Low Level Virtual Machine Intermediate Representation

数据表示：寄存器中的数据

局部变量：`%local_variable = add i32 1,2` 直接使用寄存器
必须%开头　　不可变变量且无需操作地址时

栈上的数据

汇编 {called-saved reg /可用寄存器, calling-saved reg /保留寄存器, 栈.}　LLVM：虚拟寄存器.

数据区里的数据

- 栈上变量：`%local_variable = alloc i32`

全局变量：`@global_variable = (private/interal) global i32 0.`
必须以@开头　　or constant 类型　初始化值

全局变量和栈上变量皆为指针：指向它们所处内存区域（reg 直接存值）

- 操作全局变量和栈上的变量：

```
%1 = load i32 , i32* @global_variable.
```
i32* 类型的指针

```
%2 = add i32 1, %1

store i32 1, i32* @global_variable.
```
类型 值

SSA(Static Single Assignment)：每个变量只被赋值一次

- 虚拟寄存器只存储不可变的变量.
  可变变量放到全局变量或栈内变量：

```
%stack_variable = alloc i32   指定存到栈空间
%1 = add i32 1,2
store i32 %1, i32* %stack_variable
%2 = add i32 3,4
store i32 %2, i32* %stack_variable
```

数据类型：空类型 (void)
整型 (iN) i1、i8、i16、i32、i64
浮点型 (float、double 等).

i1：ture/false
```
%boolean-variable = alloc i1
store i1 ture, i1* %boolean_variable
```

udiv：无符号整型除　sdiv：有符号整型除.
```
%1 = udiv i8 -6, 2   (256-6)/2 = 125
%2 = sdiv i8 -6, 2   (-6)/2 = -3
```

长整型转换成短整型：trunc ... to：
```
%trunc_integer = trunc i32 257 to i8
```
截去高位

短整型转换成长整型：
零扩展：zext ... to
```
%zext_integer = zext i8 -1 to i32   (255)
```
直接在高位补零

符号扩展：sext ... to
```
%sext_integer = sext i8 -1 to i32   (-1)
```
符号位不变，高位补零

浮点数与整型：fptoui ... to
fptosi ... to
uitofp ... to
sitofp ... to

指针与整型：ptrtoint ... to　　`ptrtoint i32* %x to i64`
inttoptr ... to　　`inttoptr i64 %addr to i32*`

数组类型：
```
%a = alloc [4 x i32]
@global_array = global [4 x i32] [i32 0, i32 1, i32 2, i32 3]
@global_string = global [12 x i8] c"Hello_world\00"
```

结构体：`%MyStruct = type {i32, i8}`　⇒
```
struct Mystruct {
    int x;
    char y;
}
```

```
@global_structure = global %MyStruct {i32 1, i8 0}
@global_structure = global {i32, i8} {i32 1, i8 0}
```

```
%MyStruct = type { i32, i32}
%my_structs = alloc [4 x %MyStruct]
%my_struct_ptr = getelementptr [4 x %MyStruct], [4 x %MyStruct]* %my_structs, i64 0, i64 0
```
类型 `[4 x %MyStruct]`；该类型的指针 `[4 x %MyStruct]*`；偏移值 `i64 0`；第n个 `i64 0`

```
%1 = getelementptr %MyStruct, %MyStruct* %my_struct_ptr, i64 2, i32 1
%2 = load i32, i32* %1
```

```
%1 = getelementptr [4 x %MyStruct], [4 x %MyStruct]* %my_structs, i64 0, i64 2, i32 1
```

联级调用：
```
%MyStruct = type {i32, [5 x i32]}
%my_structs = alloc [4 x %MyStruct]
%1 = getelementptr [4 x %MyStruct], [4 x %MyStruct]* %my_structs, i64 2, i32 1, i64 3
```

extracvalue、insertvalue.
```
%2 = extracvalue %MyStruct %1, 1
```
寄存器 第二个字段 (0 base)

```
%3 = insertvalue %MyStruct %1, i32 233, 1
```

控制语句：
标签　`start:` 以“:”结尾
无条件跳转　`br lable %start`
比较大小指令　`%comparison_result = icmp uge i32 %a, %b`　(a>=b)
比较方案：eq/ne、ugt/uge/ult/ule、sgt/sge/slt/sle（= != > >= < <= > >= < <=）
条件跳转.　`br i1 %comparison_result, label %A, label %B`　go to (x? A:B)

Basic block：
- 一个函数由若干基本块构成
- 每个基本块包含：
  开头标签(可省略)
  一系列指令
  ☆结尾是终结指令（改变执行顺序的指令：跳转、返回）
- 一个基本块没有标签时，会自动赋给它一个标签.

Switch：
```
switch i32 %x, lable %C [
       i32 0, lable %A
       i32 1, lable %B
]
A:
       ...
       br lable %end
B:
       ...
       br lable %end
C:
       ...
       br lable %end
end:
       ...
```
⇔
```
switch (x) {
    case 0: A
    case 1: B
    default: C
}
```

Select：
```
%result = icmp sgt i32 %x, 0
%y = select i1 %result, i32 1, i32 2
```
⇔ y = (x>0 ? 1:2) ⇔
```
int y;
if (x>0) y=1;
else y=2;
```
⇕

Phi：
```
    %result = icmp sgt i32 %x, 0
    br i1 %result, lable %bture, lable bfalse
bture:
    br lable %end
bfalse:
    br lable %end
end:
    %y = phi i32 [1, %bture], [2, %bfalse]
```
类型

函数：
定义：
```
define i32 @foo (i32 %a, i64 %b) {
    ret i32 0
}
```
返回值 `i32`；@开头函数名 `@foo`；参数列表 `(i32 %a, i64 %b)`；函数体

声明：`declare i32 @printf (i8*, ...) #1`

调用：
```
define i32 @foo (i32 %a) {
    ...
}
define void @bar ( ) {
    %1 = call i32 @foo (i32 1)
}
```

传递参数与返回值：
遵循C调用约定：参数按顺序放入指定寄存器，若寄存器不够，剩余从右往左顺序压栈.
返回值按先后顺序放入寄存器或放入调用者分配的空间中；只有一个返回值就放在 rex 里.