第四章 RISC微处理器技术

第一节 RISC技术概述

CISC — Complex Instruction Set Computer

RISC — Reduced Instruction Set Computer

一、RISC起源

- 指令集：控制器设计的基础
- 指令功能分解结合时序是设计的基本步骤

组合逻辑设计

指令功能分解成微操作(含指令流程和操作时间表)→结合时序信号→写出微信号的逻辑表达式并综合并化简→逻辑电路实现

例：

1）流程图

例1：MOV R0, R1；

FT0： M ⟶ IR,

ET0： R1 ⟶ R0

ET1： PC ⟶ MAR

例2：MOV (R0), (R1)；

FT0： M ⟶ IR, PC+1 ⟶ PC

ST0： R1 ⟶ MAR

ST1： M ⟶ MDR ⟶ C

DT0： R0 ⟶ MAR

ET0： C ⟶ MDR

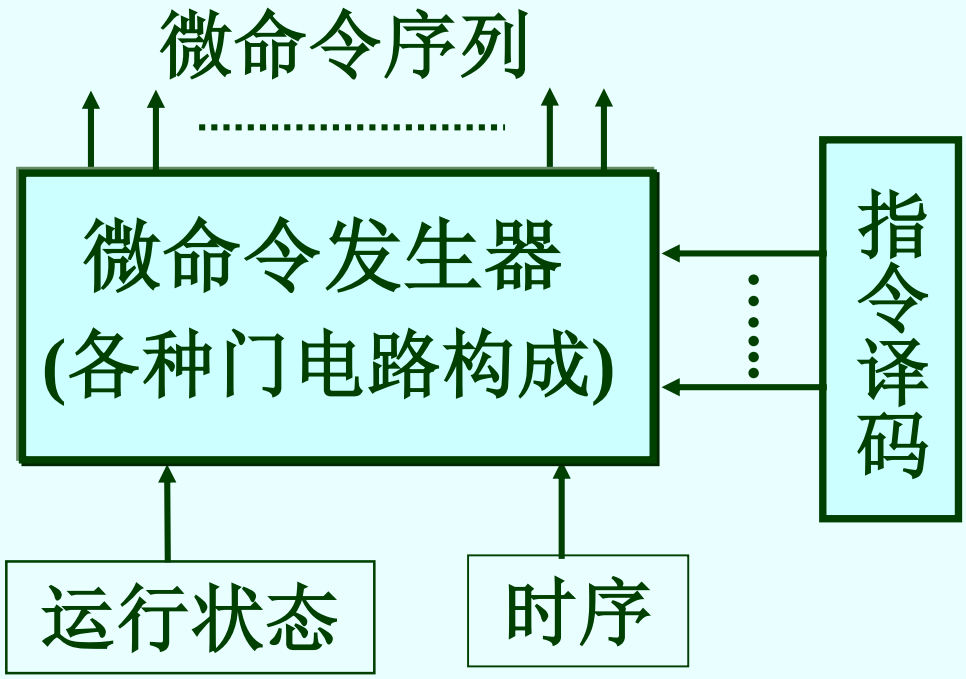
ET1： MDR ⟶ M

ET2： PC ⟶ MAR

源数

目的地址

综合简化后逻辑实现：

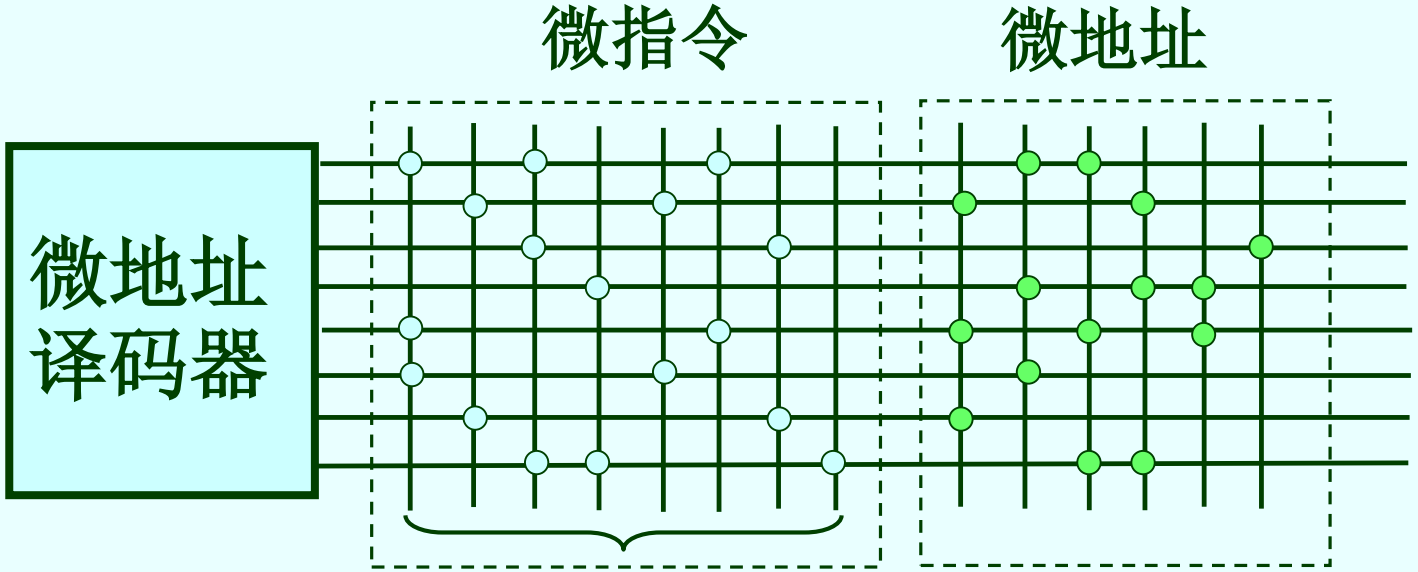


优点：硬件实现，速度快

缺点：设计过程及电路复杂；
修改和更新换代困难；

微程序设计

指令功能分解成微操作→微操作编码→存入微程序库ROM →微命令译码结合时序→完成功能



优点：设计过程及电路相对简单；修改和更新换代更容易。

缺点：微命令实现，速度较慢

采用微程序设计的另一原因：处理器与存储器速度的矛盾

传统处理器设计的难题：

速度、复杂性、设计周期等矛盾

– 对指令系统进行的研究，统计表明：

- 软件中大部分指令为简单指令(约80%)，复杂指令只占少数 (约20%)；
  (如指令：ADC [变址寻址]，立即数 ；复杂指令)
- 软件中的简单指令约占总运行时间的20%，复杂指令约占总运行时间的80%；
- 造成控制电路复杂的主要原因是由于复杂指令的存在；

实验及结论

从指令集中去掉复杂指令，而复杂指令功能由软件实现。可简化电路设计，从而去掉微程序，采用硬连控制方法，提高处理器速度。

结论：整体性能有较大提高 → **RISC处理器出现**

说明：复杂指令功能由软件实现与提高速度相矛盾？

- 复杂指令使用频率较低；
- 去掉微程序，采用硬件控制，提高了速度
- 简单指令有利于流水线执行
- 简化电路节省了芯片面积，利于增加Cache容量

二、RISC处理器特征描述

1. 简单固定的指令格式

- 指令长度固定

  指令长度无需译码，简化了译码电路并节省了长度译码时间；

  指令长度一般限定在总线宽度以内，保证取指令码在一个总线周期完成，避免了多周期取指造成的流水线阻塞；

- 指令字段位置固定

  使指令译码与取源操作数并行；

- 指令意义简单

  功能单一，简化硬件逻辑

2. 减少寻址方式和指令数量

作为简化硬件逻辑的措施之一。

3. 流水线(或超级流水线)

尽量使指令在单周期执行完成;

RISC的设计思想更利于指令流水线方式的运行。

4. 大容量高速缓存

节省的芯片面积有利于集成大容量高速缓存,减少了访存次数。

5. 大量寄存器

减少访存,上下文切换尽可能在寄存器中完成。

6. 硬连控制(去掉微程序)

以简化的指令集为基础,提高指令执行速度。

7. 采用存取式体系结构(Load/Store结构)

专门的访存指令才允许访存，避免执行周期访存造成流水线阻塞。如:不允许 ADD R_i, 20[R_j]类指令。

8. 哈佛(Harvard)总线结构

分离的指令cache和数据cache，利用双总线动态访问机构，使数据存取和指令预取可以并行。

9. 重叠寄存器窗口技术

将大量寄存器分成多个重叠窗口，用于过程调用和返回时直接传递参数，减少了访问主存所需时间。

10. 优化编译技术

如: (1) 高级语言到简单低级语言的翻译

(2) 必须访问内存时安排其它可以并行的操作

(3) 由编译来完成指令重调度(乱序)。

— RISC技术的典型产品(了解）

(1) Alpha处理器(DEC公司)

相继推出Alpha21064、21164、21264、21364等。

(2) MIPS处理器

1986年至1997年，相继推出R2000、R3000、R4000、R8000、R10000等32位和64位处理器等。

(3) HP PA-RISC处理器

64位处理器PA-8000系列。与Intel合作开发的IA-64，与已有PA-RISC和Intel x86系列芯片兼容。

(4) Sun公司的SPARC微处理器芯片

采用“SPARC标准”设计RISC处理器。

(5) IBM公司的Power PC微处理器芯片

强调系统的处理能力，而不仅仅是指令级的并行。

第二节 RISC和CISC对流水线的影响

1. 可能造成CISC处理器流水线阻塞的原因

(1) 指令太长, 一个周期不能取出全部指令码;
(2) 指令功能复杂，一个执行周期不能完成;
(3) 无条件转移指令;
(4) 条件转移指令，预测错误;
(5) 数据相关
(6) 资源冲突

(3)、(4) 控制相关

2、可能造成RISC处理器流水线阻塞的原因

RISC处理器中，通常不存在指令太长和功能太复杂。因此造成RISC流水线阻塞的原因通常是：

(1) 无条件转移指令；
(2) 条件转移指令，预测错误。
(3) 数据相关
(4) 资源冲突

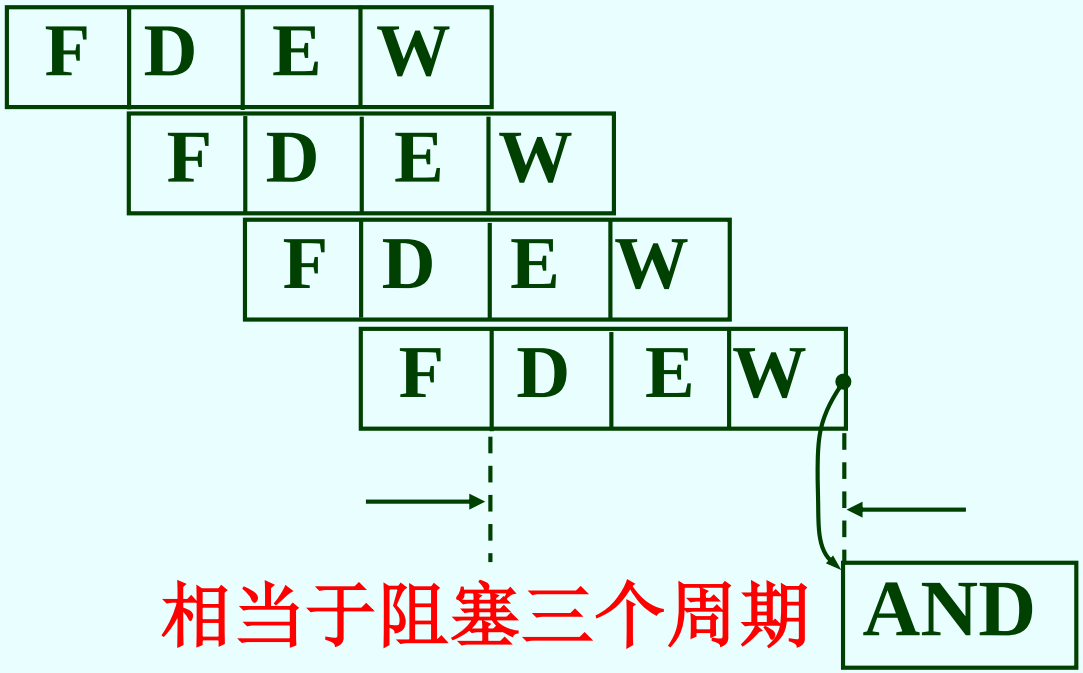
(1) 转移指令(控制相关或指令相关)

- 无条件转移指令
- 条件转移指令，预测错误

•无条件转移指令：

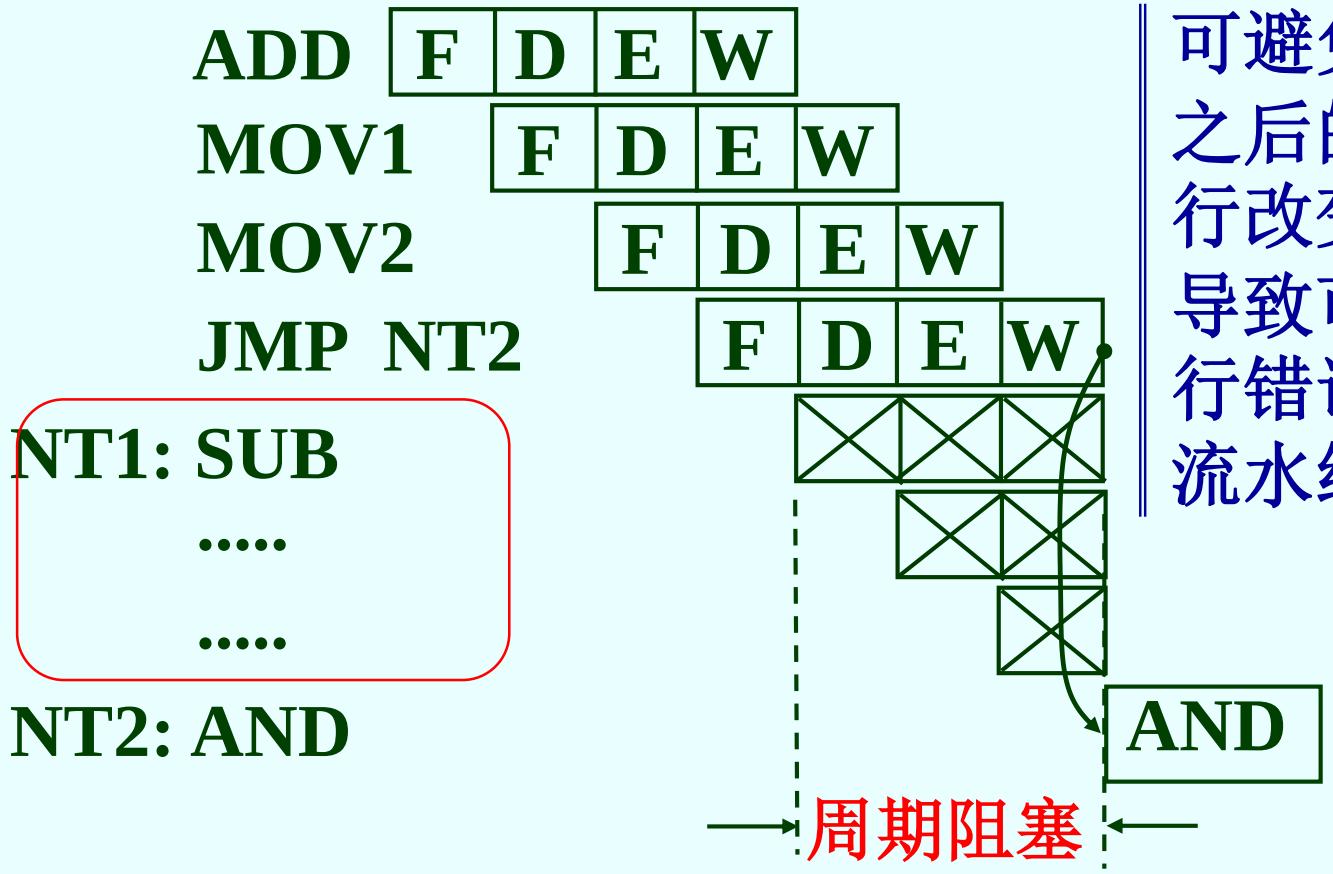
```
        ADD
        MOV1
        MOV2
        JMP  NT2
NT1:    SUB
        .....
NT2:    AND
```

流水线情况(假设四级流水线)：



解决措施:

1) 硬件自动冻结流水线周期



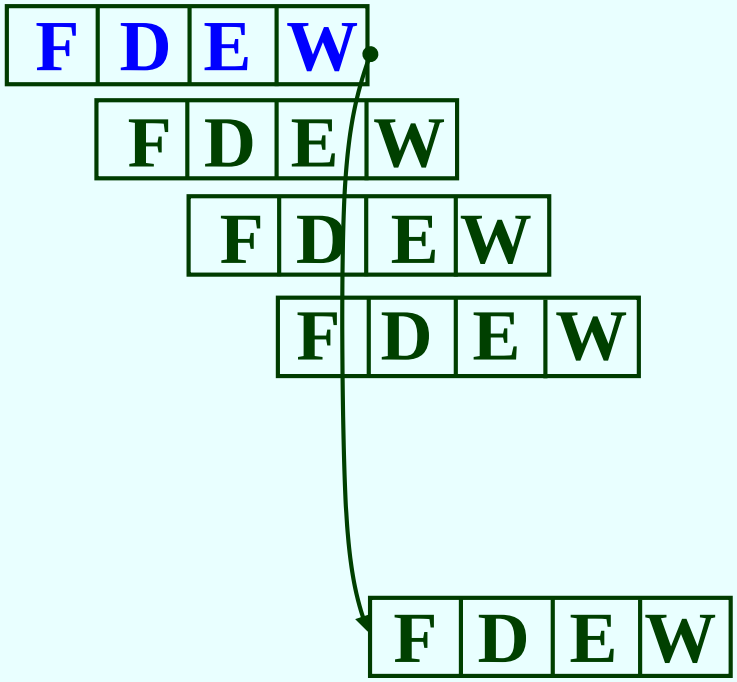
可避免NT1指令以及之后的几条指令的执行改变处理器状态,导致可能引发程序执行错误,但不能避免流水线周期损失。

2) 延迟转移技术

基本思想是调整指令执行顺序。

原程序	调整后的顺序
ADD	JMP NT2
MOV1	ADD
MOV2	MOV1
JMP NT2	MOV2
NT1: SUB	NT1: SUB
.....	.....
NT2: AND	NT2: AND

流水线情况:



这一方法也称为指令重调度。

3) 插入空操作指令(指令重调度)

```
        ADD
        MOV1
        MOV2
        JMP  NT2
        NOP
        NOP
        NOP
NT1:    SUB
         .....
NT2:    AND
```

可由硬件实现也可由软件实现

其效果等同硬件自动冻结流水线周期!

- 对条件转移指令

采用分支预测或指令重调度

例：原程序

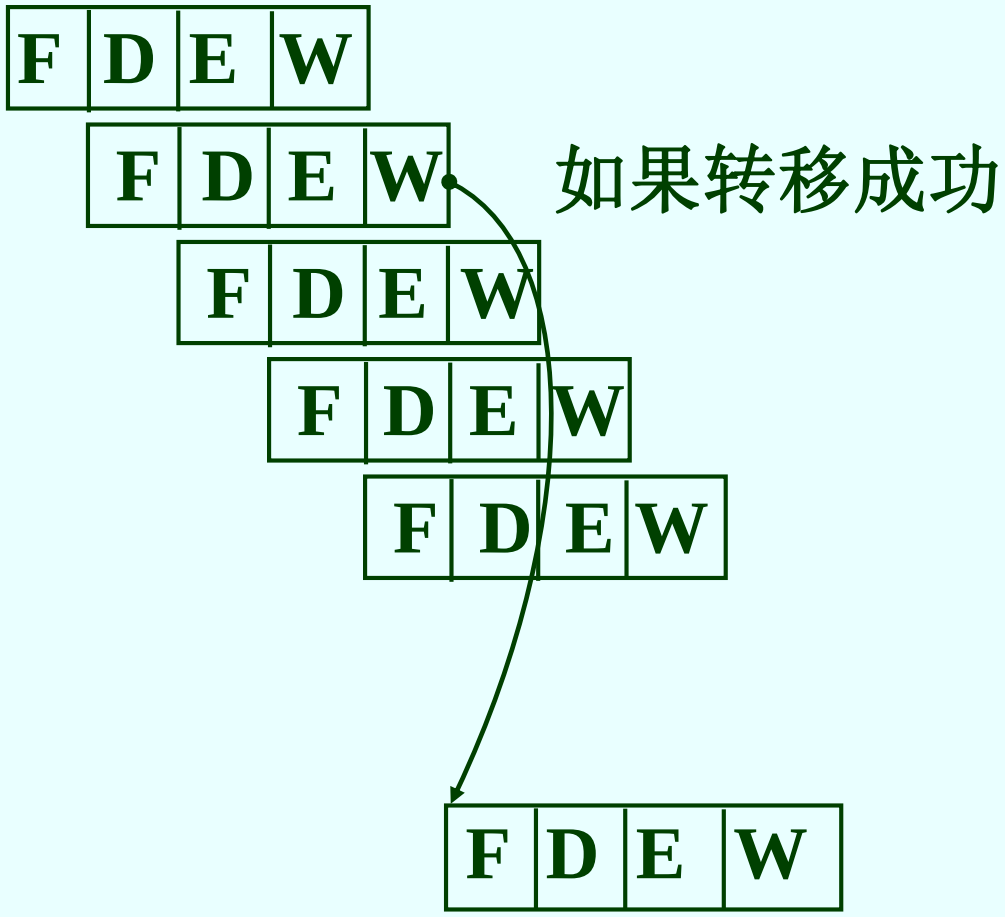
```
1:   MOV  R1,  R2
2:   MOV  R6,  a
3:   OR    R7, R6
4:   ADD   R3,  R4
5:   JNC,   N
6:   ADD   R4,  R5
        ……
N:   AND  R7,  R8
```

调整指令顺序为:

```
1:   ADD   R3,  R4
2:   JNC,   N
3:   MOV  R1,  R2
4:   MOV  R6,  a
5:   OR    R7, R6
6:   ADD   R4,  R5
        ……
N:   AND  R7,  R8
```

调整后的流水线:

```
    ADD   R3,  R4
    JNC,  N
    MOV  R1,  R2
    MOV  R6,  a
    OR     R7, R6
6:  ADD  R4,   R5
          ……
N:  AND  R7,  R8
```



(2) 数据相关

例:

```
ADD   R0, R1  ;   R0+R1 → R0
INC    R0      ;  R0+1 → R0
MOV  R2,  R3 ;   R3 → R2
   …
```

第一条指令与第二条指令出现了数据相关

解决方法:

指令重调度或插入空操作指令或硬件自动冻结周期

```
ADD   R0, R1
MOV  R2,  R3
INC    R0
   …
```

```
ADD   R0, R1
NOP
INC    R0
MOV  R2,  R3
   …
```

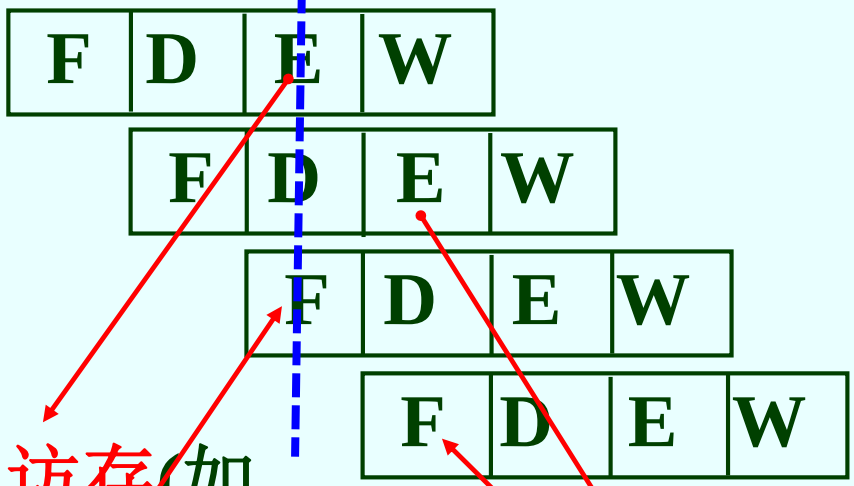
效果上与插入空操作指令相同

(3) 资源冲突

不同指令执行时, 要求使用同一资源。

资源冲突的典型情况: 不同指令对存储器的访问

如下图的指令流水线:

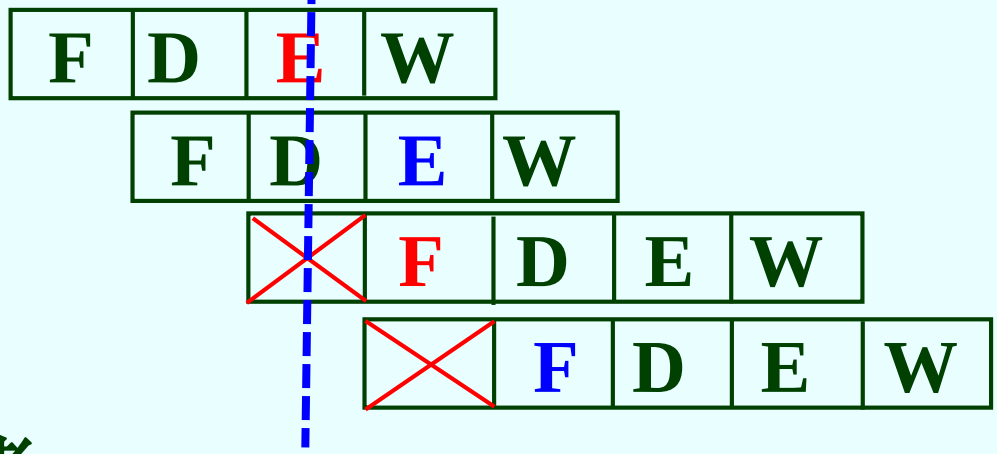


如果该阶段出现访存(如ld/St指令), 则可能与第三条指令的取指操作冲突。

若访存,可能与第四条指令的的取指操作冲突。

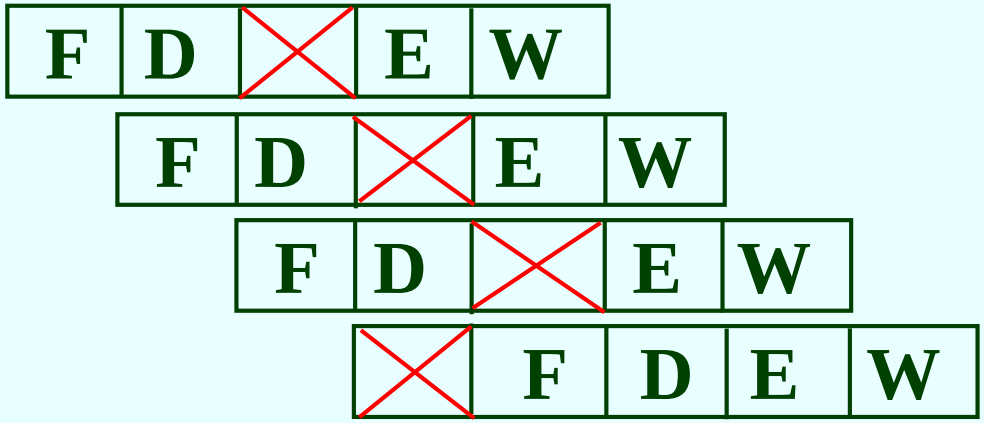
解决措施:

① 硬件自动冻结(阻塞)某一个周期



假设第一条指令的执行需要访存,则可能与其它指令的取指操作冲突

或者:



但一般要求“执行”优先于“取指”。所以一般不会采用后一种方式。

② 从结构设计上避免冲突(对总线冲突)

指令Cache和数据Cache分离(哈佛结构), 使取指令和取数据相互独立, 以减少总线资源冲突。

目前的处理器基本上都采用该结构, 如Pentium和i860处理器均采用该结构。

另一种方式是: 用一个Cache, 但指令端口和数据端口分离, 因此可以同时访问两个不同端口,指令和数据分别从两个不同端口读取。

哈佛结构是否可完全避免总线资源冲突?

资源冲突也有其它情况。例如,两条指令都要求使用一个加法部件或图形处理部件等。

解决资源冲突一般可采用的方法:

为可能导致冲突的共享资源设置一个“忙”触发器,当一条指令需要使用该资源时,首先通过资源冲突检测部件检测该触发器的状态。只有该触发器处于“空闲”状态时才能使用。一旦占有该设备,则将其置为“忙”。使用完该设备,将其置为“空闲”。

附:流水线的中断处理(自学)

附: 流水线的中断处理

(1) 指令串行执行的中断处理

(2) 流水线执行的中断处理

- “非精确断点”保护

  无论在第i条指令的哪一流水段上发生中断请求, 就不再允许后继指令进入流水线, 但已经在流水线上的所有指令, 允许它们执行完毕, 然后再转去执行中断处理。断点就是中断请求发生时, 最后进入流水线的那条指令的地址。

  因此, 断点现场的保护并不精确的对应第i条指令, 而是在第i条指令之后, 中断发生之前进入流水线的那些指令。

“非精确断点”法的硬件较简单, 缺点是中断响应时间较长, 还可能导致程序出错, 并且程序调试不太方便:

导致程序出错例:

```
i:      FADD   R1,  R2;   R1+R2→R1
i+1:    FMUL   R3,  R1;   R3×R1→R3
```

如果第i条指令发生溢出, 产生异常中断, “非精确断点法” 让已进入流水线的第i+1条指令完成(该指令使用第i条指令的结果为源操作数), 导致浮点乘法出错。

程序调试不方便:

如果在第i条指令设置断点, 而真正的中断发生在最后进入流水线的那条指令, 导致程序不能准确中断在所设置的断点处。

- “精确断点”保护

  如果第i条指令发生程序性错误(或故障), 或者在第i条指令设置断点, 则断点地址就是该指令的地址, 即中断处理程序对现场的处理都精确对应第i条指令。同时要把流水线中所有指令的执行结果和现场都保存起来, 以便之后的恢复。

  该方式需要大量寄存器来保存流水线各指令现场。

注: 对于来自I/O设备的外部中断, 由于非精确断点法并不影响程序的执行结果(仅中断响应时间较长), 一般不使用精确断点法;

由于中断发生的频率远小于程序中转移发生的频率, 因此, 不成为流水线性能下降的主要矛盾。

第三节 Intel 80860处理器 (i860)

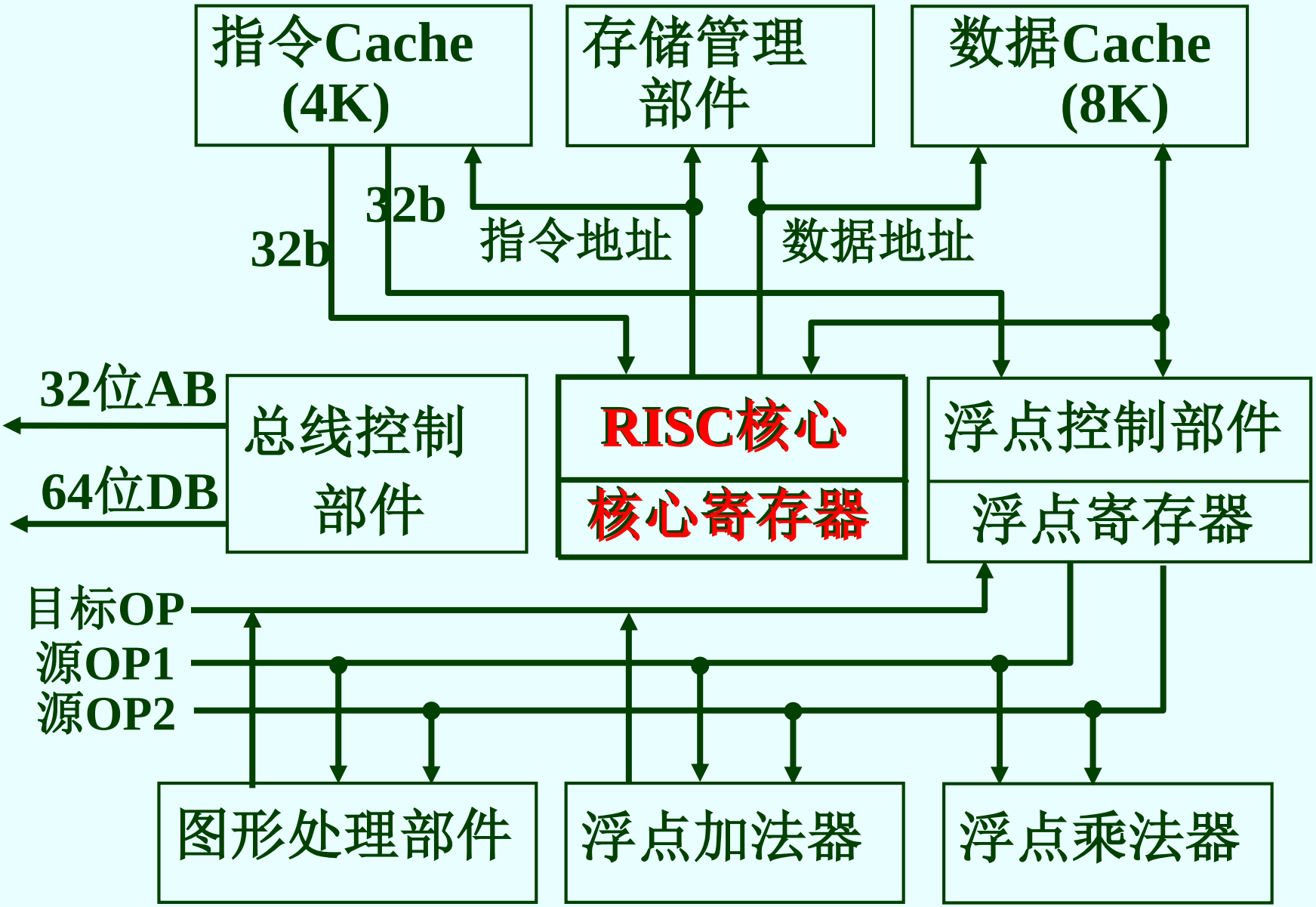
一、系统构成

(1) 4K指令Cache
(2) 8K数据Cache
(3) 总线控制部件
(4) 存储管理部件
(5) 浮点控制部件
(6) 浮点加法及乘法部件
(7) 三维图形处理部件
(8) RISC核心

- 地址总线：32位
- 数据总线：64位
- 指令Cache可同时发送两条指令，一条发送到RISC核心，另一条发送到浮点部件

系统结构示意图：

系统结构示意图：


指令Cache
(4K)
存储管理
部件
数据Cache
(8K)
32b
32b
指令地址
数据地址
32位AB
64位DB
总线控制
部件
RISC核心
核心寄存器
浮点控制部件
浮点寄存器
目标OP
源OP1
源OP2
图形处理部件
浮点加法器
浮点乘法器

(一) RISC核心

1. 功能

指令读入、数据存取、除浮点运算外的其它指令操作

2. 指令格式

32位定长指令；操作码、源操作数、目标操作数

字段位置固定 (指令译码与取源操作数并行)。

3. 寄存器

32个32位通用寄存器$r_{31} \sim r_0$

4. 指令流水线

RISC核心采用4级流水线

取指	译码	执行	写结果

5. 存取式体系结构

访存指令：Load/Store

(二) 浮点部件

1. 组成　浮点控制器　浮点加法器　浮点乘法器
2. 寄存器　f_{31}～f_0　32个32位浮点寄存器
   - 两个寄存器可在逻辑上构成一个64位寄存器;
   - 四个寄存器可在逻辑上构成一个128位寄存器;

二、i860指令系统例

两大类指令:

(1) 整数指令

整数和浮点数的存取指令

算术及逻辑运算指令

转移指令

整数寄存器与浮点寄存器间的数据传送指令

(2) 浮点指令

浮点数据加、减、比较等

浮点数乘、倒数、平方根等

(一) 整数指令

1. 整数的存取

源

目的

(1) 取数指令 ld.x isrc1(isrc2), idest

其中： isrc2和idest为r31~r0之一；

isrc1为r31~r0之一或16位立即数;

完成功能： (isrc1+isrc2) → idest

指令后缀：.x
- .b 取8位数据
- .s 取16位数据
- .l 取32位数据

取数指令例:

① ld.l 0(r14), r15

完成 (0 +r14)→r15; 间接寻址

② ld.l 8(r14), r15

③ ld.l r13(r14), r15

变址寻址

④ ld.l 64(r0), r15 r0≡0, 直接寻址

r0恒为0, 与执行的操作无关。作用如：

- subu r5, r4 r0; 产生比较指令
- 提供直接寻址
- 通过逻辑运算, 为某个寄存器赋0值

体现精简指令特点

(2) 存数指令 st.x isrc1, const(isrc2)

完成: isrc1→(const+isrc2)

其中: isrc1和isrc2为寄存器, const为16位立即数

例: st.l r1, 64(r2); r1→(64+r2)

2. 浮点数的存取

(1) 取数 fld.y isrc1(isrc2), fdest

完成: (isrc1+isrc2) → fdest

isrc1为16为立即数或整数寄存器;

isrc2为整数寄存器; fdest为浮点寄存器。

(2) 存数 fst.y freg , isrc1(isrc2)

完成: freg→(isrc1+isrc2)

freg为浮点寄存器;

isrc1和isrc2与取数指令的规定相同。

浮点数传送支持32、64和128位;后缀y指明传送宽度:

.y { .l 32位数传 .d 64位数传 .q 128位数传

例: fld.d 0(r3), f20; 64位数→f20:f21

fld.q 0(r3), f20; 128位数→f20:f21:22:f23

3. ireg与freg之间的数据传送

FxIr fsrc , idest ; 完成: fsrc → idest

IxFr isrc , fdest ; 完成: isrc → fdest

4. 算术运算指令

统一格式: Opcode isrc1 , isrc2, idest

完成运算: isrc1 op isrc2 → idest

加法: Adds isrc1 , isrc2, idest

Addu isrc1 , isrc2, idest

减法: Subs isrc1 , isrc2, idest

Subu isrc1 , isrc2, idest

isrc1为立即数或寄存器, 其余参数为寄存器

5. 逻辑运算指令

(1) “与”

“与”： And isrc1 , isrc2, idest

“高位与”： Andh const , isrc2, idest

将16位立即数const与寄存器isrc2的高16位进行“与”操作, 结果→ idest

(2) “或”

“或”： or isrc1 , isrc2, idest

“高位或”： orh const , isrc2, idest

(3) “异或”

“异或”： xor isrc1 , isrc2, idest

“高位异或”： xorh const , isrc2, idest

注：i860未提供一些实现简单功能的指令。

如：整数寄存器之间的数据传送、立即数送某个寄存器等

逻辑运算指令例：

① 完成：1234H→r5

指令：or 1234 , r0, r5

② 完成：12345678H→r5

指令：or 5678 , r0, r5
orh 1234, r5, r5

③ 完成： r2→r3

指令：or r0, r2, r3

6. 移位指令 左移：shl isrc1, isrc2, idest

完成：isrc2左移→ idest
isrc1指示移位次数(R或imm)

移位指令例：
shl 15, r16, r17
shl r1, r16, r17
shl r0, r16, r17

(二) 浮点指令

浮点指令中的参数不能有立即数。

1. 浮点加减　　加法：fadd.p　fsrc1, fsrc2, fdest
　　减法：fsub.p　fsrc1, fsrc2, fdest

p=ss/sd/dd

s表示操作数位数32位
d表示操作数位数64位

例1: fadd.sd　f6, f7, f18　完成: f6+f7 → f18:f19

例2: fsub.dd　f6, f10, f18　完成: f6:f7 – f10:f11→f18:f19

2. 浮点乘法　　fmul.p　fsrc1, fsrc2, fdest

3. 浮点倒数　　frcp.p　fsrc2, fdest

4. 浮点平方根　　freq.p　fsrc2, fdest

用i860指令系统编程例:

第三章例1:计算$(x_1+y_1)(x_2+y_2)(x_3+y_3)$, 结果存入内存单元。假设六个参数已分别在$R_0$~$R_5$寄存器中; 四级流水线, 仅在乘法运算阶段需要3Δt, 其余阶段为1Δt。

```
Add  R0, R1
Add  R2, R3
Add  R4, R5
Mul  R0, R2
Mul  R0, R4
Mov (M), R0
```

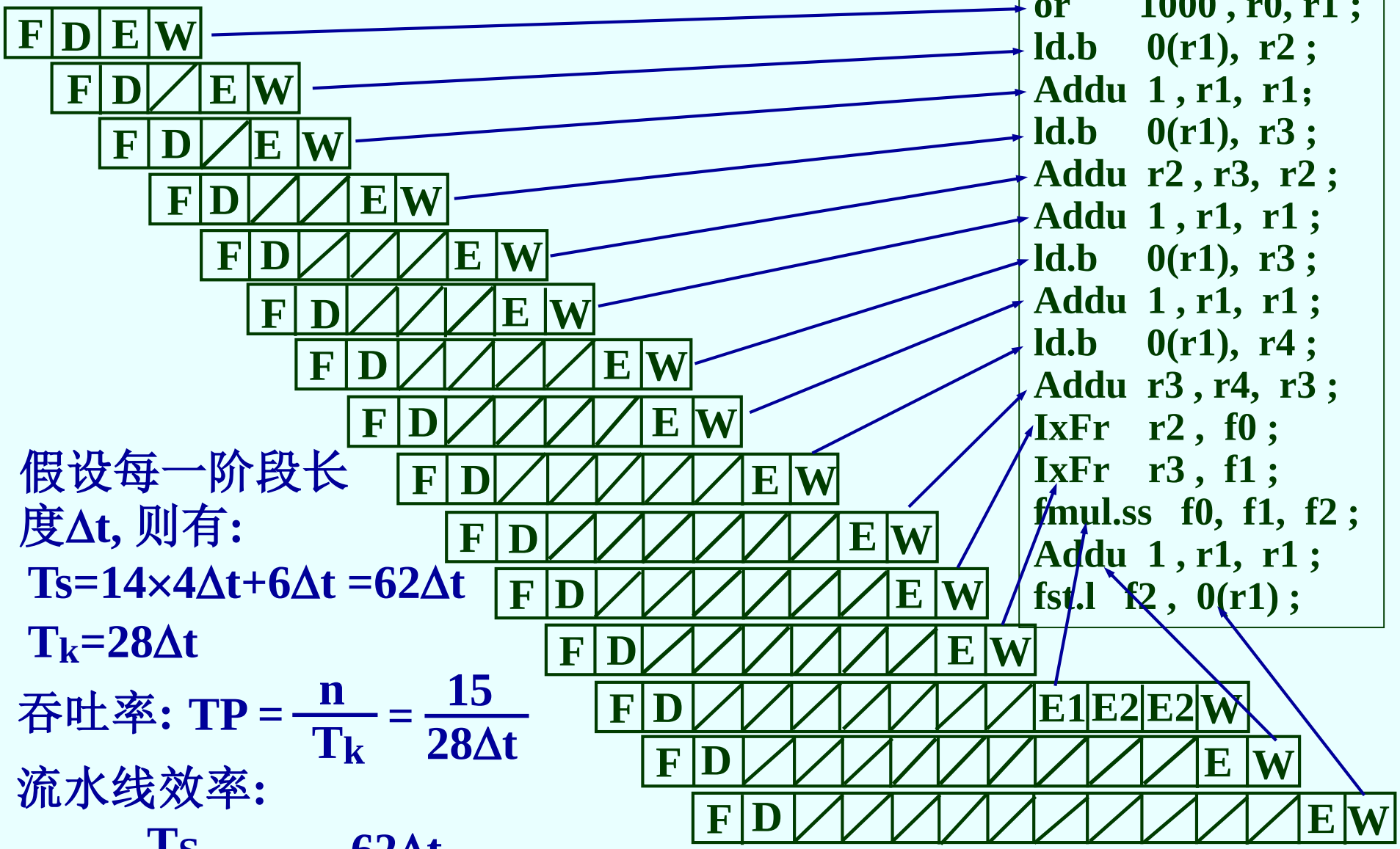
现假设: 计算$(x_1+y_1)(x_2+y_2)$, 源操作数为8位无符号数, 运算结果不超过32位。四个源数据已在内存00001000~00001003 地址单元, 结果存入00001004单元。

1. 采用i860指令系统

```
or       1000 , r0,  r1 ;   1000→r1
ld.b     0(r1),  r2 ;       x1 → r2
Addu     1 , r1,  r1;       地址加1
ld.b     0(r1),  r3 ;       y1 → r3
Addu     r2 , r3,  r2 ;     x1+y1→r2
Addu     1 , r1,  r1 ;      地址加1
ld.b     0(r1),  r3 ;       x2 → r3
Addu     1 , r1,  r1 ;      地址加1
ld.b     0(r1),  r4 ;       y2 → r4
Addu     r3 , r4,  r3 ;     x2+y2→r3
IxFr     r2 ,  f0 ;         x1+y1→f0
IxFr     r3 ,  f1 ;         x2+y2→ f1
fmul.ss  f0,  f1,  f2;      (x1+y1 )(x2+y2)→f2
Addu     1 , r1,  r1 ;      地址加1
fst.l    f2 ,  0(r1) ;      (x1+y1 )(x2+y2)→1004
```

计算$(x_1+y_1)(x_2+y_2)$, 源操作数为8位无符号数, 运算结果不超过32位。4个源数据已在内存1000~ 1003 地址单元, 结果存入1004单元。

注: 也可直接将数据取到浮点寄存器, 完成加乘运算, 避免整数寄存器与浮点寄存器之间的传输, 但浮点加法需要三个执行周期。



假设每一阶段长度Δt, 则有:

$T_S = 14 \times 4\Delta t + 6\Delta t = 62\Delta t$

$T_k = 28\Delta t$

吞吐率: $TP = \dfrac{n}{T_k} = \dfrac{15}{28\Delta t}$

流水线效率:

$$E = \frac{T_S}{k \times T_k} = \frac{62\Delta t}{4 \times 28\Delta t} = 0.5535$$

如果完全按流水线效率的定义来计算 (定义: 流水线中的功能部件的利用率, 其值为流水线功能部件的实际使用时间与整个运行时间之比) :

取指部件使用时间15Δt, 则e1=15Δt/28Δt = 0.535

译码部件使用时间15Δt, 则e2=15Δt/28Δt = 0.535

执行部件使用时间17Δt, 则e3=17Δt/28Δt = 0.607

写数部件使用时间15Δt, 则e4=15Δt/28Δt = 0.535

则整个流水线的效率:

$$E=(0.535+0.535+0.607+0.535)/4 = 0.553$$

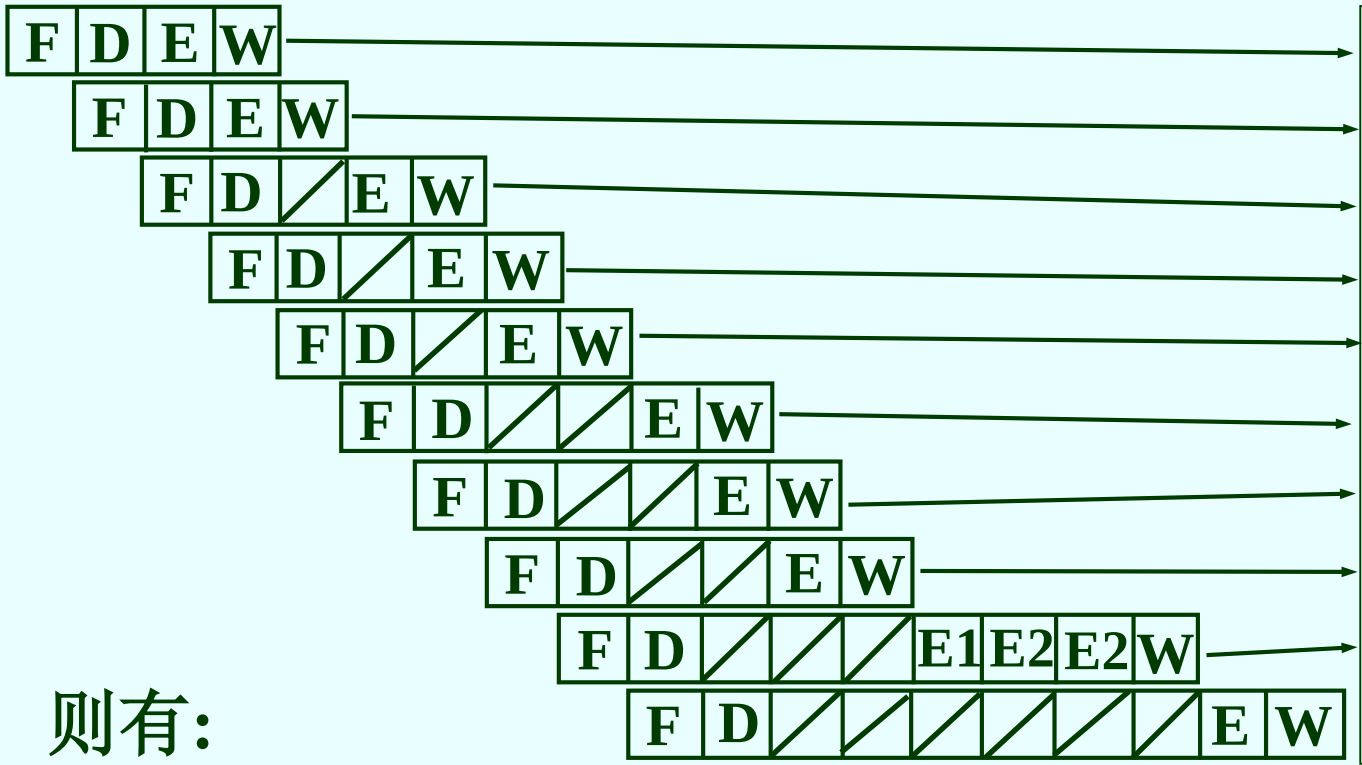
$$\sum = 62\Delta t = Ts$$

对比效率公式: $E= Ts/k\times T_k$

如果上述程序中的相应位置，采用直接寻址，如：“ld.b 1000(r0), r2”，则可减少指令数量，流水线过程以及一些指标也可能随之变化。

```
ld.b    1000(r0), r2 ;
ld.b    1001(r0), r3 ;
Addu  r2 , r3,  r2 ;
ld.b    1002(r0), r3 ;
ld.b    1003(r0), r4 ;
Addu  r3 , r4,  r3 ;
IxFr    r2 ,  f0 ;
IxFr    r3 ,  f1 ;
fmul.ss   f0,  f1,  f2;
fst.l   f2 ,  1004(r0) ;
```

共计10条指令




```
ld.b    1000(r0), r2
ld.b    1001(r0), r3
Addu  r2 , r3,  r2
ld.b    1002(r0), r3
ld.b    1003(r0), r4
Addu  r3 , r4,  r3
IxFr    r2 ,  f0
IxFr    r3 ,  f1
fmul.ss   f0,  f1,  f2
fst.l   f2,  1004(r0)
```

则有:

$T_S=9\times4\Delta t+6\Delta t=42\Delta t$, $T_k=19\Delta t$

吞吐率: $TP=\frac{n}{T_k}=\frac{10}{19\Delta t}$

流水线效率: $E=\frac{T_S}{k\times T_k}=\frac{42\Delta t}{4\times19\Delta t}=0.5526$

对照前一个程序:

$TP=\frac{15}{28\Delta t}$

效率$E=0.5535$

由此可以看出:

(1) 流水线处理器中, 数据相关是导致流水线阻塞的重要原因, 因此, 乱序执行或优化编译尤为重要;

(2) 程序中指令的调整, 将影响流水线运行时间;

(3) 流水线运行时间的减少并不一定能提高吞吐量和流水线效率。

2. 如果用80×86指令系统, 在32位系统上运行(如80486或Pentium基本型):

仍然存在数据相关、乘法也须要三个执行周期。

流水线为深度五级, 如果采用寄存器和内存单元直接相加指令如“ADD (Men), R”或“ADD (R_i), R_j”, 则需要2个执行周期(一次访存和做加法)等。

例2: 一种32位处理器(32位数据和32位地址总线), 按取指、译码、执行、写结果四级流水线方式。假设每条指令仅一个执行周期可完成执行阶段的操作, 且无资源冲突。对下面的一段程序, 试分析该程序段流水线执行情况(图示)。该流水线是否存在周期阻塞的可能？若存在, 请予以说明为什么, 并指出该程序段执行完成共需要多少个周期。

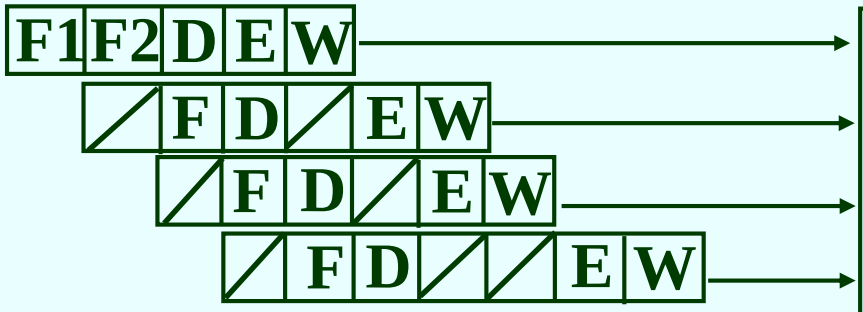
LD 2000(R1), R2 ; 变址寻址(2000+R1)→ R2 (5个字节)

ADD R2, R3, R4 ; R2+R3→ R4 (3个字节)

OR R5, R6, R2 ; R5∨R6→ R2 (3个字节)

MUL R2, R8, R9 ; R2×R8→ R9 (4个字节)

(1) 流水线执行过程



LD 2000(R1), R2; 5个字节
ADD R2, R3, R4; 3个字节
OR R5, R6, R2; 3个字节
MUL R2, R8, R9; 4个字节

(2) 流水线存在周期阻塞

① 第一条指令5个字节需要两个取指周期, 导致第二条指令取指延迟一个周期;

② 第二条指令与第一条指令存在数据相关, 使第二条指令的执行延迟一个周期;

③ 第四条指令与第三条指令存在数据相关,使第四条指令的执行延迟一个周期;

(3) 流水线执行完成共需10个周期

对比：ARM处理器基本指令集

1、ARM的基本寻址方式

1. 寄存器寻址

如：ADD R_0, R_1, R_2 ; $R_0 \leftarrow R_1+R_2$

2. 立即寻址

如ADD $R_3, R_3, \#2$; $R_3 \leftarrow R_3+2$

3. 寄存器间接寻址

如 LDR $R_0, [R_1]$; $R_0 \leftarrow [R_1]$

4. 变址寻址

如 LDR $R_0, [R_1, \#4]$; $R_0 \leftarrow [R_1+4]$

5. 堆栈寻址

使用专用寄存器(R_{13})对堆栈进行存取。

6. 多寄存器寻址

一次可传送多个寄存器的值，如一条指令传送16个寄存器的任何子集：LDMIA R_1, {R_0, R_2, R_5}

完成：$R_0 \leftarrow [R_1]$
$R_2 \leftarrow [R_1+4]$
$R_5 \leftarrow [R_1+8]$

传送的数据为32位，因此该指令将R_1所指向的连续存储单元内容送寄存器R_0、R_2、R_5

7. 寄存器移位寻址(ARM特有的一种寻址方式)

如 ADD R_3, R_2, R_1, LSL#3

将R_1的内容逻辑左移3位后与R_2相加，结果存入R_3，即$R_3 \leftarrow R_2 + R_1 \times 8$。

8. 块拷贝寻址

如 LDMIA R_0!, {R_2 – R_9}

STMIA R_1, {R_2 – R_9}

执行时, R_0自动寻址8个字, 并拷贝到R_1指向的连续存储单元。R_2~R_9作缓存(若R_2~R_9中存放有有用的值, 则应压堆栈保存)。

9. 相对寻址

由PC提供基准地址, 指令中的地址码字段作为偏移量, 两者相加得到操作数有效地址。

二、ARM指令集分类

1、访存指令LDR和STR(涉及若干种数据类型)

2、数据处理指令:含算术运算、逻辑运算、移位、寄存器传送、比较、测试等指令。

3、分支指令(控制类指令)

4、协处理指令

5、杂项指令(如软中断指令)

6、伪指令

ARM是主要应用于嵌入式系统的RISC处理器,但精简指令集特征并不明显。这说明RISC处理器具有多样性,也说明随着技术的发展,不一定强调精简指令集,性能的提升可以从多个方面来进行。

附： SPARC规范

SPARC — Scalable Processor Architecture
可伸缩的处理器结构

从硬件和软件两个方面建议了RISC处理器硬件设计以及RISC系统软件配置的规范:

- RISC处理器结构
- 操作系统/系统服务程序
- 图形用户界面
- 文件格式及调用方式等

硬件的RISC特征

- 寄存器　大量通用寄存器, 其数量能支持上下切换;
- 重叠窗口管理
  SPARC的一大特点, 将其作为RISC处理器的特征。
- 存取式体系结构
- 定长指令系统
  为支持多机系统或多任务系统设置专门指令, 如 lock/unlock 总线锁定和开锁指令;
- 浮点单元
  支持单精度、双精度、四单精度
- 协处理单元CP
  根据设计目标自行定义。

1992年SUN的Super SPARC的性能指标:

Super SPARC	指 标
寄存器	136×32位整数寄存器
指令Cache	16KB
数据Cache	16KB
IPC	3
流水线级数	8
结构	超级标量
SPEC值	SPECint=40，SPECfp=45

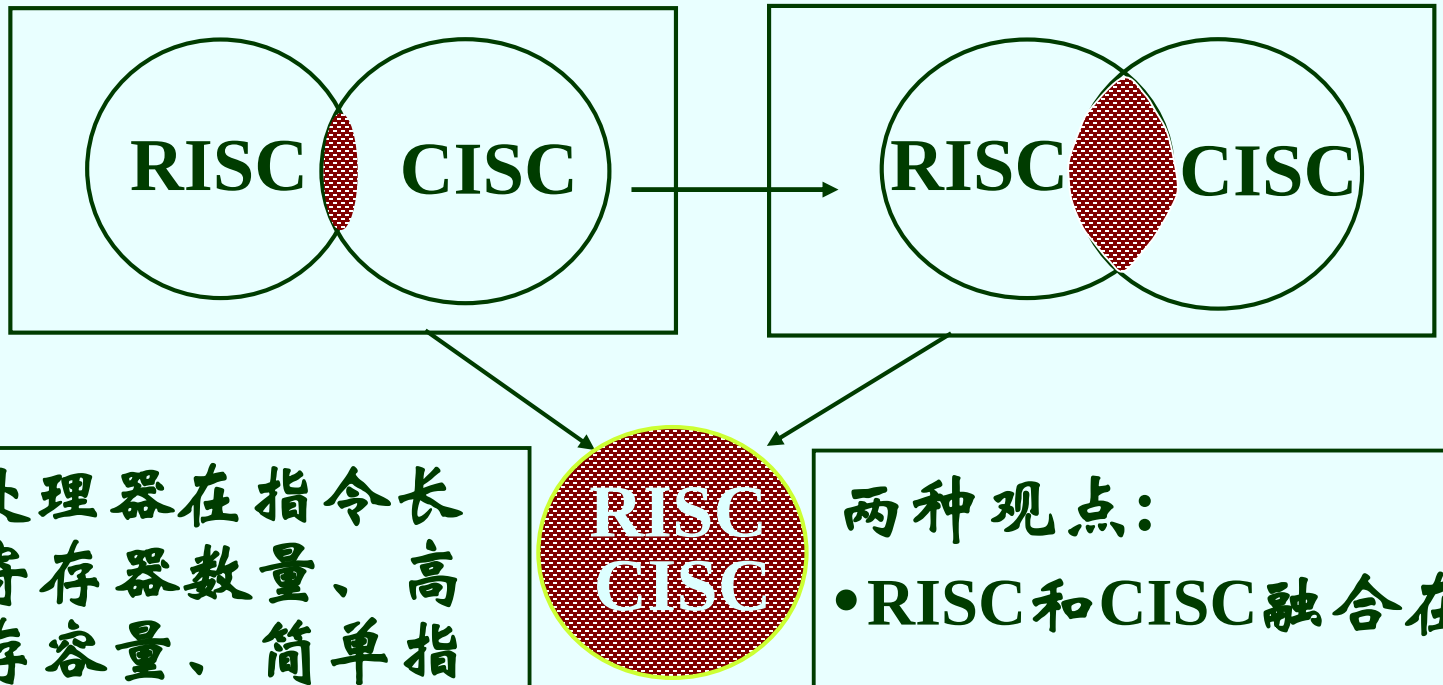
注：PentiumIII 550MHz
SPECint95=22.3， SPECfp95=15.6

近年来的研究指出:

指令执行方式以及处理器结构对处理器速度的影响远大于指令集本身。因此，RISC处理器的速度总是高于CISC处理器的速度的结论不一定正确。

随着处理器的发展，RISC变得越来越庞大和复杂，但处理器性能的提升越来越少。复杂度的迅速增长，使处理器技术又一次接近发展的瓶颈，成为处理器设计逐步向多核方向发展的原因之一。

* RISC和CISC的融合



64位处理器在指令长度、寄存器数量、高速缓存容量、简单指令的执行、解决流水线阻塞的方法等方面，两种不同的处理器设计模式已没什么差异

两种观点：

- RISC和CISC融合在64位
- 64位处理器终结RISC 结构