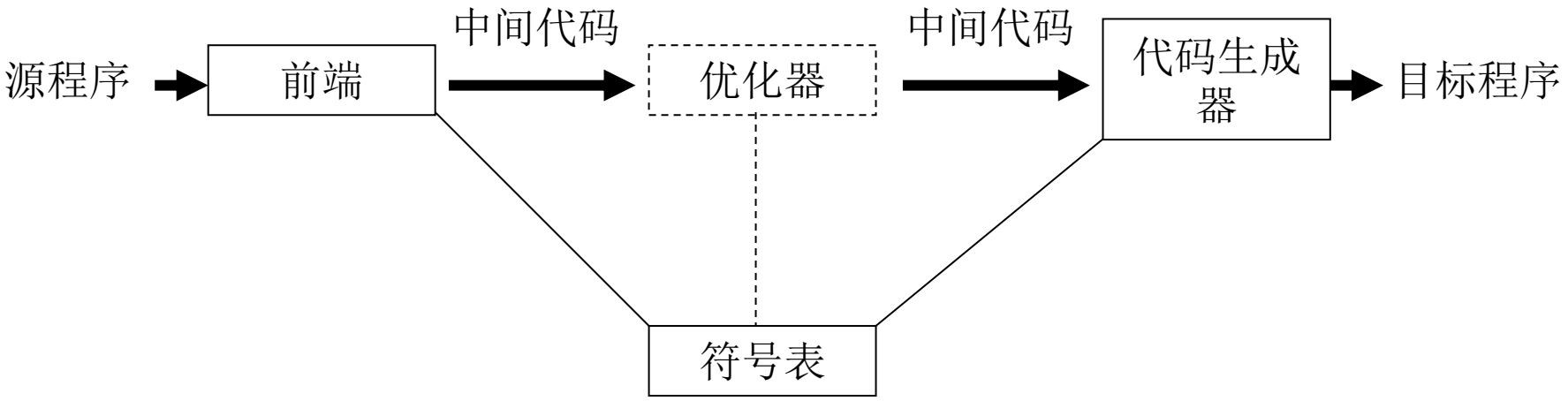
第十二章 目标代码生成

面向目标体系结构的代码生成和优化技术

史晓华
北京航空航天大学

2023.10

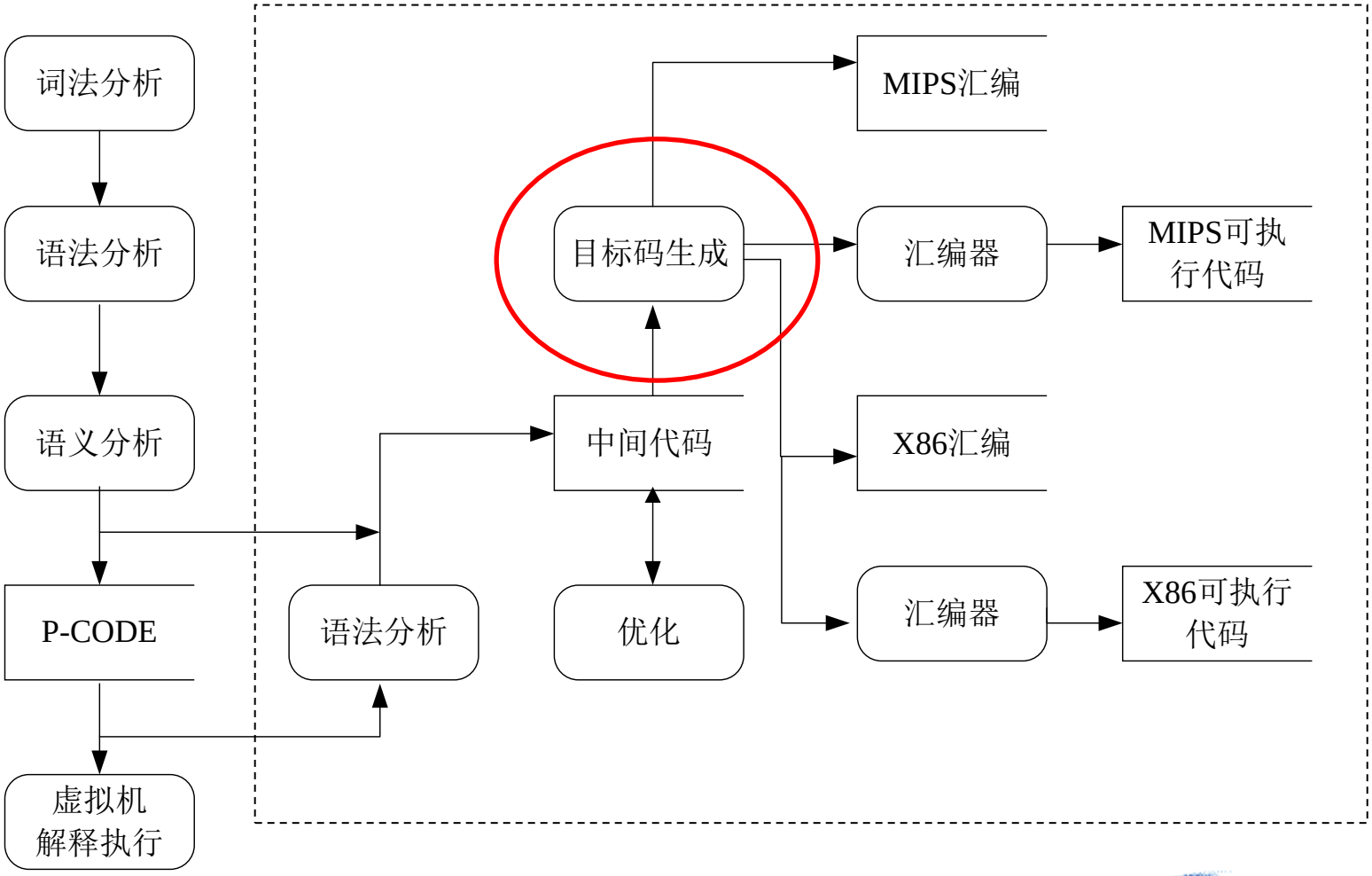
代码生成器在编译系统中的位置



代码生成器的主要任务

- 目标代码地址空间的划分，目标体系结构上存贮单元，例如寄存器和内存单元的分配和指派
- 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
- 面向目标体系结构的优化

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
- 符号表信息

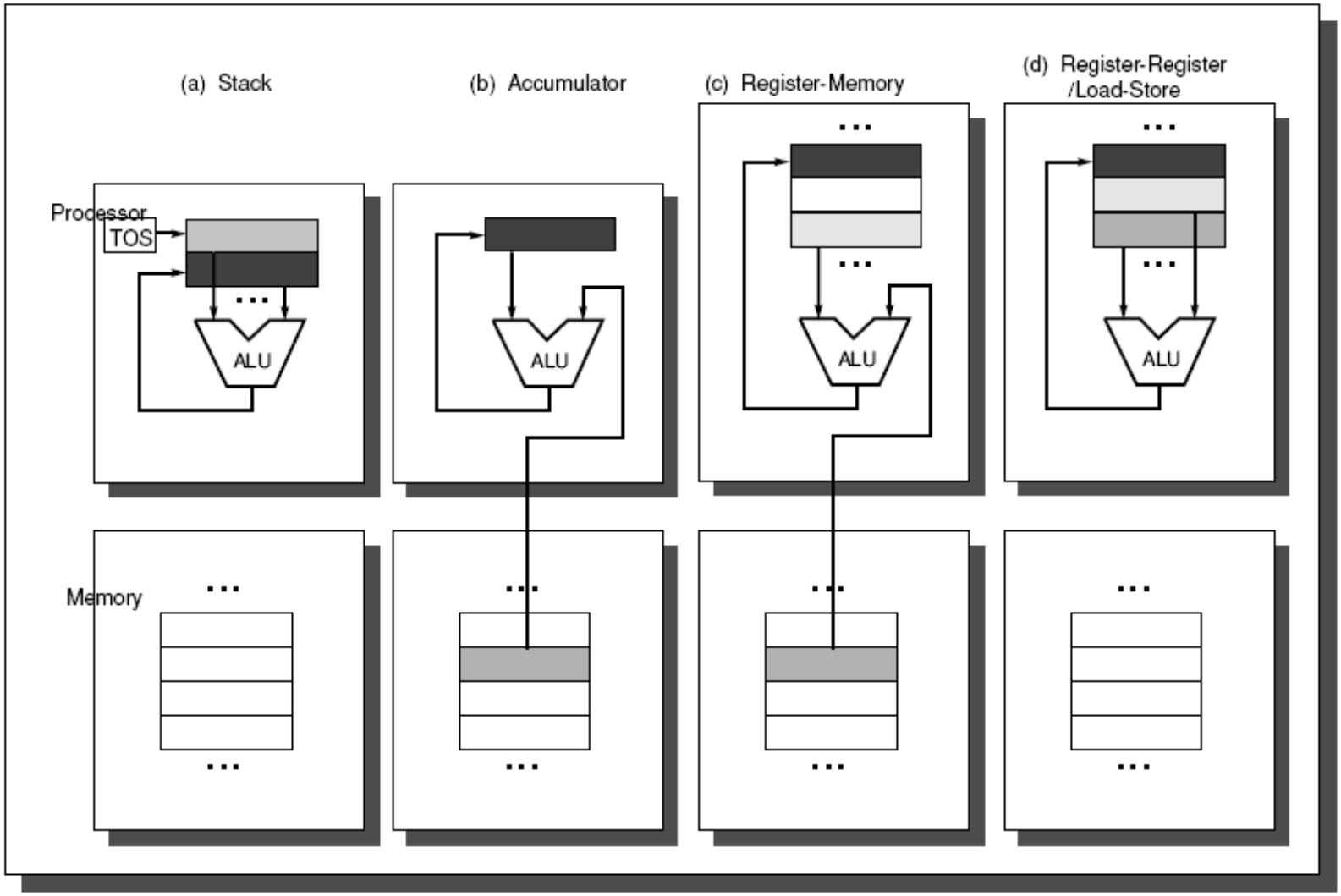
代码生成器对输入的要求

- 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，既，代码生成器可以认为输入中没有与语法或语义错误

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）

- **12.1 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



12.1.1 指令集架构

Stack	Accumulator	Register (register-memory)	Register (load-store)
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

- Accumulator-based Architecture
  - UNIVAC-I, EDSAC, IAS, 1950s
- Stack Architecture
  - B5000, B6500, 1960s
  - ALGOL
  - JVM (1990s), PL/0
- High-Level Language Computer Architecture
  - VAX-11/780, 1970s~1980s
- Complex Instruction Set Computer
  - 80x86, 1980s~
- Reduced Instruction Set Computer
  - CRAY-1, MIPS, SPARC, Intel P6 core, 1980s~

D=(A*B)+(B*C)

- 栈式架构

PUSH A
PUSH B
MUL
PUSH B
PUSH C
MUL
ADD
POP D

- 寄存器-寄存器架构

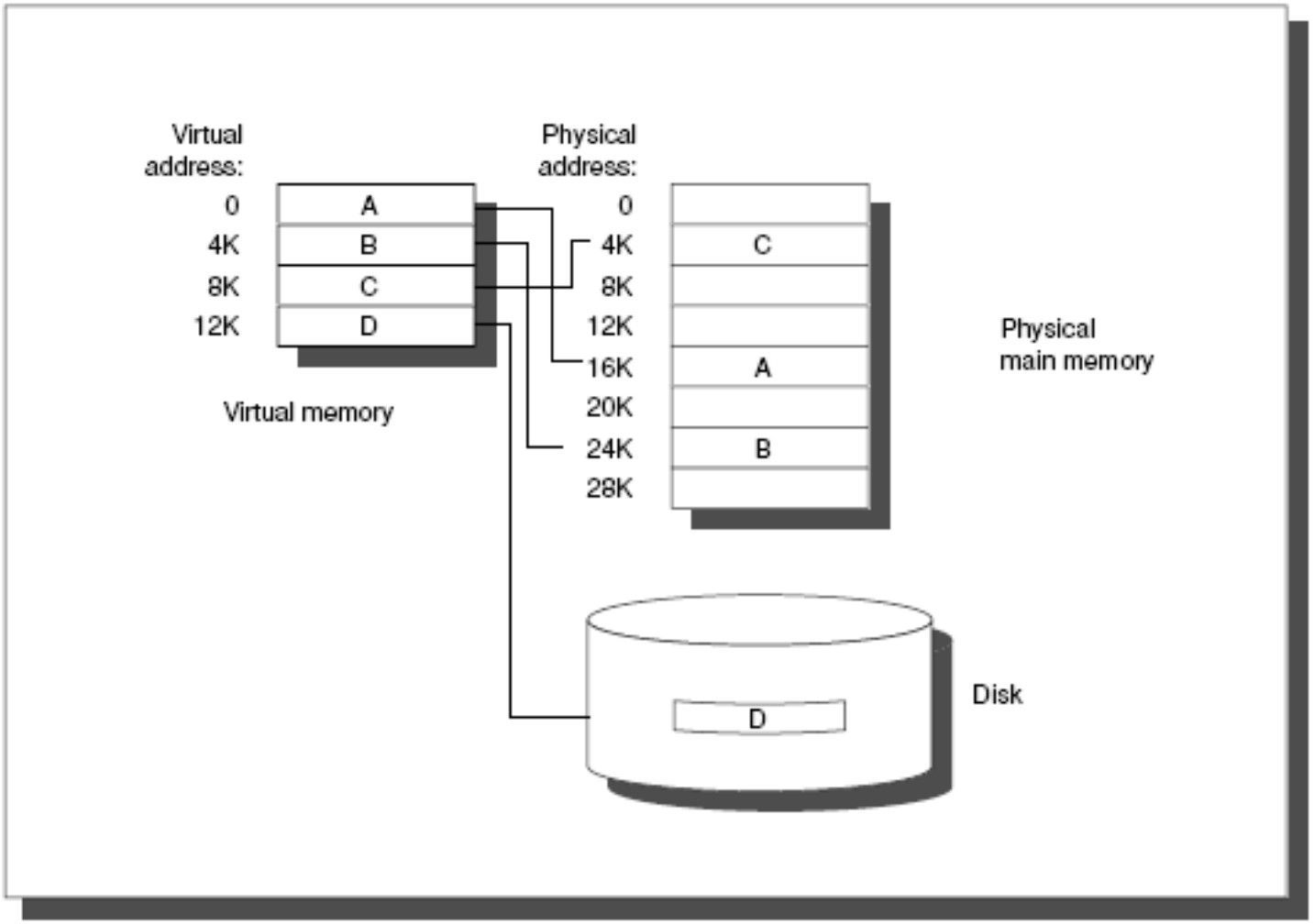
LOAD R1, A
LOAD R2, B
LOAD R3, C
MUL R1, R2, R4
MUL R2, R3, R5
ADD R4, R5, R5
STORE R5, D

12.1.2 存贮层次架构

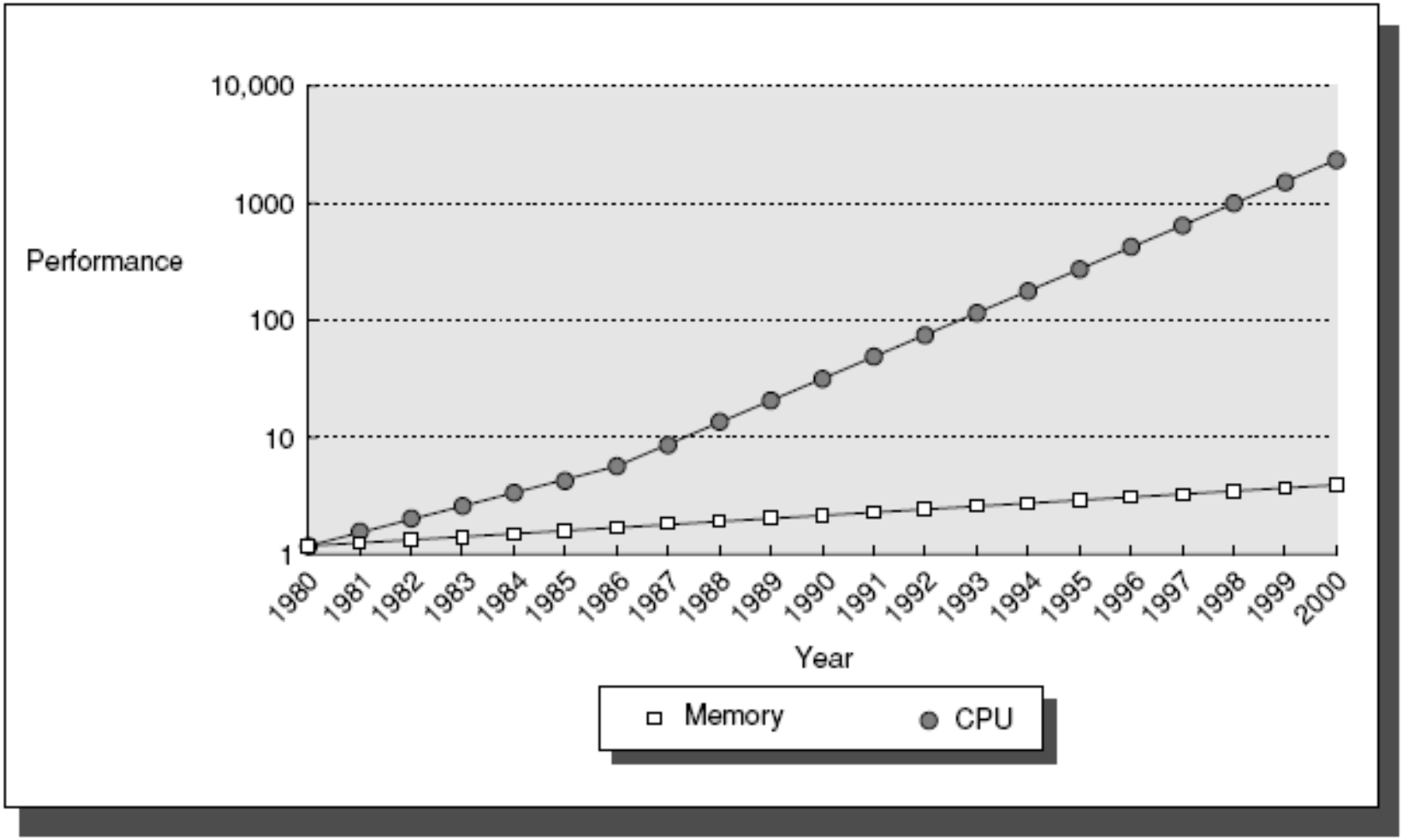
寄存器、缓存、内存、硬盘的存储访问特性

Level	1	2	3	4
Called	Registers	Cache	Main memory	Disk storage
Typical size	< 1 KB	< 16 MB	< 16 GB	> 100 GB
Implementation technology	Custom memory with multiple ports, CMOS	On-chip or off-chip CMOS SRAM	CMOS DRAM	Magnetic disk
Access time (in ns)	0.25 -0.5	0.5 to 25	80-250	5,000,000
Bandwidth (in MB/sec)	20,000-100,000	5,000-10,000	1000-5000	20-150
Managed by	Compiler	Hardware	Operating system	Operating system/operator
Backed by	Cache	Main memory	Disk	CD or Tape

虚拟内存、物理内存和磁盘



1980～2000年，CPU性能和内存访问性能的提高



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];
```

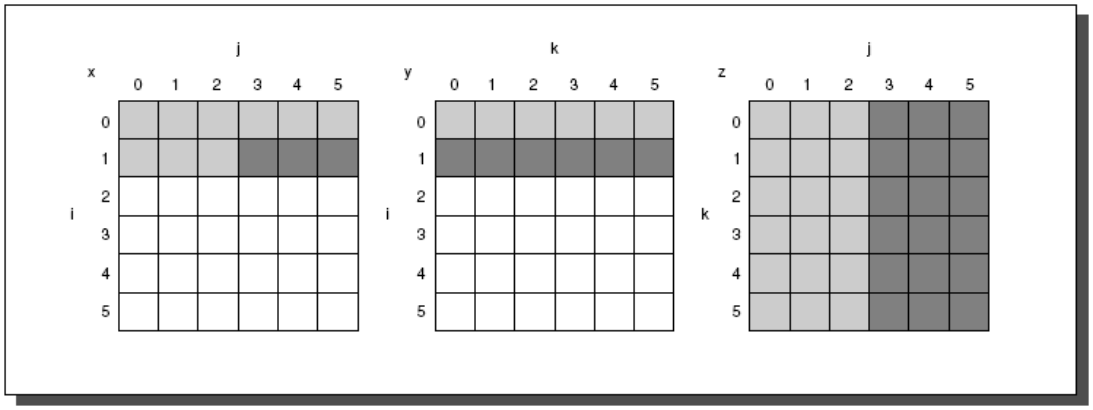
x[0][0], x[0][1], …., x[0][99], x[1][0], x[1][1], …., x[1][99], …, x[4999][0], x[4999][1], …., x[4999][99]

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

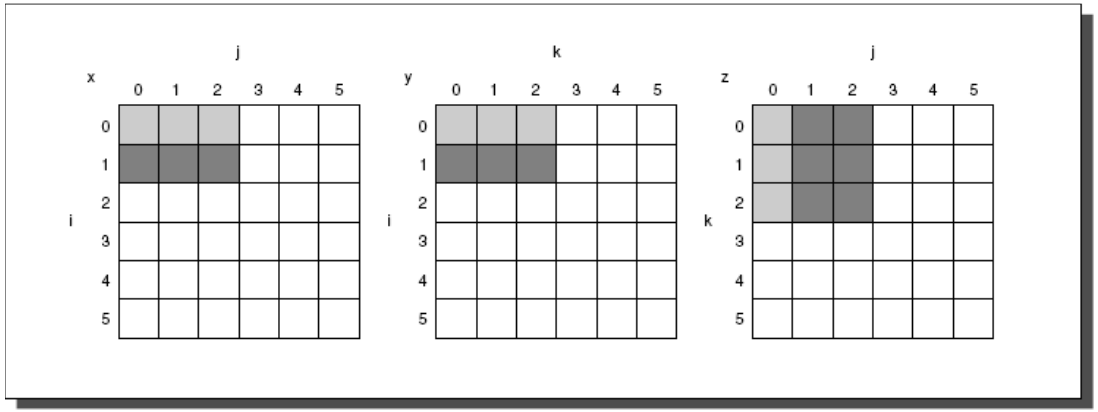
x[0][0], x[0][1], …., x[0][99], x[1][0], x[1][1], …., x[1][99], …, x[4999][0], x[4999][1], …., x[4999][99]

Tiling提高数据局部性!

```
/* Before */
for (i = 0; i < N; i = i+1)
        for (j = 0; j < N; j = j+1)
                {r = 0;
                 for (k = 0; k < N; k  = k + 1)
                        r = r + y[i][k]*z[k][j];
                 x[i][j] = r;
                };
```

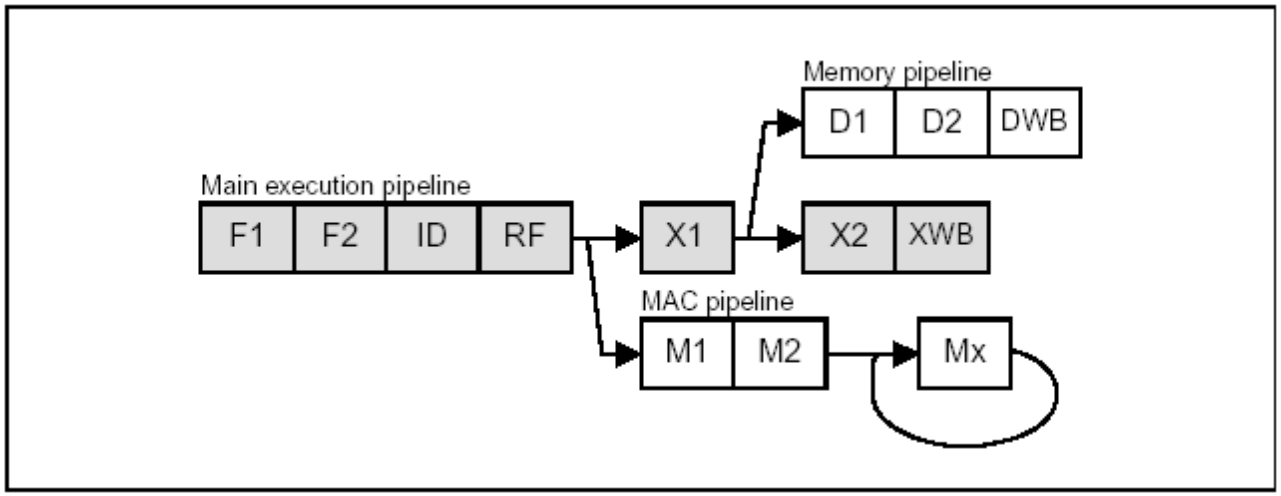



```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
        for (j = jj; j < min(jj+B,N); j = j+1)
            {r = 0;
             for (k = kk; k < min(kk+B,N); k = k + 1)
                    r = r + y[i][k]*z[k][j];
             x[i][j] = x[i][j] + r;
            };
```



12.1.3 流水线

XScale core RISC 超流水线



I0:	*add R0,R5,R6*	1	*I2:*	*ldr R2, [R4, 0x4]*	3*
I1:	*sub R1, R7,R8*	1	*I0:*	*add R0, R5, R6*	1
I2:	*ldr R2, [R4, 0x4]*	3	*I1:*	*sub R1, R7, R8*	1
I3:	*add R3, R2, R1*	1	*I3:*	*add R3, R2, R1*	1

时钟周期：1+1+3+1 = 6

时钟周期：1+1+1+1 = 4

超标量流水线

Mov ecx, eax

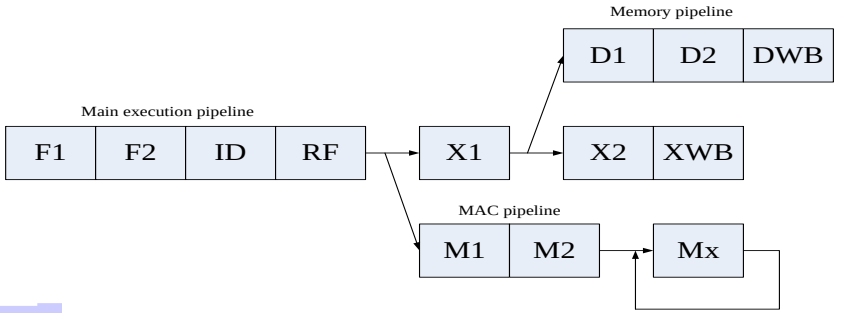
Add edx, [esp+0x20]

Mul edx, ecx

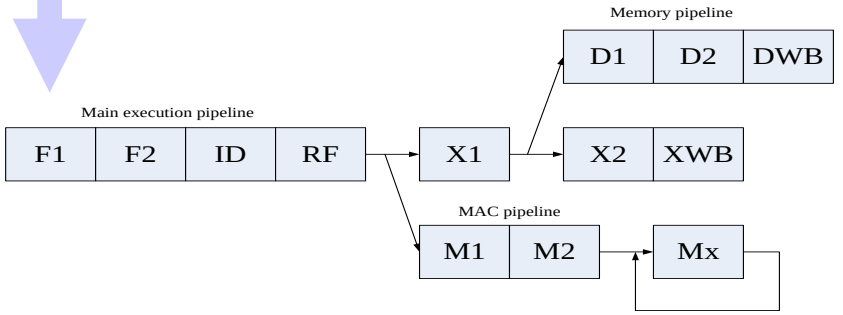
Cmp edx, esi

Jz label_1

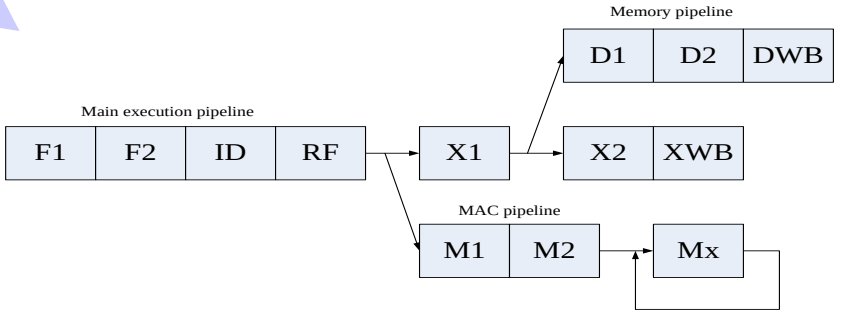
流水线1



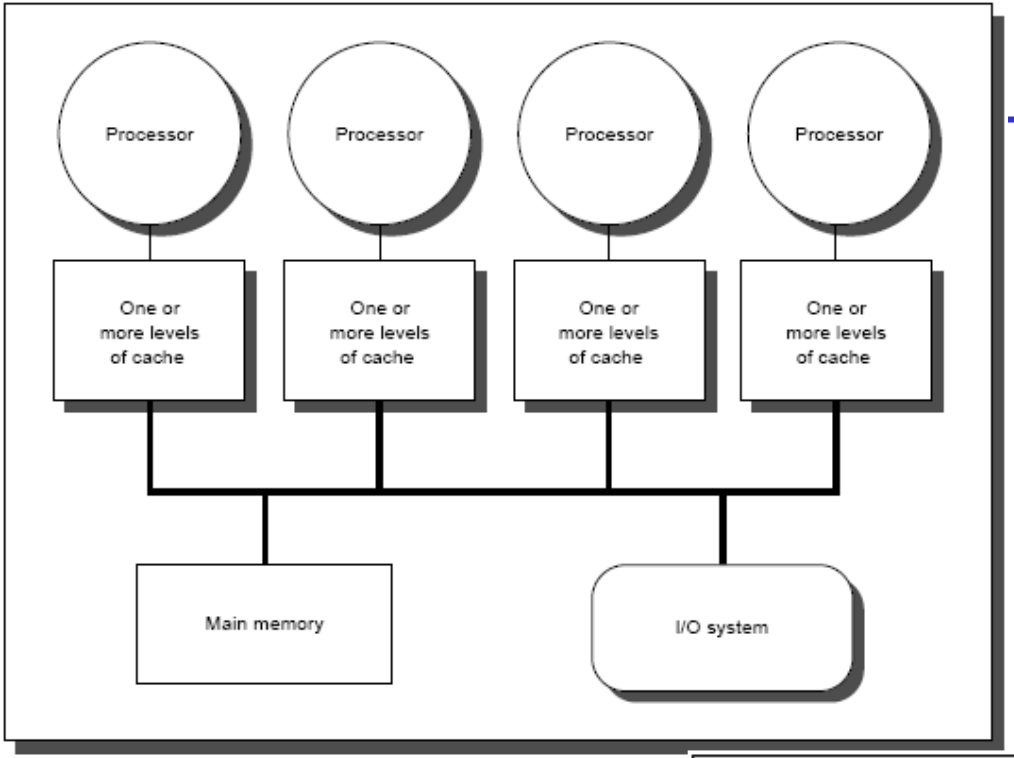
流水线2



流水线3



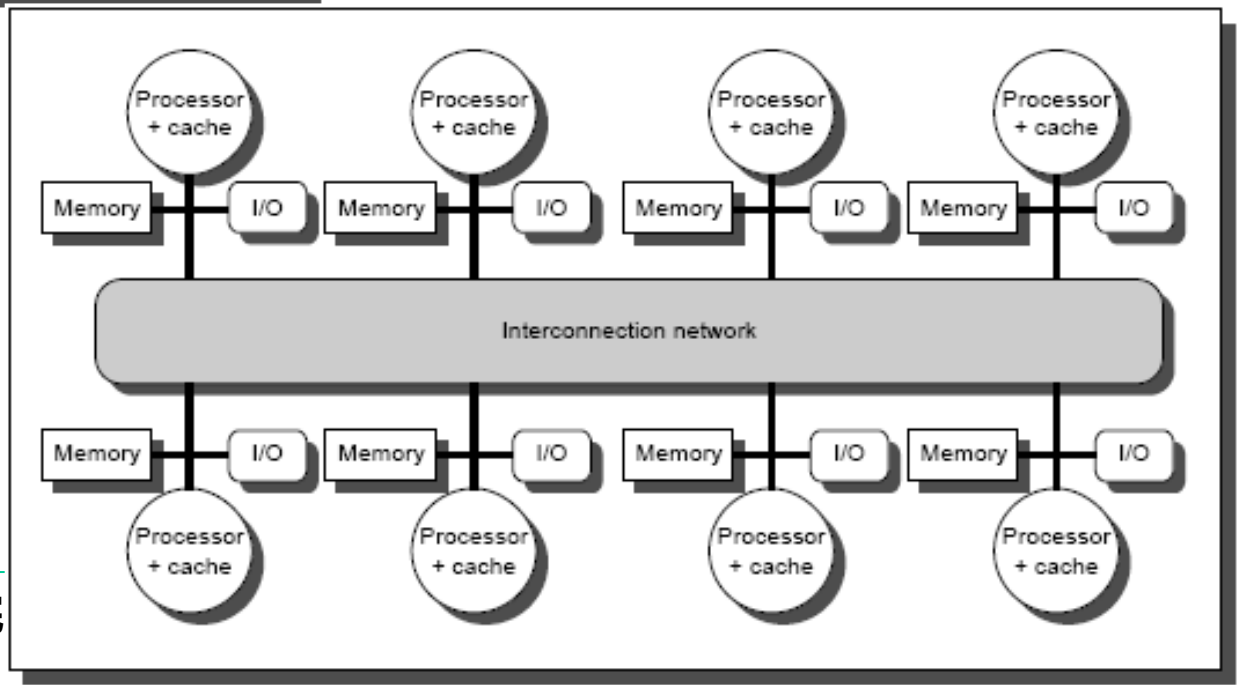
- Multiple-issued processor
  - Intel PIII/P4
- Very Long Instruction Word (VLIW)
  - i860
- Explicitly Parallel Instruction Computers (EPIC)
  - IA64



A centralized shared-memory multiprocessor

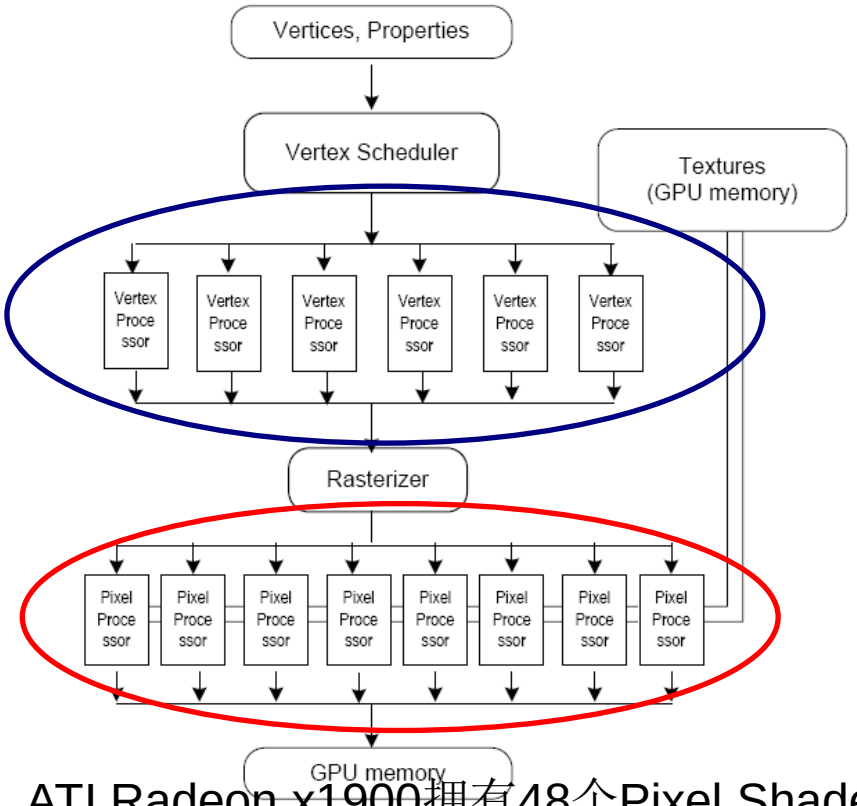
线程级并行！

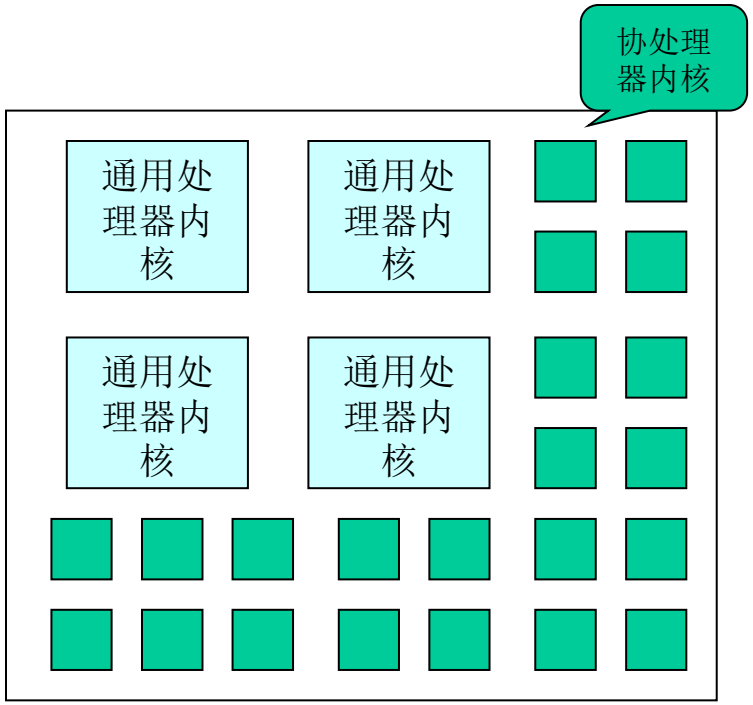
Distributed-memory multiprocessors



数据级并行！

- 多核协处理器
  - GPU，GPGPU
- 异构CMP
  - 同构CMP + 多核协处理器





ATI Radeon x1900拥有48个Pixel Shader处理器，每个都可以在1个时钟周期内处理4个浮点运算。性能达到**250 GFLOPS**。

Intel® Core™2 Extreme Processor QX9650 45nm 3GHz 12M L2, 48 GFLOPS

在100个处理器上想要达到80倍的加速比，串行部分的运行只能占原计算的0.25%！

Amdahl's Law is

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

$$80 = \frac{1}{\frac{\text{Fraction}_{\text{parallel}}}{100} + (1 - \text{Fraction}_{\text{parallel}})}$$

Simplifying this equation yields

$$0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) = 1$$

$$80 - 79.2 \times \text{Fraction}_{\text{parallel}} = 1$$

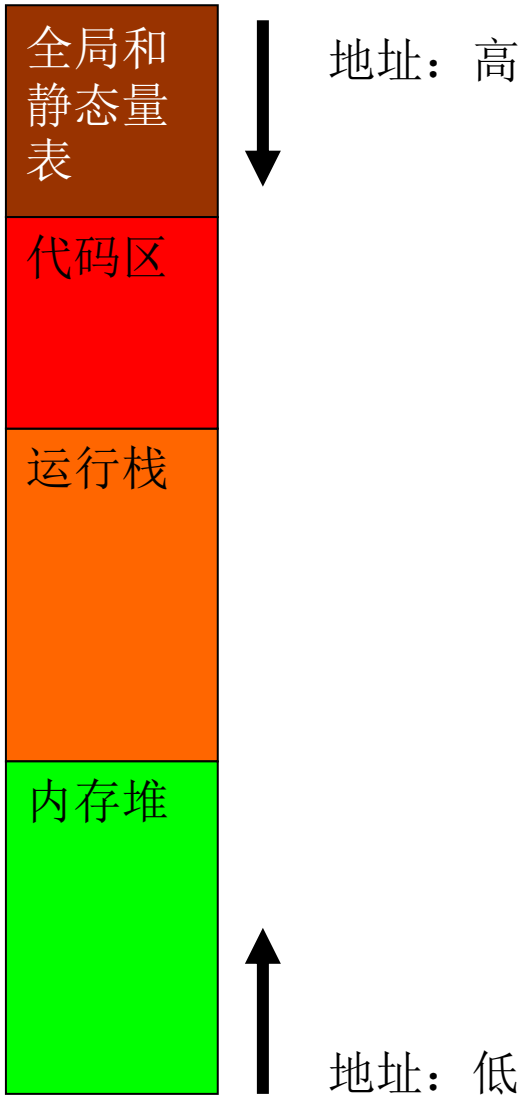
$$\text{Fraction}_{\text{parallel}} = \frac{80 - 1}{79.2}$$

$$\text{Fraction}_{\text{parallel}} = 0.9975$$

12.2 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
  - 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

- 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



12.2.1 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
```

12.2.2 程序运行栈的设计

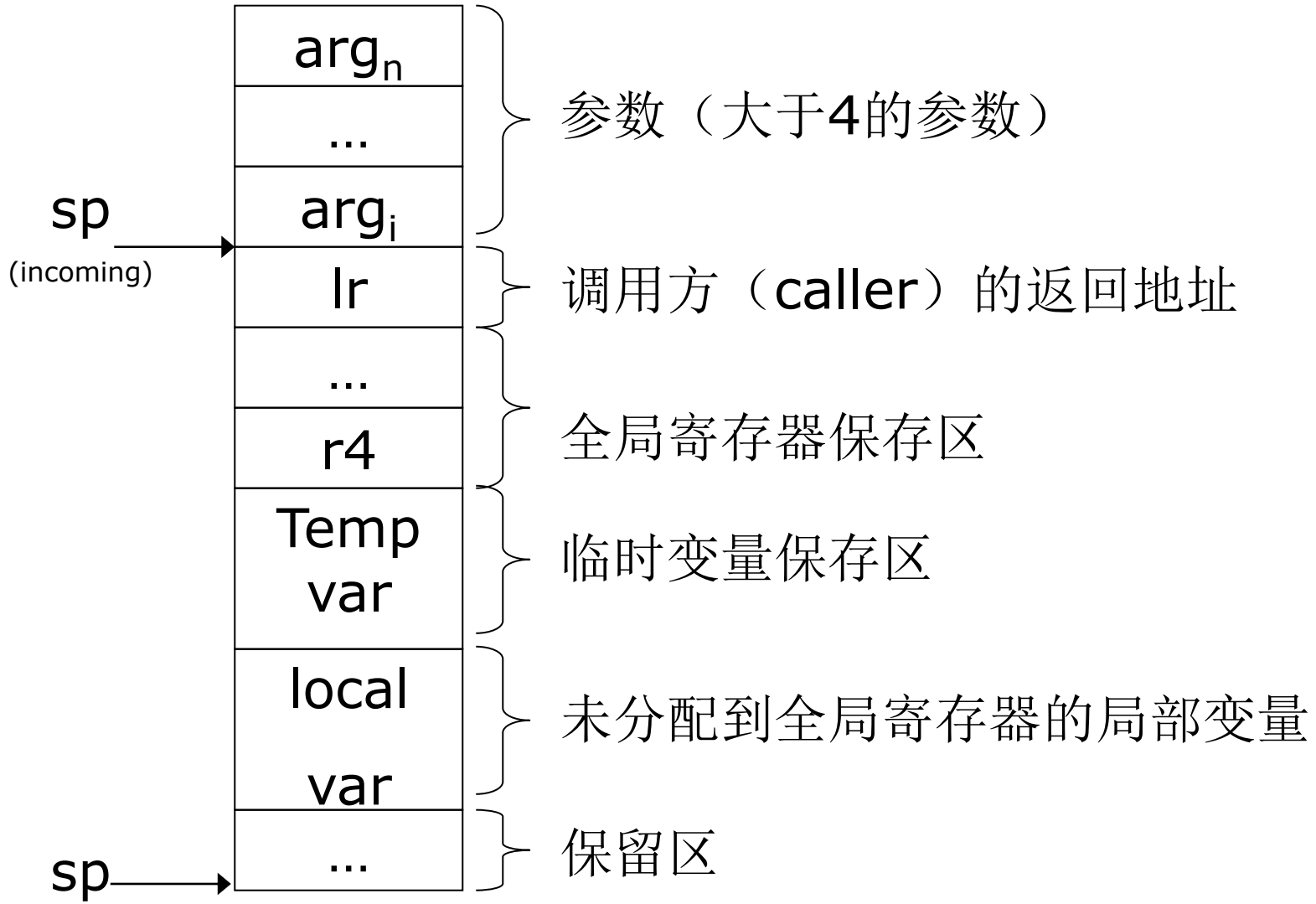
- 子程序/函数运行时所需的基本空间
- 进入子程序/函数时分配，地址空间向下生长（从高地址到低地址）
- 从子程序/函数返回时，当前运行栈将被废弃
- 递归调用的同一个子程序/函数，每次调用都将获得独立的运行栈空间

```
1// C12P1.cpp
2#include "stdafx.h"
3int global_val = 0 ;
4int 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}
14int 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
```

- 一个典型的运行栈包括
  - 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



12.4 寄存器的分配和指派

- 为什么要分配和管理寄存器？
  - 某些运算只能发生在寄存器当中
  - 寄存器的访问速度是所有存储形式中最快的
  - 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

- 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
    - ARM：R4~R11

MIPS调用规范Calling Convention

- **调用函数**时的参数传递
  - 使用MIPS仿真器**$a0-$a3**，从左到右依次传递前4个参数
  - **参数数量超过4的**，从左到右依次压在运行栈（活动记录）上
  - **返回地址**call指令自动保存在连接寄存器**$ra**中

foo(arg1, arg2, arg3, arg4, arg5, arg6) ;

$a0 = arg1, $a1 = arg2, $a2 = arg3, $a3 = arg4

push arg5

push arg6

call foo

MIPS调用规范Calling Convention

- **进入函数**时
  - 编译器决定是否将**$a0-$a3**传递进来的参数保存在运行栈，还是传递给全局寄存器
  - 除了叶子函数（leaf method，不会调用任何其他函数的函数），保存返回地址的连接寄存器**$ra**值需要编译器生成代码保存在运行栈上
  - 返回值保存在**$v0-$v1**
  - 编译器要保证返回时的**$sp寄存器**值和调用函数前（传参前）保持一致

MIPS调用规范Calling Convention

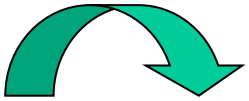
- 临时寄存器（Caller-saved register, scratch register）
  - 原则上生存范围不超过基本块，不跨越函数调用
  - $t0-$t7，$t8-$t9
  - 以下寄存器在保证上下文正确性前提下也可用作临时寄存器：$a0-$a3，$v0-$v1, $ra
- 全局寄存器（Callee-saved register, permanent register）
  - 跨越基本块，跨越函数调用
  - 一个函数使用多少全局寄存器，在入口处保存多少，返回前恢复多少
  - $s0-$s7
- 寻址及堆栈寄存器
  - 全局指针寄存器$gp
  - 栈指针寄存器$sp
  - Frame指针寄存器$fp（活动记录基地址）

12.4.1 全局寄存器分配

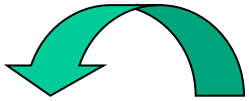
- 分配原则
  - 局部变量参与全局寄存器分配
    - 为什么全局变量/静态变量不参与全局寄存器分配？

寄存器专属于线程！

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 = 1

s_c = 1

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

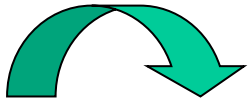
第一次被调用

a = 2

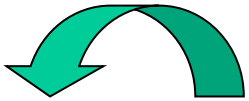
s_c = 1

s_c = 3

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 = 1

s_c (ESI) = 1

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

第一次被调用

a = 2

s_c (ESI) = 0

s_c (ESI) = 2

12.4.1.1 引用计数

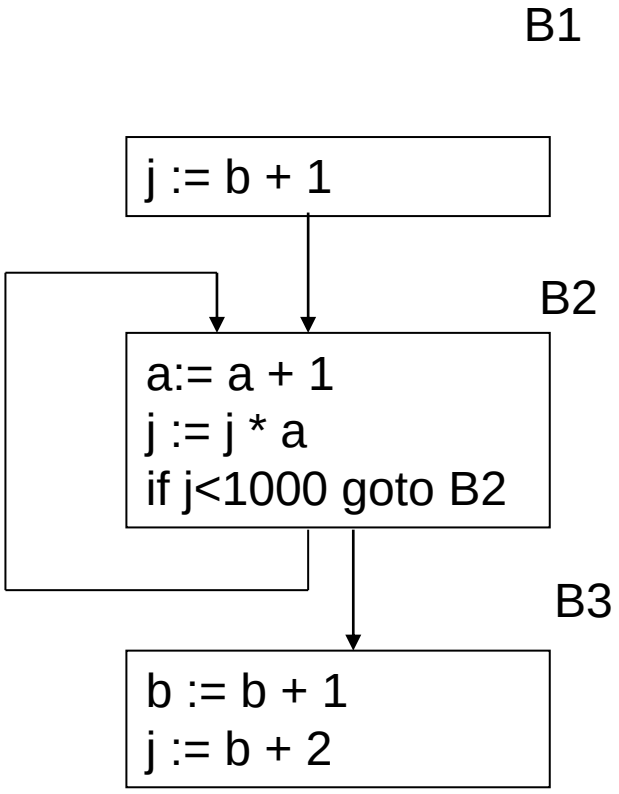
- 原则是：如果一个局部变量被访问的次数较多，那么它获得全局寄存器的机会也较大
- 需要注意的是：出现在循环，尤其是内层嵌套循环中的变量的被访问次数应该得到一定的加权

3个局部变量，2个全局寄存器可供分配，谁将获得寄存器？

j 5次

b 4次

a 3次



12.4.1.2 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？
    - 它的节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。
    - 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**

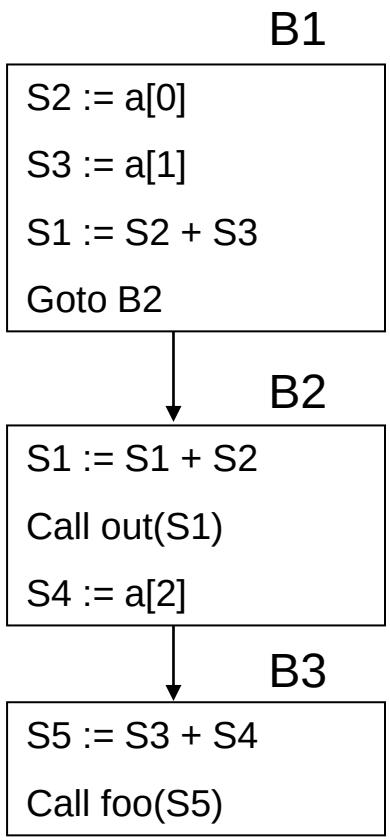
- 例：

基本块入口处的活跃变量

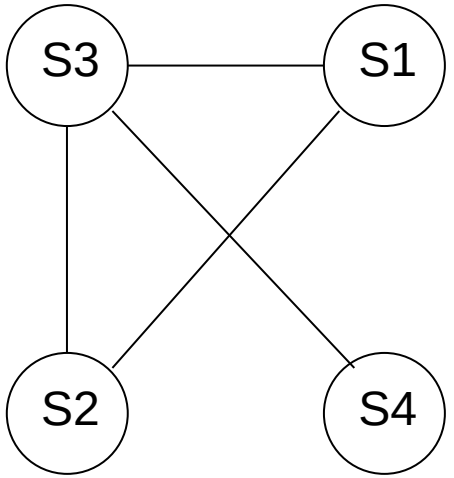
S1，S2，S3

S3, S4

流图

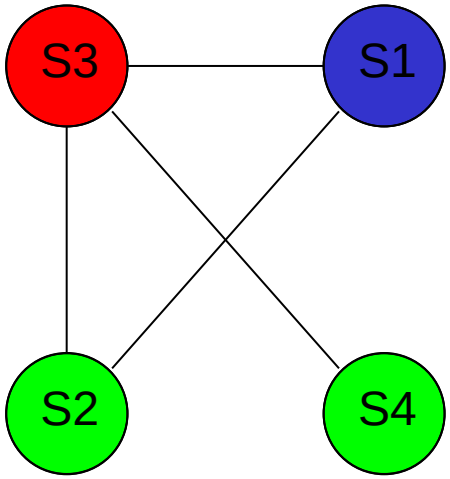


冲突图

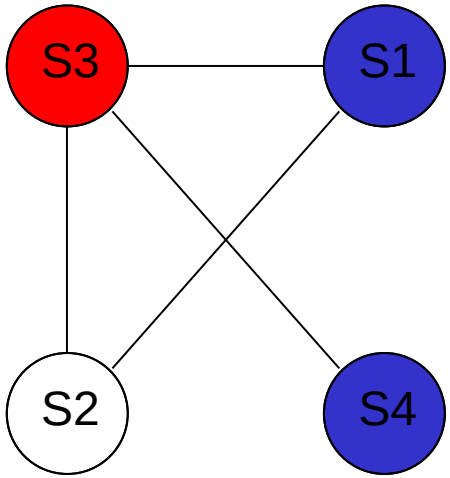


- 步骤2、如果可供分配k个全局寄存器，那么我们就尝试用k种颜色给该冲突图着色

假设1：k=3，R0,R1,R2



假设2：k=2，R0,R1



S2 := a[0]

S3 := a[1]

S1 := S2 + S3

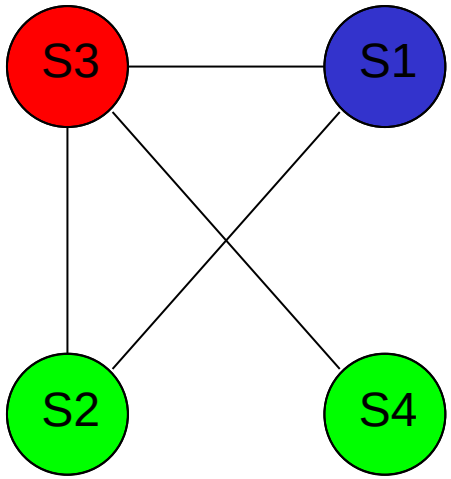
Goto B2

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

S5 := S3 + S4

Call foo(S5)

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]

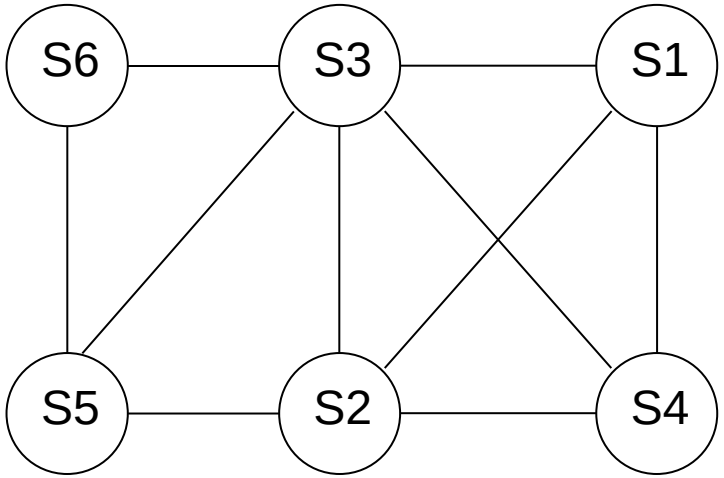
S5 := R0 + R2

Call foo (S5)

算法12.2 一种启发式图着色算法

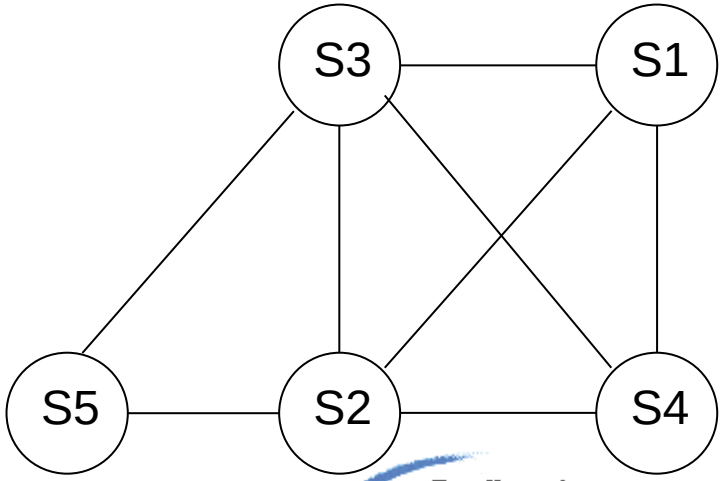
- 冲突图G
  - 寄存器数目为K
    - 假设 K=3

G

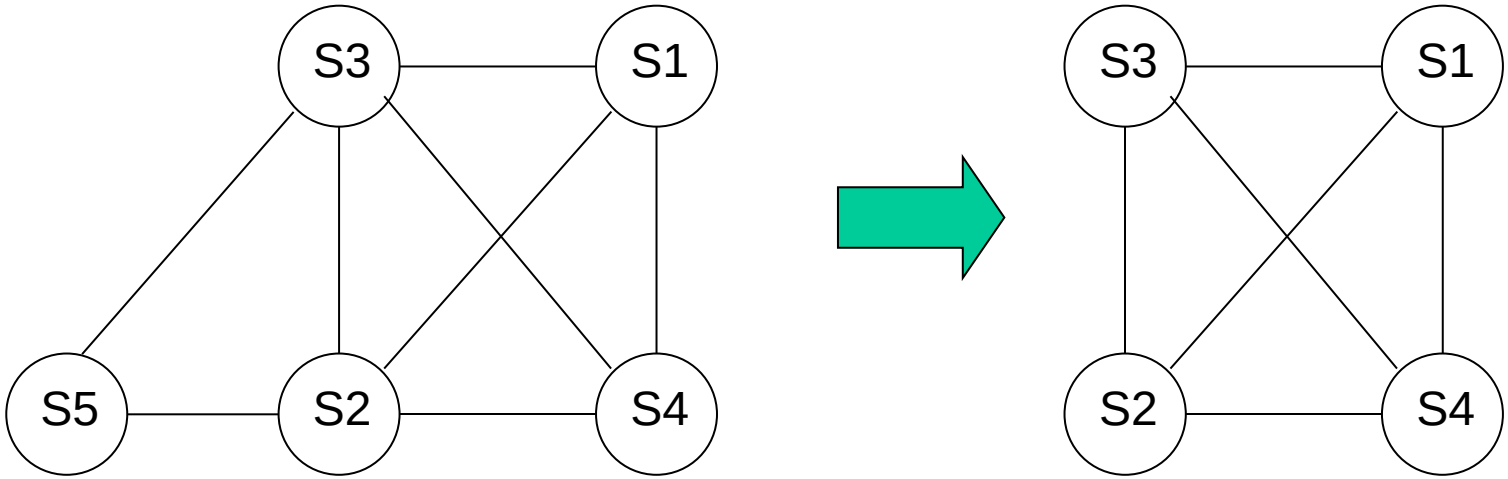


- 步骤1、找到第一个连接边数目小于K的节点，将它从图G中移走，形成图G’

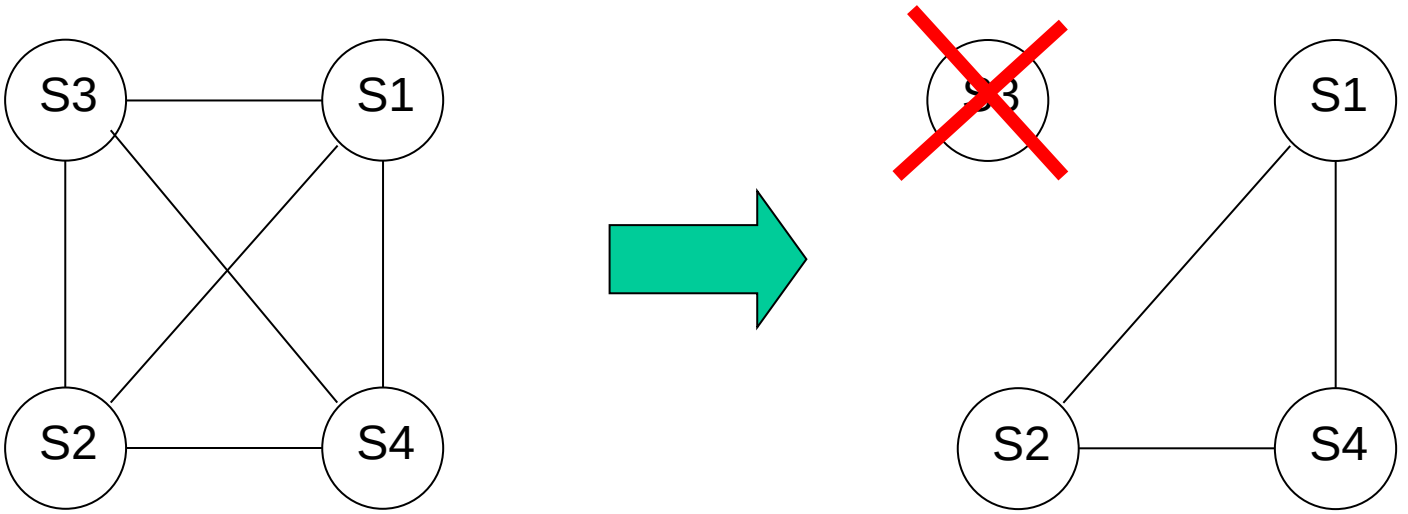
G’



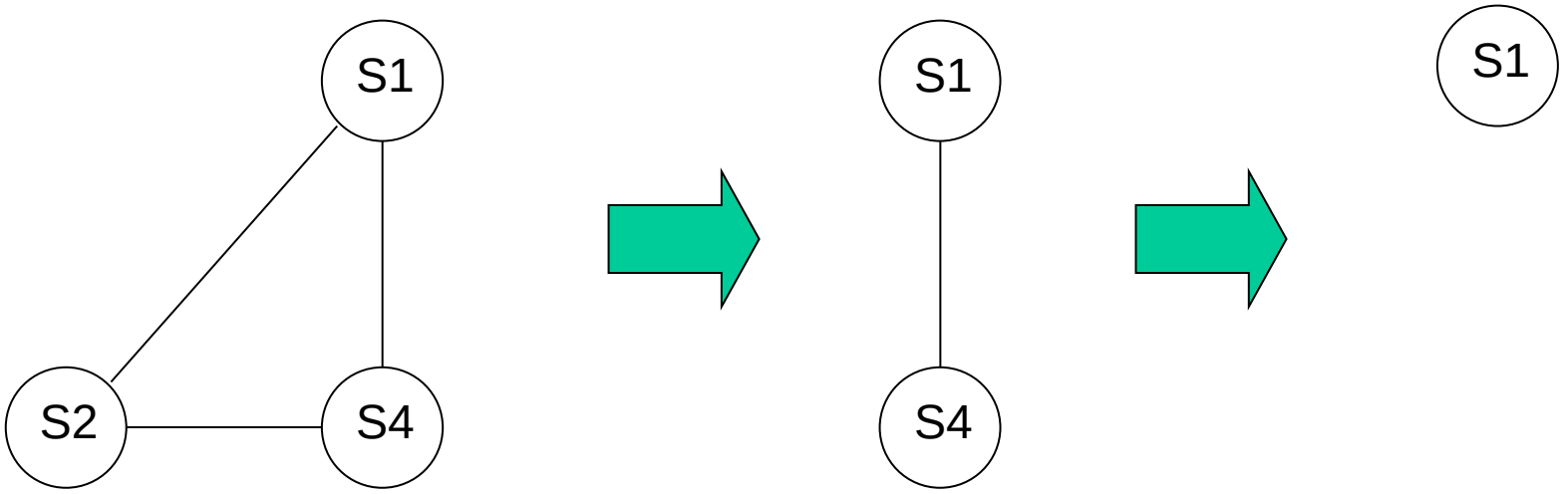
- 步骤2、重复步骤1，直到无法再从G’中移走节点



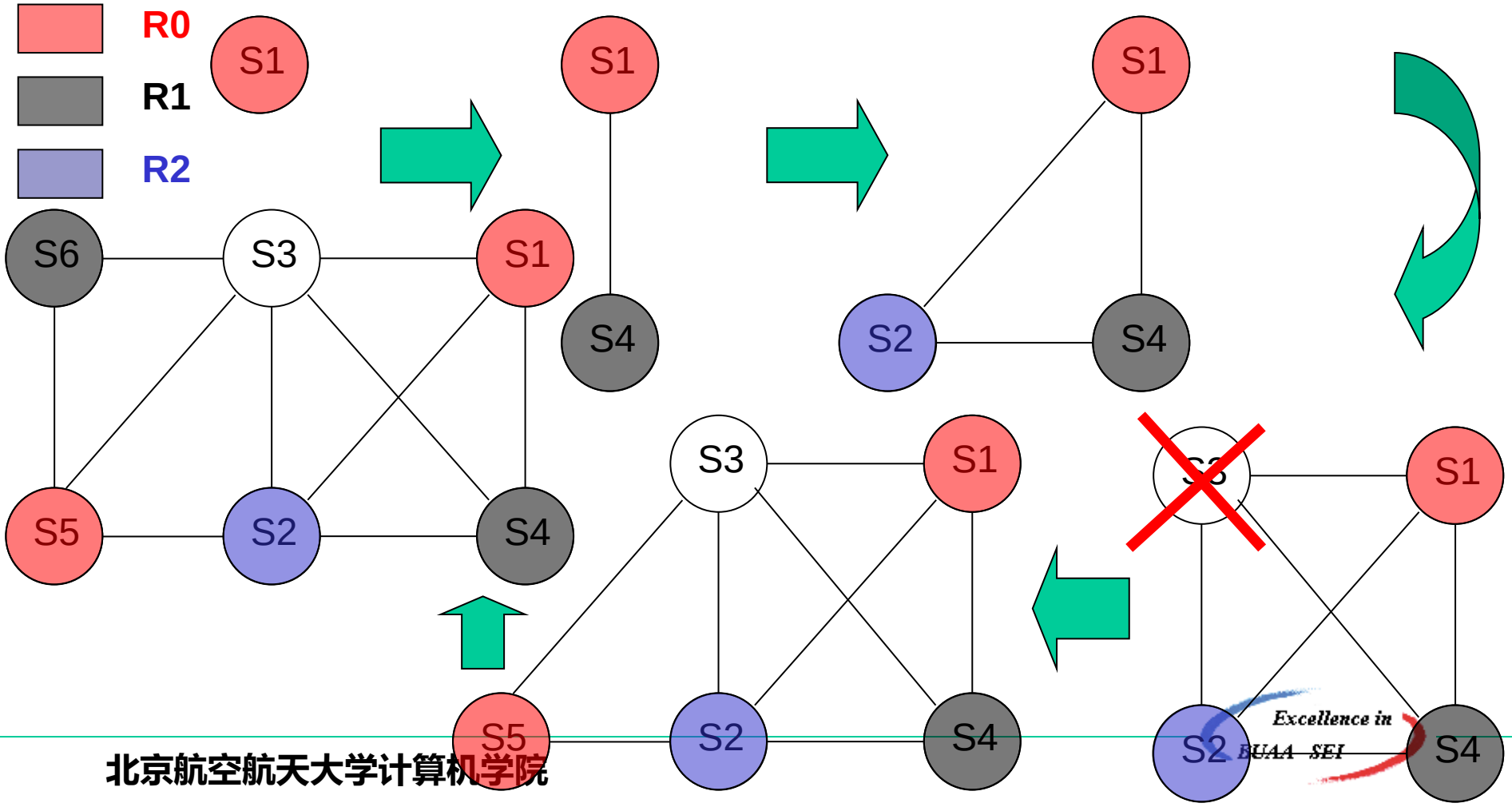
- 步骤3、在图中选取*适当*的节点，将它记录为“不分配全局寄存器”的节点，并从图中移走



- 步骤4、重复步骤1~步骤3，直到图中仅剩余1个节点



- 步骤5、给剩余的最后一个节点选取一种颜色，然后按照节点被移走的顺序，反向将节点和边添加进去，并依次给新加入的节点选取颜色



12.4.2 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

t1	EAX
	EDX

```
mov EAX, EDI
neg EAX
```

t2 := t1 - b

t1	EAX
t2	EDX

```
mov EDX, EAX
sub  EDX, ESI
```

t3 := t2 + t2

t3	**EAX**
t2	EDX

```
mov [ESP+08H], EAX
mov EAX, EDX
add EAX, EAX
```

a := t3

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

例：

(1) t1 = -c

(2) t2 = t1 – b

(3) t3 = t2 + t2

(4) a = t3

a	EBX	t3	ESP+10H
b	ESI	t2	ESP+0CH
c	EDI	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

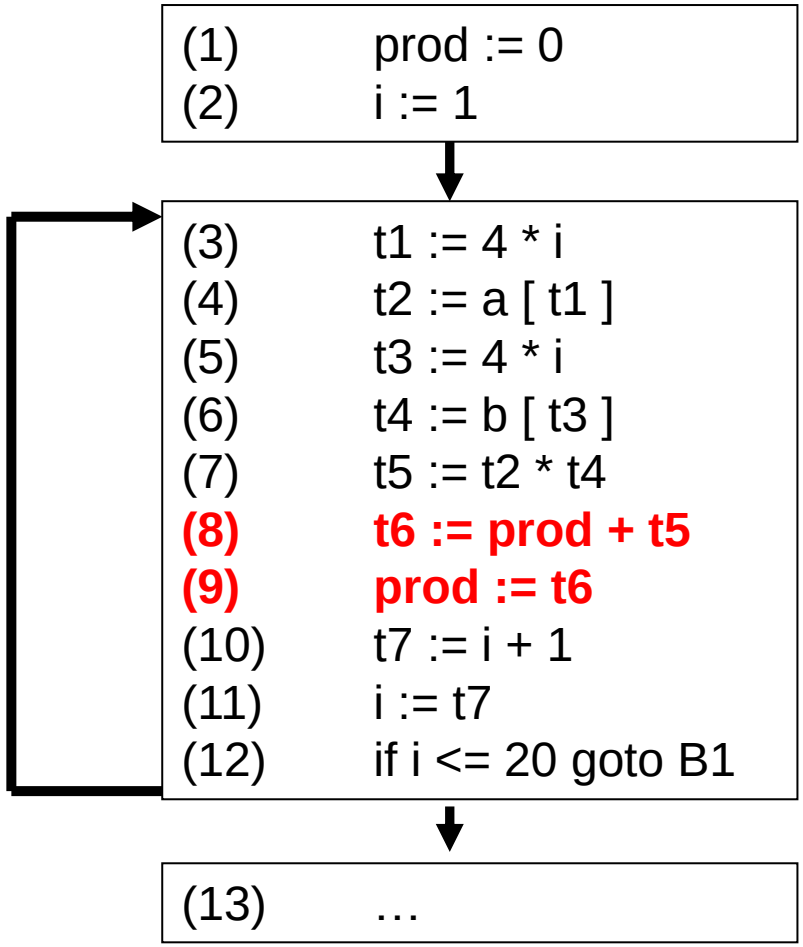
(3) mov [ESP+08H], EAX
    mov EAX, EDX
    add EAX, EAX

(4) mov EBX, EAX

12.5 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
    - Itanium

例：



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

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

作业：

15章（p381）1,4,5,6

多级缓存中缓存大小和访问未命中比率之间的关系