LoongArch Toolchain Conventions

Loongson Technology Corporation Limited

Version 1.00

Table of Contents

Note: In this document, the terms "architecture", "instruction set architecture" and "ISA" are used synonymously and refer to a certain set of instructions and the set of registers they can operate upon.

Compiler Options

Rationale

Compiler options that are specific to LoongArch should denote a change in the following compiler settings:

- **Target architecture**: the allowed set of instructions and registers to be used by the compiler.
- **Target ABI type**: the data model and calling conventions.
- **Target microarchitecture**: microarchitectural features that guides compiler optimizations.

For this model, two categories of LoongArch-specific compiler options should be implemented：

- **Basic options**: select the base configuration of the compilation target. (include only `-march` `-mabi` and `-mtune`)
- **Extended options**: make incremental changes to the target configuration.

Table 1. Basic Options

Option	Possible values	Description
`-march=`	`native` `loongarch64` `la464` `la364` `la264`	Selects the target architecture, i.e. the basic set of ISA modules to be enabled.
`-mabi=`	`lp64d` `lp64f` `lp64s` `ilp32d` `ilp32f` `ilp32s`	Selects the base ABI type.
`-mtune=`	`native` `loongarch64` `la464` `la364` `la264`	Selects the type of target microarchitecture, defaults to the value of `-march`. The set of possible values should be a superset of `-march` values.

- Valid parameter values of `-march` and `-mtune` options should correspond to actual LoongArch processor models, IP cores or product series.
- In principle, different `-march` values should not imply the same set of ISA modules. The distinctions between these values are meant to reflect the differences in hardware characteristics that have major impact on the hardware-binary compatibiliy. The compiler can adapt these features by adjusting the extended options.

Table 2. Extended Options

Option	Possible values	Description
`-msoft-float`		Prevent the compiler from generating hardware floating-point instructions, and adjust the selected base ABI type to use soft-float calling convention. (The adjusted base ABI identifier should have suffix `s`.)
`-msingle-float`		Allow generating 32-bit floating-point instructions, and adjust the selected base ABI type to use 32-bit FP calling convention. (The adjusted base ABI identifier should have suffix `f`.)
`-mdouble-float`		Allow generating 32- and 64-bit floating-point instructions. and adjust the selected base ABI type to use 64-bit FP calling convention. (The adjusted base ABI identifier should have suffix `d`.)

Option	Possible values	Description
`-mfpu=`	`64` `32` `0` `none` (equivalent to `0`)	(Optional) Selects the allowed set of basic floating-point instructions and registers. This option should not change the FP calling convention unless it's necessary. (The implementation of this option is not mandatory. It is recommended to use `-m*-float` options in software projects and scripts.)
`-mstrict-align`	`-m[no-]strict-align`	Do not generate unaligned memory accesses. Useful for targets that do not support for hardware unaligned memory access.
`-mlsx`	`-m[no-]lsx`	(Optional) Allows generating LSX 128-bit SIMD instructions.
`-mlasx`	`-m[no-]lasx`	(Optional) Allows generating LASX 256-bit SIMD instructions.

For one compilation command, the effective order of all LoongArch-specific compiler options is computed with the following rules:

1. Within each category in the above tables, only the last-seen option is effective (`-m*-float` goes into the same category).
2. Basic options always precedes the extended options.
3. On the basis of rule 1 and 2, any options with parameters (i.e. with `=`) precedes all options without parameters.
4. If the above rule failed to determine the effective order between two options, unless specified by the following table, they should have independent meanings. (i.e. the effective order between them does not affect the compiler's final configuration)

Table 3. Special processing rules for certain compiler option combinations

Option combination	Compiler behavior	Description
`-mfpu=none` `-ml[a]sx`	Abort and report error	64-bit FPU must be implemented when the target has an SIMD extension implemented
`-mfpu=0` `-ml[a]sx`		
`-mfpu=32` `-ml[a]sx`		
`-msoft-float` `-ml[a]sx`		
`-msingle-float` `-m[a]sx`		
`-mno-lsx` `-mlasx`	Abort and report error	the LSX 128-bit SIMD extension must be implemented when the target has the LASX 256-bit SIMD extension

The compiler should reach the final target configuration by applying the options in their effective order. Options that appears later in the order can override the existing ISA / ABI / uArch configuration.

Different from the extended options that makes only incremental changes, the basic options can provide a complete set of target configurations. Thus, a nominal default value is required for each basic option in case they should be absent from the command line.

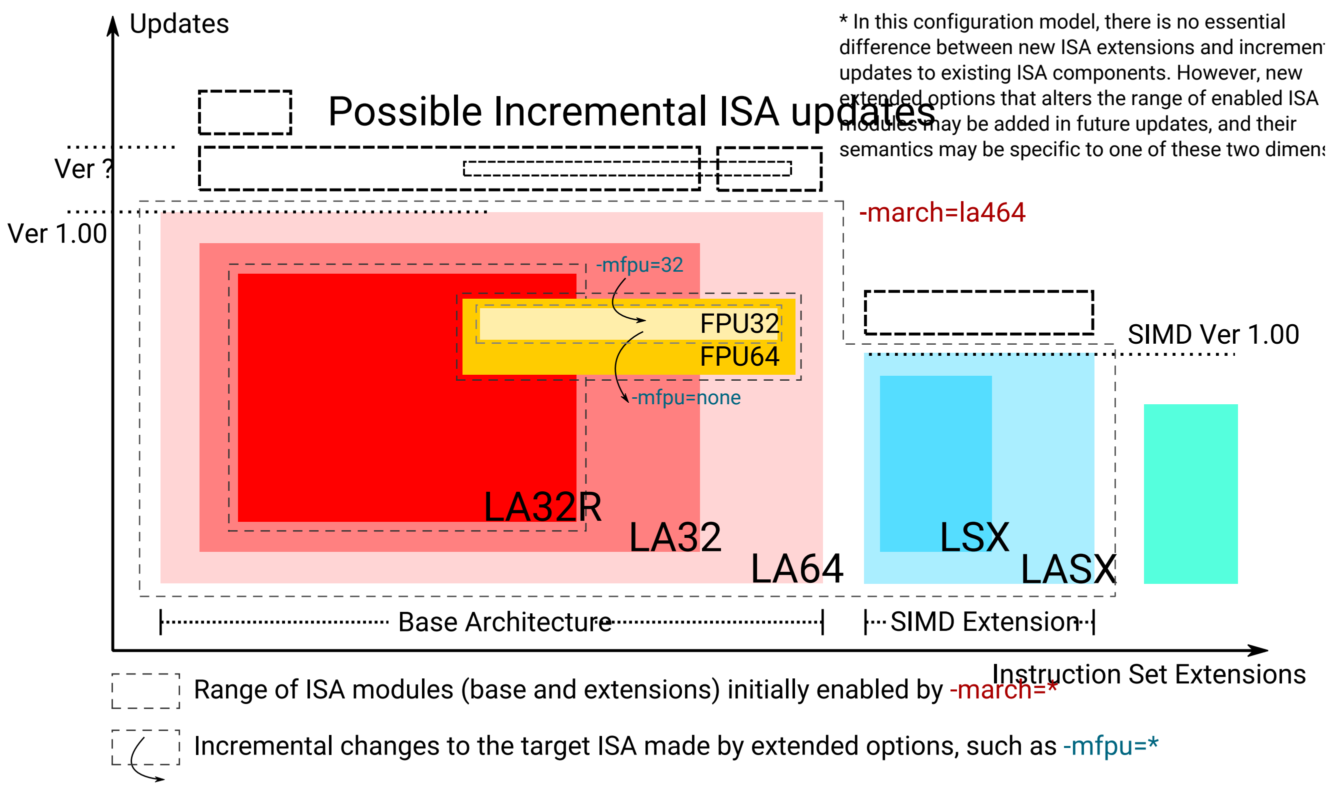
The following sections will cover the details of the target ISA / ABI configuration items.

Configuring Target ISA

The LoongArch ISA is organized in a "base-extension" manner. For future updates, each component in the *base* or *extened* part of the ISA may evolve independently while keeping compatibility with previous versions of itself.

For this purpose, the compiler should make a modular abstraction about the target ISA. The ISA modules are divided into two categories： **base architectures** and **ISA extensions**. A **base architecture** is the core component of the target ISA, which defines the *base* set of functionalities like integer and floating-point operations, and is decided by the value of `-march`. An **ISA extension** may represent either the base of a certain *extended* ISA component or an incremental update to it, and is enabled / disabled by extended options.

When determining the suitable compiler options with the target ISA, it is recommended for the user to first select the base architecture and a set of default ISA extensions with `-march`, and then add the extended options to adjust the enabled ISA extensions so that the target ISA configuration can be consistent with the actual hardware.


Updates
Possible Incremental ISA updates
* In this configuration model, there is no essential difference between new ISA extensions and incremen
updates to existing ISA components. However, new extended options that alters the range of enabled ISA modules may be added in future updates, and their semantics may be specific to one of these two dimens
Ver ?
Ver 1.00
-march=la464
-mfpu=32
FPU32
FPU64
-mfpu=none
SIMD Ver 1.00
LA32R
LA32
LA64
LSX
LASX
Base Architecture
SIMD Extension
Instruction Set Extensions
Range of ISA modules (base and extensions) initially enabled by -march=*
Incremental changes to the target ISA made by extended options, such as -mfpu=*

Among all ISA modules listed below, the compiler should at least implement one base architecture and one combination of enabled ISA extensions.

Table 4. Base Architecture

Name	`-march` values that selects this module	Description
LA64 basic architecture v1.00 (`la64v100`)	`loongarch64` `la464`	ISA defined in *LoongArch Reference Manual - Volume 1: Basic Architecture* v1.00.

The following table lists all ISA extensions that should be abstracted by the compiler and the options for adapting these hardware features.

Table 5. ISA extensions

ISA extension name	Related compiler options	Description of the option(s)
Basic Floating-Point Processing Unit (`fpu*`)	`-mfpu=*` (Possible values of `*`: `none` `32` `64`)	Selects the allowed set of basic floating-point instructions and floating-point registers. This is a constituent part of the base architecture, where it gets its default value.
Hardware support for unaligned memory access (`ual`)	`-m[no-]strict-align`	Do not or do generate unaligned memory accesses. Using `-mstrict-align` is not recommended for hardware with `ual` support.
Loongson SIMD eXtension (`lsx`)	`-m[no-]lsx`	Allow or do not allow generating LSX 128-bit SIMD instructions. Enabling `lsx` requires `fpu64` (the compiler should report an error when there's a conflict between the user-specified `-mfpu=` and `-mlsx` options).
Loongson Advanced SIMD eXtension (`lasx`)	`-m[no-]lasx`	Allow or do not allow generating LASX 256-bit SIMD instructions. Enabling `lasx` requires `lsx` (the compiler should report an error when the user specifies `-mno-lsx` and `-mlasx` together).

The following table lists the properties of all target CPU models that can serve as arguments to `-march` and `-mtune` options at the same time.

Table 6. Target CPU Models (`-march=<model>` and `-mtune=<model>`)

Name / Value	Base architecture [ISA extensions]	Target of optimization
`native`	auto-detected with `cpucfg` (only on native and cross-native compilers)	auto-detected microarchitecture model / features
`loongarch64`	`la64v100` [`ual` `fpu64`]	generic LoongArch LA64 processors
`la464`	`la64v100` [`ual` `fpu64` `lsx` `lasx`]	LA464 processor core
`la364`	`la64v100` [`ual` `fpu64` `lsx`]	LA364 processor core
`la264`	`la64v100` [`fpu64`]	LA264 processor core

Note: for `la464` and `la364` targets, the compiler may choose not to enable the SIMD extensions (`lsx` and `lasx`) if these are not implemented. This is for better `-march` compatibility across compilers.

Configuring Target ABI

Like configuring the target ISA, a complete ABI configuration of LoongArch consists of two parts, the **base ABI** and the **ABI extension**. The former describes the data model and calling convention in general, while the latter denotes an overall adjustment to the base ABI, which may require support from certain ISA extensions.

Please be noted that there is only ONE ABI extension slot in an ABI configuration. They do not combine with one another, and are, in principle, mutually incompatible.

A new ABI extension type will not be added to this document unless it implies certain significant performance / functional advantage that no compiler optimization techniques can provide without altering the ABI.

There are six base ABI types, whose standard names are the same as the `-mabi` values that select them. The compiler may choose to implement one or more of these base ABI types, possibly according to the range of implemented target ISA variants.

Table 7. Base ABI Types

Standard name	Data model	Bit-width of argument / return value GPRs / FPRs
`lp64d`	LP64	64 / 64
`lp64f`	LP64	64 / 32
`lp64s`	LP64	64 / (none)
`ilp32d`	ILP32	32 / 64
`ilp32f`	ILP32	32 / 32
`ilp32s`	ILP32	32 / (none)

The following table lists all ABI extension types and related compiler options. A compiler may choose to implement any subset of these extensions that contains `base`.

The default ABI extension type is `base` when referring to an ABI type with only the "base" component.

Table 8. ABI Extension Types

Name	Compiler options	Description
`base`	(none)	conforms to the LoongArch ELF psABI

The compiler should know the default ABI to use during its build time. If the ABI extension type is not explicitly configured, `base` should be used.

In principle, the target ISA configuration should not affect the decision of the target ABI. When certain ISA feature required by explicit (i.e. from the compiler's command-line arguments) ABI configuration cannot be met due constraints imposed by ISA options, the compiler should abort with an error message to complain about the conflict.

When the ABI is not fully constrained by the compiler options, the default configuration of either the base ABI or the ABI extension, whichever is missing from the command line, should be attempted. If this default ABI setting cannot be implemented by the explicitly configured target ISA, the expected behavior is **undefined** since the user is encouraged to specify which ABI to use when choosing a smaller instruction set than the default.

In this case, it is suggested that the compiler should abort with an error message, however, for userfriendliness, it may also choose to ignore the default base ABI or ABI extension and select a viable fallback ABI for the currently enabled ISA modules with caution. It is also recommended that the compiler should notify the user about the ABI change, optionally with a compiler warning. For example, passing `-mfpu=none` as the only command-line argument may cause a compiler configured with `lp64d` / `base` default ABI to automatically select `lp64s` / `base` instead.

When the target ISA configuration cannot be uniquely decided from the given compiler options, the build-time default should be consulted first. If the default ISA setting is insufficient for implementing the ABI configuration, the compiler should try enabling the missing ISA modules according to the following table, as long as they are not explicitly disabled or excluded from usage.

Table 9. Minimal architecture requirements for implementing each ABI type.

Base ABI type	ABI extension type	Minimal required ISA modules
`lp64d`	`base`	`la64v100` [`fpu64`]
`lp64f`	`base`	`la64v100` [`fpu32`]
`lp64s`	`base`	`la64v100` [`fpunone`]

GNU Target Triplets and Multiarch Specifiers

Target triplet is a core concept in the GNU build system. It describes a platform on which the code runs and mostly consists of three fields: the CPU family / model (`machine`), the vendor (`vendor`), and the operating system name (`os`).

Multiarch architecture apecifiers are essentially standard directory names where libraries are installed on a multiarch-flavored filesystem. These strings are normalized GNU target triplets. See debian documentation for details.

This document recognizes the following `machine` strings for the GNU triplets of LoongArch:

Table 10. LoongArch `machine` strings :

`machine`	Description
`loongarch64`	LA64 base architecture (implies `lp64*` ABI)
`loongarch32`	LA32 base architecture (implies `ilp32*` ABI)

As standard library directory names, the canonical multiarch architecture specifiers of LoongArch should contain information about the ABI type of the libraries that are meant to be released in the binary form and installed there.

While the integer base ABI is implied by the `machine` field, the floating-point base ABI and the ABI extension type are encoded with two string suffices (`<fabi-suffix><abiext-suffix>`) to the `os` field of the specifier, respectively.

Table 11. List of possible `<fabi-suffix>`

`<fabi-suffix>`	Description
(empty string)	The base ABI uses 64-bit FPRs for parameter passing. (`lp64d`)
`f32`	The base ABI uses 32-bit FPRs for parameter passing. (`lp64f`)
`sf`	The base ABI uses no FPR for parameter passing. (`lp64s`)

Table 12. List of possible `<abiext-suffix>`

`<abiext-suffix>`	ABI extension type
(empty string)	`base`

(Note: Since in principle, The default ISA configuration of the ABI should be used in this binary-release scenario, it is not necessary to reserve multiple multiarch specifiers for one OS / ABI combination.)

Table 13. List of LoongArch mulitarch specifiers

ABI type (Base ABI / ABI extension)	C Library	Kernel	Multiarch specifier
`lp64d` / `base`	glibc	Linux	`loongarch64-linux-gnu`
`lp64f` / `base`	glibc	Linux	`loongarch64-linux-gnuf32`
`lp64s` / `base`	glibc	Linux	`loongarch64-linux-gnusf`

ABI type (Base ABI / ABI extension)	C Library	Kernel	Multiarch specifier
`lp64d` / `base`	musl libc	Linux	`loongarch64-linux-musl`
`lp64f` / `base`	musl libc	Linux	`loongarch64-linux-muslf32`
`lp64s` / `base`	musl libc	Linux	`loongarch64-linux-muslsf`

C/C++ Preprocessor Built-in Macro Definitions

The definitions listed below is not specific to LoongArch. Amount of LoongArch-specific code can be minimized by utilizing them, while achieving expected portability in most of cases.

Table 14. Non-LoongArch-specific C/C++ Built-in Macros :

Name	Possible Values	Description
`__BYTE_ORDER__`	(omitted)	Byte order
`__FLOAT_WORD_ORDER__`	(omitted)	Byte order for floating-point data
`__LP64__` `_LP64`	(omitted)	Whether the ABI passes arguments in 64-bit GPRs and uses the `LP64` data model
`__SIZEOF_SHORT__`	(omitted)	Width of C/C++ `short` type, in bytes
`__SIZEOF_INT__`	(omitted)	Width of C/C++ `int` type, in bytes
`__SIZEOF_LONG__`	(omitted)	Width of C/C++ `long` type, in bytes
`__SIZEOF_LONG_LONG__`	(omitted)	Width of C/C++ `long long` type, in bytes
`__SIZEOF_INT128__`	(omitted)	Width of C/C++ `__int128` type, in bytes
`__SIZEOF_POINTER__`	(omitted)	Width of C/C++ pointer types, in bytes
`__SIZEOF_PTRDIFF_T__`	(omitted)	Width of C/C++ `ptrdiff_t` type, in bytes
`__SIZEOF_SIZE_T__`	(omitted)	Width of C/C++ `size_t` type, in bytes
`__SIZEOF_WINT_T__`	(omitted)	Width of C/C++ `wint_t` type, in bytes
`__SIZEOF_WCHAR_T__`	(omitted)	Width of C/C++ `wchar_t` type, in bytes
`__SIZEOF_FLOAT__`	(omitted)	Width of C/C++ `float` type, in bytes
`__SIZEOF_DOUBLE__`	(omitted)	Width of C/C++ `double` type, in bytes
`__SIZEOF_LONG_DOUBLE__`	(omitted)	Width of C/C++ `long double` type, in bytes

Apart from the generic definitions described above, some architecture-specific macros are still needed to convey those information strongly tied to the architecture; these macros are listed below.

Table 15. LoongArch-specific C/C++ Built-in Macros :

Name	Possible Values	Description
`__loongarch__`	`1`	Defined if the target is LoongArch.
`__loongarch_grlen`	`64`	Bit-width of general purpose registers.
`__loongarch_frlen`	`0` `32` `64`	Bit-width of floating-point registers (`0` if there is no FPU).
`__loongarch_arch`	`"loongarch64"` `"la464"` `"la364"` `"la264"`	Processor model as specified by `-march`. If `-march` is not present, the build-time default should be used. If `-march=native` is enabled (user-specified or the default value), the result is automatically detected by the compiler.
`__loongarch_tune`	`"loongarch64"` `"la464"` `"la364"` `"la264"`	Processor model as specified by `-mtune`. If `-mtune` is not present, the value should be the same as `__loongarch_arch`. If `-mtune=native` is enabled (user-specified or set with `-march=native`), the result is automatically detected by the compiler.
`__loongarch_lp64`	undefined or `1`	Defined if ABI uses the LP64 data model and 64-bit GPRs for parameter passing.
`__loongarch_hard_float`	undefined or `1`	Defined if floating-point/extended ABI type is `single` or `double`.
`__loongarch_soft_float`	undefined or `1`	Defined if floating-point/extended ABI type is `soft`.
`__loongarch_single_float`	undefined or `1`	Defined if floating-point/extended ABI type is `single`.
`__loongarch_double_float`	undefined or `1`	Defined if floating-point/extended ABI type is `double`.
`__loongarch_sx`	undefined or `1`	Defined if the compiler enables the `lsx` ISA extension.
`__loongarch_asx`	undefined or `1`	Defined if the compiler enables the `lasx` ISA extension.
`__loongarch_simd_width`	undefined, `128` or `256`	The maximum SIMD bit-width enabled by the compiler. (`128` for `lsx`, and `256` for `lasx`)

For historical reasons, the earliest LoongArch C/C++ compilers provided some MIPS-style built-in macros. Because legacy code dependent on those macros is possibly still in use, compilers conformant to this specification may provide the macros as listed below.

Because the naming style and usage of these macros are more-or-less inconsistent with the other macros described above, there is learning cost involved in using these macros. As they bring no advantage over the other macros, it is not recommended for newer compilers to implement them; portable code should not assume existence of these macros, nor use them.

Table 16. C/C++ Built-in Macros Provided for Compatibility with Historical Code

Name	Equivalent to	Description
`__loongarch64`	`__loongarch_grlen == 64`	Similar to *`mips64`*`; defined iff loongarch_grlen == 64`.
`_LOONGARCH_ARCH`	`__loongarch_arch`	n/a
`_LOONGARCH_TUNE`	`__loongarch_tune`	n/a
`_LOONGARCH_SIM`	n/a	Similar to `_MIPS_SIM` on MIPS; possible values are `_ABILP64` (in case data model is LP64) and `_ABILP32` (in case data model is ILP32; notice the omission of letter `I`).
`_LOONGARCH_SZINT`	`__SIZEOF_INT__` multiplied by 8	n/a
`_LOONGARCH_SZLONG`	`__SIZEOF_LONG__` multiplied by 8	n/a
`_LOONGARCH_SZPTR`	`__SIZEOF_POINTER__` multiplied by 8	n/a