LoongArch ELF ABI specification

Loongson Technology Corporation Limited

Version 1.00

Table of Contents

Register Convention

Table 1. General-purpose Register Convention

Name	Alias	Meaning	Preserved across calls
`$r0`	`$zero`	Constant zero	(Constant)
`$r1`	`$ra`	Return address	No
`$r2`	`$tp`	Thread pointer	(Non-allocatable)
`$r3`	`$sp`	Stack pointer	Yes
`$r4` - `$r5`	`$a0` - `$a1`	Argument registers / return value registers	No
`$r6` - `$r11`	`$a2` - `$a7`	Argument registers	No
`$r12` - `$r20`	`$t0` - `$t8`	Temporary registers	No
`$r21`		Reserved	(Non-allocatable)
`$r22`	`$fp` / `$s9`	Frame pointer / Static register	Yes
`$r23` - `$r31`	`$s0` - `$s8`	Static registers	Yes

Table 2. Floating-point Register Convention

Name	Alias	Meaning	Preserved across calls
`$f0` - `$f1`	`$fa0` - `$fa1`	Argument registers / return value registers	No
`$f2` - `$f7`	`$fa2` - `$fa7`	Argument registers	No
`$f8` - `$f23`	`$ft0` - `$ft15`	Temporary registers	No
`$f24` - `$f31`	`$fs0` - `$fs7`	Static registers	Yes

Temporary registers are also known as caller-saved registers. Static registers are also known as callee-saved registers.

Aliases for return value registers

You may see the names `$v0` `$v1` `$fv0` `$fv1` in some very early LoongArch assembly code; they simply alias to `$a0` `$a1` `$fa0` `$fa1` respectively. The aliases are initially meant to match MIPS convention with separate argument / return value registers. However, because LoongArch actually has no dedicated return value registers, such usage may lead to confusion. Hence, it is not recommended to use these aliases.

Due to implementation details, it may not be easy to give a register multiple ABI names for a given downstream project. New programs processing LoongArch assembly should not support these aliases. Portable LoongArch assembly should avoid these aliases.

NOTE

For toolchain components provided by the Loongson Corporation, the migration procedure is:

Let the version of the component at this spec's effect date be N,

1. keep support in the version N and its stable branch,
2. warn on such usage in the version N+1,
3. remove support in the version N+2.

> For the respective upstream projects of the components, the procedure above shall be followed if support for such usage is already upstream; "version N" shall be interpreted as the first release version containing LoongArch support in that case. For the components not yet upstream, and not interacting with other components that may expect such usage, support for such usage will never be implemented.

Type Size and Alignment

Table 3. LP64 Data Model (base ABI types: `lp64d` `lp64f` `lp64s`)

Scalar type	Size (Bytes)	Alignment (Bytes)
`bool` / `_Bool`	1	1
`unsigned char` / `char`	1	1
`unsigned short` / `short`	2	2
`unsigned int` / `int`	4	4
`unsigned long` / `long`	8	8
`unsigned long long` / `long long`	8	8
pointer types	8	8
`float`	4	4
`double`	8	8
`long double`	16	16

Table 4. ILP32 Data Model (base ABI types: `ilp32d` `ilp32f` `ilp32s`)

Scalar type	Size (Bytes)	Alignment (Bytes)
`bool` / `_Bool`	1	1
`unsigned char` / `char`	1	1
`unsigned short` / `short`	2	2
`unsigned int` / `int`	4	4
`unsigned long` / `long`	4	4
`unsigned long long` / `long long`	8	8
pointer types	4	4
`float`	4	4
`double`	8	8
`long double`	16	16

For all base ABI types of LoongArch, the `char` datatype is signed by default.

ELF Object Files

All common ELF definitions referenced in this section can be found in the latest SysV gABI specification.

`EI_CLASS`: File class

EI_CLASS	Value	Description
`ELFCLASS32`	`1`	ELF32 object file
`ELFCLASS64`	`2`	ELF64 object file

`e_machine`: Identifies the machine

`LoongArch (258)`

`e_flags`: Identifies ABI type and version

Bit 31 - 8	Bit 7 - 6	Bit 5 - 3	Bit 2 - 0
(reserved)	ABI version	ABI extension	Base ABI

The ABI type of an ELF object is uniquely identified by `e_flags[7:0]` in its header.

Table 5. Base ABI Types

Name	Value of `e_flags[2:0]`	Description
	`0x0`	(reserved)
`lp64s`	`0x1`	Uses 64-bit GPRs and the stack for parameter passing. Data model is LP64, where `long` and pointers are 64-bit while `int` is 32-bit.
`lp64f`	`0x2`	Uses 64-bit GPRs, 32-bit FPRs and the stack for parameter passing. Data model is LP64, where `long` and pointers are 64-bit while `int` is 32-bit.
`lp64d`	`0x3`	Uses 64-bit GPRs, 64-bit FPRs and the stack for parameter passing. Data model is LP64, where `long` and pointers are 64-bit while `int` is 32-bit.
	`0x4`	(reserved)
`ilp32s`	`0x5`	Uses 32-bit GPRs and the stack for parameter passing. Data model is ILP32, where `int`, `long` and pointers are 32-bit.
`ilp32f`	`0x6`	Uses 32-bit GPRs, 32-bit FPRs and the stack for parameter passing. Data model is ILP32, where `int`, `long` and pointers are 32-bit.
`ilp32d`	`0x7`	Uses 32-bit GPRs, 64-bit FPRs and the stack for parameter passing. Data model is ILP32, where `int`, `long` and pointers are 32-bit.

Table 6. ABI Extension types

Name	Value of `e_flags[5:3]`	Description
`base`	`0x0`	No extra ABI features.
	`0x1` - `0x7`	(reserved)

`e_flags[7:6]` marks the ABI version of an ELF object.

Table 7. ABI Version

ABI version	Value	Description
`v0`	`0x0`	Stack operands base relocation type.
`v1`	`0x1`	Another relocation type IF needed.
	`0x2` `0x3`	Reserved.

Relocations

Table 8. ELF Relocation types

Enum	ELF reloc type	Usage	Detail
0	`R_LARCH_NONE`		
1	`R_LARCH_32`	Runtime address resolving	`*(int32_t *) PC = RtAddr + A`
2	`R_LARCH_64`	Runtime address resolving	`*(int64_t *) PC = RtAddr + A`
3	`R_LARCH_RELATIVE`	Runtime fixup for load-address	`*(void **) PC = B + A`
4	`R_LARCH_COPY`	Runtime memory copy in executable	`memcpy (PC, RtAddr, sizeof (sym))`
5	`R_LARCH_JUMP_SLOT`	Runtime PLT supporting	*implementation-defined*
6	`R_LARCH_TLS_DTPMOD32`	Runtime relocation for TLS-GD	`*(int32_t *) PC = ID of module defining sym`
7	`R_LARCH_TLS_DTPMOD64`	Runtime relocation for TLS-GD	`*(int64_t *) PC = ID of module defining sym`
8	`R_LARCH_TLS_DTPREL32`	Runtime relocation for TLS-GD	`*(int32_t *) PC = DTV-relative offset for sym`
9	`R_LARCH_TLS_DTPREL64`	Runtime relocation for TLS-GD	`*(int64_t *) PC = DTV-relative offset for sym`
10	`R_LARCH_TLS_TPREL32`	Runtime relocation for TLE-IE	`*(int32_t *) PC = T`
11	`R_LARCH_TLS_TPREL64`	Runtime relocation for TLE-IE	`*(int64_t *) PC = T`
12	`R_LARCH_IRELATIVE`	Runtime local indirect function resolving	`*(void **) PC = (((void *)(*)()) (B + A)) ()`
… Reserved for dynamic linker.			
20	`R_LARCH_MARK_LA`	Mark la.abs	Load absolute address for static link.
21	`R_LARCH_MARK_PCREL`	Mark external label branch	Access PC relative address for static link.
22	`R_LARCH_SOP_PUSH_PCREL`	Push PC-relative offset	`push (S - PC + A)`
23	`R_LARCH_SOP_PUSH_ABSOLUTE`	Push constant or absolute address	`push (S + A)`
24	`R_LARCH_SOP_PUSH_DUP`	Duplicate stack top	`opr1 = pop (), push (opr1), push (opr1)`

Enum	ELF reloc type	Usage	Detail
25	R_LARCH_SOP_PUSH_GPREL	Push for access GOT entry	push (G)
26	R_LARCH_SOP_PUSH_TLS_TPREL	Push for TLS-LE	push (T)
27	R_LARCH_SOP_PUSH_TLS_GOT	Push for TLS-IE	push (IE)
28	R_LARCH_SOP_PUSH_TLS_GD	Push for TLS-GD	push (GD)
29	R_LARCH_SOP_PUSH_PLT_PCREL	Push for external function calling	push (PLT - PC)
30	R_LARCH_SOP_ASSERT	Assert stack top	assert (pop ())
31	R_LARCH_SOP_NOT	Stack top operation	push (!pop ())
32	R_LARCH_SOP_SUB	Stack top operation	opr2 = pop (), opr1 = pop (), push (opr1 - opr2)
33	R_LARCH_SOP_SL	Stack top operation	opr2 = pop (), opr1 = pop (), push (opr1 << opr2)
34	R_LARCH_SOP_SR	Stack top operation	opr2 = pop (), opr1 = pop (), push (opr1 >> opr2)
35	R_LARCH_SOP_ADD	Stack top operation	opr2 = pop (), opr1 = pop (), push (opr1 + opr2)
36	R_LARCH_SOP_AND	Stack top operation	opr2 = pop (), opr1 = pop (), push (opr1 & opr2)
37	R_LARCH_SOP_IF_ELSE	Stack top operation	opr3 = pop (), opr2 = pop (), opr1 = pop (), push (opr1 ? opr2 : opr3)
38	R_LARCH_SOP_POP_32_S_10_5	Instruction imm-field relocation	opr1 = pop (), (*(uint32_t *) PC) [14 ... 10] = opr1 [4 ... 0] with check 5-bit signed overflow
39	R_LARCH_SOP_POP_32_U_10_12	Instruction imm-field relocation	opr1 = pop (), (*(uint32_t *) PC) [21 ... 10] = opr1 [11 ... 0] with check 12-bit unsigned overflow

Enum	ELF reloc type	Usage	Detail
40	`R_LARCH_SOP_POP_32_S_10_12`	Instruction imm-field relocation	`opr1 = pop (), (*(uint32_t *) PC) [21 ... 10] = opr1 [11 ... 0]` with check 12-bit signed overflow
41	`R_LARCH_SOP_POP_32_S_10_16`	Instruction imm-field relocation	`opr1 = pop (), (*(uint32_t *) PC) [25 ... 10] = opr1 [15 ... 0]` with check 16-bit signed overflow
42	`R_LARCH_SOP_POP_32_S_10_16_S2`	Instruction imm-field relocation	`opr1 = pop (), (*(uint32_t *) PC) [25 ... 10] = opr1 [17 ... 2]` with check 18-bit signed overflow and 4-bit aligned
43	`R_LARCH_SOP_POP_32_S_5_20`	Instruction imm-field relocation	`opr1 = pop (), (*(uint32_t *) PC) [24 ... 5] = opr1 [19 ... 0]` with check 20-bit signed overflow
44	`R_LARCH_SOP_POP_32_S_0_5_10_16_S2`	Instruction imm-field relocation	`opr1 = pop (), (*(uint32_t *) PC) [4 ... 0] = opr1 [22 ... 18],` `(*(uint32_t *) PC) [25 ... 10] = opr1 [17 ... 2]` with check 23-bit signed overflow and 4-bit aligned
45	`R_LARCH_SOP_POP_32_S_0_10_10_16_S2`	Instruction imm-field relocation	`opr1 = pop (), (*(uint32_t *) PC) [9 ... 0] = opr1 [27 ... 18],` `(*(uint32_t *) PC) [25 ... 10] = opr1 [17 ... 2]` with check 28-bit signed overflow and 4-bit aligned
46	`R_LARCH_SOP_POP_32_U`	Instruction fixup	`(*(uint32_t *) PC) = pop ()` with check 32-bit unsigned overflow
47	`R_LARCH_ADD8`	8-bit in-place addition	`*(int8_t *) PC += S + A`
48	`R_LARCH_ADD16`	16-bit in-place addition	`*(int16_t *) PC += S + A`

Enum	ELF reloc type	Usage	Detail
49	`R_LARCH_ADD24`	24-bit in-place addition	`*(int24_t *) PC += S + A`
50	`R_LARCH_ADD32`	32-bit in-place addition	`*(int32_t *) PC += S + A`
51	`R_LARCH_ADD64`	64-bit in-place addition	`*(int64_t *) PC += S + A`
52	`R_LARCH_SUB8`	8-bit in-place subtraction	`*(int8_t *) PC -= S + A`
53	`R_LARCH_SUB16`	16-bit in-place subtraction	`*(int16_t *) PC -= S + A`
54	`R_LARCH_SUB24`	24-bit in-place subtraction	`*(int24_t *) PC -= S + A`
55	`R_LARCH_SUB32`	32-bit in-place subtraction	`*(int32_t *) PC -= S + A`
56	`R_LARCH_SUB64`	64-bit in-place subtraction	`*(int64_t *) PC -= S + A`
57	`R_LARCH_GNU_VTINHERIT`	GNU C++ vtable hierarchy	
58	`R_LARCH_GNU_VTENTRY`	GNU C++ vtable member usage	

Program Interpreter Path

Table 9. Standard Program Interpreter Paths

Base ABI type	ABI extension type	Operating system / C library	Program interpreter path
`lp64d`	`base`	Linux, Glibc	`/lib64/ld-linux-loongarch-lp64d.so.1`
`lp64f`	`base`	Linux, Glibc	`/lib64/ld-linux-loongarch-lp64f.so.1`
`lp64s`	`base`	Linux, Glibc	`/lib64/ld-linux-loongarch-lp64s.so.1`
`ilp32d`	`base`	Linux, Glibc	`/lib32/ld-linux-loongarch-ilp32d.so.1`
`ilp32f`	`base`	Linux, Glibc	`/lib32/ld-linux-loongarch-ilp32f.so.1`
`ilp32s`	`base`	Linux, Glibc	`/lib32/ld-linux-loongarch-ilp32s.so.1`

Procedure Calling Convention

Abbreviations

In this document, **GRLEN** is the bit width of general-purpose register, **FRLEN** is the bit width of floating-point register and **WOA** is the bit width of the argument. The general-purpose argument register is denoted as **GAR** and the floating-point argument register is denoted as **FAR**.

Argument Registers

The basic principle of the LoongArch procedure calling convention is to pass arguments in registers as much as possible (i.e. floating-point arguments are passed in floating-point registers and non floating-point arguments are passed in general-purpose registers, as much as possible); arguments are passed on the stack only when no appropriate register is available.

The argument registers are:

1. Eight floating-point registers `fa0-fa7` used for passing pass floating-point arguments, and `fa0-fa1` are also used to return values.
2. Eight general-purpose registers `a0-a7` used for passing pass integer arguments, with `a0-a1` reused to return values.

Generally, the GARs are used to pass fixed-point arguments, and floating-point arguments when no FAR is available. Bit fields are stored in little endian. In addition, subroutines should ensure that the values of general-purpose registers `s0-s9` and floating-point registers `fs0-fs7` are preserved across procedure calls.

ABI LP64D

That is, **GRLEN** = 64, **FRLEN** = 64.

C Data Types and Alignment

The C data types and alignment in the LP64D ABI are defined in the table 3.

In most cases, the unsigned integer data types are zero-extended when stored in general-purpose register, and the signed integer data types are sign-extended. However, in the **LP64D** ABI, unsigned 32-bit types, such as **`unsigned int`**, are stored in general-purpose registers as proper sign extensions of their 32-bit values.

Argument passing

Generally speaking, FARs are only used to pass floating-point arguments, GARs are used to pass non floating-point arguments and floating-point arguments when no FAR is available(`long double` type is also passed in a pair of GARs) and the reference.

Arguments passed by reference may be modified by the callee.

Scalar

There are two cases:

1. 0 < WOA ≤ GRLEN

    a. Argument is passed in a single argument register, or on the stack by value if none is available.

        i. If the argument is floating-point type, the argument is passed in FAR. if no FAR is available, it’s passed in GAR. If no GAR is available, it’s passed on the stack. When passed in registers or on the stack, floating-point types narrower than GRLEN bits are widened to GRLEN bits, with the upper bits undefined.

        ii. If the argument is integer or pointer type, the argument is passed in GAR. If no GAR is available, it’s passed on the stack. When passed in registers or on the stack, the unsigned integer scalars narrower than GRLEN bits are zero-extended to GRLEN bits, and the signed integer scalars are sign-extended.

2. GRLEN < WOA ≤ 2 × GRLEN

    a. The argument is passed in a pair of GAR, with the low-order GRLEN bits in the lower-numbered register and the high-order GRLEN bits in the higher-numbered register. If exactly one register is available, the low-order GRLEN bits are passed in the register and the high-order GRLEN bits are passed on the stack. If no GAR is available, it’s passed on the stack.

Structure

Empty structures are ignored by C compilers which support them as a non-standard extension(same as union arguments and return values). Bits unused due to padding, and bits past the end of a structure whose size in bits is not divisible by GRLEN, are undefined. And the layout of the structure on the stack is consistent with that in memory.

1. 0 < WOA ≤ GRLEN

    a. The structure has only fixed-point members. If there is an available GAR, the structure is passed through the GAR by value passing; If no GAR is available, it’s passed on the stack.

    b. The structure has only floating-point members:

        i. One floating-point member. The argument is passed in a FAR; If no FAR is available, the value is passed in a GAR; if no GAR is available, the value is passed on the stack.

        ii. Two floating-point members. The argument is passed in a pair of available FAR, with the low-order `float` member bits in the lower-numbered FAR and the high-order `float` member bits in the higher-numbered FAR. If the number of available FAR is less than 2, it’s passed in a GAR, and passed on the stack if no GAR is available.

    c. The structure has both fixed-point and floating-point members, i.e. the structure has one `float` member and…

        i. Multiple fixed-point members. If there are available GAR, the structure is passed in a GAR, and passed on the stack if no GAR is available.

        ii. Only one fixed-point member. If one FAR and one GAR are available, the floating-point member of the structure is passed in the FAR, and the integer member of the structure is passed in the GAR; If no floating-point register but one GAR is available, it’s passed in GAR; If no GAR is available, it’s passed on the stack.

2. GRLEN < WOA ≤ 2 × GRLEN

    a. Only fixed-point members.

        i. The argument is passed in a pair of available GAR, with the low-order bits in the lower-numbered GAR and the high-order bits in the higher-numbered GAR. If only one GAR is available, the low-order bits are in the GAR and the high-order bits are on the stack, and passed on the stack if no GAR is available.

    b. Only floating-point members.

        i. The structure has one `long double` member or one `double` member and two adjacent `float` members or 3-4 `float` members. The argument is passed in a pair of available GAR, with the low-order bits in the lower-numbered GAR and the high-order bits in the higher-numbered GAR. If only one GAR is available, the low-order bits are in the GAR and the high-order bits are on the stack, and passed on the stack if no GAR is available.

    ii. The structure with two `double` members is passed in a pair of available FARs. If no a pair of available FARs, it's passed in GARs. A structure with one `double` member and one `float` member is same.

  c. Both fixed-point and floating-point members.

    i. The structure has one `double` member and only one fixed-point member.

      A. If one FAR and one GAR are available, the floating-point member of the structure is passed in the FAR, and the integer member of the structure is passed in the GAR; If no floating-point registers but two GARs are available, it's passed in the two GARs; If only one GAR is available, the low-order bits are in the GAR and the high-order bits are on the stack; And it's passed on the stack if no GAR is available.

    ii. Others

      A. The argument is passed in a pair of available GAR, with the low-order bits in the lower-numbered GAR and the high-order bits in the higher-numbered GAR. If only one GAR is available, the low-order bits are in the GAR and the high-order bits are on the stack, and passed on the stack if no GAR is available.

3. WOA > 2 × GRLEN

  a. It's passed by reference and are replaced in the argument list with the address. If there is an available GAR, the reference is passed in the GAR, and passed on the stack if no GAR is available.

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

Union

Union is passed in GAR or stack.

1. 0 < WOA ≤ GRLEN
    a. The argument is passed in a GAR, or on the stack by value if no GAR is available.
2. GRLEN < WOA ≤ 2 × GRLEN
    a. The argument is passed in a pair of available GAR, with the low-order bits in the lower-numbered GAR and the high-order bits in the higher-numbered GAR. If only one GAR is available, the low-order bits are in the GAR and the high-order bits are on the stack. The arguments are passed on the stack when no GAR is available.
3. WOA > 2 × GRLEN
    a. It's passed by reference and are replaced in the argument list with the address. If there is an available GAR, the reference is passed in the GAR, and passed on the stack if no GAR is available.

Complex

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

Variadic arguments

Variadic arguments are passed in GARs in the same manner as named arguments. And 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.

1. 0 < WOA ≤ GRLEN
    a. The variadic arguments are passed in a GAR, or on the stack by value if no GAR is available.

2. GRLEN < WOA ≤ 2 × GRLEN
    a. The variadic arguments are passed in a pair of GARs. If only one GAR is available, the low-order bits are in the GAR and the high-order bits are on the stack, and passed on the stack if no GAR is available. or on the stack by value if none is available. It should be noted that `long double` data tpye is passed in an aligned GAR pair(the first register in the pair is even-numbered).
3. WOA > 2 × GRLEN
    a. It's passed by reference and are replaced in the argument list with the address. If there is an available GAR, the reference is passed in the GAR, and passed on the stack if no GAR is available.

Return values

1. Generally speaking, `a0` and `a1` are used to return non floating-point values, and `fa0` and `fa1` are used to return floating-point values.
2. 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 argument.
3. The reference of the return value is returned that is stored in GAR `a0` if the size of the return value is larger than 2×GRLEN bits.

Stack

1. In general, the stack frame for a subroutine may contain space to contain the following:
    a. Space to store arguments passed to subroutines that this subroutine calls.
    b. A place to store the subroutine's return address.
    c. A place to store the values of saved registers.
    d. A place for local data storage.
2. 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.
3. Procedures must not rely upon the persistence of stack-allocated data whose addresses lies below the stack pointer.