification. Also, the IP vendor is going to support its own ecosystem, and ensure that those ecosystem partners are in compliance with the ISA spec.

With the new, open standard RISC-V ISA, the compliance situation is different. There is no single IP vendor, no single source, so compliance is an issue that needs to be addressed by the RISC-V community. The RISC-V Foundation has formed a task group to develop the compliance procedures for the RISC-V community. The initial results of the task group – compliance methodology, test suites, reference simulators – are available to the community from a GitHub repository [4], and are described in the paper by Moore et al [5].

III. CUSTOM INSTRUCTION FLOW

Custom features, such as additional instructions, are needed because some product use case will execute more efficiently with those custom features. A comprehensive flow – characterizing and performing functional validation of the key application on the standard processor, extending the processor with custom instructions, analysis of the application execution with the custom instructions, optimization of the custom instructions – is critical to the success of the custom instructions and the target product or application. Fig. 1 shows this flow.

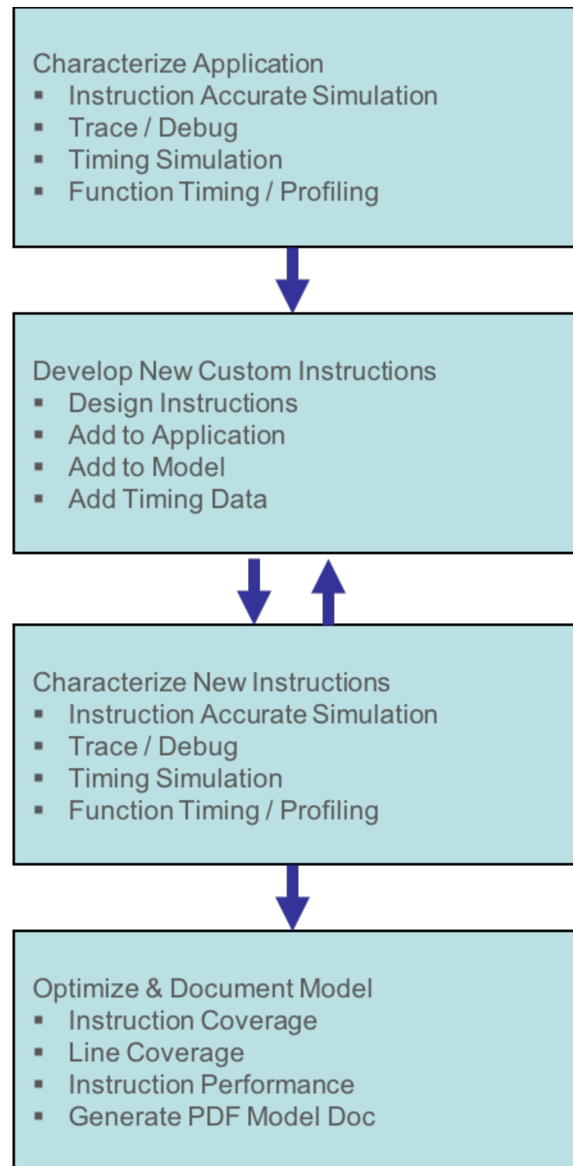


Fig. 1. Flow for adding custom features to a RISC-V processor.

The key to this flow is having a high-performance instruction accurate (IA) simulator, which supports the models needed and the tools for architectural analysis, including timing estimation. Note that this IA simulation based flow is a bit different than how most of the processor industry thinks about processor analysis and optimization. However, for most processors the analysis and optimization focuses on microarchitectural features such as the processor pipeline and on various levels of cache, and in this case, cycle accurate simulation and models for detailed timing analysis are required, along with pipeline simulators such as gem5 [6].

For RISC-V, the goal is to optimize the processor performance by adding custom instructions, so the focus is on the target applications and the instructions, and not on the microarchitecture. Tools such as instruction coverage,

instruction profiling, code coverage and timing estimation (90% accuracy required) are needed. Certainly, to achieve the absolute peak microarchitectural performance for a RISC-V processor, pipeline and cache optimization are required, and so the conventional cycle accurate simulation and pipeline simulator tools are needed. However, that is not the focus of this paper.

IV. Case Study: Adding Security Instructions

A. Case Study Setup

In this section the above flow is applied to the design of a 32-bit RISC-V processor, with a security application as the target. The base virtual platform is configured as just processor plus memory, as shown in Figs. 2 and 3. The base processor model of the RV32IM processor comes from the Open Virtual Platforms (OVP) Library [7]. This processor model is built using the OVP APIs, and is an open source model, distributed under the Apache 2.0 open source license.

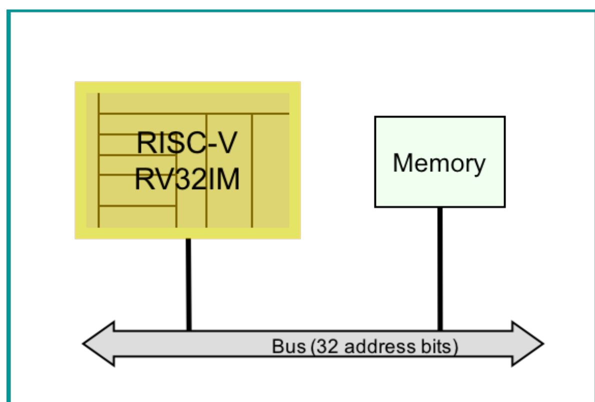


Fig. 2. Virtual platform block diagram.

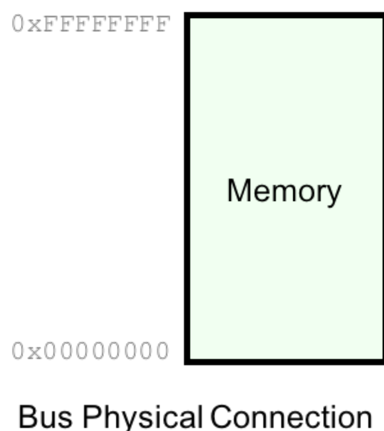


Fig. 3. Virtual platform address mapping.

The IA simulation environment includes the M*SDK and cpuDev products from Imperas [8]. These products provide such features as the IA simulator engine, software debugging, and the software and processor analysis tools required for the analysis and optimization of custom instructions.

The application is based upon the chacha20 encryption algorithm [9]; it takes data from a file and performs encryption upon it a line at a time.

B. Phase 1: Characterize the Application

The first step is to execute the application on the virtual platform, and confirm the functional correctness of the application. This is shown in Fig. 4.

The next step is timing estimation, using the Instruction Accurate + Estimation (IA+E) technology described elsewhere [5]. IA+E simulation is enabled by loading an extension library that, amongst other things, monitors the instruction stream and memory accesses and provides information back to the simulator so that it can modify the instruction execution rate accordingly.

Now that the application is verified and a timing baseline has been established, the application is profiled (based on analysis of C functions executed) to understand where the performance bottlenecks may occur. See Fig. 5 for the profiling output. From the profiling results it is seen that the application is spending a large percentage of the execution time in the *qrN_c* function that implements the core of the chacha20 algorithm, with almost as much again in the combined total of the the *qr1_c, qr2_c, qr3_c* and *qr4_c* functions.

C. Phase 2: Develop New Custom Instructions

The base processor model and the custom instructions are both developed in the same way, using the OVP VMI APIs. A key aspect of this custom instruction methodology is that the custom instructions are described in an instruction extension library. The base processor model, which has been extensively tested and verified by both developers and users, is not perturbed, so that the high-quality base model is left untouched.

The instruction extension library to be built for this application will include four custom instructions. Each uses the same base behavior, but applies a different rotation value. In the RISC-V ISA these will be R-Type instructions in custom-1 decode space, defined as shown in Table 1.

TABLE I. Decode Table for the Custom Instructions.

Bits	Bit Value	description
6 - 0	00 010 00	Custom-1 instruction class decode
11 - 7	xxxxx	Identify the result register
14 - 12	000 001 010 011 1xx	QR1 QR2 QR3 QR4 Undefined
19 - 15	xxxxx	Identify source register 1
24 - 20	xxxxx	Identify source register 2
31 - 25	0000000	Instruction decode

The instruction behavior is then implemented in the VMIOS_MORPH_FN callback. The code in Fig. 6 shows a common function is used with a rotation value passed as one of the arguments.

In a similar manner, the timing estimation library can be extended to add the timing information for the custom instructions.

D. Phase 3: Characterize the New Custom Instructions

In this phase tools such as instruction profiling and timing estimation are used to understand the execution of the application with the custom instructions.

The application can now be executed on the virtual platform, including the instruction extension library and timing extension library, as shown in Fig. 7. This shows that the application still provides the correct results, that fewer total instructions were executed, and that the simulation time has been reduced. Note that in the Imperas tool environment the addition of the custom instructions has no effect on the use of the tools; all tools work with the new custom instructions.

Function profiling results are shown in Fig. 8, showing there is no longer the appearance of the C algorithm functions we saw previously, as this behavior is now performed by the custom instructions.

There may be iteration of Phases 2 and 3 to develop and characterize the new custom instructions.

E. Phase 4: Model Optimization and Documentation

When a custom instruction has been created we need to be able to ensure that it is fully tested and that it is an efficient implementation. One useful tool is instruction coverage. Like conventional code coverage, instruction coverage is monitoring the execution to verify that code has been tested, however in the case of instruction coverage the execution being monitored is the processor model and the instruction extension library instead of the application source code. Instruction coverage results are shown in Fig. 9, including showing coverage for the custom instructions chacha20qr<N>.

The new model, including the based processor model plus the instruction extension library, should be documented. This documentation can serve as the specification for the implementation of the custom instructions.

V. Using the Model

Now that the custom instructions have been optimized in the model, there are several uses of the customized processor model. The first key use is as a golden reference model for design verification of the implemented RTL; i.e. comparing the RTL to the model. SoC verification is the most time-consuming and resource-consuming task in hardware design. For most SoCs, the processor IP has been purchased from a vendor, and does not need to go through additional verification. In this case, with a customized RISC-V processor, verification of the processor implementation is a key task in the design cycle.

The second key use is for early, pre-silicon, pre-RTL software development. The processor model can be used in a virtual platform environment for operating system and driver porting and bring up, and for firmware and application development. Because the processor model, or the virtual platform of the SoC, is available prior to RTL completion, instruction accurate software simulation can accelerate software development by months versus hardware emulators or FPGA prototypes.

VI. Conclusions

This paper has presented an overview of the complete flow for adding custom instructions to a RISC-V processor, while providing a detailed description of how instruction accurate simulation tools and models can be used to analyze and optimize custom instructions.

Acknowledgment

The authors would like to thank Duncan Graham, who was the source for the figures and tables.

References

[1] The RISC-V ISA specification is available here: https://riscv.org/specifications/.

[2] https://riscv.org/risc-v-cores/

[3] Andes Technology, Codasip, SiFive and Syntacore are among the RISC-V processor IP vendors. The RISC-V growing ecosyem directory of members can be found at https://riscv.org/members-at-a-glance/

[4] https://github.com/riscv/riscv-compliance/

[5] L. Moore, D. Graham, S. Davidmann and F. Rosa, Cycle approximate simulation of RISC-V processors, embedded world conference 2018. http://www.imperas.com/ew18-slides-on-using-an-ia-simulator-with-timing-estimation-to-provide-high-performance-cycle

[6] http://www.gem5.org/

[7] Open Virtual Platforms (OVP) Library: http://www.ovpworld.org/library/wikka.php?wakka=Library

[8] Imperas product web page: http://www.imperas.com/products

[9] https://www.cryptopp.com/wiki/ChaCha20


```
IMPERAS Instruction Set Simulator (ISS)


CpuManagerMulti (64-Bit) v20180716.0 Open Virtual Platform simulator from www.IMPERAS.com.
Copyright (c) 2005-2018 Imperas Software Ltd.  Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

CpuManagerMulti started: Tue Sep 18 08:47:27 2018
                                        Application program load
Info (OR_OF) Target 'iss/cpu0' has object file read from 'application/test_c.RISCV32.elf'
Info (OR_PH) Program Headers:
Info (OR_PH) Type           Offset     VirtAddr   PhysAddr   FileSiz    MemSiz     Flags Align
Info (OR_PD) LOAD           0x00000000 0x00010000 0x00010000 0x00017430 0x00017430 R-E   1000
Info (OR_PD) LOAD           0x00017430 0x00028430 0x00028430 0x000009c0 0x00000a24 RW-   1000
```

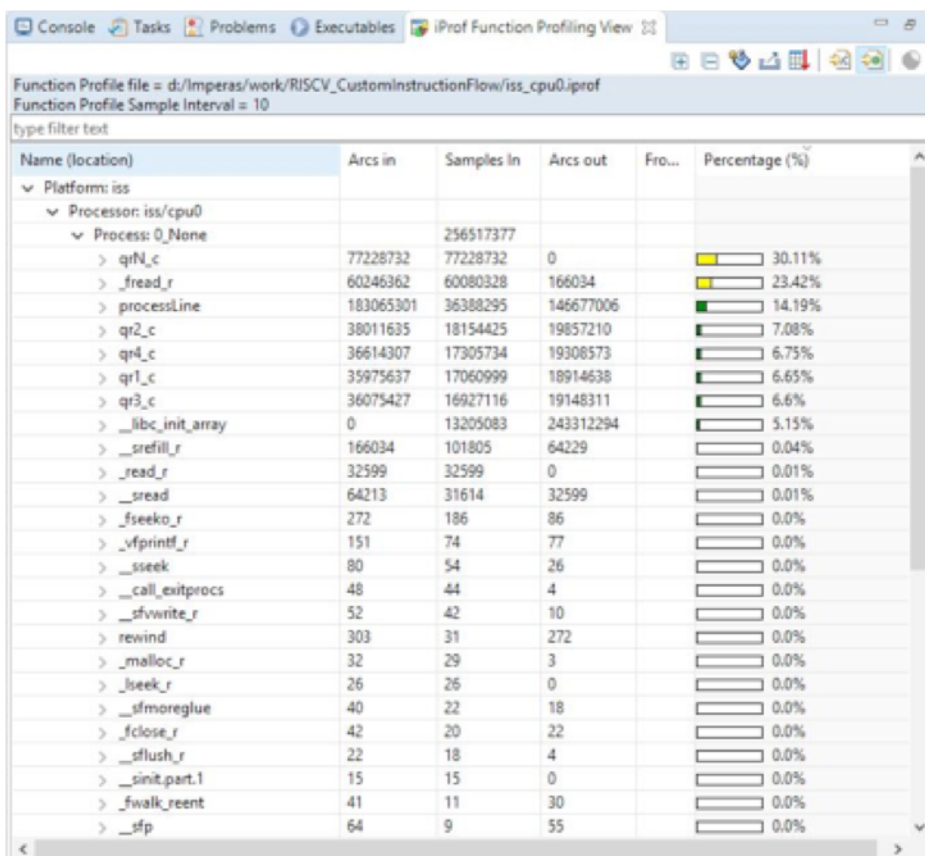
```
Info (OR_OF) Target 'iss/cpu0' has object file read from 'application/exception.RISCV32.elf'
Info (OR_PH) Program Headers:
Info (OR_PH) Type           Offset     VirtAddr   PhysAddr   FileSiz    MemSiz     Flags Align
Info (OR_PD) LOAD           0x00001000 0x00000000 0x00000000 0x0000000c 0x0000000c R-E   1000

RES = 84772366

Info
Info ---------------------------------------------------
Info CPU 'iss/cpu0' STATISTICS
Info   Type                 : riscv (RV32IM)
Info   Nominal MIPS         : 100
Info   Final program counter : 0x100ac
Info   Simulated instructions: 1,994,024,026
Info   Simulated MIPS        : 1160.2
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 19.94 seconds
Info   User time             : 1.59 seconds
Info   System time           : 0.13 seconds
Info   Elapsed time          : 1.80 seconds
Info   Real time ratio       : 11.10x faster
Info ---------------------------------------------------

CpuManagerMulti finished: Tue Sep 18 08:47:28 2018
```

Functionality verified

Execution statistics of application on processor

Execution statistics of simulation

Fig. 4. Run summary and statistics from initial run of the target application.



Function Profile file = d:/Imperas/work/RISCV_CustomInstructionFlow/iss_cpu0.iprof
Function Profile Sample Interval = 10

Name (location)	Arcs in	Samples In	Arcs out	Fro...	Percentage (%)
Platform: iss					
Processor: iss/cpu0					
Process: 0_None		256517377			
qrN_c	77228732	77228732	0		30.11%
_fread_r	60246362	60080328	166034		23.42%
processLine	183065301	36388295	146677006		14.19%
qr2_c	38011635	18154425	19857210		7.08%
qr4_c	36614307	17305734	19308573		6.75%
qr1_c	35975637	17060999	18914638		6.65%
qr3_c	36075427	16927116	19148311		6.6%
__libc_init_array	0	13205083	243312294		5.15%
__srefill_r	166034	101805	64229		0.04%
_read_r	32599	32599	0		0.01%
__sread	64213	31614	32599		0.01%
_fseeko_r	272	186	86		0.0%
_vfprintf_r	151	74	77		0.0%
__sseek	80	54	26		0.0%
__call_exitprocs	48	44	4		0.0%
__sfvwrite_r	52	42	10		0.0%
rewind	303	31	272		0.0%
_malloc_r	32	29	3		0.0%
_lseek_r	26	26	0		0.0%
__sfmoreglue	40	22	18		0.0%
_fclose_r	42	20	22		0.0%
__sflush_r	22	18	4		0.0%
__sinit.part.1	15	15	0		0.0%
_fwalk_reent	41	11	30		0.0%
__sfp	64	9	55		0.0%

Fig. 5. Function profiling results.

```
static VMIOS_MORPH_FN(doMorph) {

    // decode the instruction to get the type
    Uns32 instruction;
    riscvEnhancedInstrType type = getInstrType(object, processor, thisPC, &instruction);

    *opaque = True;
    if(type==RISCV_EIT_CHACHA20QR1) {
        emitChaCha20(processor, object, instruction, 16);

    } else if (type==RISCV_EIT_CHACHA20QR2) {
        emitChaCha20(processor, object, instruction, 12);

    } else if (type==RISCV_EIT_CHACHA20QR3) {
        emitChaCha20(processor, object, instruction, 8);

    } else if (type==RISCV_EIT_CHACHA20QR4) {
        emitChaCha20(processor, object, instruction, 7);

    } else {
        *opaque = False;
    }

    // no intercept callback specified
    return 0;
}
```

Fig. 6. Custom instruction behavior defined.

```
IMPERAS Instruction Set Simulator (ISS)


CpuManagerMulti (64-Bit) v20180716.0 Open Virtual Platform simulator from www.IMPERAS.com.
Copyright (c) 2005-2018 Imperas Software Ltd.  Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

CpuManagerMulti started: Tue Sep 18 10:22:09 2018


Info (OP_LPR) Processor  iss/cpu0
C:\Imperas\lib\Windows64\ImperasLib\riscv.ovpworld.org\processor\riscv\1.0\model
Info (OR_OF) Target 'iss/cpu0' has object file read from 'application/test_custom.RISCV32.elf'
Info (OR_PH) Program Headers:
Info (OR_PH) Type           Offset     VirtAddr   PhysAddr   FileSiz    MemSiz     Flags Align
Info (OR_PD) LOAD           0x00000000 0x00010000 0x00010000 0x00017270 0x00017270 R-E   1000
Info (OR_PD) LOAD           0x00017270 0x00028270 0x00028270 0x000009c0 0x00000a24 RW-   1000
Info (OR_OF) Target 'iss/cpu0' has object file read from 'application/exception.RISCV32.elf'
Info (OR_PH) Program Headers:
Info (OR_PH) Type           Offset     VirtAddr   PhysAddr   FileSiz    MemSiz     Flags Align
Info (OR_PD) LOAD           0x00001000 0x00000000 0x00000000 0x0000000c 0x0000000c R-E   1000
Info (OP_PEX) Extension  iss/cpu0/riscv32Newlib
C:\Imperas\lib\Windows64\ImperasLib\riscv.ovpworld.org\semihosting\riscv32Newlib\1.0\model
Info (OP_PEX) Extension  iss/cpu0/exInst                          instructionExtensionLib
RES = 84772366
Info
Info ---------------------------------------------------------
Info CPU 'iss/cpu0' STATISTICS
Info   Type                  : riscv (RV32IM)
Info   Nominal MIPS          : 100
Info   Final program counter : 0x100ac
Info   Simulated instructions: 677,012,570
Info   Simulated MIPS        : run too short for meaningful result
```

Functionality verified

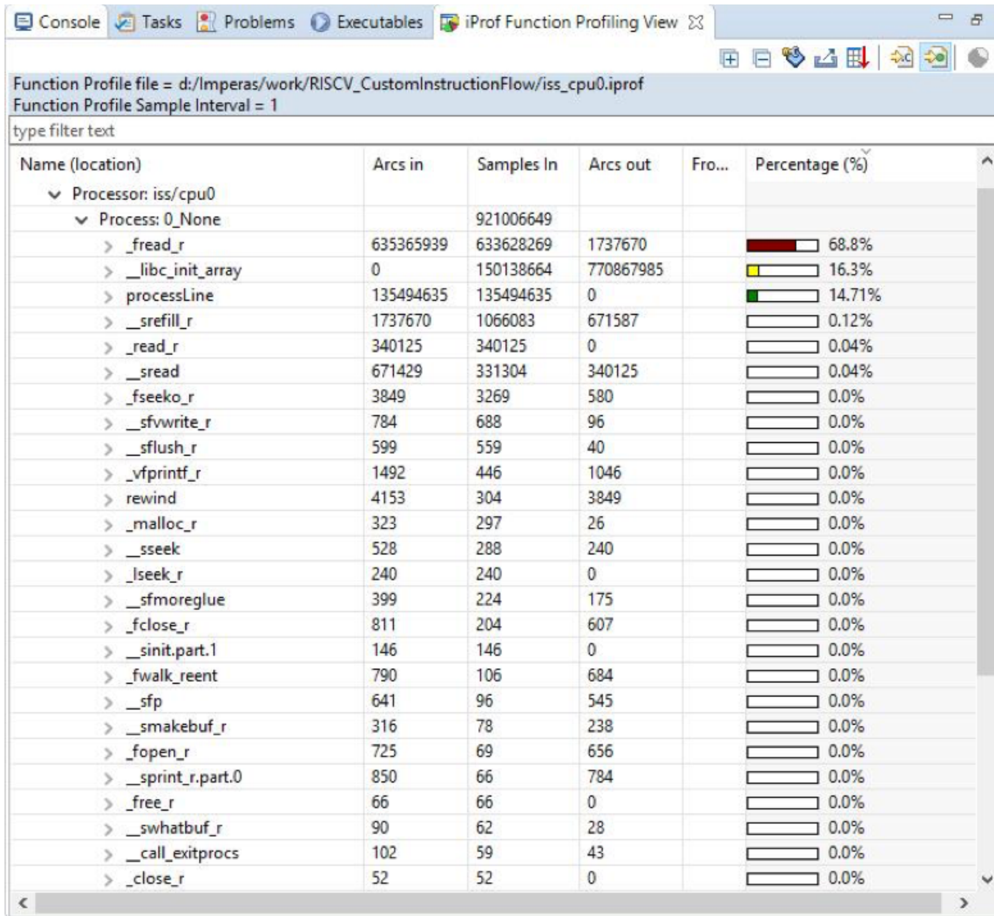
Custom instruction extension library loaded

Less instructions executed

```
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 6.77 seconds
Info   User time             : 0.25 seconds
Info   System time           : 0.09 seconds
Info   Elapsed time          : 0.34 seconds
Info   Real time ratio       : 19.74x faster
Info ---------------------------------------------------

CpuManagerMulti finished: Tue Sep 18 10:22:10 2018
```

Lower simulation time

Fig. 7. Execution summary of the application on the modified processor model, including the custom instructions.



Name (location)	Arcs in	Samples In	Arcs out	Fro...	Percentage (%)
Processor: iss/cpu0					
Process: 0_None		921006649			
_fread_r	635365939	633628269	1737670		68.8%
__libc_init_array	0	150138664	770867985		16.3%
processLine	135494635	135494635	0		14.71%
__srefill_r	1737670	1066083	671587		0.12%
_read_r	340125	340125	0		0.04%
__sread	671429	331304	340125		0.04%
_fseeko_r	3849	3269	580		0.0%
__sfvwrite_r	784	688	96		0.0%
__sflush_r	599	559	40		0.0%
_vfprintf_r	1492	446	1046		0.0%
rewind	4153	304	3849		0.0%
_malloc_r	323	297	26		0.0%
__sseek	528	288	240		0.0%
_lseek_r	240	240	0		0.0%
__sfmoreglue	399	224	175		0.0%
_fclose_r	811	204	607		0.0%
__sinit.part.1	146	146	0		0.0%
_fwalk_reent	790	106	684		0.0%
__sfp	641	96	545		0.0%
__smakebuf_r	316	78	238		0.0%
_fopen_r	725	69	656		0.0%
__sprint_r.part.0	850	66	784		0.0%
_free_r	66	66	0		0.0%
__swhatbuf_r	90	62	28		0.0%
__call_exitprocs	102	59	43		0.0%
_close_r	52	52	0		0.0%

Fig. 8. Function profiling of the application running on the modified processor model, including the custom instructions.

```
Info (ICR_OF) cpu0: instruction profile report file 'coverageReports\cpu0.icr.txt'
# iss/cpu0: Instruction Profile Totals:
#         OpCode :    Pct : Count (Total instrs=       677012570)
# -------------- : ------ : --------
            mv : 19.87% :        134497608
            lw : 16.78% :        113575358
            sw : 13.05% :         88327451
          addi :  8.70% :         58868837
          andi :  4.97% :         33620233
          beqz :  4.35% :         29426103
```

```
          bnez :  3.73% :         25248055
           add :  3.73% :         25231546
          slli :  3.10% :         21020892
           ret :  2.49% :         16843028
           jal :  2.49% :         16843026
          srli :  2.48% :         16810160
          bltu :  2.48% :         16793686
          bgeu :  1.24% :          8421494
          bltz :  1.24% :          8421470
             j :  1.24% :          8388742
   chacha20qr1 :  1.24% :          8388608
   chacha20qr2 :  1.24% :          8388608
   chacha20qr3 :  1.24% :          8388608
   chacha20qr4 :  1.24% :          8388608
            lh :  0.63% :          4243601
         auipc :  0.62% :          4210858
           sub :  0.62% :          4210835
           xor :  0.62% :          4210707
           not :  0.62% :          4194307
           beq :  0.00% :            16546
          jalr :  0.00% :            16430
          blez :  0.00% :            16407
            sb :  0.00% :              206
           lhu :  0.00% :               97
            sh :  0.00% :               82
           lui :  0.00% :               67
            or :  0.00% :               50
           and :  0.00% :               49
           lbu :  0.00% :               47
           bne :  0.00% :               34
           ori :  0.00% :               29
            jr :  0.00% :               25
           bge :  0.00% :               17
           neg :  0.00% :               16
           blt :  0.00% :               12
          srai :  0.00% :                8
          bgtz :  0.00% :                7
          bgez :  0.00% :                6
           sll :  0.00% :                4
          seqz :  0.00% :                2
         csrrc :  0.00% :                0
        csrrci :  0.00% :                0
         csrrs :  0.00% :                0
        csrrsi :  0.00% :                0
         csrrw :  0.00% :                0
```

Fig. 9. Instruction coverage results for the processor model plus the instruction extension library.