COMP130014.02 编译

第十三讲：指令调度与优化

徐 辉

xuh@fudan.edu.cn

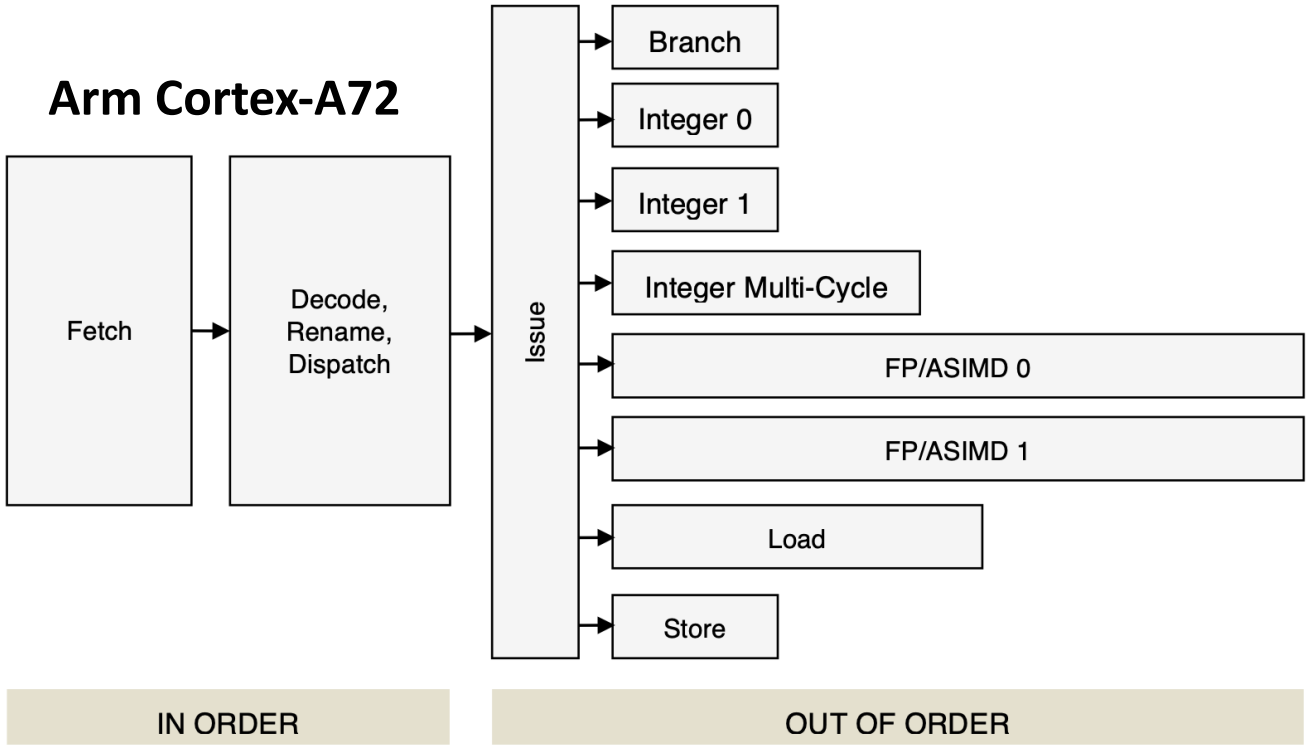
大纲

- 一、后端优化问题
- 二、指令调度优化
- 三、更多后端优化

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（μOP）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
```

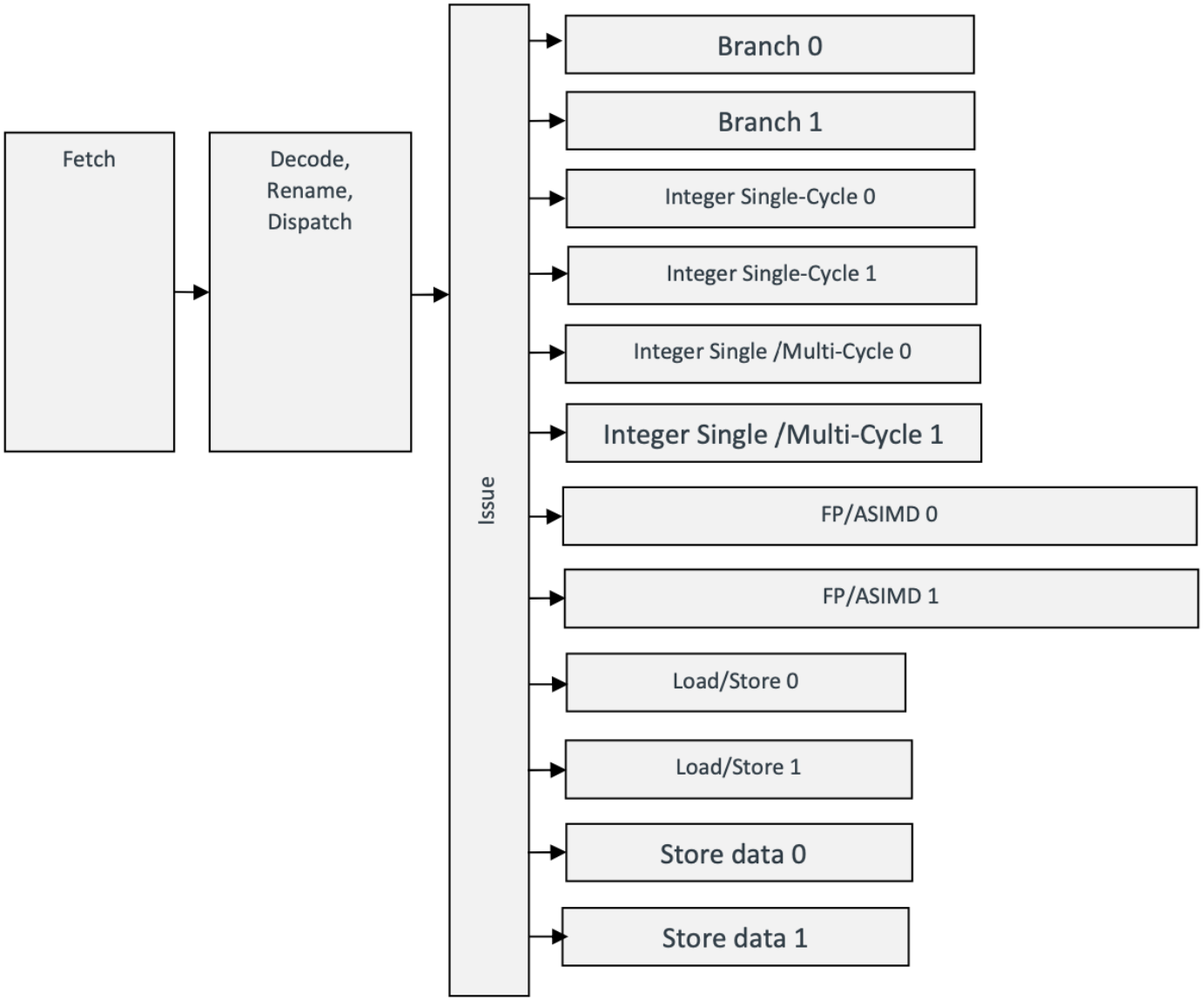
Stage	Clock Cycles							
	1	2	3	4	5	6	7	8
Fetch	add	add	mul	sub	add			
Decode		add	add	mul	sub	add		
Execute(I0)			add	add			add	sub
Execute(I1)								
Execute(M)					mul			

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

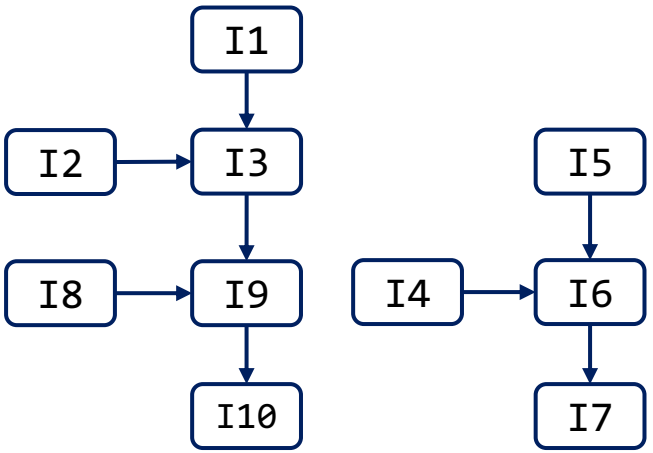
- 数据依赖关系
  - 写-读依赖（RAW/Read-After-Write）： true-dependency
  - 读-写反依赖（WAR/Write-After-Read）：anti-dependency
- 结构性影响（structural hazard）
  - 一条指令由多条微指令组成
  - 相邻指令的微指令可能会竞争ports的使用
- 控制流影响（control hazard）
  - 条件跳转或分支预测

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]

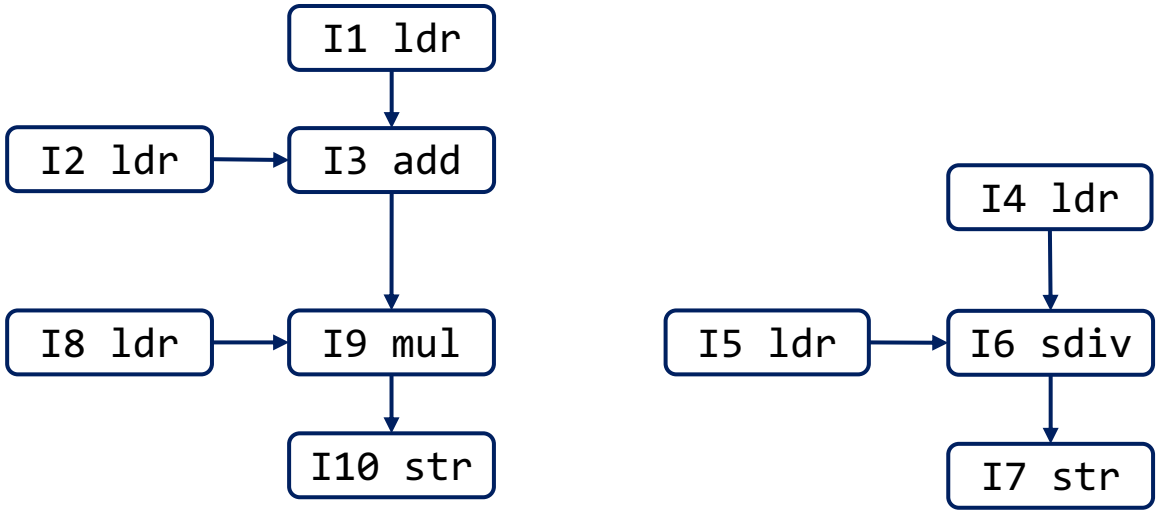


指令依赖关系

编译器的指令调度问题

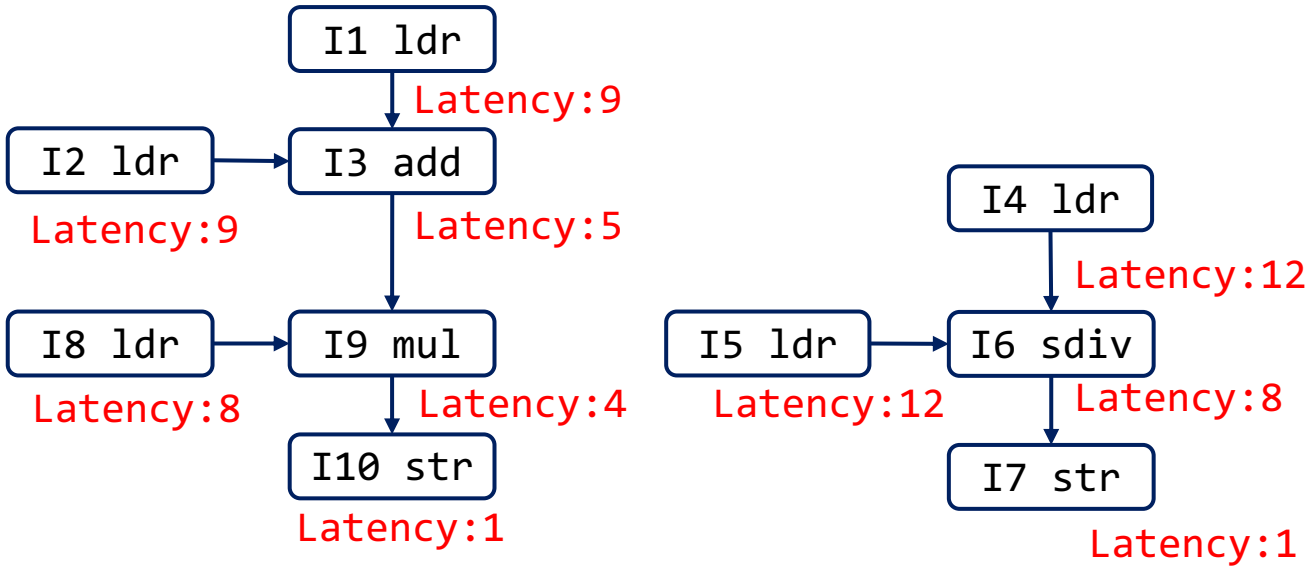
- 假设：
  - 每个cycle可以执行一条指令
  - 多条指令可以并行
  - 单条指令开销稳定
- 应如何确定最佳的指令执行序列？
  - 执行顺序应满足数据依赖关系

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



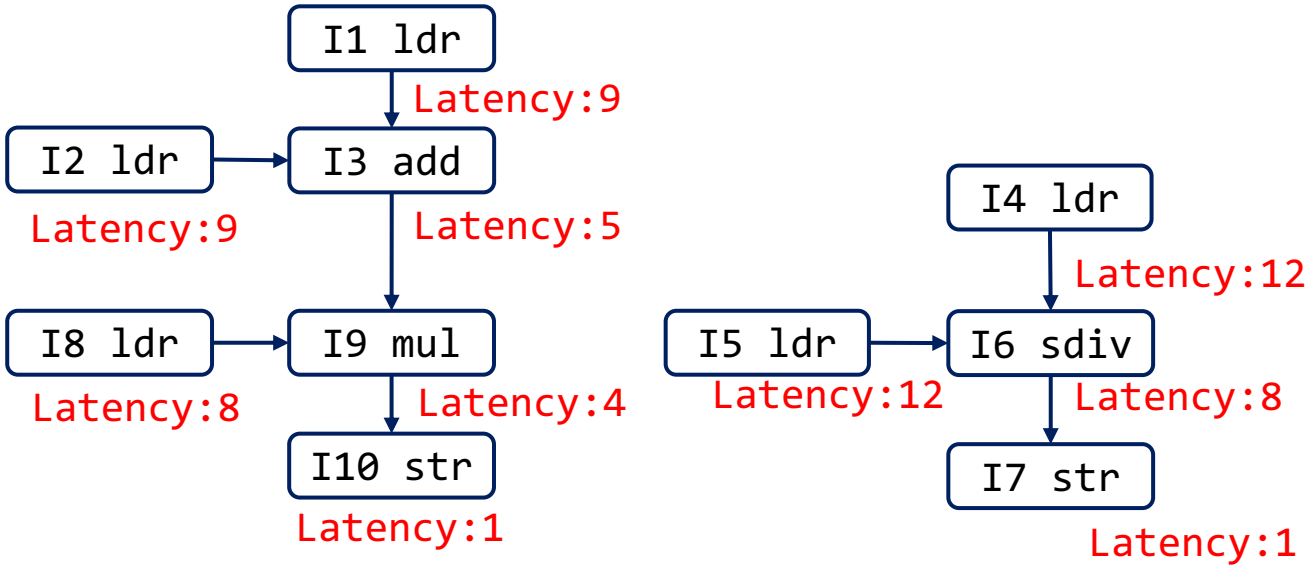
指令调度思路

- 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

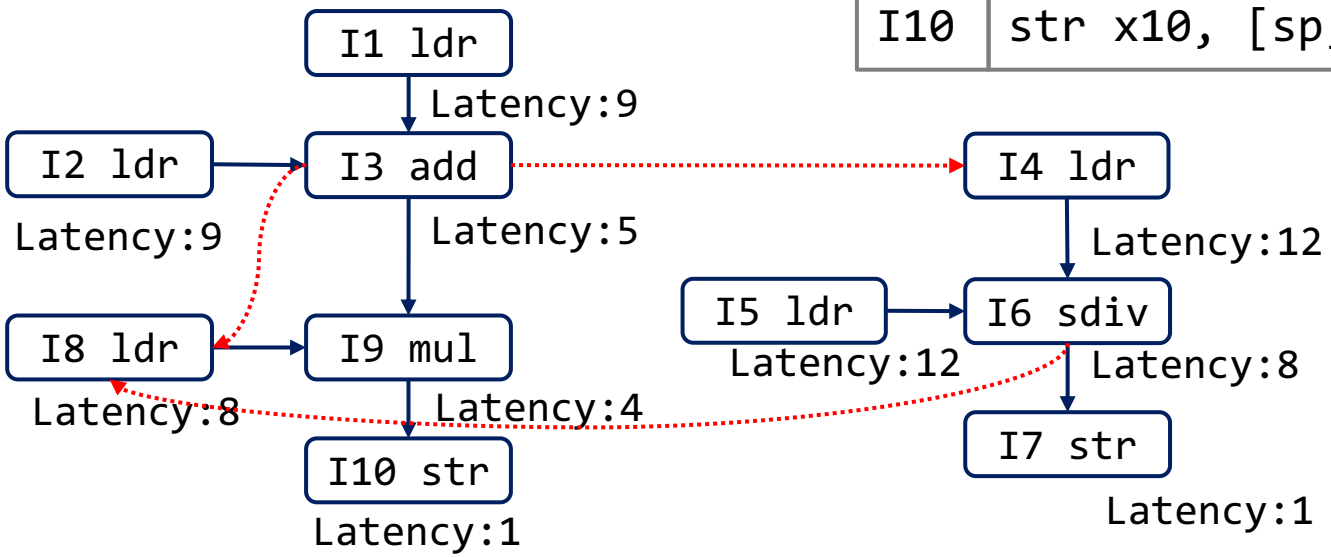
- 根据latency从大到小对指令进行排序
  - I4=I5>I6>I1=I2>I8>I3>I9>I7=I10
- 优先执行latency大的指令



读-写反依赖（WAR）问题

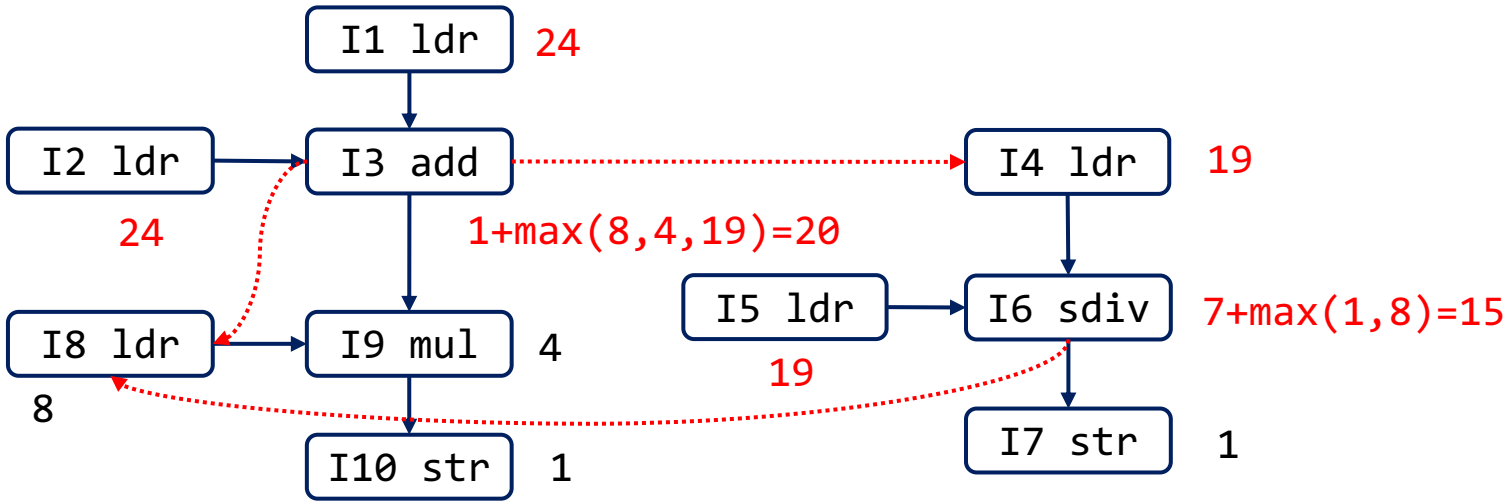
- I3执行完I4和I8才能执行
  - 否则会影响I3的计算结果
- I6执行完才能执行I8
- 寄存器分配（复用）导致

```
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

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



调度方案开销

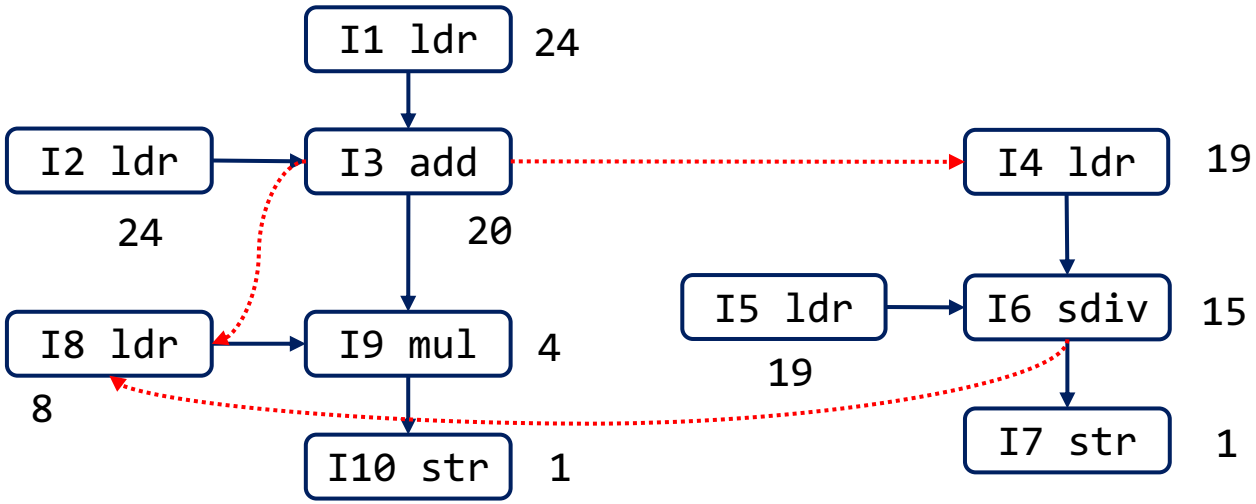
- I1=I2>I3>I4=I5>I6>I8>I9>I7=I10
  - 开销：26

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）

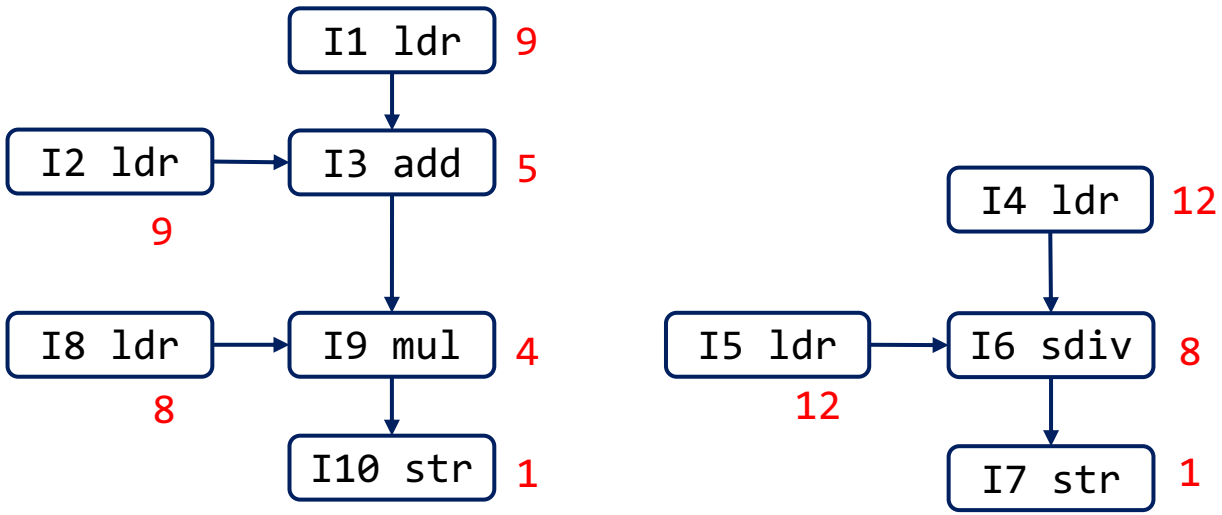
I1	ldr x9, [sp, #-12]
I2	ldr x10, [sp, #-16]
I3	add x9, x9, x10
I4	ldr x10, [sp, #-20]
I5	ldr x11, [sp, #-24]
I6	sdiv x11, x10, x11
I7	str x11, [sp, #-24]
I8	ldr x10, [sp, #-28]
I9	mul x10, x9, x10
I10	str x10, [sp, #-28]

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

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



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

- 假设：线性代码、无反依赖

```
Clock = 1
Ready = {指令依赖图的所有叶子节点}
Active = {}
While (Ready ∪ Active ≠ ∅){
    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 ≠ ∅){
        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

三、更多后端优化

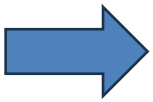
案例回顾

```
fn fac(n: int) -> int {
    let r = 1;
    while (n>0) {
        r = r * n;
        n = n-1;
    }
    ret r;
}
```

```
_fac:
    sub sp, sp, #16
    str w0, [sp, #12]
    mov w9, #1
    str w9, [sp, #8]
LBB0_1:
    ldr w9, [sp, #12]
    ldr w10, [sp, #8]
    cmp w9, #0
    b.le LBB0_3
LBB0_2:
    mul w10, w10, w9
    sub w9, w9, #1
    str w10, [sp, #8]
    str w9, [sp, #12]
    b LBB0_1
LBB0_3:
    ldr w0, [sp, #8]
    add sp, sp, #16
    ret
```

窥孔优化

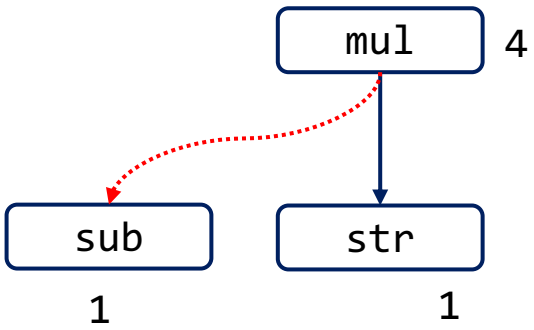
```
_fac:
    sub sp, sp, #16
    str w0, [sp, #12]
    mov w9, #1
    str w9, [sp, #8]
    ldr w9, [sp, #12]
LBB0_1:
    ldr w9, [sp, #12]
    ldr w10, [sp, #8]
    cmp w9, #0
    b.le LBB0_3
LBB0_2:
    mul w10, w10, w9
    sub w9, w9, #1
    str w10, [sp, #8]
    str w9, [sp, #12]
    b LBB0_1
LBB0_3:
    ldr w0, [sp, #8]
    add sp, sp, #16
    ret
```



```
_fac:
    sub sp, sp, #16
    mov w9, #1
    str w9, [sp, #8]
LBB0_1:
    ldr w10, [sp, #8]
    cmp w0, #0
    b.le LBB0_3
LBB0_2:
    mul w10, w10, w0
    sub w0, w0, #1
    str w10, [sp, #8]
    b LBB0_1
LBB0_3:
    ldr w0, [sp, #8]
    add sp, sp, #16
    ret
```

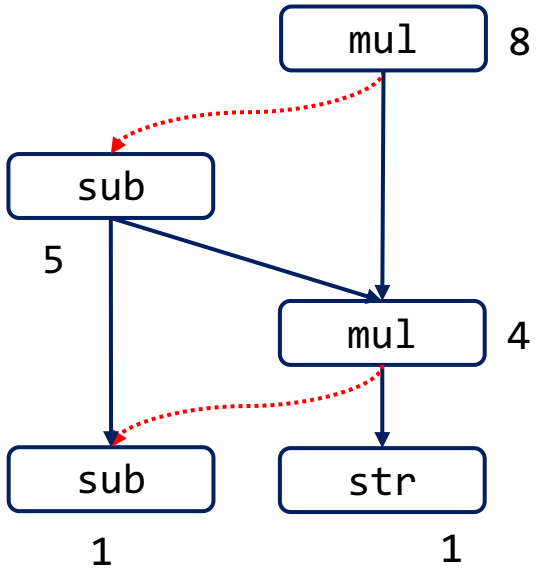
重新理解循环展开优化

```
_fac:
    sub sp, sp, #16
    mov w9, #1
    str w9, [sp, #8]
LBB0_1:
    ldr w10, [sp, #8]
    cmp w0, #1
    b.le LBB0_3
LBB0_2:
    mul w10, w10, w0
    sub w0, w0, #1
    str w10, [sp, #8]
    b LBB0_1
LBB0_3:
    ldr w0, [sp, #8]
    add sp, sp, #16
    ret
```



重新理解循环展开优化

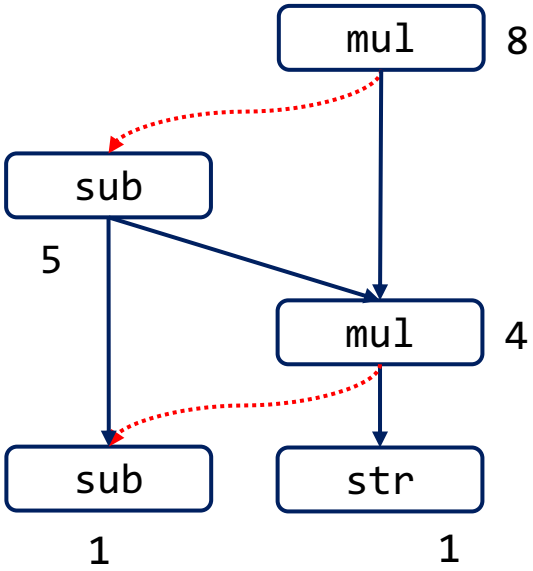
```
_fac:
    sub sp, sp, #16
    mov w9, #1
    str w9, [sp, #8]
LBB0_1:
    ldr w10, [sp, #8]
    cmp w0, #1
    b.le LBB0_3
LBB0_2:
    mul w10, w10, w0
    sub w0, w0, #1
    mul w10, w10, w0
    sub w0, w0, #1
    str w10, [sp, #8]
    b LBB0_1
LBB0_3:
    cmp w0, #1
    b.eq LBB0_4
    mul w10, w10, w0
    sub w0, w0, #1
    str w10, [sp, #8]
LBB0_4:
    ldr w0, [sp, #8]
    add sp, sp, #16
```

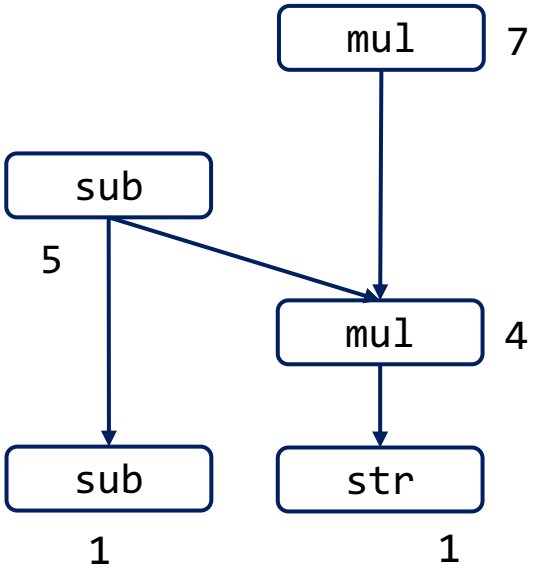


消除WAR依赖

```
_LBB0_2:
    mul w10, w10, w0
    sub w0, w0, #1
    mul w10, w10, w0
    sub w0, w0, #1
    str w10, [sp, #8]
    b LBB0_1
```

```
_LBB0_2:
    mul w10, w10, w0
    sub w11, w0, #1
    mul w10, w10, w11
    sub w0, w11, #1
    str w10, [sp, #8]
    b LBB0_1
```





内存预取：PLD/PRFM（aarch64）

```
PLD [sp, #256] //预取地址为sp+256的内存数据
//...更多指令
ldr w1, [sp, #256]
...
```

```
PRFM PLDL1KEEP, [sp, # 256] //预取到L1 Cache
// PLDL2KEEP: 预取到L2 Cache
//...更多指令
ldr w1, [x0, #256]
...
```

练习

- 设计实验测试内存预取的性能提升效果