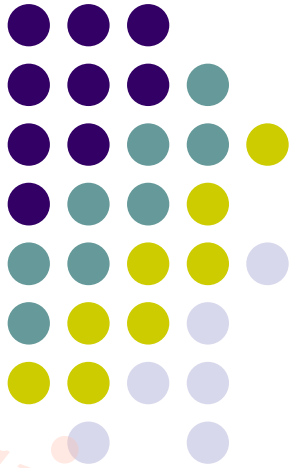
嵌入式系统设计与应用

第五章 程序设计与分析（2）

西安交通大学电信学院

孙宏滨

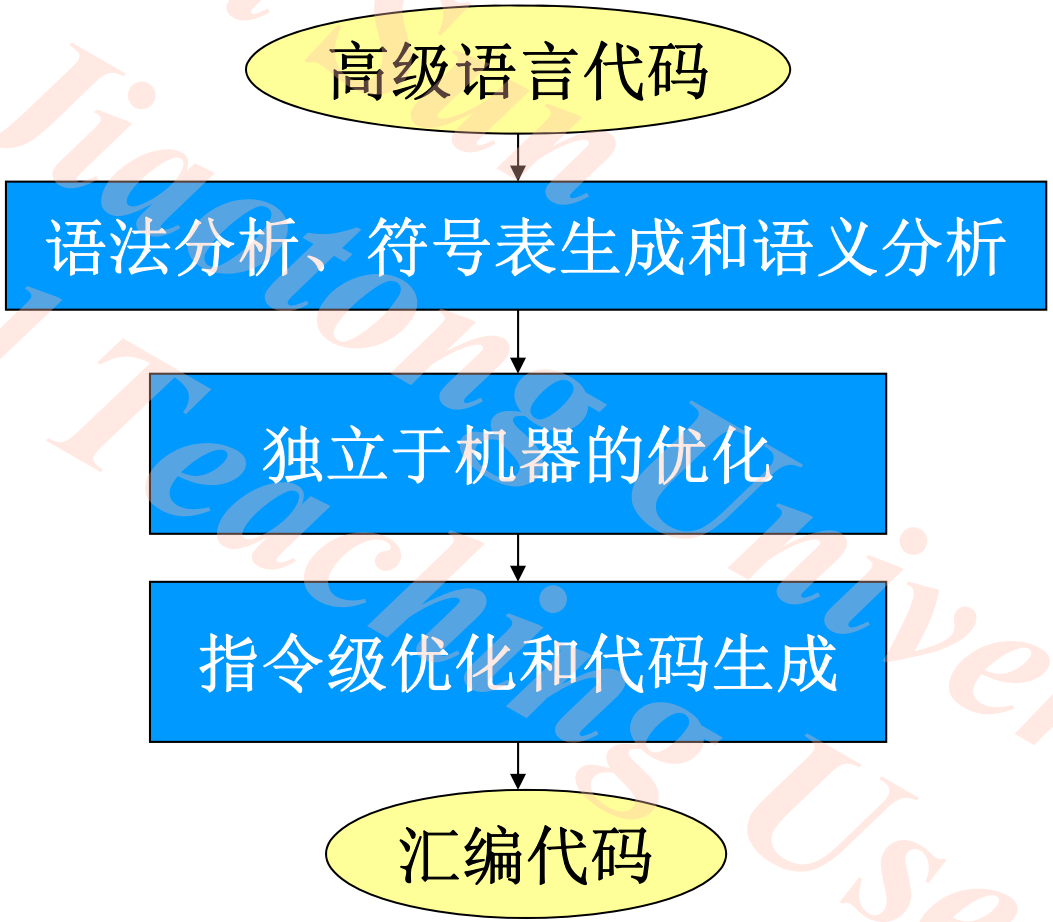


4 基本编译技术

- 编译 = 翻译 + 优化
  - 翻译：将高级语言程序翻译成低级形式的指令
  - 优化：与翻译过程中所使用的语句间相互独立的简单方法相比，优化可以生成更高指令的代码。

- 编译决定着代码质量：
  - 占用CPU资源
  - 存储器访问调度
  - 代码大小

基本编译过程



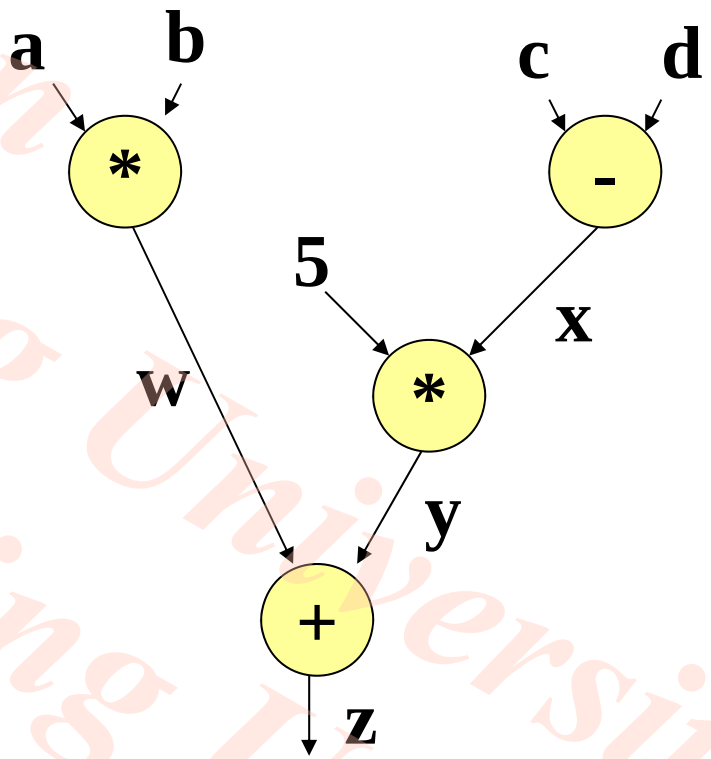
语句翻译与优化

- 源代码被翻译成类似于CDFG的中间形式
- CDFG经过变换与优化
- 通过优化决策，CDFG被翻译成指令
- 指令被进一步优化

算术表达式（1）

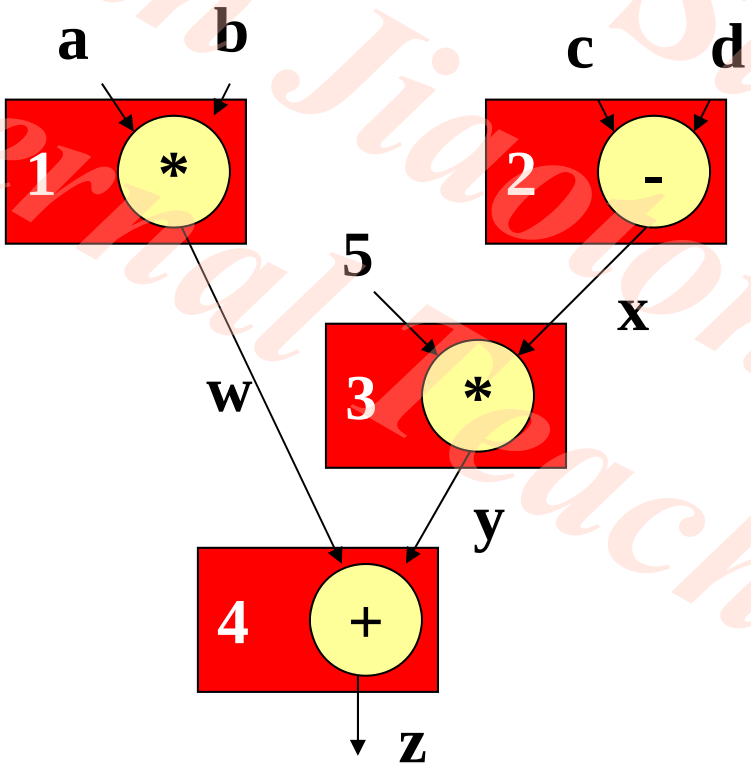
a*b + 5*(c-d)

expression



DFG

算术表达式（2）



DFG

```
ADR r9,a
MOV r1,[r9]
ADR r9,b
MOV r2,[r9]
MUL r3,r1,r2
ADR r9,c
MOV r4,[r9]
ADR r9,d
MOV r5,[r9]
SUB r6,r4,r5
MUL r7,r6,#5
ADD r8,r7,r3
```

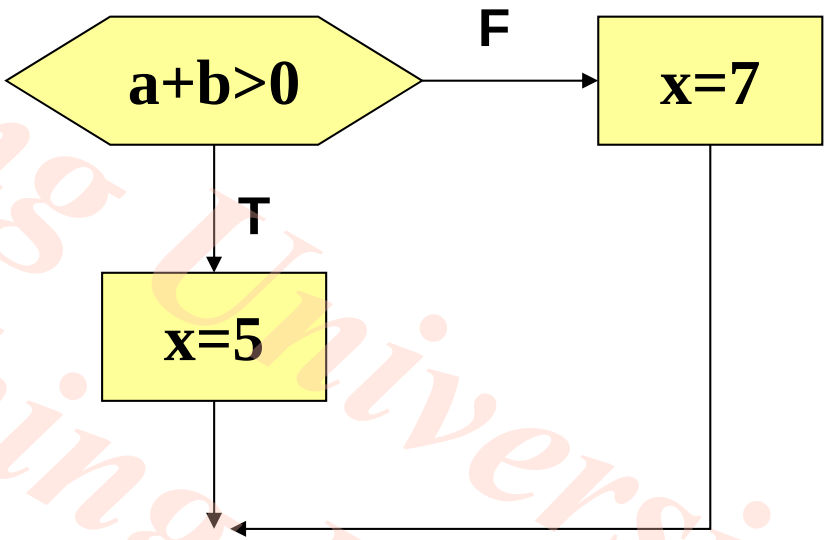
code

一个明显的优化方法是重用哪些其值不用再保存的寄存器。

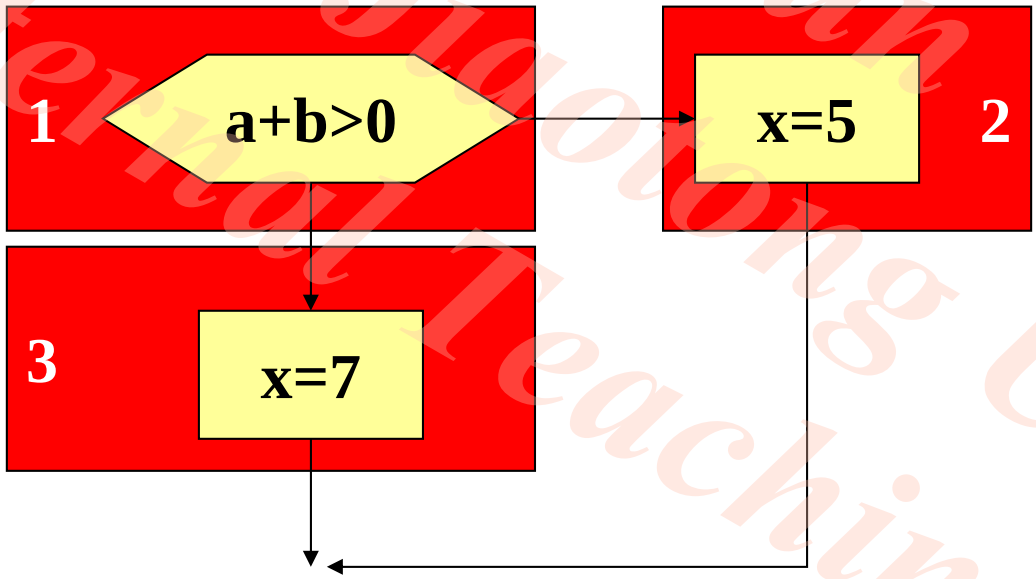
寄存器分配问题！

条件语句（1）

```
if (a+b > 0)
   x = 5;
else
   x = 7;
```



条件语句（2）




```
        ADR r5,a
        LDR r1,[r5]
        ADR r5,b
        LDR r2,[r5]
        ADD r3,r1,r2
        BLE label3
        LDR r3,#5
        ADR r5,x
        STR r3,[r5]
        B stmtend
label3  LDR r3,#7
        ADR r5,x
        STR r3,[r5]
stmtend ...
```

过程链接

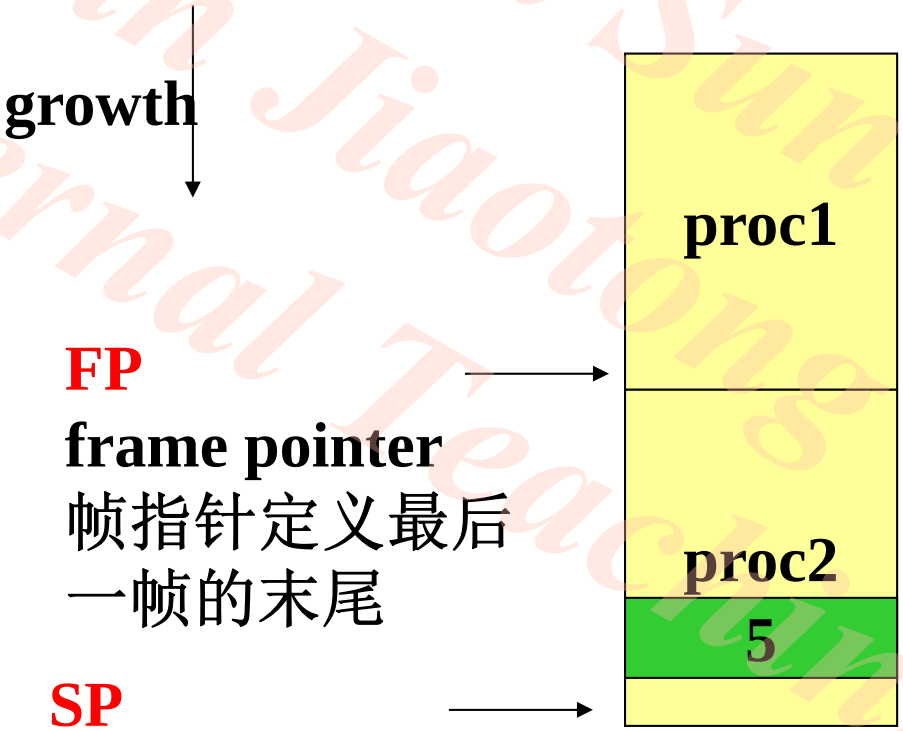
- 过程定义包括:
  - 生成处理过程调用与返回的代码
  - 传递参数和返回值
- 过程链接机制允许参数和返回值传递，并保护和恢复修改过的寄存器。
- 参数和返回值主要通过栈（**stack**）来传递
  - 少量参数也可以通过寄存器来传递

过程调用与传递参数

```
void f1 (int a) {
        f2 (a);
}
```

```
f1  LDR  r0, [r13]
    STR  r14, [r13]! ; call f2()
    STR  r0, [r13]!
    BL    f2
    SUB  r13, #4      ; return
    LDR  r15, [r13]!
```

过程栈




```
proc1(int a) {
        proc2(5);
}
```

ARM过程链接

- **ARM过程调用标准（ARM Procedure Call Standard，APCS）：**
  - **r0-r3**用于将参数传入过程。如果需要的参数超过**4**个，则将被置于栈中。
  - **r0**也用于保存返回值。
  - **r4-r7**保存寄存器变量。
  - **r11**是帧指针**fp**，**r13**是栈指针**sp**。
  - **r10**按栈的大小保存地址上限，用于检测栈的溢出。

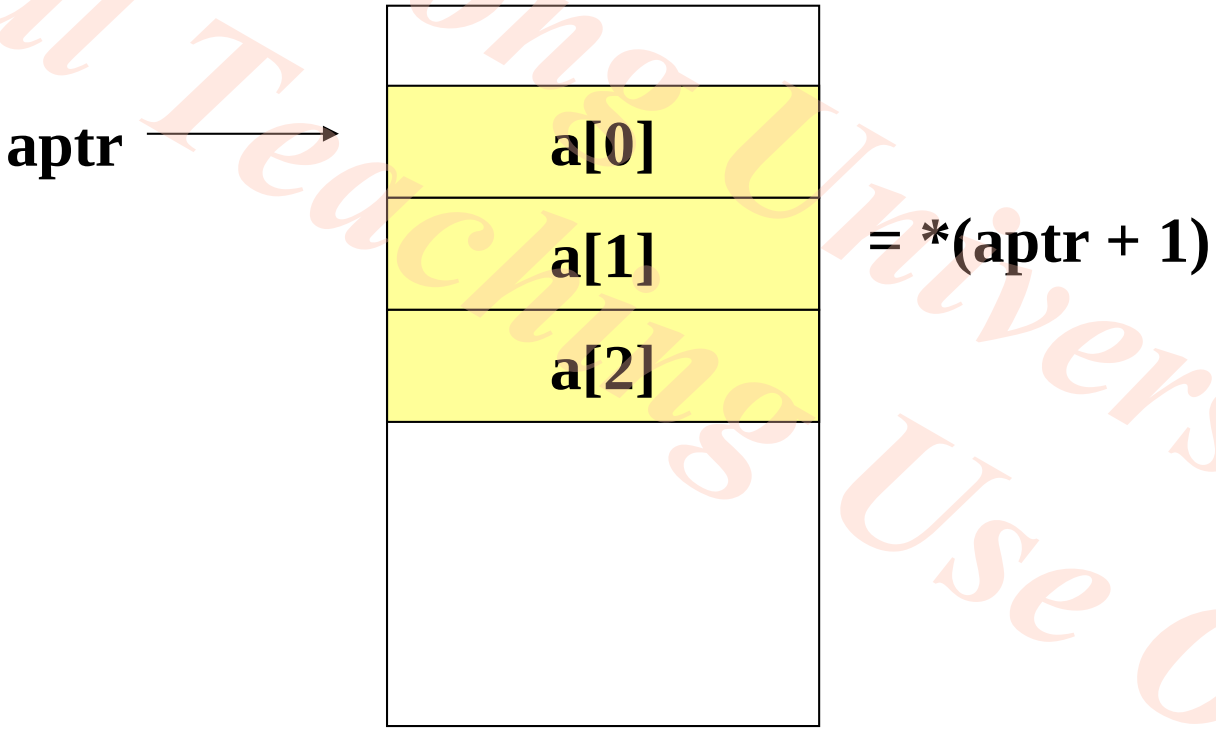
数据结构

- 编译器需要将数据结构的索引翻译成对原始内存的索引。

- 地址计算：有些计算在编译过程中完成，有些在运行时完成
  - 一维数组
  - 二维数组
  - 结构体

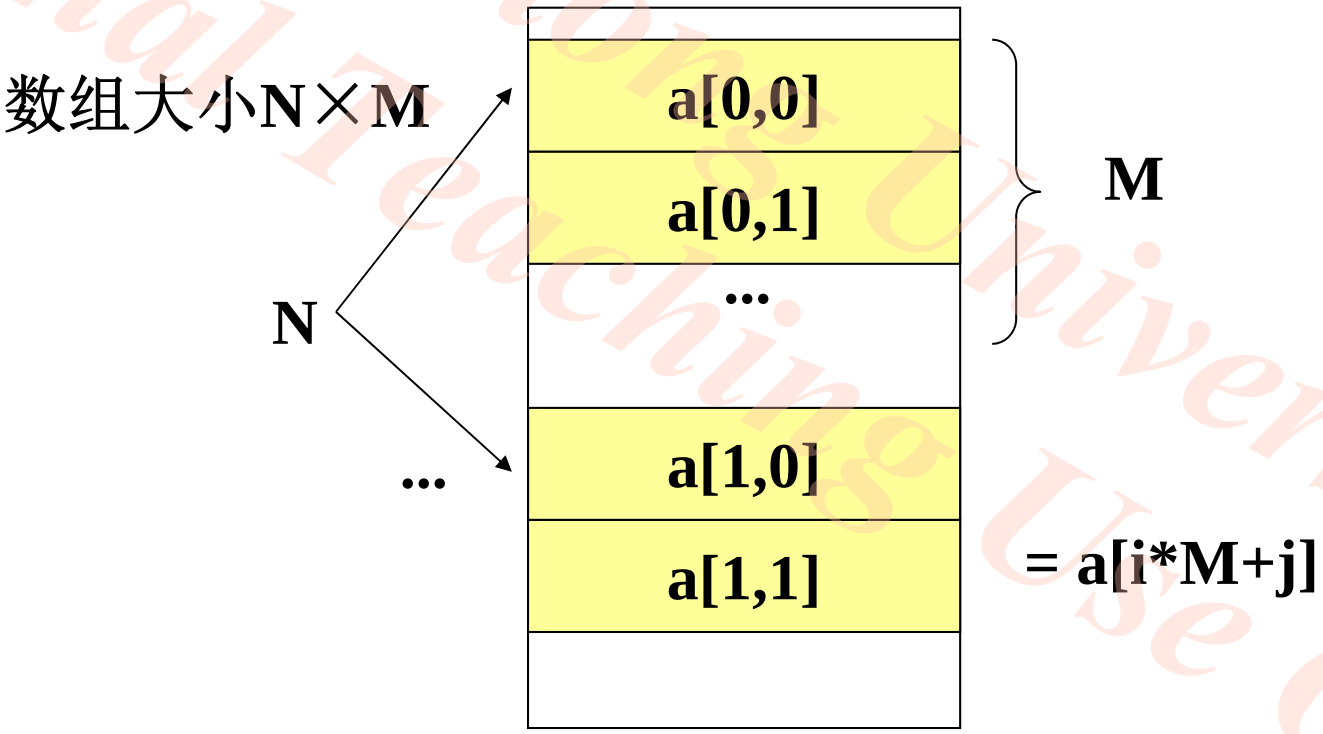
一维数组

- 一维数组**a[i]**
  - 第**0**个元素存储数组的第一个元素



二维数组

- 二维数组a[i,j]：i代表行，j代表列
- 行为主 vs. 列为主

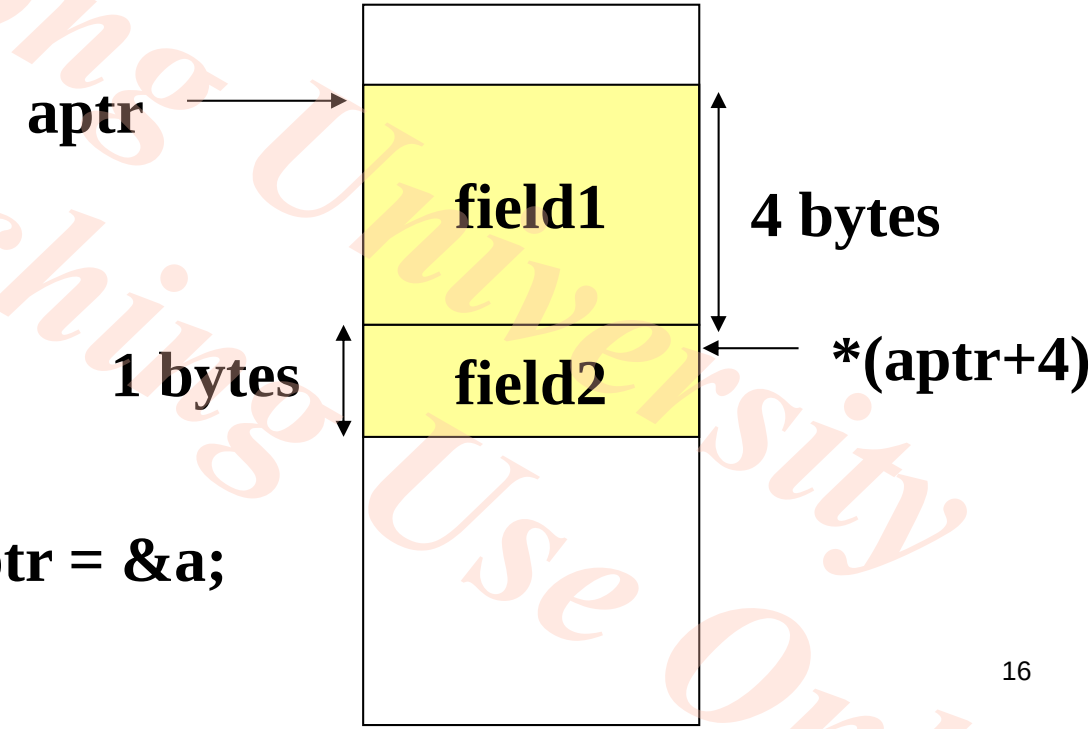


结构体

- 结构体由一个连续的存储块来实现。对结构体内的字段的访问可以通过结构体基地址+偏移量来访。

```
struct {
   int field1;
   char field2;
} mystruct;

struct mystruct a, *aptr = &a;
```



5 程序优化

- 表达式简化
- 无效代码的清除
- 过程内嵌
- 循环变换
- 寄存器分配
- 调度
- 指令选择

表达式简化

- 表达式简化是对独立于机器的变换很有效。
- 利用代数法则：
  - **a*b + a*c = a*(b+c)**　分配律使3操作简化成2操作
- 常量叠算（**constants folding**）：
  - **8+1 = 9**
  - **for (i=0; i<8+1; i++)**　　**for (i=0; i<NOPS+1; i++)**
- 强度化简:
  - **a*2 = a<<1**

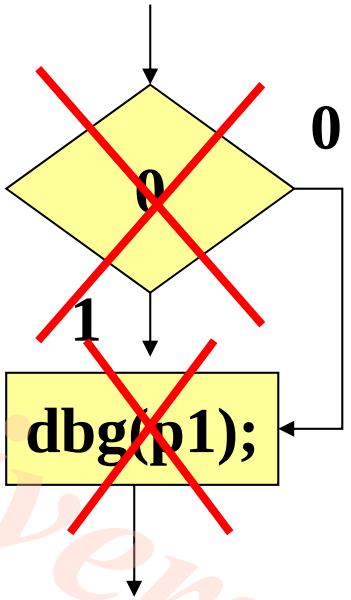
无效代码清除

- 无效代码:

```
#define DEBUG 0
if (DEBUG) dbg(p1);
```

- 可以通过控制流分析、常量叠算来消除。

- 有些无效代码是编译器引入的。



过程内嵌 procedure inlining

- 过程内嵌属于独立于机器的变换，可以去除过程链接造成的耗费。

```
int foo(a,b,c) { return a + b - c;}
z = foo(w,x,y);
```

⇨

```
z = w + x + y;
```

- 是否使用过程内嵌需要评估，不一定有益：
  - 函数多份拷贝可能造成**cache**冲突，减慢指令的存取速度
  - 增加代码量，增加存储器开销

循环变换

- 循环虽然在源代码中描述的很紧凑，但通常占用大量的计算时间。

- 有很多优化循环技术，其目标：
  - 减少循环的耗费；
  - 增加流水线执行的机会；
  - 改善存储系统的性能。

循环展开

- 减少循环的开销，使能后续的优化，比如指令并行。

```
for (i=0; i<4; i++)
    a[i] = b[i] * c[i];
```

⇨

```
for (i=0; i<2; i++) {
    a[i*2] = b[i*2] * c[i*2];
    a[i*2+1] = b[i*2+1] * c[i*2+1];
}
```

```
a[0] = b[0] * c[0];
a[1] = b[1] * c[1];
a[2] = b[2] * c[2];
a[3] = b[3] * c[3];
```

循环融合与分布

- 循环融合将两个或多个循环合并成一个：

```
for (i=0; i<N; i++) a[i] = b[i] * 5;
for (j=0; j<N; j++) w[j] = c[j] * d[j];
⇨ for (i=0; i<N; i++) {
    a[i] = b[i] * 5; w[i] = c[i] * d[i];
  }
```

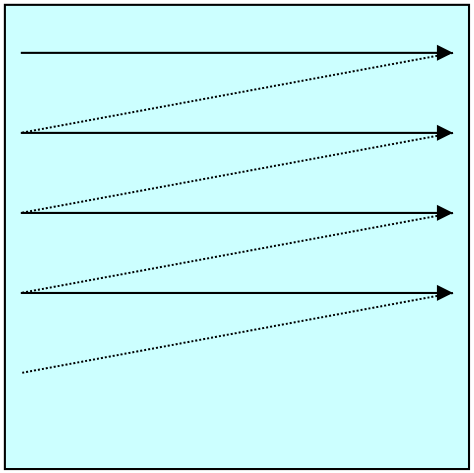
- 循环分布将一个循环分解成多个。

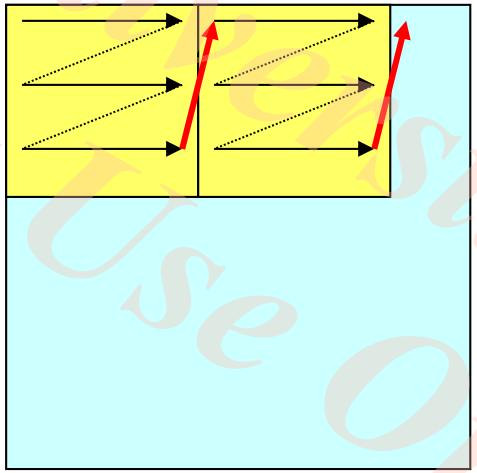
循环交叠 loop tiling

- 将一个循环拆分成一系列嵌套循环，改善**cache**

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        c[i] = a[i,j]*b[i];
```

```
for (i=0; i<N; i+=2)
  for (j=0; j<N; j+=2)
    for (ii=0; ii<min(i+2,n); ii++)
      for (jj=0; jj<min(j+2,N); jj++)
        c[ii] = a[ii,jj]*b[ii];
```





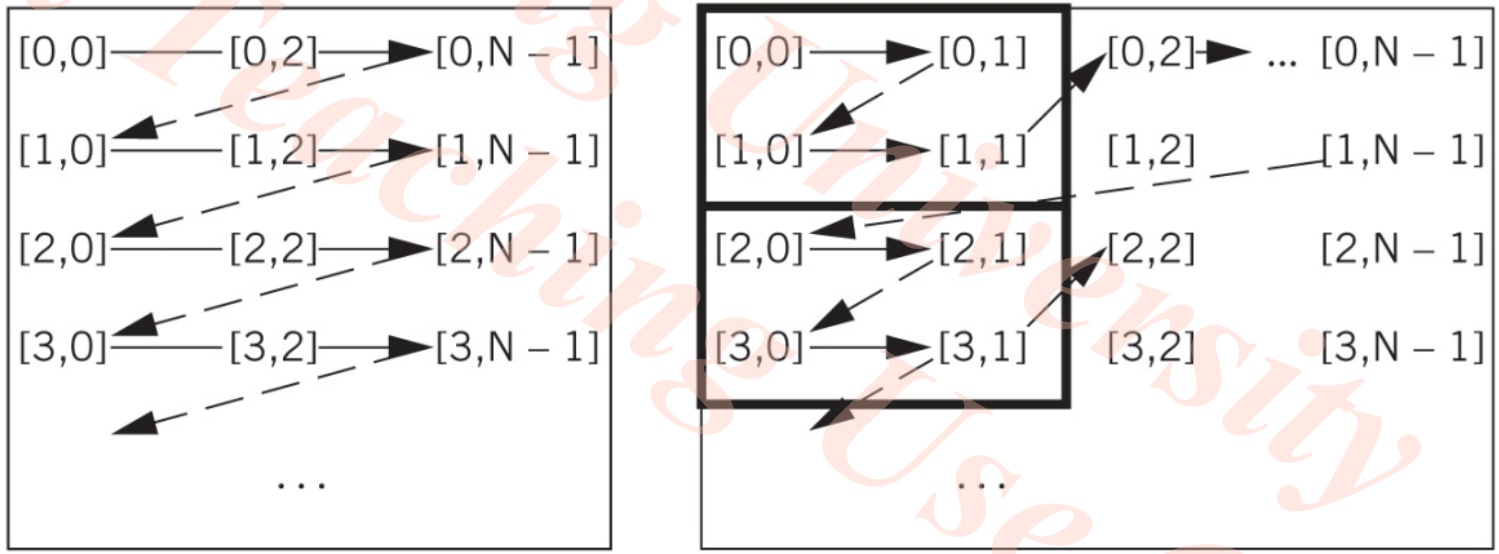
循环交叠

Code

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        c[i] = a[i,j] * b[i];
```

```
for (i = 0; i < N; i += 2)
    for (j = 0; j < N; j += 2)
        for (ii = i; ii < min(i + 2,N); i++)
            for (jj = j; jj < min(j + 2,N); j++)
                c[ii] = a[ii,jj] * b[ii];
```

Access pattern in a array



Before **After**

数组填充 Array padding

- 向循环中添加哑数据元素，改变数组在高速缓存中的布局，降低高速缓存冲突。

a[0,0]	a[0,1]	a[0,2]
a[1,0]	a[1,1]	a[1,2]

before

a[0,0]	a[0,1]	a[0,2]	a[0,2]
a[1,0]	a[1,1]	a[1,2]	a[1,2]

after

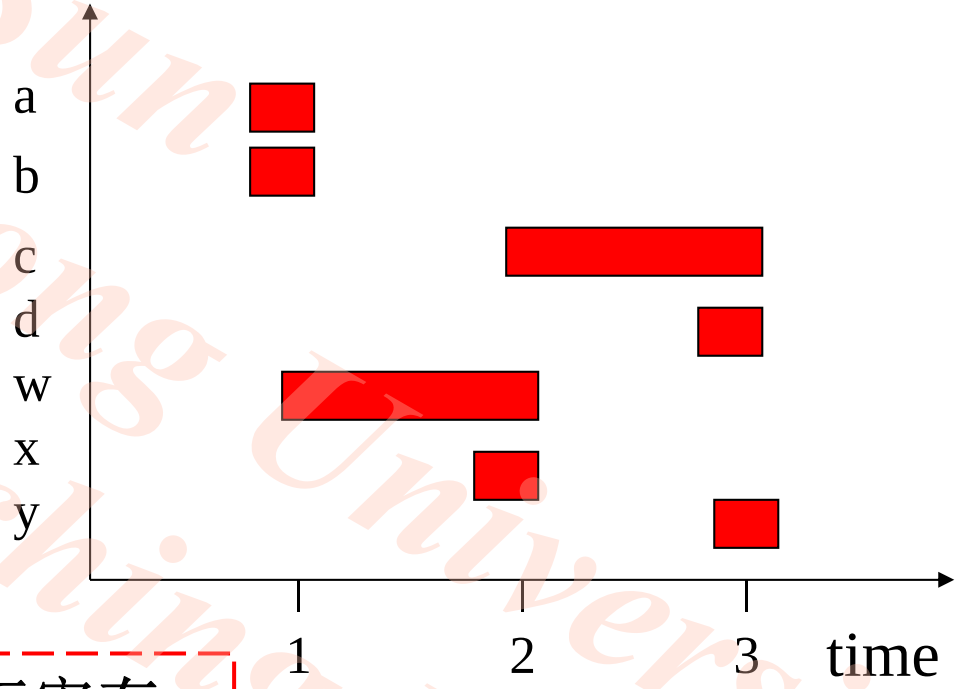
寄存器分配

- 目标:
  - 选择寄存器来保存每个变量
  - 决定变量在寄存器中的生命周期
  - 控制变量（包括声明的和临时的）在寄存器上的分配以最小化所需的寄存器总数。

- 用一个基本语句块的例子来说明

寄存器生命周期图 lifetime graph

```
w = a + b;   t=1
x = c + w;   t=2
y = c + d;   t=3
```



如果按照变量的个数进行寄存器分配，将每个变量分配给独立的寄存器，需要**7**个寄存器。

优化的寄存器分配

A	r0
B	r1
C	r2
D	r0
W	r3
X	r0
Y	r3

通过重用寄存器，我们可以编写需要不到**4**个寄存器的代码。

```
LDR r0, [p_a]
LDR r1, [p_b]
ADD r3, r0, r1
STR r3, [p_w]
LDR r2, [p_c]
ADD r0, r2, r3
STR r0, [p_x]
LDR r0, [p_d]
ADD r3, r2, r0
STR r3, [p_y]
```

寄存器溢出

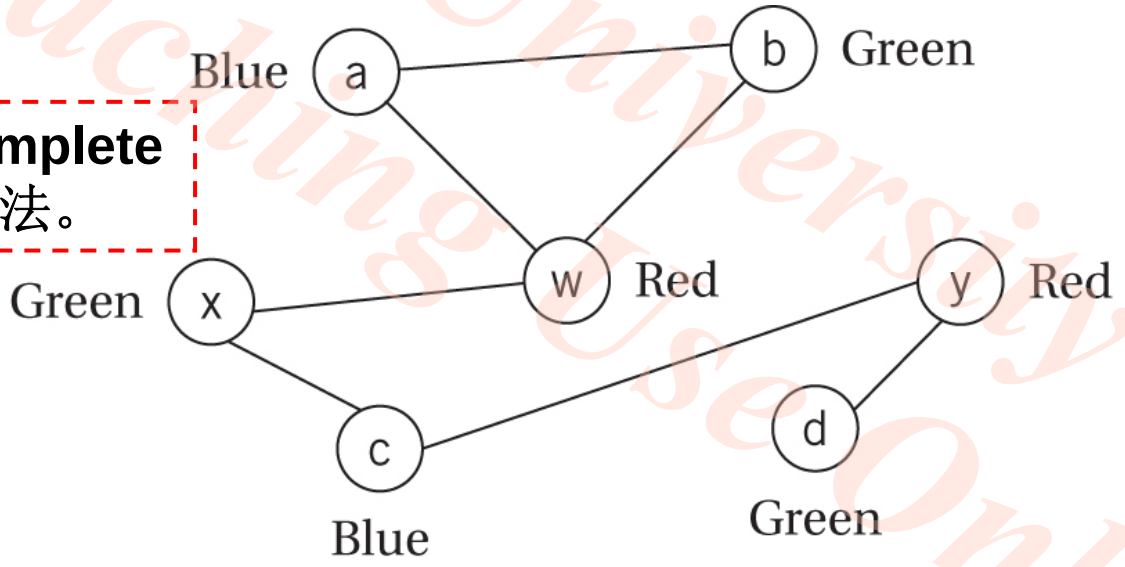
- 如果一段代码所需的寄存器数目超过可用的寄存器数，我们必须临时将一些值溢出（**spill**）到内存。
- 计算出的一些值后，我们可以将其写入临时存储单元，在其他计算中重用这些寄存器，然后重新从临时存储单元读入以前的数值继续进行计算。
- 寄存器溢出的问题：
  - 需要额外的**CPU**时间
  - 增加指令和数据存储器的使用

优化寄存器分配，避免不必要的寄存器溢出是有价值的。

寄存器分配问题

- 通过建立冲突图（**conflict graph**）并解一个图的着色问题来解决寄存器分配问题。
- 每个变量由一个节点表示，如果两个变量有相同的生存期，就在两个点之间添加一条边。
- 图的着色问题就是用最少的颜色给全部节点着色，要求两个相同颜色的节点之间没有一条直接相连的边。

图的着色问题是**NP-complete**问题，但是有启发式算法。



改善寄存器分配的操作调度

(a + b)*(c - d)

示例**5.6**

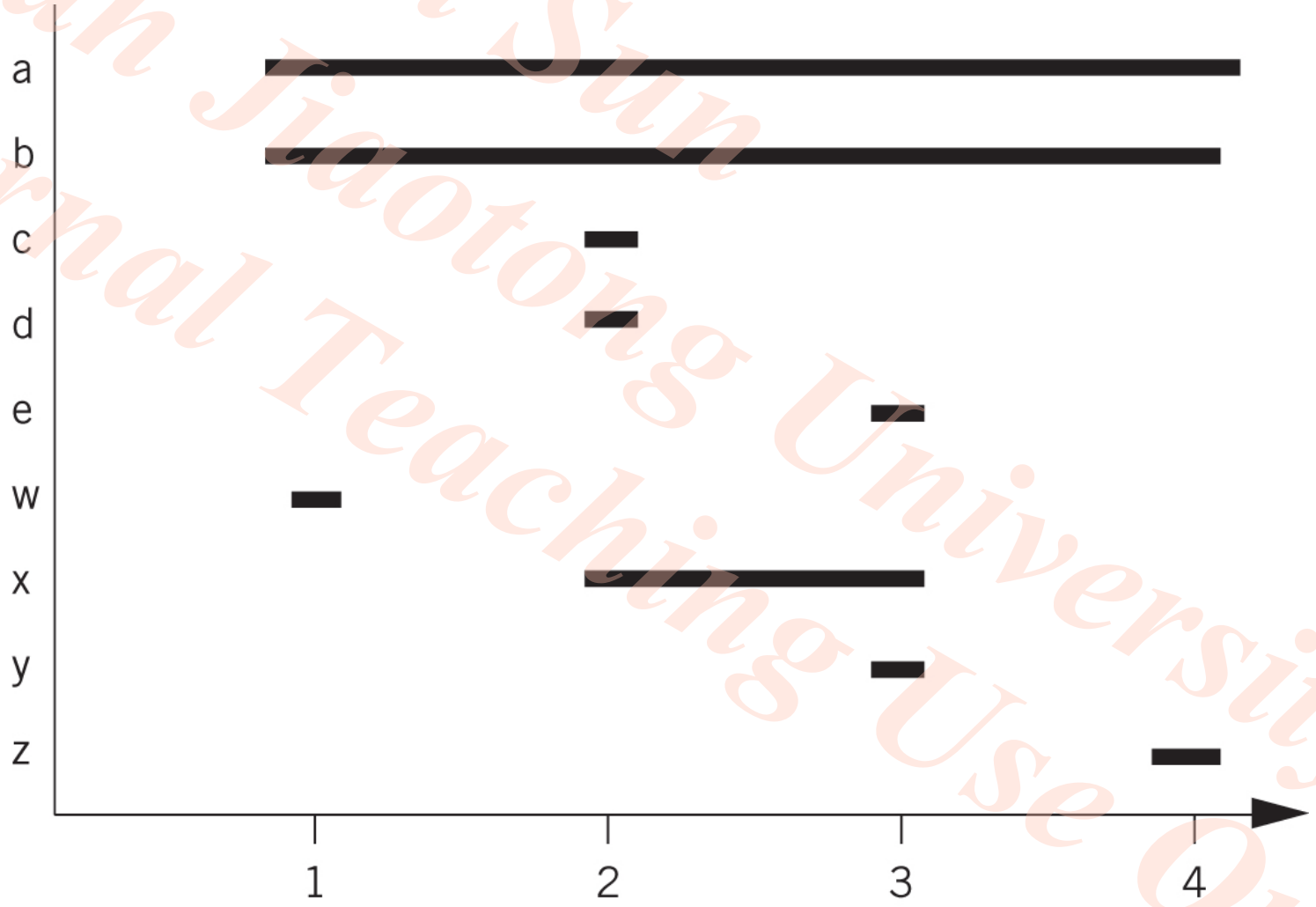
w = a + b

x = c + d

y = x + e

z = a - b

寄存器生命周期图



汇编代码（4个寄存器）

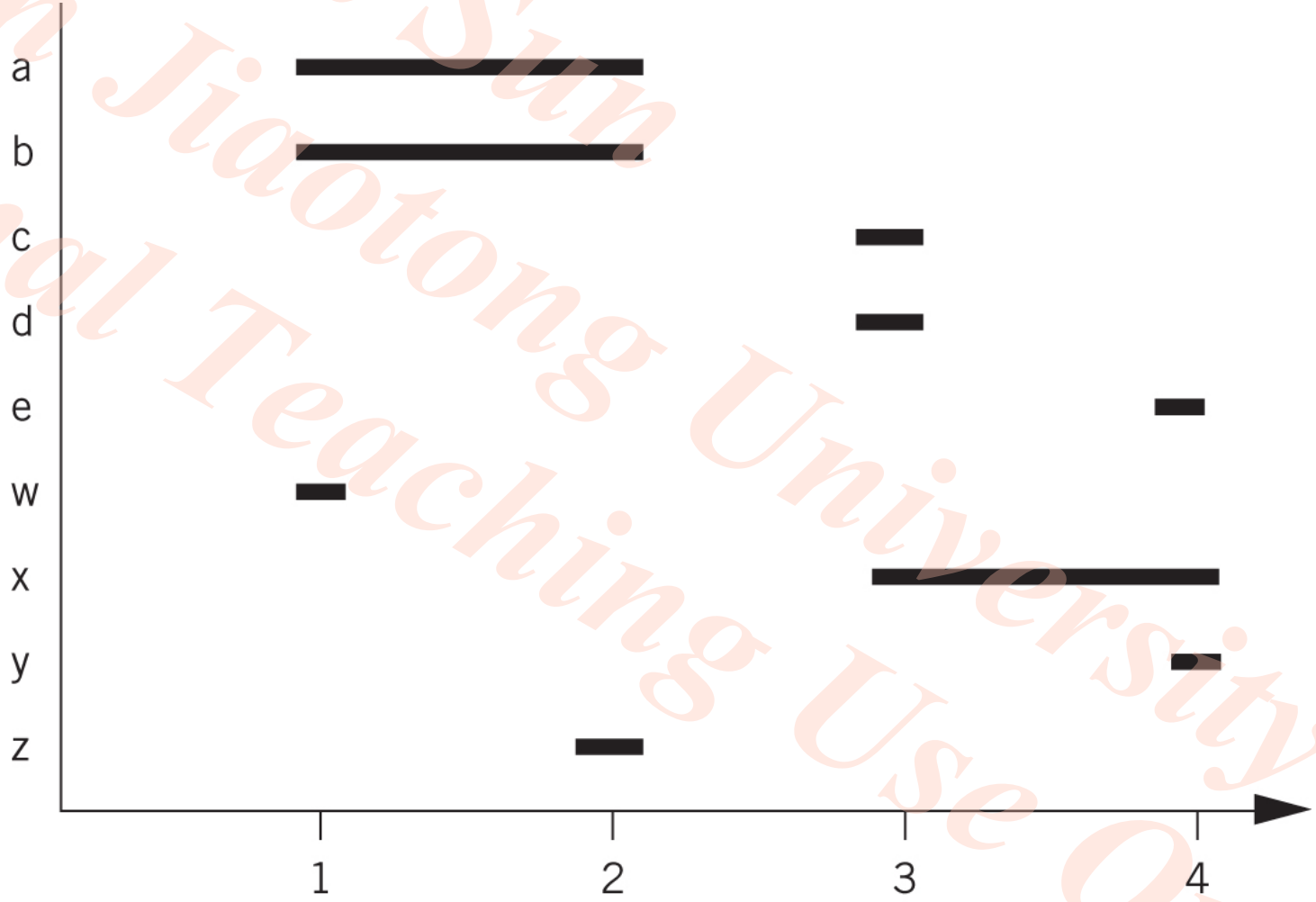
```
LDR r0,a
LDR r1,b
ADD r2,r0,r1
STR r2,w    ; w = a + b
LDRr r0,c
LDR r1,d
ADD r2,r0,r1
STR r2,x    ; x = c + d
```

```
LDR r1,e
ADD r0,r1,r2
STR r0,y    ; y = x + e
LDR r0,a    ; reload a
LDR r1,b    ; reload b
SUB r2,r1,r0
STR r2,z    ; z = a – b
```

操作调度后的C代码

```
w = a + b
z = a - b
x = c + d
y = x + e
```

调度后的生命周期图



调度后的汇编代码

```
LDR r0,a
LDR r1,b
ADD r2,r1,r0
STR r2,w ; w = a + b
SUB r2,r0,r1
STR r2,z ; z = a – b
LDR r0,c
LDR r1,d
ADD r2,r1,r0
STR r2,x ; x = c + d
LDR r1,e
ADD r0,r1,r2
STR r0,y ; y = x + e
```

调度预约表 reservation table

- 在指令调度期间用预约表跟踪**CPU**资源使用情况。
- 调度指令执行前，要检查预约表确定所有指令所需的资源都是可用的。
- 调度后，更新预约表

Time/instr	A	B
instr1	X	
instr2	X	X
instr3	X	
instr4		X

指令调度

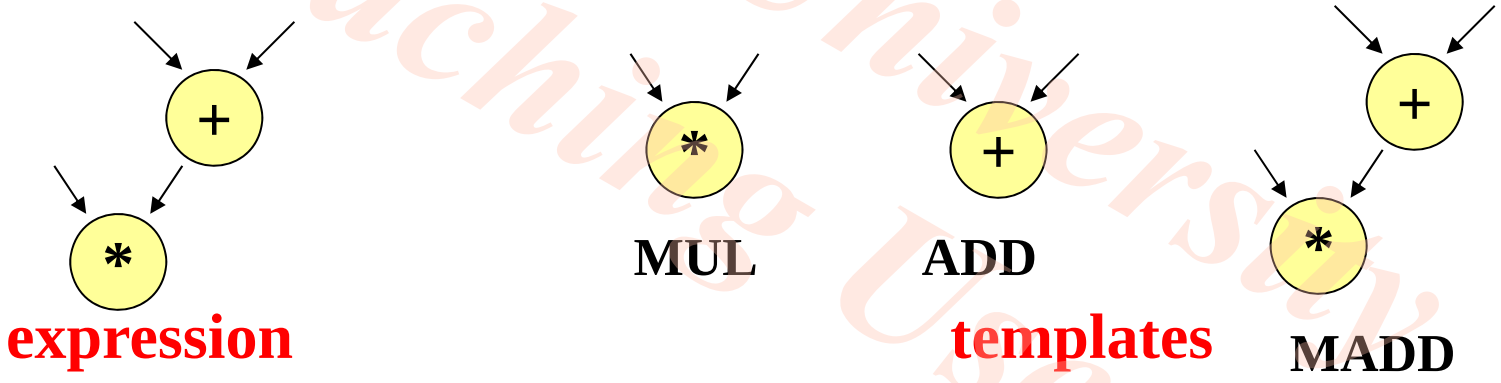
- **Non-pipelined machines do not need instruction scheduling: any order of instructions that satisfies data dependencies runs equally fast.**
- **In pipelined machines, execution time of one instruction depends on the nearby instructions: opcode, operands.**

软件流水 software pipeline

- 流水线中，如果指令执行所花费的周期比正常情况下多，就会出现流水线气泡导致性能降低
- 软件流水是一种对指令重排序的技术，跨越几个循环迭代来减少流水线气泡。
- **Reduces instruction latency in iteration i by inserting instructions from iteration i+1.**

指令选择

- 执行一个操作或一系列操作可能有几种不同的指令。执行时间/对邻近代码的影响。
- 模板匹配（**template matching**）：将操作用图表示，使用可能的指令序列进行匹配。



理解并使用你的编译器

- 研究编译器的汇编语言输出是了解编译器的好方式。
- 理解并使用编译器不同的优化级别 **(-O1, -O2, etc.)**
- 尝试使用不同方式实现同一算法，观察编译结果的区别。
- 在编译输出结果上修改，但是要注意：
  - 正确性
  - 记录手工修改

解释器与JIT编译器

- 解释器（Interpreter）：实时的翻译和执行代码.
- JIT编译器：Java与JIT