RISC-V:

An Overview of the
Instruction Set Architecture

Harry H. Porter III
Portland State University

HHPorter3@gmail.com

January 26, 2018

The RISC-V project defines and describes a standardized Instruction Set Architecture (ISA). RISC-V is an open-source specification for computer processor architectures, not a particular chip or implementation. To date, several different groups have designed and fabricated silicon implementations of the RISC-V specifications. Based on the performance of these implementations and the growing need for interoperability among vendors, it appears that the RISC-V standard will increase in importance.

This document introduces and explains the RISC-V standard, giving an informal overview of the architecture of a RISC-V processor. This document does not discuss a particular implementation; instead we discuss some of the various design options that are included in the formal RISC-V specification.

This document gives the reader an initial introduction to the RISC-V design.

Date Last Modified: January 26, 2018
Available Online: www.cs.pdx.edu/~harry/riscv

Table of Contents

List of Instructions

Chapter 1: Introduction

Introduction

An Instruction Set Architecture (ISA) defines, describes, and specifies how a particular computer processor core works. The ISA describes the registers and describes each machine-level instruction. The ISA tells exactly what each instruction does and how it is encoded into bits.

The ISA forms the interface between hardware and software. Hardware engineers design digital circuits to implement a given ISA specification. Software engineers write code (operating systems, compilers, etc.) based on a given ISA specification.

There are a number of Instruction Set Architectures in widespread use, for example:

x86-64 (AMD, Intel)
ARM (ARM Holdings)
SPARC (Sun/Oracle)

Each of these ISAs is proprietary and very complex. The details are often obscured in lengthy manuals and some details of the ISA are not made public at all. Furthermore, the widely used ISAs have been around for years and their designs carry baggage as a result, e.g., for backward compatibility. Since these legacy designs were first created, we've learned more about how to design computers. Changes in silicon hardware technology have also had an impact on which design choices are now optimal.

The RISC-V project came out of UC Berkeley to address some of these issues. RISC-V is pronounced "RISC-five".

One goal was to create a modern ISA incorporating the best current ideas in processor design. They strove to create an ISA that was much simpler than the legacy ISAs, but at the same time, was also practical and intended to accommodate really fast hardware implementations.

Another goal was to create a pure Reduced Instruction Set (RISC) architecture. The goal is to be able to execute one instruction per clock cycle and to achieve this, each instruction needs to be simple and limited.

Another goal was to create an open-source ISA. Existing ISAs are proprietary. They are owned, managed, and controlled by corporate entities like Intel, Sun Microsystems, and ARM Holdings. The open-source approach taken by RISC-V means that many different companies can provide hardware implementations of the RISC-V architecture. Creating an ecosystem in which multiple vendors can compete in implementing a single ISA should result in many of the benefits seen in other open-source projects.

The RISC-V design is not a single, fully specified, concrete ISA. Instead, RISC-V is a somewhat generalized specification which can be instantiated or fleshed-out to describe an actual silicon part.

The designers understood that there are many different markets, many different application areas for computers, many different design constraints, and so on. For example, an embedded computer for a dishwasher needs to be cheap, reliable, and simple, but doesn't require speed, support for an operating system, multiple cores, or support for 64-bit operations. On the other hand, other computers will have multiple cores, 64-bit operations, etc.

The RISC-V project approaches this plethora of design choices by introducing a number of options into the ISA. In this respect, RISC-V is really a single Instruction Set Architecture; it is a collection of related ISAs. Perhaps it can be viewed as a menu, from which a particular implementation will choose some items, but not others. The documentation provided by RISC-V is incomplete; an actual processor core will presumably come with documentation saying exactly how and which parts of the RISC-V specification it implements.

For example, the RISC-V specification does not answer these questions:

How wide are the registers?
Options are: 32 bits, 64 bits, and 128 bits
How many registers are there?
Options are: 16 or 32
How long are instructions?
32 bit instructions are mandatory; 16 bit forms are optional

Is hardware multiply/divide included?
Is floating point supported?
Options are: not supported, single, or double precision
Is support for an operating system (i.e., Supervisor Mode) included?
Are page tables supported and, if so, which option is selected?

As a result of all this parameterization, the RISC-V specification is difficult to read. For example, the length of registers is given as a variable XLEN. So instead of saying something like

"…loads a 32 bit value into bits [31:0]"

the documentation says

"…loads an XLEN bit value into bits [XLEN-1:0]".

This document is an attempt to describe RISC-V using a more traditional approach. We proceed by choosing a particular ISA that meets the RISC-V specifications and describe that. To make things more concrete, we will describe the 32 bit RISC-V variant, but make additional comments about the 64-bit and 128-bit variants.

The RISC-V Naming Conventions

As previously described, the RISC-V specification is not a single ISA. Instead, it is a collection of ISA options. A naming convention is used in which a particular ISA variation is given a coded name telling which ISA options are present and supported. A particular hardware (chip) can be described or summarized with such a coded name, indicating which RISC-V features are implemented by the chip.

For example, consider the following RISC-V name:

RV32IMAFD

The "RV" stands for "RISC-V" and all coded names begin with "RV".

The "32" indicates that registers are 32 bits wide. Other options are 64 bits and 128 bits:

RV32	32-bit machines
RV64	64-bit machines
RV128	128-bit machines

The remaining letters have these meanings:

I – Basic integer arithmetic is supported
M – Multiply and divide are supported in hardware
A – The instructions implementing atomic synchronization are supported
F – Single precision (32 bit) floating point is supported
D – Double precision (64 bit) floating point is supported

Each of these is considered to be an "extension" of the base ISA, except for the "I" (basic integer instructions), which is always required.

The letter "G" is used as an abbreviation for "IMAFD":

RV32G = RV32IMAFD

In addition, there are additional variants and extensions:

S – Supervisor mode is implemented
Q – Quad-precision (128 bit) floating point is supported
C – Compressed (i.e., 16 bit) instructions are supported
E – Embedded microprocessors, with only 16 registers

The RISC-V documentation also mentions several additional ISA design extensions, but only says these instructions will be specified at some future date. Presently, there is nothing specific for the following extensions:

L – Decimal arithmetic instructions
V – Vector arithmetic instructions
P – Packed SIMD instructions
B – Bit manipulation instructions
T – Transactional memory support

The approach we take in this document is to describe a RISC-V architecture with the following features:

- 32 registers
- Registers are 32 bits wide
- Multiply and divide instructions are present
- Single and double precision floating point instructions are present

- Atomic instructions are present
- Supervisor mode is supported
- Virtual memory is supported

Rather than specify all variants simultaneously using generalities, we will focus on a specific ISA and then comment on possible variations.

The RISC-V documentation provides an ISA "standard" which can be adopted and used freely by different groups. The standard leaves some decisions open, giving implementers several choices. For example, the implementer is free to choose the size of the registers; the standard mentions 32-bits, 64-bits, and 128-bits but does not mandate a particular choice.

In other areas, the standard is unfinished. For example, mention is made of decimal floating point instructions, but the section discussing them says "to be filled in later".

Also, the RISC-V documentation acknowledges that some implementers will choose to violate the standard or extend the standard. They use the terminology "non-standard extensions" to refer to features that might be present in a given chip, but which do not conform to the RISC-V standard.

Commentary: The RISC-V documentation contains a number of enlightening parenthetical remarks describing the various design choices they considered and offering justifications for the design decisions they made. This level of thoughtfulness is absent in most extant ISAs. In some cases, the RISC-V documentation seems to address the design space itself and offers a language and framework for future ISA development.

The Usual Disclaimer / Request For Corrections

This is a derivative work, based on:

- *The RISC-V Instruction Set Manual, Volume I: User-Level ISA Document* (Version 2.2, May 7, 2017)
- *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture* (Version 1.10, May 7, 2017)

Consult the official documentation, which prevails.

The RISC-V is complex. This document takes liberties and simplifies things. Our goal is to introduce the general design and explain the main ideas. While this may look like a manual or reference work, it is not. To make this material comprehensible…

- Some details are simplified.
- Some statements are not strictly true or complete.
- Some material may simply be incorrect.

This document uses "???" to mark incomplete or questionable information. Please contact the author if you find…

- Inaccurate information that you can correct
- Incomplete information that you can fill in
- Confusing text that needs to be reworded

Document Revision History / Permission to Copy

Version numbers are not used to identify revisions to this document. Instead the date and the author's name is used. The document history is:

Date	Author
January 26, 2018	Harry H. Porter III <Initial version>

In the spirit of RISC-V and the open-source movement, the author grants permission to freely copy and/or modify this document, with the following requirement:

> ***You must not alter this section, except to add to the revision history. You must append your date/name to the revision history.***

You are also free to adapt this document to describe a particular implementation. If you specialize this document to a specific hardware design, you should change the title to reflect this new use. Any material lifted should be referenced.

Basic Terminology

There has been some confusion in computer science documentation regarding abbreviations for large numbers. For example:

4K = ?
4,000
4,096

We use the following prefix notation for large numbers, which is becoming common in the context of computer architecture:

Prefix		Example	Value		
Ki	kibi	KiByte	2^{10}	1,024	$\sim 10^3$
Mi	mebi	MiByte	2^{20}	1,048,576	$\sim 10^6$
Gi	gibi	GiByte	2^{30}	1,073,741,824	$\sim 10^9$
Ti	tebi	TiByte	2^{40}	1,099,511,627,776	$\sim 10^{12}$
Pi	pebi	PiByte	2^{50}	1,125,899,906,842,624	$\sim 10^{15}$
Ei	exbi	EiByte	2^{60}	1,152,921,504,606,846,976	$\sim 10^{18}$

Contrast this to the standard metric prefixes, which we avoid:

Prefix		Example	Value	
K	kilo	KByte	10^3	1,000
M	mega	MByte	10^6	1,000,000
G	giga	GByte	10^9	1,000,000,000
T	tera	TByte	10^{12}	1,000,000,000,000
P	peta	PByte	10^{15}	1,000,000,000,000,000
E	exa	EByte	10^{18}	1,000,000,000,000,000,000

In this document, we use the terms "**byte**", "**halfword**", "**word**", "**doubleword**", and "**quadword**" to refer to various sizes of binary data.

```
           number    number
          of bytes  of bits        example value (in hex)
          ========  =======  ==================================
byte             1        8                      A4
halfword         2       16                    C4F9
word             4       32                  AB12CD34
doubleword       8       64              01234567 89ABCDEF
quadword        16      128      4B6D073A 9A145E40 35D0F241 DE849F03
```

The bits within an 8-bit byte are numbered from 0 (lower, least significant) to 7 (upper, most significant).

```
7654  3210
====  ====
0000  0000
```

The bits within a 16-bit halfword are numbered from 0 to 15.

```
15  12    8       4     0
 ====  ====   ====  ====
 0000  0000   0000  0000
```

The bits within a 32-bit word are numbered from 0 to 31.

```
31  28    24     20    16     12    8      4     0
 ====  ====   ====  ====   ====  ====   ====  ====
 0000  0000   0000  0000   0000  0000   0000  0000
```

Likewise, the bits within a 64-bit doubleword are numbered from 0 to 63 and the bits within a 128-bit quadword are numbered from 0 to 127.

We use the following notation to represent a range of bits:

Example	Meaning
[7:0]	Bits 0 through 7; e.g., all bits in a byte
[31:0]	Bits 0 through 31; e.g., all bits in a word
[31:28]	Bits 28 through 31; e.g., the upper 4 bits in a word
[1:0]	Bits 0 through 1; e.g., the least significant 2 bits

A single hex digit can be used to represent 4 bits (half a byte, sometimes called a “nibble”), as follows:

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

The 8 bits within a byte are conveniently expressed with two hex digits. For example:

8-bit byte	In Hex
1010 0100	A4

The 32 bits in a word are given with 8 hex digits. For example:

32-bit word	In Hex
1010 1011 0001 0010 1100 1101 0011 0100	AB12CD34

Chapter 2: Basic Organization

Main Memory

Main memory is byte addressable. Addresses are 4 bytes (i.e., 32 bits) long, allowing for up to 4 GiBytes to be addressed.

> The RISC-V documentation discusses several different address size options, but we'll start simple with 32 bit addresses.

Memory can be viewed as a sequence of bytes:

```
 address      data
(in hex)    (in hex)
========    ========
00000000       89
00000001       AB
00000002       CD
00000003       EF
00000004       01
00000005       23
00000006       45
00000007       67
  ...          ...
FFFFFFFC       E0
FFFFFFFD       E1
FFFFFFFE       E2
FFFFFFFF       E3
```

"Low" memory refers to smaller memory addresses, which will be shown higher on the page than "high" memory addresses, as in the above example.

Little Endian

The RISC-V is a little endian architecture.

As an example, assume that main memory holds the following bytes:

```
 address      data
(in hex)    (in hex)
========    ========
  ...         ...
E5000004       1A
E5000005       2B
E5000006       3C
E5000007       4D
E5000008       5E
E5000009       6F
E500000A       70
E500000B       81
E500000C       92
E500000D       A3
E500000E       B4
E500000F       C5
  ...         ...
```

Consider loading a registers that is 32 bits (4 bytes) wide. There are several LOAD instructions, which can move either a byte, a halfword, or a word from memory into a register.

Assume that a LOAD instruction loads a byte from address 0xE5000004. The register will then contain:

```
0x0000001A
```

Assume that a LOAD instruction loads a halfword (2 bytes) from the same address. The register will then contain:

```
0x00002B1A
```

Assume that a LOAD instruction loads a word (4 bytes) from the same address. The register will then contain:

```
0x4D3C2B1A
```

Note that we can view the memory as holding a sequence of properly aligned halfwords, which we can naturally represent as follows:

```
 address      data
(in hex)    (in hex)
========    ========
  ...         ...
E5000004      2B1A
E5000006      4D3C
E5000008      6F5E
E500000A      8170
E500000C      A392
E500000E      C5B4
  ...         ...
```

Or we can view this memory as holding a sequence of properly aligned words, which looks like this:

```
 address      data
(in hex)    (in hex)
========    ========
  ...         ...
E5000004    4D3C2B1A
E5000008    81706F5E
E500000C    C5B4A392
  ...         ...
```

Commentary: In a little endian architecture, the order of the bytes is changed whenever data is copied from memory to a register or stored from a register into memory. This can be a source of confusion, particularly when humans look at a printout of memory contents.

Big endian architectures are simpler to understand since the bytes are not reordered during loads and stores.

Notice that with a little endian architecture, the first byte in memory (0x1A in the example) always goes into the same bits in the register, regardless of whether the instruction is moving a byte, halfword, or word. This can result in a simplification of the circuitry.

The spec mentions that they selected little endian since it is commercially dominant. They also mention the possibility of "bi-endian" architectures, which mix big- and little-endianness.

The Registers

For concreteness, this document focuses primarily on RV32, the 32-bit variant of RISC-V. The box below discusses "Other Register Sizes".

The **general purpose registers** are 32 bits (i.e., 4 bytes or one word) in width.

There are **32 registers**.

The registers are named **x0, x1, x2, … x31**.

> **Other Register Sizes:** We mainly focus on describing the RV32 variant of RISC-V, in which the registers are **32 bits** in width.
>
> In the RV64 variant, the registers are **64 bits** (i.e., 8 bytes or a doubleword) in size.
>
> In the RV128 variant, the registers are **128 bits** (i.e., 16 bytes or a quadword) in size.
>
> In all cases, the number of registers is 32.

If floating point is supported, there will be additional **floating point registers** named **f0, f1, f2, … f31**. These registers are distinct from x0, x1, x2, … x31. Floating point registers will be discussed later.

Register x0 is a special "**zero register**". When read, its value is always 0x00000000. Whenever there is an attempt to write to x0, the data is simply discarded.

All other registers are treated identically by the ISA; there is nothing special about any register.

There is no special support for a "stack pointer," a "frame pointer," or a "link register". Although these are important concepts and are used in the implementation of programming languages, the ISA does not have any specialized instructions for them. Any register can be used for these functions.

Reduced Number of General Purpose Registers: In the RISC-V "E" extension, the number of registers is reduced to 16.

The registers are named x0, x1, x2, … x15.

This extension is only applicable to 32-bit machines. In other words, the base ISA will either be RV32I or RV32E, with the only difference being the number of general purpose registers. For 64-bit and 128-bit machines, there will always be a full sized register set with 32 registers, so "E" does not apply to RV64 or RV128.

All instruction formats are the same. In other words, there is no change to the instruction encodings between the normal RV32I and the reduced RV32E variants. The only difference is that the 5-bit fields for encoding a register number are limited to containing only the values of 0 through 15 (in binary: 00000 through 01111).

This extension is meant for simplified, embedded ("E") computers. The spec suggests that the circuitry for a full sized register set can consume 50% of the chip real estate, not counting the space used for cache. Thus, dividing the number of registers in half will result in a 25% savings in real estate.

Compressed Instructions: In the RISC-V "C" extension for compressed instructions, 8 registers (x8, x9, … x15) are easily accessible. To access the other registers, the programmer may have to use a normal, uncompressed instruction. Therefore, compilers are encouraged (but not required) to use only registers x8, x9, … x15.

In this extension, the fact that certain registers will be used as stack pointer, link register, etc. is made use of, so some registers are treated slightly differently here.

We discuss compressed instructions later.

In addition to the general purpose registers, there is also a **program counter (PC)**, whose width matches the width of the general purpose registers.

In the variant we focus on, the PC register is 32 bits wide. In the RV64 and RV128 variants, the size of the PC increases and matches the size of the general purpose registers.

Many processor ISAs include a "status register", sometimes called a "condition code register." Such a register usually contains bits such as:

- Sign / Negative Value
- Zero / Equal
- Carry Bit
- Overflow

In other ISAs, there is usually a COMPARE instruction (which will set bits in the status register) and several BRANCH instructions (which will test the status register bits and conditionally jump).

The RISC-V does not include any such "status register." Instead, the BRANCH instructions in RISC-V will perform both the test and the conditional jump.

Commentary: The normal pattern of most code in other non-RISC-V architectures is to execute a COMPARE instruction and, immediately afterward, execute a BRANCH instruction. They go together and effectively perform a single "test-and-jump" operation. By combining them into a single instruction in the RISC-V architecture, greater performance efficiency can be achieved whenever this "test-and-jump" operation must be performed.

Furthermore, by eliminating the "status register", the processing of interrupts is streamlined. Here's why: The code in any interrupt handler routine will certainly modify the status register, so therefore the status register must be automatically saved whenever an interrupt handler is invoked and restored whenever the handler returns. Additional architectural complexity would be required to support these operations (which must be atomic), but this complexity is avoided with the RISC-V approach.

Control and Status Registers (CSRs)

The **entire state** of a running RISC-V core consists of:

- The integer registers x1, x2, … x31
- The floating point registers, if floating point is supported.
- The Program Counter (PC)
- A set of "Control and Status Registers" (CSRs)

The "**Control and Status Registers**" (CSRs) are used for the protection and privilege system. The privilege system is used by the OS kernel to protect itself and manage user-level processes.

At any moment, the RISC-V processor will be executing either in user-mode or in supervisor-mode. Kernel code is executed in supervisor-mode and application programs are executed in user-mode. [Actually there are two non-user modes; we'll go into details later.]

Each "**Control and Status Register**" (CSR) has a special name and each has a unique function. Reading and/or writing a CSR will have an effect on the processor operation. Reading and writing to the CSRs is used to perform all operations that cannot be performed with normal instructions. The behavior associated with each CSR is often quite complex and each register must be understood independently.

There is a large file of CSRs: there can be up to 4,096 CSRs defined. The CSRs are read and written with just a couple of general-purpose instructions.

A few dozen CSRs are defined in the spec, giving their names, behaviors, and effects. The spec leaves open the possibility of more CSRs being defined and it is clear that there will be significant variation between implementations in the details of which CSRs are implemented and exactly how each one works.

Fortunately, the CSRs can be ignored at first. The CSRs are primarily used by OS code. For example, the CSRs are used for interrupt processing, thread switching, and page table manipulation.

In order understand the user-mode instruction set and to create user-level code, the CSRs can and should be ignored, especially on your first introduction to RISC-V.

Alignment

A "**halfword aligned**" address is an address that is a multiple of 2. The last bit of a halfword-aligned address will always be 0. Likewise, a "**word aligned**" address is a multiple of 4, and ends with the bits 00. Similarly for "doubleword alignment" and "quadword alignment".

A halfword-sized value is said to be "**properly aligned**" if it is stored at a halfword-aligned address. A word-sized value is properly aligned if it is stored at a word-aligned address, and similarly for other sizes.

RISC-V does not require data to be properly aligned for the LOAD and STORE instructions. In other words, any value may be stored at any address. However, proper alignment is good practice and encouraged. In most implementations, LOAD and STORE instructions will perform much faster when the data is properly aligned.

However, there is an alignment requirement for instructions.

Instructions are 32 bits in length and must be stored at word-aligned locations. An attempt to jump to an unaligned addressed will cause an "instruction misaligned" exception.

Commentary: In the RISC-V "C" extension for compressed instructions, halfword-sized instructions are allowed. In this case, the alignment requirement is slightly relaxed: Instructions (whether 16 or 32 bits), must only be halfword aligned, not word aligned.

Since all instructions must be at least halfword aligned, all branch target address and the program counter (PC) values must be even numbers, i.e., must end with a single 0 bit. Some instructions make use of the fact that the least significant bit is always 0, allowing a more efficient encoding of instructions by leaving this final bit implicit.

Chapter 3: Instructions

Instructions

Instructions are 32 bits (4 bytes, 1 word) in length and must be stored at word-aligned memory locations. Instructions for the RV64 and RV128 variants are still 32 bits long.

Some computer architectures are said to employ "two-address" instructions. For example, the following instruction mentions only two registers. (This is not RISC-V.)

```
ADD  x4,x5          # x4 = x4 + x5
```

RISC-V is a "three address" architecture. The ADD instruction looks like this:

```
ADD  x4,x5,x7       # x4 = x5 + x7
```

Note that in the RISC-V assembly language, the destination is typically the leftmost operand.

The Compressed Instruction Extension

This chapter describes the 32-bit instructions. However in this section, we introduce the 16-bit compressed instructions. We will cover the compressed instructions fully in a later chapter.

In the RISC-V "C" extension for compressed instructions, halfword-sized instructions are also allowed. If this extension is implemented, both 16-bit and 32-bit instructions can be used.

The 16-bit and 32-bit instructions can be intermixed. There is no "mode" bit to put the processor into "compressed-instruction" mode, as there is in other processors.

With this extension, the alignment restriction for instructions is relaxed to halfword-alignment.

Each compressed 16-bit instruction is exactly equivalent in function to a 32-bit instruction. However, there are many 32-bit instructions for which there is no equivalent 16-bit version.

Thus, each compressed 16-bit instruction can be thought of as a shorthand for some longer 32-bit instruction. The idea is that the most frequently used instructions have 16-bit versions. By using the shorter versions, the size of program code can be reduced. The RISC-V designers have done research to determine which instructions are the best candidates to be given compressed variants.

Commentary: Reducing the size of code results in increased processor performance since it allows more instructions to be cached, reducing the time to fetch instructions from main memory, which is often a performance bottleneck.

In a typical hardware implementation, when a compressed instruction is fetched and loaded into the Instruction Register (IR) prior to being executed, the hardware will notice that it is a compressed instruction. At that time, the compressed instruction will immediately be expanded from 16 bits into the equivalent 32 bit instruction. Thereafter, there is no need for any additional hardware logic to support the compressed instruction set.

To address 32 registers, 5 bits are required. Consider an instruction which refers to 3 registers, such as:

```
ADD  x8,x9,x10
```

If this instruction is encoded using 5 bits for each register, 15 bits would be consumed. With only 16 bits to work with in compressed instructions, this will clearly not work, since only 1 bit is left for encoding the opcode.

Instead, many of the compressed instructions restrict access to only 8 of the registers. With this restriction, registers can be encoded using only 3 bits.

Many of the compressed instructions only allow access to registers x8, x9, … x15. However, certain other registers are (by convention and habit) used for special purposes, such as for a "stack pointer" (x2) or "link register/return address" (x1), and some compressed instructions implicitly refer to these registers.

Some compressed instructions allow any of the 32 registers to be specified, employing a 5-bit field to encode the register. Obviously, these instructions don't have room for three such register operands.

In general, RISC-V uses three-address instructions. However, some compressed instructions use the two-address approach. For example, the instruction

```
ADD  x8,x8,x10              # Add x10 to x8
```

can be represented in compressed form since the destination register is also an operand.

More precisely, a compressed instruction of the form:

```
C.ADDW   RegD,RegB          # RegD = RegD + RegB
```

can be expanded to

```
ADDW     RegD,RegD,RegB     # Equivalent 32-bit inst
```

In assembly language, compressed instructions are identified with the "C." prefix.

The instruction opcode here ends with a "W" suffix, indicating that it operates on a 32-bit word, as opposed to 64-bit or 128-bit values.

In a typical implementation, a compressed instruction will be expanded to the equivalent 32-bit instruction by the FETCH and DECODE hardware, after an instruction is fetched from memory, directly before it is executed.

However, the expansion to a full-sized 32-bit instruction could be performed by the assembler. This would be useful when assembling for a machine that does not support the compressed extension.

Assembler Note: A sophisticated assembler will automatically generate compressed instructions whenever it can. The idea is that the programmer (or

compiler) will create only 32-bit instructions. Upon encountering a 32-bit instruction that can also be coded as a 16-bit instruction, the assembler will choose the smaller instruction. Such an assembler will relieve programmers (and compilers) from the burden of selecting compressed instructions, although a sophisticated compiler may be able to generate shorter code sequences if it is aware of which instructions can be compressed.

Instruction Encoding

Each instruction is 32-bits in length.

In other words, the ISA is built around a 32-bit instruction size. Each compressed 16-bit instruction can also be represented as a equivalent 32-bit instruction, so the set of 32-bit instruction fully describes the available instructions.

The least significant 2 bits always indicate whether the instruction is 16 or 32 bits, so the processor can immediately tell whether or not it has just loaded a compressed instruction.

Details: A bit pattern ending in 11 indicates a 32-bit instruction. The other bit patterns (00, 01, 10) indicate a compressed instruction. This approach reduces the available number of unique 16-bit instructions by ¼, and reduces the number of effective bits available for the 32-bit instructions to 30 bits: a reasonable compromise.

The distinguishing bits are in the least significant positions, which is appropriate for a little endian architecture.

There is also a provision for instructions that are longer than 32 bits, but no such instructions are specified. Such instructions would be a non-standard extension.

Longer instructions must be a multiple of 16 bits in length. An encoding scheme is specified, which shows how instructions of length 48 bits, 64 bits, etc. are to have their length encoded. Beyond the length encoding, further instructions details are left unspecified and up to the implementers of non-standard devices.

For instructions of length 32 bits, the least significant 2 bits must be 11. In addition, the next 3 bits must not be 111, since this indicates an instruction of length greater than 32-bits.

Compressed (16-bit) Instructions:

```
xxxx xxxx  xxxx xxAA      (where AA ≠ 11)
```

Normal (32-bit) Instructions:

```
xxxx xxxx  xxxx xxxx  xxxx xxxx  xxxB BB11      (where BBB ≠ 111)
```

Longer Instructions:

```
xxxx xxxx  ...  xxxx xxxx  xxxx xxxx  xxx1 1111
```

Consult the official documentation for details for longer instructions.

In this section, we will discuss only 32-bit instructions.

Since there are 32 registers, a field with a width of 5 bits is used to encode each register operand within instructions.

The prototypical instruction mentions 3 registers, which are symbolically called

RegD – The destination
Reg1 – The first operand
Reg2 – The second operand

In addition, a number of instructions contain immediate data. The immediate data value is always sign-extended to yield a 32-bit value.

The following sizes of immediate data values are used:

Notation	Field width	Number of Values	Range of values
Immed-12	12 bits	2^{12} = 4 Ki	-2,048 .. +2,047
Immed-20	20 bits	2^{20} = 1 Mi	- 524,288 .. + 524,287

Commentary: Loading an arbitrary 32 bit value into a register using instructions which are themselves only 32-bits necessarily requires two instructions.

Note that 12+20 equals 32. In RISC-V, an arbitrary 32-bit value can be split into two pieces and loaded into a register with two instructions. The designers have chosen the 12-20 split for a number of reasons. Many common constants and

offsets fall within the range of the smaller immed-12 values. Also, the RISC-V virtual memory supports a page size of 4 KiBytes.

Here are the instruction formats:

R-type instructions:

Operands:

RegD,Reg1,Reg2

Example:

```
ADD   x4,x6,x8      # x4 = x6+x8
```

I-type instructions:

Operands:

RegD,Reg1,Immed-12

Examples:

```
ADDI  x4,x6,123     # x4 = x6+123
LW    x4,8(x6)      # x4 = Mem[8+x6]
```

S-type instructions:

Operands:

Reg1,Reg2,Immed-12

Example:

```
SW    x4,8(x6)        # Mem[8+r6] = x4 (word)
```

B-type instructions (a variant of S-type):

Operands:

Reg1,Reg2,Immed-12

Example:

```
blt   x4,x6,loop      # if x4<x6, goto offset(pc)
```

U-type instructions:

Operands:

RegD,Immed-20

Example:

```
LUI   x4,0x12AB7    # x4 = value<<12
AUIPC x4,0x12AB7    # x4 = (value<<12) + pc
```

J-type instructions (a variant of U-type):

Operands:

RegD,Immed-20

Example:

```
jal   x4,foo        # call: pc=offset+pc; x4=ret addr
```

The only difference between S-type and B-type instructions is how the 12-bit immediate value is handled. In an B-type instruction, the immediate value is

multiplied by 2 (i.e., shifted left 1 bit) before being used. In the S-type instruction, the value is not shifted.

In both cases, sign-extension occurs. In particular, the bits to the left are synthesized by filling them in with a copy of the most significant bit actually present.

S-type immediate values:

Actual value used (where s=sign-extension):

```
ssss ssss ssss ssss ssss vvvv vvvv vvvv
```

Range of values:

-2,048 .. +2,047

0xFFFFF800 .. 0x000007FF

B-type immediate values:

Actual value used (where s=sign-extension):

```
ssss ssss ssss ssss sssv vvvv vvvv vvv0
```

Range of values:

-4,096 .. +4,094 (in multiples of 2)

0xFFFFF000 .. 0x00000FFE

The only difference between U-type and J-type instructions is how the 20-bit immediate value is handled. In a U-type instruction, the immediate value is shifted left by 12 bits to give a 32 bit value. In other words, the immediate value is placed in the uppermost 20 bits, and the lower 12 bits are zero-filled.

In a J-type instruction, the immediate value is shifted left by 1 bit (i.e., multiplied by 2). It is also sign-extended.

U-type immediate values:

Actual value used:

```
vvvv vvvv vvvv vvvv vvvv 0000 0000 0000
```

The value is always aligned to a multiple of 4,096.

J-type immediate values:

Actual value used (where s=sign-extension):

```
ssss ssss sssv vvvv vvvv vvvv vvvv vvv0
```

Range of values:

-1,048,576 .. +1,048,574 (in multiples of 2)

0xFFF00000 .. 0x000FFFFE

Next, we give the encodings for the different types of instructions. In the following, each letter represents a single bit, according to the following legend:

DDDDD = RegD
11111 = Reg1
22222 = Reg2
VVVVV = Immediate value
XXXXX = Op-code / function code

R-type instructions:

Operands:

RegD,Reg1,Reg2

Encoding:

```
XXXX XXX2 2222 1111 1XXX DDDD DXXX XXXX
```

I-type instructions:

Operands:

RegD,Reg1,Immed-12

Encoding:

```
VVVV VVVV VVVV 1111 1XXX DDDD DXXX XXXX
```

S-type and B-type instructions:

Operands:

Reg1,Reg2,Immed-12

Encoding:

```
VVVV VVV2 2222 1111 1XXX VVVV VXXX XXXX
```

U-type and J-type instructions:

Operands:

RegD,Immed-20

Encoding:

```
VVVV VVVV VVVV VVVV VVVV DDDD DXXX XXXX
```

Commentary: Note that Reg1, Reg2, and RegD occur in the same place in all instruction formats.

This simplifies the chip circuitry, which would be more complex if, for example, RegD was sometimes in one part of the instruction and other times in a different place in the instruction.

A consequence of keeping the register fields in that same place is that immediate data values in an instruction are sometimes not always in contiguous bits. And note that the bits encoding the immediate values in the S-type and B-type instructions are not contiguous.

I-type:

```
vvvv vvvv vvvv ---- ---- ---- ---- --
```

S-type and B-type:

```
vvvv vvv- ---- ---- ---- vvvv v--- --
```

U-type and J-type:

```
vvvv vvvv vvvv vvvv vvvv ---- ---- --
```

While both S-type and B-type have a 12-bit immediate value, the precise order of the bits in those fields differs between the two formats. While you would reasonably assume the bits are in order, they are in fact scrambled up a little in the B-type instruction format. Similarly, the bits of the 20-bit immediate value field in the J-type formats are scrambled, as compared to U-type. Consult the spec for details.

Immediate values are always sign-extended. Although the immediate value can be different sizes and may be broken into multiple fields, the sign bit is always in the same instruction bit (namely the leftmost bit, bit 31). This simplifies and speeds sign-extension circuitry.

User-Level Instructions

The RISC-V specification breaks the instructions into two broad categories: "user-level" instructions and "privileged" instructions. We begin by describing the individual user-level instructions. We'll cover the privileged instructions later.

The instruction set is quite small and tightly defined; there are not as many instructions in the RISC-V architecture as in other architectures.

Several familiar instructions (which you would find in other architectures) are simply special cases of more general RISC-V instructions.

The official RISC-V documentation focuses on the actual instructions and mentions special cases only briefly, in connection with the more general instruction. We will separate out some of these special cases and present them as useful instructions in their own right, mentioning that they are actually implemented as special cases of

other instructions. Also, some other useful "instructions" are actually expanded by the assembler into a sequence of two simpler instructions.

The RISC-V standard describe three different machine sizes:

RV32	32-bit registers
RV64	64-bit registers
RV128	128-bit registers

Any particular RISC-V hardware chip will implement only one of the above standards. However, each size is strictly more powerful than the smaller sizes. An RV64 chip will include all the instructions necessary to manipulate 32-bit data, so it can easily do anything an RV32 chip can. Likewise, an RV128 chip is a strict superset of the RV32 and RV64 chips.

You should read the instruction descriptions below assuming that the register width is 32 bits. However, the same descriptions apply and make sense for 64-bit or 128-bit registers, except where specifically noted.

Regarding 64-bit and 128-bit extensions: For the larger machine sizes, the basic 32-bit instructions work identically.

In summary, smaller data values are sign-extended to fit into larger registers. Thus, 32-bit values are sign-extended to 64 bits on an RV64 machine. This means you can run 32-bit code directly on a 64-bit machine with no changes!

Of course, some instructions from a RV64 machine will not be present on a RV32 machine, so code that truly requires 64-bits won't work.

When using a 64-bit machine to execute code written for a 32-bit machine, keep in mind that the upper 32 bits of registers will normally contain the sign extension, and not a zero extension.

Likewise, 32-bit and 64-bit values are sign-extended to 128-bits on an RV128 machine.

The 12-bit and 20-bit immediate values in instructions are sign-extended to the basic machine size. This includes the immediate values in the two U-type instructions (LUI and AUIPC).

For instructions that are not present on all machine sizes, we make special notes. Otherwise, the instruction is present in all variants.

The official documentation does not dwell on the assembler notation. The assembler notation presented here is sometimes a "best guess" of what is accepted by the typical RISC-V assembler tools.

Arithmetic Instructions (ADD, SUB, ...)

Add Immediate

General Form:

```
ADDI    RegD,Reg1,Immed-12
```

Example:

```
ADDI    x4,x9,123     # x4 = x9 + 0x0000007B
```

Description:

The immediate value (a sign-extended 12-bit value, i.e., -2,048 .. +2,047) is added to the contents of Reg1 and the result is placed in RegD.

Comments:

There is no "subtract immediate" instruction because subtraction is equivalent to adding a negative value.

Encoding:

This is an I-type instruction.

Commentary: There is no distinction between signed and unsigned addition; the same hardware yields correct result regardless of whether both values are considered to represent signed values or both are considered to represent unsigned values.

The only distinction between signed and unsigned addition is in how overflow is to be detected. The same is true of subtraction. This is not unique to RISC-V; it is true of all computers.

Overflow is ignored in RISC-V. When overflow occurs, no exceptions are raised and no status bits are set. This satisfies the needs of the "C" language, but may be problematic for languages that mandate overflow detection.

Thus, RISC-V has only one sort of addition; RISC-V does not distinguish between signed and unsigned addition. Likewise, there is no distinction between signed and unsigned subtraction.

Note that overflow detection can be programmed fairly simply. For example, the following code sequence will add two unsigned values and branch on overflow:

```
ADDI    RegD,Reg1,Immed-12
BLTU    RegD,Reg1,overflow    if RegD <U Reg1 then branch
```

The following code sequence will add two signed values and branch on overflow. However, it will only function correctly as long as we know that the immediate value is positive.

```
ADDI    RegD,Reg1,Immed-12
BLT     RegD,Reg1,overflow    if RegD <S Reg1 then branch
```

In the general case of signed addition, you can use the following code sequence, which will require a couple of additional registers:

```
ADD     RegD,Reg1,Reg2
SLTI    Reg3,Reg2,0           Reg3 = (Reg2 <S 0) ? 1 : 0
SLT     Reg4,RegD,Reg1        Reg4 = (RegD <S Reg1) ? 1 : 0
BNE     Reg3,Reg4,overflow    if Reg3 ≠ Reg4 then branch
```

Add Immediate Word

General Form:

```
ADDIW   RegD,Reg1,Immed-12
```

Example:

```
ADDIW   x4,x9,123         # x4 = x9 + 0x0000007B
```

Description:

This instruction is only present in 64-bit and 128-bit machines. The operation is performed using 32-bit arithmetic.

The immediate value (a sign-extended 12-bit value, i.e., -2,048 .. +2,047) is added to the contents of Reg1. The result is then truncated to 32-bits, signed-extended to 64 or 128 bits and placed in RegD.

Encoding:

This is an I-type instruction.

RV32 / RV64 / RV128: The ADDI instruction is present in all machine variations. The ADDI instruction is used to perform 32-bit addition on a 32-bit machine, 64-bit addition on a 64-bit machine, and 128-bit addition on a 128-bit machine.

To perform 32-bit addition on a 64-bit or 128-bit machine, the ADDIW instruction is used.

To perform 64-bit addition on a 128-bit machine, the ADDID instruction is used.

When working with 32-bit data on a 64-bit machine, the upper half of the register will contain nothing but the sign extension of the lower half. This is true regardless of whether the data in the lower 32 bits is to be interpreted as a signed integer, an unsigned integer, or something other than an integer.

To understand how the ADDI instruction differs from ADDIW on a 64-bit machine, consider this example:

```
    0x 0000 0000 0000 0001
  + 0x 0000 0000 7FFF FFFF
  -------------------------
    0x 0000 0000 8000 0000    Result of ADDI

    0x 0000 0000 0000 0001
  + 0x 0000 0000 7FFF FFFF
  -------------------------
    0x FFFF FFFF 8000 0000    Result of ADDIW
```

Add Immediate Double

General Form:

```
    ADDID   RegD,Reg1,Immed-12
```

Example:

```
    ADDID   x4,x9,123       # x4 = x9 + 0x0000007B
```

Description:

This instruction is only present in 128-bit machines. The operation is performed using 64-bit arithmetic.

The immediate value (a sign-extended 12-bit value, i.e., -2,048 .. +2,047) is added to the contents of Reg1. The result is then truncated to 64-bits, signed-extended to 128 bits and placed in RegD.

Encoding:

This is an I-type instruction.

Sign Extension: Consider the problem of sign-extending a 32-bit value to a 64-bit value on a RV64 machine. For example, the 64-bit value:

```
0x 3B4C 204E 92A7 5321
```

should be "coerced" to fit within the range of a 32-bit signed integer, resulting in:

```
0x FFFF FFFF 92A7 5321
```

The ADDIW instruction can do this. Simply adding 0 to the value will produce the desired result.

Likewise, the ADDID instruction can be used to coerce a 128-bit value into a 64-bit value.

Add

General Form:

```
ADD     RegD,Reg1,Reg2
```

Example:

```
ADD     x4,x9,x13       # x4 = x9+x13
```

Description:

The contents of Reg1 is added to the contents of Reg2 and the result is placed in RegD.

Comments:

There is no distinction between signed and unsigned. Overflow is ignored.

Encoding:

This is an R-type instruction.

RV32 / RV64 / RV128: The ADD instruction is present in all machine variations. The ADD instruction is used to perform 32-bit addition on a 32-bit machine, 64-bit addition on a 64-bit machine, and 128-bit addition on a 128-bit machine.

To perform 32-bit addition on a 64-bit or 128-bit machine, the ADDW instruction is used.

To perform 64-bit addition on a 128-bit machine, the ADDD instruction is used.

Add Word

General Form:

```
ADDW    RegD,Reg1,Reg2
```

Example:

```
ADDW    x4,x9,x13      # x4 = x9+x13
```

Description:

The contents of Reg1 is added to the contents of Reg2 and the result is placed in RegD.

RV32 / RV64 / RV128:

This instruction is only present in 64-bit and 128-bit machines. The operation is performed using 32-bit arithmetic.

Comments:

There is no distinction between signed and unsigned. Overflow beyond 32-bits is ignored. The 32-bit result is sign-extended to fill the upper bits of the destination register.

Encoding:

This is an R-type instruction.

Add Double

General Form:

```
ADDD    RegD,Reg1,Reg2
```

Example:

```
ADDD    x4,x9,x13      # x4 = x9+x13
```

Description:

The contents of Reg1 is added to the contents of Reg2 and the result is placed in RegD.

RV32 / RV64 / RV128:

This instruction is only present in 128-bit machines. The operation is performed using 64-bit arithmetic.

Comments:

There is no distinction between signed and unsigned. Overflow beyond 64-bits is ignored. The 64-bit result is sign-extended to fill the upper bits of the destination register.

Encoding:

This is an R-type instruction.

Commentary: The RISC-V authors decided to define many instructions generally, without specifying how many bits they operate on. Instead, the number of bits operated on by the instruction is determined by the register size of the machine.

For example, the ADD instruction performs 32-bit addition on a RV32 machine, 64-bit addition on an RV64 machine, and 128-bit addition on an RV128 machine.

To perform 32-bit addition on a 64-bit machine, a new instruction is included, since the upper 32 bits of the operands must be ignored and the result must have the upper 32 bits filled with the proper sign-extension.

So for the 64-bit machine (RV64), they added a second instruction (ADDW) to add 32-bit quantities. For RV128, they added a third instruction (ADDD) to add 64-bit quantities.

The following table shows how many bits each instruction operates on and highlights a slight lack of orthogonality:

	Size of Machine		
	32-bit	**64-bit**	**128-bit**
Basic Instruction Set:			
ADD	32	64	128
RV64 adds:			
ADDW		32	32
RV128 adds:			
ADDD			64

An alternate approach is the following: (This is not RISC-V.)

Specify that the basic ADD instruction will perform 32-bit addition regardless of register size; for 64-bit and 128-bit machines, this instruction will ignore the upper bits of the operands and sign-extend the result. Then, for RV64 and RV128 machines, a new instruction (not present on RV32 machines) is included to

perform 64 bit addition. Finally, for RV128 machines, a third instruction is included to perform 128 bit addition.

To illustrate this alternate approach, we can make up reasonable names for these hypothetical instructions and show which machines have which instructions with the following table:

	Size of Machine		
	32-bit	**64-bit**	**128-bit**
Basic Instruction Set:			
ADDW	32	32	32
RV64 adds:			
ADDD		64	64
RV128 adds:			
ADDQ			128

These two approaches are equivalent, having the same instructions, which only differ in their names. The names chosen for instructions is an issue orthogonal to how the instructions are encoded. The RISC-V designers selected the first naming scheme to mirror the instruction opcode encoding patterns they chose.

This issue applies to the following instruction groups:

ADDI
ADD
SUBI
SUB
NEG
SLLI
SLL
SRLI
SRL
SRAI
SRA
LD
ST
MUL
DIV
DIVU
REM

REMU

Or more specifically:

RV32	RV64	RV128
ADDI	ADDIW	ADDID
ADD	ADDW	ADDD
SUBI	SUBIW	SUBID
SUB	SUBW	SUBD
NEG	NEGW	NEGD
SLLI	SLLIW	SLLID
SLL	SLLW	SLLD
SRLI	SRLIW	SRLID
SRL	SRLW	SRLD
SRAI	SRAIW	SRAID
SRA	SRAW	SRAD
LD	LDW	LDD
ST	STW	STD
MUL	MULW	MULD
DIV	DIVW	DIVD
DIVU	DIVUW	DIVUD
REM	REMW	REMD
REMU	REMUW	REMUD

Subtract

General Form:

```
SUB     RegD,Reg1,Reg2
```

Example:

```
SUB     x4,x9,x13       # x4 = x9-x13
```

Description:

The contents of Reg2 is subtracted from the contents of Reg1 and the result is placed in RegD.

Comments:

There is no distinction between signed and unsigned. Overflow is ignored.

Encoding:

This is an R-type instruction.

Subtract Word

General Form:

```
SUBW    RegD,Reg1,Reg2
```

Example:

```
SUBW    x4,x9,x13       # x4 = x9-x13
```

Description:

The contents of Reg2 is subtracted from the contents of Reg1 and the result is placed in RegD.

RV32 / RV64 / RV128:

This instruction is only present in 64-bit and 128-bit machines. The operation is performed using 32-bit arithmetic.

Comments:

There is no distinction between signed and unsigned. Overflow beyond 32-bits is ignored. The 32-bit result is sign-extended to fill the upper bits of the destination register.

Encoding:

This is an R-type instruction.

Subtract Double

General Form:

```
SUBD    RegD,Reg1,Reg2
```

Example:

```
SUBD    x4,x9,x13       # x4 = x9-x13
```

Description:

The contents of Reg2 is subtracted from the contents of Reg1 and the result is placed in RegD.

RV32 / RV64 / RV128:

This instruction is only present in 128-bit machines. The operation is performed using 64-bit arithmetic.

Comments:

There is no distinction between signed and unsigned. Overflow beyond 64-bits is ignored. The 64-bit result is sign-extended to fill the upper bits of the destination register.

Encoding:

This is an R-type instruction.

Sign Extend Word to Doubleword

General Form:

```
SEXT.W   RegD,Reg1
```

Example:

```
SEXT.W   x4,x9        # x4 = Sign-extend(x9)
```

Description:

This instruction is only available for 64-bit and 128-bit machines.

The value in the lower 32 bits of Reg1 is signed-extended to 64 or 128 bits and placed in RegD.

Comments:

This instruction is useful when a 32-bit signed value must be "coerced" to a larger value on 64-bit and 128-bit machine.

Encoding:

This is a special case of a more general instruction. This instruction is assembled identically to:

```
ADDIW   RegD,Reg1,0
```

Commentary: The RISC-V documentation uses two distinct suffix notations to denote the size of instructions. For example:

```
ADDIW
SEXT.W
```

Sign Extend Doubleword to Quadword

General Form:

```
SEXT.D   RegD,Reg1
```

Example:

```
SEXT.D   x4,x9        # x4 = Sign-extend(x9)
```

Description:

This instruction is only available for 128-bit machines.

The value in the lower 64 bits of Reg1 is signed-extended to 128 bits and placed in RegD.

Encoding:

This is a special case of a more general instruction. This instruction is assembled identically to:

```
        ADDID   RegD,Reg1,0
```

Negate

General Form:

```
    NEG     RegD,Reg2
```

Example:

```
    NEG     x4,x9           # x4 = -x9
```

Description:

The contents of Reg2 is arithmetically negated and the result is placed in RegD.

Comments:

The result is computed by subtraction from zero. Overflow can only occur when the most negative value is negated. Overflow is ignored.

Encoding:

This is a special case of a more general instruction. This instruction is assembled identically to:

```
        SUB     RegD,x0,Reg2
```

Negate Word

General Form:

```
    NEGW    RegD,Reg2
```

Example:

```
    NEGW    x4,x9           # x4 = -x9
```

Description:

The contents of Reg2 is arithmetically negated and the result is placed in RegD.

RV64 / RV 128:

This instruction is only present in 64-bit and 128-bit machines. The operation is performed using 32-bit arithmetic, whereas the NEG instruction operates on 64-bit or 128-bit quantities in the larger machines.

Encoding:

This is a special case of a more general instruction. This instruction is assembled identically to:

```
        SUBW    RegD,x0,Reg2
```

Negate Doubleword

General Form:

```
NEGD    RegD,Reg2
```

Example:

```
NEGD    x4,x9          # x4 = -x9
```

Description:

The contents of Reg2 is arithmetically negated and the result is placed in RegD.

RV64 / RV 128:

This instruction is only present in 128-bit machines. The operation is performed using 64-bit arithmetic.

Encoding:

This is a special case of a more general instruction. This instruction is assembled identically to:

```
SUBD    RegD,x0,Reg2
```

Logical Instructions (AND, OR, XOR, ...)

And Immediate

General Form:

```
ANDI    RegD,Reg1,Immed-12
```

Example:

```
ANDI    x4,x9,123        # x4 = x9 & 0x0000007B
```

Description:

The immediate value (a sign-extended 12-bit value, i.e., -2,048 .. +2,047) is logically ANDed with the contents of Reg1 and the result is placed in RegD.

Encoding:

This is an I-type instruction.

And

General Form:

```
AND     RegD,Reg1,Reg2
```

Example:

```
AND     x4,x9,x13      # x4 = x9 & x13
```

Description:

The contents of Reg1 is logically ANDed with the contents of Reg2 and the result is placed in RegD.

Encoding:

This is an R-type instruction.

Or Immediate

General Form:

```
ORI    RegD,Reg1,Immed-12
```

Example:

```
ORI    x4,x9,123        # x4 = x9 | 0x0000007B
```

Description:

The immediate value (a sign-extended 12-bit value, i.e., -2,048 .. +2,047) is logically ORed with the contents of Reg1 and the result is placed in RegD.

Encoding:

This is an I-type instruction.

Or

General Form:

```
OR     RegD,Reg1,Reg2
```

Example:

```
OR     x4,x9,x13      # x4 = x9 | x13
```

Description:

The contents of Reg1 is logically ORed with the contents of Reg2 and the result is placed in RegD.

Encoding:

This is an R-type instruction.

Xor Immediate

General Form:

```
XORI    RegD,Reg1,Immed-12
```

Example:

```
XORI    x4,x9,123        # x4 = x9 ^ 0x0000007B
```

Description:

The immediate value (a sign-extended 12-bit value, i.e., -2,048 .. +2,047) is logical XORed with the contents of Reg1 and the result is placed in RegD.

Encoding:

This is an I-type instruction.

Xor

General Form:

```
XOR     RegD,Reg1,Reg2
```

Example:

```
XOR     x4,x9,x13      # x4 = x9 ^ x13
```

Description:

The contents of Reg1 is logically XORed with the contents of Reg2 and the result is placed in RegD.

Encoding:

This is an R-type instruction.

Not

General Form:

```
NOT     RegD,Reg1
```

Example:

```
NOT     x4,x9      # x4 = ~x9
```

Description:

The contents of Reg1 is fetched and each of the bits is flipped. The resulting value is copied into RegD.

Encoding:

This is a special case of a more general instruction. This instruction is assembled identically to:

```
    XORI   RegD,Reg1,-1    # Note that -1 = 0xFFFFFFFF
```

Shifting Instructions (SLL, SRL, SRA, ...)

Shift Left Logical Immediate

General Form:

```
SLLI    RegD,Reg1,Immed-12
```

Example:

```
SLLI    x4,x9,5     # x4 = x9<<5
```

Description:

The immediate value determines the number of bits to shift. The contents of Reg1 is shifted left that many bits and the result is placed in RegD. The shift value is not adjusted, i.e., 0 means no shifting is done. (Apparently all values, including 0, are allowed. ???)

RV64 / RV128:

For 32-bit machines, the shift amount must be within 0..31. For 64-bit machines, the shift amount must be within 0..63. For 128-bit machines, the shift amount must be within 0..127.

Encoding:

This is an I-type instruction.

Shift Left Logical

General Form:

```
SLL    RegD,Reg1,Reg2
```

Example:

```
SLL    x4,x9,x13     # x4 = x9<<x13
```

Description:

Register Reg2 contains the shift amount. The contents of Reg1 is shifted left and the result is placed in RegD.

RV64 / RV128:

For 32-bit machines, the shift amount must be within 0..31. For 64-bit machines, the shift amount must be within 0..63. For 128-bit machines, the shift amount must be within 0..127.

Encoding:

This is an R-type instruction.

Shift Right Logical Immediate

General Form:

```
SRLI    RegD,Reg1,Immed-12
```

Example:

```
SRLI    x4,x9,5     # x4 = x9>>5
```

Description:

The immediate value determines the number of bits to shift. The contents of Reg1 is shifted right that many bits and the result is placed in RegD. The shift is "logical", i.e., zero bits are repeatedly shifted in on the most-significant end.

RV64 / RV128:

For 32-bit machines, the shift amount must be within 0..31. For 64-bit machines, the shift amount must be within 0..63. For 128-bit machines, the shift amount must be within 0..127.

Encoding:

This is an I-type instruction.

Shift Right Logical

General Form:

```
SRL    RegD,Reg1,Reg2
```

Example:

```
SRL    x4,x9,x13     # x4 = x9>>x13
```

Description:

Register Reg2 contains the shift amount. The contents of Reg1 is shifted right and the result is placed in RegD. The shift is "logical", i.e., zero bits are repeatedly shifted in on the most-significant end.

RV64 / RV128:

For 32-bit machines, the shift amount must be within 0..31. For 64-bit machines, the shift amount must be within 0..63. For 128-bit machines, the shift amount must be within 0..127.

Encoding:

This is an R-type instruction.

Shift Right Arithmetic Immediate

General Form:

```
SRAI    RegD,Reg1,Immed-12
```

Example:

```
SRAI    x4,x9,5     # x4 = x9>>>5
```

Description:

The immediate value determines the number of bits to shift. The contents of Reg1 is shifted right that many bits and the result is placed in RegD. The shift is "arithmetic", i.e., the sign bit is repeatedly shifted in on the most-significant end.

RV64 / RV128:

For 32-bit machines, the shift amount must be within 0..31. For 64-bit machines, the shift amount must be within 0..63. For 128-bit machines, the shift amount must be within 0..127.

Encoding:

This is an I-type instruction.

Shift Right Arithmetic

General Form:

```
SRA    RegD,Reg1,Reg2
```

Example:

```
SRA    x4,x9,x13     # x4 = x9>>>x13
```

Description:

Register Reg2 contains the shift amount. The contents of Reg1 is shifted right and the result is placed in RegD. The shift is "arithmetic", i.e., the sign bit is repeatedly shifted in on the most-significant end.

RV64 / RV128:

For 32-bit machines, the shift amount must be within 0..31. For 64-bit machines, the shift amount must be within 0..63. For 128-bit machines, the shift amount must be within 0..127.

Encoding:

This is an R-type instruction.

RV32 / RV64 / RV128: The shift operations defined above work on the entire register, regardless of its size. For a 64-bit machine, they shift all 64 bits; for a 128-bit machine, they shift all 128 bits.

Note that when using 64-bit registers to contain 32-bit values, we cannot simply use 64-bit shift operations. We need some new instructions.

To illustrate the problem, consider the result of right-shifting the following 32-bit value by 5 on a 32-bit machine:

before:	`0x 8000 0000`
shift right logical:	`0x 0400 0000`
shift right arithmetic:	`0x FC00 0000`

What happens if we try to use 64-bit shift instructions?

If the value is represented as a 64-bit sign-extended value, we get the wrong value for the logical shift:

before:	`0x FFFF FFFF 8000 0000`
shift right logical:	`0x 07FF FFFF FC00 0000`
shift right arithmetic:	`0x FFFF FFFF FC00 0000`

On the other hand, if the value is represented as a 64-bit zero-extended value, we get the wrong value for the arithmetic shift:

before:	`0x 0000 0000 8000 0000`
shift right logical:	`0x 0000 0000 0400 0000`
shift right arithmetic:	`0x 0000 0000 0400 0000`

As a consequence, several new shift instructions are needed for 64-bit machines to perform 32-bit shifting correctly. For the RV64 architectural extension, the following instructions are added:

```
SLLIW   RegD,Reg1,Immed-12
SLLW    RegD,Reg1,Reg2
SRLIW   RegD,Reg1,Immed-12
SRLW    RegD,Reg1,Reg2
SRAIW   RegD,Reg1,Immed-12
SRAW    RegD,Reg1,Reg2
```

where the shift amounts are restricted to 5 bits (i.e., 0..31).

These new instructions perform a shift of between 0 and 31 bits and they operate only on the lower order 32 bits, leaving the upper 32 bits unmodified.

For example, when shifting right by 5 we get the correct value in the low-order 32-bits regardless of what is in the upper-bits.

before:	`0x 89AB CDEF 8000 0000`
SRLW (logical):	`0x 89AB CDEF 0400 0000`
SRAW (arithmetic):	`0x 89AB CDEF FC00 0000`

Similarly, when we go to a 128-bit machine, another group of shift instructions is needed to perform 64-bit shifting correctly. For the RV128 architecture, the following instructions are added:

```
SLLID   RegD,Reg1,Immed-12
SLLD    RegD,Reg1,Reg2
SRLID   RegD,Reg1,Immed-12
SRLD    RegD,Reg1,Reg2
SRAID   RegD,Reg1,Immed-12
SRAD    RegD,Reg1,Reg2
```

where the shift amounts are restricted to 6 bits (i.e., 0..63).

These new RV128 instructions perform a shift of between 0 and 63 bits and they operate only on the lower order 64 bits, leaving the upper 64 bits unmodified.

Shift Instructions for RV64 and RV128

General Form:

```
SLLIW   RegD,Reg1,Immed-12   # RV64 and RV128 only
SLLW    RegD,Reg1,Reg2       # RV64 and RV128 only
SRLIW   RegD,Reg1,Immed-12   # RV64 and RV128 only
SRLW    RegD,Reg1,Reg2       # RV64 and RV128 only
SRAIW   RegD,Reg1,Immed-12   # RV64 and RV128 only
SRAW    RegD,Reg1,Reg2       # RV64 and RV128 only
SLLID   RegD,Reg1,Immed-12   # RV128 only
SLLD    RegD,Reg1,Reg2       # RV128 only
SRLID   RegD,Reg1,Immed-12   # RV128 only
SRLD    RegD,Reg1,Reg2       # RV128 only
SRAID   RegD,Reg1,Immed-12   # RV128 only
SRAD    RegD,Reg1,Reg2       # RV128 only
```

Comment:
See above.

Miscellaneous Instructions

Nop

General Form:
```
NOP
```
Example:
```
NOP         # Do nothing
```
Description:
This instruction has no effect.
Comment:
There are several ways to encode the "nop" operation. Using ADDI is the canonical, recommended way. Also note that instructions with the bit encoding of 0x00000000 and 0xFFFFFFFF are specifically not used for "nop". These two values are commonly returned from memory units when the actual memory is missing (i.e., unpopulated). The two values 0x00000000 and 0xFFFFFFFF are specified as "illegal instructions" and will cause an "illegal instruction exception" if fetched and executed.
Encoding:
This is a special case of a more general instruction. This instruction is assembled identically to:
```
ADDI    x0,x0,0
```

Move (Register to Register)

General Form:
```
MV      RegD,Reg1
```
Example:
```
MV      x4,x9       # x4 = x9
```
Description:
The contents of Reg1 is copied into RegD.
Encoding:

This is a special case of a more general instruction. This instruction is assembled identically to:

```
ADDI    RegD,Reg1,0
```

Load Upper Immediate

General Form:

```
LUI     RegD,Immed-20
```

Example:

```
LUI     x4,0x12345    # x4 = 0x12345<<12   (0x12345000)
```

Description:

The instruction contains a 20-bit immediate value. This value is placed in the leftmost (i.e., upper, most significant) 20 bits of the register RegD and the rightmost (i.e., lower, least significant) 12-bits are set to zero.

RV64 / RV128:

The description above applies to 32-bit machines. Regardless of register size, the immediate value is moved into bits 31:12. In the case of 64-bit registers or 128-bit registers, the value is sign-extended to fill the upper 32 or 96 bits (i.e., bits 63:32 or bits 127:32).

Comment:

This instruction is often used directly before an instruction containing a 12-bit immediate value, which will be added in to RegD. Together, they are used to effectively make a 32-bit value. In the case of 64-bit or 128-bit machines, this will be a 32-bit signed value, in the range -2,147,483,648 .. 2,147,483,647 (i.e., -2^{31} .. $2^{31}-1$).

Encoding:

This is a U-type instruction.

Load Immediate

General Form:

```
LI      RegD,Immed-32
```

Example:

```
LI      x4,123          # x4 = 0x0000007B
```

Description:

The immediate value (which can be any 32-bit value) is copied into RegD.

Encoding:

This is a special case of more general instructions and is assembled differently depending on the actual value present.

If the immediate value is in the range of -2,048 .. +2,047, then it can be assembled identically to:

```
ADDI    RegD,x0,Immed
```

If the immediate value is not within the range of -2,048 .. +2,047 but is within the range of a 32-bit number (i.e., -2,147,483,648 .. +2,147,483,647) then it can be assembled using this two-instruction sequence:

```
LUI    RegD,Upper-20
ADDI   RegD,RegD,Lower-12
```

where "Upper-20" represents the uppermost 20 bits of the value and"Lower-12" represents the least significant 12-bits of the value.

Commentary: Notice that the assembler must be careful when computing the "Upper-20" and "Lower-12" pieces from an arbitrary 32-bit value. The assembler cannot simply break the value apart.

Why? Because the immediate 12 bit value will be sign-extended in the second (ADDI) instruction. If the most significant bit of "Lower-12" is 1, then the ADDI instruction will be working with a negative value, causing the incorrect value to be computed.

Instead, the assembler needs to compute this:

Given: Value (a 32-bit quantity)
Compute:
Lower12 = Value[11:0]
x = Value – SignExtend (Lower12)
Upper20 = (x >> 12) [19:0]

The resulting LUI/ADDI sequence will load the correct signed value on 32, 64, and 128 bit machines, since the quantities are signed-extended in both instructions.

Add Upper Immediate to PC

General Form:

```
AUIPC     RegD,Immed-20
```

Example:

```
AUIPC    x4,0x12345   # x4 = PC + (0x12345<<12)
```

Description:

The instruction contains a 20-bit immediate value. This value is moved into the leftmost (i.e., upper, most significant) 20 bits of and the rightmost (i.e., lower, least significant) 12-bits are set to zero. The number so created is then added to the contents of the Program Counter. The result is placed in RegD. The value of the PC used here is the address of the instruction that follows the AUIPC.

RV64 / RV128:

The description above applies to 32-bit machines. In the case of 64-bit or 128-bits registers, the immediate value is sign-extended before being added to the PC. The size of the PC equal to the size of the registers.

Comment:

This instruction is often used directly before an instruction containing a 12-bit immediate value, which will be added in to RegD. Together, they are used to effectively make a 32-bit PC-relative offset. This is adequate to address any location in a 32-bit (4 GiByte) address space.

In the case of 64-bit or 128-bit machines, this will be a 32-bit signed offset, in the range -2,147,483,648 .. 2,147,483,647 (i.e., -2^{31} .. $2^{31}-1$). If the address space is larger than 4 GiBytes, this technique will fail; a different instruction sequence is required.

The current PC can be obtained by using this instruction with an immediate value of zero. Use of JAL to determine the current PC is not recommended.

Encoding:

This is a U-type instruction.

Load Address

General Form:

```
LA      RegD,Address
```

Example:

```
LA      x4,MyVar      # x4 = &MyVar
```

Description:

The address of some memory location is copied into RegD. No access to memory occurs.

Encoding:

There is no actual "load address" instruction; instead the assembler substitutes a sequence of two instructions to achieve the same effect.

The "address" can refer to any location within the 32-bit memory space. The address is converted to a PC-relative address, with an offset of 32 bits. This offset is then broken into two pieces: a 20-bit piece and a 12-bit piece. The instruction is assembled using these two instructions:

```
AUIPC   RegD,Upper-20
ADDI    RegD,RegD,Lower-12
```

Set If Less Than (Signed)

General Form:

```
SLT     RegD,Reg1,Reg2
```

Example:

```
SLT     x4,x9,x13     # x4 = (x9<x13) ? 1 : 0
```

Description:

The contents of Reg1 is compared to the contents of Reg2 using signed comparison. If the value in Reg1 is less than the value in Reg2, the value 1 is stored in RegD. Otherwise, the value 0 is stored in RegD.

Encoding:

This is an R-type instruction.

Set Less Than Immediate (Signed)

General Form:

```
SLTI    RegD,Reg1,Immed-12
```

Example:

```
SLTI    x4,x9,123        # x4 = (x9<0x0000007B) ? 1 : 0
```

Description:

The immediate value (a sign-extended 12-bit value, i.e., -2,048 .. +2,047) is compared to the contents of Reg1 using signed comparison. If the value in Reg1 is less than the immediate value, the value 1 is stored in RegD. Otherwise, the value 0 is stored in RegD.

Encoding:

This is an I-type instruction.

Set If Greater Than (Signed)

General Form:

```
SGT     RegD,Reg1,Reg2
```

Example:

```
SGT     x4,x9,x13     # x4 = (x9>x13) ? 1 : 0
```

Description:

The contents of Reg1 is compared to the contents of Reg2 using signed comparison. If the value in Reg1 is greater than the value in Reg2, the value 1 is stored in RegD. Otherwise, the value 0 is stored in RegD.

Encoding:

This is a special case of a different instruction. This instruction is assembled identically to:

```
SLT   RegD,Reg2,Reg1     # Note: regs are switched
```

Note: There is no "Set If Greater Than Immediate (signed)" instruction.

Set If Less Than (Unsigned)

General Form:

```
SLTU    RegD,Reg1,Reg2
```

Example:

```
SLTU    x4,x9,x13     # x4 = (x9<x13) ? 1 : 0
```

Description:

The contents of Reg1 is compared to the contents of Reg2 using unsigned comparison. If the value in Reg1 is less than the value in Reg2, the value 1 is stored in RegD. Otherwise, the value 0 is stored in RegD.

Encoding:

This is an R-type instruction.

Set Less Than Immediate (Unsigned)

General Form:

```
SLTIU   RegD,Reg1,Immed-12
```

Example:

```
SLTIU   x4,x9,123        # x4 = (x9<0x0000007B) ? 1 : 0
```

Description:

The immediate value (a sign-extended 12-bit value, i.e., -2,048 .. +2,047) is compared to the contents of Reg1 using unsigned comparison. If the value in Reg1 is less than the immediate value, the value 1 is stored in RegD. Otherwise, the value 0 is stored in RegD.

Encoding:

This is an I-type instruction.

Set If Greater Than (Unsigned)

General Form:

```
SGTU    RegD,Reg1,Reg2
```

Example:

```
SGTU    x4,x9,x13    # x4 = (x9>x13) ? 1 : 0
```

Description:

The contents of Reg1 is compared to the contents of Reg2 using unsigned comparison. If the value in Reg1 is greater than the value in Reg2, the value 1 is stored in RegD. Otherwise, the value 0 is stored in RegD.

Encoding:

This is a special case of a different instruction. This instruction is assembled identically to:

```
SLTU  RegD,Reg2,Reg1    # Note: regs are switched
```

Note: There is no "Set If Greater Than Immediate Unsigned" instruction.

Set If Equal To Zero

General Form:

```
SEQZ    RegD,Reg1
```

Example:

```
SEQZ    x4,x9    # x4 = (x9==0) ? 1 : 0
```

Description:

If the value in Reg1 is zero, the value 1 is stored in RegD. Otherwise, the value 0 is stored in RegD.

Comment:

This instruction is implemented with an unsigned comparison against 1. Using unsigned numbers, the only value less than 1 is 0. Therefore if the less-than condition holds, the value in Reg1 must be 0.

Encoding:

This is a special case of a more general instruction. This instruction is assembled identically to:

```
SLTIU    RegD,Reg1,1
```

Set If Not Equal To Zero

General Form:

```
SNEZ    RegD,Reg2
```

Example:

```
SNEZ    x4,x9     # x4 = (x9≠0) ? 1 : 0
```

Description:

If the value in Reg2 is not zero, the value 1 is stored in RegD. Otherwise, the value 0 is stored in RegD.

Comment:

This instruction is implemented with an unsigned comparison against 0. Using unsigned numbers, the only value not less than 0 is 0. Therefore if the less-than condition holds, the value in Reg2 must be not be 0.

Encoding:

This is a special case of a more general instruction. This instruction is assembled identically to:

```
SLTU    RegD,x0,Reg2
```

Set If Less Than Zero (signed)

General Form:

```
SLTZ    RegD,Reg1
```

Example:

```
SLTZ    x4,x9     # x4 = (x9<0) ? 1 : 0
```

Description:

If the value in Reg1 is less than zero (using signed arithmetic), the value 1 is stored in RegD. Otherwise, the value 0 is stored in RegD.

Encoding:

This is a special case of a more general instruction. This instruction is assembled identically to:

```
SLT    RegD,Reg1,x0
```

Set If Greater Than Zero (signed)

General Form:

```
SGTZ    RegD,Reg2
```

Example:

```
SGTZ    x4,x9     # x4 = (x9>0) ? 1 : 0
```

Description:

If the value in Reg2 is greater than zero (using signed arithmetic), the value 1 is stored in RegD. Otherwise, the value 0 is stored in RegD.

Comment:

"Reg2 > 0" is equivalent to "0 < Reg2".

Encoding:

This is a special case of a more general instruction. This instruction is assembled identically to:

```
SLT    RegD,x0,Reg2
```

Branch and Jump Instructions

Branch if Equal

General Form:

```
BEQ     Reg1,Reg2,Immed-12
```

Example:

```
BEQ     x4,x9,MyLabel     # If x4==x9 goto MyLabel
```

Description:

The contents of Reg1 is compared to the contents of Reg2. If equal, control jumps. The target address is given as a PC-relative offset. More precisely, the offset is sign-extended, multiplied by 2, and added to the value of the PC. The value of the PC used is the address of the instruction following the branch, not the branch itself (???). The offset is multiplied by 2, since all instructions must be halfword aligned. This gives an effective range of -4,096 .. 4,094 (in multiples of 2), relative to the PC.

Comment:

Most conditional branches occur within smallish subroutines/functions, so the limited range should be adequate.

If the limited size of the offset is inadequate, code such as the following:

```
        BEQ   x4,x9,MyLabel
```

must be altered to:

```
        BNE   x4,x9,Skip
        J     MyLabel      # a "long jump"
    Skip:
```

This applies to all conditional branch instructions.

Encoding:

This is a B-type instruction.

Commentary: Instruction sizes, locations, and relative offsets are determined by the assembler and linker, not by the compiler. However, the compiler (which selects instructions and produces the assembly code sequence) does not have enough knowledge to determine whether an offset will be within range. Therefore, it cannot know whether or not a "long jump" is necessary.

What is the solution?

(1) The compiler always errs on the safe side and generates many unnecessary "long jumps." Not acceptable.

(2) The compiler ignores the problem, always uses short jumps, and occasionally generates code that will not assemble without further attention. Not acceptable.

(3) The compiler is modified to count instructions and determine when "long jumps" should be inserted. This puts a burden on the compiler, but the compiler may already be keeping track of the lengths of the code sequences it generates, in order to compare alternatives. In order for this to work, the assembler must be forbidden from doing things like compiling the LI instruction conditionally, as described earlier.

(4) The assembler detects offset overflows and quietly alters the code sequence by reversing the sense of the test and inserting a "long jump" instruction. This may be the easiest, but violates the traditional one-to-one mapping between assembler and machine code.

Branch if Not Equal

General Form:

```
BNE     Reg1,Reg2,Immed-12
```

Example:

```
BNE     x4,x9,MyLabel     # If x4≠x9 goto MyLabel
```

Description:

The contents of Reg1 is compared to the contents of Reg2. If not equal, control jumps to a PC-relative target address.

Comment:

The target location given by the offset must be within the range of -4,096 .. 4,094 (in multiples of 2), relative to the PC. See the "BEQ" instruction.

Encoding:

This is a B-type instruction.

Branch if Less Than (Signed)

General Form:

```
BLT     Reg1,Reg2,Immed-12
```

Example:

```
BLT     x4,x9,MyLabel     # If x4<x9 goto MyLabel
```

Description:

The contents of Reg1 is compared to the contents of Reg2. If Reg1 is less than Reg2 (using signed comparison), control jumps to a PC-relative target address.

Comment:

The target location given by the offset must be within the range of -4,096 .. 4,094 (in multiples of 2), relative to the PC. See the "BEQ" instruction.

Encoding:

This is a B-type instruction.

Branch if Less Than Or Equal (Signed)

General Form:

```
BLE     Reg1,Reg2,Immed-12
```

Example:

```
BLE     x4,x9,MyLabel     # If x4<=x9 goto MyLabel
```

Description:

The contents of Reg1 is compared to the contents of Reg2. If Reg1 is less than or equal to Reg2 (using signed comparison), control jumps to a PC-relative target address.

Comment:

The target location given by the offset must be within the range of -4,096 .. 4,094 (in multiples of 2), relative to the PC. See the "BEQ" instruction.

Encoding:

This is a special case of another instruction. This instruction is assembled identically to:

```
BGE   Reg2,Reg1,Immed-12  # Note: regs are swapped
```

Branch if Greater Than (Signed)

General Form:

```
BGT   Reg1,Reg2,Immed-12
```

Example:

```
BGT   x4,x9,MyLabel    # If x4>x9 goto MyLabel
```

Description:

The contents of Reg1 is compared to the contents of Reg2. If Reg1 is greater than Reg2 (using signed comparison), control jumps to a PC-relative target address.

Comment:

The target location given by the offset must be within the range of -4,096 .. 4,094 (in multiples of 2), relative to the PC. See the "BEQ" instruction.

Encoding:

This is a special case of another instruction. This instruction is assembled identically to:

```
BLT   Reg2,Reg1,Immed-12  # Note: regs are swapped
```

Branch if Greater Than Or Equal (Signed)

General Form:

```
BGE   Reg1,Reg2,Immed-12
```

Example:

```
BGE   x4,x9,MyLabel    # If x4>=x9 goto MyLabel
```

Description:

The contents of Reg1 is compared to the contents of Reg2. If Reg1 is greater than or equal to Reg2 (using signed comparison), control jumps to a PC-relative target address.

Comment:

The target location given by the offset must be within the range of -4,096 .. 4,094 (in multiples of 2), relative to the PC. See the "BEQ" instruction.

Encoding:

This is a B-type instruction.

Branch if Less Than (Unsigned)

General Form:

```
BLTU    Reg1,Reg2,Immed-12
```

Example:

```
BLTU    x4,x9,MyLabel    # If x4<x9 goto MyLabel
```

Description:

The contents of Reg1 is compared to the contents of Reg2. If Reg1 is less than Reg2 (using unsigned comparison), control jumps to a PC-relative target address.

Comment:

The target location given by the offset must be within the range of -4,096 .. 4,094 (in multiples of 2), relative to the PC. See the "BEQ" instruction.

Encoding:

This is a B-type instruction.

Branch if Less Than Or Equal (Unsigned)

General Form:

```
BLEU    Reg1,Reg2,Immed-12
```

Example:

```
BLEU    x4,x9,MyLabel    # If x4<=x9 goto MyLabel
```

Description:

The contents of Reg1 is compared to the contents of Reg2. If Reg1 is less than or equal to Reg2 (using unsigned comparison), control jumps to a PC-relative target address.

Comment:

The target location given by the offset must be within the range of -4,096 .. 4,094 (in multiples of 2), relative to the PC. See the "BEQ" instruction.

Encoding:

This is a special case of another instruction. This instruction is assembled identically to:

```
BGEU  Reg2,Reg1,Immed-12   # Note: regs are swapped
```

Branch if Greater Than (Unsigned)

General Form:

```
BGTU    Reg1,Reg2,Immed-12
```

Example:

```
BGTU    x4,x9,MyLabel    # If x4>x9 goto MyLabel
```

Description:

The contents of Reg1 is compared to the contents of Reg2. If Reg1 is greater than Reg2 (using unsigned comparison), control jumps to a PC-relative target address.

Comment:

The target location given by the offset must be within the range of -4,096 .. 4,094 (in multiples of 2), relative to the PC. See the "BEQ" instruction.

Encoding:

This is a special case of another instruction. This instruction is assembled identically to:

```
BLTU  Reg2,Reg1,Immed-12  # Note: regs are swapped
```

Branch if Greater Than Or Equal (Unsigned)

General Form:

```
BGEU    Reg1,Reg2,Immed-12
```

Example:

```
BGEU    x4,x9,MyLabel    # If x4>=x9 goto MyLabel
```

Description:

The contents of Reg1 is compared to the contents of Reg2. If Reg1 is greater than or equal to Reg2 (using unsigned comparison), control jumps to a PC-relative target address.

Comment:

The target location given by the offset must be within the range of -4,096 .. 4,094 (in multiples of 2), relative to the PC. See the "BEQ" instruction.

Encoding:

This is a B-type instruction.

Comparisons to Zero: The following abbreviations are simple variations on the previous instructions. They compare a value in a register to zero using signed comparison. The zero register (x0) is implicit in these forms.

Abbreviated Form:	Assembled As:
`BEQZ Reg1,target`	`BEQ Reg1,x0,target`
`BNEZ Reg1,target`	`BNE Reg1,x0,target`
`BLTZ Reg1,target`	`BLT Reg1,x0,target`
`BLEZ Reg1,target`	`BGE x0,Reg1,target`
`BGTZ Reg1,target`	`BLT x0,Reg1,target`
`BGEZ Reg1,target`	`BGE Reg1,x0,target`

Commentary: The RISC-V architecture does not include conditionally executed instructions. The idea with a conditionally executed instruction is that the instruction execution is predicated on some condition code(s). The instruction is executed, and will either work normally or have no effect, depending on whether the condition is true or false. The benefit is that conditional branch instructions (with their pipeline issues) can sometimes be avoided, resulting in improved efficiency. The RISC-V designers considered and rejected conditional execution.

Jump And Link (Short-Distance CALL)

General Form:

```
JAL     RegD,Immed-20
```

Example:

```
JAL     x1,MyFunct     # Goto MyFunct, x1=RetAddr
```

Description:

This instruction is used to call a subroutine (i.e., function). The return address (i.e., the PC, which is the address of the instruction following the JAL) is saved in RegD.

The target address is given as a PC-relative offset. More precisely, the offset is sign-extended, multiplied by 2, and added to the value of the PC. The value of the PC used is the address of the instruction following the JAL, not the JAL itself (???). The offset is multiplied by 2, since all instructions must be halfword aligned. This gives an effective range of ±1 MiByte, i.e., -1,048,576 .. 1,048,574 (in multiples of 2), relative to the PC.

Assembler Shorthand:

By convention, x1 is generally used as the "link register". If the register is not mentioned, then x1 is implied.

```
JAL MyFunct          # Call MyFunct
```

Comment:

The programming convention is to use register x1 as a "link register." This return address is stored in x1, rather than being pushed onto a stack in memory, thus making calls and returns faster. However, if some routine "foo" will call another routine "bar", then "foo" must save x1 before calling "bar". The "foo" routine might push x1 onto a stack in memory, or "foo" might save x1 in a "callee-saved" register in the hope of avoiding any memory access.

Exceptions:

This may generate an "instruction misaligned exception." The target address will necessarily be a multiple of 2, but it may not be multiple of 4. For machines that do not support 16-bit instructions, this will cause an alignment exception.

Encoding:

This is a J-type instruction.

Commentary: Most programs code segments are fairly small, so the limited range of JAL should be adequate.

If the program size exceeds 1 MiByte, the limited size of the offset may be inadequate. Code such as the following:

```
JAL    x1,MyFunct
```

must be altered to:

```
LUI    Reg2,Offset-20
JALR   x1,Reg2,Offset-12
```

Typically several routines are separately compiled so the compiler will not know how far away the target function is from the CALL instruction. Even the assembler will not know and only at link-time can it be determined whether a 20-bit offset is adequate. One approach is for the linker to print out an inscrutable error message.

Jump (Short-Distance)

General Form:

```
J       Immed-20
```

Example:

```
J       MyLabel     # Goto MyLabel
```

Description:

The target address is given as a PC-relative offset. The effective range is ±1 MiByte, i.e., -1,048,576 .. 1,048,574 (in multiples of 2), relative to the PC.

Exceptions:

This may generate an "instruction misaligned exception." The target address will necessarily be a multiple of 2, but it may not be multiple of 4. For machines that do not support 16-bit instructions, this will cause an alignment exception.

Encoding:

This is a special case of another instruction. This instruction is assembled identically to:

```
JAL    x0,Immed-20      # Discard return address
```

Jump And Link Register

General Form:

```
JALR    RegD,Reg1,Immed-12
```

Example:

```
JALR    x1,x4,MyFunct     # Goto MyFunct, x1=RetAddr
```

Description:

This instruction is used to call a subroutine (i.e., function). The return address (i.e., the PC, which is the address of the instruction following the JAL) is saved in RegD.

The target address is computed by adding the offset to the contents of Reg1. More precisely, the offset is sign-extended and added to the value of Reg1. The

offset is not multiplied by 2. This gives an effective range of ±2 KiByte, i.e., -2,048 .. 2,047, relative to the address in Reg1.

Comment:

This instruction can be used in several ways. See the JR, RET, CALL, and TAIL instructions.

Assembler Shorthand:

By convention, x1 is used as the "link register". The following form:

```
JALR    Reg1            # Call *Reg1
```

is used short hand for:

```
JALR    x0,Reg1,0
```

Exceptions:

This may generate an "instruction misaligned exception."

Encoding:

This is an I-type instruction.

Jump Register

General Form:

```
JR      Reg1
```

Example:

```
JR      Reg1    # Goto *Reg1, i.e., PC = Reg1
```

Description:

Jump to the address in Reg1.

Exceptions:

This may generate an "instruction misaligned exception."

Encoding:

This is a special case of another instruction. This instruction is assembled identically to:

```
JALR   x0,Reg1,0    # Discard ret addr; offset=0
```

Return

General Form:

```
RET
```

Example:

```
RET             # Goto *x1, i.e., PC = x1
```

Description:

By convention, x1 is used as the "link register" and will hold a return address. This instruction returns from a subroutine/function.

Exceptions:

This may generate an "instruction misaligned exception."

Encoding:

This is a special case of another instruction. This instruction is assembled identically to:

```
JALR    x0,x1,0     # PC=x1+0; don't save prev PC
```

Call Faraway Subroutine

General Form:

```
CALL     Immed-32
```

Example:

```
CALL     MyFunct        # PC = new addr; x1 = ret addr
```

Description:

By convention, x1 is used as the "link register" and will hold a return address. This instruction calls a subroutine/function using a PC-relative scheme, where the subroutine offset from the CALL instruction exceeds the 20-bit limit (i.e., ±1 MiByte) of the JAL instruction. This instruction modifies register x6.

Encoding:

In order to deal with the larger distance to the subroutine, this "synthetic" instruction will be assembled using the following two-instruction sequence. The target address can be expressed as a 32-bit offset from the Program Counter. This offset is broken into two pieces, which are added to the PC in two steps.

```
AUIPC   x6,Immed-20
JALR    x1,x6,Immed-12
```

The AUIPC instruction adds the PC to the upper 20-bit portion of the 32-bit offset and places the result in a temporary register. The JALR instruction adds in the lower 12 bits of the 32-bit offset and transfers control by loading the sum into the PC. It also saves the return address in x1.

The CALL instruction makes use of the convention that x1 is the link register. It also uses x6, which is a "caller-saved temporary register" by convention.

Commentary: The actual values filled into the instructions (in the Immed-20 and Immed-12 fields) must be computed by the linker, since the actual target address will, in general, not be available to the assembler. The linker must convert the target address into a 32-bit offset from the Program Counter. At runtime, the PC value used will be the address of the AUIPC (not the JALR instruction) so the value must be relative to that address.

Also note that the 12-bit immediate value will be sign extended in the JALR instruction, so simply breaking the offset apart into pieces will not work. Instead, the linker must first isolate the low-order 12 bits and sign-extend it. Then the linker must subtract it from the full-32-bit offset, yielding the upper Immed-20 portion.

This applies to the TAIL instruction, too.

Exceptions:

This sequence may generate an "instruction misaligned exception" if the target offset from which the immediate values are computed is not word aligned and the processor does not support 16-bit instructions.

Tail Recursion: Some subroutines/functions end by calling another function. For example, the last statement in a function named "foo" may be to invoke/call a function named "bar." When the called function (bar) returns, the original function (foo) immediately completes and returns. Any value returned by the called function (bar) will be used as the return value from the caller (foo).

This pattern often occurs in functional programs, where recursion is used to implement repetitive looping behavior. Without special attention, some functional programs that use recursion in this way can create very deep stacks by pushing thousands of return addresses onto the stack.

A common optimization is called the "**tail recursion optimization**". Here is how it works. The compiled code for the caller (foo) is modified to not actually call bar, but instead to simply jump to the first instruction in bar. Then, when bar returns, the processor will effectively return from foo, since no other return address was saved. Furthermore, anything returned by bar will be naturally be returned to foo's caller. The tail recursion optimization effectively turns functional programs which use recursion to perform looping activities into instruction sequences that use

"goto" instructions. It coverts recursive programs into looping programs with "goto" instructions, making recursive programming techniques practical.

Quite often a recursive function will call itself. In such cases of the tail recursion optimization, the "goto" need not be a long-distance jump and the J instruction will suffice. But in other cases, mutual recursion might involve separately compiled functions and require the long-distance jumping ability of the TAIL instruction.

Tail Call (Faraway Subroutine) / Long-Distance Jump

General Form:

```
TAIL    Immed-32
```

Example:

```
TAIL    MyFunct     # PC = new addr; Discard ret addr
```

Description:

This instruction is used to jump to a distant location using a PC-relative scheme, where the displacement from the TAIL instruction to the target instruction exceeds the 20-bit limit (i.e., ±1 MiByte) of the J instruction (jump short distance). This instruction modifies register x6.

Comment:

This instruction is nothing more than a long-distance "goto" instruction; the name "TAIL" may be confusing, but reflects how it will sometimes be used.

Encoding:

This "synthetic" instruction will be assembled using the following two-instruction sequence.

```
AUIPC   x6,Immed-20
JALR    x0,x6,Immed-12
```

See the comments for the CALL instruction. The only difference is that here the return address is discarded (x0), instead of being saved in x1.

Exceptions:

Like the CALL instruction, this sequence may generate an "instruction misaligned exception."

Load and Store Instructions

Load Byte (Signed)

General Form:

```
LB      RegD,Immed-12(Reg1)
```

Example:

```
LB      x4,1234(x9)     # x4 = Mem[x9+1234]
```

Description:

An 8-bit value is fetched from memory and moved into register RegD. The memory address is formed by adding the offset to the contents of Reg1. The value is sign-extended to the full length of the register.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

There is no alignment issue and this instruction will execute atomically.

Encoding:

This is an I-type instruction.

Load Byte (Unsigned)

General Form:

```
LBU     RegD,Immed-12(Reg1)
```

Example:

```
LBU     x4,1234(x9)     # x4 = Mem[x9+1234]
```

Description:

An 8-bit value is fetched from memory and moved into register RegD. The memory address is formed by adding the offset to the contents of Reg1. The value is zero-extended to the full length of the register.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

There is no alignment issue and this instruction will execute atomically.

Encoding:

This is an I-type instruction.

Load Halfword (Signed)

General Form:

```
LH      RegD,Immed-12(Reg1)
```

Example:

```
LH      x4,1234(x9)     # x4 = Mem[x9+1234]
```

Description:

A 16-bit value is fetched from memory and moved into register RegD. The memory address is formed by adding the offset to the contents of Reg1. The value is sign-extended to the full length of the register.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. halfword-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

Encoding:

This is an I-type instruction.

Load Halfword (Unsigned)

General Form:

```
LHU     RegD,Immed-12(Reg1)
```

Example:

```
LHU     x4,1234(x9)     # x4 = Mem[x9+1234]
```

Description:

A 16-bit value is fetched from memory and moved into register RegD. The memory address is formed by adding the offset to the contents of Reg1. The value is zero-extended to the full length of the register.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. word-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

Encoding:

This is an I-type instruction.

Load Word (Signed)

General Form:

```
LW      RegD,Immed-12(Reg1)
```

Example:

```
LW      x4,1234(x9)     # x4 = Mem[x9+1234]
```

Description:

A 32-bit value is fetched from memory and moved into register RegD. The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. word-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

RV64 / RV128:

For a machine with a register width larger than 32-bits, the value is sign-extended to the full length of the register.

Encoding:

This is an I-type instruction.

Load Word (Unsigned)

General Form:

```
LWU     RegD,Immed-12(Reg1)
```

Example:

```
LWU     x4,1234(x9)     # x4 = Mem[x9+1234]
```

Description:

This instruction is only available for 64-bit and 128-bit machines.

A 32-bit value is fetched from memory and moved into register RegD. The value is zero-extended to the full length of the register. The memory address is formed by adding the offset to the contents of Reg1.

In a machine with 32-bit registers, neither sign-extension nor zero-extension is necessary for value that is already 32 bits wide. Therefore the "signed load" instruction (LW) does the same thing as the "unsigned load" instruction (LWU), making LWU redundant.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. word-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

Encoding:

This is an I-type instruction.

Load Doubleword (Signed)

General Form:

```
LD      RegD,Immed-12(Reg1)
```

Example:

```
LD      x4,1234(x9)     # x4 = Mem[x9+1234
```

Description:

This instruction is only available for 64-bit and 128-bit machines.

A 64-bit value is fetched from memory and moved into register RegD. The memory address is formed by adding the offset to the contents of Reg1. Sign extension occurs for machines with 128-bit registers.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. word-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

Encoding:

This is an I-type instruction.

Load Doubleword (Unsigned)

General Form:

```
LDU     RegD,Immed-12(Reg1)
```

Example:

```
LDU     x4,1234(x9)     # x4 = Mem[x9+1234]
```

Description:

This instruction is only available for 128-bit machines.

A 64-bit value is fetched from memory and moved into register RegD. The value is zero-extended to 128-bits. The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. word-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

Encoding:

This is an I-type instruction.

Load Quadword

General Form:

```
LQ      RegD,Immed-12(Reg1)
```

Example:

```
LQ      x4,1234(x9)     # x4 = Mem[x9+1234]
```

Description:

This instruction is only available for 128-bit machines.

A 128-bit value is fetched from memory and moved into register RegD. The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. word-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

Encoding:

This is an I-type instruction.

Store Byte

General Form:

```
SB      Reg2,Immed-12(Reg1)
```

Example:

```
SB      x4,1234(x9)     # Mem[x9+1234] = x4
```

Description:

An 8-bit value is copied from register Reg2 to memory. The upper (more significant) bits in Reg2 are ignored. The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

There is no alignment issue and this instruction will execute atomically.

Encoding:

This is an S-type instruction.

Store Halfword

General Form:

```
SH      Reg2,Immed-12(Reg1)
```

Example:

```
SH      x4,1234(x9)     # Mem[x9+1234] = x4
```

Description:

A 16-bit value is copied from register Reg2 to memory. The upper (more significant) bits in Reg2 are ignored. The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. halfword-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

Encoding:

This is an S-type instruction.

Store Word

General Form:

```
SW      Reg2,Immed-12(Reg1)
```

Example:

```
SW      x4,1234(x9)     # Mem[x9+1234] = x4
```

Description:

A 32-bit value is copied from register Reg2 to memory. The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. word-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

RV64 / RV128:

For a machine with a register width larger than 32-bits, the upper bits of the register are ignored.

Encoding:

This is an S-type instruction.

Store Doubleword

General Form:

```
SD      Reg2,Immed-12(Reg1)
```

Example:

```
SD      x4,1234(x9)     # Mem[x9+1234] = x4
```

Description:

This instruction is only available in 64-bit and 128-bit machines.

A 64-bit value is copied from register Reg2 to memory. The memory address is formed by adding the offset to the contents of Reg1. For a 128-bit machine the upper bits of the register are ignored.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. doubleword-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

Encoding:

This is an S-type instruction.

Store Quadword

General Form:

```
SQ      Reg2,Immed-12(Reg1)
```

Example:

```
SQ      x4,1234(x9)     # Mem[x9+1234] = x4
```

Description:

This instruction is only available in 128-bit machines.

A 128-bit value is copied from register Reg2 to memory. The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. word-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

Encoding:

This is an S-type instruction.

Commentary: For the "store" instructions, the RISC-V documentation does not make it clear whether the address is formed using Reg1 or Reg2, nor whether the source register is Reg1 or Reg2. However, based on the assumption that the same register will be used for address calculation in both "load" and "store" instructions and on comments elsewhere, we conclude the order is what is shown here.

Commentary: The RISC-V documentation suggests that assembler expects the memory address operand to be the second operand, not the first. This violates the general pattern in assembly code that data movement is toward the left, i.e., from the second operand to the first.

Commentary: By convention, register x2 is used as a stack top pointer (SP). However, the RISC-V architecture does not contain any special instructions for manipulating the stack; register x2 is treated uniformly to all other registers by the ISA, even though the assembler may support some shorthand notations that facilitate use x2 as a stack pointer.

Compilers for traditional programming languages typically place local variables in a stack frame (i.e., an activation record) which is conveniently accessed using the stack top pointer. To read and write such local variables, the "load" and "store" instructions will naturally use offsets from x2:

```
LB   x5,offset(x2)   # Load from variable
SB   x5,offset(x2)   # Store into variable
ADDI x2,x2,4         # Pop 4 bytes off stack
LW   x5,0(x2)        # Fetch word on stack top
```

Assuming that stack frames tend to have modest sizes, the RISC-V instruction set, which utilizes 12-bit offsets (-2,048 .. +2,047), works well.

For global (i.e., shared or common) variables, one approach is to keep a single pointer to a block of global variables. By convention, register x3 is often used as such a "global base pointer." Again, assuming the amount of memory required for the global variables is modest, the 12-bit offset works well.

In cases where the desired offsets exceed the 12-bit limit, the "load upper immediate" instruction (LUI) comes to the rescue. Recall that LUI takes a 20-bit immediate value, shifts it left 12 bits, and moves it into a register.

As an example, the following instruction sequence can be used to load a value from a location offset from a base pointer (in this case x2), when the offset exceeds the 12-bit limit.

```
LUI   x5,Immed-20      # Grab the upper 20 bits
ADD   x5,x5,x2         # Add the base pointer
LB    x5,Immed-12(x5)  # Add the lower 12 bits
```

A similar sequence can be used for "store" instructions, although another register would be required.

Note that some care must be taken when breaking a 32-bit offset into 12 and 20 bit pieces because of the sign-extension that will occur with the 12-bit portion when it is added in.

Commentary: In some cases it will be necessary to use the "load" or "store" instructions to access arbitrary locations in memory.

In some cases, a PC-relative address is preferred; in other cases an absolute address is preferred.

Any address within a 32-bit address space can be accessed using a 32-bit offset from the PC, since a 32-bit offset (signed or not) from the PC will wrap around from 0xFFFFFFFF to 0x00000000. Any 32-bit offset can be broken into two pieces: a 20-bit upper portion and a 12-bit lower portion.

For cases where an absolute address (i.e., not a PC-relative address) is desired, recall that the "Load Upper Immediate" instruction (LUI) contains a 20-bit immediate value which is shifted left 12 bits and moved into a register.

For example, to load a halfword from an arbitrary location in memory, an instruction sequence like this can be used:

```
LUI     x4,Immed-20
LH      x4,Immed-12(x4)
```

(In the case of a "store" instruction, a second register would be needed.)

As mentioned elsewhere, care must be taken when breaking a 32-bit offset into 12 and 20 bit pieces because of the sign-extension that will occur with the 12-bit portion when it is added in.

For cases where a PC-relative address is desired, recall that the "Add Upper Immediate to PC" instruction (AUIPC) contains a 20-bit immediate value which is shifted left 12 bits, added to the PC, and stored in a register.

For example, to load a halfword using an arbitrary PC-relative offset, an instruction sequence like this can be used:

```
AUIPC   x4,Immed-20
LH      x4,Immed-12(x4)
```

PC-relative addresses are preferred in situations where (1) the code may the relocated to arbitrary locations after link-time; (2) the load instruction and the target are assembled in the same file and there some reason to complete instruction generation before link-time; and (3) the address space exceeds 32-bits (4 GiBytes), but the target location lies within ±2 GiBytes of the load instruction.

Integer Multiply and Divide

The RISC-V spec makes the multiply and divide instructions optional. This section describes these instructions, which may or may not be implemented. (There was a change between RISC-V version 1.9 and 2.0; we describe the more recent version here.)

Multiply

General Form:

```
MUL     RegD,Reg1,Reg2
```

Example:

```
MUL     x4,x9,x13       # x4 = x9*x13
```

Description:

The contents of Reg1 is multiplied by the contents of Reg2 and the result is placed in RegD.

RV32 / RV64 / RV128:

Regardless of the size of the registers, the result of their multiplication will be twice as large, and therefore require 2 registers to contain. This instruction captures the lower-order half of the result and moves it into the destination register. See the commentary and the other multiply instructions for the upper half.

Comments:

There is no distinction between signed and unsigned; the result is identical. Overflow is ignored.

Encoding:

This is an R-type instruction.

Binary Multiplication – Upper Bits, Signed vs. Unsigned

Whenever two values are multiplied, the result may require up to twice as many bits to represent. For example, consider multiplying these two 8 bit values:

```
                1101 0101
                1011 1011
```

If we view these as unsigned numbers, we get this result:

```
                1101 0101 = 213          (unsigned)
              × 1011 1011 = 187          (unsigned)
     1001 1011 1001 0111 = 39,831        (unsigned)
```

If we view these as signed values, we get this result:

```
                1101 0101 = -43          (signed)
              × 1011 1011 = -69          (signed)
     0000 1011 1001 0111 = 2,967         (signed)
```

If we view one value as signed and the other as unsigned, we get this result:

```
                1101 0101 = -43          (signed)
              × 1011 1011 = 187          (unsigned)
     1110 0000 1001 0111 = -8,041        (signed)
```

Regardless of whether we are interpreting the operands as signed or unsigned numbers, note that the least-significant half of the result is always the same bits. This is not a coincidence; it is always true.

However, the most-significant half can be different, as these three cases prove.

For this reason, there must be three additional multiply instructions to obtain the most-significant half of the result:

MULH – signed operands
MULHU – unsigned operands
MULHSU – one signed operand and one unsigned operand

We may wish to detect overflow, i.e., to make sure the result is small enough to fit into the same number of bits as the operands. For an unsigned result, we can check to make sure the bits of the most-significant half are all zero. For a signed result, we must check to make sure that the bits of the most-significant half are all the same as the sign bit of the least-significant half.

Multiply – High Bits (Signed)

General Form:

```
MULH    RegD,Reg1,Reg2
```

Example:

```
MULH    x4,x9,x13       # x4 = HighBits(x9*x13)
```

Description:

The contents of Reg1 is multiplied by the contents of Reg2 and the most-significant half of the result is placed in RegD. Both operands and the result are interpreted as signed values.

Encoding:

This is a R-type instruction.

Multiply – High Bits (Unsigned)

General Form:

```
MULHU   RegD,Reg1,Reg2
```

Example:

```
MULHU   x4,x9,x13       # x4 = HighBits(x9*x13)
```

Description:

The contents of Reg1 is multiplied by the contents of Reg2 and the most-significant half of the result is placed in RegD. Both operands and the result are interpreted as unsigned values.

Encoding:

This is an R-type instruction.

Multiply – High Bits (Signed and Unsigned)

General Form:

```
MULHSU  RegD,Reg1,Reg2
```

Example:

```
MULHSU  x4,x9,x13     # x4 = HighBits(x9*x13)
```

Description:

The contents of Reg1 is multiplied by the contents of Reg2 and the most-significant half of the result is placed in RegD. One operand is interpreted as signed and one operand is interpreted as unsigned and the result is interpreted as a signed value.

The spec suggests that:

Reg2 = multiplier = signed
Reg1 = multiplicand = unsigned

but this interpretation is also a possibility ???

Reg2 = multiplier = unsigned
Reg1= multiplicand = signed

Encoding:

This is an R-type instruction.

Recommended Usage: Typically, the programmer will want to obtain the full result of a multiplication, i.e., both upper half and lower halves. This requires two multiply instructions.

For example, the following sequence

```
MULH    x4,x9,x13   # compute upper half
MUL     x5,x9,x13   # compute lower half
```

will place the result in the register pair x4:x5.

It is recommended that the instructions always be placed in this order, i.e., with the upper half computed first and the lower half second. In some implementations, the execution unit may recognize this common programming idiom and, at the micro-architectural level, fuse these two instructions into a single multiplication operation, thereby improving performance.

Additional Multiply Instruction For 64-bit and 128-bit Machines

64-bit and 128-bit machines also add an additional multiply instruction (MULW) to properly perform 32-bit multiplication, as it would be done on a 32-bit machine.

To understand why this is necessary, consider a 64-bit machine being used to execute 32-bit code operating on 32-bit integers. A RV64 machine will store every 32-bit value in the lower-order bits of a 64-bit register, with the upper 32 bits containing the sign extension of the lower-order 32 bits.

But when we multiply two 32-bit values, the result might overflow and be larger than 32-bits. The result we desire is the low-order 32 bits of the correct answer, with a sign extension in the upper 32 bits. So, to properly emulate the behavior of a 32-bit multiply, we don't actually want the mathematically correct result.

Let's look at an example. To make our example manageable, let's consider a machine with 16-bit registers, which are being used to contain 8-bit values (rather than 64-bit registers containing 32-bit values).

Imagine that we wish to multiply 117 × 94. Both numbers can be represented as 8-bit signed values, but the result (10,998) cannot be.

```
  0000 0000 0111 0101 = +117
× 0000 0000 0101 1110 = +94
  0010 1010 1111 0110 = +10,998
```

This is a situation where overflow occurs, in the sense that the result cannot be represented in only 8 bits. In order to emulate an 8-bit machine, we want the low-order 8 bits of the correct result, with sign extension in the upper 8 bits, like this:

```
1111 1111 1111 0110
```

The MUL instruction would give us the mathematically correct result, namely

```
0010 1010 1111 0110
```

But we don't want the mathematically correct result. Instead we need a new instruction to give us the appropriate value for emulating the 8-bit multiply operation, namely the right lower-order bits, with sign extension in the upper part of the register:

```
1111 1111 1111 0110
```

Multiply Word

General Form:

```
MULW    RegD,Reg1,Reg2
```

Example:

```
MULW    x4,x9,x13      # x4 = x9*x13
```

Description:

The contents of Reg1 is multiplied by the contents of Reg2 and the result is placed in RegD. Only the lower order 32-bits of the result are used; the lower 32 bits are signed extended to the full length of the register.

Comment:

This instruction is used to properly emulate 32-bit multiplication on a 64-bit or 128-bit machine.

Note that only the least-significant 32 bits of Reg1 and Reg2 can possibly affect the result.

If you want the upper 32-bits of the full 64-bit result use the MUL instruction on a 64-bit machine.

RV32/RV64/RV128:

This instruction is only available on 64-bit and 128-bit machines.

Encoding:

This is an R-type instruction.

Commentary: Consider the following sequence when executed on a 32-bit machine:

```
MULH    x4,x9,x13    # compute upper half
MUL     x5,x9,x13    # compute lower half
```

It will place the 64-bit result in the register pair x4:x5.

Now consider executing this same sequence on a 64-bit machine. The x5 register will contain the full 64-bit value, not a 32-bit, sign-extended value. The x4 register will contain nothing but meaningless sign bits.

Previously we said that 32-bit code could be executed on a 64-bit machine with no change: The idea was that the upper 32 bits of the registers are simply ignored. This is a clear exception: in this sequence valid bits are placed in the upper 32 bits. Therefore, the code sequence to perform a 32-bit multiply with 64-bit result must be different for RV32 and RV64 machines.

Additional Multiply Instruction For 128 Bit Machines

It would seem logical for a "double" version of MUL (with the name "MULD") to exist for RV128 machines, in analogy to the MULW instruction. However the spec does not mention this instruction.

Integer Division – Basic Concepts

With division and remainder there is always question about how negative operands are treated.

Consider dividing *a* by *n* (that is, *a/n*).

```
q = a DIV n      # compute quotient
r = a REM n      # compute remainder
```

There is agreement that the quotient (*q*) and remainder (*r*) must obey these equations:

$$a = nq + r$$
$$|r| < |q|$$

Many languages (C, C++, Java) perform "truncated division":

$$q = trunc(a/n)$$
$$r = a - n\ trunc(a/n)$$

which produces these results:

```
 7 /  3 =  2        7 %  3 =  1
-7 /  3 = -2       -7 %  3 = -1
 7 / -3 = -2        7 % -3 =  1
-7 / -3 =  2       -7 % -3 = -1
```

Another reasonable definition is "floored division":

$q = \lfloor a/n \rfloor$
$r = a - n\lfloor a/n \rfloor$

which produces the following results. The dot (•) indicates differences with truncated division.

```
 7 /  3 =  2        7 %  3 =  1
-7 /  3 = -3 •     -7 %  3 =  2 •
 7 / -3 = -3 •      7 % -3 = -2 •
-7 / -3 =  2       -7 % -3 = -1
```

There is also a definition called "Euclidean division", in which the remainder is never negative. The dot (•) indicates differences with both the previous definitions.

```
 7 /  3 =  2        7 %  3 =  1
-7 /  3 = -3       -7 %  3 =  2
 7 / -3 = -2        7 % -3 =  1
-7 / -3 =  3 •     -7 % -3 =  2 •
```

The RISC-V spec makes no commitment regarding the exact meaning of DIV and REM. The following quote from Wikipedia is pertinent:

> "… Euclidean division is superior to the other ones in terms of regularity and useful mathematical properties, although floored division … is also a good definition. Despite its widespread use, truncated division is shown to be inferior to the other definitions."
>
> — Daan Leijen, *Division and Modulus for Computer Scientists*

Division Error Conditions

RISC-V supports both signed division (i.e., dividing a signed number by a signed number) and unsigned division (i.e., dividing an unsigned number by an unsigned number). With multiplication, you could mix signed and unsigned; not so with division.

Concerning the sizes of the result, note that the following must hold, since $|n| \geq 1$:

$|q| \leq |a|$
$|r| < |a|$

Thus, if the operands (*a* and *n*) are 32-bits, then the results (*q* and *r*) will almost always fit into 32-bits.

There is exactly one exception in which the results will not fit. This is called "division overflow" and it can only occur with signed division.

Let *M* represent -2^{31}, which is the most negative number representable in 32-bits. If we divide *M* by -1, the result is $+2^{31}$, which is one greater than the largest 32-bit number.

The RISC-V specifies that division overflow will be handled as follows:

Correct Result:
$q = +2^{31}$
$r = 0$
Actual Result:
$q = -2^{31}$
$r = 0$

The other well-known issue is with division by zero. The spec says that the result will be:

q = 0xFFFFFFFF
$r = a$

This can be interpreted as:

Signed Divide-By-Zero Result:
$q = -1$
$r = a$

Unsigned Divide-By-Zero Result:
$q = +2^{32}-1$ i.e., the maximum unsigned number
$r = a$

Our discussion of error conditions is for 32-bit division, but naturally extends and applies to 64-bit and 128-bit division.

Both divide-by-zero and division-overflow errors occur without causing a trap or exception and instruction execution continues with no indication. In fact there are

no arithmetic traps or exceptions. If some high-level language mandates that divide-by-zero must be caught, then the compiler must insert a single BRANCH-IF-ZERO instruction. The actual results computed by the hardware are chosen because these are the values that naturally result from a typical hardware implementation.

If the high-level language mandates that division-overflow should be caught (and they all ought to!) then at least one additional test-and-branch instruction must be executed.

Divide (Signed)

General Form:

```
DIV     RegD,Reg1,Reg2
```

Example:

```
DIV     x4,x9,x13     # x4 = x9 DIV x13
```

Description:

The contents of Reg1 is divided by the contents of Reg2 and the quotient is placed in RegD. Both operands and the result are signed values.

Comments:

Divide-by-zero and division-overflow result in mathematically incorrect results. See discussion above.

Encoding:

This is an R-type instruction.

Divide (Unsigned)

General Form:

```
DIVU    RegD,Reg1,Reg2
```

Example:

```
DIVU    x4,x9,x13     # x4 = x9 DIV x13
```

Description:

The contents of Reg1 is divided by the contents of Reg2 and the quotient is placed in RegD. Both operands and the result are unsigned values.

Comments:

Divide-by-zero produces a mathematically incorrect result. Division-overflow cannot occur. See discussion above.

Encoding:

This is an R-type instruction.

Remainder (Signed)

General Form:

```
REM     RegD,Reg1,Reg2
```

Example:

```
REM     x4,x9,x13     # x4 = x9 REM x13
```

Description:

The contents of Reg1 is divided by the contents of Reg2 and the remainder is placed in RegD. Both operands and the result are signed values.

Comments:

Divide-by-zero and division-overflow result in mathematically incorrect results. See discussion above.

Encoding:

This is an R-type instruction.

Remainder (Unsigned)

General Form:

```
REMU    RegD,Reg1,Reg2
```

Example:

```
REMU    x4,x9,x13     # x4 = x9 REM x13
```

Description:

The contents of Reg1 is divided by the contents of Reg2 and the remainder is placed in RegD. Both operands and the result are unsigned values.

Comments:

Divide-by-zero produces a mathematically incorrect result. Division-overflow cannot occur. See discussion above.

Encoding:

This is an R-type instruction.

Recommended Usage: Often the programmer will want to obtain both the quotient and remainder. It is recommended that the DIV be done first and the REM be done second.

For example:

```
    DIV     x4,x9,x13    # x4 = x9 DIV x13
    REM     x5,x9,x13    # x5 = x9 REM x13
```

In some implementations, the execution unit may recognize this common pattern and fuse these two instructions into a single division operation, thereby improving performance.

Additional Divide Instructions For 64 Bit Machines

For 64-bit machines, there are 4 additional DIVIDE/REMAINDER instructions to perform 32-bit operations. In other words, the above 4 instructions will perform 32-bit operations on a RV32 machine but will perform 64-bit operations on an RV64 machine. The following 4 instructions will work just like the above instructions would work on a 32-bit machine, by sign-extending everything up to 64-bits.

Divide Word (Signed)

General Form:

```
    DIVW    RegD,Reg1,Reg2
```

Example:

```
    DIVW    x4,x9,x13      # x4 = x9 DIV x13
```

RV32/RV64/RV128:

This instruction is only available on 64-bit and 128-bit machines.

Description:

The contents of Reg1 is divided by the contents of Reg2 and the quotient is placed in RegD. Both operands and the result are signed values. Only the low-order 32 bits of the operands are used and the 32-bit result is signed-extended to fill the destination register.

Comments:

Divide-by-zero and division-overflow result in mathematically incorrect results. See discussion above.

Encoding:

This is an R-type instruction.

Divide Word (Unsigned)

General Form:

```
    DIVUW   RegD,Reg1,Reg2
```

Example:

```
DIVUW   x4,x9,x13     # x4 = x9 DIV x13
```

RV32/RV64/RV128:

This instruction is only available on 64-bit and 128-bit machines.

Description:

The contents of Reg1 is divided by the contents of Reg2 and the quotient is placed in RegD. Both operands and the result are unsigned values. Only the low-order 32 bits of the operands are used and the 32-bit result is signed-extended (???) to fill the destination register.

Comments:

Divide-by-zero produces a mathematically incorrect result. Division-overflow cannot occur. See discussion above.

Encoding:

This is an R-type instruction.

Remainder Word (Signed)

General Form:

```
REMW    RegD,Reg1,Reg2
```

Example:

```
REMW    x4,x9,x13     # x4 = x9 REM x13
```

RV32/RV64/RV128:

This instruction is only available on 64-bit and 128-bit machines.

Description:

The contents of Reg1 is divided by the contents of Reg2 and the remainder is placed in RegD. Both operands and the result are signed values. Only the low-order 32 bits of the operands are used and the 32-bit result is signed-extended to fill the destination register.

Comments:

Divide-by-zero and division-overflow result in mathematically incorrect results. See discussion above.

Encoding:

This is an R-type instruction.

Remainder Word (Unsigned)

General Form:

```
REMUW   RegD,Reg1,Reg2
```

Example:

```
REMUW   x4,x9,x13      # x4 = x9 REM x13
```

RV32/RV64/RV128:

This instruction is only available on 64-bit and 128-bit machines.

Description:

The contents of Reg1 is divided by the contents of Reg2 and the remainder is placed in RegD. Both operands and the result are unsigned values. Only the low-order 32 bits of the operands are used and the 32-bit result is signed-extended (???) to fill the destination register.

Comments:

Divide-by-zero produces a mathematically incorrect result. Division-overflow cannot occur. See discussion above.

Encoding:

This is an R-type instruction.

Additional Divide Instructions For 128 Bit Machines

It would seem logical for “double” versions of these instructions to exist for RV128 machines. However the spec does not mention these.

Chapter 4: Floating Point Instructions

Floating Pointing – Review of Basic Concepts

This section can be skipped if you are familiar with floating point arithmetic. The discussion here is general, and not specific to RISC-V.

The IEEE 754-2008 standard describes how floating point numbers are to be represented and how floating point operations are to be executed by computers.

The standard is complicated and detailed. This section is meant to be an introduction and is not an exhaustive description. Most modern processor ISAs implement the IEEE 754-2008 specification, but the specification has options and some parts are not fully implemented on some computers.

The specification defines two important data types:

Single Precision (32-bit "float" values)
Double Precision (64-bit "double" values)

Some computers implement only single precision; other computers implement both.

In either case, the idea is to represent a real rational number in a way similar to scientific notation. For example, the following number is given in scientific notation:

6.022×10^{23} (an approximation to Avogadro's constant)

With only 32 bits of precision (or 64 bits for double precision), there are limits to the amount of precision and the size of the exponents. The available bits are used as follows:

Number of bits used for…

	Single	Double
Sign	1	1
Exponent	8	11
Value	23	52
Total	**32**	**64**

Furthermore, with floating point a numerical value is represented in binary (not decimal) and this introduces some subtleties when going back and forth between the internal bit patterns and decimal representations which humans can read.

Every positive integer can be represented with a finite number of digits and a finite number of bits. For example, here is the same number, represented both ways. Of course, this number requires a few more characters in binary, but the represented value is equal.

2,468 (decimal)
100110100100 (binary)

We commonly represent rational numbers in decimal using a "decimal point", as in:

123.456

We can also represent rational numbers in binary using a "binary point", as in:

101.0101

With decimal numbers, the position of each digit is important and we talk about the place value of the digits. The place values are all powers of 10:

…	1000	100	10	1	1/10	1/100	1/100	1/1000	…
…	10^3	10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-4}	…

We can use the place values to compute the value of a number, as you learned in primary school:

123.456
= (**1** × 10^2) + (**2** × 10^1) + (**3** × 10^0) + (**4** × 10^{-1}) + (**5** × 10^{-2}) + (**6**× 10^{-3})
= (**1**×100) + (**2**×10) + (**3**×1) + (**4**×0.1) + (**5**×0.01) + (**6**×0.001)

Likewise, with binary numbers, the value of each bit is scaled according to the place value of the bit. But with binary, the place values are all powers of 2:

…	8	4	2	1	1/2	1/4	1/8	1/16	…
…	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	…

Using this, we can convert binary numbers into decimal numbers:

101.0101
= (**1** × 2^2) + (**0** × 2^1) + (**1** × 2^0) + (**0** × 2^{-1}) + (**1** × 2^{-2}) + (**0** × 2^{-3}) + (**1** × 2^{-4})
= (**1** × 4) + (**0** × 2) + (**1** × 1) + (**0** × 1/2) + (**1** × 1/4) + (**0** × 1/8) + (**1** × 1/16)
= 4 + 1 + 1/4 + 1/16
= 5.3125

Some rational numbers require an infinite number of digits in their decimal representation. For example:

1/3 = 0.33333… = 0.3(3)*

Likewise, some rational numbers may require an infinite number of bits in their binary representation. But regardless of whether we represent a rational number in decimal or binary, the infinite strings of digits/bits will settle into a simple repeating pattern. This is true of all rational numbers, but irrational numbers (e.g., π, $\sqrt{2}$) do not have such simple decimal or binary representations. Neither their decimal nor their binary expansions will ever exhibit a repeating pattern.

Some numbers many have a finite representation in decimal but require an infinite sequence in binary. For example, the following number:

4.3

requires an infinite binary expansion to represent it, namely:

100.01001100110011… = 100.01(0011)*

It turns out that every binary number without a repeating part can be represented with a finite number of decimal digits. Furthermore, the number of digits to the right of the decimal point will never exceed the number of places to the right of the binary point. For example:

101.1101 *(binary)* = 5.8125 *(decimal)*

Turning to floating point representation, we have limited number of bits available, which means we cannot accommodate arbitrary precision. Not every number is representable, so we must round numbers to a nearby number that is representable.

For example, the number 6.022×10^{23} can only be represented approximately, even though it appears not to have a great amount of precision. Here is the closest number that can be represented using a single precision floating point:

$6.02200013124147498450944 \times 10^{23}$

On the other hand, it turns out that this number:

$2.383496609792 \times 10^{12}$

can be represented exactly using only 32 bit single precision. The next largest value that can be represented exactly happens to be:

$2.383496871936 \times 10^{12}$

We can make the following statements about IEEE 754-2008 floating point number representation:

- Every floating point numbers has a sign. Every number is either positive or negative.
- There are two representations for zero: positive zero (i.e., +0.0) and negative zero (i.e., -0.0).
- There are two representations of infinity: positive infinity (+∞ or +inf) and negative infinity (-∞ or -inf)
- The exponent may be positive or negative, allowing both very large numbers and very small numbers.
- There is a special representation called “not a number” (“NaN”). This value can represent a missing value or the result of a undefined operation, such as divide

by zero. In some implementations there are two variations, called “quiet NaN” and “signaling NaN”.

- Every 32-bit integer (i.e., every integer in the range -2,147,483,648 to +2,147,483,647) can be represented exactly with a 64 bit double precision floating point number, but not with a single precision float. In fact, the integer range is a little greater: every 54-bit integer can be represented exactly in double precision floating point. Almost all larger integers will get rounded.

- Every 25-bit integer (i.e., every integer in the range -16,777,216 to +16,777,215) can be represented exactly with a 32 bit single precision floating point number. Almost all integers outside of this range will get rounded.

Here is the range of values that can be represented. (We use decimal notation here and approximate the exact values.)

Single Precision

Largest value:	$\sim 3.40282347 \times 10^{+38}$
Smallest value above zero:	$\sim 1.17549440 \times 10^{-38}$
Digits of accuracy:	about 7

Double Precision

Largest value:	$\sim 1.7976931348623157 \times 10^{+308}$
Smallest value above zero:	$\sim 2.2250738585072014 \times 10^{-308}$
Digits of accuracy:	about 16

Remember that not every value in the above ranges can be represented. Why? Recall there is a countable infinity of rational numbers just between any two numbers, yet with only 32 or 64 bits, we only have a small number of unique representations.

Floating point arithmetic is meant to mimic mathematical arithmetic, but it must be remembered that they are only approximately the same:

- The exact value or result of an operation is **not** always representable, so the computed answer is often **not** mathematically correct.

- Floating point addition is **not** always associative, due to rounding errors. That is, (x + y) + z is not always equal to x + (y + z).

- Floating point multiplication is **not** always associative. That is, (x * y) * z is not always equal to x * (y * z).

- Floating point multiplication does **not** always distribute over addition with the exact same results. That is, x * (y + z) is not always equal to (x * y) + (x * z).

However, we can say this:

- Floating point addition and multiplication are commutative, like math. For example, x+y = y+x, so you don't have to worry about the order of operands for a single operation.

Here are the different types of things that can be represented in a floating point bit pattern:

- Positive zero (+0.0)
- Negative zero (-0.0)
- Positive infinity (+∞ or +inf)
- Negative infinity (-∞ or -inf)
- Not-a-number (NaN)
  - Quiet Nan (qNaN)
  - Signaling Nan (sNaN)
- Normal numbers (or "normalized numbers")
- Denormalized numbers (or "denormals")

Zero – Positive and Negative

There are exactly two ways to represent zero, one is positive and the other is negative. This is unlike math, where there is only a single number called zero and it is unsigned.

Here are some interesting behaviors:

1/+0 yields +∞
1/-0 yields –∞
+0 will normally compare as equal to -0 (e.g., the == in the "C" language)
Some languages provide a way to distinguish +0 and -0.

There are additional behaviors, such as:

-0/-∞ yields +0

Although +0 and -0 may compare as equal, they may also result in different outcomes in some computations. This challenges our understanding of the meaning of "equal", to say the least.

The bit pattern representation of zero is:

	single	**double**
+0.0	0x0000 0000	0x0000 0000 0000 0000
-0.0	0x8000 0000	0x8000 0000 0000 0000

Note that the floating point representation for +0.0 is bit-for-bit identical to the representation for 0 in binary integer representation (both signed and unsigned). Also, -0.0 is represented identically to the most negative signed integer.

Not a Number - NaN

Operations like the following are mathematically undefined and, when attempted, will result in a NaN result, to indicate that the result is undefined.

0/0
∞ / ∞
0 * ∞

Other operations are mathematically defined but give a complex result. Complex numbers are not handled by floating point, so operations such as the following will return NaN.

Square root of a negative number
Log of a negative number

Another use of NaN is to represent an uninitialized or missing value. If a variable is used before it is initialized, spurious incorrect results might occur, but this can be avoided if the variable contains NaN.

The IEEE spec actually mentions two kinds of NaN: "signaling NaN" and "quiet NaN". But usually, we just talk about NaN without making any distinction about whether it is signaling or quiet.

A “signaling NaN” is supposed to cause a break in the flow of execution when it is encountered in a computation. That is, a trap or exception of some sort will occur, and the normal instruction sequence will be interrupted immediately. Signaling NaNs might reasonably used for uninitialized values: their use may represent a program bug which needs attention. In theory, signaling NaNs might also be used as placeholders for values (such as complex numbers) which require special handling.

The idea with a “quiet NaN”, is that it can be used as an operand in arithmetic operations . Furthermore, a quiet NaN will be propagated. That is, if one of the arguments is a quiet NaN, the result will also be a quiet NaN. This allows a lengthy sequence of operations to be performed quickly with no special testing for problems. Once a NaN appears, as a result of some error, it will persist in the chain of computations. Each subsequent operation will complete normally, without causing an exception or trap even though some sort of error occurred earlier in the sequence. If any problems occur at any step of the computation, the final result will be a quiet NaN. Therefore, it is sufficient to perform only a single test for NaN after the entire computation to see if any errors arose at any stage of the computation.

The spec does not require signaling NaNs; they are optional. One implementation approach is for the hardware to interpret all NaN values identically, basically as quiet NaNs.

There are several bit patterns that can be used to represent NaNs, so there is not a single bit pattern for NaN.

A value is defined to represent NaN if (1) the exponent field is all 1’s, and (2) the bits of the fraction field are not all zero. (If the fraction bits are all zero, then the value is either +∞ or −∞.) The sign bit of a NaN value is ignored.

If a distinction between quiet and signaling NaN is implemented, then one of the bits in the fraction field will be used to distinguish between quiet and signaling.

The exact bit patterns for NaN are not fully specified and can vary between implementations.

We can say that a value with all bits set to one (i.e., the representation for the signed integer -1 which is 0xFFFF FFFF or 0xFFFF FFFF FFFF FFFF) will definitely represent a NaN and will almost certainly represent a quiet NaN. For example, this all-ones pattern will be a quiet NaN for Intel, AMD, SPARC, ARM, RISC-V, etc.

Mixing Single and Double Precision Using NaN

There are many bits in the fraction field, and the only requirement for NaN is that they cannot all be zero. Thus, there is room to store some additional data within the NaN. So a NaN can carry a sort of "payload" value in the fraction bits. This capability may or may not be used in a particular implementation of IEEE 754-2008.

For example, the fraction field in a double is 52 bits. Assume that one bit is reserved to be always set to indicate that this is a NaN, and assume that a second bit is reserved and used to distinguish between a quiet NaN and a signaling NaN. This leaves 50 bits that can be used to store a arbitrary value. Notice that this is enough room to store an entire single precision floating point number.

Imagine a machine that implements double precision arithmetic and uses 64-bit registers to store floating point values. How might this machine store 32-bit single precision values in these same registers?

Any 64-bit value in which the high order 32 bits are set, will be always recognized as a NaN. One approach to storing a single precision value in a 64 bit register is to store the single precision value in the least significant bits 32 bits and all 1s in the most significant 32 bits.

All single precision operations will only look at the least significant 32-bits of the operands and, for the result value, will always set the most-significant 32 bits to 1s.

Any accidental attempt to perform a double precision operation on a register containing a single precision value, will interpret that operand as a NaN.

Normal and Denormalized Numbers

Not every number is representable and the representable numbers are spaced out on the number line. So each possible floating point value is separated by a numerical distance from the next smallest number and from the next largest number. As the numbers get smaller and closer to zero, the spacing gets smaller and the numbers are closer together. As the numbers get larger, the spacing is farther apart.

For example, the following numbers differ by a very small amount:

4.567×10^{-25}
4.568×10^{-25}

On the other hand, these two numbers differ by a very large amount:

$4.567 \times 10^{+25}$
$4.568 \times 10^{+25}$

However in both examples above, the accuracy is the same: 4 digits of precision.

However, there are only a limited number of bits available to represent the exponents. Exponents cannot continue to get more negative and we cannot represent smaller and smaller numbers, ever more close to zero. Therefore, this pattern of the floating point numbers becoming spaced ever more closely as they get closer and closer to zero cannot continue. Something has to change as the numbers get smaller and approach zero.

What happens is that below some size, the representable values are simply spaced uniformly all the way down to zero. This is the role of denormalized numbers.

Most floating point numbers are "normal" numbers. Normal numbers have about 7 digits of accuracy (for single precision) and 16 digits of accuracy (for double precision). In other words, we can approximate any desired value with about 7 (or 16) digits of accuracy.

Another way to look at denormalized numbers is this: For very small values, we cannot approximate the value with full accuracy. As we get closer and closer to zero, we can approximate the true value with fewer and fewer places of accuracy. For really tiny values, we may even be forced to use 0.0 to represent the value, essentially losing all accuracy.

We can make the following statements about denormalized numbers:

- All denormalized numbers are very close to zero.
- Denormalized numbers extend on both the positive and negative sides of zero.
- +0.0 and -0.0 are themselves represented as denormalized numbers.
- All denormalized numbers are regularly and evenly spaced. (Exception: +0.0 and -0.0 have an infinitesimal difference and are considered equal.)

- The largest denormalized number is just less than the smallest positive normal number.
- Likewise, the most negative denormalized number is just greater than the least negative normal number.
- It is generally safe to ignore the distinction between normalized and denormalized numbers when using floating point in your applications.

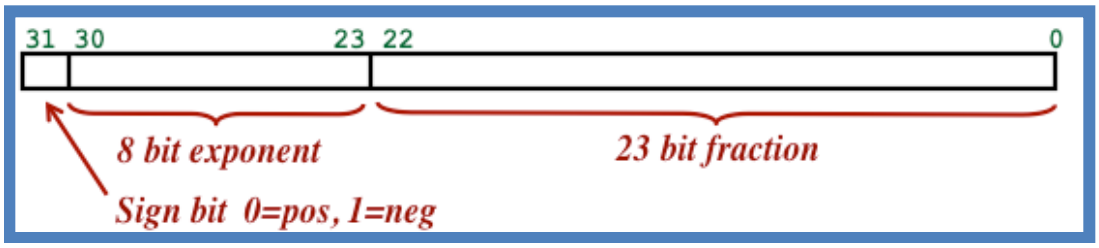
There are rules for determining the precision of the results of an arithmetic calculation involving scientific notation. But if very small values (i.e., denormalized numbers) arise during a computation, then your assumptions about precision will be violated and the final results will have reduced precision. In some cases, the final result will be a meaningless, incorrect value.

Warning: Always remember that numbers as represented in computers are NOT true mathematical numbers. Computer arithmetic is NOT mathematical arithmetic. Remember: "int"s are not integers and "floats" are not real or rational numbers.

Computer values and computation are mere approximations of mathematically pure ideals. A good programmer knows how important it is to understand and remember their differences in creating reliable software.

Representation of Single Precision Floating Point Values

A single precision floating point number is represented with a 32-bit word as shown here:



Let N be the number represented. Let "**sign**" be the most significant bit. Let "**exponent**" be the bit pattern in bits 30:23. Let "**fraction**" be the bit pattern in bits 22:0.

The number represented is:

$$N = (-1)^{\mathbf{sign}} \times 1.\mathbf{fraction} \times 2^{\mathbf{exponent}}$$

The first term simply gives the sign of the number: 0=positive and 1=negative. Note that the most significant bit holds the sign bit for both floating point numbers and signed integers.

When converting a number such as 101.0101 into floating point format, you should first shift the decimal place to just after the leftmost 1 bit. Every number (except zero) will always contain at least a single 1 bit. Thus, the most significant bit must be a 1 and representing it is redundant. This explains why we prefix the fractional part with "1.". (This trick of making one bit implicit doesn't work with decimal numbers: the leading digit can be anything except 0, so we cannot make it implicit.)

There are 8 bits in the exponent field. The interpretation of the "exponent" bit patterns is:

Bit Pattern	**Meaning of Exponent Field**
0000 0000	*Denormalized Numbers, including zero*
0000 0001	-126
...	...
0111 1110	-1
0111 1111	0
1000 0000	+1
...	...
1111 1110	+127
1111 1111	*Infinity, Not-a-Number*

For normalized numbers, the exponent has an effective range of -126 .. +127.

The smallest positive normalized number is:

In binary:

$1.00000000000000000000000 \times 2^{-126}$ (There are 23 zeros)

In bits:

0x00800000 = 0 00000001 00000000000000000000000

Decimal approximation:

$1.17549435 \times 10^{-38}$

The largest normalized number is:

In binary:

$1.11111111111111111111111 \times 2^{+127}$ (There are 1+23 ones)

In bits:

0x7F7FFFFF = 0 11111110 11111111111111111111111

Decimal approximation:

$3.4028235 \times 10^{+38}$

If the exponent is all ones (i.e., 11111111), then the value of the fraction matters. If the fraction is all zeros, then the value is +∞ or –∞ depending on the sign bit.

+∞:

0x7F800000 = 0 11111111 00000000000000000000000

-∞:

0xFF800000 = 1 11111111 00000000000000000000000

If the exponent is all ones (i.e., 11111111) and the value of the fraction is not all zeros, then NaN is represented. There are multiple representations that are to be interpreted as NaN values. The canonical, preferred representation of NaN is often this:

NaN (typical):

0xFFFFFFFF = 1 11111111 11111111111111111111111

If the exponent field is all zeros (i.e., 00000000), then the value is a denormalized number. The value of the number is:

$N = (-1)^{\mathbf{sign}} \times 0.\mathbf{fraction} \times 2^{\mathbf{-126}}$

Notice that the leading implicit "1" bit is no longer assumed; it is now "0". Also the exponent is always -126, which happens to be the smallest exponent for normalized numbers.

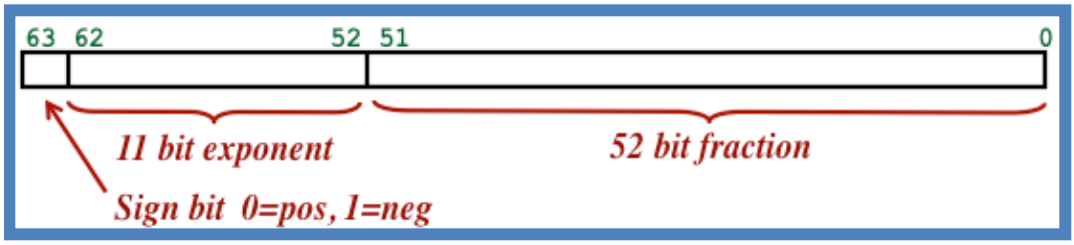
Here are some sample numbers that may help explain denormalized numbers:

Smallest normalized number:
1.00000000000000000000000 × 2^{-126} (24 bits of precision)
Largest denormalized number:
0.11111111111111111111111 × 2^{-126} (23 bits of precision)
…
Random denormalized number:
0.00000000001100101110101 × 2^{-126} (13 bits of precision)
…
Smallest denormalized number:
0.00000000000000000000001 × 2^{-126} (1 bit of precision)
+0.0:
0.00000000000000000000000 × 2^{-126} (0 bits of precision)

Representation of Double Precision Floating Point Values

Double precision floating point numbers are represented using an analogous scheme. The only difference is the number of bits in the "exponent" and "fraction" fields.

Here is the representation of a 64-bit double precision floating point value:



There are 11 bits in the exponent field, instead of 8 as in single precision. The interpretation of the "exponent" bit patterns is:

Bit Pattern	Meaning of Exponent Field
000 0000 0000	*Denormalized Numbers, including zero*
000 0000 0001	-1022
...	...
011 1111 1110	-1
011 1111 1111	0
100 0000 0000	+1
...	...
111 1111 1110	+1023
111 1111 1111	*Infinity, Not-a-Number*

For normalized numbers, the exponent has an effective range of -1022 .. +1023.

The smallest positive normalized number is:

In binary:
$1.000000000...00000000000 \times 2^{-1022}$ (There are 52 zeros)
In bits:
0x0010 0000 0000 0000
Decimal approximation:
$2.2250738585072014 \times 10^{-308}$

The largest normalized number is:

In binary:
$1.111111111...11111111111 \times 2^{+1023}$ (There are 1+52 ones)
In bits:
0x7FEF FFFF FFFF FFFF
Decimal approximation:
$1.7976931348623157 \times 10^{+308}$

The smallest denormalized number is:

In binary:
$0.000000000...00000000001 \times 2^{-1022}$
In bits:
0x0000 0000 0000 0001
Decimal approximation:
5.0×10^{-324}

Other Floating Point Sizes

In addition to the well known single precision (32-bit) and double precision (64-bit) sizes, the IEEE 754-2008 standard also describes these floating point sizes. They are much less common.

16 bits (half precision)
128 bits (quadruple precision)
256 bits (octuple precision)

There is also mention of decimal-based representations.

Rounding

The result of a computation may be a number that is not precisely representable. For example, the addition of two double numbers may not be exactly representable as a double.

For example, here are two numbers with 5 digits of precision. Their sum has 8 digits of precision.

12.345	$= 1.2345 \times 10^{1}$
+ .067891	$= 6.7891 \times 10^{-2}$
12.412891	$= 1.2412891 \times 10^{1}$

The IEEE spec says that the exact result should be "rounded" to a number that can be represented. For example, when two doubles are added, their result will be some double value.

In the above example, to bring the result back down to 5 digits of precision, some accuracy must be sacrificed. Likewise, when floating point operations are performed, there may be a loss of accuracy as a result of rounding.

The IEEE spec lists several ways that a value can be rounded to something that can be represented:

- Round to the nearest number
  (For a tie, the value with a zero in the least significant bit is chosen.)
- Round toward zero (i.e., truncate)
- Round toward positive infinity (i.e., round up)
- Round toward negative infinity (i.e., round down)

In order to perform rounding correctly, a computer may need to perform calculations (e.g., multiplication) with greater precision to first compute the correct value. Then, as the final step in the calculation, the value must be properly rounded to fit into the available floating point bits.

Here is another example, showing that the entire effect of an operation can be lost as the result of rounding.

```
   12.345              = 1.2345 × 10^1
+    .00000067891      = 6.7891 × 10^-7
   12.34500067891      = 1.234500067891 × 10^1
```

Rounding to a value with the same precision (either rounding down, rounding to zero, or rounding to the nearest) gives the initial operand, unchanged. Here is the rounded result:

```
   12.345              = 1.2345 × 10^1
```

The Floating Point Extensions

The RISC-V spec describes these extensions to support floating point arithmetic

F – Single precision floating point (32 bit values)
D – Double precision floating point (64 bit values)
Q – Quad precision floating point (128 bit values)

The “D” extension is a superset of “F”; when double precision is implemented, all instructions operating on single precision values will also be included.

Likewise, the “Q” is a superset of “D”; when the “Q” extension is implemented, all “F” and “D” instructions will also be implemented.

In this document, we intermix the descriptions of "F", "D", and "Q" extensions, rather than describe one after the other.

Floating Point Registers

There are 32 floating point registers, named f0, f1, … f31.

In the "F" extension, each register can hold one single precision floating point value. Thus, each register is 32 bits wide. All registers function identically; there is nothing special about f0, as there is with x0 of the integer registers.

In the "D" extension, these registers are all 64 bits wide and can hold either a single precision value or a double precision value. If the register contains a single precision floating point, then the most significant 32 bits of the register will all be 1. When interpreted as a double, such a value will be recognized as a NaN.

In the "Q" extension, the registers are 128 bits wide. Each register can hold either a single, double, or quad precision floating point value. In the case of smaller values, the upper bits will be set to 1 so that the value will appear to be NaN if interpreted as a larger precision value.

In addition there is a "Floating Point Control and Status Register" (called "FCSR").

The FCSR is divided into the following fields:

Bits	Width in bits	Description
0	1	NX: Inexact
1	1	UF: Underflow
2	1	OF: Overflow
3	1	DZ: Divide By Zero
4	1	NV: Invalid Operation
5:7	3	Floating Point Rounding Mode (FRM)
8:31	24	(unused)

The Floating Point Control and Status Register (FCSR) is one of the "Control and Status Registers (CSRs), which are discussed elsewhere in this document. This particular register is freely accessible regardless of the operating mode (user, supervisor, or machine mode), so this need not be a concern here.

There are some general purpose instructions used for reading and writing the CSRs and these are discussed elsewhere.

Collectively, the 5 single bit fields are referred to as the "Floating Point Flags" (FFLAGS):

FFLAGS (Floating Point Flags):

NX: Inexact
UF: Underflow
OF: Overflow
DZ: Divide By Zero
NV: Invalid Operation

These 5 flags can be reset to zero by writing to the Floating Point Control and Status Register (FCSR). They may be set to one from time-to-time as floating point instructions are executed. Once set, they will remain set. Therefore, these bits can be queried after a sequence of instructions to determine if any of several unusual conditions has occurred during the previous instruction sequence.

The meaning of each bit is straightforward.

If the result of an operation had to be rounded, then the "NX: Inexact" bit will be set. If the result is too small to fit in a normalized form and is also inexact, then the "UF: Underflow" bit will be set and the result will be returned as a denormalized value. If the result is too large to be represented, then the "OF: Overflow" bit will be set and $+\infty$ or $-\infty$ will usually be returned as the result. The "DZ: Divide By Zero" bit is set for operations like 1/0 and log(0) and $+\infty$ or $-\infty$ will be returned as the result. Certain operations are considered invalid, such as "square root of a negative number". The "NV: Invalid Operation" bit will be set and NaN (by default, quiet NaN) will be returned.

Floating Point instructions that have problems will set the FFLAGS bits but will never cause a trap or exception. Instead, instruction execution will continue uninterrupted.

If the result of an instruction is NaN, then a quiet NaN will be inserted, unless otherwise documented. RISC-V will insert the following representation for a quiet NaN:

Single Precision quiet NaN:
0x7FC00000 = 0 11111111 10000000000000000000000

The “unused” zero bits may be used for other RISC-V extensions, but normally they will appear to be zero when read. Floating point code should ignore the value of this field and not attempt to change it.

There are several different ways that a floating point result can be rounded to fit into the available precision:

- Round to the nearest number
  (For a tie, the value with a zero in the least significant bit is chosen.)
- Round toward zero (i.e., truncate)
- Round toward negative infinity (i.e., round down)
- Round toward positive infinity (i.e., round up)

The rounding mode can be either static or dynamic.

There is a “Rounding Mode” (RM) field in each floating point instruction. In static mode, the rounding method to be used is encoded in the RM bits within the instruction, according to these codes:

000	Round to the nearest number (For a tie, the value with a zero in the least significant bit is chosen.)
001	Round toward zero (i.e., truncate)
010	Round toward negative infinity (i.e., round down)
011	Round toward positive infinity (i.e., round up)
100	Round to the nearest number (For a tie, round away from zero.)
101	*invalid*
110	*invalid*
111	Use dynamic mode: Consult the CSR for the rounding method.

With the dynamic mode, the rounding mode will be determined by the Floating Point Rounding Mode (FRM) bits in the Floating Point Control and Status Register (FCSR). If the instructions RM field contains 111, then the rounding mode to be used is determined dynamically when the instruction is executed by consulting the FCSR. The FRM bits in the FCSR tell which rounding method to use, according to the following table. This table is the same as above, except for the last line.

000 Round to the nearest number
(For a tie, the value with a zero in the least significant bit is chosen.)
001 Round toward zero (i.e., truncate)
010 Round toward negative infinity (i.e., round down)
011 Round toward positive infinity (i.e., round up)
100 Round to the nearest number
(For a tie, round away from zero.)
101 *invalid*
110 *invalid*
111 ***invalid***

If a floating point instruction will not need to perform rounding but contains an RM field (for example, the FNEG instruction), the RM field should be set to 000.

The RISC-V spec does not indicate or suggest how the rounding mode should be written in assembly code. ???

The flags portion (NX, UF, OF, DZ, NV) of the Floating Point Status Register (FCSR) is accessible and mirrored in a second CSR called (FFLAGS). Likewise, the rounding mode bits of the Floating Point Status Register (FCSR) are accessible and mirrored in a third CSR called (FRM).

Control and Status Registers are covered elsewhere, but the register concerned with the floating point extension are:

CSR Addr	Name	Description
001	**fflags**	Floating pointing flags
002	**frm**	Dynamic rounding mode
003	**fcsr**	Concatenation of frm + fflags

These regs are read/write in any privilege mode.

Separate assembler instructions are defined to access these registers, but these are just shorthand (syntactic sugar) for other, more general instructions. The instructions are:

FRFLAGS - Read the FFLAGS register

```
FRFLAGS    RegD          RegD=CSR[FFLAGS]
```

FSFLAGS - Swap the FFLAGS register

```
FSFLAGS    RegD,Reg1     RegD=CSR[FFLAGS];CSR[FFLAGS]=Reg1
```

FRRM - Read the FRM (Rounding Mode) register

```
FRRM        RegD            RegD=CSR[FRM]
```

FSRM - Swap the FRM (Rounding Mode) register

```
FSRM        RegD,Reg1       RegD=CSR[FRM];CSR[FRM]=Reg1
```

Not-A-Number - NaN

The spec says that the "canonical NaN" is 0x7fc00000. Presumably, they mean a quiet NaN. The RISC-V spec mentions the existence of a signaling NaN, saying that if one is encountered in an operation, an invalid instruction exception will occur. However, the spec does not say how a signaling NaN is distinguished. ???

Floating Point Load and Store Instructions

The following instructions will move data between memory and a floating point register.

We use the notation FReg1, FReg2, and FRegD to indicate floating point registers, just as we use Reg1, Reg2, and RegD to indicate integer registers.

Floating Load (Word)

General Form:

```
FLW     FRegD,Immed-12(Reg1)
```

Example:

```
FLW     f4,1234(x9)     # f4 = Mem[x9+1234]
```

Description:

A 32-bit value is fetched from memory and moved into floating register FRegD. The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. word-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

RISC-V Extensions:

This instruction requires the "F" extension.

Encoding:

This is an I-type instruction.

Floating Load (Double)

General Form:

```
FLD     FRegD,Immed-12(Reg1)
```

Example:

```
FLD     f4,1234(x9)     # f4 = Mem[x9+1234]
```

Description:

A 64-bit value is fetched from memory and moved into floating register FRegD. The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. doubleword-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

RISC-V Extensions:

This instruction requires the "D" extension.

Encoding:

This is an I-type instruction.

Floating Load (Quad)

General Form:

```
FLQ     FRegD,Immed-12(Reg1)
```

Comment:

Analogous; Only available in “Q” quad precision floating point extension.

Floating Store (Word)

General Form:

```
FSW     FReg2,Immed-12(Reg1)
```

Example:

```
FSW     f4,1234(x9),f4     # Mem[x9+1234] = f4
```

Description:

A 32-bit value is copied from register FReg2 to memory. The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. word-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

RISC-V Extensions:

This instruction requires the “F” extension.

Encoding:

This is an I-type instruction.

Floating Store (Double)

General Form:

```
FSD     FReg2,Immed-12(Reg1)
```

Example:

```
FSD     f4,1234(x9)     # Mem[x9+1234] = f4
```

Description:

A 64-bit value is copied from register FReg2 to memory. The memory address is formed by adding the offset to the contents of Reg1.

Comment:

The target location given by the 12-bit offset must be within the range of -2,048 .. 2,047 relative to the value in Reg1.

The address of the memory location is not required to be properly aligned (i.e. doubleword-aligned), but it is assumed that this instruction will execute faster with properly aligned addresses.

This instruction is guaranteed to execute atomically if the address is properly aligned. If the address is not aligned, there is no guarantee of atomic operation.

RISC-V Extensions:

This instruction requires the "D" extension.

Encoding:

This is an I-type instruction.

Floating Store (Quad)

General Form:

```
FSQ     FReg2,Immed-12(Reg1)
```

Comment:

Analogous; Only available in "Q" quad precision floating point extension.

Basic Floating Point Arithmetic Instructions

Floating Add

General Form:

```
FADD.S   FRegD,FReg1,FReg2  (single precision)
FADD.D   FRegD,FReg1,FReg2  (double precision)
FADD.Q   FRegD,FReg1,FReg2  (quad precision)
```

Example:

```
FADD.S   f4,f9,f13     # f4 = f9+f13  (32 bits)
FADD.D   f4,f9,f13     # f4 = f9+f13  (64 bits)
FADD.Q   f4,f9,f13     # f4 = f9+f13  (128 bits)
```

Description:
The value in FReg1 is added to the value in FReg2 and the result is placed in FRegD.
Rounding Mode:
This instruction contains a 3 bit Rounding Mode (RM) field, which will determine the rounding method to be used.
RISC-V Extensions:
FADD.S requires the "F" extension.
FADD.D requires the "D" extension.
FADD.Q requires the "Q" extension.
Encoding:
This is an R-type instruction.

Floating Subtract

General Form:

```
FSUB.S   FRegD,FReg1,FReg2  (single precision)
FSUB.D   FRegD,FReg1,FReg2  (double precision)
FSUB.Q   FRegD,FReg1,FReg2  (quad precision)
```

Example:

```
FSUB.S   f4,f9,f13     # f4 = f9-f13  (32 bits)
FSUB.D   f4,f9,f13     # f4 = f9-f13  (64 bits)
FSUB.Q   f4,f9,f13     # f4 = f9-f13  (128 bits)
```

Description:
The value in FReg1 is subtracted from the value in FReg2 and the result is placed in FRegD.
Rounding Mode:
This instruction contains a 3 bit Rounding Mode (RM) field, which will determine the rounding method to be used.
RISC-V Extensions:
FSUB.S requires the "F" extension.
FSUB.D requires the "D" extension.
FSUB.Q requires the "Q" extension.
Encoding:
This is an R-type instruction.

Floating Multiply

General Form:

```
FMUL.S   FRegD,FReg1,FReg2  (single precision)
FMUL.D   FRegD,FReg1,FReg2  (double precision)
FMUL.Q   FRegD,FReg1,FReg2  (quad precision)
```

Example:

```
FMUL.S   f4,f9,f13     # f4 = f9*f13  (32 bits)
FMUL.D   f4,f9,f13     # f4 = f9*f13  (64 bits)
FMUL.Q   f4,f9,f13     # f4 = f9*f13  (128 bits)
```

Description:

The value in FReg1 is multiplied by the value in FReg2 and the result is placed in FRegD.

Rounding Mode:

This instruction contains a 3 bit Rounding Mode (RM) field, which will determine the rounding method to be used.

RISC-V Extensions:

FMUL.S requires the “F” extension.
FMUL.D requires the “D” extension.
FMUL.Q requires the “Q” extension.

Encoding:

This is an R-type instruction.

Floating Divide

General Form:

```
FDIV.S   FRegD,FReg1,FReg2  (single precision)
FDIV.D   FRegD,FReg1,FReg2  (double precision)
FDIV.Q   FRegD,FReg1,FReg2  (quad precision)
```

Example:

```
FDIV.S   f4,f9,f13     # f4 = f9/f13  (32 bits)
FDIV.D   f4,f9,f13     # f4 = f9/f13  (64 bits)
FDIV.Q   f4,f9,f13     # f4 = f9/f13  (128 bits)
```

Description:

The value in FReg1 is divided by the value in FReg2 and the result is placed in FRegD.

Rounding Mode:

This instruction contains a 3 bit Rounding Mode (RM) field, which will determine the rounding method to be used.

RISC-V Extensions:

FDIV.S requires the "F" extension.
FDIV.D requires the "D" extension.
FDIV.Q requires the "Q" extension.

Encoding:

This is an R-type instruction.

Floating Minimum

General Form:

```
FMIN.S   FRegD,FReg1,FReg2  (single precision)
FMIN.D   FRegD,FReg1,FReg2  (double precision)
FMIN.Q   FRegD,FReg1,FReg2  (quad precision)
```

Example:

```
FMIN.S   f4,f9,f13     # f4 = Min(f9,f13) (32 bits)
FMIN.D   f4,f9,f13     # f4 = Min(f9,f13) (64 bits)
FMIN.Q   f4,f9,f13     # f4 = Min(f9,f13) (128 bits)
```

Description:

The values in FReg1 and FReg2 are compared and the smaller one is placed in FRegD.

RISC-V Extensions:

FMIN.S requires the "F" extension.
FMIN.D requires the "D" extension.
FMIN.Q requires the "Q" extension.

Encoding:

This is an R-type instruction.

Floating Maximum

General Form:

```
FMAX.S   FRegD,FReg1,FReg2  (single precision)
FMAX.D   FRegD,FReg1,FReg2  (double precision)
FMAX.Q   FRegD,FReg1,FReg2  (quad precision)
```

Example:

```
FMAX.S   f4,f9,f13     # f4 = Max(f9,f13) (32 bits)
```

```
    FMAX.D   f4,f9,f13     # f4 = Max(f9,f13) (64 bits)
    FMAX.Q   f4,f9,f13     # f4 = Max(f9,f13) (128 bits)
```

Description:

The values in FReg1 and FReg2 are compared and the larger one is placed in FRegD.

RISC-V Extensions:

FMAX.S requires the "F" extension.
FMAX.D requires the "D" extension.
FMAX.Q requires the "Q" extension.

Encoding:

This is an R-type instruction.

Floating Square Root

General Form:

```
    FSQRT.S   FRegD,FReg1  (single precision)
    FSQRT.D   FRegD,FReg1  (double precision)
    FSQRT.Q   FRegD,FReg1  (quad precision)
```

Example:

```
    FSQRT.S   f4,f9 # f4 = sqrt(f9) (32 bits)
    FSQRT.D   f4,f9 # f4 = sqrt(f9) (64 bits)
    FSQRT.Q   f4,f9 # f4 = sqrt(f9) (128 bits)
```

Description:

The square root of the value in FReg1 is computed and the result is placed in FRegD.

Rounding Mode:

This instruction contains a 3 bit Rounding Mode (RM) field, which will determine the rounding method to be used.

RISC-V Extensions:

FSQRT.S requires the "F" extension.
FSQRT.D requires the "D" extension.
FSQRT.Q requires the "Q" extension.

Encoding:

This is a R-type instruction, in which the Reg2 field is all zeros.

Floating Fused Multiply-Add

General Form:

```
FMADD.S    FRegD,FReg1,FReg2,FReg3  (single precision)
FNMADD.S   FRegD,FReg1,FReg2,FReg3  (single precision)
FMADD.D    FRegD,FReg1,FReg2,FReg3  (double precision)
FNMADD.D   FRegD,FReg1,FReg2,FReg3  (double precision)
```

Example:

```
FMADD.S    f2,f5,f6,f7     # f2 = (f5*f6)+f7
FNMADD.S   f2,f5,f6,f7     # f2 = (-(f5*f6))+f7
FMADD.D    f2,f5,f6,f7     # f2 = (f5*f6)+f7
FNMADD.D   f2,f5,f6,f7     # f2 = (-(f5*f6))+f7
```

Description:

This instruction performs a multiplication and an addition. A variation (which is shown here with the opcode FNMADD) will optionally negate the product before the addition. See the spec concerning cases when the arguments are 0, ∞, and NaN.

Rounding Mode:

This instruction contains a 3 bit Rounding Mode (RM) field, which will determine the rounding method to be used.

RISC-V Extensions:

FMADD.S and FNMADD.S require the “F” extension.
FMADD.D and FNMADD.D require the “D” extension.

Encoding:

This instruction is unusual in that it has four operands, all of which are registers. Thus, it does not fit into any of the instruction formats that were described earlier. This instruction type is called an R4-type instruction. The R4-type instruction format is used only for this and the fused multiply-subtract instructions.

Floating Fused Multiply-Subtract

General Form:

```
FMSUB.S    FRegD,FReg1,FReg2,FReg3  (single precision)
FNMSUB.S   FRegD,FReg1,FReg2,FReg3  (single precision)
FMSUB.D    FRegD,FReg1,FReg2,FReg3  (double precision)
FNMSUB.D   FRegD,FReg1,FReg2,FReg3  (double precision)
```

Example:

```
FMSUB.S     f2,f5,f6,f7     # f2 = (f5*f6)-f7
FNMSUB.S    f2,f5,f6,f7     # f2 = (-(f5*f6))-f7
FMSUB.D     f2,f5,f6,f7     # f2 = (f5*f6)-f7
FNMSUB.D    f2,f5,f6,f7     # f2 = (-(f5*f6))-f7
```

Description:

This instruction performs a multiplication and a subtraction. A variation (which is shown here with the opcode FNMSUB) will optionally negate the product before the subtraction. See the spec concerning cases when the arguments are 0, ∞, and NaN.

Rounding Mode:

This instruction contains a 3 bit Rounding Mode (RM) field, which will determine the rounding method to be used.

RISC-V Extensions:

FMSUB.S and FNMSUB.S require the "F" extension.
FMSUB.D and FNMSUB.D require the "D" extension.

Encoding:

This instruction is unusual in that it has four operands, all of which are registers. Thus, it does not fit into any of the instruction formats that were described earlier. This instruction type is called an R4-type instruction. The R4-type instruction format is used only for this and the fused multiply-add instructions.

Floating Fused Multiply-Add / Subtract (Quad)

General Form:

```
FMADD.Q     FRegD,FReg1,FReg2,FReg3
FNMADD.Q    FRegD,FReg1,FReg2,FReg3
FMSUB.Q     FRegD,FReg1,FReg2,FReg3
FNMSUB.Q    FRegD,FReg1,FReg2,FReg3
```

Comment:

Analogous; Only available in "Q" quad precision floating point extension.

Floating Point Conversion Instructions

There is a set of instructions for converting from one representation to another. The following codes are used to name the different formats:

W	32-bit signed integer
WU	32-bit unsigned integer
L	64-bit signed integer
LU	64-bit unsigned integer
S	Single-precision floating point number
D	Double-precision floating point number

Floating Point Conversion (Integer to/from Single Precision)

General Forms:

```
FCVT.W.S    RegD,FReg1
FCVT.WU.S   RegD,FReg1
FCVT.L.S    RegD,FReg1    (available in RV64 or RV128 only)
FCVT.LU.S   RegD,FReg1    (available in RV64 or RV128 only)
FCVT.S.W    FRegD,Reg1
FCVT.S.WU   FRegD,Reg1
FCVT.S.L    FRegD,Reg1    (available in RV64 or RV128 only)
FCVT.S.LU   FRegD,Reg1    (available in RV64 or RV128 only)
```

Examples:

```
FCVT.W.S    r2,f5         # r2(int32)  = f5(float)
FCVT.WU.S   r2,f5         # r2(uint32) = f5(float)
FCVT.L.S    r2,f5         # r2(int64)  = f5(float)
FCVT.LU.S   r2,f5         # r2(uint64) = f5(float)
FCVT.S.W    f4,r7         # f4(float) = r7(int32)
FCVT.S.WU   f4,r7         # f4(float) = r7(uint32)
FCVT.S.L    f4,r7         # f4(float) = r7(int64)
FCVT.S.LU   f4,r7         # f4(float) = r7(uint64)
```

Description:

These instructions perform a conversion of a single precision floating format to/from an integer format.

Rounding:

These instructions contain Rounding Mode (RM) bits which determine how rounding is to be done.

When converting to an integer, the values +∞, NaN, and values that are too large to be represented in the target format are stored as the largest integer

value that can be represented. The value –∞ and values that are too negative to be represented in the target format are converted into the smallest integer value that can be represented.

RISC-V Extensions:

These instructions require the "F" extension. Furthermore, the instructions involving 64 bit integers are not available in RV32, as noted in the general forms above.

Encoding:

This is an R-type instruction, in which the Reg2 field is used for additional opcode bits.

Floating Point Conversion (Integer to/from Double Precision)

General Forms:

```
FCVT.W.D    RegD,FReg1
FCVT.WU.D   RegD,FReg1
FCVT.L.D    RegD,FReg1    (available in RV64 or RV128 only)
FCVT.LU.D   RegD,FReg1    (available in RV64 or RV128 only)
FCVT.D.W    FRegD,Reg1
FCVT.D.WU   FRegD,Reg1
FCVT.D.L    FRegD,Reg1    (available in RV64 or RV128 only)
FCVT.D.LU   FRegD,Reg1    (available in RV64 or RV128 only)
```

Examples:

```
FCVT.W.D    r2,f5         # r2(int32)  = f5(double)
FCVT.WU.D   r2,f5         # r2(uint32) = f5(double)
FCVT.L.D    r2,f5         # r2(int64)  = f5(double)
FCVT.LU.D   r2,f5         # r2(uint64) = f5(double)
FCVT.D.W    f4,r7         # f4(double) = r7(int32)
FCVT.D.WU   f4,r7         # f4(double) = r7(uint32)
FCVT.D.L    f4,r7         # f4(double) = r7(int64)
FCVT.D.LU   f4,r7         # f4(double) = r7(uint64)
```

Description:

These instructions perform a conversion of a double precision floating format to/from an integer format.

Rounding:

These instructions contain Rounding Mode (RM) bits which determine how rounding is to be done.

When converting to an integer, the values +∞, NaN, and values that are too large to be represented in the target format are stored as the largest integer value that can be represented. The value –∞ and values that are too negative to be represented in the target format are converted into the smallest integer value that can be represented.

RISC-V Extensions:

These instructions require the "D" extension. Furthermore, the instructions involving 64 bit integers are not available in RV32, as noted in the general forms above.

Encoding:

This is an R-type instruction, in which the Reg2 field is used for additional opcode bits.

Programming Hint: To store +0.0 in a floating register, use these instructions:

```
FCVT.S.W    FRegD,x0    # Single precision
FCVT.D.W    FRegD,x0    # Double precision
```

Floating Point Conversion (Integer to/from Quad Precision)

General Forms:

```
FCVT.W.Q    RegD,FReg1
FCVT.WU.Q   RegD,FReg1
FCVT.L.Q    RegD,FReg1    (available in RV64 or RV128 only)
FCVT.LU.Q   RegD,FReg1    (available in RV64 or RV128 only)
FCVT.Q.W    FRegD,Reg1
FCVT.Q.WU   FRegD,Reg1
FCVT.Q.L    FRegD,Reg1    (available in RV64 or RV128 only)
FCVT.Q.LU   FRegD,Reg1    (available in RV64 or RV128 only)
```

Comment:

Analogous; Only available in "Q" quad precision floating point extension.

Floating Point Conversion (Single/Double to/from Quad Precision)

General Forms:

```
FCVT.S.Q    FRegD,FReg1
FCVT.Q.S    FRegD,FReg1
```

```
    FCVT.D.Q    FRegD,FReg1
    FCVT.Q.D    FRegD,FReg1
```

Comment:

Analogous; Only available in "Q" quad precision floating point extension.

Floating Point Move Instructions

The next three instructions copy a signed value from one register to another register, while modifying the sign bit based on the sign from another value.

Floating Sign Injection (Single Precision)

General Forms:

```
    FSGNJ.S     FRegD,FReg1,FReg2    (inject)
    FSGNJN.S    FRegD,FReg1,FReg2    (negate)
    FSGNJX.S    FRegD,FReg1,FReg2    (exclusive-or)
```

Examples:

```
    FSGNJ.S     f2,f5,f6        # f2 = sign(f6) * |f5|
    FSGNJN.S    f2,f5,f6        # f2 = -sign(f6) * |f5|
    FSGNJX.S    f2,f5,f6        # f2 = sign(f6) * f5
```

Description:

Each of these instructions copies the value in FReg1 into FRegD. However the sign of the result is changed, based on the sign of the value in FReg2. In the "inject" instruction, the sign from FReg2 is used for the result. In the "negate" instruction, the sign from FReg2 is first flipped and then used for the result. In the "exclusive-or" instruction, the signs from FReg1 and FReg2 are XOR-ed together and then used for the result.

Comments:

These instructions are useful as implementations of special cases for the instructions FNEG.S, FABS.S, and FMV.S. In addition, the IEEE 754-2008 spec calls for a "copy sign" operation. These operations may also have use in some math library functions.

RISC-V Extensions:

These instructions require the "F" extension. Presumably, in the "D" extension, this instruction will only modify the lower order 32 bits, but it is possible that it will also set the upper 32 bits to all ones. ???

Encoding:
This is an R-type instruction.

Floating Sign Injection (Double Precision)

General Forms:

```
FSGNJ.D     FRegD,FReg1,FReg2    (inject)
FSGNJN.D    FRegD,FReg1,FReg2    (negate)
FSGNJX.D    FRegD,FReg1,FReg2    (exclusive-or)
```

Examples:

```
FSGNJ.D     f2,f5,f6          # f2 = sign(f6) * |f5|
FSGNJN.D    f2,f5,f6          # f2 = -sign(f6) * |f5|
FSGNJX.D    f2,f5,f6          # f2 = sign(f6) * f5
```

Description:
Each of these instructions copies the value in FReg1 into FRegD. However the sign of the result is changed, based on the sign of the value in FReg2. In the "inject" instruction, the sign from FReg2 is used for the result. In the "negate" instruction, the sign from FReg2 is first flipped and then used for the result. In the "exclusive-or" instruction, the signs from FReg1 and FReg2 are XOR-ed together and then used for the result.

Comments:
These instructions are useful as implementations of special cases for the instructions FNEG.D, FABS.D, and FMV.D. In addition, the IEEE 754-2008 spec calls for a "copy sign" operation. These operations may also have use in some math library functions.

RISC-V Extensions:
These instructions require the "D" extension.

Encoding:
This is an R-type instruction.

Floating Sign Injection (Double Precision)

General Forms:

```
FSGNJ.Q     FRegD,FReg1,FReg2    (inject)
FSGNJN.Q    FRegD,FReg1,FReg2    (negate)
FSGNJX.Q    FRegD,FReg1,FReg2    (exclusive-or)
```

Comment:
Analogous; Only available in "Q" quad precision floating point extension.

Floating Move

General Forms:

```
FMV.S       FRegD,FReg1  (single precision)
FMV.D       FRegD,FReg1  (double precision)
FMV.Q       FRegD,FReg1  (quad precision)
```

Examples:

```
FMV.S       f2,f5           # f2 = f5   (32 bits)
FMV.D       f2,f5           # f2 = f5   (64 bits)
FMV.Q       f2,f5           # f2 = f5   (128 bits)
```

Description:

The value in FReg1 is copied into FRegD.

RISC-V Extensions:

FMV.S requires the "F" extension.
FMV.D requires the "D" extension.
FMV.Q requires the "Q" extension.

Encoding:

These instructions are special cases of more general instructions. They are assembled identically to:

```
FSGNJ.S       FRegD,FReg1,FReg1     (inject)
FSGNJ.D       FRegD,FReg1,FReg1     (inject)
```

Floating Negate

General Forms:

```
FNEG.S      FRegD,FReg1  (single precision)
FNEG.D      FRegD,FReg1  (double precision)
FNEG.Q      FRegD,FReg1  (quad precision)
```

Examples:

```
FNEG.S      f2,f5           # f2 = -(f5) (32 bits)
FNEG.D      f2,f5           # f2 = -(f5) (64 bits)
FNEG.Q      f2,f5           # f2 = -(f5) (128 bits)
```

Description:

The value in FReg1 is negated and then copied into FRegD.

RISC-V Extensions:

FNEG.S requires the "F" extension.
FNEG.D requires the "D" extension.
FNEG.Q requires the "Q" extension.

Encoding:

These instructions are special cases of more general instructions. They are assembled identically to:

```
FSGNJN.S        FRegD,FReg1,FReg1      (negate)
FSGNJN.D        FRegD,FReg1,FReg1      (negate)
```

Floating Absolute Value

General Forms:

```
FABS.S        FRegD,FReg1   (single precision)
FABS.D        FRegD,FReg1   (double precision)
FABS.Q        FRegD,FReg1   (quad precision)
```

Examples:

```
FABS.S        f2,f5              # f2 = |f5|   (32 bits)
FABS.D        f2,f5              # f2 = |f5|   (64 bits)
FABS.Q        f2,f5              # f2 = |f5|   (128 bits)
```

Description:

The absolute value of the quantity in FReg1 is copied into FRegD.

RISC-V Extensions:

FABS.S requires the "F" extension.
FABS.D requires the "D" extension.
FABS.Q requires the "Q" extension.

Encoding:

These instructions are special cases of more general instructions. They are assembled identically to:

```
FSGNJX.S        FRegD,FReg1,FReg1      (exclusive-or)
FSGNJX.D        FRegD,FReg1,FReg1      (exclusive-or)
```

Floating Move To/From Integer Register (Single Precision)

General Forms:

```
FMV.X.W        RegD,FReg1
FMV.W.X        FRegD,Reg1
```

Examples:

```
FMV.X.W         x2,f5           # x2 = f5
FMV.W.X         f5,x2           # f5 = x2
```

Description:

32-bits are copied from a floating point register to an integer register, or from an integer register to a floating point register. The bits are unaltered; i.e., no conversion is performed.

In the case of RV64 and RV 128 (where the integer registers are larger than 32 bits), when the destination is an integer register, the upper bits of the integer register will be filled with zeros. When the source is an integer register, the upper bits will be ignored.

No error conditions can arise.

In an older version of the spec, the letter "S" was used in place of "W" in the opcode, i.e., "FMV.X.S" and "FMV.S.X".

RISC-V Extensions:

These instructions require the "F" extension.

Encoding:

This is an R-type instruction.

Floating Move To/From Integer Register (Double Precision)

General Forms:

```
FMV.X.D         RegD,FReg1
FMV.D.X         FRegD,Reg1
```

Examples:

```
FMV.X.D         x2,f5           # x2 = f5
FMV.D.X         f5,x2           # f5 = x2
```

Description:

64-bits are copied from a floating point register to an integer register, or from an integer register to a floating point register. The bits are unaltered; i.e., no conversion is performed.

In the case of RV 128 (where the integer registers are larger than 64 bits), when the destination is an integer register, the upper bits of the integer register will be filled with zeros. When the source is an integer register, the upper bits will be ignored.

No error conditions can arise.

RISC-V Extensions:

These instructions require the "D" extension.

Encoding:

This is an R-type instruction.

Note: The "Q" extension does not provide the following instructions. This is intentional.

```
FMV.X.Q         RegD,FReg1
FMV.Q.X         FRegD,Reg1
```

In order to move a quad precision value from a floating point register to an integer register, you'll need to move it to memory first, then move it to the integer register.

Floating Point Compare and Classify Instructions

Floating Point Comparison

General Forms:

```
FLT.S           RegD,FReg1,FReg2    (single precision)
FLE.S           RegD,FReg1,FReg2    (single precision)
FEQ.S           RegD,FReg1,FReg2    (single precision)

FLT.D           RegD,FReg1,FReg2    (double precision)
FLE.D           RegD,FReg1,FReg2    (double precision)
FEQ.D           RegD,FReg1,FReg2    (double precision)

FLT.Q           RegD,FReg1,FReg2    (quad precision)
FLE.Q           RegD,FReg1,FReg2    (quad precision)
FEQ.Q           RegD,FReg1,FReg2    (quad precision)
```

Examples:

```
    FLT.S       x2,f5,f6        # x2 = (f5<f6) ? 1 : 0
    FLE.S       x2,f5,f6        # x2 = (f5≤f6) ? 1 : 0
    FEQ.S       x2,f5,f6        # x2 = (f5=f6) ? 1 : 0

    FLT.D       x2,f5,f6        # x2 = (f5<f6) ? 1 : 0
    FLE.D       x2,f5,f6        # x2 = (f5≤f6) ? 1 : 0
    FEQ.D       x2,f5,f6        # x2 = (f5=f6) ? 1 : 0

    FLT.Q       x2,f5,f6        # x2 = (f5<f6) ? 1 : 0
    FLE.Q       x2,f5,f6        # x2 = (f5≤f6) ? 1 : 0
    FEQ.Q       x2,f5,f6        # x2 = (f5=f6) ? 1 : 0
```

Description:

Two floating point values in the floating point registers are compared and a value indicating the result is stored in an integer register. The instructions store a "1" if the relationship (<, ≤, or =) holds true and "0" if the relationship does not hold.

There is no need for > or ≥ instructions since the programmer can achieve the same result by using the < and ≤ instructions by simply swapping the two register operands.

Error Conditions:

If either operand is a Not-a-Number (NaN) value then a "0" will be stored in the destination register. This applies to all three comparison instructions (<, ≤, and =).

The IEEE spec requires that the < or ≤ should result in a "signaling NaN" if either operand is a NaN. For RISC-V, the instruction execution will never be interrupted, so a signaling NaN cannot be implemented by interrupting the instruction flow. RISC-V handles this case as follows: For < and ≤ comparisons, if either operand is NaN, then the NV (invalid operation) bit in the Floating Pointer Control and Status Register (FCSR) will be set to 1. Otherwise, the bit will be left unchanged.

The IEEE spec treats the = comparison differently, saying that if either operand is NaN, then the result should be a "quiet NaN". The RISC-V implements this a follows: For the = comparison, the NV (invalid operation) bit will never be modified.

RISC-V Extensions:

The ".S" instructions require the "F" extension.

The ".D" instructions require the "D" extension.
The ".Q" instructions require the "Q" extension.

Encoding:

This is an R-type instruction.

Floating Point Classify

General Forms:

```
FCLASS.S      RegD,FReg1    (single precision)
FCLASS.D      RegD,FReg1    (double precision)
FCLASS.Q      RegD,FReg1    (quad precision)
```

Examples:

```
FCLASS.S      x2,f5         # x2 = class(f5)
FCLASS.D      x2,f5         # x2 = class(f5)
FCLASS.Q      x2,f5         # x2 = class(f5)
```

Description:

The instruction looks at a floating point value and determines what sort of a number it is. For example, does the value represent +0.0, -0.0, Nan, +∞, -∞, etc?

The result of the classification of the floating point value will be stored in an integer register, as follows: All bits of the integer register will be cleared to "0", except that exactly one bit will be set to "1", to indicate what sort of thing the floating point register contains.

Here are the possible results that can be stored into the integer register:

Bit Set	Value Stored	Meaning
0	1	FReg1 contains −∞.
1	2	FReg1 contains a negative normal number.
2	4	FReg1 contains a negative subnormal number.
3	8	FReg1 contains −0.
4	16	FReg1 contains +0.
5	32	FReg1 contains a positive subnormal number.
6	64	FReg1 contains a positive normal number.
7	128	FReg1 contains +∞.
8	256	FReg1 contains a signaling NaN.
9	512	FReg1 contains a quiet NaN.

RISC-V Extensions:

The ".S" instructions require the "F" extension.
The ".D" instructions require the "D" extension.
The ".Q" instructions require the "Q" extension.

Encoding:

This is an R-type instruction.

Chapter 5: Register and Calling Conventions

Standard Usage of General Purpose Registers

The 32 bit version of the RISC-V architecture treats all registers identically so the assembly language programmer is free to use any register as he/she wishes. The only exception is register x0, which always contains zero, and can be used as a destination when the data should be discarded.

However, there is a RISC-V "standard calling convention" which specifies how registers will normally be used by the compiler and most assembly language programmers will follow this convention.

In the "C" (Compressed Instructions) extension, 16 bit instructions are supported, and each compressed instruction is a shorter, abbreviated form of a 32-bit instruction. The compressed instruction set was designed under the assumption that the registers will be used in the standard ways and only the most common instructions are given 16 bit equivalents. As such, the registers are not treated identically in the compressed instruction set.

For example, the 32 bit "call" instruction (JAL) can save the return address in any register, although the standard conventions mandate that register x1 will always be used to save the return address. Making use of this convention, the 16 bit version of the instruction (C.JAL) saves the return address in x1 and only this register. There is no compressed instruction to save the return address in any other register, which is reasonable since this is not something normally done.

The names of the general purpose registers are x0, x1, … x31. The registers are also given second, alternate names. The assembler will accept either name, so the programmer can use whichever name seems clearest.

Here are the registers:

Name	Other Name	Description	Saved Across Calls
x0	zero	Zero	
x1	ra	Return Address	
x2	sp	Stack Pointer	yes / callee saved
x3	gp	Global Pointer	
x4	tp	Thread Pointer	
x5	t0	Temp	
x6	t1	Temp	
x7	t2	Temp	
x8	s0,fp	Saved Reg / Frame Pointer	yes / callee saved
x9	s1	Saved Reg	yes / callee saved
x10	a0	Function argument	
x11	a1	Function argument	
x12	a2	Function argument	
x13	a3	Function argument	
x14	a4	Function argument	
x15	a5	Function argument	
x16	a6	Function argument	
x17	a7	Function argument	
x18	s2	Saved Reg	yes / callee saved
x19	s3	Saved Reg	yes / callee saved
x20	s4	Saved Reg	yes / callee saved
x21	s5	Saved Reg	yes / callee saved
x22	s6	Saved Reg	yes / callee saved
x23	s7	Saved Reg	yes / callee saved
x24	s8	Saved Reg	yes / callee saved
x25	s9	Saved Reg	yes / callee saved
x26	s10	Saved Reg	yes / callee saved
x27	s11	Saved Reg	yes / callee saved
x28	t3	Temp	
x29	t4	Temp	
x30	t5	Temp	
x31	t6	Temp	

The compressed instructions are designed to allow easy access to 8 registers, namely x8, x9, … x15. In the above table, these 8 registers are highlighted.

Saving Registers Across Calls

By convention, some registers are saved across function calls. For programs that follow the standard calling conventions (and we assume that almost all do), a function (say "foo") will call another function (say "bar"). The "caller" is foo and the "callee" is bar.

We normally use the terminology "callee saved" to indicate that a register will be preserved across a call. For example, function bar will not modify the register (or if it does, it will first save and then restore the register), so that the caller foo can rely on its value remaining unchanged by bar.

Presumably the caller foo will have local variables whose values must be preserved across a call to bar. Presumably the callee bar will need some temporary variables to complete its task and compute its result. If foo cannot trust bar to preserve its local variables, then foo will need to save registers and, after bar returns, restore their values. This saving and restoring is done to main memory (typically to the local stack frame), and such memory references are very slow. If foo relies on bar to do the saving and restoring the registers, then the burden is placed on bar to perform these slow memory tasks.

There are tradeoffs and generating optimal code is tricky. It may be most efficient for the caller foo to save the registers, since it may be the case that caller foo can save a register just once, make a number of calls to bar, and then restore the register. If the register had been saved in the callee bar, there would have been multiple saves and restores. On the other hand, it may be best for the saving and restoring to be performed in the callee bar. Perhaps bar will often take an execution path that does not disrupt the register, so the saving and restoring can be avoided altogether in many cases.

The tradeoff (whether to perform the saving in the caller or callee) is a decision that must be made for each of the registers in the caller foo. If both functions are programmed simultaneously by a clever programmer, then he/she can make ad hoc decisions about the saving and restoring of resisters in an attempt to produce optimal code.

However, almost all code is compiled and most compilers look at each function in complete isolation. Effectively, each function is separately compiled, so there is no

possibility of coordinating and optimizing the saving of registers between multiple functions.

Instead, the approach is to create a standard convention mandating which registers are saved by the callee bar and which are not. Registers that are not required by the convention to be preserved across a call are termed “caller saved”, which means that it is up to the caller foo to save and restore the registers, if they contain data that must be preserved across the function call to bar.

The convention dictates which registers are required to be preserved across a call, and it is assumed that compilers and programmers will follow the calling conventions. If they do, then the separately compiled or separately written code is interoperable.

Assuming that the conventions are to be followed when compiling some function foo, the compiler/programmer is free to use whichever register makes most sense. For example, a variable that must be preserved across function calls (e.g., to other functions like bar) will most likely be placed in a “callee saved” register. Therefore, the save and restore instructions are avoided in the caller’s code, and might be avoided altogether if bar doesn’t even use the register in question. On the other hand, the compiler/programmer might prefer to use a temporary (caller saved) register, especially if there are no calls within foo, or if the variable does not need to be saved across any calls that do occur.

By establishing a calling conventions about which registers are callee saved and which are not, the designers are making the important decision about the ratio between temporary callee saved registers. Assuming that there are enough registers in each class, then no register saving/restoring will be necessary.

But note that with recursive routines, there can be a conflict. Consider a local variable that must be preserved across a recursive call. By placing the variable in a temporary (caller saved) register there can be a problem. Since this register will not be saved across the recursive call, the function must save it before the call and restore it afterwards. On the other hand, if it is placed in a callee saved register we don’t have to bother with saving/restoring it around the recursive call. But since the function will be using a callee saved register, the function itself must save the previous value upon entry and restore it before return.

So either way, the register must be save and restored. This makes sense for recursive functions that have local variables that must be preserved across the recursive call.

The only question is whether to perform the save before the call instruction or after the call instruction, and likewise perform the restore after the return instruction or before the return instruction. The compiler makes this decision when it chooses to place the variable in a callee- or caller-saved register.

Note that compilers/programmers are sometimes free to violate the calling conventions order to produce superior code. This is only feasible when the compiler/programmer has access to both the called function (bar) and all potential callers (such as foo). In such cases, the compiler/programmer can choose to use the registers in a non-standard way to improve efficiency. For example, it may be that bar would benefit from having more temporary registers and none of the callers have data that must be preserved across the call to bar, so the compiler/programmer can use registers that would otherwise be "callee saved" as "caller saved". Compilers can keep all the details straight but programmers should be careful. Many assembly language programming bugs are the result of mistakes involving the proper saving and restoring of registers.

Register x0 - The Zero Register ("zero")

As mentioned earlier, register x0 is a dummy register. When read, its value is always zero; when stored into, the data is simply discarded.

Register x1 - The Return Address ("ra")

When a function is called, the return address must be saved so that the RETURN instruction can return to the correct location in the caller's code. In older computers, the CALL instruction stored the return address in memory on the stack; in modern ISAs (including RISC-V) the CALL instruction saves the return address in a register. The RETURN instruction retrieves the value from this register. The RETURN instruction is then nothing more than a JR (jump through register) instruction.

By convention, register x1 is used for this purpose.

Saving the return address in a register is faster than saving it on the stack since the memory WRITE and READ operations are avoided. This scheme is ideal for leaf functions. ("Leaf" functions do not call other functions and are thus leaf nodes in the

activation/calling tree.) Many function calls are to leaf functions so this saves a lot memory activity.

For non-leaf functions, the return address must be saved somewhere else. Since x1 will be used in subsequent calls, the return address in x1 must be moved somewhere else to avoid getting clobbered before it can be used in the RETURN instruction. The value can either be saved in a callee saved register or on the stack. (Moving the return address to a callee saved register may necessitate other register saving elsewhere, forcing a save to stack memory elsewhere. For recursive functions, there is no way to avoid using the in-memory stack; we can only shift around when the memory READs/WRITEs occurs.)

Register x2 - The Stack Pointer ("sp")

RISC-V assumes that x2 contains the stack top pointer and that the stack grows downward. The stack is assumed to always be quadword (16 byte) aligned. Some functions/methods may operate entirely using registers and may not need any stack space. The RISC-V design tries to facilitate such functions since they make no memory accesses. However, many functions will allocate stack space to hold return addresses, local variables, saved registers, etc. The local variables, etc., are stored in a "stack frame", which is sometimes called an "activation record." Recursive functions are known for needing stack space and allocating a new stack frame for each invocation.

A function that requires stack storage will grow the stack (always by a multiple of 16) by subtracting from the stack top pointer sp. Variables within the stack frame can be addressed using positive offsets from register x2. This organization motivates several of the compressed instructions, in which the offset must be positive and is never sign extended.

Register x2 is callee-saved, which means that any function that grows the stack to create a stack frame must also shrink it by an equal amount before returning. In other words, a function must have the net effect of popping everything that it pushed, leaving the stack as it found it, and effectively preserving the value of x2. Thus, x2 is said to be "callee-saved".

Register x3 - The Global Pointer ("gp")

Global variables are also called static variables, and are contrasted with local variables. Global variables are shared by all functions. Global variables remain in existence throughout the entire program, while local variables come into existence when a function is called and go out of existence when the function returns.

While local variables are often kept in registers or on the stack at unpredictable locations, global variables are placed at fixed locations in memory. Unfortunately these locations are often far away from the code that accesses them. PC-relative addressing can be used, but is often ineffective since the required offsets can be large. Absolute addresses can often be problematic too, since the absolute addresses can also be large, which would necessitate using extra instructions.

The solution supported by the RISC-V calling convention is to place the global variables together and initialize a register to point to this area. By convention, register x3 is used for this. Then the individual variables can be conveniently addressed by using a small offset from the global pointer.

The global pointer is typically initialized early in the program and never changed. So, in some sense, it is "callee-saved" although we didn't mark it as such in the table above.

Not all programs will use this technique, so this register may actually be used for something completely different.

Register x4 - The Thread Base Pointer ("tp")

In multi-threaded programs, each thread may have a collection of "thread-specific" variables, which are not local to any function. Instead, they are shared by all code running in this thread, but are invisible to other threads.

Thread-specific variables are distinct from global variables. There is only one copy of each global variable and each variable has a unique offset from the global base register, x3/gp. The thread base register (tp) has the same value in all code running in all threads, which causes all code to access that same set of variables, regardless of which thread it is executing in.

In a multi-threaded program, each thread has a unique register set. When the OS switches from one thread to another, all registers from the previously running thread will be saved to some area of memory and the newly active thread will have its registers restored from a different memory area. In general, the registers in one thread will be completely different from the registers in another thread, even though the threads are cooperating and executing in the exact same address space. For example, each thread will have its own stack, so the x2 (sp) register in each thread will contain a different value.

Since all threads share the global variables, register x3 (gp) will have the same value in all threads. In addition, each thread can have its own private set of variables, which are called "thread-specific" variables. By convention, this block of variables will be pointed to by register x4 (tp), so each thread will have a different value in its x4 register. A single section of code that accesses thread-specific variables will access a different set of variables, depending on which thread it is running in, and this works because each thread has a unique value in its x4 (tp) register.

Many applications are not multi-threaded, or contain no thread-specific variables. In such cases, it seems the convention leaves unspecified how x4 is to be used.

The thread pointer is typically initialized early in the execution of a new thread and never changed. So, in some sense, it is "callee-saved" although we didn't mark it as such in the table above.

Register x5-x7,x28-x31 - Temp Registers ("t0-t6")

By convention, there are 7 registers which are free to be used by any function without saving their previous contents. Typically the compiler will place the intermediate results of computation into temp registers. In some cases, the compiler may place local variables in the temp registers.

The temp registers are "caller saved", which means that the code within a function "foo" is not required to save their previous contents before using them. On the other hand, since this rule applies to all functions, it must be assumed that calling any other function (such as "bar") may result in these registers being changed. Thus, if the caller (foo) cares about the value of a temp register, it must save the register before calling bar.

Register x8,x9,x18-x27 - Saved Registers ("s0-s11")

By convention, there are 12 registers which are designated as callee-saved registers. The convention is that a function must not modify these registers. If some function wishes or needs to use one of these registers, that function must first save its previous value and then restore that value before returning.

This makes these register especially useful in storing local variables that must be preserved across calls to other functions. There is a reasonable chance that the called function will not touch these registers at all, so saving/restoring to main memory is often avoided completely.

Two of these registers (x8/s0 and x9/s1) are especially easy to access using the 16 bit compressed instructions. Therefore the compiler should prefer to use one of these two registers whenever possible.

Register x8 - Frame Pointer ("fp")

Register x8 (which is also s0, one of the callee-saved registers) is designated as the frame pointer and given a second name, "fp".

Not every function will need a frame pointer and, for a function that does not need a frame pointer, this register can be viewed as s0, just another callee-saved register.

To understand the purpose of a frame pointer, consider two sorts of function.

The first sort of function will have a small number of local variables. Furthermore, each of the local variables will have a fixed size, known at compile-time. For such a function, the stack frame can be laid out by the compiler and all offsets within the stack frame will be known, small numbers. This sort of function can access its variables in the stack frame by using positive offsets from the stack top pointer (x2). This sort of a function will not need a frame pointer.

The second sort of function may have a large number of local variables or may create a local variable (such as a dynamic array) whose size is not known until runtime when the function is actually called. In such a function, the compiler will not know

the layout of the frame. The local variables cannot be accessed using small positive offsets from the stack pointer register, x2.

The idea with using a frame pointer is this: Upon entry to the function, the frame pointer register will be set to a known, fixed value. Typically, this would be the old value of the stack pointer before the new stack frame is allocated. Then, the stack frame is allocated in a couple of steps. First the fixed sized variables are allocated by adjusting the stack top pointer. Then the dynamically sized variables are allocated, by decrementing the stack top pointer by values determined at runtime. After this initialization and creation of local variables, the frame pointer is left pointing to one end (the top, higher end) of the frame (or more precisely to the first byte of the previous frame) and the stack pointer is left pointing to the bottom, lower end of the frame. Offsets relative to the stack top pointer are problematic, since the compiler will not know exactly how much the stack pointer was adjusted. Instead, the local variables will be accessed using negative offsets relative to the frame pointer.

(In another approach, the local variables can be pushed onto the stack first. Then the frame pointer is initialized to the current stack top. Finally, the dynamically sized variables are pushed on to the stack. This allows the local variables to be accessed using a positive offset to the frame pointer. In the RISC-V design, positive offsets are preferred in the 16 bit compressed instruction format, so this might be a better approach.)

The frame pointer is callee saved. This means that the value of the register must be restored before the function returns. Typically, the function prologue will store the previous value of the fp register in the frame and the function epilogue will restore fp right before returning.

Register x10-x17 - Argument Registers (“a0-a7”)

Typically, arguments to functions are passed in registers and this group of registers is meant to be used for that purpose.

Also, a value that is to be returned from a function will usually be returned in a register.

The return value will be returned in x10 (a0), or if the value is too large to fit into a single register, it will be returned in the register pair x10-x11 (a0-a1). This is the same register(s) that is used to pass in the first argument.

Calling Conventions for Arguments

Arguments are normally passed to a function in registers and the calling convention details exactly how argument values are to be placed in registers.

If there are too many arguments, then the first arguments are passed in the registers, and the remainder are passed on the stack. Also, if an individual argument is too large, it will be passed in memory and the register will contain a pointer to this memory region.

If the returned value is small enough, it will be returned in a0-a1, otherwise it is returned in memory.

Here, we summarize the calling conventions for RV32, the 32 bit architecture.

- Floats are passed in the floating point registers, if they exist. Otherwise, floats are passed in the general purpose registers, just like ints.
- Other values brides floats (e.g., ints, chars, bools, pointers, structs) are passed in the 8 general purpose registers a0-a7, as long as they are no larger than 64 bits. So, if the value will fit into one or two registers, they will be passed in registers.
- Values smaller than 32 bits are sign-extended to 32 bits and passed in a register. [Apparently, they are “widened according to the sign of their type, then sign extended”; my interpretation may be incorrect. ???]
- Values of 32 bits are passed in a single register.
- Values of up to 64 bits in size are passed in a register pair. The least significant 32-bits will be placed in the register with the smaller number, in little endian style.

- Values larger than 64 bits are always passed "by reference". This means that the value is placed in a region of memory. This region is either in the caller's stack frame or is pushed on to the stack at the time of the function call. The address to this region appears in the list of arguments and is passed in a register, in place of the value. [It seems that an address is always used, regardless of whether or not there is room in the registers for the address.]

- If there are not enough registers, only the first few arguments will be passed in registers and the remainder will be passed in memory. The extra, remaining arguments will pushed on to the stack, so they will be at known offsets from the stack top pointer upon entry to the called function. Thus, they can be easily accessed using positive offsets from the frame pointer register (fp).

- If there are not quite enough registers, a 64 bit value can be split into two 32-bit parts, with the least significant word passed in a register and the most significant word passed in memory.

- Following the "C" language convention, arguments are always "passed by value", which means they are effectively local variables which can be modified; after the function returns, the modified value is lost. For example, structs passed as arguments will be copied and any changes to the stucco will be lost. If this is not what the programmer wants, then he/she can explicitly pass an address, in which case the pointer is "passed by value", and the value pointed to is effectively "passed by reference".

- All values passed in memory (e.g., arguments in the stack frame) are aligned, according to the size of the value.

- If the returned value is 64 bits or less, it will be returned in register a0-a1 (or the floating point registers, if applicable).

- If the returned value is larger than 64 bits, the caller will allocate memory space for the returned value (in the caller's stack frame) and will pass a pointer to this block of memory. The pointer will be passed as the first argument, implicitly inserted in front of the other arguments.

Six of these registers (x10-x15, i.e. a0-a5) are especially easy to access using the 16 bit compressed instructions. Therefore, the majority of functions (which have 6 or fewer arguments) can be called using registers alone.

Commentary: The argument registers are otherwise treated like temporaries, in that they are caller saved, i.e., not preserved across calls. If an argument register is not used for passing arguments, then it it is free to be used as a temporary work register.

It is unclear why it is necessary to differentiate between these two classes. Why can't temp registers be used to pass arguments in cases where there are a lot of arguments? If a function with many arguments truly needs a temporary register, then it can save some of its arguments in its stack frame. This is no extra work, since the caller would have had to save the values in its frame otherwise. And it will probably be the case that the callee function can be a bit smarter about what has to be saved in memory than the caller.

Floating Point Registers

Here are the floating point registers, with their intended usage. The 16 bit compressed instructions favor registers f8 - f15 and these are highlighted below.

Name	Other Name	Description	Saved Across Calls
f0	ft0	Temp	
f1	ft1	Temp	
f2	ft2	Temp	
f3	ft3	Temp	
f4	ft4	Temp	
f5	ft5	Temp	
f6	ft6	Temp	
f7	ft7	Temp	
f8	fs0	Saved Reg	yes / callee saved
f9	fs1	Saved Reg	yes / callee saved
f10	fa0	Function argument	
f11	fa1	Function argument	
f12	fa2	Function argument	
f13	fa3	Function argument	
f14	fa4	Function argument	
f15	fa5	Function argument	
f16	fa6	Function argument	
f17	fa7	Function argument	
f18	fs2	Saved Reg	yes / callee saved
f19	fs3	Saved Reg	yes / callee saved
f20	fs4	Saved Reg	yes / callee saved
f21	fs5	Saved Reg	yes / callee saved
f22	fs6	Saved Reg	yes / callee saved
f23	fs7	Saved Reg	yes / callee saved
f24	fs8	Saved Reg	yes / callee saved
f25	fs9	Saved Reg	yes / callee saved
f26	fs10	Saved Reg	yes / callee saved
f27	fs11	Saved Reg	yes / callee saved
f28	ft8	Temp	
f29	ft9	Temp	
f30	ft10	Temp	
f31	ft11	Temp	

Chapter 6: Compressed Instructions

Compressed Instructions

Normally, RISC-V instructions are 32 bits in length. However, the “C” (Compressed Instructions) extension adds instructions which are 16 bits long.

The “C” extension merely adds instructions to the ISA; all the other 32 bit instructions remain valid, unchanged and fully functional.

Each 16 bit instruction is an abbreviation for a longer 32 bit instruction. Thus, no new functionality is added. The purpose of the 16 bit instructions is to reduce code size: by replacing longer instructions with their equivalent shorter versions, code size is reduced.

Not all 32 bit instructions have an equivalent 16 bit counterpart. The goal of the RISC-V design is to provide 16 bit instructions for the most popular and frequently occurring 32 bit instructions, thereby reducing the code size as much as possible. A block of code in which all 32 bit instructions happen to have 16 bit counterparts can be reduced to half its size. But typical code sequences will contain some instructions which have no 16 bit counterparts, so the reduction in size will not be as great. The designers estimate an 25% reduction in code size.

In other ISA designs there is a “compressed mode”: When the processor is placed in compressed mode, the shorter instructions are executed. RISC-V does not work this way. The RISC-V design has carefully encoded the instructions so that the length of the instruction can be determined and the bytes fetched from memory can be unambiguously interpreted as either 16 bit or 32 bit instructions. Thus, the programmer can freely intermix long and short instructions.

Instruction Formats

There are several instruction formats used for the compressed instructions. In listing the formats below, we will use these abbreviations for fields:

------	Opcode Bits
DDDDD	RegD
DDD	RegD (x8..x15 only)
aaaaa	Reg 1
aaa	Reg1 (x8..x15 only)
bbbbb	Reg2
bbb	Reg2 (x8..x15 only)
VVVVV	Immediate Value
XXXXX	Offset

Here are the main instruction forms. (Each character indicates a single bit.)

Register Format

----aaaaabbbbb--

----DDDDDbbbbb--

Immediate Format

---VaaaaaVVVVV--

---VDDDDDVVVVV--

Wide Immediate Format

---VVVVVVVVDDD--

Stack-relative Store Format

---VVVVVVbbbbb--

Load Format

---VVVaaaVVDDD--

Store Format

---VVVaaaVVbbb--

Branch Format

---XXXaaaXXXXX--

Jump Format

---XXXXXXXXXXX--

These formats are schematic. For the individual instructions described in the following sections, we give the exact encoding, including the actual opcode bits, as well as other encoding constraints.

Load Instructions

C.LW - Load Word

General Form:

```
C.LW    RegD,Immed-6(Reg1)
```

Description:

Move a 32 bit value from memory into register RegD. To form the address, the Immed-6 value is zero-extended, multiplied by 4, and added as a positive offset to the base address in register Reg1.

Since the Immed-6 value is scaled by 4, this gives an effective range of 0 .. 252, in multiples of 4.

Registers RegD and Reg1 are restricted to x8 .. x15.

Encoding:

010VVVaaaVVDDD00

Where **DDD** = RegD.
Where **aaa** = Reg1.
Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
LW      RegD,Offset(Reg1)
```

C.LW - Load Doubleword

General Form:

```
C.LD    RegD,Immed-6(Reg1)
```

Description:

Move a 64 bit value from memory into register RegD. To form the address, the Immed-6 value is zero-extended, multiplied by 8, and added as a positive offset to the base address in register Reg1.

Since the Immed-6 value is scaled by 8, this gives an effective range of 0 .. 504, in multiples of 8.

Registers RegD and Reg1 are restricted to x8 .. x15.

Encoding:

011VVVaaaVVDDD00

Where **DDD** = RegD.
Where **aaa** = Reg1.
Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.
Only available for RV64 and RV128.

Equivalent 32 Bit Instruction:

```
LD          RegD,Offset(Reg1)
```

C.LQ - Load Quadword

General Form:

```
C.LQ      RegD,Immed-6(Reg1)
```

Description:

Move a 128 bit value from memory into register RegD. To form the address, the Immed-6 value is zero-extended, multiplied by 16, and added as a positive offset to the base address in register Reg1.

Since the Immed-6 value is scaled by 16, this gives an effective range of 0 .. 1,008, in multiples of 16.

Registers RegD and Reg1 are restricted to x8 .. x15.

Encoding:

001VVVaaaVVDDD00

Where **aaa** denotes Reg1.
Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.
Only available for RV128.

Equivalent 32 Bit Instruction:

```
LQ          RegD,Offset(Reg1)
```

C.FLW - Load Single Float

General Form:

```
C.FLW       FRegD,Immed-6(Reg1)
```

Description:

Move a 32 bit value from memory into floating point register FRegD. To form the address, the Immed-6 value is zero-extended, multiplied by 4, and added as a positive offset to the base address in general purpose register Reg1.

Since the Immed-6 value is scaled by 4, this gives an effective range of 0 .. 252, in multiples of 4.

Register FRegD is restricted to f8 .. f15.

Register Reg1 is restricted to x8 .. x15.

Encoding:

011VVVaaaVVDDD00

Where **aaa** denotes Reg1.
Where **VVVVVV** = Immed-6.

Availability:

Requires “C” (Compressed Instruction) extension.
Requires “F” (Single Precision Floating Point) extension.
Only available for RV32.

Equivalent 32 Bit Instruction:

```
FLW         FRegD,Offset(Reg1)
```

C.FLD - Load Double Float

General Form:

```
C.FLD       FRegD,Immed-6(Reg1)
```

Description:

Move a 64 bit value from memory into floating point register FRegD. To form the address, the Immed-6 value is zero-extended, multiplied by 8, and added as a positive offset to the base address in general purpose register Reg1.

Since the Immed-6 value is scaled by 8, this gives an effective range of 0 .. 504, in multiples of 8.

Register FRegD is restricted to f8 .. f15.

Register Reg1 is restricted to x8 .. x15.

Encoding:

001VVVaaaVVDDD00

Where **aaa** denotes Reg1.

Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.

Requires "D" (Double Precision Floating Point) extension.

Only available for RV32 and RV64.

Equivalent 32 Bit Instruction:

```
FLD         FRegD,Offset(Reg1)
```

C.LWSP - Load Word from Stack Frame

General Form:

```
C.LWSP      RegD,Immed-6
```

Description:

Move a 32 bit value from memory into register RegD. To form the address, the Immed-6 value is zero-extended, multiplied by 4, and added to the stack pointer (x2).

Since the Immed-6 value is scaled by 4, this gives an effective range of 0 .. 252, in multiples of 4.

The destination register can be any of the 32 registers, except x0.

Encoding:

010VDDDDDVVVVV10

Where **DDDDD** = RegD and cannot be x0=00000.

Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
LW          RegD,Offset(x2)
```

C.LDSP - Load Doubleword from Stack Frame

General Form:

```
C.LDSP      RegD,Immed-6
```

Description:

Move a 64 bit value from memory into register RegD. To form the address, the Immed-6 value is zero-extended, multiplied by 8, and added to the stack pointer (x2).

Since the Immed-6 value is scaled by 8, this gives an effective range of 0 .. 504, in multiples of 8.

The destination register can be any of the 32 registers, except x0.

Encoding:

011VDDDDDVVVVV10

Where **DDDDD** = RegD and cannot be x0=00000.

Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.

Only available for RV64 and RV128.

Equivalent 32 Bit Instruction:

```
LD          RegD,Offset(x2)
```

C.LQSP - Load Quadword from Stack Frame

General Form:

```
C.LQSP      RegD,Immed-6
```

Description:

Move a 128 bit value from memory into register RegD. To form the address, the Immed-6 value is zero-extended, multiplied by 16, and added to the stack pointer (x2).

Since the Immed-6 value is scaled by 16, this gives an effective range of 0 .. 1008 in multiples of 16.

The destination register can be any of the 32 registers, except x0.

Encoding:

001VDDDDDVVVVV10

Where **DDDDD** = RegD and cannot be x0=00000.

Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.

Only available for RV128.

Equivalent 32 Bit Instruction:

```
LQ          RegD,Offset(x2)
```

C.FLWSP - Load Single Float from Stack Frame

General Form:

```
C.FLWSP     FRegD,Immed-6
```

Description:

Move a 32 bit value from memory into floating point register FRegD. To form the address, the Immed-6 value is zero-extended, multiplied by 4, and added to the stack pointer (x2).

Since the Immed-6 value is scaled by 4, this gives an effective range of 0 .. 252, in multiples of 4.

The destination register can be any of the 32 floating point registers.

Encoding:

011VDDDDDVVVVV10

Where **DDDDD** = FRegD.

Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.

Requires "F" (Single Precision Floating Point) extension.

Only available for RV32.

Equivalent 32 Bit Instruction:

```
FLW         FRegD,Offset(x2)
```

C.FLDSP - Load Double Float from Stack Frame

General Form:

```
C.FLDSP     FRegD,Immed-6
```

Description:

Move a 64 bit value from memory into floating point register FRegD. To form the address, the Immed-6 value is zero-extended, multiplied by 8, and added to the stack pointer (x2).

Since the Immed-6 value is scaled by 8, this gives an effective range of 0 .. 504, in multiples of 8.

The destination register can be any of the 32 floating point registers.

Encoding:

001VDDDDDVVVVV10

Where **DDDDD** = FRegD.

Where **VVVVVV** = Immed-6.

Availability:

Requires “C” (Compressed Instruction) extension.

Requires “D” (Double Precision Floating Point) extension.

Only available for RV32 and RV64.

Equivalent 32 Bit Instruction:

```
FLD         FRegD,Offset(x2)
```

Store Instructions

C.SW - Store Word

General Form:

```
C.SW        Reg2,Immed-6(Reg1)
```

Description:

Move a 32 bit value from register Reg2 to memory. To form the address, the Immed-6 value is zero-extended, multiplied by 4, and added as a positive offset to the base address in register Reg1.

Since the Immed-6 value is scaled by 4, this gives an effective range of 0 .. 252, in multiples of 4.

Registers Reg1 and Reg2 are restricted to x8 .. x15.

Encoding:

110VVVaaaVVbbb00

Where **aaa** denotes Reg1 and **bbb** denotes Reg2.

Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
SW          Reg2,Offset(Reg1)
```

C.SD - Store Doubleword

General Form:

```
C.SD        Reg2,Immed-6(Reg1)
```

Description:

Move a 64 bit value from register Reg2 to memory. To form the address, the Immed-6 value is zero-extended, multiplied by 8, and added as a positive offset to the base address in register Reg1.

Since the Immed-6 value is scaled by 8, this gives an effective range of 0 .. 504, in multiples of 8.

Registers Reg1 and Reg2 are restricted to x8 .. x15.

Encoding:

111VVVaaaVVbbb00

Where **aaa** denotes Reg1 and **bbb** denotes Reg2.

Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.

Only available for RV64 and RV128.

Equivalent 32 Bit Instruction:

```
SD          Reg2,Offset(Reg1)
```

C.SQ - Store Quadword

General Form:

```
C.SQ        Reg2,Immed-6(Reg1)
```

Description:

Move a 128 bit value from register Reg2 to memory. To form the address, the Immed-6 value is zero-extended, multiplied by 16, and added as a positive offset to the base address in register Reg1.

Since the Immed-6 value is scaled by 16, this gives an effective range of 0 .. 1,008, in multiples of 16.

Registers Reg1 and Reg2 are restricted to x8 .. x15.

Encoding:

101VVVaaaVVbbb00

Where **aaa** denotes Reg1 and **bbb** denotes Reg2.
Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.
Only available for RV128.

Equivalent 32 Bit Instruction:

```
SQ          Reg2,Offset(Reg1)
```

C.FSW - Store Single Float

General Form:

```
C.FSW       FReg2,Immed-6(Reg1)
```

Description:

Move a 32 bit value from floating point register FReg2 to memory. To form the address, the Immed-6 value is zero-extended, multiplied by 4, and added as a positive offset to the base address in general purpose register Reg1.

Since the Immed-6 value is scaled by 4, this gives an effective range of 0 .. 252, in multiples of 4.

Register Reg1 is restricted to x8 .. x15. Register FReg2 is restricted to f8 .. f15.

Encoding:

111VVVaaaVVbbb00

Where **aaa** denotes Reg1 and **bbb** denotes FReg2.
Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.
Requires "F" (Single Precision Floating Point) extension.
Only available for RV32.

Equivalent 32 Bit Instruction:

```
FSW         FReg2,Offset(Reg1)
```

C.FSD - Store Double Float

General Form:

```
C.FSD       FReg2,Immed-6(Reg1)
```

Description:

Move a 64 bit value from floating point register FReg2 to memory. To form the address, the Immed-6 value is zero-extended, multiplied by 8, and added as a positive offset to the base address in general purpose register Reg1.

Since the Immed-6 value is scaled by 8, this gives an effective range of 0 .. 504, in multiples of 8.

Register Reg1 is restricted to x8 .. x15. Register FReg2 is restricted to f8 .. f15.

Encoding:

101VVVaaaVVbbb00

Where **aaa** denotes Reg1 and **bbb** denotes FReg2.
Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.
Requires "D" (Double Precision Floating Point) extension.
Only available for RV32 and RV64.

Equivalent 32 Bit Instruction:

```
FSD         FReg2,Offset(Reg1)
```

C.SWSP - Store Word to Stack Frame

General Form:

```
C.SWSP      Reg2,Immed-6
```

Description:

Move a 32 bit value from register Reg2 to memory. To form the address, the Immed-6 value is zero-extended, multiplied by 4, and added to the stack pointer (x2).

Since the Immed-6 value is scaled by 4, this gives an effective range of 0 .. 252, in multiples of 4.

The source register can be any of the 32 general purpose registers.

Encoding:

110VVVVVVbbbbb10

Where **bbbbb** denotes Reg2.

Where **VVVVVV** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
SW          Reg2,Offset(x2)
```

C.SDSP - Store Doubleword to Stack Frame

General Form:

```
C.SDSP      Reg2,Immed-6
```

Description:

Move a 64 bit value from register Reg2 to memory. To form the address, the Immed-6 value is zero-extended, multiplied by 8, and added to the stack pointer (x2).

Since the Immed-6 value is scaled by 8, this gives an effective range of 0 .. 504, in multiples of 8.

The source register can be any of the 32 general purpose registers.

Encoding:

111VVVVVVbbbbb10

Where **bbbbb** denotes Reg2.
Where **vvvvvv** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.
Only available for RV64 and RV128.

Equivalent 32 Bit Instruction:

```
SD          Reg2,Offset(x2)
```

C.SQSP - Store Quadword to Stack Frame

General Form:

```
C.SQSP      Reg2,Immed-6
```

Description:

Move a 128 bit value from register Reg2 to memory. To form the address, the Immed-6 value is zero-extended, multiplied by 16, and added to the stack pointer (x2).

Since the Immed-6 value is scaled by 16, this gives an effective range of 0 .. 1,016, in multiples of 16.

The source register can be any of the 32 general purpose registers.

Encoding:

101vvvvvvbbbbb10

Where **bbbbb** denotes Reg2.
Where **vvvvvv** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.
Only available for RV128.

Equivalent 32 Bit Instruction:

```
SQ          Reg2,Offset(x2)
```

C.FSWSP - Store Single Float to Stack Frame

General Form:

```
C.FSWSP     FReg2,Immed-6
```

Description:

Move a 32 bit value from register FReg2 to memory. To form the address, the Immed-6 value is zero-extended, multiplied by 4, and added to the stack pointer (x2).

Since the Immed-6 value is scaled by 4, this gives an effective range of 0 .. 252, in multiples of 4.

The source register can be any of the 32 floating point registers.

Encoding:

111vvvvvvbbbbb10

Where **bbbbb** denotes FReg2.
Where **vvvvvv** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.
Requires "F" (Single Precision Floating Point) extension.
Only available for RV32.

Equivalent 32 Bit Instruction:

```
FSW         FReg2,Offset(x2)
```

C.FSDSP - Store Double Float to Stack Frame

General Form:

```
C.FSDSP     FReg2,Immed-6
```

Description:

Move a 64 bit value from register FReg2 to memory. To form the address, the Immed-6 value is zero-extended, multiplied by 8, and added to the stack pointer (x2).

Since the Immed-6 value is scaled by 8, this gives an effective range of 0 .. 504, in multiples of 8.

The source register can be any of the 32 floating point registers.

Encoding:

101vvvvvvbbbbb10

Where **bbbbb** denotes FReg2.
Where **vvvvvv** = Immed-6.

Availability:

Requires "C" (Compressed Instruction) extension.

Requires “D” (Double Precision Floating Point) extension.
Only available for RV32 and RV64.

Equivalent 32 Bit Instruction:

```
FSD         FReg2,Offset(x2)
```

Jump, Call, and Conditional Branch Instructions

C.J - Jump (PC-relative)

General Form:

```
C.J         Immed-11
```

Description:

A jump is taken using a PC-relative offset. The 11 bit immediate value is multiplied by two (since instructions must be halfword aligned), sign-extended, and added to the program counter (PC), to give the target address.

Since the Immed-11 value is scaled by 2, this gives an effective range of -2,048 .. +2,046, in multiples of 2.

Encoding:

101XXXXXXXXXXX01

Where **XXXXXXXXXXX** denotes the Immed-11 offset.

Availability:

Requires “C” (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
JAL         x0,Offset
```

C.JAL - Jump and Link / Call (PC-relative)

General Form:

```
C.JAL       Immed-11
```

Description:

A function call is taken using a PC-relative offset. The 11 bit immediate value is multiplied by two (since instructions must be halfword aligned), sign-extended, and added to the program counter (PC), to give the target address.

Since the Immed-11 value is scaled by 2, this gives an effective range of -2,048 .. +2,046, in multiples of 2.

The return address (i.e., the address of the instruction following the C.JAL) is saved in register x1 (also known as register ra).

Encoding:

001XXXXXXXXXX01

Where **XXXXXXXXXXX** denotes the Immed-11 offset.

Availability:

Requires "C" (Compressed Instruction) extension.
Only available for RV32.

Equivalent 32 Bit Instruction:

```
JAL         x1,Offset
```

C.JR - Jump Register

General Form:

```
C.JR        Reg1
```

Description:

A jump is taken to the address in register Reg1.

Register Reg1 can be x1, x2, … x31, i.e., any general purpose register except x0.

Encoding:

1000aaaaa0000010

Where **aaaaa** denotes Reg1 and cannot be x0=00000.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
JAL         x0,Reg1,0
```

C.JALR - Jump Register and Link / Call

General Form:

```
C.JALR      Reg1
```

Description:

A function call is taken to the address in register Reg1.

The return address (i.e., the address of the instruction following the C.JALR) is saved in register x1 (also known as register ra).

Register Reg1 can be x1, x2, … x31, i.e., any general purpose register except x0.

Encoding:

1001aaaaa0000010

Where **aaaaa** denotes Reg1 and cannot be x0=00000.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
JALR        x1,Reg1,0
```

C.BEQZ - Branch if Zero (PC-relative)

General Form:

```
C.BEQZ      Reg1,Immed-8
```

Description:

A conditional branch is taken using a PC-relative offset. The branch is taken if the value in register Reg1 is zero.

The 8 bit immediate value is multiplied by two (since instructions must be halfword aligned), sign-extended, and added to the program counter (PC), to give the target address.

Since the Immed-8 value is scaled by 2, this gives an effective range of -256 .. +254, in multiples of 2.

Encoding:

110XXXaaaXXXXX01

Where **XXXXXXXX** denotes the Immed-8 offset.

Where **aaa** denotes Reg1.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
BEQ         Reg1,x0,Offset
```

C.BNEQZ - Branch if Not Zero (PC-relative)

General Form:

```
C.BNEQZ     Reg1,Immed-8
```

Description:

A conditional branch is taken using a PC-relative offset. The branch is taken if the value in register Reg1 is not zero.

The 8 bit immediate value is multiplied by two (since instructions must be halfword aligned), sign-extended, and added to the program counter (PC), to give the target address.

Since the Immed-8 value is scaled by 2, this gives an effective range of -256 .. +254, in multiples of 2.

Encoding:

111XXXaaaXXXXX01

Where **XXXXXXXX** denotes the Immed-8 offset.
Where **aaa** denotes Reg1.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
BNE         Reg1,x0,Offset
```

Load Constant into a Register

C.LI - Load Immediate

General Form:

```
C.LI        RegD,Immed-6
```

Description:

Load an immediate value in the range -32 .. +31 into register RegD.

The target register can be x1, x2, … x31, i.e., any general purpose register except x0.

The 6 bit immediate value is sign-extended.

Encoding:

010VDDDDDVVVVV01

Where **VVVVVV** denotes the Immed-6 value.
Where **DDDDD** denotes RegD, and cannot be x0=00000.

Availability:

Requires “C” (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
ADDI        RegD,x0,Value
```

C.LUI - Load Upper Immediate

General Form:

```
C.LUI       RegD,Immed-6
```

Description:

This instruction is a short form for the LUI instruction, but with a restricted range of values. This instruction loads bits [17:12] of the destination register. The upper bits [... :18] are filled with the sign extension. The lower 12 bits [11:0] are filled with zeros.

To put it another way, the immediate value is sign extended, multiplied by 4,096, and loaded into the destination register. Since the Immed-6 value is scaled by 2^{12} = 4,096, this gives an effective range of values of -131,040 .. +126,976, in multiples of 4,096.

The target register can be any general purpose register except x0 or x2. (Register x2 is normally used for the stack pointer.)

Encoding:

011VDDDDDVVVVV01

Where **VVVVVV** denotes the Immed-6 value.

Where **DDDDD** denotes RegD, and cannot be x0=00000 or x2=00010.

Availability:

Requires “C” (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
LUI         RegD,Value
```

Arithmetic, Shift, and Logic Instructions

C.ADDI - Add Immediate

General Form:

```
C.ADDI        RegD,Immed-6
```

Description:

Adds an immediate value in the range -32 .. +31 to register RegD.

The register can be x1, x2, … x31, i.e., any general purpose register except x0.

The 6 bit immediate value is sign-extended.

Encoding:

000VDDDDDVVVVV01

Where **VVVVVV** denotes the Immed-6 value and cannot be zero.
Where **DDDDD** denotes RegD, and cannot be x0=00000.

Comment:

If **VVVVVV**=000000 and **DDDDD**=00000, this encoding is identical to C.NOP.
If **VVVVVV**≠000000 and **DDDDD**=00000, this encoding can be a hint.
If **VVVVVV**=000000 and **DDDDD**≠00000, this encoding is reserved.

Availability:

Requires “C” (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
ADDI          RegD,RegD,Value
```

C.ADDIW - Add Immediate (Word)

General Form:

```
C.ADDIW       RegD,Immed-6
```

Description:

Adds an immediate value in the range -32 .. +31 to register RegD.

The register can be x1, x2, … x31, i.e., any general purpose register except x0.

The 6 bit immediate value is sign-extended.

The previously described C.ADDI instruction performs the addition in either 32, 64, or 128 bits, as determined by the architecture, RV32, RV64, or RV128.

This instruction C.ADDIW will truncate the result to 32 bits, then sign extended the 32 bits to the full register size.

A value of zero is allowed. This will simply force a larger value into a 32 bit value, sign extending to the full register size.

Encoding:

001VDDDDDVVVVV01

Where **VVVVVV** denotes the Immed-6 value. (Zero is allowed).
Where **DDDDD** denotes RegD, and cannot be x0=00000.

Comment:

If **DDDDD**=00000, this encoding can be a hint.

Availability:

Requires "C" (Compressed Instruction) extension.
Only available for RV64 and RV128.

Equivalent 32 Bit Instruction:

```
ADDIW          RegD,RegD,Value
```

C.ADDI16SP - Add Immediate to Stack Pointer

General Form:

```
C.ADDI16SP        Immed-6
```

Description:

This instruction is used to grow or shrink the stack by adjusting the stack pointer register.

The 6 bit immediate value is multiplied by 16, sign-extended, and added to the stack pointer register (x2).

It is assumed that the stack pointer is always quadword aligned, i.e., a multiple of 16. Since the Immed-6 value is scaled by 16, this gives an effective adjustment value of -512 .. +496, in multiples of 16.

A value of zero is not allowed, since adjusting the stack pointer by zero is pointless.

Encoding:

011V00010VVVVV01

Where **VVVVVV** denotes the Immed-6 value, and cannot be 000000.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
ADDI        x2,x2,Value
```

C.ADDI4SPN - Add Immediate to Stack Pointer

General Form:

```
C.ADDI14SPN      RegD,Immed-8
```

Description:

This instruction is used to load the address of a local stack variable into a register. We assume the variable is located with the stack frame, that the stack grows downward, and that the stack pointer points to the lowest address. Therefore, all local variables (located within the stack frame) will be accessed with positive offsets from register x2 (the stack pointer).

The 8 bit immediate value is multiplied by 4, zero-extended, and added to the stack pointer register (x2), and moved into the destination register RegD.

Since the Immed-8 value is scaled by 4, this gives an offset in the range -512 .. +508, in multiples of 4.

The destination register is limited to x8, x9, … x15.

A value of zero is not allowed, since a MV instruction can be used instead.

Encoding:

000VVVVVVVVDDD00

Where **VVVVVVVV** denotes Immed-8, and cannot be 00000000.
Where **DDD** denotes RegD.

Availability:

Requires “C” (Compressed Instruction) extension.
Available only for RV32 and RV64.

Equivalent 32 Bit Instruction:

```
ADDI        RegD,x2,Value
```

C.SLLI - Shift Left Logical (Immediate)

General Form:

```
C.SLLI      RegD,Immed-6
```

Description:

This instruction shifts the value in a register RegD left by an amount specified by the Immed-6 field.

For RV32 (32-bit architectures), the shift amount must be 1..31 (i.e., 000001-011111).

For RV64 (64-bit architectures), the shift amount must be 1..63 (i.e., 000001-111111).

For RV128 (128-bit architectures), the shift amount may be 1..64. (Shift amounts of 1..63 are encoded as 000001-111111; a shift amount 64 is encoded as 000000.)

The destination register can be x1, x2, … x31, i.e., any general purpose register except x0.

Encoding:

000VDDDDDVVVVV10

Where **VVVVVV** denotes Immed-6. See restrictions in the description.
Where **DDDDD** denotes RegD, and cannot be x0=00000.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
SLLI        RegD,RegD,Value
```

C.SRLI - Shift Right Logical (Immediate)

General Form:

```
C.SRLI      RegD,Immed-6
```

Description:

This instruction shifts the value in a register RegD right by an amount specified by the Immed-6 field. This is a "logical shift, i.e., zeros are shifted in from the left.

For RV32 (32-bit architectures), the shift amount must be 1..31 (i.e., 000001-011111).

For RV64 (64-bit architectures), the shift amount must be 1..63 (i.e., 000001-111111).

For RV128 (128-bit architectures), the possible shift amounts are 1..31, 64, and 96-127. [My reading of the spec suggests that the encoding of the shift amount is this: 000001-011111 means 1-31. 000000 means 64. 100000-111111 is interpreted as a signed value in the range -32..-1, which we can call X. The shift amount is right by 128-X bits. This gives a range of 96..127, resp.]

The destination register can be x8, x9 ,… x15.

Encoding:

100V00DDDVVVVV01

Where **VVVVVV** denotes Immed-6. See restrictions in the description.
Where **DDD** denotes RegD.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
SRLI        RegD,RegD,Value
```

C.SRAI - Shift Right Arithmetic (Immediate)

General Form:

```
C.SRAI      RegD,Immed-6
```

Description:

This instruction shifts the value in a register RegD right by an amount specified by the Immed-6 field. This is a "arithmetic shift, i.e., the sign bit is repeatedly shifted in from the left.

For RV32 (32-bit architectures), the shift amount must be 1..31 (i.e., 000001-011111).

For RV64 (64-bit architectures), the shift amount must be 1..63 (i.e., 000001-111111).

For RV128 (128-bit architectures), the possible shift amounts are 1..31, 64, and 96-127. [My reading of the spec suggests that the encoding of the shift amount is this: 000001-011111 means 1-31. 000000 means 64. 100000-111111 is interpreted as a signed value in the range -32..-1, which we can call X. The shift amount is right by 128-X bits. This gives a range of 96..127, resp.]

The destination register can be x8, x9 ,… x15.

Encoding:

100V01DDDVVVVV01

Where **VVVVVV** denotes Immed-6. See restrictions in the description.
Where **DDD** denotes RegD.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
SRAI        RegD,RegD,Value
```

C.ANDI - Logical AND (Immediate)

General Form:

```
C.ANDI      RegD,Immed-6
```

Description:

This instruction performs the logical AND operation with the value in a register and writes the result to that same register.

The Immed-6 value is sign-extended.

The register RegD can be x8, x9 ,… x15.

Encoding:

100V10DDDVVVVV01

Where **VVVVVV** denotes Immed-6.
Where **DDD** denotes RegD.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
ANDI        RegD,RegD,Value
```

C.MV - Move Register to Register

General Form:

```
C.MV        RegD,Reg2
```

Description:

This instruction moves a value from Reg2 to RegD.

The registers RegD and Reg2 can be x1, x2, … x31, i.e., any general purpose register except x0.

Encoding:

1000DDDDDbbbbb10

Where **DDDDD** denotes RegD.

Where **bbbbb** denotes Reg2.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
ADD         RegD,x0,Reg2
```

C.ADD - Add Register to Register

General Form:

```
C.ADD       RegD,Reg2
```

Description:

This instruction adds the values in RegD and Reg2 and places the result in RegD.

The registers RegD and Reg2 can be x1, x2, … x31, i.e., any general purpose register except x0.

Encoding:

1001DDDDDbbbbb10

Where **DDDDD** denotes RegD.

Where **bbbbb** denotes Reg2.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
ADD         RegD,RegD,Reg2
```

C.AND - Add Register to Register

General Form:

```
C.AND       RegD,Reg2
```

Description:

This instruction logically ANDs the values in RegD and Reg2 and places the result in RegD.

The registers RegD and Reg2 can be x8, x9, … x15.

Encoding:

100011DDD11bbb01

Where **DDD** denotes RegD.

Where **bbb** denotes Reg2.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
AND         RegD,RegD,Reg2
```

C.OR - Add Register to Register

General Form:

```
C.OR        RegD,Reg2
```

Description:

This instruction logically ORs the values in RegD and Reg2 and places the result in RegD.

The registers RegD and Reg2 can be x8, x9, … x15.

Encoding:

100011DDD10bbb01

Where **DDD** denotes RegD.

Where **bbb** denotes Reg2.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
OR          RegD,RegD,Reg2
```

C.XOR - Add Register to Register

General Form:

```
C.XOR       RegD,Reg2
```

Description:

This instruction logically XORs the values in RegD and Reg2 and places the result in RegD.

The registers RegD and Reg2 can be x8, x9, … x15.

Encoding:

100011DDD01bbb01

Where **DDD** denotes RegD.

Where **bbb** denotes Reg2.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
XOR         RegD,RegD,Reg2
```

C.SUB - Add Register to Register

General Form:

```
C.SUB       RegD,Reg2
```

Description:

This instruction subtracts the value in Reg2 from the value in RegD and places the result in RegD.

The registers RegD and Reg2 can be x8, x9, … x15.

Encoding:

100011DDD00bbb01

Where **DDD** denotes RegD.

Where **bbb** denotes Reg2.

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
SUB         RegD,RegD,Reg2
```

C.ADDW - Add Register to Register

General Form:

```
C.ADDW      RegD,Reg2
```

Description:

This instruction adds the value in Reg2 to the value in RegD and places the result in RegD. The result value is truncated to 32 bits and then sign extended to the full length of the register.

The registers RegD and Reg2 can be x8, x9, … x15.

Encoding:

100111DDD01bbb01

Where **DDD** denotes RegD.

Where **bbb** denotes Reg2.

Availability:

Requires "C" (Compressed Instruction) extension.

Available for RV64 and RV128 only.

Equivalent 32 Bit Instruction:

```
ADDW        RegD,RegD,Reg2
```

C.SUBW - Add Register to Register

General Form:

```
C.SUBW      RegD,Reg2
```

Description:

This instruction subtracts the value in Reg2 from the value in RegD and places the result in RegD. The result value is truncated to 32 bits and then sign extended to the full length of the register.

The registers RegD and Reg2 can be x8, x9, … x15.

Encoding:

100111DDD00bbb01

Where **DDD** denotes RegD.

Where **bbb** denotes Reg2.

Availability:

Requires "C" (Compressed Instruction) extension.

Available for RV64 and RV128 only.

Equivalent 32 Bit Instruction:

```
SUBW        RegD,RegD,Reg2
```

Miscellaneous Instructions

C.NOP - Nop Instruction

General Form:

```
C.NOP
```

Description:

This instruction does nothing. The encoding can be viewed as a special case of the C.ADDI instruction where the destination is x0 and the immediate value is 0.

Encoding:

0000000000000001

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
ADDI      x0,x0,0
```

C.EBREAK - Debugging Break Instruction

General Form:

```
C.EBREAK
```

Description:

This instruction is used by debuggers. The idea is that a debugger will temporarily replace some instruction with this instruction. When executed, an exception will occur, allowing the debugger to get control.

Encoding:

1001000000000010

Availability:

Requires "C" (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

```
EBREAK
```

C.ILLEGAL - Illegal Instruction

General Form:

```
C.ILLEGAL
```

It is doubtful that most assemblers would recognize the opcode "C.ILLEGAL", but we include it anyway.

Description:

This pattern is defined as being illegal. Any attempt to execute it will cause an "illegal instruction" exception. All zeros was intentionally chosen so that any attempt to branch into unpopulated or dysfunctional memory (which often reads as zeros) will result in an immediate trap.

Encoding:

0000000000000000

Availability:

Requires “C” (Compressed Instruction) extension.

Equivalent 32 Bit Instruction:

`ILLEGAL`

Instruction Encoding

The following table lists all the compressed instructions and is ordered in an attempt to show the complete coverage of the 16 bit instruction encoding space.

Some bit patterns are reserved by the RISC-V spec for future use; such patterns are marked <Reserved> and should not be used by implementors.

Some bit patterns are unassigned and available for specific implementations to define as they wish; such patterns are marked <NSE> (Nonstandard Extension).

Some bit patterns are marked as <Hint>; implementors are free to use these bit patterns to encode hints to the processor to the improve execution speed. In all cases, the patterns marked <Hint> are equivalent to instructions that involve modifying a register and that specifically disallow the use of register x0 as the destination register. Thus, implementors are free to ignore the <Hint> patterns and execute them in the natural way, by computing a result but sending that values to the zero register x0, causing these instructions to behave as nop instructions.

Recall that the least significant 2 bits are used to distinguish 16-bit instructions from longer instructions. If the least significant 2 bits are 11, then the instruction is not a 16 bit compressed instruction.

This table is more-or-less sorted on least significant 2 bits (00, 01, 10), followed by the most significant bits in order.

C.ILLEGAL	**0000000000000000**	
<Reserved>	**0000000000DDD00**	RegD ≠ 0
C.ADDI4SPN	**000VVVVVVVVDDD00**	Value ≠ 0

C.FLD	**001VVVaaaVVDDD00**	RV32 and RV64 only
C.LQ	**001VVVaaaVVDDD00**	RV128 only
C.LW	**010VVVaaaVVDDD00**	
C.FLW	**011VVVaaaVVDDD00**	RV32 only
C.LD	**011VVVaaaVVDDD00**	RV64 and RV128 only
<Reserved>	**100XXXXXXXXXX00**	
C.FSD	**101VVVaaaVVbbb00**	RV32 and RV64 only
C.SQ	**101VVVaaaVVbbb00**	RV128 only
C.SW	**110VVVaaaVVbbb00**	
C.FSW	**111VVVaaaVVbbb00**	RV32 only
C.SD	**111VVVaaaVVbbb00**	RV64 and RV128 only
C.ADDI	**000VDDDDDVVVVV01**	RegD ≠ 0; Value ≠ 0
<Reserved>	**0000DDDDD0000001**	RegD ≠ 0; Value = 0
<Hint>	**000V00000VVVVV01**	RegD = 0; Value ≠ 0
C.NOP	**0000000000000001**	RegD = 0; Value = 0
C.JAL	**001VVVVVVVVVV01**	RV32 only
C.ADDIW	**001VDDDDDVVVVV01**	Rv64 and RV128 only; RegD ≠ 0
<Hint>	**001V00000VVVVV01**	Rv64 and RV128 only; RegD = 0
C.LI	**010VDDDDDVVVVV01**	RegD ≠ 0
<Hint>	**010VDDDDDVVVVV01**	RegD = 0
C.LUI	**011VDDDDDVVVVV01**	RegD ≠ 0, 2; Value ≠ 0
<Reserved>	**011VDDDDDVVVVV01**	RegD ≠ 0, 2; Value = 0
<Hint>	**011V00000VVVVV01**	RegD = 0
C.ADDI16SP	**011V00010VVVVV01**	RegD = 2; Value ≠ 0
<Reserved>	**011V00010VVVVV01**	RegD = 2, Value = 0
C.SRLI	**100000DDDVVVVV01**	RV32; V ≠ 0
<Hint>	**100000DDD0000001**	RV32; V = 0
<NSE>	**100100DDDVVVVV01**	RV32;

C.SRLI	**100V00DDDVVVVV01**	RV64; V ≠ 0
<Hint>	**100000DDD0000001**	RV64; V = 0
C.SRLI	**100V00DDDVVVVV01**	RV128 only; V ≠ 0
C.SRLI64	**100000DDD0000001**	RV128 only; V = 0
C.SRAI	**100001DDDVVVVV01**	RV32; V ≠ 0
<Hint>	**100001DDD0000001**	RV32; V = 0
<NSE>	**100101DDDVVVVV01**	RV32;
C.SRAI	**100V01DDDVVVVV01**	RV64; V ≠ 0
<Hint>	**100001DDD0000001**	RV64; V = 0
C.SRAI	**100V01DDDVVVVV01**	RV128 only; V ≠ 0
C.SRAI64	**100001DDD0000001**	RV128 only; V = 0
C.SUB	**100011DDD00bbb01**	
C.XOR	**100011DDD01bbb01**	
C.OR	**100011DDD10bbb01**	
C.AND	**100011DDD11bbb01**	
C.SUBW	**100111DDD00bbb01**	RV64 and RV128 only
<Reserved>	**100111DDD00bbb01**	RV32 only
C.ADDW	**100111DDD01bbb01**	RV64 and RV128 only
<Reserved>	**100111DDD01bbb01**	RV32 only
<Reserved>	**100111XXX10XXX01**	
<Reserved>	**100111XXX11XXX01**	
C.ANDI	**100V10DDDVVVVV01**	
C.J	**101VVVVVVVVVV01**	
C.BEQZ	**110VVVaaaVVVVV01**	
C.BNEZ	**111VVVaaaVVVVV01**	
C.SLLI	**0000DDDDDVVVVV10**	RV32 only; RegD ≠ 0; V ≠ 0
<Hint>	**0000DDDDD0000010**	RV32 only; RegD ≠ 0; V = 0
<NSE>	**0001DDDDDVVVVV10**	RV32 only; RegD ≠ 0

<Hint>	**000V00000VVVVV10**	RV32 only; RegD = 0
C.SLLI	**000VDDDDDVVVVV10**	RV64 only; RegD ≠ 0; V ≠ 0
<Hint>	**0000DDDDD0000010**	RV64 only; RegD ≠ 0; V = 0
<Hint>	**000V00000VVVVV10**	RV64 only; RegD = 0
C.SLLI	**000VDDDDDVVVVV10**	RV128 only; RegD ≠ 0; V ≠ 0
C.SLLI64	**0000DDDDD0000010**	RV128 only; RegD ≠ 0; V = 0
<Hint>	**000V00000VVVVV10**	RV128 only; RegD = 0
C.FLDSP	**001VDDDDDVVVVV10**	RV32 and RV64 only
C.LQSP	**001VDDDDDVVVVV10**	RV128 only; RegD ≠ 0
<Reserved>	**001VDDDDDVVVVV10**	RV128 only; RegD = 0
C.LWSP	**010VDDDDDVVVVV10**	RegD ≠ 0
<Reserved>	**010VDDDDDVVVVV10**	RegD = 0
C.LDSP	**011VDDDDDVVVVV10**	RV64 and RV128 only; RegD ≠ 0
<Reserved>	**011VDDDDDVVVVV10**	RV64 and RV128 only; RegD = 0
C.FLWSP	**011VDDDDDVVVVV10**	RV32 only
C.JR	**1000aaaaa0000010**	RegA ≠ 0
<Reserved>	**1000aaaaa0000010**	RegA = 0
C.MV	**1000DDDDDbbbbb10**	RegB ≠ 0; RegD ≠ 0
<Hint>	**1000DDDDDbbbbb10**	RegB ≠ 0; RegD = 0
C.EBREAK	**1001000000000010**	
C.JALR	**1001aaaaa0000010**	RegA ≠ 0
C.ADD	**1001DDDDDbbbbb10**	RegB ≠ 0; RegD ≠ 0
<Reserved>	**1001DDDDDbbbbb10**	RegB ≠ 0; RegD = 0
C.FSDSP	**101VVVVVVbbbbb10**	RV32 and RV64 only
C.SQSP	**101VVVVVVbbbbb10**	RV128 only
C.SWSP	**110VVVVVVbbbbb10**	
C.FSWSP	**111VVVVVVbbbbb10**	RV32 only
C.SDSP	**111VVVVVVbbbbb10**	RV64 and RV128 only

Chapter 7: Concurrency and Atomic Instructions

Hardware Threads (HARTs)

A simple processor has a single flow-of-control. That is, it operates as a single "thread", with a FETCH-DECODE-EXECUTE loop. Instructions are executed one-after-the-other. (In the case of pipelining, several instructions may be in some stage of execution at any moment, but the instructions are completed (or "retired") one-after-the-other, and the pipelined behavior is exactly the same as if each instruction had been executed alone to completion in the order they appear in the code.)

Normally, an operating system will multiplex the processor, thereby providing many threads, via timeslicing. For clarity, we can distinguish between "hardware thread" and "software threads". The OS can multiplex a single hardware thread to provide multiple software threads.

In a multi-core or multiprocessor system, each core is running its own independent FETCH-DECODE-EXECUTE cycle. Thus, there will be a number of hardware threads equal to the number of cores. How the OS implements software threads on top of the available hardware threads is a complex topic which concerns the kernel design and organization.

In a "**superscalar**" core, there is more than one instruction stream. There is a fixed number (such as 2 or 4) of independent instruction streams. For example, some particular superscalar core might execute two independent instruction streams simultaneously. Such a core is providing two hardware threads. Each stream has its own independent set of registers and its own program counter, more-or-less as if they were separate cores in a multiprocessor system.

One benefit of the superscalar organization is that expensive and rarely used circuits (e.g., floating point divide) can be provided only in a single copy, and shared by both hardware threads. Also, there may be multiple instantiations of more critical circuits

(e.g., integer addition) which are allocated dynamically to those instructions that need them.

With superscalar organization, it is possible that more than one instruction can be retired per clock cycle, providing greater performance.

Definition of "HART": In the discussion of concurrency control and synchronization, the RISC-V documentation uses the term "**HART**" to refer to a hardware thread. The hardware threads (HARTs) might be implemented on distinct processors within a multiprocessor system, or by multiple cores in a single processor, or on a single superscalar core. The RISC-V spec covers all possibilities, including hybrid combinations.

The RISC-V Memory Model

It's a commonly accepted principle in processor design that, within a single instruction stream, data dependencies are respected. In other words, the processor executes instructions in order and will not reorder instructions.

For example, an instruction that loads a register with a value from memory and an instruction that stores the contents of that same register in memory should never be reordered. Assembly language programmers are free to assume that each instruction is executed from beginning to completion in the order it was specified. Any optimizations or reordering done by the hardware must preserve this constraint. For example, two instructions that access different registers and different memory addresses can safely be reordered or executed simultaneously to achieve faster execution.

However, between different hardware threads, the order in which operations execute may be indeterminate. For example, consider two independent processors in a multiprocessor system with shared memory. Imagine two instructions which both store into a single memory location; they may execute in arbitrary order, resulting in a different value being stored last.

Here is one common requirement for proper synchronization in a system of cooperating processes. (By "**cooperating processes**", we mean a system with multiple threads that are designed to work together concurrently executing to

complete some given programming task correctly, regardless of the vagaries of indeterminate execution.)

Synchronization of Readers and Writers: With a potentially shared piece of data, a writer must wait to write the data until all previous readers are finished reading the old value. Likewise, a reader must wait until any previous writer is finished writing the new value.

The RISC-V spec assumes that reads and writes issued by a single hardware thread (HART), must be visible to that same HART in the order executed. That is, the core's actual order of execution of a HART's instructions must be indistinguishable from the sequential order in which the programmer wrote the instructions. Perhaps the core will reorder instructions to improve performance, but any reordering must be "safe" in the sense of having the same effect as being issued in the original order.

This assumption is something that every programmer takes for granted: the computer will execute their code instructions in the order the programmer wrote them. For single threads executing in isolation, there is nothing much more to talk about. But things quickly get complicated when there are multiple threads.

As far as the apparent order of the operations as viewed by other HARTs, the RISC-V spec does not assume anything. In fact, the RISC-V spec allows the following:

> *It may appear to a second HART that the instructions executed by the first HART are done out of order.*

So it is possible that two distinct HARTs will see the same set of operations happening in a different order. This is a very real possibility on some systems when the operations in question are reads and writes to memory and the individual HARTs are operating with private caches.

For example, consider the following sequence of instructions issued by HART A to memory location X.

Write X ← 4
Read X
Write X ← 5
Read X

The first read will return 4 and the second read will return 5. However HART B might execute these instructions:

Read X
Read X

The first read might return 5 and the second read might return 4. It appears to HART B that the writes were executed out of order, probably due to inconsistencies between their caches.

Concurrency Control - Review and Background

This section is not specific to RISC-V and can be safely skipped if you are familiar with concurrency control.

In any system with multiple threads, some form of synchronization and concurrency control is required. Without it, the behavior of some code sequences may be nondeterministic and program errors may result. For example, each of several threads may need to periodically query and update a shared piece of data. Without any control, two or more threads may be trying to read and update the shared data simultaneously; there arises a possibility that some threads may see an inconsistent state of the data or that some updates may be lost or that several concurrent updates will result in the shared data being put into a invalid or inconsistent state.

In one common approach to concurrency control — particularly among software threads — a "**lock**" is created to protect the shared data. Any block of code accessing the shared data must first "acquire" (or "obtain" or "set") the lock before accessing the shared data, and must "release" (i.e., "free", "clear") the lock after the access is complete. A sequence of code which accesses the shared data is called a "**critical section**" and the programmer must remember to acquire the lock at the beginning of the critical section and release the lock at the end of the critical section.

Operating systems typically provide "locks", along with "acquire" and "release" operations. Implementing these operations on a single processor system is simple: the OS merely has to avoid a thread-switch occurring during the acquire and release operations themselves. This can be done easily and efficiently by disabling interrupts for the duration of the acquire/release operations.

Things are significantly more complex for multi-core systems with shared memory. Now there is "multiprocessing" (true concurrency) and not just "multithreading" (simulated concurrency using time-slicing). The lock itself becomes shared data and access to the lock in effect becomes a critical section of its own. Blocking interrupts on one processor is no longer adequate. There exist algorithms to solve this problem purely in software (Peterson's algorithm; Dekker's algorithm) but architectural support in the ISA is required for performance reasons.

In the past there have been a couple of instructions proposed to support the "acquire" and "release" operations for locks. One such instruction was the **"test-and-set" instruction**. This instruction includes two operands: an address and a register. The instruction reads from memory at the address given and stores the result into the register. Then the instructions stores a fixed known value (usually "1") into the same memory location. Both the read-memory and the write-memory operation are specified to be "atomic", which means that there is a guarantee that no other instruction will occur between the read-memory and write-memory from any other source (e.g., other cores or I/O devices).

Historically, hardware can enforce the atomicity guarantee by somehow blocking all accesses to all memory locations for all other cores between the read and write phases of the test-and-set instruction. This will slow other parts of the system down a little, but if the number of cores is small, it's not much of a burden.

The test-and-set instruction can be used to implement a lock very simply. Here's how: the lock is represented as a single memory location, with "0" meaning "not locked" and "1" meaning "locked". The "acquire" operation consists of executing the test-and-set instruction. After the instruction, the lock will be in the "set" state and will contain "1". The "acquire" operation then examines the previous lock value, which was retrieved from memory and stored in a register. If it was "0", then all is good: the lock was previously free. It has now been acquired and the critical section can be entered. However, if the previous value was "1", then the lock was apparently already set. The lock has not been acquired. The acquire code must try again, either immediately, or after a wait, or after thread rescheduling. The "release" operation is implemented by simply storing a "0" into the memory location representing the lock.

Another approach to ISA design is to provide a **"swap" instruction**, instead of the test-and-set instruction. Like the test-and-set instruction, the swap instruction has a read-memory phase and a write-memory phase. The instruction is takes two operands: a register and a memory address. The instruction swaps the values; after the instruction the register contains the previous value from the memory word and

the memory word contains the previous value in the register. The swap instruction is a generalization of the test-and-set instruction: by placing a "1" in the register before the swap instruction is executed, the effect will be identical to a test-and-set instruction.

Another approach is the **"compare-and-swap" (CAS) instruction**, which is used in the X-86 architecture. The CAS instruction takes four operands: a memory address, a register containing an "expected value", and a register containing a "new value", and a register in which to store a return code. The instruction does several things atomically, i.e., without the possibility of interruption or interference from other threads. First, the instruction reads a value from memory and compares it to the "expected value." If the two are different, the instruction fails and returns a code to indicate failure. But if the two values are equal, the instruction stores the "new value" into the memory location and returns a code indicating success.

The problem with all these instructions is that each requires two memory operations. This violates the general RISC principle of keeping all instructions simple, minimal, and fast. Two memory operations in a single instructions is too much and complicates the circuitry.

Furthermore, these instructions can slow other cores down since they must be executed atomically and presumably they work by freezing up the entire memory system during their execution.

The problems with atomically executing both a read and write together get worse with more cores. These problems also get worse with more frequent locking operations. To support increased concurrency, it's a good idea to have finer-grained locking, i.e., to have more locks, having each lock protect less data. Finer-grained locking implies more "acquire" and "release" operations, even if the locks are usually found to be free. The problems also get worse with increased sharing of data, a generalized trend we are seeing in many contexts.

A Review of Caching Concepts

This section is not specific to RISC-V and can be safely skipped if you are familiar with caching.

The problem of synchronization between multiple processors (e.g., cores or HARTs) gets much more complex in the presence of cache memories. In short, the problem stems from the fact that a single location in memory can have multiple values at once, because the data can be present in several caches with inconsistent values. Let's review the basic ideas behind cache memories before we discuss the RISC-V instructions.

In many computers, the bandwidth between main memory and the processor core is a performance bottleneck. To increase performance, a "**cache**" memory is placed between main memory and the core.

Whenever the core tries to fetch data from memory, the cache is checked and, if it contains the data, the data can be sent to the core immediately. Otherwise, the data must be fetched from main memory, which takes more time. The data will be sent to the core as well as being stored in cache memory to speed future requests for the same data.

When data is written from the core to memory, the data will sometimes be written to the cache memory, which speeds future accesses if the core wants to read the data again soon ("**write allocate**"). Some systems avoid caching the updated data ("**no-write-allocate**"). In either case, the updated data must be copied to main memory at some point. The write to main memory may occur immediately, at the time the core first writes the data to cache, and this is called "**write-through**". Alternatively, write to main memory can be delayed until later. At some later time, when space in the cache is needed for some other data, the updated modified data value will finally be written to main memory. This policy is called "**write-back**".

Caches do not operate on bytes individually; instead they store data in units of "**cache block**". A cache line is a block of contiguous data bytes (typically 64 bytes) which are copied together as a unit in and out of the cache. In a request for a single byte or halfword (for example), the entire cache block containing the desired bytes will either be in the cache (a "**cache hit**") or not (a "**cache miss**").

The typical alignment requirements ensure that a multiple data unit (such as a halfword) will never cross a cache block boundary; the data will either be entirely in one block or entirely in another block. Misaligned data will occasionally cross cache block boundaries and the process of reading into or writing from the core to the memory system will be complicated and generally require about twice as much time.

Caches are often organized into levels. For example, a three-level cache might consist of:

L1 Closest to the core. Fastest, smallest in capacity.
L2 Intermediate
L3 Closest to main memory. Slowest, largest in capacity.

In a system with multiple cores on a single chip, each individual core might possess its own private L1 and L2 caches, while all cores will share a larger L3 cache.

There are differences between instruction and regular data. In particular, instructions are always read-only. Since the data is never modified by the core, there is no need to provide hardware to implement the write-allocate/write-no-allocate or write-back/write-through policies.

As a typical example, each core will have two L1 caches; one for data and one for instructions. The term "**unified cache**" means that the cache holds both instructions and data. Typically the slower, larger cache (e.g., L3) is unified.

In many ISAs, including RISC-V, all access to I/O devices is "**memory-mapped I/O**" which means that the I/O devices are assigned addresses in the physical memory map. To send data or commands to an I/O device, the core will issue "store" instructions and to retrieve data or status information from an I/O device, the core will execute "load" instructions.

Many I/O devices also access main memory directly. For example, the core might move data into memory and then issue commands to an I/O device which subsequently cause the device to retrieve the same data directly from memory.

Thus, the main memory system may have several "ports" allowing data to be read and written separately by:

- Different cores on a single chip
- Different processor chips
- Various I/O devices

A modern computer system will consist of a number of busses and a number of different devices sitting on each bus.

Which all this complexity, there is a problem of **synchronization and data consistency**. For example, a core may write a byte of data to memory and expect some I/O device or other core to fetch that byte. In the presence of caches, there may be several different values for the same memory address at the same instant. Without further control, a "reader" may not see the value written by a "writer" even though the write occurs earlier in time.

Note that the data consistency problem does not occur for read-only data. If a particular byte of data never changes, then there is not a problem. Perhaps the data is cached or perhaps it must be fetched from main memory. But since there is only one value, there is never a possibility that the cache contains an outdated value. A synchronization problem can only occur when data is updated.

Note that all data is written at some point, except perhaps data stored in Read-Only Memory (ROM). We must assume that when any computer is powered up, the contents of dynamic memory are undefined; even read-only data must involve an initial write to the memory so care must be taken to avoid reading the initial undefined data.

The synchronization problem occurs because multiple values for a given memory location can be stored in various caches. If the caches are eliminated, then the data must necessarily be consistent; but caches yield tremendous performance benefits, so support for synchronization influences ISA design.

We can make a distinction between **private caches** and **shared (i.e., memory-side or global) caches**. Consider a system with several cores and a single shared main memory. Each core may have its own private memory cache, in which case a single byte of memory may be cached simultaneously in several different private caches. An update by any single core to this byte ought to be visible to all cores but (without synchronization) other cores might read outdated values from their private caches.

If, instead there is only a single shared cache sitting between main memory and all cores, then the synchronization problem goes away. A given byte can only be cached in one place, namely the shared cache. Since all cores are using this shared cache, any update to the byte will be reflected in the shared cache, and therefore seen by all other cores.

Modern systems often use a mixed of private and shared caches.

A cache memory consists of a set of "**cache lines**".

Each line in a cache contains a couple bits identifying what data is contained in the line and whether the data is "valid" or "invalid". More precisely, a cache line generally contains the following pieces of information:

- The cache line data (e.g., a block of 64 data bytes)
- The address of the data block
- A "**valid bit**", to tell whether this cache line is meaningful
- A "**dirty bit**", to tell whether the data has been modified

We discuss virtual memory and address translation elsewhere. But here we point out that each cache line must contain the address of the data stored in that cache line.

In the easiest-to-understand approach, each cache line contains the physical address of its data block. The cache is an "**associative memory**" using the physical address as the search key (i.e., the associative index). However, because of virtual memory address translation, the virtual address is known earlier in time than the physical address, so it makes sense to index the cache lines by virtual address. Real designs are more complex, but for our purposes we do not need to go further here.

A crude solution to the cache consistency problem is to periodically empty the entire cache.

"**Flushing the cache**" means writing all updated data back to main memory and marking all cache lines "invalid". So a simple approach is to flush all caches after any write. Flushing the entire cache after any write is overly conservative and not the most efficient approach, but it will solve the cache consistency problem.

The software might be a little smarter and avoid full cache flushes when not strictly required. For example, when one core writes to main memory, there is a question of whether caches belonging to other cores must be flushed. If the programmer can be certain that the data will never be read by other cores, then calling for a cache flush can be avoided.

The RISC-V designers recognize that a finer-grained control over caches and memory synchronization is required, and provide the FENCE instruction, which will be described later.

RISC-V Concurrency Control

RISC-V supports concurrency control, synchronization, and cache consistency with these instructions:

Always available:

FENCE	Synchronize data reads and writes
FENCE.I	Synchronize the instruction cache

Available in the "A" extension (for "Atomic Instructions")

LR	Load Reserved
SC	Store Conditional
AMOSWAP	Atomically swap two values
AMOADD	Atomically add two values
AMOAND	Atomically AND two values
AMOOR	Atomically OR two values
AMOXOR	Atomically XOR two values
AMOMAX	Atomically MAX two values
AMOMIN	Atomically MIN two values

Under Revision: The RISC-V spec states that their memory model is under revision, implying that these instructions may change.

The FENCE Instructions

There are two fence instructions: FENCE and FENCE.I and these instructions are always available. They do not require the "A" (Atomic Operations) extension.

The general idea with a "fence" is that all operations are partitioned into two sets: the predecessor set and the successor set.

The fence requires all operations in the predecessor set to complete before any of the operations in the successor set can proceed. In the case of RISC-V, the predecessor operations are the instructions that must complete before the fence, and the successor operations are those that may not begin until after the fence.

As an analogy, consider the Tour de France, where all the cyclists must complete each stage of the race, before any rider can begin the next stage. Only after the slowest rider finishes the previous stage of the race are any riders allowed to start the next stage. The stages are effectively separate by "fences".

The FENCE instruction in RISC-V may only affect instructions that read or write to the memory system. It is used to control concurrent accesses to shared memory.

Recall that the physical memory address space can be populated with:

- Normal, physical memory (containing stored data bytes)
- Memory-mapped I/O devices

A hardware thread (HART) can either read or write values to/from the memory space. This yields these combinations:

R Read – read data from physical memory
W Output – write data to physical memory
I Input – read data from a memory-mapped I/O device
O Output – write data to a memory-mapped I/O device

Of course many operations (such as a movement of data from one register to another) do not touch memory at all. Such operations do not fall into any of these four classifications and are ignored by the FENCE instruction.

Imagine that some HART executes two memory write ("W") operations in succession. Perhaps the first write operation is setting a lock, which is intended to protect some piece of share data. Only after the lock has been set, is it permissible to update the shared data. (This is the idea behind protecting shared data with locks; the shared data should only be accessed when the lock is held by the thread.)

Of course the HART in question will execute the write to the lock before the write to the data, but because of the relaxed memory model assumed by the RISC-V spec, it is possible that other HARTs will see the data write appearing to come before the lock write. Other HARTs could therefore see the updated data before seeing the lock getting set, thereby violating the functionality and integrity of the lock protocol. The FENCE instruction provides a way to prevent this disaster and allows programmers to implement proper concurrency control and create correct, reliable cooperating processes.

The FENCE instruction can be used to make sure that all other HARTs will see the write occurring before the read. In this example, we will use a FENCE instruction to make sure that all writes to memory that were executed before the FENCE instruction will appear to execute before all write to memory that will be executed after the FENCE instruction.

For the purposes of the FENCE instruction, we use "P" for operations occurring prior to the FENCE (predecessor operations) and "S" for operations occurring after the FENCE instruction (successor operations). Thus, we have:

Predecessor operations:

PR	Data reads occurring before the FENCE
PW	Data writes occurring before the FENCE
PI	Memory-mapped reads (inputs) occurring before the FENCE
PO	Memory-mapped writes (output) occurring before the FENCE

Successor operations:

SR	Data reads occurring after the FENCE
SW	Data writes occurring after the FENCE
SI	Memory-mapped reads (inputs) occurring after the FENCE
SO	Memory-mapped writes (output) occurring after the FENCE

In our example, we want to force all writes to data memory that were issued before the FENCE to complete before all write to data memory that will be issued after the FENCE instruction. Thus, we can insert the following FENCE instruction between the write to the lock and the read from the shared data:

```
...write to lock...
FENCE   PW,SW     # Complete past writes before future writes
...write to shared data...
```

FENCE

General Form:

```
FENCE      PI,PO,PR,PW,SI,SO,SR,SW
```

Example:

This instruction contains 8 single bit fields. Each can be enabled (set to 1) or disabled (cleared to 0). The fields are called PI, PO, PR, PW, SI, SO, SR, SW. The spec does not indicate how this instruction is to be coded in assembly. Perhaps the bits can be set by being mentioned and cleared otherwise. For example:

```
        FENCE     PR,PW,SR,SW
```

Description:

See text above.

Availability:

This instruction does not require the "A" extension (Atomic Instructions). It is always available.

Encoding:

This is a variation of an I-type instruction, where the RegD and Reg2 fields are not used and set to zero and only 8 of the Immed-12 bits are used.

Implementation of FENCE: One simple implementation of FENCE is for the hardware to simply ignore the distinction between R, W, I, and O and simply complete all operations occurring before the FENCE before starting any operations after the FENCE. In the most conservative but simplest implementation, the hardware might just flush all caches, forcing a sort of hard, system-wide fence operation. Hopefully some implementations will provide the finer-grained control permitted by the instruction.

The FLUSH instruction is the main tool for flushing the cache, but is intended to operate on only data caches. However, instructions may be cached in a separate instruction cache and there is a need to update this cache from time to time. This is the purpose of the FENCE.I instruction.

Whenever bytes are written to memory and these bytes are intended to be subsequently executed as instructions, an update to the instruction cache may be required. For example, a write (e.g., the ST store instruction) to memory location X may be executed. Subsequently the core will execute the instruction located at address X; of course the new value should be fetched. However, if the instruction cache already contains the previous contents for location X, there may be a problem. The core cannot simply use the cached value.

The effect of the FENCE.I instruction is equivalent to invalidating the entire instruction cache, thereby forcing all future fetches to go to memory. This guarantees that the newest value for location X will be fetched.

FENCE.I – (Instruction Cache Flushing)

General Form:

```
        FENCE.I
```

Example:

```
FENCE.I     # Flush the instruction cache
```

Description:

Invalidate the entire instruction cache local to the current HART, or at least do something operationally equivalent. See text.

Availability:

This instruction does not require the "A" extension (Atomic Instructions). It is always available.

Encoding:

This is a variation of an I-type instruction, where the RegD, Reg2, and Immed-12 fields are not used and set to zero.

Note: This instruction only affects the current HART. In a multi-core system, it is probable that instructions will be cached in instruction caches that are private to each core. If core A writes data to memory location X and expects to execute that same data (e.g., jump to X), then it should execute a FENCE.I instruction to make sure that its cache doesn't contain an obsolete value for location X.

However, this will not help other cores, which may have cached older data from location X in their private instruction caches. In order to flush the instruction caches of other cores, software on core A will have to somehow signal software running on the other cores, which will then each need to execute FENCE.I instructions, to flush their own private caches.

Implementation Notes: One implementation of FENCE.I is simply to flush the entire instruction cache, i.e., to set all cache lines to "invalid".

This may seem like a coarse-grained approach, but it is assumed that software typically updates a large block of instructions in memory and, only after writing all the data, will there be a jump to the newly created code. For example, consider a Just-In-Time (JIT) compiler, which only compiles a function/method at the time it is first called/invoked. There is a clear transition from compile-phase to execution-phase. Thus, only a single FENCE.I instruction would be needed, and the previous contents of the instruction cache are likely to be unneeded in the near future anyway, so invalidating them is acceptable.

Another approach assumes that the data and instruction caches are kept consistent. For example, the instruction cache might be "snooping" writes to the

data cache. Every write to the data cache would have the side-effect of updating the instruction cache when necessary. Whenever the instruction cache happened to contain data from the same address being written to, that cache line in the instruction cache would be updated or at least invalidated. If the caches are kept coherent, then the FENCE.I instruction can be implemented by simply flushing the pipeline, which may potentially contain outdated instruction data.

Load-Reserved / Store-Conditional Semantics

The RISC-V design supports an approach to concurrency control called load-reserved/store-conditional (LR/SC). This is also called "load-link/store-conditional (LL/SC).

We'll begin by describing the LR and SC instructions, which are only available in the "A" (Atomic Operations) extension to the RISC-V spec.

The idea is that an atomic operation such as test-and-set, swap, or compare-and-set is broken into two separate instructions. The first instruction ("load-reserved") performs the read phase and the second instruction ("store-conditional") performs the write phase.

The LR instruction is very similar to a typical LOAD instruction. It reads a value from memory and stores it into a register. But in addition, the instruction also "reserves" that memory location. You can imagine that this reservation consists of making a note of which core (or "HART" in RISC-V terminology) touched which memory location with an LR instruction. This reservation note is something that is managed by the memory synchronization/control hardware and not by the cores attempting to access memory.

At some later time, an SC instruction will be executed. It will store a value from a register into memory, just like a typical STORE instruction. However, the instruction may "succeed" or "fail". The SC instruction has three operands: a register containing the value to be stored in memory, a memory address, and a register into which the return code will be stored.

There are two return codes: "0" indicates success and "non-zero" indicates failure. Failure might be caused by misaligned addresses, access violations, etc., but these

are beside the point. The primary use of the success/failure has to do with concurrency control.

The proper way to use these instruction is to execute an LR instruction on some memory location and then, shortly thereafter, to execute an SC instruction on that same memory location. If no other core/HART has stored into that memory location since the LR instruction, then the SC will succeed and the value will be stored into memory. But if some other core/HART has stored anything into that memory location, then the SC will fail and nothing will be stored into memory. Presumably, the code will then loop and retry the LR/SC sequence until it succeeds.

A reservation is made by a LR instruction and remains valid for a small, finite amount of time (about 16 instructions) specified by the RISC-V spec. The SC instruction must be executed within this window of opportunity. If some other core/HART sneaks in an executes an LR on the same memory location, then the reservation is cancelled and the SC will fail.

Load Reserved (Word)

General Form:

```
LR.W     RegD,(Reg1)
```

Example:

```
LR.W     x4,(x9)    # x4 = Mem[x9] and reserve
```

Description:

A 32-bit value is fetched from memory and moved into register RegD. The memory address is in Reg1.

Comment:

This instruction places a “reservation” on a block of memory containing this word. See comments in the text regarding semantics.

The address be properly aligned. The reservation may include more than just the bytes addressed, but will at least include these bytes.

RV64 / RV128:

For a machine with a register width larger than 32-bits, the value is sign-extended to the full length of the register.

Availability:

This instruction is only available in the “A” extension (Atomic Instructions).

Encoding:

This is a variation of an R-type instruction, where Reg2 is 00000.

Load Reserved (Doubleword)

General Form:

```
LR.D      RegD,(Reg1)
```

Example:

```
LR.D      x4,(x9)     # x4 = Mem[x9] and reserve
```

Description:

A 64-bit value is fetched from memory and moved into register RegD. The memory address is in Reg1.

Comment:

See LR.W.

RV64 / RV128:

Only available on RV64 and RV128. For RV128, the value is sign-extended to the full length of the register.

Availability:

This instruction is only available in the "A" extension (Atomic Instructions).

Encoding:

This is a variation of an R-type instruction, where Reg2 is 00000.

Store Conditional (Word)

General Form:

```
SC.W      RegD,Reg2,(Reg1)
```

Example:

```
SC.W      x5,x4,(x9)     # Mem[x9]=x4; x5=success code
```

Description:

A 32-bit value is copied from register Reg2 to memory. The memory address is in Reg1. A success/fail code is placed in RegD where 0=success and non-zero=failure. On failure, memory is not changed.

Comment:

This instruction assumes a "reservation" has previously been made by an LR instruction on a block of memory containing this word. See comments in the text for details.

The address be properly aligned. The reservation may include more than just the bytes addressed, but will at least include these bytes.

The value of "1" will usually be used to indicate failure, but there is a possibility for different codes to be used to indicate different reasons for failure.

RV64 / RV128:

For a machine with a register width larger than 32-bits, the upper bits of the register are ignored.

Availability:

This instruction is only available in the "A" extension (Atomic Instructions).

Encoding:

This is an R-type instruction.

Store Conditional (Doubleword)

General Form:

```
SC.D      RegD,Reg2,(Reg1)
```

Example:

```
SC.D      x5,x4,(x9)     # Mem[x9]=x4; x5=success code
```

Description:

A 64-bit value is copied from register Reg2 to memory. The memory address is in Reg1. A success/fail code is placed in RegD where 0=success and non-zero=failure. On failure, memory is not changed.

Comment:

See SC.W

RV64 / RV128:

Only available on RV64 and RV128. For RV128, the upper bits of the register are ignored.

Availability:

This instruction is only available in the "A" extension (Atomic Instructions).

Encoding:

This is an R-type instruction.

Atomic Memory Control Bits: "aq" and "rl"

The "A" extension provides the following atomic memory operation (AMO) instructions:

LR	Load Reserved
SC	Store Conditional
AMOSWAP	Atomically swap two values
AMOADD	Atomically add two values
AMOAND	Atomically AND two values
AMOOR	Atomically OR two values
AMOXOR	Atomically XOR two values
AMOMAX	Atomically MAX two values
AMOMIN	Atomically MIN two values

Each instruction contains two additional bits to control their operation a bit further than suggested previously. These bits are called “aq” and “rl” (for acquire and release).

The bits can be set using the following assembler notation:

```
lr.w.aq     x5,(x6)   # set the “aq” bit to 1
lr.w.rl     x5,(x6)   # set the “rl” bit to 1
lr.w.aq.rl  x5,(x6)   # set both bits
```

Sometimes instructions on a single thread/HART will be reordered by the core execution unit to achieve greater performance. For example, imagine two instructions appearing in sequence: first a store instruction followed by a load instruction. If the addresses in these instructions are different, then the execution unit can reasonably perform both operations in any order. Perhaps, the core will issue both instructions simultaneously; the memory unit may complete the operation of the instructions in any order, perhaps due to the chance contents of the cache. Since the two instructions are touching different memory locations, they can be safely reordered, right?

Usually the reordering is safe, but in the presence of other concurrent threads/ HARTs, there can be issues. For example, imagine that one of the two memory locations represents a lock that is intended to control access to the other location. Now, it becomes critical that the operations are executed in the correct order.

(As an example of the problem, imagine that the store operation is being used to set a lock. Only after the lock is set, is it allowable to load values from the shared data. The shared data is only guaranteed to be in a consistent, usable state by other processes when the lock is free; whenever some process holds the lock, the state of the data may be in an inconsistent, unfinished state, and must not be accessed by

other processes. But now imagine that the load operation is performed first, before the lock is set with the store instruction. This means that the HART has accessed shared memory without first acquiring the lock. This might allow it to see a version or state of the data that it should not see; a violation of the convention to acquire locks before accessing the shared data.)

The "aq" and "rl" bits control the ordering of instructions on a single HART. They do not have any effect on the scheduling of instructions executed on different HARTs. In other words, the bits are there just to make sure that reordering of instructions on the current HART (which the execution unit might normally do to increase performance) are to be suppressed so that concurrency control code will function properly.

The "aq" bit has the following meaning. When an instruction with the "aq" (acquire) bit set is executed, it means that any instructions that follow the "aq" instruction must be delayed and cannot be executed before the "aq" instruction. The "aq" instruction will be executed first, before the instructions that follow it in the sequential flow of instructions on this HART.

The "rl" bit has the following meaning. When an instruction with the "rl" (release) bit set is executed, it means that any instruction that precedes the "rl" instruction must be done and executed to completion before the "rl" instruction. The "rl" instruction will be executed last, after the instructions that precede it in the sequential flow of instructions on this HART.

While we have just described the bits as applying to only one HART, this was a simplification. There are really multiple views of the order in which instructions are executed. First, as mentioned, there is the order that the HART itself executes the two instructions. But second, there is the order in which other HARTs observe the execution. Perhaps due to caching effects, the second order may be different.

For example, imagine that there are two store instructions to be executed by one HART. The first instruction is used to set a lock and the second instruction is an update to the shared data. Presumably, any update should only be done privately by a HART that is already holding the lock. The concern is the order in which the other HARTs observe these two memory store operations: other HARTs must observe the store to the lock to happen first, thus signaling that the lock is no longer free. Other HARTs must not be allowed to see a view in which the second store appeared to occur before the lock was set, otherwise this opens the door to allowing some HART to access data that should not be accessible.

The “aq” and “rl” bits impose ordering constraints on the views observed by all HARTs of the instruction stream of the HART using them. Other HARTs will see only the legal instruction sequences.

When both “aq” and “rl” bits are set in some instruction, something called “sequential consistency” is imposed. All other HARTs will see the same ordering as occurred in the HART containing the instruction: All instructions that were executed before the instruction in question will appear to all other HARTs as happening before the instruction. All instructions that were executed after the instruction will appear to all other HARTs as happening after the instruction.

RISC-V divides accesses to the memory system into two categories:

- Accesses to physical memory
- Accesses to memory-mapped I/O

Each AMO instruction (i.e., LR, SC, AMOSWAP, AMOADD, AMOAND, AMOOR, AMOXOR, AMOMAX, AMOMIN) will access exactly one of these two domains: either memory or I/O.

The “aq” and “rl” bits apply only to the domain that is being accessed. The bits impose ordering constrains on all accesses to the domain in question, but impose no constraints on the other domain.

Commentary: I’m guessing the motivation is that each domain (memory or I/O) will have a separate approach to implementing the synchronization constraints imposed by the “aq” and “rl” bits. The use of the bits will trigger either one mechanism or the other.

Or perhaps the implementations in each domain will impose widely different performance hits. Certainly the “aq” and “rl” bits will be required to work properly in the memory domain and presumably the implementation will endeavor to provide high performance, since locks in memory are common. But maybe their use within the I/O domain will be rare and a super-slow implementation is acceptable, as long as it can be isolated from the memory system.

Or perhaps it is envisioned that, in some implementations of the standard, the memory system will support the “aq” and “rl” semantics, but will not support the specified behavior for accesses to the I/O system.

???

Using LR and SC Instructions

Take a look at the following code sequence. We assume that there is a lock, whose address is in register x10. When the lock holds "0" the lock is free; the value of "1" is used to indicate the lock is set.

In line 2 we load the current value of the lock and place a reservation on that location. In line 3, we check to see if the lock was already set. If so, we jump to some other code. This code might simply jump back to "lock" to try again immediately; this is called a "spin" lock because the code loops, waiting for the value to become zero. Or perhaps the other code will wait, reschedule threads or whatever, and come back to try again sometime later.

Line 4 loads the value "1" into a temporary register. Line 5 is the STORE-CONDITIONAL instruction, which stores the "1" (indicating "locked") into the lock. This instruction will succeed or fail and the code is placed into x5 to tell which happened.

If some other HART managed to store into the lock, then the SC will fail and will not be updated by this HART. Otherwise, if all is good and the lock reservation is still intact, then the SC will succeed and "1" (representing "locked") will be stored into the lock.

The last line looks at the return code from the SC instruction. If the SC failed, this code loops back to the beginning to try again.

```
1  lock:
2       lr.w       x5,(x10)    # Grab the lock's value
3       bnz        x5,fail     # If already locked, fail
4       li         x7,1        # Store "1" into lock
5       sc.w       x5,x7,(x10) #   to indicate "locked"
6       bnz        x5,lock     # If failure, try again
```

To release a lock, all we need to do is store a "0" into the lock value. See the following code. Recall that x0 always holds "0".

```
7  unlock:
8       sw          x0,(x10)    # Set to "unlocked"
```

The instruction sequences above are okay in one respect, but flawed in another. First, they will atomically acquired the lock in such a way that only one HART at a time will acquire the lock. However, it is assumed that locks are used to protect critical sections of code. The critical section code will access some shared data. All accesses to the shared data must occur after a lock operation and before an unlock operation, in order for the lock to be meaningful.

But with multiple HARTs, it is possible that (due to reordering of memory operations and different views of the order) other HARTs might experience these accesses in the critical section code happening either before the lock operation or after the unlock operation. This could sabotage the lock's functionality.

So we can rewrite this code, making use of the "aq" and "rl" bits. The corrected code is shown below.

The "rl" bit is set within the LR instruction, which means that any memory operations that came earlier in the sequence will be completed (in the sense of being visible to other HARTs) before the lock sequence is begun. We don't want anything we've done that should be visible outside the critical section to be accidentally hidden.

The "aq" bit is set within the SC instruction, which means that anything that happens after the lock sequence (i.e., anything in the critical section code) will be seen by other HARTs as occurring after the lock sequence. This makes sure that no critical section stuff can leak out in front of the lock sequence.

```
1  lock:
2       lr.w.rl     x5,(x10)    # Grab the lock's value
3       bnz         x5,fail     # If already locked, fail
4       li          x7,1        # Store "1" into lock
5       sc.w.aq     x5,x7,(x10) #   to indicate "locked"
6       bnz         x5,lock     # If failure, try again
```

Likewise, we do the same within the unlock sequence, shown next.

```
7  unlock:
8       lr.w.rl     x5,(x10)    # Reserve the lock
9       sc.w.aq     x5,x0,(x10) # Set to "unlocked"
10      bnz         x5,unlock   # If failure, try again
```

We switch from using a STORE WORD instruction to using a LR/SC combination, so that we can use the "rl" and "aq" bits. We use an SC instruction to store the unlock code ("0") into the lock and we use an LR instruction, because the SC instruction will fail if we do not hold a reservation on the memory location. The "rl" bit forces everything we have done before the unlock sequence to appear to other HARTs as occurring before the lock is set to "0" (unlocked). The "aq" bit on the SC instruction forces anything we do next, to appear as occurring after the unlock sequence.

We don't bother to check the old value of the lock, since we assume that (in the absence of bugs), we would never try to unlock a lock that was not previously locked. However, we must test for the failure of the SC; it is always possible that some other HART tried to acquire the lock, touched the memory location representing the lock, and invalidated our reservation.

Deadlock and Starvation – Background

Deadlock occurs whenever two (or more) processes are each holding a resource (e.g., they have each acquired a lock) and each is waiting for a resource held by another process (e.g., trying to acquire another lock). Once a deadlock situation occurs, the processes will be frozen, pending some outside intervention. Deadlock is a concern addressed by the OS and there are a number of approaches to avoiding or dealing with it.

There is another similar problem called "starvation". (The term "livelock is sometimes used to mean starvation.) With starvation, the processes are not necessarily frozen but, due to the vagaries of chance, some processes never make forward progress.

Deadlock requires at least two resources (such as locks) that are being fought over. On the other hand, "starvation" can occur with only one resource.

To understand starvation, imagine a scenario in which two HARTs/processes (called A and B) compete for a single lock. The code running on HART A finds that the lock is currently held by B and responds by waiting and then retrying to acquire the lock. We assume that no thread/process holds a lock indefinitely (i.e., we assume these are cooperating processes without any program bugs), so B must eventually release the lock. But what if, due to the poor luck of A, HART B happens to re-acquire the lock again before A has a chance to check it. When A checks a second time, A once again finds that the lock is still unavailable. Imagine that this repeats indefinitely: B is repeatedly releasing the lock, but A is unable to make any forward process, because by the time A checks, B has already re-acquired the lock. B is well-behaved, never holding the lock indefinitely, yet A ends up being frozen and unable to make progress. This is starvation.

It seems like starvation ought to eventually resolve itself when the bad luck of A finally ends and A gets a chance to acquire the lock, but this line of reasoning is dangerous. Funny things can happen when processes/threads/HARTs are not behaving purely stochastically. The possibility of starvation must be addressed.

Starvation and LR/SC Sequences

Consider the code sequence to acquire a lock shown earlier. That sequence issues a LR instruction and then, a couple of instructions later, it issues an SC instruction. The guarantee made by the RISC-V spec is that, if the SC is successful then no other HARTs have written to the location (even if the value written happens to be the same value). If another HART managed to execute its SC first, then the SC on this HART will fail. The SC may fail for other reasons as well. But if the SC succeeds, we can be sure that the location in question has not been modified since the LR retrieved the a value.

However, the RISC-V spec makes an additional, much stronger guarantee, and it is this: If you keep retrying an LR/SC sequence, it will eventually succeed. Starvation (i.e., livelock) is guaranteed not to happen. The SC may fail a few times, but it will eventually succeed. The idea is that a “denial of service attack” by a heavy flow of requests from other HARTs will never prevent any HART from making progress.

There are some constraints placed on the LR/SC sequence in order for this guarantee to be made, and the “lock” code sequence shown above meets them. There can be code between the LR and SC (including a repeat loop to keep testing until it

succeeds). The basic restriction is that the sequence of instructions cannot be too long, namely 16 instructions. Also other memory operations are forbidden. Also exception/trap processing is forbidden. Also instructions that might take a long time to complete (such as integer multiply or floating point operations) are forbidden.

> **Commentary:** Guaranteeing that starvation will not occur is not necessarily straightforward. One approach involves maintaining a queue that somehow reflects how long a process has been waiting, and giving priority to the process that has been waiting longest. Another approach involves passing a "baton" around in sequence from process to process. Each process gets the baton in turn and, with it, a chance to move forward. But after a limited amount of time, it must pass the baton on to the next process.

The Atomic Memory Operation (AMO) Instructions

The AMOSWAP (Atomic Swap) instruction can be used to simultaneously read a value from memory and store a new value into that same location. The instruction can do this atomically, which means that no intervening instruction that tries to store into this location can sneak in and execute.

Typically, an atomic swap instruction is used to set a lock. The instruction sets the lock to the "locked" value while at the same time reading the old value to make sure the lock was previously "unlocked." The instruction must be atomic to ensure that two concurrent processes don't simultaneously store the "locked" value into an "unlocked" lock without realizing it.

Atomic Swap

General Form:

```
    AMOSWAP.W.aq.rl    RegD,Reg2,(Reg1)    # 32-bits
    AMOSWAP.D.aq.rl    RegD,Reg2,(Reg1)    # 64-bits
```

Examples:

```
    AMOSWAP.W      x5,x4,(x9)    # x5=Mem[x9]; Mem[x9]=x4
    AMOSWAP.W.aq   x5,x4,(x9)    # x5=Mem[x9]; Mem[x9]=x4
    AMOSWAP.D.rl   x5,x4,(x9)    # x5=Mem[x9]; Mem[x9]=x4
```

Description:

A value is read from the memory location whose address is in Reg1 and the value is placed into RegD. Then the value from Reg2 is written to the memory location. In the case that Reg2 and RegD are the same register, the values are swapped, i.e., the value stored in memory is the previous value of Reg2.

This instruction contains an "aq" bit and an "rl" bit, which control the ordering of instructions appearing before and after this instruction.

The memory address must be properly aligned or else an "AMO Address misaligned" exception will be raised.

RV64 / RV128:

AMOSWAP.D is only available on RV64 and RV128.

Availability:

This instruction is only available in the "A" extension (Atomic Instructions).

Encoding:

This is an R-type instruction.

Code Example: The AMOSWAP instruction can be used to implement the "lock" and "unlock" operations on a lock, as we show here. The lock will be represented as

0 = unlocked, i.e., free
1 = locked

We assume register x10 contains the address of the lock and x5 is used as a temporary. Here is code to set the lock:

```
   li             x5,1          # x5 = 1
retry:
   amoswap.w.aq   x5,x5,(x10)   # x5 = oldlock & lock = 1
   bnez           x5,retry      # Retry if previously set
```

After the critical section, which presumably accesses and modifies some shared data, we have the following code to release (i.e., free) the lock:

```
   amoswap.w.rl   x0,x0,(x10)   # Release lock by storing 0
```

The initial AMOSWAP has the "aq" bit set, which means that any instructions that follow the AMOSWAP can not be observed by other HARTs to execute before the AMOSWAP. This prevents code from the critical section from "leaking" out before the lock is set.

The final AMOSWAP has the "rl" bit set, which means that any instructions that precede the AMOSWAP can not be observed by other HARTs to execute after the AMOSWAP. This prevents code from the critical section from "leaking" out after the lock is released.

Atomic Add / AND / OR / XOR / Max / Min (Word)

General Form:

```
AMOADD.W.aq.rl    RegD,Reg2,(Reg1)   # addition
AMOAND.W.aq.rl    RegD,Reg2,(Reg1)   # logical AND
AMOOR.W.aq.rl     RegD,Reg2,(Reg1)   # logical OR
AMOXOR.W.aq.rl    RegD,Reg2,(Reg1)   # logical XOR
AMOMAX.W.aq.rl    RegD,Reg2,(Reg1)   # signed maximum
AMOMAXU.W.aq.rl   RegD,Reg2,(Reg1)   # unsigned maximum
AMOMIN.W.aq.rl    RegD,Reg2,(Reg1)   # signed minimum
AMOMINU.W.aq.rl   RegD,Reg2,(Reg1)   # unsigned minimum
```

Examples:

```
AMOADD.W      x5,x4,(x9)  # x5=Mem[x9]; Mem[x9]=x4+x5
AMOADD.W.aq   x5,x4,(x9)  # x5=Mem[x9]; Mem[x9]=x4+x5
```

Description:

A value is read from the memory location whose address is in Reg1 and the value is placed into RegD. A binary operation is then performed on the value fetched from memory and the value in Reg2 and the result is written back to the memory location. In the case that Reg2 and RegD are the same register, the operation is performed using the initial value of the register; the value stored in the register will be the value fetched from memory.

This instruction contains an "aq" bit and an "rl" bit, which control the ordering of instructions appearing before and after this instruction.

The memory address must be properly aligned or else an "AMO Address misaligned" exception will be raised.

RV64 / RV128:

The result stored into RegD will be sign-extended to the full register size, i.e., to 64 or 128 bits.

Availability:

This instruction is only available in the "A" extension (Atomic Instructions).

Encoding:

These are R-type instructions.

Atomic Add / AND / OR / XOR / Max / Min (Doubleword)

General Form:

```
AMOADD.D.aq.rl    RegD,Reg2,(Reg1)   # addition
AMOAND.D.aq.rl    RegD,Reg2,(Reg1)   # logical AND
AMOOR.D.aq.rl     RegD,Reg2,(Reg1)   # logical OR
AMOXOR.D.aq.rl    RegD,Reg2,(Reg1)   # logical XOR
AMOMAX.D.aq.rl    RegD,Reg2,(Reg1)   # signed maximum
AMOMAXU.D.aq.rl   RegD,Reg2,(Reg1)   # unsigned maximum
AMOMIN.D.aq.rl    RegD,Reg2,(Reg1)   # signed minimum
AMOMINU.D.aq.rl   RegD,Reg2,(Reg1)   # unsigned minimum
```

Examples:

```
AMOADD.D      x5,x4,(x9)  # x5=Mem[x9]; Mem[x9]=x4+x5
AMOADD.D.aq   x5,x4,(x9)  # x5=Mem[x9]; Mem[x9]=x4+x5
```

Description:

A value is read from the memory location whose address is in Reg1 and the value is placed into RegD. A binary operation is then performed on the value fetched from memory and the value in Reg2 and the result is written back to the memory location. In the case that Reg2 and RegD are the same register, the operation is performed using the initial value of the register; the value stored in the register will be the value fetched from memory.

This instruction contains an “aq” bit and an “rl” bit, which control the ordering of instructions appearing before and after this instruction.

The memory address must be properly aligned or else an “AMO Address misaligned” exception will be raised.

RV64 / RV128:

These instructions are only available on RV64 and RV128.

Availability:

This instruction is only available in the “A” extension (Atomic Instructions).

Encoding:

These are R-type instructions.

Implementation: Each AMO operation requires both a read of memory and a write to memory, as well as a simple binary operation. It may be that the read-

operate-write functionality will be implemented completely within the memory subsystem, instead of within the core. Upon encountering an AMO instruction, the core would send the initial value of Reg2 to the memory system. The memory subsystem would then complete the operation, performing the atomic read-operate-write cycle and then sending the result value back to the core, to be stored in RegD. By offloading the AMO functionality to the memory subsystem, the atomicity is guaranteed by the memory system alone.

Implementation: Another approach to implementing the Atomic Memory Operations (AMO) would make use of the more primitive LR and SC operations. The microarchitecture might, for example, implement the AMO instructions in terms of the simpler functionality, which the microarchitecture also uses to implement the LR and SC instructions.

Sequential Consistency: A series of reads and writes to memory is said to be "sequentially consistent" if the following statements hold. (1) All of the read and write operations issued by all the HARTs can be linearized into a single, undisputed order, and the same order is observed by all HARTs. (2) The operations of any one HART must of course appear in this ordering in the order the HART actually called for them, although the operations from any two HARTs can be interleaved arbitrarily. (3) Or at least the results of the computation are indistinguishable from such a global, linear ordering of all operations.

To force all reads and writes to conform to sequential consistency, the programmer can use the atomic operations as follows. This will effectively linearize all reads and writes to memory, so that every HART will agree on the order that everything is performed.

For reads such as:

```
LD.W x4,(x7)
```

Substitute:

```
LR.W.aq.rl      x4,(x7)
```

For writes such as:

```
ST.W (x7),x4
```

Substitute:

```
AMOSWAP.W.aq.rl  x0,x4,(x7)
```

Setting the “aq” and “rl” bits forces all other HARTs to see instructions that occur before the instruction to appear to execute to completion before the instruction. Also, all instructions that occur after the instruction will appear to other HARTs to execute after the instruction.

Implementation: Note that an AMOSWAP operation that discards the previous value from memory (i.e., RegD=x0), can avoid the read phase of the instruction. This is still a useful atomic instruction, distinct from a store (ST) instruction, due to the presence of the “aq” and “rl” bits. Perhaps such an instruction can be optimized by the microarchitecture to avoid the read phase.

Chapter 8: Privilege Modes

Privilege Modes

At any time, the RISC-V processor is operating in exactly one of the following "modes":

U-Mode (User Mode)	← Lowest privilege
S-Mode (Supervisor Mode)	
M-Mode (Machine Mode)	← Highest privilege

Commentary: Previous versions included a fourth privilege level, in which a hypervisor was to execute. Hypervisor Mode was pretty much identical to Supervisor Mode and was flagged as "to be specified in the future."

U-Mode (User Mode)	← Lowest privilege
S-Mode (Supervisor Mode)	
H-Mode (Hypervisor Mode)	
M-Mode (Machine Mode)	← Highest privilege

Typically, user-level applications will execute in User Mode. Whenever the application wants an OS service, it will make a system call. The OS code that handles this call will execute in Supervisor Mode, i.e., at a higher privilege level. Upon return to the application, the mode will be lowered back to User Mode.

Most instructions can be executed in any mode, but some instructions are privileged and can only be executed in a higher mode.

The privilege levels are strictly ordered. Anything that can be done at a lower privilege level can be done in any one of the higher levels. For example, anything that can be done in User Mode can be done in Supervisor Mode.

Machine Mode is the highest privilege level; all instructions are legal at this level and nothing is protected. When the processor begins operation after powering up or after a reset, it is placed in Machine Mode.

The remaining modes (User and Supervisor modes) are pretty much the same as each other. In other words, there is little to distinguish them, except their relationship to each other. If you understand User Mode, then you will also understand most of Supervisor Mode.

There is one exception to the uniformity and this regards virtual memory and page tables. This functionality is located in Supervisor Mode.

The RISC-V spec describes the privilege mechanism in somewhat general terms and it is easy to imagine the insertion of additional levels of privilege, although it should be noted that a fourth mode (Hypervisor Mode), which existed in previous versions, has been eliminated.

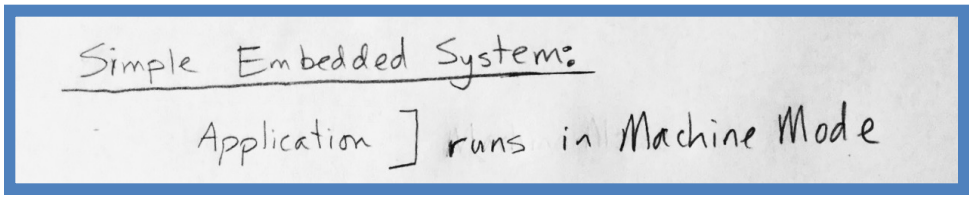
It appears that the current mode is not available in any user-visible register. In other words, it is difficult for code to determine the current mode. It cannot simply be read from a register; instead the current mode is implicit in the functionality that is available.

[This lack seems to be intentional. After all, code is typically written to run in one mode or another and rarely needs to ask which mode it is running in. For example, a hypervisor may wish to run a hosted OS is a restricted lower-privilege mode than the OS is meant to run at, thereby maintaining full control of the machine and preventing the hosted OS from corrupting the hypervisor itself or other hosted OSes. The hosted OS expects to run in a higher privilege mode than it actually is running at; the hypervisor must create and preserve the illusion that the OS is running in a higher privilege mode than it actually is.]

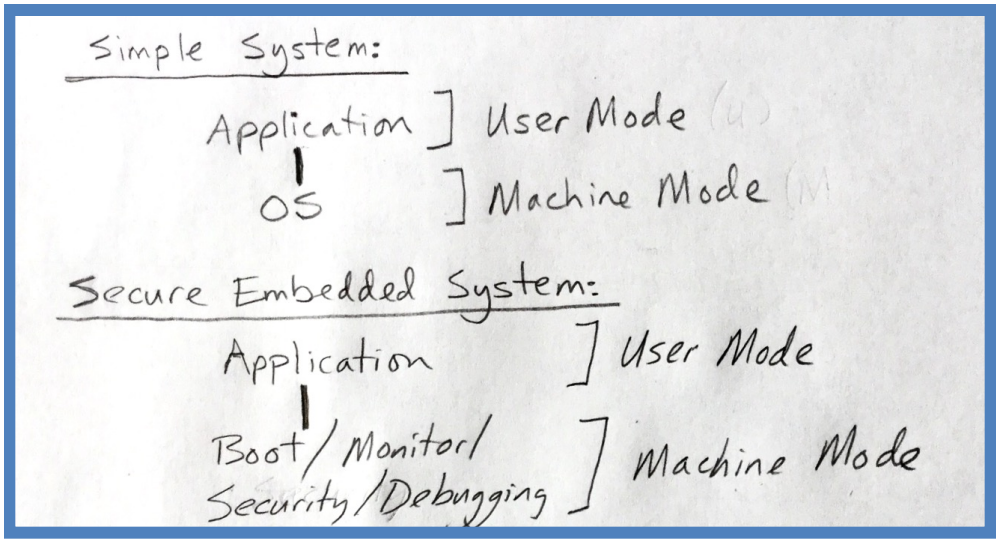
The RISC-V specification does not require all modes to be implemented. Some processors may not have all modes. Here are the legal possibilities:

Option 1	Option 2	Option 3
	U	U
		S
M	M	M

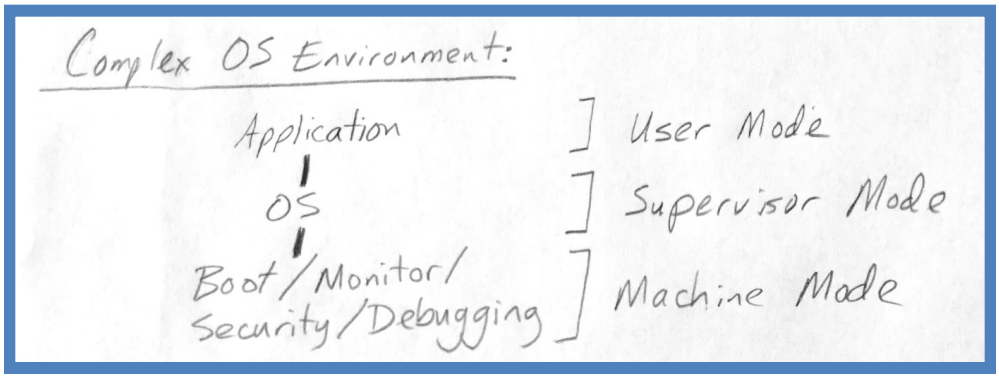
The most basic processor will have only Machine Mode. In a sense, there is no privilege system in such a simple processor since all operations can be performed at any time. An implementation like this would be suitable for an embedded microcontroller.



With option 2, there are two privilege levels: User and Machine. This might be appropriate for a simple OS with some protected security monitor running in Machine Mode and one or more applications running in User Mode.

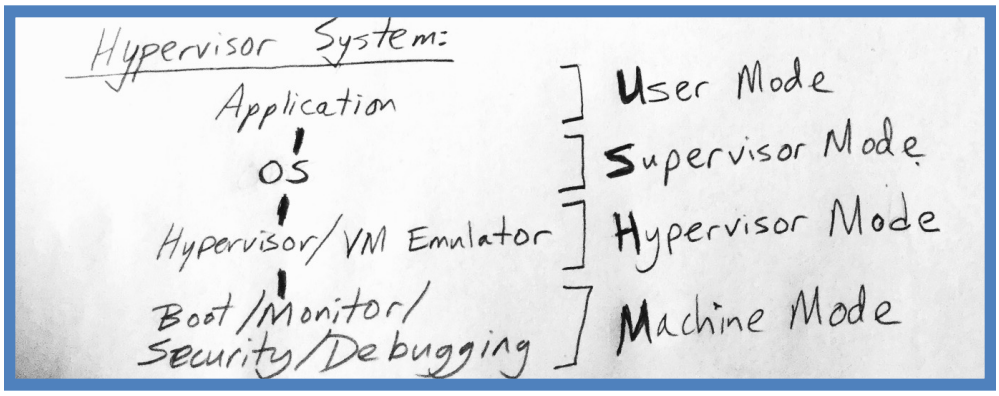


Option 3 is intended for more elaborate systems that will be running a traditional OS like Unix/Linux.



In some implementations several instructions and/or ISA features will be missing and there will be a need to emulate the missing functionality. The natural organization will place the code to emulate the missing instructions/features in Machine Mode so that the OS (which runs in Supervisor Mode) does not need to be aware of these details.

Previous RISC-V versions documented a Hypervisor Mode intended to support hypervisors. The hypervisor was to intended to run at its own privilege level, while each of the hosted OSes ran in Supervisor Mode. Although Hypervisor Mode has been eliminated, we include the following diagram anyway.

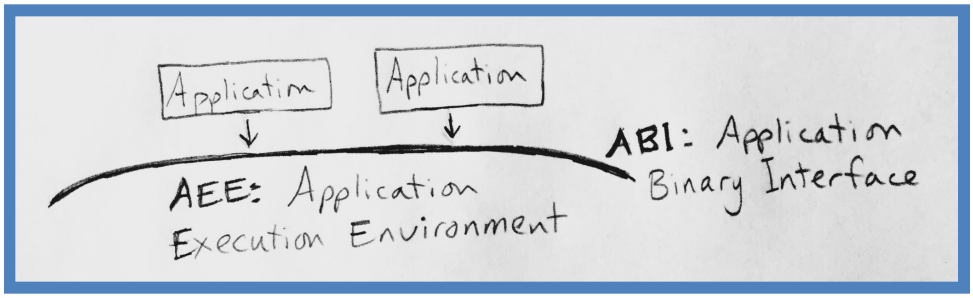


In the above diagrams we suggest which modes will be used for which sorts of code, but this is not mandatory. In particular, an operating system kernel might run in Machine Mode in some systems and in Supervisor Mode in other systems.

Interface Terminology

The RISC-V documentation discusses the interfacing between applications, operating systems, and hypervisors.

Every application program runs in a particular environment, which is called the "Application Execution Environment" (AEE). How the application interfaces with the underlying execution environment is called the "Application Binary Interface" (ABI).

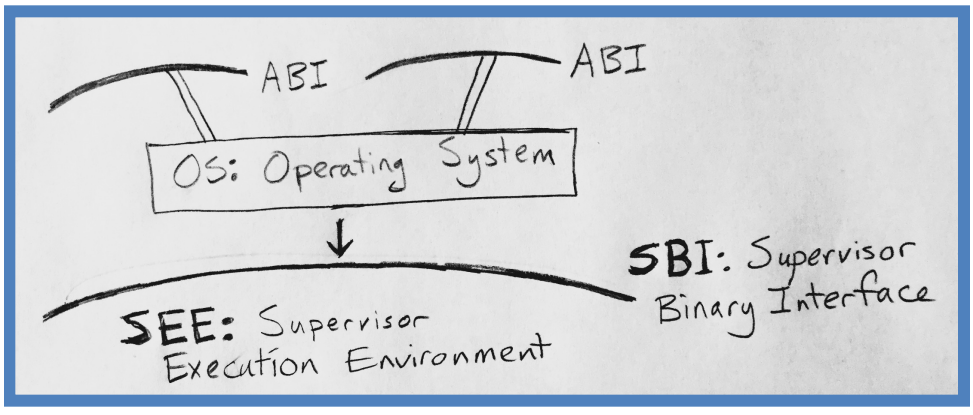


Typically an OS sits underneath the application and implements the Application Binary Interface, which consists of the User Mode instruction set along with the collection of system calls that are available to applications. The Application Binary Interface is the sum total of what the application programmer needs to understand in order to write programs; the programmer does not have to understand or know what is going on within the Application Execution Environment.

An example Application Binary Interface would combine the processor ISA along with the OS system-call interface.

If an application is ported to a new Application Execution Environment, then the application will function identically with no problems, as long as the new environment implements exactly the same Application Binary Interface.

Typically there are several applications running at any one time. The OS generally provides one Application Binary Interface and all applications are written to meet this specification. However, an OS might implement more than one Application Binary Interface, as shown in the next diagram.



The OS might be running on bare metal. Or the OS might be running on top of a hypervisor or some other monitor/security/bootstrapping software. In any case, the OS is created and written to meet the specifications of a “Supervisor Execution

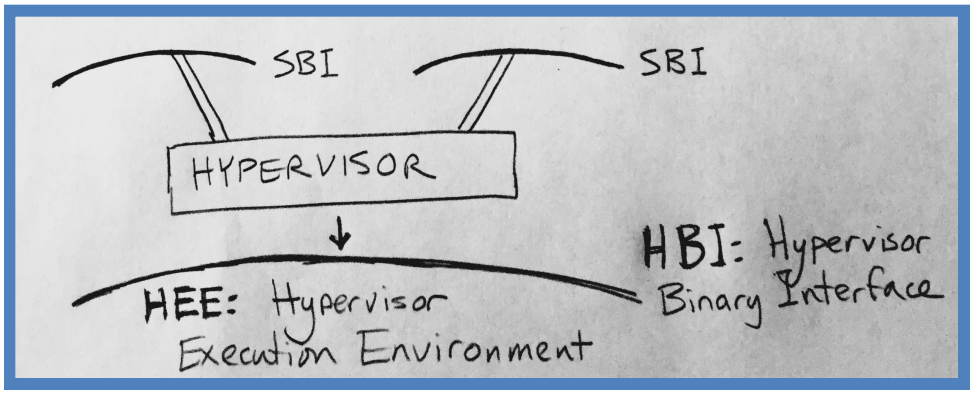
Environment" (SEE). This specification is called the "Supervisor Binary Interface" (SBI).

If the OS is ported to a new environment (for example, to new computer hardware), the OS should function identically, as long as the new execution environment implements the exact same Supervisor Binary Interface.

Typically an OS is designed to run directly on a bare machine with no underlying software. In this case, the SBI consists of a specification of the processor's ISA.

However, the OS may in fact be running on top of a hypervisor. As long as the hypervisor implements the correct Supervisor Binary Interface, the OS should be unaware that it is running on top of a hypervisor.

It is conceivable that the hypervisor provides multiple Supervisor Binary Interfaces. For example, one hypervisor might provide an SBI mimicking a PC and a second SBI mimicking an Apple computer. Such a hypervisor would than facilitate the simultaneous execution of both a MS Windows operating system and the Mac OS. Such a setup is shown below.



The hypervisor itself runs in some environment, which is called the "Hypervflaisor Execution Environment" (HEE). The underlying environment implements a particular interface, called the "Hypervisor Binary Interface" (HBI).

A "type-1 hypervisor" (or native hypervisor) is a program that is intended to run on a bare machine. In other words, there is no software below the hypervisor. The hypervisor achieves all its work using only the instructions available on the processor and does not make any system calls. For a type-1 hypervisor, the Hypervisor Execution Environment is just the processor core's instruction set architecture (ISA).

A “type-2 hypervisor” (or hosted hypervisor) is a program that is intended to run on top of other software. The hypervisor sits on top of some other operating system and, as far as the underlying OS is concerned, the hypervisor is just another application program. An example of this would be VMWare (which runs on top of the Apple MacOS and provides a Supervisor Binary Interface to mimic the PC so Windows can run. Other examples are QEMU and VirtualBox.

In order to run computer systems as efficiently as possible, hardware implementation for most instructions is desirable. The majority of common operations must be implemented directly in hardware, by instructions executed in the current mode.

However, some code will occasionally try to execute instructions that require software intervention. Perhaps a particular implementation fails to provide hardware to support certain operations; examples would be the lack of hardware support for “integer multiplication” or “floating point arithmetic”. Or perhaps the code is running in a hypervisor environment in which a particular instruction cannot be blindly executed, but must be intercepted, monitored, and/or altered before being executed. Regardless of the reason the instruction cannot be executed directly, an “exception” will occur when the instruction is encountered. The mode will be increased to a higher privilege level, software will intervene, and eventually a return to the original privilege mode will occur and instruction execution will resume.

All ISA designers try to minimize the number of operations that require software intervention and, when intervention is necessary, try to make the upcall to the exception handler quick and efficient.

Over the years we have accumulated good experience with the interface between User Mode code and operating system code. Much has been done to reduce the frequency of system calls and make the upcall interface fast and efficient.

More recently we are using more hypervisor software to support legacy operating systems. In order to implement the Supervisor Binary Interface (SBI) expected by the OS, the hypervisor may be obligated to step in frequently, whenever the OS executes a privileged instruction. Providing good ISA support for efficient hypervisors is an active research area.

Exceptions, Interrupts, Traps and Trap Handlers

During the execution of an instruction, an "exception" may occur.

Here are the different types of exceptions mentioned in the RISC-V spec:

Synchronous Exceptions:
- Illegal instruction
- Instruction address misaligned
- Instruction access fault
- Load address misaligned
- Load access fault
- Store/Atomic Memory Operation (AMO) address misaligned
- Store/ Atomic Memory Operation (AMO) access fault
- Environment call (i.e., the "syscall" trap instruction)
- Breakpoint

Asynchronous Exceptions (Interrupts):
- Timer interrupt
- Software interrupt
- External interrupt

A synchronous exception (sometimes just called an "exception") occurs as a result of trying to execute an instruction and is said to be synchronous since it is intimately connected with a particular instruction. The synchronous exception will be detected during instruction execution at the time the exceptional condition is detected. For example, during instruction decode, the hardware may detect a bad opcode field, which will initiate the "illegal instruction" exception processing.

The other type of exception is an "interrupt". During the execution of an instruction, an interrupt may occur. Interrupts are caused by events outside the current instruction execution and, as such, are asynchronous. Whenever an interrupt occurs, it will become associated with a single instruction (basically the currently executing instruction) and that instruction is chosen to receive the interrupt exception.

An exception can be handled or ignored. If the exception is handled, then a "trap" will occur. The trap processing involves a transfer of control to a "trap handler" routine. Trap processing consists of a few hardware operations, such as modifying a couple of hardware flags, saving the PC, and effecting a transfer of control to the first instruction of the trap handler routine.

A “trap handler” is a software routine and will usually be executed at a higher privilege mode than the instruction receiving the exception. For example, an “illegal instruction” exception might occur in application code running in User Mode. The trap handler will run in Supervisor Mode and, when complete, the handler may return to executing instructions in User Mode.

However, the change of mode may not always occur. In some cases, the trap handler will execute at the same privilege mode as the instruction receiving the exception. With RISC-V, some traps may be entirely contained in User Mode; the application code will be responsible for handling its own traps and supervisor code will not be involved.

A “timer interrupt” is caused when a separate timer circuit indicates that a predetermine interval has ended. The timer subsystem will interrupt the currently executing code. Timer interrupts are typically handled by the OS which uses them to implement time-sliced multithreading.

An “external interrupt” comes from outside the processor and the precise nature of the cause will depend on the application. For example, a RISC-V processor used in an embedded process control system might receive external interrupts from various sensors demanding attention.

A “software interrupt” is caused by setting a bit in the machine status word. This can be useful in a multi-core chip where a thread running on one core needs to send an interrupt signal to another core.

Control and Status Registers (CSRs)

In addition to the basic user-level instructions discussed previously, which can be executed by code running in any mode, there are a number of additional features that anyone writing an OS kernel code will need to understand.

At the center of the RISC-V privilege system are a number of **Control and Status Registers (CSRs)**, which are different from the general purpose and floating point registers. Each CSR has a unique name and specialized function.

Control of the privilege system is performed entirely by reading and writing the CSRs. OS/Kernel code can perform privileged operations solely by reading/writing to CSRs.

The RISC-V specification allows for up to 4,096 CSRs to be present, but in most implementations, not all addresses will be used. In fact, there are only a couple of dozen CSRs defined by the RISC-V specification. The other addresses are left open and different implementations may choose to add additional non-standard CSRs, up to the limit of 4,096 registers.

Each CSR has an address. Or, if you prefer, you can think of each CSR as having a number in the range 0 to 4,095.

Since there can be up to 4,096 CSRs, a 12-bit address is used to address each CSR. (Recall that 2^{12} = 4,096.) The instructions that read and write the CSRs use the I-type instruction format. This instruction format includes a 12-bit immediate field, and this field is used to contain the 12 bit address of the CSR register being operated on.

The size of the CSRs is specified to be the same size as the general purpose registers. So all the CSRs will be 32 bits in a RV32 machine. In a 64-bit machine, the CSRs are 64 bits wide. Likewise, in a machine with 128 bit registers, the CSRs are 128 bits wide.

In order to work with all sizes of machines, the specification never makes use of more than 32 bits in any CSR register. For 64-bit and 128-bit machines, the upper 32 or 96 bits are never used and are filled with zeros. [Actually, there are minor exceptions.]

Each CSR has a different meaning and use. Some CSRs are comprised of a collection of specialized fields (some fields are only 1 or 2 bits in length), while other CSRs contain a single full length value, such as a 32 bit address.

Each CSR belongs to one of the privilege modes. In other words, some CSRs are Machine Mode CSRs, some CSRs are Supervisor Mode CSRs, and the remaining CSRs are User Mode CSRs.

A Machine Mode CSR can only be read/written when the processor is in Machine Mode; it is illegal to access it when running in any other mode. A Supervisor Mode CSR can be read/written when the processor is in either Machine Mode or

Supervisor Mode, but not from code running in User Mode. A User Mode CSR can be read/written regardless of the current operating mode.

There are several instructions which are used to manage and control the privilege system, and to perform privileged operations. The instructions are:

ECALL – To make a system call from a lower privilege level to a higher mode
EBREAK – Used by debuggers to get control; similar to ECALL
URET – To return from trap handler that was running in User Mode
SRET – To return from trap handler that was running in Supervisor Mode
MRET – To return from trap handler that was running in Machine Mode
WFI – Go into sleep/low power state and Wait For Interrupt
CSR… – Instructions to read/write the Control and Status Registers (CSRs)

Writing to a CSR will change the state of the processor core. For example, to enable/disable interrupts, the program would write to one of the CSRs.

Reading/writing some CSRs is not allowed at lower privilege levels. Some CSRs are read-only at all privilege levels and are therefore set in stone by the chip designers, e.g., to describe the core's capabilities.

Key Idea: To understand the RISC-V privilege system and exception processing, it is both necessary and sufficient to understand the CSRs and their functionality.

CSR Listing

The Control and Status Registers (CSRs) are listed below for reference. Their functions will be described later.

For each CSR, we give its 12-bit address (using 3 hex digits). We also indicate whether it can be accessed in each of the privilege modes (User, Supervisor, Machine) and, if so, what sort of access (read-only or read/write) is allowed.

Recall that the processor is executing in one of the three modes at any moment. It is a privilege violation to try to access a CSR that is not visible in the current mode. Likewise, it is a privilege violation to try to write to a CSR that is read-only in the current mode. If this happens, an illegal instruction exception will be signaled.

Addr	Name	Description	Visibility in... U	S	M
	Counters:				
C00	**cycle**	Clock cycle counter	r	r	r
B00	**mcycle**				r/w
C80	**cycleh**	Upper half of cycle (RV32 only)	r	r	r
B80	**mcycleh**				r/w
C01	**time**	Current time in ticks	r	r	r
C81	**timeh**	Upper half of time (RV32 only)	r	r	r
C02	**instret**	Number of instructions retired	r	r	r
B02	**minstret**				r/w
C82	**instreth**	Upper half of instret (RV32 only)	r	r	r
B82	**minstreth**				r/w
	Exception Processing:				
000	**ustatus**	Status register	r/w	r/w	r/w
100	**sstatus**			r/w	r/w
300	**mstatus**				r/w
004	**uie**	Interrupt-enable register	r/w	r/w	r/w
104	**sie**			r/w	r/w
304	**mie**				r/w
005	**utvec**	Trap handler base address	r/w	r/w	r/w
105	**stvec**			r/w	r/w
305	**mtvec**				r/w
102	**sedeleg**	Exception delegation register		r/w	r/w
302	**medeleg**				r/w
103	**sideleg**	Interrupt delegation register		r/w	r/w
303	**mideleg**				r/w
106	**scounteren**	Counter enable		r/w	r/w

306	**mcounteren**				r/w

Trap Handling:

040	**uscratch**	Temp register for use in handler	r/w	r/w	r/w
140	**sscratch**			r/w	r/w
340	**mscratch**				r/w
041	**uepc**	Previous value of PC	r/w	r/w	r/w
141	**sepc**			r/w	r/w
341	**mepc**				r/w
042	**ucause**	Trap cause code	r/w	r/w	r/w
142	**scause**			r/w	r/w
342	**mcause**				r/w
043	**utval**	Bad address or bad instruction	r/w	r/w	r/w
143	**stval**			r/w	r/w
343	**mtval**				r/w
044	**uip**	Interrupt pending	r/w	r/w	r/w
144	**sip**			r/w	r/w
344	**mip**				r/w

Virtual Memory:

180	**satp**	Address translation and protection		r/w	r/w

Informational:

301	**misa**	ISA and extensions			r/w
F11	**mvendorid**	Vendor ID			r
F12	**marchid**	Architecture ID			r
F13	**mimpid**	Implementation ID			r
F14	**mhartid**	Hardware thread ID			r

Floating Point:

001	**fflags**	Floating pointing flags	r/w	r/w	r/w
002	**frm**	Dynamic rounding mode	r/w	r/w	r/w

003	**fcsr**	Concatenation of frm + fflags	r/w	r/w	r/w

Performance Monitoring:

323	**mhpmevent3**	Event selector #3			r/w
C03	**hpmcounter3**	Event counter #3	r	r	r
B03	**mhpmcounter3**				r/w
C83	**hpmcounter3h**	Upper half of counter (RV32 only)	r	r	r
B83	**mhpmcounter3h**				r/w
324	**mhpmevent3**	Event selector #4			r/w
C04	**hpmcounter4**	Event Counter #4	r	r	r
B04	**mhpmcounter4**				r/w
C84	**hpmcounter4h**	Upper half of counter (RV32 only)	r	r	r
B84	**mhpmcounter4h**				r/w
…	…	…	…	…	…
33F	**mhpmevent31**	Event selector #31			r/w
C1F	**hpmcounter31**	Event Counter #31	r	r	r
B1F	**mhpmcounter31**				r/w
C9F	**hpmcounter31h**	Upper half of counter (RV32 only)	r	r	r
B9F	**mhpmcounter31h**				r/w

Physical Memory Protection:

3A0	**pmpcfg0**	PMP Configuration word #0	r/w
3A1	**pmpcfg1**	PMP Configuration word #1	r/w
3A2	**pmpcfg2**	PMP Configuration word #2	r/w
3A3	**pmpcfg3**	PMP Configuration word #3	r/w
3B0	**pmpaddr0**	PMP Address #0	r/w
3B1	**pmpaddr1**	PMP Address #1	r/w
…	…	…	…
3BF	**pmpaddr15**	PMP Address #15	r/w

Debug/Trace:

7A0	**tselect**	Trigger register select	r/w
7A1	**tdata1**	Trigger data #1	r/w

7A2	**tdata2**	Trigger data #2	r/w
7A3	**tdata3**	Trigger data #3	r/w

7B0	**dcsr**	Debug control and status	r/w
7B1	**dpc**	Debug PC	r/w
7B2	**dscratch**	Scratch register	r/w

Instructions to Read/Write the CSRs

Regardless of which Control and Status Register (CSR) is involved, there are only 6 instructions that are used to read and write the registers:

CSRRW – Read and write a CSR
CSRRS – Read and set selected bits to 1
CSRRC – Read and clear selected bits to 0
CSRRWI – Read and write a CSR (from immediate value)
CSRRSI – Read and set selected bits to 1 (using immediate mask)
CSRRCI – Read and clear selected bits to 0 (using immediate mask)

There are a few other instructions that read or modify a CSR, but these additional instructions are merely special cases (i.e., shorthand or syntactic sugar) for one of the above instructions.

Each of these 6 instructions both reads and writes a single CSR in a single, atomic action.

Each of these instructions reads a CSR by copying its previous value into one of the general purpose registers (x1, x2, …). Often, you only want to modify the register and don't care about its previous value; if so, you can specify the destination register to be x0.

Each of these instructions is also capable of modifying the CSR. Often, you only want to read the register; if so, you can specify the source value as register x0. (This is a special case. The CSR is remains unchanged; it is not set to zero.)

The new value can be either a full value, or a bit mask which will be used to set or clear selected bits. The new value (or bit mask) can either come from a register or can be contained within an immediate data field within the instruction itself.

Whether code is allowed to access a particular CSR is dependent on the current privilege mode at which the code is being executed, as well as which CSR is involved. For example, if code running in User Mode attempts to read or update one of the Machine Mode CSRs (such as **mstatus**), this will not be allowed.

If an attempt is made to read and/or write to a CSR that is forbidden at the current privilege level, then an "Illegal Instruction" exception will occur. The instruction will not complete and exception processing will ensue.

Within a few of the CSRs, the protection is on a field-by-field basis. In other words, some of the bits may be protected from change (i.e., read-only), while the remaining bits can be freely changed (i.e., read-write). For example, within the **mstatus** CSR, it is possible to modify some bits but not allowable to modify other bits. (But for other CSRs, the entire register is either writable or not; that is, all bits have the same protection.) Any attempt to modify bits that are read-only will simply be ignored.

CSR Read/Write

General Form:

```
CSRRW   RegD,Reg1,Immed-12
```

Example:

```
CSRRW   x9,x4,cycle     # x9=CSR[0xC00]; CSR[0xC00]=x4
```

Description:

This instruction can be used to read from and/or write to a CSR.

The Immed-12 field is used to encode the address of one of the 4,096 Control and Status Registers (CSRs). The previous value of the CSR is copied to the destination register and the value of the source register Reg1 is copied to the CSR. This is an atomic operation.

In the above example, a CSR named "cycle" is being accessed. This CSR happens to have the 12-bit address 0xC00. This register contains a counter of the number of clock cycles since it was last written to. The counter is automatically incremented as instructions are executed.

To read a CSR without writing to it, the source register Reg1 can be specified as x0. To write a CSR without reading it, the destination register RegD can be specified as x0.

Comment:
In lower privilege modes some of the CSRs are inaccessible. An attempt to read from or write to a CSR may cause an illegal instruction exception.
Encoding:
This is an I-type instruction.

CSR Read and Set Bits

General Form:
```
CSRRS   RegD,Reg1,Immed-12
```
Example:
```
CSRRS   x9,x4,mstatus   # x9=CSR[0x300]; CSR[0x300]|=x4
```
Description:
The Immed-12 field is used to encode the address of one of the 4,096 Control and Status Registers (CSRs). The previous value of the CSR is copied to the destination register and then some selected bits of the CSR are set to 1. The value in Reg1 is used as a bit mask to select which bits are to be set in the CSR. Other bits are unchanged. This is an atomic operation.

In the above example, a CSR named "mstatus" is being accessed which has the 12-bit address 0x300. This register contains a number of bits which control which control interrupt processing.

This instruction can be used to simply read a CSR without updating it. If Reg1 is x0, then no update to the CSR will occur.
Exception:
May cause an illegal instruction exception if the current privilege mode is not high enough.
Encoding:
This is an I-type instruction.

CSR Read and Clear Bits

General Form:
```
CSRRC   RegD,Reg1,Immed-12
```
Example:
```
CSRRC   x9,x4,mstatus  # x9=CSR[0x300]; CSR[0x300]&=~x4
```

Description:

The Immed-12 field is used to encode the address of one of the 4,096 Control and Status Registers (CSRs). The previous value of the CSR is copied to the destination register and then some selected bits of the CSR are cleared to 0. The value in Reg1 is used as a bit mask to select which bits are to be cleared in the CSR. Other bits are unchanged. This is an atomic operation.

This instruction can be used to simply read a CSR without updating it. If Reg1 is x0, then no update to the CSR will occur.

Exception:

May cause an illegal instruction exception if the current privilege mode is not high enough.

Encoding:

This is an I-type instruction.

CSR Read/Write Immediate

General Form:

```
CSRRWI   RegD,Immed-5,Immed-12
```

Example:

```
CSRRWI   x9,3,mstatus   # x9=CSR[0x300]; CSR[0x300] = 3
```

Description:

The Immed-12 field is used to encode the address of one of the 4,096 Control and Status Registers (CSRs). The previous value of the CSR is copied to the destination register and then the entire CSR is written to. The 5-bit field that is normally used for Reg1 is zero-extended and used as the source value that is moved into the CSR. This is an atomic operation.

This instruction makes bits [4:0] in any CSR particularly easy to modify.

Exception:

May cause an illegal instruction exception if the current privilege mode is not high enough.

Encoding:

This is an I-type instruction.

CSR Read and Set Bits Immediate

General Form:

```
CSRRSI  RegD,Immed-5,Immed-12
```

Example:

```
CSRRSI  x9,3,mstatus  # x9=CSR[0x300]; CSR[0x300]|=3
```

Description:

The Immed-12 field is used to encode the address of one of the 4,096 Control and Status Registers (CSRs). The previous value of the CSR is copied to the destination register and then some selected bits of the CSR are set to 1. The 5-bit field that is normally used for Reg1 is zero-extended and used as a bit mask to select which bits are to be set in the CSR. Other bits are unchanged. This is an atomic operation.

This instruction makes bits [4:0] in any CSR particularly easy to set to "1".

Exception:

May cause an illegal instruction exception if the current privilege mode is not high enough.

Encoding:

This is an I-type instruction.

CSR Read and Clear Bits Immediate

General Form:

```
CSRRCI  RegD,Immed-5,Immed-12
```

Example:

```
CSRRCI  x9,3,status   # x9=CSR[0x300]; CSR[0x300]&=~3
```

Description:

The Immed-12 field is used to encode the address of one of the 4,096 Control and Status Registers (CSRs). The previous value of the CSR is copied to the destination register and then some selected bits of the CSR are cleared to 0. The 5-bit field that is normally used for Reg1 is zero-extended and used as a bit mask to select which bits are to be cleared in the CSR. Other bits are unchanged. This is an atomic operation.

This instruction makes bits [4:0] in any CSR particularly easy to clear to "0".

Exception:

May cause an illegal instruction exception if the current privilege mode is not high enough.

Encoding:
This is an I-type instruction.

Basic CSRs: CYCLE, TIME, INSTRET

There are three Control and Status Registers (CSRs) that are particularly easy to understand, so let's start with them:

cycle – counter of clock cycles
time – current real time
instret – counter of instructions retired (i.e., executed)

Each of these CSRs is automatically incremented as instructions are executed. All are readable at any privilege mode. Together they can be used to measure code speed and performance.

The CSR "cycle" counts the number of clock cycles. The CSR "time" measures real time in units of "ticks." The number of ticks per second is implementation dependent and can be determined elsewhere. The CSR "instret" counts the number of instructions retired (i.e., the number of instructions completed).

These CSRs are all User Mode registers, which means they can be read by code executing in any mode. These registers are read-only, so they cannot be written to.

We can reset these counters, but the reset operation requires higher privilege; User Mode code should not be able to write to these counters. To accommodate updating but only at higher privileges, these registers are "mirrored" by Machine Mode registers. The mirrored versions (called **mcycle**, **minstret**, …) have different names and different addresses. As such, they can only be written to by code running in Machine Mode.

To avoid possible overflow, these counters need to be 64-bits. For 64-bit and 128-bit machines, each of the three CSRs will be large enough to avoid overflow.

For 32-bit machines, the RV32 specification breaks each counter into two 32-bit CSRs. In other words, the RV32 specification introduces 3 additional registers to contain the high-order 32-bits:

cycle – clock cycles (lower 32 bits)
cycleh – clock cycles (higher 32 bits)
time – real time (lower 32 bits)
timeh – real time (higher 32 bits)
instret – instructions retired (lower 32 bits)
instreth – instructions retired (higher 32 bits)

The following instructions are short forms of other instructions. They are designed to make reading the above-mentioned counters especially easy.

Read "CYCLE"

General Form:
```
RDCYCLE  RegD
```
Example:
```
RDCYCLE  x9      # x9=CSR[cycle]
```
Encoding:
This is a simplified form of a more general instruction.

Read "CYCLEH"

General Form:
```
RDCYCLEH  RegD
```
Example:
```
RDCYCLEH  x9      # x9=CSR[cycleh]
```
Comment:
Only on 32-bit machines.
Encoding:
This is a simplified form of a more general instruction.

Read "TIME"

General Form:
```
RDTIME  RegD
```
Example:
```
RDTIME  x9      # x9=CSR[time]
```

Encoding:
This is a simplified form of a more general instruction.

Read "TIMEH"

General Form:
```
RDTIMEH  RegD
```
Example:
```
RDTIMEH  x9      # x9=CSR[timeh]
```
Comment:
Only on 32-bit machines.
Encoding:
This is a simplified form of a more general instruction.

Read "INSTRET"

General Form:
```
RDINSTRET  RegD
```
Example:
```
RDINSTRET  x9      # x9=CSR[instret]
```
Encoding:
This is a simplified form of a more general instruction.

Read "INSTRETH"

General Form:
```
RDINSTRETH  RegD
```
Example:
```
RDINSTRETH  x9      # x9=CSR[instreth]
```
Comment:
Only on 32-bit machines.
Encoding:
This is a simplified form of a more general instruction.

Example Code: The following sequence can be used to correctly read a 64-bit counter on a 32-bit machine where the counter is broken into two 32-bit pieces.

There is a possibility that, between reading the two halves, overflow from the lower-order 32 bits into the upper bits might occur.

```
again:
    rdcycleh    x3              # Read high bits
    rdcycle     x2              # Read low bits
    rdcycleh    x4              # Check for overflow
    bne         x3,x4,again     #   from low to high
```

CSR Register Mirroring

At any moment the processor is executing in some mode, i.e., in either User (U), Supervisor (S), or Machine (M) Mode.

The RISC-V spec defines a few dozen Control and Status Registers (CSRs) and they are broken into 3 groups. There is one group for each mode, so each CSR belongs to either User Mode, Supervisor Mode, or Machine Mode.

When running in Machine Mode all CSRs are accessible, regardless of which group they belong to, since Machine Mode is the highest privilege level. In Supervisor Mode, only the Supervisor and User CSRs can be accessed. Finally, when running in User Mode, only User CSRs are accessible.

To state the same thing another way, a Machine CSR can only be read or written when running in Machine Mode. A Supervisor CSR can be written when running in either Machine Mode or Supervisor Mode. Finally, a User CSR can be read or written regardless of the current privilege mode.

Each CSR has a name.

In many cases, the first letter of the register name indicates which group it is in (either “u”, “s”, or “m”).

Examples that follow the first-letter convention are **mstatus** and **misa** (accessible only in Machine Mode), **sstatus** and **satp** (accessible in Supervisor and Machine Modes), and **ustatus** and **ucause** (accessible in User, Supervisor, and Machine

Modes). Counter examples are **cycle**, **time**, and **instret** which ought to begin with "u" since they are accessible in User, Supervisor, and Machine Modes.

A particular implementation may add new CSRs in addition to those described in the RISC-V standard. Each CSR register has an address and the addresses are 12 bits wide, allowing for up to 4,096 CSRs.

Details: The address space for CSRs uses a 12-bit address, allowing for 4,096 different CSR registers. This address space is partitioned into 4 regions, with up to 1,024 registers each. There is one block of registers for each mode (U, S, and M) along with an extra "reserved" block which was formerly used for Hypervisor Mode. Hypervisor Mode has been eliminated.

Furthermore, each register also falls into one of the following access restriction categories:

registers defined by the RISC-V standard
- read-only
- read-write

extensions, not defined by the standard
- read-only
- read-write

registers used for debug mode
- standard
- non-standard extensions

This allows an implementation of the RISC-V spec to add new, non-standard CSRs without fear that their addresses will be overloaded in future revisions to the standard.

The mode (U, S, or M) and the access restriction category can be determined solely from bits in the CSR address. Thus, the hardware can easily check whether a given type of access (e.g., a read-write access from code running in User Mode) is allowed, simply by looking at the address. The encoding of bits in the CSR address is rather convoluted; consult the official documentation for details.

Some of the registers are mirrored, which means the same conceptual register is available at two or more addresses. For example, the **cycle** register, which contains a

counter of the current number of clock cycles, is mirrored in both User Mode and Machine Mode. A mirrored CSR will have two names and two addresses:

Address	Register Name
0xC00	**cycle** (read-only)
0xB00	**mcycle** (read-write)

This grouping and mirroring technique has a couple of advantages.

One advantage of mirroring the registers is that a single register can be read-only at one privilege level and yet writable at a higher privilege level.

For example, the "cycle" register (which counts how many clock cycles have occurred since last being reset) is conceptually a single register and would certainly be implemented as such. The register should be visible and readable to all code, but the reset operation (which writes to the register) should only be allowed when running at a higher privilege level. Thus, we want the "cycle" register to be read-only for User Mode code, but modifiable by Machine Mode code.

To achieve this, the register is "mirrored". There is a read-only CSR named **cycle** which can be accessed at any privilege level, and there is a different read-write CSR named **mcycle** which can only be accessed when running at the Machine Mode privilege level. Actually, there is only a single, underlying "mirrored" register, so any value written to **mcycle** will be immediately visible when reading from **cycle**.

Access to registers at a higher privilege level than the current mode is always forbidden. This allows an operating system to hide information from user-level code. For example, it might be used in a hypervisor to hide information so that an operating system can be truly fooled about what sort of processor it is running on.

This grouping mechanism is a uniform approach to preventing lower privilege code from modifying CSRs that must be protected. For example, user-level code must be prevented from modifying the virtual memory paging scheme and this happens naturally because the CSR associated with page tables is not a CSR that is accessible in User Mode.

Some registers contain a number of bit fields and these fields need to have different accessibility. For example, in the machine status register, some fields should be read-only while other fields are updatable. This is supported in the following way. An

attempt to update a field that is read-only will be ignored, even though other fields in the same CSR can be updated.

The "status" register is used in determining whether interrupts are enabled and what virtual memory mechanism is currently in effect, among other things. This register is mirrored at all privilege levels, using a different CSR at each level.

Address	Register Name
0x000	**ustatus**
0x100	**sstatus**
0x300	**mstatus**

Within the status register, some fields are invisible when running at lower privilege levels. This is supported by giving the register a different name for each mode. Some bits that are defined in the **mstatus** register will be read simply as zeros in the **sstatus** register, to reflect the fact that these bits must be accessible in Machine Mode but not in Supervisor Mode.

An Overview of Important CSRs

In this section we describe the most important Control and Status Registers (CSRs). This description goes hand-in-hand with describing features of a RISC-V pricessor not covered elsewhere and of concern only to programmers of OS and kernel code.

Recall that the first letter of the CSR register name will typically be **m**, **s**, or **u** to indicate which privilege level is required in order to access this register. For example, the "**misa**" register can only be read when running in Machine Mode, while the "**ustatus**" register may be accessed from any mode.

misa – Machine Instruction Set Architecture (ISA)

This CSR gives information about the basic architecture of the machine. It tells the register width (32, 64, or 128) and individual bits in this CSR also indicate which of the various options and extensions detailed by the RISC-V specification have been implemented. This register encodes whether this machine is an RV32IM, an RV64IMAFDQ, or some other variant of the RISC-V architecture.

The register width of the machine (either 32, 64, or 128) is encoded in the most significant two bits of this CSR. This is clever, since it makes it possible for the same code to be executed on different machines with different register widths. Using only instructions to test the sign of the register and shift the register left, the code can determine what the register size is, and then branch accordingly to different code blocks for each of the three possibilities.

This register is read-write.

Some machines may support multiple register widths. For example, an RV64 machine may be capable of running as (i.e., emulating) an RV32 machine. Upon power-on or reset, the **misa** register will be set to indicate the widest register width the core is capable of implementing. Software can set this register to effectively turn (for example) an RV64 machine into an RV32 machine.

The lower-order 26 bits correspond to the letters A, B, … Z ("A"=bit 0, "B"=bit 1, etc.) Each bit will be set to indicate whether this implementation supports the corresponding extension. For example, bit 5 will be set if the core supports the "F" single precision floating point extension.

mvendorid – Machine Vendor ID

For commercial implementations, this CSR identifies by number the vendor/ manufacturer/organization that has produced this chip. The number used here is the ID issued by a semiconductor engineering trade organization called JEDEC. For research and non-commercial implementations, this register will contain zero.

This register is read-only.

marchid – Machine architecture ID

This CSR identifies the particular architecture of the part and is essentially the "part number" or "model number". For commercial implementations, this number is assigned by the vendor. For some non-commercial or open-source projects, a number may be assigned by the RISC-V Foundation. Otherwise, this register will contain zero.

This register is read-only.

mimpid – Machine Implementation ID

Given a particular vendor (as identified in **mvendorid**) and a part/model number (as identified in **marchid**), there may be several versions. This number identifies the particular implementation or version of the processor. It may be zero.

This register is read-only.

cycle – Cycle Counter (read-only)
mcycle – Cycle Counter (writable)

The **cycle** register is accessible read-only in all modes. It counts hardware clock cycles. The counter can be reset by writing to the **mcycle** CSR, which is only accessible in Machine Mode.

instret – Instruction Counter (read-only)
minstret – Instruction Counter (writable)

The **instret** register is accessible read-only in all modes. It counts the number of instructions executed (or more precisely, the number of instructions completed "instructions retired"). The counter can be reset by writing to the **minstret** CSR, which is only allowable in Machine Mode.

time – Current Time (read-only)

The current real time in ticks. See the comments for **mtime**. This register is a shadow of **mtime**. The time CSR is accessible at all privilege levels.

Details: These registers all require 64 bits, which is problematic on RV32 machines. To deal with 32-bit machines, there are additional registers (whose names end in "**h**") to contain the upper 32 bits.

low order bits	*high order 32 bits (RV32 only)*
cycle	**cycleh**
mcycle	**mcycleh**
instret	**instreth**
minstret	**minstreth**
time	**timeh**

mtime – Current Time
mtimecmp – Time of Next Interrupt

The **mtime** register provides the current real time as a count of "ticks". The meaning of "tick" (i.e., the tick-rate of how many ticks per second and the value of time-zero, at which the counter began counting) is determined elsewhere.

The **mtimecmp** register is used to trigger the next timer interrupt. When the **mtime** register equals (or exceeds) the value in **mtimecmp**, a timer interrupt will be triggered.

Technically **mtime** and **mtimecmp** are not Control and Status Registers (CSRs), but they are listed here since they are similar. Each "register" is a 64 bit counter which is actually implemented via memory-mapping, unlike the CSRs.

Commentary: Putting a real-time clock inside the core is not practical since a core can be run at different frequencies at different times (e.g., under-clocking to reduce power consumption or over-clocking to… uh… cause it to malfunction). Typically there are several cores on a chip and the real time clock is essentially an I/O device shared by all cores, providing consistent time values to all cores. Thus, the real time clock is accessed by the use of memory-mapped accesses, like other I/O devices.

Typically processor chips are clocked by imprecise clock circuitry. In systems that aim to provide accurate real time clocks, the clock is not implemented on the same chip as the processor cores.

Presumably, the real time clock has one **mtimecmp** register per core, allowing each core to determine when its next timer interrupt will happen.

Commentary: The **mtimecmp** register is 64 bits but if the RISC-V core is only 32 bits, then there could be a problem setting it. It must be updated using two store-word (SW) instructions, each storing 32 bits. If the programmer is careless, the first load might trigger a spurious timer interrupt. The RISC-V documentation provides this code to avoid the problem:

```
    # The new value for mtimecmp is in registers x11:x10
    li    x5,-1           # 0xffffffff for lower 32 bits
    sw    x5,mtimecmp     # The 64-bit value is ≥ old value
    sw    x11,mtimecmp+4  # The 64-bit value is ≥ new value
    sw    x10,mtimecmp
```

mhartid – Machine Hardware Thread (HART) ID

RISC-V documentation defines the term HART to mean "Hardware Thread". This register does not reflect a higher level (e.g., operating system) concept of thread.

In a single-core system with a single, simple FETCH-DECODE-EXECUTE pipeline, there only one HART.

In a multi-core system, where each core will execute a single flow-of-control, each core will have its own HART. Each core's HART will execute concurrently with the other cores' HARTs. This CSR identifies which core is executing.

It may be important to identify one thread as a "master thread". One HART must be given an ID of zero.

The number of hardware threads is fixed but the application software will need an unpredictable and changing number of threads. The OS will map traditional OS threads onto the available hardware threads.

Some advanced superscalar cores may implement "hardware multithreading," in which a single core is capable of executing more than a one independent flow-of-control at a time. In other words, multi-threading is performed directly in hardware. For example, a single core might be able to execute two hardware threads at a time. The advantage of hardware multithreading is that the computational resources of the core (e.g., multipliers, adders, etc.) can be used more efficiently. In such a system, each hardware thread would be identified with a unique HART and each core would be executing more than one HART simultaneously.

This register is read-only.

mstatus – Machine Status Register
sstatus – Supervisor Status Register
ustatus – User Status Register

Conceptually, the processor "status register" is a single register, but it is mirrored at all privilege levels.

By modifying this register, the software can do things like enable/disable interrupts and change the ISA. (By setting bits in the status register, one can cause the processor to execute as if it were a 32 bit RV32 machine, even though it is actually a 64 bit RV64 machine.)

Understanding some of the field in this CSR requires understanding how exceptions are processed and trap handlers are invoked.

This CSR is complex and contains a large number of fields. As such, we delay further discussion until later.

mtvec – Machine Trap Vector Base Address
stvec – Supervisor Trap Vector Base Address
utvec – User Trap Vector Base Address

When a trap occurs (as a result of a synchronous exception or asynchronous interrupt), a jump will be taken directly to the trap handler routine.

There is a single trap handler for each privilege level. These registers contain the address of these three trap handlers. In other words, when an exception occurs (and is to be handled, not ignored), the program counter (PC) is set to the value in this CSR, causing a jump to the first instruction of the trap handler code.

The next section gives additional detail.

Exception Processing and Invoking a Trap Handler

Exceptions always trap to the Machine Mode trap handler first, regardless of the privilege mode at the time the exception occurs. The Machine Mode trap handler will execute in Machine Mode and (presumably) end with an MRET instruction, which will return to the code that was interrupted. The interrupted code may have

been running in User, Supervisor, or Machine Mode; the MRET instruction will restore the privilege level to whatever it was when the exception occurred.

However, we may not want to handle the exception in Machine Mode; we might want to handle it in Supervisor Mode or even User Mode. As such, there is a facility to "delegate" some or all exceptions to the lower privilege levels.

For example, if a particular type of exception should be handled by the Supervisor Mode trap handler, then this exception will be delegated from Machine Mode down to Supervisor Mode. The trap handler address will be determined from the **stvec** register (not **mtvec**); the handler will run in Supervisor Mode; and the handler will end with an SRET instruction.

Likewise, if a particular type of exception ought to be dealt with in User Mode, then any exception of this type will be further delegated from Supervisor to User Mode. The trap handler address will be determined from the **utvec** register; the handler will run in User Mode; and the handler will end with an URET instruction.

Whether an exception is to be delegated from Machine Mode to Supervisor Mode, or down further to User Mode, is determined by the settings of various bits in other CSRs. We describe the delegation ("deleg") CSRs elsewhere.

There is only a single trap handler at each of the three privilege levels (at least in the base specification). The **mtvec**, **stvec**, and **utvec**se CSRs are normally writable and OS software will store the address of the trap handler into a CSR before the exception occurs.

Once the exception occurs and the handler is invoked, its code must examine other CSRs to determine the nature of the exception, i.e., which exception caused the trap handler to be invoked.

(The trap handler addresses may be hardwired, in which case these CSRs are read-only. It is also allowable for there to be implementation-dependent restrictions on which values the **mtvec**/**stvec**/**utvec** CSRs can contain.)

Implementations are free to add additional trap handlers as extensions to the RISC-V spec. In particular, it may make sense for the processor to have a different trap handler for each kind of exception. This can make handling traps faster, since it eliminates the need for software to determine which type of exception caused the trap and jump conditionally.

The trap handler (or handlers, if there are more than one) must begin on word-aligned addresses. These means that any address stored in the **mtvec/stvec/utvec** CSRs must have "00" as the least significant two bits. The RISC-V spec makes use of these two bits as follows.

If the last two bits are "00", then it means the CSR contains the address of a single trap handler and it is up to the code in the handler to determine which type of exception has occurred.

If the last two bits are "01", then it means there is a collection of trap handlers, one for each type of asynchronous interrupt. This is good since we often want to handle asynchronous interrupts really quickly. More particularly, there is a jump table (i.e., an array of words, where each word contains a JUMP instruction) and the **mtvec/ stvec/utvec** CSR contains the address of this jump table.

[Details: When an asynchronous interrupt occurs, the CSR is consulted to locate the jump table. The interrupt type is consulted to determine which table entry is to be used (i.e., an index into the array). Then a jump is made to the associated entry, by loading the PC with the computed address. Presumably, each array element contains a jump instruction to the first instruction of the handler. Although there can be a different trap handler for each kind of asynchronous interrupt ("timer interrupt", "external interrupt", …), there will only be one handler for all synchronous exceptions ("illegal instruction", "address misaligned", …). The first element in the jump table (index=0) will be a trap handler for all synchronous exceptions, as well as for a "user-mode software interrupt".]

The remaining bit patterns "10" and "11" are not used.

The Non-Maskable Interrupt: Hardware Failure

Some traps are "maskable" and others are "non-maskable". A maskable interrupt can either be handled, or can be ignored, or can be passed from a higher privilege level to a lower privilege level.

A **non-maskable interrupt (NMI)** will be handled immediately and cannot be ignored. Only one event can cause a non-maskable interrupt:

- A hardware failure is detected (e.g., by error checking circuitry)

In the event of a hardware error NMI, the following happens:

- The privilege mode is set to Machine Mode.
- The previous PC will be saved in the **mepc** register.
- The PC is set to some fixed, predetermined, implementation dependent value.
- The **mcause** register is set to indicate the nature of the failure.

No other processor state is altered, which may be helpful in diagnosing the problem.

Reset Processing

The following events are not considered to be exceptions and are not handled as traps:

- Power-on-reset, triggered when the system is turned on
- Pressing a physical restart/reset button
- A watchdog timer times out
- Power levels drop below specs, triggering a "brown out" event
- A low-power sleep state is ended and the processor "wakes up"

In the above events, the following happens:

- The privilege mode is set to Machine Mode
- The MIE (Interrupt enable) bit in the status word is set to 0.
- The MPRV bit is set to 0, which turns off any address translation.
- The **mcause** register is set to indicate which event has occurred.
- The PC is set to some fixed, predetermined, implementation dependent value.

The remaining processor state is undefined.

Power-On-Reset: When power is first applied to a processor, the electronics will typically cause a "power-on-reset interrupt" to occur. Typically, this initial processing sequence will force a jump to a known, predetermined address in some read-only portion of memory by loading the Program Counter (PC) with some known value. The effect is to force a jump to the initial startup bootstrap program.

Watchdog Timer: A "watchdog timer" consists of an independent electronic circuit that is used to restart a computer that has experienced a catastrophic software failure, e.g., gone brain-dead in an infinite loop.

This technique can allow a computer to recover from otherwise fatal malfunctions such as program bugs and transient hardware errors. The watchdog timer technique has saved space probes that have had failures in flight: the probe may go silent for several days but then wakes up, performs a full reset, and continues to complete the mission.

A watch dog timer will cause a non-maskable interrupt after a certain amount of time has elapsed without the timer being reset. Resetting the timer is called “feeding the dog”. When software is working normally, the assumption is that the timer will be reset at regular intervals, thus feeding the dog. But if something goes wrong and the timer is not reset, the interrupt will occur.

Exception Delegation

We next look at the mechanism that allows exceptions to be masked and/or delegated to lower privilege levels.

By default, every exception (whether synchronous or asynchronous) is handled by a trap handler running in Machine Mode. If the kernel programmer wants the trap to be handled at a lower privilege level, then one possibility is for the programmer to write code to switch the mode and then pass control back to the lower privilege level.

However, this technique of software delegation is not very efficient and it is faster to have some exceptions automatically trap to a handler running at the lower privilege level. RISC-V supports delegation of traps by the hardware, as one possible design option. If this option is present in the implementation, then the exception will bypass the Machine Mode trap handler and go directly to a trap handler running at a lower level.

This is controlled by the following 4 delegation CSRs:

medeleg – Machine Exception Delegation Register
sedeleg – Supervisor Exception Delegation Register
and
mideleg – Machine Interrupt Delegation Register
sideleg – Supervisor Interrupt Delegation Register

We are using this terminology:

"exception" = synchronous exception (e.g., illegal instruction)
"interrupt" = asynchronous exception (e.g., timer interrupt)

If the bit corresponding to the exception type in the Machine Mode delegation registers (**medeleg** or **mideleg**) is set, then the Machine Mode trap handler will not be invoked. Instead, the trap will be immediately delegated to the next lowest privilege level, i.e., to Supervisor Mode.

Next, the appropriate bit in the Supervisor Mode delegation registers (**sedeleg** and **sideleg**) will be checked. If set, the trap will be further delegated to User Mode.

The trap cannot be further delegated, so there are no User Mode delegation CSRs.

Traps are always handled in Machine Mode, unless delegated to a lower level. The "deleg" CSRs tell which exceptions and interrupts are delegated to which privilege level.

For example, the handling of some particular trap might be delegated from Machine Mode to Supervisor Mode code. The trap might be further delegated to a trap handler running in User Mode.

These registers can be written to. They control whether individual exceptions and interrupts will be delegated, i.e., processed by a trap handler running at a lower privilege level, or will be processed at the higher level.

The **medeleg** register tells whether an synchronous exception will be handled by the Machine Mode trap handler (whose address is in the **mtvec** CSR), or will be delegated to the next lower level. If Supervisor Mode is implemented (in some chips it may not be), then the **sedeleg** register tells whether the exception will be handled in Supervisor Mode or further delegated to User Mode.

There is a bit in each "exception" delegation register (i.e., **medeleg** and **sedeleg**) for each type of synchronous exception, as follows:

Bit	Description
0	instruction address misaligned
1	instruction access fault
2	illegal instruction
3	breakpoint
4	load address misaligned
5	load access fault
6	store/atomic memory operation misaligned
7	store/ atomic memory operation access fault
8	environment call from U Mode
9	environment call from S Mode
10	*(previously used for Hypervisor Mode)*
11	environment call from M Mode

If the bit is set to 1, then the exception is delegated, i.e., handled by the next lower level. If the bit is 0, then the exception is handled at this level.

The **mideleg** register tells whether an asynchronous interrupt will be handled by the Machine Mode trap handler (whose address is in the **mtvec** CSR), or will be delegated to the next lower level. If Supervisor Mode is implemented (in some chips it may not be), then the **sideleg** register tells whether the interrupt will be handled in Supervisor Mode or further delegated to User Mode.

There is a bit in each "interrupt" delegation register (i.e., **mideleg** and **sideleg**) for each type of asynchronous interrupt, as follows:

Bit	Description
0	USIP – user software interrupt
1	SSIP – supervisor software interrupt
2	*(previously used for Hypervisor Mode)*
3	MSIP – machine software interrupt
4	UTIP – user timer interrupt
5	STIP – supervisor timer interrupt
6	*(previously used for Hypervisor Mode)*
7	MTIP – machine timer interrupt
8	UEIP – user external interrupt
9	SEIP – supervisor external interrupt
10	*(previously used for Hypervisor Mode)*
11	MEIP – machine external interrupt

If the bit is set to 1, then the interrupt is delegated, i.e., handled by the next lower level. If the bit is 0, then the exception is handled at this level.

mie – Machine Mode Interrupt Enable
mip – Machine Mode Interrupt Pending

There are three sources of interrupts (i.e., asynchronous exceptions):

- Software interrupt
- Timer interrupt
- External Interrupt

Software interrupts are used for one HART to signal another HART.

Timer interrupts occur when the real-time clock (**mtime**) reaches a preset limit value (**mtimecmp**).

External interrupts are for all other devices to interrupt a processor.

The **mie** register contains a bit for each type of asynchronous interrupt, as follows:

Bit	Description
0	USIE – user software interrupt enable
1	SSIE – supervisor…
3	MSIE – machine…
4	UTIE – user timer interrupt enable
5	STIE – supervisor…
7	MTIE – machine…
8	UEIE – user external interrupt enable
9	SEIE – supervisor…
11	MEIE – machine…

Likewise **mip** register contains a bit for each type of asynchronous interrupt, with analogous names:

Bit	Description
0	USIP – user software interrupt pending
1	SSIP – supervisor…
3	MSIP – machine…
4	UTIP – user timer interrupt pending
5	STIP – supervisor…
7	MTIP – machine…
8	UEIP – user external interrupt pending
9	SEIP – supervisor…
11	MEIP – machine…

(Bits 2, 6, and 10 were used for Hypervisor Mode and are now “reserved”. All remaining bits are unused.)

Setting a bit to 1 in the **mip** (interrupt pending) register indicates that the corresponding interrupt has occurred and should cause a trap at some point in the future. If the same bit is also set to 1 in the **mie** (interrupt enable) register, then the trap processing will occur immediately.

In addition to the pending bits discussed here, interrupts may also be globally enabled or disabled. See the **MIE** (interrupt enable) bit in the **mstatus** register.

Note that there is a bit in the status register called **MIE** and a CSR with the same name: **mie**. Likewise, there is an **SIE** bit in the status register, as well as a CSR called **sie**.

More precisely, an interrupt will be taken (i.e., the trap handler will be invoked) if and only if the corresponding bit in both **mie** and **mip** registers is set to 1, and if interrupts are globally enabled.

The timer interrupt pending bit (MTIP) is set by hardware when the timer expires, i.e., when the current time (**mtime**) reaches or exceeds the timer limit register (**mtimecmp**).

The timer interrupt pending bit (MTIP) is reset by writing to the timer comparison register. After writing to the timer comparison register, the timer interrupt will no longer be pending.

Software cannot modify the following interrupt pending bits; they are read-only and set/cleared by other circuitry:

3	MSIP – machine software interrupt pending
7	MTIP – machine timer interrupt pending
11	MEIP – machine external interrupt pending

However, software running in Machine Mode can modify these bits:

0	USIP – user software interrupt pending
1	SSIP – supervisor software interrupt pending
4	UTIP – user timer interrupt pending
5	STIP – supervisor timer interrupt pending
8	UEIP – user external interrupt pending
9	SEIP – supervisor external interrupt pending

If one of these bits is set and that interrupt type has been delegated to a lower level (see the **medeleg** and **mideleg** registers), then the interrupt will be signaled at the lower privilege level.

sip – Supervisor Mode Interrupt Pending
uip – User Mode Interrupt Pending

sie – Supervisor Mode Interrupt Enable
uie – User Mode Interrupt Enable

Each of the lower privilege levels has its own interrupt pending register and its own interrupt enable register.

Normally interrupts are processed at the highest privilege level. For example, if a timer interrupt occurs while running in User Mode, the core will enter Machine Mode and invoke the Machine Mode trap handler. When complete, the Machine Mode trap handler will return to executing in User Mode.

However, interrupts can be delegated from a higher level to a lower level. For example, the timer interrupt might be delegated from Machine Mode to Supervisor Mode. So if the interrupt occurs in User Mode, the Supervisor Mode trap handler will be invoked and Machine Mode will never be entered.

If a particular interrupt type (such as the timer interrupt) has been delegated (for example from Machine Mode to Supervisor Mode), then the corresponding interrupt pending bit is shadowed from the Machine Mode interrupt pending (**mip**) register.

Exactly what this means is unclear. Presumably the MTIP bit in **mip** shadowed as the STIP bit (and not the MTIP bit) in **sip**???

Whether the interrupt causes a trap in Supervisor Mode or not is determined by whether it is enabled in the Supervisor Mode **sie** register, as well as whether global interrupts are enabled in Supervisor Mode.

The Status Register

mstatus – Machine Status Register
sstatus – Supervisor Status Register
ustatus – User Status Register

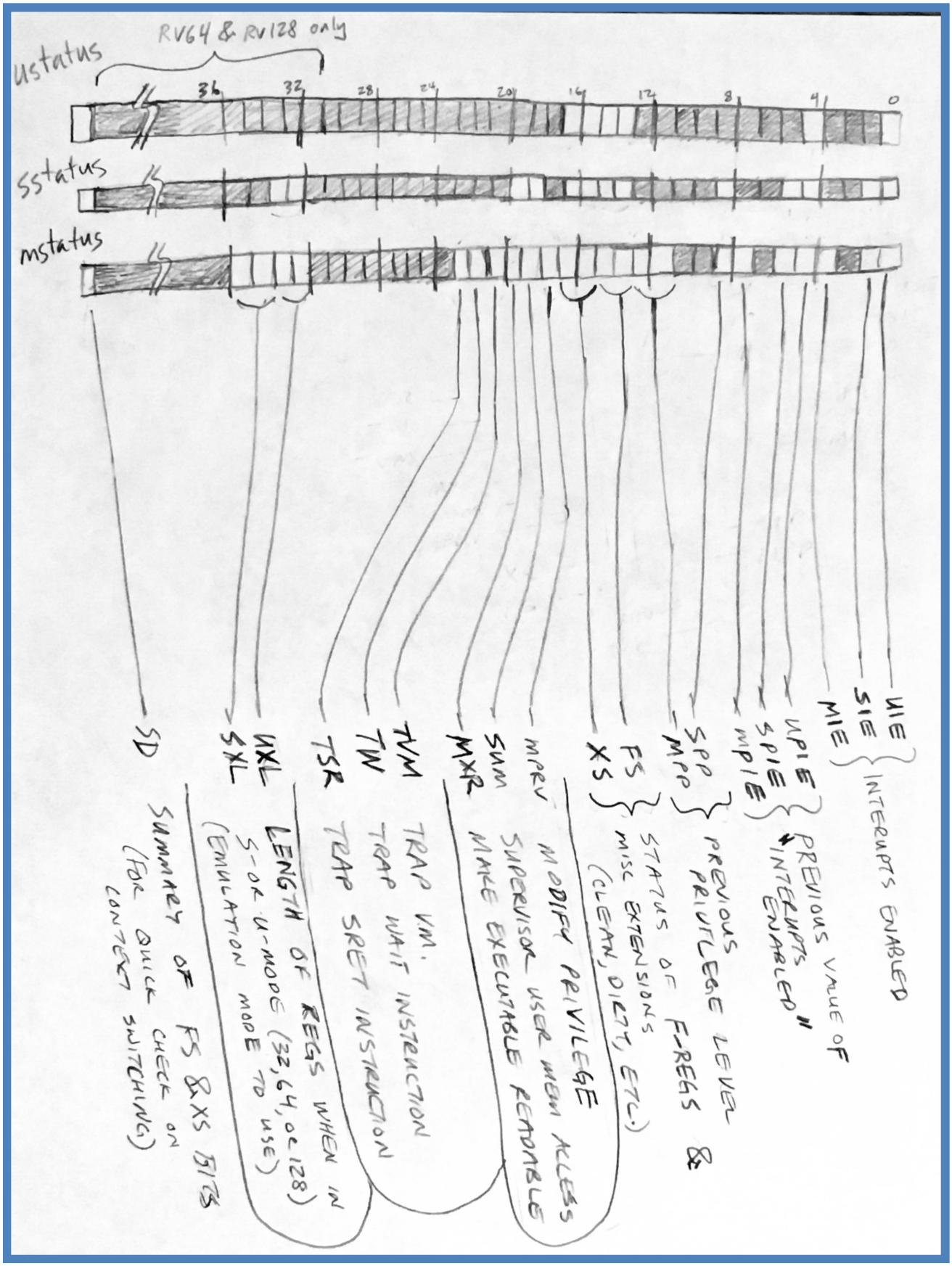
Conceptually, the processor status register is a single register, but the register is mirrored at the lower privilege levels, since some fields within the status register must be protected differently at different privilege levels.

This CSR contains a number of fields that can be read and updated. By modifying these fields, the software can do things like enable/disable interrupts and change the virtual memory model. For example, by writing to this CSR, the software can turn on virtual memory and page-table translation.

Next we show the layout of the status register, followed by a listing of the individual bit fields.

As mentioned, the status register is mirrored in the different modes. Some fields are present only at higher privilege levels. (Such bits are filled with zeros at lower privilege levels.)

Two of the fields are only used for 64 and/or 128 bit machines. These two fields reside in bits positions [35:32], so they are not even present in 32-bit machines.


RV64 & RV128 only
ustatus
sstatus
mstatus
63
32
28
24
20
16
12
8
4
0
UIE
SIE
MIE
INTERRUPTS ENABLED
UPIE
SPIE
MPIE
PREVIOUS VALUE OF "INTERRUPTS ENABLED"
SPP
MPP
PREVIOUS PRIVILEGE LEVEL
FS
XS
STATUS OF F-REGS & MISC EXTENSIONS (CLEAN, DIRTY, ETC.)
MPRV
MODIFY PRIVILEGE
SUM
SUPERVISOR USER MEM ACCESS
MXR
MAKE EXECUTABLE READABLE
TVM
TRAP VM.
TW
TRAP WAIT INSTRUCTION
TSR
TRAP SRET INSTRUCTION
UXL
SXL
LENGTH OF REGS WHEN IN S-OR U-MODE (32, 64, OR 128) (EMULATION MODE TO USE)
SD
SUMMARY OF FS & XS BITS (FOR QUICK CHECK ON CONTEXT SWITCHING)

Name	# of bits	Description
UIE	1	User Mode Interrupt Enable
SIE	1	Supervisor Mode Interrupt Enable
MIE	1	Machine Mode Interrupt Enable
UPIE	1	User Mode Previous Interrupt Enable
SPIE	1	Supervisor Mode Previous Interrupt Enable
MPIE	1	Machine Mode Previous Interrupt Enable
SPP	1	Supervisor Mode – Previous Privilege Mode
MPP	2	Machine Mode – Previous Privilege Mode
FS	2	Floating Point Status (dirty/clean/initial/off)
XS	2	User Mode Extension (dirty/clean/initial/off)
MPRV	1	Modify Privilege
SUM	1	Permit Supervisor User Memory Access
MXR	1	Make Executable Readable
TVM	1	Trap Virtual Memory
TW	1	Time-out Wait
TSR	1	Trap SRET instruction
SD	1	(FS == 11) or (XS == 11)

For RV64 and RV128 only:

UXL	2	Emulation: Register size when in User Mode
SXL	2	Emulation: Register size when in Supervisor Mode

MIE, SIE, UIE — Interrupts Enabled

Recall the term “interrupt” means asynchronous exceptions, i.e., Timer Interrupts, Software Interrupts, and External Interrupts.

If the processor is in Machine Mode, then interrupts are enabled whenever MIE = 1; interrupt processing for Supervisor and User mode is disabled.

If the processor is in Supervisor Mode, then supervisor interrupts are enabled whenever SIE = 1; interrupt processing for User mode is disabled and interrupt processing for Machine Mode is always enabled when running in Supervisor Mode.

If the processor is in User Mode, then user interrupts are enabled whenever UIE = 1; interrupt processing for Machine and Supervisor Mode is always enabled when running in User Mode.

MPIE, SPIE, UPIE, MPP, SPP — Interrupt Processing & Previous State

MPIE – Previous value of MIE (Machine Mode Interrupt Enable)
SPIE – Previous value of SIE (Supervisor Mode Interrupt Enable)
UPIE – Previous value of UIE (User Mode Interrupt Enable)
MPP – Machine Mode trap, previous Privilege Mode (either M, S, or U)
SPP – Supervisor Mode trap, previous Privilege Mode (either S or U)

When an interrupt occurs and a trap handler is to be invoked, the privilege mode will change and the "interrupt enable" bit must be set to 0 to prevent additional trap processing (at this level) while the trap handler is executing.

The processor may be executing at one privilege mode (say User) and the trap may be handled at a higher level (say Supervisor). If a Machine Mode trap then occurs before the Supervisor trap handler is complete, then the Supervisor Trap handler will be interrupted and the Machine Mode trap handler will execute. Upon return, the Supervisor Mode trap handler will resume execution; Finally, the interrupted User Mode code will be resumed.

It is necessary to save the machine state in a sort of "stack", so that upon completion of a trap handler, the processor can return to the previous state. Since a trap handler running at one level (say Supervisor) cannot be interrupted at that same level, the stack need not be too deep.

When an trap handler runs, all we need to save is the previous mode and the interrupt enable bit. Note the previous interrupt enable bit may be 0 or 1. Regardless of its previous value, the processor will change the interrupt enable bit to 0 when the trap handler is invoked and restore it when the trap handler returns.

(Of course, a non-maskable interrupt, such as a hardware failure, or a reset-type event, such as power-on-reset, might occur at any time. Such an event can never be masked and will always be handled regardless of whether or not interrupts are

enabled. But there is no problem since non-maskable interrupts are processed in Machine Mode and the previously executing code is simply abandoned.)

Consider what the processor will do upon the occurrence of an interrupt which will be handled by the Machine Mode trap handler. First, the MPIE bit will be set to hold the previous value of the interrupt enable bit for Machine Mode prior to the interrupt (i.e., to the previous value of MIE). Interrupts will then be disabled by setting MIE to 0. Also, MPP will be set to tell what privilege level the processor was running at when the interrupt occurred. The privilege mode in effect when the interrupt occurred may have been either Machine, Supervisor, or User Mode. Thus two bits are required to save the previous privilege level.

It is possible that an exception will occur during the execution of a trap handler. This means that, when exceptions are processed, the previously value of the "interrupts enabled" bit can be either 0 or 1. For example, it may be that some instruction is unimplemented and must be emulated. So when that instruction is encountered — either during normal code with interrupts enabled, or handler code with interrupt disabled — a new trap handler will be invoked. It will run with interrupts disabled and, upon completion, the "interrupts enabled" bit will be restored to its previous value.

The MIE ("Machine Interrupt Enable") bit determines whether a maskable interrupt will cause trap processing or will remain pending. If interrupts are disabled, then trap processing will be signaled but will remain pending until interrupts are once again enabled. [???]

Likewise, when an interrupt occurs that is to be handled by the Supervisor Mode trap handler, SPIE ("Supervisor Previous Interrupt Enable") will be set to the previous value of the SIE interrupt enable bit prior to the interrupt. Then SIE ("Supervisor Interrupt Enable") will be set to 0 to disable interrupts at this level. Then SPP ("Supervisor Previous Privilege") will be set to whichever privilege level the processor was running at when the interrupt occurred. Since the only choices are Supervisor or User, the SPP field is only one bit.

Similarly, when an interrupt occurs and is handled by the User Mode trap handler, UPIE ("User Previous Interrupt Enable"will be set to the previous value of the UIE ("User Interrupt Enable") interrupt enable bit prior to the interrupt. The interrupted code must have been running in User Mode, since we never interrupt a higher level mode to run a trap handler at a lower level. Therefore, there is no need for a

"UPP" ("User Previous Privilege") field to hold the previous mode; it must have been "User Mode".

Consider a trap which occurs while running in User Mode to a trap handler in Machine Mode. MPIE is set to the previous value of MIE; then MIE is set to 0; also MPP is set to "User Mode".

After the trap has been handled, an MRET instruction will be executed. MPP will be consulted to determine that the previous mode was User Mode. MIE is set to MPIE, restoring its initial value; the mode is restored to User Mode; MPIE and MPP are set to arbitrary values, since their information is effectively "popped" off the stack.

Note that the handler may choose not to return to the interrupted code; this is common in operating systems, particularly when servicing a timer interrupt; the handler will cause a thread switch and will not return to the interrupted code until much later. When this happens, the simple hardware stack system is inadequate. The handler must save the information in the status register and restore it later.

TSR — Trap SRET Instruction

If this bit is 1, then any attempt to execute an SRET instruction will cause an "illegal instruction" exception. If 0, then SRET may be executed without invoking a trap handler.

This bit is available on in the Machine Mode status word **mstatus**, not in Supervisor or User Modes.

The ability to intercept and trap an SRET instruction might be useful to hypervisor code.

TW and TVM — Time-out Wait and Trap Virtual Memory

These bits are available only in the Machine Mode status word **mstatus**, not in Supervisor or User Modes.

Their functionality is not documented in the current spec. Perhaps they have been eliminated. ???

SXL and UXL — Supervisor and User Register Size

An RV32 processor always uses the 32 ISA (e.g., 32-bit registers) regardless of what mode it is in. For 64-bit and 128-bit machines, there is a facility to execute code meant for a RISC-V processor with a smaller register size, i.e., in RV32 or RV64 mode.

The ability of larger machines to execute the RV32 or RV64 bit ISAs is not required, but if present, the SXL and UXL bits control it. If it is not implemented, then these two fields are read-only.

An RV64 bit processor always uses the 64-bit ISA when running in Machine Mode. However, it can execute in 32-bit mode when running in either Supervisor or User Mode. This is controlled by writing 01 to the SXL or UXL bits.

An RV 128 bit processor always uses the 128-bit ISA when running in Machine Mode. However, it can execute in 32-bit or 64-bit mode when running in either Supervisor or User Mode. This is controlled by writing to the SXL or UXL bits.

The encoding used for the SXL and UXL fields is:

01	32-bit ISA
10	64-bit ISA
11	128-bit ISA

FS, XS, and SD — Floating Status and Extension Status Bits

These bits of the status register are concerned with improving context switching times. First, we describe “FS” first, which is concerned with the floating point registers.

When the floating point registers have been used, they will need to be saved when the kernel switches from one software thread to another. But when the registers are unused, we’d like to avoid the overhead of saving and restoring them, since this is time consuming.

The FS bit field consists of two bits, encoded as follows:

Value	Meaning
00	Off
01	Initial Values
10	Clean, i.e., updated but saved in memory
11	Dirty, i.e., updated but not saved

These bits are set by the hardware and can also be modified by software. They also control whether a floating point instruction will cause an exception or will be executed normally.

(These bits are only present if the floating point extension is implemented. If there are no floating point registers, the FS bits are hardwired to 00.)

The FS bits should be set to 00="off" when the floating point registers are not in use. Any attempt to read or modify the registers will cause an illegal instruction exception. For a thread that will not be using the floating point registers, this is a good setting, since the OS can avoid saving and restoring the registers.

The complete state of the floating point system consists of the contents of of the 32 floating point registers and the floating point status register (fcsr). Many programs don't use floating point, so there is much to be gained by avoiding the save/restore of all this information on each context switch.

When a thread that uses floating point is created and begins life, the floating point registers should be initialized by the OS to their initial values (presumably +0.0) and the floating point status register should be initialized (presumably to zero).

The "initial" state (01) indicates that the floating point registers are usable but they have not yet been altered from their initial values. Thus, they register values do not need to be saved on the next context switch.

Whenever a floating point register is modified, hardware will change the state as reflected in FS to "11=dirty".

The "clean" state (10) is used to indicate that the registers are in use and have been modified from their initial zero values, but that they have not been modified since the last context switch. In other words, the registers contain non-zero values but the values last saved to memory are current and accurately reflect the values in the registers. Thus, at the time of the next context switch, the registers do not need to be

saved again. (This will be common since a program that uses floating point at some point will likely spend many time slices without further modifying the regs.)

In some OSes, it may be the case that processes always begin life with the floating point system enabled by default, so the "off" state is not particularly useful.

But in other OSes, processes will begin life with the floating point system disabled, under the assumption that most processes will not use floating point. Thus, the OS will set FS to 00=off. If an attempt is made to read or write the floating state, the hardware will signal an exception.

Some OSes may respond by enabling the registers, setting them to their zero values, and retrying the instruction. Other OSes may require the floating point system to be explicitly enabled by some system call and treat any accesses to disabled state as an error.

It is necessary for read (as well as write) attempts to cause an exception when the state is 00=off. The OS may occasionally leave the old state from another unrelated thread in the registers and we must prevent information from bleeding over or escaping from one thread to the next.

When a thread finishes its time slice, the OS can look at FS to determine how much state must be saved at the context switch:

Value of FS	Action
00=off	Saving not necessary
01=initial	Saving not necessary
10=clean	Saving not necessary; previously saved values are still good
11=dirty	Must save the registers

If the state was previously 11=dirty, then the OS should change it to "clean" before storing the thread's information, since it will now be true that the in-memory saved state is up-to-date.

When the OS is ready to initiate and schedule a new thread, it can look at the state of the FS from the previous thread and the state of the new thread to determine what to do. (Recall, that when the OS previously saved the regs when the thread was last de-scheduled, it changed the state from "dirty" to "clean", so the state of the new thread will never be "dirty" at this point.)

Next Thread	Previous Thread	Action
00=off	*...any...*	Do nothing
01=initial	01=initial	Do nothing (regs already zeroed)
01=initial	*...other...*	Zero the regs
10=clean	*...any...*	Restore regs from memory
11=dirty	*< not possible >*	

What we have just seen is that some section of the processor (the floating point subsystem) required attention at every context switch, either to save state or to restore state. These actions are expensive and we wish to avoid them when possible. The FS bits help determine when we can avoid saving/restoring.

Some RISC-V implementations may add non-standard extensions which are not described in the spec and the XS bits are used analogously for this additional state information.

The XS bits work the same way as the FS bits but cover all other state that should be saved / restored upon context switch. Exactly what processor state is covered by "XS" is left as "implementation defined".

It is possible that there are more than one non-standard extensions. The XS bits are meant to apply to all of them. So the meanings are changed slightly to reflect this.

Value	Name	Meaning
00	Off	All sub-systems are off
01	Initial	One or more are on; but nothing clean, nothing dirty
10	Clean	Nothing is dirty, some parts are clean
11	Dirty	At least some state has been modified

It may be that there are additional bits to determine exactly what parts of the processor state are dirty/clean/initial, but the spec leaves this to implementation specific decisions.

Finally, upon context switch, we'd like to be able to quickly determine whether we have to do anything about the floating point regs and any non-standard extensions. Might we need to save anything, or can we just ignore this issue? The question we want to answer is this: Is FS=11=dirty? Or is XS=11=dirty?

The SD bit in the status register allows the OS software to quickly answer these questions. The bit is set by the hardware and defined as:

SD = ((FS == 11) OR (XS == 11))

This bit is placed in the sign bit of the status word so a quick check can determine whether anything is dirty. If so, the OS software can look deeper to determine what state must be saved.

These fields (FS, XS, SD) are available in the Machine Mode **mstatus** and Supervisor Mode **sstatus** registers. [The diagram of the status registers incorrectly shows them in **ustatus**.]

MPRV, SUM, and MXR — Virtual Memory Enabling

The RISC-V chip supports virtual memory, via page tables. The page table system is described in a separate chapter.

Typically, user-level code will run with virtual memory mapping turned on and all memory accesses will be mapped from virtual addresses to physical addresses.

However, user-level code will occasionally make system calls, switching from execution in at a lower privilege level (e.g., User Mode) to a higher privilege (e.g., Machine Mode). Often data is passed between user-level code and the OS in memory; for example, the user-level code might pass an address pointer to the OS. This pointer is a virtual address and the OS code which services the system call will then go the memory to fetch the data.

Since the address is a virtual address while the OS is running in (say) Machine Mode with virtual memory mapping turned off, there is a problem. The OS could perform the address translation (from Virtual to Physical) in software, but this is time consuming and error-prone.

To make this task simpler, the MPRV bit can be used. Normally the MPRV is cleared to 0 and all LOAD and STORE instructions done by code running in Machine Mode will have no address translation performed. However, the OS code can set this bit to 1; any subsequent LOAD or STORE instructions will be performed with address translation turned on. More precisely, addresses will be translated as if the current

mode were that specified by MPP (the Machine Previous Privilege Mode bits in the **mstatus** register), which was the mode from which the most recent trap handler was entered.

(So a system call from User Mode, where paging was on, would pass virtual addresses. The handler, running in Machine Mode, would need address translation turned on if it wants to access the memory using these addresses. However a system call from Machine Mode itself (or Supervisor Mode with translation turned off) would pass physical addresses, since there would be no paging.)

The SUM bit is used by code running in Supervisor Mode instead of Machine Mode. It has no effect on code running in Machine Mode or in User Mode.

Address translation can be turned on by writing to the **satp** register. If turned on, then all User Mode and Supervisor Mode code will run with address translation.

Each virtual address space is divided into both user and non-user pages. In other words, some pages will be marked as "User" pages and the remainder will be inaccessible to User Mode code. (Typically the portion of the address space accessible to User Mode code is called the "bottom half" and the portion accessible only to Supervisor Mode code is called the "top half", although these two areas need not be arranged as contiguous blocks or located in any relative order.)

Normally Supervisor code will only access non-user (top half) pages in the address space. However, sometimes the supervisor code must access the user (bottom half) portion of the address space.

The SUM bit can be used to allow supervisor code to access the user pages. If SUM=0, then Supervisor Mode code cannot access pages marked as "user" pages; it can only access the non-user portion of the address space. If SUM=1, then Supervisor Mode code is free to address both user and non-user pages. This bit affects all loads, stores, and instruction fetches. Normally, supervisors code does not need to access user pages, so supervisor will run with this bit clear to prevent inadvertent accesses to user space. When the supervisor code needs to access user space, e.g., to retrieve syscall arguments, this bit can be temporarily set to 1.

Sometimes a trap handler will need to look at the instruction that was executing at the moment of an exception. For example, if the instruction is unimplemented in the hardware and must be emulated, the trap handler will need to read the instruction to determine how to emulate it.

Memory pages can be marked “readable” and/or “executable.” A page containing instructions will normally be marked “executable” but will not be marked “readable”. Therefore, any attempt to fetch a word from this page will cause an exception.

In order to allow the trap handler to fetch the instruction (as data), this exception must be suppressed. This is the function of the MXR bit in the status register. When set to 1, data reads are allowed from pages marked “executable”; when set to 0, such reads will cause an exception.

Additional CSRs

mepc – Program Counter for Machine Mode Trap Handler
mscratch – Scratch Register for Machine Mode Trap Handler
mcause – Cause Code for Machine Mode Trap Handler

When the Machine Mode trap handler is invoked, the previous value of the program counter is stored into **mepc**; this value will be used at the end of the trap handler, when an MRET instruction is executed. The **mepc** register is read/write and can be written by software (e.g., by an OS during a thread switch) to cause the MRET to jump into an arbitrary thread.

When a Machine Mode trap handler is invoked, the code must be careful not to overwrite and lose the previous values of any registers, since they were being used by the interrupted code. Instead the registers must be immediately saved. However, it is virtually impossible to do anything without the use of at least one general purpose register.

This is the purpose of the **mscratch** register, which is a read/write register. The trap handler can swap **mscratch** with any general purpose register and then use the normal ISA instructions to save the remaining registers.

Recall that the CSRRW instruction will exchange the value in an arbitrary CSR with a general purpose register. This instruction accomplishes the task of swapping **mscratch** with a general purpose register, thereby saving the general purpose register. This gives the software a register loaded with a base value, which can subsequently be used to save all remaining processor state.

The **mscratch** register cannot be changed by the execution of the code at lower privilege levels. Therefore it can be relied on to contain a known value, uncorrupted by user-level code. Typically it might contain a frame or stack pointer or a pointer to a "register save area." In any case, upon entry to the trap handler, **mscratch** can be swapped into a general purpose register and then used to save whichever registers the trap handler will be needing.

Commentary: Some earlier ISA designs used a general purpose register for this function. The idea was that one register would (by convention) be reserved for the OS kernel. Whenever an interrupt occurred, the OS kernel would be free to use this register in saving the state of the interrupted process. Therefore, this register was useless to user-level code. This register could not be used to hold a value since any time an interrupt occurred, it would be modified by the OS trap handler.

This design approach reduced the number of available general purpose registers available to user-level code.

Furthermore, the OS could not depend on the register remaining unchanged during user-level code execution; this necessitated loading the register with a known value before it could be used.

Also, if not consistently zeroed before returning from OS code to user-level code, data could potentially leak from the OS to user-level code, presenting security concerns.

The **mcause** register contains a numeric code to indicate what caused the trap. It is written by the hardware when an exception causes the Machine Mode trap handler to be invoked. Here are the possible exception cause values:

Numeric Code		Description
0		Instruction address misaligned
1		Instruction access fault
2		Illegal Instruction (or privileged instr. violation)
3		Breakpoint
4		Load address misaligned
5		Load access fault
6		Store / AMO address misaligned
7		Store / AMO access fault
8		Environment call from U-Mode
9		Environment call from S-Mode
11		Environment call from M-Mode
12		Instruction page fault
13		Load page fault
15		Store/AMO page fault
MAX+0	0x80000000	User Software Interrupt
MAX+1	0x80000001	Supervisor Software Interrupt
MAX+3	0x80000003	Machine Software Interrupt
MAX+4	0x80000004	User Timer Interrupt
MAX+5	0x80000005	Supervisor Timer Interrupt
MAX+7	0x80000007	Machine Timer Interrupt
MAX+8	0x80000008	User External Interrupt
MAX+9	0x80000009	Supervisor External Interrupt
MAX+11	0x8000000B	Machine External Interrupt

Details: Each cause has a numeric code. The code scheme uses the sign bit to show whether the cause is a synchronous exception (MSB=0) or an interrupt (MSB=1). For asynchronous interrupts, a small code number (0, 1, 2, …) is stored in the lower-order bits, and the sign-bit is also set. So in the above table, MAX stands for 2^{31} for a 32-bit machine, 2^{63} for a 64-bit machine, etc. To make this clear, we give the actual value in hex for RV32.

Commentary: A "privileged instruction violation" is an instruction that exists but cannot be executed in the current privilege mode. An illegal instruction is an invalid instruction encoding which does not represent a defined instruction. Concerning exceptions raised by either of these violations, RISC-V makes no

distinction between "privileged instruction" and "illegal instruction." Both cause the "illegal instruction" exception.

sepc – Program Counter for Supervisor Mode Trap Handler
sscratch – Scratch Register for Supervisor Mode Trap Handler
scause – Cause Code for Supervisor Mode Trap Handler

uepc – Program Counter for User Mode Trap Handler
uscratch – Scratch Register for User Mode Trap Handler
ucause – Cause Code for User Mode Trap Handler

These registers are defined analogously, to be used in the trap handlers running in Supervisor or User Mode.

Some cause values cannot show up in the cause registers. For example, a Supervisor Mode trap handler cannot ever be confronted with a Machine Mode timer interrupt.

Questions: But can't any interrupt be delegated; would it still remain a Machine Mode timer interrupt, or would it be altered to a Supervisor Mode timer interrupt ???

There is no "Load address misaligned" code for the **scause** register, implying that such a exception cannot be handled in Supervisor Mode. Why not ???

Also, the official documentation does not include a cause code for "Environment call from S-Mode" as a possible value in **scause**; is this a mistake in Table 4.2 ???

scounteren – Counter Enable Register for User Mode
mcounteren – Counter Enable Register for Supervisor or User Mode

Code running in a lower privilege level (such as User Mode) may periodically try to read the various timer and counters. Code running at a higher level (such as an OS or hypervisor) may wish to intercept all accesses to counters and timers in order to fool the lower privilege code. Perhaps the hypervisor wants to present the illusion to the lower level code that it is running on a bare machine when, in fact, it is not. This could also be useful for viruses which want to hide their existence.

These CSR register are used to control access to the following counter/times CSRs: **cycle**, **time**, **instret**, **hpmcounter3**, **hpmcounter4**, … **hpmcounter31**.

The purpose of the **scounteren** register is to allow Supervisor Mode code to determine whether accesses to one of the counter/timer registers by code running in User Mode should cause an exception or not.

The purpose of the **mcounteren** register is to allow Machine Mode code to determine whether accesses to one of the counter/timer registers by code running in User Mode or Supervisor Mode should cause an exception or not.

There is one bit for each of the 32 counters/timers. 0=disabled, cause an exception; 1=enabled, no exception.

Emulation of time / mtime Accesses: Another use of this CSR is as follows: Recall that the realtime clock **mtime** is not actually a CSR, but is expected to be implemented as a memory-mapped I/O device. However there is a true CSR named **time**, which is meant to be a mirror of **mtime**. By trapping all reads to the **time** CSR, Machine Mode code can performed the I/O to retrieve the time from **mtime** and properly emulate the read to the **time** CSR.

System Call and Return Instructions

Environment Call (System Call)

General Form:

```
ECALL
```

Example:

```
ECALL       # no operands
```

Comment:

This instruction is used to invoke a trap handler. This instruction causes one of the following exceptions, depending on the current privilege level:

- Environment Call from User Mode
- Environment Call from Supervisor Mode
- Environment Call from Machine Mode

Any and all arguments and returned values must be passed in registers.

Encoding:

This is a variant of an I-type instruction.

Break (Invoke Debugger)

General Form:

```
EBREAK
```

Example:

```
EBREAK      # no operands
```

Comment:

This instruction is used to invoke a debugger, by causing an "Breakpoint" exception. Typically the debugging software will insert this instruction at various places in the application code sequence, in order to gain control from an executing program.

Encoding:

This is a variant of an I-type instruction.

MRET: Machine Mode Trap Handler Return

General Form:

```
MRET
```

Example:

```
MRET      # no operands
```

Comment:

This instruction is used to return from a trap handler that is executing in Machine Mode. The **MPP** field of the status register will be consulted to determine which mode to return to (either m, s, or u). The value in the **MPIE** field will be copied to **MIE**, which will restore the "interrupts enabled" state to what it was when the handler was invoked. The return will be effected by copying the saved program counter from **mepc** to the Program Counter (**pc**).

This instruction may only be executed when running in Machine Mode.

Encoding:

This is a variant of an I-type instruction.

SRET: Supervisor Mode Trap Handler Return

General Form:

```
SRET
```

Example:

```
SRET        # no operands
```

Comment:

This instruction is normally used to return from a trap handler that is executing in Supervisor Mode. The **SPP** field of the status register will be consulted to determine which mode to return to (either s or u). The value in the **SPIE** field will be copied to **SIE**, which will restore the "interrupts enabled" state to what it was when the handler was invoked. The return will be effected by copying the saved program counter from **sepc** to the Program Counter (**pc**).

This instruction may be executed when running in either Supervisor Mode or Machine Mode.

Encoding:

This is a variant of an I-type instruction.

URET: User Mode Trap Handler Return

General Form:

```
URET
```

Example:

```
URET        # no operands
```

Comment:

This instruction is normally used to return from a trap handler that is executing in User Mode. User Mode trap handlers always return to User Mode code. The value in the **UPIE** field will be copied to **UIE**, which will restore the "interrupts enabled" state to what it was when the handler was invoked. The return will be effected by copying the saved program counter from **uepc** to the Program Counter (**pc**).

This instruction may be executed in any mode.

Encoding:

This is a variant of an I-type instruction.

WFI: Wait For Interrupt

General Form:

```
WFI
```

Example:

```
WFI       # no operands
```

Comment:

This instruction causes the processor to suspend instruction execution. When an asynchronous interrupt occurs, the processor will wake up and resume execution. The trap handler will be invoked and, upon return to the code sequence containing the WFI instruction, the next instruction following the WFI will be executed.

Typically, software will execute this instruction within an "idle" loop, which is only executed when there is nothing to be done. Suspending instruction execution may save power. In a system with multiple cores, this instruction may also provide a hint to the interrupt circuitry to route future interrupts to this core, since it is idling.

This instruction may be implemented as a nop. For simpler processors, it may be better to just spin in an idle loop. Complex systems may have an inexhaustible supply of background processes, making a low-power wait-state pointless.

If interrupts are (globally) disabled when a WFI instruction is executed, then this disabling is ignored. Instruction execution will resume when an interrupt occurs. See the spec for details.

Encoding:

This is a variant of an I-type instruction.

Chapter 9: Virtual Memory

Physical Memory Attributes (PMAs)

Main memory is the large range of byte-addressable locations. Each location has a "physical address".

In systems without virtual memory mapping, each and every address generated by an instruction is a physical memory address. However, many systems will support virtual memory mapping, where there is a distinction between virtual addresses and physical addresses. An instruction (e.g., in a READ or WRITE instruction) will generate a virtual address and the memory management unit (MMU) will translate this virtual address into a physical address, which is then used by the hardware to send or retrieve data to/from the main memory store.

The physical address space can be populated by:

- Normal memory
- Memory-mapped I/O devices
- Empty (i.e., unpopulated holes)

I/O devices are commonly "memory mapped", which means they sit on the same bus as normal memory. The device is assigned a specific physical address (or set of addresses). Software can send data to an I/O device by "writing" to a memory location populated by the device and can retrieve data from the device by "reading" from such an address. From the point of view of the executing software, the device looks a bit like any other memory location, but reading and writing to memory-mapped I/O locations is done, not to store data, but for the purpose of sending data to, receiving data from, and otherwise controlling an I/O device.

To handled various scenarios, the physical address space will be partitioned into regions. The regions will be contiguous, one-after-the-other, non-overlapping and will cover the entire physical address space. Every byte in the physical address space will be in exactly one physical memory region.

Each region can be characterized by different attributes. For example:

Is the region populated with main memory?
Is the region populated with memory-mapped I/O registers?
Is the region (i.e., unpopulated, empty)?
Does the region support reads (i.e., data fetches)?
Does the region support writes?
Does the region support execution (i.e., instruction fetches)?
Does the region support atomic operations?
Is the region cached?
Is the region shared with other cores, or private?

Collectively, these are "physical memory attributes" (PMAs). Some regions will have their attributes fixed and unchangeable, other regions will be configurable upon power-up (i.e., cold-pluggable), and other regions might have their attributes change during system operations (i.e., hot-pluggable).

Memory attributes must checked constantly during execution. For example, a READ from an unpopulated memory region ought to cause an exception.

Where shall the PMA information be stored? One approach is to store the information in the virtual memory tables, and require the memory management unit (MMU) to check and enforce the constraints. There may be problems when the page size of the memory management unit doesn't match the size of the regions.

In the RISC-V design, there is a separate subsystem in the core which is concerned with checking and enforcing the physical memory attributes. This subsystem is called the "PMA checker".

When an instruction attempts to access memory, the address is first translated from virtual to physical address (if virtual memory is turned on). Then, the PMA checker will verify that the type of access is legal. If not, an exception will be generated. There are several exception types for physical memory attribute (PMA) violations and these are different from the virtual memory fault exceptions.

The information about the various memory regions and their associated PMAs will be kept in processor registers. These registers (or parts of them) may be hardwired for some regions (which is appropriate when the region itself is preconfigured and not modifiable, such as an on-chip ROM), or may be readable (for querying the bus

to determine what regions are present and their PMAs), or may be writable (allowing the software to establish and mandate PMAs). Details are implementation dependent.

Concerning the atomic operations of RISC-V, it is important to determine whether a particular region can support a particular instruction.

Regions populated by main memory must support all the AMO instructions (Atomic Memory Operations) and LR/SC instructions (assuming these instructions are even implemented, of course).

Regions populated by memory mapped I/O devices are not expected to support the LR/SC instructions.

Memory-mapped regions may or may not support the AMO operations. If such a region supports any AMO instructions, then it must support AMOSWAP at a minimum. The region may also optionally support other AMO instructions. Here are the levels of AMO support that may be implemented:

- No AMO operations supported
- Only AMOSWAP supported
- AMOSWAP, AMOAND, AMOOR, AMOXOR
- All AMO operations are supported

Concerning the FENCE instructions, recall that accesses are classified as being "memory" or "I/O". To support this, every region of the physical memory address space must be classified as either:

- main memory
- I/O

It is possible that two different cores (or other devices sitting on the physical memory bus) may access the same memory locations more-or-less simultaneously. Accesses to main memory regions are "relaxed", which means that the exact ordering of the operations is indeterminate (unless atomic instructions are used, of course). The programmer can use atomic instructions to force particular orderings, but for normal instructions, there are no guarantees about ordering. A single core will see its own operations executed in the order they were issued, but reads and writes from other cores or devices can appear at any time, in any order.

Access to memory-mapped regions may either be “relaxed” or “strong”. If the region is defined to have relaxed ordering, then the ordering of operations is undefined, as just described. If the region has strong ordering, then it will appear to one core (A) that the operations executed by some other core (B) were all executed in the order they were actually issued by core (B).

This summary is a simplification of the RISC-V spec. The spec is designed to apply to a spectrum of different kinds of systems, from simple to complex. Modern complex systems may have multiple parallel channels between cores, devices, and main memory stores. Imagine that all the devices are connected by a network with the properties of the Internet: messages are sent between devices, can take different paths, and can be arbitrarily reordered in transit. On the other hand, older and simpler systems have a single, simple bus that does one thing at a time and reordering is inconceivable; these systems may use device drivers that were designed without concern for reordering and one goal is to support such architectures and legacy programming.

Cache PMAs

qqqqq

PMA Enforcement

The enforcement of Physical Memory Attributes (PMAs) is optional.

A RISC-V core may optionally provide a Physical Memory Protection (PMP) unit. Physical memory protection is implemented using a small number of Machine Mode CSRs that describe the memory regions and their PMAs.

The CSR registers encode the following information for each region:

- The starting address of the region
- Whether the region is 4, 8, 16, 32, … bytes in length
- Does the region support READ operations?
- Does the region support WRITE operations?
- Does he region support EXECUTION fetches?

- Is this region “locked”?

Up to 16 regions can be described in the registers. This information is cleverly encoded in 16 registers plus a couple of additional registers.

The PMP Unit is active when running in Supervisor or User Mode; all accesses generated are checked. The protection may or may not be enforced when running in Machine Mode.

Since these registers are Machine Mode CSRs, they can only be accessed in Machine Mode.

There is a single “Lock” bit for each one of the 16 regions and it works as follows. If the lock bit is 0, then Machine Mode accesses are not checked; only Supervisor and User Mode accesses are checked.

If the Lock bit is set, then all Machine Mode bits are checked as well.

Once a Lock bit is set, it cannot be cleared. All lock bits are initially clear; to clear a lock bit requires the system to be fully reset (e.g., POWER-ON-RESET). In other words, once a region become locked, it stays locked forever. Furthermore, all accesses, regardless of privilege mode, will be checked.

This locking scheme might be useful for secure bootstrapping code which loads OS code into memory and then locks down the region containing the OS, to prevent malware from subsequently corrupting the OS code.

The RISC-V spec suggests that the CSRs that describe the memory regions might get swapped in and out during context switches.

For example, one process (a device driver) might be allowed to access a given memory region while another process (a user thread) ought to be forbidden from accessing the same region. As another example, imagine a hypervisor running in Machine Mode that is hosting two distinct operating systems, which are running in Supervisor Mode. Each operating system will be given a range of available main memory to use. To ensure that each OS stays out of the memory region used by the other OS, the Machine Mode software will use the PMP unit to enforce all accesses. When timeslicing between two OSes, the hypervisor code running in Machine Mode will need to swap one set of PMP registers with the other set.

To describe each memory region, we must somehow specify a starting address and a length. Each memory region has certain protection attributes associated with it and these must also be given.

Recall that there can be up to 16 different memory regions. There is essentially a table with 16 entries, which can describe up to 16 different memory regions.

The information about each region (i.e., each table entry) is encoded into a single full-length "address register" (e.g., 32 bits) and an additional "configuration byte" (8 bits). The length of the region is cleverly encoded within these bits, as described next.

The names of the address registers are as follows. Each is a full CSR, e.g., 32 bits for RV32 machines, and longer for RV64.

pmpaddr0
pmpaddr1
…
pmpaddr15

The names of the configuration bytes are given next. Each is an 8 bit quantity.

pmp0cfg
pmp1cfg
…
pmp15cfg

The configuration bytes are packed into 4 registers, with four configuration bytes per register. (…at least for the RV32 machines. For 64-bit machines, only 2 CSRs are needed, since 8 configuration bytes can be packed into each register.)

Each region has a configuration byte, wherein the 8 bits are used as follows:

Number of bits	**Meaning**
1	Region is readable
1	Region is writable
1	Region is executable
1	Region is locked
2	"A" field
2	*unused*

The "A" field has these possible values:

Value	Meaning
00	This entry is not used
01	Top-of-range address encoding
10	Region is 4 bytes long
11	Region is > 4 bytes long

Not all of the 16 regions need be populated; the 00 value of the "A" field indicates an unused entry in the region table.

Each region must have a length that is a power of 2. Further more, the length must be 4 or larger; that is: 4, 8, 16, 32, 64, … The region must be naturally aligned, given its length. For example, a region of size 1,024 must start on an address that is evenly divisible by 1,024.

If the region has size 4, the "A" field will be 01 and the address field will contain the starting address of the region.

If the region has a larger size (such as 8, 16, 32, 64, …), the "A" field will be 11 and the address register will contain both the starting address and length. For example, consider a region with size 1,024. How is this encoded in the address register? The region must be naturally aligned so the starting address will look like this, in binary:

```
XXXXXXXXXXXXXXXXXXXXXX0000000000
```

The last 10 bits of the starting address will always be 0. The length of the region (1,024 in this example) will be encoded in these zero bits, as follows:

```
XXXXXXXXXXXXXXXXXXXXXX0111111111
```

The hardware, by looking at the value in the address register and by locating the rightmost 0 bit, can directly infer the size of the region.

Okay, it's more complex that this. For 32 bit machines, the region table can actually be used to describe regions anywhere in a 34 bit address space. The extra 2 bits of address are achieved as follows.

For a region of size 4 bytes, the last 2 bits must be 0, so they are not represented. All 32 bits in the address register are used for bits [33:2] of the address, with bits [1:0] implicitly being 00.

For a region of size 8 bytes, the address register will contain:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx0
```

Here there are 31 significant bits. Adding in the implicit bits 000 (since the region is 8-bytes in size), gives 34 bits of starting address.

For a region of size 16 bytes, the address register will contain:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx01
```

Here there are 30 significant bits. Adding in the implicit bits 0000 (since the region is 16-bytes in size), gives 34 bits of starting address.

And so on for larger regions. For RV64, this same shifting by 2 bits also occurs. All physical addresses are limited to 56 bits, so only the lower order 54 bits in the address registers are used.

There is a final way of describing a region's starting point and size. If the "A" field is 01 ("top-of-range") then the address register is interpreted differently. This encoding scheme can be used when the regions are contiguous and each region follows directly after the previous region. In this encoding, the address register does not specify the start of the region, but the ending address of the region. (Actually, it is one past the last address.) With this code for the "A" field, the starting address is given by the previous address register (or address 0 for the first table entry).

When the "top-of-range" encoding is used, the shifting by 2 bits still occurs. Thus, all physical addresses are 34 bits and each region must begin on a 4 byte aligned address.

The reason that 34-bit and 56-bit physical addresses are specified is that the paging systems use these sizes. In particular, the Sv32 paging scheme uses 34-bit physical addresses and the Sv39 and Sv48 paging schemes use 56-bit physical addresses.

Whenever an access to physical memory is attempted, the processor will first consult the protection table. The lowest numbered entry that matches the address

will be used to determine if the access is allowed. If the access is not allowed, then a synchronous exception will be raised (i.e., "load access exception", "store access exception", or "instruction access exception"). See the spec for complete details on which entry is selected.

Virtual Memory

Virtual Memory (i.e., page tables and virtual-to-physical address translation) is handled in Supervisor Mode.

There are several virtual memory schemes described in the RISC-V spec, called "bare", "Sv32", "Sv39", and "Sv48". The "bare" option means no virtual address translation occurs.

A 32 bit machine (RV32) may support a 2 level page table (but not 3 levels or 4 levels). A 64 bit machine (RV64) may support a 3 level table or a 4 level table (but not a 2 level table).

	Page Table Depth	Virtual Address Space	Physical Address Space
RV32:			
Bare			
Sv32	2 levels	32 bits, 4 GiBytes	34 bits, 16 GiBytes
RV64:			
Bare			
Sv39	3 levels	39 bits, 512 GiBytes	56 bits, 64 PiBytes
Sv48	4 levels	48 bits, 256 TiBytes	56 bits, 64 PiBytes
Sv57	*…to be defined later…*		
Sv64	*…to be defined later…*		

The size of the physical addresses is implementation defined. The numbers given above are the maximums, but in a particular implementation the physical address space will often be smaller.

Regardless of which virtual memory scheme is used, the page size is 4,096 (4 KiBytes). Thus, byte offsets into a page require 12 bits, since 2^{12} = 4,096. All pages must be aligned on a page boundary, so the lower 12 bits of any page address are always 000000000000.

What is a Page Table? A page table is a tree where every node is a page. These pages are each kept in main memory and addressed with physical addresses.

The pages at the leafs are called "data pages" and contain the actual bytes in the virtual address space.

The interior nodes constitute the page table itself and allow the data pages to be located. The root page plus the interior nodes constitute the page table and are not visible to the user process. The data pages constitute the virtual address space (or at least the part of it that is populated) and contain the data bytes that the user process will access.

Each interior page contains an array of "Page Table Entries" (PTEs). Each PTE points to a child node. Some PTEs are consider "leaf PTEs" and point directly to a data page. Other PTEs point to lower levels in the page table tree.

In the Sv32 addressing scheme, the page table is only 2 levels deep. Page Table Entries are 4 bytes long. Therefore, we have:

1 root page, containing 1,024 PTEs
1,024 interior pages, with 1,024 PTEs each
1,024 × 1,024 data pages, of 4,096 bytes each

Thus, a virtual address space contains 4 GiBytes.

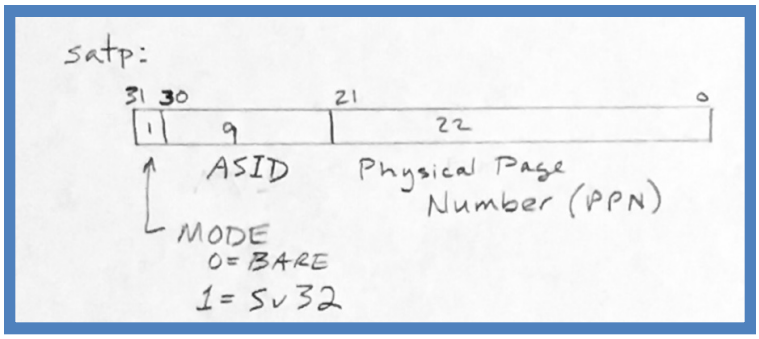
The term "page table" is often used imprecisely to mean either a single 4,096 byte interior node in the tree, or the entire collection of interior pages making up the whole tree, not including the data pages.

The **satp** (Supervisor Address and Translation Register) is the CSR that controls address translation.

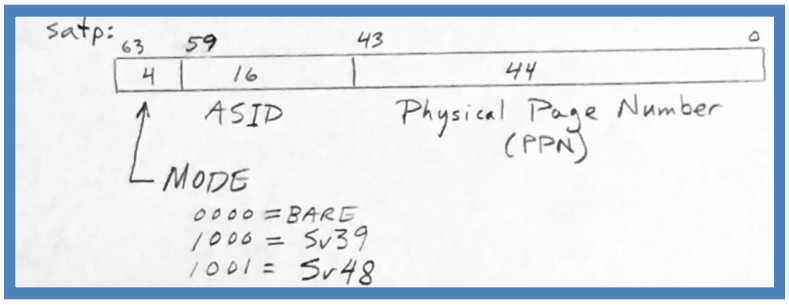
The **satp** register contains three fields:

MODE	Is address translation turned on or not?
ASID	Address Space Identifier
PPN	Physical Page Number, i.e., address of page table's root page

For RV32 systems, the **satp** register is 32 bits, arranged as follows:



For RV64 systems, the **satp** register is 64 bits, arranged as follows:



The MODE field indicates whether virtual memory address translation is turned on or not and, if so, which paging table scheme is being used.

For RV32 systems, the MODE field can have these values:

0	address translation turned off (i.e., “bare” mode)
1	Sv32 address translation is turned on

For RV64 systems, the mode field can have these values:

0000	address translation turned off (i.e., “bare” mode)
1000	Sv39 address translation is turned on
1001	Sv48 address translation is turned on
other	unused / reserved for Sv57, Sv64

The “bare” mode indicates that there is no virtual-to-physical translation. All addresses are physical. Regardless of mode, the enforcement of physical memory attributes (i.e., using the “pmp…” registers, described elsewhere) is still operative.

The Physical Page Number (PPN) field contains the physical address of the root page of the page table tree.

Since all pages must be aligned on a 4,096 byte boundary, the last 12 bits of the address of any page must be 0. For RV32, the Physical Page Number (PPN) field is 22 bits wide, which is sufficient to address any page in a 34 bit physical address space. For RV64, the PPN field is 44 bits wide, which is sufficient to address any page in a 56 bit physical address space.

Each user process will presumably run in a distinct address space. Each address space is given an Address Space Identifier (ASID). For RV32 machines, the ASID is 9 bits (allowing for 2^9 = 512 different address spaces). For RV64 machines, the ASID is 16 bits (allowing for 2^{16} = 65,536 different address spaces).

The **satp** register contains the ASID of the currently executing process.

Overview of TLBs: To make address translation fast enough for virtual memory to be feasible, page table entries must be cached in an "address translation cache". Such a cache is traditionally called a "**Translation Lookaside Buffer**", and so is abbreviated as "TLB".

The TLB will contain a small number of page table entries. When a FETCH, LOAD or STORE to memory occurs, the address translation hardware will first check the address translation cache (TLB). If the TLB contains a matching entry, the address translation hardware will use it to generate a physical address immediately, which is much faster because we avoid going to main memory to read the page table tree to locate the data page.

The TLB which is organized as an associative memory, keyed on virtual page number. If a TLB entry is present, then the entry will supply the physical page number. If the entry is absent, then the address translation process must go to memory to fetch the appropriate entry from the page table tree. Some TLB entry will be evicted and the new entry will be placed in the TLB. The address translation will then proceed.

Updating the TLB, which includes walking the in-memory page table tree to find the appropriate entry and writing the evicted entry back to memory, is a complex and time-consuming task. For faster performance, TLB updating and page-table walking are performed in hardware on many systems. However, in simpler

implementations, walking the page table and updating the TLB are performed by software.

The way software updating works is as follows: When there is no valid entry in the TLB during an address translation, an exception will be signaled. The hardware never attempts to walk the in-memory page tables. The exception is handled by a trap handler (which may be running in Machine Mode). The trap handler will essentially emulate the missing hardware in software and will walk the in-memory page table and update of the TLB directly. How this occurs is a software issue, not covered in the RISC-V spec.

From time to time context switches will occur and the currently executing process will change. Since each process can occupy a different address space, the entries in the TLB may not be appropriate for the newly executing process. The purpose of the Address Space ID is to make sure that a process only uses page table entries that apply to it. Each entry in the TLB will be tagged with an ASID; only entries that match the currently executing process's ASID (as stored in the **satp** register) will be used.

The spec does not require ASIDs to be implemented and, if they are implemented, the full range of values need not be supported.

The spec mentions that writing to the **satp** register may require careful concurrency control. It could be that address translations using older page tables operate concurrently and an update to **satp** may require the SFENCE.VMA instruction in order to ensure correctness of code.

The idea is that, as long as there are not too many distinct address spaces and the ASID field is large enough, then the ASID mechanism will prevent one process from using page table entries from the wrong address space. But with a larger number of address spaces, it may be necessary to flush the address-translation cache, i.e., to flush the TLB. We may also need to flush the TLB when changes are made to a running process's address space, i.e., to a page table that is currently in use. For example, if we remove a data page from the address space, it is not sufficient to simply invalidate the relevant Page Table Entry in the page table node stored in memory. We also need to ensure that there are no entries still sitting in the TLB.

Comment: Note that by storing the Mode, ASID, and Physical Page Number of the root of the page table in a single register, it allows all three to be updated atomically with a single CSR instruction.

SFENCE.VMA: Supervisor Fence for Virtual Memory

General Form:

```
SFENCE.VMA   vaddr,asid
```

Example:

```
SFENCE.VMA   x5,x8
```

Comment:

This instruction is used to impose order on accesses to memory and updates to page tables. In particular, it is designed to flush the address translation cache (TLB).

Whenever virtual memory is turned on and a FETCH, LOAD or STORE to memory is made, address translation must occur so there will also be implicit READs and WRITEs to the page tables stored in memory. Normally, the access to the page table in memory can be avoided and the access will be made to the address translation (TLB) cache instead. But whenever changes to the page tables in memory are made, we need to flush obsolete entries in the TLB. This is the purpose of this instruction.

When the OS updates a page table node, it will issue a WRITE to memory. In order to make sure that a WRITE to the page table is seen before all subsequent memory accesses, this instruction must be executed. It ensures that all WRITEs to memory that occur earlier in the instruction stream will be completed and visible before any implicit READs or WRITES to the page table triggered by subsequent operations in the instruction stream. Since the TLB is caching page table entries, this instruction must also invalidate affected entries, forcing the memory management unit to go back to memory to retrieve current page table entries.

[It would seem necessary to also force all earlier address translations to complete before the WRITE to the page tables occurs, but the spec does not mention this ordering constraint. ???]

To impose a finer level of granularity, this instruction can specify a specific virtual address and/or a specific address space. Reg1 contains a virtual address; Reg2 contains an Address Space ID.

If the ASID is specified as x0, then entries for all address spaces are affected; otherwise, only entries for the given address space are affected.

If the virtual address is specified as x0, then all levels of the page table are affected; otherwise, only the relevant leaf PTE in the page table is affected.

If both the ASID and the virtual address are given as x0, then it causes a complete flush of the TLB. Simple implementations are free to always ignore the ASID and virtual address space and perform a global TLB flush whenever this instruction is executed.

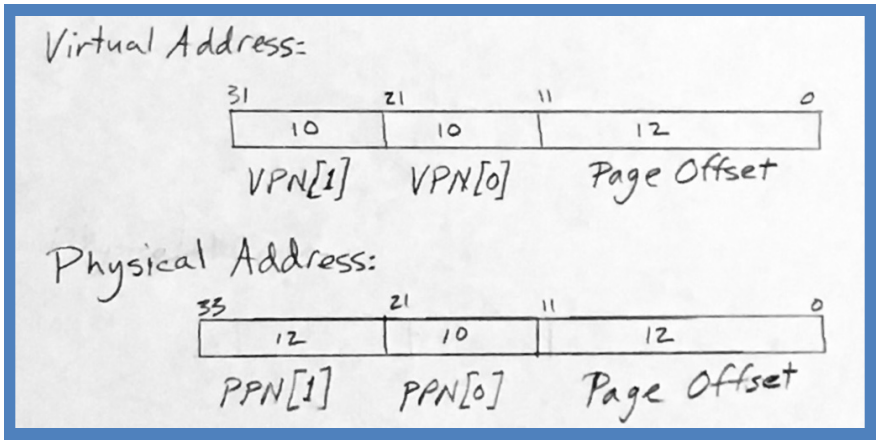
It is possible that several cores are sharing memory and that two threads running on different cores are sharing a single address space and a single page table data structure in memory. Ideally, any update to this page table should be seen by all threads on all cores. However, this instruction only affects a single core. To synchronize with other threads on other cores, the OS must use other instructions.

Encoding:

This is a variant of an R-type instruction, where the RegD field is unused.

Sv32 (Two-Level Page Tables)

In the Sv32 address translation mode, a virtual address is 32 bits. This is partitioned into a 20-bit virtual page number (VPN) and a 12 bit offset. The 20-bit Virtual Page Number is translated by the memory management unit into a 22-bit Physical Page Number. The Physical Page Number is then concatenated with the offset to give the 34-bit physical address.



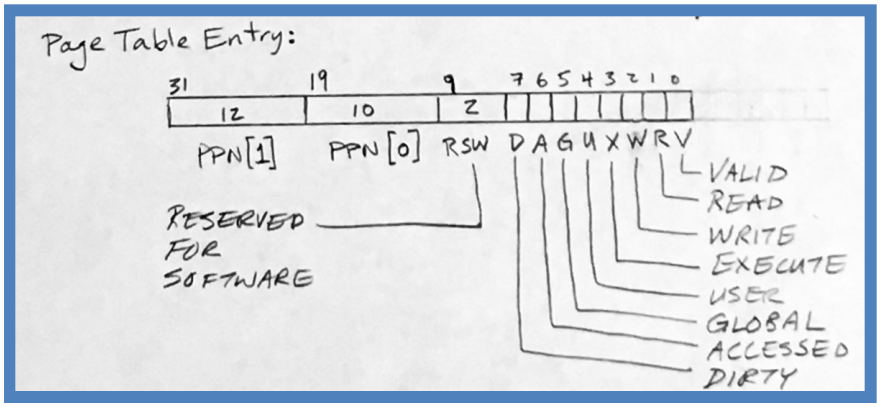
The size of each page is 4,096 bytes, and each byte within a page can be addressed by 12 bits, since 2^{12} = 4,096. All pages must be properly aligned on a 4,096 byte boundary.

Each Page Table Entry (PTE) is 4 bytes. Thus, each page can hold 1,024 entries. Within a page, each entry can be addressed by a 10-bit address, since 2^{10} = 1,024.

The page table is organized as a 2-level tree, where all nodes are interior (i.e., non-data) pages. The root page is at the top of the tree and contains 1,024 page table entries. Each entry contains a pointer to a second level page. The second level pages contain page table entries which point to the data pages. The second level pages are referred to as "leaf" pages; the data pages can be considered to lie below the leaf pages.

A Page Table Entry (PTE) is 4 bytes and contains the following fields:

Field Size	Meaning
22	Physical Page Number
2	RSW (Undefined; Reserved for Software)
1	D (Dirty)
1	A (Accessed)
1	G (Global mapping)
1	U (User accessible)
1	V (Valid)
3	XWR (Executable, Writable, Readable)

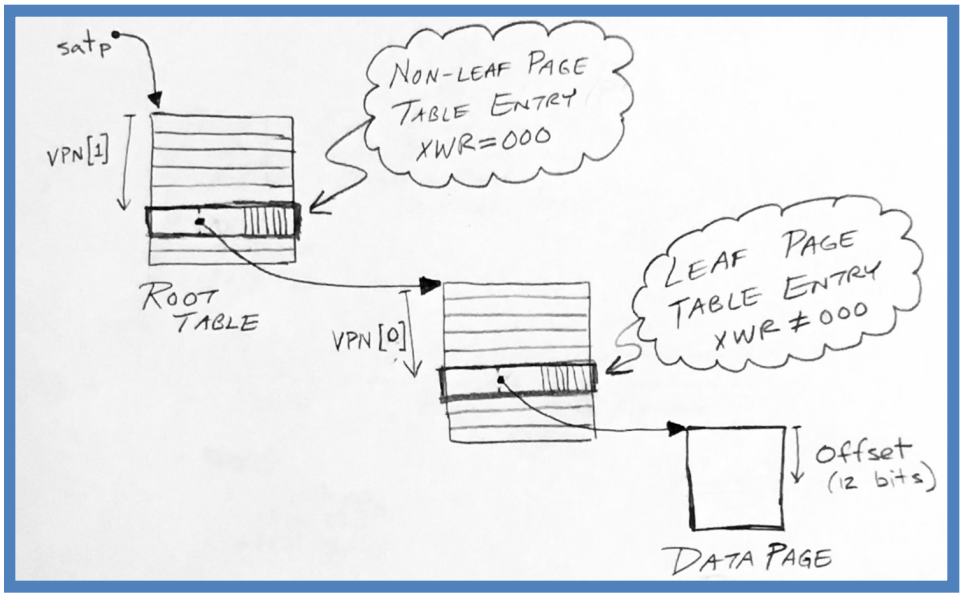


The Virtual Page Number, which is 20 bits, is divided into two fields of 10 bits each, called VPN[1] and VPN[0].

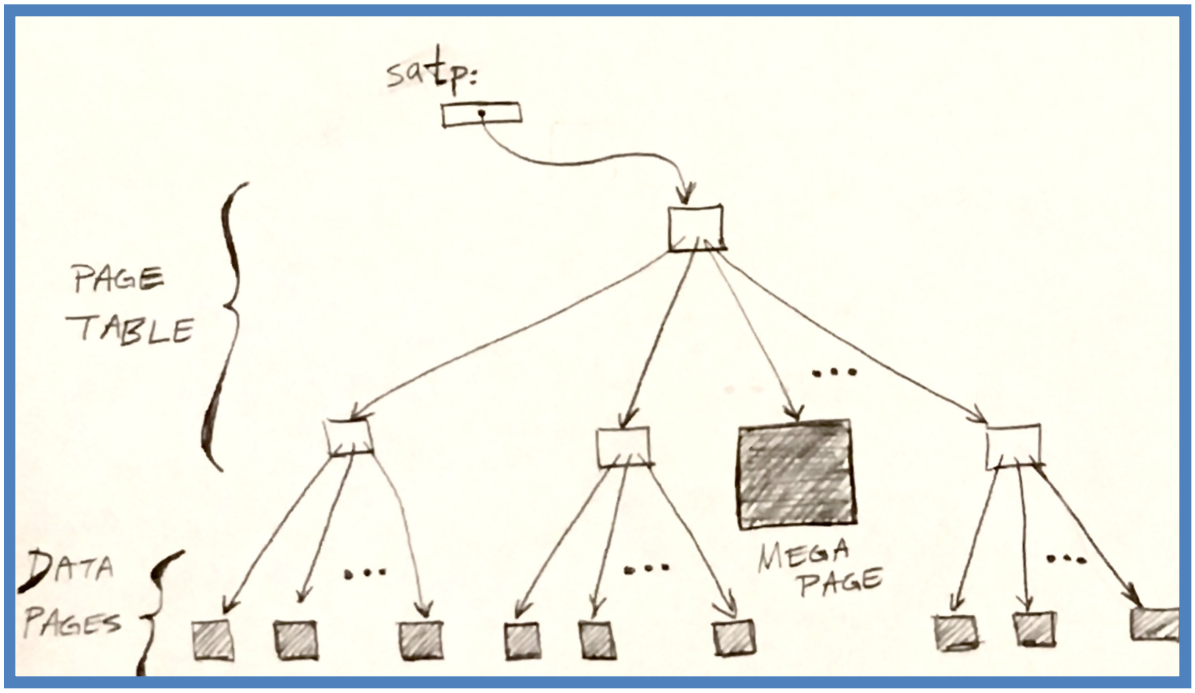
The uppermost 10 bits (VPN[1]) is used to select an entry in the root page. This entry contains a Physical Page Number (PPN) field, which addresses a second level page.

The second 10 bits of the Virtual Page Number (VPN[0]) is used to select an entry in the second level page. This yields the final page table entry (called a "leaf" page table entry), which contains the Physical Page Number of the page containing the actual data.

This diagram shows this.



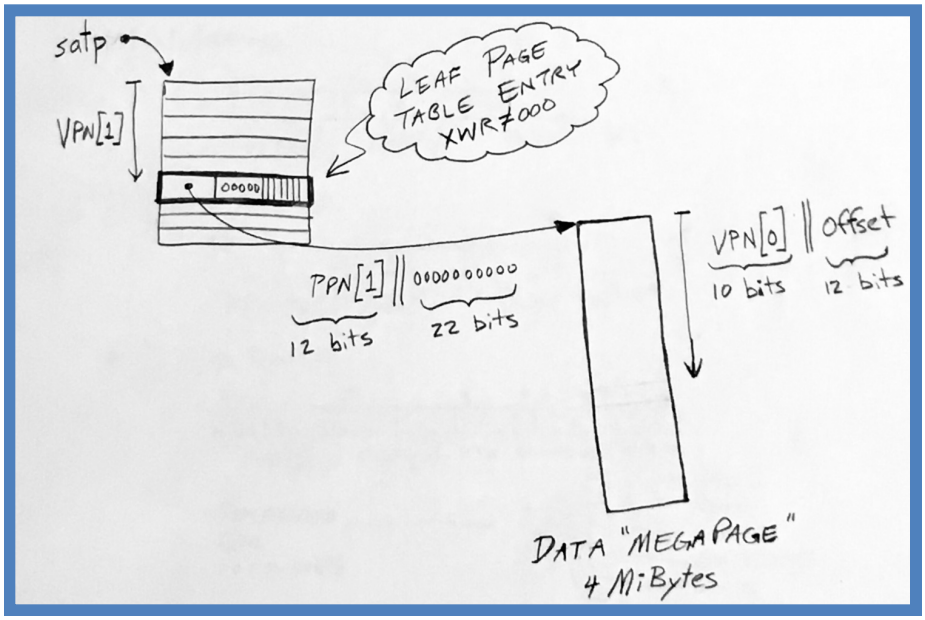
There is a single root page and up to 1,024 second level pages. Each page in the page table (that is, the root page and all second level pages) contains 1,024 Page Table Entries. The entries at the bottom of the tree are call "leaf" entries and point to the data pages. Leaf entries are distinguished from non-leaf entries by the XWR (Execute-Write-Read) permission bits. If XWR=000, then the entry is not a leaf entry.



There is a facility for “megapages”. A megapage is a data page containing 4 MiBytes. An offset into a megapage will be 22 bits, since $2^{22} = 4 \times 1{,}024 \times 1{,}024 = 4{,}194{,}304$. A megapage must be aligned to a 4 MiByte boundary.

The XWR bits in a page table entry indicate whether that entry is a leaf entry (meaning that the entry points to a data page) or a non-leaf entry (meaning that the entry points to another page in the page table tree). If the XWR bits are 000, then the entry is a non-leaf entry, pointing to the next level in the page table tree. If the XWR bits are not 000, then they indicate that the entry points to a data page and they give the “execute”, “read”, and “write” permissions for the data page.

In the case of megapages, there is no second level page in the tree. Instead, the Page Table Entry in the root page points directly to a data page, i.e., a megapage.



There are a couple of benefits of placing megapages in a virtual address space that may also contain normal 4,096 byte data pages.

First, whenever data from a page is first referenced, the Memory Management Unit must walk the page table to find the appropriate leaf page table entry. With megapages, only the root page table must be consulted, since there is no second level page involved. This reduces the number of memory accesses by one. (This could mean substantial savings if the PTE needs to be re-loaded frequently, due to TLB contention and/or needs to be updated in memory upon TLB flushing).

The second advantage is that a single entry in the address translation cache (the TLB) will cover 4 MiBytes, instead of 4 KiBytes. This means that fewer entries in the TLB will be needed for each address space. The TLB is a significant bottleneck and reducing contention is critical.

[For example, a simple address space may have one block of memory marked “read/execute” (for the program and constants) and a second block marked “read/write” (for static variables and the stack). Since neither of these will normally exceed 4 MiBytes, only 2 entries in the TLB would be needed for the entire address space. TLB entries are precious and few in number, so this reduces pressure on the TLB.]

However megapages come with drawbacks. First, they will often be larger than necessary. (This is called the “internal fragmentation problem”.) The extra space in the megapages represents real physical memory and, if not needed by the process, this memory is effectively wasted. For example, if an address space only requires 64

KiBytes, but is allocated two megapages, then approximately 99% of the memory is wasted, since $2^{22} - 2^{16} \approx 2^{22}$. If concurrent processes are doing lots of complex data sharing by sharing virtual memory pages this fragmentation problem can become worse.

Second, the OS kernel must be able to allocate large blocks of contiguous memory. This is not a problem if the memory is divided into a collection of equal sized pages. But if there are multiple page sizes (e.g., 1,024 and 4,194,304) then things get trickier. After all, this is essentially the problem that virtual memory was designed to address. (This is called the "external fragmentation problem".)

Next, we look at the meaning of the bits in the Page Table Entry (PTE).

The **Valid bit** (V) indicates whether this PTE is valid or not (1=valid, 0=invalid). If an invalid PTE is encountered by the Memory Management Unit, then an access fault is signaled. (The exception will be either a "instruction access exception", "read access exception", or "store access exception" as appropriate). If the PTE is invalid, the remaining bits are ignored and can be used by software as desired.

The **User Accessible bit** (U) controls whether the data can be accessed in Supervisor Mode or User Mode. Typically, a process's address space will be divided into two parts, sometimes called the "bottom half" and the "top half". The bottom half can only be access by code running in User Mode; data in the top half can only be accessed when running in Supervisor Mode. U=1 means bottom half; U=0 mean top half. If User Mode code attempts to access data in the top half, an access fault will occur. If Supervisor Mode code attempts to access data in the bottom half, an access fault will occur.

However, the SUM bit in the Supervisor Status word (**sstatus**) can be used to allow Supervisor Mode code to access pages in the bottom half. Normally, Supervisor code will run with SUM=0, in which case an access fault will occur. But occasionally (e.g., when data is passed to the OS in a kernel call) Supervisor Code will set SUM=1 for a short time. During this time, LOAD and STORE instructions will operate as if running in User Mode. This allows the kernel routines to retrieve or store parameters in the bottom half (i.e., in the user portion of the virtual address space).

The U bit is only meaningful for leaf PTEs; for non-leaf PTEs, this bit will be 0.

The **Global Mapping bit** (G) concerns pages that are shared among all address spaces. G=1 means globally shared by all address spaces; G=0 means local to a single

address space and not shared. If a page is globally shared, it means that it is mapped into all address spaces. In other words, the Address Space ID (ASID) checking is suppressed. For example, a set of commonly used, read-only functions and data can be mapped into all address spaces, making these routines available at essentially no cost to all address spaces. These could be user-level functions present in all "bottom halves" or could be Supervisor Mode code that is placed into all "top halves".

The spec indicates that this bit is meaningful for both leaf and non-leaf PTEs. When set in a non-leaf PTE, it means the entire page table tree below is global and shared by all address spaces.

[??? It is not clear why the G bit is specified as being meaningful in non-leaf PTEs. If a leaf PTE that is marked "global" is stored in the TLB, then that TLB entry (and the data page it refers to) is shared. But if there is no leaf PTE in the TLB, then the Memory Management Unit will need to walk the page table tree to locate the leaf PTE. During this walk, it would not seem to make any difference whether some page table page is shared or not; each page of the page table tree still has to be accessed. Perhaps the RISC-V spec envisions putting non-leaf PTEs in the TLB. ???]

The **Accessed bit** (A) is set by the Memory Management Unit whenever a byte on the data page is read, written, or fetched for execution. In some textbooks this bit is called the "referenced" bit. This bit is only meaningful for leaf PTEs; for non-leaf PTEs, this bit will be 0.

The **Dirty bit** (D) is set by the Memory Management Unit whenever a byte on the data page is written to. This bit is only meaningful for leaf PTEs; for non-leaf PTEs, this bit will be 0.

Details on Updating a Page Table Entry: We just said that the Accessed Bit and the Dirty Bit are set by the hardware whenever any byte in the data page is accessed or updated. (These are the only bits in the Page Table Entry (PTE) that would ever be modified by hardware, but any modification will require that this PTE, when evicted from the TLB in the future, must be written back to the in-memory page table.)

However the RISC-V spec also allows the hardware to work a second way. In this alternative approach, a page table entry is never modified by the hardware. This simplifies the hardware since PTEs need never be written back to memory. Instead, the "A" and "D" bits are merely checked. If they are not already set

correctly, an exception is signaled. Then software in the trap handler can take actions to write the updated page table entry back to memory.

In some cases, the "A" and "D" bits may not be needed. For example, if some data page is always resident in memory and never swapped out to backing store, then the "A" and "D" bits can be ignored. Or also, if a memory-mapped I/O device is mapped into some virtual address space, then the "A" and "D" bits can be ignored. In such cases, the bits can be preset to "1" to obviate the need for updating.

The **Executable** (X), **Writable** (W), and **Readable** (R) bits indicate whether this PTE is a leaf or non-leaf entry and, for leaf entries, they indicate the access permissions for the data on the data page. If XWR = 000, then this PTE is not a leaf entry; the Physical Page Number in the entry points to the next lower-level page in the page table tree.

The **RSW** bits are not used by the hardware and are freely available to software to be used as desired.

The Sv32 Page Table Algorithm

Whenever an access (read, write, or fetch-for-execution) occurs, the following steps are taken by the Memory Management Unit, to map a virtual address into a physical address.

(The RISC-V spec gives this algorithm in a general form applicable to Sv32, Sv39, and Sv48, i.e., two-, three-, and four-level page tables. In order to make it easier to understand, the version given here simplifies and specializes the algorithm to two-level tables.)

Inputs:

va – the virtual address to be translated (32 bits), with the following parts:
- va.VPN[1] – the uppermost 10 bits; offset into the top level page table
- va.VPN[0] – the following 10 bits; offset into the second level page table
- va.OFFSET – the lower 12 bits; offset into the data page

The desired access type (READ, WRITE, or FETCH)

The current privilege mode (U or S; no address translation is done in M mode)

satp – the Supervisor Address Translation and Protection CSR
- satp.PPN – the page number of the root page.

status.SUM – the "Permit Supervisor User Memory Access" bit in the status reg
status.MXR – the "Make Executable Readable" bit in the status reg

Outputs:

pa – physical address (34 bits)
or
access exception – signal a page fault exception and abort translation

Temp variables:

a – pointer to (i.e., physical address of) a page in memory
p – pointer to (i.e., physical address of) a PTE in memory
pte – a Page Table Entry, as fetched from memory

Algorithm:

Compute "a", the address of root top-level page in the page table
a ← satp.PPN || 000000000000
Compute "p", the address of the PTE in the top level page
p ← a + (va.VPN[1] || 00)
Read from memory "pte", a Page Table Entry from the root-level page
pte ← memory [p]
If the Physical Memory Protection (PMP) system indicates problems
then signal an exception
If this PTE is not valid (i.e., pte.V = 0 or pte.XWR is an invalid value)
then signal an exception
If pte.XWR ≠ 000 then…
/* This PTE is a leaf PTE, pointing to a megapage. */
If pte.PPN[0] ≠ 0000000000, then signal an exception
Compute the physical address
pa ← pte.PPN[1] || va.VPN[0] || va.OFFSET
Else (i.e., pte.XWR = 000)…
/* We do not have a leaf PTE, so look at the second level. */
Compute "a", the address of the second level page table page
a ← pte.PPN[1] || pte.PPN[0] || 000000000000
Compute "p", the address of the PTE in the second level page
p ← a + (va.VPN[0] || 00)
Read from memory "pte", a Page Table Entry from the second level page
pte ← memory [p]
If the Physical Memory Protection (PMP) system indicates problems
then signal an exception
If this PTE is not valid (i.e., pte.V = 0 or pte.XWR is an invalid value)

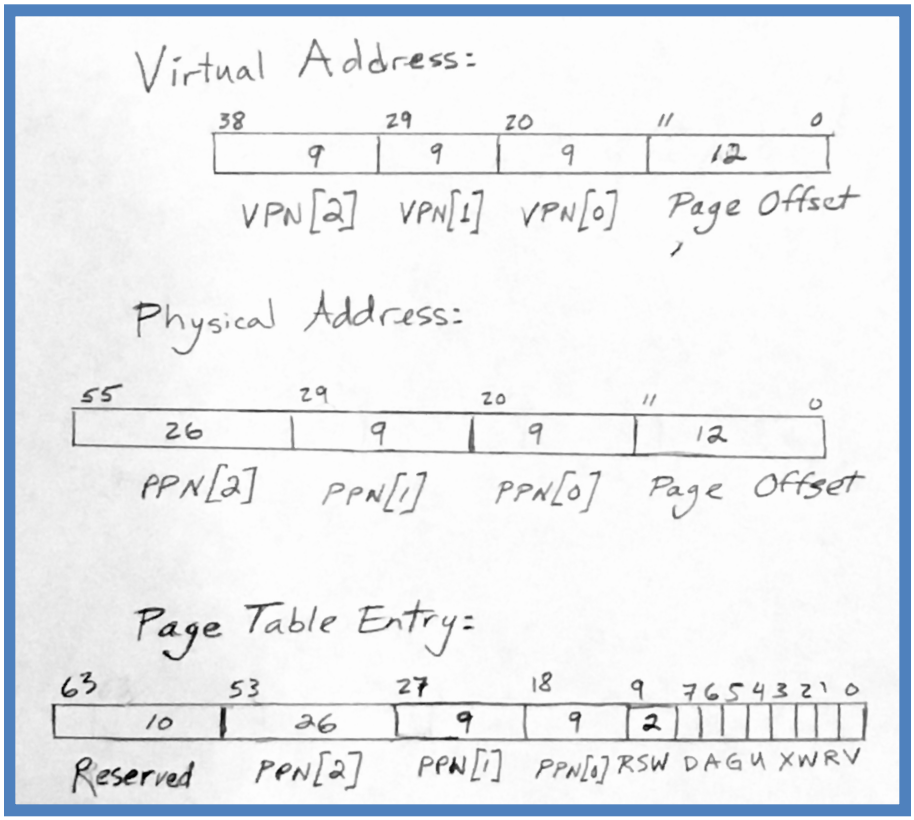
```
        then signal an exception
    If pte.XWR = 000, then we do not have a leaf PTE…
        then signal an exception
    Compute the physical address
        pa ← pte.PPN[1] || pte.PPN[0] || va.OFFSET
/* "pte" contains the leaf Page Table Entry. */
Make sure this access is allowed by checking the X, W, R, and U bits….
    If pte.R ≠ 1, then signal an exception
    If pte.W ≠ 1 and this is a WRITE, then signal an exception
    If pte.X ≠ 1 and this is a FETCH, then signal an exception
    If the current mode is "User"…
        If pte.U = 0, then signal an exception
    If the current mode is "Supervisor"…
        If this is a READ & pte.R ≠ 1 & status.MXR ≠ 1, then signal an exception
        If pte.U = 1 and status.SUM = 0, then signal an exception
Check the "accessed" bit…
    If pte.A ≠ 1, then set it
Check the "dirty" bit…
    If this is a WRITE and pte.D ≠ 1, then set it
If the pte.A or pte.D bits were changed…
    Either
        • Signal an exception (and let software update the page table)
        • Write the Page Table Entry back to memory.
          This must be atomic with respect to the earlier reading of the pte.
          If the PMP system indicates problems, then signal an exception
```

Sv39 (Three-Level Page Tables)

Both the Sv39 and Sv48 virtual addressing schemes are natural extensions of the Sv32 scheme. The primary difference is that they support larger virtual address spaces and page tables that are 3-levels (for Sv39) and 4-levels (for Sv48), instead of 2-levels.

If you understand Sv32 then – for all intents and purposes – you already understand Sv39 and Sv48.

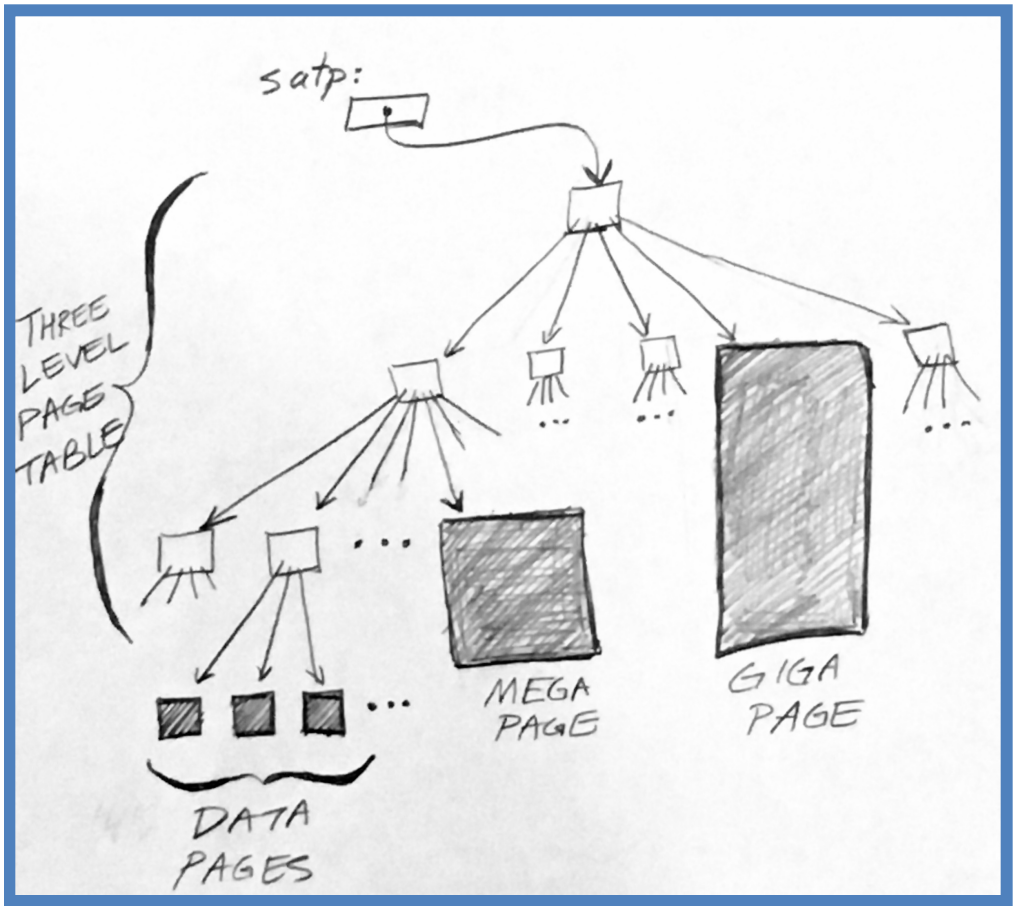
With Sv39, virtual addresses are 39 bits and physical addresses are 56 bits.



We can summarize the Sv39 virtual addressing scheme as follows: Sv39 is identical to Sv32, with these exceptions:

- It may only be implemented on RV64 systems, not on RV32 systems.
- Virtual addresses are 39 bits.
- The maximum virtual address space is 2^{39} = 512 GiBytes.
- The physical addresses are 56 bits.
- The page size is unchanged (i.e., 4,096 bytes).
- The page tables have 3 levels, instead of 2 levels.
- The Page Table Entries are 64 bits, instead of 32 bits.
- Each page of the page table contains 512 PTEs, instead of 1,204 PTEs.
- Offsets into the page tables are 9 bits (instead of 10 bits) since 2^9 = 512.
- The RSW, D, A, G, U, X, W, R, and V bits work identically.
- Megapages are 2 MiBytes, instead of 4 MiBytes.
- “Gigapages” of 1 GiByte are also supported.

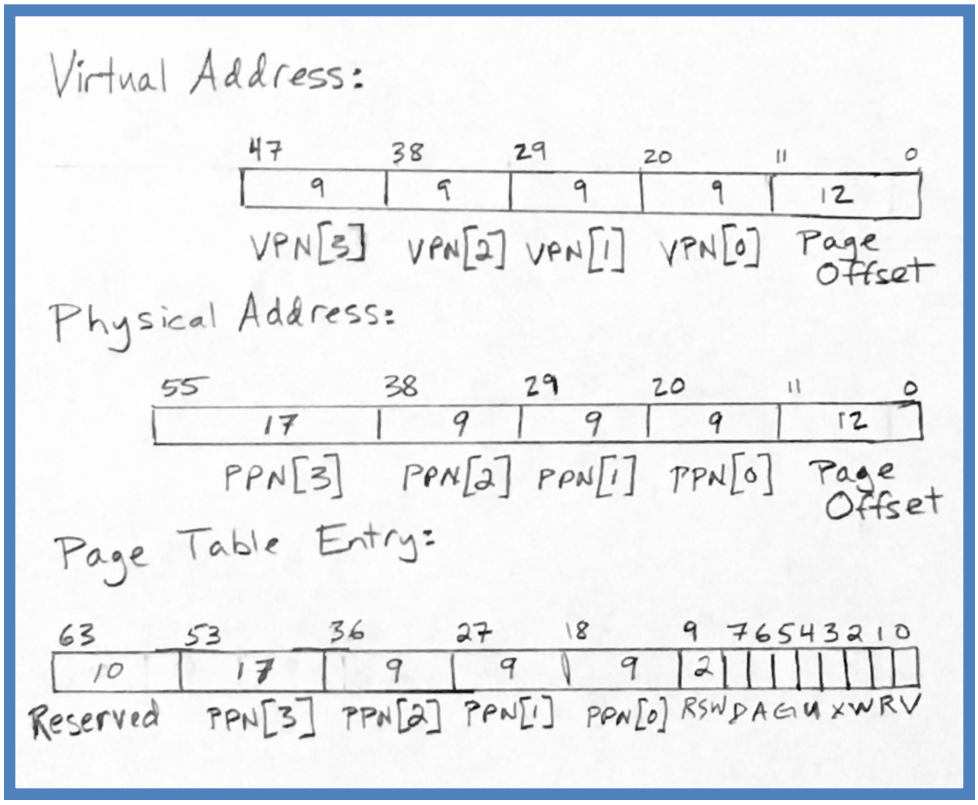
With Sv39, the page table will have three levels:



On a 64-bit machine, addresses are initially 64 bits long, since this is the register length. For Sv39, only the least significant 39 bits are used to address into the virtual address space. The remaining 25 bits must be the sign-extension (not zero-filled) of the least significant bits. Otherwise, an exception will be signaled.

Sv48 (Four-Level Page Tables)

With Sv48, virtual addresses are 48 bits and physical addresses are 56 bits.



We can summarize the Sv48 virtual addressing scheme as follows: Sv48 is identical to Sv32 and Sv39, with these exceptions:

- It may only be implemented on RV64 systems, not on RV32 systems.
- If Sv48 is supported, then Sv39 must also be supported.
- Virtual addresses are 48 bits.
- The maximum virtual address space is 2^{48} = 256 TiBytes.
- The physical addresses are 56 bits.
- The page size is unchanged (i.e., 4,096 bytes).
- The page tables have 4 levels.
- The Page Table Entries are 64 bits, the same as Sv39.
- Each page of the page table contains 512 PTEs, the same as Sv39.
- Offsets into the page tables are 9 bits, the same as Sv39.
- The RSW, D, A, G, U, X, W, R, and V bits work identically.
- Megapages are 2 MiBytes, the same as Sv39.
- Gigapages are 1 GiByte, the same as Sv39.
- “Terapages” of 512 GiBytes are also supported.

On a 64-bit machine, addresses are initially 64 bits long, since this is the register length. For Sv48, only the least significant 48 bits are used to address into the virtual

address space. The remaining 16 bits must be the sign-extension (not zero-filled) of the least significant bits. Otherwise, an exception will be signaled.

Chapter 10: What is Not Covered

Besides the extensions we have already described, the RISC-V spec mentions several additional extensions which may or may not be implemented. The specifications for these extensions are in a state of flux, so we will simply mention their existence here.

The Decimal Floating Point Extension ("L")

Most programmers are familiar with binary floating point, but there is also such a thing as a representation of floating point numbers using a decimal base. The RISC-V spec contains nothing specific for this extension and only mentions it as future work.

The Bit Manipulation Extension ("B")

The RISC-V spec contains nothing specific for this extension and only mentions it as future work.

The Dynamically Translated Languages Extension ("J")

High level languages often require dynamic typing checking, garbage collection and dynamic (just-in-time) compiling, so including specialized instructions to support these can improve performance. However, the RISC-V spec contains nothing specific for this extension and only mentions it as future work.

The Transactional Memory Extension ("T")

The RISC-V spec contains nothing specific for this extension and only mentions it as future work.

The Packed SIMD Extension ("P")

Recall that SIMD stands for "single instruction, multiple data". The goal is to perform a large number of floating point operations simultaneously. There is data parallelism, but not full concurrency since the same operation is performed on a collection of values. A single operation (such as "multiply") is performed in parallel on N values, followed by the next operation after all N multiplications have completed.

The idea with this extension is that each floating point register will hold more than one value. For this extension, the floating point registers will likely be larger than 32 or 64 bits. For example, in one implementation each floating point register might be 1,024 bits wide. Thus each register can hold more than one value. Since 16 × 64 = 1,024, each register can hold 16 double precision values at once. When an arithmetic instruction (such as "add" or "multiply") is executed, the operation is performed on all 16 values simultaneously.

The status of this extension is unsettled. It may be dropped altogether in favor of the "V" Vector SIMD Extension.

The Vector SIMD Extension ("V")

The "V" Vector SIMD extension is designed to accommodate large vectors and SIMD operation. It includes a set of 32 vector registers (v0, v1, … v31) , a number of new configuration CSR registers, and additional instructions. The RISC-V specification is difficult to understand. Furthermore, this extension is subject to revision and is not yet "frozen".

Performance Monitoring

RISC-V defines a collection of CSR registers devoted to measuring hardware performance:

mhpmcounter3
mhpmcounter4
…
mhpmcounter31

mhpmevent3
mhpmevent4
…
mhpmevent31

This mechanism is not described in the spec, beyond saying that the "events" to be counted can be selected by setting **mhpmevent3**, **mhpmevent4**,… .

Debug/Trace Mode

The RISC-V spec mentions another mode called "Debug Mode" and several related Control and Status Registers (CSRs) named **tselect**, **tdata1**, **tdata2**, and **tdata3**, but the spec does not give any detail.

Acronym List

ABI	Application Binary Interface
AEE	Application Execution Environment
AMO	Atomic Memory Operation
ASID	Address Space ID
CSR	Control and Status Register
cycle	CSR[C00]: Clock cycle counter, User Mode
cycleh	CSR[C80]: Upper half of cycle, User Mode (RV32 only)
dcsr	CSR[7B0]: Debug control and status
dpc	CSR[7B1]: Debug PC
dscratch	CSR[7B2]: Scratch register
DZ	Divide By Zero (a bit within the Floating Point Flags, FFLAGS)
fcsr	CSR[003]: Floating Point Control and Status Reg (frm || fflags)
fflags	CSR[001]: Floating pointing flags
FFLAGS	Floating Point Flags (i.e., NX, UF, OF, DZ, NV)
frm	CSR[002]: Dynamic rounding mode
FRM	Floating Point Rounding Mode
FS	Field in status register; status of floating (clean/dirty/…)
HART	Hardware Thread
HBI	Hypervisor Binary Interface
HEE	Hypervisor Execution Environment
HEIP	hypervisor external interrupt
hpmcounter3	CSR[C03]: Event counter #3, User Mode
hpmcounter31	CSR[C1F]: Event Counter #31, User Mode
hpmcounter31h	CSR[C9F]: Upper half of counter, User Mode (RV32 only)
hpmcounter3h	CSR[C83]: Upper half of counter, User Mode (RV32 only)
hpmcounter4	CSR[C04]: Event Counter #4, User Mode
hpmcounter4h	CSR[C84]: Upper half of counter, User Mode (RV32 only)
HSIP	hypervisor software interrupt
HTIP	hypervisor timer interrupt
instret	CSR[C02]: Number of instructions retired, User Mode
instreth	CSR[C82]: Upper half of instret, User Mode (RV32 only)
IR	Instruction Register
ISA	Instruction Set Architecture
LR	Link Register
marchid	CSR[F12]: Architecture ID
mcause	CSR[342]: Trap cause code, Machine Mode
mcounteren	CSR[306]: Counter enable, Machine Mode

mcycle	CSR[B00]: Clock cycle counter, Machine Mode
mcycleh	CSR[B80]: Upper half of cycle, Machine Mode (RV32 only)
medeleg	CSR[302]: Exception delegation register, Machine Mode
MEIP	machine external interrupt
mepc	CSR[341]: Previous value of PC, Machine Mode
mhartid	CSR[F14]: Hardware thread ID
mhpmcounter3	CSR[B03]: Event counter #3, Machine Mode
mhpmcounter31	CSR[B1F]: Event Counter #31, Machine Mode
mhpmcounter31h	CSR[B9F]: Upper half of counter, Machine Mode (RV32 only)
mhpmcounter3h	CSR[B83]: Upper half of counter, Machine Mode (RV32 only)
mhpmcounter4	CSR[B04]: Event Counter #4, Machine Mode
mhpmcounter4h	CSR[B84]: Upper half of counter, Machine Mode (RV32 only)
mhpmevent3	CSR[323]: Event selector #3, Machine Mode
mhpmevent3	CSR[324]: Event selector #4, Machine Mode
mhpmevent31	CSR[33F]: Event selector #31, Machine Mode
mideleg	CSR[303]: Interrupt delegation register, Machine Mode
mie	CSR[304]: Interrupt-enable register, Machine Mode
MIE	Bit in status register; Machine Mode Interrupt Enable
mimpid	CSR[F13]: Implementation ID
minstret	CSR[B02]: Number of instructions retired, Machine Mode
minstreth	CSR[B82]: Upper half of instret, Machine Mode (RV32 only)
mip	CSR[344]: Interrupt pending, Machine Mode
misa	CSR[301]: ISA and extensions
MMU	Memory Management Unit
MPIE	Bit in status register; Machine Mode Previous Interrupt Enable
MPP	Field in status register; Machine Mode – Previous Privilege Mode
MPRV	Bit in status register; Modify Privilege
mscratch	CSR[340]: Temp register for use in handler, Machine Mode
MSIP	machine software interrupt
mstatus	CSR[300]: Status register, Machine Mode
MTIP	machine timer interrupt
mtval	CSR[343]: Bad address or bad instruction, Machine Mode
mtvec	CSR[305]: Trap handler base address, Machine Mode
mvendorid	CSR[F11]: Vendor ID
MXR	Bit in status register; Make Executable Readable
NMI	Non-maskable interrupt
NSE	Nonstandard extension
NV	Invalid Operation (a bit within the Floating Point Flags, FFLAGS)
NX	Inexact (a bit within the Floating Point Flags, FFLAGS)
OF	Overflow (a bit within the Floating Point Flags, FFLAGS)

PC	Program Counter
PMA	Physical Memory Attribute
PMP	Physical Memory Protection Unit
pmpaddr0	CSR[3B0]: PMP Address #0
pmpaddr1	CSR[3B1]: PMP Address #1
pmpaddr15	CSR[3BF]: PMP Address #15
pmpcfg0	CSR[3A0]: PMP Configuration word #0
pmpcfg1	CSR[3A1]: PMP Configuration word #1
pmpcfg2	CSR[3A2]: PMP Configuration word #2
pmpcfg3	CSR[3A3]: PMP Configuration word #3
PPN	Physical Page Number
PTE	Page Table Entry
RISC	Reduced Instruction Set Computer
RM	Rounding Mode Bits
ROM	Read Only Memory
RV128 / RV128I	Instructions present only on 128 bit machines
RV32 / RV32I	The basic instruction set, present on all machines
RV64 / RV64I	Instructions present only on 64 and 128 bit machines
satp	CSR[180]: Address translation and protection
satp	Supervisor Address Translation and Protection CSR
SBI	Supervisor Binary Interface
scause	CSR[142]: Trap cause code, Supervisor Mode
scounteren	CSR[106]: Counter enable, Supervisor Mode
SD	Field in status register, summarizes FS and XS fields
sedeleg	CSR[102]: Exception delegation register, Supervisor Mode
SEE	Supervisor Execution Environment
SEIP	supervisor external interrupt
sepc	CSR[141]: Previous value of PC, Supervisor Mode
sideleg	CSR[103]: Interrupt delegation register, Supervisor Mode
sie	CSR[104]: Interrupt-enable register, Supervisor Mode
SIE	Bit in status register; Supervisor Mode Interrupt Enable
SIMD	Single instruction, multiple data
sip	CSR[144]: Interrupt pending, Supervisor Mode
SP	Stack Pointer
SPP	Field in status register; Supervisor Mode – Prev Privilege Mode
SPIE	Bit in status register; Supervisor Mode Prev Interrupt Enable
sscratch	CSR[140]: Temp register for use in handler, Supervisor Mode
SSIP	supervisor software interrupt
sstatus	CSR[100]: Status register, Supervisor Mode
STIP	supervisor timer interrupt

stval	CSR[143]: Bad address or bad instruction, Supervisor Mode
stvec	CSR[105]: Trap handler base address, Supervisor Mode
SUM	Bit in status register; Permit Supervisor User Memory Access
SXL	Field in status register; Emulation: Reg size when in S Mode
tdata1	CSR[7A1]: Trigger data #1
tdata2	CSR[7A2]: Trigger data #2
tdata3	CSR[7A3]: Trigger data #3
time	CSR[C01]: Current time in ticks
timeh	CSR[C81]: Upper half of time (RV32 only)
TLB	Translation Lookaside Buffer
tselect	CSR[7A0]: Trigger register select
TSR	Bit in status register; Trap SRET instruction
TW	Bit in status register; Time-out Wait
TVM	Bit in status register; Trap Virtual Memory
ucause	CSR[042]: Trap cause code, User Mode
UEIP	user external interrupt
uepc	CSR[041]: Previous value of PC, User Mode
UF	Underflow (a bit within the Floating Point Flags, FFLAGS)
uie	CSR[004]: Interrupt-enable register, User Mode
UIE	Bit in status register; User Mode Interrupt Enable
uip	CSR[044]: Interrupt pending, User Mode
UPIE	Bit in status register; User Mode Previous Interrupt Enable
uscratch	CSR[040]: Temp register for use in handler, User Mode
USIP	user software interrupt
ustatus	CSR[000]: Status register, User Mode
UTIP	user timer interrupt
utval	CSR[043]: Bad address or bad instruction, User Mode
utvec	CSR[005]: Trap handler base address, User Mode
UXL	Field in status register; Emulation: Reg size when in User Mode
VPN	Virtual Page Number
WFI	Wait For Interrupt
XOR	Exclusive-Or
XS	Field in status register; status of extension (clean/dirty/…)

About the Author

Professor Harry H. Porter III teaches in the Department of Computer Science at Portland State University. He has produced several video courses, notably on the Theory of Computation. Recently he built a complete computer using the relay technology of the 1940s. The computer has eight general purpose 8 bit registers, a 16 bit program counter, and a complete instruction set, all housed in mahogany cabinets as shown. Porter also designed and constructed the BLITZ System, a collection of software designed to support a university-level course on Operating Systems. Using the software, students implement a small, but complete, time-sliced, VM-based operating system kernel. Porter has habit of designing and implementing programming languages, the most recent being a language specifically targeted at kernel implementation.

Porter holds an Sc.B. from Brown University and a Ph.D. from the Oregon Graduate Center.

Porter lives in Portland, Oregon. When not trying to figure out how his computer works, he skis, hikes, travels, and spends time with his children building things.

Professor Porter's website: www.cs.pdx.edu/~harry