# RISC-V ABIs Specification

Version 1.0-rc1: Frozen

# Table of Contents

# Preamble

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 1.0-rc1', Editors Kito Cheng and Jessica Clarke, RISC-V International, December 2021.

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

# 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

# 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 | Program Linkage Table |
| PC | Program Counter |
| TLS | Thread-Local Storage |
| NTBS | Null-Terminated Byte String |

# RISC-V Calling Conventions

# Chapter 1. Register Convention

## 1.1. 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 |

*Table 1. Integer register convention*

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.

## 1.2. 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 |

*Table 2. Floating-point register convention*

*: 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>".

# 1.3. Vector Register Convention

| Name | ABI Mnemonic | Meaning | Preserved across calls? |
|---|---|---|---|
| v0-v31 | | Temporary registers | No |
| vl | | Vector length | No |
| vtype | | Vector data type register | No |

*Table 3. Vector register convention*

Vector registers are not used for passing arguments or return values; we intend to define a new calling convention variant to allow that as a future software optimization.

The `vxrm` and `vxsat` fields of `vcsr` have thread storage duration.

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

## 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 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 half is passed in a register and the second half is 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.

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 15-10 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.

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.

For the purposes of this section, FLEN refers to the width of a floating-point register in the ABI. The ABI's 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]; } g[2]; }` and `struct { float f; float g; }` 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 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); }`.

A real floating-point argument is passed in a floating-point argument register if it is no more than 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 is more than 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, without extending the integer to XLEN bits, provided the floating-point real is no more than 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. Otherwise, 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 FLEN bits wide.

## 2.3. ILP32E Calling Convention

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.

## 2.4. 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 FLEN=32 (i.e. ELFCLASS32 and EF_RISCV_FLOAT_ABI_SINGLE).

**ILP32D**

ILP32 with hardware floating-point calling convention for 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 FLEN=32 (i.e. ELFCLASS64 and EF_RISCV_FLOAT_ABI_SINGLE).

**LP64D**

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

**LP64Q**

LP64 with hardware floating-point calling convention for 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.

## 2.5. 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):

| 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 |
| _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 |

*Table 4. C/C++ type sizes and alignments for RV32*

**LP64, LP64F, LP64D, and LP64Q**

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

| Type | Size (Bytes) | Alignment (Bytes) |
|---|---:|---:|
| bool/_Bool | 1 | 1 |
| char | 1 | 1 |
| short | 2 | 2 |
| int | 4 | 4 |
| long | 8 | 8 |
| long long | 8 | 8 |
| __int128 | 16 | 16 |
| void * | 8 | 8 |

| Type | Size (Bytes) | Alignment (Bytes) |
|---|---|---|
| _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 |

*Table 5. C/C++ type sizes and alignments for RV64*

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. 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.

`_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.

## 4.3. 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.

# 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.

| Type | Size (Bytes) | Alignment (Bytes) |
|---|---:|---:|
| wchar_t | 4 | 4 |
| wint_t | 4 | 4 |

*Table 6. Linux-specific C type sizes and alignments*

## 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.

# 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. Small

The small code model, or `medlow`, allows the code to address the whole RV32 address space or the lower 2 GiB of the RV64 address space. By using the instructions `lui` and `ld` or `st`, when referring to an object, or `addi`, when calculating an address literal, for example, a 32-bit address literal can be produced. This code model is not position independent.

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)
```

## 5.2. Medium

The medium code model, or `medany`, allows the code to address the range between -2 GiB and +2 GiB from its position. By using the instructions `auipc` and `ld` or `st`, 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. This code model is position independent.

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
0:  auipc a0, %pcrel_hi(symbol)
    lw    a0, %pcrel_lo(0b)(a0)

    # Store value to a symbol
1:  auipc a0, %pcrel_hi(symbol)
    sw    a1, %pcrel_lo(1b)(a0)

    # Calculate address
2:  auipc a0, %pcrel_hi(symbol)
    addi  a0, a0, %pcrel_lo(2b)
```

# Chapter 6. Dynamic Linking

Run-time linkers that use lazy binding must preserve all argument registers used in the standard calling convention for the ABI in use. Any functions that use additional argument registers must be annotated with `STO_RISCV_VARIANT_CC`, as defined in Section 8.4.

> 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. This attribute allows vector registers to not be part of the standard calling convention so run-time linkers are not required to save/restore them and can instead eagerly bind such functions.

# Chapter 7. C++ Name Mangling

C++ name mangling for RISC-V follows the *Itanium C++ ABI* [itanium-cxx-abi]; there are no RISC-V specific mangling rules.

See the "Type encodings" section in *Itanium C++ ABI* for more detail on how to mangle types.

# 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. We only support RISC-V v2 family ISAs, this support is implicit.

**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 7 shows the layout of e_flags, and flag details are listed below.

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

*Table 7. Layout of e_flags*

**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 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 many 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.

## 8.2. Sections

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

## 8.3. String Tables

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

## 8.4. 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 8.

| Name | Mask |
| --- | --- |
| STO_RISCV_VARIANT_CC | 0x80 |

*Table 8. RISC-V-specific `st_other` flags*

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.

## 8.5. 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 9 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, omitted the prefix of `R_RISCV_` here.

**Type**

Whether the relocation is a static or runtime relocation:

- Static relocations are always resolved by the static linker
- Runtime relocations can be resolved by both static and dynamic linkers

**Field**

Describes the set of bits affected by this relocation; see Section 8.5.2 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 8.5.1

**Description**

Additional information about the relocation

| Enum | ELF Reloc Type | Type | Field / Calculation | Description |
| --- | --- | --- | --- | --- |
| 0 | NONE | None | | |
| 1 | 32 | Runtime | *word32*<br>S + A | 32-bit relocation |

| Enum | ELF Reloc Type | Type | Field / Calculation | Description |
|---|---|---|---|---|
| 2 | 64 | Runtime | *word64*<br>S + A | 64-bit relocation |
| 3 | RELATIVE | Runtime | *wordclass*<br>B + A | Relocation against a local symbol in a shared object |
| 4 | COPY | Runtime | | Must be in executable; not allowed in shared library |
| 5 | JUMP_SLOT | Runtime | *wordclass*<br>S | Indicates the symbol associated with a PLT entry |
| 6 | TLS_DTPMOD32 | Runtime | *word32*<br>TLSMODULE | |
| 7 | TLS_DTPMOD64 | Runtime | *word64*<br>TLSMODULE | |
| 8 | TLS_DTPREL32 | Runtime | *word32*<br>S + A - TLS_DTV_OFFSET | |
| 9 | TLS_DTPREL64 | Runtime | *word64*<br>S + A - TLS_DTV_OFFSET | |
| 10 | TLS_TPREL32 | Runtime | *word32*<br>S + A + TLSOFFSET | |
| 11 | TLS_TPREL64 | Runtime | *word64*<br>S + A + TLSOFFSET | |
| 16 | BRANCH | Static | *B-Type*<br>S + A - P | 12-bit PC-relative branch offset |
| 17 | JAL | Static | *J-Type*<br>S + A - P | 20-bit PC-relative jump offset |
| 18 | CALL | Static | *U+J-Type*<br>S + A - P | 32-bit PC-relative function call, macros `call`, `tail` |
| 19 | CALL_PLT | Static | *U+J-Type*<br>S + A - P | 32-bit PC-relative function call, macros `call`, `tail` (PIC) |
| 20 | GOT_HI20 | Static | *U-Type*<br>G + A - P | High 20 bits of 32-bit PC-relative GOT access, `%got_pcrel_hi(symbol)` |
| 21 | TLS_GOT_HI20 | Static | *U-Type* | High 20 bits of 32-bit PC-relative TLS IE GOT access, macro `la.tls.ie` |

| Enum | ELF Reloc Type | Type | Field / Calculation | Description |
|---|---|---|---|---|
| 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*<br>S + A - P | High 20 bits of 32-bit PC-relative reference, `%pcrel_hi(symbol)` |
| 24 | PCREL_LO12_I | Static | *I-type*<br>S - P | Low 12 bits of a 32-bit PC-relative, `%pcrel_lo(address of %pcrel_hi)`, the addend must be 0 |
| 25 | PCREL_LO12_S | Static | *S-Type*<br>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*<br>S + A | High 20 bits of 32-bit absolute address, `%hi(symbol)` |
| 27 | LO12_I | Static | *I-Type*<br>S + A | High 12 bits of 32-bit absolute address, `%lo(symbol)` |
| 28 | LO12_S | Static | *S-Type*<br>S + A | High 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*<br>V + S + A | 8-bit label addition |
| 34 | ADD16 | Static | *word16*<br>V + S + A | 16-bit label addition |
| 35 | ADD32 | Static | *word32*<br>V + S + A | 32-bit label addition |
| 36 | ADD64 | Static | *word64*<br>V + S + A | 64-bit label addition |
| 37 | SUB8 | Static | *word8*<br>V - S - A | 8-bit label subtraction |

| Enum | ELF Reloc Type | Type | Field / Calculation | Description |
|---|---|---|---|---|
| 38 | SUB16 | Static | *word16*<br>V - S - A | 16-bit label subtraction |
| 39 | SUB32 | Static | *word32*<br>V - S - A | 32-bit label subtraction |
| 40 | SUB64 | Static | *word64*<br>V - S - A | 64-bit label subtraction |
| 41 | GNU_VTINHERIT | Static | | GNU C++ vtable hierarchy |
| 42 | GNU_VTENTRY | Static | | GNU C++ vtable member usage |
| 43 | ALIGN | Static | | Alignment statement |
| 44 | RVC_BRANCH | Static | *CB-Type*<br>S + A - P | 8-bit PC-relative branch offset |
| 45 | RVC_JUMP | Static | *CJ-Type*<br>S + A - P | 11-bit PC-relative jump offset |
| 46 | RVC_LUI | Static | *CI-Type*<br>S + A | High 6 bits of 18-bit absolute address |
| 47-50 | **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*<br>V - S - A | Local label subtraction |
| 53 | SET6 | Static | *word6*<br>S + A | Local label assignment |
| 54 | SET8 | Static | *word8*<br>S + A | Local label assignment |
| 55 | SET16 | Static | *word16*<br>S + A | Local label assignment |
| 56 | SET32 | Static | *word32*<br>S + A | Local label assignment |
| 57 | 32_PCREL | Static | *word32*<br>S + A - P | 32-bit PC relative |
| 58 | IRELATIVE | Runtime | *wordclass*<br>`ifunc_resolver(B + A)` | Relocation against a local ifunc symbol in a shared object |

| Enum | ELF Reloc Type | Type | Field / Calculation | Description |
|---|---|---|---|---|
| 59-191 | **Reserved** | - | | Reserved for future standard use |
| 192-255 | **Reserved** | - | | Reserved for nonstandard ABI extensions |

*Table 9. Relocation types*

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].

### 8.5.1. Calculation Symbols

Table 10 provides details on the 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) |
| 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 |

*Table 10. Variables used in relocation calculation*

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

### 8.5.2. Field Symbols

Table 11 provides details on the 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 |
| *wordclass* | Specifies a *word32* field for ILP32 or a *word64* field for LP64 |

| Variable | Description |
|---|---|
| *B-Type* | Specifies a field as the immediate field in a B-type instruction |
| *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+J-Type* | Specifies a field as the immediate fields in a U-type and J-type instruction pair |

*Table 11. Variables used in relocation fields*

### 8.5.3. Constants

Table 12 provides details on the constants used in relocation fields:

| Name | Value |
|---|---|
| TLS_DTV_OFFSET | 0x800 |

*Table 12. Constants used in relocation fields*

### 8.5.4. 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 followed by an I-Type instruction (add immediate or load) with an `R_RISCV_LO12_I` relocation or an S-Type instruction (store) and an `R_RISCV_LO12_S` relocation. The addresses for pair of relocations are calculated like this:

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

**LO12** `symbol_address - (hi20 << 12)`

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)
```

### 8.5.5. 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 runtime 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.

### 8.5.6. Program Linkage Table

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

### 8.5.7. 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.

With 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
```

### 8.5.8. PC-Relative Jumps and Branches

Unconditional jump (U+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).

### 8.5.9. 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 followed by an I-Type instruction (add immediate or load) with an `R_RISCV_PCREL_LO12_I` relocation or an S-Type instruction (store) and an `R_RISCV_PCREL_LO12_S` relocation.

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 - (hi20 << 12)`

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)
```

## 8.6. 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 13 lists the thread local storage models:

| Mnemonic | Model | Compiler flags |
|---|---|---|
| TLS LE | Local Exec | `-ftls-model=local-exec` |
| TLS IE | Initial Exec | `-ftls-model=initial-exec` |
| TLS LD | Local Dynamic | `-ftls-model=local-dynamic` |
| TLS GD | Global Dynamic | `-ftls-model=global-dynamic` |

*Table 13. TLS models*

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.

### 8.6.1. 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.

**Compiler flag**

`-ftls-model=local-exec`

**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.

### 8.6.2. 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.

**Compiler flag**

`-ftls-model=initial-exec`

**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)
```

### 8.6.3. 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.

**Compiler flag**

`-ftls-model=global-dynamic`

**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 are defined as:

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

# 8.7. Sections

## 8.7.1. Section Types

The defined processor-specific section types are listed in Table 14.

| Name | Value | Attributes |
|---|---|---|
| SHT_RISCV_ATTRIBUTES | 0x70000003 | none |

*Table 14. RISC-V-specific section types*

## 8.7.2. Special Sections

Table 15 lists the special sections defined by this ABI.

| Name | Type | Attributes |
|---|---|---|
| .riscv.attributes | SHT_RISCV_ATTRIBUTES | none |

*Table 15. RISC-V-specific sections*

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

# 8.8. Program Header Table

The defined processor-specific segment types are listed in Table 16.

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

*Table 16. RISC-V-specific segment types*

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

# 8.9. Note Sections

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

# 8.10. Dynamic Section

The defined processor-specific dynamic array tags are listed in Table 17.

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

*Table 17. RISC-V-specific dynamic array tags*

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`.

# 8.11. Hash Table

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

# 8.12. 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).

RISC-V attributes have a string value if the tag number is odd and an integer value if the tag number is even.

## 8.12.1. List of 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 | Indicates the major version of the privileged specification. |
| Tag_RISCV_priv_spec_minor | 10 | uleb128 | Indicates the minor version of the privileged specification. |
| Tag_RISCV_priv_spec_revision | 12 | uleb128 | Indicates the revision version of the privileged specification. |
| Reserved for non-standard attribute | >= 32768 | - | - |

*Table 18. RISC-V attributes*

### 8.12.2. 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.

It will 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.

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 `RV32I2P0` in which `2P0` stands for the default version of its based ISA. On the other hand, the architecture `RV32G` has to be presented as `RV32I2P0_M2P0_A2P0_F2P0_D2P0` in which the abbreviation `G` is expanded to the IMAFD combination with default versions of the standard extensions.

#### 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.

#### Tag_RISCV_priv_spec, 8, uleb128=version

#### Tag_RISCV_priv_spec_minor, 10, uleb128=version

#### Tag_RISCV_priv_spec_revision, 12, uleb128=version

Tag_RISCV_priv_spec contains the major/minor/revision version information of the privileged specification. It will report errors if object files of different privileged specification versions are merged.

# Chapter 9. Code 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.

# 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

# 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.

| 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 |

*Table 19. DWARF register number encodings*

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.