Lecture 13

指令选择和调度

徐 辉
xuh@fudan.edu.cn

大纲

- 一、问题背景
- 二、指令选择
- 三、指令调度

一、问题背景

IR指令存在多种ASM翻译方式

一条IR指令，多种ASM翻译方式

```
%r1 = mul i32 %a, 8
```

```
mov w2, #8
mul w0, w1, w2
```

```
lsl w0, w1, #3
```

IR指令组合，多种ASM翻译方式

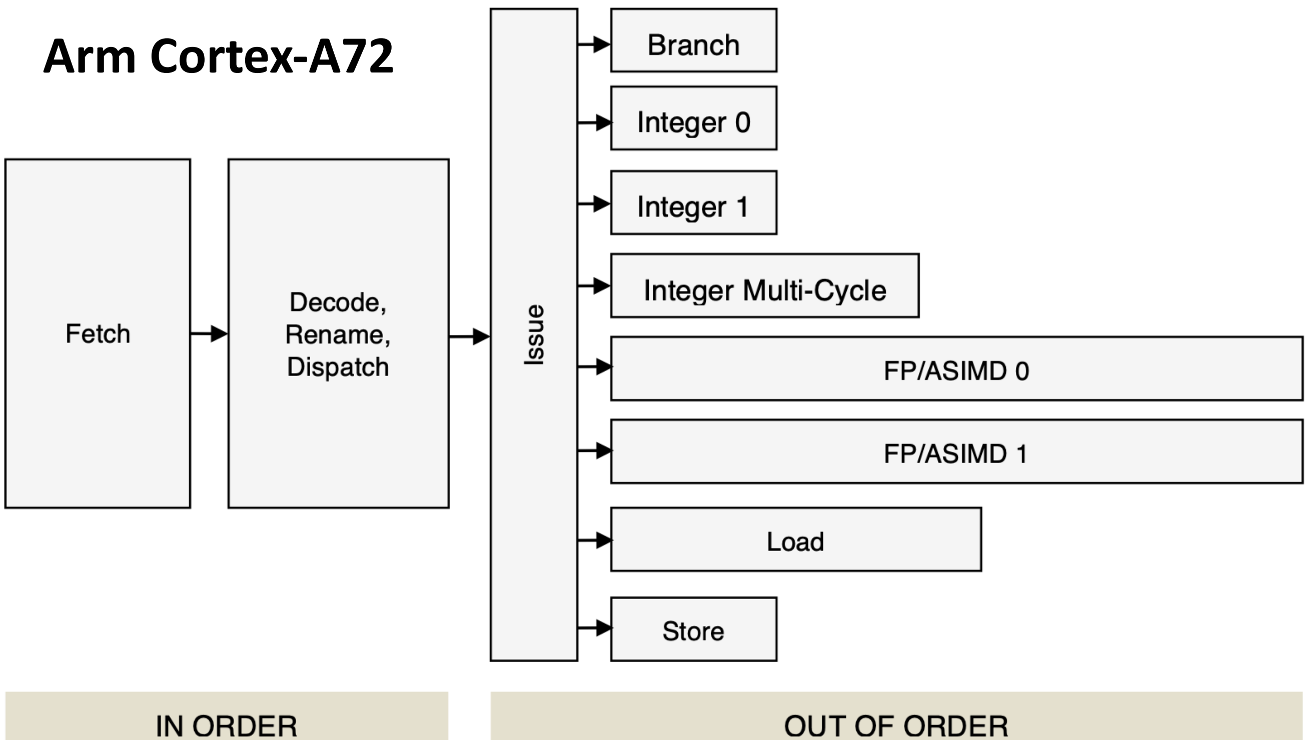
```
%p = getelementptr i32, i32* %array, i32 0, i32 1
%v = load i32, i32* %p
```

```
add x1, x1, #4
ldr w0, [x1]
```

```
ldr w0, [x1, #4]
```

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（Tomasulo算法）



指令执行顺序影响性能

- 不同指令执行效率不同
- 指令之间存在数据依赖关系

```
ADD x1, x2, x3
ADD x4, x5, x6
MUL x0, x2, x3
SUB x2, x0, x1
ADD x4, x4, x5
```

假设MUL需要3个cycles，ADD/SUB需要1个cycle

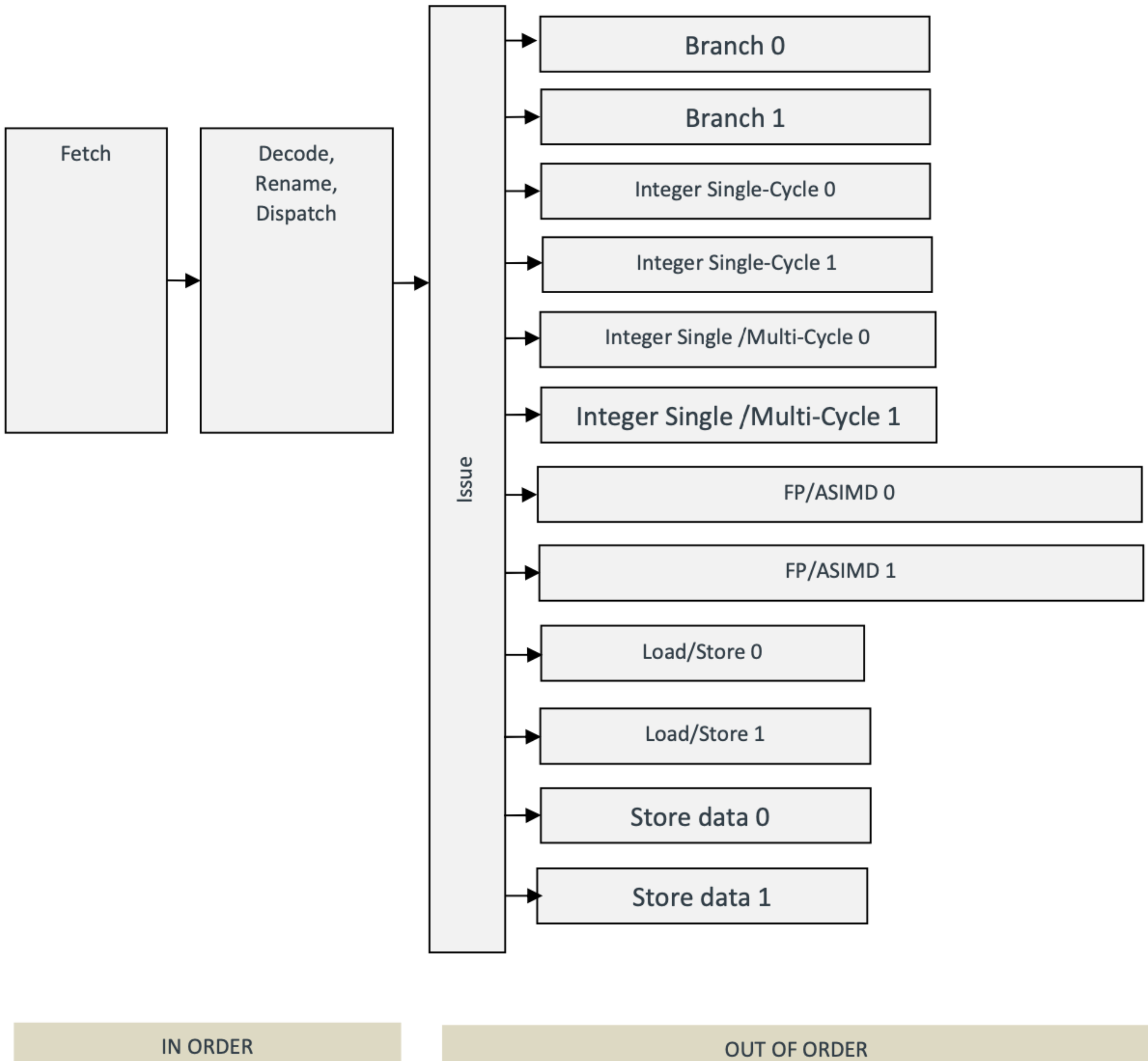
Stage	Clock Cycles							
	1	2	3	4	5	6	7	8
Fetch	ADD	ADD	MUL	SUB	ADD			
Decode		ADD	ADD	MUL	SUB	ADD		
Execute(I0)			ADD	ADD			ADD	SUB
Execute(I1)								
Execute(M)					MUL			

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/ADD	1	2	I0/I1
	SUB/SUBS	1	2	I0/I1
	MUL	3	1	M
	MADD/MSUB	3	1	M
	SDIV	4-20	1/12-1/4	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-A72 Core Software Optimization Guide

Arm Cortex-A77



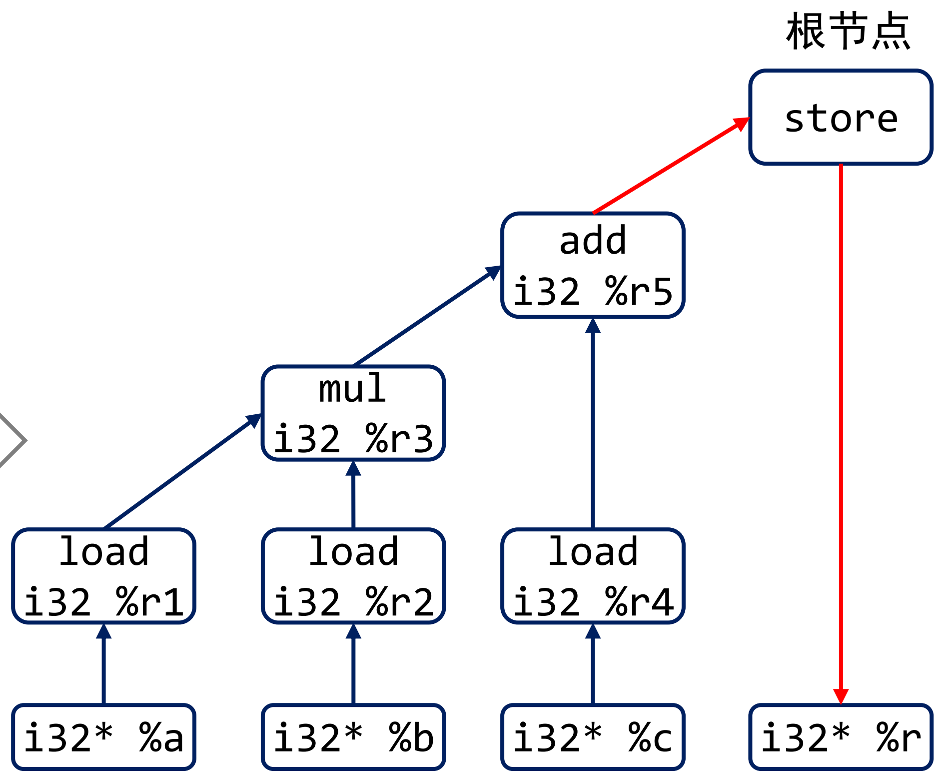
二、指令选择

IR=>DAG

- 将IR转换为表达式树
- 合并表达式树的共同节点得到表达式图DAG

```
r = a * b + c;
```

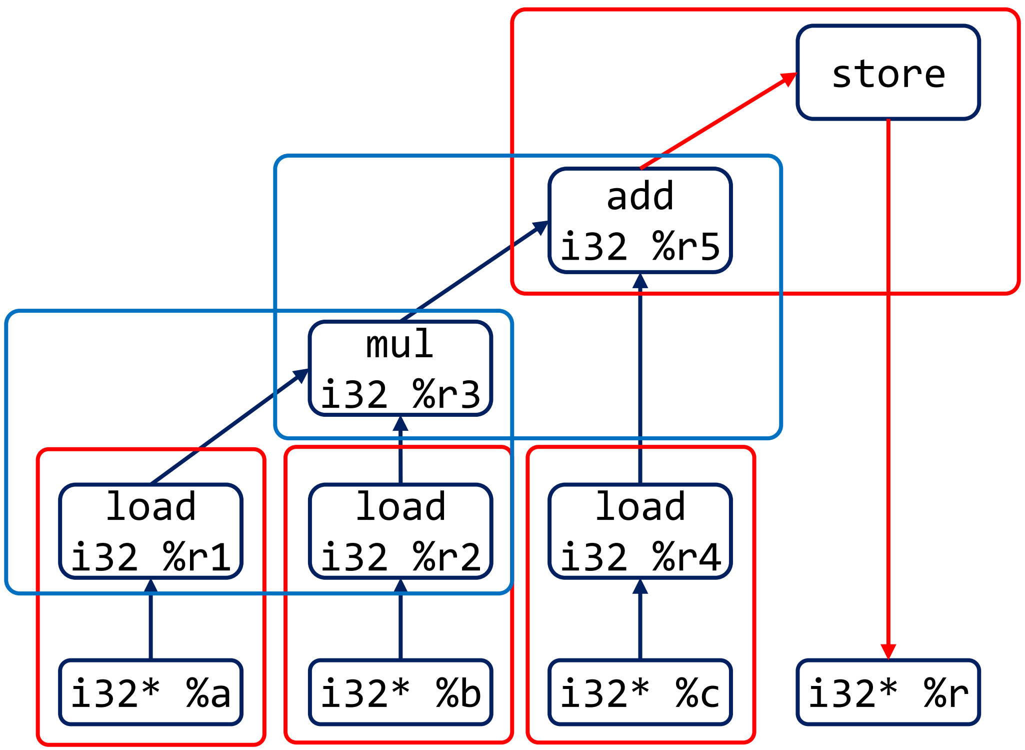
```
%r1 = load i32 %a;
%r2 = load i32 %b;
%r3 = mul i32 %r1, %r2;
%r4 = load i32 %c;
%r5 = add i32 %r3, %r4;
store i32 %r5, %r;
```



表达式DAG

指令选择问题=>铺树问题

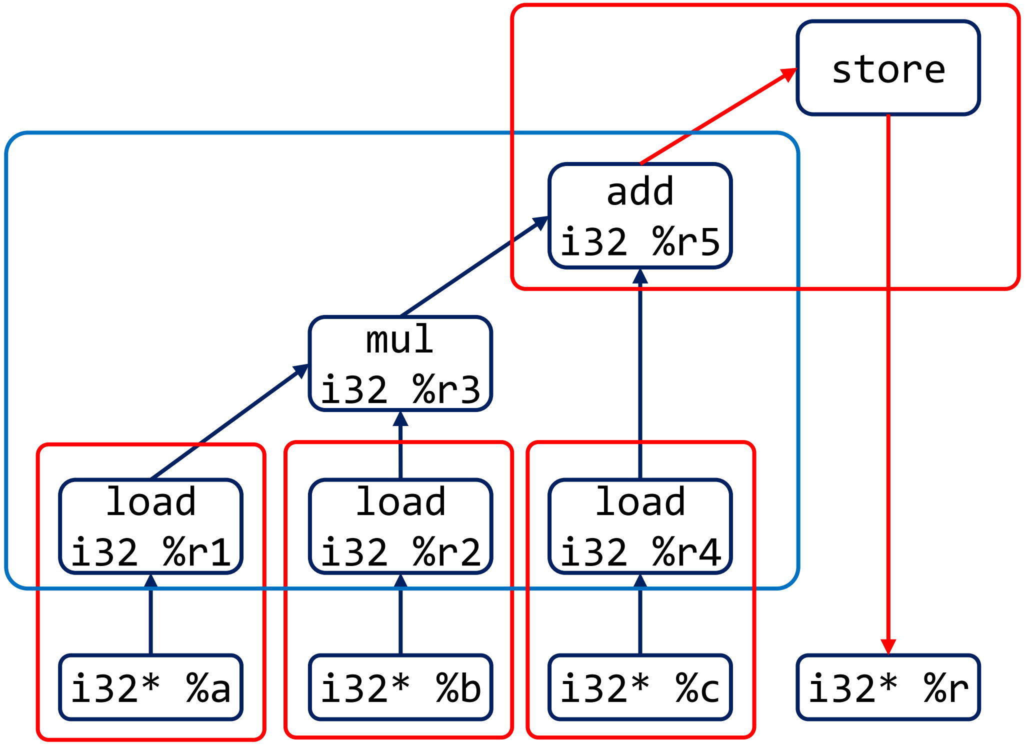
- 如何铺树使得最终的汇编代码：
  - 体积小（指令数少）
  - 运算快




```
ldr %r1, [sp, #a]
ldr %r2, [sp, #b]
ldr %r3, [sp, #c]
mul %r3, %r1, %r2
add %r5, %r3, %r4
store %r5, [sp, #r]
```

方式一

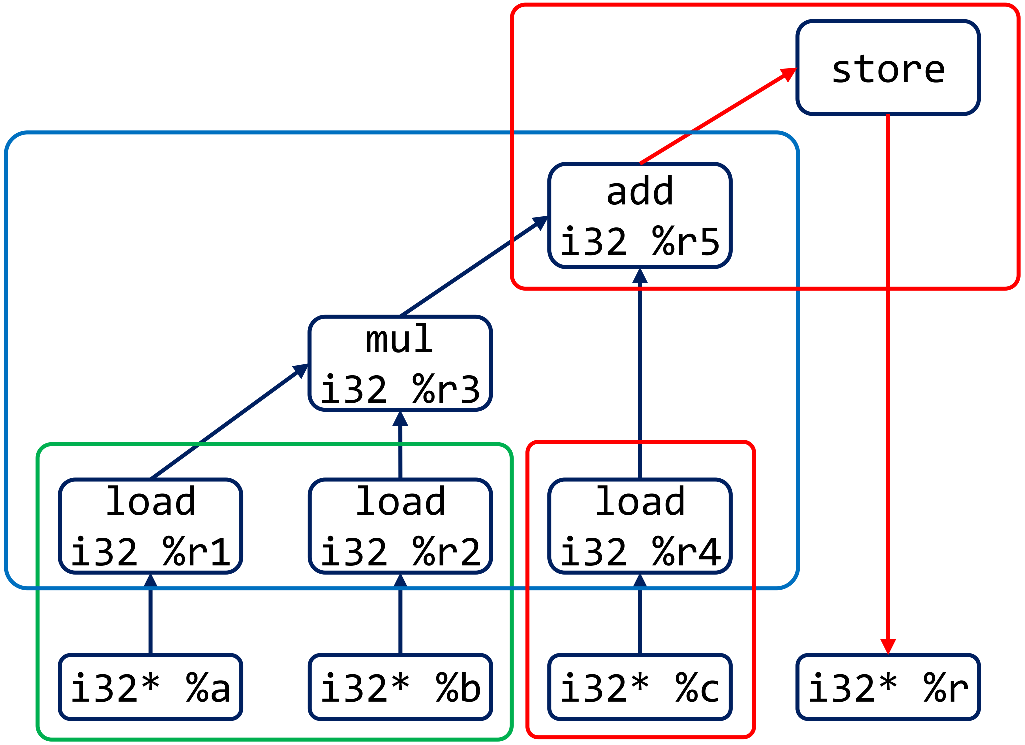
指令选择问题=>铺树问题




```
ldr %r1, [sp, #a]
ldr %r2, [sp, #b]
ldr %r4, [sp, #c]
madd %r5, %r1, %r2, %r4
store %r5, [sp, #r]
```

方式二

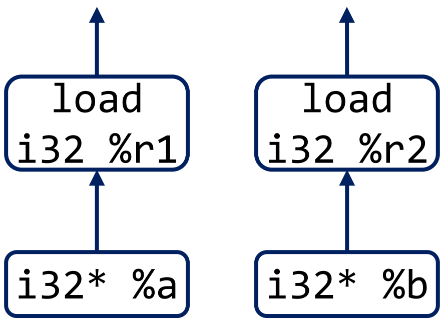
指令选择问题=>铺树问题




```
ldp %r1, %r2, [sp, #a]
ldr %r3, [sp, #c]
madd %r5, %r1, %r2, %r4
store %r5, [sp, #r]
```

方式三

load + load



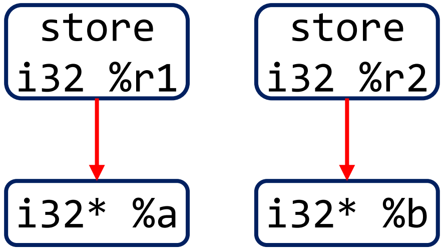

```
ldr %r1, [sp, #a]
ldr %r2, [sp, #b]
```

开销：8

```
ldp %r1, %r2, [sp, #a]
```

开销：4

store + store



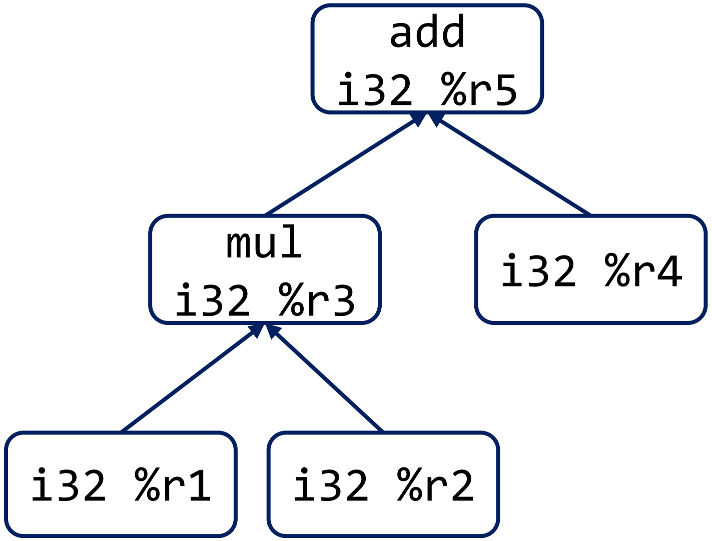

```
str %r1, [sp, #a]
str %r2, [sp, #b]
```

开销：2

```
stp %r1, %r2, [sp, #a]
```

开销：1

mul + add

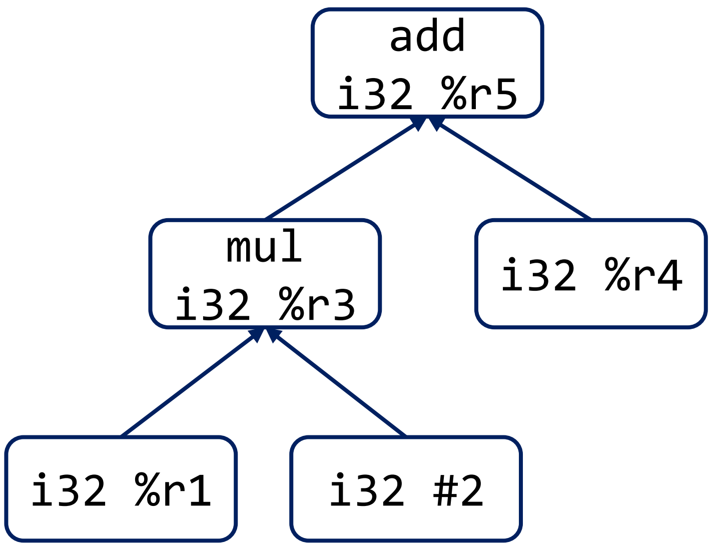



```
mul %r3, %r1, %r2
add %r5, %r3, %r4
```

开销：4

```
madd %r5, %r1, %r2, %r4
```

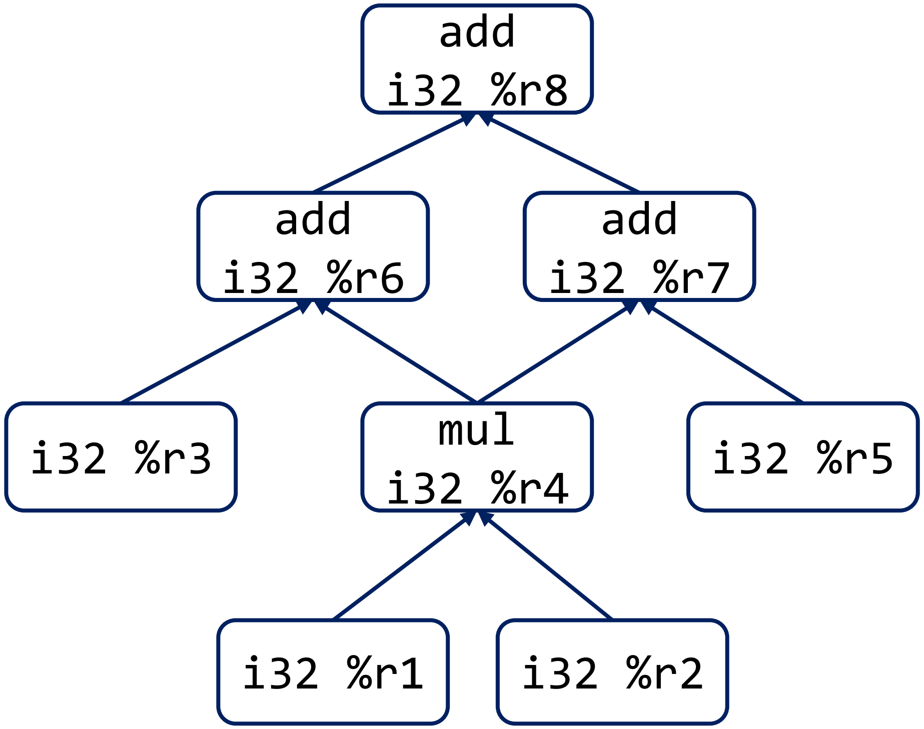
开销：3




```
mov %t1, #2
mul %r3, %r1, %t1
add %r5, %r3, %r4
```

开销：5

mul + add



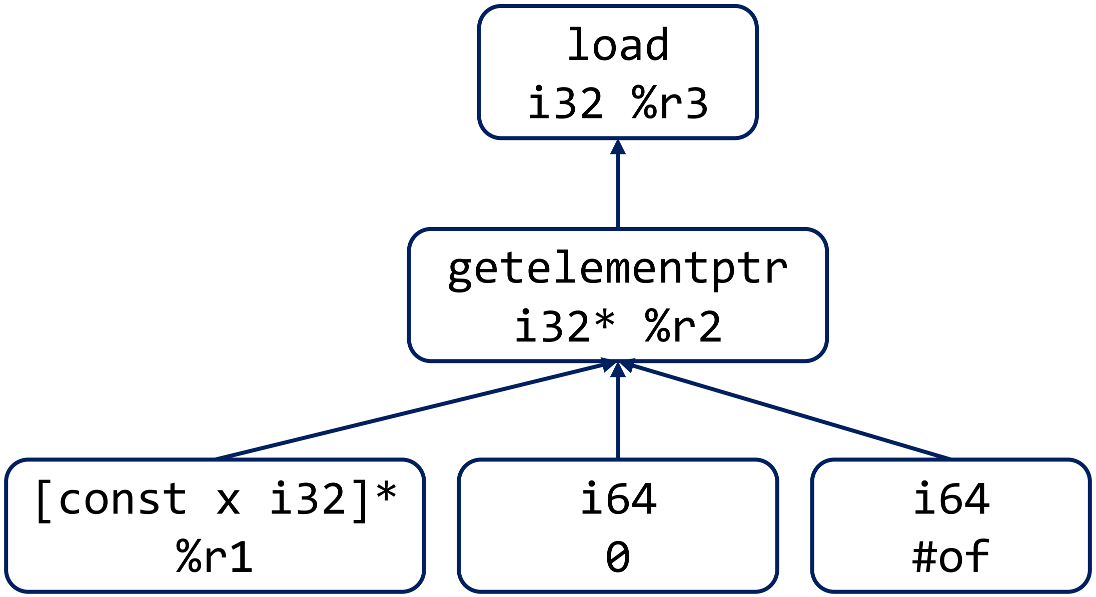

```
mul %r4, %r1, %r2
add %r6, %r3, %r4
add %r7, %r4, %r5
add %r8, %r6, %r7
```

开销：6

```
madd %r6, %r1, %r2, %r3
madd %r7, %r1, %r2, %r5
add %r8, %r6, %r7
```

开销：7

load + getelementptr：数组（常量索引）



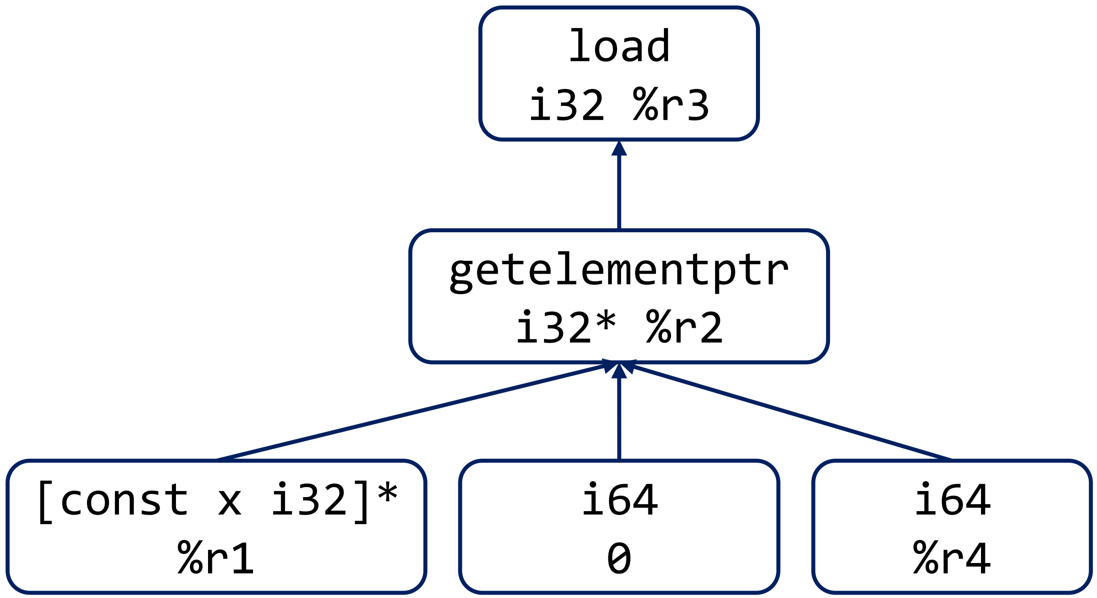

```
add %r2, %r1, #of*4
ldr %r3, [%r2]
```

开销：5

```
ldr %r3, [%r1, #of*4]
```

开销：4

load + getelementptr：数组（变量索引）




```
mov %t1, #4
mul %t2, %t1, %r4
add %r2, %r1, %t2
ldr %r3, [%r2]
```

开销：9

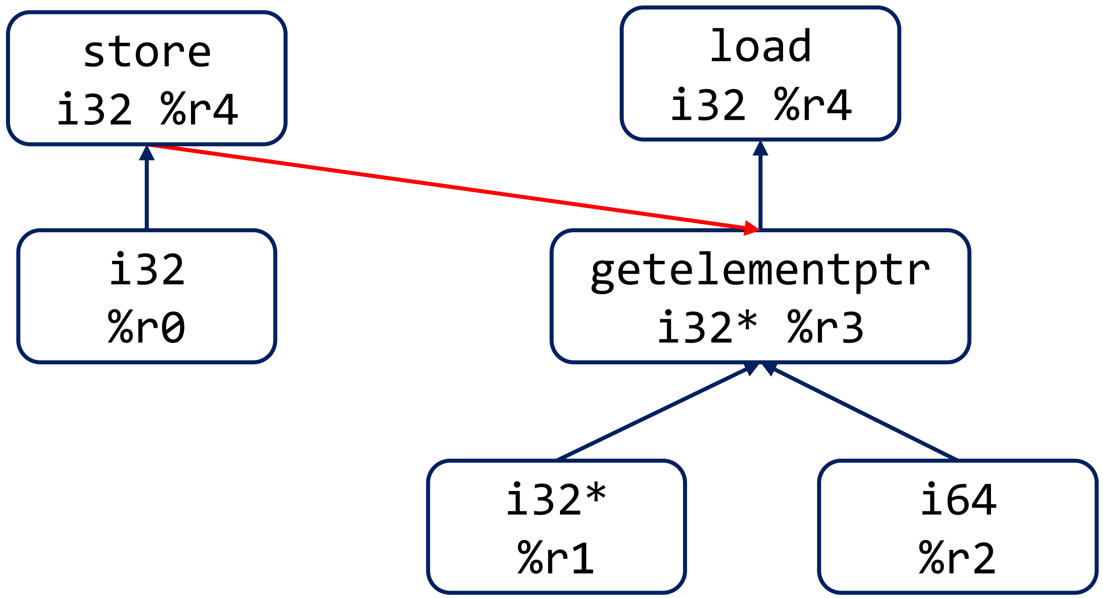
```
mov %t1, #4
mul %t2, %t1, %r4
ldr %r3, [%r1, %t2]
```

开销：8

```
ldr %r3, [%r1, %r4, lsl #2]
```

开销：4

load + getelementptr：数组（变量索引）




```
mov %t1, #4
mul %t2, %t1, %r4
add %r2, %r1, %t2
ldr %r3, [%r2]
str %r0, [%r2]
```

开销：10

```
mov %t1, #4
mul %t2, %t1, %r4
ldr %r3, [%r1, %t2]
str %r0, [%r2]
```

开销：9

```
ldr %r3, [%r1, %r4, lsl #2]
str %r0, [%r1, %r4, lsl #2]
```

开销：5

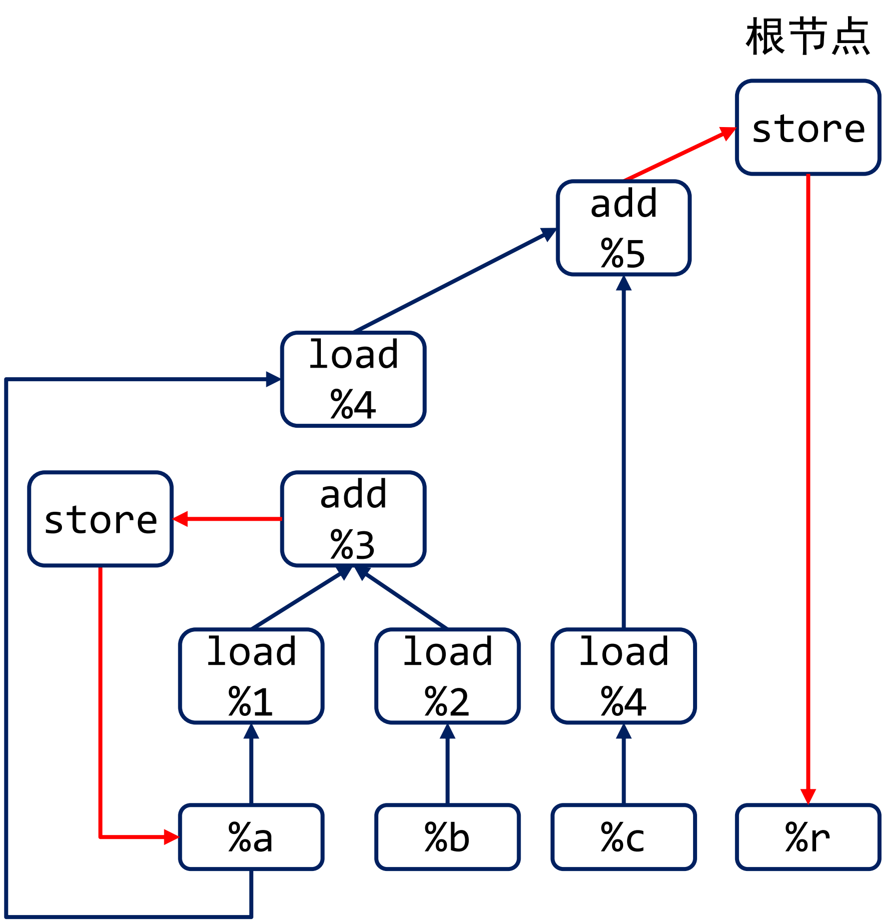
铺树问题解法

- 贪心算法：Maximal Munch
  - 从树根开始，每次选择覆盖节点最多、开销最低的规则
  - 逆序生成汇编指令
  - 局部最优
- 动态规划
  - 从树根开始，递归搜索每个节点的最优方案

同一代码块多次load的情况

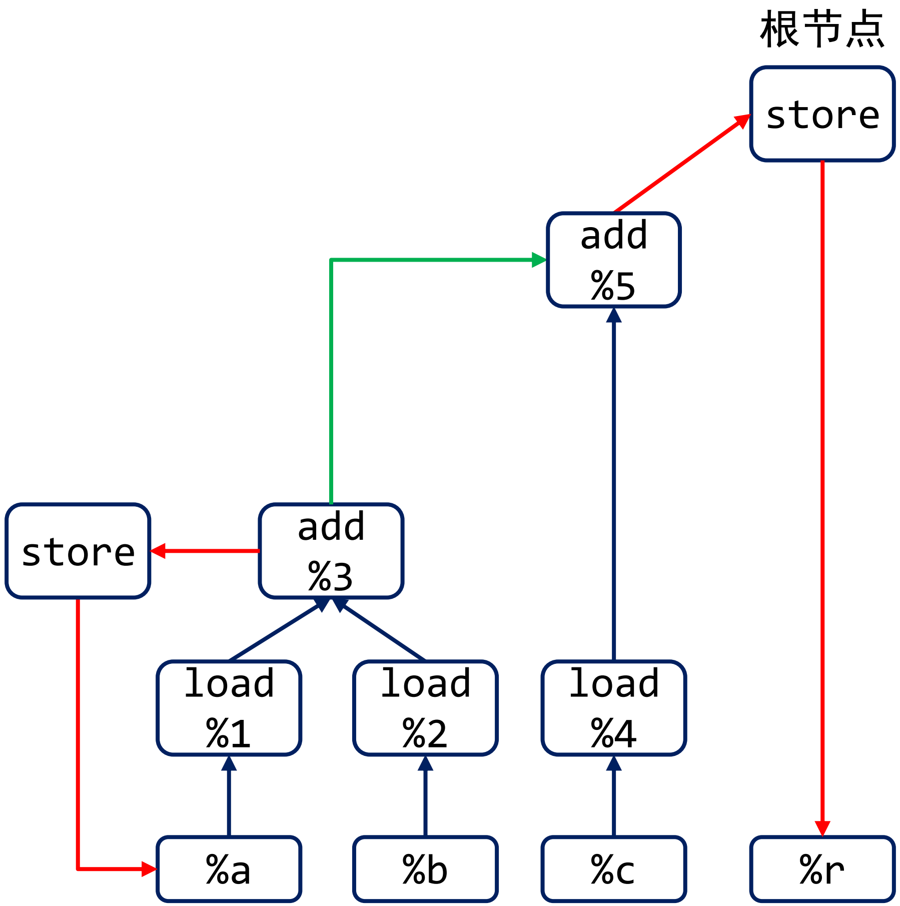
```
a = a + b;
r = a + c;
```

```
%1 = load i32 %a;
%2 = load i32 %b;
%3 = add i32 %1, %2;
store i32 %3, %a;
%4 = load i32 %a;
%5 = load i32 %c;
%6 = add i32 %4, %5;
store i32 %6, %r;
```



表达式DAG

同一代码块多次load的情况：SSA?

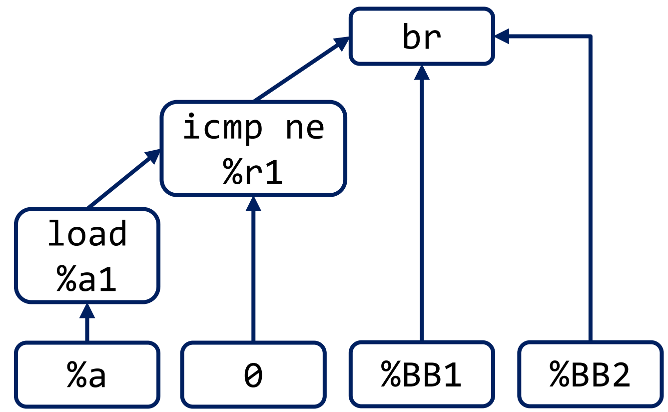


表达式DAG

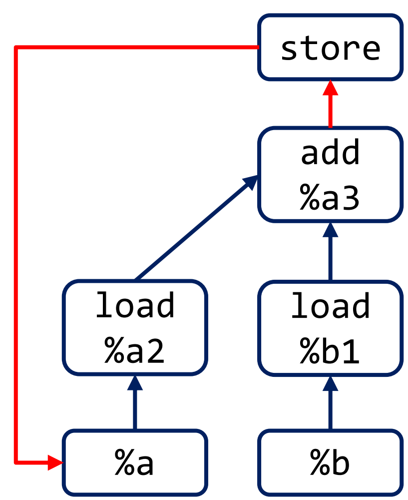
控制流

```
if(a==0)
    a = a + b;
let r = a + c;
```

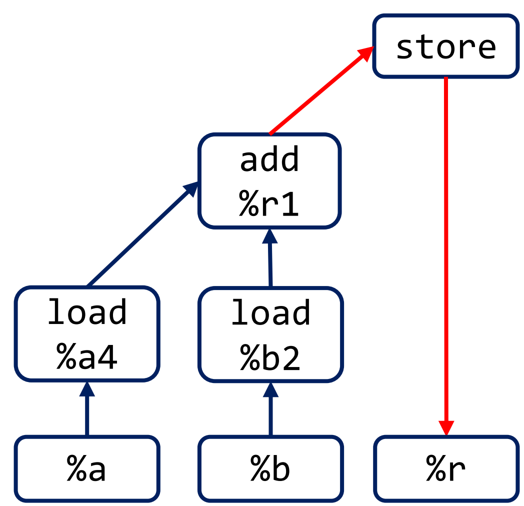
```
%BB1:
    %a1 = load i32, %a;
    %r1 = icmp eq i32 %a1, 0;
    br i1 %r1, %BB2, %BB3;
%BB2:
    %a2 = load i32, %a;
    %b1 = load i32, %b;
    %a3 = add i32 %a2, %b1;
    store i32 %a3, %a;
    br %BB2;
%BB3:
    %a4 = load i32, %a;
    %b2 = load i32, %b;
    %r1 = add i32 %a4, %b2;
    store i32 %r1, %r;
```



%BB1表达式DAG



%BB2表达式DAG

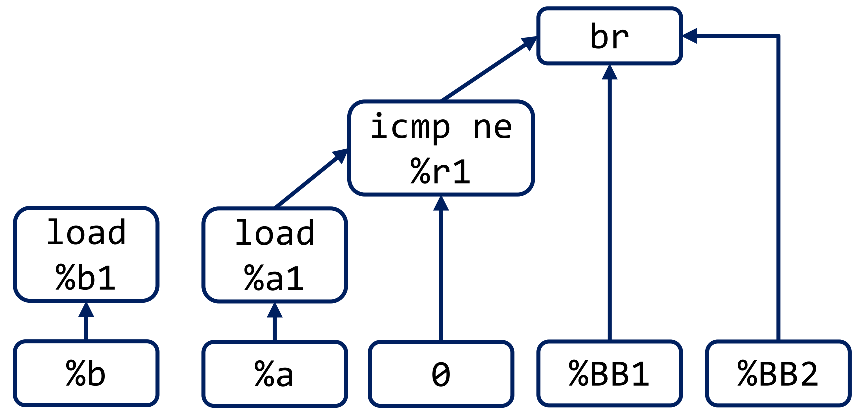


%BB3表达式DAG

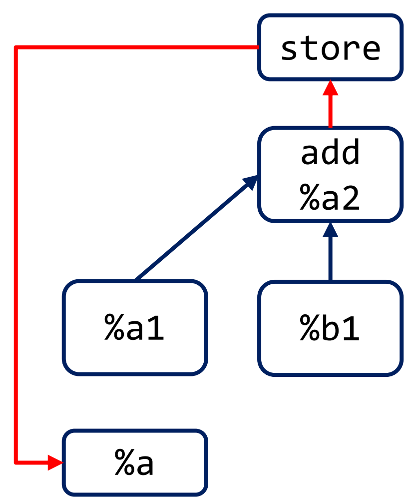
控制流：优化后

```
if(a==0)
     a = a + b;
let r = a + c;
```

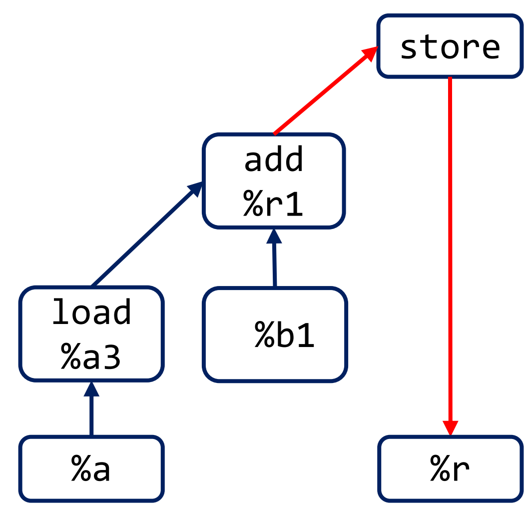
```
%BB1:
    %a1 = load i32, %a;
    %b1 = load i32, %b;
    %r1 = icmp eq i32 %8, 0;
    br i1 %r1, %BB2, %BB3;
%BB2:
    %a2 = add i32 %a1, %b1;
    store i32 %a2, %a;
    br %BB2;
%BB3:
    %a3 = load i32, %a;
    %r1 = add i32 %a3, %b1;
    store i32 %r1, %r;
```



%BB1表达式DAG



%BB2表达式DAG



%BB3表达式DAG

三、指令调度

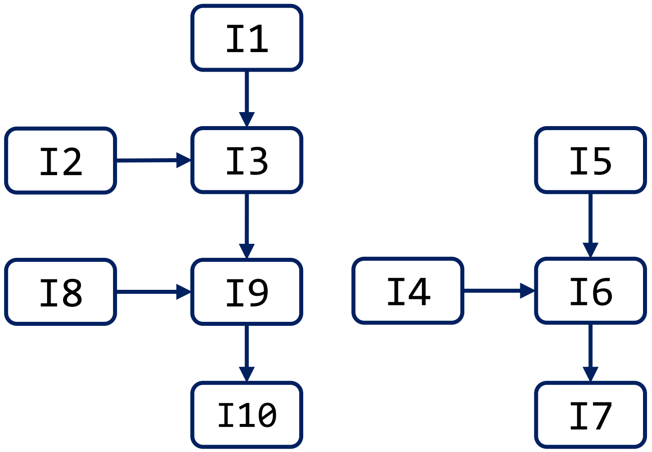
影响性能的因素

- 数据依赖关系（data dependency ）
  - 写-读依赖（true-dependency）
  - 读-写反依赖（anti-denpendency）
- 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

指令依赖关系

- 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 %r1, [%sp, #-12]
I2	ldr %r2, [%sp, #-16]
I3	add %r1, %r1, %r2
I4	ldr %r2, [%sp, #-20]
I5	ldr %r3, [%sp, #-24]
I6	sdiv %r3, %r2, %r3
I7	str %r3, [%sp, #-24]
I8	ldr %r2, [%sp, #-28]
I9	mul %r2, %r1, %r2
I10	str %r2, [%sp, #-28]

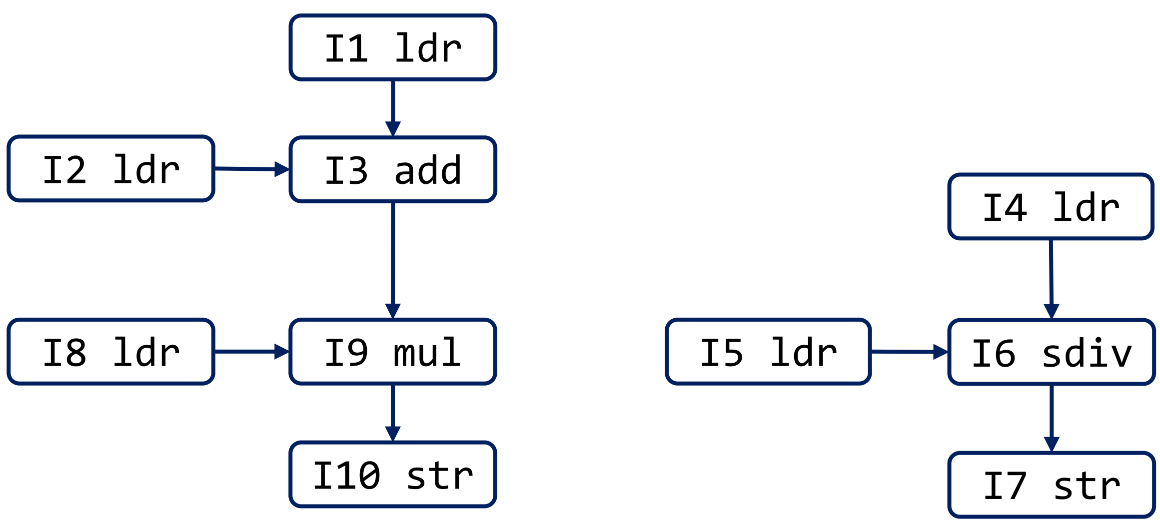


指令依赖关系

编译器的指令调度问题

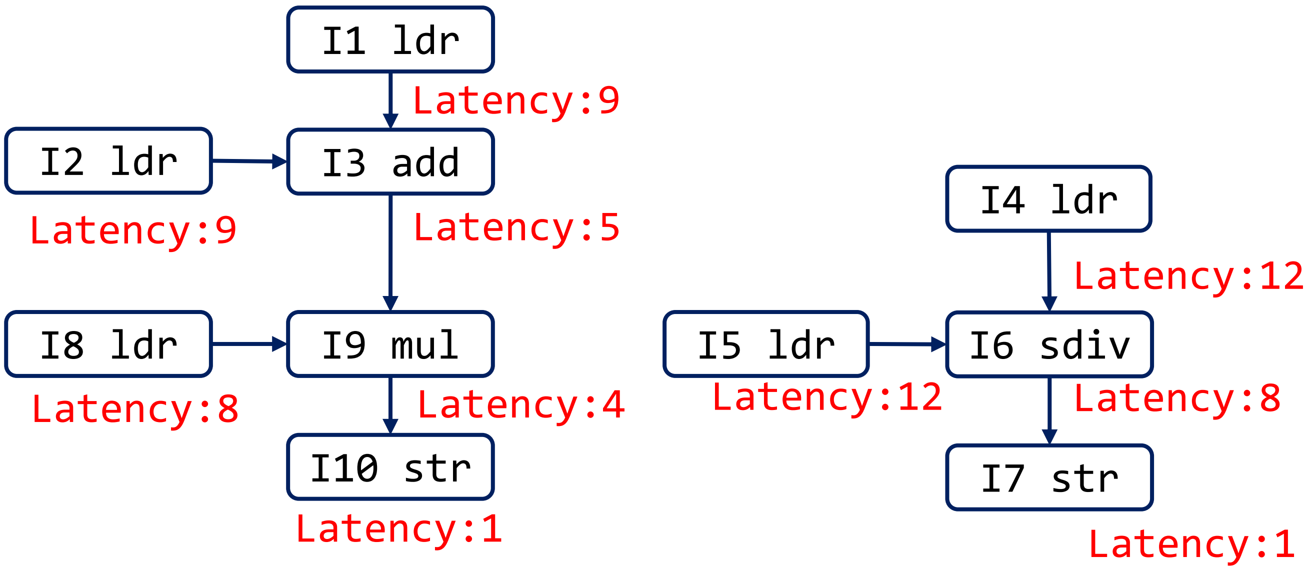
- 假设：
  - 每个cycle可以执行一条指令
  - 多条指令可以并行
  - 单条指令开销稳定
- 应如何确定最佳的指令执行序列？
  - 执行顺序应满足数据依赖关系

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

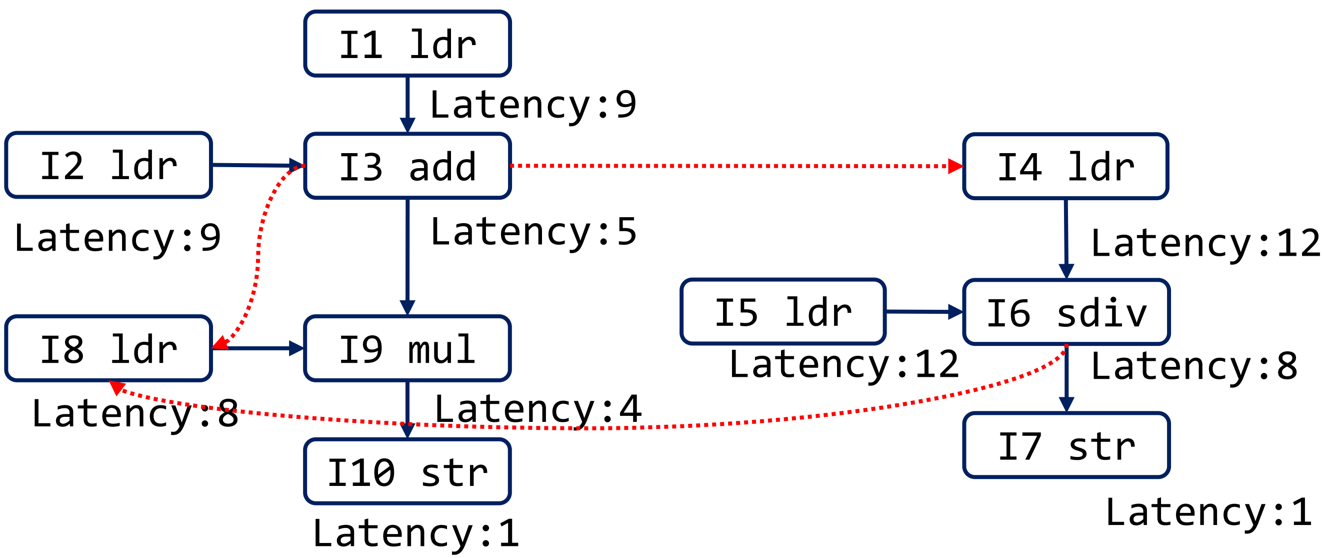
- 搜索需要尽早执行的关键指令
- 计算每条指令开始执行后，序列执行结束所需时间（latency）
  - 假设$i = v.next$，$L(v) = E_v + L(i)$
- 优先执行Latency大的指令
  - 根据latency从大到小对指令进行排序
  - I4=I5>I6>I1=I2>I8>I3>I9>I7=I10



读-写反依赖问题（非SSA）

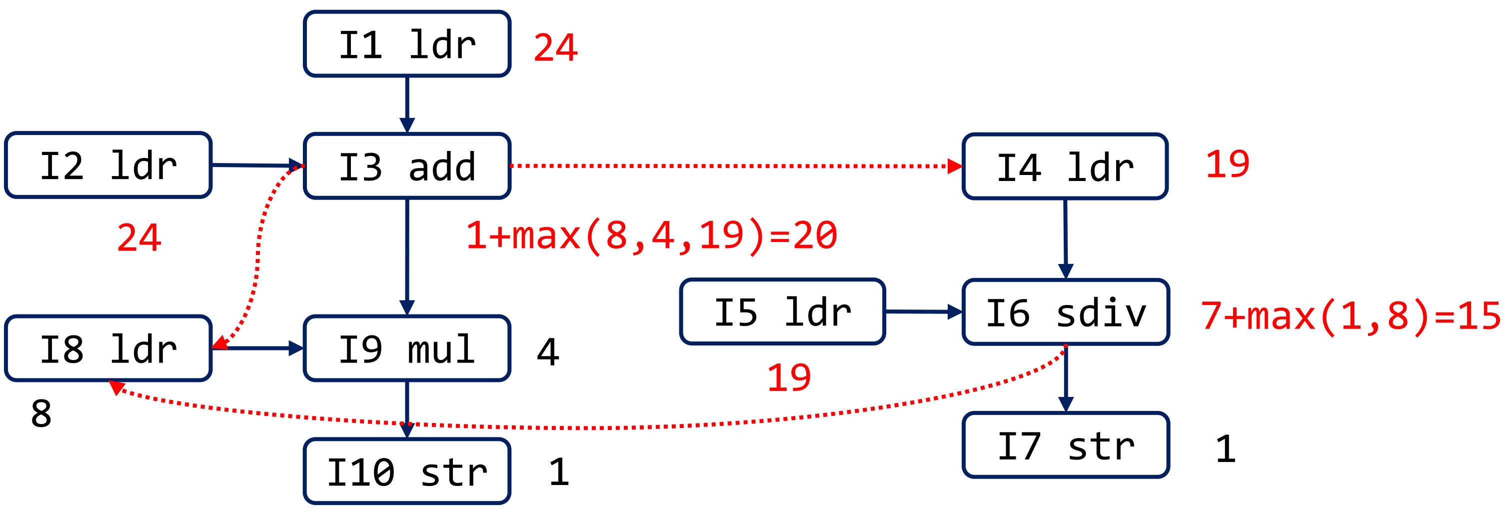
- I3执行完I4和I8才能执行
  - 否则会影响I3的计算结果
- I6执行完才能执行I8

I1	ldr %r1, [%sp, #-12]
I2	ldr %r2, [%sp, #-16]
I3	add %r1, %r1, %r2
I4	ldr %r2, [%sp, #-20]
I5	ldr %r3, [%sp, #-24]
I6	sdiv %r3, %r2, %r3
I7	str %r3, [%sp, #-24]
I8	ldr %r2, [%sp, #-28]
I9	mul %r2, %r1, %r2
I10	str %r2, [%sp, #-28]



更新Latency和执行序列

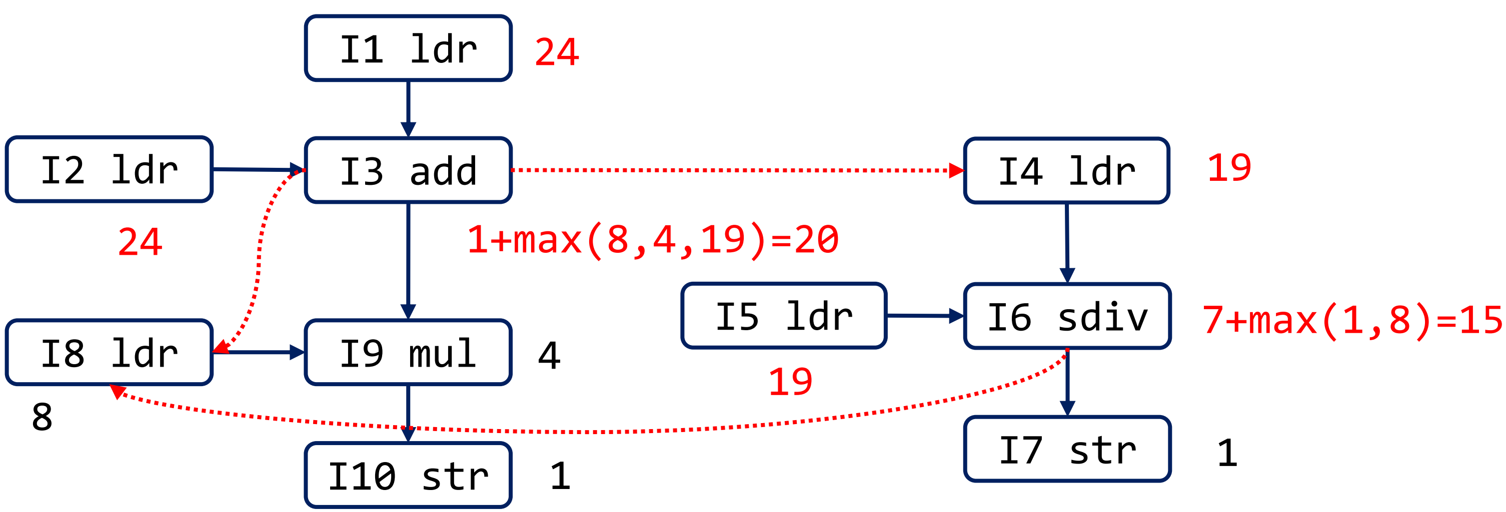
- $\forall i \in v.next, L(v) = E_v + Max(L(i))$
- 新序列：I1=I2>I3>I4=I5>I6>I8>I9>I7=I10



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 %r1, [%sp, #-12]
2	5	I2	ldr %r2, [%sp, #-16]
6	6	I3	add %r1, %r1, %r2
7	10	I4	ldr %r2, [%sp, #-20]
8	11	I5	ldr %r3, [%sp, #-24]
12	18	I6	sdiv %r3, %r2, %r3
19	22	I8	ldr %r2, [%sp, #-28]
23	25	I9	mul %r2, %r1, %r2
24	24	I7	str %r3, [%sp, #-24]
26	26	I10	str %r2, [%sp, #-28]

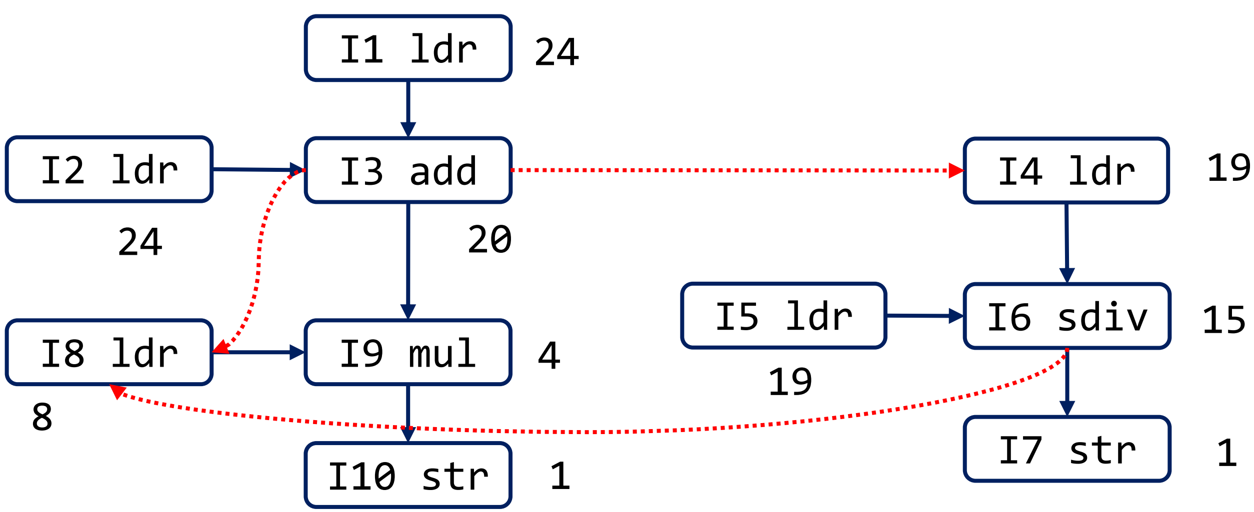


消除反依赖：重命名（CPU也支持）

I1	ldr %r1, [%sp, #-12]
I2	ldr %r2, [%sp, #-16]
I3	add %r1, %r1, %r2
I4	ldr %r2, [%sp, #-20]
I5	ldr %r3, [%sp, #-24]
I6	sdiv %r3, %r2, %r3
I7	str %r3, [%sp, #-24]
I8	ldr %r2, [%sp, #-28w]
I9	mul %r2, %r1, %r2
I10	str %r2, [%sp, #-28w]

⇨

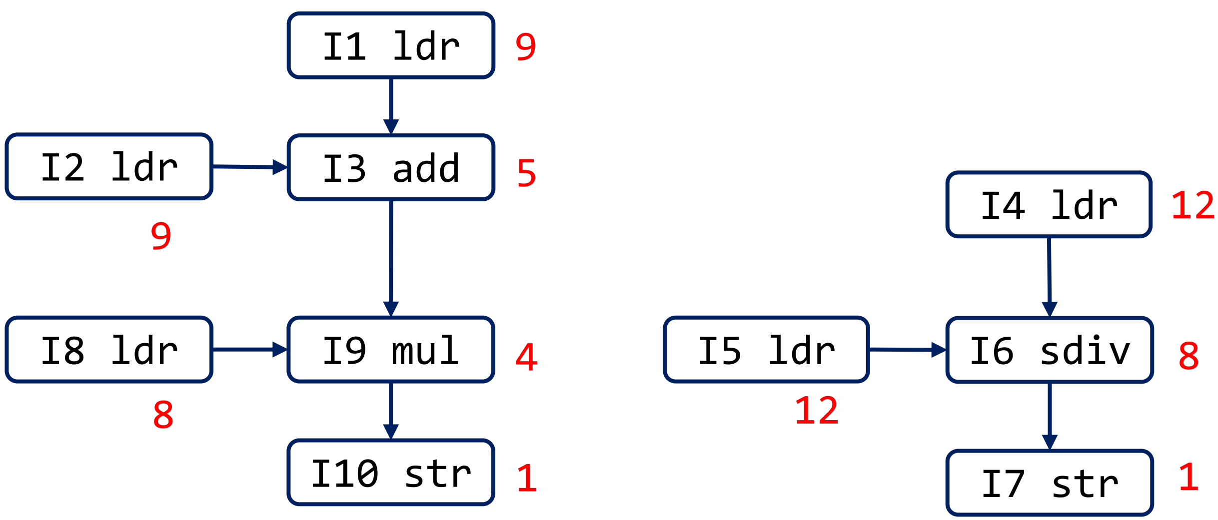
I1	ldr %r1, [%sp, #-12]
I2	ldr %r2, [%sp, #-16]
I3	add %r1, %r1, %r2
I4	ldr %r4, [%sp, #-20]
I5	ldr %r3, [%sp, #-24]
I6	sdiv %r3, %r4, %r3
I7	str [%sp, #-24], %r3
I8	ldr %r5, [%sp, #-28w]
I9	mul %r5, %r1, %r5
I10	str %r5, [%sp, #-28w]



调度方案开销

- I4=I5>I1=I2>I6=I8>I3>I9>I7=I10
  - 开销：14

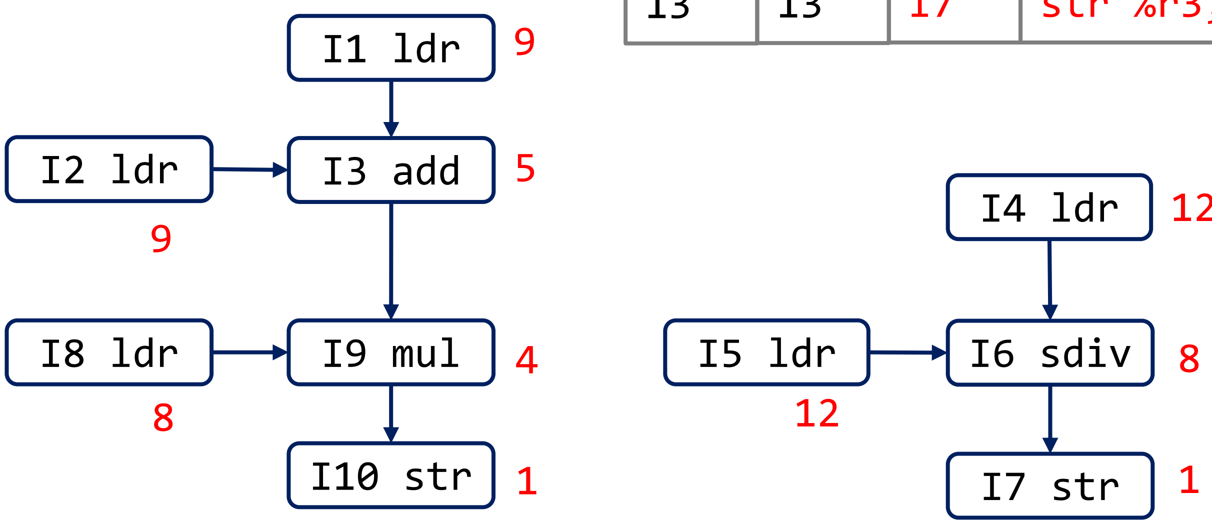
开始	结束	指令	
1	4	I4	ldr %r4, [%sp, #-20]
2	5	I5	ldr %r3, [%sp, #-24]
3	6	I1	ldr %r1, [%sp, #-12]
4	7	I2	ldr %r2, [%sp, #-16]
6	12	I6	sdiv %r3, %r4, %r3
7	10	I8	ldr %r5, [%sp, #-28w]
8	8	I3	add %r1, %r1, %r2
11	13	I9	mul %r5, %r1, %r5
13	13	I7	str %r3, [%sp, #-24]
14	14	I10	str %r5, [%sp, #-28w]



进一步优化（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 %r4, [%sp, #-20]
2	5	I5	ldr %r3, [%sp, #-24]
3	6	I1	ldr %r1, [%sp, #-12]
4	7	I2	ldr %r2, [%sp, #-16]
5	8	I8	ldr %r5, [%sp, #-28w]
6	12	I6	sdiv %r3, %r4, %r3
8	8	I3	add %r1, %r1, %r2
9	11	I9	mul %r5, %r1, %r5
12	12	I10	str %r5, [%sp, #-28w]
13	13	I7	str %r3, [%sp, #-24]



表调度算法

- 假设：
  - 线性代码
  - 无反依赖
- 动态两张表：
  - Ready表：已满足依赖的指令
  - Active表：正在执行的指令
- 算法：
  - 每次选择Ready表中的一条指令执行
  - 如果有指令执行完成，考虑将其next指令加入Ready

```
Clock = 1
Ready = {指令依赖图的所有叶子节点}
Active = {}
While (Ready ∪ Active ≠ Ø){
    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 ≠ Ø){
        Ready.remove(any I);
        Start(I) = Clock;
        Active.add(I);
    }
    Clock = Clock + 1;
}
```