Imperas RISCV Custom Instruction Flow Application Note

Imperas Software Limited

Imperas Buildings, North Weston,
Thame, Oxfordshire, OX9 2HA, UK
docs@imperas.com

Author:	Imperas Software Limited
Version:	1.0
Filename:	Imperas_RISCV_Custom_Instruction_Flow_Application_Note.doc
Project:	RISC-V Custom Instruction Flow Application Note
Last Saved:	August 7, 2020
Keywords:	OVP RISCV Custom Instruction Extension

Table of Contents

1 Preface

This document provides a user, who is intending to add a custom instruction to a processor modeled using the OVP APIs, a flow through the stages from the analysis of an application to the optimization of the extension instruction implementation.

1.1 Related Documents

The reader should be familiar with Open Virtual Platforms (OVP) VMI modeling APIs as described in the following OVP documentation:

- *OVP VMI Morph Time Reference Guide*
- *OVP VMI Run Time Reference Guide*

2 Background

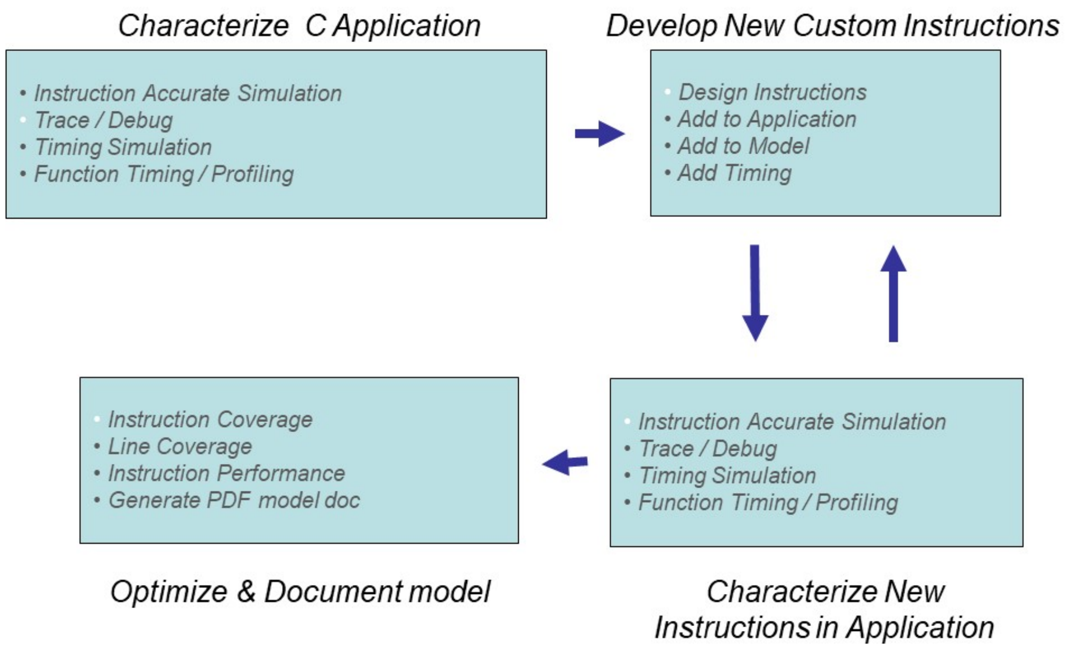
The OVP Fast Processor models are executed using an Instruction Accurate simulator. An instruction accurate simulator approximates the execution of an instruction to a single unit of time. The unit of time is based upon the processor MIPS (Millions of Instructions Per Second) rate.
An OVP Fast Processor model will contain the implementation of all the instructions defined by the processor ISA.

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

The OVP Fast processor models can be extended without modification to the pre-compiled and verified base processor model source code using one or more extension libraries. An extension library can be loaded as part of a virtual platform definition in addition to the base processor model and can provide decode and implementation of behavior for the instructions as well as additional registers.

Including more timing information for the instruction execution provides a cycle approximate simulation. This provides a better approximation to the time of the execution of an application on the actual hardware. The additional timing information is loaded into the virtual platform as an extension library so there is no change to the functional behavior provided by the processor model.

This application note will go through the complete flow of functional validation of the application, extension of the processor with custom instructions, analysis of the application execution and optimization of the custom instruction implementation and its documentation.


Characterize C Application
• Instruction Accurate Simulation
• Trace / Debug
• Timing Simulation
• Function Timing / Profiling
Develop New Custom Instructions
• Design Instructions
• Add to Application
• Add to Model
• Add Timing
• Instruction Accurate Simulation
• Trace / Debug
• Timing Simulation
• Function Timing / Profiling
Characterize New Instructions in Application
• Instruction Coverage
• Line Coverage
• Instruction Performance
• Generate PDF model doc
Optimize & Document model

3 Example Description

This section introduces the example that is found in an OVP or Imperas product installation at

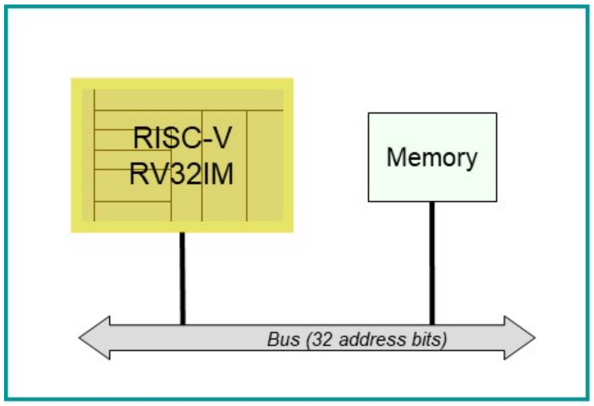
IMPERAS_HOME/Examples/Models/Processor/FeatureUsage/RISCV_CustomInstructionFlow

The example is also shown in a video that can be found on the OVP World website from the *Demos & Videos* link and titled *RISC-V Custom Instruction Design and Verification Flow* under the section '*Advanced Topic Videos*'.

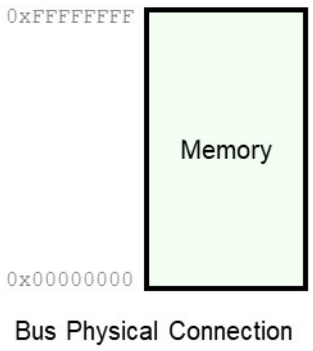
The example shows the addition of custom instructions into the RISC-V processor *custom1* decode space.

3.1 Virtual Platform

The virtual platform is created using the ISS[1], which instances a RISCV-V processor configured as an RV32I, with M extension and memory which provides the hardware definition shown in the following diagram.



Virtual Platform Block Diagram



Virtual Platform Address Mapping

3.2 Application Software

The application is based upon the chacha20 encryption algorithm; it takes data from an initialized array and preforms encryption upon it.

In the example there are two implementations, the first provides a C implementation of the algorithm and the second modifies this to use in line assembly to add custom instructions to

[1] The ISS is a virtual platform definition supporting the standard simulator command line argument and also additional specific command line arguments to define the type and number of processors to be instanced and any memory regions shared between the processors.

optimize the applications implementation. The application reads test data from the array and a function *processWord* is called in both implementations to process the data word.

Initial C implementation inner loop, *test_lib_c.c*

```
#define NOINLINE  __attribute__((noinline))

static int qrN_c(unsigned int a, unsigned int b, unsigned int rotl) {
    return ((a ^ b) << rotl) | ((a ^ b) >> (32-rotl));
}

NOINLINE int qr1_c(unsigned int a, unsigned int b) {
    return qrN_c(a, b, 16);
}

NOINLINE int qr2_c(unsigned int a, unsigned int b) {
    return qrN_c(a, b, 12);
}

NOINLINE int qr3_c(unsigned int a, unsigned int b) {
    return qrN_c(a, b, 8);
}

NOINLINE int qr4_c(unsigned int a, unsigned int b) {
    return qrN_c(a, b, 7);
}

NOINLINE unsigned int processWord(unsigned int res, unsigned int word){
    res = qr1_c(res, word);
    res = qr2_c(res, word);
    res = qr3_c(res, word);
    res = qr4_c(res, word);
    res = qr1_c(res, word);
    res = qr2_c(res, word);
    res = qr3_c(res, word);
    res = qr4_c(res, word);
    return res;
}
```

Modified implementation inner loop using in line assembly to add *custom1* instructions, *test_lib_asm.c*

```
unsigned int processWord(unsigned int input, unsigned int word){
    unsigned int res = input;
    asm __volatile__("mv x10, %0" :: "r"(res));
    asm __volatile__("mv x11, %0" :: "r"(word));
    asm __volatile__(".word 0x00B5050B\n" ::: "x10");     // QR1
    asm __volatile__(".word 0x00B5150B\n" ::: "x10");     // QR2
    asm __volatile__(".word 0x00B5250B\n" ::: "x10");     // QR3
    asm __volatile__(".word 0x00B5350B\n" ::: "x10");     // QR4
    asm __volatile__(".word 0x00B5050B\n" ::: "x10");     // QR1
    asm __volatile__(".word 0x00B5150B\n" ::: "x10");     // QR2
    asm __volatile__(".word 0x00B5250B\n" ::: "x10");     // QR3
    asm __volatile__(".word 0x00B5350B\n" ::: "x10");     // QR4
    asm __volatile__("mv %0,x10" : "=r"(res));
    return res;
}
```

3.3 Execution Control

The execution of this example is controlled from a single script RUN_STAGES.sh that can be executed on a Windows host in an MSYS shell or on a Linux host in a terminal.

When executed the script provides a choice of the run stage to execute, these are listed below:

Characterization of C Application
1: Instruction Accurate simulation
2: Cycle Approximate simulation
3: Basic Block Profile
4: Function Profile

Develop New Custom Instruction
5: Add custom instructions to application
6: Add custom instructions to model
7: Cycle Approximate simulation

Characterize New instructions in Application
8: Function Profile
9: Trace custom instructions
10: Debug custom instructions

Optimize New instruction Implementation, Analyze Test Coverage and Document
15: Documenting custom instructions (PDF - Linux Only)
16: Documenting custom instructions (TEX)
20: Custom Instruction Coverage
21: Custom Instruction Called C Function Profile
22: Custom Instruction Implementation Profile
25: Model Source Line Coverage

99: exit (without running)

The selected option controls the construction of the command line, for the ISS, which is written into a script file *lastRun.sh* and executed. This script can be used to re-run the last simulation.

4 Analyzing an Application Program

An algorithm implemented as C code can be tested using the Imperas/OVP ISS executable. This allows a virtual platform containing a processor and memory to be easily realized. It also provides the *semihosting* of host features i.e. access to the host file system, stdin, stdout etc. which greatly simplifies testing.

Initially we can run the application using Instruction Accurate simulation to prove the functional correctness.

```
$ ./RUN_STAGES.sh
… snip …
Please Select an Option: 1
Paused .. press key start simulation

IMPERAS Instruction Set Simulator (ISS)
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 0x00017338 0x00017338 R-E   1000
Info (OR_PD) LOAD           0x00017338 0x00028338 0x00028338 0x000009c0 0x00000a40 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 = 6E8D0F5A
Info 
Info ---------------------------------------------------
Info CPU 'iss/cpu0' STATISTICS
Info   Type                  : riscv (RV32I+M)
Info   Nominal MIPS          : 100
Info   Final program counter : 0x100ac
Info   Simulated instructions: 316,708,829
Info   Simulated MIPS        : run too short for meaningful result
Info ---------------------------------------------------
Info 
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 3.17 seconds
Info   User time             : 0.30 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 0.30 seconds
Info   Real time ratio       : 10.44x faster
Info ---------------------------------------------------
Use script lastRun.sh to re-run with current settings.
```

4.1 Enabling Cycle Approximate Simulation

Where a timing library has been created for the specific processor configuration and representative implementation technology, we can enable cycle approximate simulation to give more accurate execution times for the application execution.

Cycle approximate 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.

```
$ ./RUN_STAGES.sh
… snip …
Please Select an Option: 2
```

```
IMPERAS Instruction Set Simulator (ISS)
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 0x00017338 0x00017338 R-E   1000
Info (OR_PD) LOAD           0x00017338 0x00028338 0x00028338 0x000009c0 0x00000a40 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 (CMD_CC) calling 'iss/cpu0/exTT/cpucycles'
Info (CPUEST_CMD) exTT: cpucycles on: time stretch enabled   Cycle Approximate simulation enabled
Info (CMD_CC) calling 'iss/cpu0/exTT/memorycycles'
Info (CPUEST_MEM) exTT: memorycycles: Memory access cycle penalties for address range
0x28000:0x28FFF are as follows:
Info (CPUEST_MEM) exTT: memorycycles:   load  : 2
Info (CPUEST_MEM) exTT: memorycycles:   store : 1      Memory delays added
Info (CPUEST_MEM) exTT: memorycycles:   fetch : 0
RES = 6E8D0F5A
Info (CPUEST_RSLT) Estimated execution time 5.15 seconds, clock cycles 515,364,836
Info
Info ---------------------------------------------------        Cycle Approximation Summary
Info CPU 'iss/cpu0' STATISTICS
Info   Type                  : riscv (RV32I+M)
Info   Nominal MIPS          : 100
Info   Final program counter : 0x100ac
Info   Simulated instructions: 316,708,829
Info   Simulated MIPS        : 107.4
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 5.15 seconds
Info   User time             : 2.95 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 2.98 seconds
Info   Real time ratio       : 1.73x faster
Info ---------------------------------------------------
Use script lastRun.sh to re-run with current settings
```

4.2 Basic Block Profiling Application Execution

A basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit. The basic block profile tool can be used to find the most heavily executed instruction sequences in an application. This information can then be used to guide the implementation of custom instructions, compiler optimizations or other purposes.

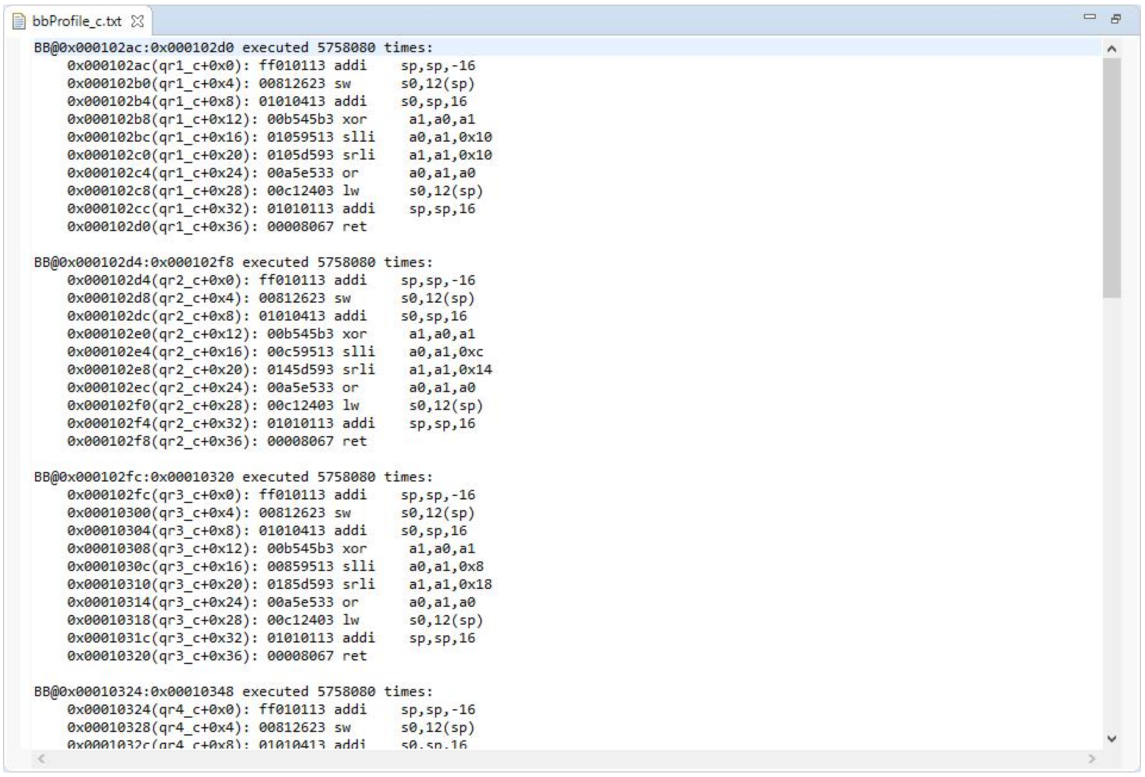
```
$ ./RUN_STAGES.sh
… snip …
Please Select an Option: 3
IMPERAS Instruction Set Simulator (ISS)
Info (OR_OF) Target 'iss/cpu0' has object file read from 'application/test_c.RISCV32.elf'
Info (OR_PH) Program Headers:
Info (OR_PH) Type           Offset     VirtAddr   PhysAddr   FileSiz    MemSiz     Flags Align
Info (OR_PD) LOAD           0x00000000 0x00010000 0x00010000 0x00017338 0x00017338 R-E   1000
Info (OR_PD) LOAD           0x00017338 0x00028338 0x00028338 0x000009c0 0x00000a40 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 (CMD_CC) calling 'iss/cpu0/exTT/cpucycles'
Info (CPUEST_CMD) exTT: cpucycles on: time stretch enabled
Info (CMD_CC) calling 'iss/cpu0/exTT/memorycycles'
Info (CPUEST_MEM) exTT: memorycycles: Memory access cycle penalties for address range
0x28000:0x28FFF are as follows:
```

```
Info (CPUEST_MEM) exTT: memorycycles:   load  : 2
Info (CPUEST_MEM) exTT: memorycycles:   store : 1
Info (CPUEST_MEM) exTT: memorycycles:   fetch : 0
RES = 6E8D0F5A
Info (BBP_WTD) iss/cpu0: Writing basic block profile data to file 'bbProfile_c.txt'.
Info (CPUEST_RSLT) Estimated execution time 5.15 seconds, clock cycles 515,364,836
Info
Info ---------------------------------------------------
Info CPU 'iss/cpu0' STATISTICS
Info   Type                  : riscv (RV32I+M)
Info   Nominal MIPS          : 100
Info   Final program counter : 0x100ac
Info   Simulated instructions: 316,708,829
Info   Simulated MIPS        : 31.0
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 5.15 seconds
Info   User time             : 10.22 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 10.23 seconds
Info ---------------------------------------------------
```

Basic Block Profile output file written

The generated profile can be viewed using any txt file viewer; the example uses the Imperas Eclipse GUI, eGui.

Once the file is opened (egui.exe -open bbProfile_c.txt) in eGui you can see the most used basic blocks are the core algorithm functions of *qr1_c*, *qr2_c*, *qr3_c* and *qr4_c*.



```
BB@0x000102ac:0x000102d0 executed 5758080 times:
    0x000102ac(qr1_c+0x0): ff010113 addi    sp,sp,-16
    0x000102b0(qr1_c+0x4): 00812623 sw      s0,12(sp)
    0x000102b4(qr1_c+0x8): 01010413 addi    s0,sp,16
    0x000102b8(qr1_c+0x12): 00b545b3 xor     a1,a0,a1
    0x000102bc(qr1_c+0x16): 01059513 slli    a0,a1,0x10
    0x000102c0(qr1_c+0x20): 0105d593 srli    a1,a1,0x10
    0x000102c4(qr1_c+0x24): 00a5e533 or      a0,a1,a0
    0x000102c8(qr1_c+0x28): 00c12403 lw      s0,12(sp)
    0x000102cc(qr1_c+0x32): 01010113 addi    sp,sp,16
    0x000102d0(qr1_c+0x36): 00008067 ret

BB@0x000102d4:0x000102f8 executed 5758080 times:
    0x000102d4(qr2_c+0x0): ff010113 addi    sp,sp,-16
    0x000102d8(qr2_c+0x4): 00812623 sw      s0,12(sp)
    0x000102dc(qr2_c+0x8): 01010413 addi    s0,sp,16
    0x000102e0(qr2_c+0x12): 00b545b3 xor     a1,a0,a1
    0x000102e4(qr2_c+0x16): 00c59513 slli    a0,a1,0xc
    0x000102e8(qr2_c+0x20): 0145d593 srli    a1,a1,0x14
    0x000102ec(qr2_c+0x24): 00a5e533 or      a0,a1,a0
    0x000102f0(qr2_c+0x28): 00c12403 lw      s0,12(sp)
    0x000102f4(qr2_c+0x32): 01010113 addi    sp,sp,16
    0x000102f8(qr2_c+0x36): 00008067 ret

BB@0x000102fc:0x00010320 executed 5758080 times:
    0x000102fc(qr3_c+0x0): ff010113 addi    sp,sp,-16
    0x00010300(qr3_c+0x4): 00812623 sw      s0,12(sp)
    0x00010304(qr3_c+0x8): 01010413 addi    s0,sp,16
    0x00010308(qr3_c+0x12): 00b545b3 xor     a1,a0,a1
    0x0001030c(qr3_c+0x16): 00859513 slli    a0,a1,0x8
    0x00010310(qr3_c+0x20): 0185d593 srli    a1,a1,0x18
    0x00010314(qr3_c+0x24): 00a5e533 or      a0,a1,a0
    0x00010318(qr3_c+0x28): 00c12403 lw      s0,12(sp)
    0x0001031c(qr3_c+0x32): 01010113 addi    sp,sp,16
    0x00010320(qr3_c+0x36): 00008067 ret

BB@0x00010324:0x00010348 executed 5758080 times:
    0x00010324(qr4_c+0x0): ff010113 addi    sp,sp,-16
    0x00010328(qr4_c+0x4): 00812623 sw      s0,12(sp)
    0x0001032c(qr4_c+0x8): 01010413 addi    s0,sp,16
```

4.3 Profiling Application Execution

The Imperas Verification, Analysis and Profiling (VAP) tools are provided as part of the Imperas professional product and include the ability to analyze the dynamic function profile of an application executing a data set without any changes to the application.

See the *VAP Tools User Guide* for additional information.

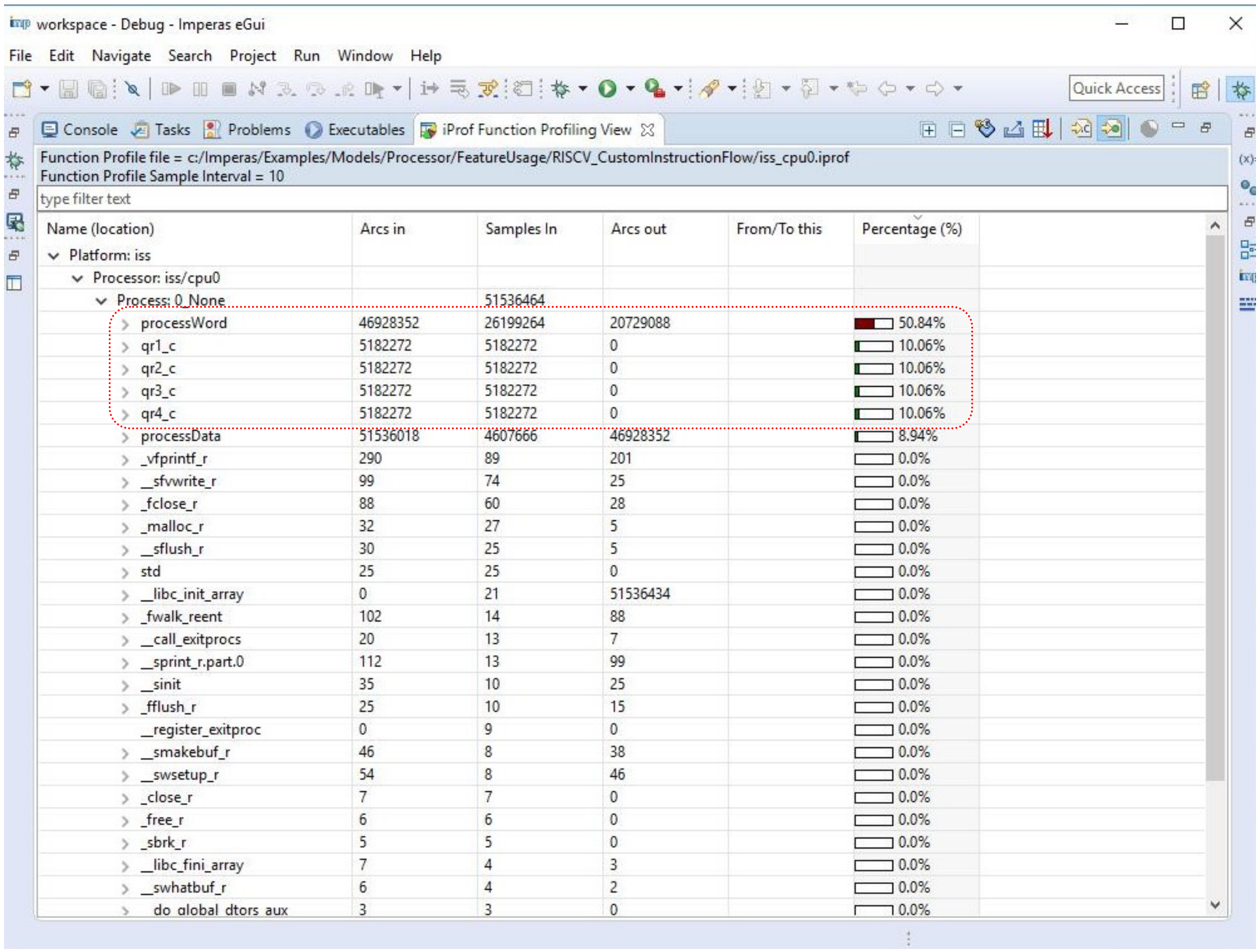
This profile tool used in conjunction with cycle approximate simulation can be used to find the 'hot spots' in an application. This information can then be used to guide the implementation of custom instructions. This data can also be used to analyze the benefits of custom instructions by comparing before and after profiles.

```
$ ./RUN_STAGES.sh
… snip …

Please Select an Option: 4
IMPERAS Instruction Set Simulator (ISS)


CpuManagerMulti (64-Bit) v20190225.0 Open Virtual Platform simulator from www.IMPERAS.com.
Copyright (c) 2005-2019 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 Feb 26 15:11:52 2019


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 0x00017338 0x00017338 R-E   1000
Info (OR_PD) LOAD           0x00017338 0x00028338 0x00028338 0x000009c0 0x00000a40 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 (CMD_CC) calling 'iss/cpu0/exTT/cpucycles'
Info (CPUEST_CMD) exTT: cpucycles on: time stretch enabled   Function Profiling enabled
Info (CMD_CC) calling 'iss/cpu0/exTT/memorycycles'
Info (CPUEST_MEM) exTT: memorycycles: Memory access cycle penalties for address range
0x28000:0x28FFF are as follows:
Info (CPUEST_MEM) exTT: memorycycles:   load  : 2
Info (CPUEST_MEM) exTT: memorycycles:   store : 1
Info (CPUEST_MEM) exTT: memorycycles:   fetch : 0
Info (CMD_CC) calling 'iss/cpu0/vapTools/functionprofile'          Profiling output file written
RES = 6E8D0F5A
Info (VAP_TOOLS) iss/cpu0: functionprofile: Writing data file 'iss_cpu0.iprof'
Info (CPUEST_RSLT) Estimated execution time 5.15 seconds, clock cycles 515,364,836
Info
Info ---------------------------------------------------
Info CPU 'iss/cpu0' STATISTICS
Info   Type                  : riscv (RV32I+M)
Info   Nominal MIPS          : 100
Info   Final program counter : 0x100ac
Info   Simulated instructions: 316,708,829
Info   Simulated MIPS        : 23.0
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 5.15 seconds
Info   User time             : 13.78 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 3114.09 seconds
Info ---------------------------------------------------

CpuManagerMulti finished: Tue Feb 26 16:03:46 2019


CpuManagerMulti (64-Bit) v20190225.0 Open Virtual Platform simulator from www.IMPERAS.com.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

Use script lastRun.sh to re-run with current settings
```

The generated profile can be viewed using the Imperas Eclipse GUI, eGui.

Once the file is opened (egui.exe -open iss_cpu0.iprof) in eGui you may be required to expand the tab (typically found in the bottom right corner with title seen as ‘iProf Fu’) and then sort for percentage to show the view below.

The profile information shows that we are spending most of the execution time within the *qrN_c* function that implements the core of the algorithm with almost as much again in the combination of the *qr1_c*, *qr2_c*, *qr3_c* and *qr4_c* functions.


workspace - Debug - Imperas eGui
File Edit Navigate Search Project Run Window Help
Quick Access
Console Tasks Problems Executables iProf Function Profiling View
type filter text

Function Profile file = c:/Imperas/Examples/Models/Processor/FeatureUsage/RISCV_CustomInstructionFlow/iss_cpu0.iprof
Function Profile Sample Interval = 10

Name (location)	Arcs in	Samples In	Arcs out	From/To this	Percentage (%)
Platform: iss					
Processor: iss/cpu0					
Process: 0_None		51536464			
processWord	46928352	26199264	20729088		50.84%
qr1_c	5182272	5182272	0		10.06%
qr2_c	5182272	5182272	0		10.06%
qr3_c	5182272	5182272	0		10.06%
qr4_c	5182272	5182272	0		10.06%
processData	51536018	4607666	46928352		8.94%
_vfprintf_r	290	89	201		0.0%
__sfvwrite_r	99	74	25		0.0%
_fclose_r	88	60	28		0.0%
_malloc_r	32	27	5		0.0%
__sflush_r	30	25	5		0.0%
std	25	25	0		0.0%
__libc_init_array	0	21	51536434		0.0%
_fwalk_reent	102	14	88		0.0%
__call_exitprocs	20	13	7		0.0%
__sprint_r.part.0	112	13	99		0.0%
__sinit	35	10	25		0.0%
_fflush_r	25	10	15		0.0%
__register_exitproc	0	9	0		0.0%
__smakebuf_r	46	8	38		0.0%
__swsetup_r	54	8	46		0.0%
_close_r	7	7	0		0.0%
_free_r	6	6	0		0.0%
_sbrk_r	5	5	0		0.0%
__libc_fini_array	7	4	3		0.0%
__swhatbuf_r	6	4	2		0.0%
__do_global_dtors_aux	3	3	0		0.0%

5 Creating Custom Instructions

5.1 Defining a custom instruction

The processor model behavior and custom instructions are both developed in the same way using the VMI APIs.
Detailed information on how to create instructions using the OVP APIs, including worked examples, is provided in the OVP Processor Modeling Guide with reference to the OVP VMI Morph Time API User Guide. This section will provide a guide to the key aspects of the custom instruction definition used in this example.

5.1.1 Instruction Decodes

A key element for a custom instruction is the decode. This includes the fixed fields defining the instruction class and the fields defining the source and result registers to be used.

This instruction extension library will be used to develop 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 the following table

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

This is used to create an attribute table in which the fixed bits are given their defined value and variable decode bits are indicated.

```
//
// This specifies attributes for each 32-bit opcode
//
const static riscvExtInstrAttrs attrsArray32[] = {
    //                                                                                      |
dec | rs2 | rs1 |fn3| rd  | dec   |
    EXT_INSTRUCTION(EXT_IT_CHACHA20QR1, "chacha20qr1", RVANY, RVIP_RD_RS1_RS2, FMT_R1_R2_R3,
"|0000000|.....|.....|000|.....|0001011|"),
    EXT_INSTRUCTION(EXT_IT_CHACHA20QR2, "chacha20qr2", RVANY, RVIP_RD_RS1_RS2, FMT_R1_R2_R3,
"|0000000|.....|.....|001|.....|0001011|"),
    EXT_INSTRUCTION(EXT_IT_CHACHA20QR3, "chacha20qr3", RVANY, RVIP_RD_RS1_RS2, FMT_R1_R2_R3,
"|0000000|.....|.....|010|.....|0001011|"),
    EXT_INSTRUCTION(EXT_IT_CHACHA20QR4, "chacha20qr4", RVANY, RVIP_RD_RS1_RS2, FMT_R1_R2_R3,
"|0000000|.....|.....|011|.....|0001011|")
};
```

The decode function uses the base model *fetchInstruction* callback to decode and instruction using the instruction attributes table.

```
//
// Decode instruction at the given address
//
static riscvExtIType decode(
    riscvP             riscv,
    vmiosObjectP       object,
    riscvAddr          thisPC,
    riscvExtInstrInfoP info
) {
    return riscv->cb.fetchInstruction(
        riscv, thisPC, info, &object->decode32, attrsArray32, EXT_IT_LAST, 32
    );
}
```

The decode function is then used in the VMIOS_MORPH_FN callback to decode the instruction read from the current PC

```
    // decode instruction
    riscvExtIType type = decode(riscv, object, thisPC, &state.info);
```

5.1.2 Instruction Behavior

The instruction behavior is created in the EXT_MORPH_FN callback. Once the instruction is decoded the correct behavior can be generated. This is accomplished using the VMI Morph Time API.

As we see in the code below the common function is used with a rotation value passed as one of the arguments.

```
//
// Dispatch table for instruction translation
//
const static riscvExtMorphAttr dispatchTable[] = {
    [EXT_IT_CHACHA20QR1] = {morph:emitCHACHA20QR, userData:(void *)16},
    [EXT_IT_CHACHA20QR2] = {morph:emitCHACHA20QR, userData:(void *)12},
    [EXT_IT_CHACHA20QR3] = {morph:emitCHACHA20QR, userData:(void *) 8},
    [EXT_IT_CHACHA20QR4] = {morph:emitCHACHA20QR, userData:(void *) 7},
};
```

The dispatch table is called using the externalMorph callback

```
        // fill translation attributes
        state.attrs = &dispatchTable[type];

        // translate instruction
        riscv->cb.morphExternal(&state, 0, opaque);
```

Setting the opaque pointer to True indicates that this behavior replaces any behavior from the underlying processor model.

The emitChaCha20 function is shown realized in two different implementations in the following sections.

5.1.2.1 Using a C algorithm function

In the initial application the algorithm was implemented as a C function.

```
static Uns32 qrN_c(Uns32 rs1, Uns32 rs2, Uns32 rotl) {
    return ((rs1 ^ rs2) << rotl) | ((rs1 ^ rs2) >> (32-rotl));
}
```

This can be used directly in the behavior for the instructions in the custom instruction extension by issuing a Morph Time function call.

```
//
// Emit code implementing CHACHA20QR instruction with rotation passed as
// userData in morph attributes structure
//
static EXT_MORPH_FN(emitCHACHA20QR) {

    riscvP riscv = state->riscv;

    // get abstract register operands
    riscvRegDesc rd  = getRVReg(state, 0);
    riscvRegDesc rs1 = getRVReg(state, 1);
    riscvRegDesc rs2 = getRVReg(state, 2);

    // get VMI registers for abstract operands
    vmiReg rdA  = getVMIReg(riscv, rd);
    vmiReg rs1A = getVMIReg(riscv, rs1);
    vmiReg rs2A = getVMIReg(riscv, rs2);

    // emit embedded call to perform operation
    UnsPS rotl = (UnsPS)state->attrs->userData;
    Uns32 bits = 32;
    vmimtArgReg(bits, rs1A);
    vmimtArgReg(bits, rs2A);
    vmimtArgUns32(rotl);
    vmimtCallResult((vmiCallFn)qrN_c, bits, rdA);

    // handle extension of result if 64-bit XLEN
    writeRegSize(riscv, rd, bits, True);
}
```

NOTE
This will have LOW simulation performance and is NOT the recommended approach when creating instruction behavior.
See the next section for implementation of instruction behavior.
This can be used to allow a staged development process.

5.1.2.2 Using VMI Morph Time Functions

The recommended approach to obtain the most efficient implementation is to use the VMI Morph Time functions directly to implement the behaviour.

As can be seen in the code for the implementation this function is the same as that shown previously except that instead of calling into the C function to implement the binary operation and rotation, we are now using the VMI MT binary operations to operate on the registers

```
//
// Emit code implementing CHACHA20QR instruction with rotation passed as
// userData in morph attributes structure
//
static EXT_MORPH_FN(emitCHACHA20QR) {

    riscvP riscv = state->riscv;

    // get abstract register operands
    riscvRegDesc rd  = getRVReg(state, 0);
    riscvRegDesc rs1 = getRVReg(state, 1);
    riscvRegDesc rs2 = getRVReg(state, 2);

    // get VMI registers for abstract operands
    vmiReg rdA  = getVMIReg(riscv, rd);
    vmiReg rs1A = getVMIReg(riscv, rs1);
    vmiReg rs2A = getVMIReg(riscv, rs2);

    // implement instruction operation
    UnsPS rotl = (UnsPS)state->attrs->userData;
    Uns32 bits = 32;
    vmimtBinopRRR(bits, vmi_XOR, rdA, rs1A, rs2A, 0);
    vmimtBinopRC(bits, vmi_ROL, rdA, rotl, 0);

    // handle extension of result if 64-bit XLEN
    writeRegSize(riscv, rd, bits, True);
}
```

5.2 Including custom instructions into C applications

The modified C application that is using in line assembly to execute the new instructions can now be executed on the virtual platform simulation that includes the RISC-V processor model and the custom instruction extension

If the modified application is executed without the custom instructions included in the processor.

```
$ ./RUN_STAGES.sh
… snip …
Please Select an Option: 5
Paused .. press key start simulation

IMPERAS Instruction Set Simulator (ISS)


CpuManagerMulti (64-Bit) v20190225.0 Open Virtual Platform simulator from www.IMPERAS.com.
Copyright (c) 2005-2019 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 Feb 26 16:05:19 2019


Info (OR_OF) Target 'iss/cpu0' has object file read from 'application/test_asm.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 0x00000a40 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
Warning (RISCV_ADF) CPU 'iss/cpu0' 0x000102c0 00b5050b custom1: Illegal instruction - extension X
(non-standard extensions present) absent or inactive
Info
Info ---------------------------------------------------
Info CPU 'iss/cpu0' STATISTICS
Info   Type                  : riscv (RV32I+M)
Info   Nominal MIPS          : 100
Info   Final program counter : 0x10238
Info   Simulated instructions: 213
Info   Simulated MIPS        : run too short for meaningful result
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 0.00 seconds
Info   User time             : 0.00 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 0.00 seconds
Info ---------------------------------------------------

CpuManagerMulti finished: Tue Feb 26 16:05:19 2019


CpuManagerMulti (64-Bit) v20190225.0 Open Virtual Platform simulator from www.IMPERAS.com.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

Use script lastRun.sh to re-run with current settings
```

Illegal instruction detected

But when we include the custom instructions into the processor model.

```
$ ./RUN_STAGES.sh
… snip …
Please Select an Option: 6
Paused .. press key start simulation

IMPERAS Instruction Set Simulator (ISS)


CpuManagerMulti (64-Bit) v20190225.0 Open Virtual Platform simulator from www.IMPERAS.com.
Copyright (c) 2005-2019 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 Feb 26 15:09:44 2019


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_asm.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 0x00000a40 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/pk
C:\Imperas\lib\Windows64\ImperasLib\riscv.ovpworld.org\semihosting\pk\1.0\model
Info (OP_PEX) Extension  iss/cpu0/exInst                          instructionExtensionLib
RES = 6E8D0F5A
Info
Info ---------------------------------------------------
Info CPU 'iss/cpu0' STATISTICS
```

Functionality verified

Custom instruction extension library loaded

```
Info   Type                  : riscv (RV32I+M)
Info   Nominal MIPS          : 100
Info   Final program counter : 0x1ff48       Less instructions executed
Info   Simulated instructions: 60,474,426
Info   Simulated MIPS        : run too short for meaningful result
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 0.60 seconds   Lower simulation time
Info   User time             : 0.03 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 0.03 seconds
Info   Real time ratio       : 18.91x faster
Info ---------------------------------------------------

CpuManagerMulti finished: Tue Feb 26 15:09:44 2019


CpuManagerMulti (64-Bit) v20190225.0 Open Virtual Platform simulator from www.IMPERAS.com.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

Use script lastRun.sh to re-run with current settings
```

6 Adding Custom Instructions to a Timing Library

A timing library is used to analyze the instruction execution stream and memory accesses performed by an application and then to use this information to determine additional cycle delays which would be caused on the actual hardware.
The timing library is created as an extension library using the OVP VMI APIs.

The timing library may be used to provide an estimation of the overall application execution or it may feedback the timing information to the simulator which will modify the execution to incorporate the reported instruction cycle delays.

6.1 Timing Library Extension

This example includes a simplified timing library that incorporates only instruction delays. The delays can be specified in a file as a pure delay or in the timing library as a delay calculated from previous instructions executed, mode of operation, data values etc.

6.1.1 Specify Delays in Timing Library Extension

A specific file *timingToolLib/timingTool.c* contains the information for the RISC-V processor timing. The timing information contained in this file was extracted from the specification documents available for the base processor configuration.

The timing information is grouped into classes of instructions with the same delays. A new class group can be added for the custom instructions that defines the four instruction mnemonics.

```
static instructionTableT instructionClasses[] = {
        // NOP
        { IC_default,        "nop"              },
… snip …
        /*Custom */
        { IC_custom,         "chacha20qr1"      },
        { IC_custom,         "chacha20qr2"      },
        { IC_custom,         "chacha20qr3"      },
        { IC_custom,         "chacha20qr4"      },
        // Default
        { IC_default,         NULL          }
};
```

The VMIOS_MORPH_FN callback is used to obtain the attributes for the instruction and use this to determine the delays, if any, that should be added to the execution.

```
VMIOS_MORPH_FN(rv32CEMorphCB) {

… snip …
    // get instruction attributes
    octiaAttrP attrs = vmiiaGetAttrs(processor, thisPC, SELECT_ATTRS, True);

… snip …
    instrClassesE iClass = getInstructionClass(object, thisPC, attrs);


    switch (iClass) {
… snip …
        case IC_custom : {
```

```
                                    // chacha20qr1-chacha20qr4 group same cycles
            cycles = 2;             // Specify cycles for instruction group
            break;
        }
        default: {
            VMI_ABORT("Invalid instructionCLassE value %d (%s)\n", iClass,
instrClassName(iClass));
            break;
        }

    }

    if (runtimeCB) { // division run-time callback
emitCycleEstimation(processor,object,thisPC,regSource1,regSource2,mduMode,iClass,runtimeCB);
    } else {
        addCycleCount(object, thisPC, cycles, iClass);    Function to update the cycle count
    }

    // Update previous morph time instruction info
    CEData->prevClassMT = iClass;
    CEData->regDestMT   = regDest;

    *opaque = False;
    return NULL;

}
```

This simple timing extension only considers the static delays for the type of instructions being executed.

The timing library uses a common set of *ce* functions that are defined in a standard Imperas library.

> *ceGetProcInfo*
> *ceAddCycles*
> *ceGetDiagnosticLevel*
> *ceGetThisInstructionClass*
> *ceGetThisInstructionData*
> *ceEmitAddCycleCountC*

The timing can be enabled or disabled and appropriate functions are called to allow updates when this is performed. The interface functions are registered in structure as shown below

```
static ceProcInfo infoRISCV = {
    .constructorCB = riscvConstructor,
    .destructorCB  = riscvDestructor,
    .docCB         = riscvDoc,
    .enableCB      = riscvEnable,
    .morphCB       = riscvMorph,
    .disableCB     = riscvDisable,
};
```

6.1.2 Adding Instruction Additional Cycles

The timing library can be modified to include instruction delays but the core timing functionality also allows for simple instruction delays to be added when there is no dependency on other instructions or on the data value. In this case we use the command *instructiondata* to specify a file to load.

The file simply contains the assembler mnemonic for the instruction and the additional delay, as shown in the following output.

```
additional delays
# arbitrary choice for delays
chacha20qr1  2
chacha20qr2  2
chacha20qr3  4
chacha20qr4  4
```

When executed these additional cycle delays are used during the simulation to extend the simulation time.

6.2 Example

We can enable cycle approximate simulation to give more accurate execution times for the application execution using the custom instructions.

```
$ ./RUN_STAGES.sh
… snip …
Please Select an Option: 7
Paused .. press key start simulation

IMPERAS Instruction Set Simulator (ISS)


CpuManagerMulti (64-Bit) v20190225.0 Open Virtual Platform simulator from www.IMPERAS.com.
Copyright (c) 2005-2019 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 Feb 26 14:34:55 2019


Info (OR_OF) Target 'iss/cpu0' has object file read from 'application/test_asm.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 0x00000a40 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 (CMD_CC) calling 'iss/cpu0/exTT/cpucycles'
Info (CPUEST_CMD) exTT: cpucycles on: time stretch enabled   Cycle Approximate simulation enabled
Info (CMD_CC) calling 'iss/cpu0/exTT/memorycycles'
Info (CPUEST_MEM) exTT: memorycycles: Memory access cycle penalties for address range
0x28000:0x28FFF are as follows:
Info (CPUEST_MEM) exTT: memorycycles:   load  : 2
Info (CPUEST_MEM) exTT: memorycycles:   store : 1
Info (CPUEST_MEM) exTT: memorycycles:   fetch : 0
Info (CMD_CC) calling 'iss/cpu0/exTT/instructiondata'
Info (CPUEST_DF1) exTT: instructiondata: Reading instruction data file
'custom_instruction_timing.txt'...
Info (CPUEST_DF3) exTT: instructiondata:     define   chacha20qr1 = 2
Info (CPUEST_DF3) exTT: instructiondata:     define   chacha20qr2 = 2   Instruction delays added
Info (CPUEST_DF3) exTT: instructiondata:     define   chacha20qr3 = 4
Info (CPUEST_DF3) exTT: instructiondata:     define   chacha20qr4 = 4
RES = 6E8D0F5A
Info (CPUEST_RSLT) Estimated execution time 1.38 seconds, clock cycles 138,210,542
Info                                                    Cycle Approximation Summary
Info --------------------------------------------------
Info CPU 'iss/cpu0' STATISTICS
Info   Type                  : riscv (RV32I+M)
Info   Nominal MIPS          : 100
Info   Final program counter : 0x100ac
Info   Simulated instructions: 60,474,242
Info   Simulated MIPS        : run too short for meaningful result
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 1.38 seconds
Info   User time             : 0.19 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 0.19 seconds
Info   Real time ratio       : 7.39x faster
Info ---------------------------------------------------

CpuManagerMulti finished: Tue Feb 26 14:34:55 2019


CpuManagerMulti (64-Bit) v20190225.0 Open Virtual Platform simulator from www.IMPERAS.com.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

Use script lastRun.sh to re-run with current settings
```

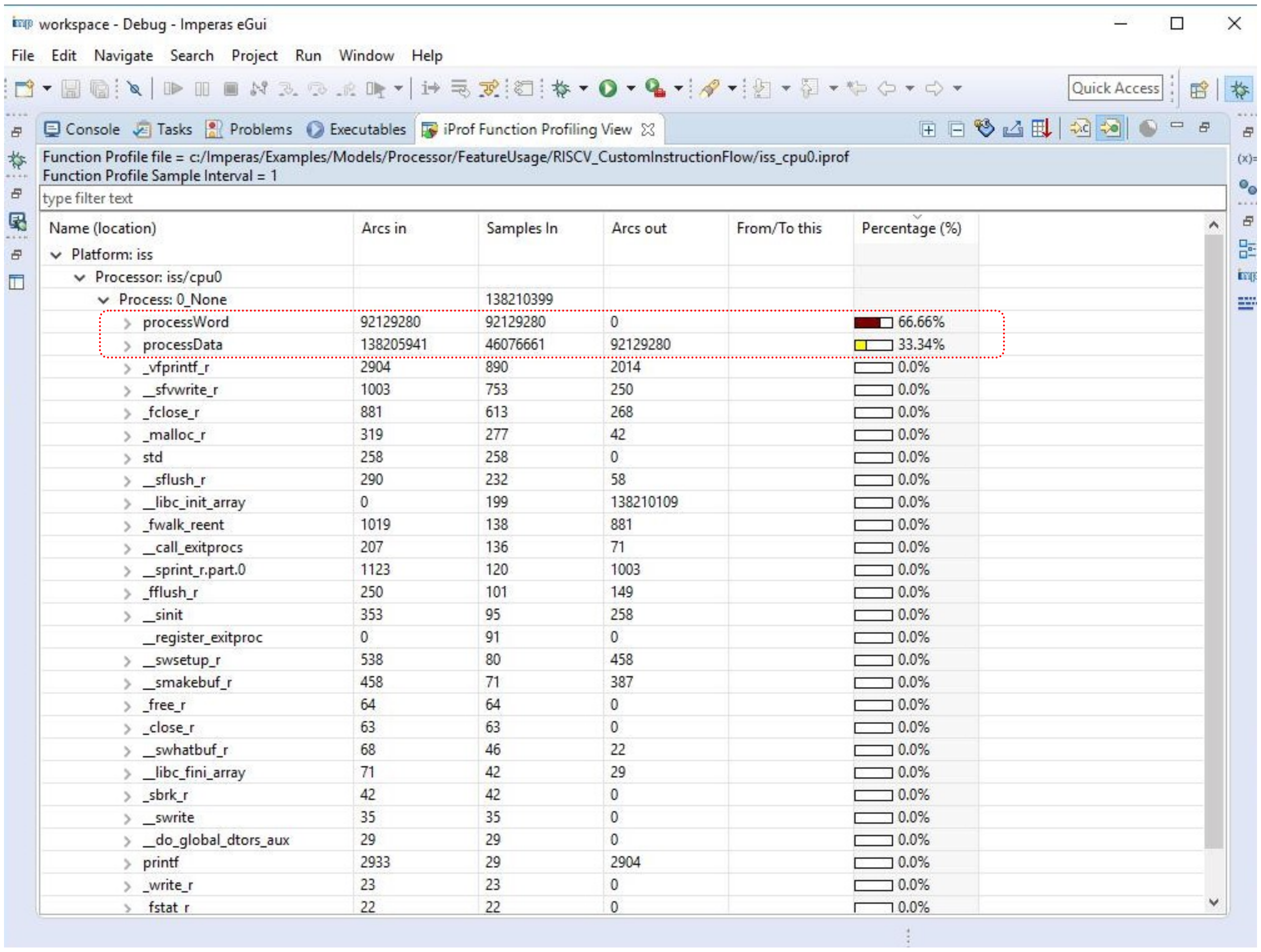
Execution time extended

7 Analyzing an Application program (that includes a custom instruction)

The same function profile tool executed on the C application in section 4.2 is used to obtain the function profile analysis when custom instructions are used.

```
$ ./RUN_STAGES.sh
… snip …
Please Select an Option: 8
IMPERAS Instruction Set Simulator (ISS)
… snip …
Info (CMD_CC) calling 'iss/cpu0/exTT/cpucycles'
Info (CPUEST_CMD) exTT: cpucycles on: time stretch enabled
Info (CMD_CC) calling 'iss/cpu0/exTT/memorycycles'
Info (CPUEST_MEM) exTT: memorycycles: Memory access cycle penalties for address range
0x28000:0x28FFF are as follows:
Info (CPUEST_MEM) exTT: memorycycles:   load  : 2
Info (CPUEST_MEM) exTT: memorycycles:   store : 1
Info (CPUEST_MEM) exTT: memorycycles:   fetch : 0
Info (CMD_CC) calling 'iss/cpu0/exTT/instructiondata'
Info (CPUEST_DF1) exTT: instructiondata: Reading instruction data file
'custom_instruction_timing.txt'...
Info (CPUEST_DF3) exTT: instructiondata:     define   chacha20qr1 = 2
Info (CPUEST_DF3) exTT: instructiondata:     define   chacha20qr2 = 2
Info (CPUEST_DF3) exTT: instructiondata:     define   chacha20qr3 = 4
Info (CPUEST_DF3) exTT: instructiondata:     define   chacha20qr4 = 4
Info (CMD_CC) calling 'iss/cpu0/vapTools/functionprofile'
RES = 6E8D0F5A
Info (VAP_TOOLS) iss/cpu0: functionprofile: Writing data file 'iss_cpu0.iprof'
Info (CPUEST_RSLT) Estimated execution time 1.38 seconds, clock cycles 138,210,542
Info
Info ---------------------------------------------------
Info CPU 'iss/cpu0' STATISTICS
Info   Type                 : riscv (RV32I+M)
Info   Nominal MIPS         : 100
Info   Final program counter : 0x100ac
Info   Simulated instructions: 60,474,242
Info   Simulated MIPS       : 33.7
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time       : 1.38 seconds
Info   User time            : 1.80 seconds
Info   System time          : 0.00 seconds
Info   Elapsed time         : 1.80 seconds
Info ---------------------------------------------------

CpuManagerMulti finished: Tue Feb 26 16:10:22 2019


CpuManagerMulti (64-Bit) v20190225.0 Open Virtual Platform simulator from www.IMPERAS.com.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

Use script lastRun.sh to re-run with current settings
```

In this execution run there is no longer the appearance of the C algorithm functions we saw previously as this behavior is now performed by the custom instructions.


workspace - Debug - Imperas eGui
File Edit Navigate Search Project Run Window Help
Quick Access
Console Tasks Problems Executables iProf Function Profiling View

Function Profile file = c:/Imperas/Examples/Models/Processor/FeatureUsage/RISCV_CustomInstructionFlow/iss_cpu0.iprof
Function Profile Sample Interval = 1

type filter text

Name (location)	Arcs in	Samples In	Arcs out	From/To this	Percentage (%)
Platform: iss					
Processor: iss/cpu0					
Process: 0_None		138210399			
processWord	92129280	92129280	0		66.66%
processData	138205941	46076661	92129280		33.34%
_vfprintf_r	2904	890	2014		0.0%
__sfvwrite_r	1003	753	250		0.0%
_fclose_r	881	613	268		0.0%
_malloc_r	319	277	42		0.0%
std	258	258	0		0.0%
__sflush_r	290	232	58		0.0%
__libc_init_array	0	199	138210109		0.0%
_fwalk_reent	1019	138	881		0.0%
__call_exitprocs	207	136	71		0.0%
__sprint_r.part.0	1123	120	1003		0.0%
_fflush_r	250	101	149		0.0%
__sinit	353	95	258		0.0%
__register_exitproc	0	91	0		0.0%
__swsetup_r	538	80	458		0.0%
__smakebuf_r	458	71	387		0.0%
_free_r	64	64	0		0.0%
_close_r	63	63	0		0.0%
__swhatbuf_r	68	46	22		0.0%
__libc_fini_array	71	42	29		0.0%
_sbrk_r	42	42	0		0.0%
__swrite	35	35	0		0.0%
__do_global_dtors_aux	29	29	0		0.0%
printf	2933	29	2904		0.0%
_write_r	23	23	0		0.0%
_fstat_r	22	22	0		0.0%

8 Tracing and Debug Support

As well as the custom instructions including the functional behavior they will also appear correctly when instructions are traced or when debugged at the instruction level.

The instruction tracing is defined in the extension library using the disassembly callback

8.1 Extension Library disassembly callback

The disassembly callback uses the same instruction decode table but generates a string that represents the current instruction.

```
//
// Disassembler callback disassembling ChaCha20 instructions
//
static VMIOS_DISASSEMBLE_FN(doDisass) {

    // decode the instruction to get the type
    Uns32                  instruction = vmicxtFetch4Byte(processor, thisPC);
    riscvEnhancedInstrType type        = vmidDecode(object->table, instruction);

    if (type != RISCV_EIT_LAST) {
        static char buffer[256];

        // extract instruction fields
        Uns32 rd  = RD(instruction);
        Uns32 rs1 = RS1(instruction);
        Uns32 rs2 = RS2(instruction);

        if(type==RISCV_EIT_CHACHA20QR1) {
            sprintf(buffer, "%-8s %s,%s,%s", "chacha20qr1", map[rd], map[rs1], map[rs2]);
            return buffer;

        } else if (type==RISCV_EIT_CHACHA20QR2) {
            sprintf(buffer, "%-8s %s,%s,%s", "chacha20qr2", map[rd], map[rs1], map[rs2]);
            return buffer;

        } else if (type==RISCV_EIT_CHACHA20QR3) {
            sprintf(buffer, "%-8s %s,%s,%s", "chacha20qr3", map[rd], map[rs1], map[rs2]);
            return buffer;

        } else if (type==RISCV_EIT_CHACHA20QR4) {
            sprintf(buffer, "%-8s %s,%s,%s", "chacha20qr4", map[rd], map[rs1], map[rs2]);
            return buffer;

        } else {
            VMI_ABORT("Invalid decode");
        }
    }

    // instruction not enhanced ChaCha20
    return 0;
}
```

8.2 Instruction Tracing

The extension instruction is implemented in the same manner as instructions in a processor model and so all the same services are provided that can be enabled on the simulator command line or interactively in the debug console.

This example uses the *traceafter* and *tracecount* argument to control tracing of instructions and modified registers over a specific period of execution that incorporates the custom instructions

```
$ ./RUN_STAGES.sh
… snip …
Please Select an Option: 9
IMPERAS Instruction Set Simulator (ISS)
… snip …
Info 1022: 'iss/cpu0', 0x000000000001023c(processData+4c): ff249ae3 bne     s1,s2,10230
Info 1023: 'iss/cpu0', 0x0000000000010230(processData+40): 0004a583 lw      a1,0(s1)
Info   a1 000062e8 -> 000013fe
Info 1024: 'iss/cpu0', 0x0000000000010234(processData+44): 078000ef jal     ra,102ac
Info 1025: 'iss/cpu0', 0x00000000000102ac(processWord): ff010113 addi    sp,sp,-16
Info   sp ffffffd0 -> ffffffc0
Info 1026: 'iss/cpu0', 0x00000000000102b0(processWord+4): 00812623 sw      s0,12(sp)
Info 1027: 'iss/cpu0', 0x00000000000102b4(processWord+8): 01010413 addi    s0,sp,16
Info   s0 fffffff0 -> ffffffd0
Info 1028: 'iss/cpu0', 0x00000000000102b8(processWord+c): 00050513 mv      a0,a0
Info 1029: 'iss/cpu0', 0x00000000000102bc(processWord+10): 00058593 mv      a1,a1
Info 1030: 'iss/cpu0', 0x00000000000102c0(processWord+14): chacha20qr1 a0,a0,a1
Info   a0 b79338c6 -> 2b38b793
Info 1031: 'iss/cpu0', 0x00000000000102c4(processWord+18): chacha20qr2 a0,a0,a1
Info   a0 2b38b793 -> 8a46d2b3
Info 1032: 'iss/cpu0', 0x00000000000102c8(processWord+1c): chacha20qr3 a0,a0,a1
Info   a0 8a46d2b3 -> 46c14d8a
Info 1033: 'iss/cpu0', 0x00000000000102cc(processWord+20): chacha20qr4 a0,a0,a1
Info   a0 46c14d8a -> 60af3a23
Info 1034: 'iss/cpu0', 0x00000000000102d0(processWord+24): chacha20qr1 a0,a0,a1
Info   a0 60af3a23 -> 29dd60af
Info 1035: 'iss/cpu0', 0x00000000000102d4(processWord+28): chacha20qr2 a0,a0,a1
Info   a0 29dd60af -> d735129d
Info 1036: 'iss/cpu0', 0x00000000000102d8(processWord+2c): chacha20qr3 a0,a0,a1
Info   a0 d735129d -> 350163d7
Info 1037: 'iss/cpu0', 0x00000000000102dc(processWord+30): chacha20qr4 a0,a0,a1
Info   a0 350163d7 -> 80b8149a
Info 1038: 'iss/cpu0', 0x00000000000102e0(processWord+34): 00050513 mv      a0,a0
Info 1039: 'iss/cpu0', 0x00000000000102e4(processWord+38): 00c12403 lw      s0,12(sp)
Info   s0 ffffffd0 -> fffffff0
Info 1040: 'iss/cpu0', 0x00000000000102e8(processWord+3c): 01010113 addi    sp,sp,16
Info   sp ffffffc0 -> ffffffd0
Info 1041: 'iss/cpu0', 0x00000000000102ec(processWord+40): 00008067 ret
Info 1042: 'iss/cpu0', 0x0000000000010238(processData+48): 00448493 addi    s1,s1,4
Info   s1 00025644 -> 00025648
Info 1043: 'iss/cpu0', 0x000000000001023c(processData+4c): ff249ae3 bne     s1,s2,10230
Info 1044: 'iss/cpu0', 0x0000000000010230(processData+40): 0004a583 lw      a1,0(s1)
Info   a1 000013fe -> 00003bbb
Info 1045: 'iss/cpu0', 0x0000000000010234(processData+44): 078000ef jal     ra,102ac
Info 1046: 'iss/cpu0', 0x00000000000102ac(processWord): ff010113 addi    sp,sp,-16
Info   sp ffffffd0 -> ffffffc0
Info 1047: 'iss/cpu0', 0x00000000000102b0(processWord+4): 00812623 sw      s0,12(sp)
Info 1048: 'iss/cpu0', 0x00000000000102b4(processWord+8): 01010413 addi    s0,sp,16
Info   s0 fffffff0 -> ffffffd0
Info 1049: 'iss/cpu0', 0x00000000000102b8(processWord+c): 00050513 mv      a0,a0
Info 1050: 'iss/cpu0', 0x00000000000102bc(processWord+10): 00058593 mv      a1,a1
Info 1051: 'iss/cpu0', 0x00000000000102c0(processWord+14): chacha20qr1 a0,a0,a1
Info   a0 80b8149a -> 2f2180b8
Info 1052: 'iss/cpu0', 0x00000000000102c4(processWord+18): chacha20qr2 a0,a0,a1
Info   a0 2f2180b8 -> 1bb032f2
Info 1053: 'iss/cpu0', 0x00000000000102c8(processWord+1c): chacha20qr3 a0,a0,a1
Info   a0 1bb032f2 -> b009491b
Info 1054: 'iss/cpu0', 0x00000000000102cc(processWord+20): chacha20qr4 a0,a0,a1
Info   a0 b009491b -> 04b95058
Info 1055: 'iss/cpu0', 0x00000000000102d0(processWord+24): chacha20qr1 a0,a0,a1
Info   a0 04b95058 -> 6be304b9
Info 1056: 'iss/cpu0', 0x00000000000102d4(processWord+28): chacha20qr2 a0,a0,a1
Info   a0 6be304b9 -> 33f026be
Info 1057: 'iss/cpu0', 0x00000000000102d8(processWord+2c): chacha20qr3 a0,a0,a1
Info   a0 33f026be -> f01d0533
Info 1058: 'iss/cpu0', 0x00000000000102dc(processWord+30): chacha20qr4 a0,a0,a1
Info   a0 f01d0533 -> 0e9f4478
Info 1059: 'iss/cpu0', 0x00000000000102e0(processWord+34): 00050513 mv      a0,a0
Info 1060: 'iss/cpu0', 0x00000000000102e4(processWord+38): 00c12403 lw      s0,12(sp)
Info   s0 ffffffd0 -> fffffff0
Info 1061: 'iss/cpu0', 0x00000000000102e8(processWord+3c): 01010113 addi    sp,sp,16
Info   sp ffffffc0 -> ffffffd0
Info 1062: 'iss/cpu0', 0x00000000000102ec(processWord+40): 00008067 ret
Info 1063: 'iss/cpu0', 0x0000000000010238(processData+48): 00448493 addi    s1,s1,4
Info   s1 00025648 -> 0002564c
Info 1064: 'iss/cpu0', 0x000000000001023c(processData+4c): ff249ae3 bne     s1,s2,10230
Info 1065: 'iss/cpu0', 0x0000000000010230(processData+40): 0004a583 lw      a1,0(s1)
Info   a1 00003bbb -> 000022a0
RES = 6E8D0F5A
```

As we can see, in the following snippet of the output, the custom instruction modifies register a0 that is identified by the simulator and included in the trace

```
Info 1051: 'iss/cpu0', 0x00000000000102c0(processWord+14): chacha20qr1 a0,a0,a1
Info   a0 80b8149a -> 2f2180b8
```

8.3 Debug

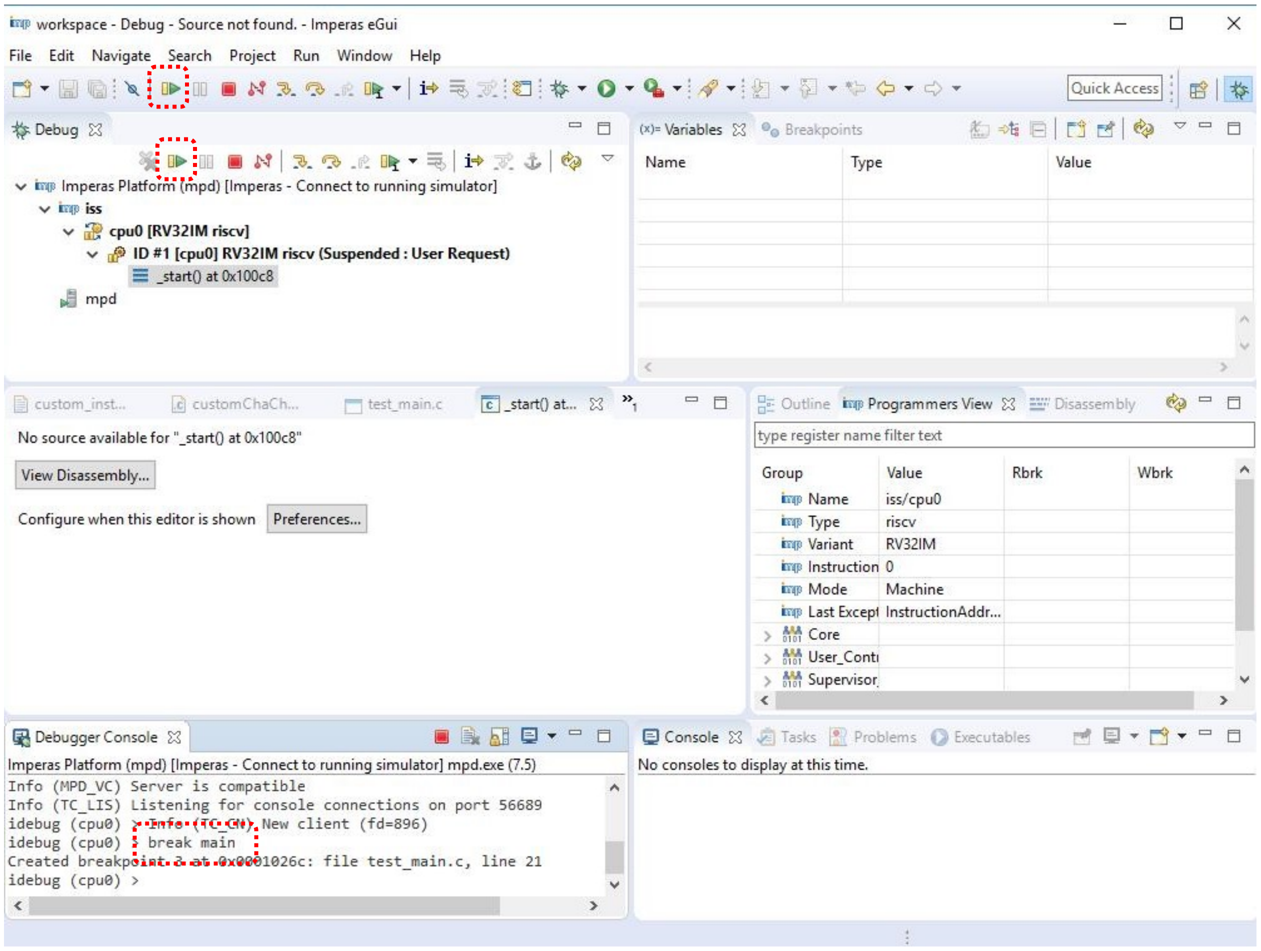
The application can be debugged using eGui Eclipse and the custom instructions can be seen correctly in the disassembly view.

Starting up the simulation using the *mpdegui* option will start the eGui and automatically connect the debug environment with the running simulator.

```
$ ./RUN_STAGES.sh
Please Select an Option: 10
IMPERAS Instruction Set Simulator (ISS)

… snip …

Info (GDBT_PORT) Host: GRAHAM-LAPTOP, Port: 63650
Info (DBC_ECL) Starting Eclipse with MPD
Info (GDBT_WAIT) Waiting for remote debugger to connect...
… snip …
Info (EGUI) egui port number = 62049
Info (GDBT_MPD) Client connected to platform
```

Now in eGui we can debug the application

Use the Debugger Console to set a breakpoint at main and run until this breakpoint


workspace - Debug - Source not found. - Imperas eGui

```
Imperas Platform (mpd) [Imperas - Connect to running simulator]
  iss
    cpu0 [RV32IM riscv]
      ID #1 [cpu0] RV32IM riscv (Suspended : User Request)
        _start() at 0x100c8
  mpd
```

No source available for "_start() at 0x100c8"

Group	Value	Rbrk	Wbrk
Name	iss/cpu0		
Type	riscv		
Variant	RV32IM		
Instruction	0		
Mode	Machine		
Last Except	InstructionAddr...		
Core			
User_Conti			
Supervisor			

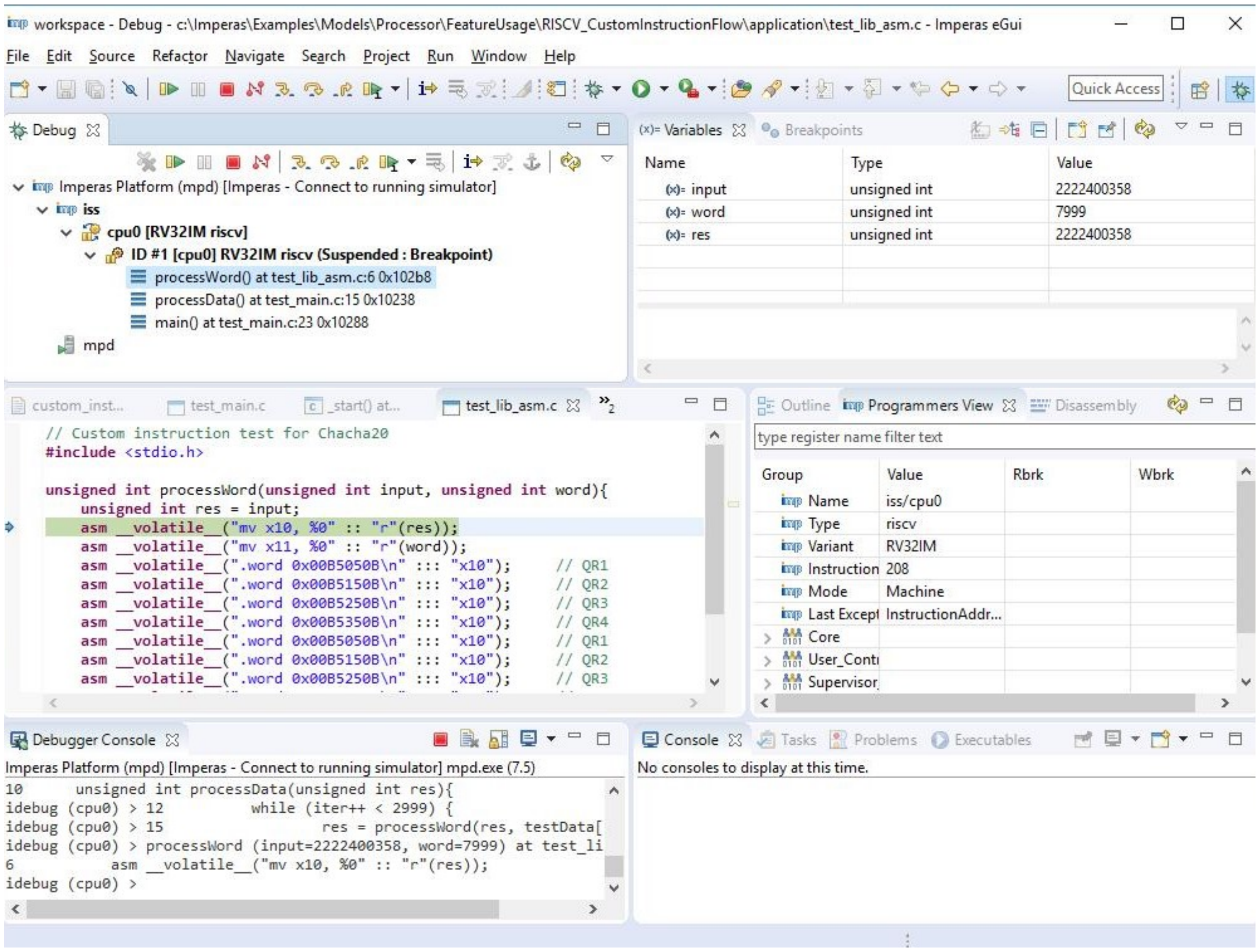
Imperas Platform (mpd) [Imperas - Connect to running simulator] mpd.exe (7.5)

```
Info (MPD_VC) Server is compatible
Info (TC_LIS) Listening for console connections on port 56689
idebug (cpu0) > Info (TC_CN) New client (fd=896)
idebug (cpu0) > break main
Created breakpoint 3 at 0x0001026c: file test_main.c, line 21
idebug (cpu0) >
```

No consoles to display at this time.

Now we want to get to the *processWord* function, step until you enter this function.

.



workspace - Debug - c:\Imperas\Examples\Models\Processor\FeatureUsage\RISCV_CustomInstructionFlow\application\test_lib_asm.c - Imperas eGui

```
Imperas Platform (mpd) [Imperas - Connect to running simulator]
  iss
    cpu0 [RV32IM riscv]
      ID #1 [cpu0] RV32IM riscv (Suspended : Breakpoint)
        processWord() at test_lib_asm.c:6 0x102b8
        processData() at test_main.c:15 0x10238
        main() at test_main.c:23 0x10288
  mpd
```

Name	Type	Value
input	unsigned int	2222400358
word	unsigned int	7999
res	unsigned int	2222400358

test_lib_asm.c

```
// Custom instruction test for Chacha20
#include <stdio.h>

unsigned int processWord(unsigned int input, unsigned int word){
    unsigned int res = input;
    asm __volatile__("mv x10, %0" :: "r"(res));
    asm __volatile__("mv x11, %0" :: "r"(word));
    asm __volatile__(".word 0x00B5050B\n" ::: "x10");     // QR1
    asm __volatile__(".word 0x00B5150B\n" ::: "x10");     // QR2
    asm __volatile__(".word 0x00B5250B\n" ::: "x10");     // QR3
    asm __volatile__(".word 0x00B5350B\n" ::: "x10");     // QR4
    asm __volatile__(".word 0x00B5050B\n" ::: "x10");     // QR1
    asm __volatile__(".word 0x00B5150B\n" ::: "x10");     // QR2
    asm __volatile__(".word 0x00B5250B\n" ::: "x10");     // QR3
```

Programmers View

Group	Value	Rbrk	Wbrk
Name	iss/cpu0		
Type	riscv		
Variant	RV32IM		
Instruction	208		
Mode	Machine		
Last Except	InstructionAddr...		
Core			
User_Conti			
Supervisor			

Debugger Console

Imperas Platform (mpd) [Imperas - Connect to running simulator] mpd.exe (7.5)

```
10      unsigned int processData(unsigned int res){
idebug (cpu0) > 12          while (iter++ < 2999) {
idebug (cpu0) > 15                  res = processWord(res, testData[
idebug (cpu0) > processWord (input=2222400358, word=7999) at test_li
6           asm __volatile__("mv x10, %0" :: "r"(res));
idebug (cpu0) >
```

Console

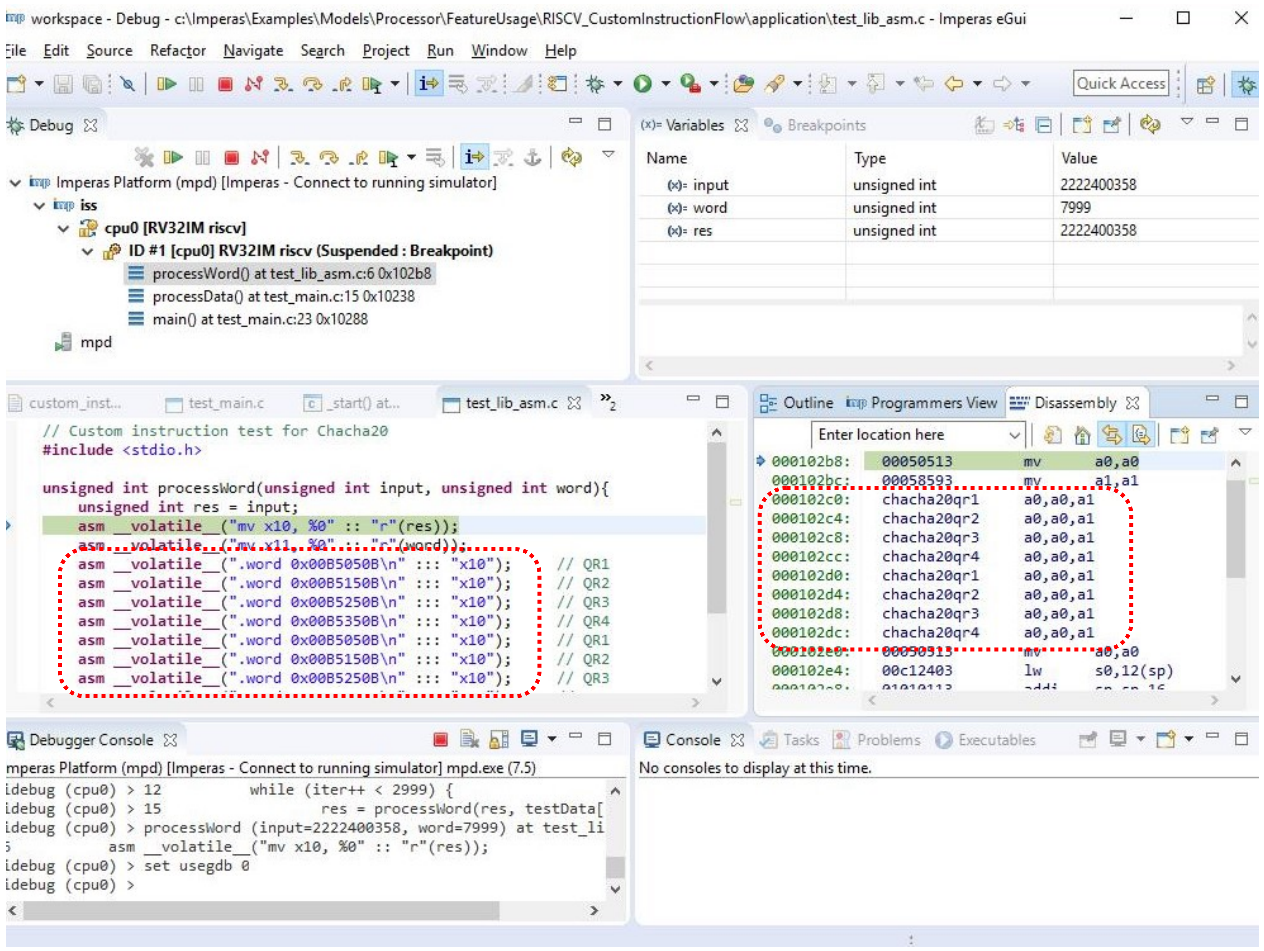
No consoles to display at this time.

When we step into this function, we can see the inline assembly that is used to include the custom instructions into the C application.
Before we open the disassembly window, we are going to turn off GDB mode so that the disassembly is generated by the custom instruction extension library.
In the Debugger Console set usegdb to zero.

```
set usegdb 0
```

If we now open the disassembly window, we can see the custom instructions. Before we step turn back into GDB mode
In the Debugger Console set usegdb to one.

```
set usegdb 1
```


workspace - Debug - c:\Imperas\Examples\Models\Processor\FeatureUsage\RISCV_CustomInstructionFlow\application\test_lib_asm.c - Imperas eGui

```
Imperas Platform (mpd) [Imperas - Connect to running simulator]
  iss
    cpu0 [RV32IM riscv]
      ID #1 [cpu0] RV32IM riscv (Suspended : Breakpoint)
        processWord() at test_lib_asm.c:6 0x102b8
        processData() at test_main.c:15 0x10238
        main() at test_main.c:23 0x10288
  mpd
```

Name	Type	Value
input	unsigned int	2222400358
word	unsigned int	7999
res	unsigned int	2222400358

test_lib_asm.c

```
// Custom instruction test for Chacha20
#include <stdio.h>

unsigned int processWord(unsigned int input, unsigned int word){
    unsigned int res = input;
    asm __volatile__("mv x10, %0" :: "r"(res));
    asm __volatile__("mv x11, %0" :: "r"(word));
    asm __volatile__(".word 0x00B5050B\n" ::: "x10");    // QR1
    asm __volatile__(".word 0x00B5150B\n" ::: "x10");    // QR2
    asm __volatile__(".word 0x00B5250B\n" ::: "x10");    // QR3
    asm __volatile__(".word 0x00B5350B\n" ::: "x10");    // QR4
    asm __volatile__(".word 0x00B5050B\n" ::: "x10");    // QR1
    asm __volatile__(".word 0x00B5150B\n" ::: "x10");    // QR2
    asm __volatile__(".word 0x00B5250B\n" ::: "x10");    // QR3
```

Disassembly

```
000102b8:   00050513       mv       a0,a0
000102bc:   00058593       mv       a1,a1
000102c0:   chacha20qr1    a0,a0,a1
000102c4:   chacha20qr2    a0,a0,a1
000102c8:   chacha20qr3    a0,a0,a1
000102cc:   chacha20qr4    a0,a0,a1
000102d0:   chacha20qr1    a0,a0,a1
000102d4:   chacha20qr2    a0,a0,a1
000102d8:   chacha20qr3    a0,a0,a1
000102dc:   chacha20qr4    a0,a0,a1
000102e0:   00050513       mv       a0,a0
000102e4:   00c12403       lw       s0,12(sp)
```

Debugger Console

```
mperas Platform (mpd) [Imperas - Connect to running simulator] mpd.exe (7.5)
idebug (cpu0) > 12          while (iter++ < 2999) {
idebug (cpu0) > 15                  res = processWord(res, testData[
idebug (cpu0) > processWord (input=2222400358, word=7999) at test_li
5           asm __volatile__("mv x10, %0" :: "r"(res));
idebug (cpu0) > set usegdb 0
idebug (cpu0) >
```

Console

No consoles to display at this time.

9 Documenting Custom Instructions

All OVP Fast Processor models include automatically generated documentation. All custom instructions can be fully documented as part of the processor model documentation by adding calls to VMI API documentation functions.

The documentation for these custom instructions is added to provide an overview and a section detailing each instruction

9.1 Definition in Extension Library

The documentation is defined by the VMIOS_DOC_FN callback. The function may add document sections and then text within those sections

```
//
// Add documentation for Custom instructions
//
VMIOS_DOC_FN(extensionDoc) {

    vmiDocNodeP custom = vmidocAddSection(0, "Instruction Extensions");

    // description
    vmidocAddText(
        custom,
        "RISCV processors may add various custom extensions to the basic "
        "RISC-V architecture. "
        "This processor has been extended, using an extension library, "
        "to add several instruction using the Custom0 opcode."
    );
    vmiDocNodeP insts = vmidocAddSection(
        custom, "Custom Instructions"
    );

    vmidocAddText(
        insts,
        "This model includes four Chacha20 acceleration instructions "
        "(one for each rotate distance) are added to encode the XOR "
        "and ROTATE parts of the quarter rounds.");

    docChaCha20(insts, "chacha20qr1", "000 (QR1)", "dst = (src1 ^ src2) <<< 16");
    docChaCha20(insts, "chacha20qr2", "001 (QR2)", "dst = (src1 ^ src2) <<< 12");
    docChaCha20(insts, "chacha20qr3", "010 (QR3)", "dst = (src1 ^ src2) <<<  8");
    docChaCha20(insts, "chacha20qr4", "011 (QR4)", "dst = (src1 ^ src2) <<<  7");

    vmidocProcessor(processor, custom);
}
```

New Chapter *custom*

Add text into this chapter

Add new section into chapter

Add text into this section

Add instruction details

The instructions themselves exist inside one of these sections and can be detailed by the specific fields.

```
//
// Add documentation for ChaCha20 instructions
//
static void docChaCha20(
    vmiDocNodeP insts,
    const char *opcode,
    const char *decode,
    const char *desc
) {
    vmiDocNodeP inst = vmidocAddFields(insts, opcode, 32);

    // fields
    vmidocAddField(inst, "Custom0 0001011",     0, 7);
    vmidocAddField(inst, "Rd",                  7, 5);
    vmidocAddField(inst, decode,               12, 3);
    vmidocAddField(inst, "Rs1",                15, 5);
    vmidocAddField(inst, "Rs2",                20, 5);
    vmidocAddField(inst, "0000000",            25, 7);

    // description
    vmidocAddText(inst, desc);
}
```

Add instruction fields

9.2 Generating Documentation

The documentation can be generated as a TeX file or on Linux the TeX file can be converted to a PDF file. For this example (on Linux) we can generated the PDF as shown below:

```
$ ./RUN_STAGES.sh
… snip …
Please Select an Option: 15
Generating PDF Document
mkdir -p pdf
texgen.exe \
        -modeltype processor -modelname riscv -modelvendor riscv.ovpworld.org -variant RV32I \
        -override add_Extensions=M -pdfdir pdf \
        -extlib instructionExtensionLib
```

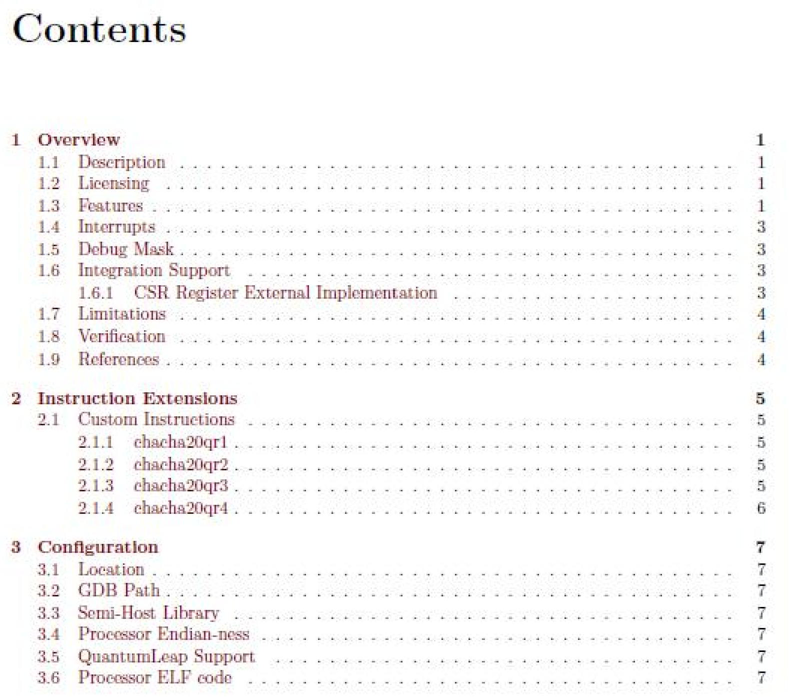
This generates the document *OVP_Model_Specific_Information_riscv_RV32I+M.pdf* in the pdf directory.

Example pages from the document are shown in the next section.

9.3 Example Documentation Pages

The API functions are used with a generation tool to create a pdf document that is based upon a base processor, in this case the RV32I WITH M EXTENSION , with the extension instructions included.

The content page shows the new chapter Instruction Extensions



Contents

The chapter contains the overview and the details for the instructions

Chapter 2

Instruction Extensions

RISCV processors may add various custom extensions to the basic RISC-V architecture. This processor has been extended, using an extension library, to add several instruction using the Custom0 opcode.

2.1 Custom Instructions

This model includes four Chacha20 acceleration instructions (one for each rotate distance) are added to encode the XOR and ROTATE parts of the quarter rounds.

2.1.1 chacha20qr1

31 25	24 20	19 15	14 12	11 7	6 0
0000000	Rs2	Rs1	000 (QR1)	Rd	Custom0 0001011

dst = (src1 ŝrc2) <<<16

2.1.2 chacha20qr2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	Rs2	Rs1	001 (QR2)	Rd	Custom0 0001011

dst = (src1 ŝrc2) <<<12

10 Analyze Custom Instruction Implementation and Test

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.

The following sections show the use of the tools available as standard tools included in the Imperas professional products.

10.1 Instruction Coverage

Instruction coverage can be used to indicate if any instructions that are defined are not being executed i.e. not being covered by the suite of tests being used to verify correct operation.

```
$ ./RUN_STAGES.sh
… snip …
Please Select an Option: 20
IMPERAS Instruction Set Simulator (ISS)
… snip …
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 = 6E8D0F5A
… snip …
Info (ICR_OF) cpu0: instruction profile report file 'coverageReports\cpu0.icr.txt'
# iss/cpu0: Instruction Profile Totals:
#         OpCode :    Pct : Count (Total instrs=        60474242)
# -------------- : ------ : --------
            addi : 19.05% :         11522537
              mv : 14.28% :          8637331
              lw :  9.52% :          5758452
     chacha20qr1 :  9.52% :          5758080
     chacha20qr2 :  9.52% :          5758080
     chacha20qr3 :  9.52% :          5758080
     chacha20qr4 :  9.52% :          5758080
              sw :  4.76% :          2879403
             ret :  4.76% :          2879120
             jal :  4.76% :          2879116
             bne :  4.76% :          2879068
            bnez :  0.01% :             3062
             add :  0.01% :             3059
            beqz :  0.00% :              113
            andi :  0.00% :               77
               j :  0.00% :               68
           auipc :  0.00% :               50
              sb :  0.00% :               50
… snip …
```

10.2 Instruction Profiling

Instruction profiling gives an indication of the amount of host system time spent executing the behavior of each instruction. This provides an indication of how well the instruction has been implemented.
As we saw earlier in section 5.1.2 Instruction Behavior we created two implementations for the behavior of the instruction. One was calling the C function implementation and the second using only the VMI Morph Time API. In this section we will show the instruction profiling results for both of these implementations.

First the C algorithm implementation of the instruction behavior

```
$ ./RUN_STAGES.sh
Please Select an Option: 21
IMPERAS Instruction Set Simulator (ISS)
… snip …
RES = 6E8D0F5A

-------------------------------------------------------------------
PROFILE REPORT
-------------------------------------------------------------------
    chacha20qr2          0.02s         5,758,080   3.5ps/instruction (iss/cpu0)
    chacha20qr4          0.02s         5,758,080   3.5ps/instruction (iss/cpu0)
    chacha20qr3          0.02s         5,758,080   3.5ps/instruction (iss/cpu0)
    addi                 0.01s        11,522,537   0.9ps/instruction (iss/cpu0)
    chacha20qr1          0.01s         5,758,080   1.7ps/instruction (iss/cpu0)
    mv                   0.00s         8,637,331   0.0ps/instruction (iss/cpu0)
    lw                   0.00s         5,758,452   0.0ps/instruction (iss/cpu0)
    sw                   0.00s         2,879,403   0.0ps/instruction (iss/cpu0)
    ret                  0.00s         2,879,120   0.0ps/instruction (iss/cpu0)
    jal                  0.00s         2,879,116   0.0ps/instruction (iss/cpu0)
    bne                  0.00s         2,879,068   0.0ps/instruction (iss/cpu0)
    bnez                 0.00s             3,062   0.0ps/instruction (iss/cpu0)
    add                  0.00s             3,059   0.0ps/instruction (iss/cpu0)
    (JIT translation)    0.00s             1,637   0.0ps/instruction
    beqz                 0.00s               113   0.0ps/instruction (iss/cpu0)
    andi                 0.00s                77   0.0ps/instruction (iss/cpu0)
… snip …
    TOTAL                0.08s        60,474,242
-------------------------------------------------------------------
… snip …
```

Now the VMI Morph Time implementation of behavior

```
$ ./RUN_STAGES.sh
Please Select an Option: 22
IMPERAS Instruction Set Simulator (ISS)

… snip …
RES = 6E8D0F5A

-------------------------------------------------------------------
PROFILE REPORT
-------------------------------------------------------------------
    lw                   0.02s         5,758,452   3.5ps/instruction (iss/cpu0)
    ret                  0.01s         2,879,120   3.5ps/instruction (iss/cpu0)
    addi                 0.00s        11,522,537   0.0ps/instruction (iss/cpu0)
    mv                   0.00s         8,637,331   0.0ps/instruction (iss/cpu0)
    chacha20qr2          0.00s         5,758,080   0.0ps/instruction (iss/cpu0)
    chacha20qr1          0.00s         5,758,080   0.0ps/instruction (iss/cpu0)
    chacha20qr4          0.00s         5,758,080   0.0ps/instruction (iss/cpu0)
    chacha20qr3          0.00s         5,758,080   0.0ps/instruction (iss/cpu0)
    sw                   0.00s         2,879,403   0.0ps/instruction (iss/cpu0)
    jal                  0.00s         2,879,116   0.0ps/instruction (iss/cpu0)
    bne                  0.00s         2,879,068   0.0ps/instruction (iss/cpu0)
    bnez                 0.00s             3,062   0.0ps/instruction (iss/cpu0)
    add                  0.00s             3,059   0.0ps/instruction (iss/cpu0)
    (JIT translation)    0.00s             1,637   0.0ps/instruction
    beqz                 0.00s               113   0.0ps/instruction (iss/cpu0)
… snip …
    TOTAL                0.03s        60,474,242
-------------------------------------------------------------------
… snip …
```

The custom instructions are highlighted in the output from the two runs. The profiling indicates that the C implementation is having a noticeable effect on the overall execution time but when implemented using the Morph Time API there is no discernable time added executing these instructions.

10.3 Code Line Coverage

This is the generation of code line coverage information for the source code of the extension library and also for that of the base RISC-V processor model.

The document *Imperas CPUGenerator Guide* describes in detail how to compile and link the custom extension library with the GCOV libraries. When your test suite is then executed, using this model, code line coverage information will be accumulated. This information is processed to provide annotated source files.

In this example we re-compile the RISC-V processor model and the custom extension library using the GCOV libraries. We then execute a single test using this model and process the code line coverage data using lcov (installation usually only available on a Linux host) to generate a file that can then be examined using eGui

10.3.1 On a Linux host

```
$ ./RUN_STAGES.sh
… snip …
Please Select an Option: 25
```

The script will compile the models with coverage libraries

```
Building Models for Code Coverage
mkdir -p /d/Imperas/work/RISCV_CustomInstructionFlow/vlnvroot
echo "OTHER_CFLAGS=-DGCOV=1 -fprofile-arcs -ftest-coverage" >
/d/Imperas/work/RISCV_CustomInstructionFlow/vlnvroot/Makefile.gcov
echo "OTHER_LDFLAGS=-lgcov" >> /d/Imperas/work/RISCV_CustomInstructionFlow/vlnvroot/Makefile.gcov
cat "C:/Imperas/ImperasLib/source/riscv.ovpworld.org/processor/riscv/1.0/model/Makefile"  >>
/d/Imperas/work/RISCV_CustomInstructionFlow/vlnvroot/Makefile.gcov
make -f /d/Imperas/work/RISCV_CustomInstructionFlow/vlnvroot/Makefile.gcov -C
C:/Imperas/ImperasLib/source/riscv.ovpworld.org/processor/riscv/1.0/model
VLNVROOT=/d/Imperas/work/RISCV_CustomInstructionFlow/vlnvroot
OBJROOT=/d/Imperas/work/RISCV_CustomInstructionFlow/vlnvroot/riscv.ovpworld.org/processor/riscv/1
.0/model/obj
make[1]: Entering directory
`/c/Imperas/ImperasLib/source/riscv.ovpworld.org/processor/riscv/1.0/model'
# Host Depending
… snip …
# Host Compiling
… snip …
# Host Linking
… snip …
# Host Compiling obj/Windows64/customChaCha20.o
# Host Linking model.dll
```

The test simulation is then executed using these models

```

IMPERAS Instruction Set Simulator (ISS)


CpuManagerMulti (64-Bit) v20180917.0 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 14:10:23 2018


Info (CF_VRM) (riscv.ovpworld.org/processor/riscv/1.0=riscv.ovpworld.org/processor/riscv-
gcov/1.0) mapped riscv.ovpworld.org/processor/riscv/ to riscv.ovpworld.org/processor/riscv-
gcov/1.0
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
RES = 84772366
Info 
Info ---------------------------------------------------
Info CPU 'iss/cpu0' STATISTICS
Info   Type                  : riscv (RV32I+M)
Info   Nominal MIPS          : 100
Info   Final program counter : 0x100ac
Info   Simulated instructions: 677,012,570
Info   Simulated MIPS        : run too short for meaningful result
Info ---------------------------------------------------
Info 
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 6.77 seconds
Info   User time             : 0.33 seconds
Info   System time           : 0.05 seconds
Info   Elapsed time          : 0.38 seconds
Info   Real time ratio       : 18.06x faster
Info ---------------------------------------------------

CpuManagerMulti finished: Tue Sep 18 14:10:25 2018


CpuManagerMulti (64-Bit) v20180917.0 Open Virtual Platform simulator from www.IMPERAS.com.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.
```

The data generated is post processed to generate lcov format files which are combined into a single file and opened in eGui
The overview indicates the percentage of code lines in the file that were executed compared to those code lines found.

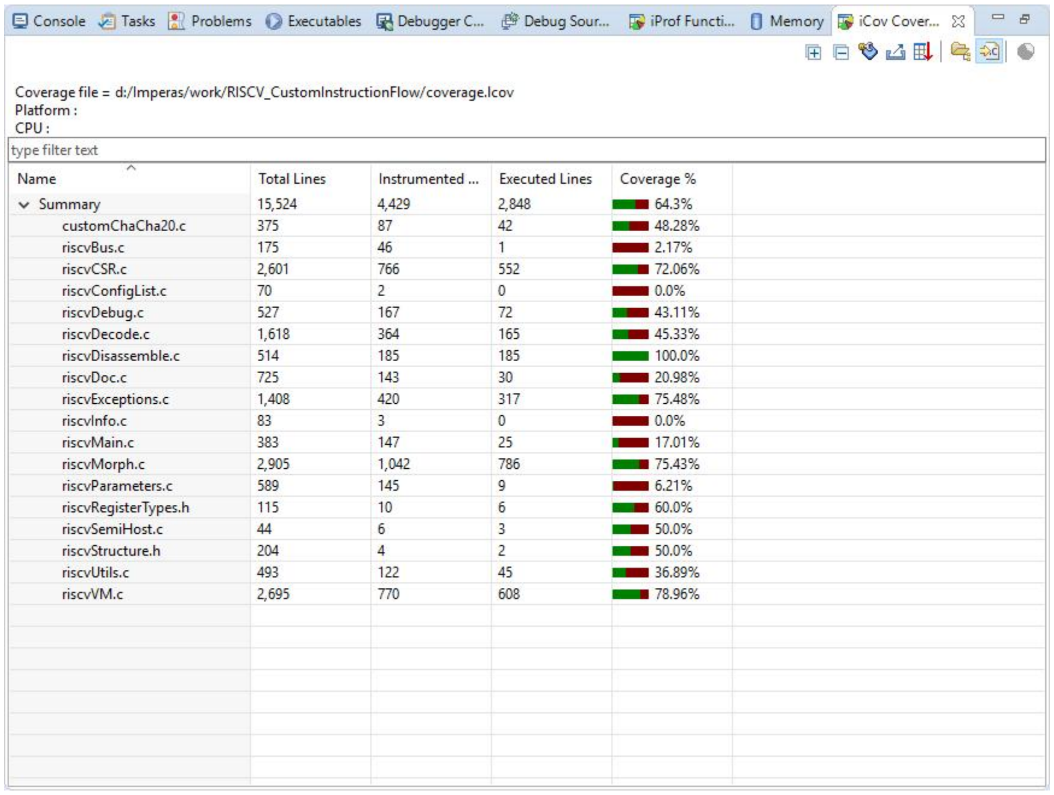
10.3.2 On a Windows host

The lcov library is not available on a Windows host to process the generated coverage data files, the script will open a pre-generated coverage file.

```
$ ./RUN_STAGES.sh
… snip …
Please Select an Option: 25
** Model Source Line Coverage not available on Windows. Opening pre-generated file
```

10.3.3 Coverage Data Output

The coverage data file (.lcov) can be opened using the eGui and presents the overview of the source lines executed.

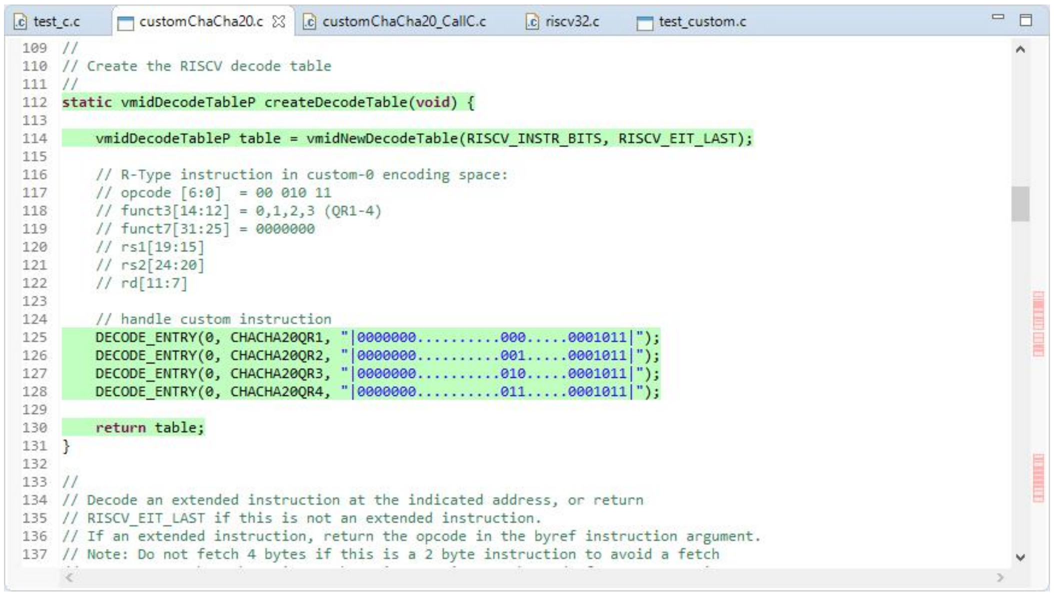


Coverage file = d:/Imperas/work/RISCV_CustomInstructionFlow/coverage.lcov
Platform :
CPU :

type filter text

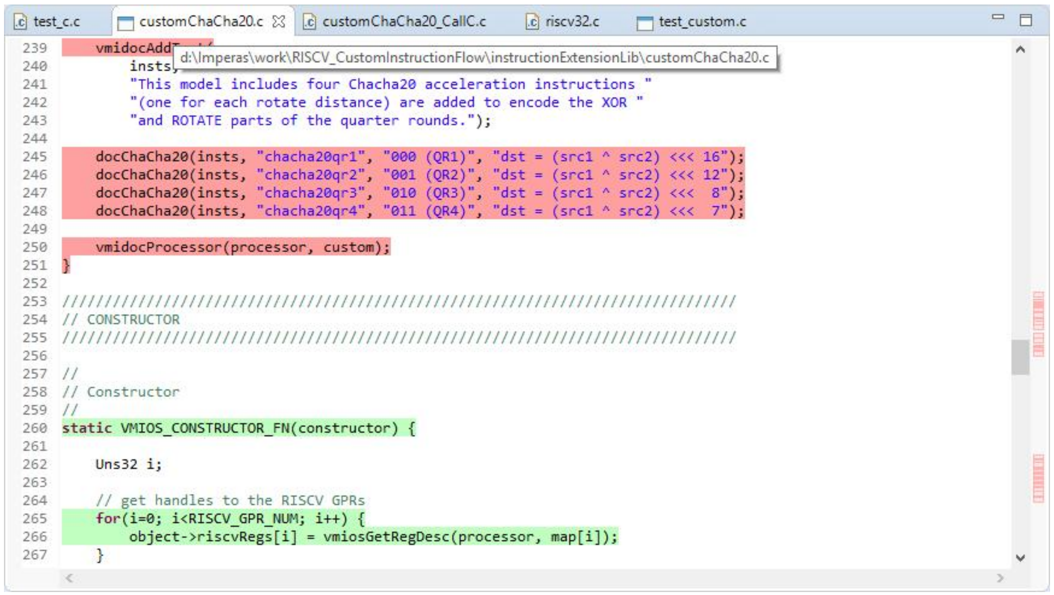
Name	Total Lines	Instrumented ...	Executed Lines	Coverage %
Summary	15,524	4,429	2,848	64.3%
customChaCha20.c	375	87	42	48.28%
riscvBus.c	175	46	1	2.17%
riscvCSR.c	2,601	766	552	72.06%
riscvConfigList.c	70	2	0	0.0%
riscvDebug.c	527	167	72	43.11%
riscvDecode.c	1,618	364	165	45.33%
riscvDisassemble.c	514	185	185	100.0%
riscvDoc.c	725	143	30	20.98%
riscvExceptions.c	1,408	420	317	75.48%
riscvInfo.c	83	3	0	0.0%
riscvMain.c	383	147	25	17.01%
riscvMorph.c	2,905	1,042	786	75.43%
riscvParameters.c	589	145	9	6.21%
riscvRegisterTypes.h	115	10	6	60.0%
riscvSemiHost.c	44	6	3	50.0%
riscvStructure.h	204	4	2	50.0%
riscvUtils.c	493	122	45	36.89%
riscvVM.c	2,695	770	608	78.96%

Each of the source files may be opened to show the actual source lines that were executed



```
//
// Create the RISCV decode table
//
static vmidDecodeTableP createDecodeTable(void) {

    vmidDecodeTableP table = vmidNewDecodeTable(RISCV_INSTR_BITS, RISCV_EIT_LAST);

    // R-Type instruction in custom-0 encoding space:
    // opcode [6:0]   = 00 010 11
    // funct3[14:12] = 0,1,2,3 (QR1-4)
    // funct7[31:25] = 0000000
    // rs1[19:15]
    // rs2[24:20]
    // rd[11:7]

    // handle custom instruction
    DECODE_ENTRY(0, CHACHA20QR1, "|0000000..........000.....0001011|");
    DECODE_ENTRY(0, CHACHA20QR2, "|0000000..........001.....0001011|");
    DECODE_ENTRY(0, CHACHA20QR3, "|0000000..........010.....0001011|");
    DECODE_ENTRY(0, CHACHA20QR4, "|0000000..........011.....0001011|");

    return table;
}

//
// Decode an extended instruction at the indicated address, or return
// RISCV_EIT_LAST if this is not an extended instruction.
// If an extended instruction, return the opcode in the byref instruction argument.
// Note: Do not fetch 4 bytes if this is a 2 byte instruction to avoid a fetch
```

And also those that were never executed


test_c.c | customChaCha20.c | customChaCha20_CallC.c | riscv32.c | test_custom.c

d:\Imperas\work\RISCV_CustomInstructionFlow\instructionExtensionLib\customChaCha20.c

```
    vmidocAdd
        insts,
        "This model includes four Chacha20 acceleration instructions "
        "(one for each rotate distance) are added to encode the XOR "
        "and ROTATE parts of the quarter rounds.");

    docChaCha20(insts, "chacha20qr1", "000 (QR1)", "dst = (src1 ^ src2) <<< 16");
    docChaCha20(insts, "chacha20qr2", "001 (QR2)", "dst = (src1 ^ src2) <<< 12");
    docChaCha20(insts, "chacha20qr3", "010 (QR3)", "dst = (src1 ^ src2) <<<  8");
    docChaCha20(insts, "chacha20qr4", "011 (QR4)", "dst = (src1 ^ src2) <<<  7");

    vmidocProcessor(processor, custom);
}

////////////////////////////////////////////////////////////////////////////////
// CONSTRUCTOR
////////////////////////////////////////////////////////////////////////////////

//
// Constructor
//
static VMIOS_CONSTRUCTOR_FN(constructor) {

    Uns32 i;

    // get handles to the RISCV GPRs
    for(i=0; i<RISCV_GPR_NUM; i++) {
        object->riscvRegs[i] = vmiosGetRegDesc(processor, map[i]);
    }
```

In this case the source lines being shown as never executed provide the disassembly output. The test was not executed using tracing and so this code was never executed.

Typically, many tests will be executed using the processor model with the extension library. The results of all tests can be accumulated to show the result after all tests are run.

11 Creating a Processor Model with Custom Instructions

Until now we have discussed the creation of an extension library that is manually added to a processor in a virtual platform simulation to extend the instantiated base processor model.

Where the custom instruction extensions are part of a fixed processor configuration a new model can be created. This provides the base instructions and the custom instructions included within a single processor instantiation.

This section will illustrate how this is done by linking to the base processor models source files and defining a mandatory extension library in the new processor model configuration information table.

In this example we will work through the steps required to create a new processor model based upon the source of the base RISC-V processor model.

11.1 Creating a New Processor Model

11.1.1 Local VLNV Library

Create a new local VLNV library structure to contain the new processor model. It is always recommended to work outside of an OVP or Imperas product installation.

For example, this can be accomplished on a Linux host or in an MSYS shell on a Windows host using the following commands to create a new library at *myLocalLib/source*.

Note: The Vendor entry should use a company specific name in place of *vendor.com* below:

```
mkdir -p myLocalLib/source/vendor.com/processor/riscv/1.0/model
cp $IMPERAS_HOME/ImperasLib/source/Makefile myLocalLib/source
```

11.1.2 Linking Existing Model Code

The new processor model source we will be based on the processor model that is located at

IMPERAS_HOME/ImperasLib/source/riscv.ovpworld.org/processor/riscv/1.0/model

For most C source files in the above directory we will create a 'link' file, for example for a file <filename> we will create a new file link.riscv.<filename>, in our new linked model directory. The 'link' file contains the inclusion of the same named source file from the original directory. Note that the header files (*.h) do not need link files created for them.

As an example, the contents of the 'link' file called link.riscv.riscvDecode.c is

```
#include "riscv.ovpworld.org/processor/riscv/1.0/model/riscvDecode.c"
```

A local version of the file riscvInfo.c must be created to contain the specific configuration information. A copy of the base model configuration file was taken and simplified as shown below

```
// VMI header files
#include "vmi/vmiAttrs.h"
#include "vmi/vmiModelInfo.h"
#include "vmi/vmiMessage.h"

// model header files
#include "riscvStructure.h"


VMI_PROC_INFO_FN(riscvProcInfo) {

    riscvP riscv = (riscvP) processor;

    static const vmiProcessorInfo info = {
        .vlnv.vendor      = "vendor.com ",
        .vlnv.library     = "processor",
        .vlnv.name        = "riscv",
        .vlnv.version     = "1.0",

        .semihost.vendor  = "riscv.ovpworld.org",
        .semihost.library = "semihosting",
        .semihost.name    = "pk",
        .semihost.version = "1.0",

        .helper.vendor    = "imperas.com",
        .helper.library   = "intercept",
        .helper.name      = "riscv32CpuHelper",
        .helper.version   = "1.0",

        .elfCode          = 243,
        .endianFixed      = True,
        .endian           = MEM_ENDIAN_LITTLE,
        .gdbPath          = "$IMPERAS_HOME/lib/$IMPERAS_ARCH/gdb/riscv64-unknown-elf-gdb"
VMI_EXE_SUFFIX,
        .gdbInitCommands  = "set architecture riscv:rv32",
        .family           = "vendor",
    };

    return &info;
}
```

This processor information is indicating the following

1) The basic 32-bit semihosting library supports this processor.
2) The basic VAP tool cpuHelper for RISCV 32-bit supports this processor.
3) The specific elf code of ‘243’ is used to identify ELF files generated for this processor.
4) The GDB provided for RISCV provided by OVP can be used to debug applications compiled for this processor.

11.1.3 Building the New Model

We must create a Makefile in the new model source directory. The Makefile is shown in the following with references to the base model in the Imperas installations VLNV library.

```
IMPERAS_HOME := $(shell getpath.exe "$(IMPERAS_HOME)")
CFLAGS+=-I$(IMPERAS_HOME)/ImperasLib/source
CFLAGS+=-I$(IMPERAS_HOME)/ImperasLib/source/riscv.ovpworld.org/processor/riscv/1.0/model
```

```
include $(IMPERAS_HOME)/ImperasLib/buildutils/Makefile.host
```

The new processor model, with source in VLNVSRC, should be built using the standard library Makefile, Makefile.Library, found in an installation, into a VLNV binary output library, in VLNVROOT, using the following commands

```
VLNVSRC=$(pwd)/myLocalLib/source
VLNVROOT=$(pwd)/lib/$IMPERAS_ARCH
mkdir -p $VLNVROOT
make -f $IMPERAS_HOME/ImperasLib/buildutils/Makefile.Library VLNVSRC=$VLNVSRC VLNVROOT=$VLNVROOT
```

This will create a binary output library that can be incorporated into a simulation using the -vlnvroot command line argument

```
-vlnvroot $(pwd)/lib/$IMPERAS_ARCH
```

11.1.4 Creating a New Configuration

As the model created above is an exact copy of the linked model it will contain all the configuration variants of the base processor model. In order that this new model contains only the variant(s) we wish to define the file link.riscv.riscvConfigList.c should be removed and a new local riscvConfigList.c created.

The following shows a file to create a new configuration definition (an OVP configuration variant) called “myRiscv”. The information to configure these arguments should be obtained from the base model.

```
//
// Defined configurations
//
static const riscvConfig configList[] = {

    {
        .name            = "myRiscv",
        .arch            = ISA_U|RV32IMAC,
        .user_version    = RVUV_2_3,
        .priv_version    = RVPV_1_11,
        .specificDocs    = "---- myRiscv Document",
        .PMP_registers   = 8,
        .tval_ii_code    = True,
        .local_int_num   = 16,
        .time_undefined  = True,
        .lr_sc_grain     = 64,
        .tvec_align      = 64,
        .d_requires_f    = True,
        .fs_always_dirty = True,
        .csrMask = {
            .cause = {u32 : {bits : 0x8000001f}},
        }
    },

    {0} // null terminator
};

//
// This returns the supported configuration list
//
riscvConfigCP riscvGetConfigList(riscvP riscv) {
    return configList;
}
```

11.1.5 Adding Extension Library

Where additional instructions are defined for this processor using an intercept library it is defined to be part of this processor in the configuration and the info structures by adding the structure member *.mandatoryExtensions* to point to a vmiVlnvInfoList containing one or more VLNV reference intercept/extension libraries.

```
VMI_PROC_INFO_FN(riscvProcInfo) {

    riscvP riscv = (riscvP) processor;

    static const vmiVlnvInfo extension = {
        .vendor      = "vendor.com",
        .library     = "intercept",
        .name        = "myRiscExtensions",
        .version     = "1.0",
    };

    static const vmiVlnvInfoList extensions = {
        0,
        &extension
    };

    static const vmiProcessorInfo info = {

        …
        .mandatoryExtensions = &extensions,

        …
    }
```

This tells the simulator that the library specified in the info structure must always be included when the processor model is instanced into a hardware definition.

##