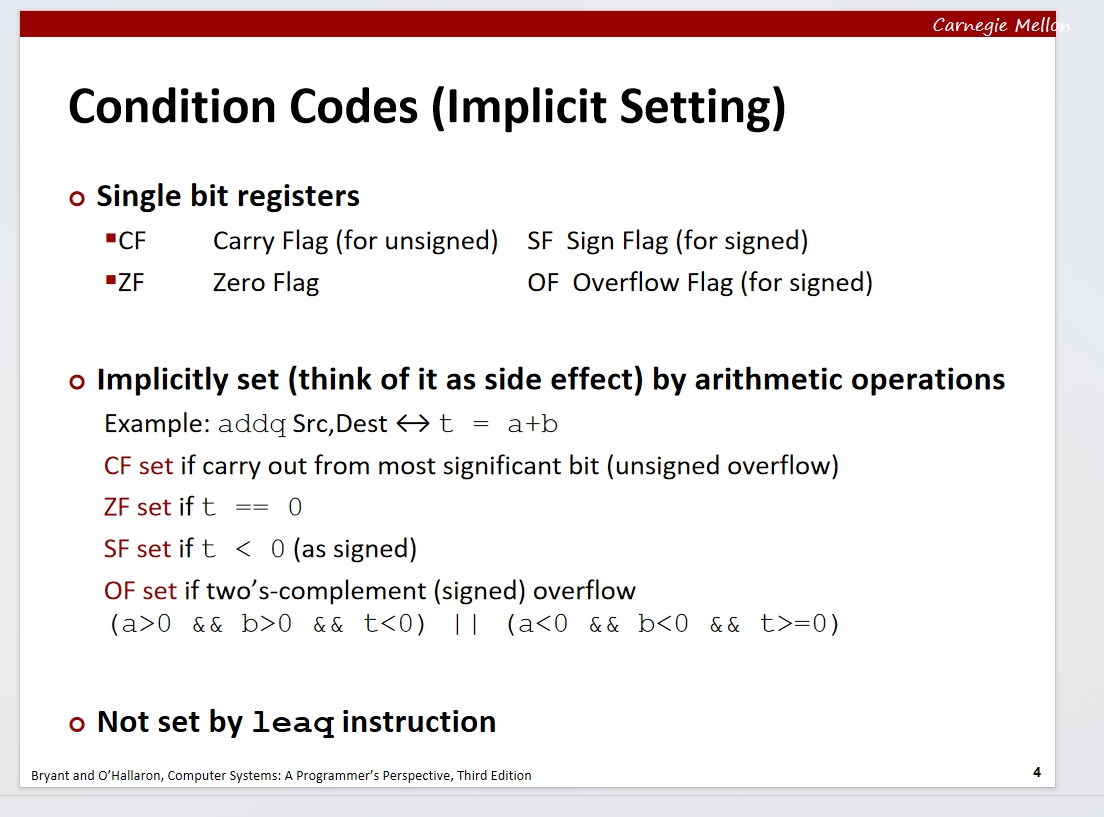


其他寄存器前面应该提到过一嘴了，这里主要讨论%rip

**PC**（程序计数器）和 **RIP**（指令指针寄存器）在功能上相似，都是用于跟踪当前执行指令的地址。

而在x86-64 架构中，PC 的功能由 RIP 寄存器承担。故在这里可以将RIP视为PC，而在其他架构中，PC的命名可能有所不同，但是它的功能是基本一样的，如PC一般，该寄存器寄存器的功能是储存当前的程序执行代码的地址，正常情况下其不能直接访问



这里对几个符号数进行解释（ai启动）

在 x86 架构中，\*\*CF\*\*（进位标志）、\*\*ZF\*\*（零标志）、\*\*OF\*\*（溢出标志）和 \*\*SF\*\*（符号标志）是用于表示运算结果状态的标志位。它们通常存储在 \*\*EFLAGS\*\* 或 \*\*RFLAGS\*\* 寄存器中。以下是每个标志的详细说明以及它们在何种情况下被设置：

### 1. \*\*CF（Carry Flag）进位标志\*\*

- \*\*含义\*\*：用于表示无符号数加法的进位或无符号数减法的借位。

- \*\*设置条件\*\*：

- \*\*加法\*\*：如果加法运算的结果超出了目标寄存器的表示范围（如从 255 加 1 得到 256），则 CF 会被设置为 1。

- \*\*减法\*\*：如果减法运算中被减数小于减数，CF 也会被设置为 1。

### 2. \*\*ZF（Zero Flag）零标志\*\*

- \*\*含义\*\*：用于表示运算结果是否为零。

- \*\*设置条件\*\*：

- 如果运算结果为零（例如，两个相同的数相减），则 ZF 被设置为 1。

- 任何情况下，只要结果为零，ZF 都会被设置。

### 3. \*\*OF（Overflow Flag）溢出标志\*\*

- \*\*含义\*\*：用于表示有符号数运算的溢出情况。

- \*\*设置条件\*\*：

- \*\*加法\*\*：如果两个正数相加得到一个负数，或者两个负数相加得到一个正数，OF 会被设置为 1。

- \*\*减法\*\*：如果从一个负数中减去一个正数得到一个正数，或者从一个正数中减去一个负数得到一个负数，OF 也会被设置为 1。

### 4. \*\*SF（Sign Flag）符号标志\*\*

- \*\*含义\*\*：用于表示运算结果的符号（正或负）。

- \*\*设置条件\*\*：

- SF 根据运算结果的最高位（符号位）进行设置。如果结果是负数（符号位为 1），SF 被设置为 1；如果结果是正数（符号位为 0），SF 被设置为 0。

### 总结：

- \*\*CF\*\*：加法中进位或减法中借位。

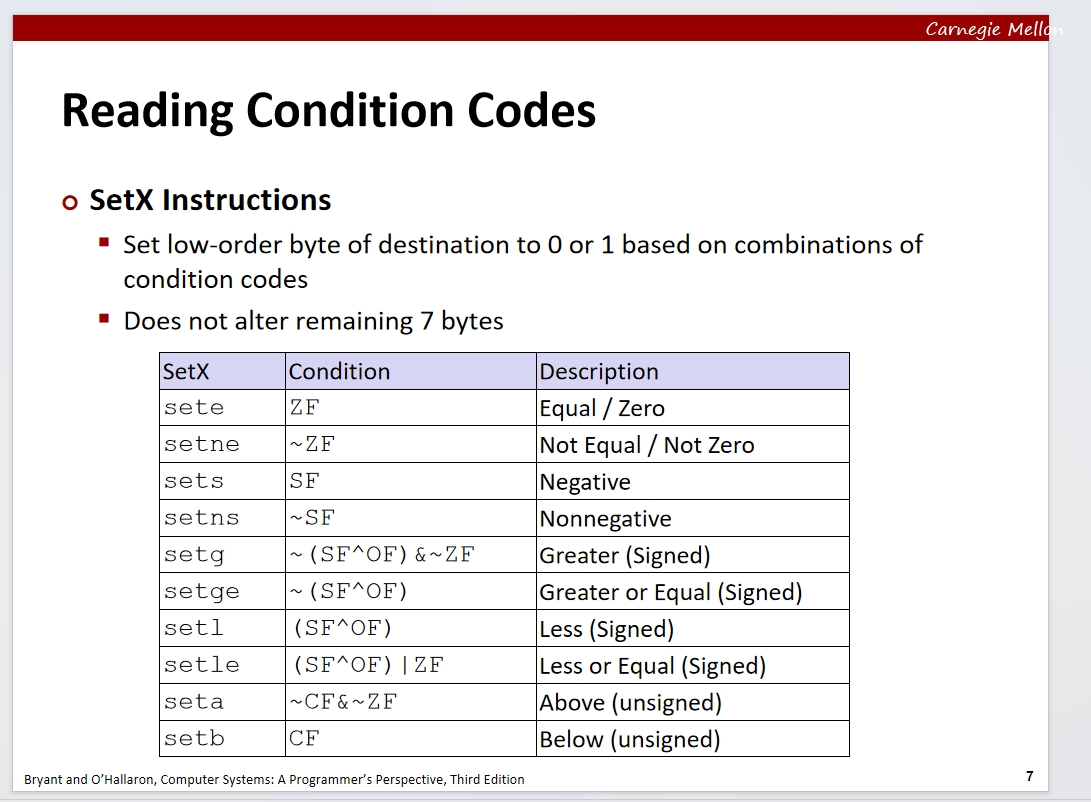
- \*\*ZF\*\*：运算结果为零。

- \*\*OF\*\*：有符号数运算溢出。

- \*\*SF\*\*：运算结果的符号（负或正）。

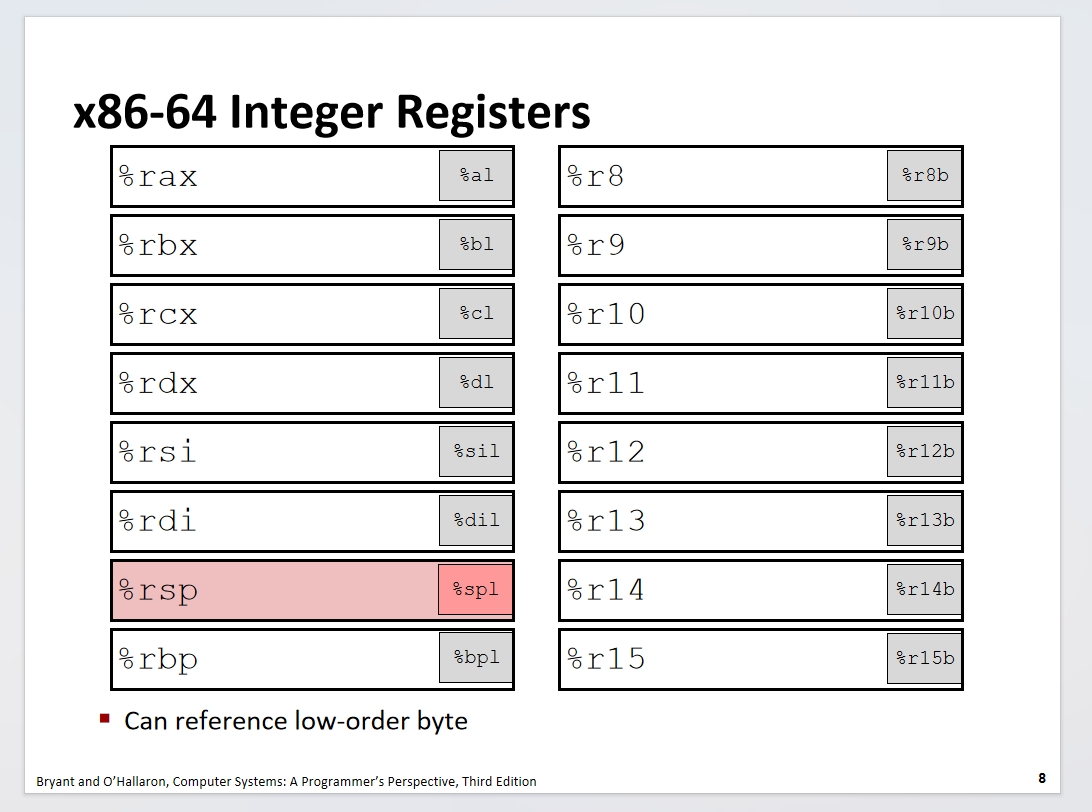
这些标志在条件跳转指令、算术运算、逻辑运算等过程中扮演重要角色，用于控制程序的执行流。

使用test指令能做的事（可能也是唯一的）：查看一个01串的属性



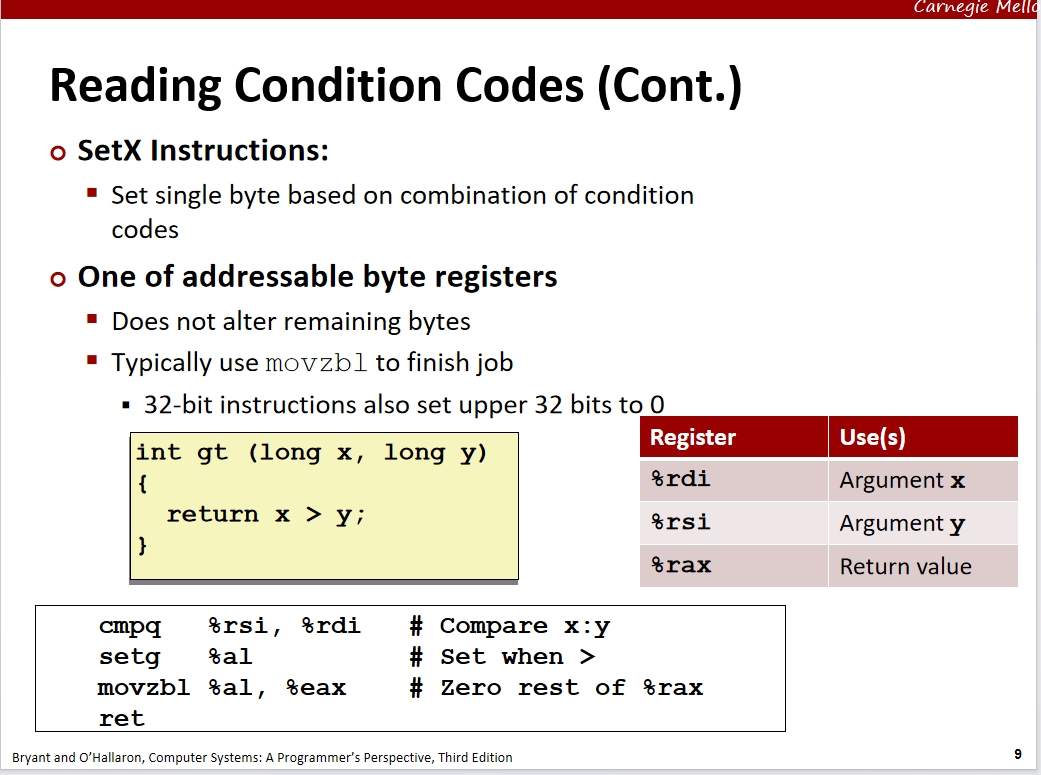
使用set可以对寄存器的某一个位进行单独的设置

即对单个寄存器的单个值设置0/1



这是是各个寄存器的情况，其中每个的右边的小框中是该寄存器中的一个子寄存器，一般都是该寄存器的低8位AL-低8位，AH-低16位的高8，EAX-由AL和AH组成的低16位的EX和高16位组成，而RAX跟EAX同理

在这些寄存器中，使用set会将该低8位寄存器中的最低一个字节进行设置1/0，也就是说这里的%al等寄存器会被设置为0/1对应的二进制码



接下来对这几行汇编代码进行分析

cmp %rsi %rdi

对rsi和rdi中的值进行比较，实际上应该是进行了一个减法的操作，接下来会将判断的结果储存到状态寄存器中，其本身运算的值并不会被储存

set %al

这个指令对上面比较指令的结果进行输出的定向，将cmp将设置的状态定向到%al寄存器中

movzbl %al %eax

这条指令是将一个寄存器中的值复制到另一个寄存器中去，并且进行高位补0

有关这里的mov指令格式解释如下：

movzbl 是指令的名称，其中：

* mov 表示移动数据。
* z 表示零扩展（zero extend）。
* b 表示操作的是字节（byte），即 8 位。
* l 表示操作的目标是双字（long），即 32 位。

对于不同优化等级的选择：-O0 -O1 -O2 -Og

编译器提供优化级别的分级（如 `-Og`、`-O1`、`-O2` 等）是为了在不同场景下平衡\*\*编译时间\*\*、\*\*执行效率\*\*和\*\*调试体验\*\*之间的关系。不同的优化级别代表编译器在生成目标代码时使用的优化策略的不同程度和种类。通过为用户提供多个优化级别，编译器可以满足从开发调试到生产环境不同的需求。

### 1. 常见的优化级别解释

在 GCC、Clang 等编译器中，常见的优化级别包括：

#### 1.1. `-O0`：无优化

- \*\*特性\*\*：完全不进行任何优化，这是编译的默认选项。每个语句都会严格按照原始的顺序编译并生成目标代码。

- \*\*用途\*\*：适合在开发和调试时使用。因为编译器不会重排代码，也不会进行任何改动，调试信息保留得最为完整，方便设置断点、逐步执行等操作。

- \*\*性能\*\*：执行速度最慢，生成的二进制文件通常比较大，但编译速度最快。

#### 1.2. `-O1`：基本优化

- \*\*特性\*\*：进行轻量级的优化，包括去除一些明显的冗余代码和死代码，减少不必要的内存访问，简单的寄存器分配等。不会进行大规模的代码重排或循环优化。

- \*\*用途\*\*：在调试过程中希望稍微提升性能但又不希望过多改变代码结构时使用。

- \*\*性能\*\*：相比 `-O0` 会有所提升，但不会对代码的结构做太大的调整，编译速度仍然较快。

#### 1.3. `-O2`：常规优化

- \*\*特性\*\*：在 `-O1` 的基础上，增加了更多的优化措施。包括更复杂的循环展开、函数内联、常量传播、移除不可达代码、提高指令流水线效率等。这是一个综合的优化级别，既提供了较好的性能提升，又不会增加太多编译时间。

- \*\*用途\*\*：适合用于一般的生产代码编译，追求性能的同时不会对调试体验产生太大的影响。

- \*\*性能\*\*：通常能显著提升程序的运行速度，编译时间也较合理，生成的代码体积通常较小。

#### 1.4. `-O3`：最大优化

- \*\*特性\*\*：在 `-O2` 的基础上进行更为激进的优化，包括更多的循环展开、更频繁的函数内联、矢量化优化等。编译器会尽量在不改变程序逻辑的前提下，使代码运行速度达到最大化。

- \*\*用途\*\*：适合对性能要求极高的场景，尤其是计算密集型应用。但此级别的优化可能会导致调试困难，因为编译器可能对代码进行大规模的重排、合并和移除操作。

- \*\*性能\*\*：通常生成的代码执行速度最快，但编译时间较长，调试时难以跟踪源代码和生成的机器代码之间的对应关系。

#### 1.5. `-Og`：调试优化

- \*\*特性\*\*：专门为调试而设计的优化级别，结合了一些性能优化（以减少调试时的代码开销），同时保留了良好的调试体验。它进行了一些基础优化（比如移除死代码），但不会对代码的结构进行复杂的重排。

- \*\*用途\*\*：适合在开发过程中使用，既可以获得比 `-O0` 更好的性能，同时又可以保留调试信息，方便排查问题。

- \*\*性能\*\*：相比 `-O0` 运行更高效，生成的代码体积较小，但不会像 `-O2` 那样进行深入的优化。

### 2. 为什么会有优化级别的分级

优化级别的分级是为了在以下几个方面取得\*\*平衡\*\*：

#### 2.1. 编译时间与执行性能之间的平衡

- 编译器的优化过程是非常复杂的，不同级别的优化可能需要更多的时间来分析代码、进行优化和生成目标代码。

- 在开发阶段，程序员更关注编译速度而不是执行速度，因此较低的优化级别（如 `-O0`、`-Og`）可以加快编译过程，减少开发等待时间。

- 在生产环境中，执行性能更为重要，较高的优化级别（如 `-O2`、`-O3`）会花费更多时间进行代码优化，但能够生成执行效率更高的程序。

#### 2.2. 调试体验与优化效果之间的平衡

- 优化过程通常会重排代码、合并指令，或者移除不必要的部分，这可能会使得编译后的代码与原始源代码不完全对应，给调试带来困难。

- 较低的优化级别（如 `-O0`）保留了完整的源代码结构，便于调试时逐行跟踪。而较高的优化级别可能导致编译器对代码进行大规模调整，调试时难以追踪代码的执行路径。

- `-Og` 则是为了解决这一问题，提供一个既可以优化性能，又能较好地支持调试的选项。

#### 2.3. 代码大小与执行效率之间的平衡

- 优化不仅影响执行速度，还会影响生成的代码体积。比如，函数内联和循环展开这些优化可以提升性能，但可能会增加生成的代码体积。

- 较高的优化级别（如 `-O3`）会生成更大的代码，因为它进行了更多的展开和内联操作，而较低的优化级别则生成较小的代码，尤其是在嵌入式设备或资源受限的系统中，这一点尤为重要。

#### 2.4. 兼容性与稳定性之间的平衡

- 较高级别的优化可能在某些极端情况下引入不可预见的问题，如某些情况下的代码行为可能与开发者的期望不同，尤其是当编译器进行激进的优化时。

- 因此，某些高性能要求的代码（如计算密集型或性能关键的部分）可以使用 `-O3`，而对于一般应用或者那些对调试和稳定性要求高的代码，可以选择 `-O2` 或 `-Og`。

### 3. 各个优化级别的具体应用场景

- \*\*`-O0`\*\*：适用于开发和调试早期阶段，尤其是当你需要逐行检查程序执行时。

- \*\*`-Og`\*\*：适用于日常开发，既能获得调试友好的信息，也能提升代码性能。

- \*\*`-O1`\*\*：适合需要轻量优化但又希望保留较好调试信息的场景。

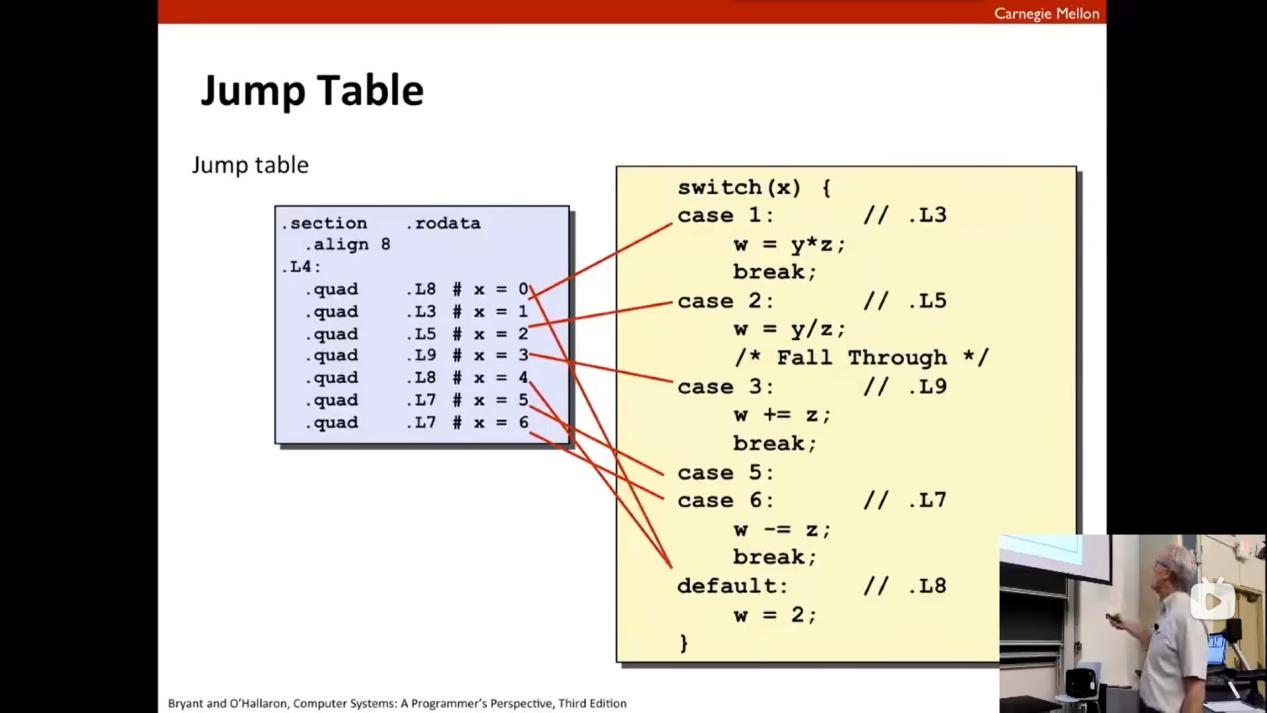
- \*\*`-O2`\*\*：适合一般的生产环境，提供较高的性能提升，是编译器推荐的优化级别之一。

- \*\*`-O3`\*\*：适用于计算密集型、性能关键的应用程序，尤其是生产环境中的性能敏感型应用。

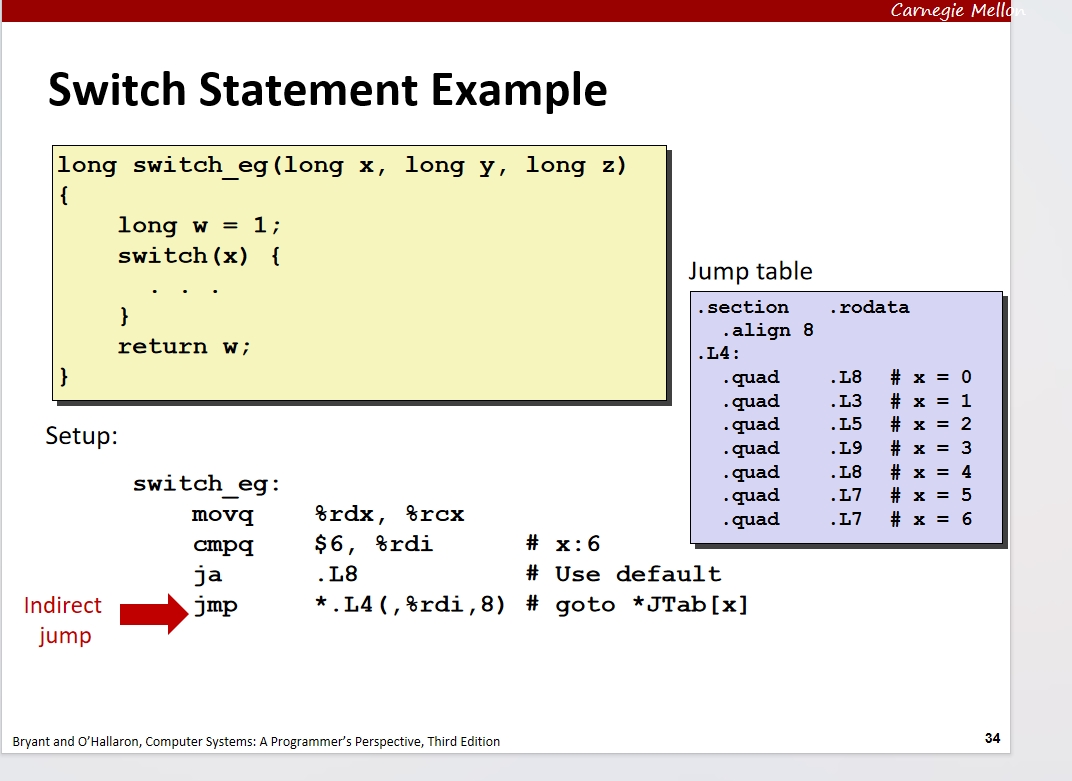
### 4. 总结

编译器提供优化级别的分级是为了满足不同开发阶段、不同应用场景下的需求。在开发阶段，程序员可能更关注调试和快速编译，因此会选择较低的优化级别。而在生产环境中，代码的执行效率至关重要，此时较高级别的优化可以发挥作用。优化级别分级的设计旨在提供灵活性，使程序员能够根据项目的需求选择最合适的编译选项。

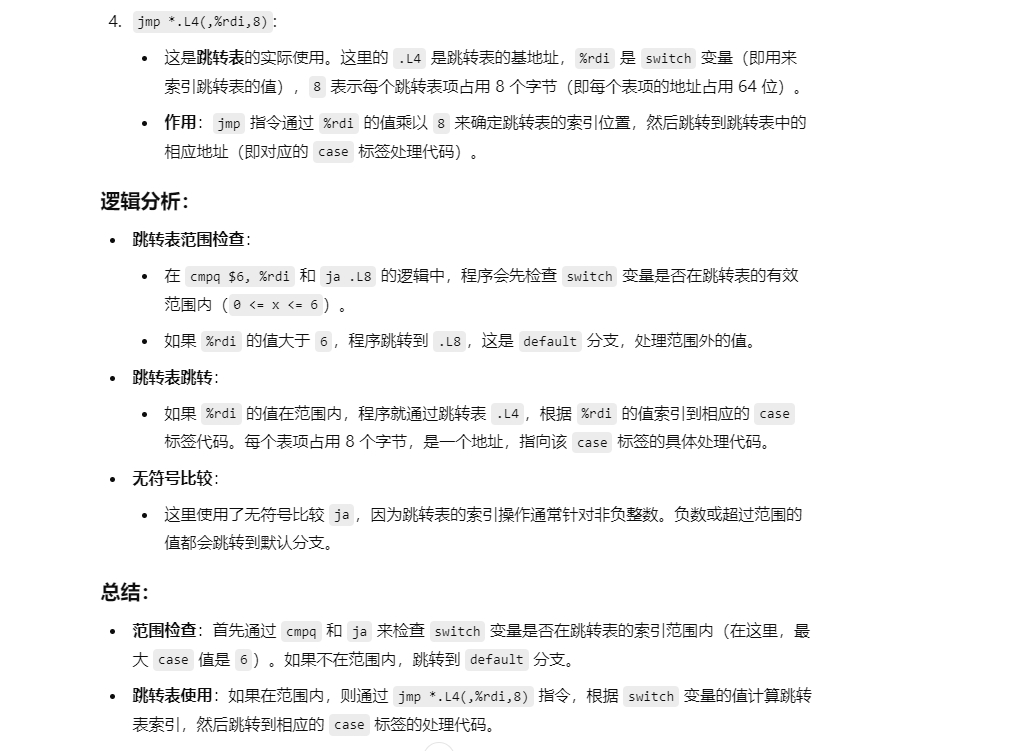
switch相对于If-else的区别，简单来说，最简单的区别就是它switch的跳转是一步到位的，其不再需要对于跳转后再进行条件判断，这使得它在某些情况下比if-else更加高效



对于swtich语句，程序会先找出给出的case中的最大值和最小值，并依据这一个形成一个跳转表，对于在case语句中这个范围内的没出现的数，程序会对这些数进行添加标签到表并且目标地是默认分支，同时在输入中，程序通过比较上下限来判断是否在索引表的范围内，在就直接跳到对应索引，对于一些不在这个跳转表范围的数，其也会被跳到默认分支(如果有)



这上面的几行汇编是真tm nb啊



这里没有对小于最小值的情况做处理，自己查ai问什么吧，不想写了，至于还需要一点，switch到底是不是跳转表的实现取决于具体的case标签值的分布情况