

# The Sail Programming Language

## *A Sail Cookbook*

William C. McSpadden, Martin Berger

# Table of Contents

List of programming examples (in increasing complexity) .....	2
1. Introduction .....	3
2. How to contribute (Bill) .....	4
2.1. Coding and indentation style .....	4
2.2. Brevity .....	4
2.3. Maintainership (when something breaks) .....	4
2.4. Syntax highlighting for Sail .....	4
3. Sail installation .....	6
3.1. Ubuntu (Bill Mc.) .....	6
3.2. MacOS (Martin) .....	6
3.3. Docker .....	6
3.4. Windows .....	6
3.5. Windows: Cygwin (Bill Mc., low priority) .....	6
3.6. Other? .....	6
4. Basic description .....	7
4.1. What Sail is .....	7
4.2. What sail is not .....	7
4.3. version management and what to expect .....	7
5. “Hello, World” example program (Bill) .....	8
5.1. "Hello, World" and the Sail interactive interpreter .....	12
6. Data types .....	14
6.1. effect annotations .....	14
6.2. Integers .....	14
6.3. type variables .....	14
6.4. Bits .....	14
6.5. Strings .....	14
6.6. Lists .....	14
6.7. Structs .....	14
6.8. mappings .....	14
6.9. Liquid data types (Martin) .....	14
7. Execution .....	15
7.1. Functions .....	15
7.2. The ``my_replicate_bits()`` function from the Sail manual (Bill) .....	15
8. Control flow .....	19
8.1. for loops (TBD) .....	19
8.2. while loops (TBD) .....	19
8.3. foreach (TBD) .....	19
8.4. repeat (TBD) .....	19

8.5. List iteration example (Bill) .....	19
8.6. match .....	21
9. Interaction between C and Sail .....	22
9.1. Sail Calls C function .....	25
10. Other stuff .....	32
10.1. Scattered definitions: Why? What are they? .....	32
10.2. <i>FILE</i> , <i>LINE</i> , <i>LOC</i> .....	36
10.3. Variable argument list support .....	36
11. Description prelude.sail .....	37
11.1. description of print, sext, equility etc. standard template stuff .....	37
11.2. the C interface .....	37
12. Simple CPU example (Martin) .....	38
13. Formal tools that analyze Sail source code .....	39
14. FAQs (Frequently Asked Questions) .....	40
14.1. Frequently Asked Questions about the Sail Programming Language .....	40
14.2. Frequently Asked Questions about the Sail RISC-V Golden Model .....	41



# List of programming examples (in increasing complexity)

The main purpose of this document, is to give the user a quick reference to Sail coding examples. The following is a list of all the programming examples found in this document.

[“Hello, World” example program \(Bill\)](#)

[The ``my\\_replicate\\_bits\(\)`` function from the Sail manual \(Bill\)](#)

[List iteration example \(Bill\)](#)

[Scattered definitions: Why? What are they?](#)

[Sail Calls C function](#)

[Simple CPU example \(Martin\)](#)

# Chapter 1. Introduction

Sail is a programming language that was developed for the purpose of clearly, concisely and completely describing a computer's Instruction Set Architecture (ISA). This includes...

- specifying the opcodes/instructions and their behaviours
- specifying the general purpose registers
- specifying the control space registers

Sail was the language chosen by RISC-V International to formally specify the RISC-V open source ISA.

This document, while not RISC-V specific, is especially targeted for engineers who are working on specifying the RISC-V ISA.

This cookbook is intended to supply the beginning Sail programmer with some simple, well-commented, bite-size program fragments that can be compiled and run.

**github** is used to host the development of Sail. You can find the repository at the following URL:

<https://github.com/rem-s-project/sail>

Currently, the work on this cookbook can be found on a branch in the above repo. This branch is:

[https://github.com/billmcspadden-riscv/sail/tree/cookbook\\_br](https://github.com/billmcspadden-riscv/sail/tree/cookbook_br)

So this is the place you should probably clone. (Eventually, this branch will be merged to the release branch.)

Other documentation:

There is another useful Sail document that you should know about. It is "The Sail instruction-set semantics specification language" by Armstrong, et. al. It can be found at:

[https://github.com/billmcspadden-riscv/sail/blob/cookbook\\_br/manual.pdf](https://github.com/billmcspadden-riscv/sail/blob/cookbook_br/manual.pdf)

While useful, the document does not contain a useful set of programming examples. That is the purpose of **this** document.

# Chapter 2. How to contribute (Bill)

We are hopeful that as you learn the Sail programming language, that you too would want to create some code snippets that you think someone might find helpful.

The simple "hello world" program (found in `cookbook/functional_code_snippets/hello_world/`) provides a template for writing a new code snippet. For an example that lives in a single Sail file, this should be sufficient. Create a test directory (with a useful name), copy the Makefile and the .sail file into that directory, and then write your code. And finally, edit this .adoc file and give a description of what the example file is intended to do.

Once you have completed your snippet and verifies that it works, you should make an entry in this document. Please see [“Hello, World” example program \(Bill\)](#) to see how you should include your snippet in this document. You should at least include the .sail file and give a brief description. Also, please make an entry in [List of programming examples \(in increasing complexity\)](#) for quick perusal by readers.

## 2.1. Coding and indentation style

We do not have a preferred coding style for these little code snippets. With regards to indentation style, the RISC-V mode follows a vaguely K&R style. Some of the program snippets (those originating with Bill McSpadden) follow the Whitesmiths indentation style. All styles are welcome.

For a list and description of popular indentation styles, steer your browser to... [https://en.wikipedia.org/wiki/Indentation\\_style](https://en.wikipedia.org/wiki/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.

Now, we do have one example in this Cookbook that somewhat violates this request. The programming example, [\[simple\\_cpu\\_exempl\]](#), is more complex. But it is meant to demonstrate the usefulness of Sail in defining the functionality of an ISA.

## 2.3. Maintainership (when something breaks)

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

## 2.4. Syntax highlighting for Sail

Syntax highlighting for several editors (emacs, vim, Visual Studio, etc) can be found at:

<https://github.com/rems-project/sail/tree/sail2/editors>

It is beyond the scope of this document to describe how to use the syntax highlighting for the

various editors.



# Chapter 3. Sail installation

Sail is supported on a number of different platforms. MacOS and Linux/Ubuntu seem to be the most used platforms.

TBD

## 3.1. Ubuntu (Bill Mc.)

TBD

## 3.2. MacOS (Martin)

TBD

## 3.3. Docker

Docker is used as a ....

## 3.4. Windows

Support of a native command line interface is not planned. If you want to run Sail under Windows, plan on running it under Cygwin.

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

If there is a demand, a port to Cygwin will be attempted.

## 3.6. Other?

Are there other OS platforms that should be supported? Other Linux distis? Or will Docker support?

# 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. “Hello, World” example program (Bill)

All example programs associated with this cookbook, can be found in `<sail_git_root>/cookbook/functional_code_snippets/`

The purpose of this simple program is to show some of the basics of Sail and to ensure that you have the Sail compiler (and the other required tools) installed in your environment.

It is assumed that you have built the sail compiler in the local area. The Makefiles in the coding examples depend on this.

The following code snippet comes from:

[https://github.com/billmcspadden-riscv/sail/tree/cookbook\\_br/cookbook/functional\\_code\\_snippets/hello\\_world](https://github.com/billmcspadden-riscv/sail/tree/cookbook_br/cookbook/functional_code_snippets/hello_world)

hello\_world.sail:

```
// =====

// Two types of comments...
// This type and ...

/*
...block comments
*/

// Whitespace is NOT significant. Yay!

default Order dec // Required. Defines whether bit vectors are increasing
                  // (inc) (MSB is index 0; AKA big-endian) or decreasing
                  // (dec) (LSB is index 0; AKA little-endian)

// The $include directive is used to pull in other Sail code.
// It functions similarly, but not exactly the same, as the
// C preprocessor directive.

// Sail is a very small language. In order to get a set
// of useful functionality (eg - print to stdout), a set
// of functions and datatypes are defined in the file
// "prelude.sail"
#include <prelude.sail>

// =====
// Function signatures (same idea as C's function prototype)
// =====

val "print" : string -> unit

val main : unit -> unit

// =====
// The entry point into the program starts at the function, main.
// =====
function main() =
{
    print("hello, world!\n") ;
    print("hello, another world!\n") ;
}
```

So... that's the code we want to compile. But how do we compile it? Remember, we want to use the sail compiler that was built in this sandbox. We use a *make* methodology for building. The first Makefile (in the same directory as the example code example) is very simple. It includes a generic Makefile (./Makefile.generic) that is used for building most of the program examples.

[Note] If you want to create and contribute your own example program and you need to deviate from our make methodology, you would do that in your own test directory by writing your own

Makefile.

The basic flow for building is:

1. Write \*.sail
2. sail -c \*.sail -o out.c
3. gcc <flags> \*.c -> executable

Makefile:

```
# vim: set tabstop=4 shiftwidth=4 noexpandtab
# =====
# Filename:      Makefile
#
# Description:   Makefile for building example code
#
# Author(s):    Bill McSpadden (bill@riscv.org)
#
# Revision:     See revision control log
#
# =====

#=====
# Includes
#=====

include ../Makefile.generic
```

Makefile.generic is the Makefile that does the work for compilation. It depends on a local compilation of sail. See the [Installation](#sail-installation) section to understand how to install in the tools for your platform.

Makefile.generic:

```
# vim: set tabstop=4 shiftwidth=4 noexpandtab
# =====
# Filename:      Makefile
#
# Description:   Makefile for building Sail example code fragments
#
#               NOTE: in order to render this file in an asciidoc
#               for the Sail cookbook, keep the line length less
#               than 86 characters, the width of the block comment line
#               of this section
#
# Author(s):    Bill McSpadden (bill@riscv.org)
#
# Revision:     See revision control log
#
```

```
# =====

#=====
# Includes
#=====

#=====
# Make variables
#=====

# The sail compiler expects that SAIL_DIR is set in the environment.
# The sh env var, SAIL_DIR, is set and exported using the make
# variable, SAIL_DIR. I hope this is not too confusing.
SAIL_DIR      := ../../..
SAIL_LIB      := ${SAIL_DIR}/lib/sail
SAIL          := ${SAIL_DIR}/sail
SAIL_OUTFILE  := out
SAIL_FLAGS    := -c -o ${SAIL_OUTFILE}

SAIL_SRC      ?= $(wildcard *.sail)

CC            := gcc
CCFLAGS       := -lgmp -lz -I ${SAIL_DIR}/lib/

# out.c is the file that sail generates as output from the
# sail compilation process. It will be compiled with
# other C code to generate an executable
# ${SAIL_DIR}/lib/*.c is a set of C code used for interaction
# with the programming environment. It also provides
# functionality that cannot be natively supported by sail.
#
C_SRC         := out.c ${SAIL_DIR}/lib/*.c

TARGET        := out

#=====
# Targets and Rules
#=====

all: run

build: out

install:

run: out
    ./out

out: out.c
    gcc ${C_SRC} ${CCFLAGS} -o $@
```

```
# gcc out.c ${SAIL_DIR}/lib/*.c -lgmp -lz -I ${SAIL_DIR}/lib -o $@

# In the following rule, the environment variable, SAIL_DIR, must be
# set in order for the sail compilation step to work correctly.
out.c: ${SAIL_SRC}
    SAIL_DIR=${SAIL_DIR} ; export SAIL_DIR ; \
    ${SAIL} ${SAIL_FLAGS} ${SAIL_SRC}

# clean: cleans only local artifacts
clean:
    rm -f out out.c out.ml

# Cleans local artifacts and the install location
clean_all:
```

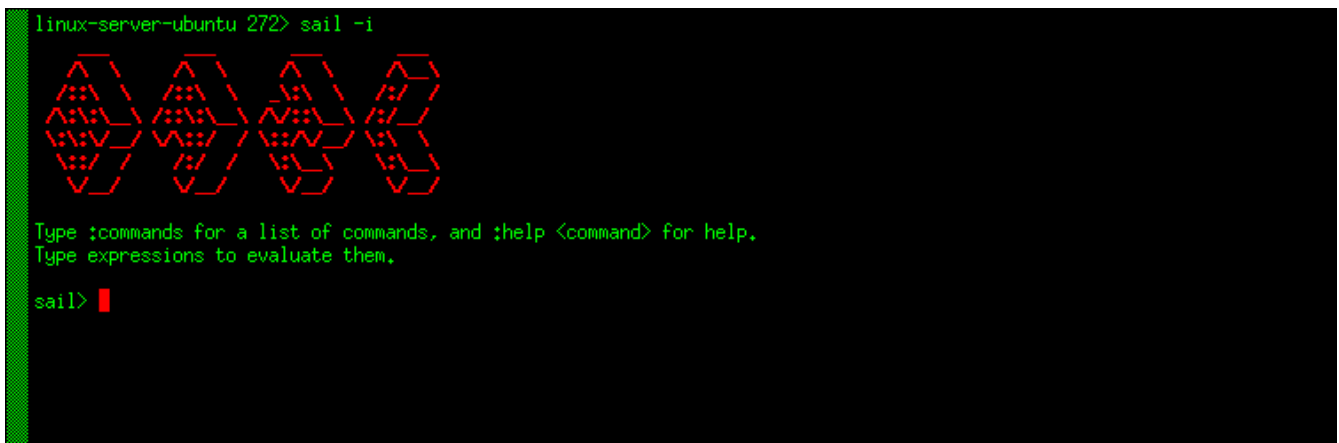
What does the compilation process look like? Under Ubuntu Linux, this is the output you can expect for compiling and running the "hello world" example program.

```
ubuntu-VirtualBox 227> make
SAIL_DIR=../../.. ; export SAIL_DIR ; \
../.././sail -c -o out hello_world.sail
gcc out.c ../.././lib/*.c -lgmp -lz -I ../.././lib/ -o out
./out
hello, world!
hello, another world!
ubuntu-VirtualBox 228>
```

Now that we've examined the Makefiles, we will make little mention of them in the rest of this document (except for the example where we discuss the C foreign function interface where we will show how Sail can call C functions).

## 5.1. "Hello, World" and the Sail interactive interpreter

Invocation:



```
linux-server-ubuntu 272> sail -i

Type :commands for a list of commands, and :help <command> for help.
Type expressions to evaluate them.

sail> █
```





# Chapter 6. Data types

## 6.1. effect annotations

## 6.2. Integers

- Int
- int
- Multi-precision

## 6.3. type variables

What does " 'n " mean?

## 6.4. Bits

## 6.5. Strings

## 6.6. Lists

## 6.7. Structs

## 6.8. mappings

## 6.9. Liquid data types (Martin)

# Chapter 7. Execution

## 7.1. Functions

## 7.2. The ``my\_replicate\_bits()`` function from the Sail manual (Bill)

First, let's look at the code that is described in the Sail manual for the function, `my_replicate_bits()`.

Note: The following code actually comes from the file `doc/examples/my_replicate_bits.sail`. It is a little bit different than what is shown in the manual for reasons that will be covered later.

```
//default Order dec          // billmc

#include <prelude.sail>

// billmc
#include "my_replicate_bits_function_signatures.sail"

infixl 7 <<
infixl 7 >>

val operator << = "shiftrl" : forall 'm. (bits('m), int) -> bits('m)
val "shiftrl" : forall 'm. (bits('m), int) -> bits('m)

val operator >> = {
  ocaml: "shiftr_ocaml",
  c: "shiftr_c",
  lem: "shiftr_lem",
  _: "shiftr"
} : forall 'm. (bits('m), int) -> bits('m)

//val "or_vec" : forall 'n. (bits('n), bits('n)) -> bits('n)
val or_vec = {c: "or_bits" } : forall 'n. (bits('n), bits('n)) -> bits('n)      //
billmc

val zero_extend = "zero_extend" : forall 'n 'm, 'm >= 'n. (bits('n), atom('m)) ->
bits('m)

overload operator | = {or_vec}

//val my_replicate_bits : forall 'n 'm, 'm >= 1 & 'n >= 1. (int('n), bits('m)) ->
bits('n * 'm)      // billmc

val zeros = "zeros" : forall 'n. atom('n) -> bits('n)

function my_replicate_bits(n, xs) = {
```

```

ys = zeros(n * length(xs));
foreach (i from 1 to n) {
  ys = ys << length(xs);
  ys = ys | zero_extend(xs, length(ys))
};
ys
}

val my_replicate_bits_2 : forall 'n 'm, 'm >= 1 & 'n >= 1. (int('n), bits('m)) ->
bits('n * 'm)

function my_replicate_bits_2(n, xs) = {
  ys = zeros('n * 'm);
  foreach (i from 1 to n) {
    ys = (ys << 'm) | zero_extend(xs, 'n * 'm)
  };
  ys
}

// The following comment is of interest for reasons other than
// functionality. The Sail syntax is still being developed.
// Attention should be paid to the issues reported to the Sail
// team (via github) and when releases are made (again via github).

// The following is deprecated per Alasdair Armstrong:
// I would just remove that example as the cast feature is now
// deprecated in the latest version (and the risc-v model has
// always used a flag fully disabling it anyway)

// val cast_extz : forall 'n 'm, 'm >= 'n. (implicit('m), bits('n)) -> bits('m)
//
//function extz(m, xs) = zero_extend(xs, m)
//
//val my_replicate_bits_3 : forall 'n 'm, 'm >= 1 & 'n >= 1. (int('n), bits('m)) ->
bits('n * 'm)
//
//function my_replicate_bits_3(n, xs) = {
//  ys = zeros('n * 'm);
//  foreach (i from 1 to n) ys = ys << 'm | xs;
//  ys
//}

```

You will see in this code, that there is no *main* function, and as such, will not compile into a C Sail model. You will get the following error message:

TODO: get the error message.

In order to get this to compile into a C Sail model, you will need to provide a main function. The following code shows the implementation of a ``main()`` function that calls `my_replicate_bits()`.

```
// vim: set tabstop=4 shiftwidth=4 expandtab
// =====
// Filename:    main.sail
//
// Description: Example sail file
//
// Author(s):   Bill McSpadden (bill@riscv.sail)
//
// Revision:    See revision control log
// =====

default Order dec
$include <prelude.sail>

val "print" : string -> unit

//val my_replicate_bits : forall 'n 'm, 'm >= 1 & 'n >= 1. (int('n), bits('m)) ->
bits('n * 'm)
$include "my_replicate_bits_function_signatures.sail"

val main : unit -> unit
function main() =
{
  v1 : bits(8)  = 0x55;
  v2 : bits(32) = 0x00000000;

  // Sail has a powerful type-checking system, but understanding it
  // is best learned by examining some examples.

  //   num : int = 4;           // CE
  //   let num : int(4) = 4;     // Works
  //   let num : int(4) = 5;     // CE
  //   let num : int(5) = 5;
  //   let num : int(4) = 3;     // CE
  let num : int(4) = 3 + 1;

  print("calling my_replicate_bits() .....\\n");

  // The compiler needs to evaluate
  //   v2 = my_replicate_bits (num, v1);
  v3 : bits(32) = my_replicate_bits (num, v1);
  //   v3 : bits(32) = my_replicate_bits (4, v1);

  print_bits("replicated bits: ", v3);

  print("returned from my_replicate_bits() .....\\n");
}
```

Because both the files, `my_replicate_bits.sail` and `main.sail`, need to have the function signatures in

order to compile (and we want them to be consistent), the function signatures have been put into a separate file that is included by both. Here is the function signature file, `my_replicate_bits_function_signatures.sail`:

```
// vim: set tabstop=4 shiftwidth=4 expandtab
// =====
// Filename:    my_replicate_bits_function_signatures.sail
//
// Description:
//
// Author(s):   Bill McSpadden (bill@riscv.org)
//
// Revision:    See revision control log
// =====

#include <prelude.sail>

val "print" : string -> unit

val my_replicate_bits : forall 'n 'm, 'm >= 1 & 'n >= 1. (int('n), bits('m)) ->
bits('n * 'm)

val main : unit -> unit
```

# Chapter 8. Control flow

## 8.1. for loops (TBD)

## 8.2. while loops (TBD)

## 8.3. foreach (TBD)

## 8.4. repeat (TBD)

## 8.5. List iteration example (Bill)

```
// vim: set tabstop=4 shiftwidth=4 expandtab
// =====
// File:      test.sail
//
// Description: test file for figuring out how to iterate through
//              a Sail list.
//
//              Using code and structure for Ben Marshall's implemetation
//              of RISC-V crypto-scalar code. (riscv_types_kext.sail).
//
// Author(s):  Bill McSpadden
//
// History:    See git log
// =====

default Order dec

$include <prelude.sail>

overload operator - = sub_bits

val not_vec = {c: "not_bits", _: "not_vec"} : forall 'n. bits('n) -> bits('n)

let aes_sbox_inv_table : list(bits(8)) =
  [
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38,
    0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87,
    0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,

    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d,
    0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2,
    0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,
```

```

0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16,
0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,
0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda,
0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,

0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a,
0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,
0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02,
0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,

0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea,
0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85,
0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,

0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89,
0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20,
0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,

0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31,
0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d,
0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,

0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0,
0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26,
0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
]

```

```

// Lookup function - takes an index and a list, and retrieves the
// x'th element of that list.

```

```

val sbbox_lookup : (bits(8), list(bits(8))) -> bits(8)
function sbbox_lookup(x, table) =
{
  match (x, table)
  {
    (0x00, head::tail) => head,
    (   y, head::tail) => sbbox_lookup(x - 0x01, tail)
  }
}

val main : unit -> unit
function main() =
{
  let x : bits(8) = 0x03;

```

```
print_bits("lookup results: ", sbbox_lookup(x, aes_sbbox_inv_table));  
}
```

## 8.6. match



# Chapter 9. Interaction between C and Sail

## Can we call Sail functions in the C model?

Short answer: yes!

In more detail, every Sail function will show up with a predictable name in the generated C (with one caveat). For example, if have the following Sail code:

```
default Order dec
#include <prelude.sail>

val giraffe1 : unit -> int
function giraffe1 () = {
    return 1
}

val giraffe2 : unit -> int

function giraffe3 () -> int = {
    return 3
}

val giraffe4 : unit -> int
function giraffe4 () = {
    return 4
}

val main : unit -> int effect {rreg, wreg}
function main () = {
    let x1 = giraffe1() in
    let x2 = giraffe2() in
    let x3 = giraffe3() in
    return 7
}
```

then we get the following C code (abbreviated).

```
void zgiraffe1(sail_int *rop, unit);

void zgiraffe1(sail_int *zcbz30, unit zgsz30)
{
    ...
}
```

for **giraffe1** (and likewise for *giraffe3*). Note that the code for **giraffe2** is simply this:

```
void zgiraffe2(sail_int *rop, unit);
```

So giraffe1 becomes `zgiraffe1`, `giraffe2` becomes `zgiraffe2` and so on. If we only provide a Sail declaration but no corresponding Sail implementation (as we do for `giraffe2`, we only get a C declaration. OTOH, if we only provide a Sail function but no separate Sail header, as we do for `giraffe3`, we still get a C implementation and a separate prototype.

Note that all the `zgiraffe*` functions are global and can be called from C. This is done for example in the RISCv model, where the Sail functions

- `tick_platform` [https://github.com/riscv/sail-riscv/blob/master/model/riscv\\_platform.sail#L495](https://github.com/riscv/sail-riscv/blob/master/model/riscv_platform.sail#L495)
- `tick_clock` [https://github.com/riscv/sail-riscv/blob/master/model/riscv\\_platform.sail#L319](https://github.com/riscv/sail-riscv/blob/master/model/riscv_platform.sail#L319)

are explicitly called in the handwritten C function

[https://github.com/riscv/sail-riscv/blob/master/c\\_emulator/riscv\\_sim.c#L935-L936](https://github.com/riscv/sail-riscv/blob/master/c_emulator/riscv_sim.c#L935-L936)

Note that if you overload a functions `f1`, ..., `fn` to a new funtion `f` and then call `f` in the Sail code, the generated C will not use `zf` but rather the appropriate `zfi`. For example

```
default Order dec
#include <prelude.sail>

val giraffe1 : unit -> int
function giraffe1 () = {
    return 1
}

function giraffe2 ( n : int ) -> int = {
    return n
}

overload giraffe = { giraffe1, giraffe2 }

val main : unit -> int effect {rreg, wreg}
function main () = {
    let x1 = giraffe() in
    let x2 = giraffe( 17 ) in
    return x2
}
```

results in the following C snippet:

```

void zgiraffe1(sail_int *rop, unit);
void zgiraffe1(sail_int *zcbz30, unit zgsz30) { ... }

void zgiraffe2(sail_int *rop, sail_int);
void zgiraffe2(sail_int *zcbz31, sail_int zn) { ... }

void zmain(sail_int *zcbz32, unit zgsz32)
{
    ...
    zgiraffe1(&z1, UNIT);
    ...
    zgiraffe2(&z2, zgsz33);
    ...
}

```

Scattered definitions (typically used in the decode and execute clauses) might be seen as a form of overloading. Here is an example of a definition of `execute``:

```

default Order dec
$include <prelude.sail>

scattered union ast
val execute : ast -> int

union clause ast = ITYPE : int
function clause execute ITYPE(i) = { return 17 }

union clause ast = BTYPE : bool
function clause execute BTYPE(b) = { return 19 }

union clause ast = RTYPE : real
function clause execute BTYPE(r) = { return 23 }

union clause ast = BVTYPE : bits(32)
function clause execute BTYPE(bv) = { return 29 }

end execute
end ast

```

Here the generated C will contain a single function `zexecute` that does a big `case`-distinction that dispatches to the relevant parts of the scattered definition:

```

void zexecute(sail_int *rop, struct zast);

void zexecute(sail_int *zcbz30, struct zast zmergez3var)
{
    ...
    if (zmergez3var.kind != Kind_zITYPE) goto case_2;
    ...
    CONVERT_OF(sail_int, mach_int)(&zgsz31, INT64_C(17));
    ...
case_2:
    ...
    CONVERT_OF(sail_int, mach_int)(&zgsz33, INT64_C(19));
    ...
case_3:
    ...
    CONVERT_OF(sail_int, mach_int)(&zgsz35, INT64_C(23));
    ...
}

```

**Warning.** The Sail compiler does aggressive dead code elimination: Sail functions, like `giraffe4` which are not used (called) get eliminated and do **not** appear in the generated C code.

Here's another example of using the C foreign language interface...

## 9.1. Sail Calls C function

Here is the sail code where we're trying to call a C function and return a value to Sail.

```
// vim: set tabstop=4 shiftwidth=4 expandtab
// =====
// Filename:    sail_calls_cfunc.sail
//
// Description: Example sail file calling C functions
//
// Author(s):   Bill McSpadden (bill@riscv.org)
//
// Revision:    See git log
// =====

default Order dec
#include <prelude.sail>

type xlenbits : Type = bits(32)

val "print"      : string -> unit
val "print_int"  : int -> unit

val cfunc_int = { c: "cfunc_int" } : bool -> int    // TODO: get rid of bool
val cfunc_str = { c: "cfunc_str" } : bool -> string // TODO: get rid of bool

val main : unit -> unit

function main() =
{
  print("hello, world!\n") ;
  print("hello, another world!\n") ;

  let ret : int = cfunc_int(true);      // TODO: get rid of argument 'true'
  print_int("cfunc_int: ", ret );

  let ret_str : string = cfunc_str(true); // TODO: get rid of argument 'true'
  print("ret_str: ");
  print(ret_str);
  print("\n");

}
```

Here is the C code, in a .c and .h file. The .h file is needed because it needs to be included in the out.c file that Sail generates for the C simulator.

First, the cfunc.h file ....

```
// vim: set tabstop=4 shiftwidth=4 expandtab
// =====
// Filename:    cfunc.h
//
// Description: Functions prototype support for cfunc
//
// Author(s):   Bill McSpadden (bill@riscv.org)
//
// Revision:    See git log
// =====
// #ifndef __CFUNC_H__
// #define __CFUNC_H__
//
//
// #pragma once
//
// #include "sail.h"
//
// #define INT_RET_TYPE    sail_int
// #define INT_RET_TYPE    int
//
// // It doesn't appear that Sail does anything with the
// // function's return value. "return values" are done
// // by passing a pointer to a return value struct, which
// // is the first element in the function's argument list.
// //
// // TODO: make the return value of type void.
//
// INT_RET_TYPE    cfunc_int(sail_int *,    bool);
// void            cfunc_str(sail_string * ,    bool);
//
// #endif
```

And now, cfunc.c, which implements the functions...

```
// vim: set tabstop=4 shiftwidth=4 expandtab
// =====
// Filename:    cfunc.c
//
// Description: Functions to be called by Sail.
//
// Author(s):   Bill McSpadden (bill@riscv.org)
//
// Revision:    See git log
// =====
//
// #include <sail.h>
// #include "cfunc.h"
// #include "string.h"
```

```

INT_RET_TYPE
cfunc_int(sail_int *zret_int, bool foo)
{
//    mpz_set_ui(zret_int, 142);
//    mpz_set_ui(*zret_int, 142);
//    mpz_set_ui(zret_int, 9223372036854775808 );           // 2 ^ 64
// works
//    mpz_set_ui(zret_int, (9223372036854775808 + 1) );       // (2 ^ 64) +
1    // works
//    mpz_set_ui(zret_int, (123456789012345678901234567890) ); // fails:
sail.test prints out incorrect number But the next example works.
//    mpz_init_set_str(*zret_int, "123 456 789 012 345 678 901 234 567 890", 10 ); //
NOTE: white space allowed in string // works

    return(42); // TODO: Nothing is done with this return value, right?
}

```

```

void
cfunc_str(sail_string * zret_str, bool foo)
{
//=====
// The following code .....
//
//    *zret_str = "i'm baaaack...\n";
//
//    return;
//
// ... yields a segmentation fault when killing
// the sail_string variable (pointed to by zret_str)
// in the calling code. The calling code assumes that
// memory has been malloc'd for the string, and when
// it's free'd, you get a seg fault. So, I re-wrote
// the code to do the actual malloc. But note the
// assymetry of the memory management: the space is
// allocated here, but free'd at the calling level.
// This is, at least, ugly code. And, at worst,
// prone to error.
//=====
char * str = "i'm baaaack....\n";
char * s;

s = malloc(strlen(str));
strcpy(s, str);
*zret_str = s;
return;

}

```

```
// TODO: Add many more return types such as...
//      void *
//      struct *
//      float
//      double
//
```

Here is the Makefile used to compile all of this.

```
# vim: set tabstop=4 shiftwidth=4 noexpandtab
# =====
# Filename:      Makefile
#
# Description:   Makefile for building.....
#
# Author(s):     Bill McSpadden (bill@riscv.org)
#
# Revision:      See revision control log
#
# =====

#=====
# Includes
#=====

DEBUG_FLAGS      := -g

#=====
# Make variables
#=====
SAIL_PATH        := /home/billmc/.opam/default
SAIL_BIN         := ${SAIL_PATH}/bin
SAIL_LIB         := ${SAIL_PATH}/lib/sail
SAIL             := ${SAIL_BIN}/sail
SAIL_OUTFILE     := out
SAIL2C_INC       := -c_include cfunc.h
#SAIL_FLAGS      := -c ${SAIL2C_INC} -o ${SAIL_OUTFILE}
SAIL_FLAGS       := ${SAIL2C_INC} -c -o ${SAIL_OUTFILE}

# TODO: fix this. Need to find an installation home for these C files.
#      Perhaps compile a library?
#SAIL_DIR        := /home/billmc/riscv/riscv_sail.git
#SAIL_DIR        := /home/billmc/riscv/riscv_sail__billmcspadden-riscv.git

SAIL_DIR         := ../../..
SAIL_LIB         := ${SAIL_DIR}/lib/sail
SAIL             := ${SAIL_DIR}/sail
SAIL_OUTFILE     := out
```



```

#SAIL_FLAGS      := -c -o ${SAIL_OUTFILE}

SAIL_SRC         := $(wildcard *.sail)

CC               := gcc
CCFLAGS          := ${DEBUG_FLAGS} -lgmp -lz -I ${SAIL_DIR}/lib/ -o out
#C_SRC           := out.c ${SAIL_DIR}/lib/*.c cfunc.c
C_SRC            := cfunc.c out.c ${SAIL_DIR}/lib/*.c

TARGET           := out

#=====
# Targets and Rules
#=====

all: run

build: out

install:

run: out
    ./out

ddd: out
    ddd ./out

out: out.c cfunc.c cfunc.h
    SAIL_DIR=${SAIL_DIR} ; export SAIL_DIR ; \
    gcc ${CCFLAGS} ${C_SRC} -lgmp -lz -I ${SAIL_DIR}/lib -o $@

#   gcc out.c ${SAIL_DIR}/lib/*.c -lgmp -lz -I ${SAIL_DIR}/lib -o $@

out.c: ${SAIL_SRC}
    SAIL_DIR=${SAIL_DIR} ; export SAIL_DIR ; \
    ${SAIL} ${SAIL_FLAGS} $^

# clean:  cleans only local artifacts
clean:
    rm -f out out.c out.ml *.o

```

```
# Cleans local artifacts and the install location  
clean_all:
```

# Chapter 10. Other stuff

## 10.1. Scattered definitions: Why? What are they?

When specifying an ISA, you'd like to coalesce the definitions of an instruction (or a set of instructions, if they have some similarity to each other) into a single file. One benefit of such organizational principles is that you can take a single Sail file and import it into a text specification when describing the instructions without having to tear apart a much larger file.

Functions, unions and mappings are definitions that can be scattered amongst multiple files. Following is an example of scattered definitions of functions, unions and mappings.

Here is the opening of the scattered definitions for this example:

```
// vim: set tabstop=4 shiftwidth=4 expandtab
// =====
// File:    scattered_definitions_begin.sail

scattered function func
scattered function print_enum_to_string
scattered mapping  enum_to_string
```

Here is the top-level Sail module. Note that it calls 2 instances of `func()`, the difference being the "argument" that is passed to it. But it's not really an argument; the "argument" is used to decide which flavor of `func()` should be called.

```
// vim: set tabstop=4 shiftwidth=4 expandtab
// =====
// File:    scattered_definitions.sail

default Order dec

// TODO: $SAIL_DIR/lib/prelude.sail does not contain the function
// 'string_length()'. There may be other functions missing as well
// but I have not investigated the list. I copied over the file
// prelude.sail from the sail-riscv repository and added it to this
// directory in order to resolve the call to string_length(). Where
// does the call to string_length() come from? The example itself
// does not use it directly. It appears to have crept in when I
// added "mapping clause enum_to_string = b_enum_e <-> "b" "
// mappings in a.sail and b.sail.
// $include <prelude.sail>
#include "./prelude.sail"

// Enums must be defined after prelude.sail for some reason.
// Question sent to Alasdair about this on 2022-07-22
//
// This enum must be defined before the function signatures
// in "scattered_definitions_include.sail" else we get a
// compilation error
enum enum_e = a_enum_e | b_enum_e | c_enum_e      // No Compile error
#include "scattered_definitions_include.sail"

scattered function func

function main() =
{
    print("hello, world!\n") ;
    print("calling function 'func'....\n");
    func(a_enum_e);
    func(b_enum_e);
}
```

The file, `scattered_definitions_include.sail`, is used to hold function signatures, which get included in several files.

```
// vim: set tabstop=4 shiftwidth=4 expandtab
// =====
// File:    scattered_definitions_include.sail

// Enums can't be in a general include file in that their definitiosn
// are bound once.
//
//enum enum_e = a_enum_e | b_enum_e | c_enum_e

val "print"          : string    -> unit
val func             : enum_e    -> unit
val print_enum_to_string : enum_e    -> unit
val main             : unit      -> unit
val enum_to_string    : enum_e    <-> string
//val "string_length" : string    -> int
val string_length = "string_length" : string -> nat
```

Here are the 2 scattered definitions for func(), found in 2 different files:

```
// vim: set tabstop=4 shiftwidth=4 expandtab
// =====
// File:    a.sail

//$include <prelude.sail>
//$include "./prelude.sail"
#include "scattered_definitions_include.sail"

function clause func(a_enum_e) =
{
  print("a.sail string: ") ;
  print("\n") ;
}

mapping clause enum_to_string = a_enum_e    <->    "a"
// {                                     // Brackets with a single mapping item gives
compile error
// a_enum_e    <->    "a"
// }

// TODO: Need a method for iterating through a mapping
function clause print_enum_to_string(a_enum_e) =
{
  print("");
}
```

```
// vim: set tabstop=4 shiftwidth=4 expandtab
// =====
// File:    b.sail

//$include <prelude.sail>
//$include "./prelude.sail"
#include "scattered_definitions_include.sail"

function clause func(b_enum_e) =
{
    print("b.sail string: ");
    print("\n");
}

mapping clause enum_to_string = b_enum_e    <->    "b"
// {                                           // Brackets with a single mapping item gives
compile error
// b_enum_e    <->    "b"
// }

// TODO: Need a method for iterating through a mapping
function clause print_enum_to_string(b_enum_e) =
{
    print("");
}

```

And here is where the scattered definition of func() is end'd. The end'ing is broken out into a separate file so that many (all?) scattered definitions can be closed in the same place. This should match the opening of scattered definitions being done in one place.

```
// vim: set tabstop=4 shiftwidth=4 expandtab
// =====
// File:    scattered_definition_end.sail

end func
end print_enum_to_string
end enum_to_string

```

And finally, there is an order of compilation of Sail code that must be observed. The scattered definitions must be compiled last. As such, the Makefile specifies the compilation order as seen here:

```
# vim: set tabstop=4 shiftwidth=4 noexpandtab
# =====

# For this example, Sail compile order is important.
# The ending of scattered definitions appear in scattered_definitions_end.sail,
# so this must be the last file.
SAIL_SRC    := scattered_definitions_begin.sail \
               scattered_definitions.sail \
               a.sail \
               b.sail \
               scattered_definitions_end.sail

include ../Makefile.generic
```

## 10.2. *FILE* , *LINE* , *LOC*

## 10.3. Variable argument list support

What support does Sail have for a variable argument list for its functions?

TBD

# Chapter 11. Description prelude.sail

prelude.sail contains the function signatures and implemenmtations of many support functions.

## 11.1. description of print, sext, equility etc. standard template stuff

## 11.2. the C interface



# Chapter 12. Simple CPU example (Martin)

- From nand2tetris

# Chapter 13. Formal tools that analyze Sail source code

coverage

# Chapter 14. FAQs (Frequently Asked Questions)

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

## 14.1. Frequently Asked Questions about the Sail Programming Language

Q: What are the purposes of "\$\<text\>" constructs, things like `$include`, `$optimize`, etc?

Q: Is there a library methodology for Sail?

Q: RVFI\_DII: What is it?

Q: What does the skid/underscore character, `_`, mean in Sail?

Q: What does *unit* mean in Sail? What is its purpose?

Q: What is the difference between *Int*, *int*, *integer*?

### 14.1.1. Q: What are the purposes of "\$\<text\>" constructs, things like `$include`, `$optimize`, etc?

A: `$<...>` runs what might be called the preprocessor (for directives like `$include <prelude.sail>`). Note that, unlike C, the Sail preprocessor works (recursively) on Sail ASTs rather than strings. Note that such directives that are used are preserved in the AST, so they also function as a useful way to pass auxiliary information to the various Sail backends.

Sail also calls those pragmas. Sail has a few pragmas that can be invoked with `$...`, see

[https://github.com/rem-s-project/sail/blob/sail2/src/process\\_file.ml#L164-L181](https://github.com/rem-s-project/sail/blob/sail2/src/process_file.ml#L164-L181)

Pragmas are useful if you want to extend the existing Sail system. We have some extensions in our internal version of Sail that are using `$...`

"\$\<text\>" is also called a "splice" because it's used to *splice* code in.

### 14.1.2. Q: Is there a library methodology for Sail?

A: Use `$include` for common code

Ideally, Sail would support a proper module system. This would be especially useful for a modular architecture like RISC-V. From a pure Sail language perspective, it is not a problem adding a well-designed module system (like OCaml's) to Sail. However, it's an open problem how to compile such a module system to Coq (IIRC). It's probably a solvable research question but nobody seems to be working on this. So for the time being, we will have to stay with "`$include <...>`"

### 14.1.3. Q: RVFI\_DII: What is it?

A: See <https://github.com/CTSRD-CHERI/TestRIG/blob/master/RVFI-DII.md>

### 14.1.4. Q: What does the skid/underscore character, `_`, mean in Sail?

A: The `_` character is the default pattern match token.

### 14.1.5. Q: What does *unit* mean in Sail? What is its purpose?

A: (From Alasdair Armstrong) *unit* is like *void \** in C.

### 14.1.6. Q: What is the difference between *Int*, *int*, *integer*?

A: (per Alisdair Armstrong) *Int* in the Sail typing system, is a *kind*. A data *kind* has parametricity. Other data *kinds* are *Type*, *Order*, *Bool*.

*int* and *integer* are datatypes. However, they are not fixed length. Sail uses a multiprecision package in order to have varying integer sizes, even greater than 64 bits, or 128 bits. The compute system provides the maximal limit on integer size.

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

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

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

Q: Is there any support for MTIMER?

[`qis_themain_loop__coded_in_Sail`]

Q: Can gdb attach to the RISC-V Golden Model to debug RISC-V code?

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?

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

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

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

### 14.2.1. 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.

#### **14.2.2. 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>

#### **14.2.3. 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 RISC\_V\_config YAML structure, the MTIMER can be assigned any address.

#### **14.2.4. Q: Is the "main loop" coded in Sail?**

A: Yes. The main execution loop can be found in `main.sail`.

#### **14.2.5. Q: Can gdb attach to the RISC\_V Golden Model to debug RISC\_V code?**

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

#### **14.2.6. 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.

#### **14.2.7. 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?

#### **14.2.8. Q: What is the meaning of life, the universe and everything?**

A: 42

#### **14.2.9. 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.