并行编译与优化
Parallel Compiler and Optimization

计算机研究所编译系统室

Lecture 12: Instruction Scheduling Part1

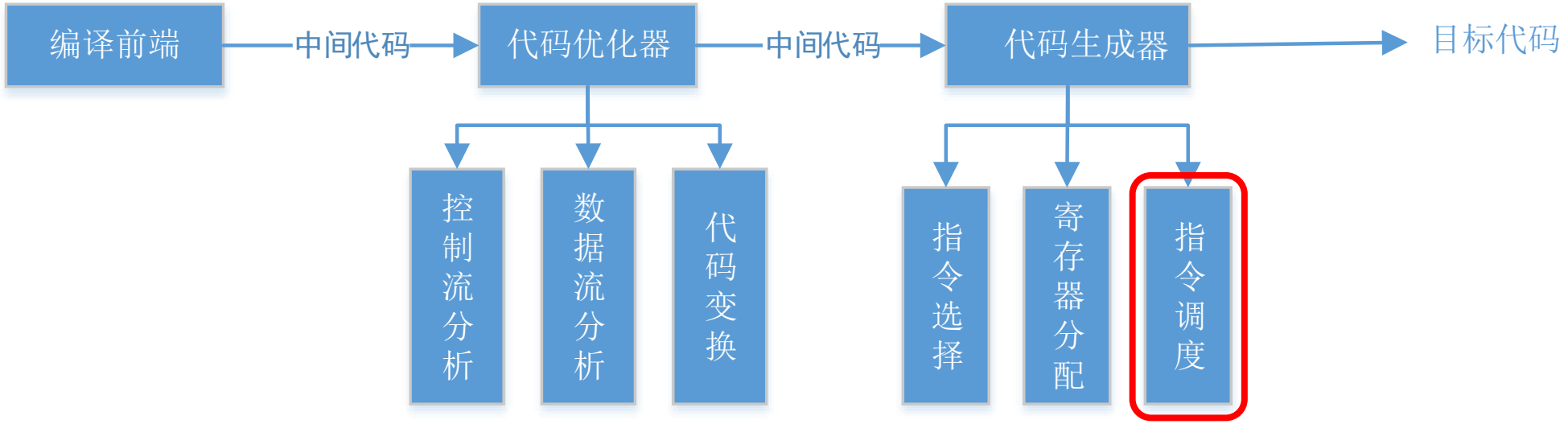
第十二课：指令调度（一）

回顾

COMP CCRG Open MP

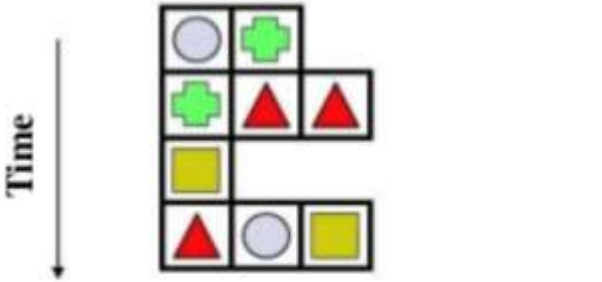
编译后端

- 指令选择: 将IR翻译为等价且成本(如执行时间)最低的目标指令序列
- 寄存器分配: 将符号寄存器分配到目标机物理寄存器或栈上
- **指令调度: 重拍目标机指令序列以最大化指令级并行**

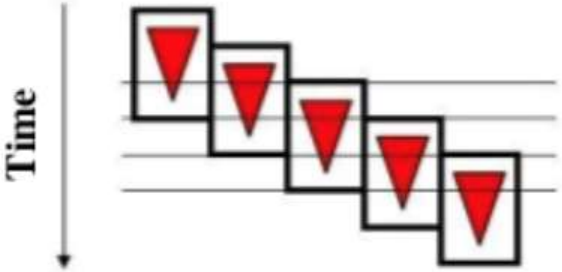


指令级并行

- 现代处理器采用多种技术以提高指令级并行，隐藏访存延迟，从而提高程序性能
  - 超标量 (Superscalar)
    - 重复设置多个功能部件
  - 多发射 (Multiple issue)
    - 每个时钟周期流出多条指令
  - 投机执行 (Speculative execution)
    - CPU分支预测器(Branch predictors)
    - 投机取指令 (Speculative loads)
  - 乱序执行 (Out-of-order execuion)
  - 更深的流水线 (Deep pipelines)



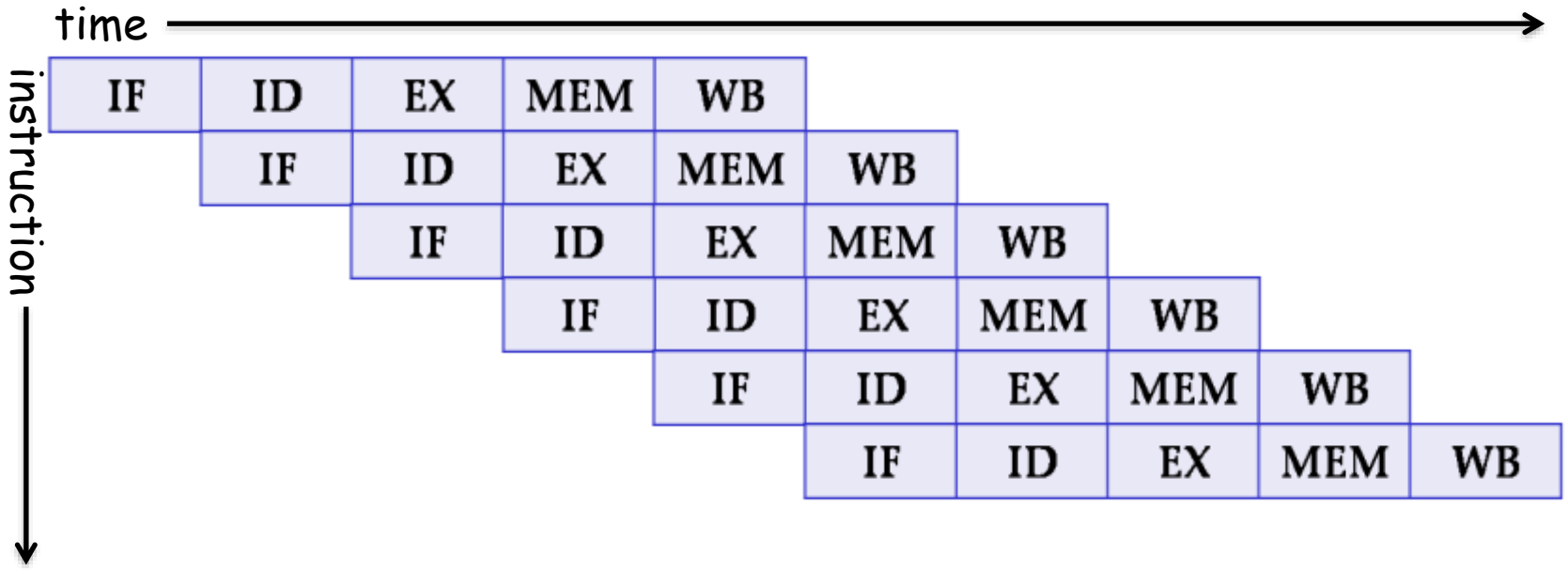
Instruction-Level Parallelism (ILP)



Pipelining

流水线

Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
Instruction fetch →	Instruction decode →	Execute →	Memory access →	Register write-back
IF →	ID/RF →	EX →	MEM →	WB



编译器：如何调度指令序列，才能提高程序的指令级并行?

学习内容

COMP
CCRG Open MP

12.1 指令调度概述

12.2 指令调度的先决条件

12.3 局部调度: 基本块的表调度方法

12.4 优化指令调度的技术

学习目标

- 掌握基本块的表调度方法，能够运用表调度方法排列出执行时间最短的指令调度
- 熟悉优化指令调度的技术，理解指令调度的约束和先决条件

12.1.1 指令调度的基本概念

- 指令调度是通过重排指令序列中各指令的执行顺序，试图减少程序总执行时间

- 指令调度的目标
  - 最大化指令级并行
  - 减少流水线中的气泡(bubbles)
- 最优指令调度是NP完全问题

小测试

- (1) 什么是流水线中的“气泡”？
- (2) 为什么会出现“气泡”？

- “气泡” 是流水线中的停顿(stalls)
- 流水线中某些指令重叠执行造成冲突，使得需要在流水线中插入“气泡”以消除冲突

12.1.2 指令调度的约束

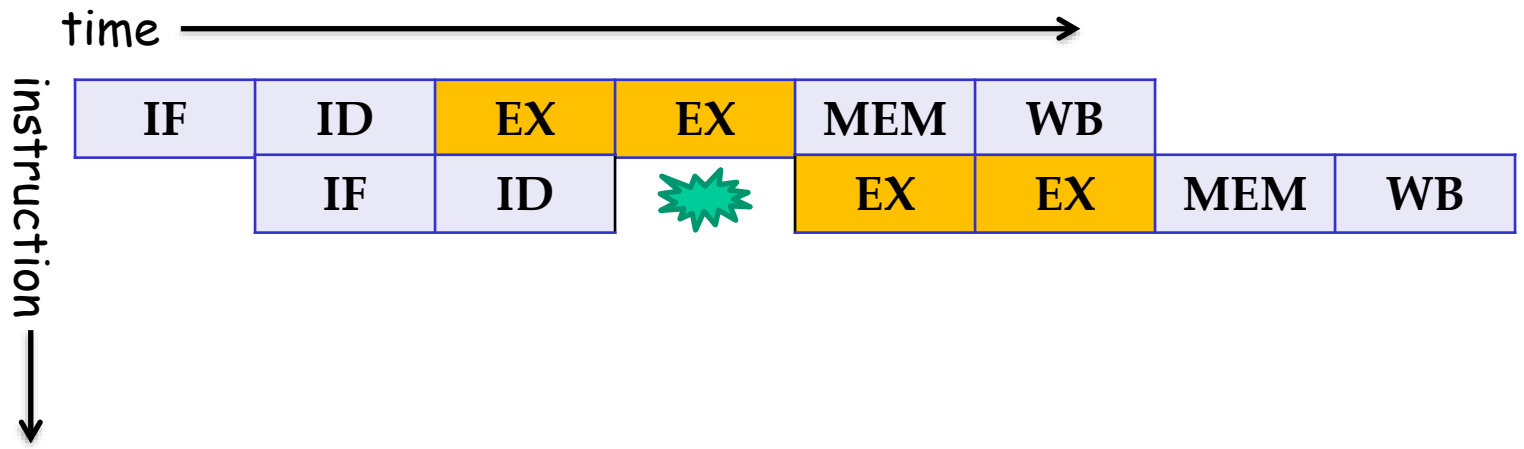
- 结构冲突 (Structural hazards)
- 数据冲突 (Data hazards)
- 控制冲突 (Control hazards)

12.1.2 指令调度的约束

结构冲突

- 由资源约束导致的冲突
- 没有足够的资源来开发并行性，如功能部件

```
mul $r1,$r2,$r3   // Suppose multiplies take two cycles
mul $r4,$r5,$r6
```

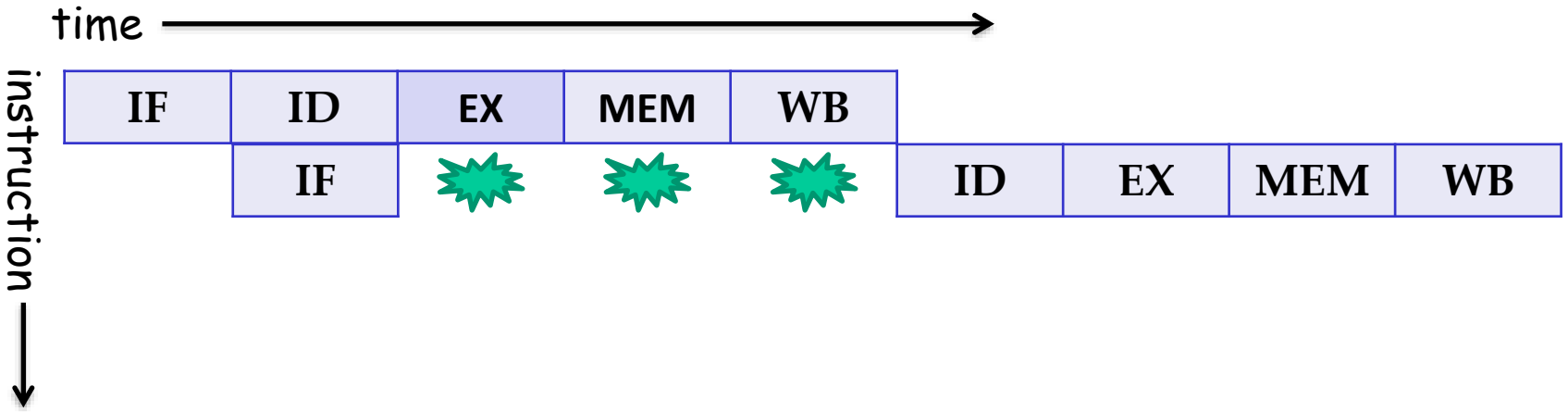


12.1.2 指令调度的约束

数据冲突

- 由数据依赖关系导致的冲突
- 指令依赖于前序指令尚未计算出或写回的结果

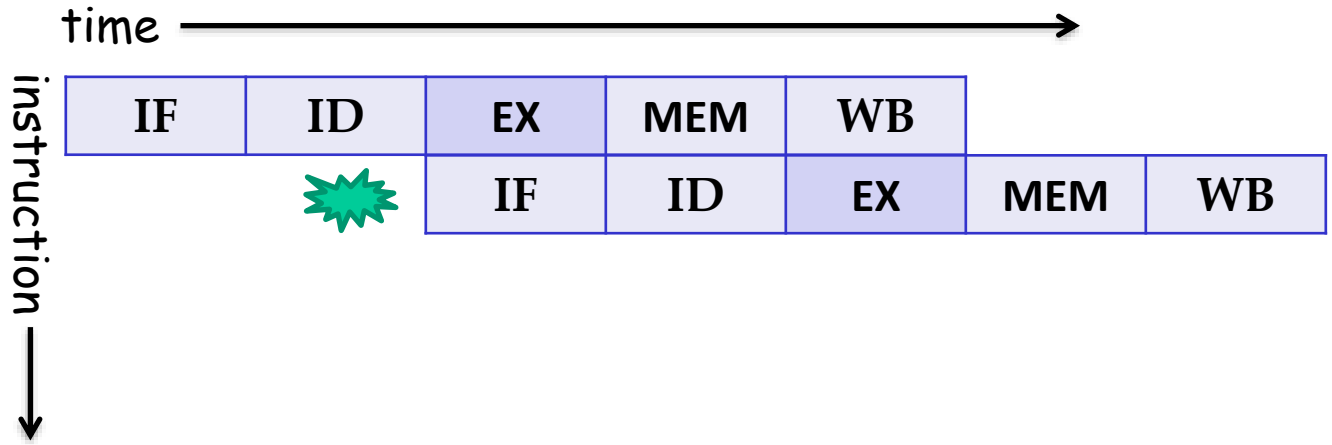
```
mul $r1, $r2, $r3    //Suppose no data forwarding
mul $r4, $r1, $r6
```



12.1.2 指令调度的约束

- 控制冲突
  - 由控制依赖关系导致的冲突
  - 分支和跳转指令修改程序计数器
  - 影响到底哪条指令应该发射到流水线上

```
bz $r1, label
add $r2, $r3, $r4
```



12.1.3 硬件解决方案

- 结构冲突
  - 复制多个功能部件 (Superscalar)
  - 更深的流水 (Deep pipelines)
- 数据冲突
  - 数据定向技术 (Data forwarding)
  - 动态指令调度，乱序执行 (Out of order execution)
- 控制冲突
  - 硬件分支预测 (Branch prediction)
  - 运行时投机执行 (Speculative execution)

学习内容

12.1 指令调度概述

12.2 指令调度的先决条件

12.3 局部调度: 基本块的表调度方法

12.4 优化指令调度的技术

指令调度的先决条件

- 资源预约表 (Resource reservation table)
  - 每条指令需要使用的资源
- 机器描述 (Machine description)
  - 对机器可用资源的描述
- 依赖图 (Dependence graph)
  - 待调度的程序需要构建依赖图

指令调度和目标机紧密相关

12.2.1 资源预约表

- 在指令调度时，使用资源预约表检查是否存在结构冲突
  - 如果两条指令S1和S2在同一时刻需要相同的资源，则S1和S2冲突

add:

Time	read src1 opnd	read src2 opnd	ALU stage 0	ALU stage 1	MULTIPLIER stage 0	MULTIPLIER stage 1	MULTIPLIER stage 2	MULTIPLIER stage 3	write result bus
0	X	X							
1			X						
2				X					
3									X

latency(add) = 4

mul:

Time	read src1 opnd	read src2 opnd	ALU stage 0	ALU stage 1	MULTIPLIER stage 0	MULTIPLIER stage 1	MULTIPLIER stage 2	MULTIPLIER stage 3	write result bus
0	X	X							
1					X				
2						X			
3							X		
4								X	
5									X

latency(mul) = 6

假设add指令在mul指令后发射，也需要在mul指令后写回，
add指令可以只落后2拍跟着mul指令吗？ No!

回顾: 依赖关系

- 在程序执行过程中，如果指令(语句)A必须在指令(语句)B之前执行，则称B依赖于A
  - 依赖关系是程序执行顺序上的约束

```
S1:          x1 =a+b
S2:          x2 = x1 *a
S3:          if (x2 > c) goto L1
S4:          x3 =x1/x2
S5:          goto L2
S6:    L1:   x3 =x2
S7:    L2:   x2 = ...
```

- S1给变量x1赋值，S2使用x1的新值，那么S1必须先于S2执行，S2数据依赖于S1
- 执行S4还是S6，取决于S3比较指令的结果，因此，S4、S6控制依赖于S3

回顾: 依赖关系

- 根据约束发生的来源
  - 如果约束是由程序的**控制流**引起，则称之为控制依赖
  - 如果约束是由程序的**数据流**引起，则称之为数据依赖
    - 由程序的定值-使用关系导致

(1) 控制依赖

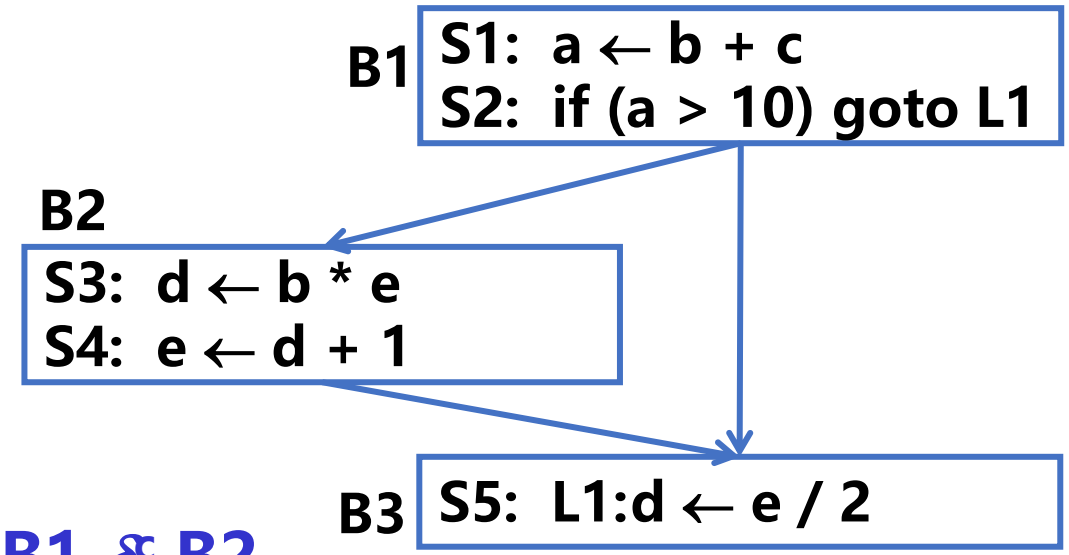
- 如果一条语句S2的执行与否依赖于S1的输出，那么S2控制依赖于S1，记作S1 δ^c S2
- 当基本块B2的执行与否依赖于基本块B1的输出，称基本块B2控制依赖于基本块B1，记作B1δ^cB2

```
S1:          a ← b + c
S2:          if (a > 10) goto L1
S3:          d ← b * e
S4:          e ← d + 1
S5:  L1:     e ← d / 2
```

S2 δ^c S3

S2 δ^c S4

S2 δ^c S5 ??



B1 δ^c B2

B1 δ^c B3 ??

(2) 数据依赖

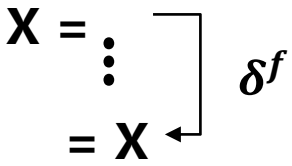
- 语句T依赖于语句S是指存在S的一个实例S'和T的一个实例T'以及一个存储单元M，满足
  - S'和T'都访问M(读或写)，且**至少一个是写操作**
  - 当程序串行执行时，S'是在T'之前执行 (S'◁ T')
  - 在同一次执行中，在S'执行结束与T'开始执行前没有对存储单元M的写操作

1. 访问同一个存储单元
2. 其中一次访问是“写”
3. 执行顺序

(2) 数据依赖

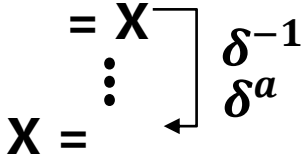
- 三类数据依赖关系

流依赖(flow-)



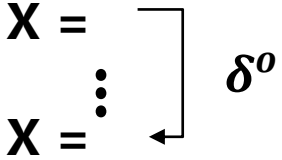
先写后读

反向依赖(anti-)



先读后写

输出依赖(output-)



先写后写

- 流依赖又称为真依赖
- 反向依赖和输出依赖可以通过寄存器重命名等优化消除

12.2.2 依赖图

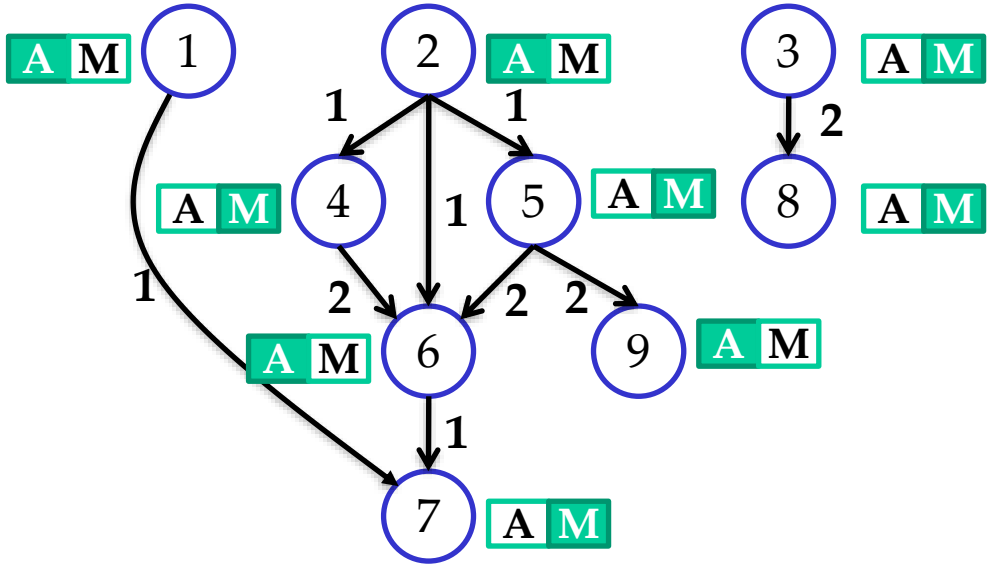
- 基本块内数据依赖关系用数据依赖图G=(N,E)表示
  - 结点是指令
    - 结点集合N中的每个结点n有一个资源预约表RT_n
  - 有向边(S→T)表示S和T之间存在数据依赖
    - S是依赖源，T是依赖槽
    - S必须在T之前执行
    - 用延迟d_e标记每一条边，表示S流出后，T至少需要延迟d_e拍后才能流出
  - 数据依赖图是有向无环图 (DAG, Directed Acyclic Graph)
    - 基本块内，动态执行顺序=静态词法顺序

12.2.2 依赖图

示例

dst src src

```
1   addi   $r2, 1, $r1
2   addi   $sp, 12, $sp
3   st     a, $r0
4   ld     $r3, -4($sp)
5   ld     $r4, -8($sp)
6   addi   $sp, 8, $(sp)
7   st     0($sp), $r2
8   ld     $r5, a
9   addi   $r4, 1, $r4
```



假设：latency(ALU op) = 1
latency(MEM op) = 2

学习内容

12.1 指令调度概述

12.2 指令调度的先决条件

12.3 局部调度: 基本块的表调度方法

12.4 优化指令调度的技术

表调度方法

- 贪婪的启发式方法
- 以基本块为单位的局部调度方法
  - 调度基本块中的指令序列（指令序列中没有分支）
- 能够发现合理的调度，容易修改以适应计算机体系结构的改变
  - 20世纪70年代末开始成为指令调度的主要范式

12.3.1 表调度基本思想

构建依赖图G

候选调度表Candidates ← G中所有根结点 (没有入边的结点)

while(当候选调度表 ≠ ∅)

从候选调度表中选择一条指令s

根据启发式信息按序选择一条指令

调度s

//将s从候选调度表中删除

Candidates ← Candidates – {s}

//候选调度表=候选调度表 ∪ “曝露”的结点

Candidates ← Candidates ∪ “exposed” nodes

“曝露”的结点是所有前驱结点都已被调度了的结点

12.3.1 表调度基本思想

- 表调度算法不会回溯
  - 对依赖图中的每个结点只进行一次调度
- 根据启发式信息给结点设置优先级，根据优先级选择下一个进行调度的结点
  - 关键路径
  - 结点后继数
  - 资源需求

12.3.2 启发式信息

- **关键路径**
  - 是依赖图中的最长路径
  - 用结点高度(从结节点开始的最长路径的长度)作为优先级度量函数
- **结点后继数**
  - 后继越多，表示该结点调度完成后，曝露的可被调度的结点越多，下一次调度的候选更多，更具灵活性
- **资源需求**
  - 优先调度需要较多资源的结点，这些结点往往容易导致更多依赖
    - 如可用的寄存器，调度使用较多寄存器的结点可以减少依赖和寄存器压力

12.3.3 表调度算法

Build dependence graph G

Assign each instruction a priority (heuristic)

Create a list (priority queue) of candidate instructions

    all predecessors already scheduled

Repeat

    Remove highest-priority instruction s from list

    add it to schedule

    add newly-eligible instructions to list (reorder)

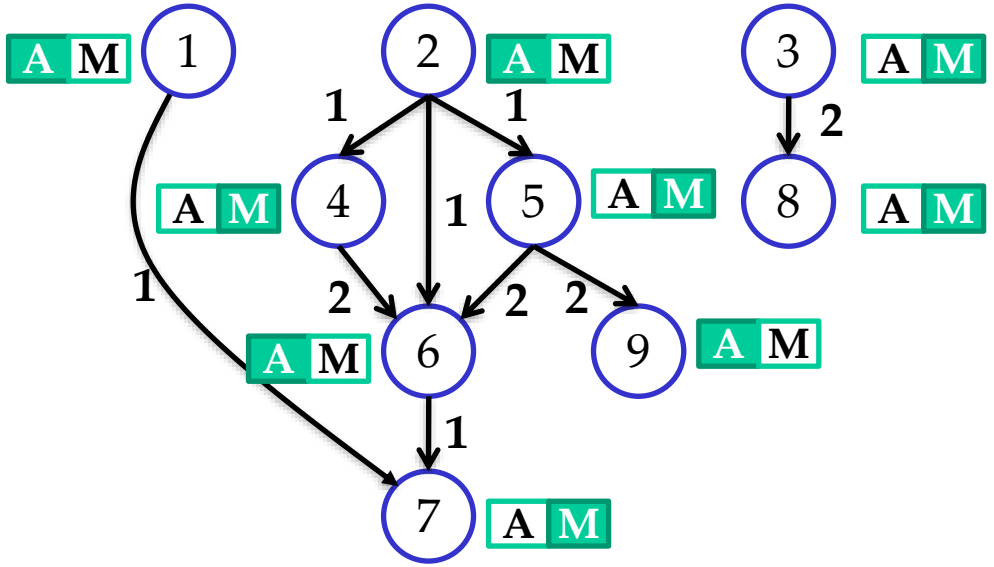
until all instructions have been scheduled

Schedule s in the earliest slot that satisfies data dependence + resource constraints with all predecessors

12.3.4 表调度算法示例

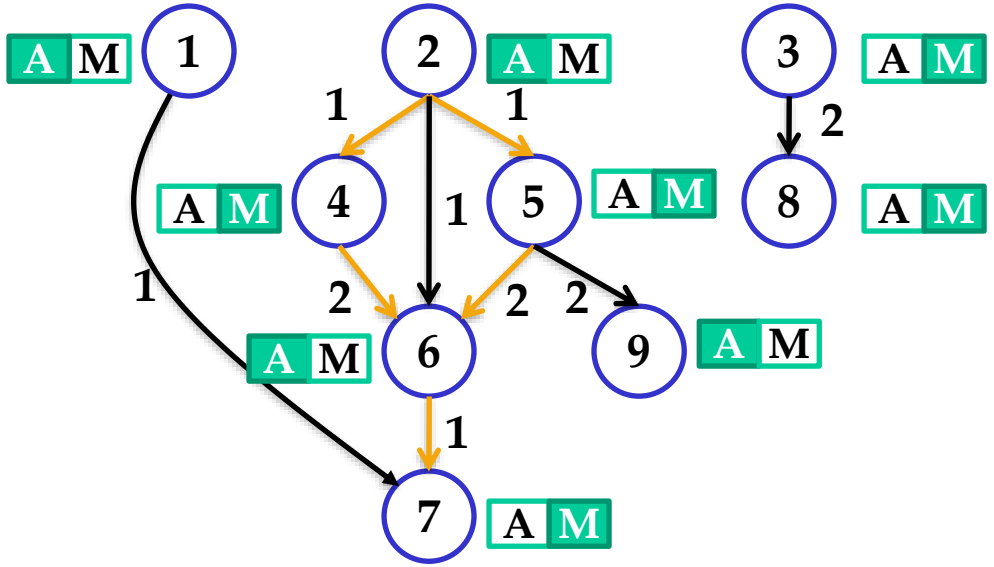
构建依赖图

```
                dst src src
                ↓   ↓   ↓
1    addi   $r2, 1, $r1
2    addi   $sp, 12, $sp
3    st     a, $r0
4    ld     $r3, -4($sp)
5    ld     $r4, -8($sp)
6    addi   $sp, 8, $(sp)
7    st     0($sp), $r2
8    ld     $r5, a
9    addi   $r4, 1, $r4
```



假设：latency(ALU op) = 1
latency(MEM op) = 2

12.3.4 表调度算法示例

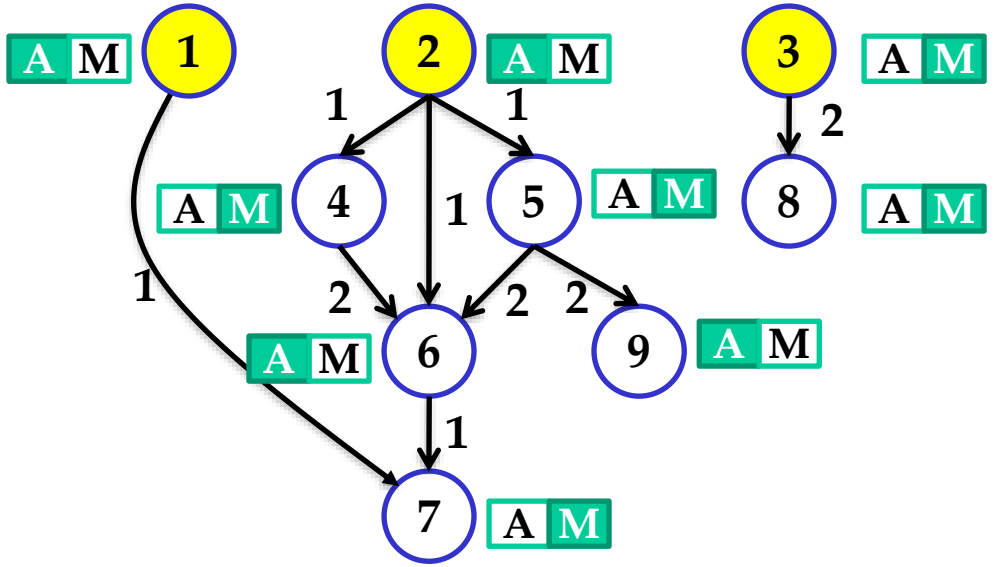


Critical Path: 2-4-6-7
2-5-6-7

Candidates list: 2→3→1
Schedule:

A	2		`
M	3		

12.3.4 表调度算法示例

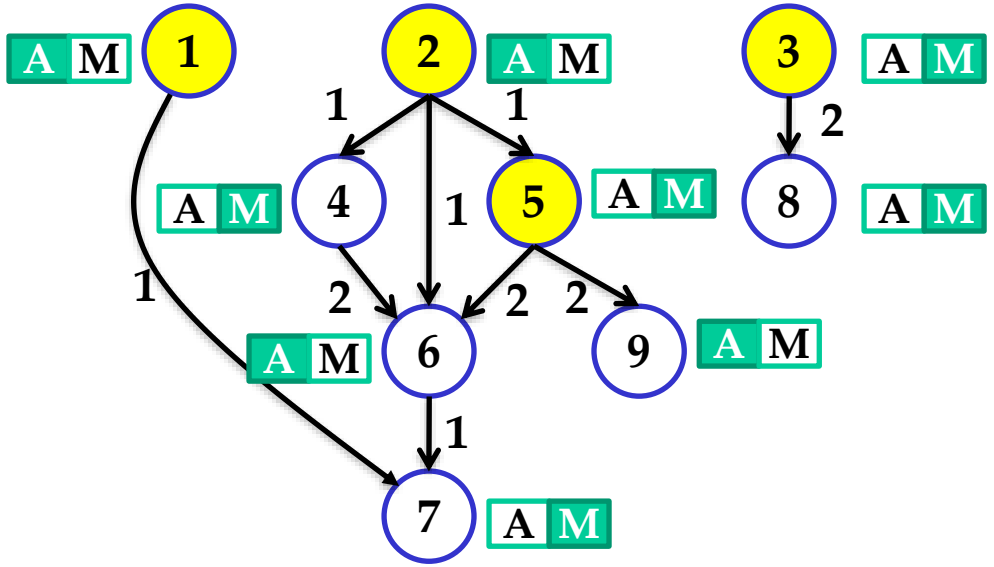


Critical Path: 2-4-6-7
2-5-6-7

Candidates list: 5→4→8
Schedule:

A	2	1	
M	3		

12.3.4 表调度算法示例

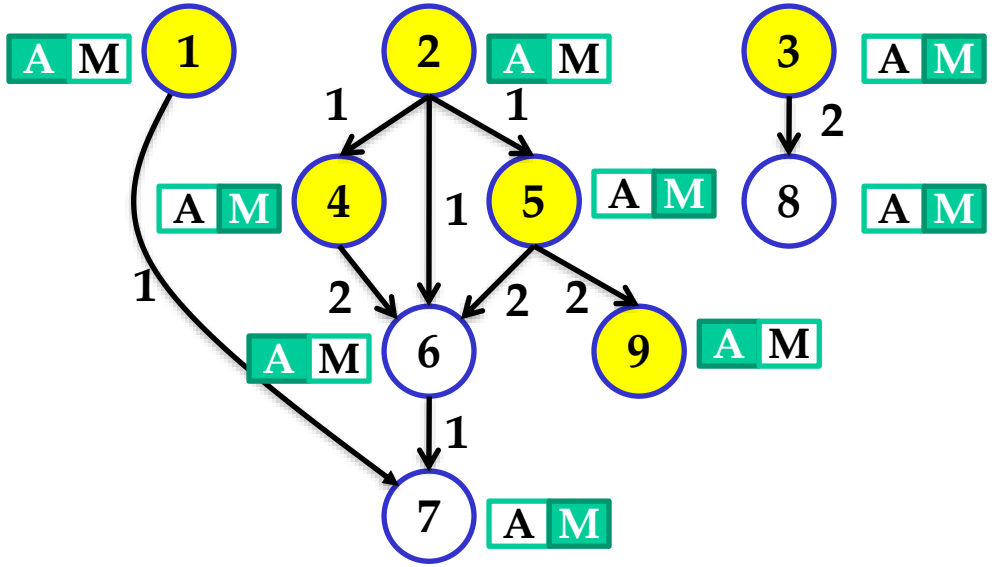


Critical Path: 2-4-6-7
2-5-6-7

Candidates list: 4→8→9
Schedule:

A	2	1			
M	3		5		

12.3.4 表调度算法示例

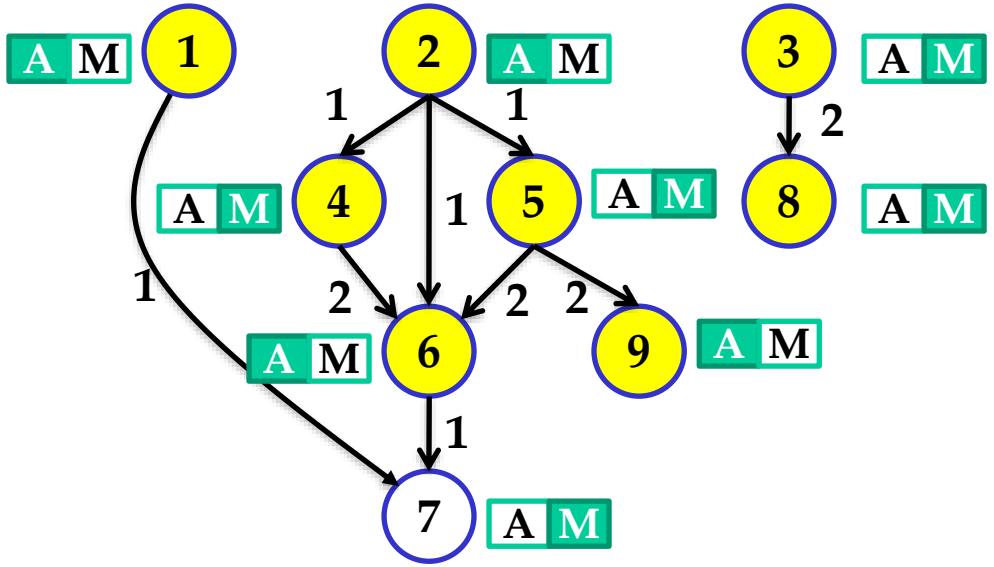


Critical Path: 2-4-6-7
2-5-6-7

Candidates list: 8→6
Schedule:

A	2	1			9		
M	3		5		4		

12.3.4 表调度算法示例

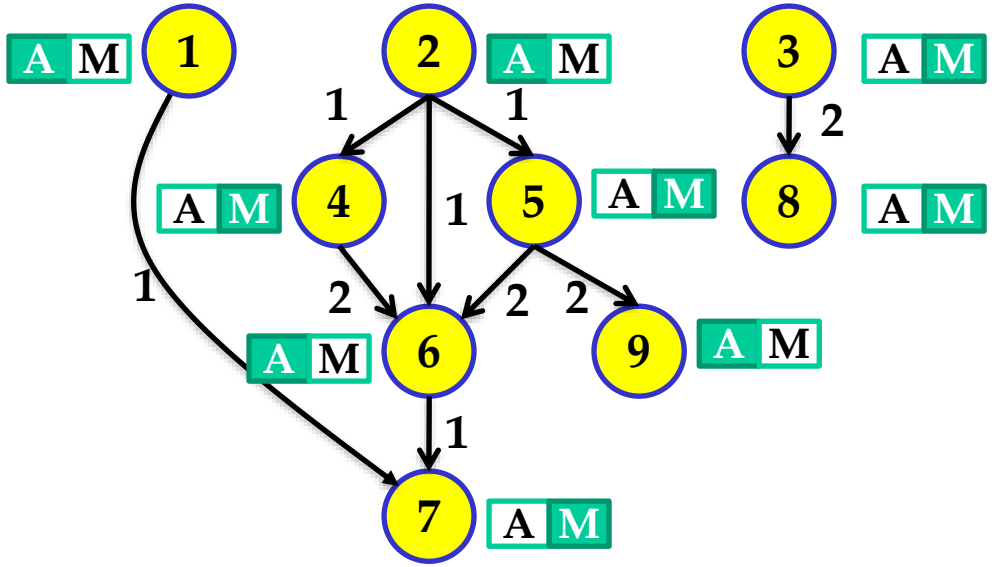


Critical Path: 2-4-6-7
2-5-6-7

Candidates list: 7
Schedule:

A	2	1			9		6			
M	3		5		4		8			

12.3.4 表调度算法示例



Critical Path: 2-4-6-7
2-5-6-7

Candidates list: 7
Schedule:

A	2	1			9		6			
M	3		5		4		8		7	

10 cycles

12.3.5 表调度算法复杂度

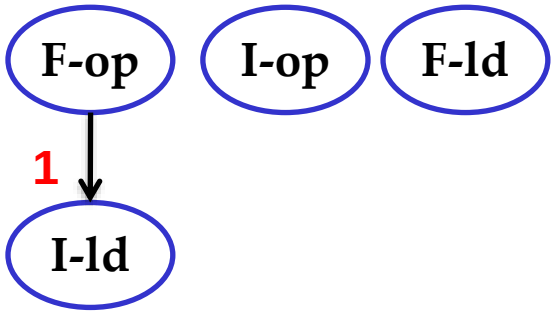
- 算法复杂度是指令数N的二次方
  - 构建依赖图是$O(N^2)$
  - 每一次调度最坏情况需要检查所有指令 O(N)
  - 实践中，算法复杂度接近于线性

12.3.6 贪婪的表调度

- 表调度算法是贪心算法
  - 尽可能用候选指令填充指令槽
  - 对于具有多流出的超标量机器或者超长指令字机器更有效
  - 但算法不一定总能发现最优调度

12.3.6 贪婪的表调度

- 假设处理器有ALU和MEM两个功能部件
  - ALU支持整型操作(I-op)和浮点操作(F-op)
  - MEM支持访存操作(F-ld, I-ld)和整型操作(I-op)
  - 所有操作延迟均为1拍



dependence graph

ALU	MEM
F-op	F-ld
I-op	I-ld

调度1： 2 cycles

ALU	MEM
F-op	I-op
	I-ld
	F-ld

调度2: 3 cycles

学习内容

COMP
CCRG Open MP

12.1 指令调度概述

12.2 指令调度的先决条件

12.3 局部调度: 基本块的表调度方法

12.4 优化指令调度的技术

优化指令调度的技术

- **寄存器重命名 (Register renaming)**
  - 消除反向依赖(WAR)和输出相关(WAW) 两种“伪相关性”
- **平衡调度 (Balanced scheduling)**
  - 隐藏访存延迟
- **循环展开 (Loop unrolling)**
  - 增加基本块的大小，从而有更多指令可以用于调度

12.4.1 寄存器重命名

- 在寄存器分配时，没有冲突的变量可以使用同一寄存器，但有可能导致产生反向依赖和输出依赖
- 通过寄存器换名(换用一组寄存器)，减少对寄存器的重用，从而消除不必要的依赖关系，增加指令调度的灵活性

```
add   $r1, $r2, 1          add   $r1, $r2, 1          add   $r1, $r2, 1
st    [$fp+52], $r1   →    st    [$fp+52], $r1   →    mul   $r4, $r3, 2
mul   $r1, $r3, 2          mul   $r4, $r3, 2          st    [$fp+52], $r1
st    [$fp+40], $r1        st    [$fp+40], $r4        st    [$fp+40], $r4
```

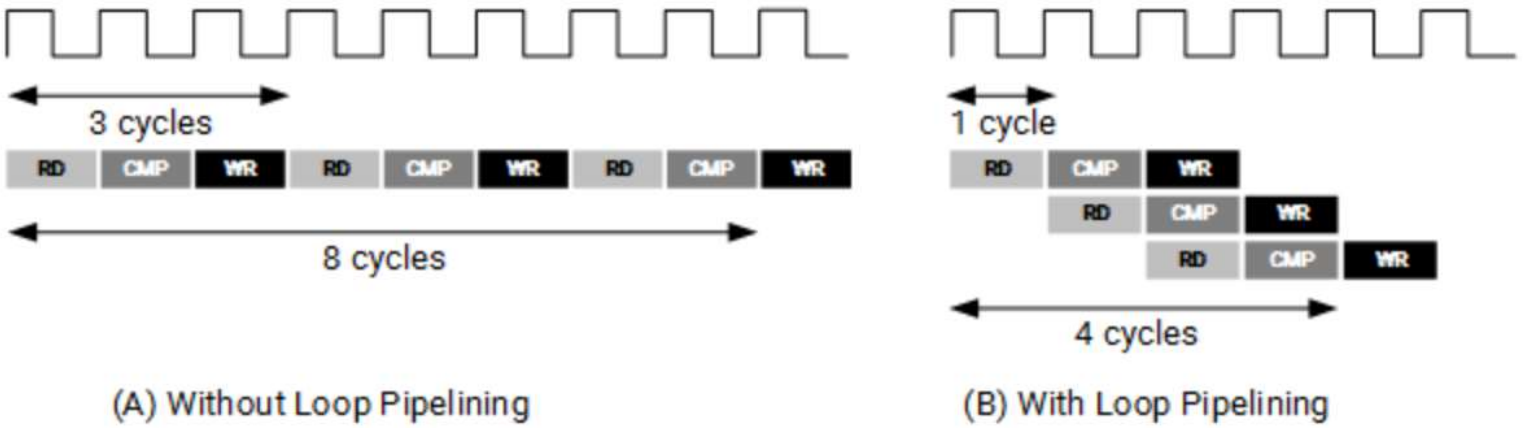
12.4.1 寄存器重命名

- 寄存器换名除通过编译器静态完成，还可通过硬件动态完成
  - Tomasulo算法乱序执行
- 寄存器轮转
  - 与循环展开配合使用

```
void func(m,n,o) {

 for (i=2;j>=0;i--) {
    op_Read;
    op_Compute;
    op_Write;

 }
}
```



(A) Without Loop Pipelining

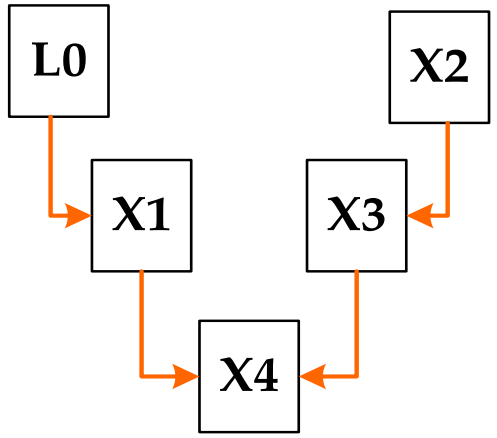
(B) With Loop Pipelining

12.4.1 寄存器重命名

- 尽可能早地调度load指令
  - load指令可能需要花费许多时钟周期
  - 访存延迟取决于L1/L2 caches命中, cache失效, TLB失效等
- 通过在load指令后插入与load无关的指令来隐藏访存延迟
- 插入多少条无关指令?
  - 根据编译分析和访存延迟确定
  - cache失效和命中情况下插入的无关指令数有区别

12.4.2 平衡调度

- load延迟
  - Cache命中: 1 cycles — 乐观
  - Cache失效: N cycles(N≥10) — 悲观



Optimistic	*Pessimistic*
L0	L0
X2	X2
X1	X3
X3	X1
X4	X4

乐观: 命中时获得最优调度(5 cycles)，失效时效果较差

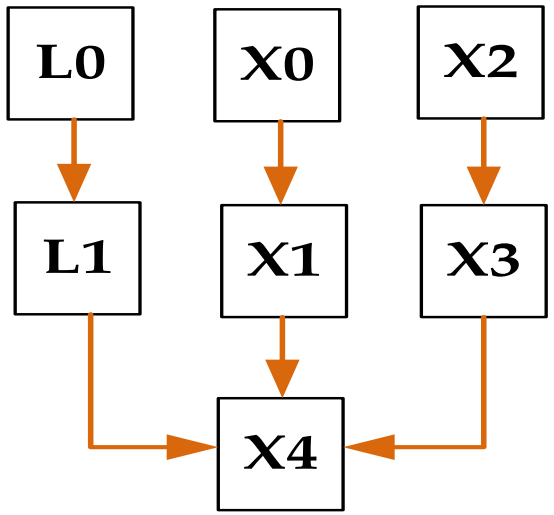
悲观: 命中时获得最优调度(5 cycles)，失效时效果较好

12.4.2 平衡调度

- 由于对访存延迟没有固定的最优估计，因此基于代码中现有的并行度进行平衡调度
  - 也叫访存级并行 (Load level parallelism)
- 基本方法
  - 计算每一条load指令的权重
  - 把load指令后的指令槽空出来，填充可以被调度的指令

12.4.2 平衡调度

- 多条laod指令的情况
  - W是对load指令权重的估计
  - W=5 – 高估
  - W=1 – 低估
- 平衡调度
  - W=3
  - 访存级并行



Traditional (W=5)	Traditional (W=1)	Balanced
L0	L0	L0
X0	L1	X0
X1	X0	X1
X2	X1	L1
X3	X2	X2
L1	X3	X3
X4	X4	X4

HPC TOP500

- TOP500排行榜根据HPL(High Performance Linpack)程序实测峰值性能排名
  - 其中计算最密集的核心函数是DGEMM双精度矩阵乘

- 专家级优化的DGEMM内核中就使用了寄存器轮转和平衡调度技术来优化指令调度

本讲小结

- 指令调度的约束
  - 结构冲突，数据冲突，控制冲突
- 指令调度的先决条件
  - 资源预约表，机器描述，依赖图
- 表调度方法
  - 局部调度，作用于基本块
  - 一次调度，贪心算法 (尽可能用更多候选指令填充指令槽)
- 优化指令调度的技术
  - 寄存器换名、平衡调度、循环展开

作业

- 使用表调度算法对下面的代码进行指令调度。假设该目标机包含3个功能部件MEM、ALU1和ALU2，其中add和mul指令既可执行在ALU1上也可执行在ALU2上。指令的延迟分别是latency(add) = 1拍，latency(mul) = 2拍，latency(ld) = 2拍，latency(st) = 3拍

(1)画出带延迟的数据依赖图

(2)用表的形式给出执行时间最短的指令调度，并给出上述代码完成所需要的周期数

备注：不考虑寄存器重命名优化

```
S1:    ld R1, 0(R2)
S2:    add R2, R2, 4
S3:    ld R4, 0(R5)
S4:    add R5, R1, R4
S5:    add R5, R5, R2
S6:    mul R1, R2, R1
S7:    st 0(R1), R5
```

Cycle	0	1	2	3	4	…
MEM						
ALU1						
ALU2						

参考资料

- 《高级编译器设计与实现》(鲸书) 第9、17章
- 《现代编译原理C语言描述》(虎书) 第20章
- 《编译器设计》第12章