

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
任鹏举



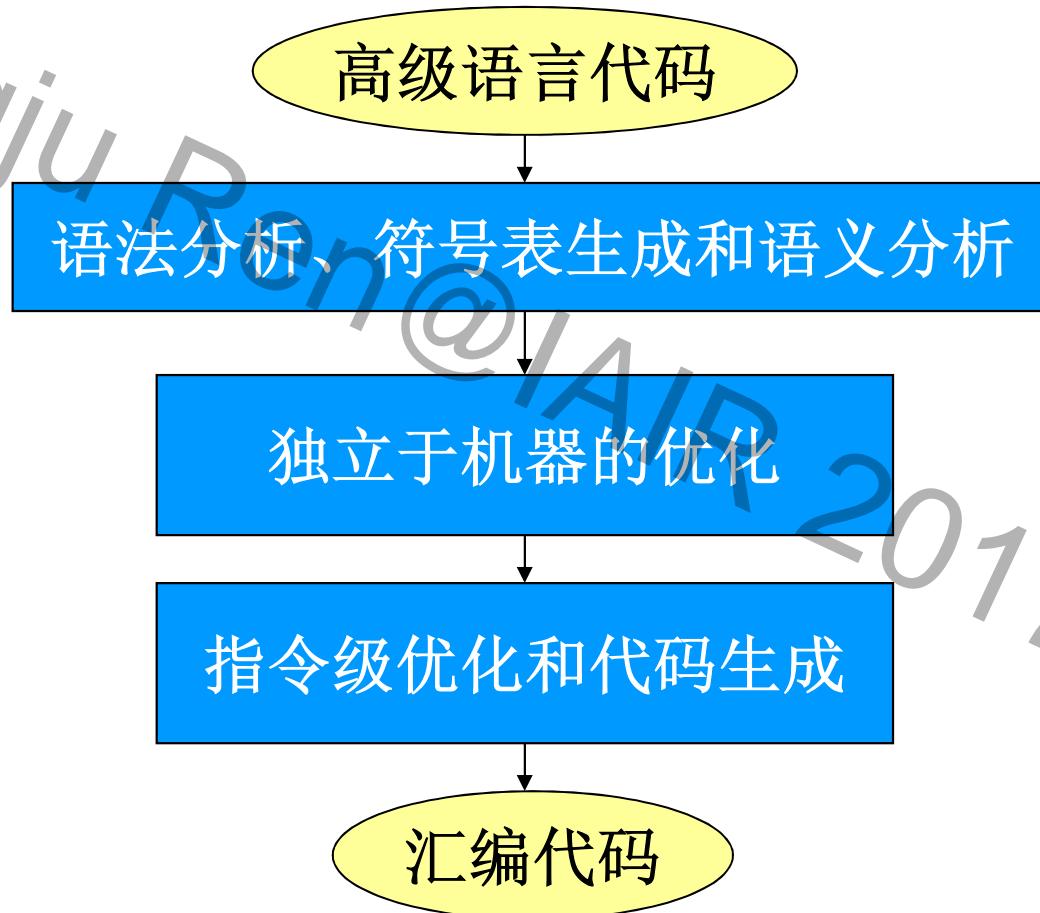


4 基本编译技术

- 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
正确的理解程序（功能）
 - 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，优化可以生成更高效指令的代码。
高效的执行程序（性能）
- 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



语句的翻译与优化

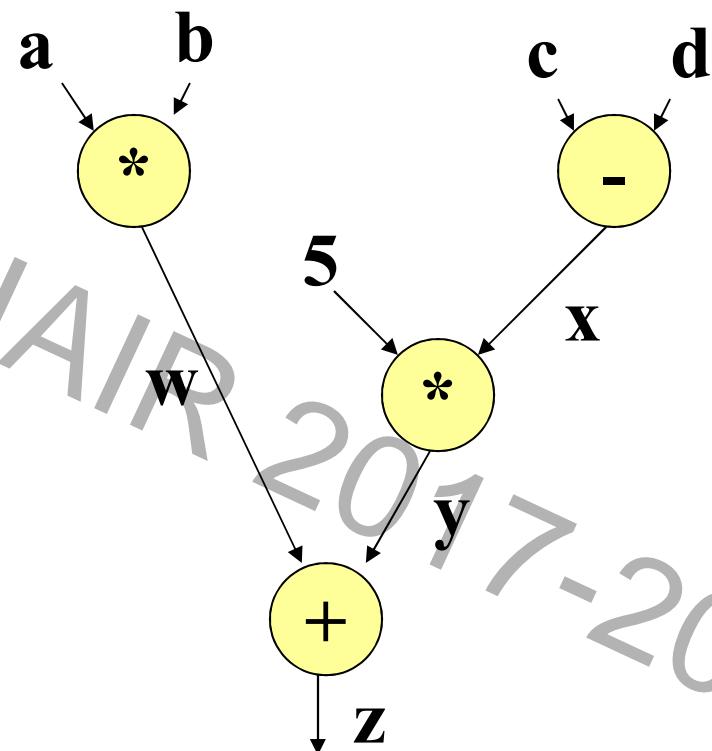
- 算术表达式;
- 控制结构-条件语句;
- 过程链接;
- 数据结构;



算术表达式 (1)

$$a^*b + 5^*(c-d)$$

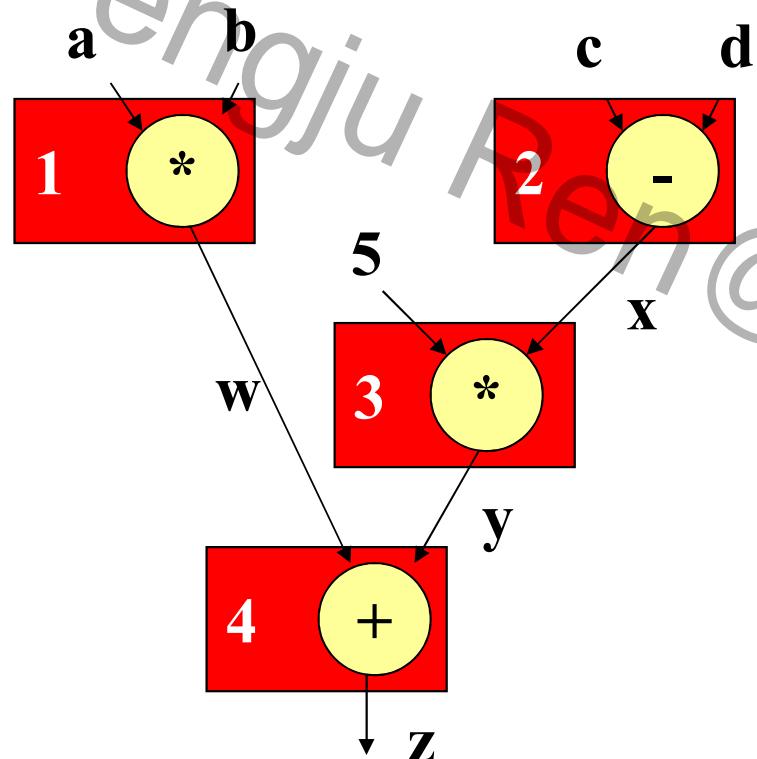
expression



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

$a/b + c*(d+e) - f^g + \log_2(h+7i) + \sin(10g+k/3) \dots$

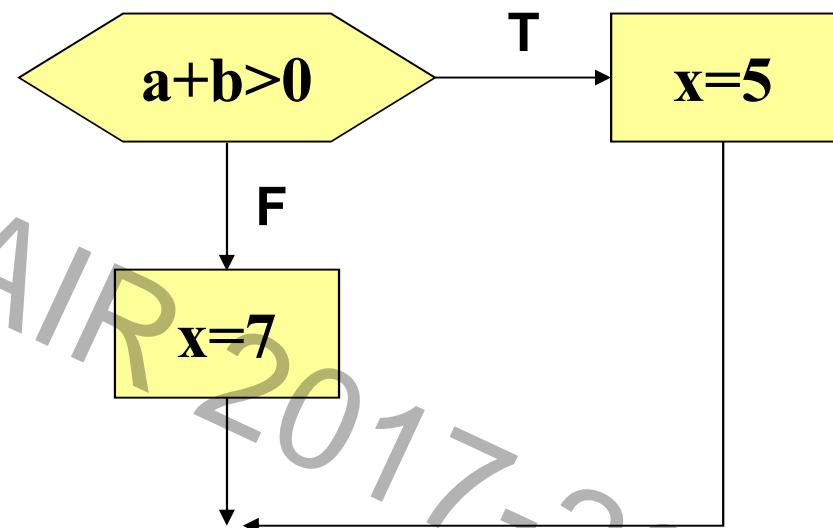
一个明显的优化方法是重用哪些其值不用再保存的寄存器。

寄存器分配问题！



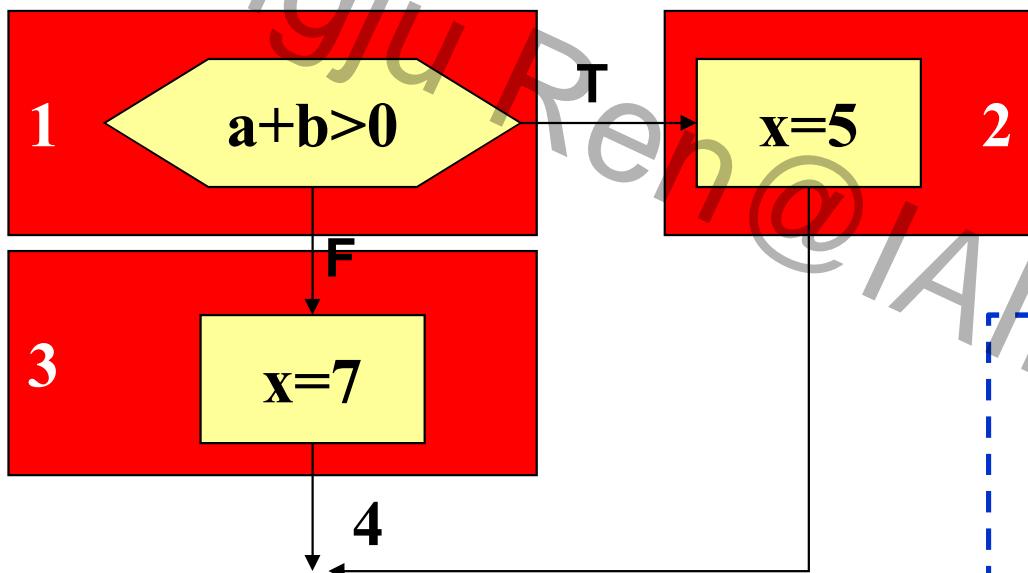
控制结构-条件语句（1）

```
PengJU Ren@IAIR 2017_2018  
if (a+b > 0)  
    x = 5;  
else  
    x = 7;
```





控制结构-条件语句（2）



ADR r5,a
LDR r1,[r5]
ADR r5,b
LDR r2,[r5]
ADD r3,r1,r2

BLE label3 *Ture block*
LDR r3,#5

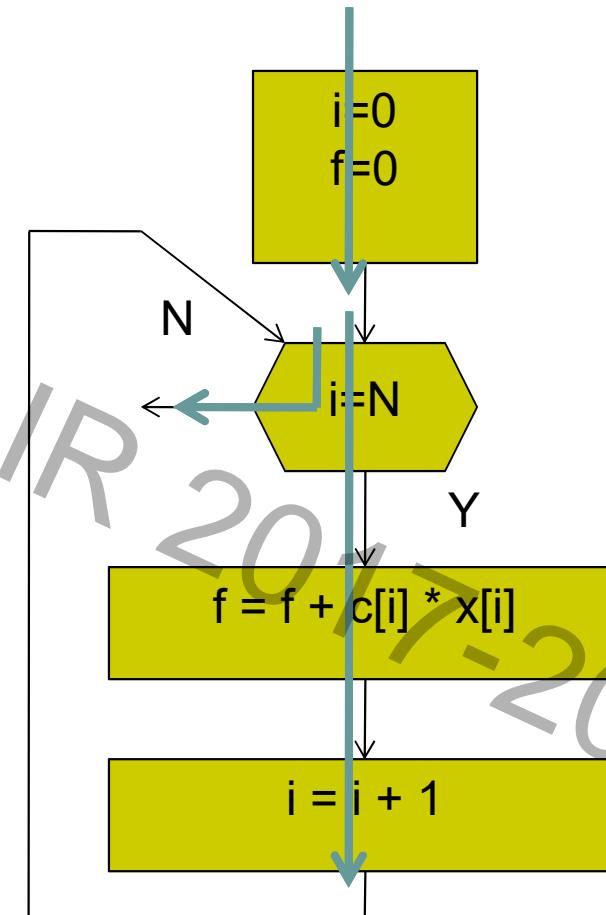
ADR r5,x
STR r3,[r5]
B stmtend

label3 LDR r3,#7
ADR r5,x
STR r3,[r5]
stmtend ... *False block*



控制结构-While语句

```
for (i=0, f=0; i<N; i++)  
    f = f + c[i] * x[i];
```





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。
- 帧栈的具体构造和数据结构取决于不同的**CPU (Machine dependent)**
- 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



堆栈回顾

- 对于高级语言程序而言，堆栈的操作往往对程序员是不可见的，对于大多数的汇编而言，需要程序员指定相应堆栈的操作。
- 堆栈的具体细节取决于：编译器，OS以及指令集



过程栈

Diagram illustrating the stack frame structure:

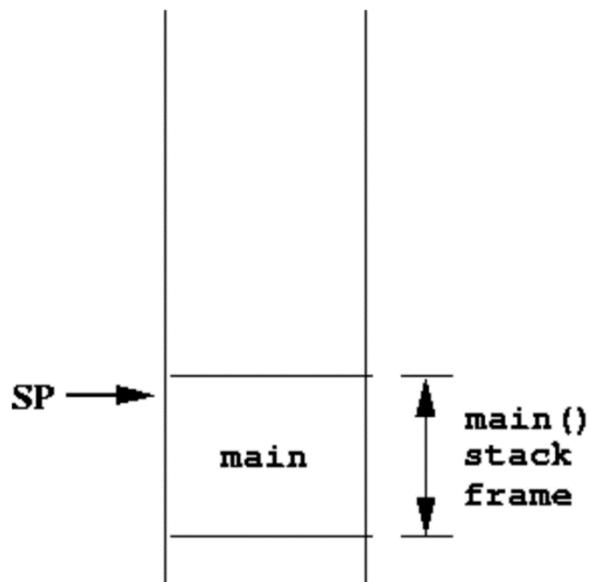
- frame pointer (FP)**: Points to the top of the current frame (proc1).
- stack pointer (SP)**: Points to the bottom of the current frame (proc2).
- The stack grows **downwards**.
- proc1** contains the call to **proc2(5);**
- proc2** contains the value **5**.
- A brace indicates the **accessed** memory relative to the **SP**.

```
proc1(int a) {  
    proc2(5);  
}  
2017-2018
```



栈帧对于程序调用的支持(1)

- 栈帧(**Stack Frames**)有时也叫做活动记录(**Activation records**);
- 对于每个函数调用，为函数保留一部分堆栈，通常称为堆栈帧(**stack frame**)。

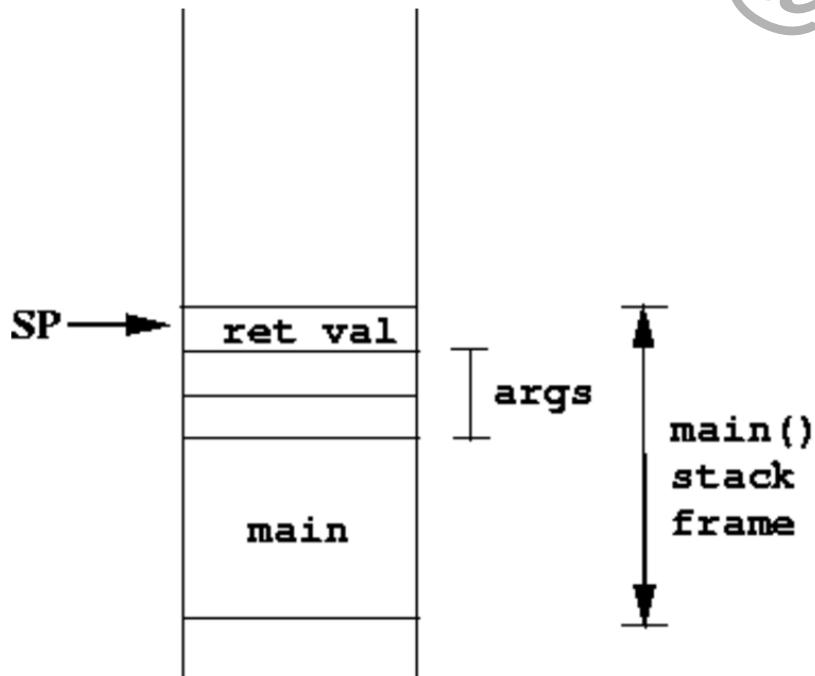


A stack frame exists whenever a function has started, but yet to complete.
每当一个函数启动但尚未完成时，就存在一个栈帧。



栈帧对于程序调用的支持(2)

- 栈帧负责子程序调用未返回的信息记录：
 - 局部变量(**Local variables**)
 - 调用函数的返回地址 (**Caller's address**)
 - 传递参数(**Parameter values**)



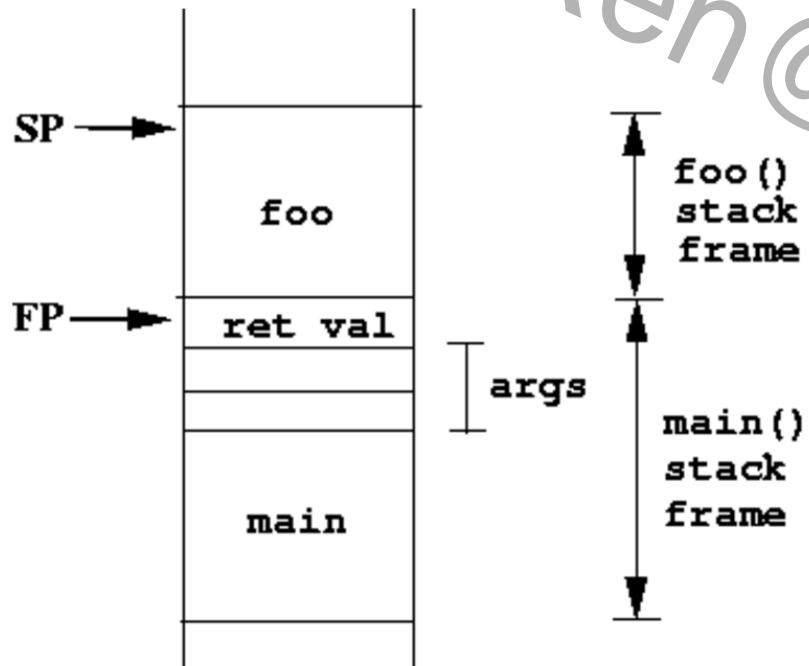
Suppose, inside of body of `main()`, there is a call to `foo(a,b)`, which needs two arguments. Then, it the `main()` to take the response to “push” arguments for `foo()` onto the stack. We also reserved some space for the return value. The return value is computed by `foo()`, so it will be filled out once `foo()` is done.

假设在`main()`体内部有一个对 `foo(a, b)` 的调用，它需要两个参数。然后，`main()`将`foo()` 的参数“推”到堆栈上。我们还保留了一些空间作为返回值。返回值由 `foo()` 计算，所以一旦`foo()`完成，它就会被填充。



栈帧对于程序调用的支持(3)

- 帧栈指针(Frame points): 方便函数的调用(args)



*We've added a new pointer called FP which stands for **frame pointer**. The FP points to the location where the stack pointer was, just before foo() moved the stack pointer for foo()'s own local variables.*

我们添加了一个名为**FP**的新指针，代表帧指针。帧指针指向堆栈指针所在的位置，就在**foo()**移动**foo()**自己局部变量的堆栈指针之前。

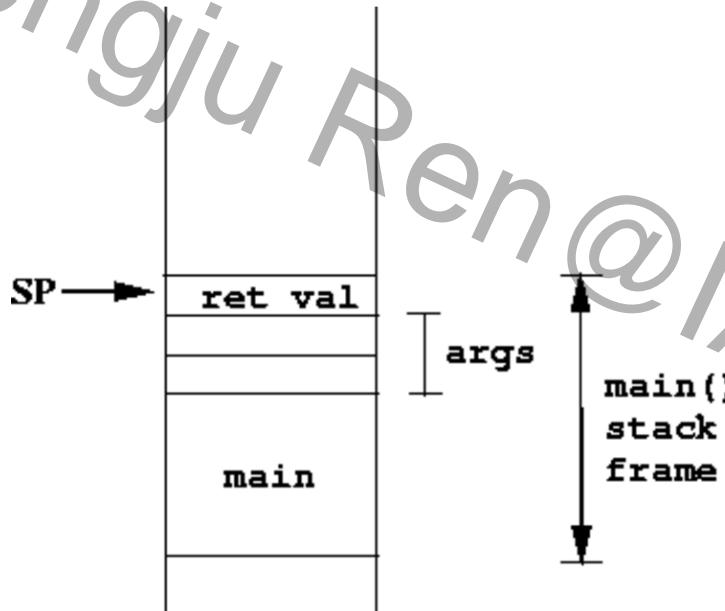
*we can use the FP to compute the locations in memory for both arguments as well as local variables. Since it doesn't move, the computations for those locations should be some **fixed offset** from the frame pointer.*

我们可以使用帧指针来计算内存中两个参数以及局部变量的位置。由于它不移动，这些位置的计算应该是从帧指针的某个固定的偏移量。



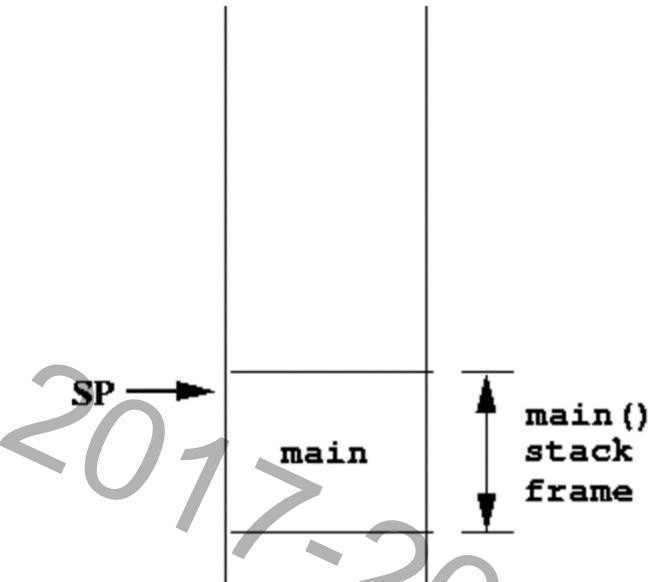
栈帧对于程序调用的支持(4)

- 帧栈指针(Frame points): 方便函数的返回(ret val)



Once it's time to exit foo(), you just have to set the stack pointer to where the frame pointer is, which effectively pops off foo()'s stack frame. It's quite handy to have a frame pointer.

一旦退出 **foo()**, 您只需将堆栈指针设置为帧指针所在的位置, 就可以有效地将 **foo()** 的堆栈框架弹出。

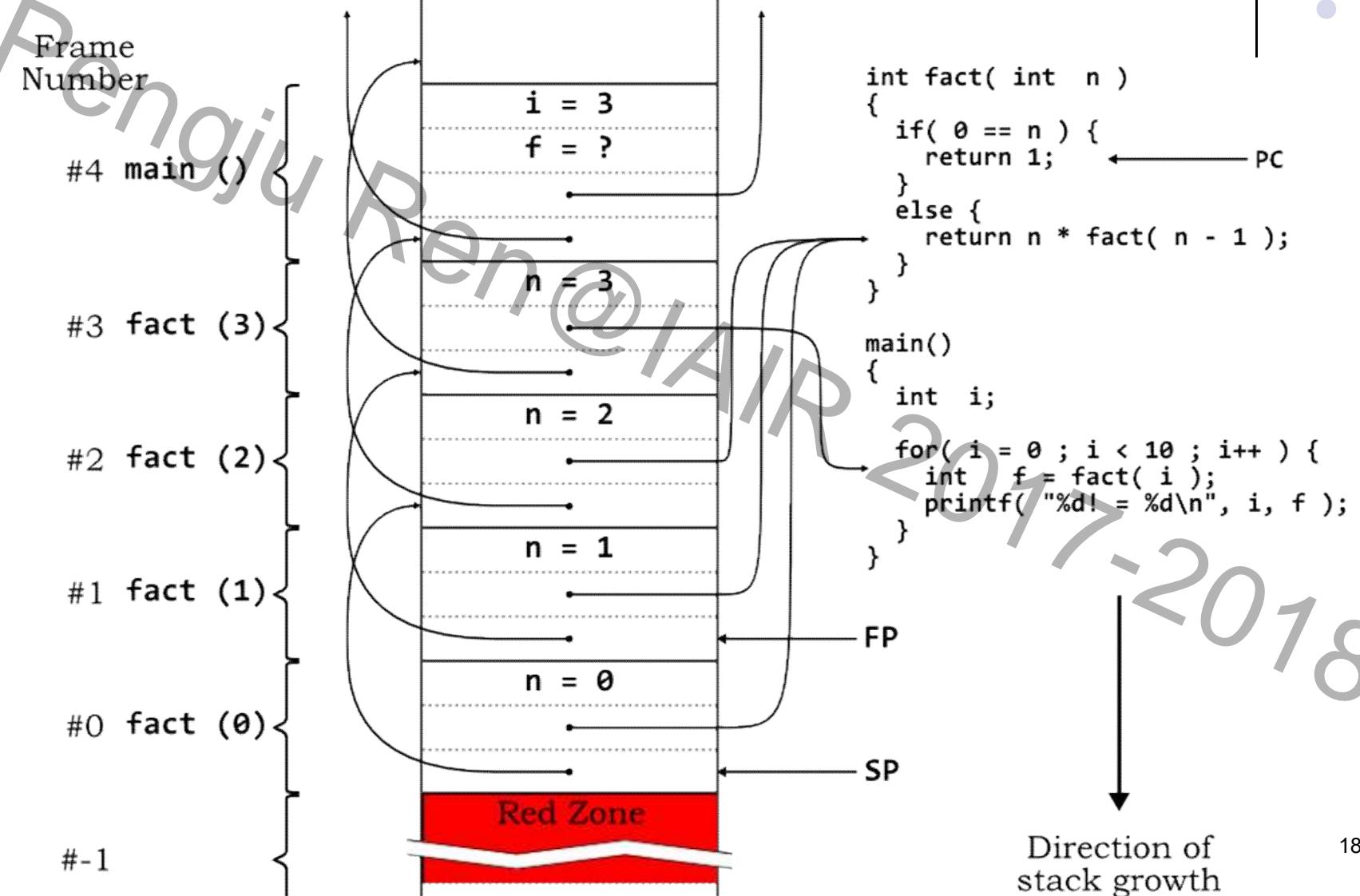


Once main() has the return value, it can pop that and the arguments to foo() off the stack.

一旦 **main()** 获得了返回值, 它可以将返回值和 **foo()** 的传递参数从堆栈中弹出。



栈帧的嵌套示例





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。



ARM过程链接

- ARM过程调用标准(ARM Procedure Call Standard, APCS)

专用寄存器
Dedicated registers

寄存器变量
callee-saved
register variables

参数传递
caller-saved
argument variables

| Register | Synonym | Special | Role in the procedure call standard |
|----------|---------|----------------|---|
| r15 | | PC | The Program Counter. |
| r14 | | LR | The Link Register. |
| r13 | | SP | The Stack Pointer. |
| r12 | | IP | The Intra-Procedure-call scratch register. |
| r11 | v8 | | Variable-register 8. |
| r10 | v7 | | Variable-register 7. |
| r9 | | v6 SB TR | Platform register. The meaning of this register is defined by the platform standard. |
| r8 | v5 | | Variable-register 5. |
| r7 | v4 | | Variable register 4. |
| r6 | v3 | | Variable register 3. |
| r5 | v2 | | Variable register 2. |
| r4 | v1 | | Variable register 1. |
| r3 | a4 | | Argument / scratch register 4. |
| r2 | a3 | | Argument / scratch register 3. |
| r1 | a2 | | Argument / result / scratch register 2. |
| r0 | a1 | | Argument / result / scratch register 1. |

Table 2, Core registers and AAPCS usage



Stack Backtrace (FP based)

MOV ip, sp ; save current sp, ready to save as old sp;

STMFD sp!, {a1-a4, v1-v5, sb, fp, ip, lr, pc};

SUB fp, ip, #4 The stack backtrace data structure has the format shown below:

SUB sp,sp,#16

| | | |
|-------------------|-------------|---------------------|
| save code pointer | [fp] | <-fp points to here |
| return link value | [fp, #-4] | |
| return sp value | [fp, #-8] | |
| return fp value | [fp, #-12] | |
| [saved v7 value] | | |
| [saved v6 value] | | |
| [saved v5 value] | | |
| [saved v4 value] | | |
| [saved v3 value] | | |
| [saved v2 value] | | |
| [saved v1 value] | | |
| [saved a4 value] | | |
| [saved a3 value] | | |
| [saved a2 value] | | |
| [saved a1 value] | | |
| [saved f7 value] | three words | |
| [saved f6 value] | three words | |
| [saved f5 value] | three words | |
| [saved f4 value] | three words | |

use the frame pointer to compute the locations in memory for both arguments as well as local variables. Since it doesn't move, the computations for those locations should be some fixed offset from the frame pointer.

使用帧指针来计算内存中的位置以及局部变量。由于它不移动，这些位置的计算应该是从帧指针的某个固定的偏移量。



C的过程链接

```
int p1 (int a, int b, int c, int d, int e) {return a+e;}
```

```
mov ip, sp  
stmfd sp!, {fp, ip, lr, pc}  
sub fp, ip, #4  
sub sp, sp, #16  
str r0, [fp, #-16]  
str r1, [fp, #-20]  
str r2, [fp, #-24]  
str r3, [fp, #-28]  
ldr r2, [fp, #-16]  
ldr r3, [fp, #4]  
add r3, r2, r3  
mov r0, r3  
ldmea fp, {fp, sp, pc}
```

```
y = p1 (a,b,c,d,x);  
  
ldr r3, [fp, #-32] ; get e  
str r3, [sp, #0] ; put into p1()'s stack frame  
ldr r0, [fp, #-16] ; put a into r0  
ldr r1, [fp, #-20] ; put b into r1  
ldr r2, [fp, #-24] ; put c into r2  
ldr r3, [fp, #-28] ; put d into r3  
bl p1 ; call p1()  
mov r3, r0 ; move return value into r3  
str r3, [fp, #-36] ; store into y in stack frame
```



C的过程链接

Caller 调用者

传递N个参数

r0,r1,r2,r3做参数传递
If N>4: 多余传递参数
压入堆栈

Callee 被调用者

保护相关指针

将fp, ip, lr, pc入栈；更新sp->ip->fp

使用M个寄存器

1:分配栈帧空间 ($sp - 4 * M$)
2:将M个寄存器数值入栈(保护现场)

实际使用X个传递参数

使用r0~r3;
如果有必要从堆栈中取值

产生Y个返回值

返回值压栈或使用r0传递

恢复相关指针

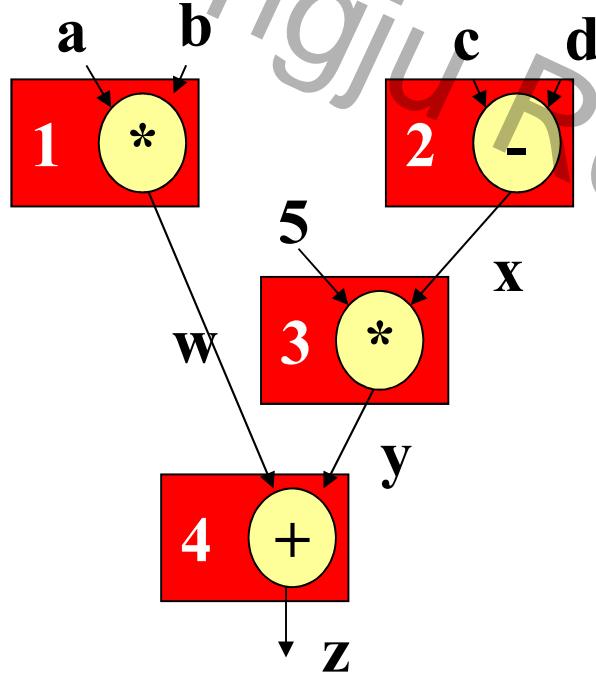
将fp, ip, lr, pc出栈

获取Y个返回值

返回值出栈



Review: 算术表达式



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

$X = a*b + 5*(c-d)$
Function $X(a,b,c,d)$

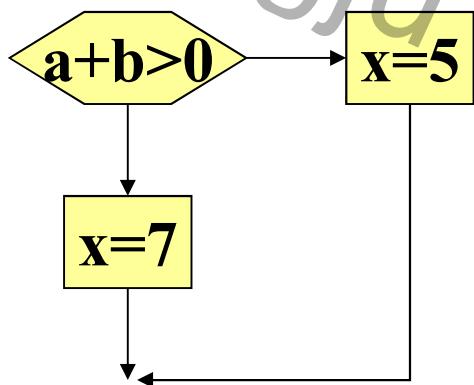
```

ldr r2, [fp, #-16]
ldr r3, [fp, #-20]
mul r1, r3, r2 ; multiply
ldr r2, [fp, #-24]
ldr r3, [fp, #-28]
rsb r2, r3, r2 ; subtract
mov r3, r2
mov r3, r3, asl #2
add r3, r3, r2 ; add
add r3, r1, r3 ; add
str r3, [fp, #-32] ; assign

```



Review: 条件语句



**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 ...

Function fun1(a,b);

```
ldr r2, [fp, #-16]
ldr r3, [fp, #-20]
add r3, r2, r3
cmp r3, #0 ; test the branch condition
ble .L3 ; branch to false block if <=
mov r3, #5 ; true block
str r3, [fp, #-32]
b .L4 ; go to end of if statement
.L3: ;
false block
mov r3, #7
str r3, [fp, #-32]
.L4:
```



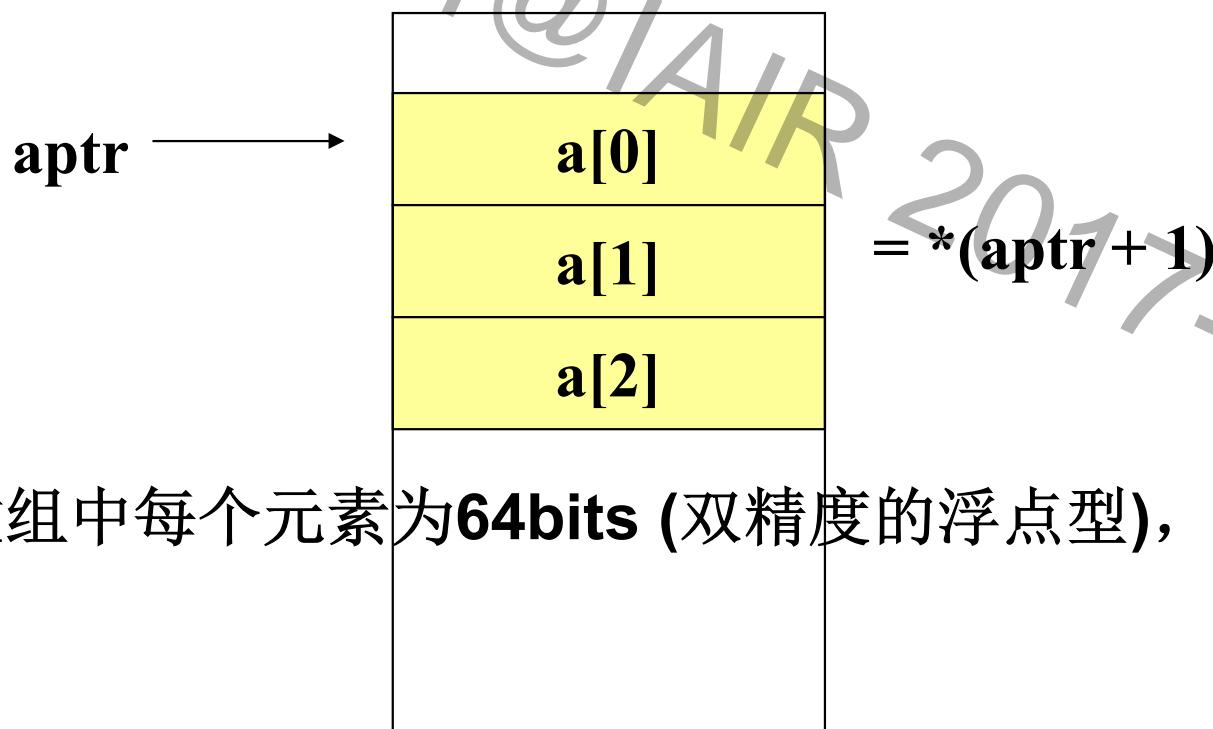
数据结构

- 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]** (每个元素为 **32bit**)
 - 第**0**个元素存储数组的第一个元素

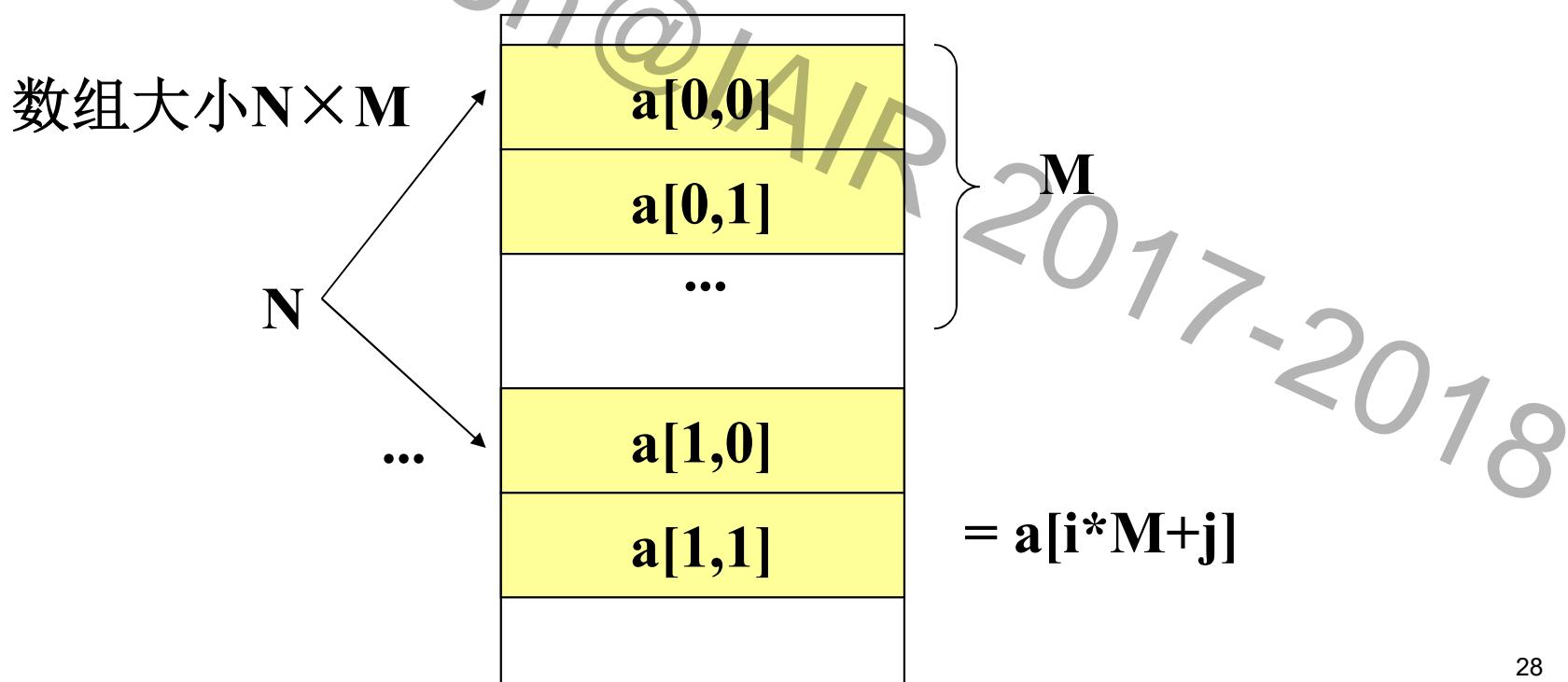


If 数组中每个元素为 **64bits** (双精度的浮点型), 如何索引?



二维数组

- 二维数组 $a[i,j]$: i 代表行, j 代表列
- 行优先 vs. 列优先



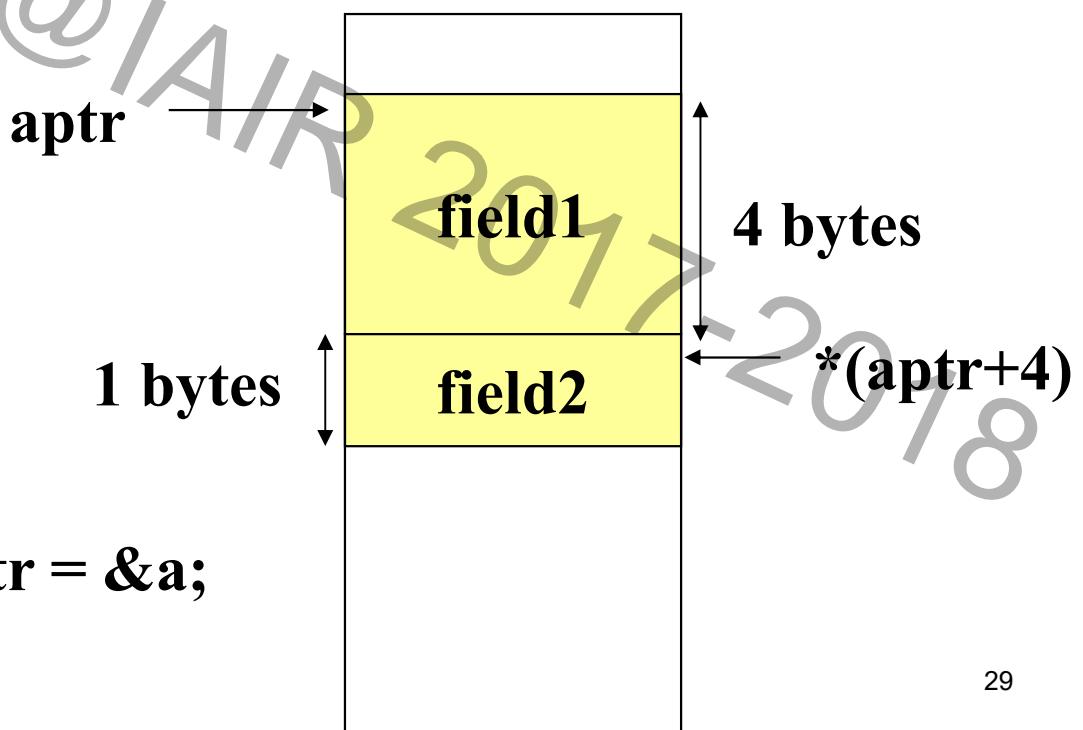


结构体

- 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**结构体地址+偏移量(静态)**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

PengjinRen@IAIR 2017-2018



表达式简化(Expression simplification)

- 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**)：
 - $8+1 = 9$
 - $\text{for } (i=0; i<8+1; i++)$ $\text{for } (i=0; i<\text{NOPS}+1; i++)$
- 强度化简：
 - $a^2 = a<<1$



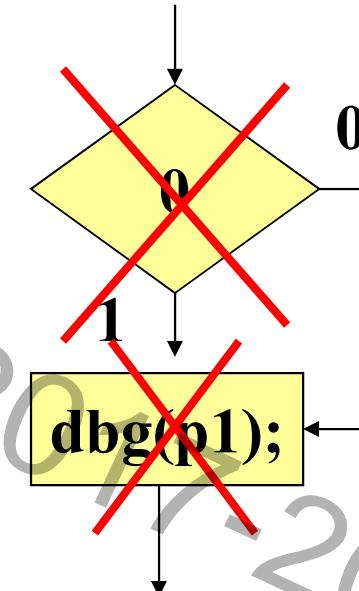
无效代码清除(Dead code elimination)

- 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(Loop transformations)

- 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(Loop unrolling)

- 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];
```



循环合并与分解(Loop fusion and distribution)

- 循环合并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];  
}
```

- 循环分解将一个循环分解成多个。



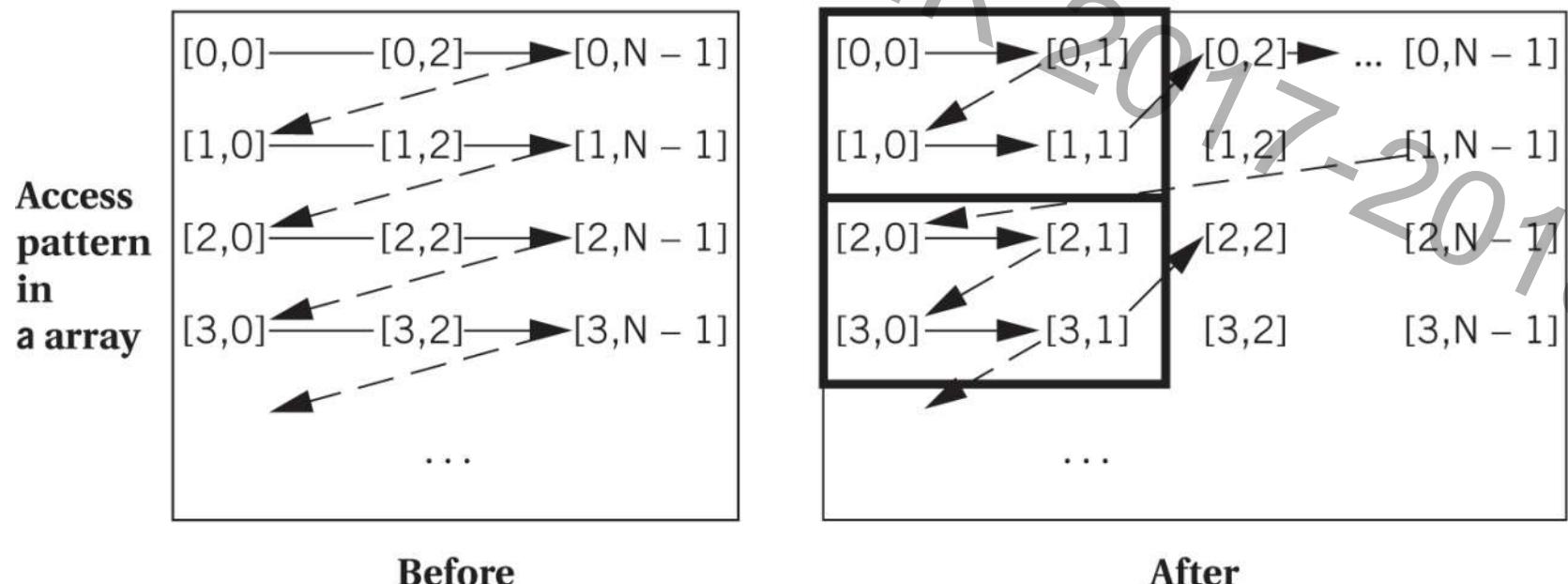
循环分块(Loop tiling)

- 将一个循环拆分成一系列嵌套循环，通过改变数组中的访问顺序，改善**cache**性能

```

Code           for (i = 0; i < N; i++)          for (i = 0; i < N; i += 2)
               for (j = 0; j < N; j++)          for (j = 0; j < N; j += 2)
                   c[i] = a[i,j] * b[j];      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];

```





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



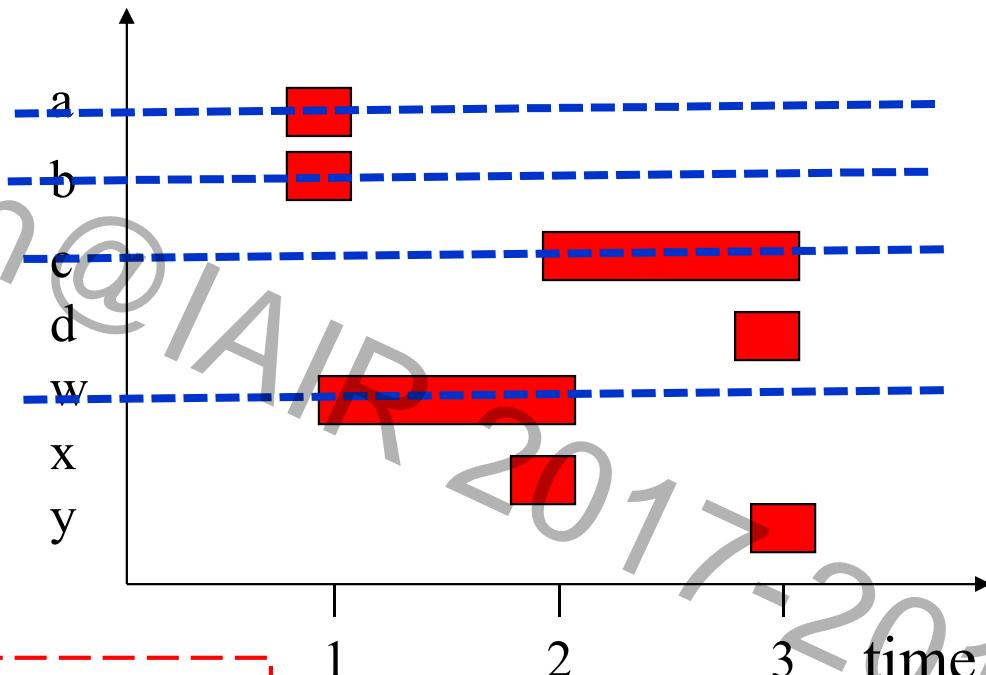
寄存器分配(Register allocation)

- 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

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



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



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

- 如果一段代码所需寄存器数目超过可用寄存器数目，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

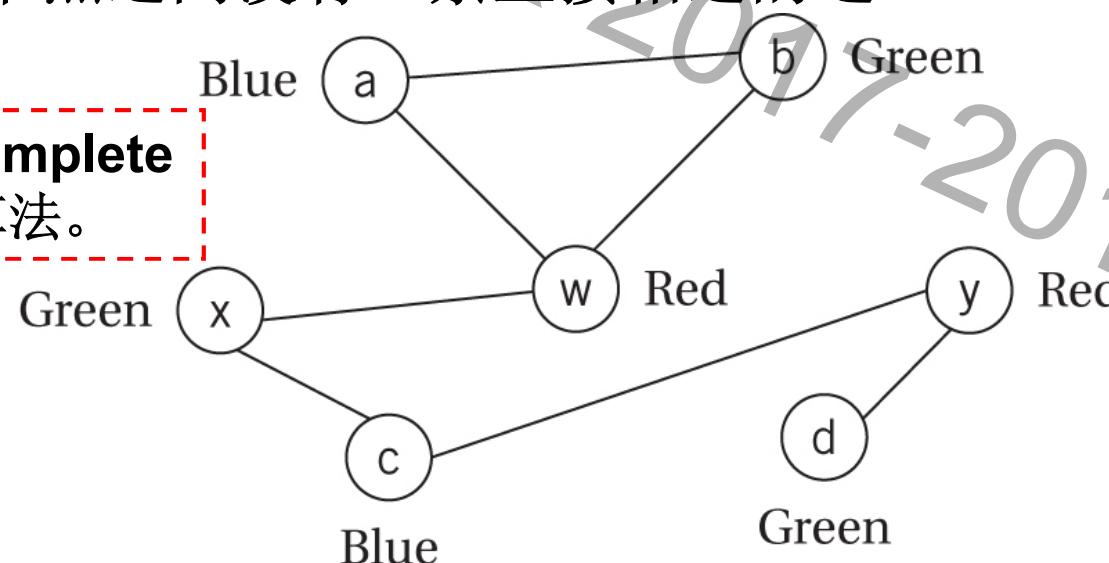
优化寄存器分配，避免不必要的
寄存器溢出是有价值的。



寄存器分配问题

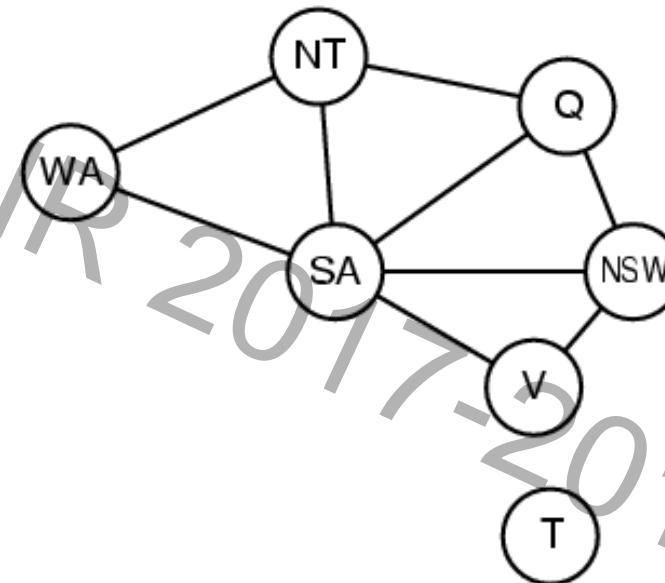
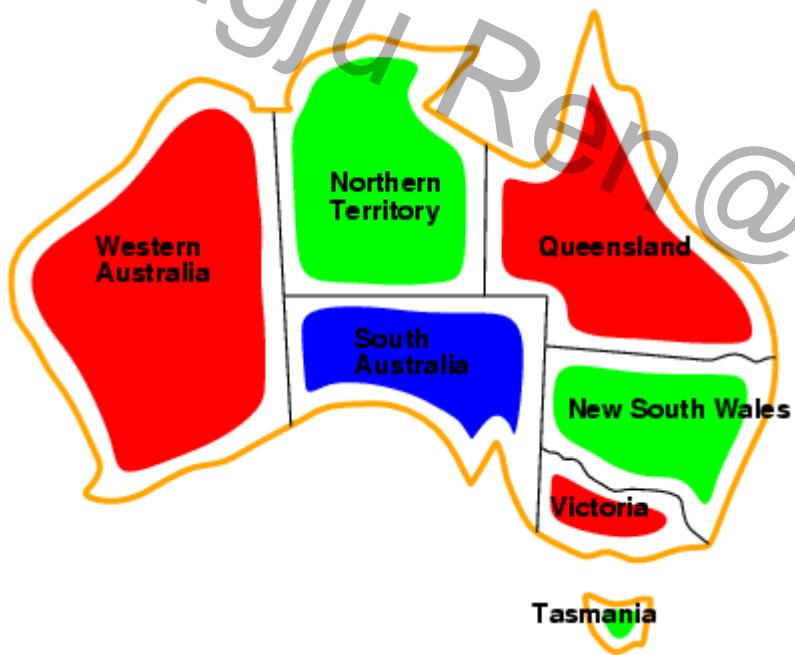
- 通过建立冲突图（**conflict graph**）并解一个图的着色问题来解决寄存器分配问题。
- 每个变量由一个节点表示，如果两个变量有相同的生存期，就在两个点之间添加一条边。
- 图的着色问题就是用最少的颜色给全部节点着色，要求两个相同颜色的节点之间没有一条直接相连的边。

图的着色问题是**NP-complete**问题，但是有启发式算法。





图着色问题



约束受限问题，请感兴趣的同学查看《人工智能-一种现代的方法》第6章内容。



改善寄存器分配的操作调度

$(a + b)^*(c - d)$

示例5.6

$$w = a + b$$

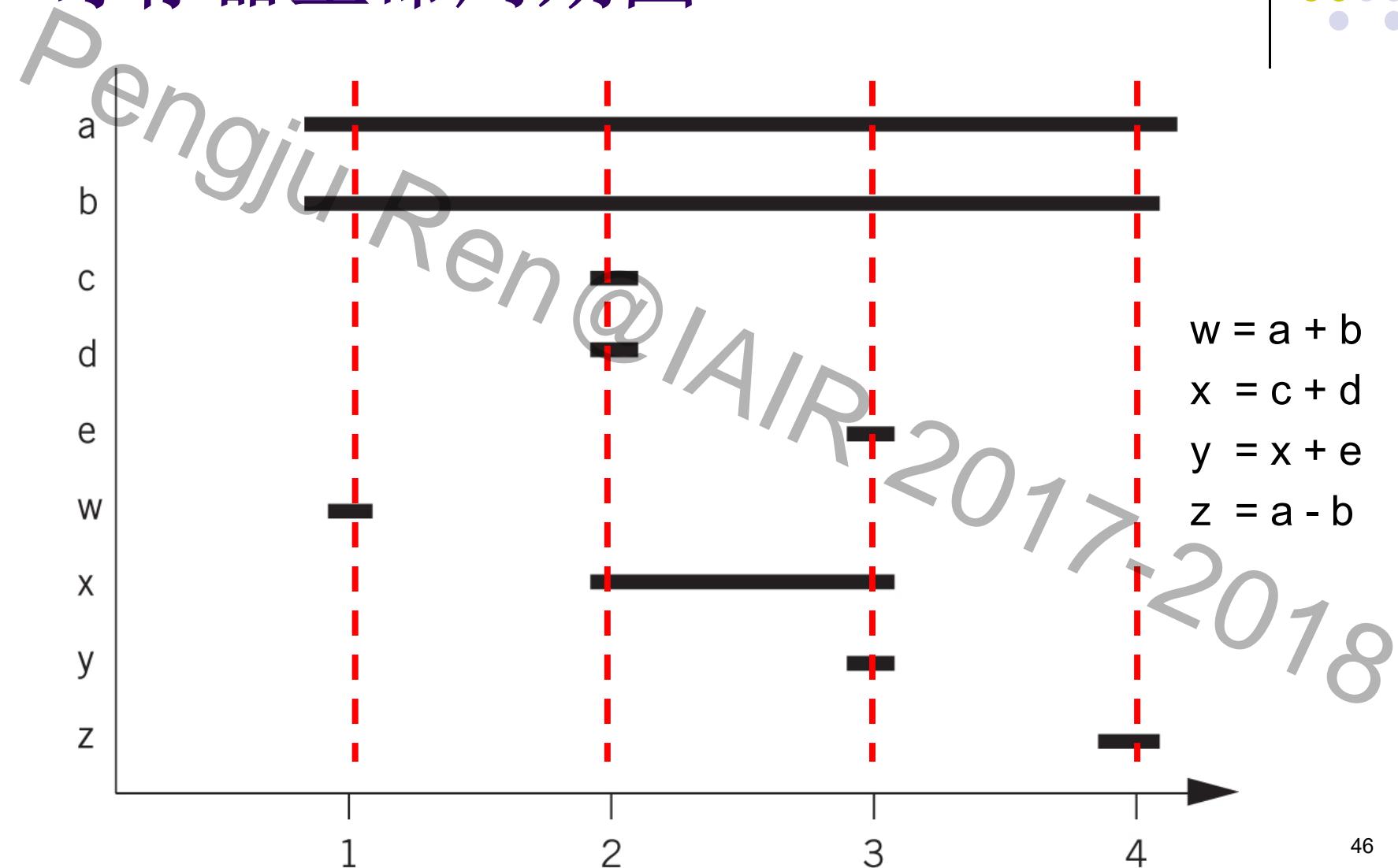
$$x = c + d$$

$$y = x + e$$

$$z = a - b$$



寄存器生命周期图





汇编代码（4个寄存器）

| | |
|----------------------|----------------------|
| LDR r0,a | LDR r1,e |
| LDR r1,b | ADD r0,r1,r2 |
| ADD r2,r0,r1 | STR r0,y ; y = x + e |
| STR r2,w ; w = a + b | LDR r0,a ; reload a |
| LDR r3,c | LDR r1,b ; reload b |
| LDR r1,d | SUB r2,r1,r0 |
| ADD r2,r0,r1 | STR r2,z ; z = a - b |
| STR r2,x ; x = c + d | |



操作调度后的C代码

w = a + b

z = a - b

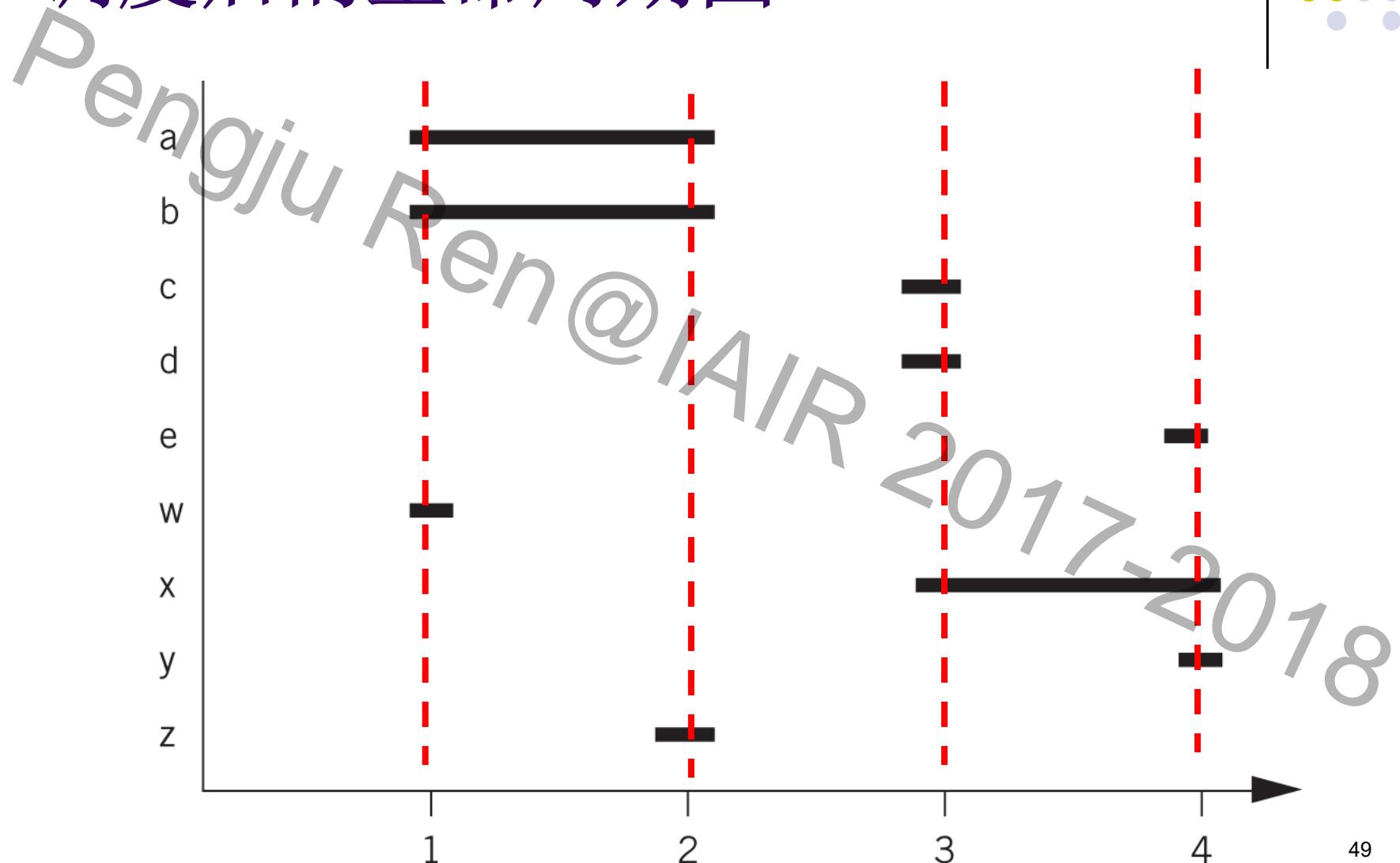
x = c + d

y = x + e

Pengju Ren@IAIR 2017-2018



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,r0,r1  
STR r2,w      ; w = a + b  
LDR r3,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
```



```
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**.



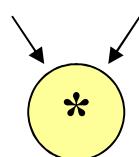
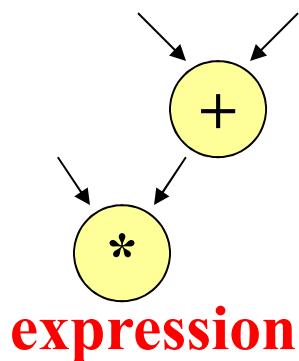
软件流水 software pipeline

- 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.**

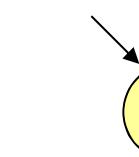


指令选择(Instruction selection)

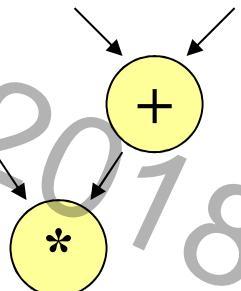
- 执行一个操作或一系列操作可能有几种不同的指令。执行时间/对邻近代码的影响。
- 模板匹配 (**template matching**)：将操作用图表表示，使用可能的指令序列进行匹配。



MUL



ADD



templates

MADD



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

Optimize options

Pengju Ren@AIR2018-2018

| Optimization | Included in Level | | | |
|----------------------------|-------------------|-----|-----|-----|
| | -O1 | -O2 | -Os | -O3 |
| defer-pop | ● | ● | ● | ● |
| thread-jumps | ● | ● | ● | ● |
| branch-probabilities | ● | ● | ● | ● |
| cprop-registers | ● | ● | ● | ● |
| guess-branch-probability | ● | ● | ● | ● |
| omit-frame-pointer | ● | ● | ● | ● |
| align-loops | ○ | ● | ○ | ● |
| align-jumps | ○ | ● | ○ | ● |
| align-labels | ○ | ● | ○ | ● |
| align-functions | ○ | ● | ○ | ● |
| optimize-sibling-calls | ○ | ● | ● | ● |
| cse-follow-jumps | ○ | ● | ● | ● |
| cse-skip-blocks | ○ | ● | ● | ● |
| gcse | ○ | ● | ● | ● |
| expensive-optimizations | ○ | ● | ● | ● |
| strength-reduce | ○ | ● | ● | ● |
| rerun-cse-after-loop | ○ | ● | ● | ● |
| rerun-loop-opt | ○ | ● | ● | ● |
| caller-saves | ○ | ● | ● | ● |
| force-mem | ○ | ● | ● | ● |
| peephole2 | ○ | ● | ● | ● |
| regmove | ○ | ● | ● | ● |
| strict-aliasing | ○ | ● | ● | ● |
| delete-null-pointer-checks | ○ | ● | ● | ● |
| reorder-blocks | ○ | ● | ● | ● |
| schedule-insns | ○ | ● | ● | ● |
| schedule-insns2 | ○ | ● | ● | ● |
| inline-functions | ○ | ○ | ○ | ● |
| rename-registers | ○ | ○ | ○ | ● |

