The RISCV Sail Golden Model

A Cookbook for the RISCV ISA

William C. McSpadden, Martin Berger

Table of Contents

List of programming examples (in increasing complexity)

The main purpose of this document, is to give the user a set of programming examples for working on the RISC-V Sail model (often referred to as the RISC-V Golden Model). The examples will show the user how to change or extend the model. And it will also show the user how to write a RISC-V program (in both assembler and C) and then run it on the Golden Model.

You should read and utilize this document after you have a good handle on the Sail programming language.

Platform Configuration example (Bill)

Chapter 1. Introduction

Chapter 2. How to contribute (Bill)

2.1. Coding and indentation style

2.2. Brevity

Program examples should be short, both in terms of number-of-lines and in terms of execution time. Each example should focus on one simple item. And the execution of the example item should be clear. The example should be short, standalone and easy to maintain.

2.3. Maintainership (when something breaks)

We would also ask that if you contribute a code example, that you would maintain it.

Chapter 3. Sail installation

TBD

3.1. Ubuntu (Bill Mc.)

TBD

3.2. MacOS (Martin)

TBD

3.3. Docker

Docker is used as a ….

3.4. Windows

3.5. Windows: Cygwin (Bill Mc., low priority)

3.6. Other?

Chapter 4. Basic description

4.1. What Sail is

Sail is a programming language that is targetted for specifying an ISA. Once specified, a set of instructions (usually found in a .elf file) can then be executed on the "model" and the results observed.

The model is a sequential model only; at this time, there are no semantics allowing for any type of parallel execution.

4.2. What sail is not

Sail is not an RTL (Register Transfer Language). There is no direct support for timing (as in clock timing) and there is no support for parallel execution, all things that an RTL contains.

4.3. version management and what to expect

TBD

Chapter 5. Platform Configuration example (Bill)

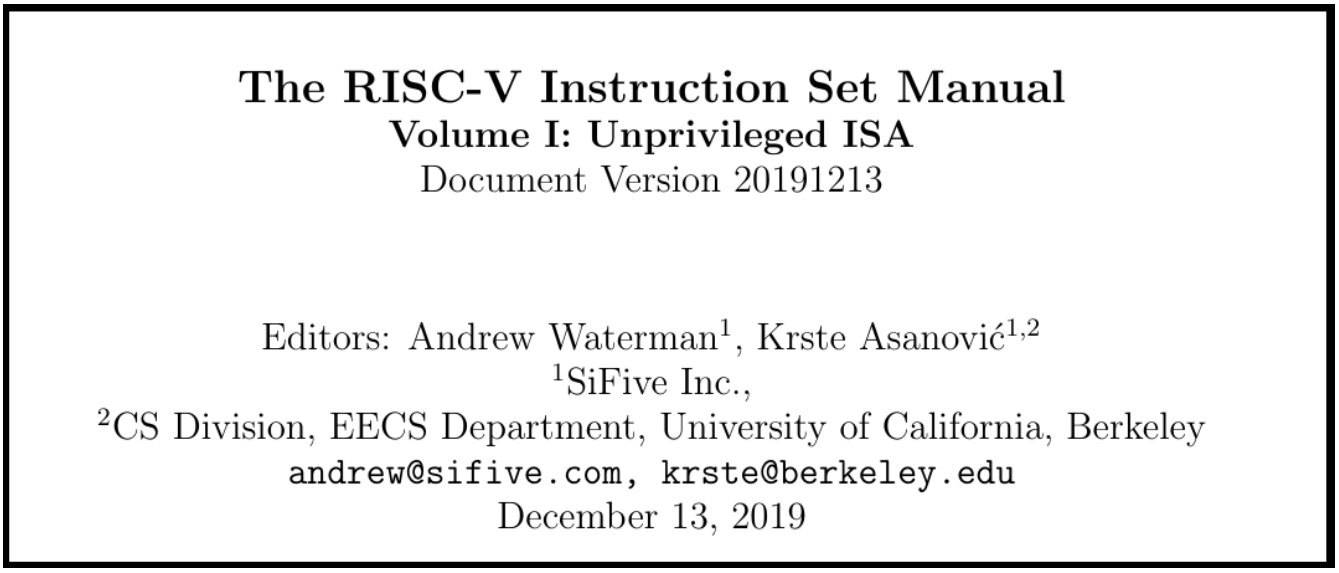
Chapter 6. Example: Add A New Extension and a New CSR

The main purpose of this cookbook, is to explain how someone can add an extension (and a CSR) to the RISC-V Sail model. This example attempts to add a very simple instruction and a very simple CSR to the model. One instruction will be added into the custom opcode space. And that instruction will be used to manipulate the new CSR, which can then be accessed by the existing CSR instructions.

This is an example of what **is**, not necessarily what it should be. This follows a pattern from the existing code.

First, we will walk through the pertinent sections of the RISC-V specifications to see what the specifications have to say about adding instructions.

Let's start with the Unprivileged Specification



The RISC-V Instruction Set Manual
Volume I: Unprivileged ISA
Document Version 20191213

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu
December 13, 2019

Chapter 26 of the Unpriv Spec ("Extending RISC-V") describes how you can extend the RISC-V instruction set.

See unpriv spec, chapter 26, "Extending RISC-V"

See unpriv spec, chapter 24, "RV32/64G Instruction Set Listings"

Chapter 24

RV32/64G Instruction Set Listings

One goal of the RISC-V project is that it be used as a stable software development target. For this purpose, we define a combination of a base ISA (RV32I or RV64I) plus selected standard extensions (IMAFD, Zicsr, Zifencei) as a "general-purpose" ISA, and we use the abbreviation G for the IMAFDZicsr_Zifencei combination of instruction-set extensions. This chapter presents opcode maps and instruction-set listings for RV32G and RV64G.

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Table 24.1: RISC-V base opcode map, inst[1:0]=11

See unpriv spec, chapter 27, "ISA Extension Naming Convention", especially setion 27.10, "Non-Standard Extension Names".

See priv spec, chapter 2, "CSR Listings", Table 2.1

CSR Address			Hex	Use and Accessibility
[11:10]	[9:8]	[7:4]		
Unprivileged and User-Level CSRs				
00	00	XXXX	0x000-0x0FF	Standard read/write
01	00	XXXX	0x400-0x4FF	Standard read/write
10	00	XXXX	0x800-0x8FF	Custom read/write
11	00	0XXX	0xC00-0xC7F	Standard read-only
11	00	10XX	0xC80-0xCBF	Standard read-only
11	00	11XX	0xCC0-0xCFF	Custom read-only
Supervisor-Level CSRs				
00	01	XXXX	0x100-0x1FF	Standard read/write
01	01	0XXX	0x500-0x57F	Standard read/write
01	01	10XX	0x580-0x5BF	Standard read/write
01	01	11XX	0x5C0-0x5FF	Custom read/write
10	01	0XXX	0x900-0x97F	Standard read/write
10	01	10XX	0x980-0x9BF	Standard read/write
10	01	11XX	0x9C0-0x9FF	Custom read/write
11	01	0XXX	0xD00-0xD7F	Standard read-only
11	01	10XX	0xD80-0xDBF	Standard read-only
11	01	11XX	0xDC0-0xDFF	Custom read-only
Hypervisor and VS CSRs				
00	10	XXXX	0x200-0x2FF	Standard read/write
01	10	0XXX	0x600-0x67F	Standard read/write
01	10	10XX	0x680-0x6BF	Standard read/write
01	10	11XX	0x6C0-0x6FF	Custom read/write
10	10	0XXX	0xA00-0xA7F	Standard read/write
10	10	10XX	0xA80-0xABF	Standard read/write
10	10	11XX	0xAC0-0xAFF	Custom read/write
11	10	0XXX	0xE00-0xE7F	Standard read-only
11	10	10XX	0xE80-0xEBF	Standard read-only
11	10	11XX	0xEC0-0xEFF	Custom read-only
Machine-Level CSRs				
00	11	XXXX	0x300-0x3FF	Standard read/write
01	11	0XXX	0x700-0x77F	Standard read/write
01	11	100X	0x780-0x79F	Standard read/write
01	11	1010	0x7A0-0x7AF	Standard read/write debug CSRs
01	11	1011	0x7B0-0x7BF	Debug-mode-only CSRs
01	11	11XX	0x7C0-0x7FF	Custom read/write
10	11	0XXX	0xB00-0xB7F	Standard read/write
10	11	10XX	0xB80-0xBBF	Standard read/write
10	11	11XX	0xBC0-0xBFF	Custom read/write
11	11	0XXX	0xF00-0xF7F	Standard read-only
11	11	10XX	0xF80-0xFBF	Standard read-only
11	11	11XX	0xFC0-0xFFF	Custom read-only

Table 2.1: Allocation of RISC-V CSR address ranges.

Other goals:

- Demonstrate the experimental switch
- Demonstrate how to code WARL fields based on settings in the YAML files.

So now that we've seen what the specifications say, let's take ka look at what that means for the Sail model.

First, we'll define a simple instruction, xmpl. This instruction

Example

Single instruction: xmpl CSR: xmpl_csr

- Takes an unsigned immediate and puts the value into the xmpl_csr
- The xmpl_csr can be read by the normal CSR instructions.
- xmpl_csr cannot be written with any form of the CSR instructions; it should generate an exception.

Files:

- (new) model/riscv_insts_custom_xampl.sail : the implmentation of the instruction and the CSR.
- (exists) Makefile : must add riscv_insts_xample.sail to the list of source files
- (exists) model/riscv_types.sail : need to add new instruction to the proper instruction opcode grouping.
- (exists) model/riscv_csr_map.sail : the address map of the CSR registers.
- (exists) mpodel/iscv_insts_zicsr.sail : need to add new CSR functionality.
- (exists) model/riscv_csr_map.sail : need to add new CSR name to the mapping
- (exists) model/riscv_sys_control.sail : need to add the new CSR name to the list found in is_CSR_defined().
- (new) cookbook/functional_code_examples/add_a_new_extension/test.S : for testing the new instruction features

riscv_insts_custom_xmpl.sail:

```
// vim: set tabstop=2 shiftwidth=2 expandtab
// ================================================================================
// Filename:    riscv_insts_custom_xmpl.sail
//
// Description: Example for adding a custom instruction, xmpl, to the RISCV model
//
// Author(s):   Bill McSpadden (bill@riscv.org)
//
// Revision:    See revision control log
// ================================================================================
```

```
/* ********************************************************************* */
/* This file specifies an example custom instruction                     */
/* It can also be used as an example when adding other ratified          */
/* extensions (while also using the ISA nomenclature).                   */

union clause ast = XTYPE : (bits(25), xop)

mapping encdec_x_xmpl : xop <-> bits(7) = {
  RISCV_X_XMPL <-> 0b0101011    // inst[6:5] == 01,  inst[4:2] == 010    --> custom-0
}

mapping clause encdec = XTYPE(imm, xop)
  <-> imm @ encdec_x_xmpl(xop)


function clause execute (XTYPE(imm, xop)) = {
  let csr_val : bitvector(25, dec) = imm;
  xmpl_csr_2->FieldWARL() = csr_val ;
  RETIRE_SUCCESS
}

mapping x_xmpl_mnemonic : xop <-> string = {
  RISCV_X_XMPL  <-> "x.xmpl"
}

mapping clause assembly = XTYPE(imm, xop)
  <-> x_xmpl_mnemonic(xop) ^ " " ^ hex_bits_25(imm)
```

Makefile (around lines 26-37):

```
SAIL_DEFAULT_INST += riscv_insts_zba.sail
SAIL_DEFAULT_INST += riscv_insts_zbb.sail
SAIL_DEFAULT_INST += riscv_insts_zbc.sail
SAIL_DEFAULT_INST += riscv_insts_zbs.sail

SAIL_DEFAULT_INST += riscv_insts_zfh.sail

SAIL_DEFAULT_INST += riscv_insts_zkn.sail
SAIL_DEFAULT_INST += riscv_insts_zks.sail

SAIL_DEFAULT_INST += riscv_insts_zbkb.sail
SAIL_DEFAULT_INST += riscv_insts_zbkx.sail

# Example custom extension (do not include this in the
#   usual model build.)
SAIL_DEFAULT_INST += riscv_insts_custom_xmpl.sail
```

model/riscv_types.sail : need to add new instruction to the proper instruction opcode grouping.

```
TODO: What changes did I make to this file????
```

model/riscv_csr_map.sail (around lines 115-120):

```
.
.
mapping clause csr_name_map = 0xF11  <-> "mvendorid"
mapping clause csr_name_map = 0xF12  <-> "marchid"
mapping clause csr_name_map = 0xF13  <-> "mimpid"
mapping clause csr_name_map = 0xF14  <-> "mhartid"
mapping clause csr_name_map = 0xFC0  <-> "xmpl_csr"    // Custom CSR example
mapping clause csr_name_map = 0xFC1  <-> "xmpl_2_csr"    // Custom CSR example
.
.
```

model/iscv_insts_zicsr.sail (around line 137):

```
.
.
    /* machine mode, custom extension example */
    (0xFC0, _)  => xmpl_csr,  // error: Xmpl_csr is not a subtype of bitvector(32,
dec)
    (0xFC1, _)  => xmpl_csr_2.bits(),
.
.
```

model/riscv_sys_control.sail (within function `is_CSR_defined()`):

```
function is_CSR_defined( csr : csreg, p : Privilege) -> bool =
.
.
    /* custom CSRs */
    0xFC0 => p == Machine,      // xmpl_csr     Example custom csr
    0xFC1 => p == Machine,      // xmpl_csr_2   Example custom csr
.
.
```

cookbook/functional_code_examples/add_a_new_extension/test.S : for testing the new instruction features

```
     1  // vim: tabstop=2  shiftwidth=2  expandtab
     2  //
--------------------------------------------------------------------------------
------
     3  /// @file       test.S
```

```
     4  ///
     5  ///
     6  /// @brief      RISC-V asm code for testing  an example custom instruction
     7  ///
     8  /// @author     Bill McSpadden (RISC-V Internation) (bill@riscv.org)
     9  //
-------------------------------------------------------------------------------
------
    10
    11  #ifndef CONFIG_BASE
    12  #error The C pre-processor variable, CONFIG_BASE, must be set.
    13  #endif
    14
    15  // -------------------------------------------------------
    16  // Support for a custom extension
    17
    18  #define X_XMPL_OPCODE   (0x2b)    // inst[6:5] == 01, inst[4:2] == 1011  -->
custom-0
    19  #define X_XMPL(__rd__, __imm__)  .word (__imm__ << 12) | (__rd__ << 7) |
(X_XMPL_OPCODE << 0)
    20
    21  #define X0    (0)
    22  #define X1    (1)
    23  #define X2    (2)
    24  #define X3    (3)
    25  #define X4    (4)
    26  #define X5    (5)
    27  #define X6    (6)
    28  #define X7    (7)
    29  #define X8    (8)
    30  #define X9    (9)
    31  #define X10   (10)
    32  #define X11   (11)
    33  #define X12   (12)
    34  #define X13   (13)
    35  #define X14   (14)
    36  #define X15   (15)
    37  #define X16   (16)
    38  #define X17   (17)
    39  #define X18   (18)
    40  #define X19   (19)
    41  #define X20   (20)
    42  #define X21   (21)
    43  #define X22   (22)
    44  #define X23   (23)
    45  #define X24   (24)
    46  #define X25   (25)
    47  #define X26   (26)
    48  #define X27   (27)
    49  #define X28   (28)
    50  #define X29   (29)
```

```
51  #define X30   (30)
52  #define X31   (31)
53
54
55
56
57  // ------------------------------------------------------
58  // Memory-mapped machine timer registers and other support
59  //  for generating a timer interrupt
60
61  //#define MMR_MTIMEL      (CONFIG_BASE + 0x0000)
62  //#define MMR_MTIMEH      (CONFIG_BASE + 0x0004)
63  //#define MMR_MTIMECMPL   (CONFIG_BASE + 0x0008)
64  //#define MMR_MTIMECMPH   (CONFIG_BASE + 0x000C)
65
66  #define MMR_MTIMEL      (CONFIG_BASE + 0xbff8)
67  #define MMR_MTIMEH      (CONFIG_BASE + 0xbffc)
68  #define MMR_MTIMECMPL   (CONFIG_BASE + 0x4000)
69  #define MMR_MTIMECMPH   (CONFIG_BASE + 0x4004)
70
71  #define TIMER_COUNT     (100)
72  #define WATCHDOG_COUNT  (100000)
73
74  #define MSTATUS_MIE     0x00000008
75  #define MSTATUS_FS      0x00006000
76  #define MSTATUS_XS      0x00018000
77
78  #define MIE_MTIE        0x80
79
80
81  // ------------------------------------------------------
82  // mcause bit definitions
83
84  #define MCAUSE_SUPERVISOR_SOFTWARE_INTERRUPT    (0x1 << (__riscv_xlen - 1) +
1)
85  #define MCAUSE_MACHINE_TIMER_INTERRUPT          (0x1 << (__riscv_xlen - 1) +
7)
86  #define MCAUSE_ILLEGAL_INSTRUCTION              (0x0 << (__riscv_xlen - 1) +
2)
87
88  // ------------------------------------------------------
89  // Support for tohost/fromhost
90
91  #define PASS_CODE       1
92  #define FAIL_CODE       1337
93
94
95  // ------------------------------------------------------
96  // Support for 32/64 bit compilation.
97
98  #if __riscv_xlen == 64
```

```
 99  # define LREG ld
100  # define SREG sd
101  # define REGBYTES 8
102  #else
103  # define LREG lw
104  # define SREG sw
105  # define REGBYTES 4
106  #endif
107
108  #define XMPL_CSR    (0xfc0)
109  #define XMPL_CSR_2  (0xfc1)
110
111  // -------------------------------------------------------
112  // Following power-on reset, we start executing at _start.
113  //  We jump to "reset_vector"
114  //
115    .section ".text.init"
116    .globl _start
117  _start:
118    la    x5,   reset_vector
119    jr    x5
120  // -------------------------------------------------------
121
122
123  // -------------------------------------------------------
124  // Initialization of the processor, starting with the
125  //  register file.
126  reset_vector:
127    li    x1,       0
128    li    x2,       0
129    li    x3,       0
130    li    x4,       0
131    li    x5,       0
132    li    x6,       0
133    li    x7,       0
134    li    x8,       0
135    li    x9,       0
136    li    x10,      0
137    li    x11,      0
138    li    x12,      0
139    li    x13,      0
140    li    x14,      0
141    li    x15,      0
142    li    x16,      0
143    li    x17,      0
144    li    x18,      0
145    li    x19,      0
146    li    x20,      0
147    li    x21,      0
148    li    x22,      0
149    li    x23,      0
```

```
150   li    x24,      0
151   li    x25,      0
152   li    x26,      0
153   li    x27,      0
154   li    x28,      0
155   li    x29,      0
156   li    x30,      0
157   li    x31,      0
158
159 // -------------------------------------------------------
160 // PMP configuration
161
162   # configure pmp to enable all accesses
163   li    t0,       0x1f
164   csrw  pmpcfg0,  t0
165   li    t0,       0xffffffff
166   csrw  pmpaddr0, t0
167
168 // -------------------------------------------------------
169 // initialize machine trap vector
170   la    x5,       machine_trap_entry
171   csrw  mtvec,    x5
172
173
174 // -------------------------------------------------------
175 //  The test!
176
177 the_test_begin:
178   X_XMPL(X2, 0x0dead)
179   csrr  x3, XMPL_CSR_2
180
181 //  li    x4, 0x76543210
182 //  csrw  XMPL_CSR_2, x4    // Q: What happens to a write to a read-only csr?
183 //                          // A: illegal_instruction trap
184 the_test_end:
185
186
187
188
189 // -------------------------------------------------------
190 // PASS: The end of the test,  if successful
191 j_target_end_pass:
192   // exit code construction
193   li    x10,      PASS_CODE
194   la    x13,      tohost
195   sw    x10,      0(x13)
196   la    x5,       j_target_end_pass
197   jalr  x5
198   j     j_target_end_fail                       // should never be taken
199
200 // -------------------------------------------------------
```

```
201
202  // -------------------------------------------------------
203  // FAIL: The end of the test,  if unsuccessful
204  j_target_end_fail:
205    // exit code construction
206    li    x10,      FAIL_CODE
207    la    x13,      tohost
208    sw    x10,      0(x13)
209    la    x5,       j_target_end_fail
210    jalr  x5
211
212
213  // -------------------------------------------------------
214  // In support of vectored interrupt,  although it's not
215  //  being used in this test.
216
217    .align 4
218  machine_trap_entry:
219    j     machine_trap_entry_0
220    .align 2
221    j     machine_trap_entry_1
222    .align 2
223    j      machine_trap_entry_2
224    .align 2
225    j      machine_trap_entry_3
226    .align 2
227    j      machine_trap_entry_4
228    .align 2
229    j      machine_trap_entry_5
230    .align 2
231    j      machine_trap_entry_6
232    .align 2
233    j      machine_trap_entry_7
234    .align 2
235    j      machine_trap_entry_8
236    .align 2
237    j      machine_trap_entry_9
238    .align 2
239    j      machine_trap_entry_10
240    .align 2
241    j      machine_trap_entry_11
242  // -------------------------------------------------------
243
244
245  // -------------------------------------------------------
246    .align 2
247  machine_trap_entry_0:
248    csrr    x7,       mcause
249    li      x6,       MCAUSE_MACHINE_TIMER_INTERRUPT
250    bne     x7,       x6,     not_a_timer_interrupt
251    li      x6,       0x1
```

```
   252    la      x7,       timer_interrupt_flag
   253    sw      x6,       0(x7)
   254
   255    // Turn off timer interrupt. No longer needed
   256    addi    x7,       x0,     MIE_MTIE
   257    csrc    mie,      x7
   258
   259    // Clear interrupt
   260    li      x7,       MSTATUS_MIE
   261    csrc    mstatus,  x7
   262
   263    // and return
   264    mret
   265
   266  not_a_timer_interrupt:
   267    // Do not try and correct the opcode,  and do not
   268    //    do an mret. This should probably be the last
   269    //    part of this simple test.
   270    csrr    x7,       mcause
   271    li      x6,       MCAUSE_ILLEGAL_INSTRUCTION
   272    j       j_target_end_fail
   273  // ------------------------------------------------------
   274
   275  // ------------------------------------------------------
   276  // None of these machine traps should have been taken
   277  //  Jump to test failure
   278  machine_trap_entry_1:
   279  machine_trap_entry_2:
   280  machine_trap_entry_3:
   281  machine_trap_entry_4:
   282  machine_trap_entry_5:
   283  machine_trap_entry_6:
   284  machine_trap_entry_7:
   285  machine_trap_entry_8:
   286  machine_trap_entry_9:
   287  machine_trap_entry_10:
   288  machine_trap_entry_11:
   289    csrr    x7,       mcause        // Do the read so that it appears in the log
file for debug.
   290    j       j_target_end_fail
   291  // ------------------------------------------------------
   292
   293
   294
   295  // ------------------------------------------------------
   296  // Memory locations for specific usage.
   297  .section ".tdata.begin"
   298  .globl _tdata_begin
   299  _tdata_begin:
   300
   301  .section ".tdata.end"
```

```
302  .globl _tdata_end
303  _tdata_end:
304
305  .section ".tbss.end"
306  .globl _tbss_end
307  _tbss_end:
308
309  .section ".tohost","aw",@progbits
310  .align 6
311  .globl tohost
312  tohost: .dword 0
313
314  .section ".fromhost","aw",@progbits
315  .align 6
316  .globl fromhost
317  fromhost: .dword 0
318
319  .align 6
320  .global timer_interrupt_flag
321  timer_interrupt_flag: .dword 0
322
323
324
325
```

You will probably have to add command line switches to enable/disable extensions/functionality. Files that need to be touched are:

- (exists) c_emulator/riscv_sim.c : implements the longopts functionality
- (exists) model/riscv_sys_regs.sail : function signatures for sys_enable_XXX() functionms.
- (exists) c_emulator/riscv_platform_impl.* : global variables for holding enabled state vars
- (exists) c_emulator/riscv_platform.c : implements the C functions that will be made available to Sail; functions like sys_enable_zfinx().

What does the test.dump file look like? Remember, the RISC-V assembler knows nothing about the custom instruction we have added.

cookbook/functional_code_examples/add_a_new_extension/test.dump:

```
.
.
89  80000062 <the_test_begin>:
90  80000062:   0dead12b            0xdead12b
91  80000066:   fc1021f3            csrr    gp,0xfc1
.
.
```

What does the Sail log file look like?

```
.
.
   424  model/riscv_step.sail
   425  model/riscv_step.sail:75.25-75.32
   426  entering step() function...
   427
   428  mem[X,0x80000062] -> 0xD12B
   429  mem[X,0x80000064] -> 0x0DEA
   430  [41] [M]: 0x80000062 (0x0DEAD12B) x.xmpl 1824162
   431
   432
   433  model/riscv_step.sail
   434  model/riscv_step.sail:75.25-75.32
   435  entering step() function...
   436
   437  mem[X,0x80000066] -> 0x21F3
   438  mem[X,0x80000068] -> 0xFC10
   439  [42] [M]: 0x80000066 (0xFC1021F3) csrrs gp, xmpl_2_csr, zero
   440  CSR xmpl_2_csr -> 0x001BD5A2
   441  x3 <- 0x001BD5A2
.
.
```

Chapter 7. FAQs (Frequently Asked Questions)

Following are a set of FAQs that were generated via set of questions to the Sail developers.

7.1. Frequently Asked Questions about the Sail RISC-V Golden Model

7.1.1. Q: Is there support for multi-HART or multi-Core simulation?

A: There is no inherent support for multi-HART or multi-Core within the existing RISC-V Sail model. There are future plans for adding this kind of simulation. It is needed in order to simulate (in a meaningful way) the atomic memory operations and to evaluate memory consistency and coherency.

The model isn't directly about testing. Testing is a separate activity. The point of the model is to be as clear as possible. and we should keep testing and the model separate.

7.1.2. Q: What are .ml files? What are their purpose?

A: These are OCaml files. They are to the ocaml emulator what the .c files are to the c emulator. I question the need for an OCaml emulator ,see also https://github.com/riscv/sail-riscv/issues/138

7.1.3. Q: Is there any support for MTIMER?

A: Yes. MTIMER functionality lives in riscv_platform.sail. At this date (2022-05-27) it lives at a fixed MMIO space as specified by the MCONFIG CSR. In the future, once the Golden Model supports the RISCV_config YAML structure, the MTIMER can be assigned any address.

7.1.4. Q: Is the "main loop" coded in Sail?

A: The initial answer to this question ("The main execution loop can be found in `main.sail`.") is incorrect. main.sail is not executed in the RISC-V model, even though it is compiled into the model.

The main loop is actually found on the C side in the file `c_emulator/riscv_sim.c` in the function `run_sail()`. In this function, the Sail function, `zstep()`, is called (which is the Sail function, `step()`)

7.1.5. Q: Can gdb attach to the RISCV Golden Model to debug RISCV code?

A: Not at this time (2022-05-27). It is being looked at as an enhancement.

7.1.6. Q: There are two C executables built: riscv_sim_RV32 and riscv_sim_RV64. Is there a reason why we need two executables? Can't XLEN be treated as a run-time setting rather than a compile time setting?

A: (Response from Martin Berger) I think this would require a redesign of the Sail code because of the way Sail's liquid types work. Currently xlen is a global type constant, that is used, directly or indirectly, everywhere. As a type-constant it is used during type checking. The typing system might (note the subjunctive) be flexible enough to turn this into a type-parameter, but probably not without major code surgery. I think we should ask the Cambridge team why they decided on the current approach.

7.1.7. Q: Is there support in the model for misaligned memory accesses?

A: (Response from Martin Berger) Short answer: I don't know. Alignment stuff is distributed all over the code base. riscv_platform.sail has some configuration options for this. Maybe that's a place to start looking?

7.1.8. Q: What is the meaning of life, the universe and everything?

A: 42

7.1.9. Q: What does the answer to "What is the meaning of life, the universe and everything" mean?

A: One must construct an experimental, organic computer to compute the meaning. Project *Earth* is one such computer. Timeframe for an expected answer is… soon.

Chapter 8. Colophon

This document was prepared on an Ubuntu Linux workstation using Microsofts VSCode for editing and rendering the asciidoc text.

`shutter` was used for screenshots of various parts of the RISCV specifications and were saved in PNG format.

These screenshots were then edited using `gimp` to highlight the pertinent sections of the screenshot.

`asciidoctor-reducer` was used to combine and resolve all cross-document references and put them into one .adoc file, TheRISCVSailCookbook_Complate.adoc.

The pdf was created using `asciidoctor-pdf` .

See the Makefile, `cookbook/doc/Makefile`, for the recipe for building the document.