RISC-V ELF psABI specification

Table of Contents

Copyright and license information

Register Convention

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.

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>".

Procedure Calling Convention

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.)

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.

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.

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.

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 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.

- **on RV32G**: ILP32D

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.

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.

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.

Compact

The compact code model allows the code to address the whole 64-bit address space, especially when code and data are located far apart. By using the Global Offset Table, or GOT, to hold the 64-bit address literals, any memory position can be referred. By using the instructions `lui` and `addi`, a signed 32-bit offset, relative to the value of the `gp` register, can be produced, referring to address literals in the GOT. This code model is position independent.
Does not apply to the ILP32 ABIs.

C type details

C type sizes and alignments

There are two conventions for C type sizes and alignments.

- **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
float	4	4
double	8	8
long double	16	16
float _Complex	8	4
double _Complex	16	8
long double _Complex	32	16

- **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
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.

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.

`_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.

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.

ELF Object Files

File Header

- e_ident
  - EI_CLASS: Specifies the base ISA, either RV32 or RV64. We don't let users link RV32 and RV64 code together.
    - ELFCLASS64: ELF-64 Object File
    - ELFCLASS32: ELF-32 Object File
- e_type: Nothing RISC-V specific.
- 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.

Bit 0	**Bit 1 - 2**	**Bit 3**	**Bit 4**	**Bit 5 - 31**
RVC	Float ABI	RVE	TSO	*Reserved*

- 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 (0xffffffe0) are allocated by future versions of this specification, they shall not be set by standard software.

Sections

String Tables

Symbol Table

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.

The following table provides details of the RISC-V ELF relocations (instruction specific relocations show the instruction type in the Details column):

Enum	ELF Reloc Type	Description	Field	Calculation	Details
0	R_RISCV_NONE	None			
1	R_RISCV_32	Runtime relocation	*word32*	S + A	
2	R_RISCV_64	Runtime relocation	*word64*	S + A	
3	R_RISCV_RELATIVE	Runtime relocation	*wordclass*	B + A	
4	R_RISCV_COPY	Runtime relocation			Must be in executable; not allowed in shared library
5	R_RISCV_JUMP_SLOT	Runtime relocation	*wordclass*	S	Handled by PLT unless `LD_BIND_NOW`
6	R_RISCV_TLS_DTPMOD32	TLS relocation	*word32*	S->TLSINDEX	
7	R_RISCV_TLS_DTPMOD64	TLS relocation	*word64*	S->TLSINDEX	
8	R_RISCV_TLS_DTPREL32	TLS relocation	*word32*	S + A + TLS - TLS_TP_OFFSET	
9	R_RISCV_TLS_DTPREL64	TLS relocation	*word64*	S + A + TLS - TLS_TP_OFFSET	
10	R_RISCV_TLS_TPREL32	TLS relocation	*word32*	S + A + TLS + S_TLS_OFFSET - TLS_DTV_OFFSET	
11	R_RISCV_TLS_TPREL64	TLS relocation	*word64*	S + A + TLS + S_TLS_OFFSET - TLS_DTV_OFFSET	
16	R_RISCV_BRANCH	PC-relative branch	*B-Type*	S + A - P	
17	R_RISCV_JAL	PC-relative jump	*J-Type*	S + A - P	
18	R_RISCV_CALL	PC-relative call	*J-Type*	S + A - P	Macros `call`, `tail`
19	R_RISCV_CALL_PLT	PC-relative call (PLT)	*J-Type*	S + A - P	Macros `call`, `tail` (PIC)
20	R_RISCV_GOT_HI20	PC-relative GOT reference	*U-Type*	G + A - P	`%got_pcrel_hi(symbol)`
21	R_RISCV_TLS_GOT_HI20	PC-relative TLS IE GOT offset	*U-Type*		Macro `la.tls.ie`
22	R_RISCV_TLS_GD_HI20	PC-relative TLS GD reference	*U-Type*		Macro `la.tls.gd`
23	R_RISCV_PCREL_HI20	PC-relative reference	*U-Type*	S + A - P	`%pcrel_hi(symbol)`
24	R_RISCV_PCREL_LO12_I	PC-relative reference	*I-type*	S + A - P	`%pcrel_lo(address of %pcrel_hi)`
25	R_RISCV_PCREL_LO12_S	PC-relative reference	*S-Type*	S + A - P	`%pcrel_lo(address of %pcrel_hi)`
26	R_RISCV_HI20	Absolute address	*U-Type*	S + A	`%hi(symbol)`
27	R_RISCV_LO12_I	Absolute address	*I-Type*	S + A	`%lo(symbol)`
28	R_RISCV_LO12_S	Absolute address	*S-Type*	S + A	`%lo(symbol)`
29	R_RISCV_TPREL_HI20	TLS LE thread offset	*U-Type*		`%tprel_hi(symbol)`
30	R_RISCV_TPREL_LO12_I	TLS LE thread offset	*I-Type*		`%tprel_lo(symbol)`
31	R_RISCV_TPREL_LO12_S	TLS LE thread offset	*S-Type*		`%tprel_lo(symbol)`
32	R_RISCV_TPREL_ADD	TLS LE thread usage			`%tprel_add(symbol)`
33	R_RISCV_ADD8	8-bit label addition	*word8*	V + S + A	

Enum	ELF Reloc Type	Description	Field	Calculation	Details
34	R_RISCV_ADD16	16-bit label addition	*word16*	V + S + A	
35	R_RISCV_ADD32	32-bit label addition	*word32*	V + S + A	
36	R_RISCV_ADD64	64-bit label addition	*word64*	V + S + A	
37	R_RISCV_SUB8	8-bit label subtraction	*word8*	V - S - A	
38	R_RISCV_SUB16	16-bit label subtraction	*word16*	V - S - A	
39	R_RISCV_SUB32	32-bit label subtraction	*word32*	V - S - A	
40	R_RISCV_SUB64	64-bit label subtraction	*word64*	V - S - A	
41	R_RISCV_GNU_VTINHERIT	GNU C++ vtable hierarchy			
42	R_RISCV_GNU_VTENTRY	GNU C++ vtable member usage			
43	R_RISCV_ALIGN	Alignment statement			
44	R_RISCV_RVC_BRANCH	PC-relative branch offset	*CB-Type*	S + A - P	
45	R_RISCV_RVC_JUMP	PC-relative jump offset	*CJ-Type*	S + A - P	
46	R_RISCV_RVC_LUI	Absolute address	*CI-Type*	S + A	
47	R_RISCV_GPREL_I	GP-relative reference	*I-Type*	S + A - GP	
48	R_RISCV_GPREL_S	GP-relative reference	*S-Type*	S + A - GP	
49	R_RISCV_TPREL_I	TP-relative TLS LE load	*I-Type*		
50	R_RISCV_TPREL_S	TP-relative TLS LE store	*S-Type*		
51	R_RISCV_RELAX	Instruction pair can be relaxed			
52	R_RISCV_SUB6	Local label subtraction	*word6*	V - S - A	
53	R_RISCV_SET6	Local label assignment	*word6*	S + A	
54	R_RISCV_SET8	Local label assignment	*word8*	S + A	
55	R_RISCV_SET16	Local label assignment	*word16*	S + A	
56	R_RISCV_SET32	Local label assignment	*word32*	S + A	
57	R_RISCV_32_PCREL	PC-relative reference	*word32*	S + A - P	
58	R_RISCV_IRELATIVE	Runtime relocation	*wordclass*	`ifunc_resolver(B + A)`	
59-191	*Reserved*	Reserved for future standard use			

Enum	ELF Reloc Type	Description	Field	Calculation	Details
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.

Calculation Symbols

The following table 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

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

Field Symbols

The following table 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
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

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)
```

GP-Relative Relocations: If `symbol` is within the range of a signed 12-bit immediate offset from `__global_pointer$`, then the address can be loaded with a single instruction which has one relocation, a `R_RISCV_GPREL_I` or `R_RISCV_GPREL_S`.

The instruction is an I-Type instruction (add immediate or load) with an `R_RISCV_GPREL_I` or an S-type instruction (store) with an `R_RISCV_GPREL_S` relocation. The following assembly show loading a gp-relative address:

```
      addi a0, gp, 0          # R_RISCV_GPREL_I (symbol)
```

This relies on the value of `__global_pointer$` being loaded into `gp` (aka `x3`). This can be used by linker relaxation to delete the `lui` instruction.

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.

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
```

Procedure Calls

`R_RISCV_CALL` or `R_RISCV_CALL_PLT` and `R_RISCV_RELAX` relocations are associated with pairs of instructions (`AUIPC+JALR`) generated by the `CALL` or `TAIL` pseudoinstructions.

In position dependent code (`-fno-pic`) the `AUIPC` instruction in the `AUIPC+JALR` pair has both a `R_RISCV_CALL` relocation and a `R_RISCV_RELAX` relocation indicating the instruction sequence can be relaxed during linking.

In position independent code (`-fPIC`, `-fpic` or `-fpie`) the `AUIPC` instruction in the `AUIPC+JALR` pair has both a `R_RISCV_CALL_PLT` relocation and a `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
```

expands to the following assembly and relocation:

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

and when `-fpic` is enabled it expands to:

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

PC-Relative Jumps and Branches

Unconditional jump (UJ-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).

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 with a `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)
```

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 https://www.akkadia.org/drepper/tls.pdf.

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) held in the `tp` register.

There are various thread local storage models for statically allocated or dynamically allocated thread local storage. The following table 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`

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.

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.

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)
```

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;
```

Program Header Table

Note Sections

Dynamic Table

Hash Table

DWARF

Dwarf Register Numbers

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.

There space for 4096 CSRs. Each CSR is assigned a DWARF register number corresponding to its CSR number given in **Volume II: Privileged Architecture** of **The RISC-V Instruction Set Manual** plus 4096.

Linux-specific ABI

This section of the RISC-V ELF psABI 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.

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

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.

Terms and Definitions

- **ABI**: Application binary interface.
  Here a combination of the **gABI** and the **psABI**
- **gABI**: generic ABI
- **psABI**: processor specific ABI