RISC-V ABIs Specification

Version 1.1, January 30, 2024: Pre-release

Table of Contents

Preamble

This document is in the [Development state](#)

Assume everything can change.

Contributors to all versions of the spec in alphabetical order:

Alex Bradbury, Andrew Burgess, Chih-Mao Chen, Zakk Chen, Kito Cheng, Nelson Chu, Michael Clark, Jessica Clarke, Palmer Dabbelt, Sam Elliott, Gonzalo Brito Gadeschi, Sebastian Huber, Roger Ferrer Ibanez, Quey-Liang Kao, Nick Knight, Luís Marques, Evandro Menezes, Max Nordlund, Stefan O'Rear, Konrad Schwarz, Fangrui Song, Hsiangkai Wang, Andrew Waterman, Jim Wilson

Please cite as: `RISC-V ABIs Specification, Document Version January 30, 2024-draft', Editors Kito Cheng and Jessica Clarke, RISC-V International, January 2024.

The latest version of this document can be found here: github.com/riscv-non-isa/riscv-elf-psabi-doc.

This specification is written in collaboration with the development communities of the major open-source toolchain and operating system communities, and as such specifies what has been agreed upon and implemented. As a result, any changes to this specification that are not backwards-compatible would break ABI compatibility for those toolchains, which is not permitted unless for features explicitly marked as experimental, and so will not be made unless absolutely necessary, regardless of whether the specification is a pre-release version, ratified version or anything in between. This means any version of this specification published at the above link can be regarded as stable in the technical sense of the word (but not necessarily in the official RISC-V International specification state meaning), with the official specification state being an indicator of the completeness, clarity and general editorial quality of the specification.

Introduction

This specification provides the processor-specific application binary interface document for RISC-V.

This specification consists of the following three parts:

- Calling convention
- ELF specification
- DWARF specification

A future revision of this ABI will include a canonical set of mappings for memory model synchronization primitives.

Terms and Abbreviations

This specification uses the following terms and abbreviations:

Term	Meaning
ABI	Application Binary Interface
gABI	Generic System V Application Binary Interface
ELF	Executable and Linking Format
psABI	Processor-Specific ABI
DWARF	Debugging With Arbitrary Record Formats
GOT	Global Offset Table
PLT	Procedure Linkage Table
PC	Program Counter
TLS	Thread-Local Storage
NTBS	Null-Terminated Byte String
XLEN	The width of an integer register in bits
FLEN	The width of a floating-point register in bits
Linker relaxation	A mechanism for optimizing programs at link-time, see Chapter 9 for more detail.
RVWMO	RISC-V Weak Memory Order, as defined in the RISC-V specification.

Status of ABI

ABI Name	Status
ILP32	Ratified
ILP32F	Ratified
ILP32D	Ratified
ILP32E	Draft
LP64	Ratified
LP64F	Ratified
LP64D	Ratified
LP64Q	Ratified

ABI for big-endian is **NOT** included in this specification, we intend to define that in future version of this specification.

RISC-V Calling Conventions

Chapter 1. Register Convention

1.1. Integer Register Convention

Table 1. Integer register convention

Name	ABI Mnemonic	Meaning	Preserved across calls?
x0	zero	Zero	— (Immutable)
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	— (Unallocatable)
x4	tp	Thread pointer	— (Unallocatable)
x5 - x7	t0 - t2	Temporary registers	No
x8 - x9	s0 - s1	Callee-saved registers	Yes
x10 - x17	a0 - a7	Argument registers	No
x18 - x27	s2 - s11	Callee-saved registers	Yes
x28 - x31	t3 - t6	Temporary registers	No

In the standard ABI, procedures should not modify the integer registers tp and gp, because signal handlers may rely upon their values.

The presence of a frame pointer is optional. If a frame pointer exists, it must reside in x8 (s0); the register remains callee-saved.

If a platform requires use of a dedicated general-purpose register for a platform-specific purpose, it is recommended to use gp (x3). The platform ABI specification must document the use of this register. For such platforms, care must be taken to ensure all code (compiler generated or otherwise) avoids using gp in a way incompatible with the platform specific purpose, and that global pointer relaxation is disabled in the toolchain.

1.2. Frame Pointer Convention

The presence of a frame pointer is optional. If a frame pointer exists, it must reside in x8 (s0); the register remains callee-saved.

Code that uses a frame pointer will construct a linked list of stack frames, where each frame links to its caller using a "frame record". A frame record consists of two XLEN values on the stack; the return address and the link to the next frame record. The frame pointer register will point to the innermost frame, thereby starting the linked list. By convention, the lowest XLEN value shall point to the previous frame, while the next XLEN value shall be the return address. The end of the frame record chain is indicated by the address zero appearing as the next link in the chain.

After the prologue, the frame pointer register will point to the Canonical Frame Address or CFA, which is the stack pointer value on entry to the current procedure. The previous frame pointer and return address pair will reside just prior to the current stack address held in `fp`. This puts the return address at `fp - XLEN/8`, and the previous frame pointer at `fp - 2 * XLEN/8`.

It is left to the platform to determine the level of conformance with this convention. A platform may choose:

- not to maintain a frame chain and use the frame pointer register as a general purpose callee-saved register.
- to allow the frame pointer register be used as a general purpose callee-saved register, but provide a platform specific mechanism to reliably detect this condition.
- to use a frame pointer to address a valid frame record at all times, but allow any procedure to choose to forgo creating a frame record.
- to use the frame pointer to address a valid frame record at all times, except leaf functions, who may elect to forgo creating a frame record.

1.3. Floating-point Register Convention

Table 2. Floating-point register convention

Name	ABI Mnemonic	Meaning	Preserved across calls?
f0 - f7	ft0 - ft7	Temporary registers	No
f8 - f9	fs0 - fs1	Callee-saved registers	Yes*
f10 - f17	fa0 - fa7	Argument registers	No
f18 - f27	fs2 - fs11	Callee-saved registers	Yes*
f28 - f31	ft8 - ft11	Temporary registers	No

*: Floating-point values in callee-saved registers are only preserved across calls if they are no larger than the width of a floating-point register in the targeted ABI. Therefore, these registers can always be considered temporaries if targeting the base integer calling convention.

The Floating-Point Control and Status Register (fcsr) must have thread storage duration in accordance with C11 section 7.6 "Floating-point environment <fenv.h>".

1.4. Vector Register Convention

Table 3. Vector register convention for standard calling convention

Name	ABI Mnemonic	Meaning	Preserved across calls?
v0-v31		Temporary registers	No
vl		Vector length	No
vtype		Vector data type register	No
vxrm		Vector fixed-point rounding mode register	No
vxsat		Vector fixed-point saturation flag register	No

*Table 4. Vector register convention for standard vector calling convention variant**

Name	ABI Mnemonic	Meaning	Preserved across calls?
v0		Argument register	No
v1-v7		Callee-saved registers	Yes
v8-v23		Argument registers	No
v24-v31		Callee-saved registers	Yes
vl		Vector length	No
vtype		Vector data type register	No
vxrm		Vector fixed-point rounding mode register	No
vxsat		Vector fixed-point saturation flag register	No

*: Functions that use vector registers to pass arguments and return values must follow this calling convention. Some programming languages can require extra functions to follow this calling convention (e.g. C/C++ functions with attribute `riscv_vector_cc`).

Please refer to the Section 2.3 section for more details about standard vector calling convention variant.

The `vxrm` and `vxsat` fields of `vcsr` are not preserved across calls and their values are unspecified upon entry.

Procedures may assume that `vstart` is zero upon entry. Procedures may assume that `vstart` is zero upon return from a procedure call.

Application software should normally not write `vstart` explicitly. Any procedure that does explicitly write `vstart` to a nonzero value must zero `vstart` before either returning or calling another procedure.

Chapter 2. Procedure Calling Convention

This chapter defines standard calling conventions and standard calling convention variants, and describes how to pass arguments and return values.

Functions must follow the register convention defined in calling convention: the contents of any register without specifying it as an argument register in the calling convention are unspecified upon entry, and the content of any register without specifying it as a return value register or callee-saved in the calling convention are unspecified upon exit, the contents of all callee-saved registers must be restored to what was set on entry, and the contents of any fixed registers like `gp` and `tp` never change,

> Calling convention for big-endian is **NOT** included in this specification yet, we intend to define that in future version of this specification.

2.1. Integer Calling Convention

The base integer calling convention provides eight argument registers, a0-a7, the first two of which are also used to return values.

Scalars that are at most XLEN bits wide are passed in a single argument register, or on the stack by value if none is available. When passed in registers or on the stack, integer scalars narrower than XLEN bits are widened according to the sign of their type up to 32 bits, then sign-extended to XLEN bits. When passed in registers or on the stack, floating-point types narrower than XLEN bits are widened to XLEN bits, with the upper bits undefined.

Scalars that are 2×XLEN bits wide are passed in a pair of argument registers, with the low-order XLEN bits in the lower-numbered register and the high-order XLEN bits in the higher-numbered register. If no argument registers are available, the scalar is passed on the stack by value. If exactly one register is available, the low-order XLEN bits are passed in the register and the high-order XLEN bits are passed on the stack.

Scalars wider than 2×XLEN bits are passed by reference and are replaced in the argument list with the address.

Aggregates whose total size is no more than XLEN bits are passed in a register, with the fields laid out as though they were passed in memory. If no register is available, the aggregate is passed on the stack. Aggregates whose total size is no more than 2×XLEN bits are passed in a pair of registers; if only one register is available, the first XLEN bits are passed in a register and the remaining bits are passed on the stack. If no registers are available, the aggregate is passed on the stack. Bits unused due to padding, and bits past the end of an aggregate whose size in bits is not divisible by XLEN, are undefined.

Aggregates or scalars passed on the stack are aligned to the greater of the type alignment and XLEN bits, but never more than the stack alignment.

Aggregates larger than 2×XLEN bits are passed by reference and are replaced in the argument list with the address, as are C++ aggregates with nontrivial copy constructors, destructors, or vtables.

Fixed-length vectors are treated as aggregates.

Empty structs or union arguments or return values are ignored by C compilers which support them as a non-standard extension. This is not the case for C++, which requires them to be sized types.

Bitfields are packed in little-endian fashion. A bitfield that would span the alignment boundary of its integer type is padded to begin at the next alignment boundary. For example, `struct { int x : 10; int y : 12; }` is a 32-bit type with `x` in bits 9-0, `y` in bits 21-10, and bits 31-22 undefined. By contrast, `struct { short x : 10; short y : 12; }` is a 32-bit type with `x` in bits 9-0, `y` in bits 27-16, and bits 31-28 and bits 15-10 undefined.

Bitfields may larger than its integer type, bits excess than its integer type will treat as padding bits, then padding to begin at the next alignment boundary. For example `struct { char x : 9; char y; }` is a 24 byte type with `x` in bits 7-0, `y` in bit 23-16, and bits 15-8 undefined, `struct { char x : 9; char y : 2 }` is a 16-bit type with `x` in bits 7-0, `y` in bit 10-9, and bit 8, bits 15-11 is undefined.

Arguments passed by reference may be modified by the callee.

Floating-point reals are passed the same way as aggregates of the same size; complex floating-point numbers are passed the same way as a struct containing two floating-point reals. (This constraint changes when the integer calling convention is augmented by the hardware floating-point calling convention.)

In the base integer calling convention, variadic arguments are passed in the same manner as named arguments, with one exception. Variadic arguments with 2×XLEN-bit alignment and size at most 2×XLEN bits are passed in an **aligned** register pair (i.e., the first register in the pair is even-numbered), or on the stack by value if none is available. After a variadic argument has been passed on the stack, all future arguments will also be passed on the stack (i.e. the last argument register may be left unused due to the aligned register pair rule).

Values are returned in the same manner as a first named argument of the same type would be passed. If such an argument would have been passed by reference, the caller allocates memory for the return value, and passes the address as an implicit first parameter.

> There is no requirement that the address be returned from the function and so software should not assume that a0 will hold the address of the return value on return.

The stack grows downwards (towards lower addresses) and the stack pointer shall be aligned to a 128-bit boundary upon procedure entry. The first argument passed on the stack is located at offset zero of the stack pointer on function entry; following arguments are stored at correspondingly higher addresses.

In the standard ABI, the stack pointer must remain aligned throughout procedure execution. Non-standard ABI code must realign the stack pointer prior to invoking standard ABI procedures. The operating system must realign the stack pointer prior to invoking a signal handler; hence, POSIX signal handlers need not realign the stack pointer. In systems that service interrupts using the interruptee's stack, the interrupt service routine must realign the stack pointer if linked with any code that uses a non-standard stack-alignment discipline, but need not realign the stack pointer if all code adheres to the standard ABI.

Procedures must not rely upon the persistence of stack-allocated data whose addresses lie below the stack pointer.

Registers s0-s11 shall be preserved across procedure calls. No floating-point registers, if present, are preserved across calls. (This property changes when the integer calling convention is augmented by the hardware floating-point calling convention.)

2.2. Hardware Floating-point Calling Convention

The hardware floating-point calling convention adds eight floating-point argument registers, fa0-fa7, the first two of which are also used to return values. Values are passed in floating-point registers whenever possible, whether or not the integer registers have been exhausted.

The remainder of this section applies only to named arguments. Variadic arguments are passed according to the integer calling convention.

ABI_FLEN refers to the width of a floating-point register in the ABI. The ABI_FLEN must be no wider than the ISA's FLEN. The ISA might have wider floating-point registers than the ABI.

For the purposes of this section, "struct" refers to a C struct with its hierarchy flattened, including any array fields. That is, `struct { struct { float f[1]; } a[2]; }` and `struct { float f0; float f1; }` are treated the same. Fields containing empty structs or unions are ignored while flattening, even in C++, unless they have nontrivial copy constructors or destructors. Fields containing zero-length bit-fields or zero-length arrays are ignored while flattening. Attributes such as `aligned` or `packed` do not interfere with a struct's eligibility for being passed in registers according to the rules below, i.e. `struct { int i; double d; }` and `struct __attribute__((__packed__)) { int i; double d }` are treated the same, as are `struct { float f; float g; }` and `struct { float f; float g __attribute__ ((aligned (8))); }`.

> One exceptional case for the flattening rule is an array of empty structs or unions; C treats it as an empty field, but C++ treats it as a non-empty field since C++ defines the size of an empty struct or union as 1. i.e. for `struct { struct {} e[1]; float f; }` as the first argument, C will treat it like `struct { float f; }` and pass `f` in `fa0` as described below, whereas C++ will pass the pass the entire aggregate in `a0` (XLEN = 64) or `a0` and `a1` (XLEN = 32), as described in the integer calling convention. Zero-length arrays of empty structs or union will be ignored for both C and C++. i.e. For `struct { struct {} e[0]; float f; };`, as the first argument, C and C++ will treat it like `struct { float f; }` and pass `f` in `fa0` as described below.

A real floating-point argument is passed in a floating-point argument register if it is no more than ABI_FLEN bits wide and at least one floating-point argument register is available. Otherwise, it is passed according to the integer calling convention. When a floating-point argument narrower than FLEN bits is passed in a floating-point register, it is 1-extended (NaN-boxed) to FLEN bits.

A struct containing just one floating-point real is passed as though it were a standalone floating-point real.

A struct containing two floating-point reals is passed in two floating-point registers, if neither real is more than ABI_FLEN bits wide and at least two floating-point argument registers are available. (The registers need not be an aligned pair.) Otherwise, it is passed according to the integer calling convention.

A complex floating-point number, or a struct containing just one complex floating-point number, is passed as though it were a struct containing two floating-point reals.

A struct containing one floating-point real and one integer (or bitfield), in either order, is passed in a floating-point register and an integer register, provided the floating-point real is no more than ABI_FLEN bits wide and the integer is no more than XLEN bits wide, and at least one floating-point argument register and at least one integer argument register is available. If the struct is passed in this manner, and the integer is narrower than XLEN bits, the remaining bits are unspecified. If the struct is not passed in this manner, then it is passed according to the integer calling convention.

Unions are never flattened and are always passed according to the integer calling convention.

Values are returned in the same manner as a first named argument of the same type would be passed.

Floating-point registers fs0-fs11 shall be preserved across procedure calls, provided they hold values no more than ABI_FLEN bits wide.

2.3. Standard Vector Calling Convention Variant

The *RISC-V V Vector Extension*[riscv-v-extension] defines a set of thirty-two vector registers, v0-v31. The *RISC-V Vector Extension Intrinsic Document*[rvv-intrinsic-doc] defines vector types which include vector mask types, vector data types, and tuple vector data types. A value of vector type can be stored in vector register groups.

The remainder of this section applies only to named vector arguments, other named arguments and return values follow the standard calling convention. Variadic vector arguments are passed by reference.

v0 is used to pass the first vector mask argument to a function, and to return vector mask result from a function. v8-v23 are used to pass vector data arguments, tuple vector data arguments and the rest vector mask arguments to a function, and to return vector data and vector tuple results from a function.

It must ensure that the entire contents of v1-v7 and v24-v31 are preserved across the call.

Each vector data type and vector tuple type has an LMUL attribute that indicates a vector register group. The value of LMUL indicates the number of vector registers in the vector register group and requires the first vector register number in the vector register group must be a multiple of it. For example, the LMUL of `vint64m8_t` is 8, so v8-v15 vector register group can be allocated to this type, but v9-v16 can not because the v9 register number is not a multiple of 8. If LMUL is less than 1, it is treated as 1. If it is a vector mask type, its LMUL is 1.

Each vector tuple type also has an NFIELDS attribute that indicates how many vector register groups the type contains. Thus a vector tuple type needs to take up LMUL×NFIELDS registers.

The rules for passing vector arguments are as follows:

1. For the first vector mask argument, use v0 to pass it.
2. For vector data arguments or rest vector mask arguments, starting from the v8 register, if a vector register group between v8-v23 that has not been allocated can be found and the first register number is a multiple of LMUL, then allocate this vector register group to the argument and mark these registers as allocated. Otherwise, pass it by reference and are replaced in the argument list with the address.
3. For tuple vector data arguments, starting from the v8 register, if NFIELDS consecutive vector register groups between v8-v23 that have not been allocated can be found and the first register number is a multiple of LMUL, then allocate these vector register groups to the argument and mark these registers as allocated. Otherwise, pass it by reference and are replaced in the argument list with the address.

> The registers assigned to the tuple vector data argument must be consecutive. For example, for the function `void foo(vint32m1_t a, vint32m2_t b, vint32m1x2_t c)`, v8 will be allocated to `a`, v10-v11 will be allocated to `b`, v12-v13 instead of v9 and v12 will beallocated to `c`.

> It should be stressed that the search for the appropriate vector register groups starts at v8 each time and does not start at the next register after the registers are allocated for the previous vector argument. Therefore, it is possible that the vector register number allocated to a vector argument can be less than the vector register number allocated to previous vector arguments. For example, for the function `void foo (vint32m1_t a, vint32m2_t b, vint32m1_t c)`, according to the rules of allocation, v8 will be allocated to `a`, v10-v11 will be allocated to `b` and v9 will be allocated to `c`. This approach allows more vector registers to be allocated to arguments in some cases.

Vector values are returned in the same manner as the first named argument of the same type would be passed.

Vector types are disallowed in struct or union.

Vector arguments and return values are disallowed to pass to an unprototyped function.

> Functions that use the standard vector calling convention variant must be marked with `STO_RISCV_VARIANT_CC`, see Chapter 6 for the meaning of `STO_RISCV_VARIANT_CC`.

> `setjmp`/`longjmp` follow the standard calling convention, which clobbers all vector registers. Hence, the standard vector calling convention variant won't disrupt the `jmp_buf` ABI.

2.4. ILP32E Calling Convention

> RV32E is not a ratified base ISA and so we cannot guarantee the stability of ILP32E,

> in contrast with the rest of this document. This documents the current implementation in GCC as of the time of writing, but may be subject to change.

The ILP32E calling convention is designed to be usable with the RV32E ISA. This calling convention is the same as the integer calling convention, except for the following differences. The stack pointer need only be aligned to a 32-bit boundary. Registers x16-x31 do not participate in the calling convention, so there are only six argument registers, a0-a5, only two callee-saved registers, s0-s1, and only three temporaries, t0-t2.

If used with an ISA that has any of the registers x16-x31 and f0-f31, then these registers are considered temporaries.

The ILP32E calling convention is not compatible with ISAs that have registers that require load and store alignments of more than 32 bits. In particular, this calling convention must not be used with the D ISA extension.

2.5. Named ABIs

This specification defines the following named ABIs:

ILP32

Integer calling-convention only, hardware floating-point calling convention is not used (i.e. ELFCLASS32 and EF_RISCV_FLOAT_ABI_SOFT).

ILP32F

ILP32 with hardware floating-point calling convention for ABI_FLEN=32 (i.e. ELFCLASS32 and EF_RISCV_FLOAT_ABI_SINGLE).

ILP32D

ILP32 with hardware floating-point calling convention for ABI_FLEN=64 (i.e. ELFCLASS32 and EF_RISCV_FLOAT_ABI_DOUBLE).

ILP32E

ILP32E calling-convention only, hardware floating-point calling convention is not used (i.e. ELFCLASS32, EF_RISCV_FLOAT_ABI_SOFT, and EF_RISCV_RVE).

LP64

Integer calling-convention only, hardware floating-point calling convention is not used (i.e. ELFCLASS64 and EF_RISCV_FLOAT_ABI_SOFT).

LP64F

LP64 with hardware floating-point calling convention for ABI_FLEN=32 (i.e. ELFCLASS64 and EF_RISCV_FLOAT_ABI_SINGLE).

LP64D

LP64 with hardware floating-point calling convention for ABI_FLEN=64 (i.e. ELFCLASS64 and EF_RISCV_FLOAT_ABI_DOUBLE).

LP64Q

LP64 with hardware floating-point calling convention for ABI_FLEN=128 (i.e. ELFCLASS64 and EF_RISCV_FLOAT_ABI_QUAD).

The ILP32* ABIs are only compatible with RV32* ISAs, and the LP64* ABIs are only compatible with RV64* ISAs. A future version of this specification may define an ILP32 ABI for the RV64 ISA, but currently this is not a supported operating mode.

The *F ABIs require the *F ISA extension, the *D ABIs require the *D ISA extension, and the LP64Q ABI requires the Q ISA extension.

This means code targeting the Zfinx extension always uses the ILP32, ILP32E or LP64 integer calling-convention only ABIs as there is no dedicated hardware floating-point register file.

2.6. Default ABIs

While various different ABIs are technically possible, for software compatibility reasons it is strongly recommended to use the following default ABIs for specific architectures:

on RV32G ILP32D

on RV64G LP64D

Although RV64GQ systems can technically use LP64Q, it is strongly recommended to use LP64D on general-purpose RV64GQ systems for compatibility with standard RV64G software.

Chapter 3. Calling Convention for System Calls

The calling convention for system calls does not fall within the scope of this document. Please refer to the documentation of the RISC-V execution environment interface (e.g OS kernel ABI, SBI).

Chapter 4. C/C++ type details

4.1. C/C++ type sizes and alignments

There are two conventions for C/C++ type sizes and alignments.

ILP32, ILP32F, ILP32D, and ILP32E

Use the following type sizes and alignments (based on the ILP32 convention):

Table 5. C/C++ type sizes and alignments for RV32

Type	Size (Bytes)	Alignment (Bytes)
bool/_Bool	1	1
char	1	1
short	2	2
int	4	4
long	4	4
long long	8	8
void *	4	4
__bf16	2	2
_Float16	2	2
float	4	4
double	8	8
long double	16	16
float _Complex	8	4
double _Complex	16	8
long double _Complex	32	16

LP64, LP64F, LP64D, and LP64Q

Use the following type sizes and alignments (based on the LP64 convention):

Table 6. C/C++ type sizes and alignments for RV64

Type	Size (Bytes)	Alignment (Bytes)
bool/_Bool	1	1
char	1	1
short	2	2
int	4	4

Type	Size (Bytes)	Alignment (Bytes)
long	8	8
long long	8	8
__int128	16	16
void *	8	8
__bf16	2	2
_Float16	2	2
float	4	4
double	8	8
long double	16	16
float _Complex	8	4
double _Complex	16	8
long double _Complex	32	16

The alignment of `max_align_t` is 16.

`CHAR_BIT` is 8.

Structs and unions are aligned to the alignment of their most strictly aligned member. The size of any object is a multiple of its alignment.

4.2. Fixed-length vector

Various compilers have support for fixed-length vector types, for example GCC and Clang both support declaring a type with `attributevector_size(N`, where N is a positive number larger than zero.

The alignment requirement for the fixed length vector shall be equivalent to the alignment requirement of its elemental type.

The size of the fixed length vector is determined by multiplying the size of its elemental type by the total number of elements within the vector.

4.3. C/C++ type representations

`char` is unsigned.

Booleans (`bool`/`_Bool`) stored in memory or when being passed as scalar arguments are either `0` (`false`) or `1` (`true`).

A null pointer (for all types) has the value zero.

`_Float16` is as defined in the C ISO/IEC TS 18661-3 extension.

`__bf16` has the same parameter passing and return rules as for `_Float16`.

`_Complex` types have the same layout as a struct containing two fields of the corresponding real type (`float`, `double`, or `long double`), with the first member holding the real part and the second member holding the imaginary part.

The type `size_t` is defined as `unsigned int` for RV32 and `unsigned long` for RV64.

The type `ptrdiff_t` is defined as `int` for RV32 and `long` for RV64.

4.4. va_list, va_start, and va_arg

The `va_list` type is `void*`. A callee with variadic arguments is responsible for copying the contents of registers used to pass variadic arguments to the vararg save area, which must be contiguous with arguments passed on the stack. The `va_start` macro initializes its `va_list` argument to point to the start of the vararg save area. The `va_arg` macro will increment its `va_list` argument according to the size of the given type, taking into account the rules about 2×XLEN aligned arguments being passed in "aligned" register pairs.

4.5. Vector type sizes and alignments

This section defines the sizes and alignments for the vector types defined in the *RISC-V Vector Extension Intrinsic Document*[rvv-intrinsic-doc]. The actual size of each type is determined by the hardware configuration, which is based on the content of the `vlenb` register.

There are three classes of vector types: the vector mask types, the vector data types and the vector tuple types.

Table 7. Type sizes and alignments for vector mask types

Internal Name	Type	Size (Bytes)	Alignment (Bytes)
__rvv_vbool1_t	vbool1_t	VLENB	1
__rvv_vbool2_t	vbool2_t	VLENB / 2	1
__rvv_vbool4_t	vbool4_t	VLENB / 4	1
__rvv_vbool8_t	vbool8_t	ceil(VLENB / 8)	1
__rvv_vbool16_t	vbool16_t	ceil(VLENB / 16)	1
__rvv_vbool32_t	vbool32_t	ceil(VLENB / 32)	1
__rvv_vbool64_t	vbool64_t	ceil(VLENB / 64)	1

Table 8. Type sizes and alignments for vector data types

Internal Name	Type	Description	__rvv_vint8mf8_t
vint8mf8_t	(VLEN / 8) / 8	1	__rvv_vuint8mf8_t

Internal Name	Type	Description	__rvv_vint8mf8_t
vuint8mf8_t	(VLEN / 8) / 8	1	__rvv_vfloat8mf8_t
vfloat8mf8_t	(VLEN / 8) / 8	1	__rvv_vint8mf4_t
vint8mf4_t	(VLEN / 8) / 4	1	__rvv_vuint8mf4_t
vuint8mf4_t	(VLEN / 8) / 4	1	__rvv_vfloat8mf4_t
vfloat8mf4_t	(VLEN / 8) / 4	1	__rvv_vint8mf2_t
vint8mf2_t	(VLEN / 8) / 2	1	__rvv_vuint8mf2_t
vuint8mf2_t	(VLEN / 8) / 2	1	__rvv_vfloat8mf2_t
vfloat8mf2_t	(VLEN / 8) / 2	1	__rvv_vint8m1_t
vint8m1_t	(VLEN / 8)	1	__rvv_vuint8m1_t
vuint8m1_t	(VLEN / 8)	1	__rvv_vfloat8m1_t
vfloat8m1_t	(VLEN / 8)	1	__rvv_vint8m2_t
vint8m2_t	(VLEN / 8) * 2	1	__rvv_vuint8m2_t
vuint8m2_t	(VLEN / 8) * 2	1	__rvv_vfloat8m2_t
vfloat8m2_t	(VLEN / 8) * 2	1	__rvv_vint8m4_t
vint8m4_t	(VLEN / 8) * 4	1	__rvv_vuint8m4_t
vuint8m4_t	(VLEN / 8) * 4	1	__rvv_vfloat8m4_t
vfloat8m4_t	(VLEN / 8) * 4	1	__rvv_vint8m8_t
vint8m8_t	(VLEN / 8) * 8	1	__rvv_vuint8m8_t
vuint8m8_t	(VLEN / 8) * 8	1	__rvv_vfloat8m8_t

Internal Name	Type	Description	__rvv_vint8mf8_t
vfloat8m8_t	(VLEN / 8) * 8	1	__rvv_vint16mf8_t
vint16mf8_t	(VLEN / 8) / 8	2	__rvv_vuint16mf8_t
vuint16mf8_t	(VLEN / 8) / 8	2	__rvv_vbfloat16mf8_t
vbfloat16mf8_t	(VLEN / 8) / 8	2	__rvv_vint16mf4_t
vint16mf4_t	(VLEN / 8) / 4	2	__rvv_vuint16mf4_t
vuint16mf4_t	(VLEN / 8) / 4	2	__rvv_vbfloat16mf4_t
vbfloat16mf4_t	(VLEN / 8) / 4	2	__rvv_vint16mf2_t
vint16mf2_t	(VLEN / 8) / 2	2	__rvv_vuint16mf2_t
vuint16mf2_t	(VLEN / 8) / 2	2	__rvv_vbfloat16mf2_t
vbfloat16mf2_t	(VLEN / 8) / 2	2	__rvv_vint16m1_t
vint16m1_t	(VLEN / 8)	2	__rvv_vuint16m1_t
vuint16m1_t	(VLEN / 8)	2	__rvv_vbfloat16m1_t
vbfloat16m1_t	(VLEN / 8)	2	__rvv_vint16m2_t
vint16m2_t	(VLEN / 8) * 2	2	__rvv_vuint16m2_t
vuint16m2_t	(VLEN / 8) * 2	2	__rvv_vbfloat16m2_t
vbfloat16m2_t	(VLEN / 8) * 2	2	__rvv_vint16m4_t
vint16m4_t	(VLEN / 8) * 4	2	__rvv_vuint16m4_t
vuint16m4_t	(VLEN / 8) * 4	2	__rvv_vbfloat16m4_t
vbfloat16m4_t	(VLEN / 8) * 4	2	__rvv_vint16m8_t

Internal Name	Type	Description	__rvv_vint8mf8_t
vint16m8_t	(VLEN / 8) * 8	2	__rvv_vuint16m8_t
vuint16m8_t	(VLEN / 8) * 8	2	__rvv_vbfloat16m8_t
vbfloat16m8_t	(VLEN / 8) * 8	2	__rvv_vint32mf8_t
vint32mf8_t	(VLEN / 8) / 8	4	__rvv_vuint32mf8_t
vuint32mf8_t	(VLEN / 8) / 8	4	__rvv_vfloat32mf8_t
vfloat32mf8_t	(VLEN / 8) / 8	4	__rvv_vint32mf4_t
vint32mf4_t	(VLEN / 8) / 4	4	__rvv_vuint32mf4_t
vuint32mf4_t	(VLEN / 8) / 4	4	__rvv_vfloat32mf4_t
vfloat32mf4_t	(VLEN / 8) / 4	4	__rvv_vint32mf2_t
vint32mf2_t	(VLEN / 8) / 2	4	__rvv_vuint32mf2_t
vuint32mf2_t	(VLEN / 8) / 2	4	__rvv_vfloat32mf2_t
vfloat32mf2_t	(VLEN / 8) / 2	4	__rvv_vint32m1_t
vint32m1_t	(VLEN / 8)	4	__rvv_vuint32m1_t
vuint32m1_t	(VLEN / 8)	4	__rvv_vfloat32m1_t
vfloat32m1_t	(VLEN / 8)	4	__rvv_vint32m2_t
vint32m2_t	(VLEN / 8) * 2	4	__rvv_vuint32m2_t
vuint32m2_t	(VLEN / 8) * 2	4	__rvv_vfloat32m2_t
vfloat32m2_t	(VLEN / 8) * 2	4	__rvv_vint32m4_t
vint32m4_t	(VLEN / 8) * 4	4	__rvv_vuint32m4_t

Internal Name	Type	Description	__rvv_vint8mf8_t
vuint32m4_t	(VLEN / 8) * 4	4	__rvv_vfloat32m4_t
vfloat32m4_t	(VLEN / 8) * 4	4	__rvv_vint32m8_t
vint32m8_t	(VLEN / 8) * 8	4	__rvv_vuint32m8_t
vuint32m8_t	(VLEN / 8) * 8	4	__rvv_vfloat32m8_t
vfloat32m8_t	(VLEN / 8) * 8	4	__rvv_vint64mf8_t
vint64mf8_t	(VLEN / 8) / 8	8	__rvv_vuint64mf8_t
vuint64mf8_t	(VLEN / 8) / 8	8	__rvv_vfloat64mf8_t
vfloat64mf8_t	(VLEN / 8) / 8	8	__rvv_vint64mf4_t
vint64mf4_t	(VLEN / 8) / 4	8	__rvv_vuint64mf4_t
vuint64mf4_t	(VLEN / 8) / 4	8	__rvv_vfloat64mf4_t
vfloat64mf4_t	(VLEN / 8) / 4	8	__rvv_vint64mf2_t
vint64mf2_t	(VLEN / 8) / 2	8	__rvv_vuint64mf2_t
vuint64mf2_t	(VLEN / 8) / 2	8	__rvv_vfloat64mf2_t
vfloat64mf2_t	(VLEN / 8) / 2	8	__rvv_vint64m1_t
vint64m1_t	(VLEN / 8)	8	__rvv_vuint64m1_t
vuint64m1_t	(VLEN / 8)	8	__rvv_vfloat64m1_t
vfloat64m1_t	(VLEN / 8)	8	__rvv_vint64m2_t
vint64m2_t	(VLEN / 8) * 2	8	__rvv_vuint64m2_t
vuint64m2_t	(VLEN / 8) * 2	8	__rvv_vfloat64m2_t

Internal Name	Type	Description	__rvv_vint8mf8_t
vfloat64m2_t	(VLEN / 8) * 2	8	__rvv_vint64m4_t
vint64m4_t	(VLEN / 8) * 4	8	__rvv_vuint64m4_t
vuint64m4_t	(VLEN / 8) * 4	8	__rvv_vfloat64m4_t
vfloat64m4_t	(VLEN / 8) * 4	8	__rvv_vint64m8_t
vint64m8_t	(VLEN / 8) * 8	8	__rvv_vuint64m8_t
vuint64m8_t	(VLEN / 8) * 8	8	__rvv_vfloat64m8_t

| Internal Name | Type | Size (Bytes) | Alignment (Bytes) | rvv_vint8mf8x2_t | vint8mf8x2_t | (VLEN / 8) / 4 | 1 | rvv_vuint8mf8x2_t | vuint8mf8x2_t | (VLEN / 8) / 4 | 1 | rvv_vfloat8mf8x2_t | vfloat8mf8x2_t | (VLEN / 8) / 4 | 1 | rvv_vint8mf8x3_t | vint8mf8x3_t | (VLEN / 8) * 0.375 | 1 | rvv_vuint8mf8x3_t | vuint8mf8x3_t | (VLEN / 8) * 0.375 | 1 | rvv_vfloat8mf8x3_t | vfloat8mf8x3_t | (VLEN / 8) * 0.375 | 1 | rvv_vint8mf8x4_t | vint8mf8x4_t | (VLEN / 8) / 2 | 1 | rvv_vuint8mf8x4_t | vuint8mf8x4_t | (VLEN / 8) / 2 | 1 | rvv_vfloat8mf8x4_t | vfloat8mf8x4_t | (VLEN / 8) / 2 | 1 | rvv_vint8mf8x5_t | vint8mf8x5_t | (VLEN / 8) * 0.625 | 1 | rvv_vuint8mf8x5_t | vuint8mf8x5_t | (VLEN / 8) * 0.625 | 1 | rvv_vfloat8mf8x5_t | vfloat8mf8x5_t | (VLEN / 8) * 0.625 | 1 | rvv_vint8mf8x6_t | vint8mf8x6_t | (VLEN / 8) * 0.75 | 1 | rvv_vuint8mf8x6_t | vuint8mf8x6_t | (VLEN / 8) * 0.75 | 1 | rvv_vfloat8mf8x6_t | vfloat8mf8x6_t | (VLEN / 8) * 0.75 | 1 | rvv_vint8mf8x7_t | vint8mf8x7_t | (VLEN / 8) * 0.875 | 1 | rvv_vuint8mf8x7_t | vuint8mf8x7_t | (VLEN / 8) * 0.875 | 1 | rvv_vfloat8mf8x7_t | vfloat8mf8x7_t | (VLEN / 8) * 0.875 | 1 | rvv_vint8mf8x8_t | vint8mf8x8_t | (VLEN / 8) | 1 | rvv_vuint8mf8x8_t | vuint8mf8x8_t | (VLEN / 8) | 1 | rvv_vfloat8mf8x8_t | vfloat8mf8x8_t | (VLEN / 8) | 1 | rvv_vint8mf4x2_t | vint8mf4x2_t | (VLEN / 8) / 2 | 1 | rvv_vuint8mf4x2_t | vuint8mf4x2_t | (VLEN / 8) / 2 | 1 | rvv_vfloat8mf4x2_t | vfloat8mf4x2_t | (VLEN / 8) / 2 | 1 | rvv_vint8mf4x3_t | vint8mf4x3_t | (VLEN / 8) * 0.75 | 1 | rvv_vuint8mf4x3_t | vuint8mf4x3_t | (VLEN / 8) * 0.75 | 1 | rvv_vfloat8mf4x3_t | vfloat8mf4x3_t | (VLEN / 8) * 0.75 | 1 | rvv_vint8mf4x4_t | vint8mf4x4_t | (VLEN / 8) | 1 | rvv_vuint8mf4x4_t | vuint8mf4x4_t | (VLEN / 8) | 1 | rvv_vfloat8mf4x4_t | vfloat8mf4x4_t | (VLEN / 8) | 1 | rvv_vint8mf4x5_t | vint8mf4x5_t | (VLEN / 8) * 1.25 | 1 | rvv_vuint8mf4x5_t | vuint8mf4x5_t | (VLEN / 8) * 1.25 | 1 | rvv_vfloat8mf4x5_t | vfloat8mf4x5_t | (VLEN / 8) * 1.25 | 1 | rvv_vint8mf4x6_t | vint8mf4x6_t | (VLEN / 8) * 1.5 | 1 | rvv_vuint8mf4x6_t | vuint8mf4x6_t | (VLEN / 8) * 1.5 | 1 | rvv_vfloat8mf4x6_t | vfloat8mf4x6_t | (VLEN / 8) * 1.5 | 1 | rvv_vint8mf4x7_t | vint8mf4x7_t | (VLEN / 8) * 1.75 | 1 | rvv_vuint8mf4x7_t | vuint8mf4x7_t | (VLEN / 8) * 1.75 | 1 | rvv_vfloat8mf4x7_t | vfloat8mf4x7_t | (VLEN / 8) * 1.75 | 1 | rvv_vint8mf4x8_t | vint8mf4x8_t | (VLEN / 8) * 2 | 1 | rvv_vuint8mf4x8_t | vuint8mf4x8_t | (VLEN / 8) * 2 | 1 | rvv_vfloat8mf4x8_t | vfloat8mf4x8_t | (VLEN / 8) * 2 | 1 | rvv_vint8mf2x2_t | vint8mf2x2_t | (VLEN / 8) | 1 | rvv_vuint8mf2x2_t | vuint8mf2x2_t | (VLEN / 8) | 1 | rvv_vfloat8mf2x2_t | vfloat8mf2x2_t | (VLEN / 8) | 1 | rvv_vint8mf2x3_t | vint8mf2x3_t | (VLEN / 8) * 1.5 | 1 | rvv_vuint8mf2x3_t | vuint8mf2x3_t | (VLEN / 8) * 1.5 | 1 | rvv_vfloat8mf2x3_t | vfloat8mf2x3_t | (VLEN / 8) * 1.5 | 1 | rvv_vint8mf2x4_t | vint8mf2x4_t | (VLEN / 8) * 2 | 1 |

rvv_vuint8mf2x4_t | vuint8mf2x4_t | (VLEN / 8) * 2 | 1 | rvv_vfloat8mf2x4_t | vfloat8mf2x4_t | (VLEN / 8) * 2 | 1 | rvv_vint8mf2x5_t | vint8mf2x5_t | (VLEN / 8) * 2.5 | 1 | rvv_vuint8mf2x5_t | vuint8mf2x5_t | (VLEN / 8) * 2.5 | 1 | rvv_vfloat8mf2x5_t | vfloat8mf2x5_t | (VLEN / 8) * 2.5 | 1 | rvv_vint8mf2x6_t | vint8mf2x6_t | (VLEN / 8) * 3 | 1 | rvv_vuint8mf2x6_t | vuint8mf2x6_t | (VLEN / 8) * 3 | 1 | rvv_vfloat8mf2x6_t | vfloat8mf2x6_t | (VLEN / 8) * 3 | 1 | rvv_vint8mf2x7_t | vint8mf2x7_t | (VLEN / 8) * 3.5 | 1 | rvv_vuint8mf2x7_t | vuint8mf2x7_t | (VLEN / 8) * 3.5 | 1 | rvv_vfloat8mf2x7_t | vfloat8mf2x7_t | (VLEN / 8) * 3.5 | 1 | rvv_vint8mf2x8_t | vint8mf2x8_t | (VLEN / 8) * 4 | 1 | rvv_vuint8mf2x8_t | vuint8mf2x8_t | (VLEN / 8) * 4 | 1 | rvv_vfloat8mf2x8_t | vfloat8mf2x8_t | (VLEN / 8) * 4 | 1 | rvv_vint8m1x2_t | vint8m1x2_t | (VLEN / 8) * 2 | 1 | rvv_vuint8m1x2_t | vuint8m1x2_t | (VLEN / 8) * 2 | 1 | rvv_vfloat8m1x2_t | vfloat8m1x2_t | (VLEN / 8) * 2 | 1 | rvv_vint8m1x3_t | vint8m1x3_t | (VLEN / 8) * 3 | 1 | rvv_vuint8m1x3_t | vuint8m1x3_t | (VLEN / 8) * 3 | 1 | rvv_vfloat8m1x3_t | vfloat8m1x3_t | (VLEN / 8) * 3 | 1 | rvv_vint8m1x4_t | vint8m1x4_t | (VLEN / 8) * 4 | 1 | rvv_vuint8m1x4_t | vuint8m1x4_t | (VLEN / 8) * 4 | 1 | rvv_vfloat8m1x4_t | vfloat8m1x4_t | (VLEN / 8) * 4 | 1 | rvv_vint8m1x5_t | vint8m1x5_t | (VLEN / 8) * 5 | 1 | rvv_vuint8m1x5_t | vuint8m1x5_t | (VLEN / 8) * 5 | 1 | rvv_vfloat8m1x5_t | vfloat8m1x5_t | (VLEN / 8) * 5 | 1 | rvv_vint8m1x6_t | vint8m1x6_t | (VLEN / 8) * 6 | 1 | rvv_vuint8m1x6_t | vuint8m1x6_t | (VLEN / 8) * 6 | 1 | rvv_vfloat8m1x6_t | vfloat8m1x6_t | (VLEN / 8) * 6 | 1 | rvv_vint8m1x7_t | vint8m1x7_t | (VLEN / 8) * 7 | 1 | rvv_vuint8m1x7_t | vuint8m1x7_t | (VLEN / 8) * 7 | 1 | rvv_vfloat8m1x7_t | vfloat8m1x7_t | (VLEN / 8) * 7 | 1 | rvv_vint8m1x8_t | vint8m1x8_t | (VLEN / 8) * 8 | 1 | rvv_vuint8m1x8_t | vuint8m1x8_t | (VLEN / 8) * 8 | 1 | rvv_vfloat8m1x8_t | vfloat8m1x8_t | (VLEN / 8) * 8 | 1 | rvv_vint8m2x2_t | vint8m2x2_t | (VLEN / 8) * 4 | 1 | rvv_vuint8m2x2_t | vuint8m2x2_t | (VLEN / 8) * 4 | 1 | rvv_vfloat8m2x2_t | vfloat8m2x2_t | (VLEN / 8) * 4 | 1 | rvv_vint8m2x3_t | vint8m2x3_t | (VLEN / 8) * 6 | 1 | rvv_vuint8m2x3_t | vuint8m2x3_t | (VLEN / 8) * 6 | 1 | rvv_vfloat8m2x3_t | vfloat8m2x3_t | (VLEN / 8) * 6 | 1 | rvv_vint8m2x4_t | vint8m2x4_t | (VLEN / 8) * 8 | 1 | rvv_vuint8m2x4_t | vuint8m2x4_t | (VLEN / 8) * 8 | 1 | rvv_vfloat8m2x4_t | vfloat8m2x4_t | (VLEN / 8) * 8 | 1 | rvv_vint8m4x2_t | vint8m4x2_t | (VLEN / 8) * 8 | 1 | rvv_vuint8m4x2_t | vuint8m4x2_t | (VLEN / 8) * 8 | 1 | rvv_vfloat8m4x2_t | vfloat8m4x2_t | (VLEN / 8) * 8 | 1 | rvv_vint16mf8x2_t | vint16mf8x2_t | (VLEN / 8) / 4 | 2 | rvv_vuint16mf8x2_t | vuint16mf8x2_t | (VLEN / 8) / 4 | 2 | rvv_vbfloat16mf8x2_t | vbfloat16mf8x2_t | (VLEN / 8) / 4 | 2 | rvv_vint16mf8x3_t | vint16mf8x3_t | (VLEN / 8) * 0.375 | 2 | rvv_vuint16mf8x3_t | vuint16mf8x3_t | (VLEN / 8) * 0.375 | 2 | rvv_vbfloat16mf8x3_t | vbfloat16mf8x3_t | (VLEN / 8) * 0.375 | 2 | rvv_vint16mf8x4_t | vint16mf8x4_t | (VLEN / 8) / 2 | 2 | rvv_vuint16mf8x4_t | vuint16mf8x4_t | (VLEN / 8) / 2 | 2 | rvv_vbfloat16mf8x4_t | vbfloat16mf8x4_t | (VLEN / 8) / 2 | 2 | rvv_vint16mf8x5_t | vint16mf8x5_t | (VLEN / 8) * 0.625 | 2 | rvv_vuint16mf8x5_t | vuint16mf8x5_t | (VLEN / 8) * 0.625 | 2 | rvv_vbfloat16mf8x5_t | vbfloat16mf8x5_t | (VLEN / 8) * 0.625 | 2 | rvv_vint16mf8x6_t | vint16mf8x6_t | (VLEN / 8) * 0.75 | 2 | rvv_vuint16mf8x6_t | vuint16mf8x6_t | (VLEN / 8) * 0.75 | 2 | rvv_vbfloat16mf8x6_t | vbfloat16mf8x6_t | (VLEN / 8) * 0.75 | 2 | rvv_vint16mf8x7_t | vint16mf8x7_t | (VLEN / 8) * 0.875 | 2 | rvv_vuint16mf8x7_t | vuint16mf8x7_t | (VLEN / 8) * 0.875 | 2 | rvv_vbfloat16mf8x7_t | vbfloat16mf8x7_t | (VLEN / 8) * 0.875 | 2 | rvv_vint16mf8x8_t | vint16mf8x8_t | (VLEN / 8) | 2 | rvv_vuint16mf8x8_t | vuint16mf8x8_t | (VLEN / 8) | 2 | rvv_vbfloat16mf8x8_t | vbfloat16mf8x8_t | (VLEN / 8) | 2 | rvv_vint16mf4x2_t | vint16mf4x2_t | (VLEN / 8) / 2 | 2 | rvv_vuint16mf4x2_t | vuint16mf4x2_t | (VLEN / 8) / 2 | 2 | rvv_vbfloat16mf4x2_t | vbfloat16mf4x2_t | (VLEN / 8) / 2 | 2 | rvv_vint16mf4x3_t | vint16mf4x3_t | (VLEN / 8) * 0.75 | 2 | rvv_vuint16mf4x3_t | vuint16mf4x3_t | (VLEN / 8) * 0.75 | 2 | rvv_vbfloat16mf4x3_t | vbfloat16mf4x3_t | (VLEN / 8) * 0.75 | 2 | rvv_vint16mf4x4_t | vint16mf4x4_t | (VLEN / 8) | 2 | rvv_vuint16mf4x4_t | vuint16mf4x4_t | (VLEN / 8) | 2 | rvv_vbfloat16mf4x4_t | vbfloat16mf4x4_t | (VLEN / 8) | 2 | rvv_vint16mf4x5_t | vint16mf4x5_t | (VLEN / 8) * 1.25 | 2 | rvv_vuint16mf4x5_t | vuint16mf4x5_t | (VLEN / 8) * 1.25 | 2 | rvv_vbfloat16mf4x5_t | vbfloat16mf4x5_t | (VLEN / 8) * 1.25

| 2 | rvv_vint16mf4x6_t | vint16mf4x6_t | (VLEN / 8) * 1.5 | 2 | rvv_vuint16mf4x6_t | vuint16mf4x6_t | (VLEN / 8) * 1.5 | 2 | rvv_vbfloat16mf4x6_t | vbfloat16mf4x6_t | (VLEN / 8) * 1.5 | 2 | rvv_vint16mf4x7_t | vint16mf4x7_t | (VLEN / 8) * 1.75 | 2 | rvv_vuint16mf4x7_t | vuint16mf4x7_t | (VLEN / 8) * 1.75 | 2 | rvv_vbfloat16mf4x7_t | vbfloat16mf4x7_t | (VLEN / 8) * 1.75 | 2 | rvv_vint16mf4x8_t | vint16mf4x8_t | (VLEN / 8) * 2 | 2 | rvv_vuint16mf4x8_t | vuint16mf4x8_t | (VLEN / 8) * 2 | 2 | rvv_vbfloat16mf4x8_t | vbfloat16mf4x8_t | (VLEN / 8) * 2 | 2 | rvv_vint16mf2x2_t | vint16mf2x2_t | (VLEN / 8) | 2 | rvv_vuint16mf2x2_t | vuint16mf2x2_t | (VLEN / 8) | 2 | rvv_vbfloat16mf2x2_t | vbfloat16mf2x2_t | (VLEN / 8) | 2 | rvv_vint16mf2x3_t | vint16mf2x3_t | (VLEN / 8) * 1.5 | 2 | rvv_vuint16mf2x3_t | vuint16mf2x3_t | (VLEN / 8) * 1.5 | 2 | rvv_vbfloat16mf2x3_t | vbfloat16mf2x3_t | (VLEN / 8) * 1.5 | 2 | rvv_vint16mf2x4_t | vint16mf2x4_t | (VLEN / 8) * 2 | 2 | rvv_vuint16mf2x4_t | vuint16mf2x4_t | (VLEN / 8) * 2 | 2 | rvv_vbfloat16mf2x4_t | vbfloat16mf2x4_t | (VLEN / 8) * 2 | 2 | rvv_vint16mf2x5_t | vint16mf2x5_t | (VLEN / 8) * 2.5 | 2 | rvv_vuint16mf2x5_t | vuint16mf2x5_t | (VLEN / 8) * 2.5 | 2 | rvv_vbfloat16mf2x5_t | vbfloat16mf2x5_t | (VLEN / 8) * 2.5 | 2 | rvv_vint16mf2x6_t | vint16mf2x6_t | (VLEN / 8) * 3 | 2 | rvv_vuint16mf2x6_t | vuint16mf2x6_t | (VLEN / 8) * 3 | 2 | rvv_vbfloat16mf2x6_t | vbfloat16mf2x6_t | (VLEN / 8) * 3 | 2 | rvv_vint16mf2x7_t | vint16mf2x7_t | (VLEN / 8) * 3.5 | 2 | rvv_vuint16mf2x7_t | vuint16mf2x7_t | (VLEN / 8) * 3.5 | 2 | rvv_vbfloat16mf2x7_t | vbfloat16mf2x7_t | (VLEN / 8) * 3.5 | 2 | rvv_vint16mf2x8_t | vint16mf2x8_t | (VLEN / 8) * 4 | 2 | rvv_vuint16mf2x8_t | vuint16mf2x8_t | (VLEN / 8) * 4 | 2 | rvv_vbfloat16mf2x8_t | vbfloat16mf2x8_t | (VLEN / 8) * 4 | 2 | rvv_vint16m1x2_t | vint16m1x2_t | (VLEN / 8) * 2 | 2 | rvv_vuint16m1x2_t | vuint16m1x2_t | (VLEN / 8) * 2 | 2 | rvv_vbfloat16m1x2_t | vbfloat16m1x2_t | (VLEN / 8) * 2 | 2 | rvv_vint16m1x3_t | vint16m1x3_t | (VLEN / 8) * 3 | 2 | rvv_vuint16m1x3_t | vuint16m1x3_t | (VLEN / 8) * 3 | 2 | rvv_vbfloat16m1x3_t | vbfloat16m1x3_t | (VLEN / 8) * 3 | 2 | rvv_vint16m1x4_t | vint16m1x4_t | (VLEN / 8) * 4 | 2 | rvv_vuint16m1x4_t | vuint16m1x4_t | (VLEN / 8) * 4 | 2 | rvv_vbfloat16m1x4_t | vbfloat16m1x4_t | (VLEN / 8) * 4 | 2 | rvv_vint16m1x5_t | vint16m1x5_t | (VLEN / 8) * 5 | 2 | rvv_vuint16m1x5_t | vuint16m1x5_t | (VLEN / 8) * 5 | 2 | rvv_vbfloat16m1x5_t | vbfloat16m1x5_t | (VLEN / 8) * 5 | 2 | rvv_vint16m1x6_t | vint16m1x6_t | (VLEN / 8) * 6 | 2 | rvv_vuint16m1x6_t | vuint16m1x6_t | (VLEN / 8) * 6 | 2 | rvv_vbfloat16m1x6_t | vbfloat16m1x6_t | (VLEN / 8) * 6 | 2 | rvv_vint16m1x7_t | vint16m1x7_t | (VLEN / 8) * 7 | 2 | rvv_vuint16m1x7_t | vuint16m1x7_t | (VLEN / 8) * 7 | 2 | rvv_vbfloat16m1x7_t | vbfloat16m1x7_t | (VLEN / 8) * 7 | 2 | rvv_vint16m1x8_t | vint16m1x8_t | (VLEN / 8) * 8 | 2 | rvv_vuint16m1x8_t | vuint16m1x8_t | (VLEN / 8) * 8 | 2 | rvv_vbfloat16m1x8_t | vbfloat16m1x8_t | (VLEN / 8) * 8 | 2 | rvv_vint16m2x2_t | vint16m2x2_t | (VLEN / 8) * 4 | 2 | rvv_vuint16m2x2_t | vuint16m2x2_t | (VLEN / 8) * 4 | 2 | rvv_vbfloat16m2x2_t | vbfloat16m2x2_t | (VLEN / 8) * 4 | 2 | rvv_vint16m2x3_t | vint16m2x3_t | (VLEN / 8) * 6 | 2 | rvv_vuint16m2x3_t | vuint16m2x3_t | (VLEN / 8) * 6 | 2 | rvv_vbfloat16m2x3_t | vbfloat16m2x3_t | (VLEN / 8) * 6 | 2 | rvv_vint16m2x4_t | vint16m2x4_t | (VLEN / 8) * 8 | 2 | rvv_vuint16m2x4_t | vuint16m2x4_t | (VLEN / 8) * 8 | 2 | rvv_vbfloat16m2x4_t | vbfloat16m2x4_t | (VLEN / 8) * 8 | 2 | rvv_vint16m4x2_t | vint16m4x2_t | (VLEN / 8) * 8 | 2 | rvv_vuint16m4x2_t | vuint16m4x2_t | (VLEN / 8) * 8 | 2 | rvv_vbfloat16m4x2_t | vbfloat16m4x2_t | (VLEN / 8) * 8 | 2 | rvv_vint32mf8x2_t | vint32mf8x2_t | (VLEN / 8) / 4 | 4 | rvv_vuint32mf8x2_t | vuint32mf8x2_t | (VLEN / 8) / 4 | 4 | rvv_vfloat32mf8x2_t | vfloat32mf8x2_t | (VLEN / 8) / 4 | 4 | rvv_vint32mf8x3_t | vint32mf8x3_t | (VLEN / 8) * 0.375 | 4 | rvv_vuint32mf8x3_t | vuint32mf8x3_t | (VLEN / 8) * 0.375 | 4 | rvv_vfloat32mf8x3_t | vfloat32mf8x3_t | (VLEN / 8) * 0.375 | 4 | rvv_vint32mf8x4_t | vint32mf8x4_t | (VLEN / 8) / 2 | 4 | rvv_vuint32mf8x4_t | vuint32mf8x4_t | (VLEN / 8) / 2 | 4 | rvv_vfloat32mf8x4_t | vfloat32mf8x4_t | (VLEN / 8) / 2 | 4 | rvv_vint32mf8x5_t | vint32mf8x5_t | (VLEN / 8) * 0.625 | 4 | rvv_vuint32mf8x5_t | vuint32mf8x5_t | (VLEN / 8) * 0.625 | 4 | rvv_vfloat32mf8x5_t | vfloat32mf8x5_t | (VLEN / 8) * 0.625 | 4 | rvv_vint32mf8x6_t | vint32mf8x6_t | (VLEN / 8) * 0.75 | 4 | rvv_vuint32mf8x6_t | vuint32mf8x6_t |

(VLEN / 8) * 0.75 | 4 | rvv_vfloat32mf8x6_t | vfloat32mf8x6_t | (VLEN / 8) * 0.75 | 4 | rvv_vint32mf8x7_t | vint32mf8x7_t | (VLEN / 8) * 0.875 | 4 | rvv_vuint32mf8x7_t | vuint32mf8x7_t | (VLEN / 8) * 0.875 | 4 | rvv_vfloat32mf8x7_t | vfloat32mf8x7_t | (VLEN / 8) * 0.875 | 4 | rvv_vint32mf8x8_t | vint32mf8x8_t | (VLEN / 8) | 4 | rvv_vuint32mf8x8_t | vuint32mf8x8_t | (VLEN / 8) | 4 | rvv_vfloat32mf8x8_t | vfloat32mf8x8_t | (VLEN / 8) | 4 | rvv_vint32mf4x2_t | vint32mf4x2_t | (VLEN / 8) / 2 | 4 | rvv_vuint32mf4x2_t | vuint32mf4x2_t | (VLEN / 8) / 2 | 4 | rvv_vfloat32mf4x2_t | vfloat32mf4x2_t | (VLEN / 8) / 2 | 4 | rvv_vint32mf4x3_t | vint32mf4x3_t | (VLEN / 8) * 0.75 | 4 | rvv_vuint32mf4x3_t | vuint32mf4x3_t | (VLEN / 8) * 0.75 | 4 | rvv_vfloat32mf4x3_t | vfloat32mf4x3_t | (VLEN / 8) * 0.75 | 4 | rvv_vint32mf4x4_t | vint32mf4x4_t | (VLEN / 8) | 4 | rvv_vuint32mf4x4_t | vuint32mf4x4_t | (VLEN / 8) | 4 | rvv_vfloat32mf4x4_t | vfloat32mf4x4_t | (VLEN / 8) | 4 | rvv_vint32mf4x5_t | vint32mf4x5_t | (VLEN / 8) * 1.25 | 4 | rvv_vuint32mf4x5_t | vuint32mf4x5_t | (VLEN / 8) * 1.25 | 4 | rvv_vfloat32mf4x5_t | vfloat32mf4x5_t | (VLEN / 8) * 1.25 | 4 | rvv_vint32mf4x6_t | vint32mf4x6_t | (VLEN / 8) * 1.5 | 4 | rvv_vuint32mf4x6_t | vuint32mf4x6_t | (VLEN / 8) * 1.5 | 4 | rvv_vfloat32mf4x6_t | vfloat32mf4x6_t | (VLEN / 8) * 1.5 | 4 | rvv_vint32mf4x7_t | vint32mf4x7_t | (VLEN / 8) * 1.75 | 4 | rvv_vuint32mf4x7_t | vuint32mf4x7_t | (VLEN / 8) * 1.75 | 4 | rvv_vfloat32mf4x7_t | vfloat32mf4x7_t | (VLEN / 8) * 1.75 | 4 | rvv_vint32mf4x8_t | vint32mf4x8_t | (VLEN / 8) * 2 | 4 | rvv_vuint32mf4x8_t | vuint32mf4x8_t | (VLEN / 8) * 2 | 4 | rvv_vfloat32mf4x8_t | vfloat32mf4x8_t | (VLEN / 8) * 2 | 4 | rvv_vint32mf2x2_t | vint32mf2x2_t | (VLEN / 8) | 4 | rvv_vuint32mf2x2_t | vuint32mf2x2_t | (VLEN / 8) | 4 | rvv_vfloat32mf2x2_t | vfloat32mf2x2_t | (VLEN / 8) | 4 | rvv_vint32mf2x3_t | vint32mf2x3_t | (VLEN / 8) * 1.5 | 4 | rvv_vuint32mf2x3_t | vuint32mf2x3_t | (VLEN / 8) * 1.5 | 4 | rvv_vfloat32mf2x3_t | vfloat32mf2x3_t | (VLEN / 8) * 1.5 | 4 | rvv_vint32mf2x4_t | vint32mf2x4_t | (VLEN / 8) * 2 | 4 | rvv_vuint32mf2x4_t | vuint32mf2x4_t | (VLEN / 8) * 2 | 4 | rvv_vfloat32mf2x4_t | vfloat32mf2x4_t | (VLEN / 8) * 2 | 4 | rvv_vint32mf2x5_t | vint32mf2x5_t | (VLEN / 8) * 2.5 | 4 | rvv_vuint32mf2x5_t | vuint32mf2x5_t | (VLEN / 8) * 2.5 | 4 | rvv_vfloat32mf2x5_t | vfloat32mf2x5_t | (VLEN / 8) * 2.5 | 4 | rvv_vint32mf2x6_t | vint32mf2x6_t | (VLEN / 8) * 3 | 4 | rvv_vuint32mf2x6_t | vuint32mf2x6_t | (VLEN / 8) * 3 | 4 | rvv_vfloat32mf2x6_t | vfloat32mf2x6_t | (VLEN / 8) * 3 | 4 | rvv_vint32mf2x7_t | vint32mf2x7_t | (VLEN / 8) * 3.5 | 4 | rvv_vuint32mf2x7_t | vuint32mf2x7_t | (VLEN / 8) * 3.5 | 4 | rvv_vfloat32mf2x7_t | vfloat32mf2x7_t | (VLEN / 8) * 3.5 | 4 | rvv_vint32mf2x8_t | vint32mf2x8_t | (VLEN / 8) * 4 | 4 | rvv_vuint32mf2x8_t | vuint32mf2x8_t | (VLEN / 8) * 4 | 4 | rvv_vfloat32mf2x8_t | vfloat32mf2x8_t | (VLEN / 8) * 4 | 4 | rvv_vint32m1x2_t | vint32m1x2_t | (VLEN / 8) * 2 | 4 | rvv_vuint32m1x2_t | vuint32m1x2_t | (VLEN / 8) * 2 | 4 | rvv_vfloat32m1x2_t | vfloat32m1x2_t | (VLEN / 8) * 2 | 4 | rvv_vint32m1x3_t | vint32m1x3_t | (VLEN / 8) * 3 | 4 | rvv_vuint32m1x3_t | vuint32m1x3_t | (VLEN / 8) * 3 | 4 | rvv_vfloat32m1x3_t | vfloat32m1x3_t | (VLEN / 8) * 3 | 4 | rvv_vint32m1x4_t | vint32m1x4_t | (VLEN / 8) * 4 | 4 | rvv_vuint32m1x4_t | vuint32m1x4_t | (VLEN / 8) * 4 | 4 | rvv_vfloat32m1x4_t | vfloat32m1x4_t | (VLEN / 8) * 4 | 4 | rvv_vint32m1x5_t | vint32m1x5_t | (VLEN / 8) * 5 | 4 | rvv_vuint32m1x5_t | vuint32m1x5_t | (VLEN / 8) * 5 | 4 | rvv_vfloat32m1x5_t | vfloat32m1x5_t | (VLEN / 8) * 5 | 4 | rvv_vint32m1x6_t | vint32m1x6_t | (VLEN / 8) * 6 | 4 | rvv_vuint32m1x6_t | vuint32m1x6_t | (VLEN / 8) * 6 | 4 | rvv_vfloat32m1x6_t | vfloat32m1x6_t | (VLEN / 8) * 6 | 4 | rvv_vint32m1x7_t | vint32m1x7_t | (VLEN / 8) * 7 | 4 | rvv_vuint32m1x7_t | vuint32m1x7_t | (VLEN / 8) * 7 | 4 | rvv_vfloat32m1x7_t | vfloat32m1x7_t | (VLEN / 8) * 7 | 4 | rvv_vint32m1x8_t | vint32m1x8_t | (VLEN / 8) * 8 | 4 | rvv_vuint32m1x8_t | vuint32m1x8_t | (VLEN / 8) * 8 | 4 | rvv_vfloat32m1x8_t | vfloat32m1x8_t | (VLEN / 8) * 8 | 4 | rvv_vint32m2x2_t | vint32m2x2_t | (VLEN / 8) * 4 | 4 | rvv_vuint32m2x2_t | vuint32m2x2_t | (VLEN / 8) * 4 | 4 | rvv_vfloat32m2x2_t | vfloat32m2x2_t | (VLEN / 8) * 4 | 4 | rvv_vint32m2x3_t | vint32m2x3_t | (VLEN / 8) * 6 | 4 | rvv_vuint32m2x3_t | vuint32m2x3_t | (VLEN / 8) * 6 | 4 | rvv_vfloat32m2x3_t | vfloat32m2x3_t | (VLEN / 8) * 6 | 4 | rvv_vint32m2x4_t | vint32m2x4_t | (VLEN / 8) * 8 | 4 | rvv_vuint32m2x4_t |

vuint32m2x4_t | (VLEN / 8) * 8 | 4 | rvv_vfloat32m2x4_t | vfloat32m2x4_t | (VLEN / 8) * 8 | 4 | rvv_vint32m4x2_t | vint32m4x2_t | (VLEN / 8) * 8 | 4 | rvv_vuint32m4x2_t | vuint32m4x2_t | (VLEN / 8) * 8 | 4 | rvv_vfloat32m4x2_t | vfloat32m4x2_t | (VLEN / 8) * 8 | 4 | rvv_vint64mf8x2_t | vint64mf8x2_t | (VLEN / 8) / 4 | 8 | rvv_vuint64mf8x2_t | vuint64mf8x2_t | (VLEN / 8) / 4 | 8 | rvv_vfloat64mf8x2_t | vfloat64mf8x2_t | (VLEN / 8) / 4 | 8 | rvv_vint64mf8x3_t | vint64mf8x3_t | (VLEN / 8) * 0.375 | 8 | rvv_vuint64mf8x3_t | vuint64mf8x3_t | (VLEN / 8) * 0.375 | 8 | rvv_vfloat64mf8x3_t | vfloat64mf8x3_t | (VLEN / 8) * 0.375 | 8 | rvv_vint64mf8x4_t | vint64mf8x4_t | (VLEN / 8) / 2 | 8 | rvv_vuint64mf8x4_t | vuint64mf8x4_t | (VLEN / 8) / 2 | 8 | rvv_vfloat64mf8x4_t | vfloat64mf8x4_t | (VLEN / 8) / 2 | 8 | rvv_vint64mf8x5_t | vint64mf8x5_t | (VLEN / 8) * 0.625 | 8 | rvv_vuint64mf8x5_t | vuint64mf8x5_t | (VLEN / 8) * 0.625 | 8 | rvv_vfloat64mf8x5_t | vfloat64mf8x5_t | (VLEN / 8) * 0.625 | 8 | rvv_vint64mf8x6_t | vint64mf8x6_t | (VLEN / 8) * 0.75 | 8 | rvv_vuint64mf8x6_t | vuint64mf8x6_t | (VLEN / 8) * 0.75 | 8 | rvv_vfloat64mf8x6_t | vfloat64mf8x6_t | (VLEN / 8) * 0.75 | 8 | rvv_vint64mf8x7_t | vint64mf8x7_t | (VLEN / 8) * 0.875 | 8 | rvv_vuint64mf8x7_t | vuint64mf8x7_t | (VLEN / 8) * 0.875 | 8 | rvv_vfloat64mf8x7_t | vfloat64mf8x7_t | (VLEN / 8) * 0.875 | 8 | rvv_vint64mf8x8_t | vint64mf8x8_t | (VLEN / 8) | 8 | rvv_vuint64mf8x8_t | vuint64mf8x8_t | (VLEN / 8) | 8 | rvv_vfloat64mf8x8_t | vfloat64mf8x8_t | (VLEN / 8) | 8 | rvv_vint64mf4x2_t | vint64mf4x2_t | (VLEN / 8) / 2 | 8 | rvv_vuint64mf4x2_t | vuint64mf4x2_t | (VLEN / 8) / 2 | 8 | rvv_vfloat64mf4x2_t | vfloat64mf4x2_t | (VLEN / 8) / 2 | 8 | rvv_vint64mf4x3_t | vint64mf4x3_t | (VLEN / 8) * 0.75 | 8 | rvv_vuint64mf4x3_t | vuint64mf4x3_t | (VLEN / 8) * 0.75 | 8 | rvv_vfloat64mf4x3_t | vfloat64mf4x3_t | (VLEN / 8) * 0.75 | 8 | rvv_vint64mf4x4_t | vint64mf4x4_t | (VLEN / 8) | 8 | rvv_vuint64mf4x4_t | vuint64mf4x4_t | (VLEN / 8) | 8 | rvv_vfloat64mf4x4_t | vfloat64mf4x4_t | (VLEN / 8) | 8 | rvv_vint64mf4x5_t | vint64mf4x5_t | (VLEN / 8) * 1.25 | 8 | rvv_vuint64mf4x5_t | vuint64mf4x5_t | (VLEN / 8) * 1.25 | 8 | rvv_vfloat64mf4x5_t | vfloat64mf4x5_t | (VLEN / 8) * 1.25 | 8 | rvv_vint64mf4x6_t | vint64mf4x6_t | (VLEN / 8) * 1.5 | 8 | rvv_vuint64mf4x6_t | vuint64mf4x6_t | (VLEN / 8) * 1.5 | 8 | rvv_vfloat64mf4x6_t | vfloat64mf4x6_t | (VLEN / 8) * 1.5 | 8 | rvv_vint64mf4x7_t | vint64mf4x7_t | (VLEN / 8) * 1.75 | 8 | rvv_vuint64mf4x7_t | vuint64mf4x7_t | (VLEN / 8) * 1.75 | 8 | rvv_vfloat64mf4x7_t | vfloat64mf4x7_t | (VLEN / 8) * 1.75 | 8 | rvv_vint64mf4x8_t | vint64mf4x8_t | (VLEN / 8) * 2 | 8 | rvv_vuint64mf4x8_t | vuint64mf4x8_t | (VLEN / 8) * 2 | 8 | rvv_vfloat64mf4x8_t | vfloat64mf4x8_t | (VLEN / 8) * 2 | 8 | rvv_vint64mf2x2_t | vint64mf2x2_t | (VLEN / 8) | 8 | rvv_vuint64mf2x2_t | vuint64mf2x2_t | (VLEN / 8) | 8 | rvv_vfloat64mf2x2_t | vfloat64mf2x2_t | (VLEN / 8) | 8 | rvv_vint64mf2x3_t | vint64mf2x3_t | (VLEN / 8) * 1.5 | 8 | rvv_vuint64mf2x3_t | vuint64mf2x3_t | (VLEN / 8) * 1.5 | 8 | rvv_vfloat64mf2x3_t | vfloat64mf2x3_t | (VLEN / 8) * 1.5 | 8 | rvv_vint64mf2x4_t | vint64mf2x4_t | (VLEN / 8) * 2 | 8 | rvv_vuint64mf2x4_t | vuint64mf2x4_t | (VLEN / 8) * 2 | 8 | rvv_vfloat64mf2x4_t | vfloat64mf2x4_t | (VLEN / 8) * 2 | 8 | rvv_vint64mf2x5_t | vint64mf2x5_t | (VLEN / 8) * 2.5 | 8 | rvv_vuint64mf2x5_t | vuint64mf2x5_t | (VLEN / 8) * 2.5 | 8 | rvv_vfloat64mf2x5_t | vfloat64mf2x5_t | (VLEN / 8) * 2.5 | 8 | rvv_vint64mf2x6_t | vint64mf2x6_t | (VLEN / 8) * 3 | 8 | rvv_vuint64mf2x6_t | vuint64mf2x6_t | (VLEN / 8) * 3 | 8 | rvv_vfloat64mf2x6_t | vfloat64mf2x6_t | (VLEN / 8) * 3 | 8 | rvv_vint64mf2x7_t | vint64mf2x7_t | (VLEN / 8) * 3.5 | 8 | rvv_vuint64mf2x7_t | vuint64mf2x7_t | (VLEN / 8) * 3.5 | 8 | rvv_vfloat64mf2x7_t | vfloat64mf2x7_t | (VLEN / 8) * 3.5 | 8 | rvv_vint64mf2x8_t | vint64mf2x8_t | (VLEN / 8) * 4 | 8 | rvv_vuint64mf2x8_t | vuint64mf2x8_t | (VLEN / 8) * 4 | 8 | rvv_vfloat64mf2x8_t | vfloat64mf2x8_t | (VLEN / 8) * 4 | 8 | rvv_vint64m1x2_t | vint64m1x2_t | (VLEN / 8) * 2 | 8 | rvv_vuint64m1x2_t | vuint64m1x2_t | (VLEN / 8) * 2 | 8 | rvv_vfloat64m1x2_t | vfloat64m1x2_t | (VLEN / 8) * 2 | 8 | rvv_vint64m1x3_t | vint64m1x3_t | (VLEN / 8) * 3 | 8 | rvv_vuint64m1x3_t | vuint64m1x3_t | (VLEN / 8) * 3 | 8 | rvv_vfloat64m1x3_t | vfloat64m1x3_t | (VLEN / 8) * 3 | 8 | rvv_vint64m1x4_t | vint64m1x4_t | (VLEN / 8) * 4 | 8 | rvv_vuint64m1x4_t | vuint64m1x4_t | (VLEN / 8) * 4 | 8 | rvv_vfloat64m1x4_t | vfloat64m1x4_t | (VLEN / 8) * 4 | 8 | rvv_vint64m1x5_t | vint64m1x5_t | (VLEN / 8) * 5 | 8 |

rvv_vuint64m1x5_t | vuint64m1x5_t | (VLEN / 8) * 5 | 8 | *rvv_vfloat64m1x5_t | vfloat64m1x5_t | (VLEN / 8) * 5 | 8* | rvv_vint64m1x6_t | vint64m1x6_t | (VLEN / 8) * 6 | 8 | *rvv_vuint64m1x6_t | vuint64m1x6_t | (VLEN / 8) * 6 | 8* | rvv_vfloat64m1x6_t | vfloat64m1x6_t | (VLEN / 8) * 6 | 8 | *rvv_vint64m1x7_t | vint64m1x7_t | (VLEN / 8) * 7 | 8* | rvv_vuint64m1x7_t | vuint64m1x7_t | (VLEN / 8) * 7 | 8 | *rvv_vfloat64m1x7_t | vfloat64m1x7_t | (VLEN / 8) * 7 | 8* | rvv_vint64m1x8_t | vint64m1x8_t | (VLEN / 8) * 8 | 8 | *rvv_vuint64m1x8_t | vuint64m1x8_t | (VLEN / 8) * 8 | 8* | rvv_vfloat64m1x8_t | vfloat64m1x8_t | (VLEN / 8) * 8 | 8 | *rvv_vint64m2x2_t | vint64m2x2_t | (VLEN / 8) * 4 | 8* | rvv_vuint64m2x2_t | vuint64m2x2_t | (VLEN / 8) * 4 | 8 | *rvv_vfloat64m2x2_t | vfloat64m2x2_t | (VLEN / 8) * 4 | 8* | rvv_vint64m2x3_t | vint64m2x3_t | (VLEN / 8) * 6 | 8 | *rvv_vuint64m2x3_t | vuint64m2x3_t | (VLEN / 8) * 6 | 8* | rvv_vfloat64m2x3_t | vfloat64m2x3_t | (VLEN / 8) * 6 | 8 | *rvv_vint64m2x4_t | vint64m2x4_t | (VLEN / 8) * 8 | 8* | rvv_vuint64m2x4_t | vuint64m2x4_t | (VLEN / 8) * 8 | 8 | *rvv_vfloat64m2x4_t | vfloat64m2x4_t | (VLEN / 8) * 8 | 8* | rvv_vint64m4x2_t | vint64m4x2_t | (VLEN / 8) * 8 | 8 | *rvv_vuint64m4x2_t | vuint64m4x2_t | (VLEN / 8) * 8 | 8* | rvv_vfloat64m4x2_t | vfloat64m4x2_t | (VLEN / 8) * 8 | 8

> The vector mask types utilize a portion of the space, while the remaining content may be undefined, both in the register and in memory.

> Size must be a positive integer.

Appendix A: Linux-specific ABI

> This section of the RISC-V calling convention specification only applies to Linux-based systems.

In order to ensure compatibility between different implementations of the C library for Linux, we provide some extra definitions which only apply on those systems. These are noted in this section.

A.1. Linux-specific C type sizes and alignments

The following definitions apply for all ABIs defined in this document. Here there is no differentiation between ILP32 and LP64 ABIs.

Table 9. Linux-specific C type sizes and alignments

Type	Size (Bytes)	Alignment (Bytes)
wchar_t	4	4
wint_t	4	4

A.2. Linux-specific C type representations

The following definitions apply for all ABIs defined in this document. Here there is no differentiation between ILP32 and LP64 ABIs.

`wchar_t` is signed. `wint_t` is unsigned.

References

- [riscv-v-extension] "RISC-V V vector extension specification" github.com/riscv/riscv-v-spec
- [rvv-intrinsic-doc] "RISC-V Vector Extension Intrinsic Document" github.com/riscv-non-isa/rvv-intrinsic-doc

RISC-V ELF Specification

Chapter 5. Code models

The RISC-V architecture constrains the addressing of positions in the address space. There is no single instruction that can refer to an arbitrary memory position using a literal as its argument. Rather, instructions exist that, when combined together, can then be used to refer to a memory position via its literal. And, when not, other data structures are used to help the code to address the memory space. The coding conventions governing their use are known as code models.

However, some code models can't access the whole address space. The linker may raise an error if it cannot adjust the instructions to access the target address in the current code model.

5.1. Medium low code model

The medium low code model, or `medlow`, allows the code to address the whole RV32 address space or the lower 2 GiB and highest 2 GiB of the RV64 address space (`0xFFFFFFFF7FFFF800` ~ `0xFFFFFFFFFFFFFFFF` and `0x0` ~ `0x000000007FFFF7FF`). By using the `lui` and load / store instructions, when referring to an object, or `addi`, when calculating an address literal, for example, a 32-bit address literal can be produced.

The following instructions show how to load a value, store a value, or calculate an address in the `medlow` code model.

```
    # Load value from a symbol
    lui  a0, %hi(symbol)
    lw   a0, %lo(symbol)(a0)

    # Store value to a symbol
    lui  a0, %hi(symbol)
    sw   a1, %lo(symbol)(a0)

    # Calculate address
    lui  a0, %hi(symbol)
    addi a0, a0, %lo(symbol)
```

The ranges on RV64 are not `0x0` ~ `0x000000007FFFFFFF` and `0xFFFFFFFF80000000` ~ `0xFFFFFFFFFFFFFFFF` due to RISC-V's sign-extension of immediates; the following code fragments show where the ranges come from:

```
# Largest postive number:
lui a0, 0x7ffff # a0 = 0x7ffff000
addi a0, 0x7ff # a0 = a0 + 2047 = 0x000000007FFFF7FF

# Smallest negative number:
lui a0, 0x80000 # a0 = 0xffffffff80000000
addi a0, a0, -0x800 # a0 = a0 + -2048 = 0xFFFFFFFF7FFFF800
```

5.2. Medium any code model

The medium any code model, or `medany`, allows the code to address the range between -2 GiB and +2 GiB from its position. By using `auipc` and load / store instructions, when referring to an object, or `addi`, when calculating an address literal, for example, a signed 32-bit offset, relative to the value of the `pc` register, can be produced.

As a special edge-case, undefined weak symbols must still be supported, whose addresses will be 0 and may be out of range depending on the address at which the code is linked. Any references to possibly-undefined weak symbols should be made indirectly through the GOT as is used for position-independent code. Not doing so is deprecated and a future version of this specification will require using the GOT, not just advise.

> This is not yet a requirement as existing toolchains predating this part of the specification do not adhere to this, and without improvements to linker relaxation support doing so would regress performance and code size.

The following instructions show how to load a value, store a value, or calculate an address in the medany code model.

```
        # Load value from a symbol
.Ltmp0: auipc a0, %pcrel_hi(symbol)
        lw    a0, %pcrel_lo(.Ltmp0)(a0)

        # Store value to a symbol
.Ltmp1: auipc a0, %pcrel_hi(symbol)
        sw    a1, %pcrel_lo(.Ltmp1)(a0)

        # Calculate address
.Ltmp2: auipc a0, %pcrel_hi(symbol)
        addi  a0, a0, %pcrel_lo(.Ltmp2)
```

> Although the generated code is technically position independent, it's not suitable for ELF shared libraries due to differing symbol interposition rules; for that, please use the medium position independent code model below.

5.3. Medium position independent code model

This model is similar to the medium any code model, but uses the global offset table (GOT) for non-local symbol addresses.

```
        # Load value from a local symbol
.Ltmp0: auipc a0, %pcrel_hi(symbol)
        lw    a0, %pcrel_lo(.Ltmp0)(a0)

        # Store value to a local symbol
.Ltmp1: auipc a0, %pcrel_hi(symbol)
```

```
        sw    a1, %pcrel_lo(.Ltmp1)(a0)

        # Calculate address of a local symbol
.Ltmp2:  auipc a0, %pcrel_hi(symbol)
        addi  a0, a0, %pcrel_lo(.Ltmp2)

        # Calculate address of non-local symbol
.Ltmp3:  auipc  a0, %got_pcrel_hi(symbol)
        l[w|d] a0, a0, %pcrel_lo(.Ltmp3)
```

Chapter 6. Dynamic Linking

Any functions that use registers in a way that is incompatible with the calling convention of the ABI in use must be annotated with `STO_RISCV_VARIANT_CC`, as defined in Section 8.3.

> Vector registers have a variable size depending on the hardware implementation and can be quite large. Saving/restoring all these vector arguments in a run-time linker's lazy resolver would use a large amount of stack space and hurt performance. `STO_RISCV_VARIANT_CC` attribute will require the run-time linker to resolve the symbol directly to prevent saving/restoring any vector registers.

Chapter 7. C++ Name Mangling

C++ name mangling for RISC-V follows the *Itanium C++ ABI* [itanium-cxx-abi]; plus mangling for RISC-V vector data types and vector mask types, which are defined in the following section.

See the "Type encodings" section in *Itanium C++ ABI* for more detail on how to mangle types. Note that `__bf16` is mangled in the same way as `std::bfloat16_t`.

7.1. Name Mangling for Vector Data Types, Vector Mask Types and Vector Tuple Types.

The vector data types and vector mask types, as defined in the section Section 4.5, are treated as vendor-extended types in the *Itanium C++ ABI* [itanium-cxx-abi]. These mangled name for these types is `"u"<len>"rvv_"<type-name>. Specifically, prefixing the type name withrvv_`, which is prefixed by a decimal string indicating its length, which is prefixed by "u".

For example:

```
void foo(vint8m1_t x);
```

is mangled as

```
_Z3foou15__rvv_vint8m1_t
```

```
mangled-name = "u" len "__rvv_" type-name

len = nonzero *DIGIT
nonzero = "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"

type-name = identifier-nondigit *identifier-char
identifier-nondigit = ALPHA / "_"
identifier-char = identifier-nondigit / "_"
```

Chapter 8. ELF Object Files

The ELF object file format for RISC-V follows the *Generic System V Application Binary Interface* [gabi] ("gABI"); this specification only describes RISC-V-specific definitions.

8.1. File Header

The section below lists the defined RISC-V-specific values for several ELF header fields; any fields not listed in this section have no RISC-V-specific values.

e_ident

EI_CLASS

Specifies the base ISA, either RV32 or RV64. Linking RV32 and RV64 code together is not supported.

ELFCLASS64 ELF-64 Object File

ELFCLASS32 ELF-32 Object File

EI_DATA

Specifies the endianness; either big-endian or little-endian. Linking big-endian and little-endian code together is not supported.

ELFDATA2LSB Little-endian Object File

ELFDATA2MSB Big-endian Object File

e_machine

Identifies the machine this ELF file targets. Always contains EM_RISCV (243) for RISC-V ELF files.

e_flags

Describes the format of this ELF file. These flags are used by the linker to disallow linking ELF files with incompatible ABIs together, Table 10 shows the layout of e_flags, and flag details are listed below.

Table 10. Layout of e_flags

Bit 0	Bits 1 - 2	Bit 3	Bit 4	Bits 5 - 23	Bits 24 - 31
RVC	Float ABI	RVE	TSO	**Reserved**	**Non-standard extensions**

EF_RISCV_RVC (0x0001)

This bit is set when the binary targets the C ABI, which allows instructions to be aligned to 16-bit boundaries (the base RV32 and RV64 ISAs only allow 32-bit instruction alignment). When linking objects which specify EF_RISCV_RVC, the linker is permitted to use RVC instructions such as C.JAL in the linker relaxation process.

EF_RISCV_FLOAT_ABI_SOFT (0x0000)

EF_RISCV_FLOAT_ABI_SINGLE (0x0002)

EF_RISCV_FLOAT_ABI_DOUBLE (0x0004)

EF_RISCV_FLOAT_ABI_QUAD (0x0006)

These flags identify the floating point ABI in use for this ELF file. They store the largest floating-point type that ends up in registers as part of the ABI (but do not control if code generation is allowed to use floating-point internally). The rule is that if you have a floating-point type in a register, then you also have all smaller floating-point types in registers. For example _DOUBLE would store "float" and "double" values in F registers, but would not store "long double" values in F registers. If none of the float ABI flags are set, the object is taken to use the soft-float ABI.

EF_RISCV_FLOAT_ABI (0x0006)

This macro is used as a mask to test for one of the above floating-point ABIs, e.g., `(e_flags & EF_RISCV_FLOAT_ABI) == EF_RISCV_FLOAT_ABI_DOUBLE`.

EF_RISCV_RVE (0x0008)

This bit is set when the binary targets the E ABI.

EF_RISCV_TSO (0x0010)

This bit is set when the binary requires the RVTSO memory consistency model.

Until such a time that the **Reserved** bits (0x00ffffe0) are allocated by future versions of this specification, they shall not be set by standard software. Non-standard extensions are free to use bits 24-31 for any purpose. This may conflict with other non-standard extensions.

> There is no provision for compatibility between conflicting uses of the e_flags bits reserved for non-standard extensions, and many standard RISC-V tools will ignore them. Do not use them unless you control both the toolchain and the operating system, and the ABI differences are so significant they cannot be done with a .RISCV.attributes tag nor an ELF note, such as using a different syscall ABI.

==== Policy for Merge Objects With Different File Headers

This section describe the behavior when the inputs files come with different file headers.

`e_ident` and `e_machine` should have exact same value otherwise linker should raise an error.

`e_flags` has different different policy for different fields:

RVC

Input file could have different values for the RVC field; the linker should set this field into EF_RISCV_RVC if any of the input objects has been set.

Float ABI

Linker should report errors if object files of different value for float ABI field.

RVE

Linker should report errors if object files of different value for RVE field.

TSO

Input files can have different values for the TSO field; the linker should set this field if any of the input objects have the TSO field set.

> The static linker may ignore the compatibility checks if all fields in the `e_flags` are zero and all sections in the input file are non-executable sections.

8.2. String Tables

There are no RISC-V specific definitions relating to ELF string tables.

8.3. Symbol Table

st_other

The lower 2 bits are used to specify a symbol's visibility. The remaining 6 bits have no defined meaning in the ELF gABI. We use the highest bit to mark functions that do not follow the standard calling convention for the ABI in use.

The defined processor-specific `st_other` flags are listed in Table 11.

Table 11. RISC-V-specific `st_other` flags

Name	Mask
STO_RISCV_VARIANT_CC	0x80

See Chapter 6 for the meaning of `STO_RISCV_VARIANT_CC`.

`__global_pointer$` must be exported in the dynamic symbol table of dynamically-linked executables if there are any GP-relative accesses present in the executable.

8.4. Relocations

RISC-V is a classical RISC architecture that has densely packed non-word sized instruction immediate values. While the linker can make relocations on arbitrary memory locations, many of the RISC-V relocations are designed for use with specific instructions or instruction sequences. RISC-V has several instruction specific encodings for PC-Relative address loading, jumps, branches and the RVC compressed instruction set.

The purpose of this section is to describe the RISC-V specific instruction sequences with their associated relocations in addition to the general purpose machine word sized relocations that are used for symbol addresses in the Global Offset Table or DWARF meta data.

Table 12 provides details of the RISC-V ELF relocations; the meaning of each column is given below:

Enum

The number of the relocation, encoded in the r_info field

ELF Reloc Type

The name of the relocation, omitting the prefix of `R_RISCV_`.

Type

Whether the relocation is a static or dynamic relocation:

- A static relocation relocates a location in a relocatable file, processed by a static linker.
- A dynamic relocation relocates a location in an executable or shared object, processed by a run-time linker.
- `Both`: Some relocation types are used by both static relocations and dynamic relocations.

Field

Describes the set of bits affected by this relocation; see Section 8.4.2 for the definitions of the individual types

Calculation

Formula for how to resolve the relocation value; definitions of the symbols can be found in Section 8.4.1

Description

Additional information about the relocation

Table 12. Relocation types

Enum	ELF Reloc Type	Type	Field / Calculation	Description
0	NONE	None		
1	32	Both	*word32* S + A	32-bit relocation
2	64	Both	*word64* S + A	64-bit relocation
3	RELATIVE	Dynamic	*wordclass* B + A	Adjust a link address (A) to its load address (B + A)
4	COPY	Dynamic		Must be in executable; not allowed in shared library
5	JUMP_SLOT	Dynamic	*wordclass* S	Indicates the symbol associated with a PLT entry
6	TLS_DTPMOD32	Dynamic	*word32* TLSMODULE	

Enum	ELF Reloc Type	Type	Field / Calculation	Description
7	TLS_DTPMOD64	Dynamic	*word64*	
			TLSMODULE	
8	TLS_DTPREL32	Dynamic	*word32*	
			S + A - TLS_DTV_OFFSET	
9	TLS_DTPREL64	Dynamic	*word64*	
			S + A - TLS_DTV_OFFSET	
10	TLS_TPREL32	Dynamic	*word32*	
			S + A + TLSOFFSET	
11	TLS_TPREL64	Dynamic	*word64*	
			S + A + TLSOFFSET	
12	TLSDESC	Dynamic	See Section 8.5.4	
			TLSDESC(S+A)	
16	BRANCH	Static	*B-Type*	12-bit PC-relative branch offset
			S + A - P	
17	JAL	Static	*J-Type*	20-bit PC-relative jump offset
			S + A - P	
18	CALL	Static	*U+I-Type*	**Deprecated, please use CALL_PLT instead** 32-bit PC-relative function call, macros `call`, `tail`
			S + A - P	
19	CALL_PLT	Static	*U+I-Type*	32-bit PC-relative function call, macros `call`, `tail` (PIC)
			S + A - P	
20	GOT_HI20	Static	*U-Type*	High 20 bits of 32-bit PC-relative GOT access, `%got_pcrel_hi(symbol)`
			G + GOT + A - P	
21	TLS_GOT_HI20	Static	*U-Type*	High 20 bits of 32-bit PC-relative TLS IE GOT access, macro `la.tls.ie`
22	TLS_GD_HI20	Static	*U-Type*	High 20 bits of 32-bit PC-relative TLS GD GOT reference, macro `la.tls.gd`
23	PCREL_HI20	Static	*U-Type*	High 20 bits of 32-bit PC-relative reference, `%pcrel_hi(symbol)`
			S + A - P	
24	PCREL_LO12_I	Static	*I-type*	Low 12 bits of a 32-bit PC-relative, `%pcrel_lo(address of %pcrel_hi)`, the addend must be 0
			S - P	

Enum	ELF Reloc Type	Type	Field / Calculation	Description
25	PCREL_LO12_S	Static	*S-Type* S - P	Low 12 bits of a 32-bit PC-relative, `%pcrel_lo(address of %pcrel_hi)`, the addend must be 0
26	HI20	Static	*U-Type* S + A	High 20 bits of 32-bit absolute address, `%hi(symbol)`
27	LO12_I	Static	*I-Type* S + A	Low 12 bits of 32-bit absolute address, `%lo(symbol)`
28	LO12_S	Static	*S-Type* S + A	Low 12 bits of 32-bit absolute address, `%lo(symbol)`
29	TPREL_HI20	Static	*U-Type*	High 20 bits of TLS LE thread pointer offset, `%tprel_hi(symbol)`
30	TPREL_LO12_I	Static	*I-Type*	Low 12 bits of TLS LE thread pointer offset, `%tprel_lo(symbol)`
31	TPREL_LO12_S	Static	*S-Type*	Low 12 bits of TLS LE thread pointer offset, `%tprel_lo(symbol)`
32	TPREL_ADD	Static		TLS LE thread pointer usage, `%tprel_add(symbol)`
33	ADD8	Static	*word8* V + S + A	8-bit label addition
34	ADD16	Static	*word16* V + S + A	16-bit label addition
35	ADD32	Static	*word32* V + S + A	32-bit label addition
36	ADD64	Static	*word64* V + S + A	64-bit label addition
37	SUB8	Static	*word8* V - S - A	8-bit label subtraction
38	SUB16	Static	*word16* V - S - A	16-bit label subtraction
39	SUB32	Static	*word32* V - S - A	32-bit label subtraction
40	SUB64	Static	*word64* V - S - A	64-bit label subtraction

Enum	ELF Reloc Type	Type	Field / Calculation	Description
41	GOT32_PCREL	Static	*word32* G + GOT + A - P	32-bit difference between the GOT entry for a symbol and the current location
42	**Reserved**	-		Reserved for future standard use
43	ALIGN	Static		Alignment statement. The addend indicates the number of bytes occupied by `nop` instructions at the relocation offset. The alignment boundary is specified by the addend rounded up to the next power of two.
44	RVC_BRANCH	Static	*CB-Type* S + A - P	8-bit PC-relative branch offset
45	RVC_JUMP	Static	*CJ-Type* S + A - P	11-bit PC-relative jump offset
46	**Reserved**	-		Reserved for future standard use
47	GPREL_LO12_I	Static	*I-type* S + A - GP	Low 12 bits of a 32-bit GP-relative address, `%gprel_lo(symbol)`
48	GPREL_LO12_S	Static	*S-Type* S + A - GP	Low 12 bits of a 32-bit GP-relative address, `%gprel_lo(symbol)`
49	GPREL_HI20	Static	*U-Type* S + A - GP	High 20 bits of a 32-bit GP-relative address, `%gprel_hi(symbol)`
50	**Reserved**	-		Reserved for future standard use
51	RELAX	Static		Instruction can be relaxed, paired with a normal relocation at the same address
52	SUB6	Static	*word6* V - S - A	Local label subtraction
53	SET6	Static	*word6* S + A	Local label assignment
54	SET8	Static	*word8* S + A	Local label assignment
55	SET16	Static	*word16* S + A	Local label assignment
56	SET32	Static	*word32* S + A	Local label assignment

Enum	ELF Reloc Type	Type	Field / Calculation	Description
57	32_PCREL	Static	*word32* S + A - P	32-bit PC relative
58	IRELATIVE	Dynamic	*wordclass* `ifunc_resolver(B + A)`	Relocation against a non-preemptible ifunc symbol
59	PLT32	Static	*word32* S + A - P	32-bit relative offset to a function or its PLT entry
60	SET_ULEB128	Static	*ULEB128* S + A	Must be placed immediately before a SUB_ULEB128 with the same offset. Local label assignment *note
61	SUB_ULEB128	Static	*ULEB128* V - S - A	Must be placed immediately after a SET_ULEB128 with the same offset. Local label subtraction *note
62	TLSDESC_HI20	Static	*U-Type* S + A - P	High 20 bits of a 32-bit PC-relative offset into a TLS descriptor entry, `%tlsdesc_hi(symbol)`
63	TLSDESC_LOAD_LO12	Static	*I-Type* S - P	Low 12 bits of a 32-bit PC-relative offset into a TLS descriptor entry, `%tlsdesc_load_lo(address of %tlsdesc_hi)`, the addend must be 0
64	TLSDESC_ADD_LO12	Static	*I-Type* S - P	Low 12 bits of a 32-bit PC-relative offset into a TLS descriptor entry, `%tlsdesc_add_lo(address of %tlsdesc_hi)`, the addend must be 0
65	TLSDESC_CALL	Static		Annotate call to TLS descriptor resolver function, `%tlsdesc_call(address of %tlsdesc_hi)`, for relaxation purposes only
66-191	**Reserved**	-		Reserved for future standard use
192-255	**Reserved**	-		Reserved for nonstandard ABI extensions

Nonstandard extensions are free to use relocation numbers 192-255 for any purpose. These relocations may conflict with other nonstandard extensions.

This section and later ones contain fragments written in assembler. The precise assembler syntax, including that of the relocations, is described in the *RISC-V Assembly Programmer's Manual* [rv-asm].

The assembler must allocate sufficient space to accommodate the final value for the `R_RISCV_SET_ULEB128` and `R_RISCV_SUB_ULEB128` relocation pair and fill the space with a single ULEB128-encoded value. This is achieved by prepending the redundant `0x80` byte as necessary. The linker must not alter the length of the ULEB128-encoded value.

8.4.1. Calculation Symbols

Table 13 provides details on the variables used in relocation calculation:

Table 13. Variables used in relocation calculation

Variable	Description
A	Addend field in the relocation entry associated with the symbol
B	Base address of a shared object loaded into memory
G	Offset of the symbol into the GOT (Global Offset Table)
GOT	Address of the GOT (Global Offset Table)
P	Position of the relocation
S	Value of the symbol in the symbol table
V	Value at the position of the relocation
GP	Value of `__global_pointer$` symbol
TLSMODULE	TLS module index for the object containing the symbol
TLSOFFSET	TLS static block offset (relative to `tp`) for the object containing the symbol

Global Pointer: It is assumed that program startup code will load the value of the `__global_pointer$` symbol into register `gp` (aka `x3`).

8.4.2. Field Symbols

Table 14 provides details on the variables used in relocation fields:

Table 14. Variables used in relocation fields

Variable	Description
word6	Specifies the 6 least significant bits of a *word8* field
word8	Specifies an 8-bit word
word16	Specifies a 16-bit word
word32	Specifies a 32-bit word
word64	Specifies a 64-bit word
ULEB128	Specifies a variable-length data encoded in ULEB128 format.
wordclass	Specifies a *word32* field for ILP32 or a *word64* field for LP64
B-Type	Specifies a field as the immediate field in a B-type instruction

Variable	Description
CB-Type	Specifies a field as the immediate field in a CB-type instruction
CI-Type	Specifies a field as the immediate field in a CI-type instruction
CJ-Type	Specifies a field as the immediate field in a CJ-type instruction
I-Type	Specifies a field as the immediate field in an I-type instruction
S-Type	Specifies a field as the immediate field in an S-type instruction
U-Type	Specifies a field as the immediate field in an U-type instruction
J-Type	Specifies a field as the immediate field in a J-type instruction
U+I-Type	Specifies a field as the immediate fields in a U-type and I-type instruction pair

8.4.3. Constants

Table 15 provides details on the constants used in relocation fields:

Table 15. Constants used in relocation fields

Name	Value
TLS_DTV_OFFSET	0x800

8.4.4. Absolute Addresses

32-bit absolute addresses in position dependent code are loaded with a pair of instructions which have an associated pair of relocations: `R_RISCV_HI20` plus `R_RISCV_LO12_I` or `R_RISCV_LO12_S`.

The `R_RISCV_HI20` refers to an `LUI` instruction containing the high 20-bits to be relocated to an absolute symbol address. The `LUI` instruction is used in conjunction with one or more I-Type instructions (add immediate or load) with `R_RISCV_LO12_I` relocations or S-Type instructions (store) with `R_RISCV_LO12_S` relocations. The addresses for pair of relocations are calculated like this:

HI20 `(symbol_address + 0x800) >> 12`

LO12 `symbol_address`

The following assembly and relocations show loading an absolute address:

```
lui  a0, %hi(symbol)     # R_RISCV_HI20 (symbol)
addi a0, a0, %lo(symbol) # R_RISCV_LO12_I (symbol)
```

A symbol can be loaded in multiple fragments using different addends, where multiple instructions associated with `R_RISCV_LO12_I`/`R_RISCV_LO12_S` share a single `R_RISCV_HI20`. The HI20 values for the multiple fragments must be identical, a condition met when the symbol is sufficiently aligned.

```
lui a0, 0       # R_RISCV_HI20 (symbol)
lw a1, 0(a0)    # R_RISCV_LO12_I (symbol)
```

```
    lw a2, 0(a0)    # R_RISCV_LO12_I (symbol+4)
    lw a3, 0(a0)    # R_RISCV_LO12_I (symbol+8)
    lw a0, 0(a0)    # R_RISCV_LO12_I (symbol+12)
```

8.4.5. Global Offset Table

For position independent code in dynamically linked objects, each shared object contains a GOT (Global Offset Table), which contains addresses of global symbols (objects and functions) referred to by the dynamically linked shared object. The GOT in each shared library is filled in by the dynamic linker during program loading, or on the first call to extern functions.

To avoid dynamic relocations within the text segment of position independent code the GOT is used for indirection. Instead of code loading virtual addresses directly, as can be done in static code, addresses are loaded from the GOT. This allows runtime binding to external objects and functions at the expense of a slightly higher runtime overhead for access to extern objects and functions.

8.4.6. Procedure Linkage Table

The PLT (Procedure Linkage Table) exists to allow function calls between dynamically linked shared objects. Each dynamic object has its own GOT (Global Offset Table) and PLT (Procedure Linkage Table).

The first entry of a shared object PLT is a special entry that calls `_dl_runtime_resolve` to resolve the GOT offset for the called function. The `_dl_runtime_resolve` function in the dynamic loader resolves the GOT offsets lazily on the first call to any function, except when `LD_BIND_NOW` is set in which case the GOT entries are populated by the dynamic linker before the executable is started. Lazy resolution of GOT entries is intended to speed up program loading by deferring symbol resolution to the first time the function is called. The first entry in the PLT occupies two 16 byte entries:

```
1:  auipc  t2, %pcrel_hi(.got.plt)
    sub    t1, t1, t3               # shifted .got.plt offset + hdr size + 12
    l[w|d] t3, %pcrel_lo(1b)(t2)    # _dl_runtime_resolve
    addi   t1, t1, -(hdr size + 12) # shifted .got.plt offset
    addi   t0, t2, %pcrel_lo(1b)    # &.got.plt
    srli   t1, t1, log2(16/PTRSIZE) # .got.plt offset
    l[w|d] t0, PTRSIZE(t0)          # link map
    jr     t3
```

Subsequent function entry stubs in the PLT take up 16 bytes and load a function pointer from the GOT. On the first call to a function, the entry redirects to the first PLT entry which calls `_dl_runtime_resolve` and fills in the GOT entry for subsequent calls to the function:

```
1:  auipc   t3, %pcrel_hi(function@.got.plt)
    l[w|d]  t3, %pcrel_lo(1b)(t3)
    jalr    t1, t3
    nop
```

8.4.7. Procedure Calls

R_RISCV_CALL and R_RISCV_CALL_PLT relocations are associated with pairs of instructions (AUIPC+JALR) generated by the CALL or TAIL pseudoinstructions. Originally, these relocations had slightly different behavior, but that has turned out to be unnecessary, and they are now interchangeable, R_RISCV_CALL is deprecated, suggest using R_RISCV_CALL_PLT instead.

With linker relaxation enabled, the AUIPC instruction in the AUIPC+JALR pair has both a R_RISCV_CALL or R_RISCV_CALL_PLT relocation and an R_RISCV_RELAX relocation indicating the instruction sequence can be relaxed during linking.

Procedure call linker relaxation allows the AUIPC+JALR pair to be relaxed to the JAL instruction when the procedure or PLT entry is within (-1MiB to +1MiB-2) of the instruction pair.

The pseudoinstruction:

```
call symbol
call symbol@plt
```

expands to the following assembly and relocation:

```
auipc ra, 0          # R_RISCV_CALL (symbol), R_RISCV_RELAX (symbol)
jalr  ra, ra, 0
```

and when symbol has an @plt suffix it expands to:

```
auipc ra, 0          # R_RISCV_CALL_PLT (symbol), R_RISCV_RELAX (symbol)
jalr  ra, ra, 0
```

8.4.8. PC-Relative Jumps and Branches

Unconditional jump (J-Type) instructions have a R_RISCV_JAL relocation that can represent an even signed 21-bit offset (-1MiB to +1MiB-2).

Branch (SB-Type) instructions have a R_RISCV_BRANCH relocation that can represent an even signed 13-bit offset (-4096 to +4094).

8.4.9. PC-Relative Symbol Addresses

32-bit PC-relative relocations for symbol addresses on sequences of instructions such as the AUIPC+ADDI instruction pair expanded from the la pseudoinstruction, in position independent code typically have an associated pair of relocations: R_RISCV_PCREL_HI20 plus R_RISCV_PCREL_LO12_I or R_RISCV_PCREL_LO12_S.

The R_RISCV_PCREL_HI20 relocation refers to an AUIPC instruction containing the high 20-bits to be relocated to a symbol relative to the program counter address of the AUIPC instruction. The AUIPC instruction is used in conjunction with one or more I-Type instructions (add immediate or load) with R_RISCV_PCREL_LO12_I relocations or S-Type instructions (store) with R_RISCV_PCREL_LO12_S relocations.

The R_RISCV_PCREL_LO12_I or R_RISCV_PCREL_LO12_S relocations contain a label pointing to an instruction in the same section with an R_RISCV_PCREL_HI20 relocation entry that points to the target symbol:

- At label: R_RISCV_PCREL_HI20 relocation entry → symbol
- R_RISCV_PCREL_LO12_I relocation entry → label

To get the symbol address to perform the calculation to fill the 12-bit immediate on the add, load or store instruction the linker finds the R_RISCV_PCREL_HI20 relocation entry associated with the AUIPC instruction. The addresses for pair of relocations are calculated like this:

HI20 `(symbol_address - hi20_reloc_offset + 0x800) >> 12`

LO12 `symbol_address - hi20_reloc_offset`

The successive instruction has a signed 12-bit immediate so the value of the preceding high 20-bit relocation may have 1 added to it.

Note the compiler emitted instructions for PC-relative symbol addresses are not necessarily sequential or in pairs. There is a constraint is that the instruction with the R_RISCV_PCREL_LO12_I or R_RISCV_PCREL_LO12_S relocation label points to a valid HI20 PC-relative relocation pointing to the symbol.

Here is example assembler showing the relocation types:

```
label:
    auipc t0, %pcrel_hi(symbol)   # R_RISCV_PCREL_HI20 (symbol)
    lui t1, 1
    lw t2, t0, %pcrel_lo(label)   # R_RISCV_PCREL_LO12_I (label)
    add t2, t2, t1
    sw t2, t0, %pcrel_lo(label)   # R_RISCV_PCREL_LO12_S (label)
```

8.4.10. Relocation for Alignment

The relocation type R_RISCV_ALIGN marks a location that must be aligned to N-bytes, where N is the smallest power of two that is greater than the value of the addend field, e.g. R_RISCV_ALIGN with addend value 2 means align to 4 bytes, R_RISCV_ALIGN with addend value 4 means align to 8 bytes; this relocation is only required if the containing section has any R_RISCV_RELAX relocations, R_RISCV_ALIGN points to the beginning of the padding bytes, and the instruction that actually needs to be aligned is located at the point of R_RISCV_ALIGN plus its addend.

To ensure the linker can always satisfy the required alignment solely by deleting bytes, the compiler or assembler must emit a R_RISCV_ALIGN relocation and then insert N - [IALIGN] padding bytes before the location where we need to align, it could be mark by an alignment directive like .align, .p2align or .balign or emit by compiler directly, the addend value of that relocation is the number of padding bytes.

The compiler and assembler must ensure padding bytes are valid instructions without any side-effect like `nop` or `c.nop`, and make sure those instructions are aligned to IALIGN if possible.

The linker may remove part of the padding bytes at the linking process to meet the alignment requirement, and must make sure those padding bytes still are valid instructions and each instruction is aligned to at least IALIGN byte.

Here is example to showing how `R_RISCV_ALIGN` is used:

```
0x0    c.nop           # R_RISCV_ALIGN with addend 2
0x2    add t1, t2, t3  # This instruction must align to 4 byte.
```

`R_RISCV_ALIGN` relocation is needed because linker relaxation can shrink preceding code during the linking process, which may cause an aligned location to become mis-aligned.

IALIGN means the instruction-address alignment constraint. IALIGN is 4 bytes in the base ISA, but some ISA extensions, including the compressed ISA extension, relax IALIGN to 2 bytes. IALIGN may not take on any value other than 4 or 2. This term is also defined in `The RISC-V Instruction Set Manual` with a similar meaning, the only difference being it is specified in terms of the number of bits instead of the number of bytes.

Here is pseudocode to decide the alignment of `R_RISCV_ALIGN` relocation:

```
# input:
#   addend: addend value of relocation with R_RISCV_ALIGN type.
# output:
#   Alignment of this relocation.

def align(addend):
  ALIGN = 1
  while addend >= ALIGN:
    ALIGN *= 2
  return ALIGN
```

8.5. Thread Local Storage

RISC-V adopts the ELF Thread Local Storage Model in which ELF objects define `.tbss` and `.tdata` sections and `PT_TLS` program headers that contain the TLS "initialization images" for new threads. The `.tbss` and `.tdata` sections are not referenced directly like regular segments, rather they are copied or allocated to the thread local storage space of newly created threads. See *ELF Handling For Thread-Local Storage* [tls].

In The ELF Thread Local Storage Model, TLS offsets are used instead of pointers. The ELF TLS sections are initialization images for the thread local variables of each new thread. A TLS offset defines an offset into the dynamic thread vector which is pointed to by the TCB (Thread Control Block). RISC-V uses Variant I as described by the ELF TLS specification, with `tp` containing the address one past the end of the TCB.

There are various thread local storage models for statically allocated or dynamically allocated thread local storage. Table 16 lists the thread local storage models:

Table 16. TLS models

Mnemonic	Model
TLS LE	Local Exec
TLS IE	Initial Exec
TLS LD	Local Dynamic
TLS GD	Global Dynamic

The program linker in the case of static TLS or the dynamic linker in the case of dynamic TLS allocate TLS offsets for storage of thread local variables.

`Global Dynamic` model is also known as `General Dynamic` model.

8.5.1. Local Exec

Local exec is a form of static thread local storage. This model is used when static linking as the TLS offsets are resolved during program linking.

Variable attribute

`__thread int i __attribute__((tls_model("local-exec")));`

Example assembler load and store of a thread local variable `i` using the `%tprel_hi`, `%tprel_add` and `%tprel_lo` assembler functions. The emitted relocations are in comments.

```
    lui  a5,%tprel_hi(i)           # R_RISCV_TPREL_HI20 (symbol)
    add  a5,a5,tp,%tprel_add(i)    # R_RISCV_TPREL_ADD (symbol)
    lw   t0,%tprel_lo(i)(a5)       # R_RISCV_TPREL_LO12_I (symbol)
    addi t0,t0,1
    sw   t0,%tprel_lo(i)(a5)       # R_RISCV_TPREL_LO12_S (symbol)
```

The `%tprel_add` assembler function does not return a value and is used purely to associate the `R_RISCV_TPREL_ADD` relocation with the `add` instruction.

8.5.2. Initial Exec

Initial exec is is a form of static thread local storage that can be used in shared libraries that use thread local storage. TLS relocations are performed at load time. `dlopen` calls to libraries that use thread local storage may fail when using the initial exec thread local storage model as TLS offsets must all be resolved at load time. This model uses the GOT to resolve TLS offsets.

Variable attribute

`__thread int i __attribute__((tls_model("initial-exec")));`

ELF flags

DF_STATIC_TLS

Example assembler load and store of a thread local variable `i` using the `la.tls.ie` pseudoinstruction, with the emitted TLS relocations in comments:

```
    la.tls.ie a5,i
    add  a5,a5,tp
    lw   t0,0(a5)
    addi t0,t0,1
    sw   t0,0(a5)
```

The assembler pseudoinstruction:

```
    la.tls.ie a5,symbol
```

expands to the following assembly instructions and relocations:

```
label:
    auipc a5, 0                  # R_RISCV_TLS_GOT_HI20 (symbol)
    {ld,lw} a5, 0(a5)            # R_RISCV_PCREL_LO12_I (label)
```

8.5.3. Global Dynamic

RISC-V local dynamic and global dynamic TLS models generate equivalent object code. The Global dynamic thread local storage model is used for PIC Shared libraries and handles the case where more than one library uses thread local variables, and additionally allows libraries to be loaded and unloaded at runtime using `dlopen`. In the global dynamic model, application code calls the dynamic linker function `__tls_get_addr` to locate TLS offsets into the dynamic thread vector at runtime.

Variable attribute

`__thread int i __attribute__((tls_model("global-dynamic")));`

Example assembler load and store of a thread local variable `i` using the `la.tls.gd` pseudoinstruction, with the emitted TLS relocations in comments:

```
    la.tls.gd a0,i
    call  __tls_get_addr@plt
    mv   a5,a0
    lw   t0,0(a5)
    addi t0,t0,1
```

```
    sw   t0,0(a5)
```

The assembler pseudoinstruction:

```
    la.tls.gd a0,symbol
```

expands to the following assembly instructions and relocations:

```
label:
    auipc a0,0                    # R_RISCV_TLS_GD_HI20 (symbol)
    addi  a0,a0,0                 # R_RISCV_PCREL_LO12_I (label)
```

In the Global Dynamic model, the runtime library provides the `__tls_get_addr` function:

```
extern void *__tls_get_addr (tls_index *ti);
```

where the type tls_index is defined as:

```
typedef struct
{
  unsigned long int ti_module;
  unsigned long int ti_offset;
} tls_index;
```

8.5.4. TLS Descriptors

TLS Descriptors (TLSDESC) are an alternative implementation of the Global Dynamic model that allows the dynamic linker to achieve performance close to that of Initial Exec when the library was not loaded dynamically with `dlopen`.

The linker reserves a consecutive pair of pointer-sized entry in the GOT for each `TLSDESC` relocation. At runtime, the dynamic linker fills in the TLS descriptor entry as defined below:

```
typedef struct
{
  unsigned long (*entry)(tls_descriptor *);
  unsigned long arg;
} tls_descriptor;
```

Upon accessing the thread local variable, the `entry` function is called with the address of `tls_descriptor` containing it, returning `<address of thread local variable> - tp`.

The TLS descriptor `entry` is called with a special calling convention, specified as follows:

- `a0` is used to pass the argument and return value.
- `t0` is used as the link register.
- Any other registers are callee-saved. This includes any vector registers when the vector extension is supported.

Example assembler load and store of a thread local variable `i` using the `%tlsdesc_hi`, `%tlsdesc_load_lo`, `%tlsdesc_add_lo` and `%tlsdesc_call` assembler functions. The emitted relocations are in the comments.

```
label:
    auipc tX, %tlsdesc_hi(symbol)         // R_RISCV_TLSDESC_HI20 (symbol)
    lw    tY, tX, %tlsdesc_load_lo(label) // R_RISCV_TLSDESC_LOAD_LO12 (label)
    addi  a0, tX, %tlsdesc_add_lo(label)  // R_RISCV_TLSDESC_ADD_LO12 (label)
    jalr  t0, tY, %tlsdesc_call(label)    // R_RISCV_TLSDESC_CALL (label)
```

`tX` and `tY` in the example may be replaced with any combination of two general purpose registers.

The `%tlsdesc_call` assembler function does not return a value and is used purely to associate the `R_RISCV_TLSDESC_CALL` relocation with the `jalr` instruction.

The linker can use the relocations to recognize the sequence and to perform relaxations. To ensure correctness, only the following changes to the sequence are allowed:

- Instructions outside the sequence that do not clobber the registers used within the sequence may be inserted in-between the instructions of the sequence (known as instruction scheduling).
- Instructions in the sequence with no data dependency may be reordered. In the preceding example, the only instructions that can be reordered are `lw` and `addi`.

8.6. Sections

8.6.1. Section Types

The defined processor-specific section types are listed in Table 17.

Table 17. RISC-V-specific section types

Name	Value	Attributes
SHT_RISCV_ATTRIBUTES	0x70000003	none

8.6.2. Special Sections

Table 18 lists the special sections defined by this ABI.

Table 18. RISC-V-specific sections

Name	Type	Attributes
.riscv.attributes	SHT_RISCV_ATTRIBUTES	none
.riscv.jvt	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

.riscv.attributes names a section that contains RISC-V ELF attributes.

.riscv.jvt is a linker-created section to store table jump target addresses. The minimum alignment of this section is 64 bytes.

8.7. Program Header Table

The defined processor-specific segment types are listed in Table 19.

Table 19. RISC-V-specific segment types

Name	Value	Meaning
PT_RISCV_ATTRIBUTES	0x70000003	RISC-V ELF attribute section.

`PT_RISCV_ATTRIBUTES` describes the location of RISC-V ELF attribute section.

8.8. Note Sections

There are no RISC-V specific definitions relating to ELF note sections.

8.9. Dynamic Section

The defined processor-specific dynamic array tags are listed in Table 20.

Table 20. RISC-V-specific dynamic array tags

Name	Value	d_un	Executable	Shared Object
DT_RISCV_VARIANT_CC	0x70000001	d_val	Platform specific	Platform specific

An object must have the dynamic tag `DT_RISCV_VARIANT_CC` if it has one or more `R_RISCV_JUMP_SLOT` relocations against symbols with the `STO_RISCV_VARIANT_CC` attribute.

`DT_INIT` and `DT_FINI` are not required to be supported and should be avoided in favour of `DT_PREINIT_ARRAY`, `DT_INIT_ARRAY` and `DT_FINI_ARRAY`.

8.10. Hash Table

There are no RISC-V specific definitions relating to ELF hash tables.

8.11. Attributes

Attributes are used to record information about an object file/binary that a linker or runtime loader needs to check compatibility.

Attributes are encoded in a vendor-specific section of type SHT_RISCV_ATTRIBUTES and name .riscv.attributes. The value of an attribute can hold an integer encoded in the uleb128 format or a null-terminated byte string (NTBS). The tag number is also encoded as uleb128.

In order to improve the compatibility of the tool, the attribute follows below rules:

- RISC-V attributes have a string value if the tag number is odd and an integer value if the tag number is even.
- The tag is mandatory; If the tool does not recognize this attribute and the tag number modulo 128 is less than 64 (`(N % 128) < 64`), errors should be reported.
- The tag is optional; If the tool does not recognize this attribute and the tag number modulo 128 is greater than or equal to 64 (`(N % 128) >= 64`), the tag can be ignored.

8.11.1. Layout of .riscv.attributes section

The attributes section start with a format-version (uint8 = 'A') followed by vendor specific sub-section(s). A sub-section starts with sub-section length (uint32), vendor name (NTBS) and one or more sub-sub-section(s).

A sub-sub-section consists of a tag (uleb128), sub-sub-section length (uint32) followed by actual attribute tag,value pair(s) as specified above. Sub-sub-section Tag Tag_file (value 1) specifies that contained attibutes relate to whole file.

A sub-section with name "riscv\0" is mandatory. Vendor specific sub-sections are allowed in future. Vendor names starting with "[Aa]non" are reserved for non-standard ABI extensions.

8.11.2. List of attributes

Table 21. RISC-V attributes

Tag	Value	Parameter type	Description
Tag_RISCV_stack_align	4	uleb128	Indicates the stack alignment requirement in bytes.
Tag_RISCV_arch	5	NTBS	Indicates the target architecture of this object.
Tag_RISCV_unaligned_access	6	uleb128	Indicates whether to impose unaligned memory accesses in code generation.
Tag_RISCV_priv_spec	8	uleb128	**Deprecated**, indicates the major version of the privileged specification.

Tag	Value	Parameter type	Description
Tag_RISCV_priv_spec_minor	10	uleb128	**Deprecated**, indicates the minor version of the privileged specification.
Tag_RISCV_priv_spec_revision	12	uleb128	**Deprecated**, indicates the revision version of the privileged specification.
Tag_RISCV_atomic_abi	14	uleb128	Indicates which version of the atomics ABI is being used.
Tag_RISCV_x3_reg_usage	16	uleb128	Indicates the usage definition of the X3 register.
Reserved for non-standard attribute	>= 32768	-	-

8.11.3. Detailed attribute description

How does this specification describe public attributes?

Each attribute is described in the following structure: <Tag name>, <Value>, <Parameter type 1>=<Parameter name 1>[, <Parameter type 2>=<Parameter name 2>]

Tag_RISCV_stack_align, 4, uleb128=value

Tag_RISCV_stack_align records the N-byte stack alignment for this object. The default value is 16 for RV32I or RV64I, and 4 for RV32E.

Merge Policy

The linker should report erros if link object files with different `Tag_RISCV_stack_align` values.

Tag_RISCV_arch, 5, NTBS=subarch

Tag_RISCV_arch contains a string for the target architecture taken from the option `-march`. Different architectures will be integrated into a superset when object files are merged.

Tag_RISCV_arch should be recorded in lowercase, and all extensions should be separated by underline(`_`).

Note that the version information for target architecture must be presented explicitly in the attribute and abbreviations must be expanded. The version information, if not given by `-march`, must agree with the default specified by the tool. For example, the architecture `rv32i` has to be recorded in the attribute as `rv32i2p1` in which `2p1` stands for the default version of its based ISA. On the other hand, the architecture `rv32g` has to be presented as `rv32i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zifencei2p0` in which the abbreviation `g` is expanded to the `imafd_zicsr_zifencei` combination with default versions of the standard extensions.

The toolchain should normalized the architecture string into canonical order which defined in *The*

RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document [riscv-unpriv] , expanded with all required extension and should add shorthand extension into architecture string if all expanded extensions are included in architecture string.

> A shorthand extension is an extension that does not define any actual instructions, registers or behavior, but requires other extensions, such as the `zks` extension, which is defined in the cryptographic extension, `zks` extension is shorthand for `zbkb`, `zbkc`, `zbkx`, `zksed` and `zksh`, so the toolchain should normalize `rv32i_zbkb_zbkc_zbkx_zksed_zksh` to `rv32i_zbkb_zbkc_zbkx_zks_zksed_zksh`; `g` is an exception and does not apply to this rule.

Merge Policy

The linker should merge the different architectures into a superset when object files are merged, and should report errors if the merge result contains conflict extensions.

This specification does not mandate rules on how to merge ISA strings that refer to different versions of the same ISA extension. The suggested merge rules are as follows:

- Merge versions into the latest version of all input versions that are ratified without warning or error.
- The linker should emit a warning or error if input versions have different versions and any extension versions are not ratified.
- The linker may report a warning or error if it detects incompatible versions, even if it's ratified.

> Example of conflicting merge result: `RV32IF` and `RV32IZfinx` will be merged into `RV32IFZfinx`, which is an invalid architecture since `F` and `Zfinx` conflict.

Tag_RISCV_unaligned_access, 6, uleb128=value

Tag_RISCV_unaligned_access denotes the code generation policy for this object file. Its values are defined as follows:

0 This object does not perform any unaligned memory accesses.

1 This object may perform unaligned memory accesses.

Merge policy

Input file could have different values for the Tag_RISCV_unaligned_access; the linker should set this field into 1 if any of the input objects has been set.

Tag_RISCV_priv_spec, 8, uleb128=version

Tag_RISCV_priv_spec_minor, 10, uleb128=version

Tag_RISCV_priv_spec_revision, 12, uleb128=version

> Those three attributes are deprecated since RISC-V using extensions with version

rather than a single privileged specification version scheme for privileged ISA.

Tag_RISCV_priv_spec contains the major/minor/revision version information of the privileged specification.

Merge policy

The linker should report errors if object files of different privileged specification versions are merged.

Tag_RISCV_atomic_abi, 14, uleb128=version

Tag_RISCV_atomic_abi denotes the atomic ABI used within this object file. Its values are defined as follows:

Value	Symbolic Name	Description
0	UNKNOWN	This object uses unknown atomic ABI.
1	A6C	This object uses the A6 classical atomic ABI, which is defined in table A.6 in [riscv-unpriv-20191213].
2	A6S	This object uses the strengthened A6 ABI, which uses the atomic mapping defined by [Mappings from C/C++ primitives to RISC-V primitives] and does not rely on any note 3 annotated mappings.
3	A7	This object uses the A7 atomic ABI, which uses the atomic mapping defined by [Mappings from C/C++ primitives to RISC-V primitives] and may rely on note 3 annotated mappings.

Merge policy

The linker should report errors if object files with incompatible atomics ABIs are merged; the compatibility rules for atomic ABIs can be found in the compatibility column in the following table.

Input Values	Compatible?	Ouput Value
UNKNOWN and A6C	Yes	A6C
UNKNOWN and A6S	Yes	A6S
UNKNOWN and A7	Yes	A7
A6C and A6S	Yes	A6C
A6C and A7	No	-
A6S and A7	Yes	A7

Merging object files with the same ABI will result in the same ABI.

Programs that implement atomic operations without relying on the A-extension are classified as UNKNOWN for now. A new value for those may be defined in the future.

Tag_RISCV_x3_reg_usage, 16, uleb128=value

Tag_RISCV_x3_reg_usage indicates the usage of `x3`/`gp` register. `x3`/`gp` could be used for global pointer relaxation, as a reserved platform register, or as a temporary register.

0 — This object uses `x3` as a fixed register with unknown purpose.

1 — This object uses `x3` as the global pointer, for relaxation purposes.

2 — This object uses `x3` as the shadow stack pointer.

3 — This object uses `X3` as a temporary register.

4~1023 — Reserved for future standard defined platform register.

1024~2047 — Reserved for nonstandard defined platform register.

Merge policy

The linker should issue errors when object files with differing `gp` usage are combined. However, an exception exists: the value `0` can merge with `1` or `2` value. After the merge, the resulting value will be the non-zero one.

8.12. Mapping Symbol

The section can have a mixture of code and data or code with different ISAs. A number of symbols, named mapping symbols, describe the boundaries.

Symbol Name	Meaning
$d	Start of a sequence of data.
$d.<any>	
$x	Start of a sequence of instructions.
$x.<any>	
$x<ISA>	Start of a sequence of instructions with <ISA> extension.
$x<ISA>.<any>	

The mapping symbol should set the type to `STT_NOTYPE`, binding to `STB_LOCAL`, and the size of symbol to zero.

The mapping symbol for data(`$d`) indicates the start of a sequence of data bytes.

The mapping symbol for instruction(`$x`) indicates the start of a sequence of instructions. and it has an optional ISA string, which means the following code regions are using ISA is different than the ISA recorded in the arch attribute; the ISA information will used until the next instruction mapping symbol; an instruction mapping symbol without ISA string means using ISA configuration from ELF attribute.

Format and rule of the optional ISA string are same as `Tag_RISCV_arch`, must having explicit version, more detailed rule please refer to Section 8.11.

The mapping symbol can be followed by an optional uniquifier, which is prefixed with a dot (`.`).

> The use case for mapping symbol for instruction(`$x`) with ISA information is used with ifunc, e.g. libraries are built with `rv64gc`, but few functions like memcpy provides two versions, one built with `rv64gc`, and one built with `rv64gcv`, and select by ifunc mechanism at run-time; however, the arch attribute is recording for minimal execution environment requirements, so the ISA information from arch attribute is not enough for the disassembler to disassemble the `rv64gcv` version correctly.

Chapter 9. Linker Relaxation

At link time, when all the memory objects have been resolved, the code sequence used to refer to them may be simplified and optimized by the linker by relaxing some assumptions about the memory layout made at compile time.

Some relocation types, in certain situations, indicate to the linker where this can happen. Additionally, some relocation types indicate to the linker the associated parts of a code sequence that can be thusly simplified, rather than to instruct the linker how to apply a relocation.

The linker should only perform such relaxations when a R_RISCV_RELAX relocation is at the same position as a candidate relocation.

As this transformation may delete bytes (and thus invalidate references that are commonly resolved at compile-time, such as intra-function jumps), code generators must in general ensure that relocations are always emitted when relaxation is enabled.

Linkers should adjust relocations that refer to symbols whose addresses have been updated.

ULEB128 value with relocation must be padding to the same length even if the data can be encoded with a shorter byte sequence after linker relaxation, The linker should report errors if the length of ULEB128 byte sequence is more extended than the current byte sequence.

9.1. Linker Relaxation Types

The purpose of this section is to describe all types of linker relaxation, the linker may implement a part of linker relaxation type, and can be skipped the relaxation type is unsupported.

Each candidate relocation might fit more than one relaxation type, the linker should only apply one relaxation type.

In the linker relaxation optimization, we introduce a concept called relocation group; a relocation group consists of 1) relocations associated with the same target symbol and can be applied with the same relaxation, or 2) relocations with the linkage relationship (e.g. `R_RISCV_PCREL_LO12_S` linked with a `R_RISCV_PCREL_HI20`); all relocations in a single group must be present in the same section, otherwise will split into another relocation group.

Every relocation group must apply the same relaxation type, and the linker should not apply linker relaxation to only part of the relocation group.

> Applying relaxation on the part of the relocation group might result in a wrong execution result; for example, a relocation group consists of `lui t0, 0 # R_RISCV_HI20 (foo)`, `lw t1, 0(t0) # R_RISCV_LO12_I (foo)`, and we only apply global pointer relaxation on first instruction, then remove that instruction, and didn't apply relaxation on the second instruction, which made the load instruction reference to an unspecified address.

9.1.1. Function Call Relaxation

Target Relocation

R_RISCV_CALL, R_RISCV_CALL_PLT.

Description

This relaxation type can relax `AUIPC+JALR` into `JAL`.

Condition

The offset between the location of relocation and target symbol or the PLT stub of the target symbol is within +-1MiB.

Relaxation

- Instruction sequence associated with `R_RISCV_CALL` or `R_RISCV_CALL_PLT` can be rewritten to a single JAL instruction with the offset between the location of relocation and target symbol.

Example

Relaxation candidate:

```
auipc ra, 0          # R_RISCV_CALL_PLT (symbol), R_RISCV_RELAX
jalr  ra, ra, 0
```

Relaxation result:

```
jal  ra, 0           # R_RISCV_JAL (symbol)
```

Using address of PLT stubs of the target symbol or address target symbol directly will resolve by linker according to the visibility of the target symbol.

9.1.2. Compressed Function Call Relaxation

Target Relocation

R_RISCV_CALL, R_RISCV_CALL_PLT.

Description

This relaxation type can relax `AUIPC+JALR` into `C.JAL` instruction sequence.

Condition

The offset between the location of relocation and target symbol or the PLT stub of the target symbol is within +-2KiB and rd operand of second instruction in the instruction sequence is `X1`/`RA` and if it is RV32.

Relaxation

- Instruction sequence associated with `R_RISCV_CALL` or `R_RISCV_CALL_PLT` can be rewritten to a single `C.JAL` instruction with the offset between the location of relocation and target symbol.

Example

Relaxation candidate:

```
auipc ra, 0          # R_RISCV_CALL_PLT (symbol), R_RISCV_RELAX
jalr  ra, ra, 0
```

Relaxation result:

```
c.jal  ra, <offset-between-pc-and-symbol>
```

9.1.3. Compressed Tail Call Relaxation

Target Relocation

R_RISCV_CALL, R_RISCV_CALL_PLT.

Description

This relaxation type can relax `AUIPC+JALR` into `C.J` instruction sequence.

Condition

The offset between the location of relocation and target symbol or the PLT stub of the target symbol is within +-2KiB and rd operand of second instruction in the instruction sequence is `X0`.

Relaxation

- Instruction sequence associated with `R_RISCV_CALL` or `R_RISCV_CALL_PLT` can be rewritten to a single `C.J` instruction with the offset between the location of relocation and target symbol.

Example

Relaxation candidate:

```
auipc ra, 0          # R_RISCV_CALL_PLT (symbol), R_RISCV_RELAX
jalr  x0, ra, 0
```

Relaxation result:

```
c.j  ra, <offset-between-pc-and-symbol>
```

9.1.4. Global-pointer Relaxation

Target Relocation

R_RISCV_HI20, R_RISCV_LO12_I, R_RISCV_LO12_S, R_RISCV_PCREL_HI20, R_RISCV_PCREL_LO12_I, R_RISCV_PCREL_LO12_S

Description

This relaxation type can relax a sequence of the load address of a symbol or load/store with a symbol reference into global-pointer-relative instruction.

Condition

Global-pointer relaxation requires that Tag_RISCV_x3_reg_usage must be 0 or 1, and offset between global-pointer and symbol is within +-2KiB, `R_RISCV_PCREL_LO12_I` and `R_RISCV_PCREL_LO12_S` resolved as indirect relocation pointer. It will always point to another `R_RISCV_PCREL_HI20` relocation, the symbol pointed by `R_RISCV_PCREL_HI20` will be used in the offset calculation.

Relaxation

- Instruction associated with `R_RISCV_HI20` or `R_RISCV_PCREL_HI20` can be removed.
- Instruction associated with `R_RISCV_LO12_I`, `R_RISCV_LO12_S`, `R_RISCV_PCREL_LO12_I` or `R_RISCV_PCREL_LO12_S` can be replaced with a global-pointer-relative access instruction.

Example

Relaxation candidate (`tX` and `tY` can be any combination of two general purpose registers):

```
lui tX, 0       # R_RISCV_HI20 (symbol), R_RISCV_RELAX
lw tY, 0(tX)    # R_RISCV_LO12_I (symbol), R_RISCV_RELAX
```

Relaxation result:

```
lw tY, <gp-offset-for-symbol>(gp)
```

A symbol can be loaded in multiple fragments using different addends, where multiple instructions associated with `R_RISCV_LO12_I`/`R_RISCV_LO12_S` share a single `R_RISCV_HI20`. The HI20 values for the multiple fragments must be identical and all the relaxed global-pointer offsets must be in range.

Relaxation candidate:

```
lui tX, 0       # R_RISCV_HI20 (symbol), R_RISCV_RELAX
lw tY, 0(tX)    # R_RISCV_LO12_I (symbol), R_RISCV_RELAX
lw tZ, 0(tX+4)  # R_RISCV_LO12_I (symbol+4), R_RISCV_RELAX
lw tW, 0(tX+8)  # R_RISCV_LO12_I (symbol+8), R_RISCV_RELAX
lw tX, 0(tX+12) # R_RISCV_LO12_I (symbol+12), R_RISCV_RELAX
```

Relaxation result:

```
lw tY, <gp-offset-for-symbol>(gp)
lw tZ, <gp-offset-for-symbol+4>(gp)
lw tW, <gp-offset-for-symbol+8>(gp)
lw tX, <gp-offset-for-symbol+12>(gp)
```

> The global-pointer refers to the address of the `__global_pointer$` symbol, which is the content of `gp` register.

> This relaxation requires the program to initialize the `gp` register with the address of `__global_pointer$` symbol before accessing any symbol address, strongly recommended initialize `gp` at the beginning of the program entry function like `_start`, and code fragments of initialization must disable linker relaxation to prevent initialization instruction relaxed into a NOP-like instruction (e.g. `mv gp, gp`).

```
    # Recommended way to initialize the gp register.
    .option push
    .option norelax
1:  auipc gp, %pcrel_hi(__global_pointer$)
    addi  gp, gp, %pcrel_lo(1b)
    .option pop
```

> The global pointer is referred to as the global offset table pointer in many other targets, however, RISC-V uses PC-relative addressing rather than access GOT via the global pointer register (`gp`), so we use `gp` register to optimize code size and performance of the symbol accessing.

> Tag_RISCV_x3_reg_usage is treated as 0 if it is not present.

9.1.5. GOT load relaxation

Target Relocation

R_RISCV_GOT_HI20, R_RISCV_PCREL_LO12_I

Description

This relaxation can relax a GOT indirection into load immediate or PC-relative addressing. This relaxation is intended to optimize the `lga` assembly pseudo-instruction (and thus `la` for PIC objects), which loads a symbol's address from a GOT entry with an `auipc` + `l[w|d]` instruction pair.

Condition

- Both `R_RISCV_GOT_HI20` and `R_RISCV_PCREL_LO12_I` are marked with `R_RISCV_RELAX`.
- The symbol pointed to by `R_RISCV_PCREL_LO12_I` is at the location to which `R_RISCV_GOT_HI20` refers.
- If the symbol is absolute, its address is within `0x0` ~ `0x7ff` or `0xfffffffffffff800` ~ `0xffffffffffffffff` for RV64 and `0xfffff800` ~ `0xffffffff` for RV32. Note that an undefined weak symbol satisfies this condition because such a symbol is handled as if it were an absolute symbol at address 0.
- If the symbol is relative, it's bound at link time to be within the object. It should not be of the

GNU ifunc type. Additionally, the offset between the location to which R_RISCV_GOT_HI20 refers and the target symbol should be within a range of +-2GiB.

Relaxation

- The auipc instruction associated with R_RISCV_GOT_HI20 can be removed if the symbol is absolute.
- The instruction or instructions associated with R_RISCV_PCREL_LO12_I can be rewritten to either c.li or addi to materialize the symbol's address directly in a register.
- If this relaxation eliminates all references to the symbol's GOT slot, the linker may opt not to create a GOT slot for that symbol.

Example

Relaxation candidate:

```
label:
    auipc   tX, 0       # R_RISCV_GOT_HI20 (symbol), R_RISCV_RELAX
    l[w|d]  tY, 0(tX)  # R_RISCV_PCREL_LO12_I (label), R_RISCV_RELAX
```

Relaxation result (absolute symbol whose address can be represented as a 6-bit signed integer and if the RVC instruction is permitted):

```
    c.li    tY, <symbol-value>
```

Relaxation result (absolute symbol and did not meet the above condition to use c.li):

```
    addi    tY, zero, <symbol-value>
```

Relaxation result (relative symbol):

```
    auipc   tX, <hi>
    addi    tY, tX, <lo>
```

9.1.6. Zero-page Relaxation

Target Relocation

R_RISCV_HI20, R_RISCV_LO12_I, R_RISCV_LO12_S

Description

This relaxation type can relax a sequence of the load address of a symbol or load/store with a symbol reference into shorter instruction sequence if possible.

Condition

The symbol address located within 0x0 ~ 0x7ff or 0xfffffffffffff800 ~ 0xffffffffffffffff for RV64 and 0xfffff800 ~ 0xffffffff for RV32.

Relaxation

- Instruction associated with `R_RISCV_HI20` can be removed if the symbol address satisfies the x0-relative access.
- Instruction associated with `R_RISCV_LO12_I` or `R_RISCV_LO12_S` can be relaxed into x0-relative access.

Example

Relaxation candidate:

```
lui t0, 0       # R_RISCV_HI20 (symbol), R_RISCV_RELAX
lw t1, 0(t0)    # R_RISCV_LO12_I (symbol), R_RISCV_RELAX
```

Relaxation result:

```
lw t1, <address-of-symbol>(x0)
```

9.1.7. Compressed LUI Relaxation

Target Relocation

R_RISCV_HI20, R_RISCV_LO12_I, R_RISCV_LO12_S

Description

This relaxation type can relax a sequence of the load address of a symbol or load/store with a symbol reference into shorter instruction sequence if possible.

Condition

The symbol address can be presented by a `C.LUI` plus an `ADDI` or load / store instruction.

Relaxation

- Instruction associated with `R_RISCV_HI20` can be replaced with `C.LUI`.
- Instruction associated with `R_RISCV_LO12_I` or `R_RISCV_LO12_S` should keep unchanged.

Example

Relaxation candidate:

```
lui t0, 0       # R_RISCV_HI20 (symbol), R_RISCV_RELAX
lw t1, 0(t0)    # R_RISCV_LO12_I (symbol), R_RISCV_RELAX
```

Relaxation result:

```
c.lui t0, <non-zero>  # RVC_LUI (symbol), R_RISCV_RELAX
lw t1, 0(t0)          # R_RISCV_LO12_I (symbol), R_RISCV_RELAX
```

9.1.8. Thread-pointer Relaxation

Target Relocation

R_RISCV_TPREL_HI20, R_RISCV_TPREL_ADD, R_RISCV_TPREL_LO12_I, R_RISCV_TPREL_LO12_S.

Description

This relaxation type can relax a sequence of the load address of a symbol or load/store with a thread-local symbol reference into a thread-pointer-relative instruction.

Condition

Offset between thread-pointer and thread-local symbol is within +-2KiB.

Relaxation

- Instruction associated with `R_RISCV_TPREL_HI20` or `R_RISCV_TPREL_ADD` can be removed.
- Instruction associated with `R_RISCV_TPREL_LO12_I` or `R_RISCV_TPREL_LO12_S` can be replaced with a thread-pointer-relative access instruction.

Example

Relaxation candidate:

```
lui t0, 0        # R_RISCV_TPREL_HI20 (symbol), R_RISCV_RELAX
add t0, t0, tp   # R_RISCV_TPREL_ADD (symbol), R_RISCV_RELAX
lw t1, 0(t0)     # R_RISCV_TPREL_LO12_I (symbol), R_RISCV_RELAX
```

Relaxation result:

```
lw t1, <tp-offset-for-symbol>(tp)
```

9.1.9. TLS Descriptors → Initial Exec Relaxation

Target Relocation

R_RISCV_TLSDESC_HI20, R_RISCV_TLSDESC_LOAD_LO12, R_RISCV_TLSDESC_ADD_LO12, R_RISCV_TLSDESC_CALL

Description

This relaxation can relax a sequence loading the address of a thread-local symbol reference into a GOT load instruction.

Condition

- Linker output is an executable.

Relaxation

- Instruction associated with `R_RISCV_TLSDESC_HI20` or `R_RISCV_TLSDESC_LOAD_LO12` can be removed.
- Instruction associated with `R_RISCV_TLSDESC_ADD_LO12` can be replaced with load of the high half of the symbol's GOT address.

- Instruction associated with R_RISCV_TLSDESC_CALL can be replaced with load of the low half of the symbol's GOT address.

Example

Relaxation candidate (tX and tY can be any combination of two general purpose registers):

```
label:
    auipc tX, <hi>      // R_RISCV_TLSDESC_HI20 (symbol), R_RISCV_RELAX
    lw    tY, tX, <lo>  // R_RISCV_TLSDESC_LOAD_LO12 (label), R_RISCV_RELAX
    addi  a0, tX, <lo>  // R_RISCV_TLSDESC_ADD_LO12 (label), R_RISCV_RELAX
    jalr  t0, tY        // R_RISCV_TLSDESC_CALL (label), R_RISCV_RELAX
```

Relaxation result:

```
    auipc   a0, <pcrel-got-offset-for-symbol-hi>
    {ld,lw} a0, <pcrel-got-offset-for-symbol-lo>(a0)
```

9.1.10. TLS Descriptors → Local Exec Relaxation

Target Relocation

R_RISCV_TLSDESC_HI20, R_RISCV_TLSDESC_LOAD_LO12, R_RISCV_TLSDESC_ADD_LO12, R_RISCV_TLSDESC_CALL

Description

This relaxation can relax a sequence loading the address of a thread-local symbol reference into a thread-pointer-relative instruction sequence.

Condition

- Short form only: Offset between thread-pointer and thread-local symbol is within +-2KiB.
- Linker output is an executable.
- Target symbol is non-preemptible.

Relaxation

- Instruction associated with R_RISCV_TLSDESC_HI20 or R_RISCV_TLSDESC_LOAD_LO12 can be removed.
- Instruction associated with R_RISCV_TLSDESC_ADD_LO12 can be replaced with the high TP-relative offset of symbol (long form) or be removed (short form).
- Instruction associated with R_RISCV_TLSDESC_CALL can be replaced with the low TP-relative offset of symbol.

Example

Relaxation candidate (tX and tY can be any combination of two general purpose registers):

```
label:
    auipc tX, <hi>      // R_RISCV_TLSDESC_HI20 (symbol), R_RISCV_RELAX
    lw    tY, tX, <lo>  // R_RISCV_TLSDESC_LOAD_LO12 (label), R_RISCV_RELAX
```

```
addi  a0, tX, <lo>  // R_RISCV_TLSDESC_ADD_LO12 (label), R_RISCV_RELAX
jalr  t0, tY        // R_RISCV_TLSDESC_CALL (label), R_RISCV_RELAX
```

Relaxation result (long form):

```
lui a0, <tp-offset-for-symbol-hi>
addi a0, a0, <tp-offset-for-symbol-lo>
```

Relaxation result (short form):

```
addi a0, zero, <tp-offset-for-symbol>
```

9.1.11. Table Jump Relaxation

Target Relocation

R_RISCV_CALL, R_RISCV_CALL_PLT, R_RISCV_JAL.

Description

This relaxation type can relax a function call or jump instruction into a single table jump instruction with the index of the target address in table jump section (Table 18). Before relaxation, the linker scans all relocations and calculates whether additional gains can be obtained by using table jump instructions, where expected size saving from function-call-related relaxations and the size of jump table will be taken into account. If there is no additional gain, then table jump relaxation is ignored. Otherwise, this relaxation is switched on. Compressed Tail Call Relaxation and Compressed Function Call Relaxation are always prefered during relaxation, since table jump relaxation has no extra size saving over these two relaxations and might bring a performance overhead.

Condition

The `zcmt` extension is required, the linker output is not position-independent and the rd operand of a function call or jump instruction is `X0` or `RA`.

Relaxation

- Instruction sequence associated with `R_RISCV_CALL` or `R_RISCV_CALL_PLT` can be rewritten to a table jump instruction.
- Instruction associated with `R_RISCV_JAL` can be rewritten to a table jump instruction.

Example

Relaxation candidate:

```
auipc ra, 0          # R_RISCV_CALL (symbol), R_RISCV_RELAX (symbol)
jalr  ra, ra, 0

auipc ra, 0          # R_RISCV_CALL_PLT (symbol), R_RISCV_RELAX (symbol)
jalr  x0, ra, 0
```

```
jal ra, 0              # R_RISCV_JAL (symbol), R_RISCV_RELAX (symbol)

jal x0, 0              # R_RISCV_JAL (symbol), R_RISCV_RELAX (symbol)
```

Relaxation result:

```
cm.jalt  <index-for-symbol>

cm.jt    <index-for-symbol>

cm.jalt  <index-for-symbol>
```

The `zcmt` extension cannot be used in position-independent binaries.

Jump or call instructions with the rd operand `RA` will be relaxed into `cm.jalt` and instructions with the rd operand `X0` will be relaxed into `cm.jt`. The table jump section holds target addresses for these two instructions separately. More details are available in the *ZC* extension specification* [riscv-zc-extension-group].

This relaxation requires programs to initialize the `jvt` CSR with the address of the `__jvt_base$` symbol before executing table jump instructions. It is recommended to initialize `jvt` CSR immediately after global pointer initialization.

```
    # Recommended way to initialize the jvt CSR.
1:  auipc a0, %pcrel_hi(__jvt_base$)
    addi  a0, a0, %pcrel_lo(1b)
    csrw  jvt, a0
```

References

- [gabi] "Generic System V Application Binary Interface" www.sco.com/developers/gabi/latest/contents.html
- [itanium-cxx-abi] "Itanium C++ ABI" itanium-cxx-abi.github.io/cxx-abi/
- [rv-asm] "RISC-V Assembly Programmer’s Manual" github.com/riscv-non-isa/riscv-asm-manual
- [tls] "ELF Handling For Thread-Local Storage" www.akkadia.org/drepper/tls.pdf, Ulrich Drepper
- [riscv-unpriv] "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document", Editors Andrew Waterman and Krste Asanovi´c, RISC-V International.
- [riscv-unpriv-20191213] "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document release 20191213", Editors Andrew Waterman and Krste Asanovi´c, RISC-V International.
- [riscv-zc-extension-group] "ZC* extension specification" github.com/riscv/riscv-code-size-reduction
- [rvv-intrinsic-doc] "RISC-V Vector Extension Intrinsic Document" github.com/riscv-non-isa/rvv-intrinsic-doc

RISC-V DWARF Specification

Chapter 10. DWARF Debugging Format

The DWARF debugging format for RISC-V follows the standard DWARF specification; this specification only describes RISC-V-specific definitions.

Chapter 11. DWARF Register Numbers

The table below lists the mapping from DWARF register numbers to machine registers.

Table 22. DWARF register number encodings

DWARF Number	Register Name	Description
0 - 31	x0 - x31	Integer Registers
32 - 63	f0 - f31	Floating-point Registers
64		Alternate Frame Return Column
65 - 95		Reserved for future standard extensions
96 - 127	v0 - v31	Vector Registers
128 - 3071		Reserved for future standard extensions
3072 - 4095		Reserved for custom extensions
4096 - 8191		CSRs

The alternate frame return column is meant to be used when unwinding from signal handlers, and stores the address where the signal handler will return to.

The RISC-V specification defines a total of 4096 CSRs (see [riscv-priv]). Each CSR is assigned a DWARF register number corresponding to its specified CSR number plus 4096.

References

- [riscv-priv] "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document", Editors Andrew Waterman, Krste Asanovi´c, and John Hauser, RISC-V International.

RISC-V Run-time ABI Specification

Chapter 12. Run-time ABI

This document defines the run-time helper function ABI for RISC-V, which includes compiler helper functions, but does not cover the language standard library functions.

RISC-V Atomics ABI Specification

Chapter 13. RISC-V atomics mappings

This specifies mappings of C and C++ atomic operations to RISC-V machine instructions. Other languages, for example Java, provide similar facilities that should be implemented in a consistent manner, usually by applying the mapping for the corresponding C++ primitive.

> Because different programming languages may be used within the same process, these mappings must be compatible across programming languages. For example, Java programmers expect memory ordering guarantees to be enforced even if some of the actual memory accesses are performed by a library written in C.

> Though many mappings are possible, not all of them will interoperate correctly. In particular, many mapping combinations will not correctly enforce ordering between a C++ `memory_order_seq_cst` store and a subsequent `memory_order_seq_cst` load.

> These mappings are very similar to those that originally appeared in the appendix of the RISC-V "unprivileged" architecture specification as "Mappings from C/C++ primitives to RISC-V Primitives", which we will refer to by their 2019 historical label of "Table A.6". That mapping may be used, *except* that `atomic_store(memory_order_seq_cst)` must have an an extra trailing fence for compatibility with the "Hypothetical mappings …" table in the same section, which we similarly refer to as "Table A.7". As a result, we allow the "Table A.7" mappings as well.

> Our primary design goal is to maximize performance of the "Table A.7" mappings. These require additional load-acquire and store-release instructions, and are this not immediately usable. By requiring the extra store fence. or equivalent, we avoid an ABI break when moving to the "Table A.7" mappings in the future, in return for a small performance penalty in the short term.

For each construct, we provide a mapping that assumes only the A extension. In some cases, we provide additional mappings that assume a future load-acquire and store-release extension, as denoted by note 1 in the table.

All mappings interoperate correctly, and with the original "Table A.6" mappings, *except* that mappings marked with note 3 do not interoperate with the original "Table A.6" mappings.

We present the mappings as a table in 3 sections. The first deals with translations for loads, stores, and fences. The next two sections address mappings for read-modify-write operations like `fetch_add`, and `exchange`. The second section deals with operations that have direct `amo` instruction equivalents in the RISC-V A extension. The final section deals with other read-modify-write operations that require the `lr` and `sc` instructions.

Table 23. Mappings from C/C++ primitives to RISC-V primitives

C/C++ Construct	RVWMO Mapping	Notes
Non-atomic load	l{b|h|w|d}	
atomic_load(memory_order_relaxed)	l{b|h|w|d}	
atomic_load(memory_order_acquire)	l{b|h|w|d}; fence r,rw	
atomic_load(memory_order_acquire)	<RCsc atomic load-acquire>	1, 2
atomic_load(memory_order_seq_cst)	fence rw,rw; l{b|h|w|d}; fence r,rw	
atomic_load(memory_order_seq_cst)	<RCsc atomic load-acquire>	1, 3
Non-atomic store	s{b|h|w|d}	
atomic_store(memory_order_relaxed)	s{b|h|w|d}	
atomic_store(memory_order_release)	fence rw,w; s{b|h|w|d}	
atomic_store(memory_order_release)	<RCsc atomic store-release>	1, 2
atomic_store(memory_order_seq_cst)	fence rw,w; s{b|h|w|d}; fence rw,rw;	
atomic_store(memory_order_seq_cst)	amoswap.rl{w|d};	4
atomic_store(memory_order_seq_cst)	<RCsc atomic store-release>	1
atomic_thread_fence(memory_order_acquire)	fence r,rw	
atomic_thread_fence(memory_order_release)	fence rw,w	
atomic_thread_fence(memory_order_acq_rel)	fence.tso	
atomic_thread_fence(memory_order_seq_cst)	fence rw,rw	

C/C++ Construct	RVWMO AMO Mapping	Notes
atomic_<op>(memory_order_relaxed)	amo<op>.{w|d}	4
atomic_<op>(memory_order_acquire)	amo<op>.{w|d}.aq	4
atomic_<op>(memory_order_release)	amo<op>.{w|d}.rl	4
atomic_<op>(memory_order_acq_rel)	amo<op>.{w|d}.aqrl	4
atomic_<op>(memory_order_seq_cst)	amo<op>.{w|d}.aqrl	4

C/C++ Construct	RVWMO LR/SC Mapping	Notes
atomic_<op>(memory_order_relaxed)	loop:lr.{w|d}; <op>; sc.{w|d}; bnez loop	4
atomic_<op>(memory_order_acquire)	loop:lr.{w|d}.aq; <op>; sc.{w|d}; bnez loop	4
atomic_<op>(memory_order_release)	loop:lr.{w|d}; <op>; sc.{w|d}.rl; bnez loop	4
atomic_<op>(memory_order_acq_rel)	loop:lr.{w|d}.aq; <op>; sc.{w|d}.rl; bnez loop	4
atomic_<op>(memory_order_seq_cst)	loop:lr.{w|d}.aqrl; <op>; sc.{w|d}.rl; bnez loop	4
atomic_<op>(memory_order_seq_cst)	loop:lr.{w|d}.aq; <op>; sc.{w|d}.rl; bnez loop	3, 4

13.1. Meaning of notes in table

1) Depends on a load instruction with an RCsc aquire annotation, or a store instruction with an RCsc release annotation. These are currently under discussion, but the specification has not yet

been approved.

2) An RCpc load or store would also suffice, if it were to be introduced in the future.

3) Incompatible with the original "Table A.6" mapping. Do not combine these mappings with code generated by a compiler using those older mappings. (This was mostly used by the initial LLVM implementations for RISC-V.)

4) Currently only directly possible for 32- and 64-bit operands.

Chapter 14. Ztso atomics mappings

This specifies additional mappings of C and C++ atomic operations to RISC-V machine instructions.

For each construct, we provide a mapping that assumes only the A and Ztso extension.

All mappings interoperate correctly with the RVWMO mappings, and with the original "Table A.6" mappings.

We present the mappings as a table in 3 sections, as above.

Table 24. Mappings with Ztso extension from C/C++ primitives to RISC-V primitives

C/C++ Construct	Ztso Mapping	Notes
atomic_load(memory_order_acquire)	l{b|h|w|d}	5
atomic_load(memory_order_seq_cst)	fence rw,rw; l{b|h|w|d}	5
atomic_store(memory_order_release)	s{b|h|w|d}	5
atomic_store(memory_order_seq_cst)	s{b|h|w|d}; fence rw, rw	5
atomic_thread_fence(memory_order_acquire)	nop	5
atomic_thread_fence(memory_order_release)	nop	5
atomic_thread_fence(memory_order_acq_rel)	nop	5

C/C++ Construct	Ztso AMO Mapping	Notes
atomic_<op>(memory_order_acquire)	amo<op>.{w|d}	4, 5
atomic_<op>(memory_order_release)	amo<op>.{w|d}	4, 5
atomic_<op>(memory_order_acq_rel)	amo<op>.{w|d}	4, 5
atomic_<op>(memory_order_seq_cst)	amo<op>.{w|d}	4, 5

C/C++ Construct	Ztso LR/SC Mapping	Notes
atomic_<op>(memory_order_acquire)	loop:lr.{w|d}; <op>; sc.{w|d}; bnez loop	4, 5
atomic_<op>(memory_order_release)	loop:lr.{w|d}; <op>; sc.{w|d}; bnez loop	4, 5
atomic_<op>(memory_order_acq_rel)	loop:lr.{w|d}; <op>; sc.{w|d}; bnez loop	4, 5

14.1. Meaning of notes in table

4) Currently only directly possible for 32- and 64-bit operands.

5) Requires the Ztso extension.

Chapter 15. Other conventions

It is expected that the RVWMO and Ztso AMO Mappings will be used for atomic read-modify-write operations that are directly supported by corresponding AMO instructions, and that LR/SC mappings will be used for the remainder, currently including compare-exchange operations. Compare-exchange LR/SC sequences on the containing 32-bit word should be used for shorter operands. Thus, a `fetch_add` operation on a 16-bit quantity would use a 32-bit LR/SC sequence.

It is acceptable, but usually undesirable for performance reasons, to use LR/SC mappings where an AMO mapping would suffice.

Atomics do not imply any ordering for IO operations. IO operations should include sufficient fences to prevent them from being visibly reordered with atomic operations.

Float and double atomic loads and stores should be implemented using the integer sequences.

Float and double read-modify-write instructions should consist of a loop performing an initial plain load of the value, followed by the floating point computation, followed by an integer compare-and-swap sequence to try to store back the updated value. This avoids floating point instructions between LR and SC instructions. Depending on language requirements, it may be necessary to save and restore floating-point exception flags in the case of an operation that is later redone due to a failed SC operation.

> The "Eventual Success of Store-Conditional Instructions" section in the ISA specification provides that essential progress guarantee only if there are no floating point instructions between the LR and matching SC instruction. By compiling such sequences with an "extra" ordinary load, and performing the floating point computation before the LR, we preserve the guarantee.