第二章　指令系统原理与实例

- 2.1 简介
- 2.2 指令集系统结构的分类
- 2.3 存储器寻址
- 2.4 操作数的类型
- 2.5 指令系统的操作
- 2.6 控制流指令
- 2.7 指令系统的编码
- 2.8 编译器的角色
- 2.9 MIPS系统结构
- 2.10 谬误和易犯的错误
- 2.11 结论

2.8 编译器的角色

编译器功能：将高级、抽象表示方式逐步转换成低级表示形式，最终到达机器目标指令代码。

- 机器运行的大多数指令代码是编译器的输出。
- 早期：编译器是在机器实现后再开发的。
- 现在：机器性能不仅是其原始速度，还包括编译器如何利用系统结构特征。（编译器在处理器设计阶段开发，编译后的代码在模拟器上运行，代码用来评估系统结构特征）

2.8编译器的角色

- 编译器开发的两个主要目标：

  * 正确性

  * 编译后的代码的执行速度。重要的是针对当前系统结构的优化技术。

- 与编译相关的主要系统结构特征

（1）并行性

指令级并行：编译器通过重新调度指令顺序，配合硬件使指令高效执行。

处理器并行：程序员编写多线程，也可以编译器自动生成并行代码。

（2）存储层次结构：寄存器—Cache—主存—外存

高效使用寄存器是优化处理的重要问题。

对于数组计算，改变数据布局或数据访问顺序提高访存效率，减少访存次数；改变代码布局提高访问命中率。

2.8 编译器的角色

指令系统的设计方案的选择影响：

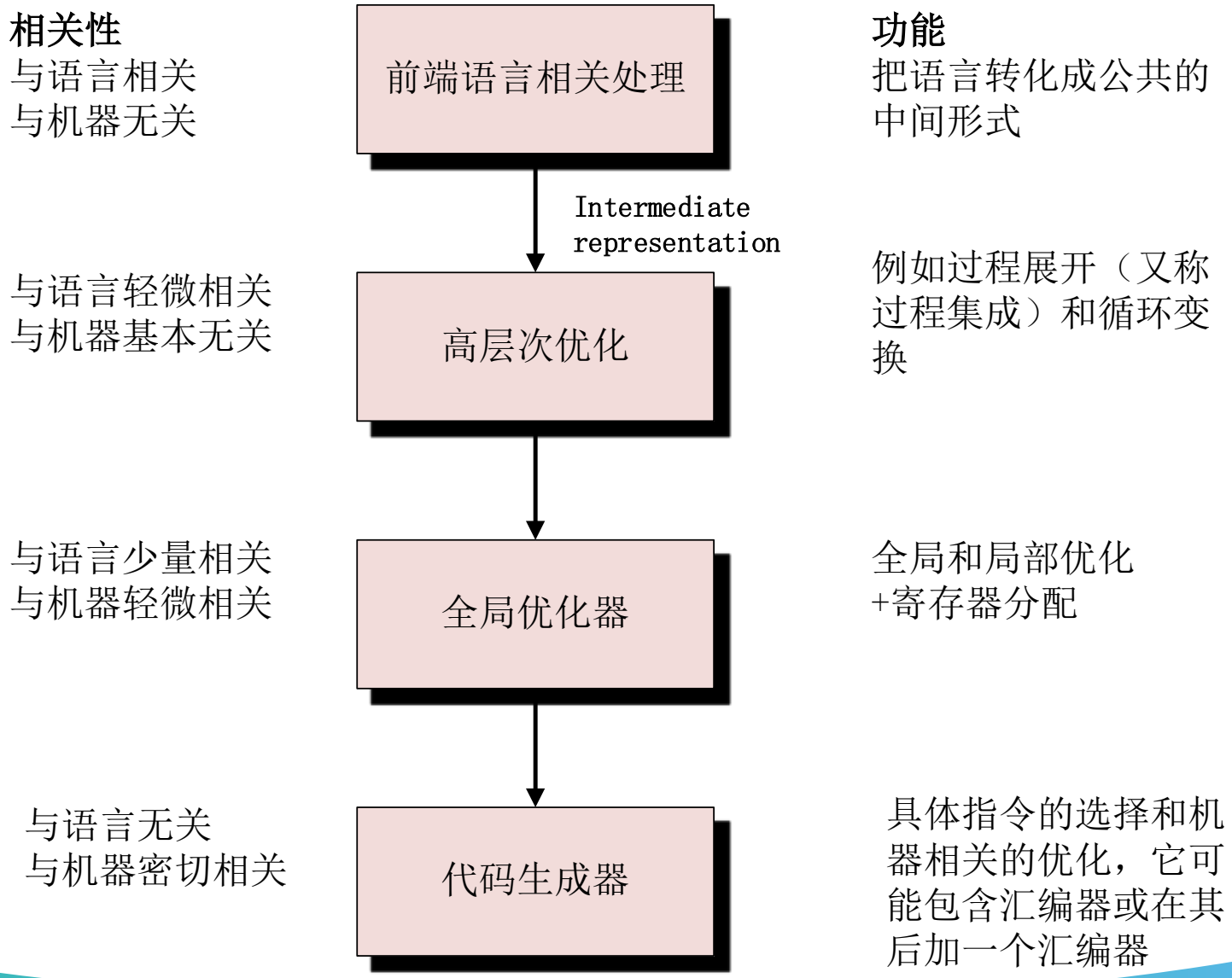
- 编译器产生**指令代码的质量**，如代码大小、代码复杂度、代码效率。
- 构建一个高效编译器**的复杂度**。

例如：指令系统是CISC为主，还是RISC为主？

2.8编译器的角色

❖ 编译器的结构

Figure A.19



2.8编译器的角色

- 近来编译器的结构

近代编译器一般包括2~4遍扫描，而性能更高的编译器包括更多次的扫描。

* 编译目标----较高编译速度、较低代码质量可接受时，优化扫描可以被跳过。一遍扫描即可。

* 优化扫描：被设计成一个可选项。

由于优化扫描是独立的，因此同一种ISA上不同的语言可以使用相同的优化与代码生成步骤。对于一个新的语言，只需要一个新的前端程序就可以了。

2.8编译器的角色

- 编译器优化方式的分类

  高级语言优化，一般在源码上进行，同时把输出传递给以后的优化扫描步骤。

  局部优化，仅在一系列代码片段之内将代码优化。

  全局优化，将局部优化扩展为跨越分支，并且引入一组针对优化循环的转换。

  寄存器分配，将寄存器与变量联系起来。

  处理器相关的优化，充分利用特定的系统结构。

2.8编译器的角色

❖ 优化对性能的影响（下图列出了优化转换用于源程序的频率）

优化名称	说明	在优化转换中所占比例
高级语言层	接近或者在源代码级，与机器无关	
过程集成	*用过程体替代过程调用*	*未被测试*
局部	在一系列代码中	
公共子表达式消去	*用一份副本替代一个计算的两个实例*	*18%*
常量传递	*把所有分配了一个常量的变量的实例用该常量替换*	*22%*
栈高度缩减	*重新安排表达式树使表达式计算所需要的资源最少*	*未被测试*
全局	跨越分支	
全局公共子表达式消去	*与局部的方法相同，但是这里会跨越分支*	*13%*
副本传递	*用X（如A=X）替换已经赋值以X的变量A的所有实例*	*11%*
代码移动	*把循环中每次迭代计算同一变量的代码从循环中移动*	*16%*
归纳变量消去	*在循环中简化或者消去数组地址的计算*	*2%*
处理器相关	依赖处理器结构	
长度缩减	*有很多例子，如用一个常数的加法和移位来代替乘法*	*未被测试*
流水线调度	*重新对指令进行排序以改进流水线的性能*	*未被测试*
分支偏移优化	*选择到达目标的最短分支位移路径*	*未被测试*

Figure A.20

2.8编译器的角色

- 优化对性能的影响

后图为编译优化级别不同时，SPEC2000中的lucas（fp）和mcf（int）两个测试程序指令条数的变化。这些实验是在Alpha的编译器上运行的。

0级---未优化代码。

1级---局部优化、代码调度和局部寄存器分配。

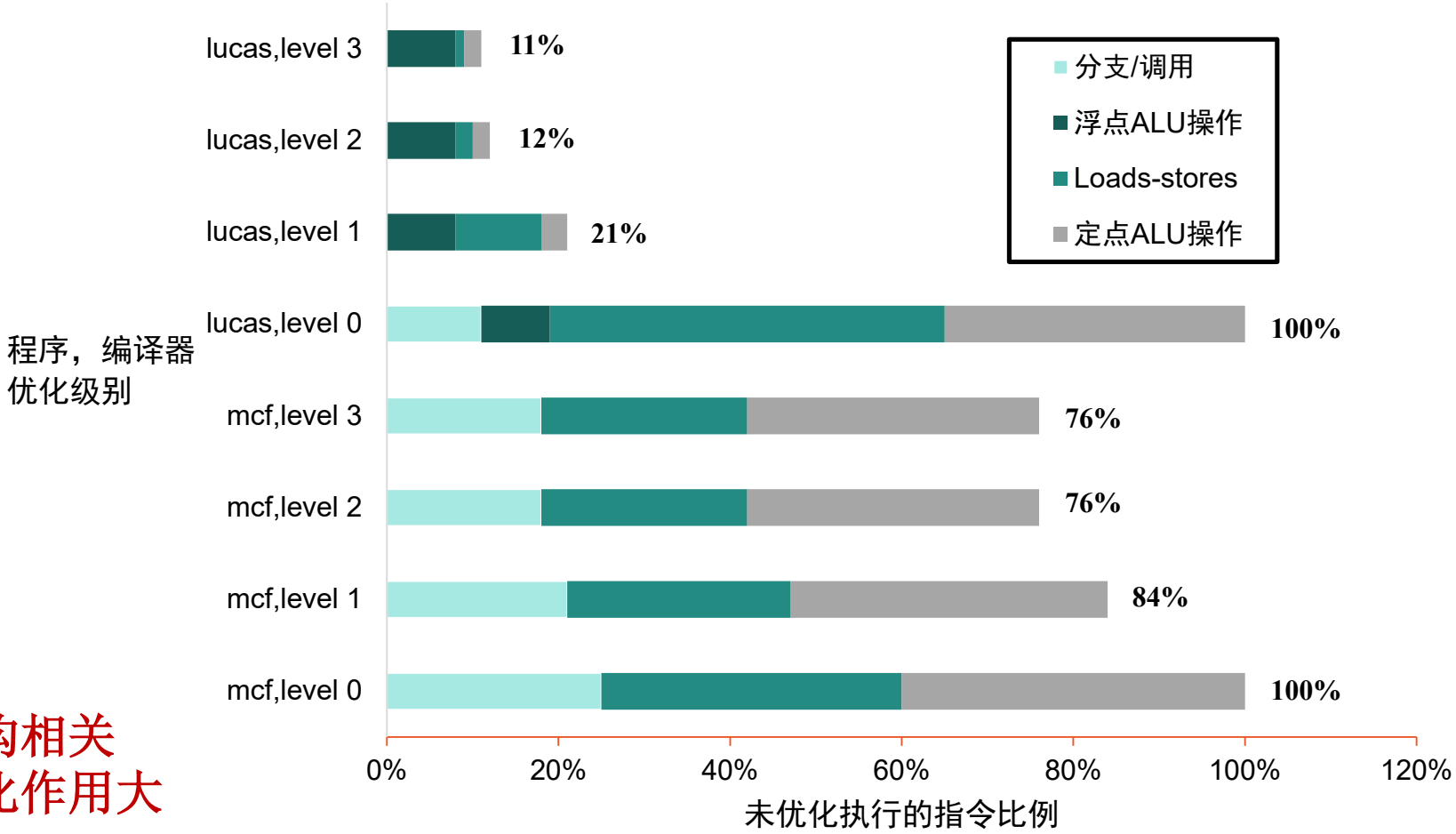
2级---在1级上增加了全局优化、循环转换（软件流水线），以及全局寄存器分配。

3级---在2级上增加了过程集成。

公共子表达式：表达式E在某次出现之前已被计算过，E中变量的值自计算后未改变。

2.8编译器的角色

- **优化对性能的影响**（下图为两个程序采用不同的优化方法对指令数目的影响。）



与结构相关的优化作用大

Figure A.21

2.8编译器的角色

- 编译技术对系统结构设计者的决策所产生的影响

  两个重要问题：变量是如何分配和寻址的？需要多少个寄存器才能有效地分配变量？

  高级语言保存数据的区域：

  * 栈---被用来分配局部变量。

  * 全局数据区---用来分配静态声明的对象，如全局变量和常量。

  * ---用来分配那些不适于放在栈中的动态对象（用指针访问）。

2.8编译器的角色

- 编译技术对系统结构设计者的决策所产生的影响
- 局部变量（栈分配的对象）---寄存器分配比全局变量有效。
- 分配对象---基本不能进行寄存器分配，因为对象是使用指针进行访问的。
- 有“别名”的全局变量以及变量---不能进行寄存器分配，因为它们具有“别名”，即有多种不同方式可以访问这个变量的地址，这使得将它放入寄存器是非法的。

2.8编译器的角色

- 寄存器分配

分配算法基于图着色的算法，其基本思想是：构造一个图，用它来代表分配寄存器的各个可能候选方案，然后用此图来分配寄存器。即如何用有限的颜色使图中相邻的节点分别着以不同的颜色。这种方法要强调是要100%地完成活动变量的分配。

图着色正常工作：在全局分配中有16个以上通用寄存器用于整型变量；同时另有其他的寄存器用于浮点数。

2.8编译器的角色

- 系统结构设计者对编译器设计者的技术支持

  遵循原则：确保经常出现的事件要尽量快（生成代码质量要高）；而不经常出现的事件则一定正确（生成代码质量一般）。

  有助于编译器设计者的指令系统特性（指导性）：

  * 规则性---操作、数据类型和寻址方式是正交的。
  * 提供单纯功能，而不是复杂功能指令----多种高级语言；
  * 在取舍时考虑简化的折中---帮助了解可替换代码的代价（源代码对应不同的指令序列）；
  * 对于编译时作为常量的数值量，提供能将其确定为常量的指令。

Example

- 下面的例子将一个向量计算机与MMX进行对比，将像素的色彩表示方式由RGB（红、绿、蓝）转换为YUV(发光度、色度），每个像素用3个字节表示。这种转换只需要三行C代码，放在循环中即可：

```
Y = (9798*R + 19235*G + 3736*B) / 32768;
U = (-4784*R - 9437*G + 4221*B) / 32768 + 128;
V = (20218*R - 16941*G - 3277*B) / 32768 + 128;
```

宽度为64位的向量计算机可以同时计算8个像素。一个采用自增寻址的媒体向量计算机将执行以下操作

- 3次向量载入（以获得RGB)
- 3次向量相乘（转换R)
- 6次向量乘加（转换G和B)
- 3次向量移位（除以32768)
- 2次向量加（加上128)
- 3次向量存储（存储YUV)

总共有20条指令用于执行前面C代码中转换8个像素的20个操作[Kozyrakis 2000]。（由于向量可能有32个64位寄存器，这一代码实际上可以转换最多32*8=256个像素。）

与之相对，Intel网站显示一个对8个像素执行相同计算的库例程使用了116条MMX 指令和6个80x86指令[Intel2001]。指令数之所以会增加到6倍是因为没有自增寻址存储器访问，需要大量的指令来载入RGB像素并解包，然后再打包并存储YUV像素。

Exercise A.5

考虑下面这个由3行语句组成的高级代码序列:

- A = B + C;
- B = A + C;
- D = A - B;

使用副本传递技术将此代码序列转换为所有操作数都不是计算值的情况。

优化名称	说明	在优化转换中所占比例
高级语言层	接近或者在源代码级，与机器无关	
过程集成	*用过程体替代过程调用*	*未被测试*
局部	在一系列代码中	
公共子表达式消去	*用一份副本替代一个计算的两个实例*	*18%*
常量传递	*把所有分配了一个常量的变量的实例用该常量替换*	*22%*
栈高度缩减	*重新安排表达式树使表达式计算所需要的资源最少*	*未被测试*
全局	跨越分支	
全局公共子表达式消去	*与局部的方法相同，但是这里会跨越分支*	*13%*
副本传递	*用X（如A=X）替换已经赋值以X的变量A的所有实例*	*11%*
代码移动	*把循环中每次迭代计算同一变量的代码从循环中移动*	*16%*
归纳变量消去	*在循环中简化或者消去数组地址的计算*	*2%*
处理器相关	依赖处理器结构	
长度缩减	*有很多例子，如用一个常数的加法和移位来代替乘法*	*未被测试*
流水线调度	*重新对指令进行排序以改进流水线的性能*	*未被测试*
分支偏移优化	*选择到达目标的最短分支位移路径*	*未被测试*

Exercise A.5

Consider this high-level code sequence of three statements:

- `A = B + C;`
- `B = A + C;`
- `D = A - B;`

Use the technique of copy propagation (see Figure A.20) to transform the code sequence to the point where no operand is a computed value.

Solution:

A = B + C;

B = A + C = B + C + C ;

D = A – B = (B + C) – (B + C + C) = – C ;

COD_5e Exercise 2.1

- For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables f, g, h, and i are given and could be considered 32-bit integers as declared in a C program. Use a minimal number of MIPS assembly instructions.
- 下面的C语言表达式对应的MIPS汇编语言代码是什么？假设给定变量f、g、h和i, 像在C程序中声明的一样。它们都是32位的整数， 使用最少的MIPS汇编指令。
- f = g + (h − 5);

Answer(MIPS)

- For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables f, g, h, and i are given and could be considered 32-bit integers as declared in a C program. Use a minimal number of MIPS assembly instructions.
- f = g + (h − 5);
- ```
  lw $t0 , g
  lw $t1 , h

  addi $t2 , $t1 , -5
  add  $t2 , $t0,  $t2

  sw $t2 , f
  ```
  ```

COD 5e Exercise 2.6

- The table below shows 32-bit values of an array stored in memory.
- 下表表示在主存中存放的一个数组的32位数据。

Address	Data
24	2
28	4
32	3
36	6
40	1

2.6.1

- For the memory locations in the table above, write C code to sort the data from lowest to highest, placing the lowest value in the smallest memory location shown in the figure. Assume that the data shown represents the C variable called `Array`, which is an array of type `int`, and that the first number in the array shown is the first element in the array. Assume that this particular machine is a byte-addressable machine and a word consists of four bytes.
- 基于上表中数据在存储器中的位置， 编写一段C代码， 将数据从小到大排序， 最小的数放在地址最低的位置。（假设这段数据代表了C的一个int型数组Array, 并且这台机器是按照字节寻址的，且一个字包含4字节。）

Address	Data
24	2
28	4
32	3
36	6
40	1

```
int mydata[11] = { 0, 0, 0, 0, 0, 0, 2 , 4 , 3 , 6 , 1 } ;

int main() {
  int temp0 , temp1 ;

  temp0  = mydata[6] ;
  temp1 = mydata[7] ;

  mydata[6] = mydata[10] ;
  mydata[7] = temp0 ;

  mydata[10] = mydata[9] ;
  mydata[9] = temp1 ;

  return 0;
}
```

Address	Data
24	2
28	4
32	3
36	6
40	1

Answer 2.6.1 C code

2.6.2

- For the memory locations in the table above, write MIPS code to sort the data from lowest to highest, placing the lowest value in the smallest memory location. Use a minimum number of MIPS instructions. Assume the base address of `Array` is stored in register `$s6`.
- 基于上表中数据在存储器中的位置， 编写一段MIPS代码，将数据从小到大排序，最小的数放在地址最低的位置。（使用最少的MIPS汇编指令，假设Array的基地址保存在寄存器$s6中。）

2.6.2

- For the memory locations in the table above, write MIPS code to sort the data from lowest to highest, placing the lowest value in the smallest memory location. Use a minimum number of MIPS instructions. Assume the base address of `Array` is stored in register `$s6`.

Address	Data
24	2
28	4
32	3
36	6
40	1

Address	Data
24	2
28	4
32	3
36	6
40	1

```
int mydata[11] = { 0, 0, 0, 0, 0, 0, 2 , 4 , 3 , 6 , 1 } ;

//------------------
// main()
//------------------
int main() {
  int temp0 , temp1 ;

  temp0  = mydata[6] ;
  temp1 = mydata[7] ;

  mydata[6] = mydata[10] ;
  mydata[7] = temp0 ;

  mydata[10] = mydata[9] ;
  mydata[9] = temp1 ;

  return 0;
}
```

```
.data
mydata :  .space 24
          .word  2 ,  4 ,  3 ,  6 ,  1
.text
.globl main
.ent main
main :
  la $s6 , mydata

  lw $t0 , 24($s6)
  lw $t1 , 28($s6)

  lw $t4 , 40($s6)
  sw $t4 , 24($s6)

  sw $t0 , 28($s6)

  lw $t3 , 36($s6)
  sw $t3 , 40($s6)

  sw $t1 , 36($s6)
.end main
```

Answer 2.6.2
MIPS assembly code