# 指令选择与指令调度

徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

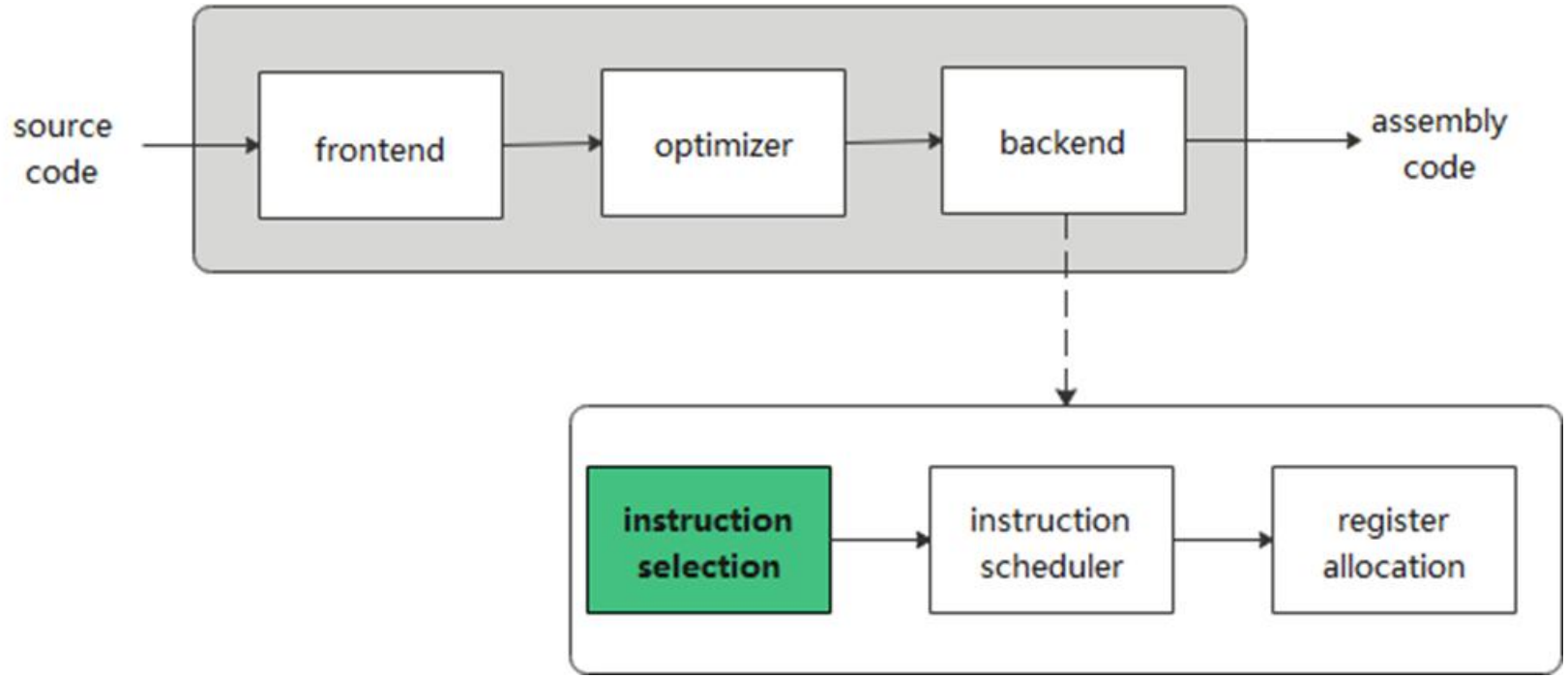
先进技术研究院、计算机科学与技术学院

2025年11月27日

# 指令选择

## ❑指令选择（instruction selection）

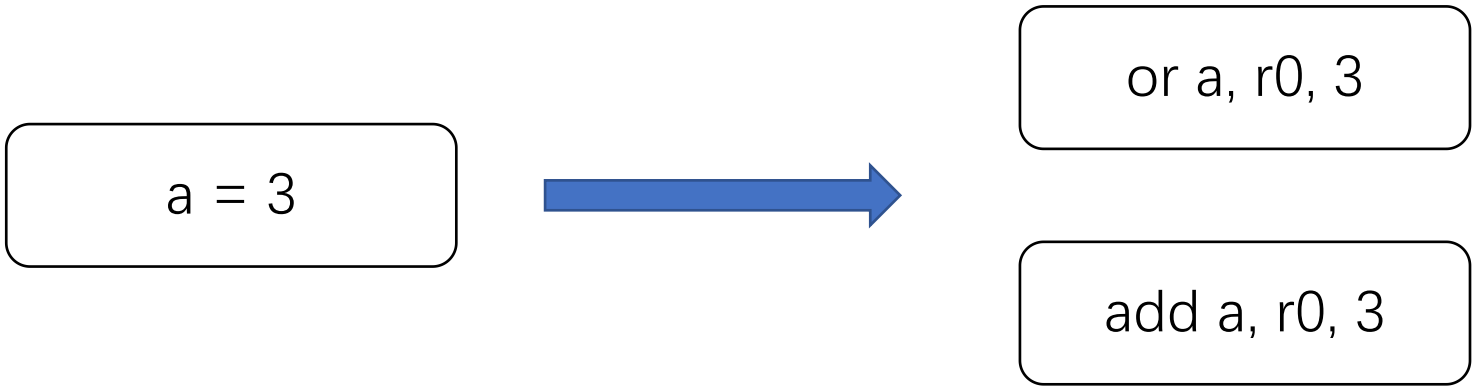
- 作为一个独立概念于二十世纪六十年代被提出
- 编译器将平台无关的中间语言转换为平台相关的机器指令的过程。其目标是尽量高效的选择最优的可用的机器指令。



# 指令选择

## ❑指令选择（instruction selection）

- 将编译器的IR映射到目标ISA
- 是一个模式匹配问题
- 其复杂性源自常见的ISA为操作提供的大量备选实现方案。



# 指令选择

## ❑转换规则

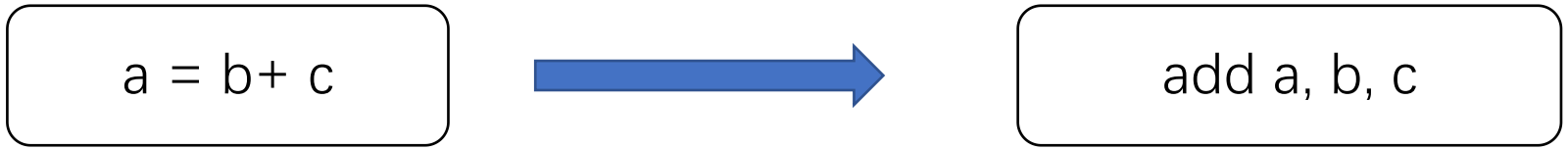
- ■ 可能很简单，也可能很复杂


简单方式：一对一示意图

# 指令选择

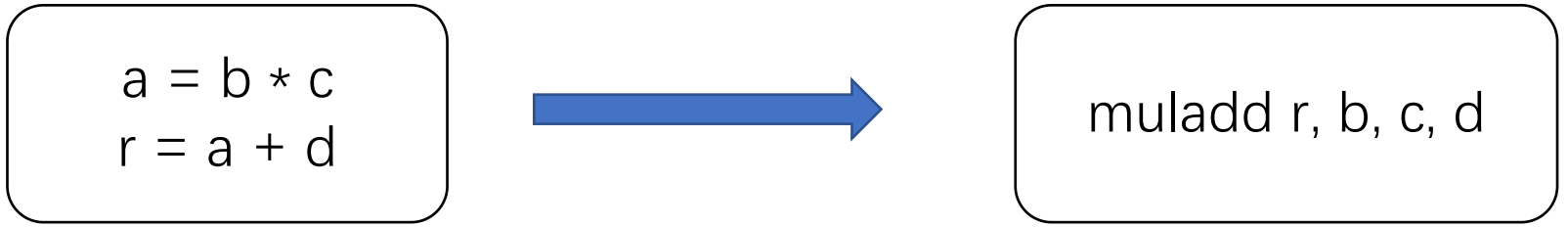
## ❑转换规则

- 可能很简单，也可能很复杂




简单方式：一对一示意图

- 可能很简单，也可能很复杂




复杂方式：多条高级语言操作生成一条目标指令
也存在一条高级语言操作对应多条指令等情况

# 指令选择算法

## 指令选择算法

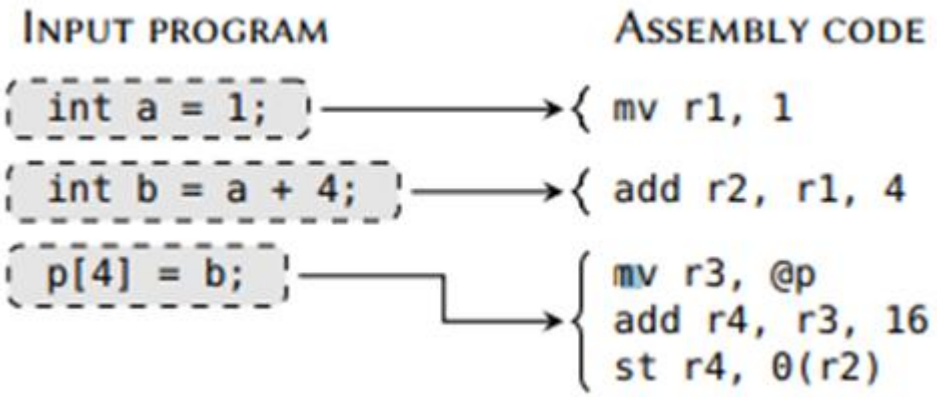
- 宏展开
- 树覆盖
- DAG匹配

# 指令选择算法

## ❑指令选择算法

### ■ 宏展开

- 也被叫做模板匹配(Template Matching)，即对于每一条IR或AST结构，都有一条或多条机器指令与其相对应，编译器直接使用预制好的指令或指令模板替换对应的每一条输入IR即可。




优点：简单粗暴，实现简单，易于理解

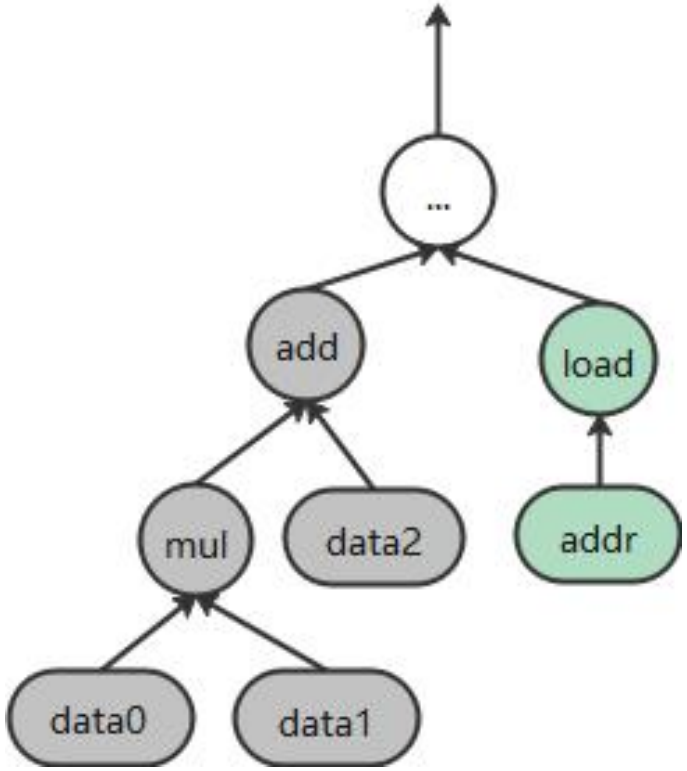
缺点：只支持1:1或1:N的情况，无法处理N:1或N:M的场景即多条IR或AST对应一条或者多条机器指令的场景，致使指令结果不优

# 指令选择算法

## ❑指令选择算法

### ■ 树覆盖

- 解决了宏展开的不支持N:1或N:M的场景
- 将IR或AST的前端语法与后端的机器指令都转换为树结构。这样就把指令选择问题转换为机器指令树覆盖全IR语法树的问题




指令选择可以将其想象为铺地板的过程

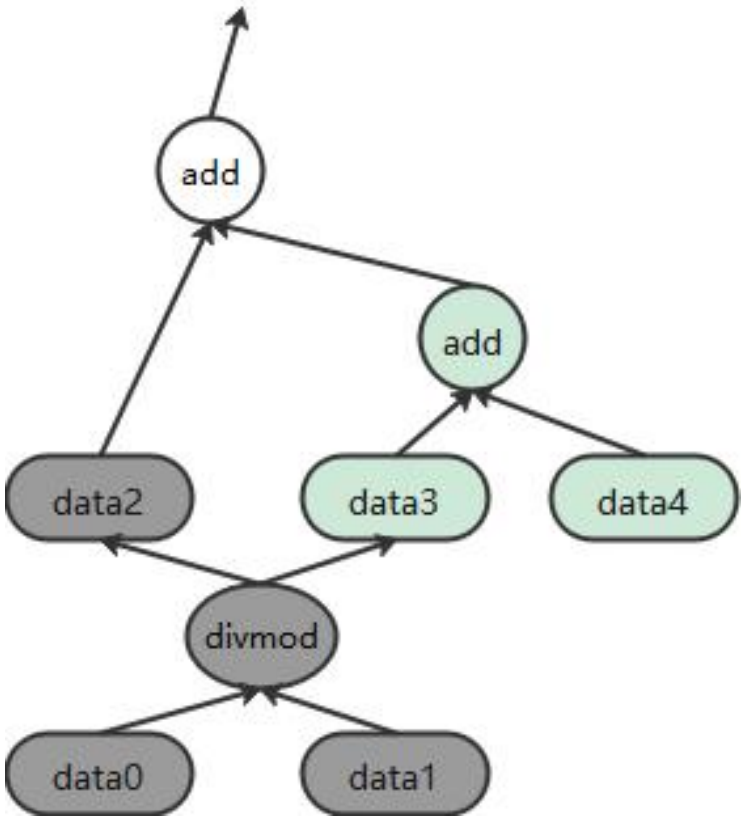
缺点：无法表示多个输出边的情况

# 指令选择算法

## ❑指令选择算法

### ■ DAG匹配

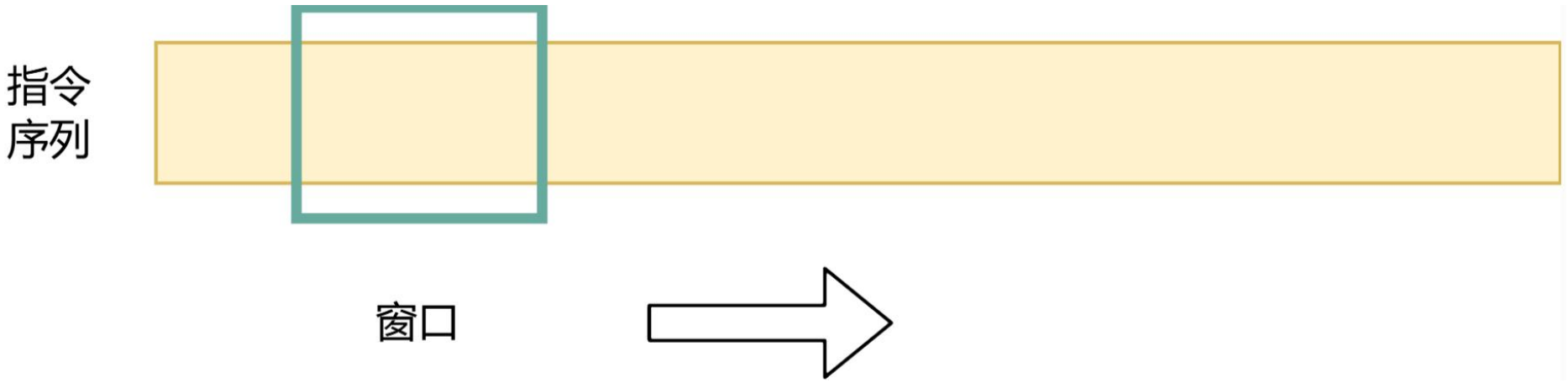
- DAG(directed acyclic graph)即为有向无环图
- 把树覆盖中的数据结构转换为DAG，主要思路与树覆盖类似采用pattern覆盖
- DAG覆盖可以解决树无法表示多个输出边的情况。



# 窥孔优化

## ❑窥孔优化（Peephole Optimization）

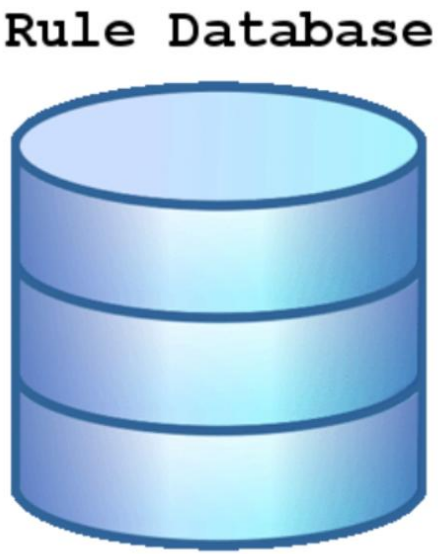
- 编译器中的一个技术，用于优化生成的中间代码或目标代码
- 通过查看代码的小部分（或称为“窥孔”）来识别并提供更高效的代码替代方案



# 窥孔优化

## ❑窥孔优化（Peephole Optimization）

- 窥孔优化类似于滑动窗口，针对一小组编译器生成的指令序列进行优化
- 不需要过多关注上下文、控制流等




比如对指令进行替换，提前定好了匹配的规则来进行操作

```
    %0 = mul i32 %1, 8
=> %0 = lsh i32 %1, 3
```

乘 8 的操作指令，可以优化替换为左移 3 的操作指令
提升了指令执行的性能。

# 窥孔优化

## ❑窥孔优化优化规则

### ■ 冗余消除(Redundant load and store elimination)

```
****Initial code:****
y = x + 5;
i = y;
z = i;
w = z * 3;
```

```
****Optimized code:****
y = x + 5;
w = y * 3; //* there is no i now
```

消除还应该判断 *def-use* 链路的相关指令，避免误删

# 窥孔优化

## ❑窥孔优化优化规则

### ■ 常量折叠(Constant folding)

```
****Initial code:****
x = 2 * 3;
```

```
****Optimized code:****
x = 6;
```

# 窥孔优化

## ❑窥孔优化优化规则

### ■ 强度消减(Strength Reduction)

****Initial code:****

```
y = x * 2;
```

****Optimized code:****

```
y = x + x;    or    y = x << 1;
```

# 窥孔优化

## ❑窥孔优化优化规则

### ■ 删除无用代码

```
****Initial code:****
int foo(int x) {
    if (x == 0) {
        return 0;
    } else {
        return 2 * x;
    }
}
```

```
****Optimized code:****
int foo(int x) {
    return 2 * x;
}
```

# 窥孔优化

## 窥孔优化步骤

- 定义窥孔大小
- 模式识别
- 模式替换
- 维护窥孔表
- 反复应用
- 验证优化的正确性

# 窥孔优化

## 定义窥孔大小

- 确定一个“窥孔”的大小。要考察的连续指令的数量。一个窥孔可以是一个、两个、三个或更多的连续指令
- 大小选择的关键是权衡：更大的窥孔可以识别更多的优化机会，但同时也增加了搜索和匹配的复杂性

## 模式识别

- 滑动窗口遍历整个代码片段，以检查预定义的低效或冗余代码模式
- 这些模式可能包括无用的指令、冗余的加载和存储操作、可以简化的算术操作等

# 窥孔优化

## ❑模式替换

- 一旦识别到预定义模式，就用更高效的代码替换它
- 例如，连续的加载和存储操作可以被单一的复制指令替换，或者连续的算术操作可以被一个等效但更简单的操作替换

## ❑维护窥孔表

- 编译器会维护一个窥孔表，列出可以识别和替换的模式，以及它们的替代代码
- 可以基于经验进行构建，也可以基于具体的体系结构或平台进行调整

# 窥孔优化

## 反复应用

- 窥孔优化可能会为进一步的窥孔优化创造新的机会
- 一次优化的结果可能会产生新的连续指令，这些指令再次适用于窥孔优化
- 因此，窥孔优化通常会反复应用，直到没有进一步的优化机会为止

## 验证优化的正确性

- 优化后的代码应该产生与原始代码相同的结果
- 通常需要进行额外的验证步骤，确保替换是正确的并没有引入任何新的错误
- LLVM 可以使用 Alive 工具来进行正确性的验证

# 窥孔优化

## ❑示例

```
****Initial code:****
LOAD R1, a          ; R1 = a
LOAD R2, b          ; R2 = b
MUL R3, R1, 1       ; R3 = R1 * 1
ADD R4, R2, 0       ; R4 = R2 + 0
STORE R3, c         ; c = R3
LOAD R5, c          ; R5 = c
ADD R5, R5, R4      ; R5 = R5 + R4
STORE R5, d         ; d = R5
```

改进点：

1、乘以1或加0是没有必要的。

2、STORE R3, c 之后的 LOAD R5, c 是冗余的。

# 窥孔优化

## ❑示例

```
****Initial code:****
LOAD R1, a        ; R1 = a
LOAD R2, b        ; R2 = b
MUL R3, R1, 1     ; R3 = R1 * 1
ADD R4, R2, 0     ; R4 = R2 + 0
STORE R3, c       ; c = R3
LOAD R5, c        ; R5 = c
ADD R5, R5, R4    ; R5 = R5 + R4
STORE R5, d       ; d = R5
```

改进点：

1、乘以1或加0是没有必要的

2、STORE R3, c 之后的 LOAD R5, c 是冗余的

应用窥孔优化：

1、去掉乘以1和加0的操作

2、消除冗余的存储和加载指令

# 窥孔优化

## ❑示例

```
****Initial code:****
LOAD R1, a        ; R1 = a
LOAD R2, b        ; R2 = b
MUL R3, R1, 1     ; R3 = R1 * 1
ADD R4, R2, 0     ; R4 = R2 + 0
STORE R3, c       ; c = R3
LOAD R5, c        ; R5 = c
ADD R5, R5, R4    ; R5 = R5 + R4
STORE R5, d       ; d = R5
```

改进点：

1、乘以1或加0是没有必要的

2、STORE R3, c 之后的 LOAD R5, c 是冗余的

应用窥孔优化：

1、去掉乘以1和加0的操作

2、消除冗余的存储和加载指令

```
****Optimized code:****
LOAD R1, a        ; R1 = a
LOAD R2, b        ; R2 = b
ADD R1, R1, R2    ; R1 = R1 + R2
STORE R1, d       ; d = R1
```

# 指令调度

## 指令调度

- 对程序块或过程中的操作进行排序以有效利用处理器资源的任务
- 目的：通过重排指令，提高指令级并行性，使得程序在拥有指令流水线的CPU上更高效的运行
- 必要前提：CPU硬件支持指令并行，否则，指令调度是毫无意义的

# 指令调度

## 根据指令调度发生的阶段划分

### 静态调度

- 发生在程序编译时期
- 由编译器完成，在生成可执行文件之前通过指令调度相关优化，完成指令重排

### 动态调度

- 发生在程序运行时期
- 需要提供相应的硬件支持，比如乱序执行，此时指令的发射顺序和执行顺序可能是不一致，但CPU会保证程序执行的正确性

# 现代计算机的指令并行方案

- 现代计算机的三种并行模式
  - 流水线
  - 超标量
  - 多核

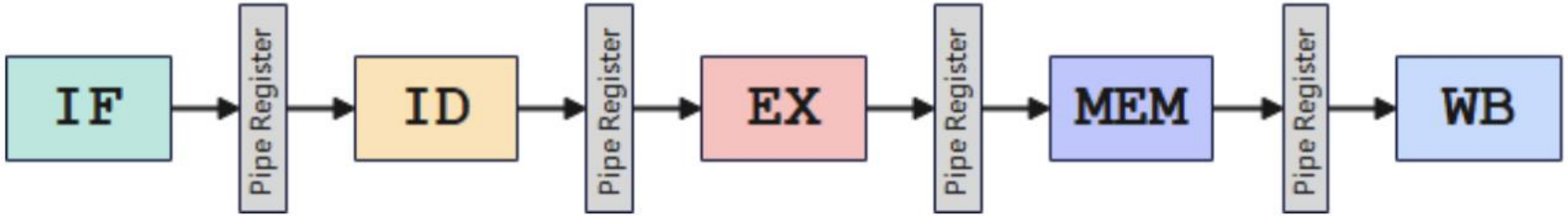
# 现代计算机的指令并行方案

## ❑流水线

- 将指令执行过程分成多个阶段
- 每个阶段使用不同的硬件资源，从而使得多条指令的执行时间可以重叠

## ❑经典五段式流水线

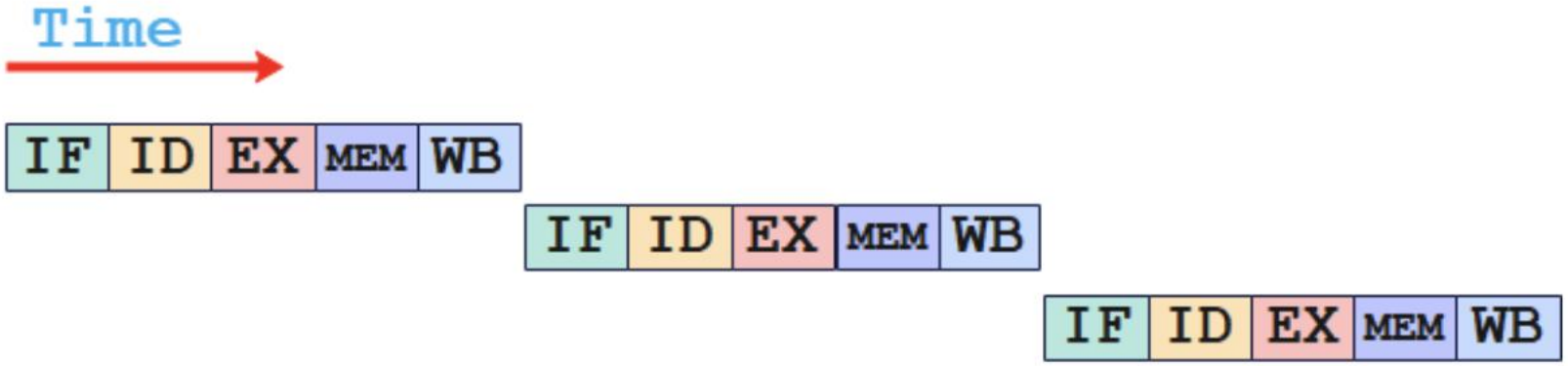
- 在五段式流水线中将一条指令的执行过程分成了5个阶段
- IF（取指）、ID（译码）、EX（执行）、MEM（访存）、WB（回写）



# 现代计算机的指令并行方案

## ❑流水线

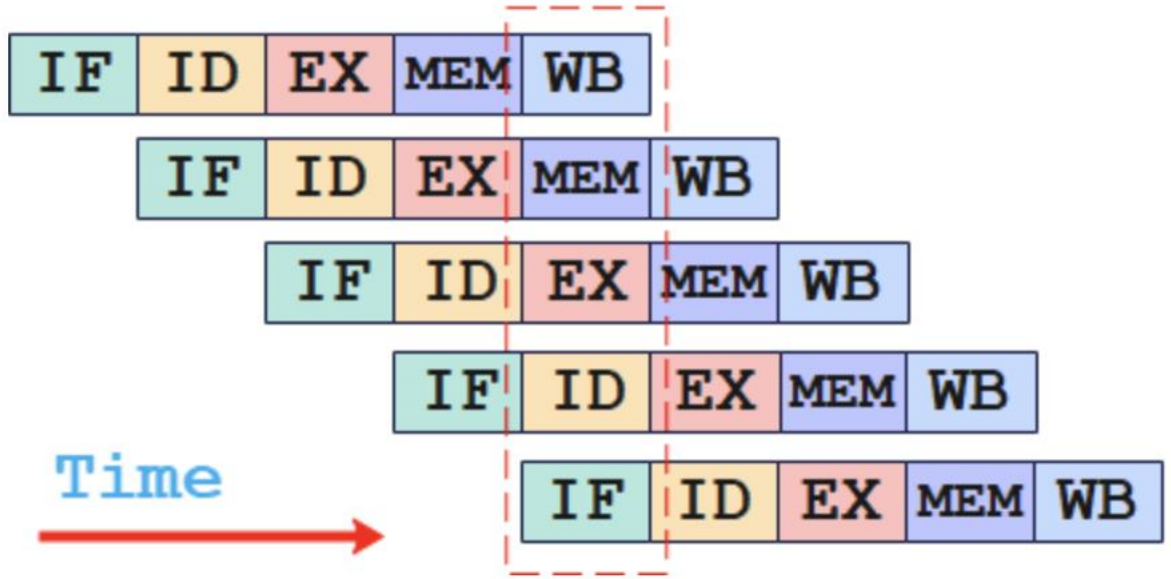
- 使能流水线之前
- IF（取指）、ID（译码）、EX（执行）、MEM（访存）、WB（回写）



# 现代计算机的指令并行方案

## ❑流水线

- 使能流水线后
- IF（取指）、ID（译码）、EX（执行）、MEM（访存）、WB（回写）

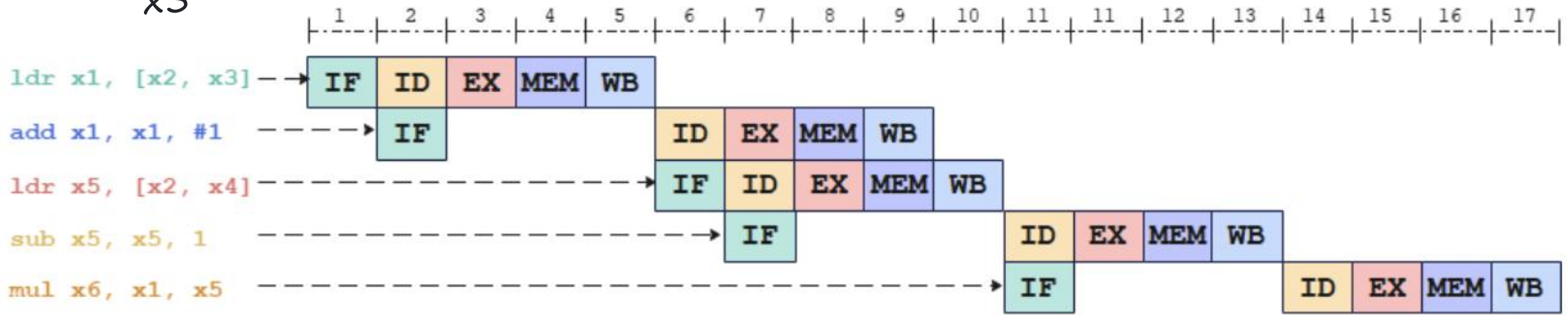


# 现代计算机的指令并行方案

- 流水线示例

```
ldr   x1, [x2, x3]
add   x1, x1, #1
ldr   x5, [x2, x4]
sub   x5, x5, #1
mul   x6, x1, x5
```

指令调度之前，耗时17个cycle



# 现代计算机的指令并行方案

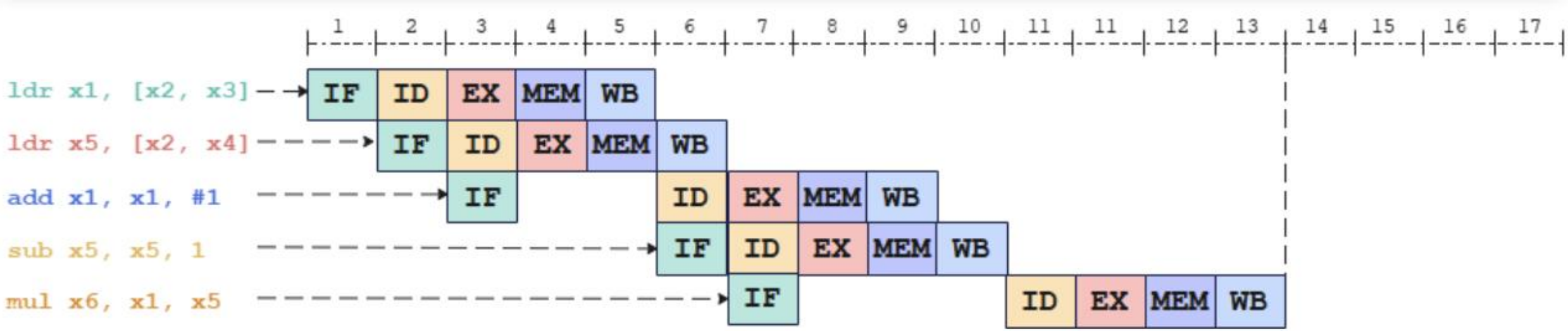
- 流水线示例

```
ldr   x1, [x2, x3]
add   x1, x1, #1
ldr   x5, [x2, x4]
sub   x5, x5, #1
mul   x6, x1, x5
```

指令调度

```
ldr   x1, [x2, x3]
ldr   x5, [x2, x4]
add   x1, x1, #1
sub   x5, x5, #1
mul   x6, x1, x5
```

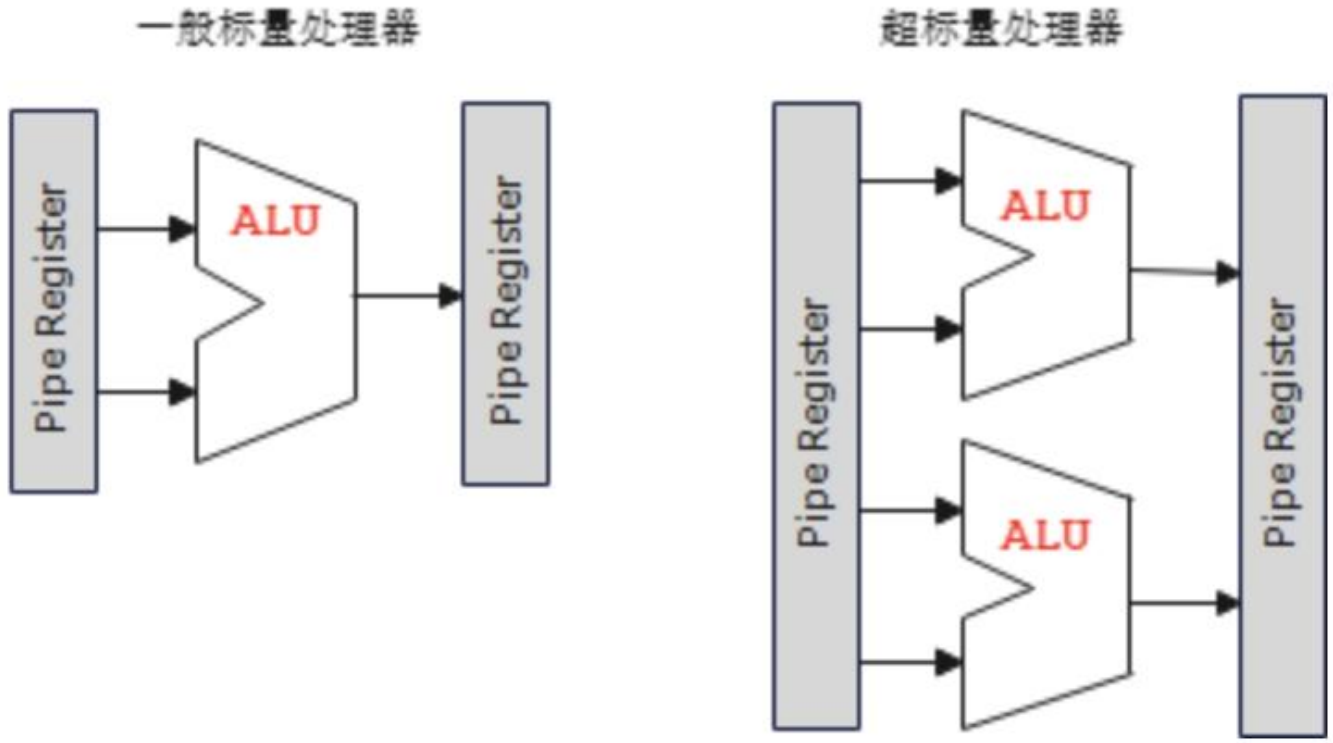
指令调度之前，耗时13个cycle



# 现代计算机的指令并行方案

## ❑超标量

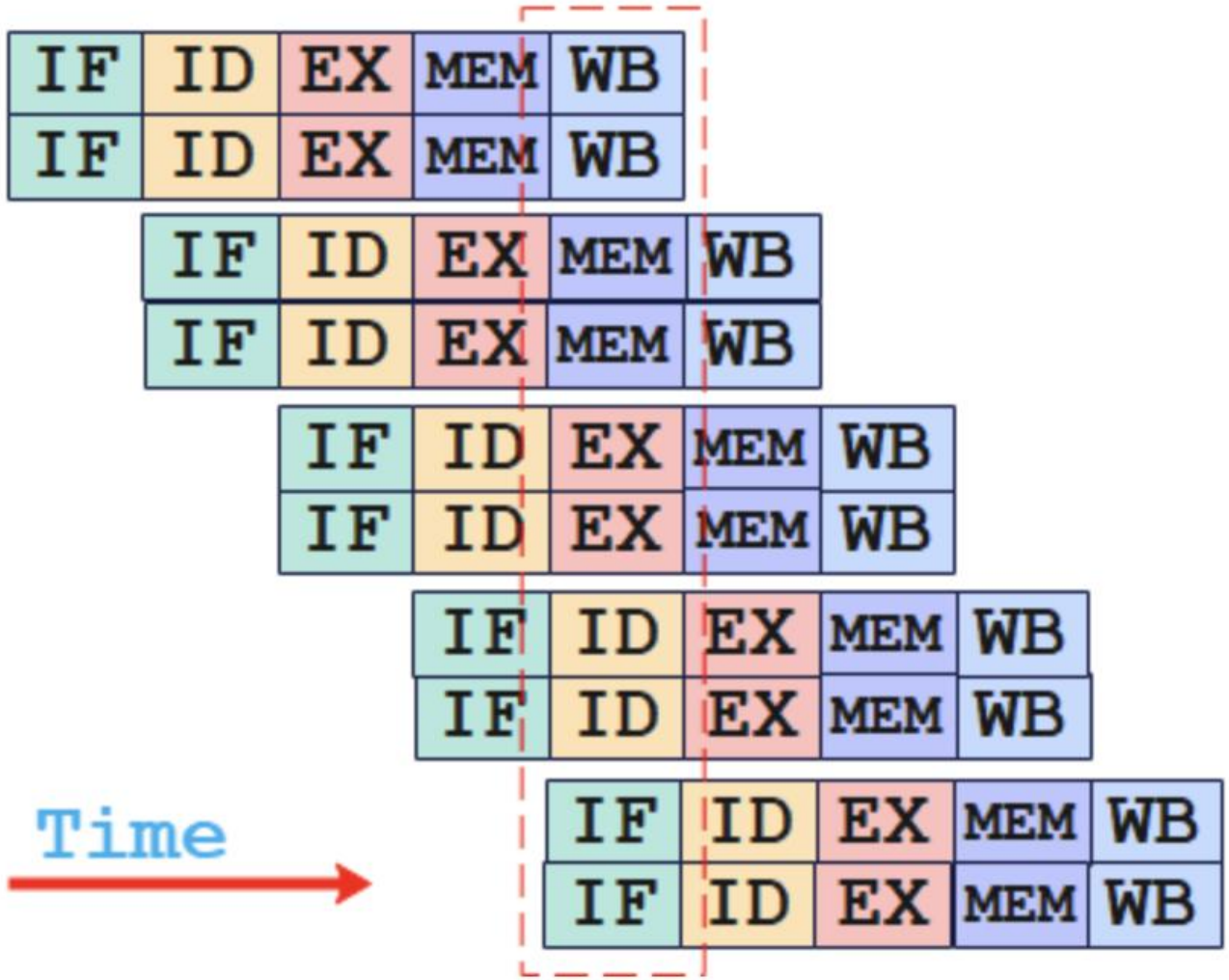
- 具备超标量结构的CPU在一个内核上集成了多个译码器、ALU等单元
- 相比于具备普通流水线技术的CPU，具备超标量技术的CPU可以在同一个阶段执行多条处在相同阶段的指令



# 现代计算机的指令并行方案

## ❑超标量

### ■超标量流水线



# 指令调度

## 指令调度与寄存器分配的关系

- 指令调度通过重排指令顺序，降低指令间依赖，提高程序的并行度

  →改变指令的执行时机也会改变指令所使用的寄存器的生命周期

- 寄存器分配尽量缩短寄存器的生命周期，让更多的数据直接存储在寄存器中
- 当对寄存器的需求超过寄存器数量时，会增加访存指令，访存指令需要纳入到指令调度的考虑

两者相互约束

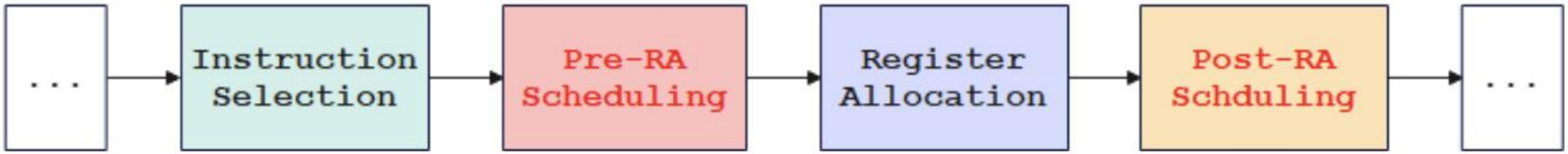
# 指令调度

- 将指令调度问题和寄存器分配问题进行联合求解相对更优的

- 由于指令调度/寄存器分配，都是NP完全问题，编译器一般分别处理

# 指令调度

- LLVM编译器的设计中，寄存器分配之前和后都会执行指令调度
  - 寄存器分配之前指令调度
    - 当前LLVM IR中分配的寄存器为虚拟寄存器，寄存器数量不受限制
    - 此时指令调度受到的约束最小，可以更大程度上提高指令并行度
  - 寄存器分配之后指令调度
    - 寄存器复用等情况会增加指令间依赖，破坏在寄存器分配之前做好的指令调度优化
    - 寄存器分配之后还要再次执行指令调度



# 指令调度的问题与约束

- 指令调度受到数据依赖约束、功能部件约束、寄存器约束等
- 在这些约束下，寻找到最优解，降低指令流水间的stall（停滞），就是指令调度的终极目标
- 指令流水间的stall主要由数据型冒险、结构性冒险、控制型冒险引起

# 指令调度的问题与约束

## 数据型冒险

- 当前指令的执行依赖与上一条指令执行结果
- 数据型冒险共有三种：写后读（RAW）、读后写（WAR）、写后写（WAW）。
- 数据冒险可能产生数据流依赖

## 结构性冒险

- 多条指令同时访问一个硬件单元的时候，由于缺少相应的资源，导致结构型冒险

## 控制型冒险

- 存在分支跳转，无法预测下一条要执行的指令，导致其产生的控制型冒险

# 指令调度的问题与约束

## ❑三种数据型冒险

### ■写后读（RAW）

- 一条指令读取前一条指令的写入结果
- 写后读是最常见的一种数据依赖类型，这种依赖被称为真数据依赖（true dependence）

```
x = 1;
y = x;
```

# 指令调度的问题与约束

## ❑三种数据型冒险

### ■读后写（WAR）

- 一条指令写入数据到前一条指令的操作数。这种依赖被称为反依赖或反相关（anti dependence）

```
y = x;
x = 1;
```

# 指令调度的问题与约束

## ❑三种数据型冒险

### ■写后写（WAW）

- 两条指令写入同一个目标。这种依赖被称为输出依赖（output dependence）

```
x = 1;
x = 2;
```

编译器解决上述冒险的常用方法就是通过插入 NOP 指令，增加流水间的*stall*来化解冒险

# 指令调度算法之表调度

## ❑表调度（List Scheduling）

- 贪心+启发式方法
- 用以调度基本块中的各个指令，是基本块中指令调度的最常见方法
- 基于基本块的指令调度不需要考虑程序控制流，主要考虑数据依赖、硬件资源

## ❑表调度思想

- 维护用来存储已经准备执行的指令的**ready**列表和正在执行指令的**active**列表
- **ready**列表的构建主要基于数据依赖约束和硬件资源信息；根据调度算法以周期为单位来执行具体的指令调度，包括从列表中选择及调度指令，更新列表信息

# 指令调度算法之表调度

## 表调度算法

- 根据指令间依赖，建立依赖关系图
- 根据当前指令节点到根节点的长度以及指令的latency，计算每个指令的优先级
- 不断选择一个指令，并调度
  - 使用两个队列维护ready的指令和正在执行的active的指令
  - 在每个周期：选择一个满足条件的ready指令并调度它，更新ready队列；检查active的指令是否执行完毕，更新active队列

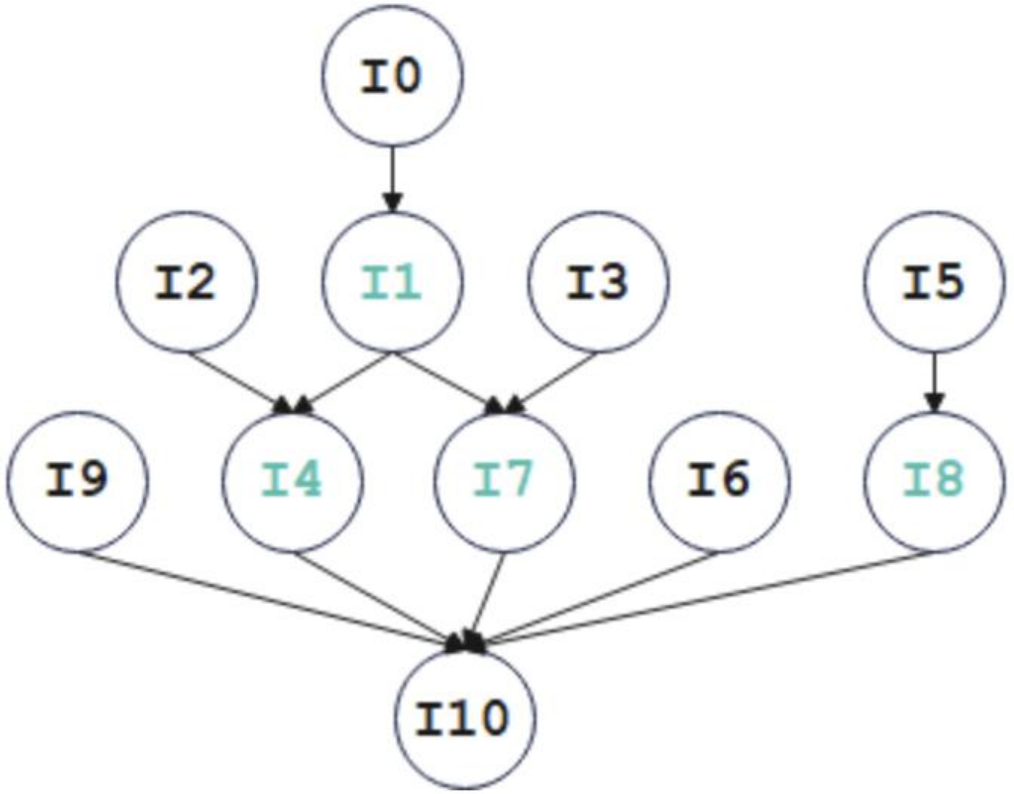
# 指令调度算法之表调度

## ❑表调度算法示例

- 假设当前CPU有两个计算单元（即每个周期可以执行两条指令）；加法指令的latency为 2 cycles，其他指令为 1 cycle

步骤一、根据数据依赖关系构建出依赖关系图。

```
I0: a = 1
I1: f = a + x
I2: b = 7
I3: c = 9
I4: g = f + b
I5: d = 13
I6: e = 19
I7: h = f + c
I8: j = d + y
I9: z = -1
I10: JMP L1
```



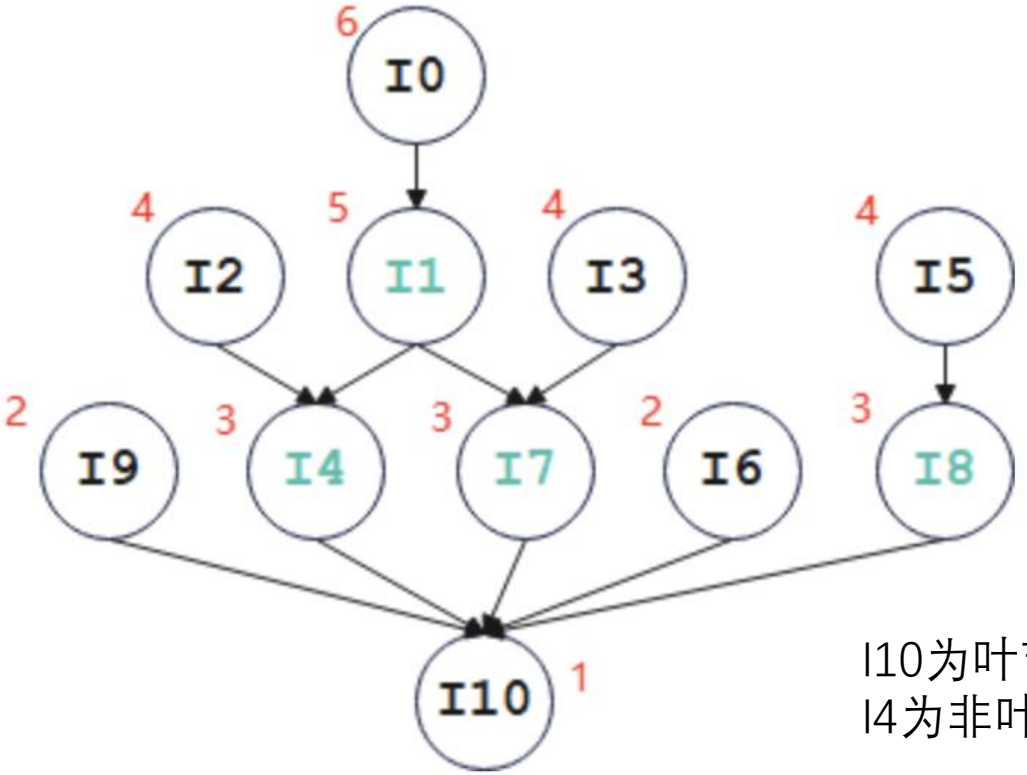
# 指令调度算法之表调度

## ❑表调度算法示例

步骤二、计算指令节点优先级

x表示当前指令节点
y表示x的子节点
E表示“true dependency”，表示“anti-dependency”（

$$priority(x) = \begin{cases} latency(x) & \text{if x is a leaf} \\ max(latency(x) + max_{(x,y)\in E}(priority(y)), max_{(x,y)\in E'}(priority(y))) & \text{otherwise} \end{cases}$$




I10为叶节点，优先级为其latency，为1
I4为非叶节点，优先级为当前节点latency（2）+ 子节点的优先级，为3

# 指令调度算法之表调度

## ❑表调度算法示例

步骤三、执行调度

选出延迟最大的指令序列是最简单的调度算法（暴力）
可以通过添加其他度量标准进一步优化优先级计算方案

| Cycle | | |
|---|---|---|
| 0 | I0 | I2 |
| 1 | I1 | I5 |
| 2 | I3 | I8 |
| 3 | I4 | I7 |
| 4 | I6 | I9 |
| 5 | I10 | --- |
| 6 | | |

| | | |
|---|---|---|
| 0 | I0 | I2 |
| 1 | I1 | I3 |
| 2 | I5 | I6 |
| 3 | I4 | I7 |
| 4 | I8 | I9 |
| 5 | --- | --- |
| 6 | I10 | --- |

```
I0: a = 1
I1: f = a + x
I2: b = 7
I3: c = 9
I4: g = f + b
I5: d = 13
I6: e = 19
I7: h = f + c
I8: j = d + y
I9: z = -1
I10: JMP L1
```

# 一起努力
# 打造国产基础软硬件体系！

徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

先进技术研究院、计算机科学与技术学院

2025年11月21日