Assembly Language Programming Guide

LoongArch

Loongson Technology Corporation Limited

Version 0.9(draft), 2023-08-15

Contents

Preamble

This is the official documentation of the `Assembly Language Programming Guide` LoongArch for the `LoongArch` Architecture.

Preface

This preface introduces the `LoongArch` Assembly Language Programming Guide.

This document aims to provide guidance to assembly programmers targeting the standard `LoongArch` assembly language, which common open-source assemblers like `GNU` as and `LLVM`'s assembler support. Other assemblers might not support the same directives or pseudo instructions; their dialects are outside the scope of this document.

About This Book

This book provides tutorial and reference information on the `LoongArch` Assembly Language Program. This includes `LoongArch` data type, register, address, and inline assemblers in the C and C++ compilers. It describes the assembly language mnemonics, the pseudo-instructions, the macros, and directives available to assembly language programmers. It also describes the format of the `LoongArch` ELF file.

Intended Audience

This book is written for all developers who are producing applications using compilation tools. It assumes that you are an experienced assembly language developer and that you are familiar with the `LoongArch` architecture.

Using This Book

NOTE For more information, please refer to:

LoongArch Architecture manual

LoongArch Application Binary Interface manual

Change History

Table 1. Change History Table

Date	Version	Note
2023-08-15	V0.9	`Draft version`.

CHAPTER 1. Introduction

This chapter introduces the `LoongArch` Architecture.

1.1 `LoongArch` Architecture Overview

`LoongArch` has the typical characteristics of RISC. `LoongArch` instructions are of fixed size and have regular instruction formats. Most of the instructions have two source operands and one destination operand. `LoongArch` is a load-store architecture; this means only the load/store instructions can access memory the operands of the other instructions are within the processor core or the immediate number in the instruction opcode.

`LoongArch` is divided into two versions, the 32-bit version (`LA32`) and the 64-bit version (`LA64`). `LA64` applications are “application-level backward binary compatibility” with `LA32` applications. That means `LA32` applications can run directly on the machine compatible with `LA64`, but the behavior of system softwares (such as the kernel) on the machine compatible with `LA32` is not guaranteed to be the same as on the machine compatible with `LA64`.

`LoongArch` is composed of a basic part (`Loongson Base`) and an expanded part, as shown in the figure. The expansion part includes Loongson Binary Translation (`LBT`), Loongson VirtualiZation (`LVZ`), Loongson SIMD EXtension (`LSX`), and Loongson Advanced SIMD EXtension(`LASX`).

1.2 Introduction to `LoongArch` Series CPU

This section mainly introduces the CPU models that are compatible with the `LoongArch` assembly program design guide manual.

Tip

Retain this section

CHAPTER 2. Data Representation

2.1 `LoongArch` Data Representation

There are five data types for basic integer instruction operations, namely:

- `bit` (short for 1 bit)
- `byte` (short for `B`, length 8 bits)
- `halfword` (short for `H`, length 16 bits)
- `word` (short for `W`, length 32 bits)
- `doubleword` (short for `D`, length 64 bits)

Under the `LA32` architecture, there are no integer instructions for manipulating doublewords. Byte, half word, and word data types are encoded using binary complement.

Floating-point data types include single precision floating-point numbers and double precision floating-point numbers, both of which follow the definition in the `IEEE 754-2008 standard` specification.

2.2 Sign Extend & Zero Extend

In `LoongArch` computing instructions, immediate values require both `Sign-Extend` and `Zero-Extend`.

- 32-bit sign extend: Fills the high 32-n bits of an n-bit immediate with the highest bit of the immediate.
- 64 bit sign extend: Fills the high 64-n bits of an n-bit immediate with the highest bit of the immediate.
- 32-bit unsigned extend: Fill the high 32-n bits of the n-bit immediate with 0.
- 64 bit unsigned extend: Fill the high 64-n bits of the n-bit immediate with 0.

In the subsequent instructions, **SignExtend**(x, len) represents sign extend, x represents the extend object, and len represents the number of digits in the final length of x after sign extend. **ZeroExtend**(x, len) uses 0 to extend x to len.

CHAPTER 3. Register

This chapter describes the naming and usage conventions that the assembler applies to the CPU and FPU registers. For detailed information, please refer to the `LoongArch architecture manual` and the `LoongArch Application Binary Interface manual`.

3.1 General Purpose Register

There are thirty-two General Registers (`GR`), denoted as `$r0`-`$r31`, where the value of register `$r0` is always **0**. The bit width of `GR` is denoted as **GRLEN**. The bit width of `GR` on the `LA32` architecture is 32 bits, while on the `LA64` architecture, the bit width of `GR` is 64 bits. The basic integer instruction has an orthogonal relationship with the `GR`. From the `LoongArch` architectural perspective, any register operand in these instructions can take any of the 32 `GR` s. The only exception is that the destination register implicit in the `BL` instruction must be the first register `$r1`(`$ra`).

3.2 PC Register

There is only one `PC` that records the address of the current instruction. The `PC` register cannot be directly modified by instructions, it can only be indirectly modified by transfer instructions, exception trapping, and exception return instructions. `PC` registers can be directly read as source operands for some non transfer instructions. The width of `PC` always matches the width of `GR`.

3.3 Floating-Point Register

`LoongArch` has a total of 32 `FRs`, denoted as `$f0`-`$f31`, each of which can be read and written. The bit width of `FR` is 32 bits, which only implements floating-point instructions that operate on single precision floating-point numbers and word integers. Usually, the bit width of `FR` is 64 bits, regardless of whether it is an `LA32` or `LA64` architecture. The basic floating-point instruction has an orthogonal relationship with floating-point registers, meaning that from an architectural perspective, any floating-point register operand in these instructions can take any of the 32 `FR` s. When a single floating-point number or word integer is recorded in a floating-point register, the data always load in the [`31`: `0`] bit of the floating-point register, and the [`63`: `32`] bit of the floating-point register can be any value.

3.4 Condition Flag Register

`LoongArch` has a total of 8 `CFR`, denoted as `$fcc0`-`$fcc7`, each of which can be read and written. The bit width of `CFR` is **1** bit. The result of the floating-point comparison will be written to the condition flag register, set to **1** when the comparison result is true, otherwise set to **0**. The judgment condition for floating-point branch instructions comes from the condition flag register.

3.5 Floating-Point Control Status Register

`LoongArch` has a total of 4 `FCSRs`, denoted as `$fcsr0`-`$fcsr3`, with a bitwidth of 32 bits. Among them, `$fcsr0`-`$fcsr3` are aliases for the central domain of `$fcsr0`, that is, accessing `$fcsr0`-`$fcsr3` is actually accessing certain domains of `$fcsr0`. When the software writes `$fcsr0`-`$fcsr3`, the corresponding fields in `$fcsr0` are modified while the remaining bits remain unchanged.

3.6 Register Usage Convention

`LoongArch ABI` has established usage conventions for the functions of registers.

3.6.1 Generic Register Alias

Name	Alias	Usage	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
`$r22`	`$s0` - `$s8`	Static registers	YES

3.6.2 Floating-Point Register Alias

Name	Alias	Usage	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
`$f23` - `$f31`	`$s0` - `$s8`	Static registers	YES

3.6.3 Register Function Introduction

3.6.3.1 Zero Register

The zero register, `$r0`, is a constant register that always returns **0** when read, regardless of what is written. To take the opposite number of a variable, you can use the zero register and the register where the variable is located to subtract, reducing the loading operation on the immediate **0**. `$zero` can complete some synthesis instructions, such as the macro instruction move in `LoongArch`.

```
# macro instruction                  # instruction
move  $t0, $t1                       # or     $t0, $t1, $zero
```

3.6.3.2 Function Call Register

When `LoongArch`(`LP64D ABI`) makes a function call, registers `$a0` - `$a7` are used to pass 8 integer or pointer parameters. Registers `$fa0` - `$fa7` are used to pass 8 float-point parameters. Among them, `$a0` / `$fa0` and `$a1` / `$fa1` are also used to return values, and register `$ra` stores the return address.

3.6.3.3 Temporary Register & Save Register

Temporary registers are mainly used as temporary variables. When using these temporary registers in a function, there is no need to consider saving old values.

To save registers, the current function needs to ensure that the values of these registers are consistent with the function entry when the function returns. If one or more registers from `$s0` to `$s8` are to be used within a function, their old values need to be stored on the stack. And load the old value in the save register before the function returns.

Regarding the physical mapping of registers `$rd`, `$rj`, `$rk` in the assembly instruction description, the physical registers that can be used when writing assembly are as follows:

Name	Alias	Note
`$r12` - `$r20`	`$t0` - `$t8`	The role of temporary variables in functions does not require consideration of the preservation of old values.
`$r23` - `$r31`	`$s0` - `$s8`	The function of a temporary variable requires storing its old value on the stack before use and restoring the old value before the function returns.
`$f8` - `$f23`	`$ft0` - `$ft15`	The role of temporary variables in functions does not require consideration of the preservation of old values.
`$f23` - `$f31`	`$s0` - `$s8`	The role of temporary variables in functions does not require consideration of the preservation of old values.

3.6.3.4 TP Register

The `$tp` register is used to support thread local storage. `TLS` is a storage method for thread local variables, ensuring that variables are globally accessible within the thread, but cannot be accessed by other threads. `LoongArch ABI` specifically occupies a register to point to the `TLS` region of the current thread, with the aim of quickly locating and accessing variables within this region, and improving program execution efficiency. The user program is not recomened to modify this register.

3.6.3.5 Function Stack & SP FP Register

In a data structure, a stack is a dynamic storage space that only allows insertion and deletion operations on the same end. According to the principle of first in, last out, data is stored, where the data that enters first is pressed at the bottom of the stack, and the data that enters last is at the top of the stack. The function stack is mainly used to store local variables and related registers within a function, but its usage is not as strict as the stack in the data structure. Each function has different stack space sizes depending on the number of parameters and local variables.

The frame pointer of a function with immutable stack frames is `$sp`; The frame pointer of the variable stack frame function is `$fp` . In the entire function, the determination of the frame pointer, the storage register, and the backup of `$ra` (for non leaf functions) are in a basic block called prologue. Once the frame pointer is determined, it will not change until the function returns.

CHAPTER 4. Addressing

This chapter describes the formats that you can use to specify addresses. The machine uses a byte addressing scheme.

4.1 Address Range

The memory address space on `LoongArch` is a continuous linear address space, which is addressed in bytes.

In `LA32`, the specification of the memory address space that application can access is: 0-2^{31}-1.

In `LA64`, the range of memory address space accessible by application is: 0-$2^{VALEN-1}$-1. Generally `VALEN` is in the range of [`40`,`48`]. Application can determine the specific value of `VALEN` by executing the `CPUCFG` instruction to read the `VALEN` field of the 0x1 configuration word.

When the virtual address of the instruction fetch or memory access instruction in the application exceeds the above range, ADEF or ADEM will be triggered.

4.2 Addressing Method

`Register`

- **Format** : `reg`[R1] = `reg`[R1] + `reg`[R2]
- **Instruction example** : `ADD R1`, `R2`
- **When to use** : Value in register

`Immediate`

- **Format** : `reg`[R1] = `reg`[R1] + 2
- **Instruction example** : `ADD R1`, `#2`
- **When to use** : For constants

`Displacement`

- **Format** : `reg`[R1] = `reg`[R1] + **mem**[`reg`[R2] + 100]
- **Instruction example** : `ADD R1`, `100(R2)`
- **When to use** : Accessing Local Variables (Analog Register Indirect, Direct Addressing)

`Indexed`

- **Format** : `reg`[R1] = `reg`[R1] + **mem**[`reg`[R2] + `reg`[R3]]
- **Instruction example** : `ADD R1`, `(R2+R3)`
- **When to use** : Used for array addressing, with R1 as the array base address and R2 as the index value.

CHAPTER 5. Assembler Directives

This chapter mainly describes assembler commands and assembler instructions.

For more command line parameters of the assembler, please refer to the documentation for the assembler on the `GCC` and `LLVM` official websites.

5.1 Symbol Definition Directives

If an identifier is not defined to the assembler (only referenced), the assembler assumes that the identifier is an external symbol. The assembler treats the identifier like a `.globl` pseudo-operation. If the identifier is defined to the assembler and the identifier has not been specified as global, the assembler assumes that the identifier is a local symbol.

5.1.1 Set Symbol Type

The assembler directive that defines a symbol type is `.type`, which is often followed by types `@function` and `@object`, respectively, indicating that the current symbol is a function and a variable.

```
.type   test,       @function
.type   var,        @object
```

5.1.2 Set Symbol Size

The assembler directive `.size` name, expression is used to set the size of symbols, where name is the symbol name. When setting the variable size, expression is a positive integer. When setting the function size, expression is usually a "`.-name`" expression.

```
.size  short,       2
.size  main,        .-main
```

5.1.3 Set Symbol Align

The assembler directive `.align` expr is used to specify the alignment of symbols, where expr is a positive integer used to indicate the alignment of subsequent data storage addresses in the target file.

```
.align expr
```

Pad the location counter (in the current subsection) to a particular storage boundary. Expr is a positive integer indicating the alignment of the data storage address in the target file. In `LoongArch`:

```
.align 4                     # Align to the 16 bytes
```

In order to simplify the hardware design between processors and memory systems, many computer systems impose restrictions on the address of memory access operations, requiring that the address of the accessed memory must be a multiple of the data type, i.e. naturally aligned.

- To read or write a `halfword` (`2 bytes`) of data from memory, the access address must be a multiple of **2**.
- To read or write a `word` (`4 bytes`) of data from memory, the access address must be a multiple of **4**.
- To read or write a `doubleword` (`8 bytes`) of data from memory, the access address must be a multiple of **8**.

`$r5` = `0x120000000`, `ld.w $r4`, `$r5`, `0x3` Address `0x120000003` Not divisible by **4**, therefore non aligned access. `ld.w $r4`, `$r5`, `0x8` Address `0x120000008` It can be divided by **4**, so it is an aligned access. `ld.d $r4`, `$r5`, `0x5` Address `0x120000005` Not divisible by **4**, therefore non aligned access.

`LoongArch` supports hardware processing of non aligned memory data access. Although there is non aligned access in the above example, the processor can still function properly and obtain correct results without throwing non aligned exceptions. However, for better performance, it is recommended to align the data as much as possible. Generally, the compiler will automatically align the data.

The assembler directive `.align` expr is used to specify the alignment of symbols, where expr is a positive integer used to indicate the alignment of subsequent data storage addresses in the target file.

```
.align expr
```

Expr is a positive integer indicating the alignment of the data storage address in the target file. In `LoongArch` :

```
.align 4          # Align to the 16byte
```

In the above command, different architectures of .align have different definitions of expr, and two other variants, `.balgin` and `.p2align`, can be used. The instruction `.balgin` 4 represents 4-byte alignment in any architecture.

- .align https://sourceware.org/binutils/docs/as/Align.html

Two other variants, `.balign` and `.p2align`, can be used. Regarding `.balgin` and `.p2align` :

- .balign https://sourceware.org/binutils/docs/as/Balign.html
- .p2align https://sourceware.org/binutils/docs/as/p2align.html

5.1.4 Set Symbol Location

When defining a variable or function symbol in the assembly source file, its scope should also be declared to identify the scope of the current symbol. By default, the current symbol scope is not specified, and the symbol scope is visible within the current assembly source file. Other compiler instructions that need to be used in other situations include:

```
.globl symbol     # Global visibility
.global symbol    # Global visibility
.common symbol    # Universal symbol, similar to uninitialized global variables
.local symbol     # Similar to uninitialized local static variables
```

`.globl` / `.global` specifies the symbol symbol symbol as a global variable or non-static member function, which is globally visible and visible to other source files in the linker.

The `.common` declaration is a universal symbol, similar to an uninitialized global variable in C language. A universal symbol with the same name that appears in multiple assembly source files may be merged during the compiler's compilation phase, resulting in the preservation of the one with the largest footprint.

`.local` is used to declare an uninitialized local static variable definition similar to that in a language.

5.2 Logic Control Directives

The assembler will translate the assembly instructions into machine instructions and store them in the target file. The assembler directive is different from the assembly instruction, which is used to guide the assembler on how to define variables and functions, and how to store assembly instructions in the target file. The assembler directive is the instruction that guides the work of the assembler.

5.2.1 Set Symbol Data Storage Segment

Use assembler directives such as .data subsection and `.text` subsection in the assembly source file to specify the data and code segments where the following statements are stored in the target file. When it is necessary to specify a more refined segment type. You can use the `.section` name.

```
.data   # Specify the data segment to store the next data in the target file
str:
     .ascii "test\000"
var:
     .word  10
.text   # Specify the code snippet for storing the next data in the target file
add:
```

5.2.2 Constant Declaration

The assembler directive `.set` symbol, expression is used for constant settings.

```
.set    FLAG, 0
.equ    FLAG, 0
```

5.2.3 Conditional Compilation

In conjunction with constant settings, use the assembler directives `.if`, `.else`, and `.endif` to achieve conditional compilation.

```
      .set              FLAG, 0
.LC0:
      .ascii            "test1\000"
.LC1:
      .ascii            "test2\000"
main:
      addi.d            $sp,    $sp,    -8
      st.d              $ra,    $sp,    0
.if FLAG == 1
      la.local          $r4,    .LC0
.else
      la.local          $r4,    .LC1
.endif
      bl                %plt(puts)
```

For conditional compilation, preprocessing commands such as `#ifdef`, `#else`, and `#endif` in C language can also be directly used in the asm source file. When using the C language pre-processing command, the assembly source file cannot be compiled directly with the assembler, but needs to call the compiler's preprocessing tool in advance to translate the pre-processing command.

Table 2. Condition

Command	Function
`.ifdef` symbol	If the symbol symbol has already been defined, assemble the following code.
`.ifndef` symbol	If the symbol symbol has not been defined before, assemble the following code, which is equivalent to. ifnotdef symbol.
`.ifc` str1, str2	If two strings are the same, assemble the following code, which is equivalent to .ifeqs str1, str2.
`.ifnc` str1, str2	If two strings are different, assemble the following code.
`.ifeq` expression	If the expression value is 0, assemble the following code.
`.ifge` expression	If the expression value is greater than or equal to 0, assemble the following code.
`.ifgt` expression	If the expression value is greater than 0, assemble the following code.
`.ifle` expression	If the expression value is less than or equal to 0, assemble the following code.
`.iflt` expression	If the expression value is less than 0, assemble the following code.

5.2.4 Compile Debug

The information output instructions that can be used during the compilation process of the assembler include:

- `.print` string
- `.fail` expression
- `.error` string & `.err`

5.2.4.1 .print string

Will cause the assembler to output a string on standard output.

5.2.4.2 .fail expression

An error or warning message will be generated, and when the expression value is greater than or equal to 500, the assembler will output a warning message; When the value of expression is less than 500, the assembler will output an error message. The default value of expression is 0, and the `.fail` parameter can be written directly and left blank.

5.2.4.3 .error string & .err

`.err` can output a default error message during the assembly process. If you want to customize the error message type, you can use the `.error` string directive.

5.2.5 File Include

There are two ways to reference other files in the assembly source file. One method is to use the assembler directive `.include` '`file`', which defaults to the current directory as the reference file path. When the path of the referenced file is not in the same directory, the search path can be controlled through the compiler's command-line option parameter '`- I`'; Another method is to use the C language preprocessing command `#include`, which requires the assembler file to be `.S` and preprocessed through the front-end preprocessing tool.

```
#ref.S
       .text
test:
       .print "test"
       .jr $r1
#main.S
       .include "ref.S"
```

5.2.6 Loop Unrolling

assembler directives `.rept` count and `.endr` can be used to loop through their internal statements count times.

```
.rept 3
nop
.endr
```

The above code is equivalent to notifying the assembler to generate three nop instructions in the target file. When it is necessary to insert different numbers of your nop instructions according to the actual situation to achieve address alignment, using the loop unrolling instruction is very convenient.

assembler directives `.irp` symbol, values and `.endr` can loop through its internal statements.

```
.irp n,4,5,6,7,8,9,10,11,12
st.d $r\n, $sp, \n*8
.endr
```

is equivalent to :

```
st.d $r4, $sp, 0x20
st.d $r5, $sp, 0x28
st.d $r6, $sp, 0x30
st.d $r7, $sp, 0x38
st.d $r8, $sp, 0x40
st.d $r9, $sp, 0x48
st.d $r10, $sp, 0x50
st.d $r11, $sp, 0x58
st.d $r12, $sp, 0x60
```

5.2.7 Macro Define

The assembler directive `.macro` name args is similar in function to the macro definition function in C language, where name is the macro name, args is the parameter, and ends with `.endm` .

For example, implementing a macro definition that can generate different numbers of nop instructions based on different parameters:

```
.text
.macro INSERT_NOP a
.rept \a
nop
.endr
.endm
```

Here, `.text` is used to indicate that the following instructions are stored in the code snippet of the target file. Macro name is `INSERT_NOP`, parameter is a. The format for using parameters in the macro definition body is "parameter", such as a. The parameters of the macro can be 0 or multiple. When there are multiple parameters, use commas or spaces to separate them. When the program is in use, simply call the macro.

```
INSERT_NOP 3
INSERT_NOP 7
```

CHAPTER 6. ASM File & ELF File

This chapter mainly discusses the format of the `Loongarch` assembler source file and the ELF file.

```
.data
.LC0:
    .ascii          "test\0"
    .text
    .align          2
    .globl          main
    .type           main,   @function

main:
    addi.d          $sp,    $sp,    -32
    st.d            $ra,    $sp,    24
    st.d            $fp,    $sp,    16
    addi.d          $fp,    $sp,    32
    or              $t0,    $a0,    $zero
    st.d            $a1,    $fp,    -32
    slli.w          $t0,    $t0,    0
    st.w            $t0,    $fp,    -20
    la.local        $a0,    .LC0
    bl              %plt(puts)
    or              $t0,    $zero,  $zero
    or              $a0,    $t0,    $zero
    ld.d            $ra,    $sp,    24
    ld.d            $fp,    $sp,    16
    addi.d          $sp,    $sp,    32
    jr              $ra
    .size           main, .-main
    .section        .note.GNU-stack, "", @progbits
```

A `relocatable file` holds code and data suitable for linking with other object files to create an executable or a shared object file.

An `executable file` holds a program suitable for execution; the file specifies how exec(LoongArch) creates a program's process image.

A `shared object file` holds code and data suitable for linking in two contexts. First, the link editor see ld(LoongArch) processes the shared object file with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image.

For more information on ELF files, please refer to SysV gABI and LoongArch Application Binary Interface manual.

CHAPTER 7. Inline Assembly

The asm keyword allows you to embed assembler instructions within C code. Compiler provides two forms of inline asm statements. A basic assembly statement is one with no operands, and an Extended assembly statement which includes one or more operands to interact with C variables.

In the process of program development, assembly language can achieve functions that cannot be achieved by C language in certain situations, which requires the use of extended asm.

7.1 Basic Assembly

Refer to the "Instruction Set" section in Programming Manual document to get details of all instructions.

Here is a example:

Inline assembly code is used to write pure assembly code in a `C/C++` program:

```
int main()
{
    asm volatile("move $r23, $r24");
    asm volatile("addi $r23, $r24, 1");
}
```

Use in block of instructions, note the `\n\t` at the end of each instruction:

```
int main()
{
    asm volatile("move $r23, $r24\n\t"
                 "addi $r23, $r24, 1\n\t");
}
```

Usage :

We will write a simple code to:

- Load values from 2 addresses `0x20001000` and `0x20001004`
- Store the sum of those numbers to a new address `0x20001008`

```
int main(void) {
  __asm volatile(
    "ld.d  $r23, $r0,  0x120001000\n\t"
    "ld.d  $r24, $r0,  0x120001004\n\t"
    "add.d $r25, $r23, $r24\n\t"
    "st.d  $r25, $r0,  0x120001008\n\t");
}
```

7.2 Extended Assembly

The Inline Assembly full syntax is :

```
asm volatile ( AssemblerTemplate
             : OutputOperands
             : InputOperands
             : Clobbers )
```

```
asm goto(AssemblerTemplate
       : OutputOperands
       : InputOperands
       : Clobbers
       : GotoLabels)
```

`AssemblerTemplate` :

- This is a literal string that is the template for the assembler code. It is a combination of fixed text and tokens that refer to the input, output, and goto parameters.

```
asm volatile("move $r23, $r24"); // is the same as
asm volatile("move $r23, $r24":::);
```

`OutputOperands` :

- A comma-separated list of the C variables modified by the instructions in the Assembler Template. An empty list is permitted.

Here is a example:

```
int result;
int d1, d2;
... ...
asm volatile("add.w %0 , %1 , %2 \n\t"
           : "=r" (result)
           : "r" (d1), "r" (d2):);
```

In the above example, the instructions and operands used by extended asm are first described. By using '`:`' to inform the compiler of the meaning of this assembly instruction and the number of operands required by the assembly instruction. '`=r`' and '`r`' indicate that this is a register operand, referred to as an operand constraint. '`r`' represents a fixed point register operand, '`=`' represents an output operand, otherwise it is an input operand. After containing the above information, the compiler can understand how to convert this extended asm into actual assembly instructions.

If the extended asm is bound to a register variable, it should be noted that if the extended asm uses a temporary register to save the result, or if a register variable is bound to a temporary register, it may be changed during the function call, as the value of the temporary register will not be maintained by the system during the function call and manual instructions need to be added for maintenance. So it is recommended to use save registers as much as possible in extended asm.

`InputOperands` :

- A comma-separated list of C expressions read by the instructions in the AssemblerTemplate. An empty list is permitted.
- Sometimes we need to use designated registers in assembly instructions. A typical example is a system call, where the system call number and parameters must be placed in the designated registers.
- To achieve this goal, we need to use extended syntax when declaring variables.

Here is a example:

```
register int a asm("$a0") = 1;
register int b asm("$a1") = 2;

asm volatile("addi.w %0,%1,0xf\n\t"
              :"=r"(a)
              :"r"(b));
```

If the register used by extended asm is specified in the instruction description, it should be noted that general-purpose registers can be represented as `$GR` or `GR`, the '`$`' symbol is not necessary, floating-point registers must contain the '`$`' symbol, and floating-point registers must be represented as `$FR`. Please do not use its alias for `FR` .

`Clobbers` :

- A comma-separated list of registers or other values changed by the AssemblerTemplate, beyond those listed as outputs. An empty list is permitted.
- Some assembly instructions may implicitly modify some registers that are not in the instruction operand. In order to make the compiler aware of this situation, the implicit change of register rules is listed after the input rules.

Here is a example:

```
asm volatile("xor $r25, $r25, %0\n\t"
            ::"r"(a):"r25");
```

`GotoLabels` :

With extended asm you can read and write C variables from assembler and perform jumps from assembler code to C labels. Extended asm syntax uses colons ('`:`') to delimit the operand parameters after the assembler template:

```
asm goto( AssemblerTemplate
        : OutputOperands
        : InputOperands
        : Clobbers
        : GotoLabels)
```

Here is a example:

```
    ra = 0;
    asm goto("beqz  %0, %l[labelbeqz] \n\t"
             :
             :"r"(ra):
             :labelbeqz);
    // code
labelbeqz:
    // code
```

"`%l[labelbeqz]`" indicates the target label to jump to in the C language source code, while embedding the assembly in the format of "`goto`":

This is useful for above cases:

- Move the content of C variable to an `LoongArch` register.
- Move the content of an `LoongArch` register to a C variable.
- Access assembly instructions that are not readily available to C programs.

For more extended asm related content, please refer to:

- Basic Asm — Assembler Instructions Without Operands
- Extended Asm - Assembler Instructions with C Expression Operands

CHAPTER 8. Instruction Set

The `LoongArch` architecture is divided into two versions: 32-bit and 64-bit, respectively referred to as `LA32` architecture and `LA64` architecture. The `LA64` architecture is application level down binary compatible with the `LA32` architecture. The so-called "application level down binary compatibility" refers to the fact that the binaries of application software using the `LA32` architecture can directly run on machines compatible with the `LA64` architecture and obtain the same running results. On the other hand, it refers to the fact that this down binary compatibility is limited to application software. The architecture specification does not guarantee that the binary of system software (such as the operating system kernel) running on machines compatible with `LA32` architecture always obtains the same running result when running directly on machines compatible with `LA64` architecture.

The hexadecimal representation of the integer range involved in this book must contain signed, for example, [-`0x800`, `0x7ff`] indicates that the range is [-`2048`, `2047`], where the minus sign of -`0x800` must exist. Otherwise, the assembler overflow error will occur. If it is necessary to not write signed to represent this range, the `0x800` and `0x7ff` should be signed extended to `GRLEN`, corresponding to [`0xfffff800`, `0x000007ff`] of `LA32` and [`0xfffffffffffff800`, `0x00000000000007ff`] of `LA64`.

8.1 Base Instruction Introduction

The basic part of the `LoongArch` architecture consists of two parts: the non privileged instruction set and the privileged instruction set. The non privileged instruction set defines commonly used integer and floating-point instructions, which can fully support the generation of efficient object code by existing mainstream compilation systems.

8.1.1 Base Integer Instruction

This section will describe the functionality of application level basic integer instructions in the `LA64` architecture. For the `LA32` architecture, only one subset needs to be implemented. Due to the fact that the bit width of `GR` under the `LA32` architecture is only 32 bits, the signed extension operation in the subsequent instruction description of "writing the 32-bit result signed extension to the general register `$rd`" is not required.

8.1.1.1 Arithmetic Operation Instructions

8.1.1.1.1 `ADD.{W/D}`, `SUB.{W/D}`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`add.w`, `sub.w`, `add.d`, `sub.d`	`$rd`	`$rj`	`$rk`

Description :

{`add`/`sub`}.`w` : `$rd` = **SignExtend**(`$rj`[31:0] +/- `$rk`[31:0], GRLEN)

{`add`/`sub`}.`d` : `$rd` = `$rj`[63:0] +/- `$rk`[63:0]

Usage :

```
{add/sub}.{w/d}    $r23,  $r24,  $r25
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.1.1` .

8.1.1.1.2 `ADDI.{W/D}`, `ADDU16I.D`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`addi.w`, `addi.d`	`$rd`	`$rj`	`si12`
`addu16i.d`	`$rd`	`$rj`	`si16`

Description :

`addi.w` : `$rd` = **SignExtend**(`$rj`[31:0] + **SignExtend**(`si12`, 32), GRLEN)

`addi.d` : `$rd` = `$rj`[63:0] + **SignExtend**(`si12`, 64)

- si12 : 12 bit immediate, Signed value range(integer) : [-2048, 2047] or [-0x800, 0x7ff]

addu16i.d : $rd = $rj[63:0] + **SignExtend**({si16, 16’b0}, 64)

- si16 : 16 bit immediate, Signed value range(integer) : [-32768, 32767] or [-0x8000, 0x7fff]
  - The input si16 is the value before the offset operation.

Usage :

```
li.w             $r24, -2048              # $t1 = -2048
addi.w/addi.d    $r23, $r24, -2048        # $t0 = -2040
addu16i.d        $r23, $r24, -32768       # $t0 = -2147483640
```

- **Explanation :**
  - The LA64 of -32768 is 0x8000, shifts the 16-bit immediate sil6 logic to the left by 16 bits, shifts the 16-bit immediate si16 logic to the left by 16 bits, the result is 0x80000000 or -2147483648 .

NOTE For more information, refer to the LoongArch instruction manual:2.2.1.2 .

8.1.1.1.3 ALSL.{W[U]/D}, ALSL.D

Syntax:

```
opcode    dest,   src1,   src2,   ShiftAmount
```

opcode	dest	src1	src2	ShiftAmount
alsl.w, alsl.wu, alsl.d	$rd	$rj	$rk	1,2,3,4

Description :

alsl.w : $rd = **SignExtend**((($rj[31:0]<<(ShiftAmount)) + $rk[31:0])[31:0], GRLEN)

alsl.wu : $rd = **ZeroExtend**((($rj[31:0]<<(ShiftAmount)) + $rk[31:0])[31:0], GRLEN)

alsl.d : $rd = (($rj[63:0]<<(ShiftAmount)) + $rk[63:0])[63:0]

Usage :

```
li.w     $r24, 8                    # $r24 = 8
li.w     $r25, 4                    # $r25 = 4
alsl.w   $r23, $r24, $r25, 2        # $r23 = 8<<2 + 4 = 36
```

NOTE For more information, refer to the LoongArch instruction manual:2.2.1.3 .

8.1.1.1.4 `LU12I.W`, `LU32I.D`, `LU52I.D`

Syntax:

```
opcode    dest,  src1,  {src2}
```

opcode	dest	src1	src2
`lu12i.w`, `lu32i.d`	`$rd`	`si20`	
`lu52i.d`	`$rd`	`$rj`	`si12`

Description :

`lu12i.w` : `$rd` = **SignExtend**({`si20`, 12’b0}, GRLEN)

`lu32i.d` : `$rd` = {**SignExtend**(`si20`, 32), `$rd`[31:0]}

- `si20` : 20 bit immediate, Signed value range(`integer`) : [`-524288`, `524287`] or [`-0x80000`, `0x7ffff`]

`lu52i.d` : `$rd` = {`si12`, `$rj`[51:0]}

- `si12` : 12 bit immediate, Signed value range(`integer`) : [`-2048`, `2047`] or [`-0x800`, `0x7ff`]

Usage :

```
lu12i.w $r23, 0x54321          # $r12 = 0x54321000
lu32i.d $r23, 0xa9876          # $r12 = 0xa987654321000
lu52i.d $r23, $r23, 0xdcb      # $r12 = 0xdcba987654321000
```

- **Explanation :**
  - The loading of immediate number in LoongArch is very cumbersome, and pseudo instructions are generally used when writing assembly files:
    - `li.w` `$rd`, `imm32`
    - `li.d` `$rd`, `imm64`

NOTE For more information, refer to the `LoongArch instruction manual:2.2.1.4` .

Usage :

```
li.w   $r23, 100000              # $r12 = 100000
li.d   $r23, 1000000             # $r12 = 1000000
```

8.1.1.1.5 SLT[U]

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
slt, sltu	$rd	$rj	$rk

Description :

slt : $rd = (signed($rj) < signed($rk)) ? 1 : 0

sltu : $rd = (unsigned($rj) < unsigned($rk)) ? 1 : 0

Usage :

```
slt/sltu  $r23, $r24, $r25
```

NOTE For more information, refer to the LoongArch instruction manual:2.2.1.5 .

8.1.1.1.6 SLT[U]I

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
slti, sltui	$rd	$rj	si12

Description :

slti : $rd = (signed($rj) < signed(**SignExtend**(si12, GRLEN))) ? 1 : 0

sltui : $rd = (unsigned($rj) < unsigned(**SignExtend**(si12, GRLEN))) ? 1 : 0

- si12 : 12 bit immediate, Signed value range(integer) : [-2048, 2047] or [-0x800, 0x7ff]

Usage :

```
slti/sltui  $r23, $r24, 1
```

NOTE For more information, refer to the LoongArch instruction manual:2.2.1.6 .

8.1.1.1.7 `PCADDI`, `PCADDU12I`, `PCADDU18I`, `PCALAU12I`

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
`pcaddi`, `pcaddu12i`, `pcaddu18i`, `pcalau12i`	`$rd`	`si20`

Description :

`pcaddi` : `$rd` = **PC** + **SignExtend**({`si20`, 2’b0}, GRLEN)

`pcaddu12i` : `$rd` = **PC** + **SignExtend**({`si20`, 12’b0}, GRLEN)

`pcaddu18i` : `$rd` = **PC** + **SignExtend**({`si20`, 18’b0}, GRLEN)

`pcalau12i` : `$rd` = {(**PC** + **SignExtend**({`si20`, 12’b0}, GRLEN))[GRLEN-1:12], 12’b0}

- `si20` : 20 bit immediate, Signed value range(`integer`) : [`-524288`, `524287`] or [`-0x80000`, `0x7ffff`]
  - The input `si20` is the value `before` the `offset operation`.

Usage :

```
pcaddi    $r24, 0xf    # PC = 120000ba0; $r24 = 120000bdc
pcaddu12i $r24, 0xf    # PC = 120000bb8; $r24 = 12000fbb8
pcaddu18i $r24, 0xf    # PC = 120000bd0; $r24 = 1203c0bd0
pcalau12i $r24, 0xf    # PC = 120000be8; $r24 = 12000f000
```

- **Explanation :**
  - The `PC` value saved in `$r14` has actually increased by `0x3c`
  - The `PC` value saved in `$r14` has actually increased by `0xf000`
  - The `PC` value saved in `$r14` has actually increased by `0x3c0000`
  - The `PC` value saved in `$r14` has actually increased by `0xe418`, And store it in `$r14` after the low bit of 12 is 0.

NOTE For more information, refer to the `LoongArch instruction manual:2.2.1.7` .

8.1.1.1.8 `AND`, `OR`, `NOR`, `XOR`, `ANDN`, `ORN`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
and, or, nor, xor, andn, orn	$rd	$rj	$rk

Description :

and : $rd = $rj & $rk

or : $rd = $rj | $rk

nor : $rd = ~($rj | $rk)

xor : $rd = $rj ^ $rk

andn : $rd = $rj & ~($rk)

orn : $rd = $rj | ~($rk)

Usage :

```
li.d   $r24, 0x00000000ffad1235    # $r24 = 0x00000000ffad1235
li.d   $r25, 0x00000000ccdd2345    # $r25 = 0x00000000ccdd2345
and    $r23, $r24, $r25            # $r23 = 0x00000000cc8d0205
or     $r23, $r24, $r25            # $r23 = 0x00000000fffd3375
nor    $r23, $r24, $r25            # $r23 = 0xffffffff0002cc8a
xor    $r23, $r24, $r25            # $r23 = 0x0000000033703170
andn   $r23, $r24, $r25            # $r23 = 0x0000000033201030
orn    $r23, $r24, $r25            # $r23 = 0xffffffffffafdebf
```

NOTE For more information, refer to the LoongArch instruction manual:2.2.1.8 .

8.1.1.1.9 ANDI, ORI, XORI

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
andi, ori, xori	$rd	$rj	ui12

Description :

andi : $rd = $rj & **ZeroExtend**(ui12, GRLEN)

ori : $rd = $rj | **ZeroExtend**(ui12, GRLEN)

xori : $rd = $rj ^ **ZeroExtend**(ui12, GRLEN)

- ui12 : 12 bit immediate, Unsigned value range(integer) : [0, 4095] or [0x000, 0xfff]

Usage :

```
li.d   $r24, 0xfffffffffad1f0f    # $r24 = 0xfffffffffad1f0f
andi   $r23, $r24, 0xff0          # $r23 = 0x0000000000000f00
ori    $r23, $r24, 0xff0          # $r23 = 0xfffffffffad1fff
xori   $r23, $r24, 0xff0          # $r23 = 0xfffffffffad10ff
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.1.9` .

8.1.1.1.10 `NOP`

Syntax:

```
nop
```

Description :

`nop` : `andi $zero`, `$zero`, 0

NOTE For more information, refer to the `LoongArch instruction manual:2.2.1.10` .

8.1.1.1.11 `MUL.{W/D}`, `MULH.{W[U]/D[U]}`,

Syntax:

```
opcode    dest,   src1,   src2
```

opcode	dest	src1	src2
`mul.w`, `mulh.w`, `mulh.wu`, `mul.d`, `mulh.d`, `mulh.du`	`$rd`	`$rj`	`$rk`

Description :

`mul.w` : `$rd` = **SignExtend**((signed(`$rj`[31:0]) × signed(`$rk`[31:0]))[31:0], GRLEN)

`mulh.w` : `$rd` = **SignExtend**((signed(`$rj`[31:0]) × signed(`$rk`[31:0]))[63:31], GRLEN)

`mulh.wu` : `$rd` = **SignExtend**((unsigned(`$rj`[31:0]) × unsigned(`$rk`[31:0]))[63:32], GRLEN)

`mul.d` : `$rd` = (signed(`$rj`[63:0]) × signed(`$rk`[63:0]))[63:0]

`mulh.d` : `$rd` = (signed(`$rj`[63:0]) × signed(`$rk`[63:0]))[127:64]

`mulh.du` : `$rd` = (unsigned(`$rj`[63:0]) × unsigned(`$rk`[63:0]))[127:64]

Usage :

```
li.d      $r26, 0x000000000000000f   # $r26 = 0x000000000000000f
li.d      $r27, 0xffffffff80000000   # $r27 = 0xffffffff80000000
mul.w     $r23, $r26, $r27           # $r23 = 0xffffffff80000000
mulh.w    $r24, $r26, $r27           # $r24 = 0xfffffffffffffff8
mulh.wu   $r25, $r26, $r27           # $r25 = 0x0000000000000007
li.d      $r26, 0x000000000000000f   # $r26 = 0x000000000000000f
li.d      $r27, 0x8000000000000000   # $r27 = 0x8000000000000000
mul.d     $r23, $r26, $r27           # $r23 = 0x8000000000000000
mulh.d    $r24, $r26, $r27           # $r24 = 0xfffffffffffffff8
mulh.du   $r25, $r26, $r27           # $r25 = 0x0000000000000007
```

- **Explanation :**
    - The signed operation result of `$r26` multiplied by `$r27` is `0xfffff88000000`, and the unsigned operation result is `0x00000078000000`. Because the operation results of `$23`, `$24`, and `$25` are all stored in registers after signed extend, only `31 bit` to `0 bit` are taken when viewing the operation results.

NOTE For more information, refer to the `LoongArch instruction manual:2.2.1.11` .

8.1.1.1.12 `MULW.D.W[U]`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`mulw.d.w`, `mulw.d.wu`	`$rd`	`$rj`	`$rk`

Description :

`mulw.d.w` : `$rd` = (signed(`$rj`[31:0]) × signed(`$rk`[31:0]))[63:0]

`mulw.d.wu` : `$rd` = (unsigned(`$rj`[31:0]) × unsigned(`$rk`[31:0]))[63:0]

Usage :

```
li.d        $r26, 0x000000000000000f    # $r26 = 0x000000000000000f
li.d        $r27, 0xffffffff80000000    # $r27 = 0xffffffff80000000
mulw.d.w    $r23, $r26, $r27            # $r23 = 0xfffffff880000000
mulw.d.wu   $r24, $r26, $r27            # $r24 = 0x0000000780000000
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.1.12` .

8.1.1.1.13 DIV.{W[U]/D[U]}, MOD.{W[U]/D[U]}

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
div.w, div.wu, mod.w, mod.wu, div.d, div.du, mod.d, mod.du	$rd	$rj	$rk

Description :

div.w : $rd = **SignExtend**((signed($rj[31:0]) / signed($rk[31:0]))[31:0], GRLEN)

div.wu : $rd = **SignExtend**((unsigned($rj[31:0]) / unsigned($rk[31:0]))[31:0], GRLEN)

mod.w : $rd = **SignExtend**((signed($rj[31:0]) % signed($rk[31:0]))[31:0], GRLEN)

mod.wu : $rd = **SignExtend**((unsigned($rj[31:0]) % unsigned($rk[31:0]))[31:0], GRLEN)

div.d : $rd = signed($rj[63:0]) / signed($rk[63:0])

div.du : $rd = unsigned($rj[63:0]) / unsigned($rk[63:0])

mod.d : $rd = signed($rj[63:0]) % signed($rk[63:0])

mod.du : $rd = unsigned($rj[63:0]) % unsigned($rk[63:0])

Usage :

```
li.d      $r26, 0x000000000000000f    # $r26 = 0x000000000000000f
li.d      $r27, 0xffffffff80000000    # $r27 = 0xffffffff80000000
div.w     $r23, $r26, $r27            # $r23 = 0xfffffffff7777778
div.wu    $r23, $r26, $r27            # $r23 = 0x0000000008888888
mod.w     $r23, $r26, $r27            # $r23 = 0xfffffffffffffff8
mod.wu    $r23, $r26, $r27            # $r23 = 0x0000000000000008
li.d      $r26, 0x000000000000000f    # $r26 = 0x000000000000000f
li.d      $r27, 0x8000000000000000    # $r27 = 0x8000000000000000
div.d     $r23, $r26, $r27            # $r23 = 0xf777777777777778
div.du    $r23, $r26, $r27            # $r23 = 0x0888888888888888
mod.d     $r23, $r26, $r27            # $r23 = 0xfffffffffffffff8
mod.du    $r23, $r26, $r27            # $r23 = 0x0000000000000008
```

NOTE For more information, refer to the LoongArch instruction manual:2.2.1.13 .

8.1.1.2 Bit-shift Instructions

- **SLL** : Shift data logic left.
- **SRL** : Shift data logic right.
- **SRA** : Arithmetic shift of data to the right.
- **ROTR** : Rotate data to the right.

8.1.1.2.1 `SLL.W`, `SRL.W`, `SRA.W`, `ROTR.W`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`sll.w`, `srl.w`, `sra.w`, `rotr.w`	`$rd`	`$rj`	`$rk`

Description :

`sll.w` : `$rd` = **SignExtend**(`SLL`(`$rj`[31:0], `$rk`[4:0]), GRLEN)

`srl.w` : `$rd` = **SignExtend**(`SRL`(`$rj`[31:0], `$rk`[4:0]), GRLEN)

`sra.w` : `$rd` = **SignExtend**(`SRA`(`$rj`[31:0], `$rk`[4:0]), GRLEN)

`rotr.w` : `$rd` = **SignExtend**(`ROTR`(`$rj`[31:0], `$rk`[4:0]), GRLEN)

- The shift amount of the above-mentioned shift instruction is all [4:0] (Unsigned value range(`integer`) : [`0`, `31`]) bit data in the general register `rk`, and is regarded as an unsigned number.

Usage :

```
li.d     $r23, 0x00000000f000000e    # $r23 = 0x00000000f000000e
li.d     $r24, 0x0000000000000002    # $r24 = 0x0000000000000002
sll.w    $r25, $r23, $r24            # $r25 = 0xffffffffc0000038
srl.w    $r25, $r23, $r24            # $r25 = 0x000000003c000003
sra.w    $r25, $r23, $r24            # $r25 = 0xfffffffffc000003
rotr.w   $r25, $r23, $r24            # $r25 = 0xffffffffbc000003
```

- **Explanation :**
  - Please note that the above instructions first perform Bit-shift operations and then perform signed extend.

NOTE For more information, refer to the `LoongArch instruction manual:2.2.2.1` .

8.1.1.2.2 `SLLI.W`, `SRLI.W`, `SRAI.W`, `ROTRI.W`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`slli.w`, `srli.w`, `srai.w`, `rotri.w`	`$rd`	`$rj`	`ui5`

Description :

`slli.w` : `$rd` = **SignExtend**(`SLL`(`$rj`[31:0], `ui5`), GRLEN)

`srli.w` : `$rd` = **SignExtend**(`SRL`(`$rj`[31:0], `ui5`), GRLEN)

`srai.w` : `$rd` = **SignExtend**(`SRA`(`$rj`[31:0], `ui5`), GRLEN)

`rotri.w` : `$rd` = **SignExtend**(`ROTR`(`$rj`[31:0], `ui5`), GRLEN)

- `ui5` : 5 bit immediate, Unsigned value range(`integer`) : [`0`, `31`] or [`0x0`, `0x1f`]
    - The shift amounts of the above shift instructions are all 5-bit unsigned immediate `ui5` in the instruction code.

Usage :

```
li.d      $r23, 0x00000000f000000e    # $r23 = 0x00000000f000000e
slli.w    $r25, $r23, 2               # $r25 = 0xffffffffc0000038
srli.w    $r25, $r23, 2               # $r25 = 0x000000003c000003
srai.w    $r25, $r23, 2               # $r25 = 0xfffffffffc000003
rotri.w   $r25, $r23, 2               # $r25 = 0xffffffffbc000003
```

- **Explanation :**
    - Please note that the above instructions first perform Bit-shift operations and then perform signed extend.

NOTE For more information, refer to the `LoongArch instruction manual:2.2.2.2` .

8.1.1.2.3 `SLL.D`, `SRL.D`, `SRA.D`, `ROTR.D`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`sll.d`, `srl.d`, `sra.d`, `rotr.d`	`$rd`	`$rj`	`$rk`

Description :

`sll.d` : `$rd` = `SLL`(`$rj`[63:0], `$rk`[5:0])

`srl.d` : `$rd` = `SRL`(`$rj`[63:0], `$rk`[5:0])

`sra.d` : `$rd` = `SRA`(`$rj`[63:0], `$rk`[5:0])

`rotr.d` : `$rd` = `ROTR`(`$rj`[63:0], `$rk`[5:0])

- The shift amount of the above-mentioned shift instruction is all [5:0] (Unsigned value range(`integer`) : [`0`, `63`]) bit data in the general register `rk`, and is regarded as an unsigned number.

Usage :

```
li.d     $r23, 0xf00000000000000e    # $r23 = 0xf00000000000000e
li.d     $r24, 0x0000000000000002    # $r24 = 0x0000000000000002
sll.d    $r25, $r23, $r24            # $r25 = 0xc000000000000038
srl.d    $r25, $r23, $r24            # $r25 = 0x3c00000000000003
sra.d    $r25, $r23, $r24            # $r25 = 0xfc00000000000003
rotr.d   $r25, $r23, $r24            # $r25 = 0xbc00000000000003
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.2.3` .

8.1.1.2.4 `SLLI.D`, `SRLI.D`, `SRAI.D`, `ROTRI.D`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`slli.d`, `srli.d`, `srai.d`, `rotri.d`	`$rd`	`$rj`	`ui6`

Description :

`slli.d` : `$rd` = `SLL`(`$rj`[63:0], `ui6`)

`srli.d` : `$rd` = `SRL`(`$rj`[63:0], `ui6`)

`srai.d` : `$rd` = `SRA`(`$rj`[63:0], `ui6`)

`rotri.d` : `$rd` = `ROTR`(`$rj`[63:0], `ui6`)

- `ui6` : 6 bit immediate, Unsigned value range(`integer`) : [`0`, `63`] or [`0x0`, `0x3f`]
    - The shift amount of the above-mentioned shift instruction is the 6-bit unsigned immediate `ui6` in the instruction code.

Usage :

```
li.d     $r23, 0xf00000000000000e    # $r23 = 0xf00000000000000e
slli.d   $r25, $r23, 2               # $r25 = 0xc000000000000038
srli.d   $r25, $r23, 2               # $r25 = 0x3c00000000000003
srai.d   $r25, $r23, 2               # $r25 = 0xfc00000000000003
rotri.d  $r25, $r23, 2               # $r25 = 0xbc00000000000003
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.2.4` .

8.1.1.3 Bit-manipulation Instructions

8.1.1.3.1 `EXT.W.{B/H}`

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
`ext.w.b`, `ext.w.h`	`$rd`	`$rj`

Description :

`ext.w.b` : `$rd` = **SignExtend**(`$rj`[7:0], GRLEN)

`ext.w.h` : `$rd` = **SignExtend**(`$rj`[15:0], GRLEN)

Usage :

```
li.d       $r23, 0x1111111111118111    # $r23 = 0x1111111111118111
ext.w.b    $r25, $r23                  # $r25 = 0xc000000000000011
ext.w.h    $r25, $r23                  # $r25 = 0xffffffffffff8111
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.3.1` .

8.1.1.3.2 `CL{O/Z}.{W/D}`, `CT{O/Z}.{W/D}`

This section involves four functions, namely CLO, CLZ, CTO, and CTZ.

- The **CLO** function performs the operation that for the data of bit [63/31:0] in the general register rj, the number of continuous bits 1 is measured from bit 63/31 to bit 0, and the result is written into the general register rd.
- The **CLZ** function performs the operation that for the data of bit [63/31:0] in the general register rj, the number of continuous bits 0 is measured from bit 63/31 to bit 0, and the result is written into the general register rd.
- The **CTO** function performs the operation that for the data of bit [63/31:0] in the general register rj, the number of continuous bits 1 is measured from bit 0 to bit 63/31, and the result is written into the general register rd.
- The **CTZ** function performs the operation that for the data of bit [63/31:0] in the general register rj, the number of continuous bits 0 is measured from bit 0 to bit 63/31, and the result is written into the general register rd.

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
`clo.w`, `clo.d`, `clz.w`, `clz.d`, `cto.w`, `cto.d`, `ctz.w`, `ctz.d`	`$rd`	`$rj`

Description :

`clo.w` : `$rd` = **CLO**(`$rj`[31:0])

`clo.d` : `$rd` = **CLO**(`$rj`[63:0])

`clz.w` : `$rd` = **CLZ**(`$rj`[31:0])

`clz.d` : `$rd` = **CLZ**(`$rj`[63:0])

`cto.w` : `$rd` = **CTO**(`$rj`[31:0])

`cto.d` : `$rd` = **CTO**(`$rj`[63:0])

`ctz.w` : `$rd` = **CTZ**(`$rj`[31:0])

`ctz.d` : `$rd` = **CTZ**(`$rj`[63:0])

Usage :

```
li.d     $r23, 0xfff00000f00fffff    # $r23 = 0xfff00000f00fffff
clo.w    $r25, $r23                  # $r25 = 4
clo.d    $r25, $r23                  # $r25 = 12
```

- **Explanation:**
  - Calculate the number of consecutive bit 1 from bit 31 to bit 0. The operands involved in the calculation are `0xf00fffff`, and stop when the first bit 0 occurs. So the result obtained by running the instruction is 4.
  - Calculate the number of consecutive bit 1 from bit 63 to bit 0. The operands involved in the calculation are `0xfff00000f00fffff`, and stop when the first bit 0 occurs. So the result obtained by running the instruction is 12.

```
li.d     $r23, 0x000fffff0000ffff    # $r23 = 0x000fffff0000ffff
clz.w    $r25, $r23                  # $r25 = 16
clz.d    $r25, $r23                  # $r25 = 12
```

- **Explanation:**
  - Calculate the number of consecutive bit 0 from bit 31 to bit 0. The operands involved in the calculation are `0x0000ffff`, and stop when the first bit 0 occurs. So the result obtained by running the instruction is 16.
  - Calculate the number of consecutive bit 0 from bit 63 to bit 0. The operands involved in the calculation are `0x000fffff0000ffff`, and stop when the first bit 0 occurs. So the result obtained by running the instruction is 12.

```
li.d     $r23, 0x000fffffffffffff    # $r23 = 0x000fffffffffffff
cto.w    $r25, $r23                  # $r25 = 32
cto.d    $r25, $r23                  # $r25 = 52
```

- **Explanation:**
  - Calculate the number of consecutive bit 1 from bit 0 to bit 31. The operands involved in the calculation are `0xffffffff`, and stop when the first bit 0 occurs. So the result obtained by running the instruction is 32.
  - Calculate the number of consecutive bit 1 from bit 0 to bit 63. The operands involved in the calculation are `0x000fffffffffffff`, and stop when the first bit 0 occurs. So the result obtained by running the instruction is 52.

```
li.d     $r23, 0xfff0000000000000    # $r23 = 0xfff0000000000000
ctz.w    $r25, $r23                  # $r25 = 32
ctz.d    $r25, $r23                  # $r25 = 52
```

- **Explanation:**
  - Calculate the number of consecutive bit 0 from bit 0 to bit 31. The operands involved in the calculation are `0x00000000`, and stop when the first bit 0 occurs. So the result obtained by running the instruction is 32.
  - Calculate the number of consecutive bit 0 from bit 0 to bit 63. The operands involved in the calculation are `0xfff0000000000000`, and stop when the first bit 0 occurs. So the result obtained by running the instruction is 52.

NOTE For more information, refer to the `LoongArch instruction manual:2.2.3.2` .

8.1.1.3.3 `BYTEPICK.{W/D}`

Syntax:

```
opcode    dest,  src1,  src2,  ShiftAmount
```

opcode	dest	src1	src2	ShiftAmount
`bytepick.w`	`$rd`	`$rj`	`$rk`	{`0`,`1`,`2`,`3`}
`bytepick.d`	`$rd`	`$rj`	`$rk`	{`0`,`1`,`2`,`3`,`4`,`5`,`6`,`7`}

Description :

`bytepick.w` : `$rd` = **SignExtend**({ `$rk`[8×(4-`ShiftAmount`)-1:0], `$rj`[31:8×(4-`ShiftAmount`)] }[31:0], GRLEN)

`bytepick.d` : `$rd` = { `$rk`[8×(8-`ShiftAmount`)-1:0], `$rj`[63:8×(8-`ShiftAmount`)] }

Usage :

```
li.d        $r23, 0x0000000001230000    # $r23 = 0x0000000001230000
li.d        $r24, 0x0000000000004567    # $r24 = 0x0000000000004567
bytepick.w  $r25,$r23,$r24,sa2          # $r25 = 0x0000000045670123
li.d        $r23, 0x0123456700000000    # $r23 = 0x0123456700000000
li.d        $r24, 0x0000000089abcdef    # $r24 = 0x0000000089abcdef
bytepick.d  $r25,$r23,$r24,sa3          # $r25 = 0x89abcdef01234567
```

- **Explanation:**
  - When `ShiftAmount`=2:
    - `bytepick.w` : `$r25` = **SignExtend**({`$r24`[15:0], `$r23`[31:16]}[31:0], GRLEN)
    - `$r25` = `0x0000000045670123`
  - When `ShiftAmount`=4:
    - `bytepick.w` : `$r25` = {`$r24`[31:0], `$r23`[63:32]}
    - `$r25` = `0x89abcdef01234567`

NOTE For more information, refer to the `LoongArch instruction manual:2.2.3.3` .

8.1.1.3.4 `REVB.{2H/4H/2W/D}`

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
`revb.2h`, `revb.4h`, `revb.2w`, `revb.d`	`$rd`	`$rj`

Description :

`revb.2h` :

- `temp0` = {`$rj`[7 : 0], `$rj`[15: 8]}
- `temp1` = {`$rj`[23:16], `$rj`[31:24]}
- `$rd` = **SignExtend**({`temp1`, `temp0`}, GRLEN)

`revb.4h` :

- `temp0` = {`$rj`[7 : 0], `$rj`[15: 8]}
- `temp1` = {`$rj`[23:16], `$rj`[31:24]}
- `temp2` = {`$rj`[39:32], `$rj`[47:40]}
- `temp3` = {`$rj`[55:48], `$rj`[63:56]}

- `$rd` = {`temp3`, `temp2`, `temp1`, `temp0`}

`revb.2w` :

- `temp0` = {`$rj`[7 : 0], `$rj`[15: 8], `$rj`[23:16], `$rj`[31:24]}
- `temp1` = {`$rj`[39:32], `$rj`[47:40], `$rj`[55:48], `$rj`[63:56]}
- `$rd` = {`temp1`, `temp0`}

`revb.d` :

- `$rd` = {`$rj`[7:0], `$rj`[15:8], `$rj`[23:16], `$rj`[31:24], `$rj`[39:32], `$rj`[47:40], `$rj`[55:48], `$rj`[63:56]}

Usage :

```
li.d       $r23, 0xfedcba9876543210    # $r23 = 0xfedcba9876543210
revb.2h    $r25, $r23                  # $r25 = 0x0000000054761032
revb.4h    $r25, $r23                  # $r25 = 0xdcfe98ba54761032
revb.2w    $r25, $r23                  # $r25 = 0x98badcfe10325476
revb.d     $r25, $r23                  # $r25 = 0x1032547698badcfe
```

- **Explanation:**
  - Function description of the `revb` series instructions: Reverse the byte data within a specified range, with different suffixes determining different ranges.
  - `revb.2h` represents dividing the data into two halfwords, and reversing the bytes in each of the two halfwords.
    - When using the `revb.h` instruction to process `0xfedcba9876543210`, only data between bit 31 and bit 0 will be processed. `0x76543210` will be divided into two halfwords, namely `0x7654` and `0x3210`, and the bytes in the two will be arranged in reverse to obtain `0x5476` and `0x1032`. The final result is `0x0000000054761032`.
    - `0xfedcba98 7654 3210` → `revb`(`7654`, `3210`) → `0x0000000054761032`
  - `revb.4h` means dividing the data into four halfwords and arranging the bytes in reverse order in each of the two halfwords.
    - `0xfedc ba98 7654 3210` → `revb`(`fedc`, `ba98`, `7654`, `3210`) → `0xdcfe98ba54761032`
  - `revb.2w` means dividing the data into two words and arranging the bytes in reverse in each word.
    - `0xfedcba98 76543210` → `revb`(`fedcba98`, `76543210`) → `0x98badcfe10325476`
  - `revb.d` represents the reverse arrangement of bytes in the entire doubleword data.
    - `0xfedcba9876543210` → `revb`(`fedcba9876543210`) → `0x1032547698badcfe`

NOTE For more information, refer to the `LoongArch instruction manual:2.2.3.4` .

8.1.1.3.5 REVH.{2W/D}

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
revh.2w, revh.d	$rd	$rj

Description :

revh.2w :

- temp0 = {$rj[15: 0], $rj[31:16]}
- temp1 = {$rj[47:32], $rj[63:48]}
- $rd = {temp1, temp0}

revh.d :

- $rd = {$rj[15: 0], $rj[31:16], $rj[47:32], $rj[63:48]}

Usage :

```
li.d      $r23, 0xfedcba9876543210    # $r23 = 0xfedcba9876543210
revh.2w   $r25, $r23                  # $r25 = 0xba98fedc32107654
revh.d    $r25, $r23                  # $r25 = 0x32107654ba98fedc
```

NOTE For more information, refer to the LoongArch instruction manual:2.2.3.5 .

8.1.1.3.6 BITREV.{4B/8B}

The bitrev($rj[a : b]) performs the operation that the [a : b] bit in general register rj is arranged in reverse order.

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
bitrev.4b, bitrev.8b	$rd	$rj

Description :

bitrev.4b :

- temp3 = **bitrev**($rj[31:24])

- `temp2` = **bitrev**(`$rj`[23:16])
- `temp1` = **bitrev**(`$rj`[16 : 8])
- `temp0` = **bitrev**(`$rj`[7 : 0])
- `$rd` = **SignExtend**({`temp3`, `temp2`, `temp1`, `temp0`}, GRLEN)

`bitrev.8b` :

- `temp7` = **bitrev**(`$rj`[63:56])
- `temp6` = **bitrev**(`$rj`[55:48])
- `temp5` = **bitrev**(`$rj`[47:40])
- `temp4` = **bitrev**(`$rj`[39:32])
- `temp3` = **bitrev**(`$rj`[31:24])
- `temp2` = **bitrev**(`$rj`[23:16])
- `temp1` = **bitrev**(`$rj`[16 : 8])
- `temp0` = **bitrev**(`$rj`[7 : 0])
- `$rd` = {`temp7`, `temp6`, `temp5`, `temp4`, `temp3`, `temp2`, `temp1`, `temp0`}

Usage :

```
li.d         $r23, 0xfedcba9876543210    # $r23 = 0xfedcba9876543210
bitrev.4b    $r25, $r23                  # $r25 = 0x000000006e2a4c08
bitrev.8b    $r25, $r23                  # $r25 = 0x7f3b5d196e2a4c08
```

- **Explanation:**
  - `bitrev.8b`
    - Divide bit 31 to bit 0 into 4 bytes to perform a bitwise reverse order operation.
    - `0x10` → `0b00010000` → **bitrev**(`0b00010000`) → `0b00001000` → `0x08`
    - `0x32` → `0b00110010` → **bitrev**(`0b00110010`) → `0b01001100` → `0x4c`
    - `0x54` → `0b01010100` → **bitrev**(`0b01010100`) → `0b00101010` → `0x2a`
    - `0x76` → `0b01110110` → **bitrev**(`0b01110110`) → `0b01101110` → `0x6e`
    - `0x98` → `0b10011000` → **bitrev**(`0b10011000`) → `0b00011001` → `0x19`
    - `0xba` → `0b10111010` → **bitrev**(`0b10111010`) → `0b01011101` → `0x5d`
    - `0xdc` → `0b11011100` → **bitrev**(`0b11011100`) → `0b00111011` → `0x3b`
    - `0xfe` → `0b11111110` → **bitrev**(`0b11111110`) → `0b01111111` → `0x7f`
    - `0xfedcba9876543210` → `0x7f3b5d196e2a4c08`

NOTE For more information, refer to the `LoongArch instruction manual:2.2.3.6` .

8.1.1.3.7 BITREV.{W/D}

The bitrev($rj[a : b]) performs the operation that the [a : b] bit in general register rj is arranged in reverse order.

Syntax:

```
opcode    dest,   src1
```

opcode	dest	src1
bitrev.w, bitrev.d	$rd	$rj

Description :

bitrev.w : $rd = **SignExtend**(**bitrev**($rj[31:0]), GRLEN)

bitrev.d : $rd = **bitrev**($rj[63:0])

Usage :

```
li.d        $r23, 0xfedcba9876543210    # $r23 = 0xfedcba9876543210
bitrev.w    $r25, $r23                  # $r25 = 0x00000000084c2a6e
bitrev.d    $r25, $r23                  # $r25 = 0x084c2a6e195d3b7f
```

NOTE For more information, refer to the LoongArch instruction manual:2.2.3.7 .

8.1.1.3.8 BSTRINS.{W/D}

Syntax:

```
opcode    dest,   src1,   src2,   src3
```

opcode	dest	src1	src2	src3
bstrins.w	$rd	$rj	msbw	lsbw
bstrins.d	$rd	$rj	msbd	lsbd

Description :

bstrins.w : $rd = **SignExtend**({$rd[31: msbw+1], $rj[msbw-lsbw:0], $rd[lsbw-1: 0]}, GRLEN)

- msbw, lsbw : Unsigned value range(integer) : **31** > msbw > lsbw > **0**

bstrins.d : $rd = {$rd[63: msbd+1], $rj[msbd-lsbd:0], $rd[lsbd-1: 0]}

- msbd, lsbd : Unsigned value range(integer) : **63** > msbd > lsbd > **0**

Usage :

```
li.d       $r23, 0x0123456789abcdef    # $r23 = 0x0123456789abcdef
li.d       $r25, 0xfedcba9876543210    # $r25 = 0xfedcba9876543210
bstrins.w  $r25, $r23, 15, 8
bstrins.d  $r25, $r23, 51, 8
```

- **Explanation:**
  - `bstrins.w`
    - `$r25`[31:16] = `0x7654`, `$r23`[7 : 0] = `0xef`, `$r25`[7 : 0] = `0x10`
    - `$r25`[31: 0] = {`7654`, `ef`, `10`} = `0x000000007654ef10`
  - `bstrins.d`
    - `$r25`[63:52] = `0xfed`, `$r23`[43: 0] = `0x56789abcdef`, `$r25`[7 : 0] = `0x10`
    - `$r25`[31: 0] = {`fed`, `56789abcdef`, `10`} = `0xfed56789abcdef10`

NOTE For more information, refer to the `LoongArch instruction manual:2.2.3.8` .

8.1.1.3.9 `BSTRPICK.{W/D}`

Syntax:

```
opcode    dest,   src1,   src2,   src3
```

opcode	**dest**	**src1**	**src2**	**src3**
`bstrpick.w`	`$rd`	`$rj`	`msbw`	`lsbw`
`bstrpick.d`	`$rd`	`$rj`	`msbd`	`lsbd`

Description :

`bstrpick.w` : `$rd` = **SignExtend** (**ZeroExtend**(`$rj`[`msbw` : `lsbw`], 32), GRLEN)

- `msbw`, `lsbw` : Unsigned value range(`integer`) : **31** > `msbw` > `lsbw` > **0**

`bstrpick.d` : `$rd` = **ZeroExtend**(`$rj`[`msbd` : `lsbd`], 64)

- `msbd`, `lsbd` : Unsigned value range(`integer`) : **63** > `msbd` > `lsbd` > **0**

Usage :

```
li.d        $r23, 0x0123456789abcdef    # $r23 = 0x0123456789abcdef
bstrpick.w  $r25, $r23, 15, 8
bstrpick.d  $r25, $r23, 51, 8
```

- **Explanation:**
  - `bstrpick.w`
    - `$r23`[15: 8] = `0xcd`, `$r25` = `0x00000000000000cd`
  - `bstrpick.d`
    - `$r23`[51: 8] = `0x3456789abcd`, `$r25` = `0x000003456789abcd`

NOTE For more information, refer to the `LoongArch instruction manual:2.2.3.9` .

8.1.1.3.10 `MASKEQZ`, `MASKNEZ`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`maskeqz`, `masknez`	`$rd`	`$rj`	`$rk`

Description :

`maskeqz` : `$rd` = (`$rk` == 0) ? 0 : `$rj`

`masknez` : `$rd` = (`$rk` ! = 0) ? 0 : `$rj`

Usage :

```
maskeqz  $r25, $r23, $r24
masknez  $r25, $r23, $r24
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.3.10` .

8.1.1.4 Branch Instructions

8.1.1.4.1 `BEQ`, `BNE`, `BLT[U]`, `BGE[U]`

Syntax:

```
opcode   src1,  src2,  src3
```

opcode	src1	src2	src3
`beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`	`$rd`	`$rj`	`si18` | `symbol`

Description :

`beq` : **if** (`$rj` == `$rd`) **jump** (`si18` | `symbol`)

`bne` : **if** (`$rj` **!** = `$rd`) **jump** (`si18` | `symbol`)

`blt` : **if** (signed(`$rj`) < signed(`$rd`)) **jump** (`si18` | `symbol`)

`bge` : **if** (signed(`$rj`) >= signed(`$rd`)) **jump** (`si18` | `symbol`)

`bltu` : **if** (unsigned(`$rj`) < unsigned(`$rd`)) **jump** (`si18` | `symbol`)

`bgeu` : **if** (unsigned(`$rj`) >= unsigned(`$rd`)) **jump** (`si18` | `symbol`)

- `si18` : a 4-bytes aligned 18-bits signed immediate value in range :
  - [`-131072`, `131068`] or [`-0x20000`, `0x1fffc`]
- `symbol` : Tags in assembly for jump.

Usage :

```
beq    $r23,  $r24,  (si18 | symbol)
bne    $r23,  $r24,  (si18 | symbol)
blt    $r23,  $r24,  (si18 | symbol)
bge    $r23,  $r24,  (si18 | symbol)
bltu   $r23,  $r24,  (si18 | symbol)
bgeu   $r23,  $r24,  (si18 | symbol)
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.4.1` .

8.1.1.4.2 `BEQZ`, `BNEZ`

Syntax:

```
opcode   src1,  src2
```

opcode	src1	src2
beqz, bnez	$rd	si23 | symbol

Description :

beqz : **if** ($rj == 0) **jump** (si23 | symbol)

bnez : **if** ($rj ! = 0) **jump** (si23 | symbol)

- si23 : a 4-bytes aligned 23-bits signed immediate value in range :
  - [-4194304, 4194300] or [-0x400000, 0x3ffffc]
- symbol : Tags in assembly for jump.

Usage :

```
beqz    $r23,  (si23 | symbol)
bnez    $r23,  (si23 | symbol)
```

NOTE For more information, refer to the LoongArch instruction manual:2.2.4.2 .

8.1.1.4.3 B

Syntax:

```
opcode    src1
```

opcode	src1
b	si28 | symbol

Description :

b : **jump** (si28 | symbol)

- si28 : a 4-bytes aligned 28-bits signed immediate value in range :
  - [-134217728, 134217724] or [-0x8000000, 0x7ffffffc]
- symbol : Tags in assembly for jump.

Usage :

```
b       (si28 | symbol)
```

NOTE For more information, refer to the LoongArch instruction manual:2.2.4.3 .

8.1.1.4.4 BL

Syntax:

```
opcode    src1
```

opcode	src1
bl	si28 | symbol

Description :

bl : **jump** (si28 | symbol)

- si28 : a 4-bytes aligned 28-bits signed immediate value in range :
  - [-134217728, 134217724] or [-0x8000000, 0x7ffffc]
- symbol : Tags in assembly for jump.

Usage :

```
bl      (si28 | symbol)
```

NOTE For more information, refer to the LoongArch instruction manual:2.2.4.4 .

8.1.1.4.5 JIRL

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
jirl	$rd	$rj	si28

Description :

jirl : $rd = PC + 4, **jump** si28 + $rj

- si28 : a 4-bytes aligned 28-bits signed immediate value in range :
  - [-134217728, 134217724] or [-0x8000000, 0x7ffffc]
- symbol : Tags in assembly for jump.

NOTE For more information, refer to the LoongArch instruction manual:2.2.4.5 .

8.1.1.5 Common Memory Access Instructions

8.1.1.5.1 `LD.{B[U]/H[U]/W[U]/D}`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`ld.b`, `ld.h`, `ld.w`, `ld.d`, `ld.bu`, `ld.hu`, `ld.wu`	`$rd`	`$rj`	`si12`

Description :

`ld.b` : `$rd` = **SignExtend**(**MemoryLoad**((`$rj` + **SignExtend**(`si12`, GRLEN)), BYTE), GRLEN)

`ld.h` : `$rd` = **SignExtend**(**MemoryLoad**((`$rj` + **SignExtend**(`si12`, GRLEN)), HALFWORD), GRLEN)

`ld.w` : `$rd` = **SignExtend**(**MemoryLoad**((`$rj` + **SignExtend**(`si12`, GRLEN)), WORD), GRLEN)

`ld.d` : `$rd` = **MemoryLoad**((`$rj` + **SignExtend**(`si12`, GRLEN)), DOUBLEWORD)

`ld.bu` : `$rd` = **ZeroExtend**(**MemoryLoad**((`$rj` + **SignExtend**(`si12`, GRLEN)), BYTE), GRLEN)

`ld.hu` : `$rd` = **ZeroExtend**(**MemoryLoad**((`$rj` + **SignExtend**(`si12`, GRLEN)), HALFWORD), GRLEN)

`ld.wu` : `$rd` = **ZeroExtend**(**MemoryLoad**((`$rj` + **SignExtend**(`si12`, GRLEN)), WORD), GRLEN)

- `si12` : 12 bit immediate, Signed value range(`integer`) : [`-2048`, `2047`] or [`-0x800`, `0x7ff`]

Usage :

```
                            # memory[$r22 - 40] = 0xfedcba9876543210
ld.b    $r23, $r22, -40     # $r23 = 0x0000000000000010
ld.h    $r23, $r22, -40     # $r23 = 0x0000000000003210
ld.w    $r23, $r22, -40     # $r23 = 0x0000000076543210
ld.d    $r23, $r22, -40     # $r23 = 0xfedcba9876543210
ld.bu   $r23, $r22, -40     # $r23 = 0x0000000000000010
ld.hu   $r23, $r22, -40     # $r23 = 0x0000000000003210
ld.wu   $r23, $r22, -40     # $r23 = 0x0000000076543210
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.5.1` .

8.1.1.5.2 `ST.{B/H/W/D}`

Syntax:

```
opcode    src1,  src2,  src3
```

opcode	src1	src2	src3
`st.b`, `st.h`, `st.w`, `st.d`	`$rd`	`$rj`	`si12`

Description :

`st.b` : **MemoryStore**(`$rd`[7 :0], (`$rj` + **SignExtend**(`si12`, GRLEN)), BYTE)

`st.h` : **MemoryStore**(`$rd`[15:0], (`$rj` + **SignExtend**(`si12`, GRLEN)), HALFWORD)

`st.w` : **MemoryStore**(`$rd`[31:0], (`$rj` + **SignExtend**(`si12`, GRLEN)), WORD)

`st.d` : **MemoryStore**(`$rd`[63:0], (`$rj` + **SignExtend**(`si12`, GRLEN)), DOUBLEWORD)

- `si12` : 12 bit immediate, Signed value range(`integer`) : [`-2048`, `2047`] or [`-0x800`, `0x7ff`]

Usage :

```
li.w $r23, 0xfedcba9876543210    # $r23 = 0xfedcba9876543210
st.b $r23, $r22, -24             # memory[$r22 - 24] = 0x0000000000000010
st.h $r23, $r22, -24             # memory[$r22 - 24] = 0x0000000000003210
st.w $r23, $r22, -24             # memory[$r22 - 24] = 0x0000000076543210
st.d $r23, $r22, -24             # memory[$r22 - 24] = 0xfedcba9876543210
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.5.1` .

8.1.1.5.3 `LDX.{B[U]/H[U]/W[U]/D}`

Syntax:

```
opcode    dest,   src1,   src2
```

opcode	dest	src1	src2
`ldx.b`, `ldx.h`, `ldx.w`, `ldx.d`, `ldx.bu`, `ldx.hu`, `ldx.wu`	`$rd`	`$rj`	`$rk`

Description :

`ldx.b` : `$rd` = **SignExtend**(**MemoryLoad**((`$rj` + `$rk`), BYTE), GRLEN)

`ldx.h` : `$rd` = **SignExtend**(**MemoryLoad**((`$rj` + `$rk`), HALFWORD), GRLEN)

`ldx.w` : `$rd` = **SignExtend**(**MemoryLoad**((`$rj` + `$rk`), WORD), GRLEN)

`ldx.d` : `$rd` = **MemoryLoad**((`$rj` + `$rk`), DOUBLEWORD)

`ldx.bu` : `$rd` = **ZeroExtend**(**MemoryLoad**((`$rj` + `$rk`), BYTE), GRLEN)

`ldx.hu` : `$rd` = **ZeroExtend**(**MemoryLoad**((`$rj` + `$rk`), HALFWORD), GRLEN) `ldx.wu` : `$rd` = **ZeroExtend**(**MemoryLoad**((`$rj` + `$rk`), WORD), GRLEN)

Usage :

```
                              # memory[$r22 - $r24] = 0xfedcba9876543210
li.w    $r24, -40             # $r24 = -40
ldx.b   $r23, $r22, $r24      # $r23 = 0x0000000000000010
ldx.h   $r23, $r22, $r24      # $r23 = 0x0000000000003210
ldx.w   $r23, $r22, $r24      # $r23 = 0x0000000076543210
ldx.d   $r23, $r22, $r24      # $r23 = 0xfedcba9876543210
ldx.bu  $r23, $r22, $r24      # $r23 = 0x0000000000000010
ldx.hu  $r23, $r22, $r24      # $r23 = 0x0000000000003210
ldx.wu  $r23, $r22, $r24      # $r23 = 0x0000000076543210
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.5.2` .

8.1.1.5.4 `STX.{B/H/W/D}`

Syntax:

```
opcode    src1,  src2,  src3
```

opcode	src1	src2	src3
`stx.b`, `stx.h`, `stx.w`, `stx.d`	`$rd`	`$rj`	`$rk`

Description :

`stx.b` : **MemoryStore**(`$rd`[7 :0], (`$rj` + `$rk`), BYTE)

`stx.h` : **MemoryStore**(`$rd`[15:0], (`$rj` + `$rk`), HALFWORD)

`stx.w` : **MemoryStore**(`$rd`[31:0], (`$rj` + `$rk`), WORD)

`stx.d` : **MemoryStore**(`$rd`[63:0], (`$rj` + `$rk`), DOUBLEWORD)

Usage :

```
li.w   $r23, 0xfedcba9876543210   # $r23 = 0xfedcba9876543210
li.w   $r24, -40                  # $r24 = -40
stx.b  $r23, $r22, $r24           # memory[$r22 - $r24] = 0x0000000000000010
stx.h  $r23, $r22, $r24           # memory[$r22 - $r24] = 0x0000000000003210
stx.w  $r23, $r22, $r24           # memory[$r22 - $r24] = 0x0000000076543210
stx.d  $r23, $r22, $r24           # memory[$r22 - $r24] = 0xfedcba9876543210
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.5.2` .

8.1.1.5.5 `LDPTR.{W/D}`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`ldptr.w`, `ldptr.d`	`$rd`	`$rj`	`si14`

Description :

`ldptr.w` : `$rd` = **SignExtend**(**MemoryLoad**((`$rj` + **SignExtend**(`si16`, GRLEN)), WORD), GRLEN)

`ldptr.d` : `$rd` = **MemoryLoad**((`$rj` + **SignExtend**(`si16`, GRLEN)), WORD)

- `si16` : a 4-bytes aligned 16-bits signed immediate value in range :
    - [`-32768`, `32764`] or [`-0x8000`, `0x7ffc`]

Usage :

```
ldptr.w $r23, $r22, -40    # $r23 = 0x0000000076543210
ldptr.d $r23, $r22, -40    # $r23 = 0xfedcba9876543210
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.5.3` .

8.1.1.5.6 `STPTR.{W/D}`

Syntax:

```
opcode    src1,  src2,  src3
```

opcode	src1	src2	src3
`stptr.w`, `stptr.d`	`$rd`	`$rj`	`si14`

Description :

`stptr.w` : **MemoryStore**(`$rd`[31:0], (`$rj` + **SignExtend**({`si14`, 2’b0}, GRLEN)), WORD)

`stptr.d` : **MemoryStore**(`$rd`[63:0], (`$rj` + **SignExtend**({`si14`, 2’b0}, GRLEN)), DOUBLEWORD)

- `si16` : a 4-bytes aligned 16-bits signed immediate value in range :
    - [`-32768`, `32764`] or [`-0x8000`, `0x7ffc`]

Usage :

```
stptr.w $r23, $r22, -40    # memory[$r22 - 40] = 0x0000000076543210
stptr.d $r23, $r22, -40    # memory[$r22 - 40] = 0xfedcba9876543210
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.5.3`.

8.1.1.5.6 `PRELD`, `PRELDX`

Syntax:

```
opcode    src1,  src2,  src3
```

opcode	src1	src2	src3
`preld`	`hint`	`$rj`	`si12`
`preldx`	`hint`	`$rj`	`$rk`

Description :

`preld` :

- The processor learns from the hint in the `PRELD` instruction what type will be acquired and which level of `Cache` the data to be taken back fill in, `hint` has 32 optional values (0 to 31), 0 represents load to level 1 `Cache`, and 8 represents store to level 1 `Cache`. The remaining `hint` values are not defined and are processed for nop instructions when the processor executes.
- `si12` : 12 bit immediate, Signed value range(`integer`) : [`-2048`, `2047`] or [`-0x800`, `0x7ff`]

`preldx` :

- The `PRELDX` instruction continuously prefetches data from memory into the Cache according to the configuration parameters, and the continuously prefetched data is a `block` (`block`) of length `block_size` starting from the specified base `address` (`base`) with a number of (`block_num`) spacing stride. The `base address` is the sum of the [63:0] bits in the general register `rj` and the sign extension [15:0] bits in the general register `rk`. The [I16] bits in general register `rk` are the address sequence ascending and descending flag bits, with 0 indicating address ascending and 1 indicating address descending. The value of bits [25:20] in general register `rk` is `block_size`, the basic unit of `block_size` is 16 bytes, so the maximum length of a single `block` is 1KB. The value of bits [39:32] in general register `rk` is `block_num-1`, so a single instruction can prefetch up to 256 `blocks`. The value of bits [59:44] in the block general register `rk` is treated as a signed number and defines the stride between adjacent blocks, the basic unit of stride is 1 byte. The value of bits [39:32] in `rk` is `block.num-1`, so a single instruction can prefetch up to 256 blocks. The value of bits [59:44] in general register `rk` is regarded as a signed number, which defines the corresponding The basic unit of stride and stride between adjacent blocks is 1 byte.
- `hint` in the `PRELDX` instruction indicates the type of prefetch and the level of `Cache` into which the fetched data is to be filled. hint has 32 selectable values from 0 to 31. Currently, `hint=0` is defined as load prefetch to level 1 data `Cache`, `hint=2` is defined as load prefetch to level 3 `Cache`, `hint=8` is

defined as store prefetch to level 1 data `Cache`. The meaning of the rest of `hint` values is not defined yet, and the processor executes it as `NOP` instruction.

NOTE For more information, refer to the `LoongArch instruction manual:2.2.5.4` / `2.2.5.5` .

8.1.1.6 Bound Check Memory Access Instructions

8.1.1.6.1 `LD{GT/LE}.{B/H/W/D}`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`ldgt.b`, `ldgt.h`, `ldgt.w`, `ldgt.d`, `ldle.b`, `ldle.h`, `ldle.w`, `ldle.d`	`$rd`	`$rj`	`$rk`

Description :

`ldgt.{b/h/w/d}` :

- **if**(`$rj` > `$rk`)`$rd` = **SignExtend**(**MemoryLoad** (`$rj`, `byte`/`halfword`/`word`/`doubleword`), GRLEN)
- **else** : **RaiseException**(BCE)

`ldle.{b/h/w/d}` :

- **if**(`$rj` ⇐ `$rk`)`$rd` = **SignExtend**(**MemoryLoad** (`$rj`, `byte`/`halfword`/`word`/`doubleword`), GRLEN)
- **else** : **RaiseException**(BCE)

Usage :

```
ldgt.b     $r23,  $r24,  $r25    # $r23 = 0x0000000000000010
ldgt.h     $r23,  $r24,  $r25    # $r23 = 0x0000000000003210
ldgt.w     $r23,  $r24,  $r25    # $r23 = 0x0000000076543210
ldgt.d     $r23,  $r24,  $r25    # $r23 = 0xfedcba9876543210
ldle.b     $r23,  $r24,  $r25    # $r23 = 0x0000000000000010
ldle.h     $r23,  $r24,  $r25    # $r23 = 0x0000000000003210
ldle.w     $r23,  $r24,  $r25    # $r23 = 0x0000000076543210
ldle.d     $r23,  $r24,  $r25    # $r23 = 0xfedcba9876543210
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.6.1` .

8.1.1.6.2 `ST{GT/LE}.{B/H/W/D}`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	src1	src2	src3
`stgt.b`, `stgt.h`, `stgt.w`, `stgt.d`, `stle.b`, `stle.h`, `stle.w`, `stle.d`	`$rd`	`$rj`	`$rk`

Description :

`stgt.{b/h/w/d}` :

- **if**(`$rj` > `$rk`)
  - **MemoryStore**(`$rd`[7 :0],`$rj`,`byte`)/
  - **MemoryStore**(`$rd`[15:0],`$rj`,`halfword`)/
  - **MemoryStore**(`$rd`[31:0],`$rj`,`word`)/
  - **MemoryStore**(`$rd`[63:0],`$rj`,`doubleword`)
- **else** :
  - **RaiseException**(BCE)

`stle.{b/h/w/d}` :

- **if**(`$rj` < = `$rk`)
  - **MemoryStore**(`$rd`[7 :0],`$rj`,`byte`)/
  - **MemoryStore**(`$rd`[15:0],`$rj`,`halfword`)/
  - **MemoryStore**(`$rd`[31:0],`$rj`,`word`)/
  - **MemoryStore**(`$rd`[63:0],`$rj`,`doubleword`)
- **else** :
  - **RaiseException**(BCE)

Usage :

```
stgt.b     $r23,  $r24,  $r25    # memory[$r24] = 0x0000000000000010
stgt.h     $r23,  $r24,  $r25    # memory[$r24] = 0x0000000000003210
stgt.w     $r23,  $r24,  $r25    # memory[$r24] = 0x0000000076543210
stgt.d     $r23,  $r24,  $r25    # memory[$r24] = 0xfedcba9876543210
stle.b     $r23,  $r24,  $r25    # memory[$r24] = 0x0000000000000010
stle.h     $r23,  $r24,  $r25    # memory[$r24] = 0x0000000000003210
stle.w     $r23,  $r24,  $r25    # memory[$r24] = 0x0000000076543210
stle.d     $r23,  $r24,  $r25    # memory[$r24] = 0xfedcba9876543210
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.6.1` .

8.1.1.7 Atomic Memory Access Instructions

8.1.1.7.1 AM{SWAP/ADD/AND/OR/XOR/MAX/MIN}[_DB].{W/D}, AM{MAX/MIN}[_DB].{WU/DU}

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
amswap.w, amswap_db.w, amswap.d, amswap_db.d	$rd	$rk	$rj
amadd.w, amadd_db.w, amadd.d, amadd_db.d	$rd	$rk	$rj
amand.w, amand_db.w, amand.d, amand_db.d	$rd	$rk	$rj
amor.w, amor_db.w, amor.d, amor_db.d	$rd	$rk	$rj
amxor.w, amxor_db.w, amxor.d, amxor_db.d	$rd	$rk	$rj
ammax.w, ammax_db.w, ammax.d, ammax_db.d	$rd	$rk	$rj
ammin.w, ammin_db.w, ammin.d, ammin_db.d	$rd	$rk	$rj
ammax.wu, ammax_db.wu, ammax.du, ammax_db.du	$rd	$rk	$rj
ammin.wu, ammin_db.wu, ammin.du, ammin_db.du	$rd	$rk	$rj

Description :

amswap.w : $rd = **MemoryLoad**($rj, word), **MemoryStore**($rk, $rj, word)

amswap_db.w : $rd = **MemoryLoad**($rj, word), **MemoryStore**($rk, $rj, word)

amswap.d : $rd = **MemoryLoad**($rj, doubleword), **MemoryStore**($rk, $rj, doubleword)

amswap_db.d : $rd = **MemoryLoad**($rj, doubleword), **MemoryStore**($rk, $rj, doubleword)

Usage :

```
li.d       $r26, 0x0123456789abcdef
li.d       $r24, &(0xfedcba9876543210)
amswap.d   $r25, $r26, $r24
```

- **Explanation :**
    - $r25 = 0xfedcba9876543210
    - $r24 = address → 0x0123456789abcdef

Description :

amadd.w : $rd = **MemoryLoad**($rj, word), **MemoryStore**($rd + $rk, $rj, word)

amadd_db.w : $rd = **MemoryLoad**($rj, word), **MemoryStore**($rd + $rk, $rj, word) `amadd.d` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(`$rd` + `$rk`, `$rj`, doubleword)

`amadd_db.d` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(`$rd` + `$rk`, `$rj`, doubleword)

`amand.w` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(`$rd` & `$rk`, `$rj`, word)

`amand_db.w` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(`$rd` & `$rk`, `$rj`, word)

`amand.d` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(`$rd` & `$rk`, `$rj`, doubleword)

`amand_db.d` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(`$rd` & `$rk`, `$rj`, doubleword)

`amor.w` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(`$rd` | `$rk`, `$rj`, word)

`amor_db.w` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(`$rd` | `$rk`, `$rj`, word)

`amor.d` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(`$rd` | `$rk`, `$rj`, doubleword)

`amor_db.d` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(`$rd` | `$rk`, `$rj`, doubleword)

`amxor.w` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(`$rd` ^ `$rk`, `$rj`, word)

`amxor_db.w` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(`$rd` ^ `$rk`, `$rj`, word)

`amxor.d` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(`$rd` ^ `$rk`, `$rj`, doubleword)

`amxor_db.d` : `$rd` = **MemoryLoad**(`$rj`,doubleword), **MemoryStore**(`$rd` ^ `$rk`, `$rj`, doubleword)

`ammax.w` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(max{`$rd`, `$rk`}, `$rj`, word)

`ammax_db.w` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(max{`$rd`, `$rk`}, `$rj`, word)

`ammax.d` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(max{`$rd`, `$rk`}, `$rj`, doubleword)

`ammax_db.d` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(max{`$rd`, `$rk`}, `$rj`, doubleword)

`ammin.w` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(min{`$rd`, `$rk`}, `$rj`, word)

`ammin_db.w` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(min{`$rd`, `$rk`}, `$rj`, word)

`ammin.d` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(min{`$rd`, `$rk`}, `$rj`, doubleword)

`ammin_db.d` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(min{`$rd`, `$rk`}, `$rj`, doubleword)

`ammax.wu` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(max{`$rd`, `$rk`}, `$rj`, word)

`ammax_db.wu` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(max{`$rd`, `$rk`}, `$rj`, word)

`ammax.du` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(max{`$rd`, `$rk`}, `$rj`, doubleword)

`ammax_db.du` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(max{`$rd`, `$rk`}, `$rj`, doubleword)

`ammin.wu` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(min{`$rd`, `$rk`}, `$rj`, word) `ammin_db.wu` : `$rd` = **MemoryLoad**(`$rj`, word), **MemoryStore**(min{`$rd`, `$rk`}, `$rj`, word)

`ammin.du` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(min{`$rd`, `$rk`}, `$rj`, doubleword)

`ammin_db.du` : `$rd` = **MemoryLoad**(`$rj`, doubleword), **MemoryStore**(min{`$rd`, `$rk`}, `$rj`, doubleword)

NOTE For more information, refer to the `LoongArch instruction manual:2.2.7.1` .

8.1.1.7.2 `LL.{W/D}`, `SC.{W/D}`

Syntax:

```
opcode    dest,   src1,   src2
```

opcode	dest	src1	src2
`ll.w`, `ll.d`, `sc.w`, `sc.d`	`$rd`	`$rj`	`si16`

- **Explanation :**

`ll.w` : `$rd`[31:0] = **MemoryLoad**(`$rj` + {`si16`}, word)

`ll.d` : `$rd` = **MemoryLoad**(`$rj` + {`si16`}, doubleword)

`sc.w` : **MemoryStore**(`$rd`[31:0], `$rj` + {`si16`}, word)

`sc.d` : **MemoryStore**(`$rd`, `$rj` + {`si16`}, doubleword)

- `si16` : a 4-bytes aligned 16-bits signed immediate value in range :
  - [`-32768`, `32764`] or [`-0x8000`, `0x7ffc`]

Usage :

```
ll.w    $r25, $r12, 0
addi.w  $r25, $r25, 5
sc.w    $r25, $r12, 0
ll.d    $r25, $r12, 0
addi.d  $r25, $r25, 10
sc.d    $r25, $r12, 0
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.7.2` .

8.1.1.8 Barrier Instructions

8.1.1.8.1 DBAR

Syntax:

```
opcode    src1
```

opcode	src1
dbar	hint

Description :

The DBAR instruction is used to complete the barrier function between load/store memory access operations. The immediate hint it carries is used to indicate the synchronization object and synchronization degree of the barrier.

A hint value of 0 is mandatory by default, and it indicates a fully functional synchronization barrier. Only after all previous load/store access operations are completely executed, the DBAR 0 instruction can be executed; and only after the execution of DBAR 0 is completed, all subsequent load/store access operations can be executed.

If there is no special function implementation, all other hint values must be executed according to hint=0.

NOTE For more information, refer to the LoongArch instruction manual:2.2.8.1 .

8.1.1.8.2 IBAR

Syntax:

```
opcode    src1
```

opcode	src1
ibar	hint

Description :

The IBAR instruction is used to complete the synchronization between the store operation and the instruction fetch operation within a single processor core. The immediate hint it carries is used to indicate the synchronization object and synchronization degree of the barrier.

A hint value of 0 is mandatory by default. It can ensure that the instruction fetch after the IBAR 0 instruction must be able to observe the execution effect of all store operations before the IBAR 0 instruction.

NOTE For more information, refer to the LoongArch instruction manual:2.2.8.2 .

8.1.1.9 CRC Check Instructions

8.1.1.9.1 `CRC[C].W.{B/H/W/D}.W`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`crc.w.b.w`, `crc.w.h.w`, `crc.w.w.w`, `crc.w.d.w`	`$rd`	`$rj`	`$rk`
`crcc.w.b.w`, `crcc.w.h.w`, `crcc.w.w.w`, `crcc.w.d.w`	`$rd`	`$rj`	`$rk`

Description :

`crc.w.b.w` : `$rd` = **SignExtend**(**CRC32**(`$rk`[31:0], `$rj`[7 :0], 8 , 0xEDB88320), GRLEN)

`crc.w.h.w` : `$rd` = **SignExtend**(**CRC32**(`$rk`[31:0], `$rj`[15:0], 16, 0xEDB88320), GRLEN)

`crc.w.w.w` : `$rd` = **SignExtend**(**CRC32**(`$rk`[31:0], `$rj`[31:0], 32, 0xEDB88320), GRLEN)

`crc.w.d.w` : `$rd` = **SignExtend**(**CRC32**(`$rk`[31:0], `$rj`[63:0], 64, 0xEDB88320), GRLEN)

`crcc.w.b.w` : `$rd` = **SignExtend**(**CRC32**(`$rk`[31:0], `$rj`[7 :0], 8 , 0x82F63B78), GRLEN)

`crcc.w.h.w` : `$rd` = **SignExtend**(**CRC32**(`$rk`[31:0], `$rj`[15:0], 16, 0x82F63B78), GRLEN)

`crcc.w.w.w` : `$rd` = **SignExtend**(**CRC32**(`$rk`[31:0], `$rj`[31:0], 32, 0x82F63B78), GRLEN)

`crcc.w.d.w` : `$rd` = **SignExtend**(**CRC32**(`$rk`[31:0], `$rj`[63:0], 64, 0x82F63B78), GRLEN)

Usage :

```
crc.w.b.w    $r24, $r25, $r26
crc.w.h.w    $r24, $r25, $r26
crc.w.w.w    $r24, $r25, $r26
crc.w.d.w    $r24, $r25, $r26
crc.w.b.w    $r24, $r25, $r26
crc.w.h.w    $r24, $r25, $r26
crc.w.w.w    $r24, $r25, $r26
crc.w.d.w    $r24, $r25, $r26
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.9.1` .

8.1.1.10 Other Miscellaneous Instructions

8.1.1.10.1 `SYSCALL`

Syntax:

```
opcode    src1
```

opcode	src1
`syscall`	`code`

Description :

Executing the SYSCALL instruction will immediately and unconditionally trigger the system call exception.

NOTE For more information, refer to the `LoongArch instruction manual:2.2.10.1` .

8.1.1.10.2 `BREAK`

Syntax:

```
opcode    src1
```

opcode	src1
`break`	`code`

Description :

Executing the BREAK instruction will immediately and unconditionally trigger the breakpoint exception.

NOTE For more information, refer to the `LoongArch instruction manual:2.2.10.2` .

8.1.1.10.3 `ASRT{LE/GT}.D`

Syntax:

```
opcode    src1,  src2
```

opcode	src1	src2
`asrtle.d`, `asrtgt.d`	`$rj`	`$rk`

Description :

`asrtle.d` : **if** (`$rj` > `$rk`) **RaiseException**(BCE) `asrtgt.d` : **if** (`$rj` < = `$rk`) **RaiseException**(BCE)

NOTE For more information, refer to the `LoongArch instruction manual:2.2.10.3` .

8.1.1.10.4 `RDTIME{L/H}.W`, `RDTIME.D`

Syntax:

```
opcode    dest,   dest
```

opcode	dest	dest
`rdtimel.w`, `rdtimeh.w`, `rdtime.d`	`$rd`	`$rj`

Description :

`rdtimel.w` : `$rd` = **Stable_Counter**[31:0], `$rj` = **Counter ID**

`rdtimeh.w` : `$rd` = **Stable_Counter**[63:32], `$rj` = **Counter ID**

`rdtime.d` : `$rd` = **Stable_Counter**[63:0], `$rj` = **Counter ID**

NOTE For more information, refer to the `LoongArch instruction manual:2.2.10.4` .

8.1.1.10.5 `CPUCFG`

Syntax:

```
opcode    src1,   src2
```

opcode	dest	dest
`cpucfg`	`$rd`	`$rj`

Description :

`cpucfg` : When using the `CPUCFG` instruction, the source operand register `rj` stores the number of the configuration information word to be accessed, and the configuration information word information read after the instruction is executed is written into the general register `rd`. In **LA64**, each configuration information word is 32 bits, which is written into the result register after the sign extension.

NOTE For more information, refer to the `LoongArch instruction manual:2.2.10.5` .

8.1.2 Base Floating-point Instruction

This chapter will introduce the floating-point instructions in the non privileged subset foundation of the LoongArch architecture. Functional definition of the Basic Floating-point Instruction in the LoongArch Architecture.

Comply with `IEEE 754-2008 standard`. The basic floating-point instruction cannot be implemented separately from the basic integer instruction. Both the basic integer instruction and the basic floating-point instruction need to be implemented simultaneously. Whether the implementation of basic floating-point instructions includes instructions for manipulating double precision floating-point numbers and doubleword integers is independent of whether the architecture is `LA32` or `LA64`.

8.1.2.1 Floating-Point Arithmetic Operation Instructions

8.1.2.1.1 `F{ADD/SUB/MUL/DIV}.{S/D}`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`fadd.s`, `fadd.d`, `fsub.s`, `fsub.d`, `fmul.s`, `fmul.d`, `fdiv.s`, `fdiv.d`	`$fd`	`$fj`	`$fk`

Description :

`fadd.s` : `$fd`[31:0] = **FP32_addition**(`$fj`[31:0], `$fk`[31:0])

`fadd.d` : `$fd` = **FP64_addition**(`$fj`, `$fk`)

`fsub.s` : `$fd`[31:0] = **FP32_subtraction**(`$fj`[31:0], `$fk`[31:0])

`fsub.d` : `$fd` = **FP64_subtraction**(`$fj`, `$fk`)

`fmul.s` : `$fd`[31:0] = **FP32_multiplication**(`$fj`[31:0], `$fk`[31:0])

`fmul.d` : `$fd` = **FP64_multiplication**(`$fj`, `$fk`)

`fdiv.s` : `$fd`[31:0] = **FP32_division**(`$fj`[31:0], `$fk`[31:0])

`fdiv.d` : `$fd` = **FP64_division**(`$fj`, `$fk`)

- When the operand is a single-precision floating-point number, the upper 32 bits of the resulting floating-point register can be any value.
- The floating-point `addition` / `subtraction` / `multiplication` / `division` operation follows the `addition`(x, y) / `subtraction`(x, y) / `multiplication`(x, y) / `division`(x, y) operation specification in the `IEEE 754-2008 standard` .

Usage :

```
fadd.s   $f23, $f24, $f25
fadd.d   $f23, $f24, $f25
fsub.s   $f23, $f24, $f25
fsub.d   $f23, $f24, $f25
fmul.s   $f23, $f24, $f25
fmul.d   $f23, $f24, $f25
fdiv.s   $f23, $f24, $f25
fdiv.d   $f23, $f24, $f25
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.1.1`.

8.1.2.1.2 `F{MADD/MSUB/NMADD/NMSUB}.{S/D}`

Syntax:

```
opcode    dest,   src1,   src2,   src3
```

opcode	dest	src1	src2	src3
`fmadd.s`, `fmadd.d`, `fmsub.s`, `fmsub.d`	`$fd`	`$fj`	`$fk`	`$fa`
`fnmadd.s`, `fnmadd.d`, `fnmsub.s`, `fnmsub.d`	`$fd`	`$fj`	`$fk`	`$fa`

Description :

`fmadd.s` : `$fd`[31:0] = **FP32_fusedMultiplyAdd**(`$fj`[31:0], `$fk`[31:0], `$fa`[31:0])

`fmadd.d` : `$fd` = **FP64_fusedMultiplyAdd**(`$fj`, `$fk`, `$fa`)

`fmsub.s` : `$fd`[31:0] = **FP32_fusedMultiplyAdd**(`$fj`[31:0], `$fk`[31:0], -`$fa`[31:0])

`fmsub.d` : `$fd` = **FP64_fusedMultiplyAdd**(`$fj`, `$fk`, -`$fa`)

`fnmadd.s` : `$fd`[31:0] = **- FP32_fusedMultiplyAdd**(`$fj`[31:0], `$fk`[31:0], `$fa`[31:0])

`fnmadd.d` : `$fd` = **- FP64_fusedMultiplyAdd**(`$fj`, `$fk`, `$fa`)

`fnmsub.s` : `$fd`[31:0] = **- FP32_fusedMultiplyAdd**(`$fj`[31:0], `$fk`[31:0], -`$fa`[31:0])

`fnmsub.d` : `$fd` = **- FP64_fusedMultiplyAdd**(`$fj`, `$fk`, -`$fa`)

- The above four floating-point `fusion multiply-add` operations follow the specification of the `fusedMultiplyAdd`(x,y,z) operation in the `IEEE 754-2008 standard`.

Usage :

```
fmadd.s    $f23, $f24, $f25, $f26
fmadd.d    $f23, $f24, $f25, $f26
fmsub.s    $f23, $f24, $f25, $f26
fmsub.d    $f23, $f24, $f25, $f26
fnmadd.s   $f23, $f24, $f25, $f26
fnmadd.d   $f23, $f24, $f25, $f26
fnmsub.s   $f23, $f24, $f25, $f26
fnmsub.d   $f23, $f24, $f25, $f26
```

- **Explanation :**
  - `fmadd.s` : `$f23` = `$f24` × `$f25` + `$f26`

NOTE For more information, refer to the `LoongArch instruction manual:3.2.1.2` .

8.1.2.1.3 `F{MAX/MIN}{S/D}`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`fmax.s`, `fmax.d`, `fmin.s`, `fmin.d`	`$fd`	`$fj`	`$fk`

Description :

`fmax.s` : `$fd`[31:0] = **FP32_maxNum**(`$fj`[31:0], `$fk`[31:0])

`fmax.d` : `$fd` = **FP64_maxNum**(`$fj`, `$fk`)

- `FMAX{S/D}` follows the specification of `maxNum`(x,y) operation in `IEEE 754-2008 standard` .

`fmin.s` : `$fd`[31:0] = **FP32_minNum**(`$fj`[31:0], `$fk`[31:0])

`fmin.d` : `$fd` = **FP64_minNum**(`$fj`, `$fk`)

- `FMIN{S/D}` follows the specification of `minNum`(x,y) operation in `IEEE 754-2008 standard` .

Usage :

```
fmax.s    $f23, $f24, $f25  # $f23 = max{$f24, $f25}
fmax.d    $f23, $f24, $f25  # $f23 = max{$f24, $f25}
fmin.s    $f23, $f24, $f25  # $f23 = min{$f24, $f25}
fmin.d    $f23, $f24, $f25  # $f23 = min{$f24, $f25}
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.1.3`.

8.1.2.1.4 `F{MAXA/MINA}.{S/D}`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`fmaxa.s`, `fmaxa.d`, `fmina.s`, `fmina.d`	`$fd`	`$fj`	`$fk`

Description :

`fmaxa.s` : `$fd`[31:0] = **FP32_maxNumMag**(`$fj`[31:0], `$fk`[31:0])

`fmaxa.d` : `$fd` = **FP64_maxNumMag**(`$fj`, `$fk`)

- `FMAXA{S/D}` follows the specification of `maxNumMag`(x,y) operation in `IEEE 754-2008 standard`.

`fmina.s` : `$fd`[31:0] = **FP32_minNumMag**(`$fj`[31:0], `$fk`[31:0])

`fmina.d` : `$fd` = **FP64_minNumMag**(`$fj`, `$fk`)

- `FMINA{S/D}` follows the specification of `minNumMag`(x,y) operation in `IEEE 754-2008 standard`.

Usage :

```
fmaxa.s    $f23, $f24, $f25  # $f23 = max{|$f24|, |$f25|}
fmaxa.d    $f23, $f24, $f25  # $f23 = max{|$f24|, |$f25|}
fmina.s    $f23, $f24, $f25  # $f23 = min{|$f24|, |$f25|}
fmina.d    $f23, $f24, $f25  # $f23 = min{|$f24|, |$f25|}
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.1.4`.

8.1.2.1.5 `F{ABS/NEG}.{S/D}`

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
`fabs.s`, `fabs.d`, `fneg.s`, `fneg.d`	`$fd`	`$fj`

Description :

`fabs.s` : `$fd`[31:0] = **FP32_maxNumMag**(`$fj`[31:0]) `fabs.d` : `$fd` = **FP64_maxNumMag**(`$fj`)

- `FABS.{S/D}` follows the specification of `abs`(x) operation in `IEEE 754-2008 standard` .

`fneg.s` : `$fd`[31:0] = **FP32_minNumMag**(`$fj`[31:0])

`fneg.d` : `$fd` = **FP64_minNumMag**(`$fj`)

- `FNEG.{S/D}` follows the specification of `negate`(x) operation in `IEEE 754-2008 standard` .

Usage :

```
fabs.s    $f23, $f24    # $f23 = |$f24|
fabs.d    $f23, $f24    # $f23 = |$f24|
fneg.s    $f23, $f24    # $f23 = -$f24
fneg.d    $f23, $f24    # $f23 = -$f24
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.1.5` .

8.1.2.1.6 `F{SQRT/RECIP/RSQRT}.{S/D}`

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
`fsqrt.s`, `fsqrt.d`, `frecip.s`, `frecip.d`, `frsqrt.s`, `frsqrt.d`	`$fd`	`$fj`

Description :

`fsqrt.s` : `$fd`[31:0] = **FP32_squareRoot**(`$fj`[31:0])

`fsqrt.d` : `$fd` = **FP64_squareRoot**(`$fj`)

- The floating-point `square root` operation follows the specifications of the `squareRoot`(x) operation in the `IEEE 754-2008 standard` .

`frecip.s` : `$fd`[31:0] = **FP32_division**(1.0, `$fj`[31:0])

`frecip.d` : `$fd` = **FP64_division**(1.0, `$fj`)

- `FP32_Division` / `FP64_division` is equivalent to the `division`(1.0, x) in the `IEEE 754-2008 standard` .

`frsqrt.s` : `$fd`[31:0] = **FP32_division**(1.0, **FP32_squareRoot**(`$fj`[31:0]))

`frsqrt.d` : `$fd` = **FP64_division**(1.0, **FP64_squareRoot**(`$fj`))

- The floating-point `square root` inversion operation follows the specifications of `rSqrt`(x) operation in `IEEE 754-2008 standard`.

Usage :

```
fsqrt.s     $f23, $f24
fsqrt.d     $f23, $f24
frecip.s    $f23, $f24
frecip.d    $f23, $f24
frsqrt.s    $f23, $f24
frsqrt.d    $f23, $f24
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.1.6` .

8.1.2.1.7 `F{SCALEB/LOGB/COPYSIGN}.{S/D}`

Syntax:

```
opcode    dest,  src1,  {src2}
```

opcode	dest	src1	src2
`flogb.s`, `flogb.d`	`$fd`	`$fj`	
`fscaleb.s`, `fscaleb.d`, `fcopysign.s`, `fcopysign.d`	`$fd`	`$fj`	`$fk`

Description :

`flogb.s` : `$fd`[31:0] = **FP32_logB**(`$fj`[31:0])

`flogb.d` : `$fd` = **FP64_logB**(`$fj`)

- `LOGB.{S/D}` follows the specification of `logB`(x) operation in `IEEE 754-2008 standard` .

`fscaleb.s` : `$fd`[31:0] = **FP32_scaleB**(`$fj`[31:0], `$fk`[31:0])

`fscaleb.d` : `$fd` = **FP64_scaleB**(`$fj`, `$fk`)

- `FSCALEB.{S/D}` follows the specification of `scaleB`(x, N) operation in `IEEE 754-2008 standard` .

`fcopysign.s` : `$fd`[31:0] = **FP32_copySign**(`$fj`[31:0], `$fk`[31:0])

`fcopysign.d` : `$fd` = **FP64_copySign**(`$fj`, `$fk`)

- `COPYSIGN.{S/D}` follows the specification of `copySign`(x, y) operation in `IEEE 754-2008 standard` .

Usage :

```
flogb.s       $f23, $f24
flogb.d       $f23, $f24
fscaleb.s     $f23, $f24, $f25
fscaleb.d     $f23, $f24, $f25
fcopysign.s   $f23, $f24, $f25
fcopysign.d   $f23, $f24, $f25
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.1.7` .

8.1.2.1.8 `FCLASS.{S/D}`

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
`fclass.s`, `fclass.d`	`$fd`	`$fj`

Description :

`fclass.s` : `$fd`[31:0] = **FP32_class**(`$fj`[31:0])

`fclass.d` : `$fd` = **FP64_class**(`$fj`)

- `FCLASS.{S/D}` follows the specification of `class`(x) operation in `IEEE 754-2008 standard` .

Usage :

```
flogb.s       $f23, $f24
flogb.d       $f23, $f24
```

This instruction determines the category of floating-point numbers in the floating-point register `fj`, and the resulting judgment result consists of a total of 10 bits of information. The meaning of each bit is as follows:

bit `0`	bit `1`	bit `2`	bit `3`	bit `4`	bit `5`	bit `6`	bit `7`	bit `8`	bit `9`
SNaN	QNaN	negative value				positive value			
		∞	normal	subnormal	0	∞	normal	subnormal	0

NOTE For more information, refer to the `LoongArch instruction manual:3.2.1.8` .

8.1.2.2 Floating-Point Comparison Instructions

8.1.2.2.1 `FCMP.cond.{S/D}`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`fcmp.cond.s`, `fcmp.cond.d`	`$fcc[ca]`	`$fj`	`$fk`

This is a floating-point comparison instruction, which stores the result of the comparison into the specified status code (CC). There are 22 types of cond for this instruction. These comparison conditions and judgment standards are listed in the following table .

Mnemonic	Meaning	True Condition	QNaN Exception
`CAF`	`None`	`None`	`No`
`CUN`	`Incomparable`	`UN`	`No`
`CEQ`	`Equal`	`EQ`	`No`
`CUEQ`	`Equal`, `incomparable`	`UN`, `EQ`	`No`
`CLT`	`Less than`	`LT`	`No`
`CULT`	`Less than`, `incomparable`	`UN`,`LT`	`No`
`CLE`	`Less than`, `Equal`	`LT`, `EQ`	`No`
`CULE`	`Less than`, `Equal`, `incomparable`	`UN`, `LT`, `EQ`	`No`
`CNE`	`Vary`	`GT`, `LT`	`No`
`COR`	`Orderly`	`GT`, `LT`, `EQ`	`No`
`CUNE`	`Incomparable`, `unequal`	`UN`, `GT`, `LT`	`No`
`SAF`	`None`	`None`	`Yes`
`SUN`	Not `greater than`, Not `equal`	`UN`	`Yes`
`SEQ`	`Equal`	`EQ`	`Yes`
`SUEQ`	Not `greater than`, Not `less than`	`UN`, `EQ`	`Yes`
`SLT`	`Less than`	`LT`	`Yes`
`SULT`	Not `greater than`, Not `equal`	`UN`,`LT`	`Yes`
`SLE`	`Less than`, `Equal`	`LT`, `EQ`	`Yes`
`SULE`	Not `greater than`	`UN`, `LT`, `EQ`	`Yes`
`SNE`	`Vary`	`GT`, `LT`	`Yes`
`SOR`	`Orderly`	`GT`, `LT`, `EQ`	`Yes`
`SUNE`	`Incomparable`, `unequal`	`UN`, `GT`, `LT`	`Yes`

Usage :

```
fcmp.slt.s   $fcc0, $f23, $f24
bceqz        $fcc0, .L128
nop
.L128:
nop
```

- **Explanation :**
  - `fcmp.slt.s`
    - **if** (`$f23` < `$f24`) `$fcc0` = **1**
    - **else** `$fcc0` = **0**
  - `bceqz`
    - **if** (`$fcc0` == **0**) **jump** `.L128`

NOTE For more information, refer to the `LoongArch instruction manual:3.2.2.1` .

8.1.2.3 Floating-Point Conversion Instructions

8.1.2.3.1 FCVT.S.D, FCVT.D.S

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
fcvt.s.d, fcvt.d.s	$fd	$fj

Description :

fcvt.s.d : $fd[31:0] = **FP32_convertFormat**($fj, FP64)

fcvt.d.s : $fd = **FP64_convertFormat**($fj[31:0], FP32)

- The floating-point format conversion operation follows the specification of the convertFormat(x) operation in the IEEE 754-2008 standard .

Usage :

```
fcvt.s.d   $f23, $f27
fcvt.d.s   $f26, $f24
```

NOTE For more information, refer to the LoongArch instruction manual:3.2.3.1 .

8.1.2.3.2 FFINT.{S/D}.{W/L}, FTINT.{W/L}.{S/D}

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
ffint.s.w, ffint.s.l, ffint.d.w, ffint.d.l	$fd	$fj
ftint.w.s, ftint.w.d, ftint.l.s, ftint.l.d	$fd	$fj

Description :

ffint.s.w : $fd[31:0] = **FP32_convertFromInt**($fj[31:0], SINT32)

ffint.s.l : $fd[31:0] = **FP32_convertFromInt**($fj, SINT64)

ffint.d.w : $fd = **FP64_convertFromInt**($fj[31:0], SINT32)

ffint.d.l : $fd = **FP64_convertFromInt**($fj, SINT64)

- The `FFINT{S/D}.{W/L}` instruction selects the `integer`/`long-integer` fixed-point number in the floating-point register `fj` and converts it into a `single`/`double` floating-point number, and the obtained `single`/`double` floating-point number is written to Floating-point register `fd`. The floating-point `format conversion` operation follows the specifications of the `convertFromInt`(x) operation in the `IEEE 754-2008 standard` .

`ftint.w.s` : `$fd`[31:0] = **FP32convertToSint32**(`$fj`[31:0], FCSR.Enables.I, FCSR.RM)

`ftint.w.d` : `$fd` = **FP64convertToSint32**(`$fj`, FCSR.Enables.I, FCSR.RM)

`ftint.l.s` : `$fd`[31:0] = **FP32convertToSint64**(`$fj`[31:0], FCSR.Enables.I, FCSR.RM)

`ftint.l.d` : `$fd` = **FP64convertToSint64**(`$fj`, FCSR.Enables.I, FCSR.RM)

- `FTINT{W/L}.{S/D}` instruction selects the `single`/`double` floating-point number in the floating-point register `fj` to be converted into an `integer`/`long-integer` fixed-point number, and the obtained `integer`/`long-integer` fixed-point number is written To the floating-point memory `fd`. According to the different states in `FCSR`, the operations in the `IEEE 754-2008 standard` followed by this floating-point `format conversion` operation are shown in the following table.

rounding mode	Operations in IEEE 754-2008 Standard
Round to the nearest even number	`convertToIntegerExactTiesToEven`(x)
Round to zero	`convertToIntegerExactTowardZero`(x)
Round towards positive infinity	`convertToIntegerExactTowardPositive`(x)
Round towards negative infinity	`convertToIntegerExactTowardNegative`(x)

- **FP32convertToSint32** :

```
{bits(32) } FP32convertToSint32(bits(32) x, bits(2) rm):
 case {rm} of:
  {2'd0}: return Sint32_convertToIntegerExactTiesToEven(x)
  {2'd1}: return Sint32_convertToIntegerExactTowardZero(x)
  {2'd2}: return Sint32_convertToIntegerExactTowardPositive(x)
  {2'd3}: return Sint32_convertToIntegerExactTowardNegative(x)
```

- **FP64convertToSint32** :

```
{bits(64) } FP32convertToSint64(bits(32) x, bits(2) rm):
 case {rm} of:
  {2'd0}: return Sint32_convertToIntegerExactTiesToEven(x)
  {2'd1}: return Sint32_convertToIntegerExactTowardZero(x)
  {2'd2}: return Sint32_convertToIntegerExactTowardPositive(x)
  {2'd3}: return Sint32_convertToIntegerExactTowardNegative(x)
```

- **FP32convertToSint64** :

```
{bits(64) } FP64convertToSint32(bits(64) x, bits(2) rm):
 case {rm} of:
  {2'd0}: return Sint64_convertToIntegerExactTiesToEven(x)
  {2'd1}: return Sint64_convertToIntegerExactTowardZero(x)
  {2'd2}: return Sint64_convertToIntegerExactTowardPositive(x)
  {2'd3}: return Sint64_convertToIntegerExactTowardNegative(x)
```

- **FP64convertToSint64** :

```
{bits(64) } FP64convertToSint64(bits(64) x, bits(2) rm):
 case {rm} of:
  {2'd0}: return Sint64_convertToIntegerExactTiesToEven(x)
  {2'd1}: return Sint64_convertToIntegerExactTowardZero(x)
  {2'd2}: return Sint64_convertToIntegerExactTowardPositive(x)
  {2'd3}: return Sint64_convertToIntegerExactTowardNegative(x)
```

Usage :

```
.LC282:
    .word    1083179008        # 4.5
    .align   3
la.local     $r12, .LC282      # $r12 = &(.LC282)
fld.s        $f24, $r12,  0    # $f24 = 4.5
ftint.w.s    $f25, $f24        # $f25 = 4 (0x4)
ffint.s.w    $f26, $f25        # $f26 = 4.0
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.3.2` .

8.1.2.3.3 `FTINT{RM/RP/RZ/RNE}.{W/L}.{S/D}`

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
`ftintrm.w.s`, `ftintrm.w.d`, `ftintrm.l.s`, `ftintrm.l.d`	`$fd`	`$fj`
`ftintrp.w.s`, `ftintrp.w.d`, `ftintrp.l.s`, `ftintrp.l.d`	`$fd`	`$fj`
`ftintrz.w.s`, `ftintrz.w.d`, `ftintrz.l.s`, `ftintrz.l.d`	`$fd`	`$fj`
`ftintrne.w.s`, `ftintrne.w.d`, `ftintrne.l.s`, `ftintrne.l.d`	`$fd`	`$fj`

Description :

`ftintrm.w.s` : `$fd`[31:0] = **FP32convertToSint32**(`$fj`[31:0], FCSR.Enables.I, 3)

`ftintrm.w.d` : `$fd` = **FP64convertToSint32**(`$fj`, FCSR.Enables.I, 3)

`ftintrm.l.s` : `$fd`[31:0] = **FP32convertToSint64**(`$fj`[31:0], FCSR.Enables.I, 3)

`ftintrm.l.d` : `$fd` = **FP64convertToSint64**(`$fj`, FCSR.Enables.I, 3)

- `FTINTRM.{W/L}.{S/D}` instruction selects the `single`/`double` floating-point number in the floating-point register `fj` and converts it to `integer`/`long-integer` fixed point number, and the resulting `integer`/`long-integer` fixed point number is written to the floating-point register `fd`, using the "`round to negative infinity`" mode.

Usage :

```
fld.s        $f24, $r12,  0    # $f24 = 4.6
ftintrm.w.s  $f26, $f24        # $f26 = 4 (0x4)
fld.s        $f24, $r12,  0    # $f24 = -4.6
ftintrm.w.s  $f26, $f24        # $f26 = -5
fld.d        $f24, $r12,  0    # $f24 = 4.6
ftintrm.l.d  $f26, $f24        # $f26 = 4
fld.d        $f25, $r12,  0    # $f25 = -4.6
ftintrm.l.d  $f26, $f25        # $f26 = -5
```

Description :

`ftintrp.w.s` : `$fd`[31:0] = **FP32convertToSint32**(`$fj`[31:0], FCSR.Enables.I, 2)

`ftintrp.w.d` : `$fd` = **FP64convertToSint32**(`$fj`, FCSR.Enables.I, 2)

`ftintrp.l.s` : `$fd`[31:0] = **FP32convertToSint64**(`$fj`[31:0], FCSR.Enables.I, 2)

`ftintrp.l.d` : `$fd` = **FP64convertToSint64**(`$fj`, FCSR.Enables.I, 2)

- `FTINTRP.{W/L}.{S/D}` instruction selects the `single`/`double` floating-point number in the floating-point register `fj`, converts it to `integer`/`long-integer` fixed point number, and writes the `integer` /`long-integer` fixed point number into the floating-point register `fd`, using the "`rounding to positive infinity`" method.

Usage :

```
fld.s        $f24, $r12,  0    # $f24 = 4.6
ftintrp.w.s  $f26, $f24        # $f26 = 5
fld.s        $f25, $r12,  0    # $f25 = -4.6
ftintrp.w.s  $f26, $f25        # $f26 = -4
fld.d        $f25, $r12,  0    # $f25 = -4.6
ftintrp.l.d  $f26, $f25        # $f26 = -4
```

Description :

`ftintrz.w.s` : `$fd`[31:0] = **FP32convertToSint32**(`$fj`[31:0], FCSR.Enables.I, 1)

`ftintrz.w.d` : `$fd` = **FP64convertToSint32**(`$fj`, FCSR.Enables.I, 1)

`ftintrz.l.s` : `$fd`[31:0] = **FP32convertToSint64**(`$fj`[31:0], FCSR.Enables.I, 1)

`ftintrz.l.d` : `$fd` = **FP64convertToSint64**(`$fj`, FCSR.Enables.I, 1)

- `FTINTRZ.{W/L}.{S/D}` instruction selects the `single`/`double` floating-point number in floating-point register `fj`, converts it to `integer`/`long-integer` fixed-point number, and writes the obtained `integer`/`long-integer` fixed-point number to floating-point register `fd`, using the "`rounding to zero`" method.

Usage :

```
fld.s        $f24, $r12,  0    # $f24 = 4.6
ftintrz.w.s  $f26, $f24        # $f26 = 4
fld.s        $f24, $r12,  0    # $f24 = -4.6
ftintrz.w.s  $f26, $f24        # $f26 = -4
```

Description :

`ftintrne.w.s` : `$fd`[31:0] = **FP32convertToSint32**(`$fj`[31:0], FCSR.Enables.I, 0)

`ftintrne.w.d` : `$fd` = **FP64convertToSint32**(`$fj`, FCSR.Enables.I, 0)

`ftintrne.l.s` : `$fd`[31:0] = **FP32convertToSint64**(`$fj`[31:0], FCSR.Enables.I, 0)

`ftintrne.l.d` : `$fd` = **FP64convertToSint64**(`$fj`, FCSR.Enables.I, 0)

- `FTINTRNE.{W/L}{S/D}` instruction selects the `single`/`double` floating-point number in floating-point register `fj`, converts it to `integer`/`long-integer` fixed point number, and writes the obtained `integer`/`long-integer` fixed point number to floating-point register `fd`, using the "`rounding to the nearest even number`" method.

Usage :

```
fld.s         $f24, $r12,  0    # $f24 = 4.6
ftintrne.w.s  $f26, $f24        # $f26 = 5
fld.s         $f24, $r12,  0    # $f24 = -4.6
ftintrne.w.s  $f26, $f24        # $f26 = -5
fld.d         $f25, $r12,  0    # $f25 = -4.6
ftintrne.l.d  $f26, $f25        # $f26 = -5
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.3.3` .

8.1.2.3.4 `FRINT.{S/D}`

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
`frint.s`, `frint.d`	`$fd`	`$fj`

Description :

`frint.s` : `$fd`[31:0] = **FP32_roundToInteger**(`$fj`, FCSR.Enables.I, FCSR.RM)

`frint.d` : `$fd` = **FP64_roundToInteger**(`$fj`, FCSR.Enables.I, FCSR.RM)

- The operations in `IEEE 754-2008 standard` for floating-point format conversion operations are shown in the table below..

rounding mode	`Operations in IEEE 754-2008 Standard`
Round to the nearest even number	`roundToIntegralExact`(x)
Round to zero	
Round towards positive infinity	
Round towards negative infinity	

- **FP32_roundToInteger** :

```
{bits(32) } FP32_roundToInteger(bits(N) x):
  return FP32_roundToIntegralExact(x)
```

- **FP64_roundToInteger** :

```
{bits(64) } FP64_roundToInteger(bits(N) x):
  return FP64_roundToIntegralExact(x)
```

Usage :

```
fld.s        $f24, $r12,  0    # $f24 = 4.5
frint.s      $f26, $f24        # $f26 = 4.0
fld.d        $f24, $r12,  0    # $f24 = 4.6
frint.d      $f26, $f24        # $f26 = 5.0
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.3.4` .

8.1.2.4 Floating-Point Move Instructions

8.1.2.4.1 `FMOV.{S/D}`

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
`fmov.d`, `fmov.s`	`$fd`	`$fj`

Description :

`fmov.s` : `$fd`[31:0] = `$fj`[31:0]

`fmov.d` : `$fd` = `$fj`

- `FMOV{S/D}` Write the value of floating-point register `fj` in single/double precision floating-point format to floating-point register `fd`. If the value of `fj` is not in `single`/`double` floating-point format, the result is uncertain.
- The above instruction operations are non arithmetic and do not raise IEEE 754 exceptions, nor do they modify the Cause and Flags fields of the floating-point control status register.

Usage :

```
fld.s     $f24, $r12,  0    # $f24 = 4.5
fmov.s    $f25, $f24        # $f25 = 4.5
fld.d     $f26, $r12,  0    # $f26 = 4.5
fmov.d    $f27, $f26        # $f27 = 4.5
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.4.1` .

8.1.2.4.2 `FSEL`

Syntax:

```
opcode    dest,  src1,  src2,  src3
```

opcode	dest	src1	src2	src3
`fsel`	`$fd`	`$fj`	`$fk`	`$fcc[ca]`

Description :

`fsel` : `$fd` = `$fcc[ca]` ? `$fk` : `$fj`

- The `FSEL` instruction performs conditional assignment operations. When `FSEL` is executed, if the value of the condition flag register ca is equal to 0, the value of floating-point register fj is written to fd; otherwise, the value of floating-point register fk is written to fd.

Usage :

```
fld.d       $f25,  $r12,  0                   # $f25  = 4.5
fld.d       $f26,  $r12,  -8                  # $f26  = -4.5
fcmp.slt.s  $fcc0, $f25,  $f26                # $fcc0 = 0
fsel        $f24,  $f25,  $f26,  $fcc0        # $f25  = 4.5
```

NOTE For more information, refer to the LoongArch instruction manual:3.2.4.2 .

8.1.2.4.3 MOVGR2FR.{W/D}, MOVGR2FRH.W

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
movgr2fr.w, movgr2fr.d, movgr2frh.w	$fd	$rj

Description :

movgr2fr.w : $fd[31:0] = $rj[31:0]

movgr2fr.d : $fd = $rj

movgr2frh.w : $fd[63:32] = $rj[31:0]

Usage :

```
li.w       $r25, 0x76543210           # r25 = 0x76543210
movgr2fr.w $f24, $r25                 # f24 = 0x0000000076543210
li.d       $r25, 0xfedcba9876543210   # r25 = 0xfedcba9876543210
movgr2fr.d $f24, $r25                 # f24 = 0xfedcba9876543210
movgr2frh.2 $f24, $r25                # f24 = 0x7654321076543210
```

NOTE For more information, refer to the LoongArch instruction manual:3.2.4.3 .

8.1.2.4.4 MOVFR2GR.{S/D}, MOVFRH2GR.S

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
`movfr2gr.s`, `movfr2gr.d`, `movfrh2gr.s`	`$rd`	`$fj`

Description :

`movfr2gr.s` : `$rd` = **SignExtend**(`$fj`[31:0], GRLEN)

`movfr2gr.d` : `$rd` = `$fj`

`movfrh2gr.s` : `$rd` = **SignExtend**(`$fj`[63:32], GRLEN)

Usage :

```
fld.d        $f24,  $r12,  0         # $f24 = 0x4028ad72ffd1dcd7
movfr2gr.s   $r25,  $f24             # $f24 = 0xffffffffffd1dcd7
movfr2gr.d   $r25,  $f24             # $f24 = 0x4028ad72ffd1dcd7
movfrh2gr.s  $r25,  $f24             # $f24 = 0x000000004028ad72
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.4.4` .

8.1.2.4.5 `MOVGR2FCSR`, `MOVFCSR2GR`

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
`movgr2fcsr`	`$FSCR`[`fcsr`]	`$rj`
`movfcsr2gr`	`$rd`	`$FSCR`[`fcsr`]

Description :

`movgr2fcsr` : `$FCSR`[`fcsr`] = `$rd`[31:0]

`movfcsr2gr` : `$rd` = **SignExtend**(`$FCSR`[`fcsr`], GRLEN)

Usage :

```
movgr2fcsr   $fcsr0, $r25
movfcsr2gr   $r25, $fcsr0
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.4.5` .

8.1.2.4.6 MOVFR2CF, MOVCF2FR

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
movfr2cf	$FCC[cd]	$fj
movcf2fr	$fd	$FCC[cj]

Description :

movfr2cf : $fcc[cd] = $fj[0]

movcf2fr : $fd[0] = $fcc[cj]

Usage :

```
movfr2cf   $fcc0, $f25
movcf2fr   $f25, $fcc0
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.4.6` .

8.1.2.4.7 MOVGR2CF, MOVCF2GR

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
movgr2cf	$FCC[cd]	$rj
movcf2gr	$rd	$FCC[cj]

Description :

movgr2cf : $fcc[cd] = $rj[0]

movcf2gr : $rd[0] = $fcc[cj]

Usage :

```
movgr2cf   $fcc0, $r25
movcf2gr   $r25, $fcc0
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.4.7` .

8.1.2.5 Floating-Point Branch Instructions

8.1.2.5.1 `BCEQZ`, `BCNEZ`

Syntax:

```
opcode    dest,  src1
```

opcode	dest	src1
`bceqz`, `bcnez`	`cj`	`si23` | `symbol`

Description :

`bceqz` : **if** (`$FCC`[cj] == 0) **jump** (`si23` | `symbol`)

`bcnez` : **if** (`$FCC`[cj] ! = 0) **jump** (`si23` | `symbol`)

- `si23` : a 4-bytes aligned 23-bits signed immediate value in range :
  - [`-4194304`, `4194300`] or [`-0x400000`, `0x3ffffc`]

Usage :

```
bceqz    $fcc0,  (si23 | symbol)
bcnez    $fcc0,  (si23 | symbol)
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.5.1` .

8.1.2.6 Floating-Point Common Memory Access Instructions

8.1.2.6.1 `FLD.{S/D}`, `FST.{S/D}`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`fld.s`, `fld.d`, `fst.d`, `fst.d`	`fd`	`rj`	`si12`

Description :

`fld.s` : `$fd`[31:0] = **MemoryLoad**((`$rj` + **SignExtend**(`si12`, GRLEN)), word)

`fld.d` : `$fd` = **MemoryLoad**((`$rj` + **SignExtend**(`si12`, GRLEN)), doubleword)

`fst.s` : **MemoryStore**(`$fd`[31:0], (`$rj` + **SignExtend**(`si12`, GRLEN)), word)

`fst.d` : **MemoryStore**(`$fd`, (`$rj` + **SignExtend**(`si12`, GRLEN)), doubleword)

- `si12` : 12 bit immediate, Signed value range(`integer`) : [`-2048`, `2047`] or [`-0x800`, `0x7ff`]

Usage :

```
fst.s $f24, $r22, -24
fld.s $f26, $r22, -24
fst.d $f25, $r22, -24
fld.d $f27, $r22, -24
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.6.1` .

8.1.2.6.2 `FLDX.{S/D}`, `FSTX.{S/D}`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`fldx.s`, `fldx.d`, `fstx.d`, `fstx.d`	`fd`	`rj`	`rk`

Description :

`fldx.s` : `$fd`[31:0] = **MemoryLoad**((`$rj` + `$rk`), word)

`fldx.d` : `$fd` = **MemoryLoad**((`$rj` + `$rk`), doubleword) `fstx.s` : **MemoryStore**(`$fd`[31:0], (`$rj` + `$rk`), word)

`fstx.d` : **MemoryStore**(`$fd`, (`$rj` + `$rk`), doubleword)

Usage :

```
fstx.s $f24, $r22, $r23
fldx.s $f26, $r22, $r23
fstx.d $f25, $r22, $r23
fldx.d $f27, $r22, $r23
```

NOTE For more information, refer to the `LoongArch instruction manual:3.2.6.2` .

8.1.2.7 Floating-Point Bound Check Memory Access Instructions

8.1.2.7.1 `FLD{GT/LE}.{S/D}`, `FST{GT/LE}.{S/D}`

Syntax:

```
opcode    dest,  src1,  src2
```

opcode	dest	src1	src2
`fldgt.s`, `fldgt.d`, `fldle.s`, `fldle.d`, `fstgt.s`, `fstgt.d`, `fstle.s`, `fstle.d`	`fd`	`rj`	`rk`

Description :

`fldgt.{s/d}` :

```
if($rj > $rk)
  $fd = SignExtend(MemoryLoad($rj, single/double),GRLEN)
else :
  RaiseException(BCE)
```

`fldle.{s/d}` :

```
if($rj <= $rk)
  $fd = SignExtend(MemoryLoad($rj, single/double),GRLEN)
else :
  RaiseException(BCE)
```

`fstgt.{s/d}` :

```
if $rj > $rk :
  MemoryStore($fd[31:0],$rj,single)/
  MemoryStore($fd[63:0],$rj,double)
else :
  RaiseException(BCE)
```

`fstle.{s/d}` :

```
if $rj <= $rk :
  MemoryStore($fd[31:0],$rj,single)/
  MemoryStore($fd[63:0],$rj,double)
else :
  RaiseException(BCE)
```

Usage :

```
fstle.s  $f24, $r25, $r24
fldle.s  $f25, $r25, $r24
fstle.d  $f26, $r25, $r24
fldle.d  $f27, $r25, $r24
```

NOTE For more information, refer to the `LoongArch instruction manual:2.2.7.1`.

CHAPTER 9. Pseudo Instructions

The LoongArch assembler supports a number of pseudo-instructions that are translated into the appropriate combination of LoongArch instructions at assembly time.

Pseudo Instruction	Machine Instruction	Meaning
jr $rd	jirl $zero, $rd, 0	Direct register jump
ret (jr $ra)	jirl $zero, $ra, 0	Function return
bgt $rj, $rd, (si18 | symbol)	blt $rd, $rj, (si18 | symbol)	if($rj > $rd)jump (si18 | symbol)
bgtu $rj, $rd, (si18 | symbol)	bltu $rd, $rj, (si18 | symbol)	if($rj > $rd)jump (si18 | symbol)
ble $rj, $rd, (si18 | symbol)	bge $rd, $rj, (si18 | symbol)	if($rj < = $rd)jump (si18 | symbol)
bleu $rj, $rd, (si18 | symbol)	bgeu $rd, $rj, (si18 | symbol)	if($rj < = $rd)jump (si18 | symbol)
bltz $rd, (si18 | symbol)	blt $rd, $zero, (si18 | symbol)	if($rd < 0)jump (si18 | symbol)
bgtz $rd, (si18 | symbol)	blt $zero, $rd, (si18 | symbol)	if($rd > 0)jump (si18 | symbol)
blez $rd, (si18 | symbol)	bge $zero, $rd, (si18 | symbol)	if($rd < = 0)jump (si18 | symbol)
bgez $rd, (si18 | symbol)	bge $rd, $zero, (si18 | symbol)	if($rd > = 0)jump (si18 | symbol)
move $rd, $rj	or $rd, $rj, $zero	Assign the value of $rj to $rd

Pseudo Instruction	Machine Instruction	Meaning
li.w $rd, imm32	lu12i.w $rd, si20	Load a 32-bit immediate
	ori $rd, $rd, si12	
li.d $rd, imm64	lu12i.w $rd, si20	Load a 64-bit immediate
	ori $rd, $rd, si12	
	lu32i.w $rd, si20	
	lu52i.w $rd, si12	

Macros Instruction	Feature	Machine Instructions
la $rd, sym	NORMAL	pcalau12i $rd, %got_pc_hi20(sym)
		ld.d $rd, $rd, %got_pc_lo12(sym)
	GTOPCR	pcalau12i $rd, %pc_hi20(sym)
		addi.d $rd, $rd, %pc_lo12(sym)
	GTOABS	lu12i.w $rd, %abs_hi20(sym)
		ori $rd, $rd, %abs_lo12(sym)
		lu32i.d $rd, %abs64_lo20(sym)
		lu52i.d $rd, $rd, %abs64_hi12(sym)

Macros Instruction	Feature	Machine Instructions
`la.global $rd, sym_global`	`NORMAL`	`pcalau12i $rd, %got_pc_hi20(sym_global)`
		`ld.d $rd, $rd, %got_pc_lo12(sym_global)`
	`GTOPCR`	`pcalau12i $rd, %pc_hi20(sym_global)`
		`addi.d $rd, $rd, %pc_lo12(sym_global)`
	`GTOABS`	`lu12i.w $rd, %abs_hi20(sym_global)`
		`ori $rd, $rd, %abs_lo12(sym_global)`
		`lu32i.d $rd, %abs64_lo20(sym_global)`
		`lu52i.d $rd, $rd, %abs64_hi12(sym_global)`

Macros Instruction	Feature	Machine Instructions
`la.global $rd, $rj, sym_global_large`	`NORMAL`	`pcalau12i $rd, %got_pc_hi20(sym_global_large)`
		`addi.d $rj, $zero, %got_pc_lo12(sym_global_large)`
		`lu32i.d $rj, %got64_pc_lo20(sym_global_large)`
		`lu52i.d $rj, $rj, %got64_pc_hi12( sym_global_large)`
		`ldx.d $rd, $rd, $rj`
	`GTOPCR`	`pcalau12i $rd, %pc_hi20(sym_global_large)`
		`addi.d $rj, $zero, %pc_lo12(sym_global_large)`
		`lu32i.d $rj, %pc64_lo20(sym_global_large)`
		`lu52i.d $rj, $rj, %pc64_hi12(sym_global_large)`
		`add.d $rd, $rd, $rj`
	`GTOABS`	`lu12i.w $rd, %abs_hi20(sym_global_large)`
		`ori $rd, $rd, %abs_lo12(sym_global_large)`
		`lu32i.d $rd, %abs64_lo20(sym_global_large)`
		`lu52i.d $rd, $rd, %abs64_hi12(sym_global_large)`

Feature :

- `GTOPCR` : `la-global-with-pcrel`
- `GTOABS` : `la-global-with-abs`

Macros Instruction	Feature	Machine Instructions
la.local $rd, sym_local	NORMAL	pcalau12i $rd, %pc_hi20(sym_local)
		addi.d $rd, $rd, %pc_lo12(sym_local)
	LTOABS	lu12i.w $rd, %abs_hi20(sym_local)
		ori $rd, $rd, %abs_lo12(sym_local)
		lu32i.d $rd, %abs64_lo20(sym_local)
		lu52i.d $rd, $rd, %abs64_hi12(sym_local)

Macros Instruction	Feature	Machine Instructions
la.local $rd, $rj, sym_local_large	NORMAL	pcalau12i $rd, %pc_hi20(sym_local)
		addi.d $rj, $zero, %pc_lo12(sym_local)
		lu32i.d $rj, %pc64_lo20(sym_local)
		lu52i.d $rj, $rj, %pc64_hi12(sym_local)
		add.d $rd, $rd, $rj
	LTOABS	lu12i.w $rd, %abs_hi20(sym_local)
		ori $rd, $rd, %abs_lo12(sym_local)
		lu32i.d $rd, %abs64_lo20(sym_local)
		lu52i.d $rd, $rd, %abs64_hi12(sym_local)

Feature :

- LTOABS : la-local-with-abs

Macros Instruction	Machine Instructions
la.abs $rd, sym_abs	lu12i.w $rd, %abs_hi20(sym_abs)
	ori $rj, $zero, %abs_lo12(sym_abs)
	lu32i.d $rj, %abs64_lo20(sym_abs)
	lu52i.d $rj, $rj, %abs64_hi12(sym_abs)

Macros Instruction	Machine Instructions
la.pcrel $rd, sym_pcrel	pcalau12i $rd, %pc_hi20(sym_pcrel)
	addi.d $rd, $rd, %pc_lo12(sym_pcrel)
la.pcrel $rd, $rj, sym_pcrel_large	pcalau12i $rd, %pc_hi20(sym_pcrel_large)
	addi.d $rj, $zero, %pc_lo12(sym_pcrel_large)
	lu32i.d $rj, %pc64_lo20(sym_pcrel_large)
	lu52i.d $rj, $rj, %pc64_hi12(sym_pcrel_large)
	add.d $rd, $rd, $rj

Macros Instruction	Machine Instructions
la.got $rd, sym_got	pcalau12i $rd, %got_pc_hi20(sym_got)
	ld.d $rd, $rd, %got_pc_lo12(sym_got)
la.got $rd, $rj, sym_got_large	pcalau12i $rd, %got_pc_hi20(sym_got_large)
	addi.d $rj, $zero, %got_pc_lo12(sym_got_large)
	lu32i.d $rj, %got64_pc_lo20(sym_got_large)
	lu52i.d $rj, $rj, %got64_pc_hi12(sym_got_large)
	ldx.d $rd, $rd, $rj

Macros Instruction	Machine Instructions
la.tls.le $rd, sym_le	lu12i.w $rd, %le_hi20(sym_le)
	ori $rd, $rd, %le_lo12(sym_le)
la.tls.ie $rd, sym_ie	pcalau12i $rd, %ie_pc_hi20(sym_ie)
	ld.d $rd, $rd, %ie_pc_lo12(sym_ie)
la.tls.ie $rd, $rj, sym_ie_large	pcalau12i $rd, %ie_pc_hi20(sym_ie_large)
	addi.d $rj, $zero, %ie_pc_lo12(sym_ie_large)
	lu32i.d $rj, %ie64_pc_lo20(sym_ie_large)
	lu52i.d $rj, $rj, %ie64_pc_hi12(sym_ie_large)
	ldx.d $rd, $rd, $rj

Macros Instruction	Machine Instructions
la.tls.ld $rd, sym_ld	pcalau12i $rd, %ld_pc_hi20(sym_ld)
	ld.d $rd, $rd, %got_pc_lo12(sym_ld)
la.tls.ld $rd, $rj, sym_ld_large	pcalau12i $rd, %ld_pc_hi20(sym_ld_large)
	addi.d $rj, $zero, %got_pc_lo12(sym_ld_large)
	lu32i.d $rj, %got64_pc_lo20(sym_ld_large)
	lu52i.d $rj, $rj, %got64_pc_hi12(sym_ld_large)
	ldx.d $rd, $rd, $rj

Macros Instruction	Machine Instructions
la.tls.gd $rd, sym_gd	pcalau12i $rd, %gd_pc_hi20(sym_gd)
	ld.d $rd, $rd, %got_pc_lo12(sym_gd)

Macros Instruction	Machine Instructions
`la.tls.gd $rd, $rj, sym_gd_large`	`pcalau12i $rd, %gd_pc_hi20(sym_gd_large)`
	`addi.d $rj, $zero, %got_pc_lo12(sym_gd_large)`
	`lu32i.d $rj, %got64_pc_lo20(sym_gd_large)`
	`lu52i.d $rj, $rj, %got64_pc_hi12(sym_gd_large)`
	`ldx.d $rd, $rd, $rj`

Operand	ELF reloc type	Usage
`%abs_hi20`	`R_LARCH_ABS_HI20`	[31 … 12] bits of 32/64-bit absolute address
`%abs_lo12`	`R_LARCH_ABS_LO12`	[11 … 0] bits of 32/64-bit absolute address
`%abs64_lo20`	`R_LARCH_ABS64_LO20`	[51 … 32] bits of 64-bit absolute address
`%abs64_hi12`	`R_LARCH_ABS64_HI12`	[63 … 52] bits of 64-bit absolute address
`%pc_hi20`	`R_LARCH_PCALA_HI20`	[31 … 12] bits of 32/64-bit PC-relative offset
`%pc_lo12`	`R_LARCH_PCALA_LO12`	[11 … 0] bits of 32/64-bit PC-relative offset
`%pc64_lo20`	`R_LARCH_PCALA64_LO20`	[51 … 32] bits of 64-bit PC-relative offset
`%pc64_hi12`	`R_LARCH_PCALA64_HI12`	[63 … 52] bits of 64-bit PC-relative offset
`%got_pc_hi20`	`R_LARCH_GOT_PC_HI20`	[31 … 12] bits of 32/64-bit PC-relative offset to GOT entry
`%got_pc_lo12`	`R_LARCH_GOT_PC_LO12`	[11 … 0] bits of 32/64-bit PC-relative offset to GOT entry
`%got64_pc_lo20`	`R_LARCH_GOT64_PC_LO20`	[51 … 32] bits of 64-bit PC-relative offset to GOT entry
`%got64_pc_hi12`	`R_LARCH_GOT64_PC_HI12`	[63 … 52] bits of 64-bit PC-relative offset to GOT entry
`%le_hi20`	`R_LARCH_TLS_LE_HI20`	[31 … 12] bits of 32/64-bit offset from TP register
`%le_lo12`	`R_LARCH_TLS_LE_LO12`	[11 … 0] bits of 32/64-bit offset from TP register
`%ie_pc_hi20`	`R_LARCH_TLS_IE_PC_HI20`	[31 … 12] bits of 32/64-bit PC-relative offset to TLS IE GOT entry
`%ie_pc_lo12`	`R_LARCH_TLS_IE_PC_LO12`	[11 … 0] bits of 32/64-bit PC-relative offset to TLS IE GOT entry
`%ie64_pc_lo20`	`R_LARCH_TLS_IE64_PC_LO20`	[51 … 32] bits of 64-bit PC-relative offset to TLS IE GOT entry
`%ie64_pc_hi12`	`R_LARCH_TLS_IE64_PC_HI12`	[63 … 52] bits of 64-bit PC-relative offset to TLS IE GOT entry
`%ld_pc_hi20`	`R_LARCH_TLS_LD_PC_HI20`	[51 … 32] bits of 64-bit PC-relative offset to TLS LD GOT entry
`%gd_pc_hi20`	`R_LARCH_TLS_GD_PC_HI20`	[63 … 52] bits of 64-bit PC-relative offset to TLS GD GOT entry