90年代讲处理器要讲3超
超标量
超流水
超长指令字

高等计算机系统结构

如今超流水技术已不再作为独立的技术来讲，通常都会在superscalar中实现超流水

VLIW/EPIC
基于静态调度的ILP

（第六讲）

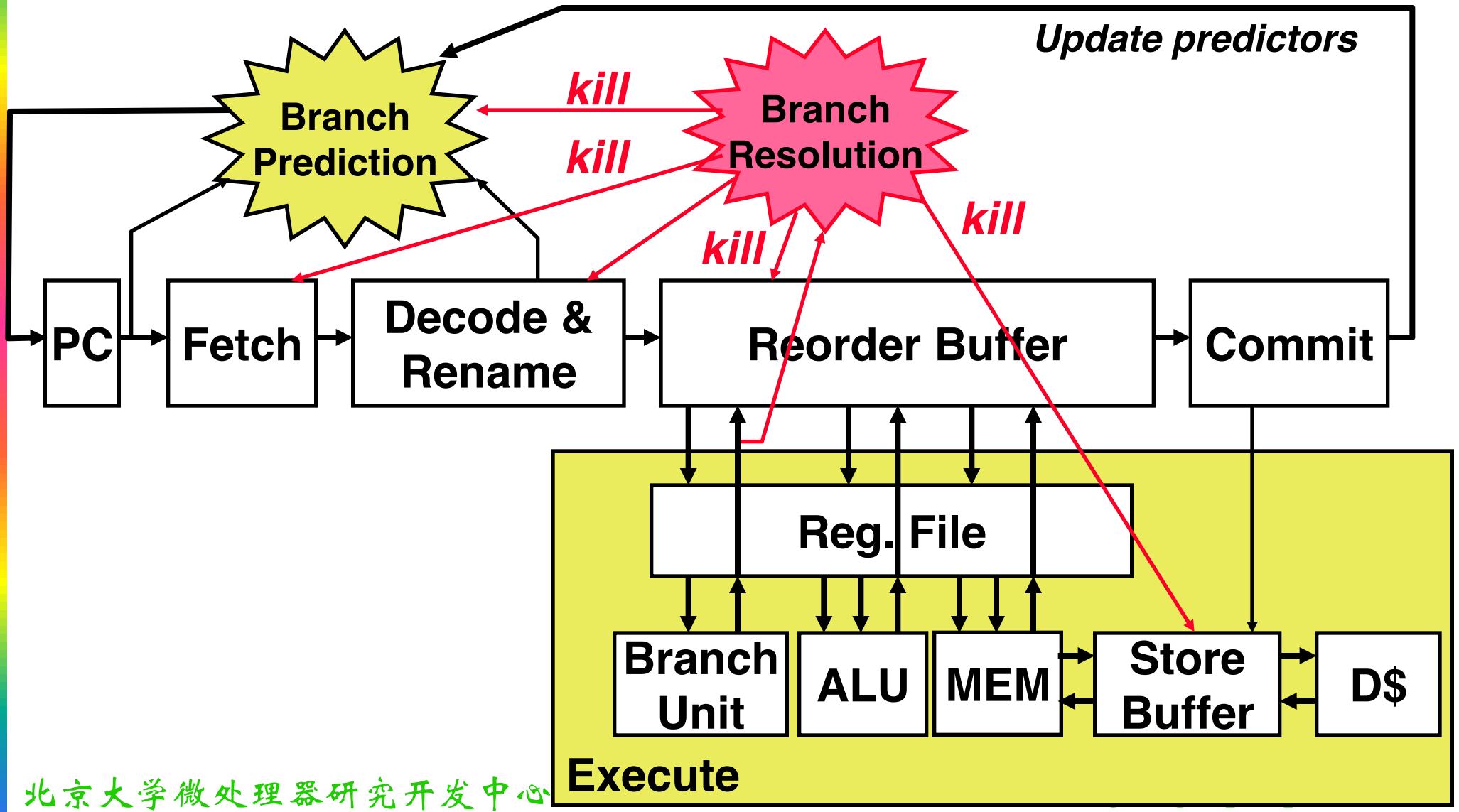
程 旭

2014年4月21日

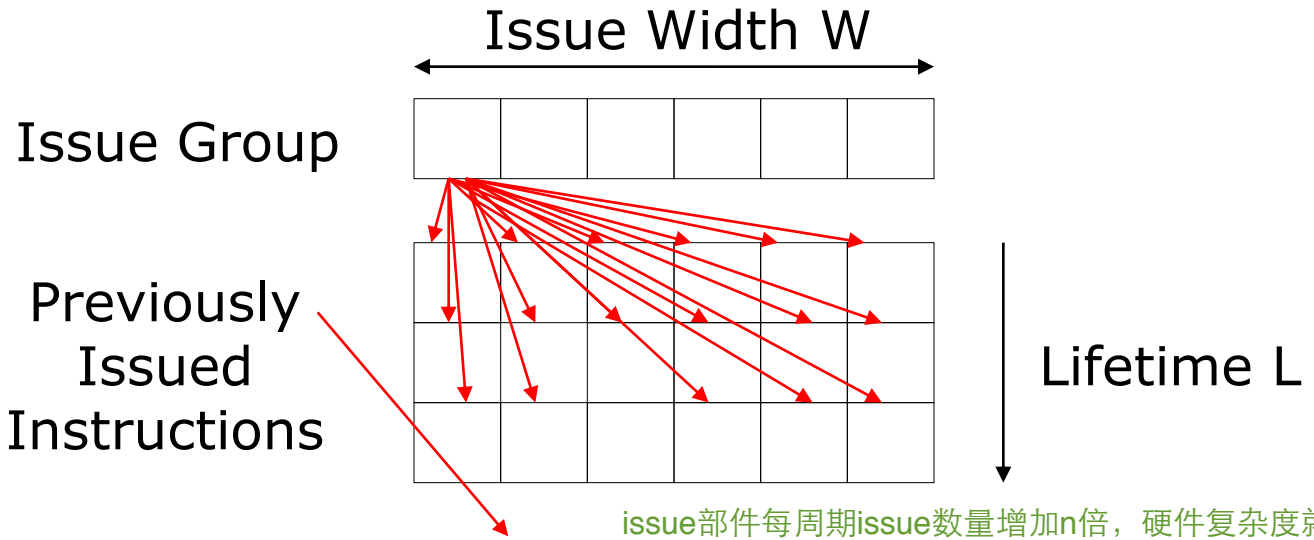
复习：先进超标量技术

- 简单的转移预测技术也会有不错的效果
- 基于路径（Path-based）的预测器可取得>95%的正确率
- BTB可以在流水线的前期就改变控制流，BHT的每个表项成本小，但需要等到指令译码
- 转移错误预测的恢复需要使用流水线的快照（snapshots）以减少损失
- 统一物理寄存器堆的设计，可以避免从不同地方(ROB+arch regfile)读取数据
- 超标量处理器可以在一个时钟周期对多条相关指令进行重命名
- 需要采用推测式存储缓冲器（speculative store buffer）来避免等待存储操作的确认

复习：转移预测和推测式执行



超标量处理器控制逻辑的可扩展性

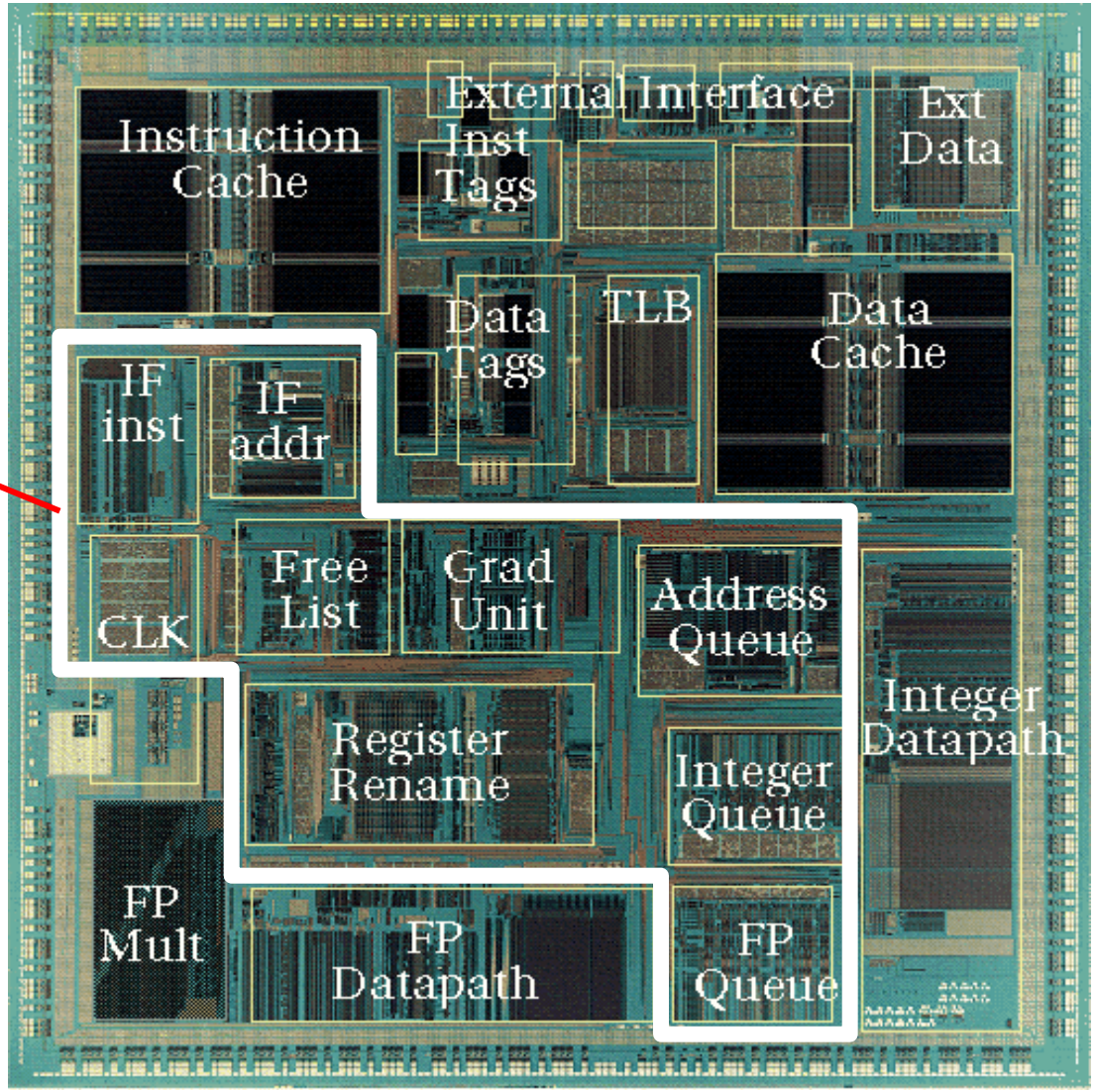


issue部件每周期issue数量增加n倍，硬件复杂度就增加2^n倍
硬件复杂度太高会导致时钟频率不能太高
因此有人说能不能尽量编译优化多做一点事情，让硬件复杂度少一些

- 每条被发射的指令都必须检查与W*L条指令间的关系，即硬件的增长∝ W*(W*L)
- 对于按序机器，L与流水线延迟相关
- 对于乱序机器，L还包括指令缓冲器消耗的时间（指令窗口 或 ROB）
- 随着W的增加，需要更大的指令窗口来找都维持机器忙碌所需的指令级并行性
  => L变得更大

=> 乱序控制逻辑的复杂程度增长速度大于 W^2 （~W^3）

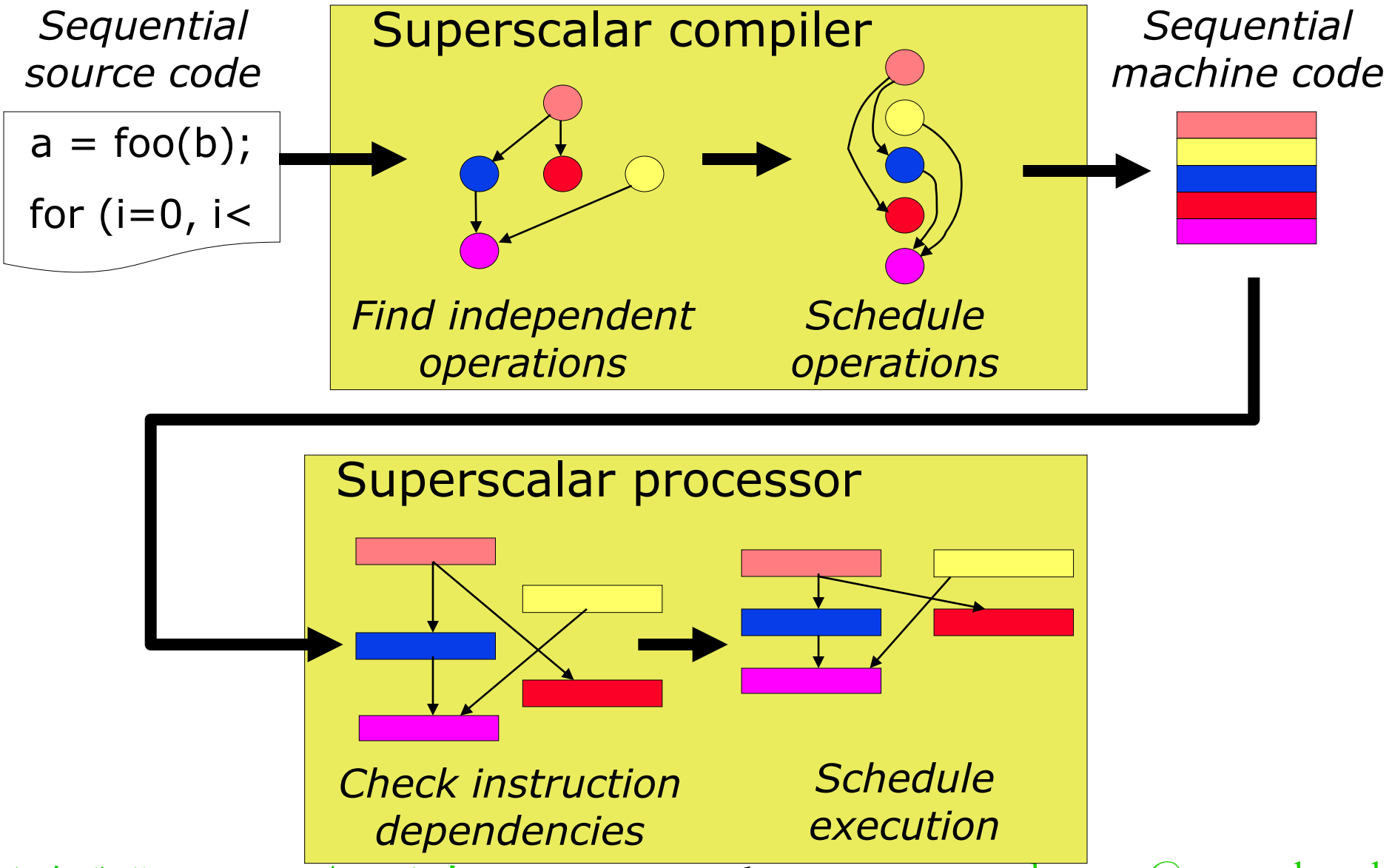
乱序控制的复杂度:MIPS R10000



[SGI/MIPS Technologies Inc., 1995]

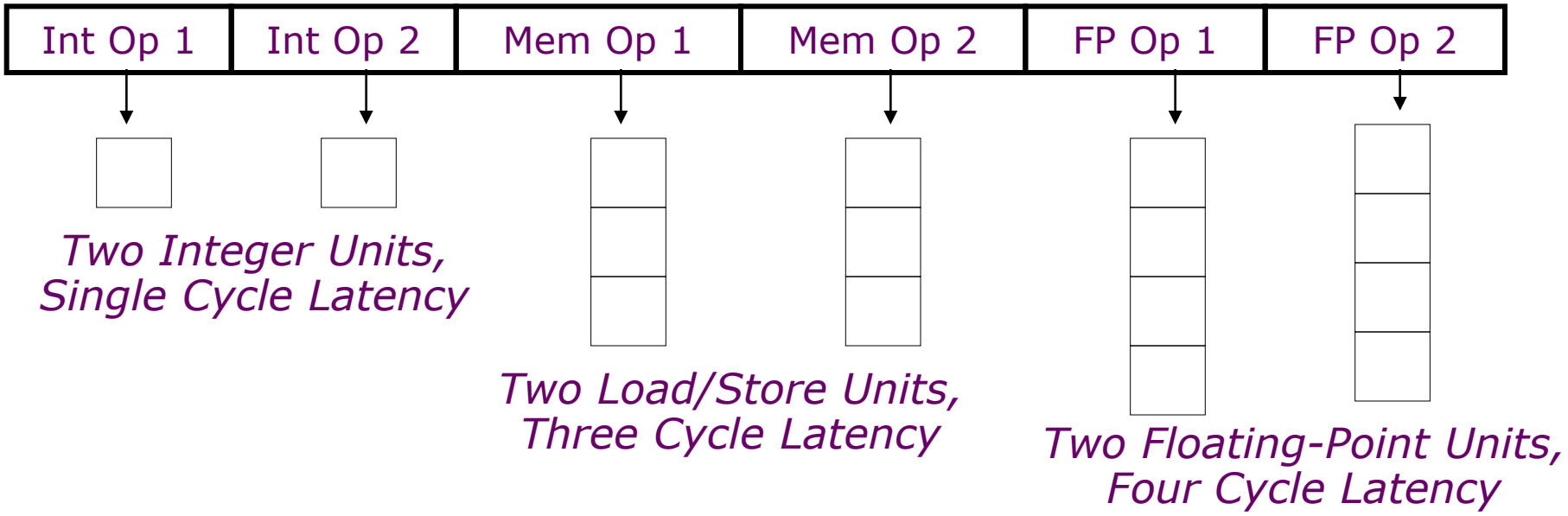
串行ISA的瓶颈

前边讲superscalar的思想：在动态执行过程中，先把串行的汇编代码构造成一个流图，实际上根据流图来执行
但流图的构造过程实际上编译阶段就做过了



VLIW: Very Long Instruction Word

硬件里头有多个并行部件，比如两个浮点部件，好几个定点部件。能不能以编译来保证没有hazard，让那些能同时执行的指令放在同一个超长指令中同时执行



- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction => no x-operation RAW check
  - No data use before data ready => no data interlocks

VLIW 编译器的职责

编译器:

- 调度成最大并行执行
- 确保指令内的并行性
- 避免数据冒险(无互锁)
  - 通常用显式的NOP指令来分隔实际操作

早期VLIW机器

- FPS AP120B (1976)
  - scientific attached array processor
  - first commercial wide instruction machine
  - hand-coded vector math libraries using software pipelining and loop unrolling
- Multiflow Trace (1987)
  - commercialization of ideas from Fisher's Yale group including "trace scheduling"
  - available in configurations with 7, 14, or 28 operations/instruction
  - 28 operations packed into a 1024-bit instruction word
- Cydrome Cydra-5 (1987)
  - 7 operations encoded in 256-bit instruction word
  - rotating register file

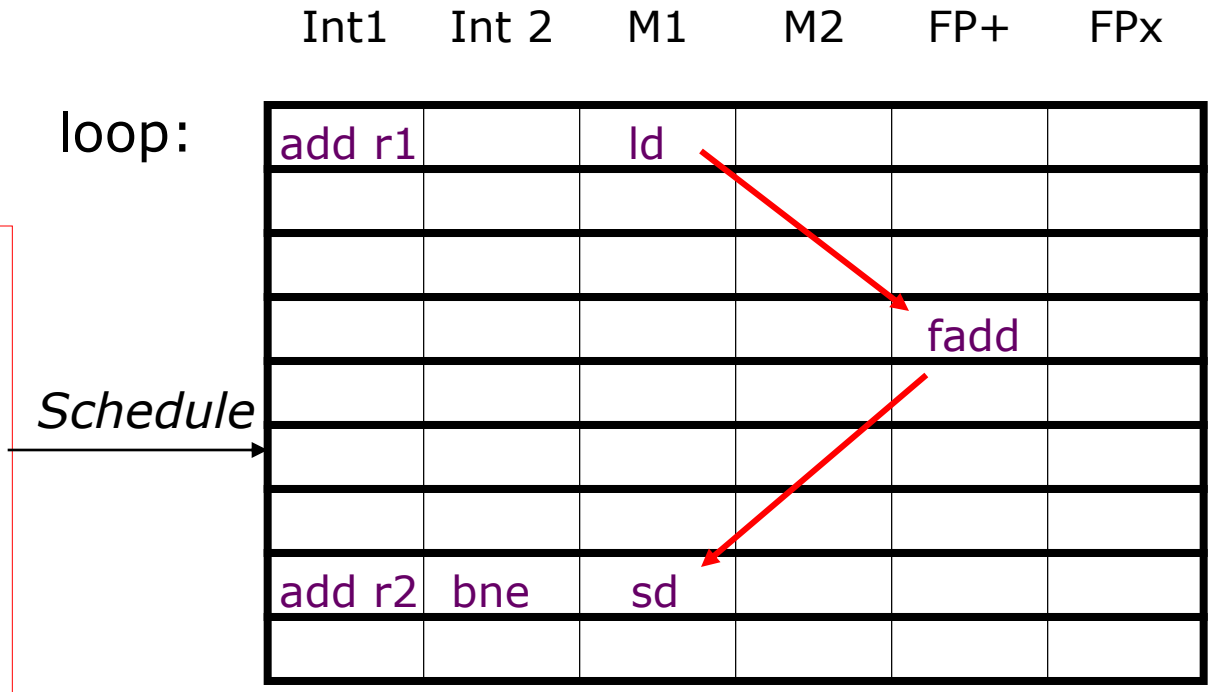
循环的执行

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Compile

```
loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       sd f2, 0(r2)
       add r2, 8
       bne r1, r3, loop
```

Schedule



How many FP ops/cycle?

1 fadd / 8 cycles = 0.125

循环展开（Loop Unrolling）

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Unroll inner loop to perform 4 iterations at once

```
for (i=0; i<N; i+=4)
{
    B[i]   = A[i] + C;
    B[i+1] = A[i+1] + C;
    B[i+2] = A[i+2] + C;
    B[i+3] = A[i+3] + C;
}
```

Need to handle values of N that are not multiples of unrolling factor with final cleanup loop

循环展开后代码的调度

一旦unrolling，就必须要显示寄存器换名，否则就出现太多naming dep，而不能像前边动态隐式换名了

Unroll 4 ways

```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
        add r2, 32
       bne r1, r3, loop
```

Schedule →

	Int1	Int 2	M1	M2	FP+	FPx
loop:			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4		fadd f5	
					fadd f6	
					fadd f7	
					fadd f8	
			sd f5			
			sd f6			
			sd f7			
	add r2	bne	sd f8			

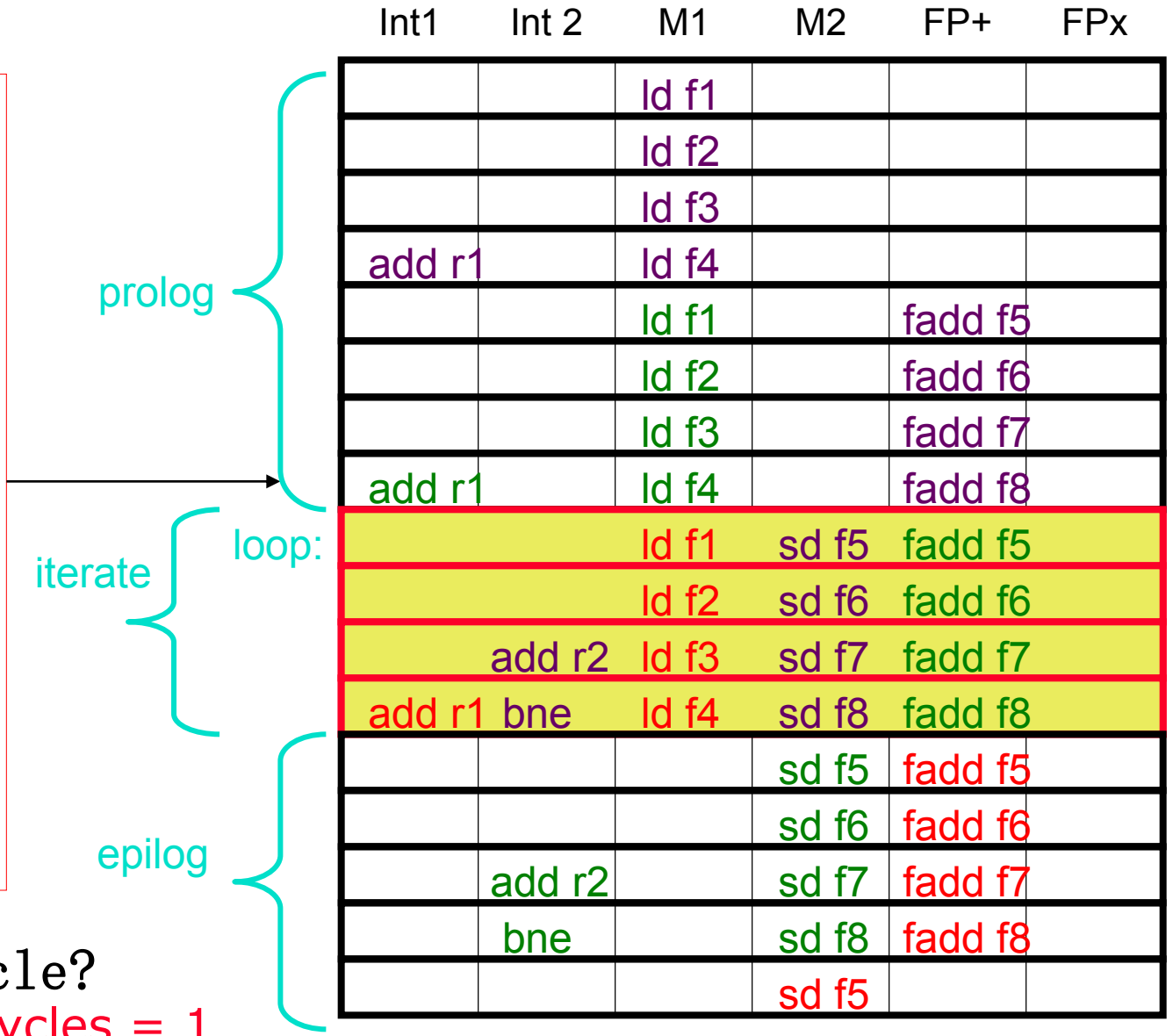
How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36

软件流水（Software Pipelining）

Unroll 4 ways first

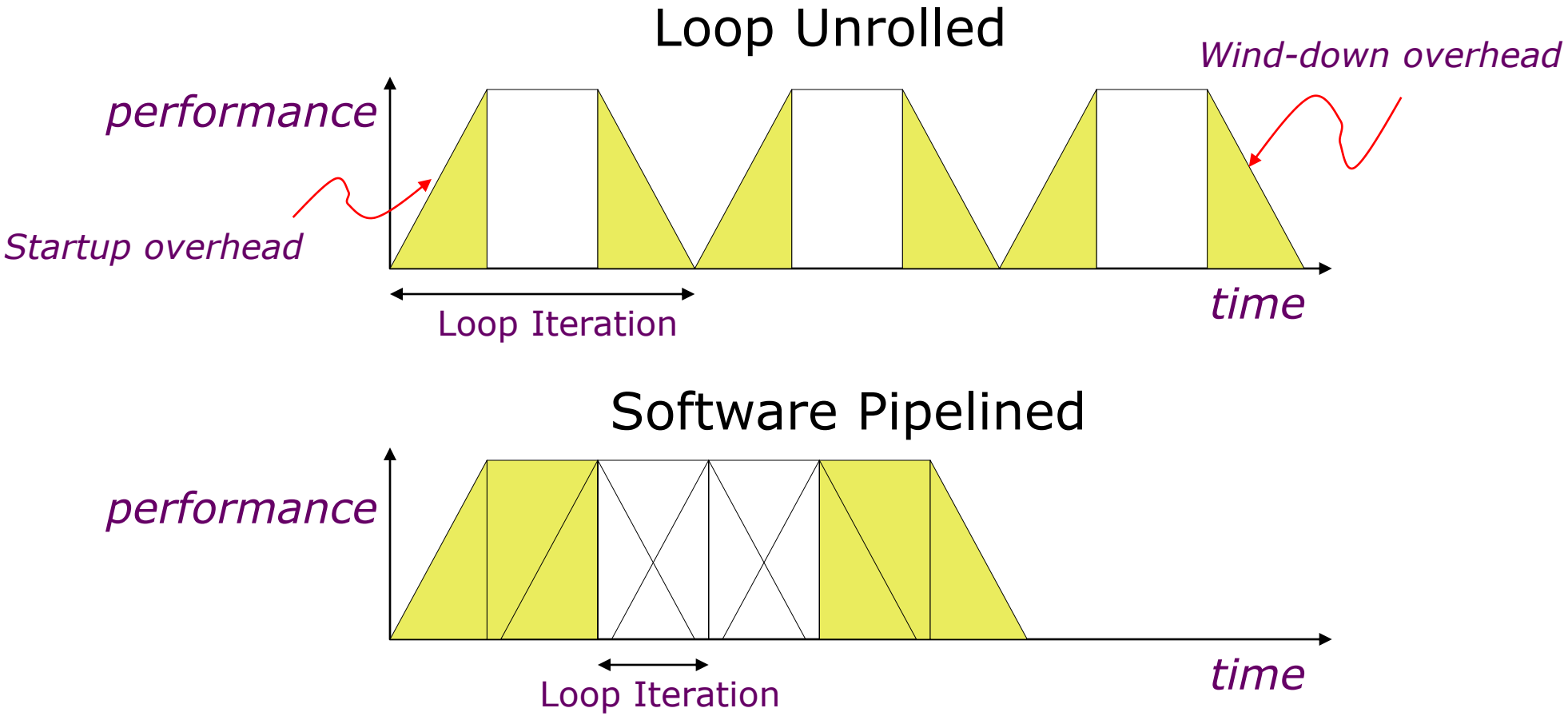
```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       add r2, 32
       sd f8, -8(r2)
       bne r1, r3, loop
```



How many FLOPS/cycle?

4 fadds / 4 cycles = 1

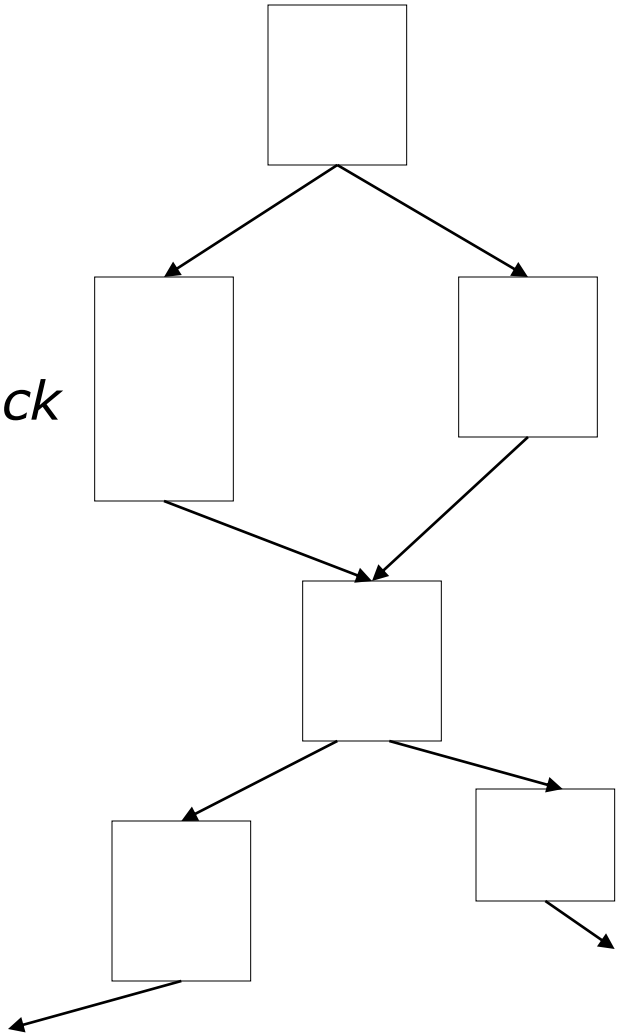
软件流水与循环展开



Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

如果没有循环，怎么办?

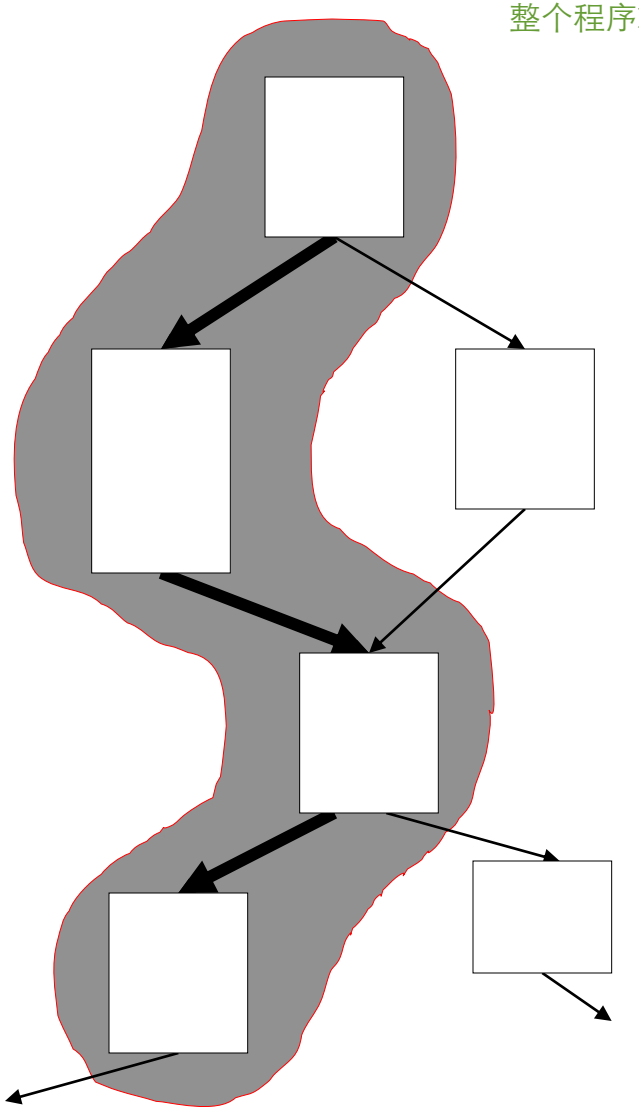
要对通用程序有用，有些程序没有那么多循环



- 转移指令限制了控制流敏感的非规则代码中基本块的大小
- 很难从单独的基本块中发现ILP

踪迹调度（Trace Scheduling） *[Fisher, Ellis]*

通过剖视和静态分析，我能找到一条最经常跑的路径，猜测每次都是跑这条路径
整个程序就可以视为一个没有brunch的basic block，就可以发现足够多的可以调度的指令

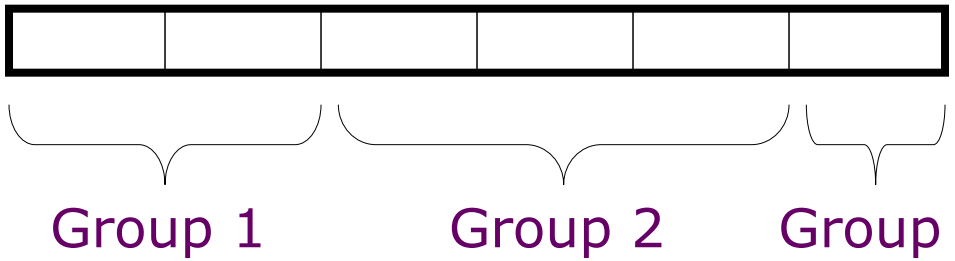


- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole "trace" at once
- Add fixup code to cope with branches jumping out of trace

“典型”VLIW的问题

- 目标代码的兼容性 可执行代码与机器的微结构绑定过死，要把各个功能部件的latency等等暴露给编译器，不同微结构的机器都得需要重新编译才能跑
  - 需要针对每种机器重新编译所有代码，甚至对于同一代中的两种不同机器也需要如此 重新编译一下当然可以，但是有很多程序没有源代码只有二进制代码
- 目标代码的大小 指令太长，有时候不需要同时执行很多操作也得放一个长指令在那里
  - 指令填充浪费指令存取空间（存储器/cache）
  - 循环展开/软件流水将复制代码
- 需要调度访存延迟可变的操作 每次执行load/store要多少latency呢？这是可变的，cache命中不命中很大不同，根据latency要编制不同的指令字，但latency可变你怎么搞
  - caches 和/或 主存块的冲突可能产生静态时刻不可预知的变化
- 确认转移发生概率
  - 动态剖视（Profiling）需要增加巨大的一步来进行静态转移预测
- 对于静态时刻不可预测的转移进行调度 因为其目标是让编译器尽可能多做一些事情嘛
  - 根据转移路径的不同，最佳调度策略也不同

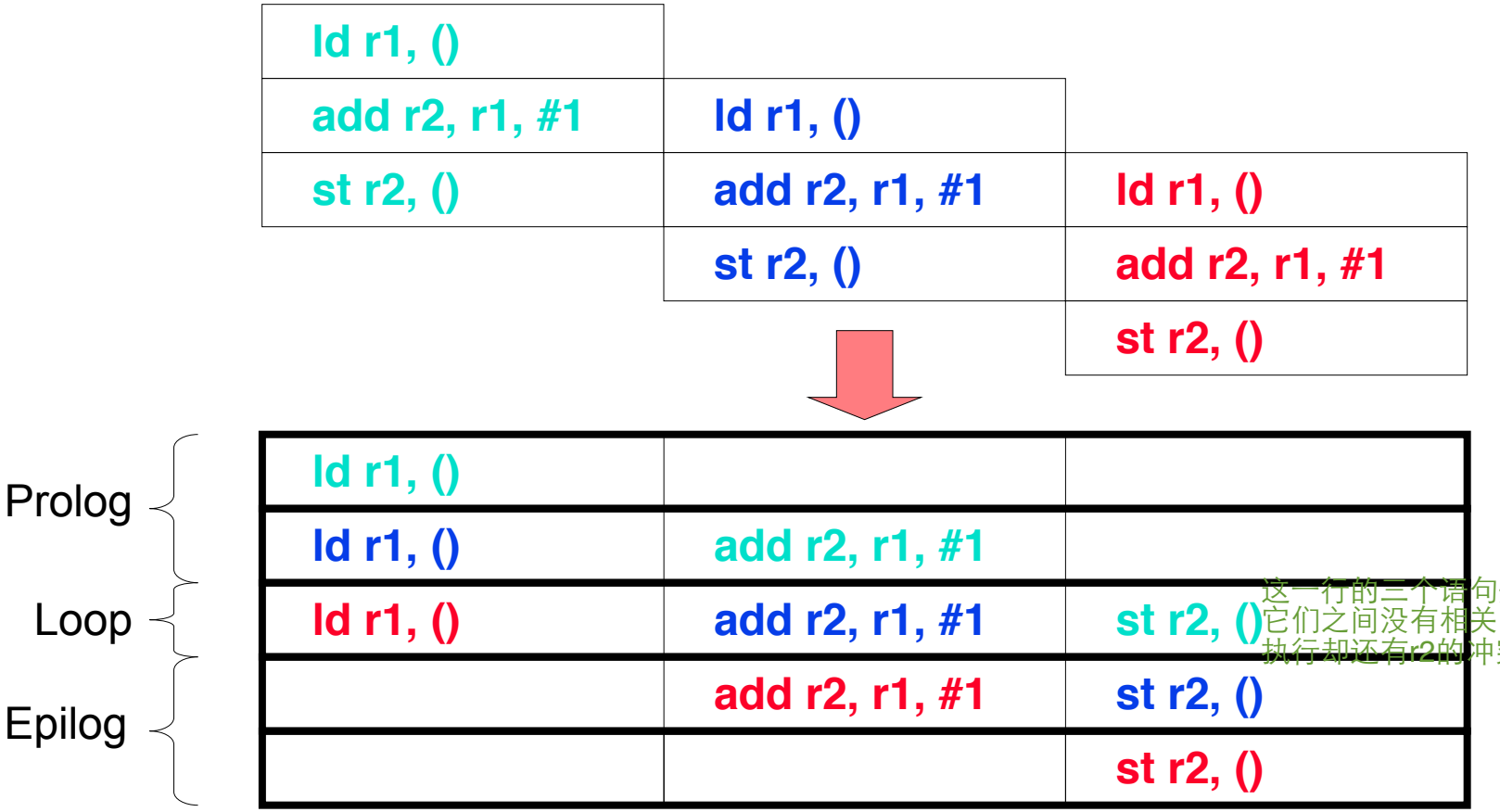
VLIW指令的编码



- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
    - used in Multiflow Trace
    - introduces instruction addressing challenge
  - Mark parallel groups
    - used in TMS320C6x DSPs, Intel IA-64
  - Provide a single-op VLIW instruction
    - Cydra-5 UniOp instructions

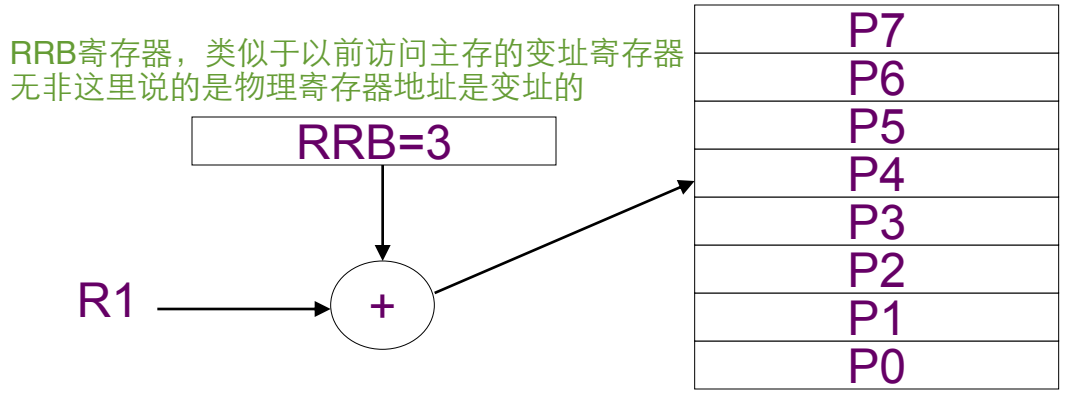
Rotating Register Files

Problems: Scheduled loops require lots of registers,
Lots of duplicated code in prolog, epilog

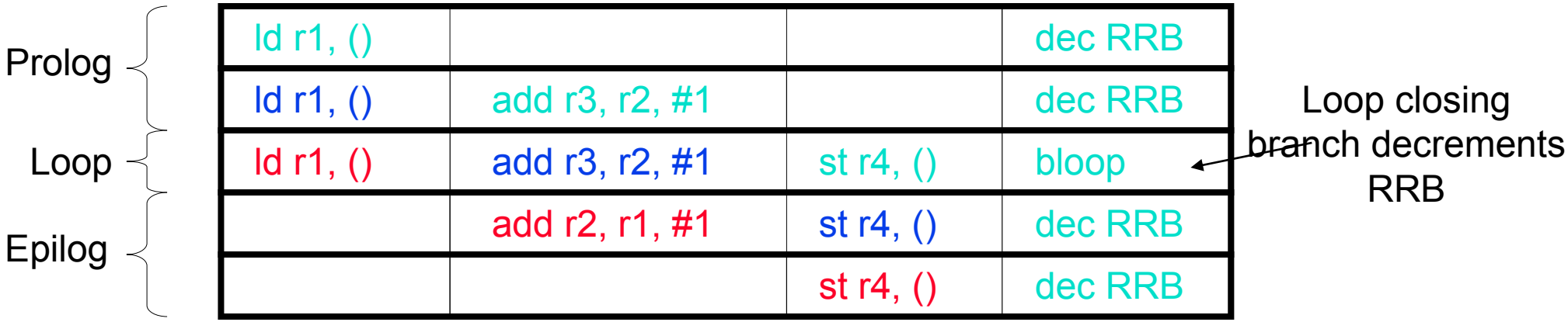


Solution: Allocate new set of registers for each loop iteration

Rotating Register File



Rotating Register Base (RRB) register points to base of current register set. Value added on to logical register specifier to give physical register number. Usually, split into rotating and non-rotating registers.



Rotating Register File
(Previous Loop Example)

Three cycle load latency encoded as difference of 3 in register specifier number (f4 - f1 = 3)

Four cycle fadd latency encoded as difference of 4 in register specifier number (f9 – f5 = 4)

ld f1, ()	fadd f5, f4, ...	sd f9, ()	bloop	这一行是原本的代码
ld P9, ()	fadd P13, P12,	sd P17, ()	bloop	RRB=8
ld P8, ()	fadd P12, P11,	sd P16, ()	bloop	RRB=7
ld P7, ()	fadd P11, P10,	sd P15, ()	bloop	RRB=6
ld P6, ()	fadd P10, P9,	sd P14, ()	bloop	RRB=5
ld P5, ()	fadd P9, P8,	sd P13, ()	bloop	RRB=4
ld P4, ()	fadd P8, P7,	sd P12, ()	bloop	RRB=3
ld P3, ()	fadd P7, P6,	sd P11, ()	bloop	RRB=2
ld P2, ()	fadd P6, P5,	sd P10, ()	bloop	RRB=1

这些指令每次执行时再也没有WAW、WAR了

每做一次RRB自减1

实际执行时动态第重定向到不同物理寄存器

Cydra-5:
存储延迟寄存器（Memory Latency Register：MLR）

问题：装入操作可能出现时延变化

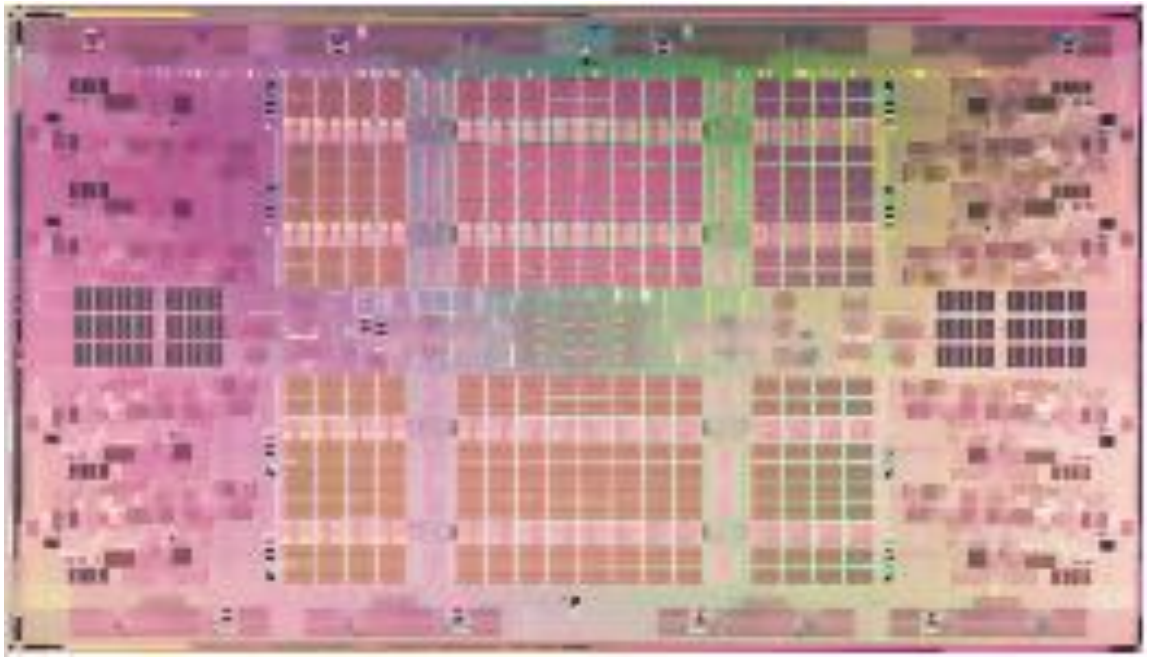
解决方案：让软件选择所需时延

- 编译对代码进行调度以满足最大装入-使用（load-use）距离
- 软件将MLR设置为机器代码调度的延迟
- 硬件保证装入操作在MLR要求的周期将所需数值返回给处理器流水线
  - 硬件将缓存提前返回的装入数值
  - 如果装入没有按时返回数值，硬件将暂停处理器

Intel Itanium, EPIC IA-64

- 与传统CISC和RISC相比，EPIC是一种新型的体系结构
  显式并行指令计算
  这个就是EPIC的缩写，EPIC允许处理器根据编译器的调度并行执行而不用增加硬件复杂性
  该架构由超长指令字架构发展而来，做了大量改进。
  - Explicitly Parallel Instruction Computing
- IA-64是Intel采用的一种不用于x86的ISA
  - IA-64 = Intel Architecture 64-bit
  - 一种目标代码兼容的VLIW
- Itanium（最早称为Merced)是采用IA-64的第一种处理器
  - 1995年预计第一种产品在1997问世（实际上为2001）
  - McKinley是2002年推出的第二类产品
  - 2011年发布的最新版本Poulson, 采用32nm工艺, 集成八核

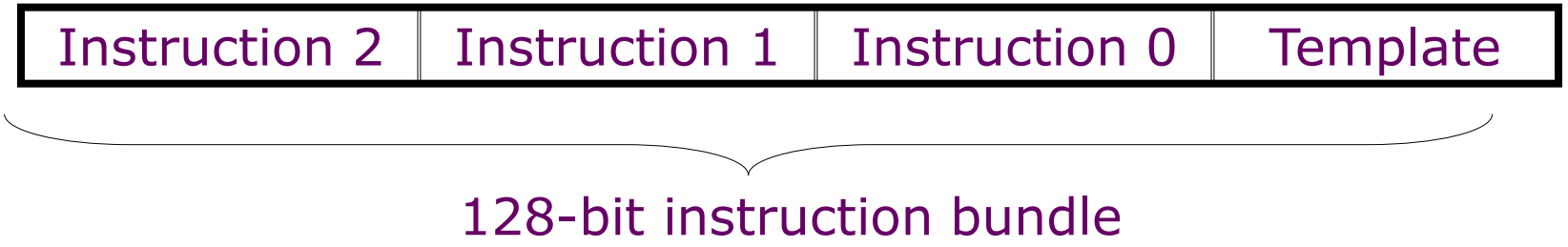
Eight Core Itanium "Poulson" *[Intel 2011]*



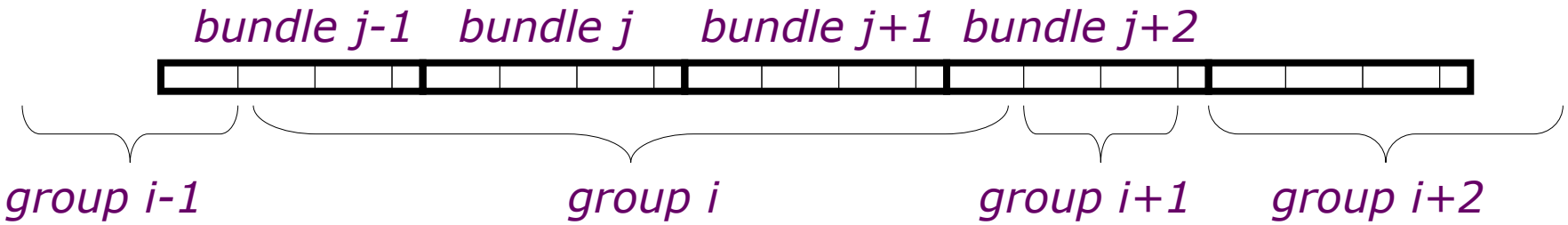
- 8 cores
- 1-cycle 16KB L1 I&D caches
- 9-cycle 512KB L2 I-cache
- 8-cycle 256KB L2 D-cache
- 32 MB shared L3 cache
- 544mm^2 in 32nm CMOS
- Over 3 billion transistors
- Cores are 2-way multithreaded
- 6 instruction/cycle fetch
  - Two 128-bit bundles
- Up to 12 insts/cycle execute

IA-64 的指令格式

任何一条指令给128位，其中几位为template来指示前边几个指令能怎样地并行，第一条和第二条能否并行等等，以及指示这128位和后边的128位能否并行。这就是所谓“显式并行”



- Template bits describe grouping of these instructions with others in adjacent bundles
- Each group contains instructions that can execute in parallel



IA-64中的寄存器

- 128个通用64位定点寄存器
- 128个通用64/80位浮点寄存器
- 64个1位预测寄存器

- GPR“旋转（rotate）”以减小软件流水化循环的代码大小
  - 旋转（rotate）是支持寄存器换名的一种简化模式，它可以使同一指令在循环的每次不同迭代过程访问不同的物理寄存器

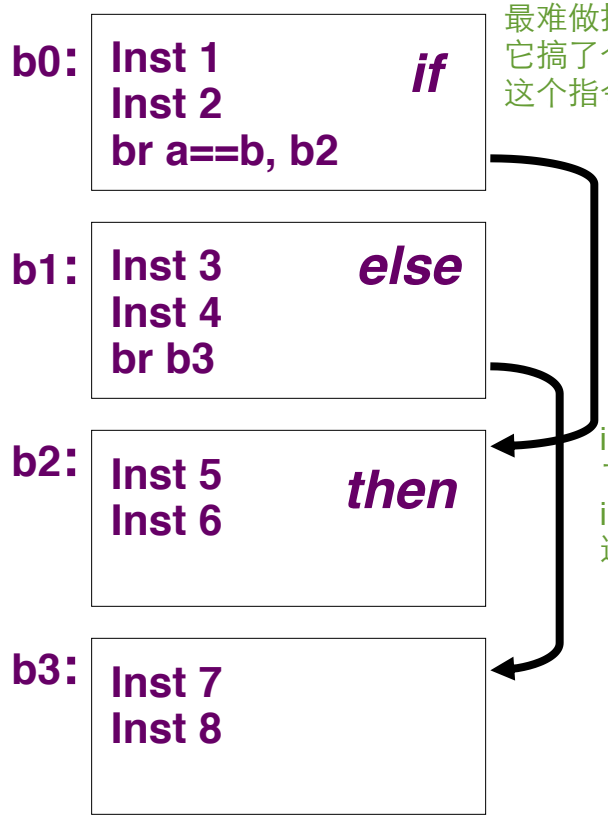
IA-64 Predicated Execution

Problem: Mispredicted branches limit ILP

Solution: Eliminate hard to predict branches with predicated execution

- Almost all IA-64 instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false

最难做指令调度的是brunch，如果能把控制相关转移成data dep，就可以在全局做指令调度
它搞了个conditional instruction。对于任何一个指令都有一个predictor，如果p为true，就执行这个指令，如果条件p为false，就执行其它的嘛



Four basic blocks

Predication

inst3和inst5就可以同时执行了，判断p1和p2的真假然后决定inst3和inst5的结果谁保留
这样整个程序就没有控制相关了

```
Inst 1
Inst 2
p1,p2 <- cmp(a==b)
(p1) Inst 3    ||   (p2) Inst 5
(p1) Inst 4    ||   (p2) Inst 6
Inst 7
Inst 8
```

p1为true就执行inst3

One basic block

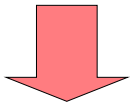
Mahlke et al, ISCA95: On average >50% branches removed

Predicate Software Pipeline Stages

Single VLIW Instruction

(p1) ld r1	(p2) add r3	(p3) st r4	(p1) bloop

这样就可以同时执行很多指令



Dynamic Execution

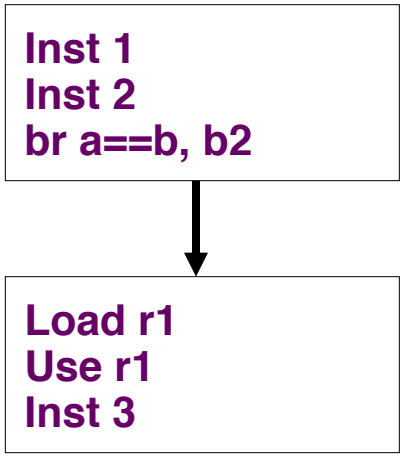
(p1) ld r1			(p1) bloop
(p1) ld r1	(p2) add r3		(p1) bloop
(p1) ld r1	(p2) add r3	(p3) st r4	(p1) bloop
(p1) ld r1	(p2) add r3	(p3) st r4	(p1) bloop
(p1) ld r1	(p2) add r3	(p3) st r4	(p1) bloop
	(p2) add r3	(p3) st r4	(p1) bloop
		(p3) st r4	(p1) bloop

Software pipeline stages turned on by rotating predicate registers ➔ Much denser encoding of loops

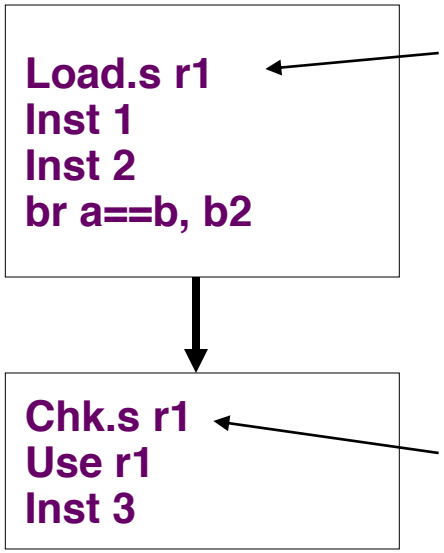
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don't cause exceptions



Can't move load above branch because might cause spurious exception



Speculative load never causes exception, but sets "poison" bit on destination register

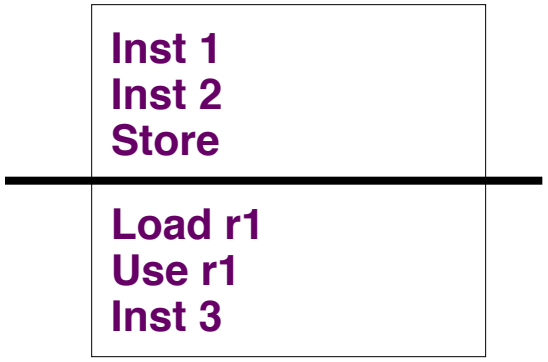
Check for exception in original home block jumps to fixup code if exception detected

跟踪那个地址
在这一地方检查有没有任何操作改过存储器这个地址的内容
只要没有任何东西改过这个地址

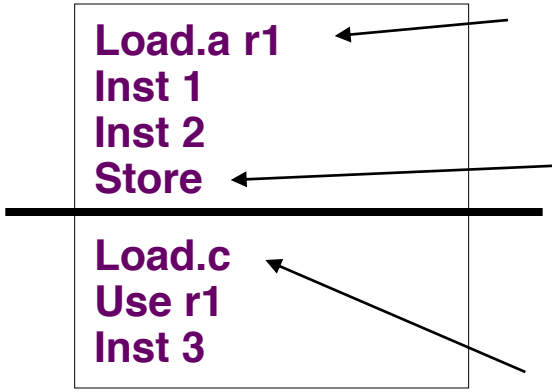
Particularly useful for scheduling long latency loads early

IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling
Solution: Hardware to check pointer hazards



Can't move load above store because store might be to same address



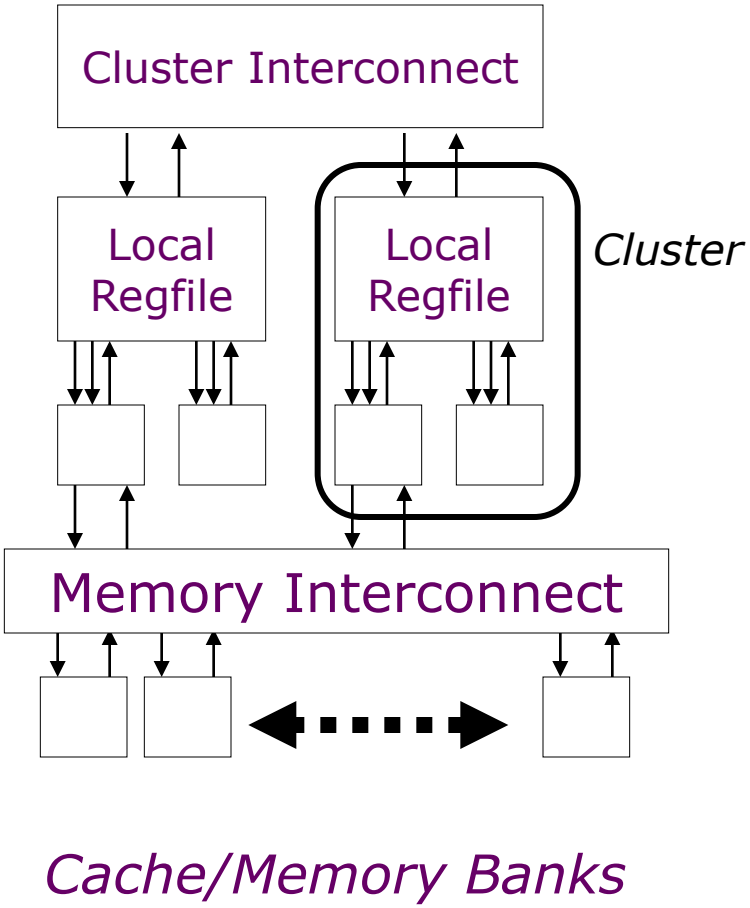
Data speculative load adds address to address check table

Store invalidates any matching loads in address check table

Check if load invalid (or missing), jump to fixup code if so

Requires associative hardware in address check table

集群化VLIW（Clustered VLIW）



- 将机器划分成具有局部寄存器堆和局部功能部件的集群
- 集群间采用低带宽/高时延的互联
- 由软件负责进行计算划分，以实现最小化通讯开销（communication overhead）
- 在商用嵌入式VLIW处理器中经常使用，例如，TI C6x DSPs，HP Lx处理器
- (在一些超标量处理器中也使用该技术，例如，Alpha 21264)

静态调度的制约

- 不可预知的转移
- 可变存储时延(不可预知的cache失效)
- 代码大小爆炸（Code size explosion）
- 编译的复杂性

问题:

对于传统RISC/CISC处理器，VLIW能够激发哪些可采用的技术?