龙芯架构工具链约定

龙芯中科技术股份有限公司

Version 1.00

目录

注：在本文档中，"架构"、"指令集架构"、"*ISA*" 均表示某一指令集和可操作的寄存器集合。

编译器命令行选项

概述

与龙芯架构相关的编译器选项含义包括以下三方面：

- 配置目标架构：允许使用的指令集和寄存器范围；
- 配置目标 **ABI**：标准数据类型的表示方法，函数调用传参、返回的实现方式；
- 配置优化参数：用于指导编译器优化的微架构特性。

为此，编译器应实现以下两类命令行选项：

- 基础选项：选择编译目标的基本配置，包括 `-march` `-mabi` `-mtune`；
- 扩展选项：对基础选项或基础选项默认值的配置进行增量调整。

Table 1. 基础选项

选项	可用值	描述
`-march=`	`native` `loongarch64` `la464`	选择目标架构：设定默认可用的指令集和寄存器范围 （即默认使用的 指令集模块 集合）
`-mabi=`	`lp64d` `lp64f` `lp64s` `ilp32d` `ilp32f` `ilp32s`	选择基础 ABI 类型
`-mtune=`	`native` `loongarch64` `la464`	选择目标微架构：设定微架构相关的性能调优参数； 取值范围是 `-march` 选项的超集，默认值与 `-march` 值相同

- `-march` 和 `-mtune` 参数的取值应代表实现龙芯架构的处理器或产品系列。
- 原则上，不同 `-march` 取值所代表的默认指令集互不相同。

Table 2. 扩展选项

选项	可用值	描述
`-msoft-float`		禁止使用浮点数指令， 并对当前选择的基础 ABI 进行调整，以采用软浮点调用惯例。 (调整后的基础 ABI 名称后缀为 `s`)
`-msingle-float`		允许使用 32 位浮点数指令， 并对当前选择的基础 ABI 进行调整，以采用 32 位浮点调用惯例。 (调整后的基础 ABI 名称后缀为 `f`)
`-mdouble-float`		允许使用 32 位和 64 位浮点数指令， 并对当前选择的基础 ABI 进行调整，以采用 64 位浮点调用惯例。 (调整后的基础 ABI 名称后缀为 `d`)
`-mfpu=`	`64` `32` `0` `none` (等同于 `0`)	(可选实现) 选择可用的基础浮点数指令和寄存器范围，非必要不调整浮点调用惯例。 (在软件项目或脚本中，建议直接使用 `-m*-float'。)

在一条编译命令中，各龙芯架构相关选项的总体配置效果等同于它们按先后顺序依次生效的结果。 唯一的例外是 `-m*-float` ：它们对浮点指令集和调用惯例的配置不会被除 `-m*-float` 之外 其他类型的后续选项改变。

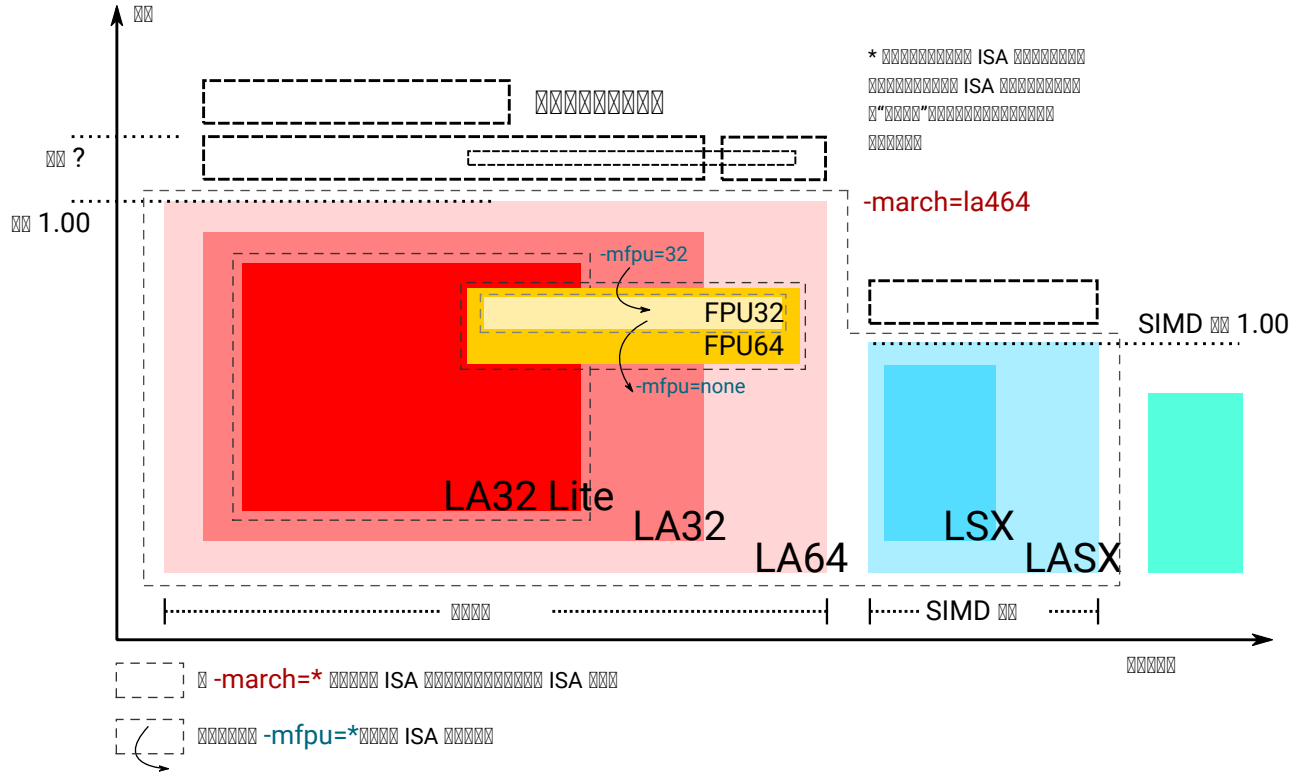
目标指令集架构 (ISA) 的构成

龙芯架构采用 基础部分 加 扩展部分 的组织形式， 在后续更新过程中，基础部分或扩展部分中的各功能子集都可以独立地演进， 并保证高版本总是二进制兼容低版本。

针对这一特点，编译器应当对目标 ISA 进行模块化抽象。 约定 ISA 模块分为两类： 基础架构 和 **ISA** 扩展特性 。

其中，基础架构 为目标 ISA 的核心部分，包含基础整数指令、基础浮点数指令等功能， 由 `-march` 选项的取值唯一确定。**ISA** 扩展特性 可能对应一种单独的指令集扩展， 也可能对应基础架构或指令集扩展的增量/演进部分，由扩展选项控制是否开启。

在确定目标 ISA 配置时，应以 基础架构 隐含的 ISA 模块为基础， 再根据选用 / 关闭 ISA 扩展的命令行选项进行调整，得出结果。



在以下列举的所有的 ISA 模块中，编译器必须至少实现一种基础架构。

Table 3. 基础架构

名称	选择该基础架构的 `-march` 值	描述
LA64 基础架构 v1.00 (`la64v100`)	`loongarch64 la464`	由 《龙芯架构参考手册 - 卷1 - 基础架构》 v1.00 定义的指令集架构

下表列举了由编译器抽象的全体 ISA 扩展特性，以及选用/关闭这些特性的命令行选项。

Table 4. ISA 扩展特性

名称	编译器选项	描述
基础浮点运算单元 (`fpu*`)	`-mfpu=*` (`*` 可能取值为 `none 32 64`)	选择编译器可用的基础浮点数指令和浮点寄存器，属于基础架构的一部分， 默认值由基础架构决定。

下表列举了所有可同时作为 `-march` 和 `-mtune` 选项参数的目标 CPU 类型 及其相关属性。

Table 5. 目标 CPU

名称 / 选项值	默认选择的 **ISA** 模块	性能调优目标
`native`	由 `cpucfg` 指令自动检测（仅适用于本地编译器）	由 `cpucfg` 自动检测的处理器类型
`loongarch64`	`la64v100` [`fpu64`]	通用 64 位龙芯架构 (LA64) 处理器
`la464`	`la64v100` [`fpu64`]	LA464 处理器核

应用二进制接口 (ABI) 的构成

对于龙芯架构编译器，完整的 ABI 配置应包含两个部分： 基础 **ABI** 和 **ABI** 扩展特性 。前者描述了 ABI 中整型和浮点数据的表示、 传参和返回方式，后者则代表对基础 ABI 进行的总体调整，可能需要特定 ISA 扩展支持。

需要注意的是，不同 ABI 扩展特性之间是 互斥 的，不能相互叠加； 具有不同扩展特性的 ABI 配置之间也 互不兼容 。

原则上本文档不会增加新的 ABI 扩展特性，除非它能提供 其他编译器优化技术不能单独实现的功能或性能优势。

基础 ABI 共有六种，编译器可根据实现的目标架构范围，选择实现其中的一种或多种， 其标准名称和对应的 `-mabi` 选项值一致。

Table 6. 基础 ABI 类型

标准名称	数据模型	可用于传参、返回的通用/浮点寄存器宽度
`lp64d`	LP64	64 / 64
`lp64f`	LP64	64 / 32
`lp64s`	LP64	64 / （无）
`ilp32d`	ILP32	32 / 64
`ilp32f`	ILP32	32 / 32
`ilp32s`	ILP32	32 / （无）

下表列举了全体 ABI 扩展特性类型及其相关命令行选项，除 `base` 必须实现之外， 编译器可选择实现或不实现其中任何一种。

当引述一种 ABI 名称时，若仅给出基础 ABI 类型， 则 ABI 扩展特性类型默认为 `base` 。

Table 7. ABI 扩展特性类型

名称	编译器选项	含义
`base`	（无）	符合 龙芯架构 ELF psABI 规范

编译器的默认 ABI 应在构建时确定。此时若未明确配置 ABI 扩展特性类型， 则采用 `base` 。

原则上，实际配置的目标架构不应该对目标 ABI 的确定造成影响， 当命令行选项对 ABI 的明确约束导致实现它所需的指令集特性 超出了编译选项对 ISA 配置的约束范围。编译器应报错退出。

若命令行选项未声明或未完整声明目标 ABI 类型， 缺失的部分（基础 ABI 或 ABI 扩展类型）应当取构建时确

定的默认值。 当编译选项中明确约束的 ISA 范围不足以实现该默认 ABI 配置时， 编译器的实际行为 不确定，因为在缩减默认可用指令集范围的同时， 用户应该通过命令行选项明确表示使用何种 ABI 。

对于编译器实现来说，此时推荐的行为是报错退出，但出于易用性的考量， 也可以在默认配置允许的范围内谨慎地选择一种当前可用指令集能够实现的 备选 ABI。对于这种情况，建议编译器输出提示或警告信息以通知用户。 例如，对于默认 ABI 为 `lp64d` / `base` 的编译器， 若编译选项为 `-mfpu=none` ，则可能自动调整 ABI 到 `lp64s` / `base` 。

当编译选项不能唯一确定目标架构时，编译器应当首先检查默认值是否能满足 ABI 配置的需要。若不能，则应根据下表，在默认可用指令集基础上增加选用 缺失的模块，但不违反已给出编译选项对目标架构的明确约束。

Table 8. 实现各 *ABI* 类型所需的最小目标架构

基础 **ABI** 类型	**ABI** 扩展特性类型	最小目标架构包含的 **ISA** 模块
`lp64d`	`base`	`la64v100` [`fpu64`]
`lp64f`	`base`	`la64v100` `fpu32`
`lp64s`	`base`	`la64v100` `fpunone`

GNU 目标三元组和 Multiarch 架构标识符

GNU 目标三元组 (target triplet) 是 GNU 构建系统用于描述目标平台的字符串， 一般包含三个字段：处理器类型 (`machine`)， 系统厂商 (`vendor`)，操作系统 (`os`)。

Multiarch 架构标识符 是用于 multiarch 库安装路径的目录名称， 可以看作规范的 GNU 目标三元组，参见 Debian 文档 。

对于龙芯架构的合法 GNU 目标三元组，约定 `machine` 字段的取值范围及其含义如下：

Table 9. 龙芯架构 `machine` 字符串

`machine` 字符串	含义
`loongarch64`	LA64 基础架构，基础 ABI 为 `lp64*`
`loongarch32`	LA32 基础架构，基础 ABI 为 `ilp32*`

作为标准的库路径名称，龙芯架构的标准 multiarch 架构标识符至少应该反映 发行到对应目录的二进制库 ABI 类型。

原则上，在编译以二进制形式发行的库时，应当采用 所选 ABI 对应的默认目标指令集架构， 因此 multiarch 架构标识符应与目标 ABI 配置一一对应。 其中，关于整型 ABI 的部分由 `machine` 字段隐含， 基础 ABI 的浮点部分和 ABI 扩展特性则分别由连续附加在 multiarch 标识符 `os` 字段后的两个字符串后缀 (`<fabi_suffix><abiext_suffix>`) 标记。

Table 10. Multiarch `os` 字段，`<fabi_suffix>` 后缀标记及其含义

`<fabi_suffix>` 字符串	含义
（空）	基础 ABI 使用 64 位浮点寄存器传参 (`lp64d`)
`f32`	基础 ABI 使用 32 位浮点寄存器传参 (`lp64f`)
`sf`	基础 ABI 不使用浮点寄存器传参 (`lp64s`)

Table 11. Multiarch `os` 字段，`<abiext_suffix>` 后缀标记及其对应的 ABI 扩展特性

`<abiext_suffix>` 字符串	**ABI** 扩展特性
（空）	`base`

Table 12. 全体 Mulitarch 标识符列表

ABI 类型（基础 **ABI** / **ABI** 扩展特性）	**C** 库	内核	**Multiarch** 架构标识符
`lp64d` / `base`	glibc	Linux	`loongarch64-linux-gnu`
`lp64f` / `base`	glibc	Linux	`loongarch64-linux-gnuf32`
`lp64s` / `base`	glibc	Linux	`loongarch64-linux-gnusf`
`lp64d` / `base`	musl libc	Linux	`loongarch64-linux-musl`

ABI 类型（基础 **ABI** / **ABI** 扩展特性）	**C** 库	内核	**Multiarch** 架构标识符
`lp64f` / `base`	musl libc	Linux	`loongarch64-linux-muslf32`
`lp64s` / `base`	musl libc	Linux	`loongarch64-linux-muslsf`

C/C++ 预处理器内建宏定义

下表列举的预处理器内建宏定义并非 LoongArch 独有。 通过使用它们，用户可以尽量减少为 LoongArch 特殊编写的代码量， 而往往足以取得架构适配的预期结果。

Table 13. 非特定于 LoongArch 的 C/C++ 预处理器内建宏

名称	值	描述
`__BYTE_ORDER__`	（略）	字节序
`__FLOAT_WORD_ORDER__`	（略）	浮点数据的字节序
`__LP64__ _LP64`	（略）	ABI 是否使用 64 位通用寄存器传参，采用 LP64 数据模型
`__SIZEOF_SHORT__`	（略）	C/C++ `short` 类型位宽，单位为字节
`__SIZEOF_INT__`	（略）	C/C++ `int` 类型位宽，单位为字节
`__SIZEOF_LONG__`	（略）	C/C++ `long` 类型位宽，单位为字节
`__SIZEOF_LONG_LONG__`	（略）	C/C++ `long long` 类型位宽，单位为字节
`__SIZEOF_INT128__`	（略）	C/C++ `__int128` 类型位宽，单位为字节
`__SIZEOF_POINTER__`	（略）	C/C++ 指针类型位宽，单位为字节
`__SIZEOF_PTRDIFF_T__`	（略）	C/C++ `ptrdiff_t` 类型位宽，单位为字节
`__SIZEOF_SIZE_T__`	（略）	C/C++ `size_t` 类型位宽，单位为字节
`__SIZEOF_WINT_T__`	（略）	C/C++ `wint_t` 类型位宽，单位为字节
`__SIZEOF_WCHAR_T__`	（略）	C/C++ `wchar_t` 类型位宽，单位为字节
`__SIZEOF_FLOAT__`	（略）	C/C++ `float` 类型位宽，单位为字节
`__SIZEOF_DOUBLE__`	（略）	C/C++ `double` 类型位宽，单位为字节
`__SIZEOF_LONG_DOUBLE__`	（略）	C/C++ `long double` 类型位宽，单位为字节

在上述通用的定义之外，为了沟通那些与架构强相关的信息，仍然需要提供一些 LoongArch 平台特有的预处

理器内建宏。下表列举了这些架构相关的宏定义。

Table 14. LoongArch 架构相关 C/C++ 预处理器内建宏

名称	值	描述
`__loongarch__`	`1`	目标为龙芯架构
`__loongarch_grlen`	`64` `32`	通用寄存器位宽
`__loongarch_frlen`	`0` `32` `64`	浮点寄存器位宽（无 FPU 则为 `0` ）
`__loongarch_arch`	`"loongarch64"` `"la464"`	`-march` 指定的目标 CPU 名称， 若未指定则为编译器构建时指定的默认值
`__loongarch_tune`	`"loongarch64"` `"la464"`	`-mtune` 指定的目标 CPU 名称， 若未指定则为编译器构建时指定的默认值
`__loongarch_lp64`	未定义或 `1`	ABI 使用 64 位通用寄存器传参，采用 LP64 数据模型
`__loongarch_hard_float`	未定义或 `1`	ABI 使用浮点寄存器传参
`__loongarch_soft_float`	未定义或 `1`	ABI 不使用浮点寄存器传参
`__loongarch_single_float`	未定义或 `1`	ABI 仅使用 32 位浮点寄存器传参
`__loongarch_double_float`	未定义或 `1`	ABI 使用 64 位浮点寄存器传参

由于历史原因，最早期的 LoongArch C/C++ 编译器提供了一批 MIPS 风格的预处理器内建宏。 因为用到这些宏的旧代码可能仍在被使用，符合本规范的编译器实现可以选择提供下表所述的预处理器内建宏。

由于这些宏的命名风格、使用姿势多多少少都与上表中的宏不一致， 且使用它们并不会有额外好处，还造成额外的学习成本，因此不建议新的编译器实现这些宏。 可移植的代码不应当假定这些宏存在，也不应当使用它们。

Table 15. 用于兼容早期移植代码的 C/C++ 预处理器内建宏

名称	等价于	备注
`__loongarch64`	`__loongarch_grlen == 64`	行为类似 *`mips64`* ，当且仅当 `loongarch_grlen == 64` 时被定义。
`_LOONGARCH_ARCH`	`__loongarch_arch`	（无）
`_LOONGARCH_TUNE`	`__loongarch_tune`	（无）

名称	等价于	备注
`_LOONGARCH_SIM`	（无）	行为类似于 MIPS 架构上的 `_MIPS_SIM` ；其取值形如 `_ABILP64` （对应数据模型为 LP64 的情况）、 `_ABILP32` （对应数据模型为 ILP32 的情况；注意取值中少了个 `I` ）。
`_LOONGARCH_SZINT`	`__SIZEOF_INT__` 乘以 8	（无）
`_LOONGARCH_SZLONG`	`__SIZEOF_LONG__` 乘以 8	（无）
`_LOONGARCH_SZPTR`	`__SIZEOF_POINTER__` 乘以 8	（无）