

COMP130014 编译

# 第十三讲：后端优化

徐 辉

xuh@fudan.edu.cn



# 大纲

一、后端优化问题

二、指令调度优化

三、窥孔优化

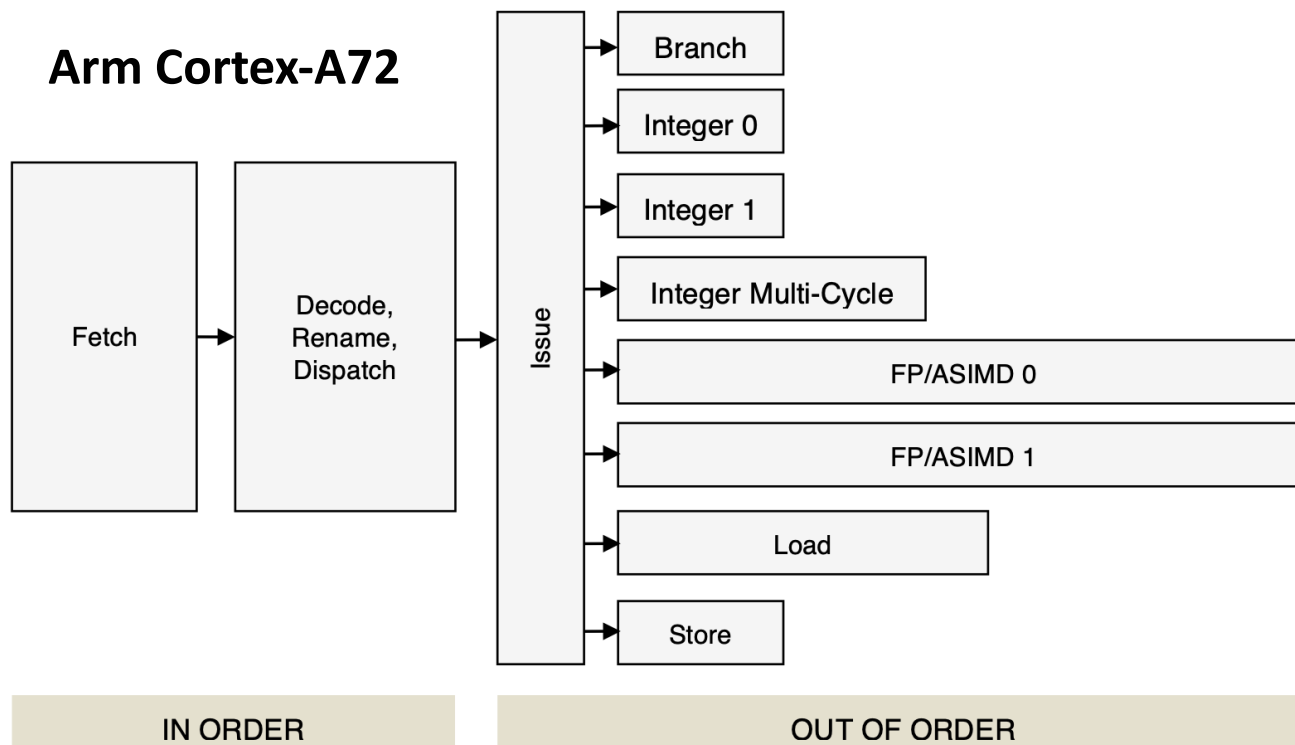
四、利用CPU特性优化

# 一、后端优化问题

---

# CPU流水线和乱序执行

- 流水线 => 指令级并行
  - 每个指令由1个或多个微指令（ $\mu$ OP）组成
  - 一个周期可以同时执行多条微指令，数据依赖满足便可执行



# 指令执行顺序影响性能

- 指令之间存在数据依赖关系
- 不同指令执行效率不同
- CPU优化能力有限

```
add x1, x2, x3
add x4, x5, x6
mul x0, x2, x3
sub x2, x0, x1
add x4, x4, x5
```

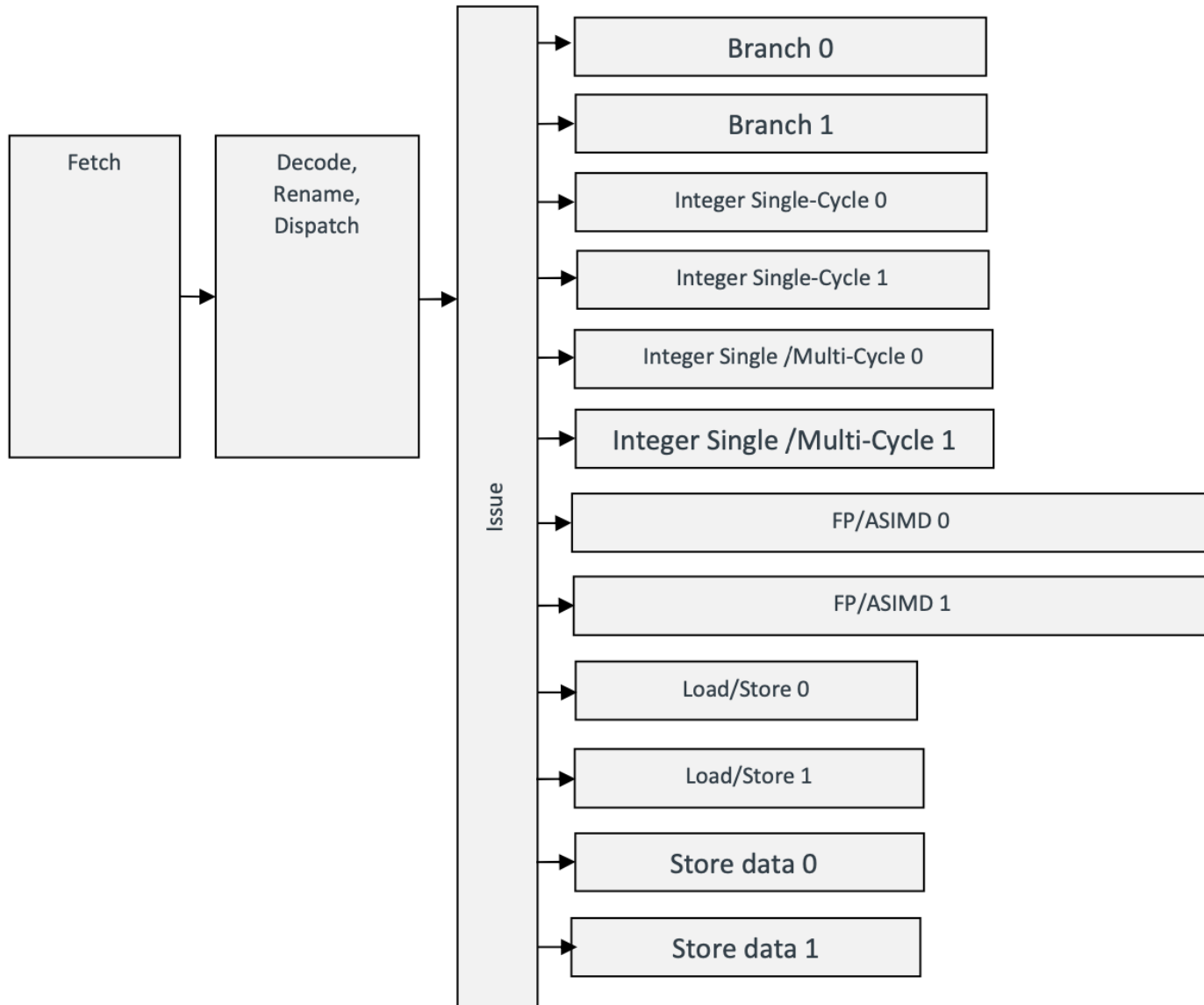
Stage	Clock Cycles							
	1	2	3	4	5	6	7	8
Fetch	add	add	mul	sub	add			
Decode		add	add	mul	sub	add		
Execute(I0)			add	add			add	sub
Execute(I1)								
Execute(M)					mul			

假设执行mul需要3个cycles，执行add/sub需要1个cycle

# Arm Cortex-A72指令开销

指令组	指令	延迟	吞吐	Pipeline
数据存取	ldr	4	1	L
	str	1	1	S
算数运算	add	1	2	I0/I1
	sub	1	2	I0/I1
	mul	3	1	M
	madd/msub	3	1	M
	sdiv	4-20	1/20-1/4	M
移动	mov	1	2	I0/I1
取地址	adr/adrp	1	2	I0/I1
跳转	b/bl/ret	1	1	B
	cbz/tbz	1	1	B

# Arm Cortex-A77



# 影响性能的因素

- 数据依赖关系
  - 写-读依赖（RAW/Read-After-Write）： true-dependency
  - 读-写反依赖（WAR/Write-After-Read）： anti-dependency
- 结构性影响（structural hazard）
  - 一条指令由多条微指令组成
  - 相邻指令的微指令可能会竞争ports的使用
- 控制流影响（control hazard）
  - 条件跳转或分支预测



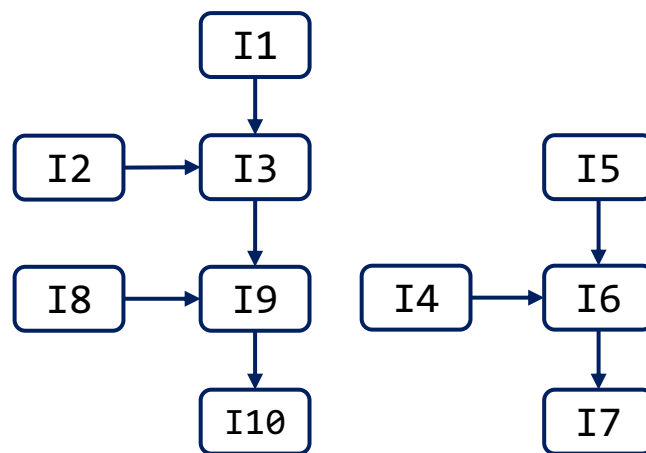
## 二、指令调度优化

---

# 指令依赖关系：写-读依赖（RAW）

- 场景：单个程序块，无跳转指令
- 如果指令I2使用I1的结果，那么I2依赖I1
- 叶子节点没有任何依赖，可以尽早执行
  - I1、I2、I4、I7

I1	ldr x9, [sp, -12]
I2	ldr x10, [sp, -16]
I3	add x9, x9, x10
I4	ldr x10, [sp, -20]
I5	ldr x11, [sp, -24]
I6	sdiv x11, x10, x11
I7	str x11, [sp, -24]
I8	ldr x10, [sp, -28]
I9	mul x10, x9, x10
I10	str x10, [sp, -28]

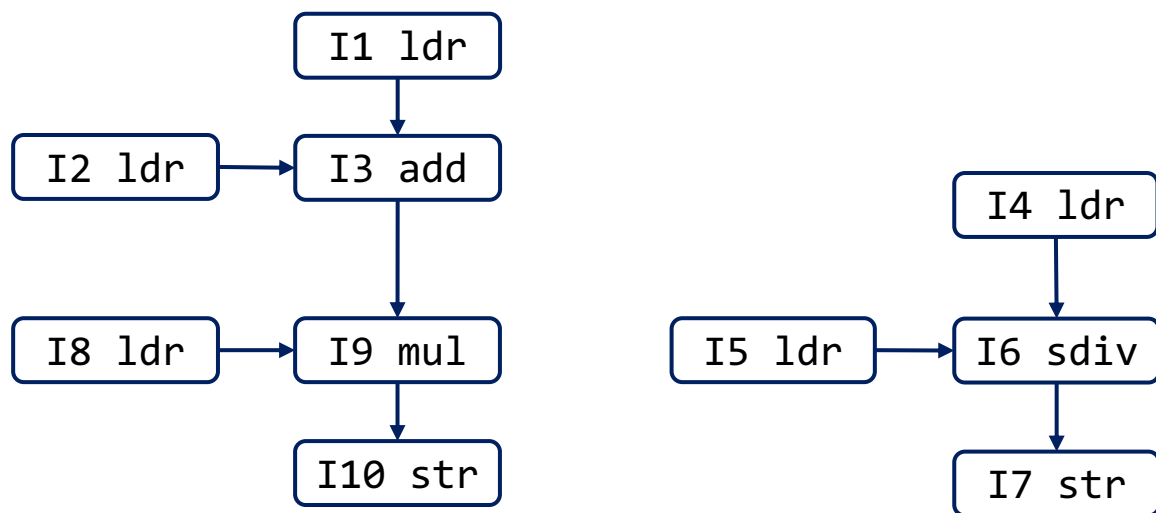


指令依赖关系

# 编译器的指令调度问题

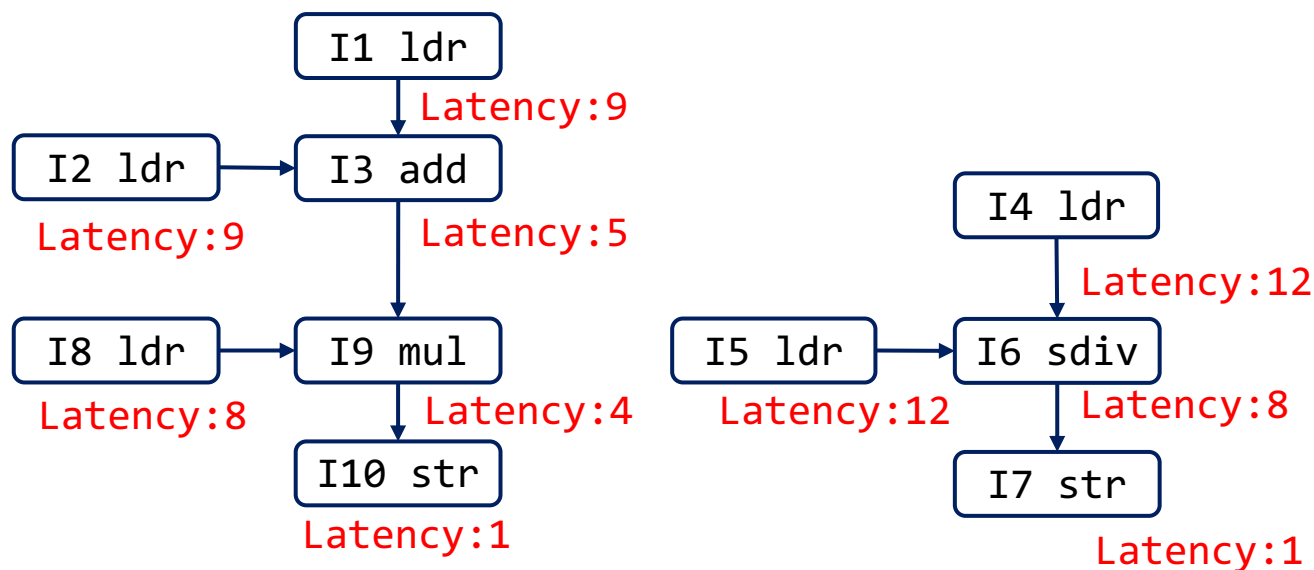
- 假设：
  - 每个cycle可以执行一条指令
  - 多条指令可以并行
  - 单条指令开销稳定
- 应如何确定最佳的指令执行序列？
  - 执行顺序应满足数据依赖关系

指令	延迟	吞吐
ldr	4	不限
str	1	不限
add	1	不限
sub	1	不限
mul	3	不限
sdiv	7	不限
mov	1	不限



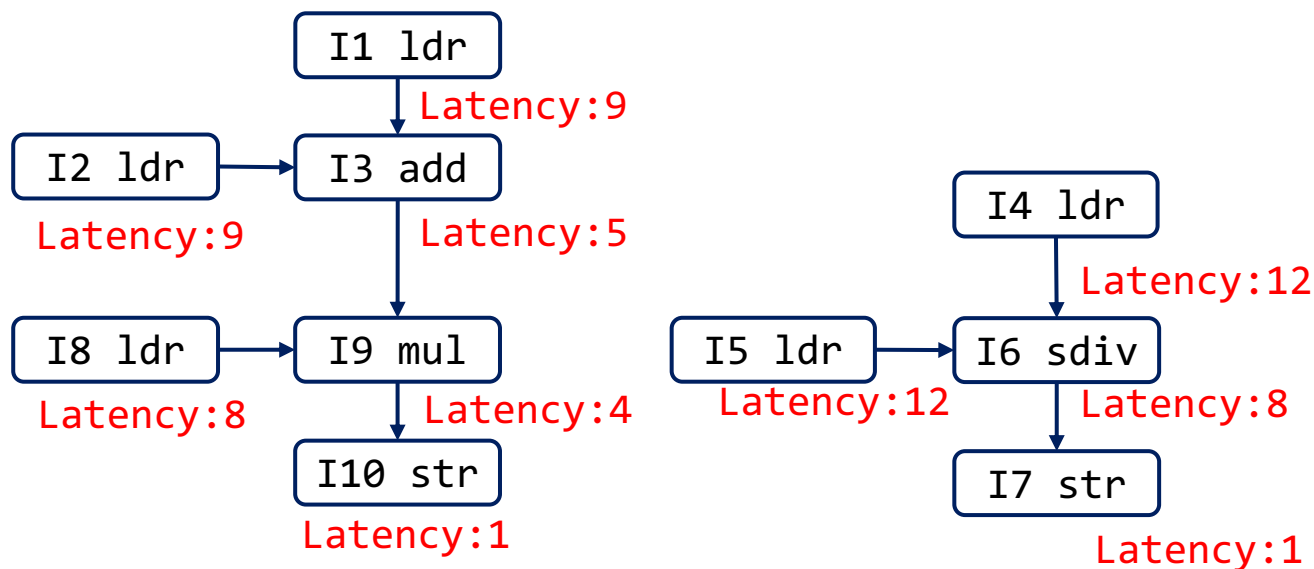
# 指令调度思路

- 计算每条指令开始执行后，序列执行结束所需时间（latency）
  - 假设  $i = v.next$ ,  $L(v) = E_v + L(i)$



# 指令调度思路

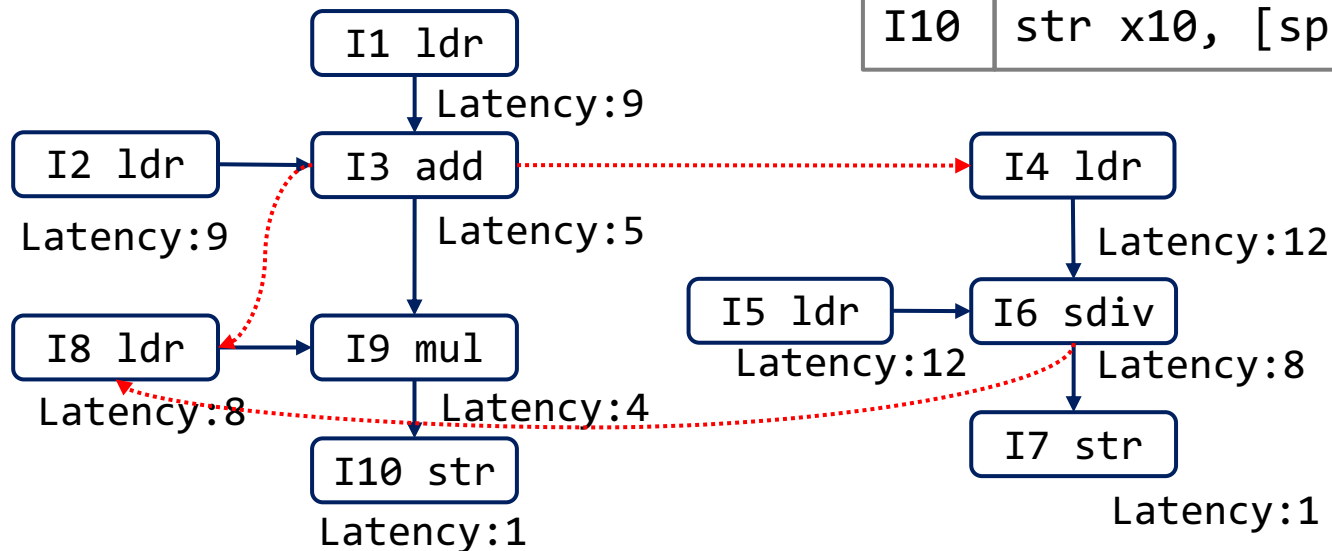
- 根据latency从大到小对指令进行排序
  - $I4=I5>I6>I1=I2>I8>I3>I9>I7=I10$
- 优先执行latency大的指令



# 读-写反依赖（WAR）问题

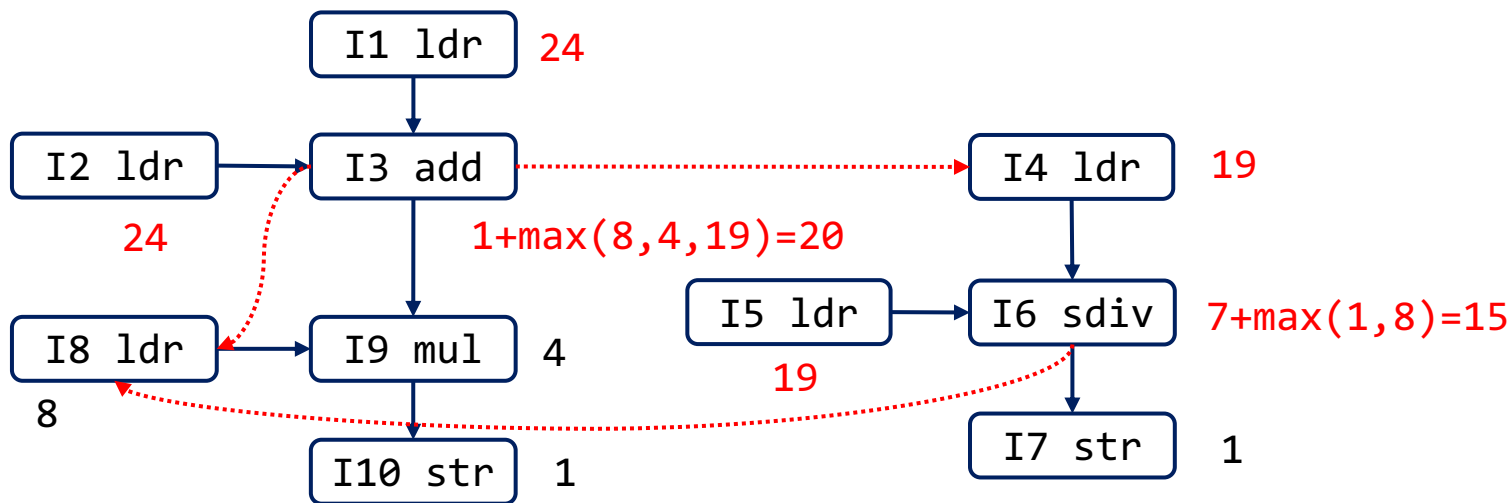
- I3执行完I4和I8才能执行
  - 否则会影响I3的计算结果
- I6执行完才能执行I8
- 寄存器分配（复用）导致

I1	ldr x9, [sp, -12]
I2	ldr x10, [sp, -16]
I3	add x9, x9, x10
I4	ldr x10, [sp, -20]
I5	ldr x11, [sp, -24]
I6	sdiv x11, x10, x11
I7	str x11, [sp, -24]
I8	ldr x10, [sp, -28]
I9	mul x10, x9, x10
I10	str x10, [sp, -28]



# 更新Latency并排序

- $\forall i \in v.next, L(v) = E_v + \text{Max}(L(i))$
- 重新排序: I1=I2>I3>I4=I5>I6>I8>I9>I7=I10



# 调度方案开销

- I1=I2>I3>I4=I5>I6>I8>I9>I7=I10

- 开销：26

开始 结束 指令

1	4	I1	ldr x9, [sp, -12]
2	5	I2	ldr x10, [sp, -16]
6	6	I3	add x9, x9, x10
7	10	I4	ldr x10, [sp, -20]
8	11	I5	ldr x11, [sp, -24]
12	18	I6	sdiv x11, x10, x11
19	22	I8	ldr x10, [sp, -28]
23	25	I9	mul x10, x9, x10
24	24	I7	str x11, [sp, -24]
26	26	I10	str x10, [sp, -28]

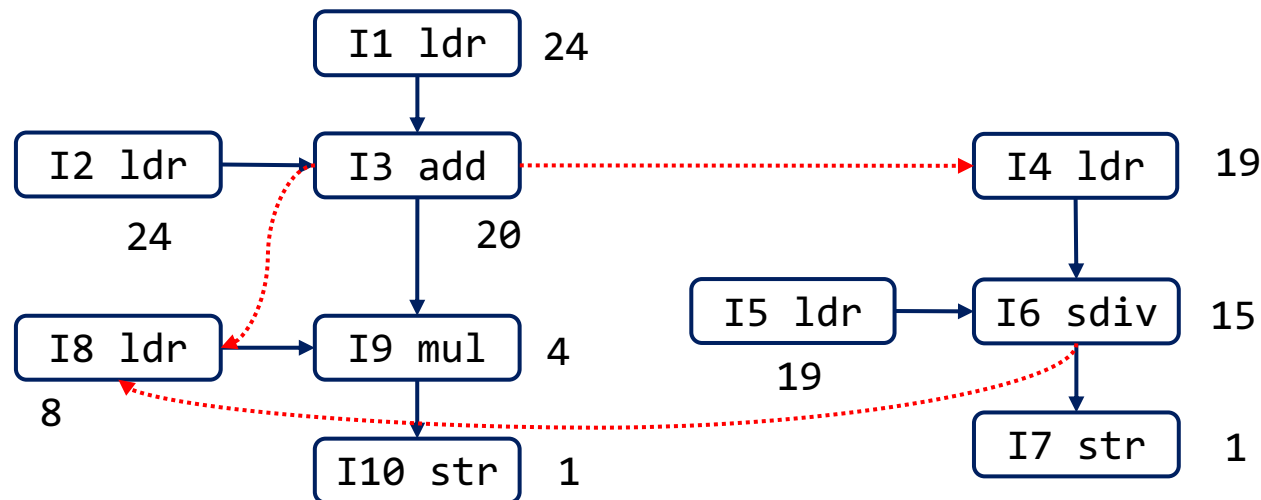


# 消除反依赖：重命名（vs Tomasulo）

I1	ldr x9, [sp, -12]
I2	ldr x10, [sp, -16]
I3	add x9, x9, x10
I4	ldr x10, [sp, -20]
I5	ldr x11, [sp, -24]
I6	sdiv x11, x10, x11
I7	str x11, [sp, -24]
I8	ldr x10, [sp, -28]
I9	mul x10, x9, x10
I10	str x10, [sp, -28]

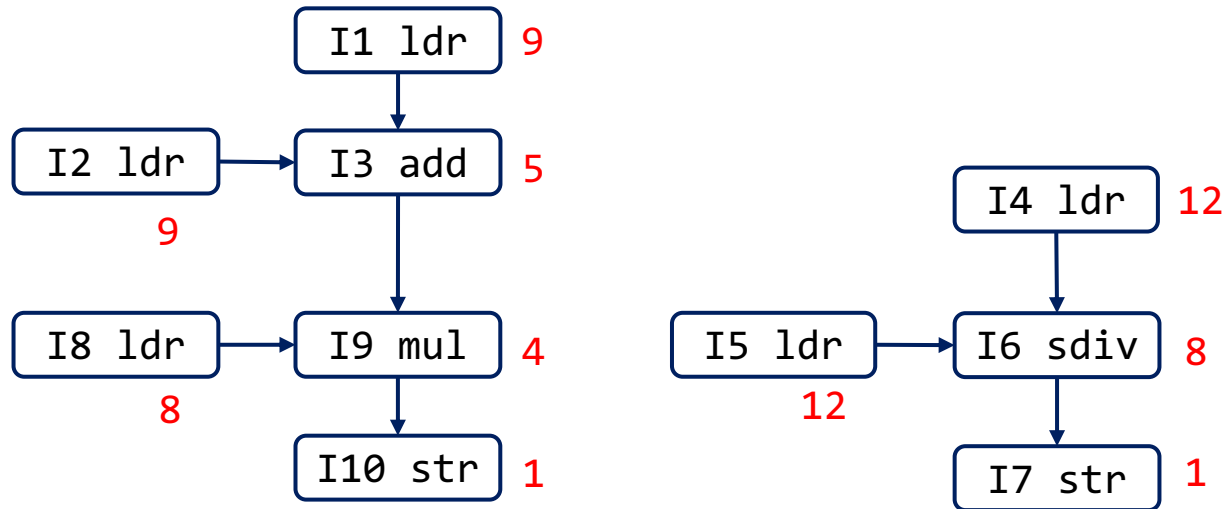


I1	ldr x9, [sp, -12]
I2	ldr x10, [sp, -16]
I3	add x9, x9, x10
I4	ldr x12, [sp, -20]
I5	ldr x11, [sp, -24]
I6	sdiv x11, x12, x11
I7	str [sp, -24], x11
I8	ldr x13, [sp, -28]
I9	mul x13, x9, x13
I10	str x13, [sp, -28]



# 更新Latency并排序

- $I_4=I_5 > I_1=I_2 > I_6=I_8 > I_3 > I_9 > I_7=I_{10}$



# 调度方案开销

- I4=I5>I1=I2>I6=I8>I3>I9>I7=I10

- 开销: 14

开始	结束	指令	
1	4	I4	ldr x12, [sp, -20]
2	5	I5	ldr x11, [sp, -24]
3	6	I1	ldr x9, [sp, -12]
4	7	I2	ldr x10, [sp, -16]
6	12	I6	sdiv x11, x12, x11
7	10	I8	ldr x13, [sp, -28]
8	8	I3	add x9, x9, x10
11	13	I9	mul x13, x9, x13
13	13	I7	str x11, [sp, -24]
14	14	I10	str x13, [sp, -28]

# 进一步优化（vs CPU乱序执行）

- 可尽早执行已经满足了依赖的指令
- I6和I8互换，I7和I0互换
  - 开销：13

开始	结束	指令	
1	4	I4	ldr x12, [sp, -20]
2	5	I5	ldr x11, [sp, -24]
3	6	I1	ldr x9, [sp, -12]
4	7	I2	ldr x10, [sp, -16]
5	8	I8	ldr x13, [sp, -28]
6	12	I6	sdiv x11, x12, x11
8	8	I3	add x9, x9, x10
9	11	I9	mul x13, x9, x13
12	12	I10	str x13, [sp, -28]
13	13	I7	str x11, [sp, -24]

# 表调度算法

```
Clock = 1
Ready = {指令依赖图的所有叶子节点}
Active = {}
While (Ready  $\cup$  Active  $\neq \emptyset$ ){
    foreach I in Active {
        if Start(I) + Cost(I) < Clock {
            remove I;
            foreach C in I.next {
                if C isReady
                    Ready.add(C);
            }
        }
    }
    if (Ready  $\neq \emptyset$ ){
        Ready.remove(any I);
        Start(I) = Clock;
        Active.add(I);
    }
    Clock = Clock + 1;
}
```

保存已满足依赖的指令

保存正在执行的指令

指令执行完成

分析其next指令是否满足依赖

执行Ready表中的一条指令

# 思考

- 对比CPU乱序执行和编译器指令调度
  - 参考：<https://people.eecs.berkeley.edu/~pattrsn/252F96/Lecture04.pdf>
- 分析寄存器分配和指令调度的先后顺序对结果的影响

### 三、窥孔优化

---

# 窥孔优化

- 编译器实现时难以考虑周全，统一的处理规则容易产生冗余
- 基于局部的几条指令（窗口大小）进行模式匹配和优化
- 基于案例分析：
  - [https://github.com/hxuhack/compiler\\_project/blob/24f-assignment5/src/example/BFS.S](https://github.com/hxuhack/compiler_project/blob/24f-assignment5/src/example/BFS.S)



# 窥孔优化：窗口大小1

bb1:

```
...  
mov     w11, w0  
mov     w9, 0          # %r255 = add i32 0, 0  
mov     w9, 0          # %r258 = add i32 0, 0  
mov     w9, w9  
mov     w10, 0         # %r256 = add i32 0, 0  
mov     w10, 0         # %r259 = add i32 0, 0  
mov     w10, w10  
mov     w10, w10       #bb2: %r260 = phi i32 ...  
mov     w12, w11       #bb2: %r261 = phi i32 ...  
b       bb2           # br label %bb2
```

bb2:

冗余模式：

mov r1, r1



优化方式：

删除

# 窥孔优化：窗口大小2

bb1:

...

mov w11, w0

mov w9, 0

mov w9, 0

mov w10, 0

mov w10, 0

mov w12, w11

b bb2

bb2:

# %r255 = add i32 0, 0

# %r258 = add i32 0, 0

# %r256 = add i32 0, 0

# %r259 = add i32 0, 0

#bb2: %r261 = phi i32 ...

# br label %bb2

冗余模式:

mov r1, ?

mov r1, ?



优化方式:

删除第一条

冗余模式:

b bb2?

bb2:



优化方式:

删除跳转指令

# 窥孔优化：窗口大小2

quickread函数：

bb1:

```
mov    x16, 0
sub    sp, sp, x16
mov    w9, 0
stp    x9, x10, [sp, 16]!
stp    x11, x12, [sp, -16]!
stp    x13, x14, [sp, -16]!
str    x15, [sp, #-8]!
stp    x29, x30, [sp, -16]!
```

冗余模式：

```
mov r1, 0
sub r?, r?, r1
```



优化方式：

删除sub指令

# 思考

- 更多窥孔优化模式？

## 四、利用CPU特性优化

---

# 内存预取：PRFM（aarch64）

```
# PRFM <type>, [<base>, <offset>]  
PRFM PLDL1KEEP, [sp, 256] # 读操作, 预取到L1 Cache  
# PLDL2KEEP: 读操作, 预取到L2 Cache  
# STL2KEEP: 写操作, 预取到L2 Cache  
#...更多指令  
ldr w1, [x0, 256]
```

```
void __builtin_prefetch (const void *addr, int rw, int locality);
```

❑ rw（Read/Write hint，读/写提示）：

- 0 表示数据主要用于读取（读取预取）
- 1 表示数据主要用于写入（写入预取）

❑ locality（预取局部性）：

- 0 表示较低的预取局部性，即只预取当前访问的附近数据
- 1 表示较高的预取局部性，通常会预取更多的连续数据
- 3 表示最强的预取局部性，可能会预取较远的数据

# 应用示例

```
#define ARRAY_SIZE 10000
void no_prefetch(int *arr) {
    for (int i = 0; i < ARRAY_SIZE; i++) {
        usleep(100);
        arr[i] = arr[i] * 2;
    }
}

void with_prefetch(int *arr) {
    for (int i = 0; i < ARRAY_SIZE; i+=4) {
        __builtin_prefetch(&arr[i], 1, 1);
        usleep(100);
        arr[i] = arr[i] * 2;
        arr[i+1] = arr[i+1] * 2;
        arr[i+2] = arr[i+2] * 2;
        arr[i+3] = arr[i+3] * 2;
    }
}
```

```
#: clang prefetch.c
```

```
#: ./a.out
```

```
No prefetch time: 0.0153 seconds
```

```
With prefetch time: 0.0041 seconds
```

# 并行指令特化：SIMD

- SIMD: Single Instruction Multiple Data
  - ARM: Neon, 含有32个128bit寄存器v0-v31
  - X86: SSE、AVX
- 典型应用场景: 向量运算



# 示例：向量运算

```
#include <arm_neon.h>
int x[4] = {1, 2, 3, 4};
int y[4] = {5, 6, 7, 8};
void avadd(int* z) {
    int32x4_t vec_x = vld1q_s32(x);
    int32x4_t vec_y = vld1q_s32(y);
    int32x4_t vec_z = vaddq_s32(vec_x, vec_y);
    vst1q_s32(z, vec_z);
}
```

```
#: clang -march=armv8-a+simd test.c -O2 -S
```



```
_avadd:
    adrp x8, _x@PAGE
    ldr q0, [x8, _x@PAGEOFF]
    adrp x8, _y@PAGE
    ldr q1, [x8, _y@PAGEOFF]
    add.4s v0, v1, v0
    str q0, [x0]
    ret
```

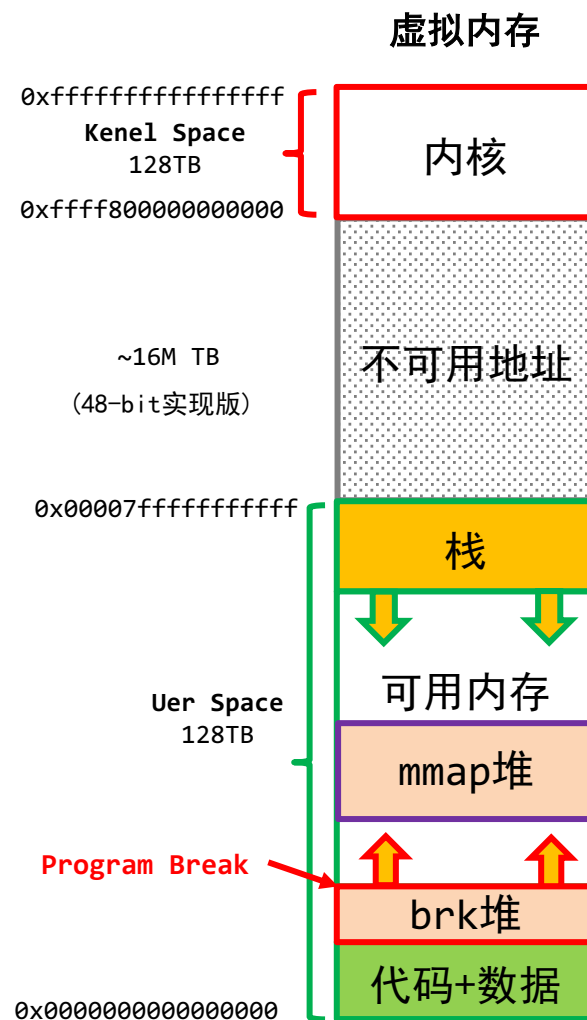
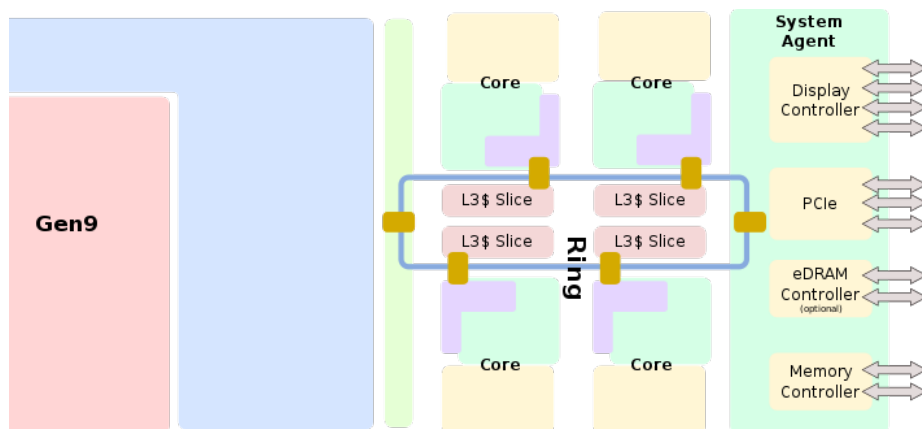
# 并行计算：多核处理器（multicore）

- 多线程并行计算

- 独立栈空间
- 共享堆空间

- 关键问题：

- 任务分解：数据分块
- 数据更新同步



# 数据竞争问题：多线程的例子

```
#include <pthread.h>
#include <assert.h>
#include <stdio.h>
#define NUM 100
int global_cnt = 0;

void *mythread(void *in) {
    for (int i=0; i<NUM; i++)
        global_cnt++;
}

int main(int argc, char** argv) {
    pthread_t tid[NUM];
    for (int i=0; i<NUM; i++){
        assert(pthread_create(&tid[i], NULL, mythread, NULL)==0);
    }
    for (int i=0; i<NUM; i++){
        pthread_join(tid[i], NULL);
    }
    assert(global_cnt==NUM*NUM);
}
```

多个线程并发访问

assertion fail

# 修复方式：原子访问

方式一：声明为原子变量类型

```
define NUM 100  
atomic_int global_cnt;
```

```
void *mythread(void *from) {  
    //__atomic_fetch_add(&global_cnt, 1, __ATOMIC_SEQ_CST);  
    for (int i=0; i<NUM; i++)  
        global_cnt++;  
}
```

方式二：使用原子运算API

```
int main(int argc, char** argv) {  
    pthread_t tid[NUM];  
    for (int i=0; i<NUM; i++){  
        assert(pthread_create(&tid[i], NULL, mythread, NULL)==0);  
    }  
    for (int i=0; i<NUM; i++){  
        pthread_join(tid[i], NULL);  
    }  
    assert(global_cnt==NUM*NUM);  
}
```

# 原子访问的实现方式：原子指令

- ldaddal: 原子加法指令

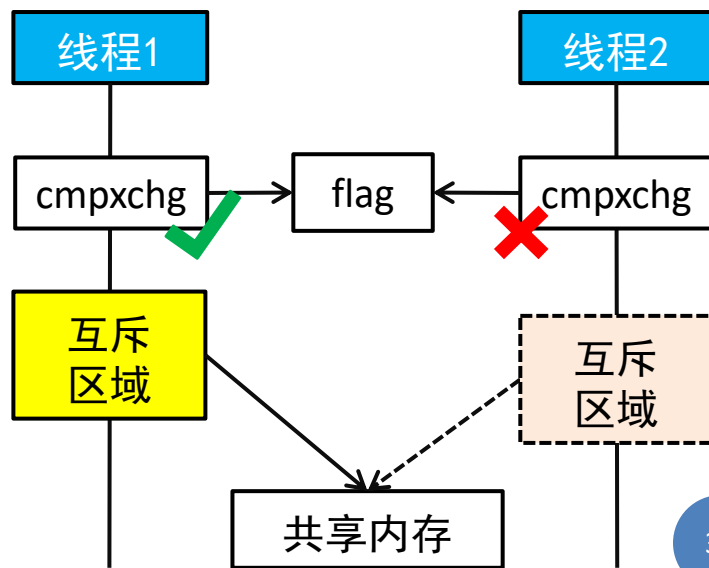
```
adrp x9, _global_cnt@PAGE
ldr w8, [x9, _global_cnt@PAGEOFF]
add w8, w8, #1
str w8, [x9, _global_cnt@PAGEOFF]
b LBB0_3
```



```
mov w8, #1
adrp x9, _global_cnt@PAGE
add x9, x9, _global_cnt@PAGEOFF
ldaddal w8, w8, [x9]
```

# 如何实现一段连续指令的原子性？

- 基于Compare and Set/Swap机制
- ldaxr：从内存地址加载一个值，标记该地址为“独占访问”
  - 不会失败，允许并发读
  - 其它线程的ldaxr并发操作会清除“独占访问”状态
- stlxr：将一个值存储到内存，该地址需为“独占访问”状态。
  - 可能会失败



# 锁的例子

```
#include <stdatomic.h>
#include <stdbool.h>
volatile atomic_int lock = 0;
// 自旋锁，初始值为0（未锁定）
void acquire_lock(volatile atomic_int *lock) {
    int expected = 0;
    while (!atomic_compare_exchange_weak(lock, &expected, 1)) {
        expected = 0; // 如果交换失败，需要重置 expected 的值
    }
}

void release_lock(volatile atomic_int *lock) {
    atomic_store(lock, 0); // 释放锁
}

int main() {
    acquire_lock(&lock);    // 临界区代码
    release_lock(&lock);
    return 0;
}
```

# 锁的例子

acquire\_lock:

mov w1, 1

.L2:

ldaxr w2, [x0]

cmp w2, 0

bne .L6

stlxr w3, w1, [x0]

cmp w3, 0

.L6:

bne .L2

ret

标记[x0]为“独占访问”

如果“独占访问”标记被其它线程清除，则指令失败，w3!=0

release\_lock:

stlr wzr, [x0]

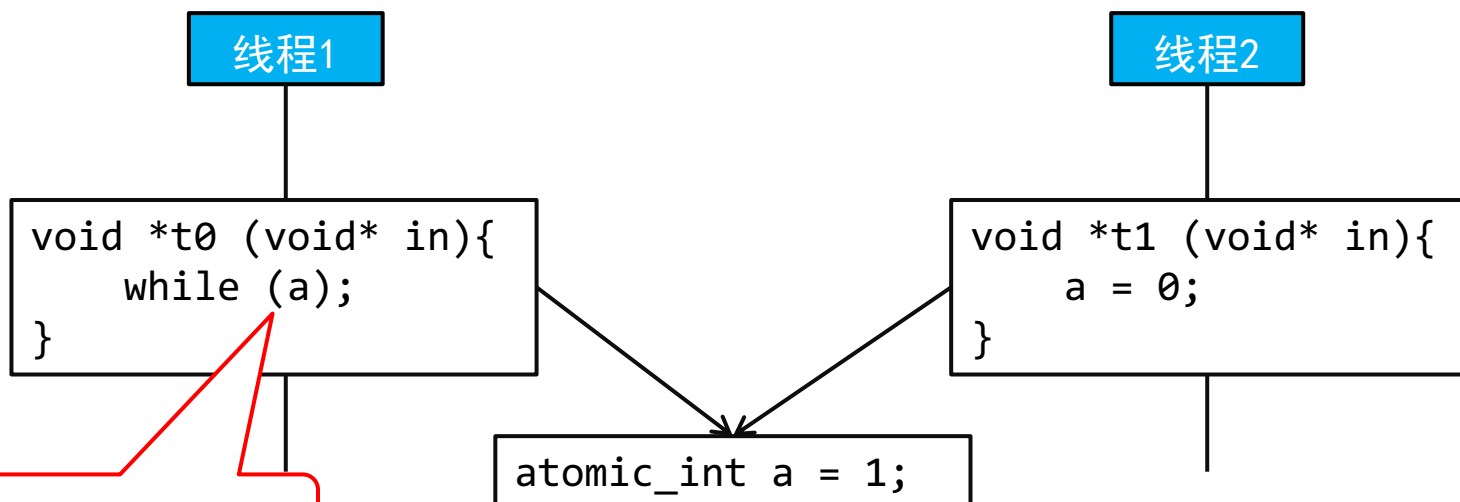
ret



# 并发程序架构



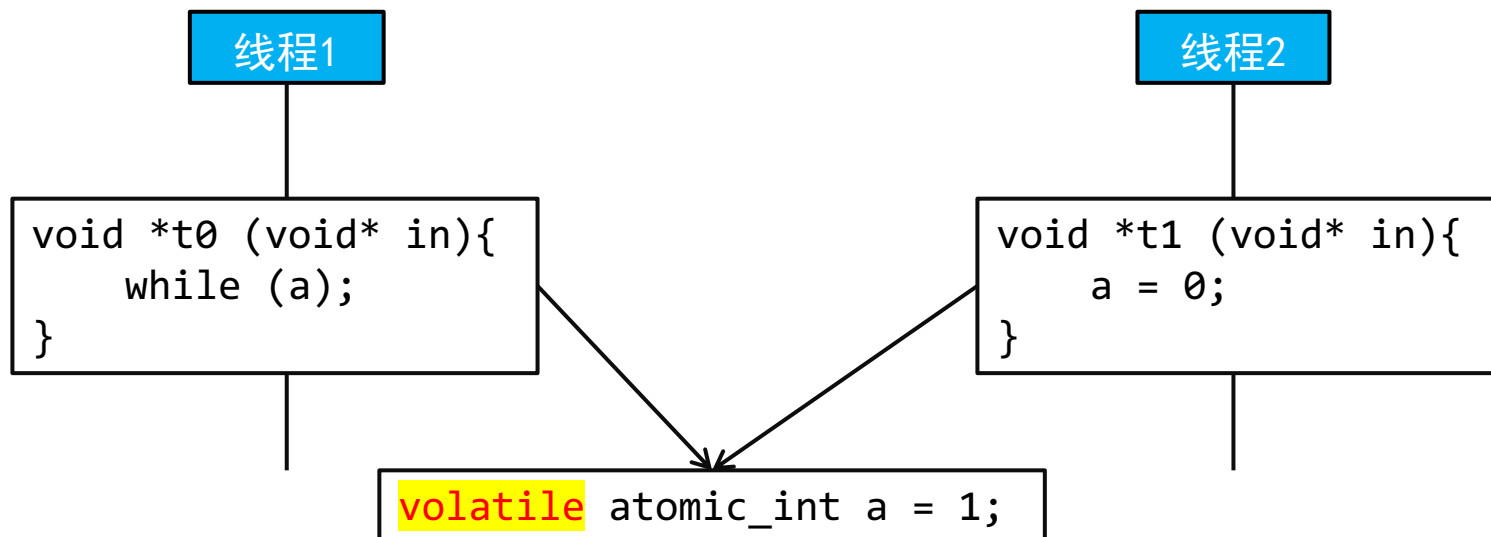
# 编译优化可能会产生副作用：举例



常量优化：a恒等于1？

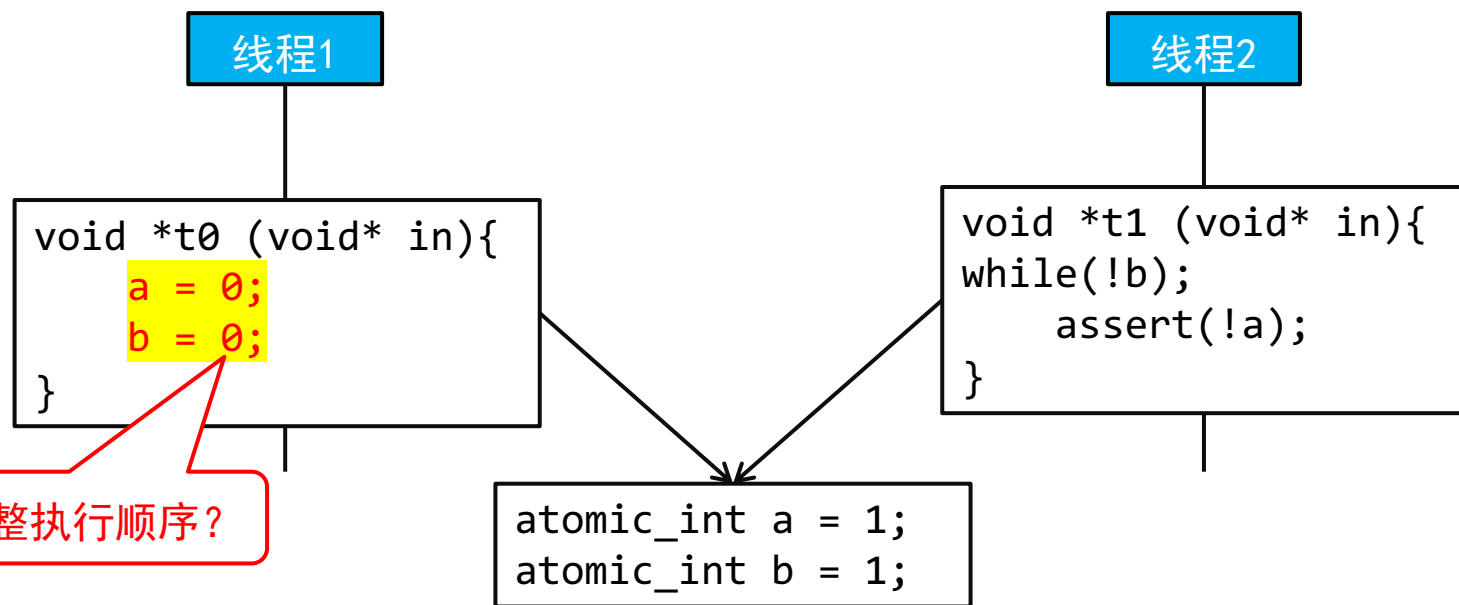
# 易变内存访问：Volatile

- 告诉编译器丢弃寄存器中的值，重新从内存加载



# 指令重排问题

- 指令执行的先后顺序不同会对其它线程产生影响
- 编译优化可能会误将指令重排



# 内存屏障：Memory Barrier/Fence

- DMB屏障：编译器确保屏障之前的指令在其之前完成
- DSB屏障：内存访问同步，强制寄存器刷新

//空指令

```
define barrier() __asm__ volatile("dmb sy" ::: "memory");
```

线程1

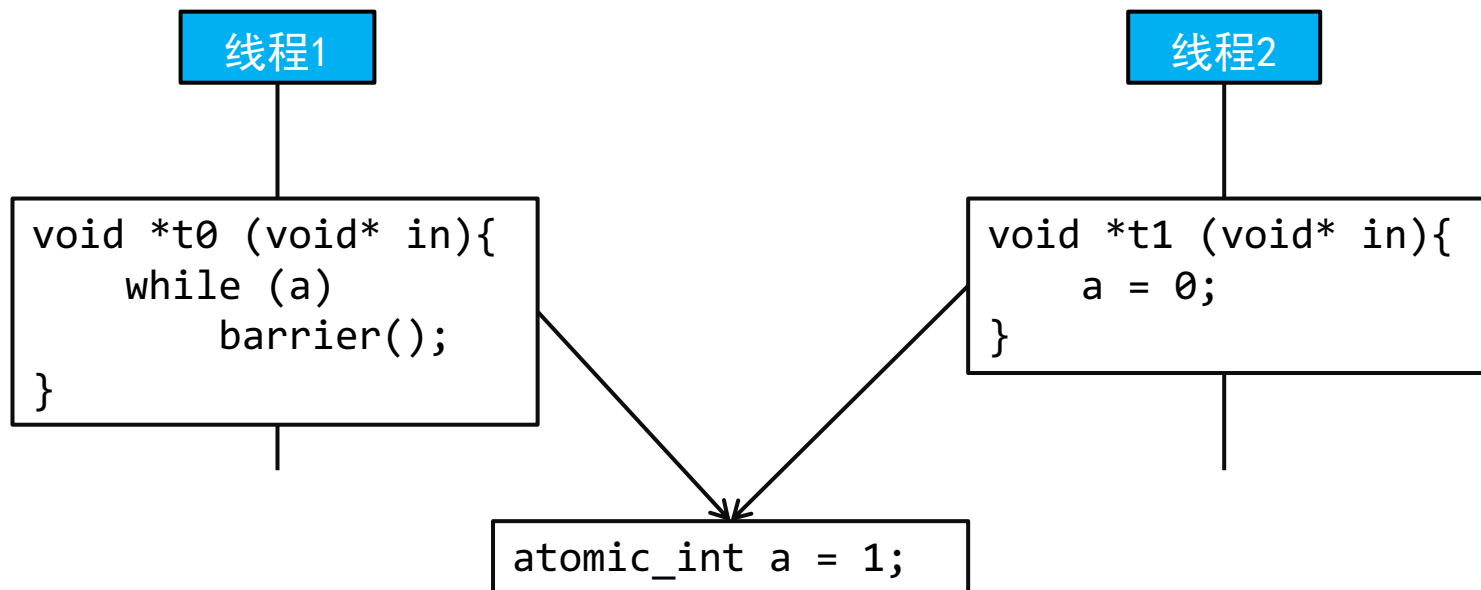
```
void *t0 (void* in){  
    a = 0;  
    barrier();  
    b = 0;  
}
```

线程2

```
void *t1 (void* in){  
    while(!b);  
    assert(!a);  
}
```

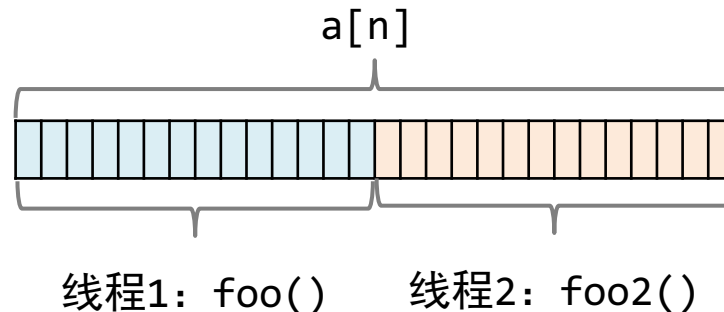
```
atomic_int a = 1;  
atomic_int b = 1;
```

# 使用内存屏障



# 传统多线程

```
int test(){
    int a[n];
    for (int i = 0; i < n; i++) {
        a[i] = 2 * i;
    }
    return 0;
}
```



```
int paratest(){
    pthread_t t1;
    pthread_t t2;
    pthread_create(&t1, NULL, foo1, NULL);
    pthread_create(&t2, NULL, foo2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

```
int a[n];
foo1(){
    for (int i=0; i<n/2; i++)
        a[i] = 2 * i;
}
foo2(){
    for (int i=n/2; i<n; i++)
        a[i] = 2 * i;
}
```

# OpenMP应用举例


```
int test(){
    unsigned long long start = rdtsc();
    int a[100000];
    pragma omp parallel for num_threads(2)
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }
    unsigned long long cycles = rdtsc() - start;
    printf("cycles = %d\n", cycles);
    return 0;
}
```



# 循环中的数据依赖问题： 示例1

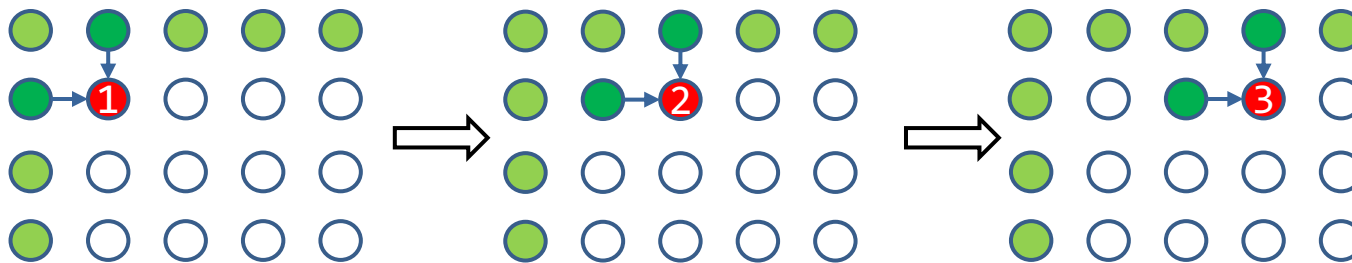
```
int test(){  
    int a[20];  
    a[0] = 1;  
    a[1] = 1;  
    pragma omp parallel for num_threads(4)  
    for (int i = 0; i < 20; i++) {  
        a[i] = a[i-1] + a[i-2];  
    }  
    return 0;  
}
```

数据依赖，无法并行

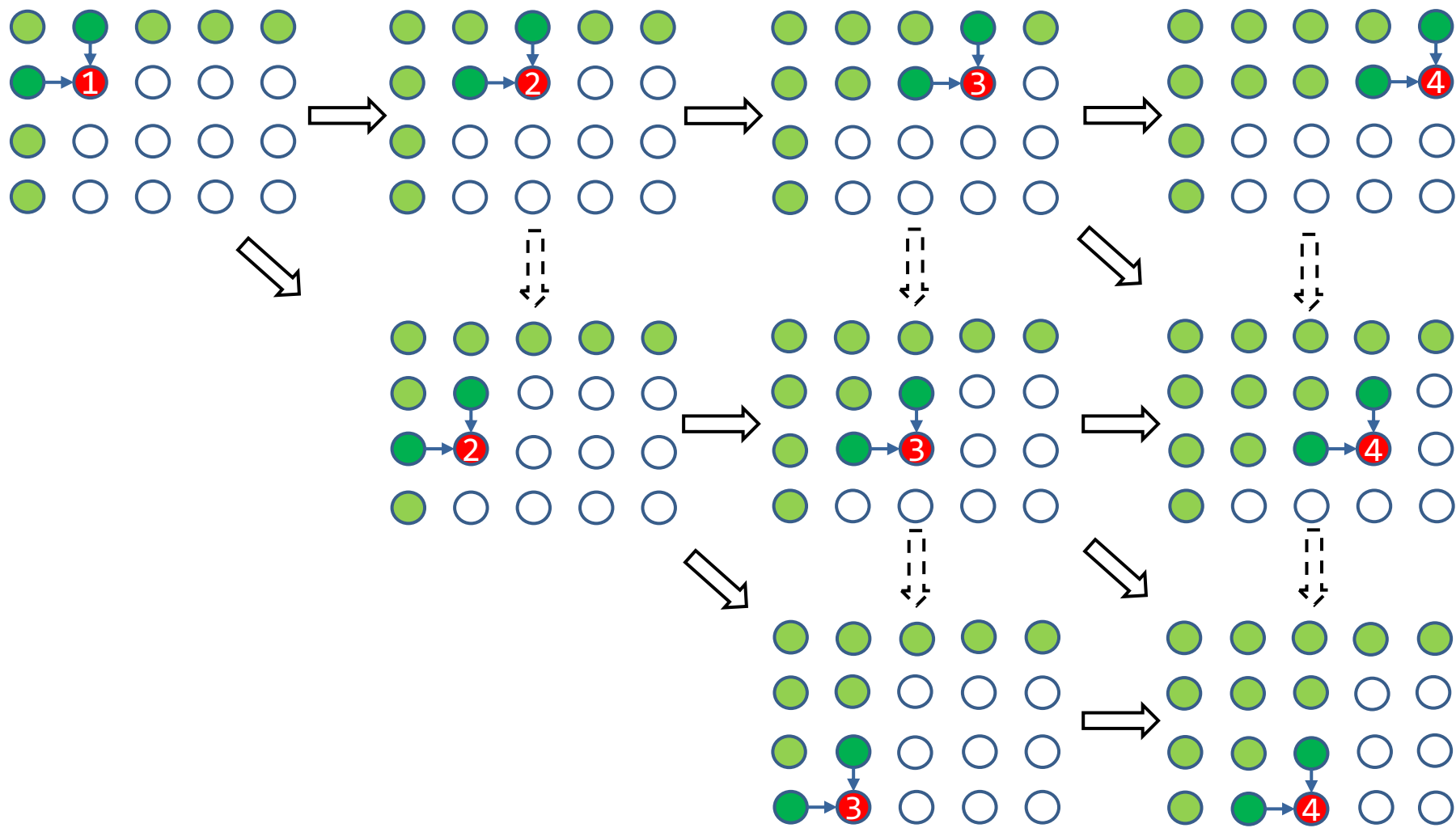


## 循环中的数据依赖问题： 示例2

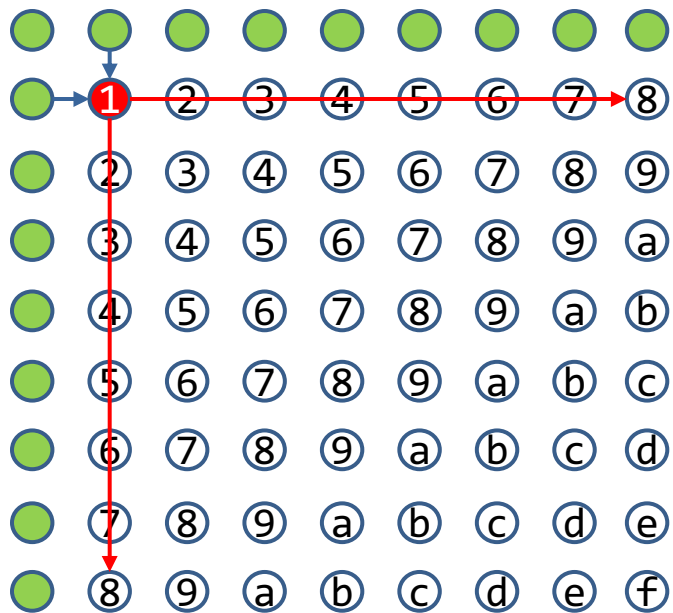
```
for(int i = 1; i < n; i++) {  
    for(int j = 1; j < n; j++) {  
        a[i][j] = a[i-1][j] + a[i][j-1];  
    }  
}
```



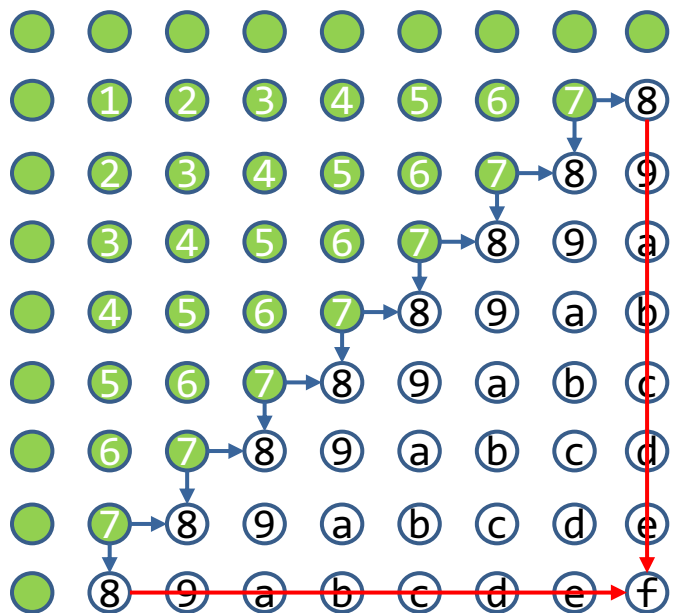
# 依赖分析



# 依赖优化: Polyhedral model



```
for(int i = 1; i < n-1; i++) {  
    for(int j = 1; j < i+1; j++) {  
        a[i-j+1][j] = a[i-j][j]  
            + a[i-j+1][j-1];  
    }  
}
```



```
for(int i = 1; i < n-1; i++) {  
    for(int j = n-1; j > i-1; j--) {  
        a[j-i+1][j] = a[j-i+1][j]  
            + a[j-i][j];  
    }  
}
```