VLIW/EPIC: 静态调度的指令级并行

Krste Asanovic

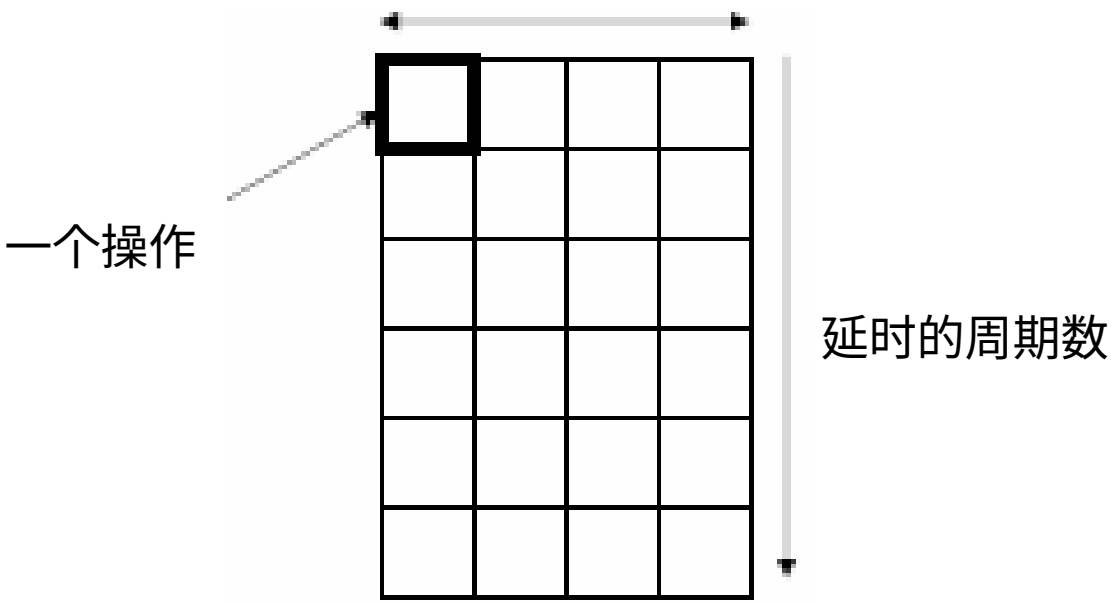
麻省理工学院

计算机科学实验室

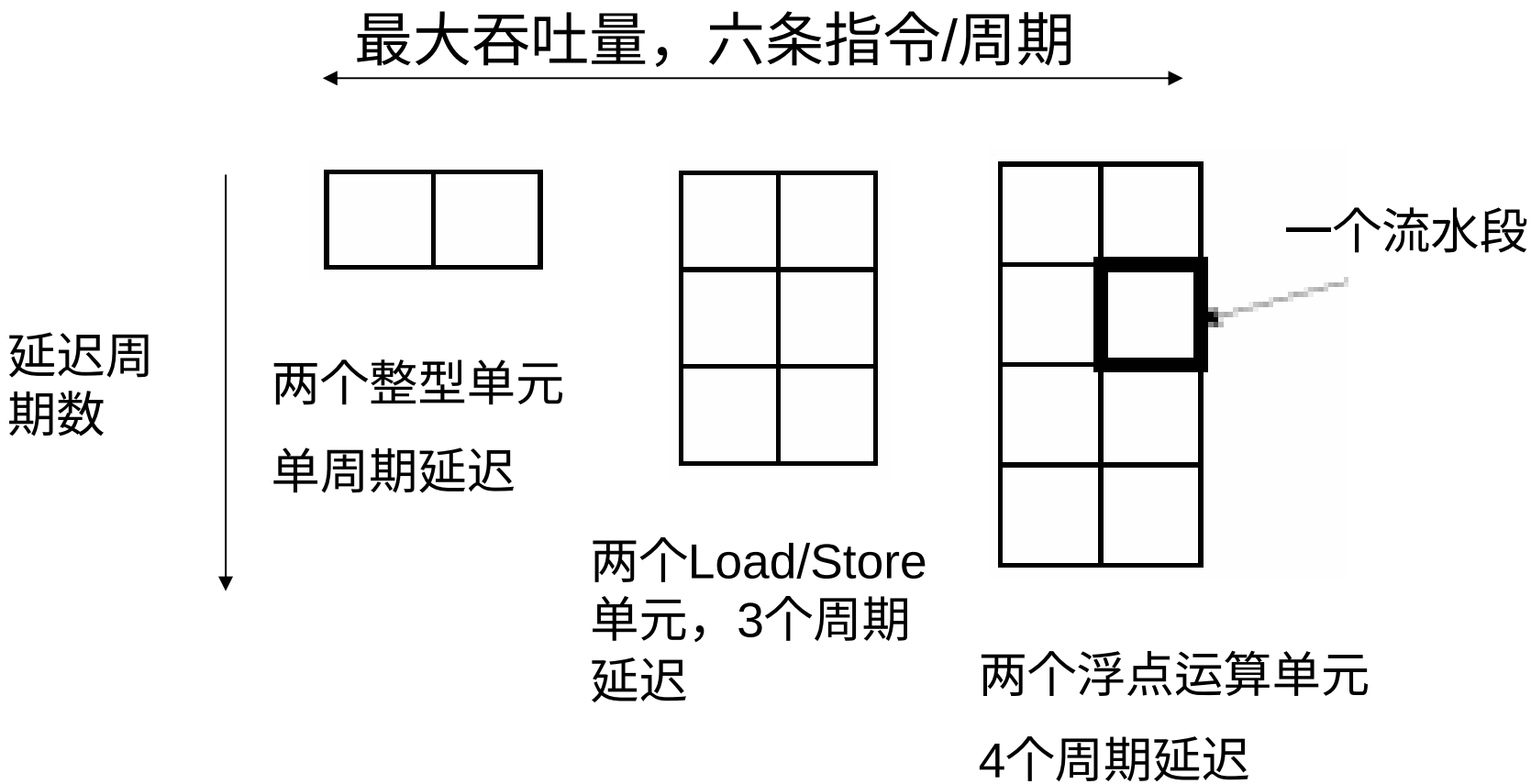
Little定律

并行度＝吞吐率×运行时间

每个周期的吞吐量

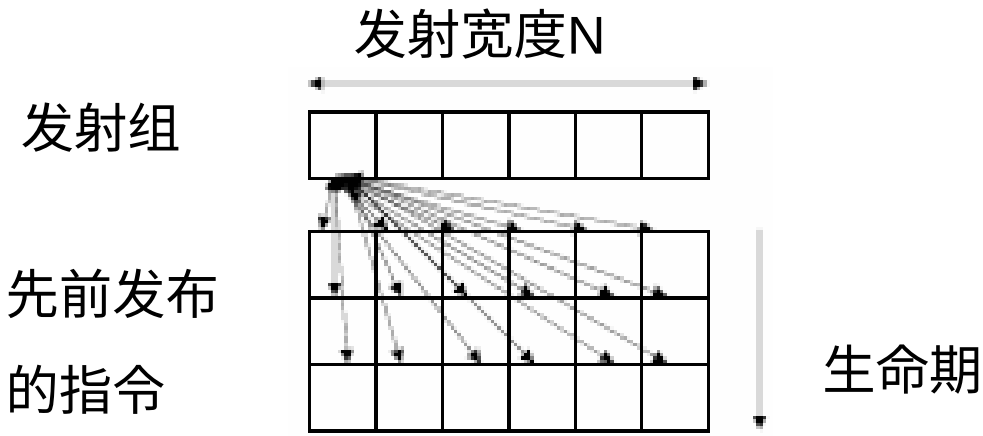


流水线ILP计算机例子



为使流水线保持繁忙，指令级并行度（ILP）需要是多少？

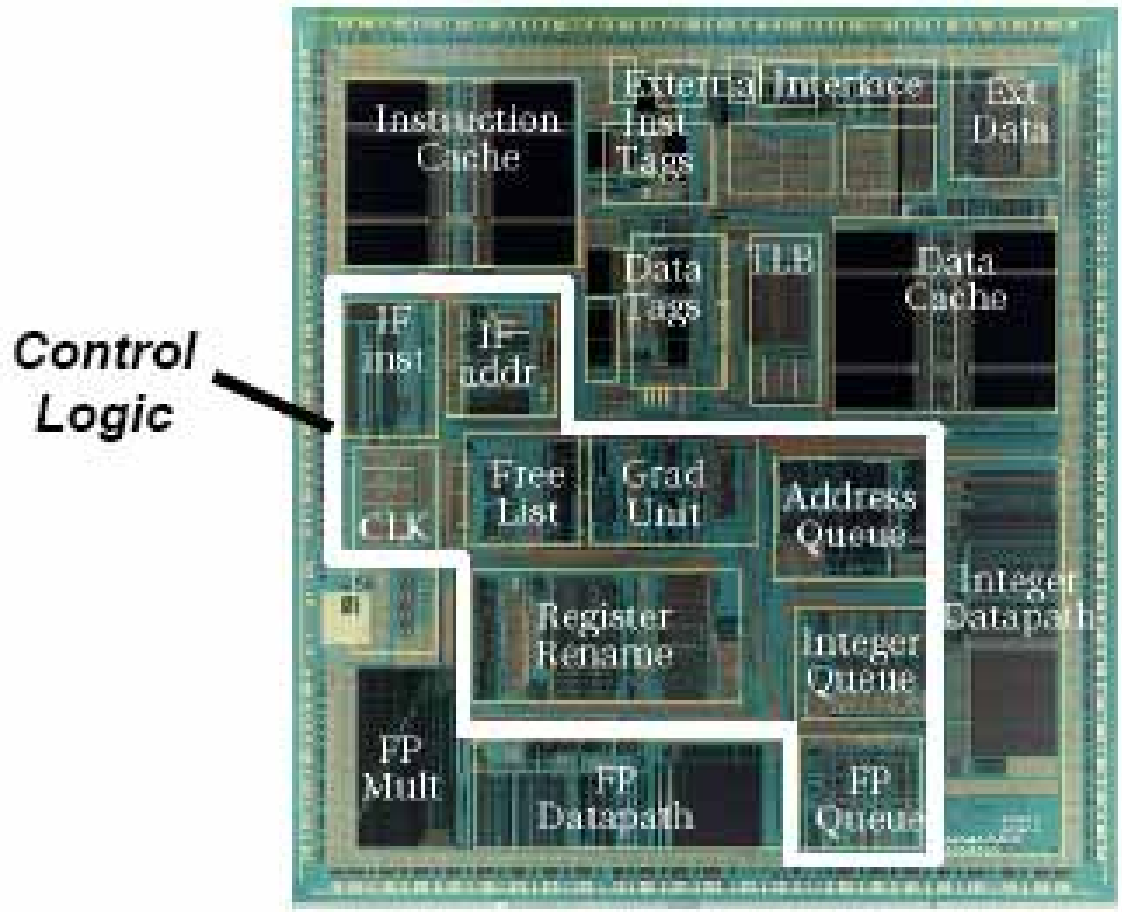
超标量控制逻辑比例变化



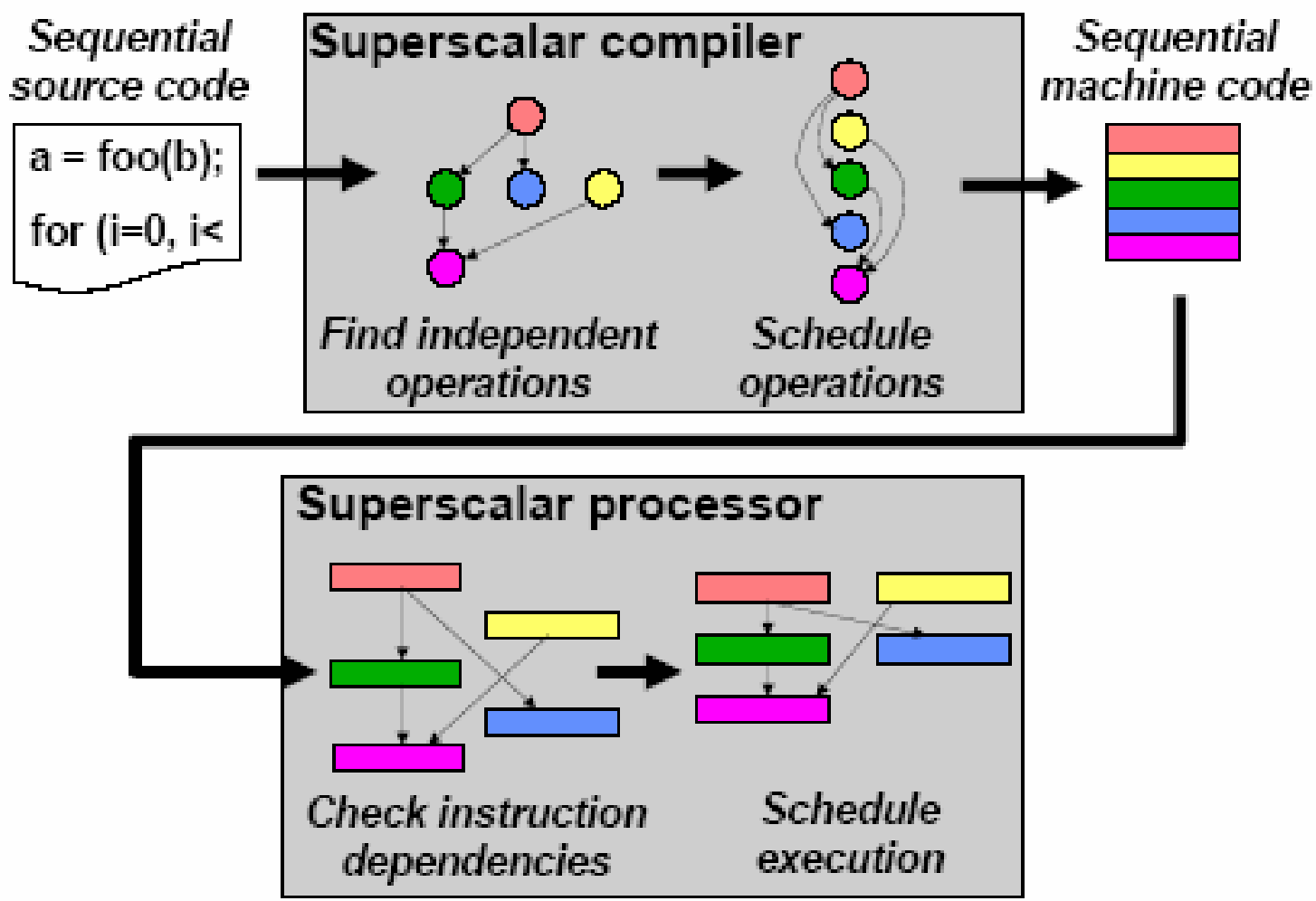
- 互锁检查和标签比较的数目以N*(N *L)增长，L是机器中指令的生命期
  - －每发布或执行完N条指令中的一个就必须检查N *L条运行中的指令
- 对于按序发射的计算机，生命期L与流水时延有关
- 对于乱序发射的计算机，L还包括花在指令缓冲区里的时间（指令窗或ROB）
- 随着N增加，需要更大的指令窗以获得足够的并行度使机器保持繁忙＝>更长的生命期L
- ＝>乱序控制逻辑增长快于N^2(~N^3)

乱序控制的复杂性

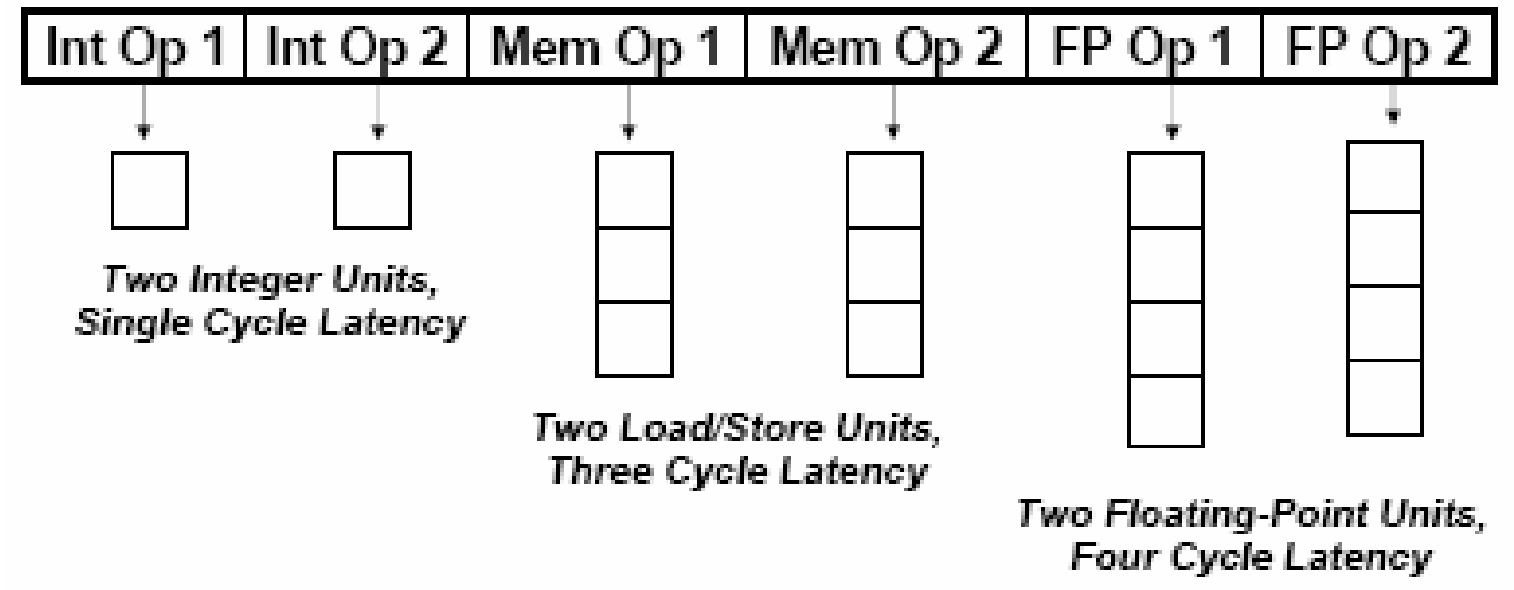
MIPS R10000



顺序ISA的瓶颈


Sequential source code
a = foo(b);
for (i=0, i<
Superscalar compiler
Find independent operations
Schedule operations
Sequential machine code
Superscalar processor
Check instruction dependencies
Schedule execution

VLIW:超长指令字

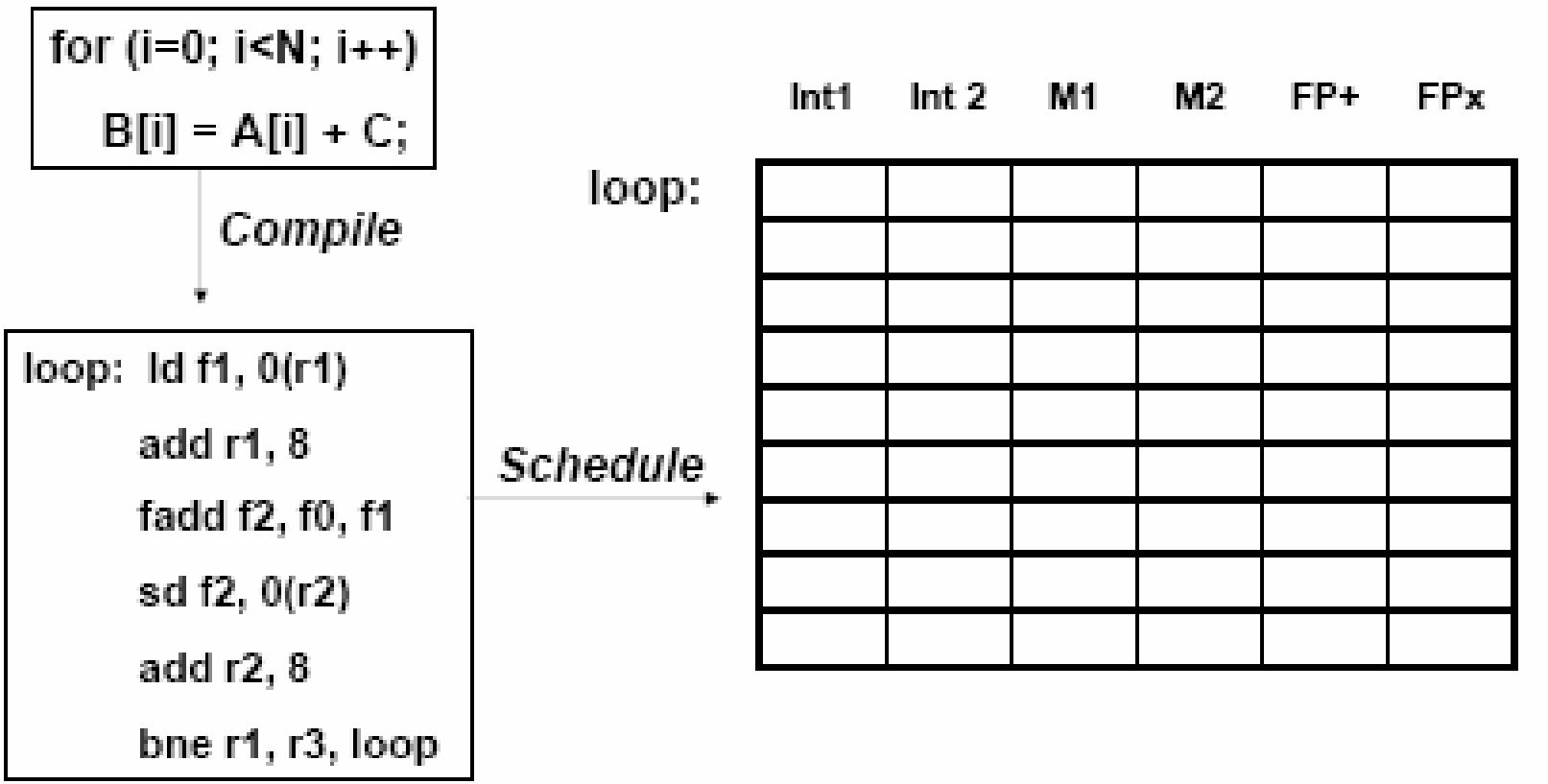


- 编译器调度并行执行
- 多个并行操作压缩到一个长指令字中
- 编译器必须避免数据冲突（没有互锁）

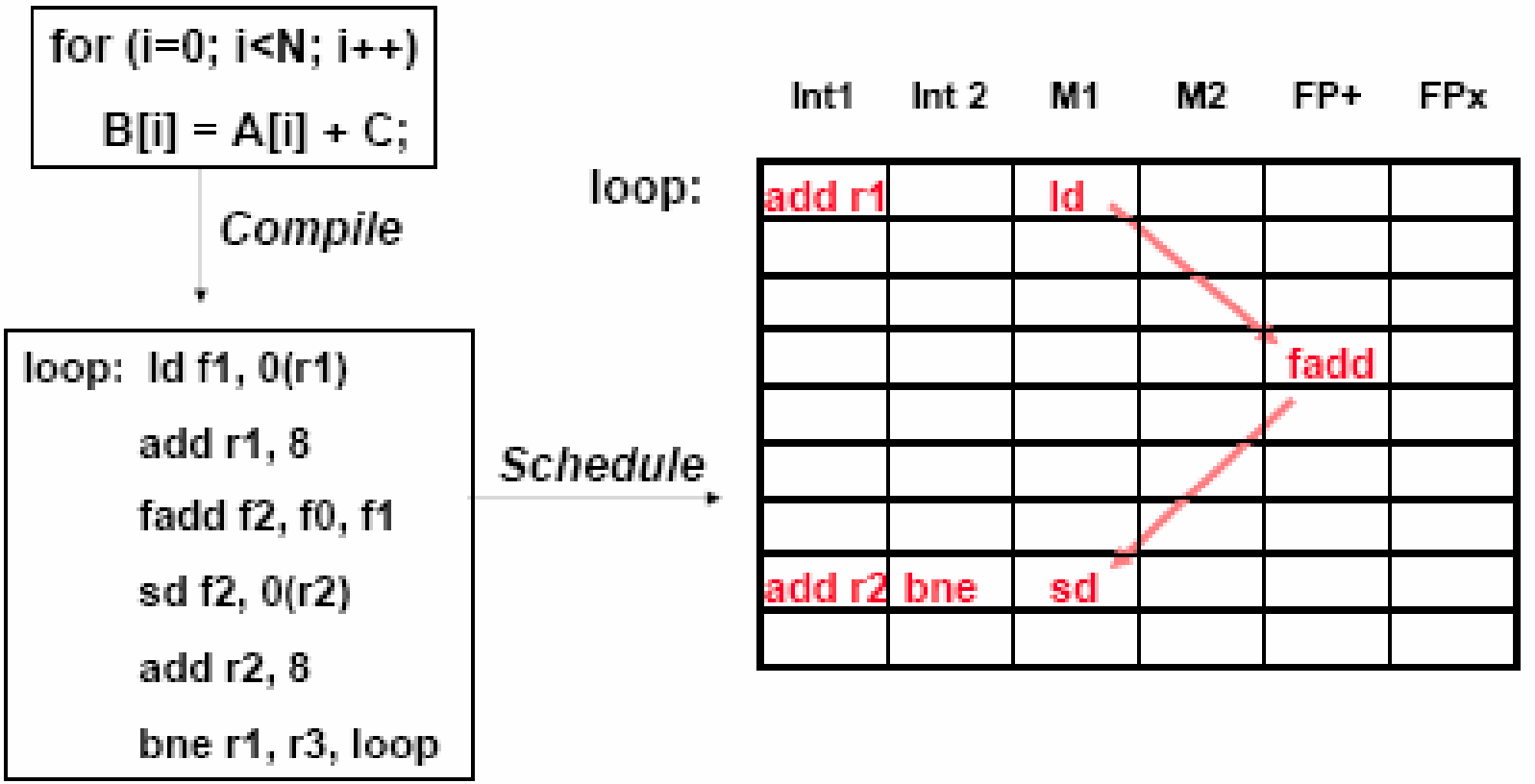
早期的VLIW机

- FPS AP120B(1976)
  - 科学计算阵列协处理器
  - 第一台商业宽指令机
  - 手工编码的向量算术库，使用软件流水和循环展开技术
- 多流追踪（1987）
  - 来自fisher的耶鲁小组的思想，包括迹调度，商业化了
    - 每条指令配置7，14，28个操作均是有效的
    - 28个操作紧缩成一个1024位的指令字
- Cydrome Cydra-5(1987)
  - 7个操作编码为256位指令字
  - 循环寄存器模式

循环执行


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
Int1
Int 2
M1
M2
FP+
FPx
loop:

循环执行



- 多少个浮点操作/周期？

1 fadd / 8 cycles = 0.125

展开循环

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

```
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;
}
```

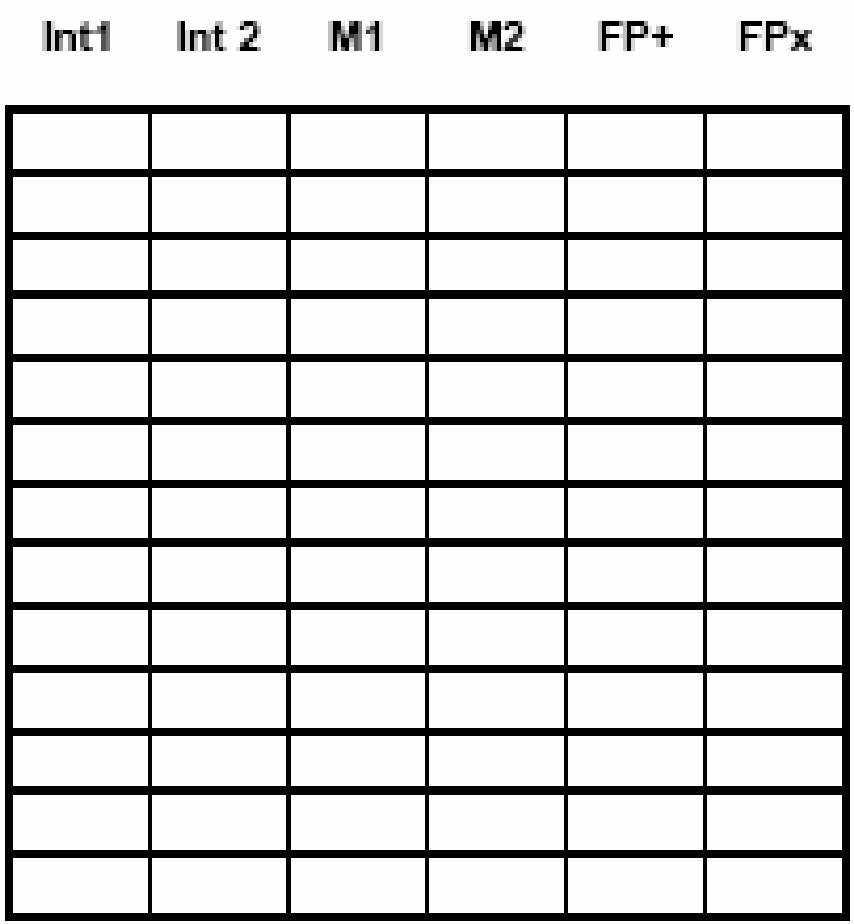
打开内循环，一次执行4个迭代

- 需要处理N的值，它不是最终清除循环的展开因子的倍数，

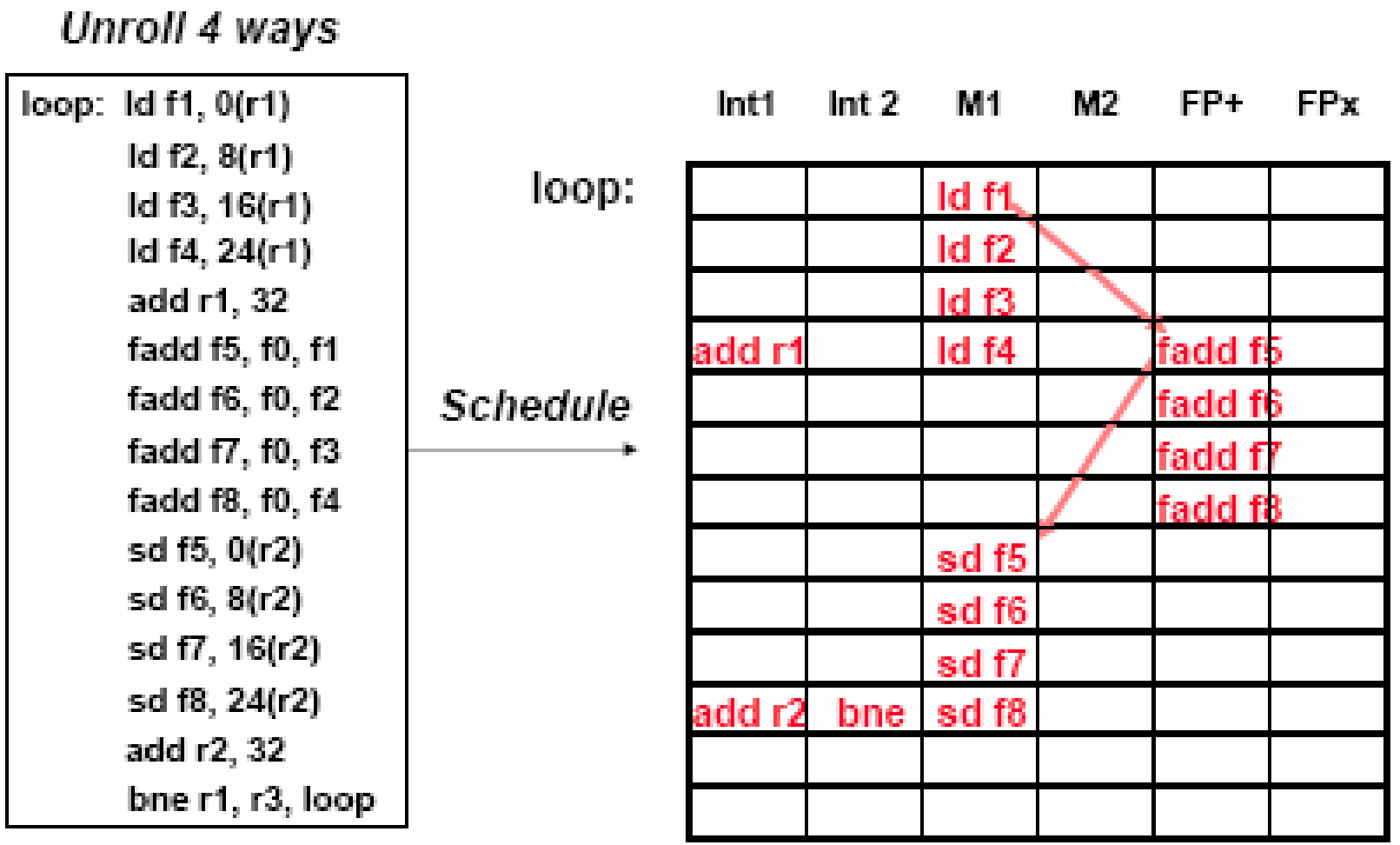
展开循环代码的调度

```
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 →



展开循环代码的调度

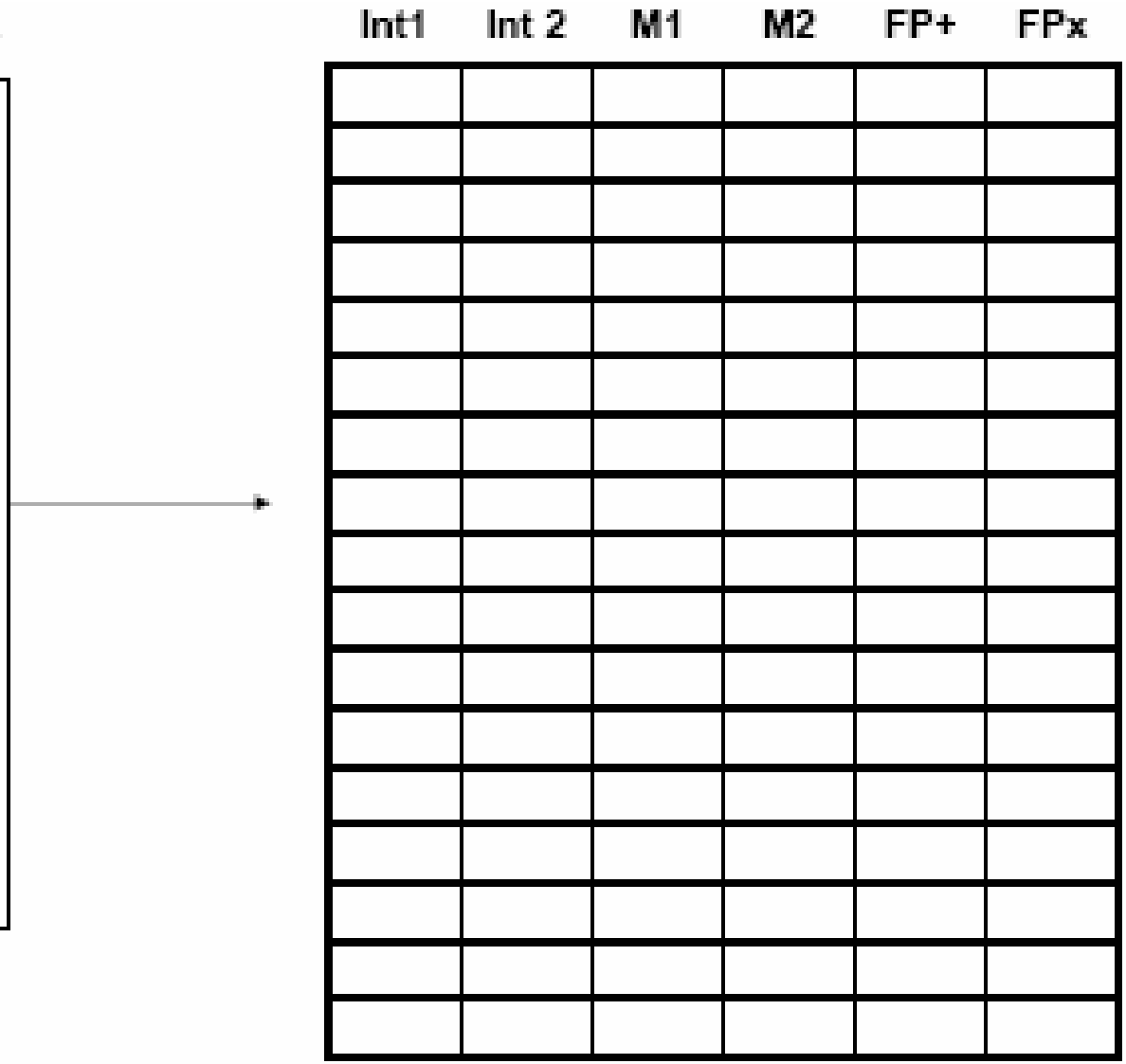


- 多少FLOPS/周期？

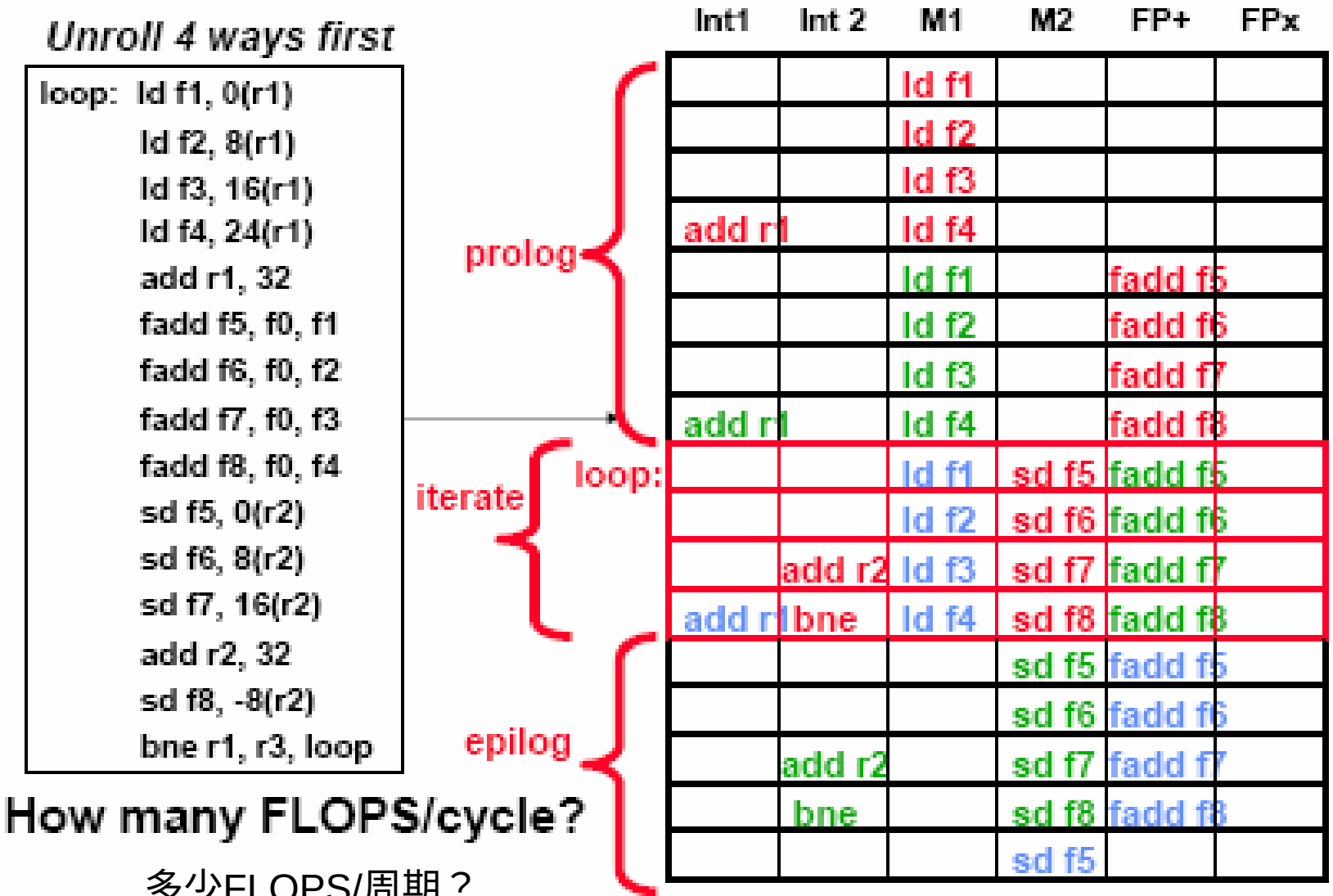
软件流水

Unroll 4 ways first

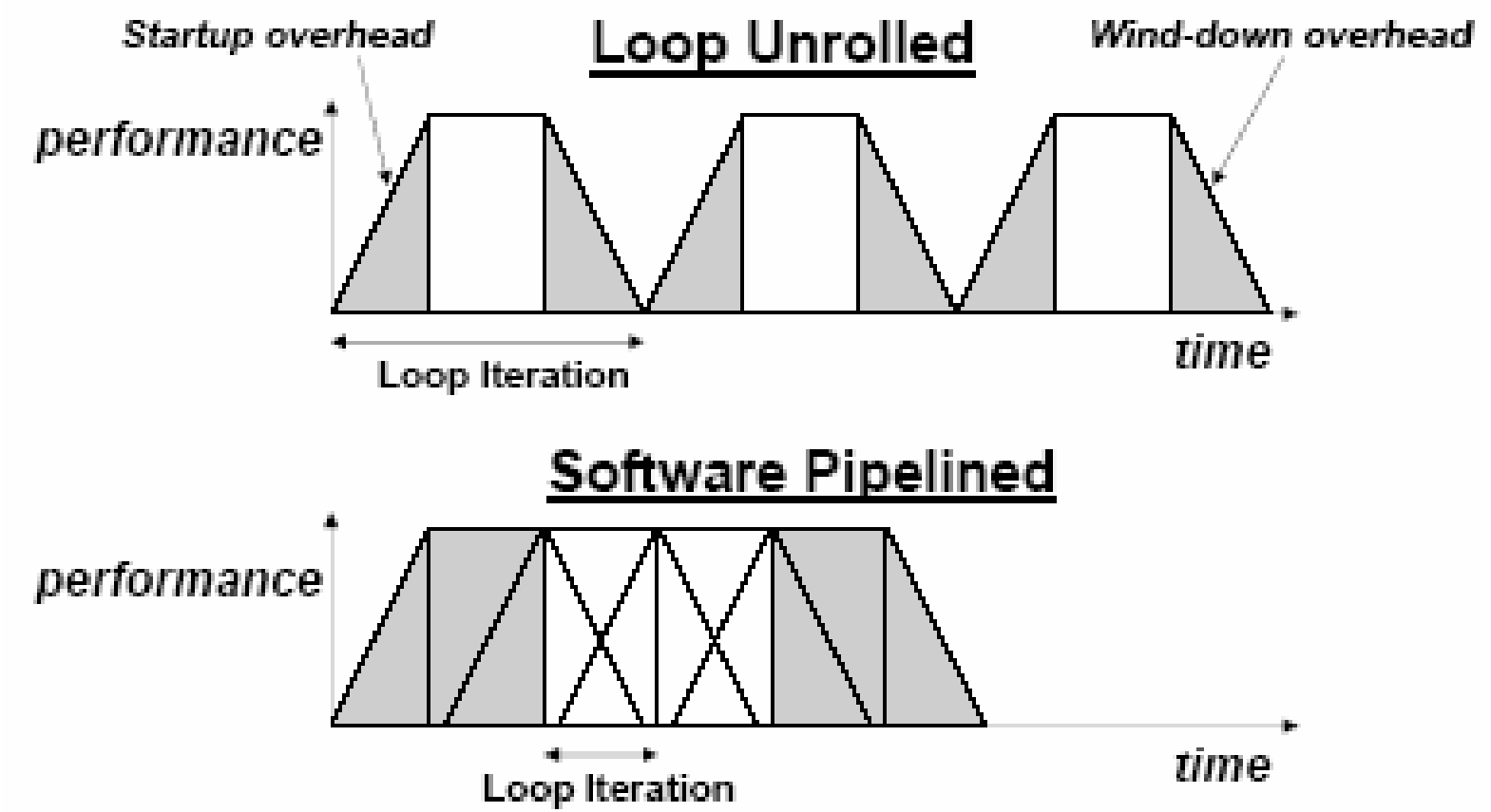
```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       add r2, 32
       sd f8, -8(r2)
    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



软件流水

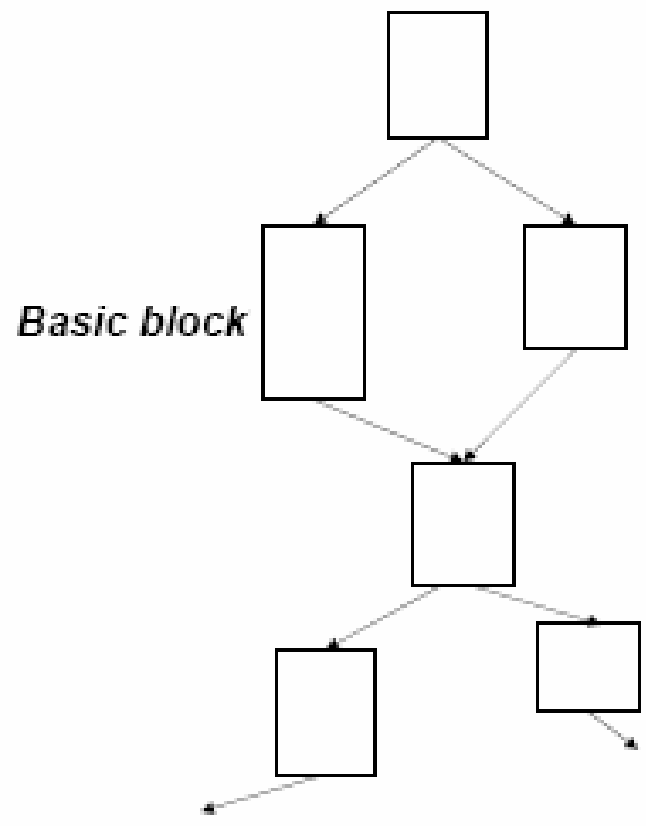

Unroll 4 ways first
loop: ld f1, 0(r1)
ld f2, 8(r1)
ld f3, 16(r1)
ld f4, 24(r1)
add r1, 32
fadd f5, f0, f1
fadd f6, f0, f2
fadd f7, f0, f3
fadd f8, f0, f4
sd f5, 0(r2)
sd f6, 8(r2)
sd f7, 16(r2)
add r2, 32
sd f8, -8(r2)
bne r1, r3, loop
How many FLOPS/cycle?
多少FLOPS/周期？
prolog
iterate
epilog
loop:
Int1
Int 2
M1
M2
FP+
FPx
ld f1
ld f2
ld f3
add r1
ld f4
ld f1
fadd f5
ld f2
fadd f6
ld f3
fadd f7
add r1
ld f4
fadd f8
ld f1
sd f5
fadd f5
ld f2
sd f6
fadd f6
add r2
ld f3
sd f7
fadd f7
add r1
bne
ld f4
sd f8
fadd f8
sd f5
fadd f5
sd f6
fadd f6
add r2
sd f7
fadd f7
bne
sd f8
fadd f8
sd f5

软件流水与循环展开



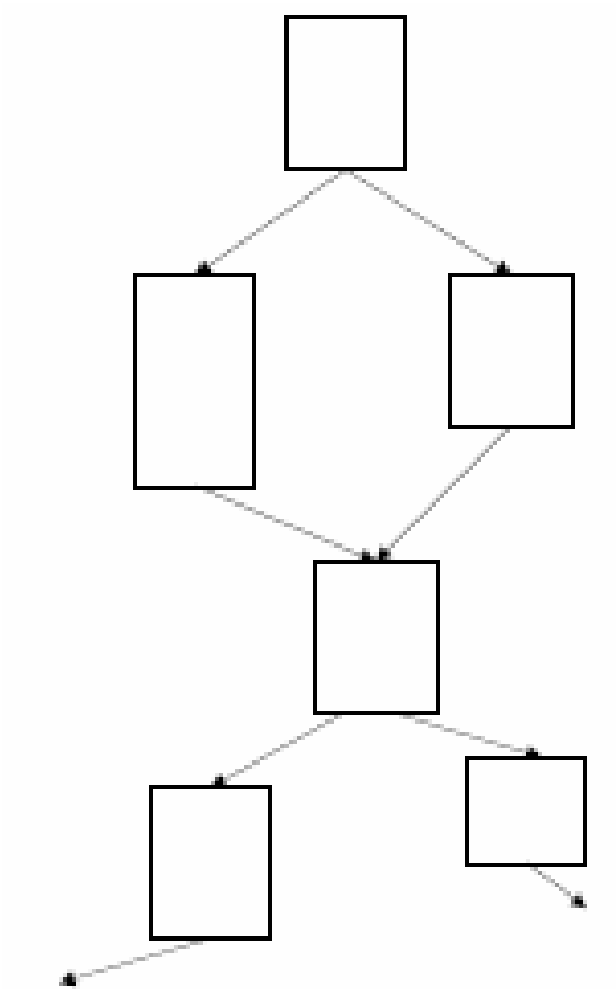
- 软件流水每个循环花 一次启动/停止的代价，而不是每次迭代开销一次

如果没有循环的话怎么样？

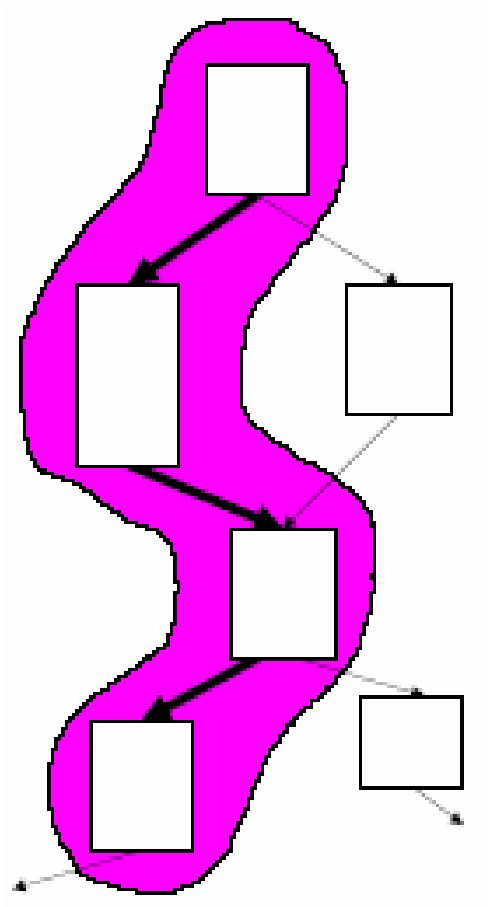


- 在控制流非常没有规律的代码中，转移限制了基本块的大小
- 在单独的基本块里很难找到ILP

迹调度*[Fisher, Ellis]*



迹调度*[Fisher, Ellis]*



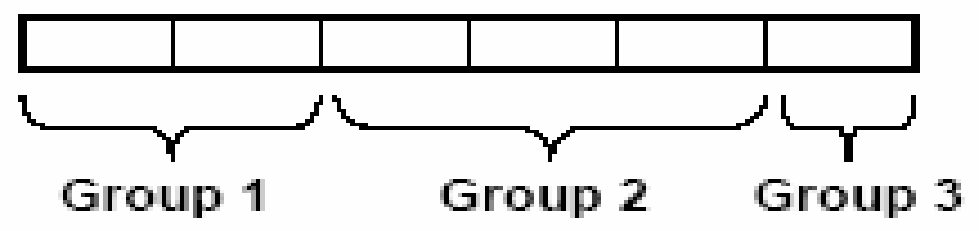
- 选择一串基本块，一个轨迹，表示最频繁转移的路径
- 使用概貌反馈，或编译启发，寻找共同的转移路径
- 一次调度整个“迹”
- 加入固定的代码来处理跳出迹的转移

典型VLIW的问题

- 目标代码的兼容性
  - 必须对每台机器重新编译所有的代码，即使是两台同一代的机器
- 目标代码的大小
  - 指令填充浪费流了指令内存/cache
  - 循环展开/指令流水复制了代码
- 调度可变延时的内存操作
  - caches和/或内存体冲突，造成静态不可预料的变化
- 围绕静态不可预料的转移的调度
  - 转移路径不同，优化的调度也不同

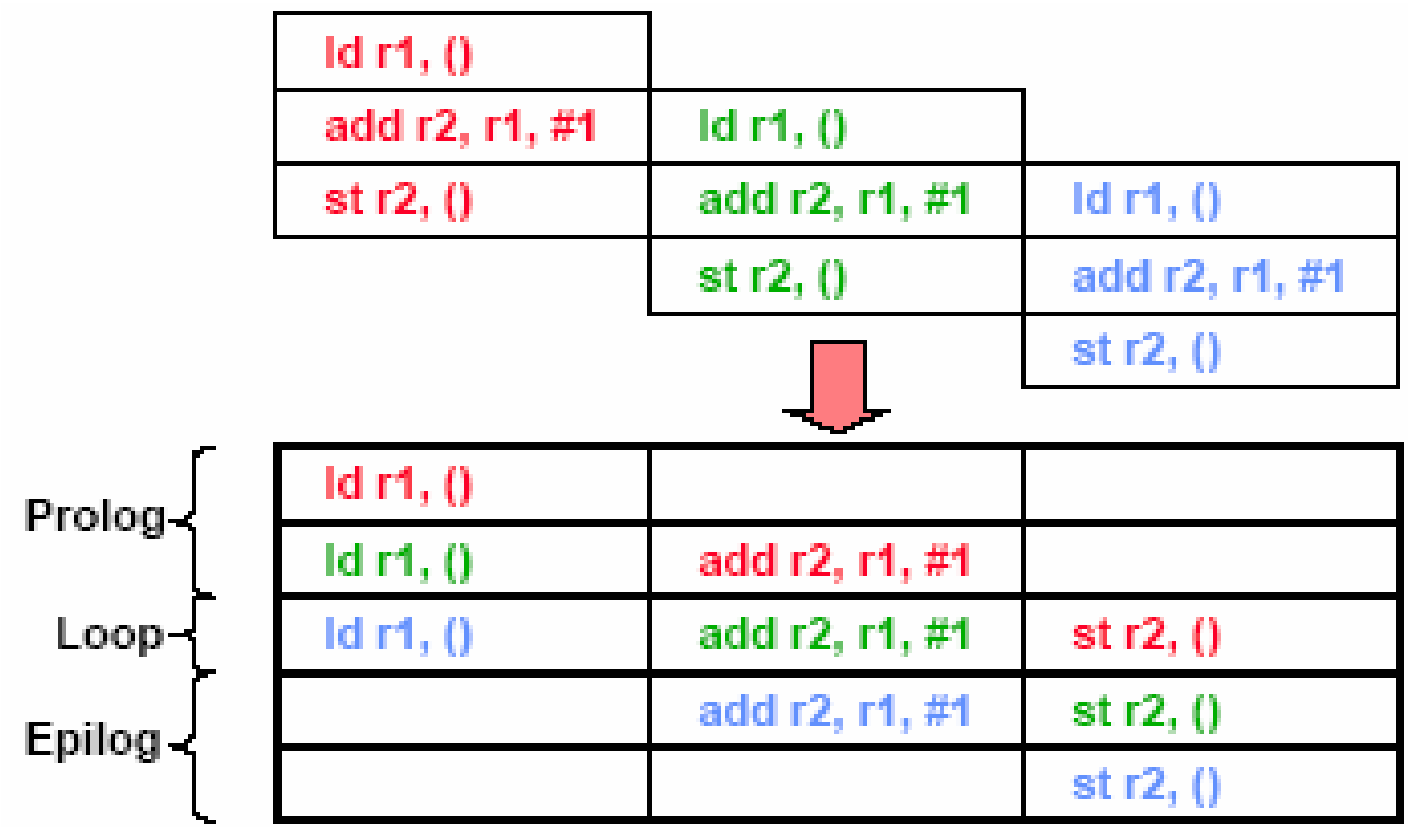
VLIW指令编码

- 各种减少无用字段影响的方案
  - －内存中采用压缩格式，扩展I-cache的再装入
  - －Cydra-5多操作指令：以顺序操作执行VLIW
  - －标出并行组（应用于TMS320C6x DSPs, Intel IA-64）



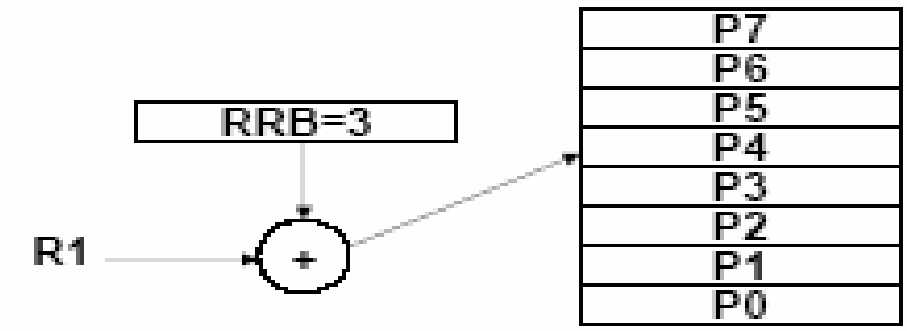
循环寄存器方式

问题：调度的循环需要很多寄存器，在开头和结尾需要大量重复的代码

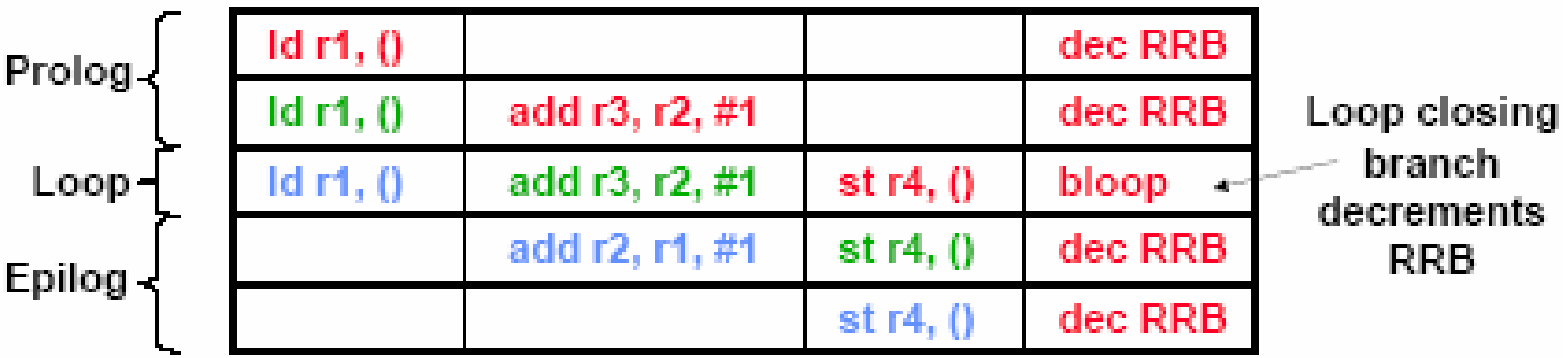


解决方案：为每个循环迭代分配一组新的寄存器

循环寄存器方式



- 循环寄存器基指针（RRB）指向当前寄存器组的基地址。给逻辑寄存器描述符加上一个值，就能得到一个物理寄存器号。通常，分为循环和非循环寄存器



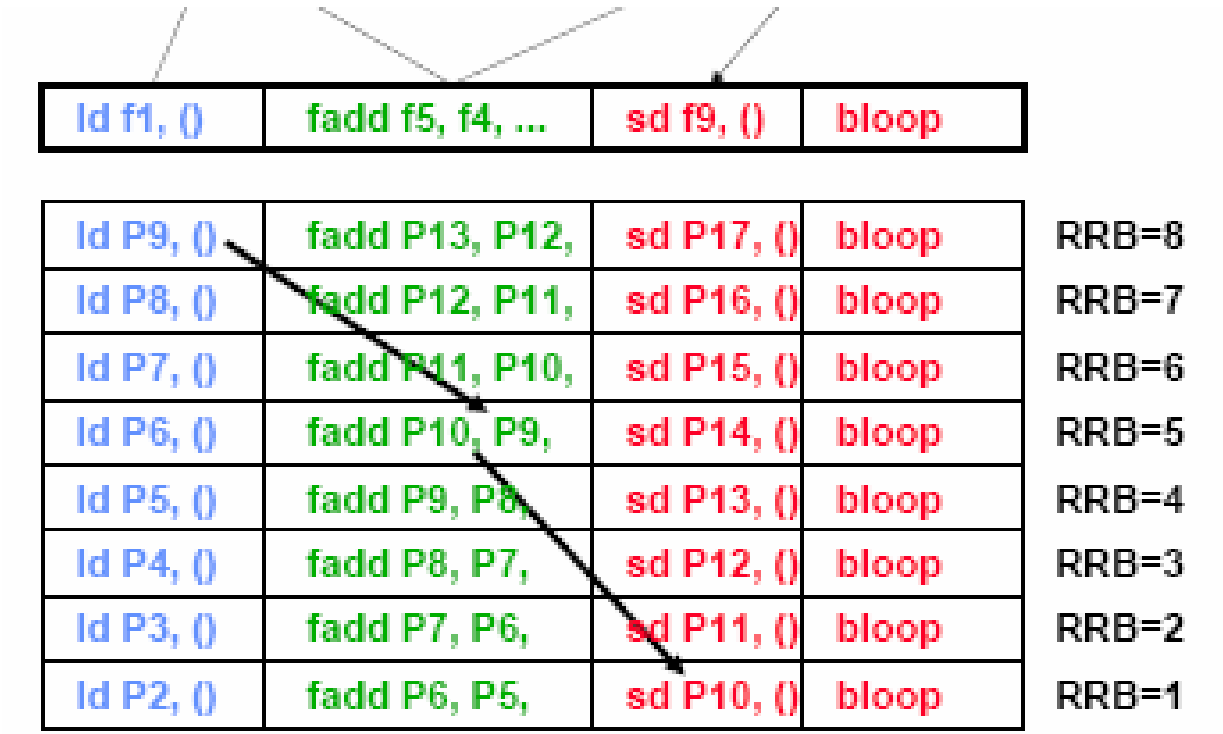
循环寄存器方式

3个周期的装入延时在寄存器描述符号中被编码为值为3的差

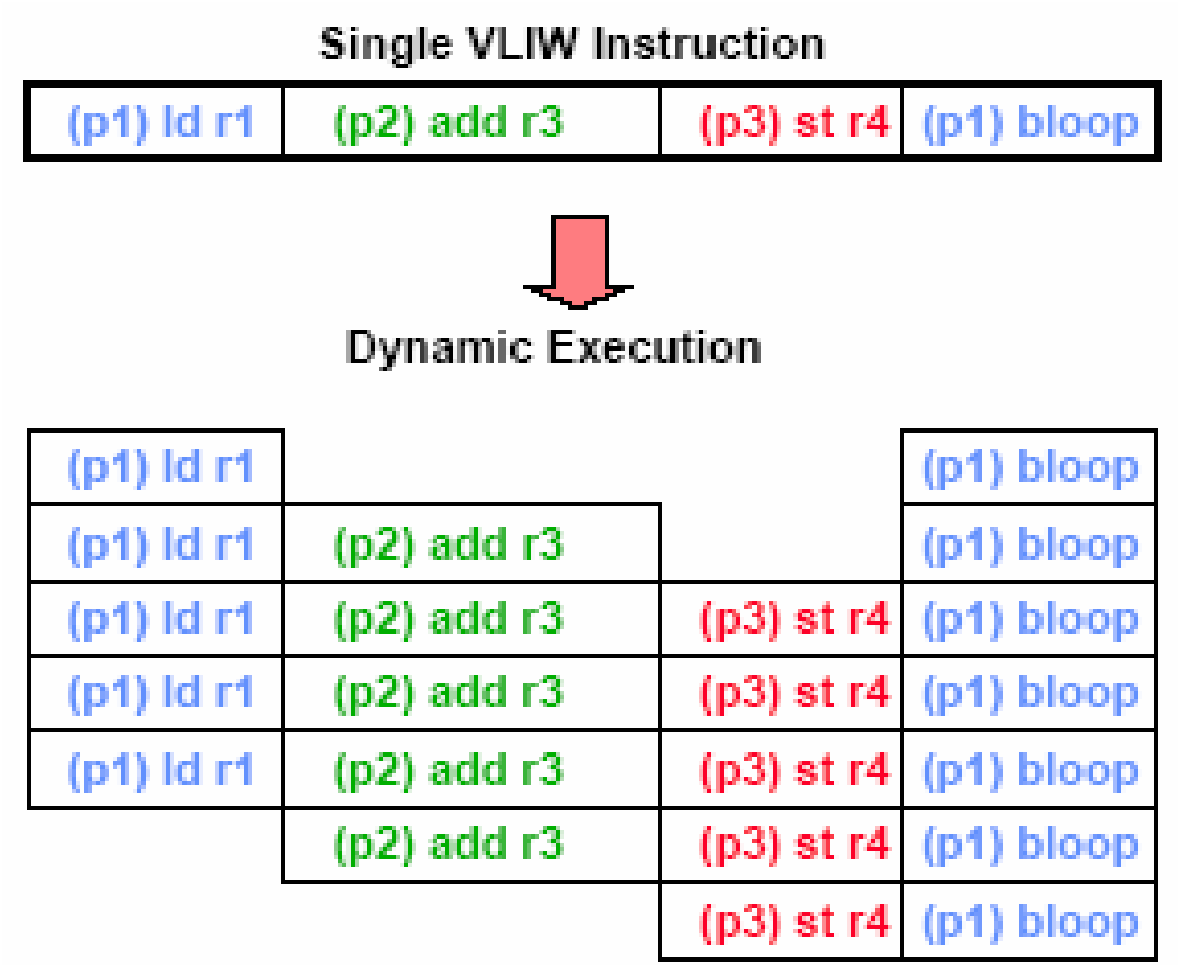
(f4 -f1 = 3)

4个周期的浮点加延时在寄存器描述符号中被编码为值为4的差

number (f9 –f5 = 4)



断定软件流水线阶段



软件流水线阶段由循环断定寄存器启动→更密的循环编码

内存延迟寄存器（MLR）

问题：装入有可变的延迟

解决：让软件选择期望的内存延迟

- 编译器试图以最大的使用距离调度代码（提早装入，译者注）
- 软件对匹配代码调度的延迟设置MLR
- 硬件保证装入只花费MLR周期，把值返回给处理器的流水线

  —硬件缓冲返回早的装入

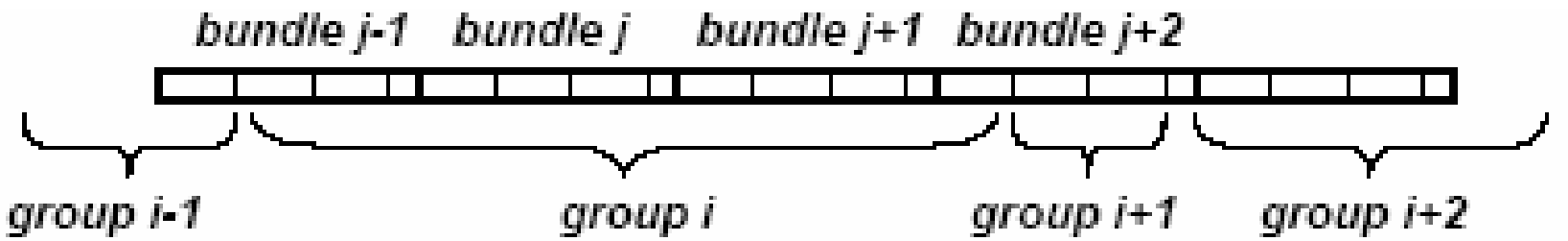
  —如果返回太晚，硬件停滞处理器

Intel EPIC IA-64

- EPIC是一种体系结构风格（比较CISC,RISC）
  一明确的并行指令计算
- IA-64 Intel另外一种ISA（比较x86,MIPS）
  – IA-64 = Intel 64-bit 结构
  – 一种兼容VLIW的目标码
- Itanium (即， Merced) 是第一代产品 (cf. 8086)
  – 第一台产品应该是 2001
  – McKinley 会是其第二代产品，2002年完成

IA-64指令格式

- 模板位描述指令与相邻捆中指令的组合关系
- 每组包含可以并行执行的指令



IA-64寄存器

- 128个通用64位整型寄存器
- 128个通用64/80位浮点寄存器
- 64个1位寄存器断定寄存器

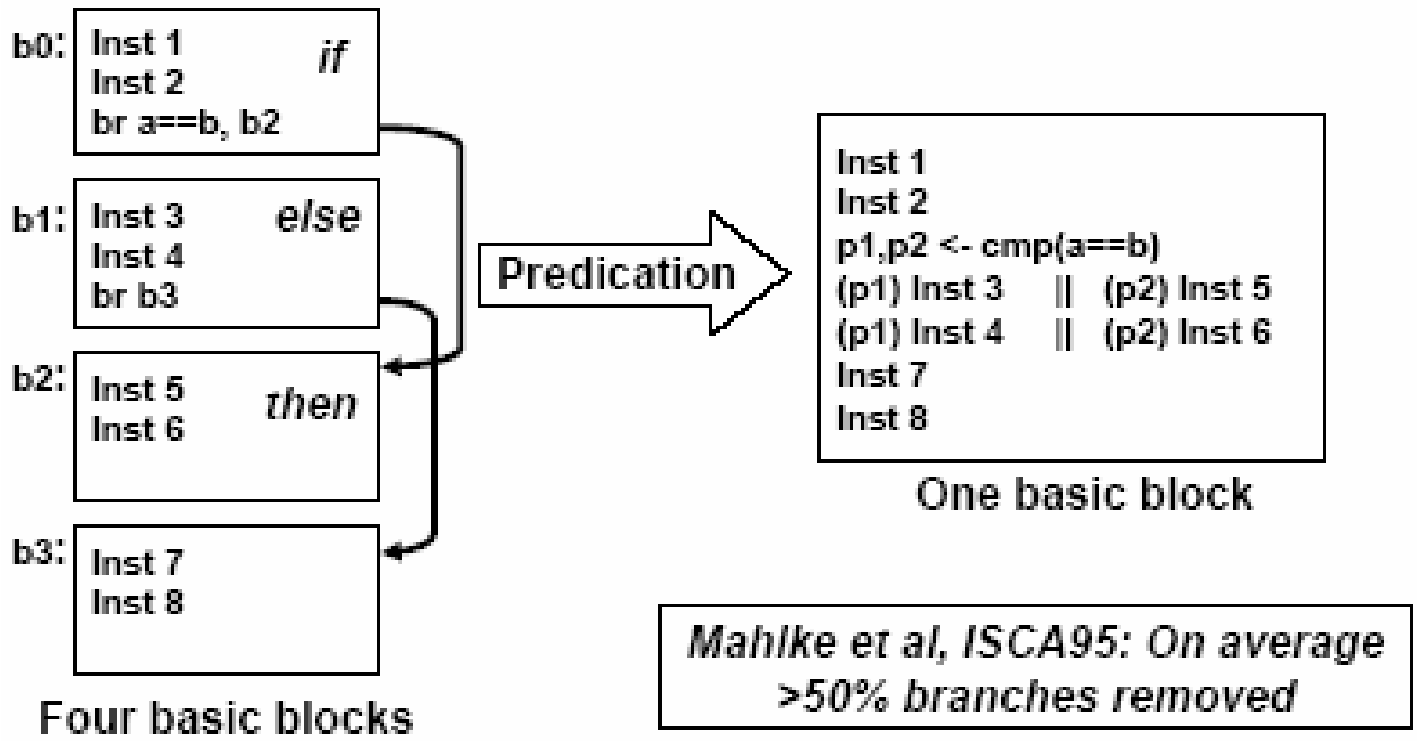
- GPRs轮流使用，以减少软件流水循环的代码大小

IA-64的断定执行

问题：误预测转移制约了ILP

解决：采用断定执行消除难于预测的转移

- 几乎所有的IA-64指令都能通过断定来条件执行
- 如果断定寄存器出错，那么指令变为NOP

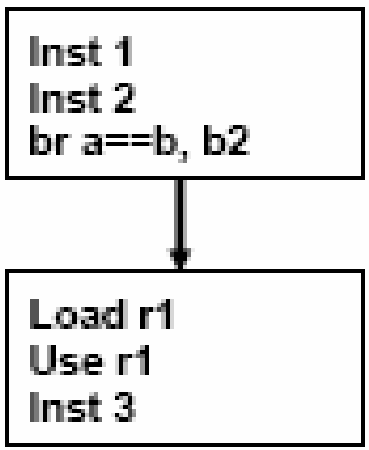


平均消除超过50%的转移

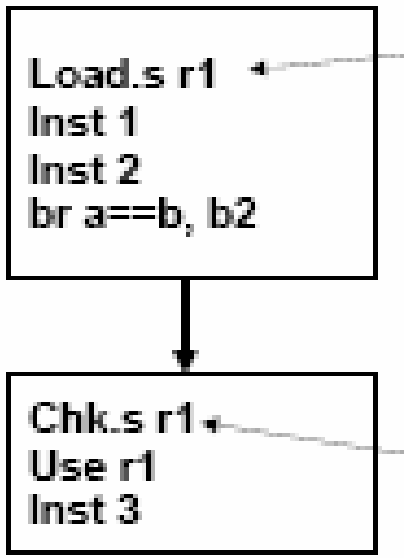
IA-64推测执行

问题：转移制约了编译器代码的运行

解决：不引起异常的推测操作



不能把装入指令移到转移指令之上
因为会引起欺骗性异常



推测装入决不能引起异常，
在可以在墓地寄存器里设置
“毒”位

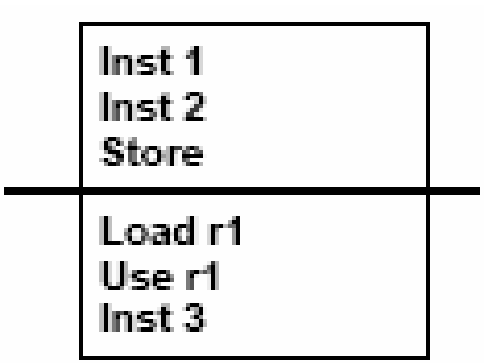
检查原始序列块中的异常，
如果检测到异常，跳转到
固定代码

对提早调度长延迟装入尤其有用

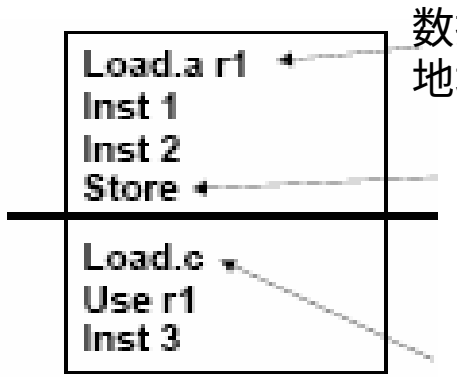
IA-64数据推测

问题：可能的存储冲突限制了代码调度

解决：硬件检查指针冲突



不能把装入指令移到存储指令之上，因为存储会写到统一地址



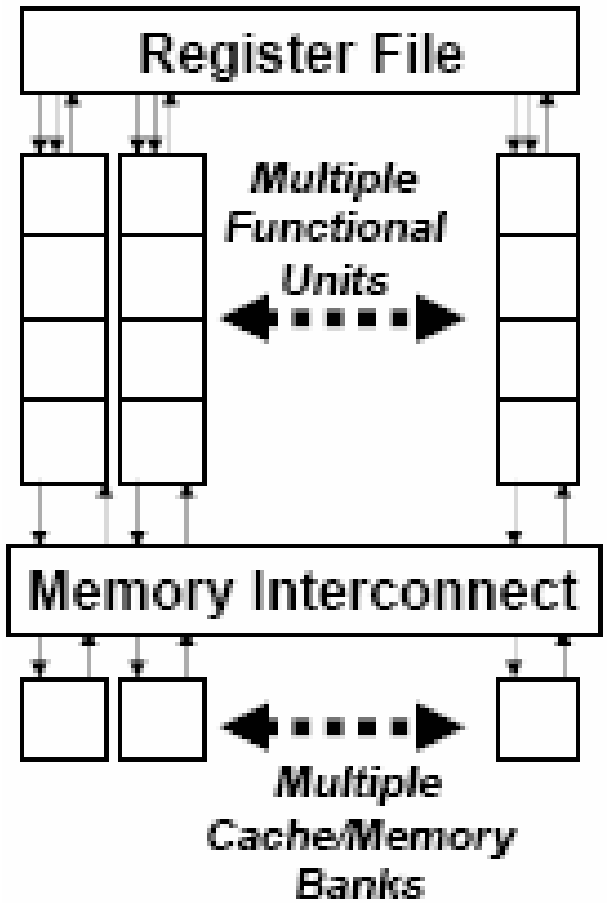
数据推测装入把地址加进地址检查表

存储会把与地址检查表匹配的装入变为无效

如果检查到装入无效（或不命中），跳转到固定代码

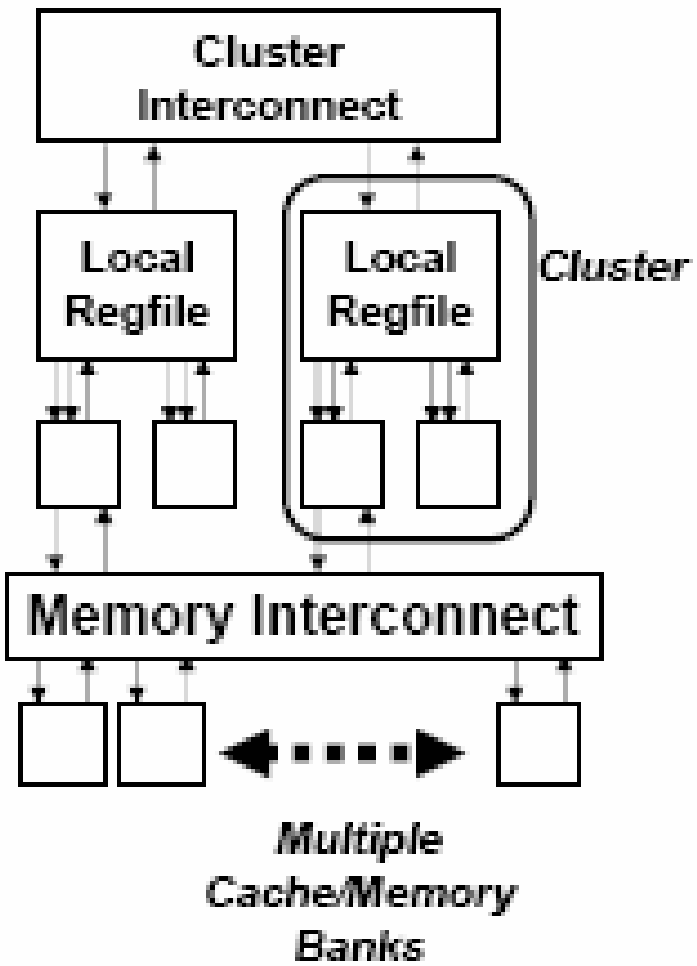
地址检查表中需要关联硬件

ILP数据路径硬件比例变化



- 复制功能单元和cache/内存体是直接的做法，且能够线性扩充
- 对N个功能单元，寄存器组的端口和旁路逻辑以二次方（N*N）增长
- N个功能单元的内存互联以及内存体数也以二次方增长
- （对于大N，可以尝试 O(N logN)互联方案）
- 技术扩充：与门电路相比，导线变得更慢
- 复杂的互联增加了延迟和面积
  =>需要更大的并发度来隐藏延迟

群式VLIW



- 将机器分为本地寄存器组和本地功能单元的群
- 群之间低带宽/高延迟互联
- 软件负责映射计算以减少通讯开销

静态调度的局限

静态调度的四个主要的缺点：

- 不可预料的转移
- 可变的内存延迟（无法预料的cache不命中）
- 代码大小的爆炸
- 编译器的复杂性