

# 编译技术



胡春明

[hucm@buaa.edu.cn](mailto:hucm@buaa.edu.cn)

2019.9-2019.12

# 第十五章 目标代码生成

面向目标体系结构的代码生成和优化技术



编译过程是指将**高级语言程序**翻译为等价的**目标程序**的过程。

习惯上是将编译过程划分为5个基本阶段：



## 寄存器分配 (Register Allocation)

- **改进目标:**

- 尽可能映射更多的变量到寄存器
- 减少内存读写次数

- **要解决的问题:**

- 把哪个变量放到哪个寄存器?
- 如果寄存器用完了, 如何替换?

## 全局寄存器分配：图着色算法

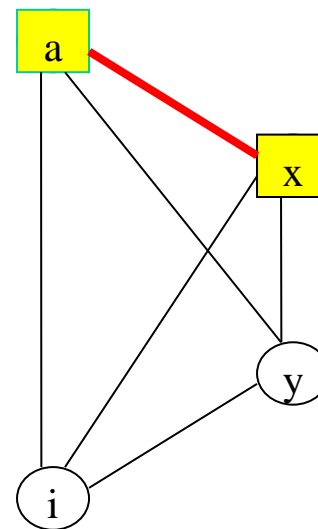
# 冲突图

	in[B]	out[B]
B1	a, b	a,x,y,i
B2	a,x,y,i	a,x,y,i
B3	a,x,y,i	a,x,y,i
B4	a,x,y,i	a,x,y,i
B5	a,x,y,i	a,x,y,i
B <sub>exit</sub>	∅	∅

**节点a:** 待分配全局寄存器的变量a

**节点x:** 待分配全局寄存器的变量x

**边a-x:** 变量 a 在变量 x 定义和使用处是活跃的 (即a、x同时活跃)

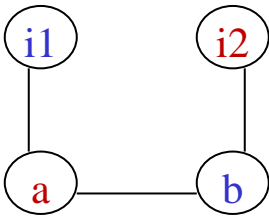
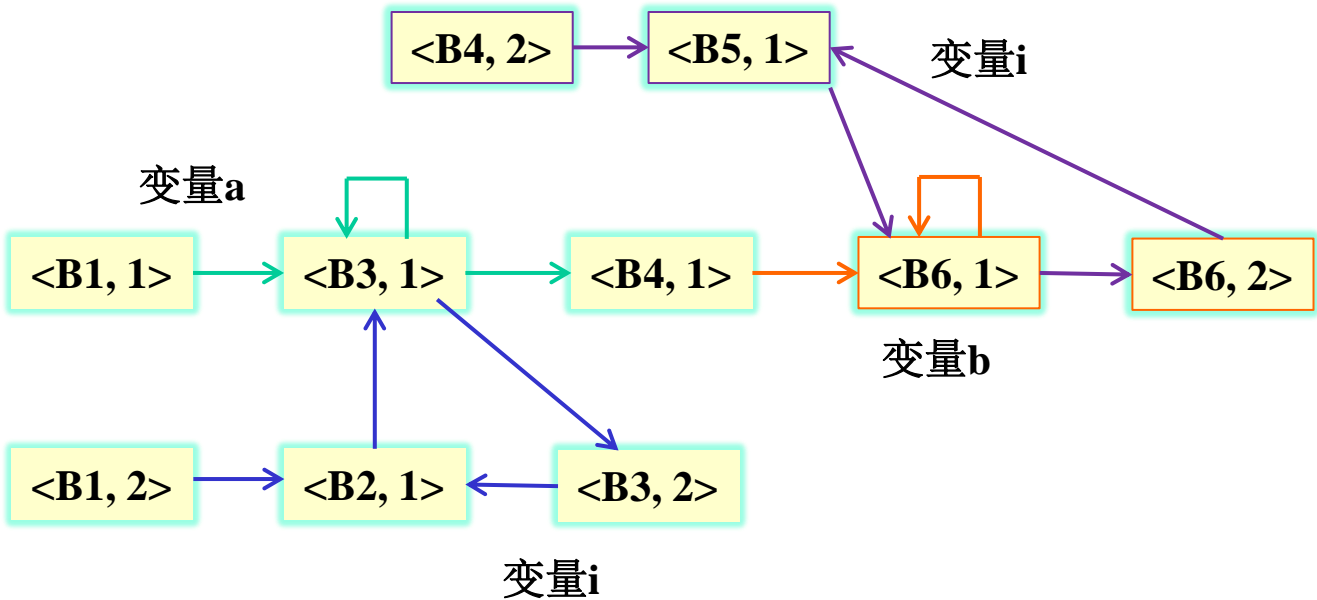


变量a: W1 { L1 {<B1, 1>, <B3, 1>, <B4, 1>}, L2 {<B3, 1>, <B3, 1>, <B4, 1>}}

变量b: W2 { L3 {<B4, 1>, <B6, 1>}, L4 {<B6, 1>, <B6, 1>}}

变量i: W3 { L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}, L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}}

W4 { L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}, L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}}



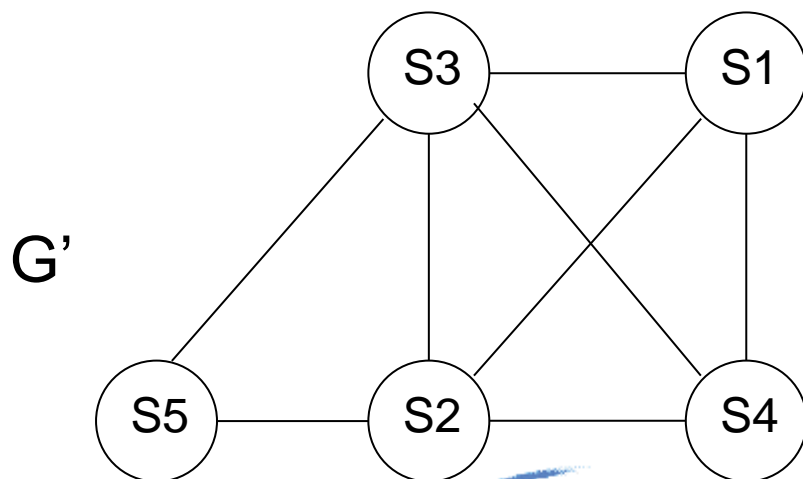
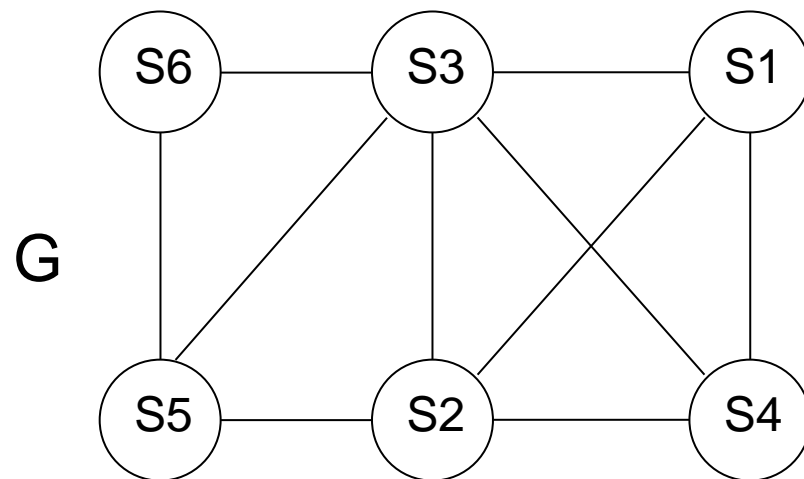
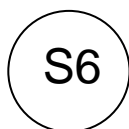


# 一种启发式图着色算法：Chaitin-Briggs算法

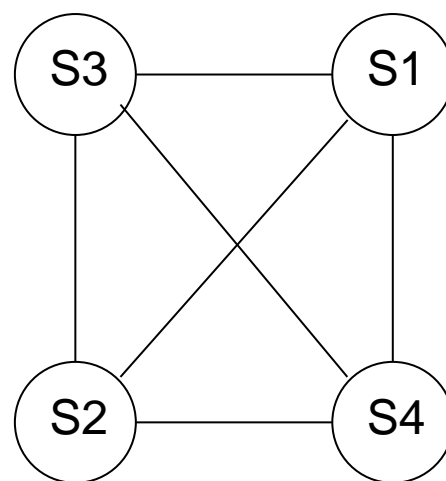
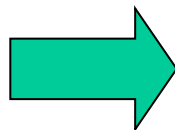
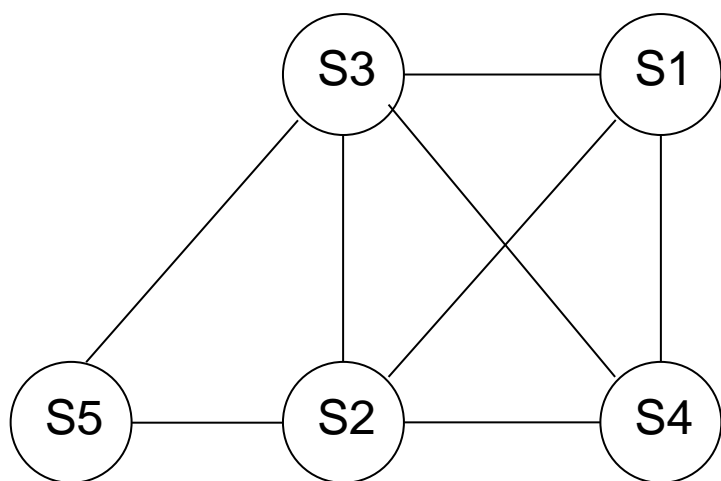
- 冲突图G
  - 寄存器数目为K
    - 假设  $K=3$

■ **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'


已移走节点



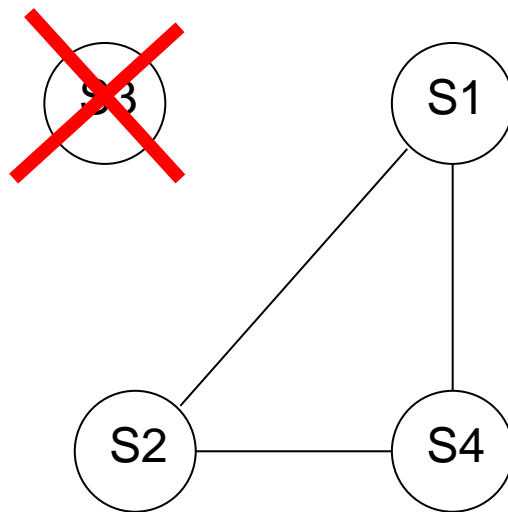
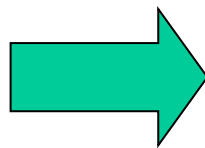
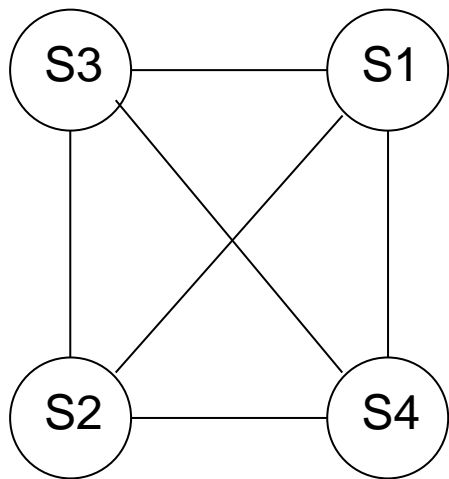
## ■ 步骤2、重复步骤1，直到无法再从 $G'$ 中移走结点






已移走节点 

已移走节点  

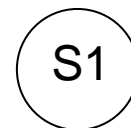
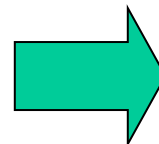
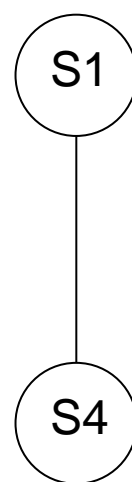
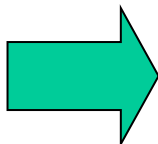
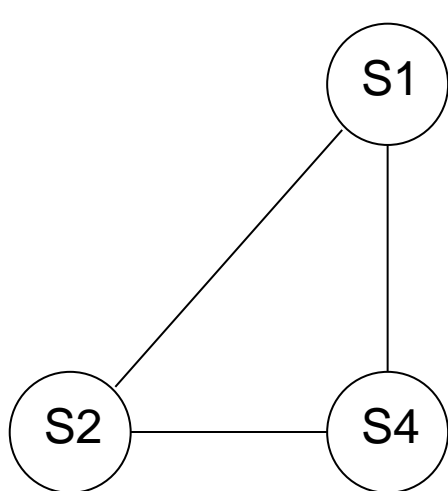
- **步骤3**、在图中选取**适当**的结点，将它记录为“不分配全局寄存器”的结点，并从图中移走



已移走节点  

已移走节点   

## ■ 步骤4、重复上述步骤，直到图中仅剩余1个结点

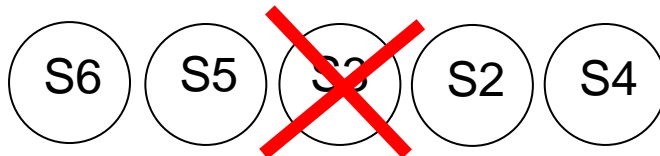


已移走 S6 S5 ~~S3~~

已移走 S6 S5 ~~S3~~ S2 S4

## ■ 步骤5 按照结点移走的反向顺序将点和边添加回去，并分配颜色

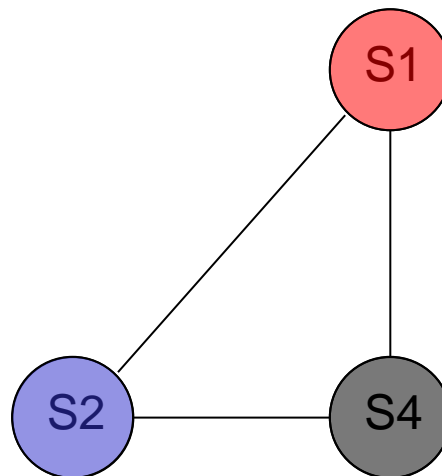
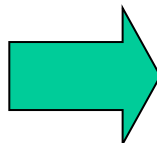
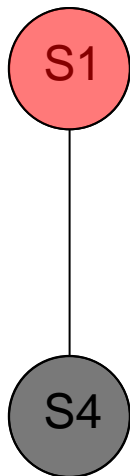
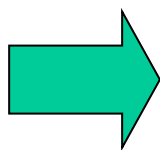
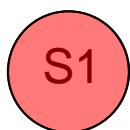
移走顺序



 R0

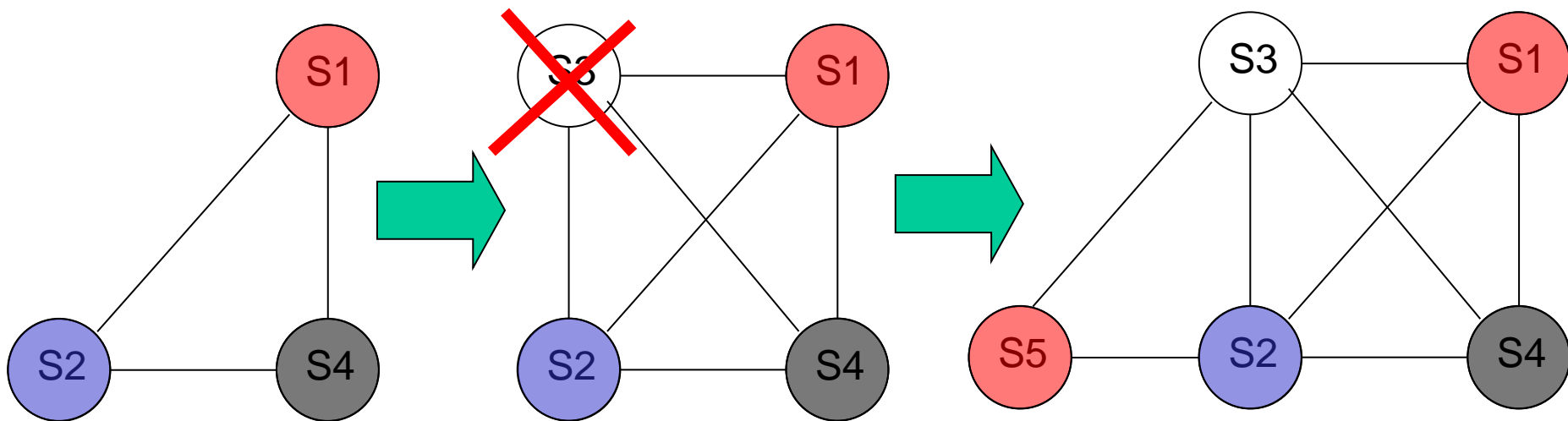
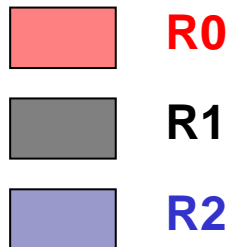
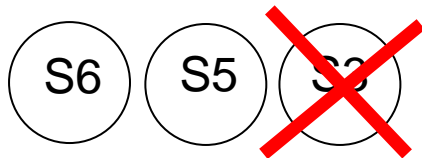
 R1

 R2



## ■ 步骤5 按照结点移走的反向顺序将点和边添加回去，并分配颜色

移走顺序



## ■ 步骤5 按照结点移走的反向顺序将点和边添加回去，并分配颜色

移走顺序

S6

R0

R1

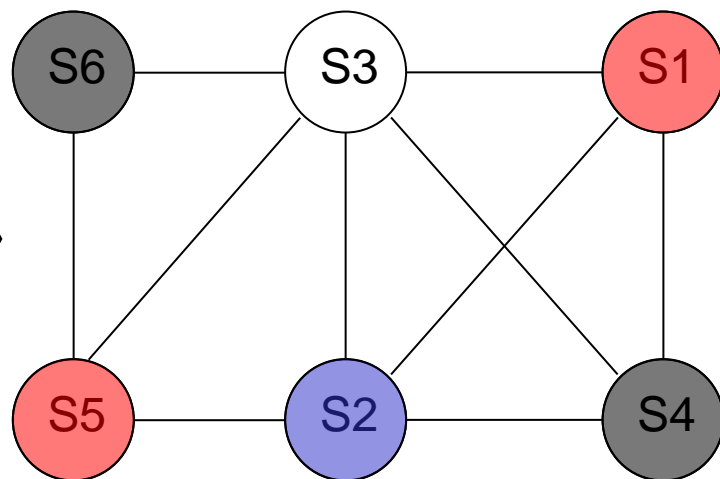
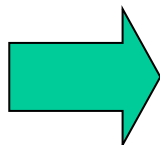
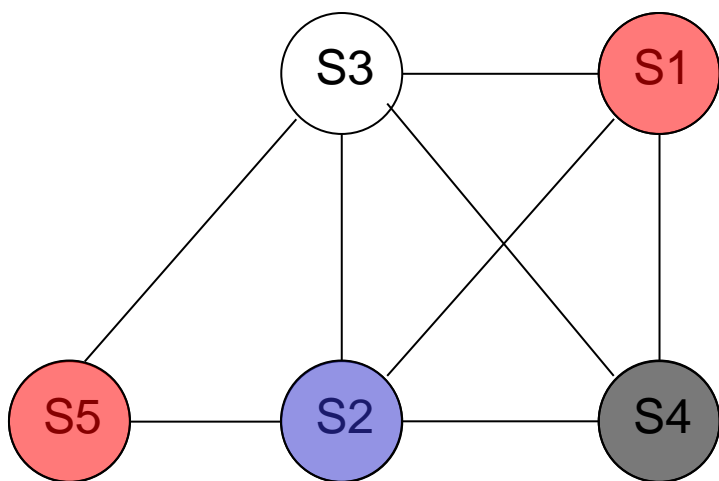
R2

R0: S1, S5

R1: S4, S6

R2: S2

变量 S3 不分配寄存器



## 临时寄存器分配：寄存器池



# 临时寄存器池：基本思想 FIFO

- **进入基本块：**清空临时寄存器池
- **全局变量、局部变量使用临时寄存器：**向临时寄存器池申请
- **申请处理：**
  - 有空闲寄存器：分配申请，做标识
  - 没有空闲寄存器：（启发式）**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
- **退出基本块（或函数调用发生前）：**将寄存器池中的值写回内存，清空临时寄存器池

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

例:

- (1)  $t1 = -c$
- (2)  $t2 = t1 - b$
- (3)  $t3 = t2 + t2$
- (4)  $a = t3$

a
b
c

EBX
ESI
EDI

t3
t2
t1

ESP+10H
ESP+0CH
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]`

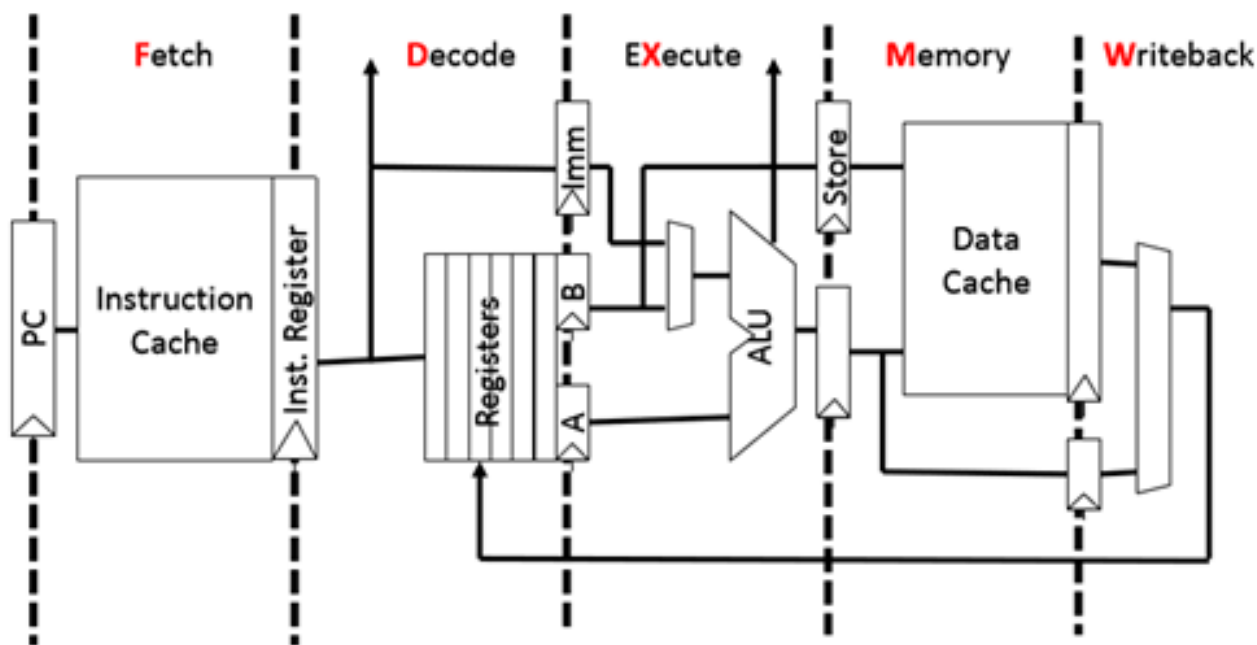
## 逐句转换+临时寄存器池:

- (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`

## 流水线

# 理解流水线

- Why? 
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$
- 典型的RISC五级流水线

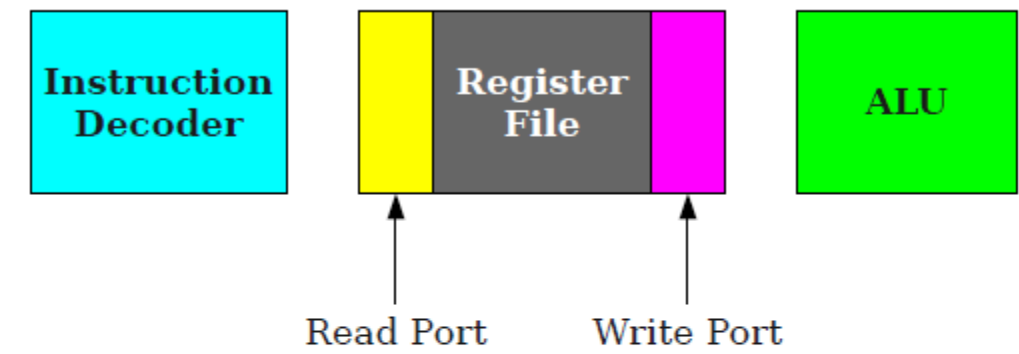


Source: <https://www.cnblogs.com/dragonir/p/6196602.html>

# Compiler

## 流水线

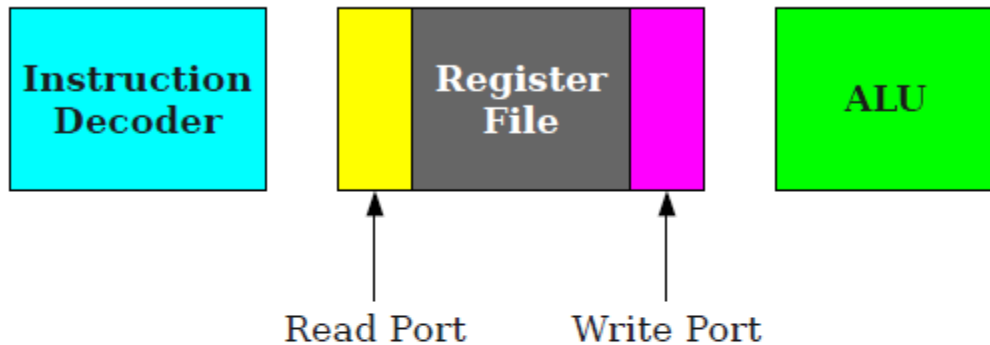
- 操作数准备好了吗？



```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t5, $t3, $t4    # $t5 = $t3 + $t4
add $t8, $t6, $t7    # $t8 = $t6 + $t7
```

ID	RR	ALU	RW

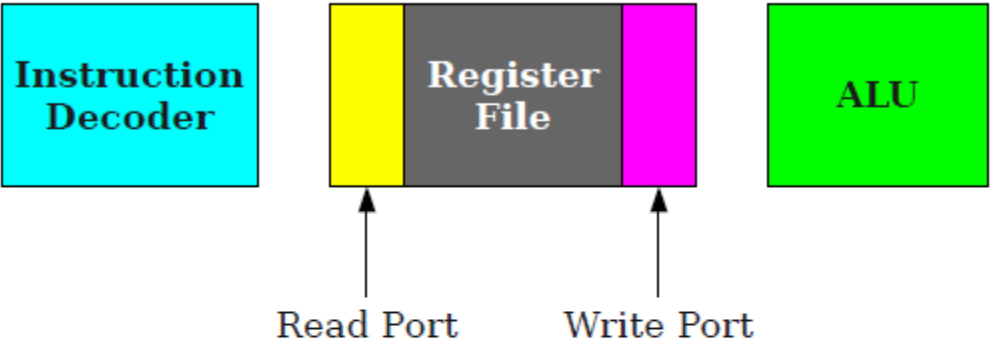
- 操作数准备好了吗?



```
add $t2, $t0, $t1 # $t2 = $t0 + $t1
add $t5, $t3, $t4 # $t5 = $t3 + $t4
add $t8, $t6, $t7 # $t8 = $t6 + $t7
```

[illegible]

• 操作数准备好了吗？

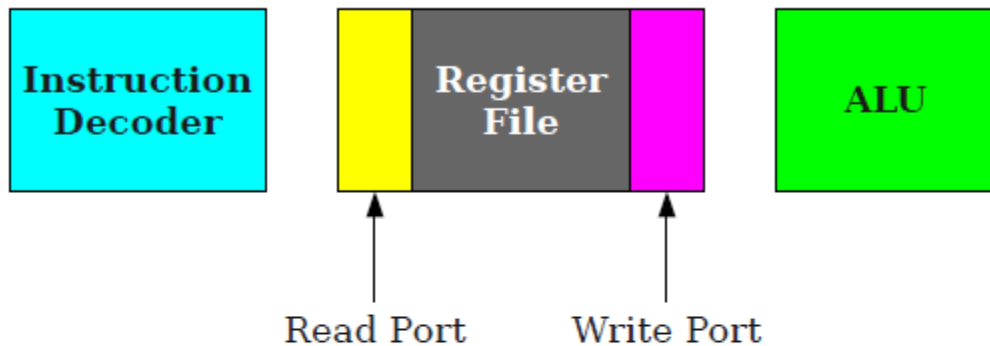


```
add $t2, $t0, $t1 # $t2 = $t0 + $t1
add $t4, $t3, $t2 # $t5 = $t3 + $t2
add $t7, $t5, $t6 # $t7 = $t5 + $t6
add $t0, $t0, $t7 # $t0 = $t0 + $t7
```

ID	RR	ALU	RW



- 操作数准备好了吗?



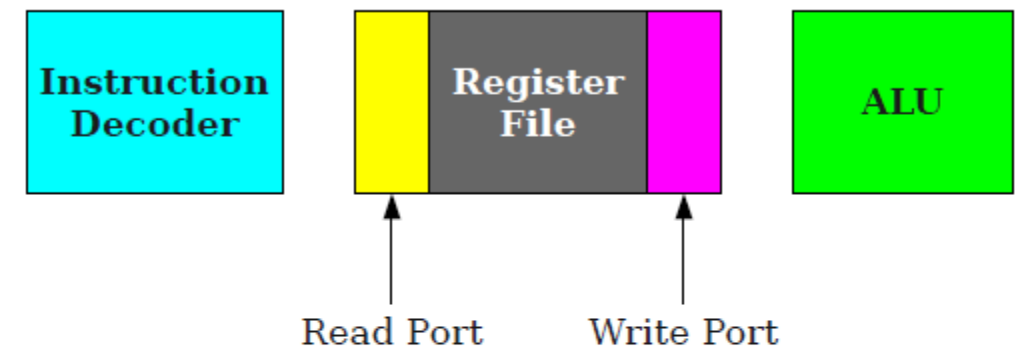
```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
add $t4, $t3, $t2    # $t5 = $t3 + $t2
add $t7, $t5, $t6    # $t7 = $t5 + $t6
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

# Compiler

## 流水线

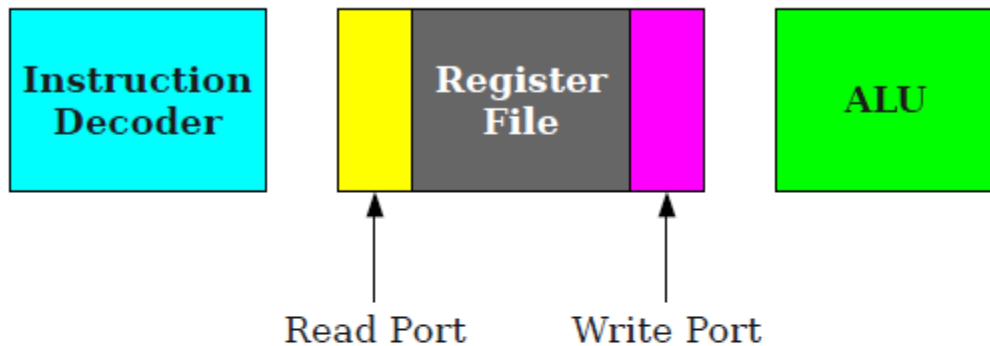
- 操作数准备好了吗？



```
add $t2, $t0, $t1 # $t2 = $t0 + $t1
add $t7, $t5, $t6 # $t7 = $t5 + $t6
add $t4, $t3, $t2 # $t5 = $t3 + $t2
add $t0, $t0, $t7 # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

- 操作数准备好了吗?



```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
```

```
add $t7, $t5, $t6    # $t7 = $t5 + $t6
```

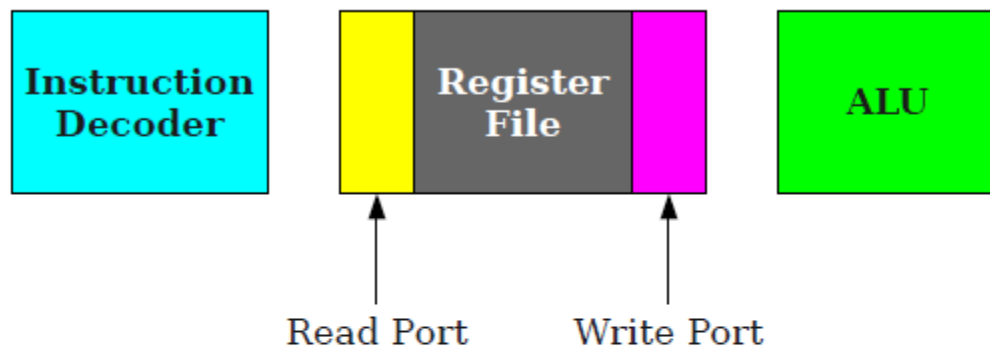
```
add $t4, $t3, $t2    # $t5 = $t3 + $t2
```

```
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

## 流水线优化：核心是“指令调度”

- **寻找让操作数准备好的指令序列**



```
add $t2, $t0, $t1    # $t2 = $t0 + $t1
```

```
add $t7, $t5, $t6    # $t7 = $t5 + $t6
```

```
add $t4, $t3, $t2    # $t5 = $t3 + $t2
```

```
add $t0, $t0, $t7    # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

# 流水线优化：核心是“指令调度”

- 优化：语义的正确性 + 避免流水线冲突
- 用“数据流的思想去处理”

$$t0 = t1 + t2$$

$$t1 = t0 + t1$$

$$t3 = t2 + t4$$

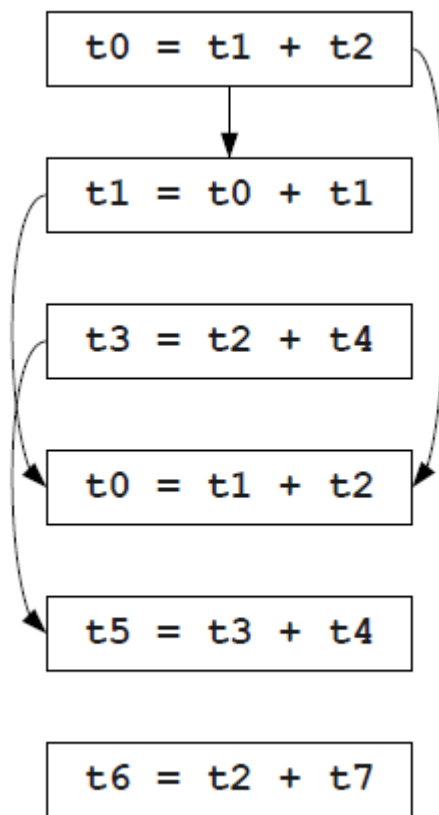
$$t0 = t1 + t2$$

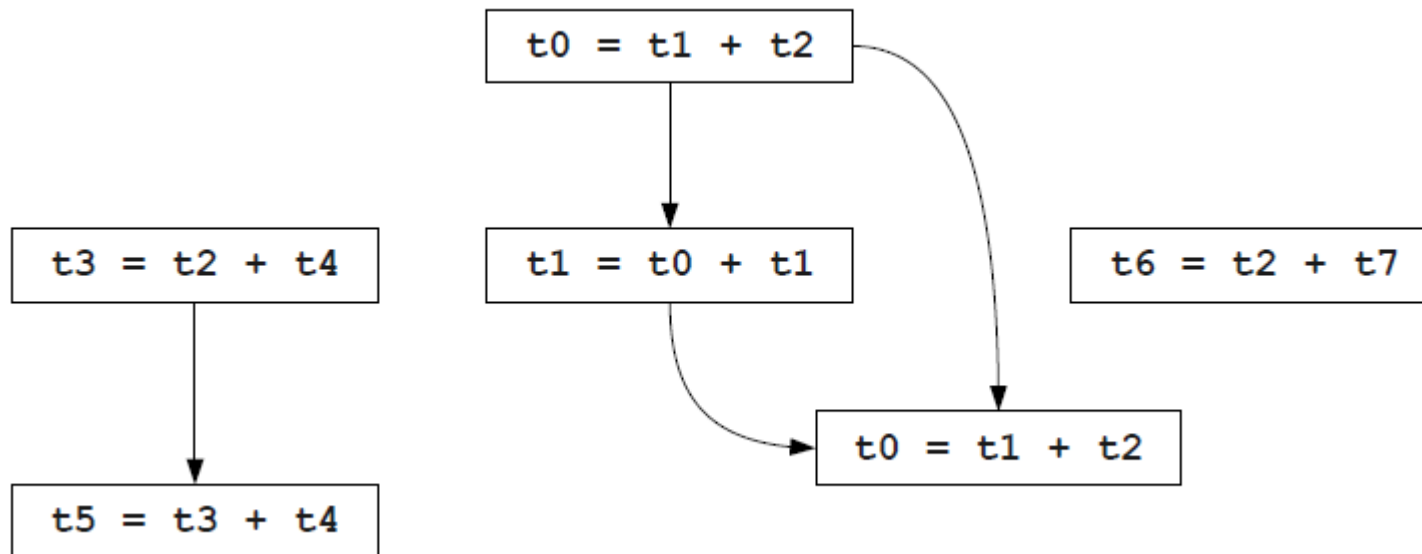
$$t5 = t3 + t4$$

$$t6 = t2 + t7$$

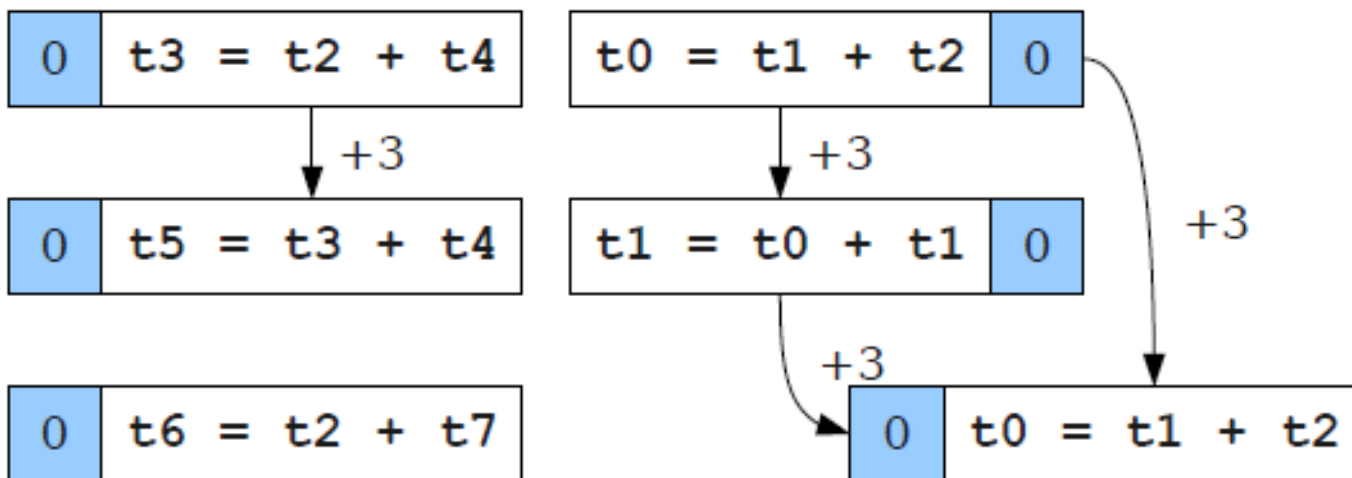
# 流水线优化：核心是“指令调度”

- 优化：语义的正确性 + 避免流水线冲突
- 用“数据流的思想去处理”





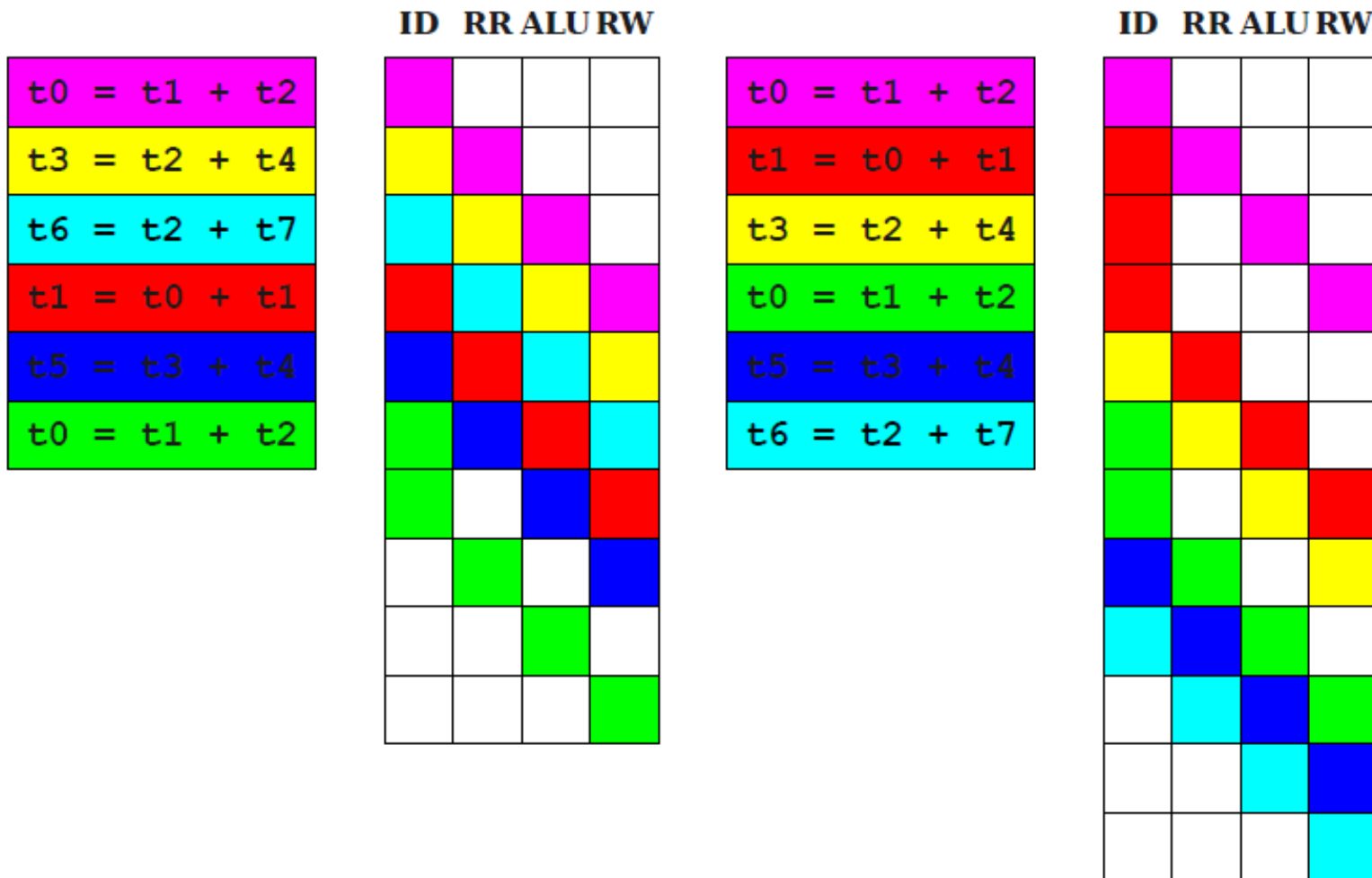
$t3 = t2 + t4$
$t5 = t3 + t4$
$t0 = t1 + t2$
$t1 = t0 + t1$
$t0 = t1 + t2$
$t6 = t2 + t7$



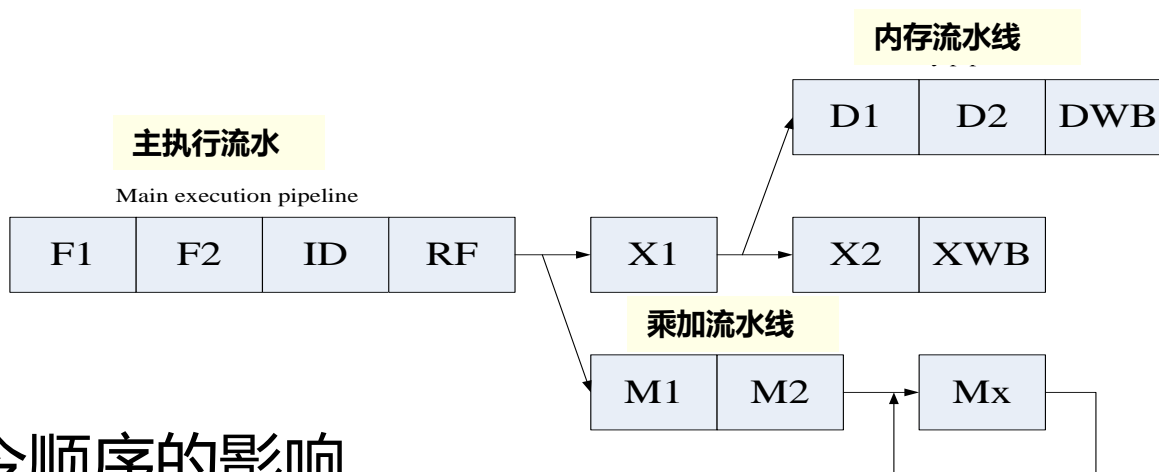
$t3 = t2 + t4$
$t5 = t3 + t4$
$t0 = t1 + t2$
$t1 = t0 + t1$
$t0 = t1 + t2$
$t6 = t2 + t7$



# 流水线优化：核心是“指令调度”



# XScale core RISC 超流水线



## 不同指令顺序的影响

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

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

时钟周期:  $1 + 1 + 3 + 1 = 6$

时钟周期:  $1 + 1 + 1 + 1 = 4$

## 缓存利用

# 数据访问的时空局部性

- 通常的缓存替换策略： LRU

```
arr[0] = 5;
arr[2] = 6;
arr[10] = 13;
arr[1] = 4;
```

arr[0]	0
arr[1]	0
arr[2]	0
arr[3]	0
arr[4]	0
arr[5]	0
arr[6]	0
arr[7]	0
arr[8]	0
arr[9]	0
arr[10]	0
arr[11]	0

Memory Cache	
arr[0]	0
arr[1]	0
arr[2]	0
arr[3]	0

# 数据访问的时空局部性

- 例如：数组的存储与访问

```
int[][] array;
for (j = 0; j < 4; j = j + 1)
    for (i = 0; i < 4; i = i + 1)
        array[i][j] = 0;
```

00	01	02	03	10	11	12	13	20	21	22	23	30	31	32	33
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

```
int[][] array;
for (j = 0; j < 4; j = j + 1)
    for (i = 0; i < 4; i = i + 1)
        array[i][j] = 0;
```

00	01	02	03	10	11	12	13	20	21	22	23	30	31	32	33
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

## 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

$x[0][0], x[0][1], \dots, x[0][99], x[1][0], x[1][1], \dots, x[1][99], \dots, x[4999][0], x[4999][1], \dots, x[4999][99]$

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

$x[0][0], x[0][1], \dots, x[0][99], x[1][0], x[1][1], \dots, x[1][99], \dots, x[4999][0], x[4999][1], \dots, x[4999][99]$

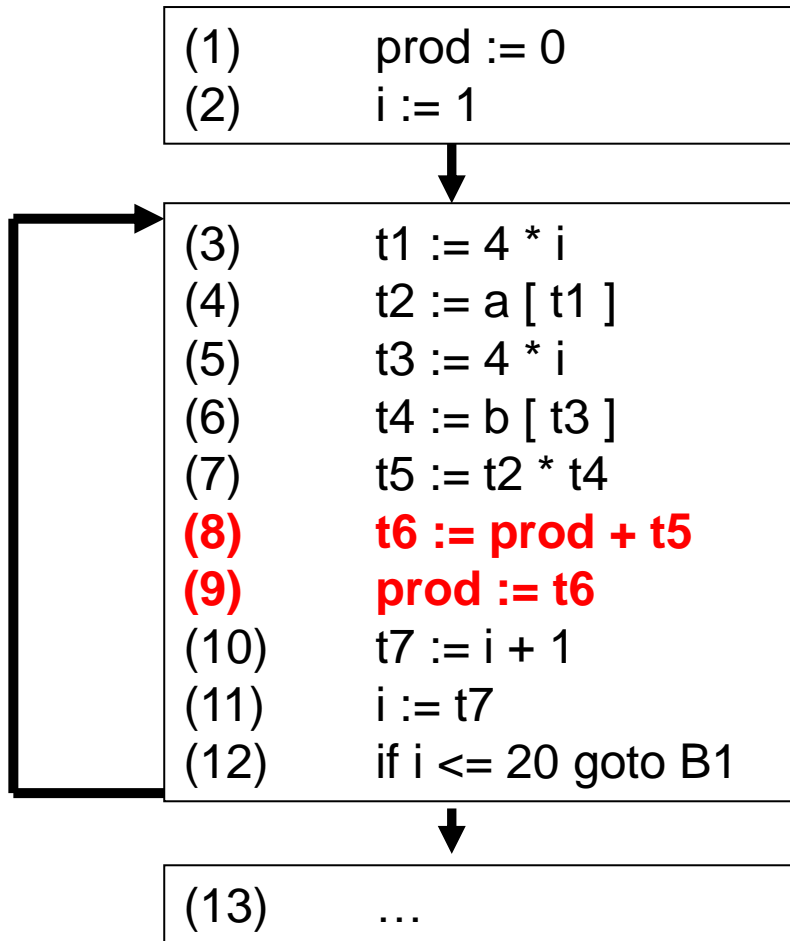
## 其他优化：指令选择

## 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, 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
```

## 并行优化

- 处理器的并行处理能力
- 向量处理
- GPU/NPUs

# 优化部分小结

## 第11章 中间代码优化

与机器无关的优化独立于机器的（中间）代码优化

优化分为  
两大类

## 第12章 目标代码生成

与机器有关的优化目标代码上的优化（与具体机器有关）

## 中间代码优化方法:

- **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

## 中间代码优化方法:

- **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

# 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
- 理解流水线（指令调度）
- 理解Cache优化（利用时空局部性）

谢谢!