`LAB5` 为两个主体部分：**指令选择** 和 **寄存器分配**

一、寄存器使用

```
#include "register_rules.h"
```

- 寄存器类别
  - w0-w31

    在llvm中，一个i32可以使用一个32位寄存器存储
  - x0-x31

    地址类型的数据使用64位寄存器存储。

    对地址进行计算时，所有操作数都要使用64位寄存器

    当地址作为函数参数时，传递参数和读取参数时都需要使用64位寄存器

    当对存储栈指针、帧指针、链接寄存器时，都需要使用64位寄存器
- x30、x29，caller-save函数调用前全部保存
- 保留16-19用作spill时加载和保存临时变量

> spill是当物理寄存器不够使用时，将这个虚拟寄存器放在栈中，对应use、def更改为ldr、str。

```
#ifndef REGISTER_RULRES
#define REGISTER_RULRES

#include <set>
#include <array>
const std::set<int> allocateRegs{9, 10, 11, 12, 13, 14, 15};//caller-save
const int XXna = 28;
const int XXnb = 27;
const std::array<int, 8> paramRegs = {0, 1, 2, 3, 4, 5, 6, 7};
```

```
const int XXn1 = 16;
const int XXn2 = 17;
const int XXn3 = 18;
const int XXn4 = 19;
const int FPR = 29;
const int RETR = 0;
const int LNR = 30;
const int INT_LENGTH = 4;
#endif
```

寄存器抽象

```
enum AS_type{
    IMM,//立即数 #10
    SP,//栈指针 sp
    Xn,//x0-x30
    Wn,//w0-w30
    ADR//实现基址寄存器模式、偏移量寻址模式  [x29] [sp,#-30] [x29,x17]
};
struct AS_reg {
    AS_type type;
    union
    {
        int offset;
        struct AS_address *add;
    } u;
};
struct AS_address
{
    AS_reg *base;//<SP|Xn>
    int imm;
    AS_reg *reg;//最多只有一个起作用
};
```

二、指令翻译

从 LLVM 转换成 ARM 的指令选择（Instruction Selection，简称 ISel）是一个将 LLVM IR（中间表示）转换为目标架构（如 ARM）汇编代码的关键步骤。这个过程是编译器后端的核心部分，负责将抽象的 IR 转换成特定平台上可以执行的机器指令。LLVM 中的指令选择模块负责根据目标架构的指令集选择适当的机器指令。

```
#include "llvm2asm.h"
#include "llvm2asm.cpp"
```

可选择指令

```
#include "asm_arm.h"
#include "asm_arm.cpp"
```

```
enum class AS_stmkind {
    BINOP,//二元运算
    MOV,
    MOVZ;//movz    xn, imm, LSL #16 在这里专门设置大数字的高16位
    MOVK;//movk    xn, imm, LSL #0 在这里专门设置大数字的低16位
    LDR;//一般情况下用于加载内存的变量，特殊情况下从堆栈弹出sp并调整sp
    LDP;//从堆栈中弹出X0和X1： LDP X0, X1, [SP], #16
    STR;//一般情况下用于存储寄存器，特殊情况下将一个寄存器压入堆栈并调整sp
    STP;//设计用来存储寄存器 将X0和X1压入堆栈： STP X0, X1, [SP, #-16]!
    LABEL,
    B,
    BCOND,
    BL,
    CMP,
    RET,
    ADR,
    LLVMIR//注释，用来debug
};
```

寻址模式详解

可以查看`printASM.cpp`来理解其效果

```
#include "printASM.h"
#include "printASM.cpp"
```

```
struct AS_ldr {
    AS_reg *dst;
    AS_reg *ptr;
    int post_index;//后索引（Post-index）寻址模式：使用后索引寻址，从基指针中的地址加载值，然后更新指针。
};
struct AS_ldp {
    AS_reg *dst1;
    AS_reg *dst2;
    AS_reg *ptr;
    int post_index;//后索引（Post-index）寻址模式：使用后索引寻址，从基指针中的地址加载值，然后更新指针。
};
struct AS_str {
    AS_reg *src;
    AS_reg *ptr;
    int pre_index;//预索引（Pre-index）寻址模式：使用预索引寻址，先更新基指针，之后从更新后的基指针的地址加载值。
```

```
};
struct AS_stp {
    AS_reg *src1;
    AS_reg *src2;
    AS_reg *ptr;
    int pre_index;//预索引（Pre-index）寻址模式：使用预索引寻址，先更新基指针，之后从更新后的基指针的地址加载值。
};

```

流程：

优化流程介绍提供了一个从函数翻译到寄存器分配的详细流程，其中涵盖了栈操作、虚拟寄存器使用、SSA保持、指令翻译等方面。下面对每个步骤进行更详细的说明。

1. 计算结构体占据空间和每个变量的偏移量

- **目标**：确定程序中结构体占用的内存空间，以及结构体内每个变量的偏移量。
- **操作**：通过分析结构体的成员变量，计算出每个成员变量相对于结构体起始位置的偏移量。该步骤非常重要，因为它决定了结构体成员在栈上的存储位置。

2. 声明函数

- **目标**：声明函数。
- **操作**：此步骤一般用于函数的原型声明或定义，确保函数有明确的接口和返回类型。

3. 全局变量初始化

- **目标**：初始化全局变量。
- **操作**：根据程序需求初始化全局变量，可以是在数据段中为全局变量分配空间，并赋予初始值。

4. 函数翻译

- 这是整个流程的核心部分，涉及从源代码到汇编代码的转换。包括加载参数、调整栈指针、翻译函数体等步骤。

4.1 将函数参数加载到虚拟寄存器

- **当函数参数 `<=8`**：直接从寄存器 `x0` 到 `x7` 获取。这是因为在 ARM 架构中，最多 8 个函数参数通过寄存器传递，寄存器 `x0` 到 `x7` 用来传递这些参数。
- **当函数参数 `>8`**：其余的参数通过栈传递。超过 8 个的参数存储在栈中，因此需要相应调整栈指针并加载参数。

4.2 调整栈指针 `sp`

- **计算局部变量占据的空间**：局部变量需要在栈上分配空间，这会影响栈指针的位置。调整栈指针时要确保所有的局部变量空间都已分配。
- **调整栈指针的时机**：通常在函数调用时，栈指针会被调整为栈帧的起始位置，并且栈指针要在退出时进行恢复。
- **相对于栈帧指针的偏移量**：局部变量的存储地址是相对于栈帧指针（`fp`，即 `x29`）计算的。这个偏移量需要在编译时确定并记录。

4.3 翻译函数体

- **虚拟寄存器使用**：在生成中间表示时，通常会使用虚拟寄存器来表示变量。在翻译时，如果需要更多寄存器，会通过 `new AS_reg(AS_type::Xn, Temp_newtemp_int()->num)` 来获取新的虚拟寄存器。
- **保持 SSA（静态单赋值形式）**：SSA 是一种将每个变量赋值限制在一个地方的编译技术。在翻译过程中，需要保持 SSA 的结构。若需要多个赋值，则通过新的虚拟寄存器实现。

4.4 指令翻译

4.4.1 getelementptr 指令翻译

- `getelementptr` 是一种在指针基础上计算偏移量的指令。在 ARM 汇编中，需要将其翻译为指针和偏移量的计算，可能涉及到地址计算和寄存器加载。

4.4.2 Call | Ret

- **Call**：调用其他函数时，需要保存当前寄存器状态并调整栈指针。通过 `bl` 指令调用函数，并且栈上需要为函数参数分配空间。
- **Ret**：返回时，需要恢复栈指针并释放当前函数的局部变量空间，返回指令是 `ret`。

4.4.3 phi 指令翻译

- **Phi 指令**通常出现在 SSA 中，用于合并不同控制流路径的变量值。为了保持 SSA 形式，在 ARM 汇编中可以将其转换为 `move` 指令，放在前驱 block 的最后，确保变量值的正确性。
  - 将 `phi` 指令改为 `move` 是为了保持变量赋值的清晰性和避免复杂的控制流依赖。
- **为什么将 `phi` 改为 `move` 指令**：
  - 在前驱块的最后插入 `move` 指令可以保证程序流的正确性。虽然这样做会增加一些冗余的定义，但它并不改变活跃分析的结果，因为 `move` 只是一个数据传输操作，不会影响其他的活跃变量分析。

4.4.4 Br 指令翻译

- **Br** 指令（分支）与条件判断一起翻译，通常和 `cmp`（比较）指令联合使用。在 ARM 汇编中，可以通过 `cmp` 指令进行条件比较，并使用 `b.ge`（条件跳转）或其他分支指令来跳转到不同的目标标签。
- **代码示例**：

```
//  %r112 = icmp sge i32 %r266, 48
mov     x11, #48
cmp     x12, x11
//  br i1 %r112, label %bb12, label %bb11
b.ge     bb12
b     bb11
```

在这里，`cmp` 比较寄存器 `x12` 和常数 48 的值，如果大于等于 48，跳转到 `bb12`，否则跳转到 `bb11`。

函数调用约定

函数调用约定定义了在函数调用过程中如何传递参数、保存返回地址、处理局部变量和保存寄存器的状态。针对 ARM 架构和类似架构，调用约定有时被称为 **AArch64 调用约定**。以下是一个典型的 ARM 64 位架构中的函数调用约定的详细说明：

---

1. 函数调用过程（Caller）

在函数调用过程中，调用者需要做一系列的准备工作来确保函数调用后的恢复。步骤如下：

1.1 保存现场：

- **Caller-save 寄存器入栈**：调用者（Caller）保存需要保持的寄存器状态，尤其是 **Caller-save** 寄存器（例如 `x19–x29`）。这些寄存器在函数调用中可能被修改，因此需要在调用前保存在栈中，防止被函数修改后丢失。
- **作用**：此步骤确保调用的函数不会破坏调用者（Caller）使用的寄存器。

1.2 当前帧指针和 `x30` 入栈：

- **保存当前帧指针**：将当前栈帧的指针（即 `x29`）压栈。`x29` 被用作函数栈帧的基准，指向栈帧的顶部（即局部变量和返回地址所在的位置）。
- **保存返回地址 `x30`**：`x30` 存储着返回地址，它包含跳转到函数返回位置的地址。在函数调用后，`x30` 会被修改，所以需要将其压栈。

1.3 多余的实参入栈：

- 如果函数有超过 8 个参数，额外的参数会通过栈传递。栈上将为这些参数分配空间并传递给被调用函数。

1.4 设置帧指针：

- **更新帧指针**：设置 `x29` 为当前栈顶，即当前函数的栈帧起始位置。这样，所有的局部变量和参数都可以相对于帧指针来访问。

1.5 调用函数（`bl` 到相关函数）：

- 通过 `bl`（branch with link）指令调用函数。此时，`bl` 指令会将返回地址（即 `x30`）保存到寄存器中，并跳转到被调用函数的地址。

1.6 加载帧指针和 `x30`：

- 被调用函数执行完后，控制返回到调用者。在返回前，需要恢复栈帧，恢复 `x29` 和 `x30` 寄存器的值。

1.7 恢复现场：

- 恢复调用者的寄存器状态，恢复之前保存的 **Caller-save** 寄存器和 `x29`、`x30` 寄存器。

1.8 从 `x0` 获取返回值进入虚拟寄存器：

- ARM 函数调用约定规定，返回值通常通过 `x0` 寄存器传递。所以，调用者将从 `x0` 寄存器中获取返回值，并将其传递给虚拟寄存器。

---

2. 函数被调用（Callee）

在函数被调用时，函数内部会进行相应的参数接收、栈调整、返回值处理等操作。步骤如下：

2.1 保存函数参数到虚拟寄存器：

- 被调用函数会将传递给它的参数（通常是通过 `x0-x7` 寄存器传递）保存到虚拟寄存器中。对于更多的参数，可能会通过栈传递。

2.2 调整栈指针 `sp`：

- **分配局部变量空间**：函数的局部变量通常是通过栈来存储的。在函数体内，会调整栈指针来为局部变量分配空间。

2.3 返回过程：

- **返回值移入物理寄存器 `x0`**：如果函数有返回值，则返回值需要存储在寄存器 `x0` 中。这个值是调用者从被调用函数中获取返回值时的方式。
- **根据帧指针调整栈指针到函数调用前的状态**：在函数返回之前，栈指针（`sp`）会根据栈帧指针（`x29`）进行调整，将栈指针恢复到函数调用之前的状态。通过 `mov sp, x29` 实现这一操作。
- **`ret` 指令**：使用 `ret` 指令返回到调用者的代码中。`ret` 指令会跳转到 `x30` 中存储的返回地址。

示例

```
        //  %r257 = call i32 @getch()

        stp     x9, x10, [sp, #-16]!
        stp     x11, x12, [sp, #-16]!
        stp     x13, x14, [sp, #-16]!
        str     x15, [sp, #-8]!
        stp     x29, x30, [sp, #-16]!
        mov     x29, sp
```

```
        bl      getch
        ldp     x29, x30, [sp], #16
        ldr     x15, [sp], #8
        ldp     x13, x14, [sp], #16
        ldp     x11, x12, [sp], #16
        ldp     x9, x10, [sp], #16
        mov     x11, x0
```

`getelementptr` 指令翻译

`getelementptr` 指令用于计算指针偏移，它通常用于获取结构体、数组或其他数据类型中元素的地址。由于 AArch64 中的数据都是无类型的，因此 `getelementptr` 主要涉及对寄存器的加减乘除操作来计算偏移地址。

下面是如何将 `getelementptr` 指令翻译成 AArch64 汇编的步骤，具体包括处理局部变量、全局变量、函数参数和中间变量。

1. 局部变量的 `getelementptr` 翻译

局部变量通常位于栈上，其地址是相对于当前帧指针（`fp` 或 `x29`）计算的。过程如下：

- **计算偏移地址**：
  1. 首先，根据 `fpOffset`（局部变量的偏移量）计算出局部变量的首地址，即 `base_ptr`。这个偏移量是相对于帧指针 `x29` 的。
  2. 然后，根据类型的大小（例如，对于一个整数，大小为 4 字节）计算偏移，得到新的地址 `new_ptr`。
- **代码示例**：

  假设有一个局部变量 `a`，并且它在栈上相对于帧指针有一个偏移 `offset`。

```
// base_ptr = fp + offset
add     x0, x29, offset        // x0 保存计算出的首地址 base_ptr

// new_ptr = base_ptr + element_offset
add     x1, x0, element_offset // x1 保存最终地址 new_ptr
```

在计算过程中，如果需要更多的寄存器来存储中间结果，使用新的虚拟寄存器，虚拟寄存器的编号可以通过 `Temp_newtemp_int()->num` 获取。

2. 全局变量的 `getelementptr` 翻译

全局变量在程序的整个生命周期内都有固定的地址。要获取全局变量的地址，可以使用 `adr` 指令来获得全局变量的地址。

- **计算偏移地址**：
  1. 使用 `adr` 指令来获得全局变量的首地址 `base_ptr`， `adr` 指令会将符号地址加载到寄存器中。
  2. 根据类型的大小，计算偏移，得到 `new_ptr`。
- **代码示例**：

假设全局变量 `global_var` 在程序中的地址为 `global_var_addr`，并且我们要计算全局数组 `arr` 中第 `i` 个元素的地址：

```
// 获取全局变量的首地址 base_ptr
adr     x0, global_var_addr // x0 保存全局变量的首地址

// new_ptr = base_ptr + (i * element_size)
mul     x1, x2, element_size // x2 保存索引 i，element_size 为元素大小
add     x1, x0, x1           // x1 保存最终地址 new_ptr
```

同样，在计算过程中，可能需要新的虚拟寄存器来存储中间结果。

3. 函数参数与中间变量的 `getelementptr` 翻译

函数参数和中间变量在大多数情况下是直接传递给寄存器的，特别是前 8 个参数会通过 `x0-x7` 寄存器传递。如果参数是通过栈传递的，则需要使用栈指针来访问。

- **计算偏移地址**：
  1. 对于函数参数和中间变量，直接将其作为 `base_ptr` 使用，不需要额外的计算。
  2. 通过类型大小计算偏移，得到 `new_ptr`。
- **代码示例**：

  假设函数参数 `param` 是通过 `x0` 寄存器传递的，我们要访问它的某个成员或偏移：

```
// base_ptr = x0  // 假设 param 已经存在于寄存器 x0 中
// new_ptr = base_ptr + offset
add     x1, x0, offset // x1 保存最终地址 new_ptr
```

对于中间变量，其偏移量通常是由编译器在生成过程中计算出来的，直接使用寄存器即可。

4. 保持 SSA 形式

在上述过程中，尤其是在处理指针和偏移时，保持 **SSA（静态单赋值）形式** 是非常重要的。SSA 形式要求每个变量只能被赋值一次，这意味着在翻译过程中，如果需要创建新的寄存器以存储中间计算结果，则必须使用新的虚拟寄存器。

- 新的虚拟寄存器可以通过 `Temp_newtemp_int()->num` 创建。这将确保每个寄存器有唯一的标识符，并且不会引入重复的赋值。

示例

```
        //%r271->x10
        //%r134->x11
        //%r135->x11
//  %r134 = getelementptr [1005 x i32 ], [1005 x i32 ]* @head, i32 0, i32 %r271

        mov     x11, #8
        mul     x12, x10, x11
        adrp     x11, head
        add     x11, x11, #:lo12:head
        add     x11, x11, x12
        //  %r135 = load i32, i32* %r134

        ldr     x11, [x11]
```

BCOND

ARMv8架构中的BCOND指令是一条条件分支指令，用于根据特定条件跳转到目标地址。这在控制程序的执行流时尤其有用，比如在实现循环、条件判断等结构时。

条件分支

BCOND指令会根据指定的条件进行跳转。ARMv8架构提供了一组条件码，可以用来决定分支是否执行。常见的条件码包括：

```
enum class AS_relopkind {
    EQ_,
    NE_,
    LT_,
    GT_,
    LE_,
    GE_,
};
```

三、寄存器分配

寄存器分配是编译器优化过程中的一个关键步骤，它的目标是将程序中的变量和临时计算结果（通常存储在内存中）映射到处理器的有限寄存器中。通过合理的寄存器分配，可以显著提高程序的执行效率，减少内存访问的次数，并最大化处理器的计算能力。

```
#include "allocReg.h"
#include "allocReg.cpp"
```

预分配的物理寄存器

- 编号`<100`
- 如x0-x7,x29,x30

待分配的虚拟寄存器

- 编号`>=100`

流程：

1. 计算实现构建干扰图（ok）
2. **删除没有使用过的变量(ok)**

   > 如果一个变量只定义没有use，那么它就可以任意染色，因此可能ntr别人的寄存器

3. 寄存器分配
   1. 线性扫描算法
   2. 图染色算法
   3. 基于单纯消除序列的方法
4. 有位置的寄存器color
5. **spill** 没有位置的寄存器要修改代码。
   - **再次调整栈指针**。
   - 记录临时变量相对于`sp`的偏移量
   - **对use变量a地方，每次从栈中ldr， def变量a的地方，str到栈中**
   - 需要额外计算，使用保留的物理寄存器

图染色算法

图着色算法是寄存器分配中的核心步骤，它通过将变量映射到有限数量的寄存器上来解决寄存器分配问题。在这个过程中，节点代表变量，边代表变量之间的冲突（即它们在程序中有重叠的活跃区间），而颜色代表寄存器。图着色算法旨在为每个变量分配一个寄存器（颜色），确保两个冲突的变量不会被分配到同一个寄存器。

图着色算法通常由以下几个步骤组成：

1. 活跃分析

活跃分析的目的是分析程序中每个变量的活跃周期，也就是变量从定义到最后使用之间的时间段。通过活跃分析，编译器可以确定哪些变量在某个时间点是活跃的，哪些变量可以存储在寄存器中，哪些变量可能需要溢出到内存中。

2. 构建冲突图

图染色的第一步是构建**冲突图（Interference Graph）**。在冲突图中，每个节点表示一个程序中的变量，每一条边表示两个变量在程序执行时有重叠的活跃区间，因此它们不能被分配到同一个寄存器中。

- **节点**：每个变量。
- **边**：如果两个变量在程序的某个部分有重叠的活跃区间，则这两个变量之间有一条边（表示它们冲突，不能共享一个寄存器）。

对于冲突图的构建，常见的做法是遍历程序中的每一条指令，分析变量的使用情况，若某两个变量在同一时刻都需要被保留在寄存器中（即它们的活跃区间重叠），就将它们之间添加一条边。

3. 简化（Simplify）

简化阶段的目标是将冲突图中的某些节点从图中移除，以便为其他节点腾出空间。简化的基本策略是：反复从图中删除度数小于 `K`（即寄存器数）的节点，并将这些节点添加到栈中。

- **度数**：节点的度数是指与该节点相连的边的数量，表示该变量与多少个其他变量冲突。
- **度数< K**：如果一个节点的度数小于 `K`，则说明它的冲突较少，能够比较容易地分配寄存器。此时可以从图中删除这个节点，并将它放入栈中。

在这一阶段，简化操作不会改变图中的边，只是删除了度数较小的节点，减少了图中的复杂性。简化的目的是通过逐步减少冲突的节点，为后续的寄存器分配过程铺平道路。

4. 潜在溢出（Potential Spill）

在简化阶段之后，图中会残留一些度数大于等于 `K` 的节点，这些节点代表了在当前寄存器数下无法通过简化分配寄存器的变量。对于这些变量，编译器必须做出决策，将它们从图中删除并标记为溢出（spilled）。

- **溢出（Spilling）**：当一个节点的度数大于等于 `K` 时，表示它的活跃区间过长，无法直接存储在寄存器中。此时，可以选择将该变量存储到内存中。此操作称为溢出。

潜在溢出阶段的目的是识别这些“难以分配寄存器”的变量，并为它们做出合适的处理决策。

5. 选择（Select）

在**选择**阶段，编译器从栈中恢复节点，并尝试为每个节点分配寄存器。在恢复的过程中，编译器会检查每个节点的邻居节点（即与当前节点有边的节点，表示它们有冲突）是否已经被染色（已经分配了寄存器）。如果有邻居节点已经分配了寄存器，则不能将当前节点分配到同样的寄存器。

具体步骤如下：

1. 从栈中弹出一个节点。
2. 检查该节点的邻居节点，哪些已经分配了寄存器。
3. 从剩余的寄存器中选择一个未被使用的颜色为该节点染色。
4. 如果无法为该节点分配寄存器，则将其标记为溢出，并将它存储到内存中。

这个过程的核心是根据当前图中的状态（已经染色的节点）来选择一个适当的寄存器。这个过程是贪心的，尽量选择未被占用的寄存器来为变量分配寄存器。

6. 实际溢出（Actual Spill）

如果在选择过程中发现某些节点无法被分配到寄存器（由于冲突或寄存器数量不足），编译器会将这些节点标记为溢出，并将它们存储在内存中。这些节点会在程序的执行过程中通过内存访问来获取值，可能会导致性能下降。

实际溢出步骤包括：

- **内存分配**：为溢出的变量分配内存位置。
- **生成内存加载/存储指令**：在程序的适当位置插入内存加载和存储指令，将溢出的变量从内存加载到寄存器，或者将寄存器中的值存储到内存。

线性扫描算法

线性扫描（Linear Scan）是一种用于寄存器分配的算法，它是一种较为简单和高效的寄存器分配算法，常用于简单的寄存器分配场景。与图染色算法相比，线性扫描不需要构建复杂的干涉图，而是通过对活跃区间的线性扫描来分配寄存器。这使得线性扫描在某些情况下可以更高效地运行，尤其是在寄存器较少且程序较简单的情况下。

线性扫描算法的基本步骤

1. **活跃区间分析**：
   - 与图染色算法一样，线性扫描也需要进行活跃分析，以确定每个变量的活跃区间。活跃区间指的是变量在程序中从被定义到最后使用之间的时间段。
2. **按时间顺序遍历指令**：
   - 程序中的每一条指令都包含一组变量，这些变量在执行该指令时是活跃的。线性扫描算法通过按顺序遍历程序中的指令来分析每个变量的活跃状态。
3. **按活跃区间排序**：
   - 所有变量按照它们的活跃区间的起始时间和结束时间进行排序，通常使用区间的开始时间进行排序。排序后的变量表示它们出现在程序中的时间顺序。
4. **分配寄存器**：
   - 线性扫描通过遍历排序后的活跃区间来分配寄存器。如果一个变量的活跃区间与已经分配寄存器的变量的区间不重叠，且有足够的空闲寄存器，则为该变量分配一个寄存器。
   - 如果发现所有寄存器都已经被使用，而且当前变量的活跃区间与已经分配寄存器的变量的活跃区间重叠，那么需要**溢出**（spill）。溢出的变量会被存储到内存中，并且会生成相应的加载/存储指令。
5. **溢出处理（Spilling）**：
   - 当没有足够的寄存器来分配时，线性扫描会选择一个溢出（spill）策略。通常，选择活跃区间最早结束的变量进行溢出，因为它们较早不再使用。
   - 这些溢出的变量将会被存储到内存中，并且会生成内存加载（load）和存储（store）指令。
6. **寄存器回收**：
   - 在程序的执行过程中，当一个变量的活跃区间结束时，已经分配给该变量的寄存器会被回收，供其他变量使用。

线性扫描的工作流程

1. **活跃区间计算**：
   - 对程序进行遍历，计算每个变量的活跃区间。记录每个变量在程序中的使用情况以及它们的生命周期。
2. **排序**：
   - 按照活跃区间的开始时间对变量进行排序。
3. **寄存器分配**：
   - 在排序后的活跃区间中，从前到后遍历。如果当前变量的活跃区间与已分配寄存器的区间没有重叠，且有空闲寄存器，则分配一个寄存器；否则，需要溢出。
4. **溢出处理**：
   - 如果发现寄存器不足，选择一个合适的变量进行溢出，通常选择结束时间最早的变量溢出。将溢出变量存储到内存，并更新相应的加载/存储指令。
5. **回收寄存器**：
   - 当一个变量的活跃区间结束时，释放该变量所占用的寄存器。

基于单纯消除序列的方法（Simple Elimination Sequence）

基于单纯消除序列（Simple Elimination Sequence，简称 SES）的方法是寄存器分配中的一种技术，它属于一种基于图的寄存器分配算法。该方法通过消除冲突图中的节点，寻找一个合适的顺序来分配寄存器。该方法的灵感源自于图的**拓扑排序**，通过逐步移除干涉图（interference graph）中的节点来找到合适的寄存器分配。

基本概念

- **冲突图（Interference Graph）**：冲突图的每个节点表示一个变量，而图中的每条边表示两个变量在程序中存在冲突（即它们的活跃区间重叠，无法同时分配到同一个寄存器）。
- **消除序列（Elimination Sequence）**：消除序列是指在冲突图中逐步移除节点的顺序，这个顺序应该能够确保移除节点时不会破坏剩余节点的合法性。通过消除节点的顺序，可以决定哪些变量在某一时刻会被分配到寄存器。

单纯消除序列的步骤

1. **构建冲突图**：
   - 首先，分析程序的变量，构建变量间的冲突图。冲突图中的每个节点代表一个变量，两个变量之间有一条边如果它们的活跃区间有交集（即它们不能共用一个寄存器）。
2. **寻找可消除节点**：
   - 在冲突图中，找出可以消除的节点。节点是可消除的当且仅当它的邻居节点的度数小于等于可用寄存器数目（K）。度数小于等于K的节点意味着这些节点的邻居数不超过可用寄存器数目，因此可以在不引发冲突的情况下分配寄存器。
3. **消除节点并更新冲突图**：
   - 一旦找到可消除的节点，将该节点从图中移除，并更新它的邻居节点的度数。因为该节点被移除，原本与该节点相邻的变量之间的冲突关系被解除，从而可能使其他节点变得可消除。
4. **重复消除过程**：
   - 重复第3步，直到所有的节点都被消除。如果图中剩下的节点的度数大于可用寄存器数目（K），则会触发溢出（spill）操作，将某些变量从寄存器中溢出到内存中。
5. **分配寄存器**：
   - 在所有节点都被消除后，剩下的节点按消除顺序依次恢复。每个恢复的节点都尝试分配一个可用的寄存器。如果该节点的邻居已经有寄存器分配，则尝试为其分配剩余的寄存器。
6. **处理溢出**：
   - 如果在分配过程中发现没有足够的寄存器来分配一个节点，且无法通过图的消除顺序来避免溢出，则将溢出的变量存储到内存中。

关键思想

- **消除节点的顺序**：通过选择合适的消除节点顺序，可以确保图中较为“简单”的部分首先被处理，而剩下的部分更加复杂。在算法过程中，通过尽可能地删除度数小于K的节点来减少冲突，从而减少溢出的发生。
- **消除顺序的保证性**：单纯消除序列的核心思想是保证每次删除的节点是合法的，也就是说删除的节点不会导致后续的分配不可能进行。因此，消除的顺序对于整个寄存器分配的正确性非常重要。

四、示例

- https://godbolt.org/一个交互式在线编译器，可以展示编译后的汇编输出，支持多种语言，包括C++、Rust、Go等。

- /example下有BFS的完整实例。

五、调试

使用 `qemu-aarch64` 配合 `gdb` 来调试 64 位 ARM（AArch64）二进制文件。以下是具体步骤：

1. 安装必要的工具

确保你已经安装了 `qemu-aarch64` 和 `gdb-multiarch` 或 `gdb`（支持 AArch64 的 GDB）。

在 Debian/Ubuntu 系统上，可以使用以下命令：

`sh sudo apt-get update sudo apt-get install qemu-user-static gdb-multiarch`

2. 启动 QEMU 并启用 GDB 调试

使用 `qemu-aarch64` 启动你的程序，并启用 GDB 调试模式。

指定 QEMU 在某个端口上等待 GDB 连接。

`sh qemu-aarch64 -g 1234 ./int_split`

这里，`-g 1234` 表示 QEMU 将在端口 `1234` 上等待 GDB 的连接。

3. 启动 GDB 并连接到 QEMU 在另一个终端中

启动 `gdb-multiarch` 或 `gdb`： `sh gdb-multiarch ./int_split`

然后，在 GDB 提示符下连接到 QEMU：

```
target remote localhost:1234
```

其他常用 GDB 命令

设置断点：

`(gdb) break main`

`(gdb)  break bb1`

`(gdb)  break *0x500067bc`

查看寄存器值：

`(gdb) info registers`

`(gdb) print $pc`

`(gdb) print $x10`

`(gdb) p/o $x10`

/x /d /o /t

查看内存内容：

`(gdb) x/10i $pc  // 查看当前 $pc（程序计数器）处的 10 条指令`

`(gdb) x/20x $sp  // 查看当前 $sp（栈指针）处的 20 个字节的内容` -

单步执行指令：

`gdb  (gdb) stepi`

`(gdb) step 10//执行10步源代码语句，并进入函数内部`

`(gdb) next 10//执行10步源代码语句，而不进入函数内部`

继续执行：

`gdb  (gdb) continue`

跳出当前函数：

`gdb  (gdb) finish`

查看当前栈帧:

`(gdb) backtrace`