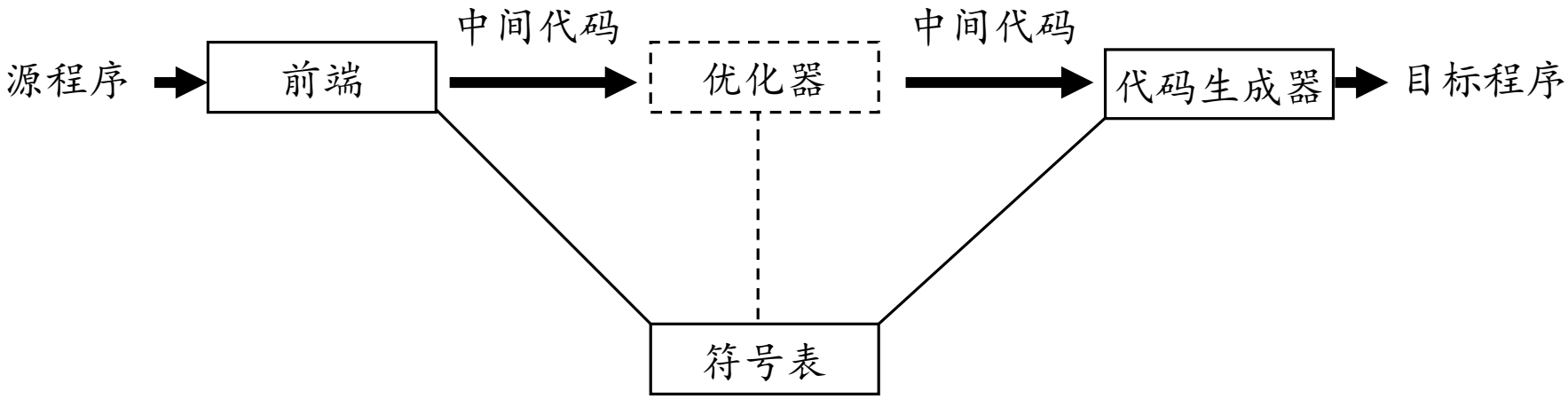
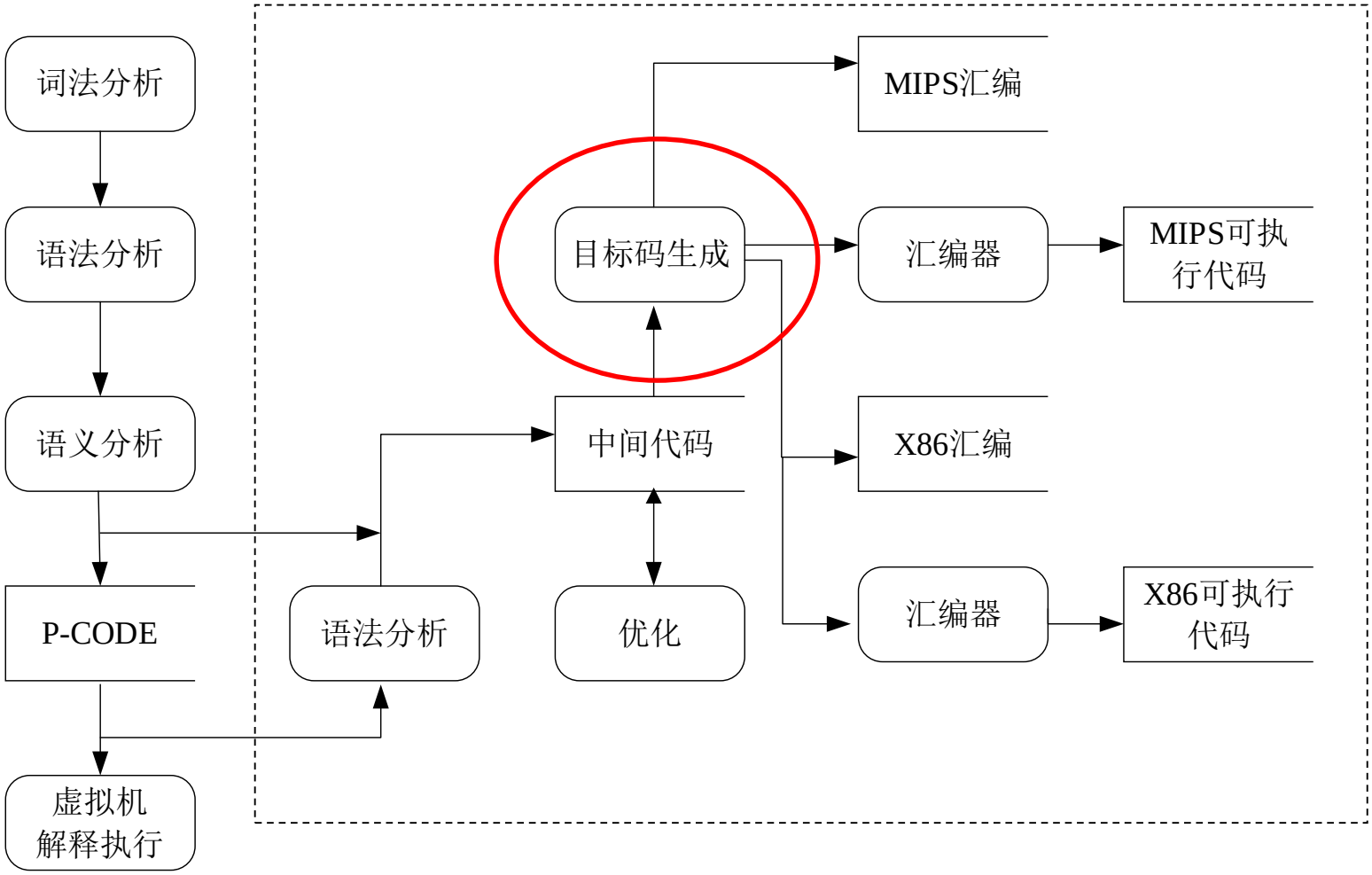
第十五章 目标代码生成及优化

面向目标体系结构的代码生成和优化技术

代码生成器在编译系统中的位置



教学编译器架构


词法分析
语法分析
语义分析
P-CODE
虚拟机
解释执行
语法分析
中间代码
优化
目标码生成
MIPS汇编
汇编器
MIPS可执行代码
X86汇编
汇编器
X86可执行代码

代码生成器的输入

- 源程序的中间表示
  - 线性表示（波兰式）
  - 三地址码（四元式）
  - 栈式中间代码（P-CODE/Java Bytecode）
  - 图形表示（如AST、DAG）
- 符号表信息

代码生成器对输入的要求

- 编译器前端已经将源程序扫描、分析和翻译成足够详细的**中间表示**。
- 中间语言中的**标识符**表示为目标机器能够直接操作的变量（位、整数、浮点数、指针等）。
- 完成了必要的类型检查，类型转换/检测操作已经加入到中间语言的必要位置。
- **完成语法和必要的语义检查**，代码生成器可以认为输入中没有语法或语义错误。

目标程序的种类

- 汇编语言
  - 生成宏汇编代码，再由汇编程序进行编译、连接，从而生成最终代码（.S/.ASM文件）
- 包含绝对地址的机器语言
  - 执行时必须被载入到地址空间中（相对）固定的位置
  - EXE (MS-WIN)、COM (MS-WIN)、A.OUT (Linux)
- 可重定位的机器语言
  - 一组可重定位的模块/子程序可以用连接器装配后生成最终的目标程序（.obj/.o文件组）
  - 可动态加载的模块/子程序（DLL/.SO动态连接库）

面向特定的目标体系结构生成目标代码

- **目标体系结构**可以是：
  - 某种微处理器，如X86、MIPS、ARM等
  - 某种被精心设计和定义的虚拟机或运行时系统，如Java虚拟机、C#运行时系统、P-code虚拟机等。

- **虚拟机：**
  - 第十章介绍的面向P-code虚拟机，采用自顶向下的属性翻译文法生成代码的方法适用于其它虚拟机
  - 虚拟机的代码需要**解释器**解释或者**即时编译器**编译后才能运行

本章内容

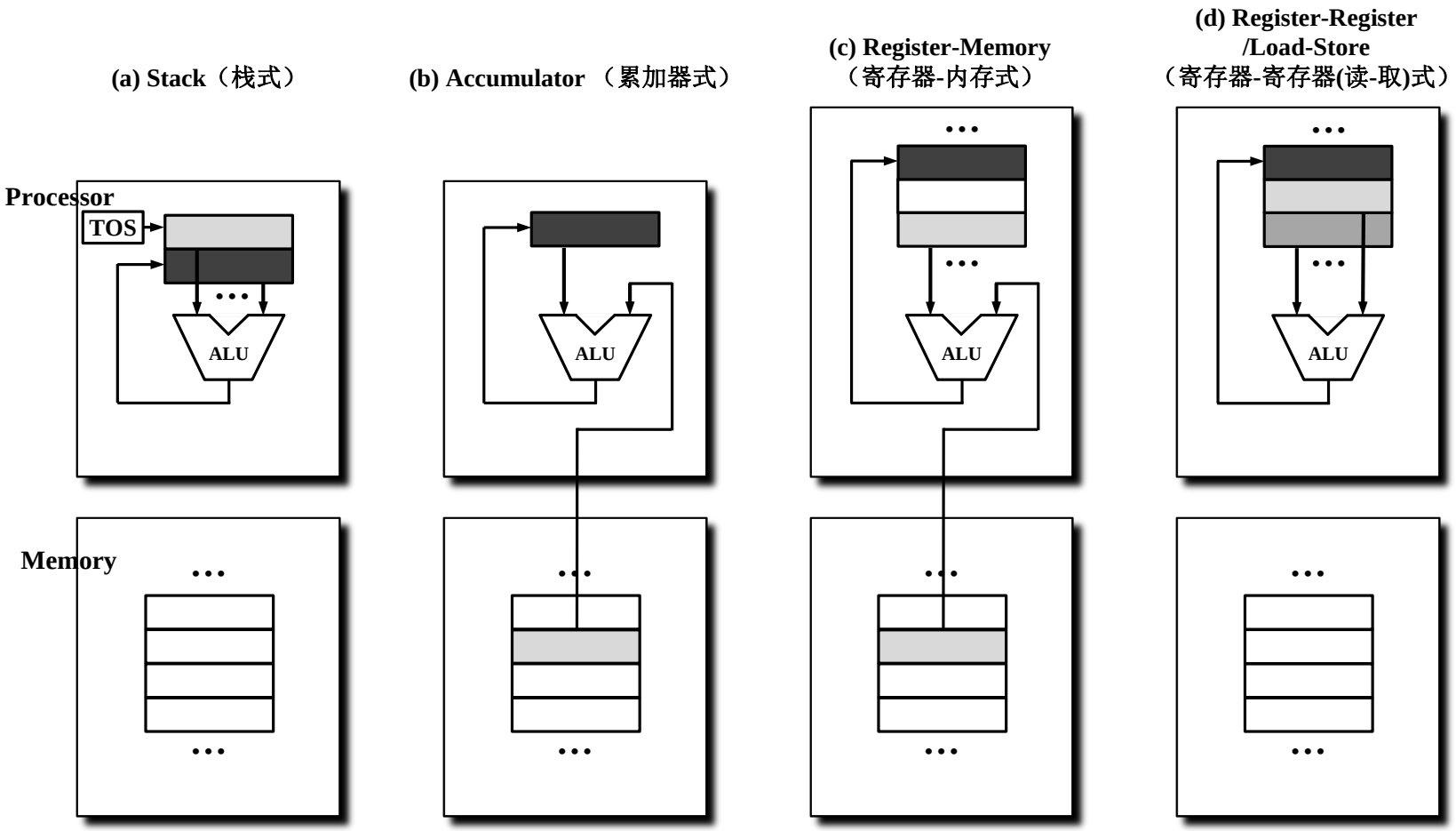
面向微处理器体系结构的代码生成技术

- 主要内容：
  1. 目标代码地址空间的划分，目标体系结构上存贮单元（如寄存器和内存单元）的分配和指派
  2. 从中间代码（或者源代码）到目标代码转换过程中所进行的指令选择
  3. 面向目标体系结构的优化

15.1 现代微处理器体系结构简介

- 指令集
  - Instruction Set
- 流水线和指令级并行
  - Pipeline and Instruction Level Parallelism
- 存储结构和I/O
  - Memory Hierarchy and I/O Systems
- 多处理器和线程级并行
  - Multiprocessor and Thread Level Parallelism

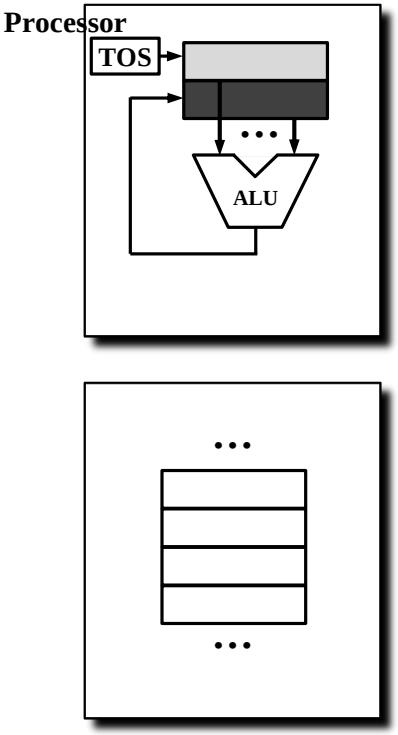
15.1.1 指令集架构



不同：算术逻辑单元ALU对存储单元的访问方式不同

1、栈式指令集架构

(a) Stack（栈式）



类似于P-code和Java虚拟机

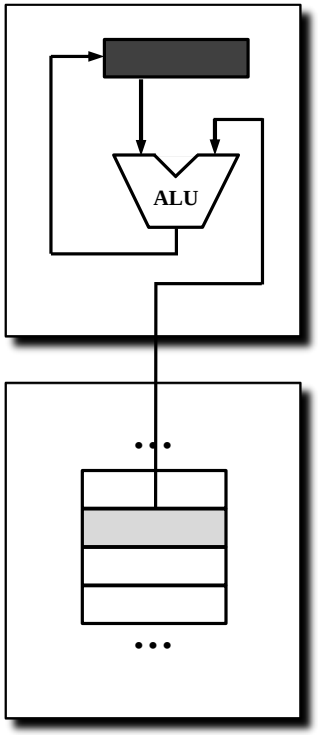
C=A+B

代码：

PUSH A
PUSH B
ADD
POP C

2、累加器式指令集架构

(b) Accumulator （累加器式）



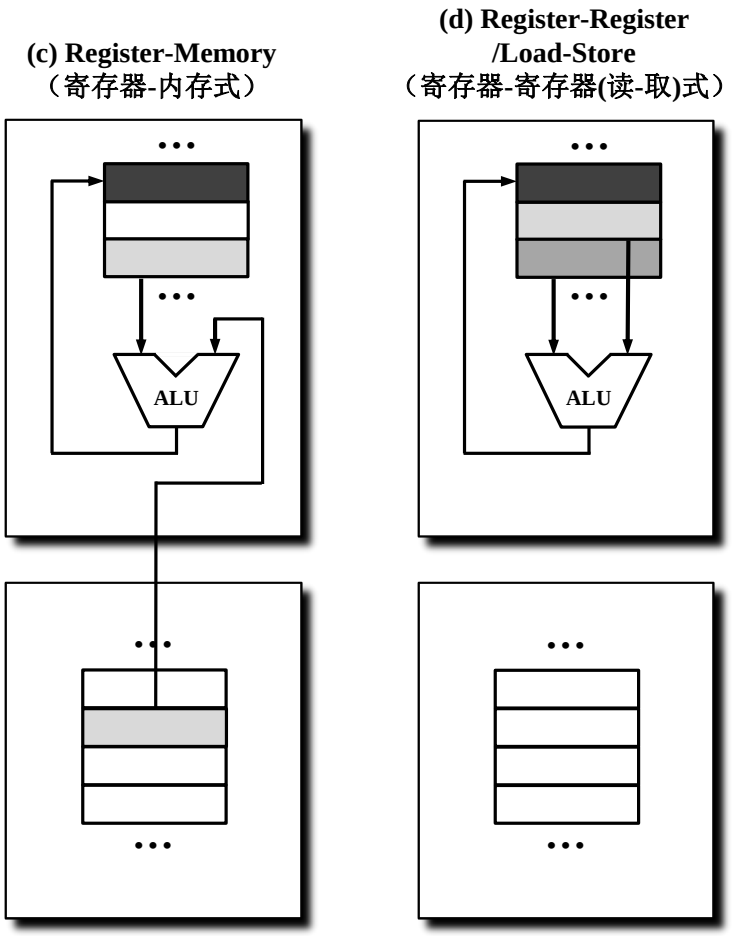
C=A+B

代码：

LOAD A
ADD B
STORE C

代码短了，但开销不一定小。
直接访问内存

3、寄存器架构



- 寄存器-内存指令集架构
- 寄存器-寄存器指令架构

- **区别：** 是否可以直接内存寻址
- **共性：** 内部有多个寄存器可以直接作为ALU指令的**任一**操作数
- 优点：
  - 减少内存访问
  - 减少指令数

计算C=A+B的代码：

栈式	累加器式	寄存器-内存式	寄存器-寄存器式
PUSH A	LOAD A	LOAD R1, A	LOAD R1, A
PUSH B	ADD B	ADD R3, R1, B	LOAD R2, B
ADD	STORE C	STORE R3, C	ADD R3, R1, R2
POP C			STORE R3, C

例：D=(A*B)+(B*C)

栈式架构	寄存器-寄存器架构
PUSH A	LOAD R1, A
PUSH B	LOAD R2, B
MUL	LOAD R3, C
PUSH B	MUL R1, R2, R4
PUSH C	MUL R2, R3, R5
MUL	ADD R4, R5, R5
ADD	STORE R5, D
POP D	
8条指令，5条内存访问	7条指令，4条存储访问

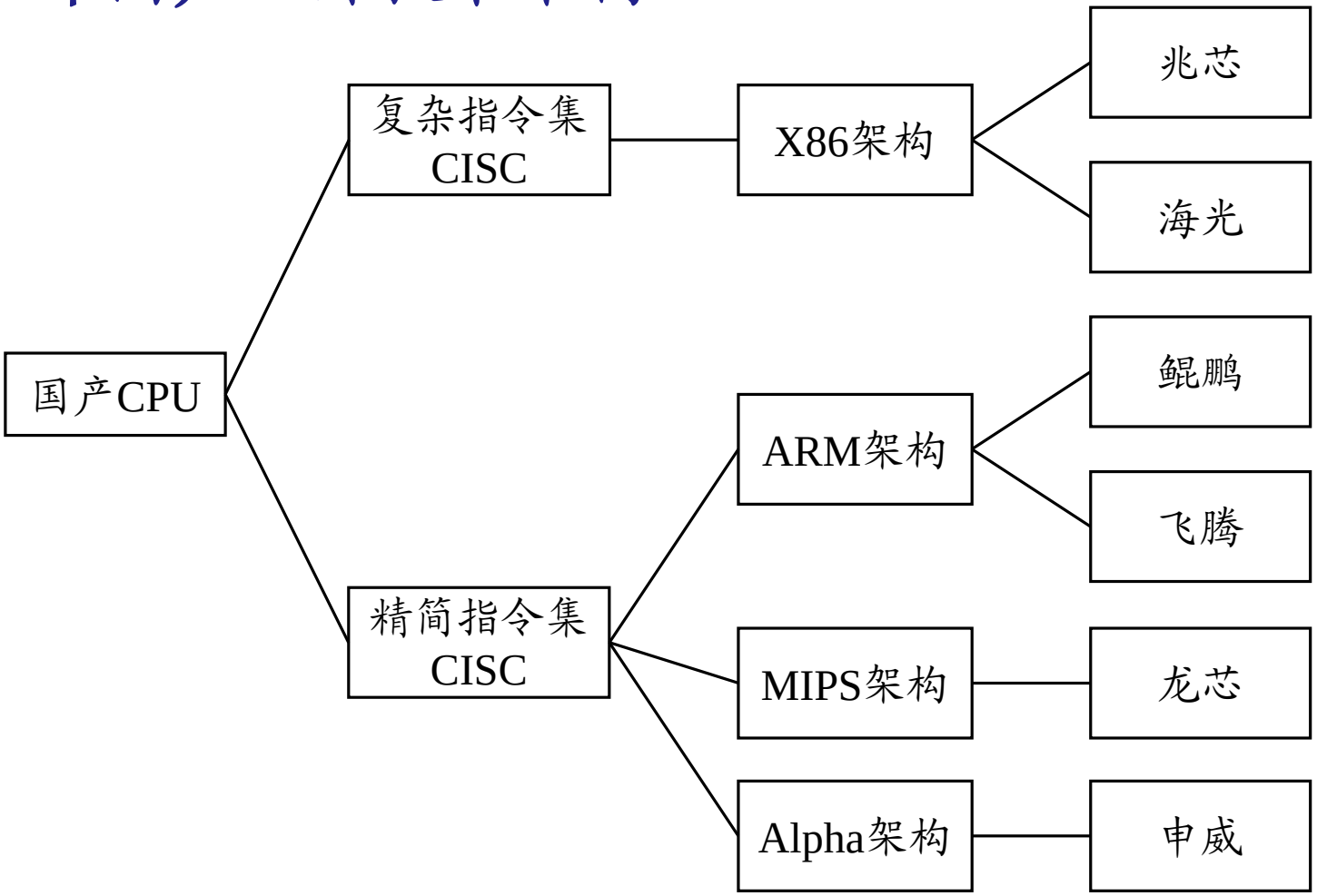
- **寄存器-内存指令集架构处理器**：称为Complex Instruction Set Computers (CISC) 架构计算机

  包括使用最广泛的Intel X86架构处理器、曾经十分辉煌，但现在已经退出历史舞台的DEC VAX系列计算机。

- **寄存器-寄存器指令集架构处理器**：称为Reduced Instruction Set Computer (RISC) 架构计算机

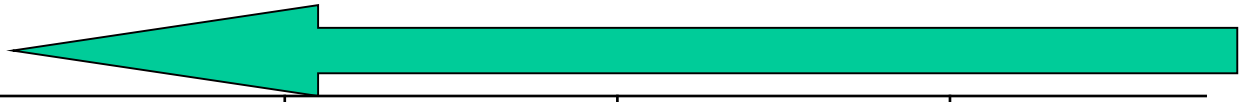
  包括正在广泛使用的Alpha、ARM,、MIPS、 PowerPC、SPARC等微处理器。

2023年国产芯片技术架构



15.1.2 存储层次架构

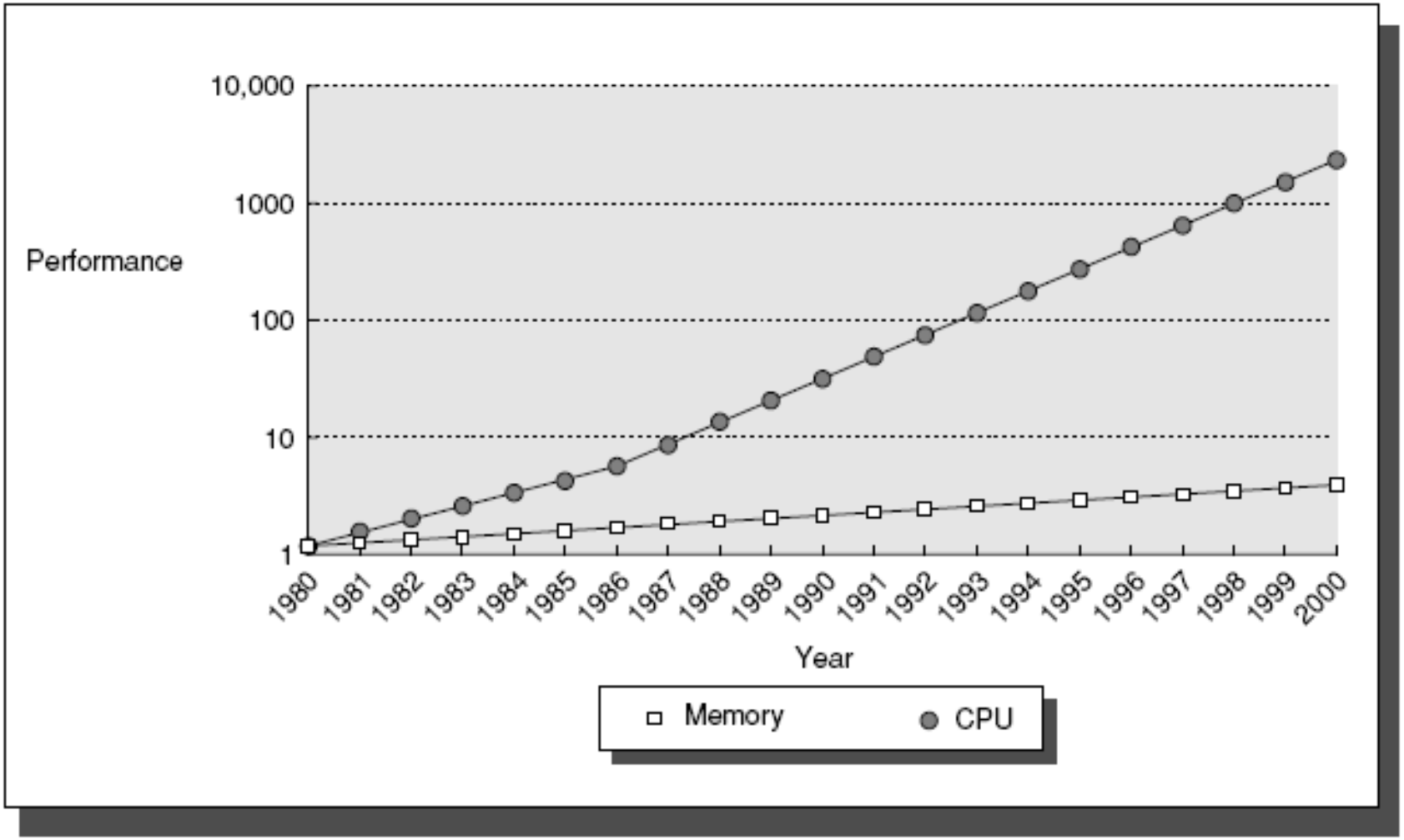
寄存器、缓存、内存、硬盘的存储访问特性



存储层次	1	2	3	4
名称	寄存器	缓存	内存	硬盘
典型容量	< 1 KB	< 16 MB	< 16 GB	> 100 GB
制造技术	CMOS	CMOS SRAM	CMOS DRAM	磁介质
访问时间 (ns)	0.25 - 0.5	0.5 - 25	80 - 250	5,000,000
带宽 (MB/sec)	20,000 - 100,000	5,000 - 10,000	1000 - 5000	20 - 150
管理方式	编译器	硬件	操作系统	操作系统
备份位置	缓存	内存	硬盘	CD 或磁带

寄存器的访问速度基本可以保证处理器在每个时钟周期内访问到需要的数据

1980～2000年，CPU性能和内存访问性能的提高



尽可能少地访问寄存器之外的存储设备!

- **但是，寄存器的数量极其有限**
  - 32位X86微处理器上有8个通用寄存器。
  - XScale/ARM，MIPS：大约16~32个通用寄存器。
  - 分配策略很重要。
- **对缓存的利用，对于大型数据结构有用**
  - 缓存的管理单位是：缓存行（数十或上百字节）。
  - 每次从内存载入的是一组地址连续的数据，而不仅仅是被访问数据。

通过循环交换（Loop Interchange）优化提高缓存命中率

```
for (j = 0; j < 100; j = j+1)
    for (i = 0; i < 5000; i = i+1)
        x[i][j] = 2 * x[i][j];
```

C语言存储：行优先

x[0][0], x[0][1], …., x[0][99], x[1][0], x[1][1], …., x[1][99], …, x[4999][0], x[4999][1], …., x[4999][99]

```
for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
        x[i][j] = 2 * x[i][j];
```

x[0][0], x[0][1], …., x[0][99], x[1][0], x[1][1], …., x[1][99], …, x[4999][0], x[4999][1], …., x[4999][99]

Caches and programs

```
double A[MAX][MAX], x[MAX],
. . .
/* Initialize A and x, assig
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
      for(j = 0; j < MAX; j++)
            y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
      for (i = 0; i < MAX; i++)
            y[i] += A[i][j]*x[j];
```

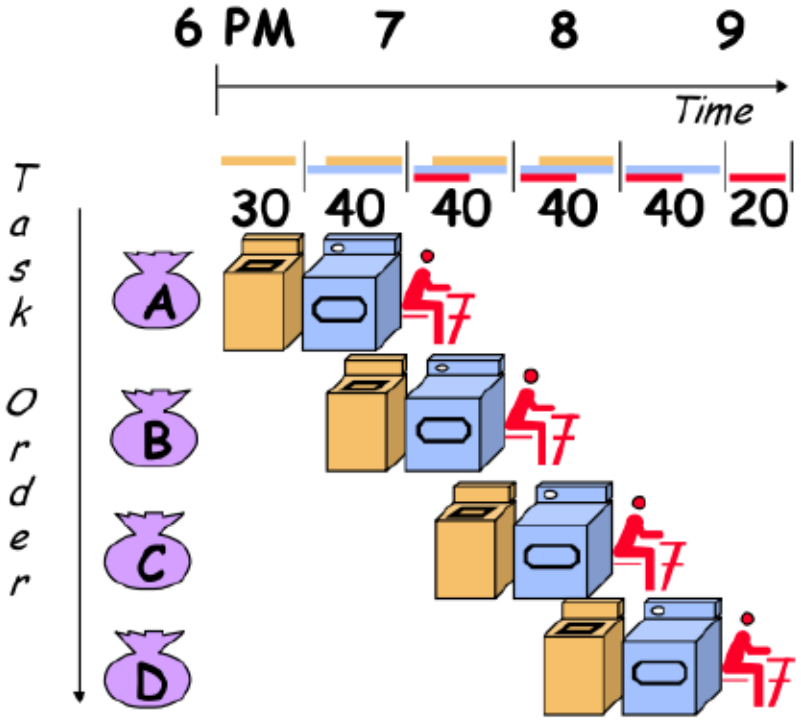
Cache line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Pipelining

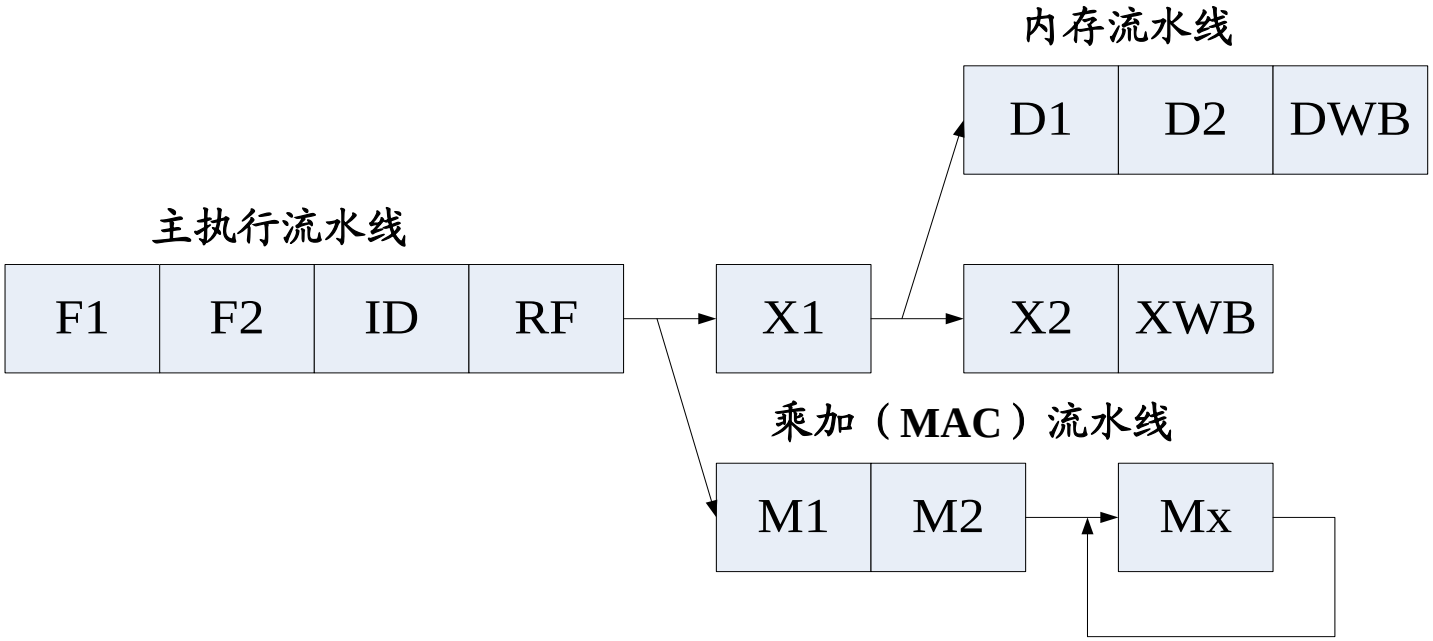
Dave Patterson's Laundry example: 4 people doing laundry
wash (30 min) + dry (40 min) + fold (20 min) = 90 min Latency

- In this example:
  - Sequential execution takes 4 * 90min = 6 hours
  - Pipelined execution takes 30+4*40+20 = 3.5 hours
- Bandwidth = loads/hour
  - BW = 4/6 Non-pipelining
  - BW = 4/3.5 pipelining
  - BW <= 1.5 pipelining, when more total loads
- Pipelining helps bandwidth but not latency (90 min)
- Bandwidth limited by slowest pipeline stage
- Potential speedup = Number of pipe stages



15.1.3 流水线

Intel XScale core RISC 超流水线



F1/F2：2级指令提取；　ID：指令译码；　RF：寄存器文件/操作数移位
X1：ALU执行　X2：状态执行；　XWB：写回

15.2 地址空间

- 代码区
  - 存放目标代码
- 静态数据区
  - 全局变量
  - 静态变量
  - 部分常量，例如字符串
- 动态内存区
  - 也被称为内存堆Heap
  - 程序员管理：C、C++
  - 自动管理（内存垃圾收集器）：Java、Ada
- 程序运行栈
  - 活动记录
  - 函数调用的上下文现场
    - 由调用方保存的一些临时寄存器
    - 被调用方保存的一些全局寄存器

15.2.1 程序地址空间的实例分析

```
1   // C12P1.cpp
2   #include "stdafx.h"
3   int global_val = 0 ;
4   int foo(int n){
5       static int static_val = 0 ;
6       int i ;
7       for(i=0 ; i<100; i++){
8           n=n*i+global_val+static_val;
9       }
10      static_val = n / 2;
11      printf("static_val is %x\n", static_val) ;
12      return n ;
13  }
14  int main(int argc, char* argv[]){
15      global_val = 99 ;
16      int n = foo(100) ;
17      printf("foo(100) is %x\n", n);
18      return 0;
19  }
```

```
1   TITLE   C12P1.cpp
2   .386P
3   .model FLAT
4
5   PUBLIC  ?global_val@@3HA                    ; global_val
6   _BSS    SEGMENT
7   ?global_val@@3HA DD 01H DUP (?)             ; global_val
8   _?static_val@?1??foo@@YAHH@Z@4HA DD 01H DUP (?)
9   _BSS    ENDS
10  PUBLIC  ?foo@@YAHH@Z                        ; foo
11  PUBLIC  ??_C@_0BC@BPOF@static_val?5is?5?$CFd?6?$AA@    ; `string'
12  EXTRN   _printf:NEAR
13  _DATA   SEGMENT
14  ??_C@_0BC@BPOF@static_val?5is?5?$CFd?6?$AA@ DB 'static_val is %x',
    0aH, 00H ; `string'
15  _DATA   ENDS
16  _TEXT   SEGMENT
17  _n$ = 8
18  ?foo@@YAHH@Z PROC NEAR                      ; foo
19  …; foo的函数体被省略
20  ?foo@@YAHH@Z ENDP                           ; foo
21  _TEXT   ENDS
22
23  PUBLIC  _main
24  PUBLIC  ??_C@_0BA@OOBN@foo?$CI100?$CJ?5is?5?$CFd?6?$AA@; `string'
25  _DATA   SEGMENT
26  ??_C@_0BA@OOBN@foo?$CI100?$CJ?5is?5?$CFd?6?$AA@ DB 'foo(100) is
    %x', 0aH, 00H ; `string'
27  _DATA   ENDS
28  _TEXT   SEGMENT
29  _main   PROC NEAR                           ;
30  …; main的函数体被省略
31  _main   ENDP
32  _TEXT   ENDS
33  END
```

全局变量和静态变量

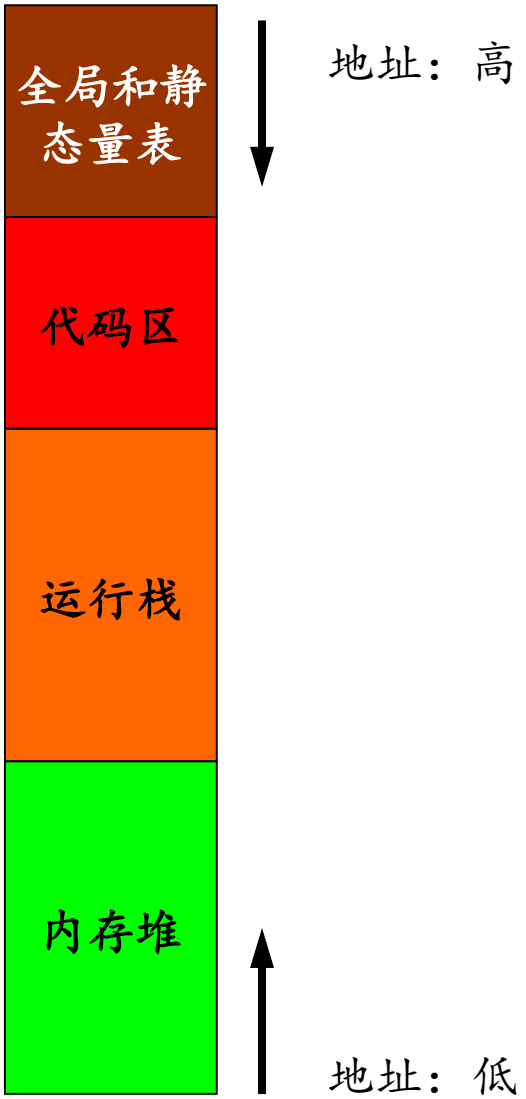
字符串常量

代码区

字符串常量

- 以MS-Win下的应用程序为例，从高地址到低地址，自上而下的是：
  - 静态数据区
    - 全局和静态量表
  - 代码区
  - 程序运行栈
  - 动态内存区
    - 内存堆

栈是向下生长



15.2.2 程序运行栈的设计

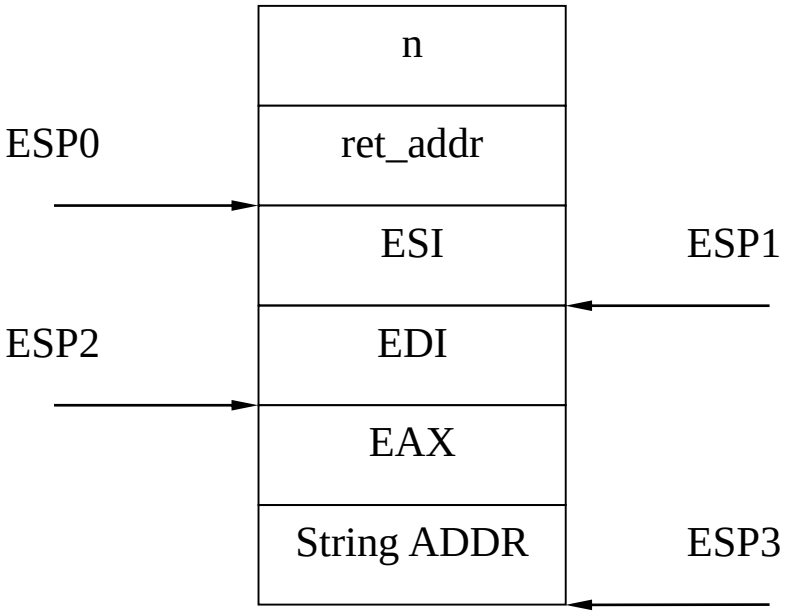
- **子程序/函数**运行时所需的基本空间：活动记录
- **进入**子程序/函数时分配，地址空间向下生长（从高地址到低地址）
- 从子程序/函数**返回时**，当前运行栈将被废弃
- 递归调用的同一个子程序/函数，每次调用都将获得**独立的**运行栈空间，从而保证了递归程序和多线程程序的正确运行。

```
1  // C12P1.cpp
2  #include "stdafx.h"
3  int global_val = 0 ;
4  int foo(int n){
5     static int static_val = 0 ;
6     int i ;
7     for(i=0 ; i<100; i++){
8        n = n * i + global_val + static_val;
9     }
10    static_val = n / 2;
11    printf("static_val is %x\n", static_val) ;
12    return n ;
13 }
14 int main(int argc, char* argv[]){
15    global_val = 99 ;
16    int n = foo(100) ;
17    printf("foo(100) is %x\n",n);
18    return 0;
19 }
```

```
1  PUBLIC     ?foo@@YAHH@Z                                  ; foo
2  PUBLIC     ??_C@_0BC@BPOF@static_val?5is?5?$CFd?6?$AA@   ; `string'
3  EXTRN      _printf:NEAR
4  _DATA      SEGMENT
5  ??_C@_0BC@BPOF@static_val?5is?5?$CFd?6?$AA@ DB 'static_val is %x',
   0aH, 00H                    ; 'string'
6  _DATA      ENDS
7  _TEXT      SEGMENT
8  _n$ = 8
9  ?foo@@YAHH@Z PROC NEAR                                   ; foo
10    mov     ecx, DWORD PTR ?global_val@@3HA                ; global_val
11    mov     edx, DWORD PTR _?static_val@?1??foo@@YAHH@Z@4HA
12    push    esi
13    mov     esi, DWORD PTR _n$[esp]
14    push    edi
15    xor     eax, eax
16 $L533:
17    mov     edi, eax
18    imul    edi, esi
19    add     edi, edx
20    add     edi, ecx
21    inc     eax
22    cmp     eax, 100          ; 00000064H
23    mov     esi, edi
24    jl      SHORT $L533
25    mov     eax, esi
26    cdq
27    sub     eax, edx
28    sar     eax, 1
29    push    eax
30    push    OFFSET FLAT:??_C@_0BC@BPOF@static_val?5is?5?$CFd?6?$AA@ ; `string'
31    mov     DWORD PTR _?static_val@?1??foo@@YAHH@Z@4HA, eax
32    call    _printf
33    add     esp, 8
34    mov     eax, esi
35    pop     edi
36    pop     esi
37    ret     0
38 ?foo@@YAHH@Z ENDP                        ; foo
39 _TEXT      ENDS
```

; foo函数代码

函数foo的运行栈示例



X86共有8个通用寄存器：

EAX: 局部

EBX: 全局

ECX: 局部

EDX: 局部

ESI: 全局，通常在内存操作指令中作为“源地址指针”使用。

EDI: 全局，通常在内存操作指令中作为“目的地址指针”使用。

ESP: 栈指针

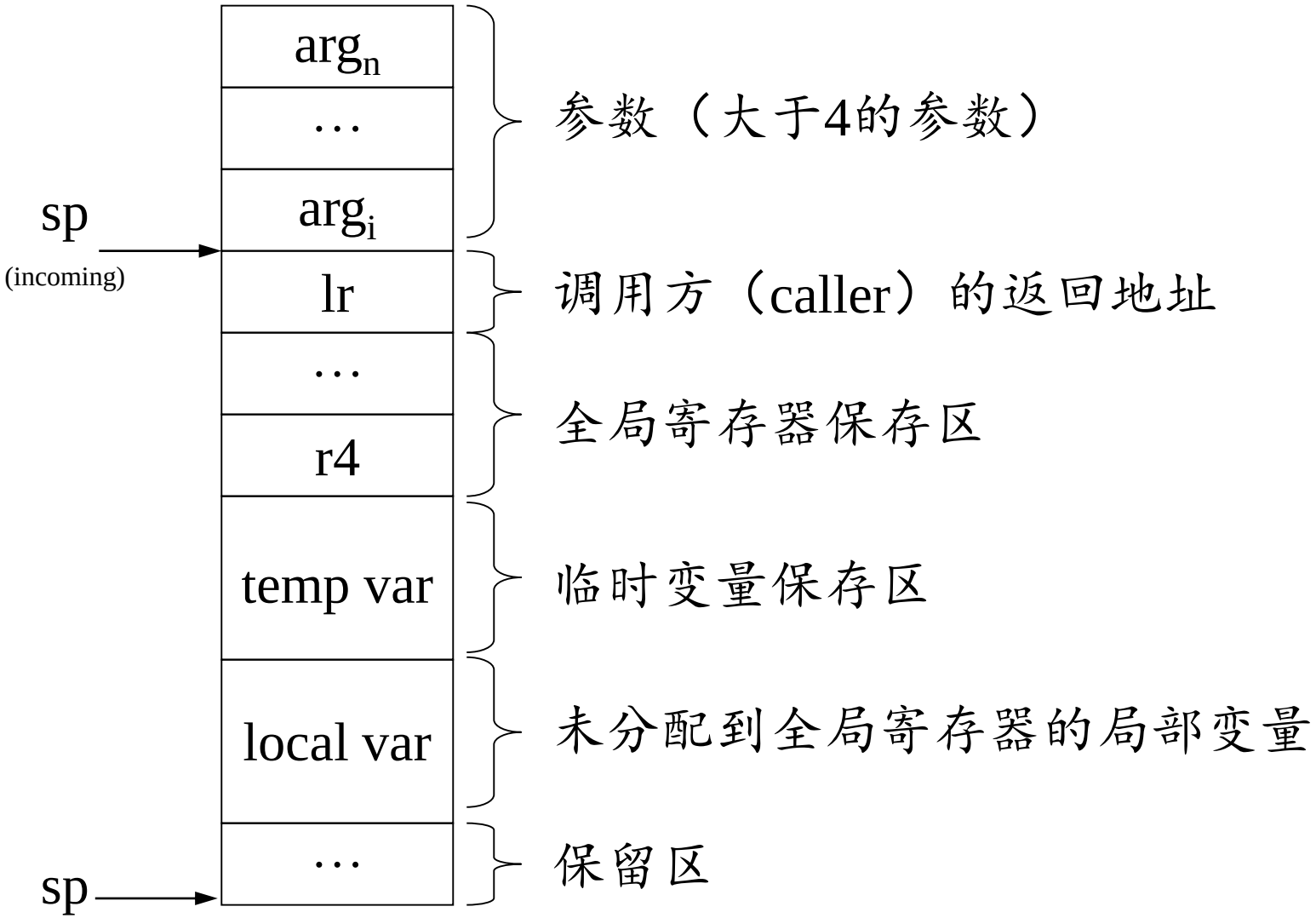
EBP: 当前函数基地址

一个典型的运行栈包括（活动记录）

- 函数的返回地址
- 全局寄存器的保存区
- 临时变量的保存区
- 未分配到全局寄存器的局部变量的保存区
- 其他辅助信息的保存区

  例，PASCAL/PL-I类语言的DISPLAY区

一个典型的RISC架构上的运行栈



15.3 寄存器的分配和指派

- 为什么要分配和管理寄存器？
  - 寄存器的访问速度是所有存储形式中最快的
  - 某些运算只能发生在寄存器当中
  - 从程序优化的角度来说，我们希望所有指令的执行都仅在寄存器中完成
  - 而资源是有限的

- 寄存器通常分为
  - 通用寄存器
    - X86：EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, etc
    - ARM: R0~R15, etc
  - 专用寄存器
    - X86：浮点寄存器栈, etc
- 通用寄存器
  - 保留寄存器
    - 例如，X86的ESP栈指针寄存器，ARM的返回寄存器LR
  - 调用方保存的寄存器——临时寄存器
    - caller-saved register
    - X86：EAX, ECX, EDX
    - ARM：R0~R3, R12, LR
  - 被调用方保存的寄存器——全局寄存器
    - callee-saved register
    - X86：EBX, ESI, EDI, EBP （ESP为运行栈寄存器，不参与寄存器分配）
    - ARM：R4~R11

15.3.1 全局寄存器分配

- 全局寄存器分配
  - “全局”相对于“基本块”而言，不是“程序全局”。
  - 全局寄存器分配的对象主要是函数的局部变量，包括函数入口参数。
- 分配原则

寄存器专属于线程！

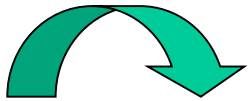
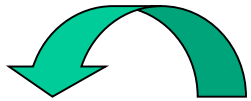
  - 优先分配给跨基本块仍然活跃的变量，尤其是循环体内最活跃的变量。
  - 局部变量参与全局寄存器分配。
  - 为了线程安全，全局变量/静态变量一般不参与全局寄存器分配。

如果全局/静态量参与寄存器分配？

全局变量和静态变量一般不参与全局寄存器分配，即使他们在某个循环体中被多次访问。

- 这是因为当一个全局变量或静态变量的值保存在于寄存器中时，如果发生线程切换，当前的寄存器状态将作为线程现场被保存，切入线程将恢复其此前保存的寄存器状态，这就导致了其他线程无法得到该寄存器在此前线程中的值，程序运行可能会发生不可预知的错误。

如果全局/静态量参与寄存器分配？

Thread 1　　Thread 2

第一次被调用

a = 2

s_c = 1

s_c = 3

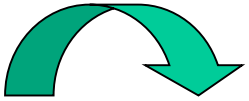
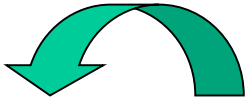
第一次被调用

a = 1

s_c = 1

```
void foo(int a)
{
    static int s_c = 0 ;
    s_c += a ;
}
```

如果s_c被分配给全局寄存器ESI

Thread 1　　Thread 2

Context switch!

第一次被调用

a = 2

s_c (ESI) = 0

s_c (ESI) = 2

第一次被调用

a = 1

s_c (ESI) = 1

```
void foo(int a)
{
    static int s_c = 0 ;
    s_c += a ;
}
```

常用全局寄存器分配方法

- 引用计数
  - 通过统计变量在函数内被引用的次数，并根据被引用的特点赋予不同的权重，最终为每个变量计算出一个唯一的权值，根据权值的大小排序，将全局寄存器依次分配给权值最大的变量
- 着色图算法
  - 通过构建变量之间的冲突图，在图上应用着色算法，将不同的全局寄存器分配给有冲突的变量。

一、引用计数

- **原则**：如果一个局部变量被访问的次数较多，那么它获得全局寄存器的机会也较大
- **注意**：出现在循环，尤其是内层嵌套循环中的变量的被访问次数应该得到一定的加权

如权值为10。则：

a 30次

b 3次

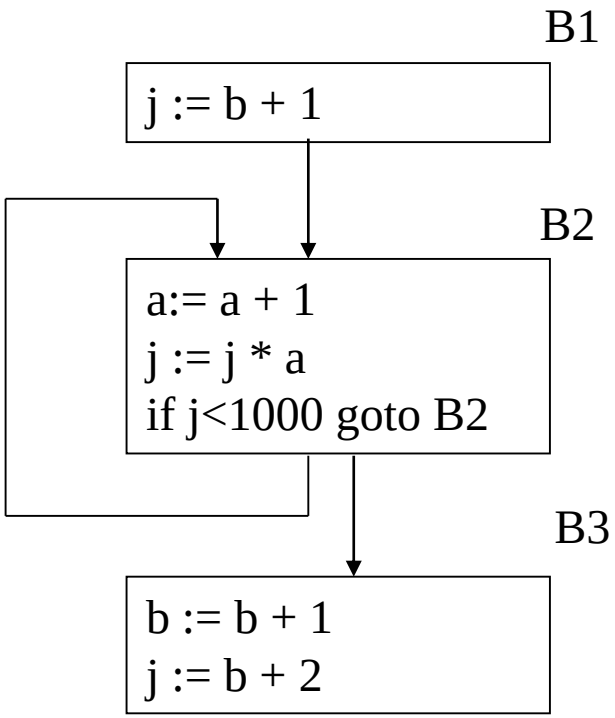
j 32次

3个局部变量，2个全局寄存器可供分配，谁将获得寄存器？

a 3次

b 4次

j 5次



引用计数

- **分配算法：**如果有N个全局寄存器可供分配，则前N个变量拥有全局寄存器，其余变量在程序运行栈（活动记录）分配存贮单元
- **问题：**不再使用的变量不能及时释放寄存器
  - 如变量a 在前期大量使用，后端程序中不使用了
- 解决办法：
  - 活跃变量分析、冲突图
  - 着色算法

二、图着色算法

一种简化的图着色算法

步骤：

1. 通过数据流分析，构建变量的冲突图

2. 如果可供分配k个全局寄存器，那么我们就尝试用k种颜色给该冲突图着色

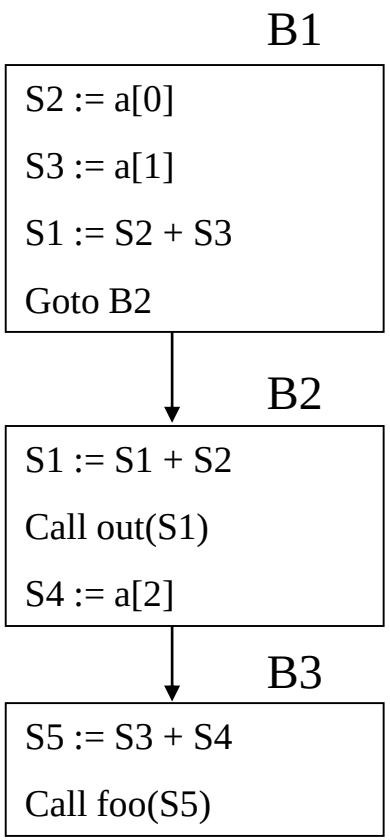
步骤1、通过数据流分析，构建变量的冲突图

- 什么是变量的冲突图？
  - 它的结点是待分配全局寄存器的变量
  - 当两个变量中的一个变量在另一个变量定义（赋值）处是活跃的，它们之间便有一条边连接。所谓变量i在代码n处活跃，是指程序运行时变量i在n处拥有的值，在从n出发的某条路径上会被使用（活跃变量分析）。
  - 直观的理解，可以认为有边相连的变量，它们无法共用一个全局寄存器，或者同一存贮单元，否则程序运行将可能出错
  - 无连接关系的变量，即便它们占用同一全局寄存器，或同一存贮单元，程序运行也不会出错

- 例:

```
for 每个基本块B do in[B] = ∅;
while 集合in发生变化 do
    for 每个基本块B do begin
        out[B] = ∪ B的所有后继S in[S]
        in[B] = use[B] ∪(out[B] – def[B])
    end
```

流图

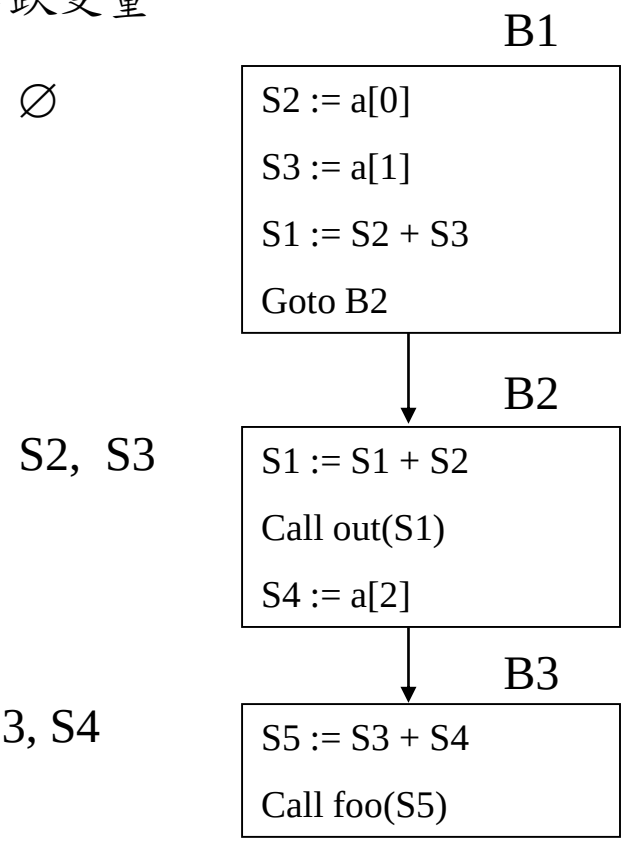


	def[B]	use[B]	in[B]	out[B]
B1	S1,S2,S3	∅	∅	S1,S2,S3
B2	S4	S1,S2	S1,S2,S3	S3,S4
B3	S5	S3,S4	S3,S4	∅
			∅	∅

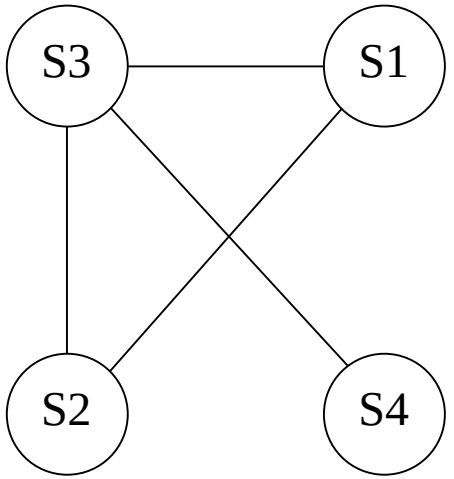
例:

基本块入口处的活跃变量

流图

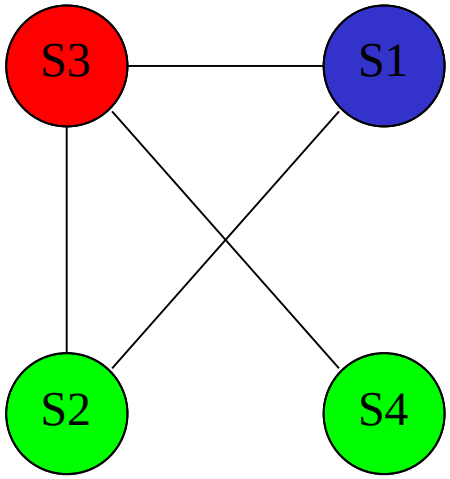


冲突图

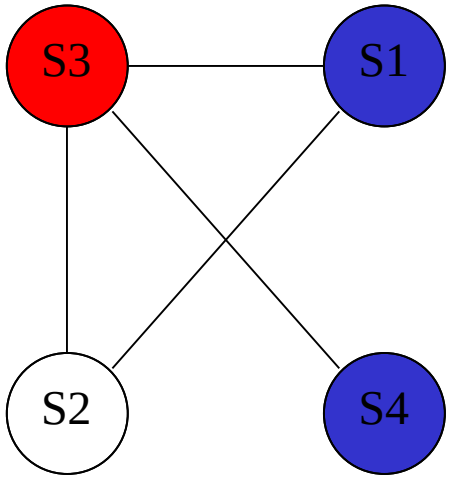


步骤2、如果可供分配k个全局寄存器，那么我们就尝试用k种颜色给该冲突图着色

假设1：k=3，R0, R1, R2



假设2：k=2，R0, R1



S2 := a[0]

S3 := a[1]

S1 := S2 + S3

Goto B2

S1 := S1 + S2
Call out(S1)
S4 := a[2]

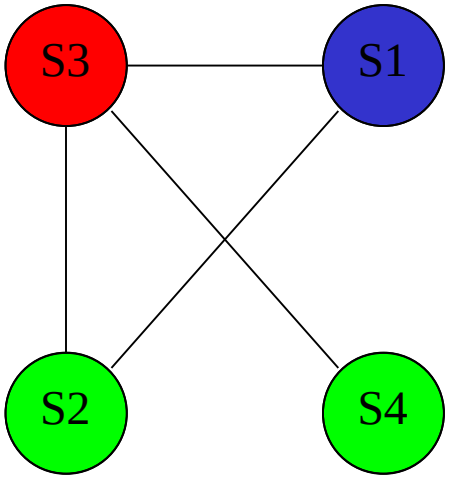
S5 := S3 + S4

Call foo(S5)

S3, **S1**, **S2**

S4

假设1：k=3，R0, R1, R2



R2 := a[0]

R0 := a[1]

R1 := **R2** + **R0**

Goto B2

R1 := **R1** + **R2**

Call out(**R1**)

R2 := a[2]

S5 := **R0** + **R2**

Call foo (**S5**)

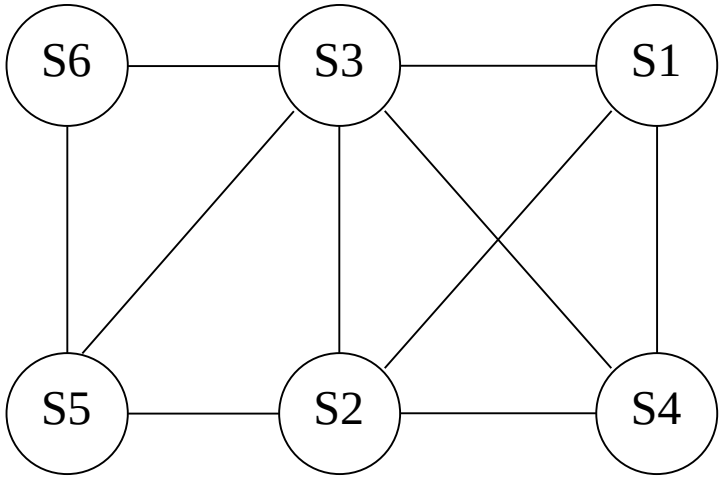
算法15.2 一种启发式图着色算法

冲突图G

- 寄存器数目为K

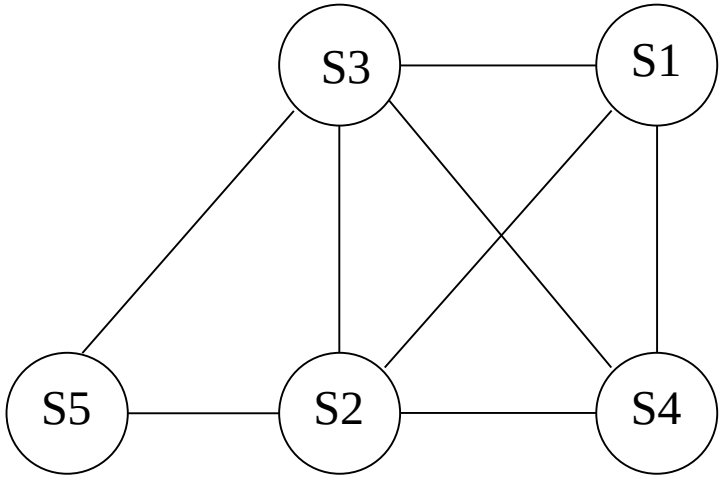
  假设 K=3

G

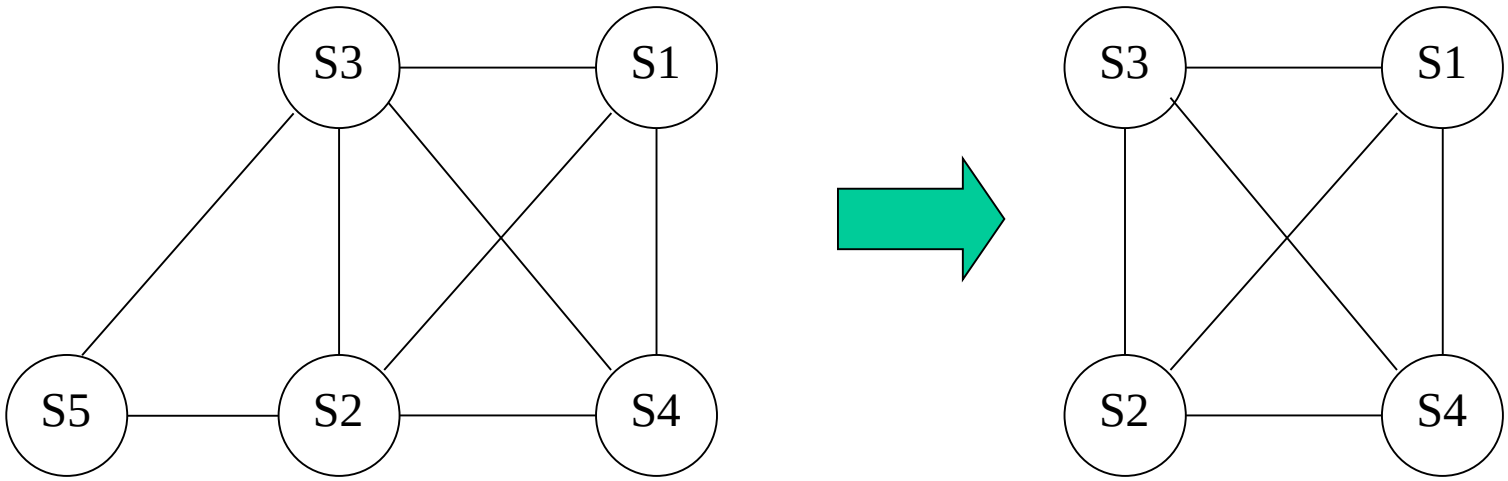


步骤1、找到第一个连接边数目小于K的结点，将它从图G中移走，形成图G’

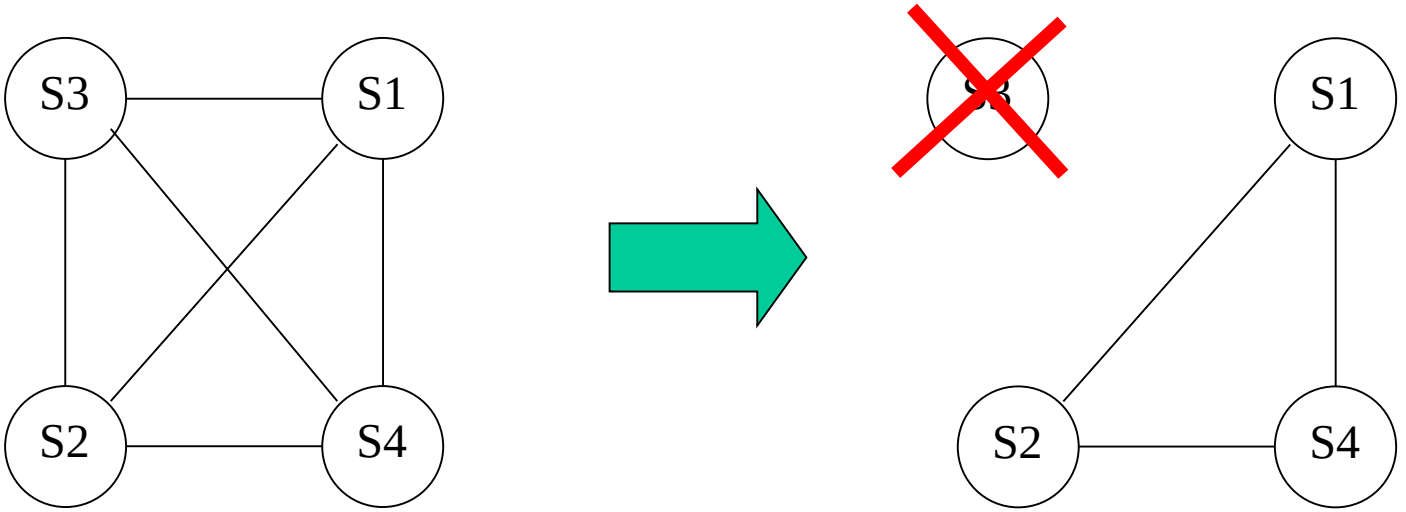
G’



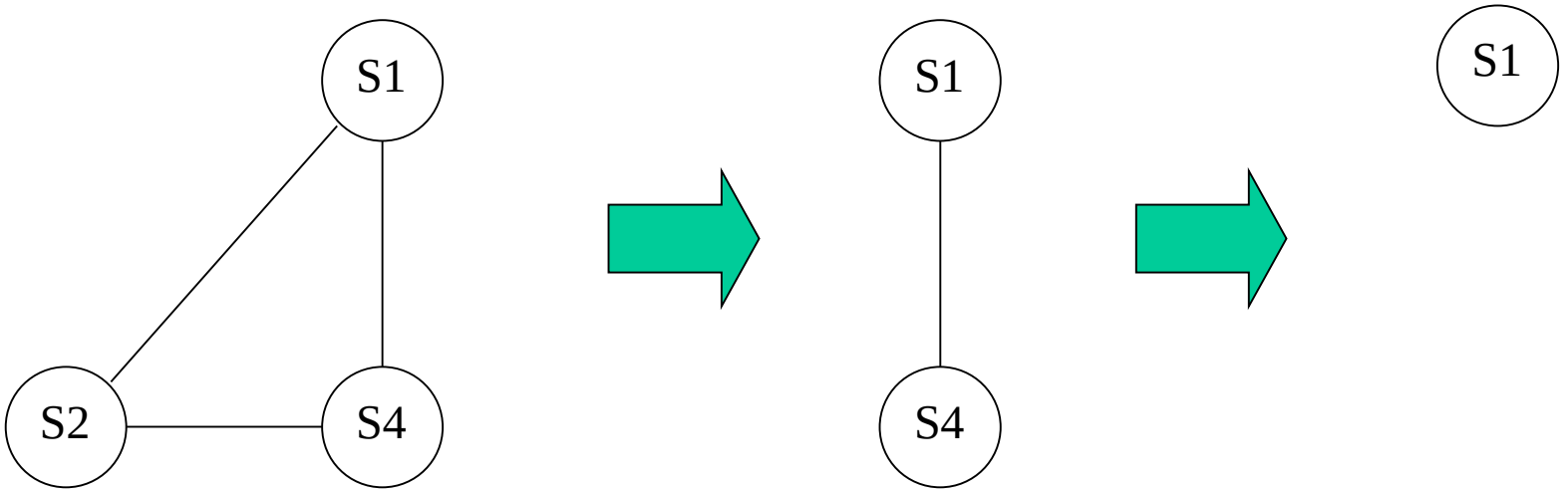
步骤2、重复步骤1，直到无法再从G'中移走结点



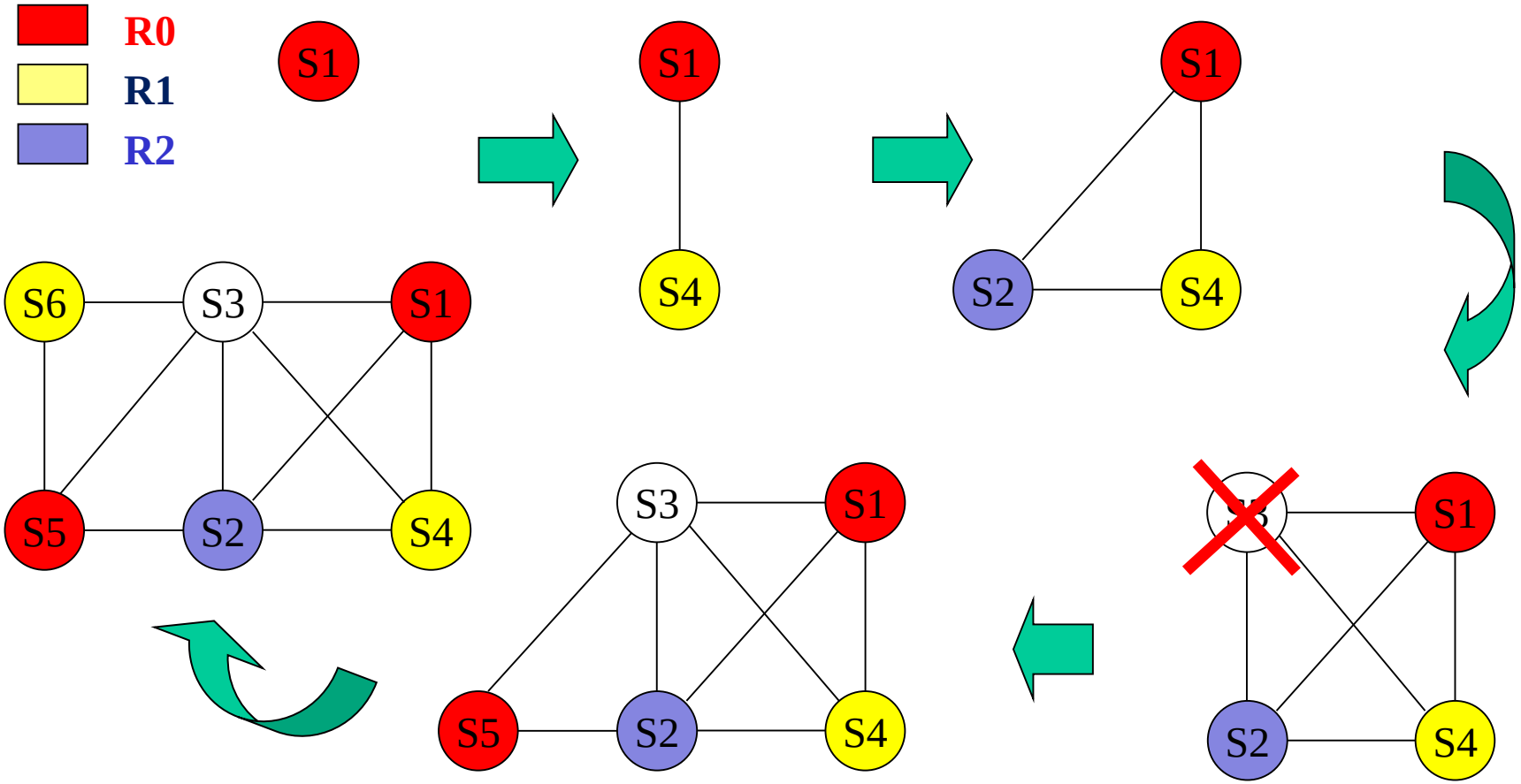
步骤3、在图中选取**适当**的结点，将它记录为“不分配全局寄存器”的结点，并从图中移走



步骤4、重复步骤1~步骤3，直到图中仅剩余1个结点



步骤5、给剩余的最后一个结点选取一种颜色，然后按照结点被移走的顺序，反向将结点和边添加进去，并依次给新加入的结点选取颜色。（保证有链接边的结点着不同的颜色）



15.3.2 临时寄存器分配

- 为什么在代码生成过程中，需要对临时寄存器进行管理？
  - 因为生成某些指令时，必须使用指定寄存器
  - 临时寄存器中保存有此前的计算中间结果

- 以X86为例，生成代码时可用的临时寄存器
  - EAX, ECX, EDX等

临时寄存器的管理原则和方法

- 临时寄存器的生存范围
  - 不超越基本块
  - 不跨越函数调用
- 临时寄存器的管理方法
  - 寄存器池

临时寄存器池的使用方法：设EAX, EDX可用

全局寄存器分配结果：

a	EBX
b	ESI
c	EDI

临时变量在运行栈上的保存地址：

t3	ESP+10H
t2	ESP+0CH
t1	ESP+08H

寄存器池：

t1 := -c

t2 := t1 - b

t3 := t2 + t2

a := t3

t1	EAX
	EDX

mov EAX, EDI

neg EAX

t1	EAX
t2	EDX

mov EDX, EAX

sub EDX, ESI

t3	**EAX**
t2	EDX

mov [ESP+08H], EAX

mov EAX, EDX

add EAX, EAX

t3	EAX
t2	EDX

mov EBX, EAX

临时寄存器的分配和管理方法

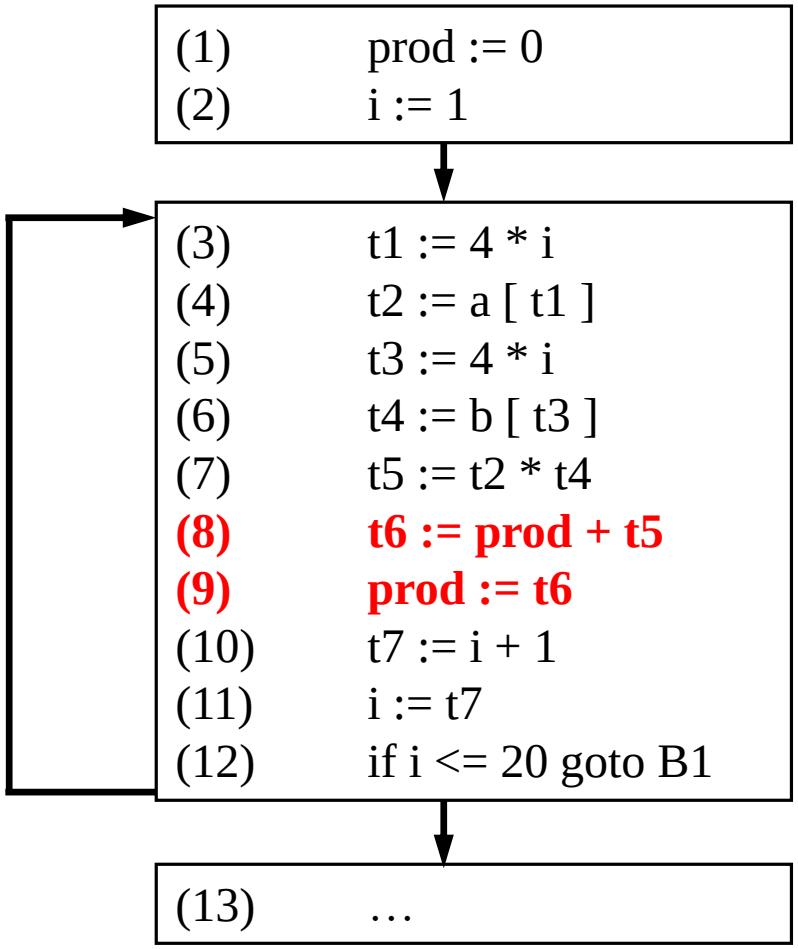
- 进入基本块时，清空临时寄存器池
- 为当前中间代码生成目标代码时，无论临时变量还是局部变量（亦或全局变量和静态变量），如需使用临时寄存器，都可以向临时寄存器池申请
- 临时寄存器池接到申请后
  - 如寄存器池中有空闲寄存器，则可将该寄存器标识为被该申请变量占用，并返回该空闲寄存器
  - 如寄存器池中没有空闲寄存器，则选取一个在即将生成代码中不会被使用的寄存器写回相应的内存空间，标识该寄存器被新的变量占用，返回该寄存器
- 在基本块结尾，或者函数调用发生前，将寄存器池中所有被占用的临时寄存器写回相应的内存空间，清空临时寄存器池

15.4 指令选择

不同的体系结构采用了不同类型的指令集，由于体系结构和指令集的差异，使得在生成代码时需要采用不同的指令选择策略。

- RISC
  - ARM, MIPS
- CISC
  - X86
- VLIW/EPIC (**V**ery **L**ong **I**nstruction)
  - Itanium (**W**ord**E**xplicitly **P**arallel **I**nstruction **C**omputing)

例：



- RISC: ARM
  - prod = R5
  - t5 = [SP+8]
  - t6 = R2

```
ldr  R3, [SP, #8]      ; R3 = t5
add R5, R2, R3         ; prod = t6 + R3
```

- CISC: X86
  - prod = EBX
  - t5 = [ESP+8]
  - t6 = ECX

```
mov ECX, EBX           ; t6 = prod
add ECX, [ESP+8]       ; t6 = prod + t5
mov EBX, ECX           ; prod = t6
```

例：
(1) t1 = -c
(2) t2 = t1 – b
(3) t3 = t2 + t2
(4) a = t3

a	EBX
b	ESI
c	EDI

t3	ESP+10H
t2	ESP+0CH
t1	ESP+08H

逐句转换：

(1) mov ECX, EDI
neg ECX
mov [ESP+08H], ECX
(2) mov ECX, [ESP+08H]
sub ECX, ESI
mov [ESP+0CH], ECX
(3) mov ECX, [ESP+0CH]
add ECX, [ESP+0CH]
mov [ESP+10H], ECX
(4) mov EBX, [ESP+10H]

逐句转换+临时寄存器池：

(1) mov EAX, EDI
neg EAX
(2) mov EDX, EAX
sub EDX, ESI
(3) mov [ESP+08H], EAX
mov EAX, EDX
add EAX, EAX
(4) mov EBX, EAX

作业：P381：教材第十五章 1, 5

优化部分小结

优化分为两大类

- 与机器无关的优化独立于机器的（中间）代码优化
- 与机器有关的优化目标代码上的优化（与具体机器有关）

优化方法的分类2：

- **局部优化技术**
  - 指在基本块内进行的优化
  - 例如，局部公共子表达式删除
- **全局优化技术**
  - 函数/过程内进行的优化
  - 跨越基本块
  - 例如，全局数据流分析
- **跨函数优化技术**
  - 整个程序
  - 例如，跨函数别名分析，逃逸分析等

第14章 代码优化

- **DAG图**
  - 消除局部公共子表达式
  - 从DAG图导出中间代码的启发式算法
- **数据流分析**
  - 到达定义分析
  - 活跃变量分析
- **构建冲突图**
  - 变量冲突的基本概念
  - 通过活跃变量分析构建（精度不太高的）冲突图

第15章 目标代码生成及优化

- 微处理器体系结构基础知识
- **全局寄存器分配算法**
  - 引用计数
  - **图着色**
- 临时寄存器池管理方法与指令选择