冒泡排序汇编

源程序

```
// bubble_sort.sy
// 功能：读取一系列整数，使用冒泡排序算法将其排序，然后输出结果。

const int MAX_SIZE = 100; // 定义数组的最大容量
int data_array[MAX_SIZE]; // 用于存储数据的全局数组

/*
 * 函数名：bubble_sort
 * 功能：对数组的前 len 个元素执行冒泡排序（升序）。
 * 参数：arr[] - 待排序的整数数组
 *       len   - 数组中有效元素的个数
 * 返回值：void（无返回值）
 */
void bubble_sort(int arr[], int len) {
    int i = 0;
    int j = 0;
    int temp = 0; // 用于交换元素的临时变量

    // 外层循环控制排序的轮数
    // 每一轮都会将一个最大的元素“冒泡”到末尾
    i = 0;
    while (i < len - 1) {

        // 内层循环负责比较和交换相邻元素
        j = 0;
        while (j < len - 1 - i) {
            if (arr[j] > arr[j + 1]) {
                // 如果前一个元素大于后一个，则交换它们
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
            j = j + 1;
        }

        i = i + 1;
    }
}

/*
 * 函数名：main
```

```
 * 功能: 程序主入口
 * 返回值: 0 表示成功
 */
int main() {
    int n = 0; // 存储实际输入的元素个数

    // 从标准输入读取一串整数到全局数组 data_array 中
    // getarray 会先读取一个整数 N，然后读取 N 个整数到数组
    // 返回值就是读取的元素个数 N
    n = getarray(data_array);

    // 调用排序函数对数组进行排序
    bubble_sort(data_array, n);

    // 输出排序后的数组
    // putarray 的第一个参数是要输出的元素个数
    // 第二个参数是要输出的数组
    putarray(n, data_array);

    return 0;
}
```

路径规划依据

1. 汇编语言没有**变量**的概念，只有**内存地址**和**寄存器。**

   - **高级语言的便利**: 你可以轻松地写 int n;，编译器会帮你处理好一切。
   - **汇编语言的现实**: 汇编根本不知道 n 是什么。它只认识像 32(sp) (栈上某个偏移地址) 或 0x10010000 (某个全局地址) 这样的具体位置，以及像 s0, t1 这样的寄存器。

   ➡️ 这直接催生了我们的【第 1 阶段：高级分析与规划】

   这个阶段的唯一目的，就是建立一个从“高级语言的变量名”到“汇编语言的存储位置”的映射表。

   - **对全局变量 data_array**: 我们规划出它在 .bss 段的标签和大小。这就是它的“地址”。
   - **对局部变量 n, i, j**: 我们设计好栈帧，为它们分配好在栈上的具体“坑位”（如 0(sp), 16(sp)）。这就是它们的“地址”。
   - **对频繁使用的变量**: 我们决定用哪个 s 寄存器来“缓存”它，这就是它的“临时家”。

   此阶段的产出是一个“内存地图”。完成了这一步，我们就把“变量”这个抽象概念彻底从大脑里剥离了，接下来的步骤就可以只思考地址和寄存器了。

2. 汇编语言没有**结构化控制流**，只有**比较**和**跳转。**

   - **高级语言的便利**: 你可以用 while(...) { ... } 和 if (...) { ... } else { ... } 来清晰地组织逻辑。

- **汇编语言的现实**: 汇编只有“线性”的指令流。它能做的只有：
  - 比较两个值 (bge, ble 等)。
  - 如果满足条件，就**无情地跳转**到一个完全不同的代码位置（一个标签）。

➡️ **这直接催生了我们的【第 2 阶段：逻辑到“伪指令”的翻译】**

这个阶段的**唯一目的**，就是将**“结构化的、嵌套的逻辑”**打散，重组成**“扁平化的、带标签和跳转的逻辑流程”**。

- **对于 while 循环**: 我们把它拆解成：
  - 一个 loop_start 标签。
  - 循环条件的比较与**条件跳转**（如果不满足就跳到 loop_end）。
  - 循环体。
  - 一个**无条件跳转**（跳回到 loop_start）。
  - 一个 loop_end 标签。
- **对于 if-else**: 我们把它拆解成：
  - if 条件的比较与**条件跳转**（如果不满足就跳到 else_block）。
  - if 的代码块。
  - 一个**无条件跳转**（跳过 else 代码块，直接到 endif）。
  - 一个 else_block 标签。
  - else 的代码块。
  - 一个 endif 标签。

此阶段的产出是一个“逻辑流程图”或“控制流图”（CFG）。我们用伪指令搭建好了程序的骨架。完成了这一步，我们就把复杂的嵌套逻辑变成了简单的线性步骤，为最后一步的机械翻译铺平了道路。

3. 汇编语言没有**函数**，只有**调用约定**和**指令序列**。

- **高级语言的便利**: 你可以定义函数 bubble_sort(arr, len)，调用时编译器会帮你处理好参数传递、返回地址保存等一切繁琐事务。
- **汇编语言的现实**: 汇编只知道 call 指令会把下一条指令的地址存入 ra 并跳转。它完全不知道：
  - 参数 arr 和 len 应该放哪里？
  - ra 寄存器里的地址会不会被后续的嵌套调用覆盖掉？

- 局部变量 i, j, temp 存哪里?
- 哪些寄存器可以安全使用而不会影响调用者?

➡️ **这也同时被【第 1 阶段】和【第 3 阶段】共同解决**

- **第 1 阶段的栈帧规划**: 我们提前设计好了函数调用的“合同”（即ABI的应用）。我们规定了 ra 必须存入栈中，s 寄存器也必须保存和恢复。这就像在盖房子前就设计好了水管和电路的预留通道。
- **第 3 阶段的最终汇编**: 这是最机械的一步。因为前面两步已经把所有困难的、需要“思考”的问题都解决了（变量在哪？逻辑怎么走？），这一步我们只需要**“查表”**一样，把伪指令翻译成具体的 RISC-V 指令就行了。
  - “加载 n 到 a1” -> ld a1, 0(sp)
  - “跳转到 loop_start” -> j .L_outer_loop_start
  - “i = i + 1” -> addi s2, s2, 1

此阶段的产出就是最终的可执行汇编代码。

第一步：高级分析与规划

在这一步，我们不写任何汇编代码，只做分析和设计，就像编译器在构建符号表和进行高级规划一样。

1.1 全局符号分析

- **常量 MAX_SIZE**: 这是一个编译时常量，值为100。它在最终的汇编代码中不会作为变量存在，我们只需要在心里记住它的值。
- **全局变量 data_array**: 这是一个全局数组，包含100个int。因为它没有初始值，所以它属于 .bss 段（未初始化数据段）。
  - **内存大小**: 100个整数 * 4字节/整数 = 400字节。
  - **规划**: 我们需要在汇编中定义一个标签 data_array，并为其预留400字节的空间。
- **函数**:
  - main: 我们的程序入口。
  - bubble_sort: 我们需要自己实现的排序函数。
  - getarray, putarray: 运行时库函数，我们只需要 call 它们。

1.2 main 函数分析与栈帧规划

- **局部变量**: 只有一个，int n。

- **函数调用**: 调用了 getarray, bubble_sort, putarray。因为它调用了其他函数，我们**必须**在栈上保存返回地址 ra。
- **栈帧设计**:
  - 为 ra 预留空间：8字节。
  - 为局部变量 n 预留空间：4字节 (但为了对齐，我们通常也给它8字节的空间)。
  - 总需求：8 + 8 = 16 字节。这正好是 RISC-V 要求的16字节对齐。
  - **布局**:
    - 8(sp): 用于存放 ra。
    - 0(sp): 用于存放 n。

1.3 bubble_sort 函数分析与栈帧规划

- **参数**: int arr[] (地址) 和 int len (整数)。它们将分别通过 a0 和 a1 寄存器传入。
- **局部变量**: int i, int j, int temp。
- **寄存器使用策略 (关键!)**:
  - 这个函数有嵌套循环，如果每次都从栈里读写 i, j, len 和数组地址，效率会非常低。
  - **最佳策略**: 使用**被调用者保存** (s) 寄存器来“缓存”这些重要的值。
    - s0: 保存 arr 的基地址。
    - s1: 保存 len 的值。
    - s2: 保存外层循环变量 i 的值。
    - s3: 保存内层循环变量 j 的值。
  - 因为我们使用了 s0, s1, s2, s3，所以我们**必须**在函数开头将它们原来的值保存到栈上，在函数结尾恢复它们。
- **栈帧设计**:
  - 为 ra 预留空间：8字节。
  - 为 s0, s1, s2, s3 预留空间：4 * 8 = 32字节。
  - 局部变量 temp 可以直接使用临时寄存器 t，无需栈空间。
  - 总需求：8 + 32 = 40 字节。按16字节对齐，我们需要分配 **48字节**。
  - **布局**:
    - 40(sp): 存放 ra

- 32(sp): 存放 s0
- 24(sp): 存放 s1
- 16(sp): 存放 s2
- 8(sp): 存放 s3

第二步：逻辑到“伪指令”的翻译

现在，我们用更接近自然语言的方式，把 C 代码的逻辑流程写下来。

2.1 main函数伪指令

```
main:
  // 序言: 设置栈帧
  分配 16 字节栈空间
  保存 ra 到 8(sp)

  // n = getarray(data_array)
  加载全局变量 data_array 的地址到 a0
  调用 getarray
  getarray的返回值 (n) 在 a0 中，把它保存到栈上 0(sp)

  // bubble_sort(data_array, n)
  加载全局变量 data_array 的地址到 a0
  从栈 0(sp) 加载 n 到 a1
  调用 bubble_sort

  // putarray(n, data_array)
  从栈 0(sp) 加载 n 到 a0
  加载全局变量 data_array 的地址到 a1
  调用 putarray

  // return 0
  设置返回值 a0 = 0

  // 尾声: 恢复栈帧
  从 8(sp) 恢复 ra
  释放 16 字节栈空间
  返回
```

bubble_sort函数伪指令

```
bubble_sort:
  // 序言: 设置栈帧
  分配 48 字节栈空间
  保存 ra 到 40(sp)
  保存 s0 到 32(sp), s1 到 24(sp), s2 到 16(sp), s3 到 8(sp)

  // 将参数存入 s 寄存器
  移动 a0 (arr地址) 到 s0
  移动 a1 (len) 到 s1

  // i = 0
  设置 s2 = 0

.L_outer_loop_start:
  // while (i < len - 1)
  计算 `len - 1` 到 t0
  如果 s2 >= t0, 跳转到 .L_outer_loop_end

  //   j = 0
  设置 s3 = 0

.L_inner_loop_start:
  //   while (j < len - 1 - i)
  计算 `len - 1` 到 t0
  计算 `t0 - i` 到 t0
  如果 s3 >= t0, 跳转到 .L_inner_loop_end

  //     if (arr[j] > arr[j + 1])
  计算 arr[j] 的地址到 t1
  计算 arr[j+1] 的地址到 t2
  加载 arr[j] 的值到 t3
  加载 arr[j+1] 的值到 t4
  如果 t3 <= t4, 跳转到 .L_skip_swap

  //       交换
  存储 t4 (arr[j+1]的值) 到 arr[j]的地址 (t1)
  存储 t3 (arr[j]的值) 到 arr[j+1]的地址 (t2)

.L_skip_swap:
  //     j = j + 1
  s3 = s3 + 1
  跳转到 .L_inner_loop_start

.L_inner_loop_end:
  //   i = i + 1
  s2 = s2 + 1
```

```
  跳转到 .L_outer_loop_start

.L_outer_loop_end:
  // 尾声: 恢复栈帧
  从栈上恢复 ra, s0, s1, s2, s3
  释放 48 字节栈空间
  返回
```

第三步：最终汇编代码生成

现在，我们把第二步的“伪指令”精确地翻译成 RISC-V 汇编

```
# bubble_sort.s
# 功能: 实现冒泡排序算法

# --- 数据段 ---
.section .bss
.align 3                # 8字节对齐
.globl data_array       # 声明为全局符号
data_array:
    .space 400          # 为100个int分配空间 (100 * 4 bytes)

# --- 代码段 ---
.section .text
.globl main

# --- main 函数 ---
# 功能: 读取数据, 调用排序, 打印结果
main:
    # --- 函数序言 ---
    addi sp, sp, -16    # 分配16字节栈帧
    sd   ra, 8(sp)      # 保存返回地址

    # --- n = getarray(data_array) ---
    lla  a0, data_array # 参数1: 数组地址
    call getarray       # 调用getarray, 返回值 n 在 a0 中
    sd   a0, 0(sp)      # 将 n 保存到栈上

    # --- bubble_sort(data_array, n) ---
    lla  a0, data_array # 参数1: 数组地址
    ld   a1, 0(sp)      # 参数2: 从栈加载 n
    call bubble_sort

    # --- putarray(n, data_array) ---
    ld   a0, 0(sp)      # 参数1: 从栈加载 n
```

```
    lla  a1, data_array # 参数2: 数组地址
    call putarray

    # --- return 0 ---
    li   a0, 0          # 设置返回值为 0

    # --- 函数尾声 ---
    ld   ra, 8(sp)      # 恢复返回地址
    addi sp, sp, 16     # 释放栈帧
    ret                 # 返回

# --- bubble_sort 函数 ---
# 参数: a0 = arr[] (地址), a1 = len (整数)
# 使用的 s 寄存器: s0=arr, s1=len, s2=i, s3=j
.globl bubble_sort
bubble_sort:
    # --- 函数序言 ---
    addi sp, sp, -48    # 分配48字节栈帧
    sd   ra, 40(sp)     # 保存 ra
    sd   s0, 32(sp)     # 保存 s0
    sd   s1, 24(sp)     # 保存 s1
    sd   s2, 16(sp)     # 保存 s2
    sd   s3, 8(sp)      # 保存 s3

    # 将参数移动到 s 寄存器
    mv   s0, a0         # s0 = arr base address
    mv   s1, a1         # s1 = len

    # i = 0
    li   s2, 0          # s2 = i = 0

.L_outer_loop_start:
    # while (i < len - 1)
    addi t0, s1, -1     # t0 = len - 1
    bge  s2, t0, .L_outer_loop_end # if (i >= len - 1) goto end

    #   j = 0
    li   s3, 0          # s3 = j = 0

.L_inner_loop_start:
    #   while (j < len - 1 - i)
    sub  t0, t0, s2     # t0 = (len - 1) - i
    bge  s3, t0, .L_inner_loop_end # if (j >= len - 1 - i) goto end

    #     if (arr[j] > arr[j + 1])
    #     计算 arr[j] 的地址
    slli t1, s3, 2      # t1 = j * 4 (byte offset)
    add  t1, s0, t1     # t1 = addr(arr[j])
```

```
    #     计算 arr[j+1] 的地址
    addi t2, s3, 1      # t2 = j + 1
    slli t2, t2, 2      # t2 = (j + 1) * 4
    add  t2, s0, t2     # t2 = addr(arr[j+1])
    #     加载值
    lw   t3, 0(t1)      # t3 = arr[j]
    lw   t4, 0(t2)      # t4 = arr[j+1]

    ble  t3, t4, .L_skip_swap # if (arr[j] <= arr[j+1]) skip

    #       交换
    sw   t4, 0(t1)      # arr[j] = arr[j+1]
    sw   t3, 0(t2)      # arr[j+1] = arr[j]

.L_skip_swap:
    #     j = j + 1
    addi s3, s3, 1
    j    .L_inner_loop_start

.L_inner_loop_end:
    #   i = i + 1
    addi s2, s2, 1
    j    .L_outer_loop_start

.L_outer_loop_end:
    # --- 函数尾声 ---
    ld   ra, 40(sp)     # 恢复 ra
    ld   s0, 32(sp)     # 恢复 s0
    ld   s1, 24(sp)     # 恢复 s1
    ld   s2, 16(sp)     # 恢复 s2
    ld   s3, 8(sp)      # 恢复 s3
    addi sp, sp, 48     # 释放栈帧
    ret                 # 返回
```

测试运行汇编程序

正确性及性能测试命令

1. 对标准程序`bubble_sort.s`的正确性进行测试

> ```
> riscv64-unknown-elf-gcc bubble_sort.s -o bubble_sort_exec -L
> ../../SysY_Compiler_2025/lib -lsysy_riscv -static -mcmodel=medany
> qemu-riscv64 ./bubble_sort_exec
> 运行结果
> lalazhuuby@LAPTOP-C4QFSRSK:~/prehw/sort_riscv$ qemu-riscv64
> ```

```
./bubble_sort_exec
5
6
7
3
2
1
5: 1 2 3 6 7
TOTAL: 0H-0M-0S-0us
```

2. 量化baseline性能

`generate_input.py` 生成50000万个倒序的数字，然后将其存入input.txt作为冒泡排序的输入：

```
import sys

# --- 可配置的参数 ---
NUM_ELEMENTS = 50000

# --- 脚本主逻辑 ---
# 第一行输出：要排序的元素总数
print(NUM_ELEMENTS)

# 接下来输出 N 到 1 的数字序列，用空格隔开
# 这构成了冒泡排序的最坏情况
numbers = [str(i) for i in range(NUM_ELEMENTS, 0, -1)]
print(" ".join(numbers))

#运行 python3 generate_input.py > input.txt
#计时 time qemu-riscv64 ./bubble_sort_exec < input.txt
```

3. 解读时间

- real: **墙上时钟时间 (Wall-clock time)**。从你按下回车键到命令执行完毕，真实世界流逝的时间。它会受到你电脑上其他正在运行的程序的影响。
- user: **用户态 CPU 时间 (User CPU time)**。这是我们的**黄金指标**！它表示 CPU 花在**执行你的程序代码（以及QEMU本身）**上的总时间。这个值能最精确地反映你的排序算法的计算量。
- sys: **内核态 CPU 时间 (System CPU time)**。CPU 花在为你的程序执行内核调用（比如I/O操作）上的时间。对于计算密集型的排序任务，这个值通常很小。

确认正确性后新的baseline：消除输出

无输出源代码

```
# bubble_sort_no_output.s
# 功能: 实现冒泡排序算法，仅读取和排序，不输出结果，用于性能基准测试。

# --- 数据段 ---
.section .bss
.align 3                # 8字节对齐
.globl data_array       # 声明为全局符号
data_array:
    .space 200000       # 为50000个int分配空间

# --- 代码段 ---
.section .text
.globl main

# --- main 函数 ---
# 功能: 读取数据，调用排序
main:
    # --- 函数序言 ---
    addi sp, sp, -16    # 分配16字节栈帧
    sd   ra, 8(sp)      # 保存返回地址

    # --- n = getarray(data_array) ---
    lla  a0, data_array # 参数1: 数组地址
    call getarray       # 调用getarray, 返回值 n 在 a0 中
    sd   a0, 0(sp)      # 将 n 保存到栈上

    # --- bubble_sort(data_array, n) ---
    lla  a0, data_array # 参数1: 数组地址
    ld   a1, 0(sp)      # 参数2: 从栈加载 n
    call bubble_sort

    # --- [已移除] putarray(n, data_array) ---
    # ld   a0, 0(sp)
    # lla  a1, data_array
    # call putarray

    # --- return 0 ---
    li   a0, 0          # 设置返回值为 0

    # --- 函数尾声 ---
    ld   ra, 8(sp)      # 恢复返回地址
    addi sp, sp, 16     # 释放栈帧
    ret                 # 返回

# --- bubble_sort 函数 ---
```

```
# （此部分与之前修正后的正确版本完全相同，无需改动）
.globl bubble_sort
bubble_sort:
    # --- 函数序言 ---
    addi sp, sp, -48
    sd   ra, 40(sp)
    sd   s0, 32(sp)
    sd   s1, 24(sp)
    sd   s2, 16(sp)
    sd   s3, 8(sp)

    # 将参数移动到 s 寄存器
    mv   s0, a0
    mv   s1, a1

    # i = 0
    li   s2, 0

.L_outer_loop_start:
    # while (i < len - 1)
    addi t0, s1, -1
    bge  s2, t0, .L_outer_loop_end

    #   j = 0
    li   s3, 0

.L_inner_loop_start:
    #   while (j < len - 1 - i)
    sub  t1, t0, s2
    bge  s3, t1, .L_inner_loop_end

    #     if (arr[j] > arr[j + 1])
    slli t2, s3, 2
    add  t2, s0, t2
    addi t3, s3, 1
    slli t3, t3, 2
    add  t3, s0, t3
    lw   t4, 0(t2)
    lw   t5, 0(t3)

    ble  t4, t5, .L_skip_swap

    #       交换
    sw   t5, 0(t2)
    sw   t4, 0(t3)

.L_skip_swap:
    #     j = j + 1
```

```
    addi s3, s3, 1
    j    .L_inner_loop_start

.L_inner_loop_end:
    #   i = i + 1
    addi s2, s2, 1
    j    .L_outer_loop_start

.L_outer_loop_end:
    # --- 函数尾声 ---
    ld   ra, 40(sp)
    ld   s0, 32(sp)
    ld   s1, 24(sp)
    ld   s2, 16(sp)
    ld   s3, 8(sp)
    addi sp, sp, 48
    ret
```

测试baseline性能命令

```
riscv64-unknown-elf-gcc bubble_sort_no_output.s -o benchmark_exec -L
../../SysY_Compiler_2025/lib -lsysy_riscv -static -mcmodel=medany
time qemu-riscv64 ./benchmark_exec < input.txt
```

baseline性能

```
real    0m1.798s
user    0m1.914s
sys     0m0.005s
```

确立黄金标准

找到“标准最优汇编代码”执行时间的最权威、最直接的方法，就是**利用高度优化的专业编译器（如GCC或Clang）来为我们生成一个参考答案。**

```
// bubble_sort_ref.c
// 用于生成高度优化的 RISC-V 汇编参考

// SysY 运行时库函数的外部声明
// 我们需要告诉C编译器这些函数存在，但不需要头文件
extern int getarray(int a[]);
extern void putarray(int n, int a[]);
```

```c
// 全局数组
int data_array[50000];

void bubble_sort(int arr[], int len) {
    int i = 0;
    int j = 0;
    int temp = 0;

    i = 0;
    while (i < len - 1) {
        j = 0;
        while (j < len - 1 - i) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
            j = j + 1;
        }
        i = i + 1;
    }
}

int main() {
    int n = 0;
    n = getarray(data_array);
    bubble_sort(data_array, n);
    // 我们在基准测试中不需要输出，所以这里注释掉
    // putarray(n, data_array);
    return 0;
}
```

优化级汇编代码生成及性能测试

```
riscv64-unknown-elf-gcc -S -O2 bubble_sort_ref.c -o bubble_sort_gcc_o2.s
riscv64-unknown-elf-gcc bubble_sort_gcc_o2.s -o benchmark_exec_gcc_o2 -L
../../SysY_Compiler_2025/lib -lsysy_riscv -static -mcmodel=medany
time qemu-riscv64 ./benchmark_exec_gcc_o2 < input.txt

real    0m1.672s
user    0m1.620s
sys     0m0.000s
```

对baseline的剖析

main 函数剖析

这个函数只在程序开始时执行一次，它的总耗时在整个程序中**可以忽略不计**，但它的正确性至关重要。

汇编代码	指令功能分析	性能评估
addi sp, sp, -16	**分配栈帧**: 将栈指针 sp 向下移动16字节，为 ra 和 n 预留空间。	**极快**: 1个时钟周期。
sd ra, 8(sp)	**保存返回地址**: 将 ra 寄存器（64位）的值存入栈上 sp+8 的位置。	**较慢**: 内存写操作。
lla a0, data_array	**加载地址**: 将全局数组 data_array 的地址加载到 a0 寄存器。	**快**: 1-2个时钟周期。
call getarray	**调用函数**: 跳转到 getarray 函数执行。	**开销大**: 包含跳转、参数传递、以及函数内部的所有I/O和计算。这是 main 函数中最耗时的部分。
sd a0, 0(sp)	**保存返回值**: 将 getarray 的返回值 n (在a0中) 存入栈上 sp+0 的位置。	**较慢**: 内存写操作。
lla a0, data_array	**加载地址**: 再次加载 data_array 的地址到 a0，准备传给 bubble_sort。	**快**: 1-2个时钟周期。
ld a1, 0(sp)	**加载参数**: 从栈上 sp+0 的位置加载 n 的值到 a1 寄存器。	**较慢**: 内存读操作。
call bubble_sort	**调用函数**: 跳转到 bubble_sort 执行。	**核心耗时**: 整个程序的绝大部分时间 (~1.914s) 都将消耗在这里面。
li a0, 0	**设置返回值**: 将立即数0加载到 a0 寄存器。	**极快**: 1个时钟周期。
ld ra, 8(sp)	**恢复返回地址**: 从栈上 sp+8 将之前保存的 ra 值加载回 ra 寄存器。	**较慢**: 内存读操作。
addi sp, sp, 16	**释放栈帧**: 将栈指针 sp 恢复原位。	**极快**: 1个时钟周期。
ret	**返回**: 跳转到 ra 寄存器中的地址，结束 main 函数。	**快**: 1-2个时钟周期。

main 函数总结: 逻辑正确，严格遵守调用约定。主要开销在于 call getarray 和 call bubble_sort。

---

bubble_sort 函数剖析 (性能瓶颈的核心)

这个函数将被调用一次，但其内部的循环会执行**数十亿次**指令。

函数序言 (Prologue)

汇编代码	指令功能分析	性能评估
addi sp, sp, -48	**分配栈帧**: 为 ra 和 4个 s 寄存器分配48字节空间。	**极快**: 1周期。
sd ra, 40(sp) ...	**保存寄存器**: 5条 sd 指令，将 ra, s0-s3 的值依次存入栈中。	**慢**: 5次内存写操作。这是GCC -O2 版本完全避免的开销。
mv s0, a0 / mv s1, a1	**参数“缓存”**: 将传入的 arr 地址和 len 存入 s 寄存器，以便在函数体内安全使用。	**极快**: 2周期。
li s2, 0	**初始化 i**: i = 0。	**极快**: 1周期。

外层循环 (.L_outer_loop_start)

汇编代码	指令功能分析	性能评估
addi t0, s1, -1	**计算边界**: t0 = len - 1。	**极快**: 1周期。
bge s2, t0, ...	**循环判断**: if (i >= len - 1) 则跳转结束。	**快**: 1-3周期，取决于分支预测是否成功。
li s3, 0	**初始化 j**: j = 0。	**极快**: 1周期。

外层循环总结: 这部分代码执行 n-1 次（约5万次），每次的开销都非常小。

内层循环 (.L_inner_loop_start) - 性能风暴中心

这部分代码的**总执行次数约为 12.5 亿次**！这里的每一个时钟周期都至关重要。

汇编代码	指令功能分析	性能评估
sub t1, t0, s2	**计算内循环边界**: t1 = (len - 1) - i。	**极快**: 1周期。
bge s3, t1, ...	**内循环判断**: if (j >= len - 1 - i) 则跳转结束。	**快**: 1-3周期。
slli t2, s3, 2	**地址计算1**: t2 = j * 4 (字节偏移)。	**极快**: 1周期。
add t2, s0, t2	**地址计算2**: t2 = base_addr + j * 4 (得到 arr[j] 的地址)。	**极快**: 1周期。
addi t3, s3, 1	**地址计算3**: t3 = j + 1。	**极快**: 1周期。
slli t3, t3, 2	**地址计算4**: t3 = (j + 1) * 4。	**极快**: 1周期。
add t3, s0, t3	**地址计算5**: t3 = base_addr + (j + 1) * 4 (得到 arr[j+1] 的地址)。	**极快**: 1周期。
lw t4, 0(t2)	**加载 arr[j]**: 从内存加载 arr[j] 的值到 t4。	**慢**: **性能瓶颈 #1**。这是12.5亿次内存读操作之一。
lw t5, 0(t3)	**加载 arr[j+1]**: 从内存加载 arr[j+1] 的值到 t5。	**慢**: **性能瓶颈 #2**。这是另一次内存读操作。
ble t4, t5, ...	**比较**: if (arr[j] <= arr[j+1]) 则跳过交换。	**快**: 1-3周期。
sw t5, 0(t2)	**存储1 (交换)**: 将 arr[j+1] 的值写入 arr[j] 的内存位置。	**慢**: **性能瓶颈 #3**。在最坏情况下，会发生约12.5亿次内存写操作。
sw t4, 0(t3)	**存储2 (交换)**: 将 arr[j] 的值写入 arr[j+1] 的内存位置。	**慢**: **性能瓶颈 #4**。
addi s3, s3, 1	**j++**: j = j + 1。	**极快**: 1周期。
j .L_inner_loop_start	**跳转**: 无条件跳回内循环开头。	**快**: 1-2周期。

对O2版本黄金标准的剖析

整体印象

- **极致的寄存器使用**: 代码几乎完全在寄存器中进行计算，极力避免访问栈内存。
- **指针化思维**: 完全抛弃了整数索引 i 和 j，代之以指针的移动和比较。
- **精简的函数调用**: 省略了所有不必要的栈帧操作。
- **复杂的循环结构**: 使用了更复杂的递减指针和边界计算，虽然阅读起来更困难，但执行效率更高。

---

main 函数剖析 (GCC -O2 版)

汇编代码	指令功能分析	优化点解读
addi sp,sp,-16	分配16字节栈帧。	**标准操作。**
sd s0,0(sp)	保存 s0 寄存器。	**编译器防御性策略。**GCC 发现 main 虽然自身逻辑简单，但调用了外部函数 getarray 和 bubble_sort，它无法保证这些外部函数不会修改 s0，所以为了安全，还是保存了它。
lui s0,%hi(data_array)	加载 data_array 的**高20位**地址到 s0。	**标准地址加载 (第1步)。**这是 RISC-V 加载一个32位（或更高）地址的标准两步法。
addi a0,s0,%lo(data_array)	将 s0 中的高位地址与 data_array 的**低12位**地址相加，得到完整地址放入 a0。	**标准地址加载 (第2步)。**lui+addi 组合拳，比 lla 伪指令更底层。
sd ra,8(sp)	保存返回地址 ra。	**标准操作**，因为调用了其他函数。
call getarray	调用 getarray。	标准。
mv a1,a0	将 getarray 的返回值 n (在a0中) 直接移动到 a1。	**寄存器优化**！你的代码是将 n 存入栈再取出，GCC直接在寄存器之间传递，**避免了1次写内存和1次读内存**，非常高效。
addi a0,s0,%lo(data_array)	**地址复用**！再次用 addi 快速从 s0 计算出 data_array 的地址到 a0。	比再次执行 lla 更快，因为 s0 中已经缓存了地址的高位。
call bubble_sort	调用 bubble_sort。	标准。
ld ra,8(sp) / ld s0,0(sp)	恢复 ra 和 s0。	标准。
li a0,0	设置返回值为0。	标准。
addi sp,sp,16	释放栈帧。	标准。
jr ra	返回。(ret 的底层实现)	标准。

main 函数总结: GCC 在 main 函数中做的最亮眼的优化就是**通过寄存器直接传递返回值 n**，避免了不必要的内存读写。

---

bubble_sort 函数剖析 (GCC -O2 版) - 核心部分

函数序言 & 边界情况优化

汇编代码	指令功能分析	优化点解读
li a5,1	加载立即数1到 a5。	准备常数1。
ble a1,a5,.L1	if (len <= 1)，则直接跳转到 .L1 (函数末尾的ret)。	**边界情况优化。**如果数组只有0或1个元素，根本不需要排序。你的代码没有这个判断，会白跑一遍循环设置。这是一个简单但有效的优化。

循环初始化 (最复杂、最精妙的部分)

GCC 将 while (i < len-1) 和 while (j < len-1-i) 这两个嵌套循环的控制，完全重构成了基于**结束指针**的递减操作。

汇编代码	指令功能分析	优化点解读
addiw a5,a1,-2	a5 = len - 2 (32位加法)。这实际上是外层循环的总趟数。	**循环计数器优化。**
slli a4,a1,2	a4 = len * 4 (数组总字节数)。	**预计算。**
...	(slli, srli, sub) 这几条指令是复杂的64位地址计算，我们简化理解。	**高级地址计算。**
最终结果:		
a1	被计算为**外层循环的结束指针。**它指向的位置大约是 arr + (len - 1) * 4。	
a2	被计算为**第一轮内层循环的结束指针。**它指向的位置是 arr + (len - 1) * 4。	

循环体

汇编代码	指令功能分析	优化点解读
.L3:	**外层循环**的标签。	
mv a5,a0	a5 作为**当前内层循环的指针 p_j**，初始化为数组的起始地址 arr。	**指针化。**
.L5:	**内层循环**的标签。	
lw a4,0(a5)	a4 = *p_j。	**指针解引用**，非常高效。
lw a3,4(a5)	a3 = *(p_j + 4)。	**指针偏移+解引用**，利用了硬件的立即数偏移寻址模式，一条指令完成。
ble a4,a3,.L4	if (a4 <= a3)，跳过交换。	标准比较。
sw a3,0(a5) / sw a4,4(a5)	**交换。**	**指针化存储**，高效。
.L4:	跳过交换的标签。	
addi a5,a5,4	p_j = p_j + 4。	**指针移动**，用廉价的 addi 代替了你的 j++ 和一堆地址计算。
bne a5,a2,.L5	if (p_j != end_ptr_inner)，继续内层循环。	**指针比较**，这是内层循环的控制。
addi a2,a2,-4	**内层循环结束指针递减。**end_ptr_inner--。	**外层循环的体现**！外层循环的 i++ 被巧妙地转换成了内层循环边界的递减。
bne a2,a1,.L3	if (end_ptr_inner != end_ptr_outer)，继续外层循环。	**指针比较**，这是外层循环的控制。
.L1:		
ret	**返回。**	**无栈帧恢复**，极致精简。

bubble_sort 函数总结:
GCC的优化是**革命性**的，它完全抛弃了你源代码的结构，重新设计了一个功能等价但硬件亲和度极高的实现：

- **完全消除了栈帧操作**，因为它识别出这是一个叶子函数。
- **将所有循环控制都转换为了指针操作**，用廉价的指针移动 (addi) 和比较 (bne) 替代了昂贵的整数索引地址计算 (slli, add)。
- **通过递减结束指针**来巧妙地实现了嵌套循环的逻辑，代码更紧凑，分支更少。

对baseline的瓶颈优化——像编译器一样思考

无果