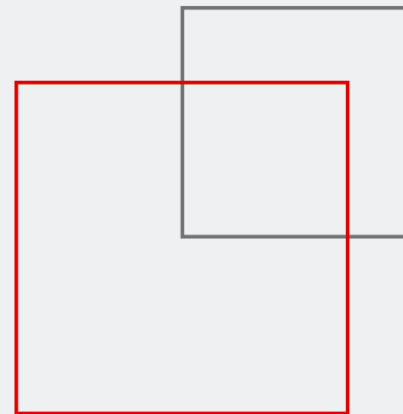


# 多发射处理器

静态多发射处理器

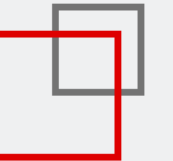
# 什么是多发射？

- 1 同时读出多条指令
- 2 同时对多条指令进行译码
- 3 同时执行多条指令



# 指令级并行性

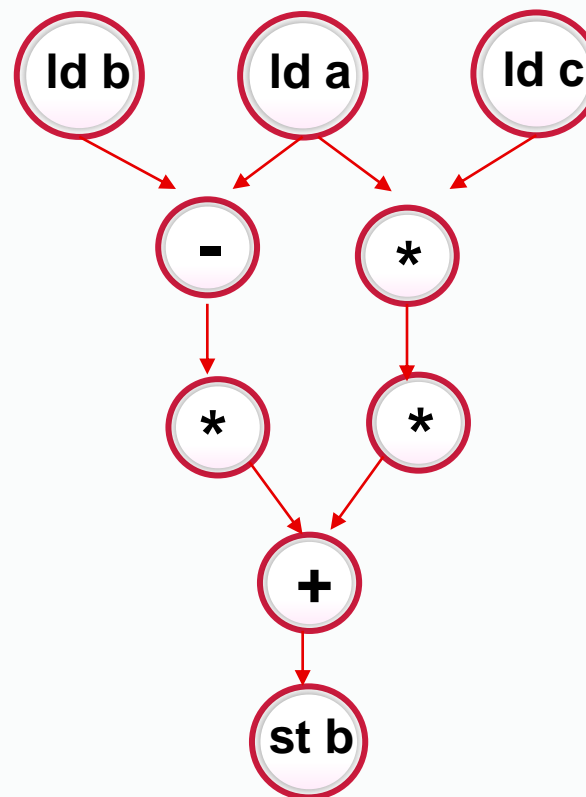
## (Instruction Level Parallelism: ILP)



# 多发射:开发指令级并行 (ILP)

$$D = 3 * (a - b) + 7 * a * c ;$$

*ld a*  
*ld b*  
*sub a-b*  
*mul 3\*(a-b)*  
*ld c*  
*mul a\*c*  
*mul 7\*a\*c*  
*add 3(a-b) + 7\*a\*c*  
*st d*



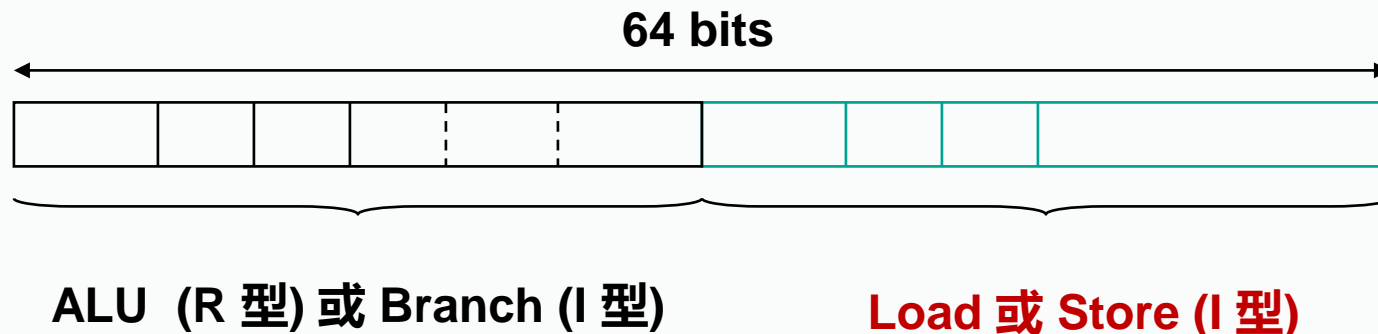
# 现代处理器的微结构

处理器	年份	时钟频率	流水级数	发射宽度	乱序执行?	核数	功耗
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103 W
Intel Core	2006	2930 MHz	14	4	Yes	2	75 W
Sun USPARC III	2003	1950 MHz	14	4	No	1	90 W
Sun T1 (Niagara)	2005	1200 MHz	6	1	No	8	70 W

# 多发射处理器的实现和主要特点

常用名	发射结构	冲突检测	调度方式	特点	处理器举例
静态超标量 superscalar (static)	动态	硬件	静态	按序执行	大部分嵌入式处理器, 例如ARM cortex-A8
动态超标量 superscalar (dynamic)	动态	硬件	动态	乱序执行	目前无
推测执行超标量 superscalar (speculative)	动态	硬件	带推测的动态	乱序、推测执行	大部分通用处理器, 如Intel Core i3,i5,i7
超长指令字 (VLIW)	静态	主要由软件完成	静态	编译器(隐式)完成冲突检测、指令调度	某些特定领域,如信号处理器 TI C6x
显式并发指令 运算(EPIC)	主要为静态	主要由软件完成	主要为静态	编译器(显式)完成冲突检测、指令调度	Intel 安腾 Itanium处理器

# 举例：一个 VLIW MIPS



- ❑ 双发射的 MIPS 处理器，两条指令组成一个指令束
- ❑ 指令束中的指令成对取指、译码和发射
- ❑ 由编译器安排指令束，选取每次同时发射的两条指令
- ❑ 如果找不到合适的指令，就用空指令noop代替



# 代码调度举例

以下代码

```
lp:  lw    $t0, 0($s1)    # $t0=array element
      addu  $t0, $t0, $s2  # add scalar in $s2
      sw    $t0, 0($s1)    # store result
      addi  $s1, $s1, -4   # decrement pointer
      bne   $s1, $0, lp    # branch if $s1 != 0
```

假设： 总能正确预测转移指令的转移方向

编译器：

- 将两条指令打包成一束长指令
- 在一束长指令上的两条指令必须无关
- Load-use 指令必须间隔一周期





# 代码调度举例

以下代码

```
lp:  lw    $t0, 0($s1)    # $t0=array element
      addu  $t0, $t0, $s2  # add scalar in $s2
      sw    $t0, 0($s1)    # store result
      addi  $s1, $s1, -4   # decrement pointer
      bne   $s1, $0, lp    # branch if $s1 != 0
```

	ALU 或 branch	数据传送	时钟周期
lp:		<u>lw \$t0, 0(\$s1)</u>	1
	<u>addi \$s1, \$s1, -4</u>		2
	<u>addu \$t0, \$t0, \$s2</u>		3
	<u>bne \$s1, \$0, lp</u>	<u>sw \$t0, 4(\$s1)</u>	4

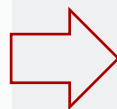
- 5 条指令花费4个周期, CPI= 0.8 (最好情况下是0.5)

# 代码调度举例

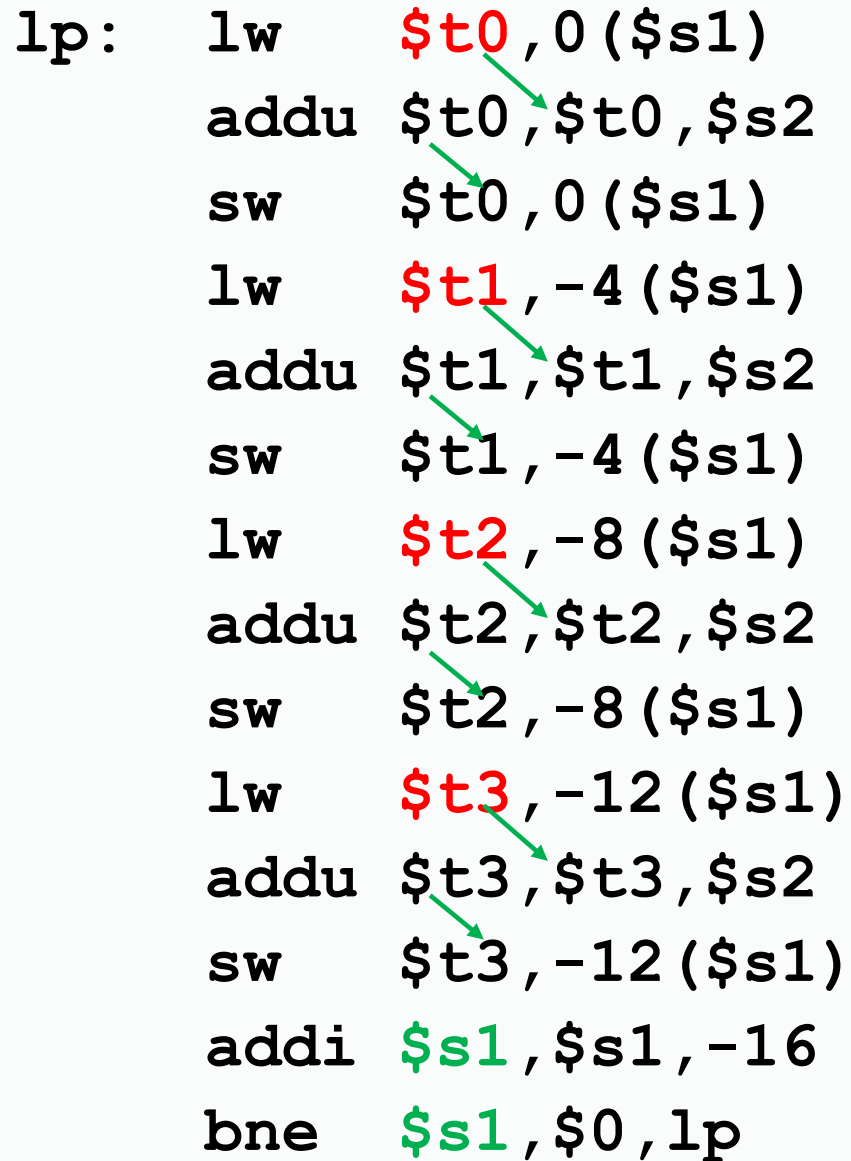
```
lp:
    lw      $t0, 0($s1)
    addu    $t0, $t0, $s2
    sw      $t0, 0($s1)
    addi    $s1, $s1, -4
    bne     $s1, $0, lp
```

- 指令条数增加了: 增加指令级并行性 (ILP)
- 转移指令减少了: 降低转移指令引起的开销
- 寄存器换名: 消除由于寄存器名字引起的相关性

循环  
展开



```
lp:    lw      $t0, 0($s1)
        addu    $t0, $t0, $s2
        sw      $t0, 0($s1)
        lw      $t1, -4($s1)
        addu    $t1, $t1, $s2
        sw      $t1, -4($s1)
        lw      $t2, -8($s1)
        addu    $t2, $t2, $s2
        sw      $t2, -8($s1)
        lw      $t3, -12($s1)
        addu    $t3, $t3, $s2
        sw      $t3, -12($s1)
        addi    $s1, $s1, -16
        bne     $s1, $0, lp
```



```

lp:  lw  $t0, 0($s1)
     addu $t0, $t0, $s2
     sw  $t0, 0($s1)
     lw  $t1, -4($s1)
     addu $t1, $t1, $s2
     sw  $t1, -4($s1)
     lw  $t2, -8($s1)
     addu $t2, $t2, $s2
     sw  $t2, -8($s1)
     lw  $t3, -12($s1)
     addu $t3, $t3, $s2
     sw  $t3, -12($s1)
     addi $s1, $s1, -16
     bne $s1, $0, lp

```

```

lp:  lw  $t0, 0($s1)
     lw  $t1, -4($s1)
     lw  $t2, -8($s1)
     lw  $t3, -12($s1)
     addu $t0, $t0, $s2
     addu $t1, $t1, $s2
     addu $t2, $t2, $s2
     addu $t3, $t3, $s2
     sw  $t0, 0($s1)
     sw  $t1, -4($s1)
     sw  $t2, -8($s1)
     sw  $t3, -12($s1)
     addi $s1, $s1, -16
     bne $s1, $0, lp

```



lp:    lw     \$t0, 0(\$s1)  
      lw     \$t1, -4(\$s1)  
      lw     \$t2, -8(\$s1)  
      lw     \$t3, -12(\$s1)  
      addu   \$t0, \$t0, \$s2  
      addu   \$t1, \$t1, \$s2  
      addu   \$t2, \$t2, \$s2  
      addu   \$t3, \$t3, \$s2  
      sw     \$t0, 0(\$s1)  
      sw     \$t1, -4(\$s1)  
      sw     \$t2, -8(\$s1)  
      sw     \$t3, -12(\$s1)  
      addi   \$s1, \$s1, -16  
      bne    \$s1, \$0, lp

ALU or branch	Data transfer	CC
<u>addi    \$s1, \$s1, -16</u>	<u>lw    \$t0, 0(\$s1)</u>	1
	<u>lw    \$t1, 12(\$s1)</u>	2
<u>addu    \$t0, \$t0, \$s2</u>	<u>lw    \$t2, 8(\$s1)</u>	3
addu    \$t1, \$t1, \$s2	lw    \$t3, 4(\$s1)	4
addu    \$t2, \$t2, \$s2	sw    \$t0, 16(\$s1)	5
addu    \$t3, \$t3, \$s2	sw    \$t1, 12(\$s1)	6
	sw    \$t2, 8(\$s1)	7
bne     \$s1, \$0, lp	sw    \$t3, 4(\$s1)	8

- 14 条指令8个周期,
- CPI =0.57 (最佳情况是0.5)

# VLIW的局限性

## 1 编译复杂、编译时间长

- 循环展开、冲突检测、指令调度
- 将if then else 结构转化为可预测得转移指令
- 存储器访问地址预测

## 2 代码膨胀

- 空指令浪费内存存储空间
- 循环展开后，也需要更多存储空间
- 需要更大内存带宽

## 3 锁步 (lock step) 机制

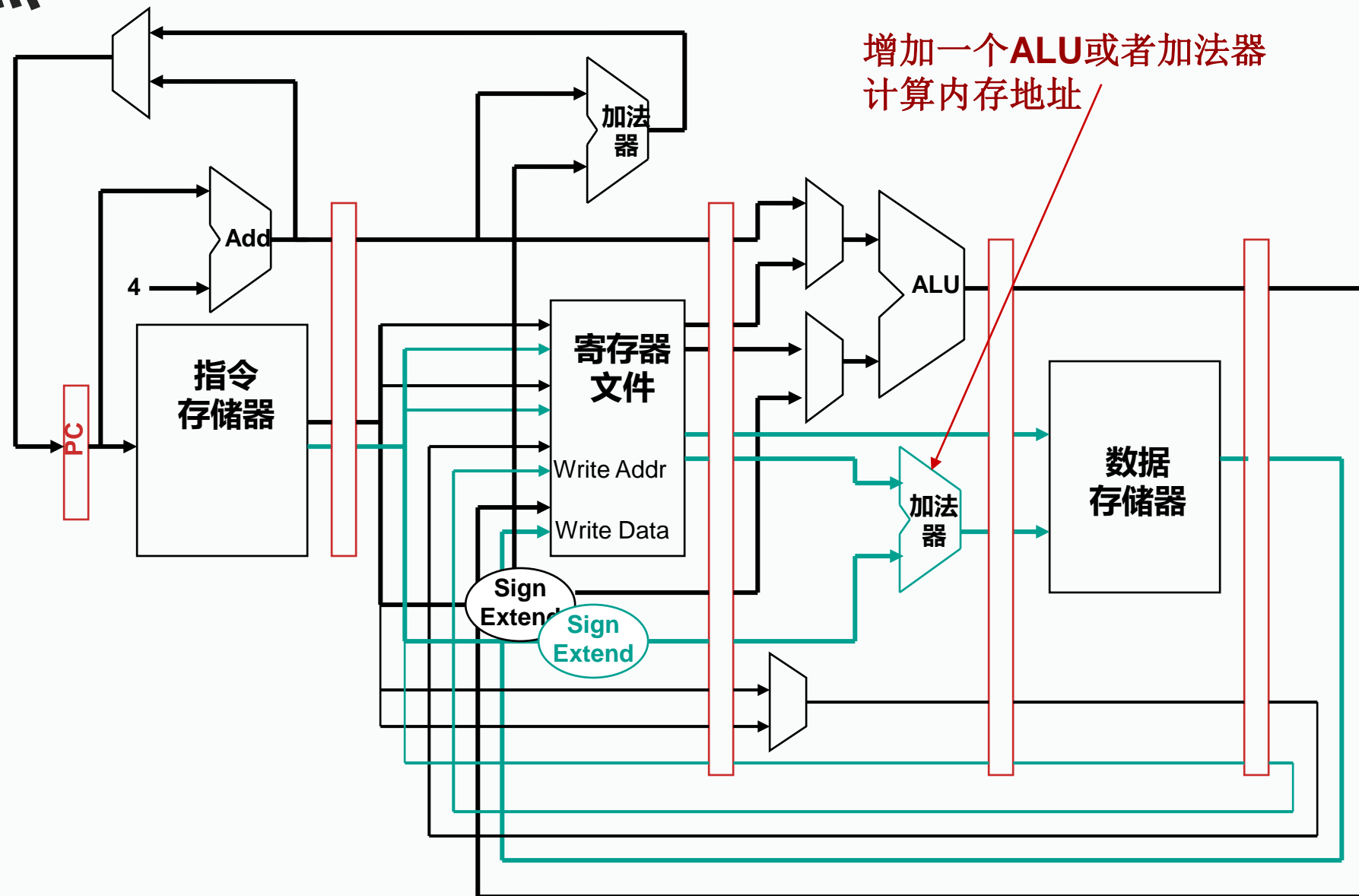
- 一条指令阻塞，其后所有指令都阻塞
- 相关性解除了，才允许发射有相关的指令
- 流水线段数越多，相关性越多

## 4 目标代码不兼容

- 市场接受意愿低

# VLIW的优点

- 硬件简单
- 成本低
- 能耗少



# 静态多发射处理器的现状

常用名	发射结构	冲突检测	调度方式	特点	处理器举例
静态超标量 superscalar (static)	动态	硬件	静态	按序执行	大部分嵌入式处理器，例如ARM cortex-A8
动态超标量 superscalar (dynamic)	动态	硬件	动态	乱序执行	目前无
推测执行超标量 superscalar (speculative)	动态	硬件	带推测的动态	乱序、推测执行	大部分通用处理器，如Intel Core i3,i5,i7
<b>超长指令字</b> (VLIW)	静态	主要由软件完成	静态	编译器（隐式）完成冲突检测、指令调度	某些特定领域,如信号处理器 <b>TI C6x</b>
<b>显式并发指令运算</b> (EPIC)	主要为静态	主要由软件完成	主要为静态	编译器（显式）完成冲突检测、指令调度	<b>Intel 安腾 Itanium</b> 处理器



# 小结

## 指令级并行

多发射处理器：开发程序中的指令级并行性

## 静态多发射



编译器决定哪些指令可以在同一周期内执行

## 编译器的作用

冲突检测、指令调度、循环展开、转移预测等

## 优缺点

通用处理器更多采用动态多发射：即超标量结构



# 谢谢观看

上海交通大学