目前推测的 LoongArch 指令 (v20210311)

https://github.com/loongson-community/docs/tree/master/unofficial/loongarch

- roriw 32 位循环右移（立即数）
- rori 原生宽度循环右移（立即数）
- ext.w 32 位位域提取
- mask 位域掩码
- fadd.w 浮点加（单精度）
- fadd.d 浮点加（双精度）
- fsub.w 浮点减（单精度）
- fsub.d 浮点减（双精度）
- fmul.w 浮点乘（单精度）
- fmul.d 浮点乘（双精度）
- fdiv.w 浮点除（单精度）
- fdiv.d 浮点除（双精度）
- slti 有符号小于立即数则置位
- sltiu 无符号小于立即数则置位
- addiw 32 位加（立即数）
- addi 原生宽度加（立即数）
- ati 最高位立即数加（Add Top Immediate）
- andi 按位与（立即数）
- ori 按位或（立即数）
- xori 按位异或（立即数）
- lui 装载高位立即数
- ahi 更高位立即数加（Add Higher Immediate）
- auipc PC-relative 高位立即数加
- lw.2 另一个 32 位读
- sw.2 另一个 32 位写
- ld.2 另一个 64 位读
- sd.2 另一个 64 位写
- lb 符号扩展 8 位读
- lh 符号扩展 16 位读
- lw 符号扩展 32 位读
- ld 64 位读
- sb 8 位写
- sh 16 位写
- sw 32 位写

注意事项

- 此文档系基于一定量的 LoongArch 二进制代码逆向工程的结果，并非来自龙芯官方。一切以龙芯或将发布的完整指令集文档为准。
- 此文档使用的助记符和语法借鉴了 RISC-V 和 MIPS 的汇编语言。部分新颖的指令助记符为生造，此时会附上简短的英文全称。

记法与约定

- 位的编号从 0 开始，0 为最低位（LSB）。
- `xxx[HI:LO]` 表示 `xxx` 的从低位 `LO`（含）到高位 `HI`（含）的位域。
- `simm` 表示该立即数域应被视作有符号数，如被用于更宽的操作，高位应作符号扩展。
- `uimm` 表示该立即数域应被视作无符号数，如被用于更宽的操作，高位应作零扩展。
- “原生宽度”指整数寄存器的宽度。
- 在 C 伪代码中，原生宽度的有符号、无符号数用 `intptr_t` 或 `uintptr_t` 类型表示。
- `PC` 是程序计数器，其值为当前执行的指令最低字节（LSB）所位于的虚拟地址。

- **UNPREDICTABLE** 含义与 MIPS 指令集手册中一致。

基本特性

- 有 32 个整数寄存器，32 个浮点寄存器。不清楚 FPU 是否必然存在。
- 目前应该仅定义了小端序（Little-endian）的 ABI 与硬件。不清楚是否存在大端序（Big-endian）硬件。
- 目前到手的二进制采用原生宽度均为 64 位。不清楚是否存在原生宽度为 32 位的 ABI 与硬件（但已推入上游的 triples 显然包含了类似 MIPS o32 与 n32 的 ABI）。三种 ABI 应该分别叫 64、32、x32。
- 推测 LoongArch 的指针宽度（虚拟地址空间大小）与原生宽度相同（除 x32 ABI 外）。
- 所有跳转指令均没有延迟槽，目前已知的跳转指令都是 PC-relative 的。
- 所有位运算、逻辑运算如未明确说明，均操作整个原生宽度。

关于 FPU：

- FPU 如果存在，按照一切常识，都应当遵循 IEEE-754 2008 规范。
- 浮点寄存器至少有 32 位，不清楚是否存在阉割版的 FPU（例如不支持双精度），也不清楚是否比 64 位还宽（复用为向量寄存器）。

本文档中假定浮点寄存器最宽为 64 位，但在叙述中不排除其更宽的可能性。

寄存器规范

整数寄存器

编号	别名	保存方	备注
0	zero	-	读取时固定为 0，写入为空操作（但不影响可能产生的副作用）
1	ra	Callee	返回地址
2	tp	-	非常可能但不确定，线程指针；用户态不应修改
3	sp	Callee	栈指针
4-11	a0-a7	Caller	入参/临时量
12-20	t0-t8	Caller	临时量

编号	别名	保存方	备注
21	gp	-	不确定，可能为全局指针；用户态不应修改
22	s9/fp	Callee	保证不被过程调用覆盖；可用作帧指针
23-31	s0-s8	Callee	保证不被过程调用覆盖

注：

- 返回值寄存器复用 `a0-a1`，用于返回最多两倍原生宽度的数据。

浮点寄存器

编号	别名	保存方	备注
0-23	f0-f23	Caller	具体用途分类不确定
24-31	fs0-fs7	Callee	保证不被过程调用覆盖

指令列表

`sext.h` 符号扩展 16 位 -> 原生宽度

语法：`sext.h rd, rj`

行为：将 rj 的低 16 位符号扩展到原生宽度，存入 rd

`sext.b` 符号扩展 8 位 -> 原生宽度

语法：`sext.b rd, rj`

行为：将 rj 的低 8 位符号扩展到原生宽度，存入 rd

`addw` 32 位加（三寄存器）

语法：`addw rd, rj, rk`

行为：`rd = ((int32_t) rj) + ((int32_t) rk)` 结果符号扩展

注：2 的补码表示中，有符号与无符号加法行为相同。

`add` 原生宽度加（三寄存器）

语法：`add rd, rj, rk`

行为：`rd = rj + rk`

注：2 的补码表示中，有符号与无符号加法行为相同。

`subw` 32 位有符号减（三寄存器）

语法：`subw rd, rj, rk`

行为：`rd = ((int32_t) rj) - ((int32_t) rk)` 结果符号扩展

`sub` 原生宽度有符号减（三寄存器）

语法：`sub rd, rj, rk`

行为：`rd = ((intptr_t) rj) - ((intptr_t) rk)`

`selnez` 非零则选择

语法：`selnez rd, rj, rk`

行为：`rd = rk ? rj : 0`

注：

可配合 `seleqz` 与 `or` 实现三目运算符的语义。欲计算 `R = C ? T : F` 可采用以下写法（会 clobber T 与 F）：

```
selnez T, T, C  // T = C ? T : 0
seleqz F, F, C  // F = C ? 0 : F
or     R, T, F  // T 与 F 有且仅有一个为全零
```

`seleqz` 为零则选择

语法：`seleqz rd, rj, rk`

行为：`rd = rk ? 0 : rj`

注：

可配合 `selnez` 与 `or` 实现三目运算符的语义。欲计算 `R = C ? T : F` 可采用以下写法（会 clobber T 与 F）：

```
selnez T, T, C  // T = C ? T : 0
```

```
seleqz F, F, C  // F = C ? 0 : F
or     R, T, F  // T 与 F 有且仅有一个为全零
```

`nor` 按位或非（三寄存器）

语法：`nor rd, rj, rk`

行为：`rd = rj | (~rk)`

注：

- 习惯上可用 `rj = zero` 起到逻辑非的作用，此时可写作 `not rd, rk`。
- 目前分析过的少量该指令对应的操作均为逻辑非，因此不能排除该指令实际上为逻辑异或非（xnor）的可能性，尽管该可能性非常渺茫。

`and` 按位与（三寄存器）

语法：`and rd, rj, rk`

行为：`rd = rj & rk`

`or` 按位或（三寄存器）

语法：`or rd, rj, rk`

行为：`rd = rj | rk`

注：习惯上，可用 rj = zero 起到寄存器间移动的作用，此时可写作伪指令 `mv rd, rk`。

`xor` 按位异或（三寄存器）

语法：`xor rd, rj, rk`

行为：`rd = rj ^ rk`

`sll` 逻辑左移（三寄存器）

语法：`sll rd, rj, rk`

行为：`rd = rj << rk`

注意：`rk < 0` 或 `rk >= 原生宽度` 时的行为未知。

`sbs` 如位域有置位则置位（Set if Bits Set）

语法：`sbs rd, rj, rk`

行为：`rd = (rj & rk) != 0 ? 1 : 0`

`srl` 逻辑右移（三寄存器）

语法：`srl rd, rj, rk`

行为：将 rj 逻辑右移 rk 位，存入 rd

注意：`rk < 0` 或 `rk >= 原生宽度` 时的行为未知。

`mul` 原生宽度乘（三寄存器）

语法：`mul rd, rj, rk`

行为：rj 与 rk 相乘，低原生宽度位存入 rd

注：2 的补码表示中，有符号与无符号乘法行为相同。

`syscall` 系统调用

语法：`syscall`

行为：陷入内核态，执行系统调用后返回到下一条指令继续执行

注：

- 目前到手的 Linux 二进制分析结果显示，系统调用号置于 `a7`，参数按顺序置于 `a0`-`a6`，返回值置于 `a0`，其他寄存器值显然不变。

`ofs.w` 记录偏移量（4 字节长）（Offset by Words）

语法：`ofs.w rd, rj, rk`

行为：`rd = 4 * rj + rk`

`slliw` 32 位逻辑左移（立即数）

语法：`slliw rd, rj, uimm5`

行为：rj 低 32 位逻辑左移 uimm5 位，存入 rd 低 32 位，rd 其余位置零（如有）

注：习惯上，可用 `slliw rd, rj, 0` 起到零扩展 32 位到原生宽度的作用。

`slli` 原生宽度逻辑左移（立即数）

语法：`slli rd, rj, uimm6`

行为：`rd = rj << uimm6`

`srliw` 32 位逻辑右移（立即数）

语法：`srliw rd, rj, uimm5`

行为：rj 低 32 位逻辑右移 uimm5 位，存入 rd 低 32 位，rd 其余位置零（如有）

`srli` 原生宽度逻辑右移（立即数）

语法：`srli rd, rj, uimm6`

行为：`rd = ((uintptr_t) rj) >> uimm6`

`sraiw` 32 位算术右移（立即数）

语法：`sraiw rd, rj, uimm5`

行为：rj 低 32 位算术右移 uimm5 位，存入 rd 低 32 位，rd 其余位置为符号扩展（如有）

`srai` 原生宽度算术右移（立即数）

语法：`srai rd, rj, uimm6`

行为：`rd = rj >> uimm6`

`roriw` 32 位循环右移（立即数）

语法：`roriw rd, rj, uimm5`

行为：rj 低 32 位循环右移 uimm5 位，存入 rd 低 32 位，rd 其余位置零（如有）

`rori` 原生宽度循环右移（立即数）

语法：`rori rd, rj, uimm6`

行为：rj 循环右移 uimm6 位，存入 rd

`ext.w` 32 位位域提取

语法：`ext.w rd, rj, uimm5a, uimm5b`

行为：提取 rj 低 32 位从第 uimm5a 位（含）到第 uimm5b 位（含）的位域内容，存入 rd。

即 `rd = (((uint32_t) rj) >> uimm5a) & ((1 << (uimm5b + 1) - 1)` 高位零扩展

注：

- `uimm5a > uimm5b` 时的行为未知，按照常识，应该为 **UNPREDICTABLE**。

`mask` 位域掩码

语法：`mask rd, rj, uimm6a, uimm6b`

行为：只保留 rj 从第 uimm6a 位（含）到第 uimm6b 位（含）的位域内容，其余位置零，存入 rd。

即 `rd = rj & ((1 << (uimm6b + 1)) - (1 << uimm6a))`

注：

- `uimm6a > uimm6b` 时的行为未知，按照常识，应该为 **UNPREDICTABLE**。

`fadd.w` 浮点加（单精度）

语法：`fadd.w fd, fj, fk`

行为：`fd[31:0] = fj[31:0] + fk[31:0]` 高位 **UNPREDICTABLE**

`fadd.d` 浮点加（双精度）

语法：`fadd.d fd, fj, fk`

行为：`fd[63:0] = fj[63:0] + fk[63:0]` 高位 **UNPREDICTABLE**

`fsub.w` 浮点减（单精度）

语法：`fsub.w fd, fj, fk`

行为：`fd[31:0] = fj[31:0] - fk[31:0]` 高位 **UNPREDICTABLE**

`fsub.d` 浮点减（双精度）

语法：`fsub.d fd, fj, fk`

行为：`fd[63:0] = fj[63:0] - fk[63:0]` 高位 **UNPREDICTABLE**

`fmul.w` 浮点乘（单精度）

语法：`fmul.w fd, fj, fk`

行为：`fd[31:0] = fj[31:0] * fk[31:0]` 高位 **UNPREDICTABLE**

`fmul.d` 浮点乘（双精度）

语法：`fmul.d fd, fj, fk`

行为：`fd[63:0] = fj[63:0] * fk[63:0]` 高位 **UNPREDICTABLE**

`fdiv.w` 浮点除（单精度）

语法：`fdiv.w fd, fj, fk`

行为：`fd[31:0] = fj[31:0] / fk[31:0]` 高位 **UNPREDICTABLE**

`fdiv.d` 浮点除（双精度）

语法：`fdiv.d fd, fj, fk`

行为：`fd[63:0] = fj[63:0] / fk[63:0]` 高位 **UNPREDICTABLE**

`slti` 有符号小于立即数则置位

语法：`slti rd, rj, simm12`

行为：`rd = ((intptr_t) rj) < simm12 ? 1 : 0`

`sltiu` 无符号小于立即数则置位

语法：`sltiu rd, rj, uimm12`

行为：`rd = ((uintptr_t) rj) < uimm12 ? 1 : 0`

`addiw` 32 位加（立即数）

语法：`addiw rd, rj, simm12`

行为：`rd = ((int32_t) rj) + simm12` 结果符号扩展

注：

- 习惯上，用 `rj = zero` 的该指令表达装载小立即数的语义，此时可写作伪指令 `li rd, simm12`。

`addi` 原生宽度加（立即数）

语法：`addi rd, rj, simm12`

行为：`rd = rj + simm12`

`ati` 最高位立即数加（Add Top Immediate）

语法：`ati rd, rj, uimm12`

行为：`rd = rj + (uimm12 << 52)`

注：

- 本指令与 MIPS R6 的 `dati` 指令类似，区别在于立即数域的宽度和装载的位域。
- 本指令可按需与 `lui ahi ori` 配合，以构造任意的 64 位立即数。

`andi` 按位与（立即数）

语法：`andi rd, rj, uimm12`

行为：`rd = rj & uimm12`

注：

- 习惯上，使用 `andi zero, zero, 0x0` 表示空操作，此时可写作 `nop`。

`ori` 按位或（立即数）

语法：`ori rd, rj, uimm12`

行为：`rd = rj | uimm12`

注：

- 本指令可按需与 `lui ahi ati` 配合，以构造任意的 64 位立即数。

`xori` 按位异或（立即数）

语法：`xori rd, rj, uimm12`

行为：`rd = rj ^ uimm12`

`lui` 装载高位立即数

语法：`lui rd, uimm20`

行为：`rd = uimm20 << 12` 高位零扩展

注：

- 本指令与 MIPS、RISC-V 的同名指令类似，区别在于立即数域的宽度和装载的位域。
- 本指令可按需与 `ahi ati ori` 配合，以构造任意的 64 位立即数。

`ahi` 更高位立即数加（Add Higher Immediate）

语法：`ahi rd, uimm20`

行为：`rd += uimm20 << 32`

注：

- 本指令与 MIPS R6 的 `dahi` 指令类似，区别在于立即数域的宽度和装载的位域。
- 本指令可按需与 `lui ati ori` 配合，以构造任意的 64 位立即数。

`auipc` PC-relative 高位立即数加

语法：`auipc rd, uimm20`

行为：`rd = PC + uimm20 << 12`

`lw.2` 另一个 32 位读

语法：`lw.2 rd, simm14(rj)`

行为：**不确定**，大致为 `rd = *((int32_t *)(rj + (simm14 << 2)))` 高位可能为符号扩展

注：

- 偏移量为立即数左移 2 位，因此实质上可表示的偏移量为 16 位宽的 4 的倍数。
- 与 `lw` 的其他区别未知。

`sw.2` 另一个 32 位写

语法：`sw.2 rd, simm14(rj)`

行为：**不确定**，大致为 `*((int32_t *)(rj + (simm14 << 2))) = rd`

注：

- 偏移量为立即数左移 2 位，因此实质上可表示的偏移量为 16 位宽的 4 的倍数。
- 与 `sw` 的其他区别未知。

`ld.2` 另一个 64 位读

语法：`ld.2 rd, simm14(rj)`

行为：**不确定**，大致为 `rd = *((int64_t *)(rj + (simm14 << 2)))` 高位可能为符号扩展

注：

- 偏移量为立即数左移 2 位，因此实质上可表示的偏移量为 16 位宽的 4 的倍数。
- 与 `ld` 的其他区别未知。

`sd.2` 另一个 64 位写

语法：`sd.2 rd, simm14(rj)`

行为：**不确定**，大致为 `*((int64_t *)(rj + (simm14 << 2))) = rd`

注：

- 偏移量为立即数左移 2 位，因此实质上可表示的偏移量为 16 位宽的 4 的倍数。
- 与 `sd` 的其他区别未知。

`lb` 符号扩展 8 位读

语法：`lb rd, simm12(rj)`

行为：`rd = *((int8_t *)(rj + simm12))`（高位符号扩展）

`lh` 符号扩展 16 位读

语法：`lh rd, simm12(rj)`

行为：`rd = *((int16_t *)(rj + simm12))`（高位符号扩展）

`lw` 符号扩展 32 位读

语法：`lw rd, simm12(rj)`

行为：`rd = *((int32_t *)(rj + simm12))`（高位符号扩展（如原生宽度大于 32 位））

`ld` 64 位读

语法：`ld rd, simm12(rj)`

行为：`rd = *((int64_t *)(rj + simm12))`

`sb` 8 位写

语法：`sb rd, simm12(rj)`

行为：`*((int8_t *)(rj + simm12)) = (int8_t)rd`

`sh` 16 位写

语法：`sh rd, simm12(rj)`

行为：`*((int16_t *)(rj + simm12)) = (int16_t)rd`

`sw` 32 位写

语法：`sw rd, simm12(rj)`

行为：`*((int32_t *)(rj + simm12)) = (int32_t)rd`

`sd` 64 位写

语法：`sd rd, simm12(rj)`

行为：`*((int64_t *)(rj + simm12)) = (int64_t)rd`

`lbu` 零扩展 8 位读

语法：`lbu rd, simm12(rj)`

行为：`rd = *((uint8_t *)(rj + simm12))`（高位零扩展）

`lhu` 零扩展 16 位读

语法：`lhu rd, simm12(rj)`

行为：`rd = *((uint16_t *)(rj + simm12))`（高位零扩展）

`flw` 浮点 32 位读

语法：`flw fd, simm12(rj)`

行为：`fd[31:0] = *((int32_t *)(rj + simm12))` 高位 **UNPREDICTABLE**

`fsw` 浮点 32 位写

语法：`fsw fd, simm12(rj)`

行为：`*((int32_t *)(rj + simm12)) = fd[31:0]`

`fld` 浮点 64 位读

语法：`fld fd, simm12(rj)`

行为：`fd[63:0] = *((int64_t *)(rj + simm12))` 高位 **UNPREDICTABLE**

`fsd` 浮点 64 位写

语法：`fsd fd, simm12(rj)`

行为：`*((int64_t *)(rj + simm12)) = fd[63:0]`

`beqz` 为零则跳

语法：`beqz rj, label`

行为：`if (rj == 0) PC += simm21 * 4`

`bnez` 非零则跳

语法：`bnez rj, LABEL`

行为：`if (rj != 0) PC += simm21 * 4`

`jalr` 跳并链接（寄存器）

语法：`jalr rd, rj`

行为：`rd = PC + 4; PC = rj`

注：

- 按照跳转指令的编码规律，所有其他跳转指令都带一个偏移量，那么该指令的未知位域也应该是偏移量（这样一来就与 RISC-V 的 `jalr` 用法完全相同），但目前观测到的该指令的该域均为全 0，因此不能排除其实际上为保留位的可能性。
- 如 `rd = ra` 则为常规的过程调用，汇编上可简化为 `jalr rj`。
- 如 `rd = zero` 则不会记录返回地址，此时汇编上可简化为 `jr rj`。
- `jalr zero, ra` 汇编上可简化为 `ret`。

`j` 跳

语法：`j LABEL`

行为：`PC += simm25 * 4`

注：

- 目前观测到的该指令标 ? 的位均保持与立即数最高位相同（即符号扩展）。因此尚不能确定该指令的立即数究竟为 25 位（与老胡 PPT 保持一致）还是 26 位。

`jal` 跳并链接（立即数）

语法：`jal LABEL`

行为：`R1 = PC + 4; PC += simm25 * 4`

注：

- 隐式指定的链接寄存器为 r1（ra）。
- 目前观测到的该指令标 ? 的位均保持与立即数最高位相同（即符号扩展）。因此尚不能确定该指令的立即数究竟为 25 位（与老胡 PPT 保持一致）还是 26 位。

`beq` 相等则跳

语法：`beq rd, rj, LABEL`

行为：`if (rd == rj) PC += simm16 * 4`

`bne` 不等则跳

语法：`bne rd, rj, LABEL`

行为：`if (rd != rj) PC += simm16 * 4`

`bgt` 有符号大于则跳

语法：`bgt rd, rj, LABEL`

行为：`if ((intptr_t)rd > (intptr_t)rj) PC += simm16 * 4`

`ble` 有符号小于等于则跳

语法：`ble rd, rj, LABEL`

行为：`if ((intptr_t)rd <= (intptr_t)rj) PC += simm16 * 4`

`bgtu` 无符号大于则跳

语法：`bgtu rd, rj, LABEL`

行为：`if ((uintptr_t)rd > (uintptr_t)rj) PC += simm16 * 4`

`bleu` 无符号小于等于则跳

语法：`bleu rd, rj, LABEL`

行为：`if ((uintptr_t)rd <= (uintptr_t)rj) PC += simm16 * 4`