12 寄存器分配

徐辉，xuh@fudan.edu.cn

本章学习目标：

- 了解寄存器分配问题
- 掌握 AArch64 中主要寄存器的用法
- 了解着色问题和启发式着色思路
- 掌握基于单纯消除序列的寄存器分配算法

一般来讲，寄存器分配问题指的是将汇编代码中的虚拟寄存器转化为物理寄存器，使得汇编代码可在目标 CPU 上运行。该过程涉及寄存器预分配、通用寄存器分配和寄存器溢出环节。

12.1 预分配

AArch64 架构有 31 个 64-bit 通用寄存器 x0-x30，如果仅使用低 32-bit 则使用标识符 w0-w30。虽称为通用寄存器，但是这些寄存器在函数调用场景应当满足一些使用约定。如表 12.1所示，寄存器 x0-x1 用于参数和返回值传递，x30 用于保存函数返回地址，x29 一般用于保存栈帧基地址。另外，x9-x15 和 x19-x28 寄存器用途不限，但前者是 Caller-saved，后者是 Callee-saved。Caller-saved 指的是 Caller 负责保障函数调用前后寄存器的值不变，即调用前备份，调用后还原；Callee-saved 则是由 Callee 负责，即使用前备份，使用后还原。

表 12.1: AArch64 中的寄存器用法约定

寄存器名称	**调用规约**	**注释**
x0-x1	参数 1-2/返回值	
x2-x7	参数 3-8	
x8	特殊用途：间接调用返回地址	
x9-x15	普通寄存器	Caller-saved
x16-x18	特殊用途	
x19-x28	普通寄存器	Callee-saved
x29	栈帧基地址	
x30	返回地址	

预分配指的是寄存器的用法应满足该指令集架构下的一些使用约定，可总结为以下几点：

- **参数和返回值传递**：LLVM IR 中的函数调用和返回均是由单条 IR 指令完成的，参数和返回值传递未考虑调用规约的问题，将其翻译为汇编代码时应：函数调用前先将参数按照顺序依次拷贝到 x0-x7 中；函数返回前将返回值拷贝到 x0-x1 中。
- **返回地址**：如果当前函数涉及一处或多处函数调用，应在函数入口处将 x30 寄存器内容保存到栈上，函数返回前将其还原。否则，Callee 会改写 x30 的值，导致无法正常返回。

- **其它调用规约**：如果使用 x9-x15 这类 Caller-saved 寄存器并涉及函数调用，应当在函数调用前将其备份，调用后还原；对于 x19-x28 这类 Callee-saved 寄存器，应当在使用前将其备份，函数返回前还原。

12.2 寄存器分配

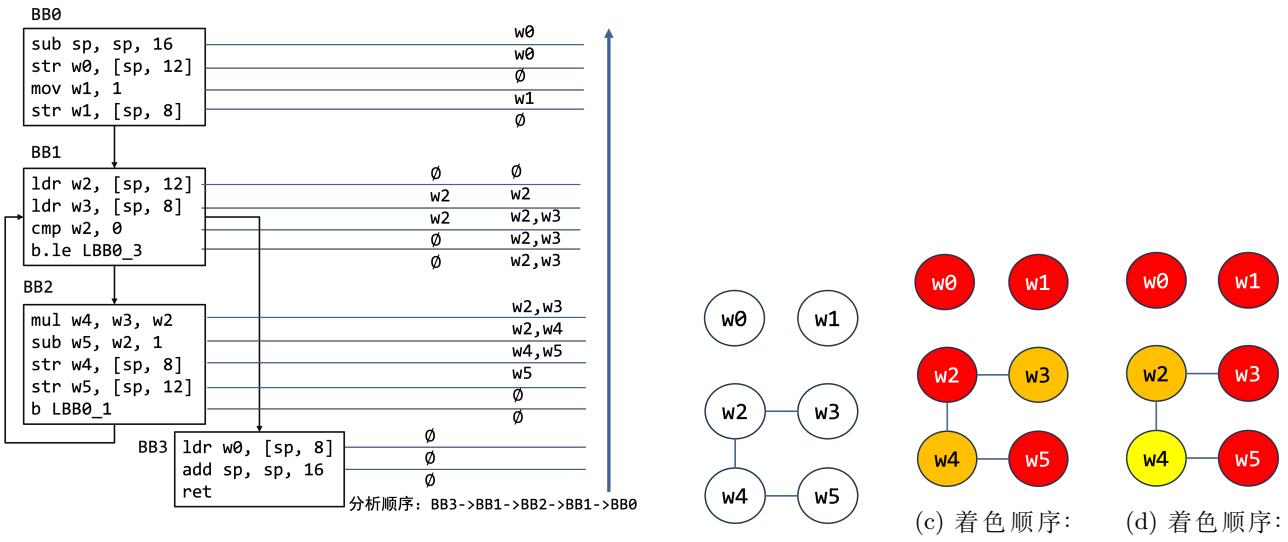
指令翻译时无需考虑虚拟寄存器数量限制，然而实际的物理寄存器数量是有限的。例如，AArch64 架构中一般使用 x9-x15 寄存器或 x19-x28 寄存器；X86 架构可用的寄存器则更少。因此，如何将虚拟寄存器翻译为物理寄存器非常关键。

下面我们使用干扰图对寄存器分配问题进行建模，并将其转化为着色问题。

12.2.1 干扰图构建

定义 1 (RIG)**.** 寄存器干扰图（Register Interference Graph）$\{V, E\}$ 是一个无向图，其中每一个点 $v_i \in V$ 表示一个虚拟寄存器，如果 $v_i, v_j \in V$ 同时活跃，则存在边 e_{ij}。存在干扰关系的虚拟寄存器应分配不同的物理寄存器。

干扰图是基于活跃性分析构建获得的，即分析每个虚拟寄存器的活跃区间。我们可以采用循环迭代数据流分析方法：对控制流图进行逆序分析；遇到 use(v_i) 则认为虚拟寄存器 v_i 在此之前都必须是活跃的，遇到 def(v_i)，则认为该虚拟寄存器最早活跃至此；遇到合并节点取并集即可。将同时存在活跃关系的虚拟寄存器两两相连便得到了干扰图。图 12.1a和 12.1b分别展示了一个活跃性分析示例及其干扰图构建结果。



(a) 活跃性分析　(b) 干扰图　(c) 着色顺序：0-1-2-3-4-5　(d) 着色顺序：0-1-3-2-5-4

图 12.1: 干扰图构建和着色示例

12.2.2 着色问题

基于干扰图的寄存器分配问题是一个着色问题。

定义 2 (着色问题)**.** 对无向图 $\{V, E\}$ 的所有点 $v_1, ..., v_n \in V$ 进行着色，要求相邻的点不能采用同样的颜色，请问至少需要多少个颜色？或是否存在至多使用 K 个颜色的着色方案？该问题又称为 K-colorable 问题，其中 K 个颜色代表 K 个物理寄存器。

K-colorable 问题在 $K \geq 3$ 时是 NP-Complete 的问题。

12.2.3 着色算法

我们假设存在颜色数组 Color = [红, 澄, 黄, 绿, 青, 蓝, 紫]，每次着色均采用当前编号最低的可能的颜色，即算法 1。则着色问题本质上是着色顺序选取的问题。如图 12.1c和 12.1d分别对应两种着色顺序，一个需要使用两个颜色，另外一个则需要使用三个颜色。

算法 1 颜色选取方法

```
1: procedure GreedyColoring(G(V, E))
2:     let C = {c_0, ..., c_k} be K colors
3:     for each v_i  s ∈ V do:
4:         let c_j be the lowest color not used in Adj(v_i)
5:         Col(v_i) = c_j
6:     end for
7:     Return G(V, C, E)
8: end procedure
```

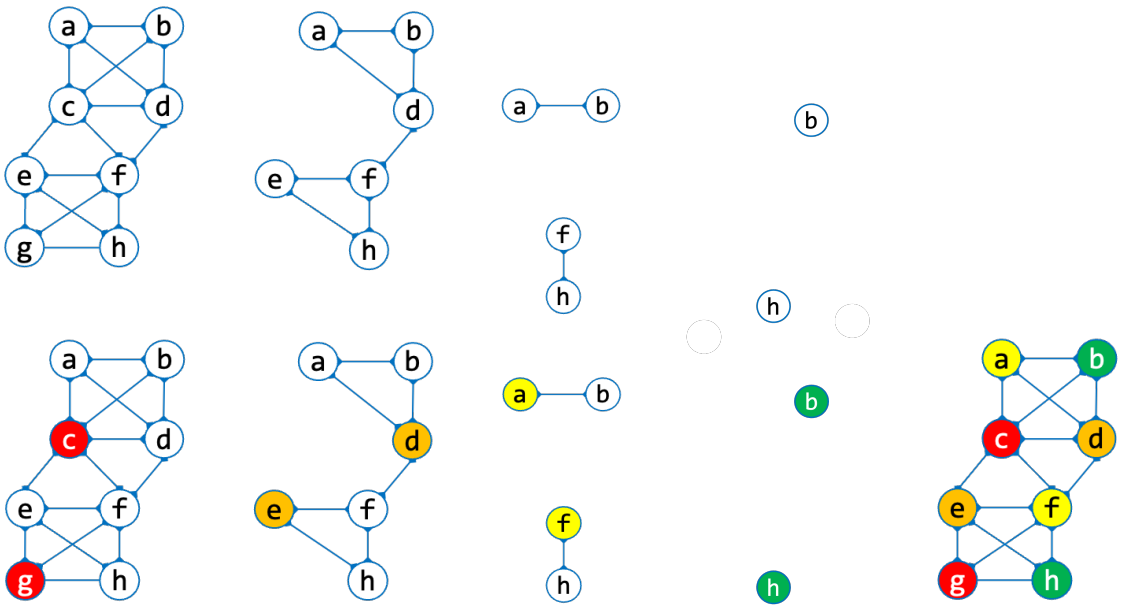
针对着色问题有大量的贪心算法研究。比如线性扫描算法采用先到先得的思想，对于先遇到的寄存器先分配颜色。该方法的有点是非常块，无需维护干扰图的边数信息。另外还有一些基于干扰图边数选取着色顺序的启发式算法，下面介绍一种经典的 RLF（Recursive Largest First）算法 [1]。如算法 2所示，该方法分为多轮次递归进行。每一轮次从图中选取度数最大（边数最的）的点进行着色；如果其余点中存在与该点不相连的点，则继续从中选取度数最大的点并采用相同的颜色着色，直至选择不出不相连的点为止。重复该过程直至全部点都被着色。

算法 2 Recursive largest first 算法

```
1: let S be the stack of nodes to be colored in one round; init S with empty.
2: let C = {c_0, ..., c_k} be K colors
3: procedure RLF(G(V, E))
4:     Find v_i ∈ G with the max degree
5:     S.push(v_i)
6:     Let T be the rest nodes in G non-adjacent to any node in S
7:     while T is not NULL do
8:         Find v_j ∈ T with the max degree
9:         S.push(v_j)
10:        Update T
11:    end while
12:    GreedyColoring(S)
13:    Remove S from G
14:    Set S to empty
15:    RLF(G)
16: end procedure
```

12.3 基于单纯消除序列着色

下面介绍一类特殊的着色问题：即当虚拟寄存器满足 SSA 形式时，则该着色问题不是 NP-hard 问题。我们可以采用基于单纯消除序列的方法选取最优着色顺序，确定最优着色方案 [2]。



(a) 第 1 轮　(b) 第 2 轮　(c) 第 3 轮　(d) 第 4 轮　(e) 结果

图 12.2: 应用 RLF 算法进行着色示例

下面首先定义单纯消除序列的相关概念。

定义 3 (单纯点). 无向图 $\{V, E\}$ 中的如果点 v_i 的所有邻居的邻居节点组成一个团（clique），则 v_i 是一个单纯点。

定义 4 (完美消除序列). 按照该序列消除的每一个点都是单纯点

定义 5 (单纯消除序列). 完美消除序列的逆序

定义 6 (弦图（Chordal Graph）). 无向图 $\{V, E\}$ 中的任意长度大于 3 的环都有弦。

当虚拟寄存器满足 SSA 形式时，其干扰图为弦图。弦图一定存在单纯消除序列。给定一个弦图，找单纯消除序列可以采用最大势算法，具体可参考算法 3。表 12.2展示了应用最大式算法找图 12.2e的单纯消除序列的过程。

表 12.2: 应用最大式算法找图 12.2e的单纯消除序列。

步骤		a	b	c	d	e	f	g	h
0	初始化	0	0	0	0	0	0	0	0
1	选取 a		1	1	1	0	0	0	0
2	选取 b			2	2	0	0	0	0
3	选取 c				3	1	1	0	0
4	选取 d					1	2	0	0
5	选取 f					2		1	1
6	选取 e							2	2
7	选取 g								3
8	选取 h								

算法 3

最大优势（Maximum Cardinality Search）算法

```
1: procedure MCS(G(V, E))
2:     for each v_i ∈ V do
3:         w(v_i) = 0;
4:     end for
5:     W = V;
6:     for each i ∈ [1..n] do
7:         Let v be a node with max weight in W
8:         for each u ∈ Neighbor(v) do
9:             w(u) = w(u) + 1
10:        end for
11:        W = W - v;
12:    end for
13: end procedure
```

12.4 寄存器溢出

当干扰图所需的颜色数量超过实际可用的物理寄存器时，部分寄存器的值需要溢出（spill）到内存中，以释放寄存器供其他用途。在值被溢出后，若再次需要使用，必须将其从内存重新加载到寄存器中。由于寄存器溢出会增加程序的运行开销，关键在于合理选择溢出的寄存器，以尽可能降低溢出代价。具体的寄存器选择通常与着色算法相关，同时也可以采用一些启发式方法，例如优先选择干扰图中最大团的顶点或度数最高的顶点进行溢出。

练习

1. 构造一个非弦图，使得 RLF 算法无法找到最优解。
2. 如果一个图是弦图，RLF 算法是否一定能找到最优解?

Bibliography

[1] Frank Thomson Leighton, “A graph coloring algorithm for large scheduling problems.” Journal of research of the national bureau of standards 84, no. 6 (1979): 489.

[2] Fernando Magno Quintao Pereira, and Jens Palsberg. “Register allocation via coloring of chordal graphs.” Asian Symposium on Programming Languages and Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.