第 5 章 指令级并行及其开发——硬件方法

内容提要

(1) 指令级并行的概念;
(2) 相关与指令级并行;
(3) 指令的动态调度;
(4) 动态分支预测技术;
(5) 多指令流出技术。

指令级并行(Instruction-Level Parallelism,ILP)是指指令之间存在的一种并行性,利用它,计算机可以并行执行两条或两条以上的指令。开发 ILP 的途径有两种:一种是资源重复,重复设置多个处理部件,让它们同时执行相邻或相近的多条指令;另一种是采用流水线技术,使指令重叠并行执行。开发指令级并行性是提高计算机性能的一种重要方法,在近三十多年来推出的计算机中,几乎都采用了这种技术。

本章及第 6 章研究如何利用各种技术来开发更多的指令级并行。本章着重讨论硬件的方法,而第 6 章则介绍软件的方法。

5.1 指令级并行的概念

开发指令级并行的方法可以分为两大类:主要基于硬件的动态开发方法以及基于软件的静态开发方法。在实际应用中,这种区分不是绝对的,有些思想是两者都可以采用。特别是,为了能充分开发程序中潜在的指令级并行,应该把硬件技术与软件技术相结合,把动态方法与静态方法相结合。

流水线处理机的实际 CPI 等于理想流水线的 CPI 加上各类停顿的时钟周期数。

$$\text{CPI}_{\text{流水线}} = \text{CPI}_{\text{理想}} + \text{停顿}_{\text{结构冲突}} + \text{停顿}_{\text{数据冲突}} + \text{停顿}_{\text{控制冲突}}$$

其中,理想 CPI 是衡量流水线最高性能的一个指标。通过减少该式右边的各项,就能减少总的 CPI,从而提高 IPC。IPC 是 Instructions Per Cycle 的缩写,其含义是每个时钟周期完成的指令条数。它是 CPI 的倒数。

如果一串连续的代码除了入口和出口以外,没有其他的分支指令和转入点,则称之为一个基本程序块(Basic Block)。由于程序中往往每隔 4～7 条指令就会有一个分支,而且指令之间还可能存在相关,因此在基本程序块中能开发出的并行性是很有限的,很可能比基本块的平均大小要小得多。为了明显地提高性能,必须跨越多个基本块开发指令级并行。

最简单和最常用的增加指令之间的并行性的方法，是开发循环的不同迭代之间存在的并行性。这种并行性称为循环级并行性(Loop-Level Parallelism)。例如，考虑下述语句：

```
for (i=1; i<=500; i=i+1)
    a[i]=a[i]+s;
```

这里，在每一次循环的内部，都没有任何的并行性可言，但每一次循环都可以与其他的循环重叠并行执行。第6章将讨论如何把这种循环级并行性转换为ILP。

开发循环级并行性的重要方法是采用向量指令和向量数据表示。第4章已经对向量处理机做了比较详细的讨论。虽然向量处理思想的出现要早于许多后面将介绍的开发ILP的方法，但目前开发ILP的处理机在通用应用领域占统治地位，几乎是完全替代了向量处理机。当然，向量处理机在某些特定的应用领域还有广泛的应用，例如图形、数字信号处理、多媒体应用等。

5.2 相关与指令级并行

在第3章中讲过，确定程序中指令之间存在什么样的相关，对于确定程序中有多少并行性以及如何开发这些并行性有重要的意义。如果两条指令相关，它们就不能并行执行，或只能部分重叠执行。相关有三种类型：数据相关、名相关、控制相关。

流水线冲突(Hazard)是指对于具体的流水线来说，由于相关的存在，使得指令流中的下一条指令不能在指定的时钟周期执行。流水线冲突有三种类型：结构冲突、数据冲突、控制冲突。结构冲突是因硬件资源冲突造成的，数据冲突是由数据相关和名相关造成的，控制冲突是由控制相关造成的。

相关是程序固有的一种属性，它反映了程序中指令之间的相互依赖关系。而具体的一次相关是否会导致实际冲突的发生以及该冲突会带来多长的停顿，则是流水线的属性。

数据相关限制了所能开发的ILP，本章论述的主要内容之一就是如何克服这些限制。可以从以下两个方面来解决相关问题。

(1) 保持相关，但避免发生冲突。

(2) 进行代码变换，消除相关。

指令调度就是一种用来避免冲突的主要方法，它并不改变相关。第3章已经介绍了一些依靠编译器静态地进行指令调度的方法，本章将论述动态调度代码的硬件方法。我们还将看到，有些类型的相关是可以消除的。

由于相关的存在，必须保持所谓的程序顺序。程序顺序(Program Order)是指：由原来程序确定的在完全串行方式下指令的执行顺序。但是，并不需要在所有存在相关的地方都保持程序顺序。后面要介绍的各种软硬件技术的目标是尽可能地开发并行性，只有在可能会导致错误的情况下，才保持程序顺序。

对于提高性能来说，控制相关本身并不是一个主要的限制。当存在控制相关时，如果不遵守控制相关的依赖关系、执行本来不该执行的指令对程序的正确性没有影响，那么就完全可以这么做(如果有好处的话)，所以控制相关并不是一个必须严格保持的关键属性。为了保证程序执行的正确性，必须保持的最关键的两个属性是数据流和异常行为。

保持异常行为(Exception Behavior)是指：无论怎么改变指令的执行顺序，都不能改变

程序中异常的发生情况。即原来程序中是怎么发生的，改变执行顺序后应该还是那样发生。这个条件经常被弱化为：指令执行顺序的改变不能导致程序中发生新的异常。

数据流(Data Flow)是指数据值从其产生者指令到其消费者指令的实际流动。分支指令使得数据流具有动态性，因为一条指令有可能数据相关于多条先前的指令。分支指令的执行结果决定了哪条指令真正是所需数据的产生者。

有时，不遵守控制相关既不影响异常行为，也不改变数据流。这时就可以大胆地进行指令调度，把失败分支中的指令调度到分支指令前。例如，考虑下面这个例子：

```
          DADDU  R1,R2,R3
          BEQZ   R12,Skipnext
          DSUBU  R4,R5,R6
          DADDU  R5,R4,R9
Skipnext: OR     R7,R8,R9
```

假设我们知道R4在Skipnext之后不再被使用，而且DSUBU不会产生异常，那么就可以把DSUBU移到分支指令之前。这是因为这个移动不会改变数据流。如果分支成功，尽管按原程序DSUBU指令是不该执行的，但现在执行了也没关系，因为它不会影响程序的执行结果。

后面将讨论的前瞻执行不仅能解决异常问题，而且能够在保持数据流的情况下，减少控制相关对开发ILP的影响。

5.3 指令的动态调度

静态调度的流水线依靠编译器对代码进行静态调度，以减少相关和冲突。之所以称为静态调度(Static Scheduling)，是因为它不是在程序执行的过程中，而是在编译期间进行代码调度和优化的。静态调度通过把相关的指令拉开“距离”来减少可能产生的停顿。

第3章中论述的流水线属于静态调度的流水线。在这样的流水线中，当取出的指令与已经在流水线中执行的指令不存在数据相关，或者虽存在数据相关，但可以通过定向机制将相关隐藏时，就可以流出这条指令。如果数据相关不能被隐藏，冲突检测硬件就会从使用该数据的指令开始，使流水线停顿(Stall)，不再取指令和流出指令。

与静态调度不同，动态调度(Dynamic Scheduling)是在程序的执行过程中，依靠专门硬件对代码进行调度。这是一种很重要的技术，许多现代的处理器都采用了这种技术。动态调度能在保持数据流和异常行为的情况下，通过硬件对指令执行顺序进行重新安排，减少数据相关导致的停顿。动态调度有许多优点：①能够处理一些编译时情况不明的相关(比如涉及存储器访问的相关)，并简化了编译器；②能够使本来是面向某一流水线优化编译的代码在其他动态调度的流水线上也能高效地执行。当然，动态调度的这些优点是以硬件复杂性的显著增加为代价的。

扫一扫

视频讲解

5.3.1 动态调度的基本思想

第3章中讨论的5段流水线有一个很大的局限性，即其指令是按序流出(In-order Issue)和按序执行(In-order Execution)的。如果某条指令在流水线中被停顿了，其后面所有的指令也就都停止前进了。如果系统中有多个功能部件，那么这些部件就很可能因为没

有指令可处理而处于空闲状态，系统效率低下。

考虑下述例子：

```
DIV.D  F4,F0,F2
ADD.D  F10,F4,F8
SUB.D  F12,F6,F14
```

ADD.D 指令与 DIV.D 指令关于 F4 相关，导致流水线停顿。SUB.D 指令也因此受阻，而实际上它与流水线中的任何指令都没有关系。这是按序流出和按序执行带来的局限性。如果可以不按程序顺序执行指令，就能够进一步提高性能。

在第 3 章的流水线中，结构冲突和数据冲突都是在译码（ID）段进行检测的。只有当既没有结构冲突也没有数据冲突时，指令才能够流出。为了使上述指令序列中的 SUB.D 指令能继续执行下去，必须把指令流出的工作拆分为以下两步。

（1）检测结构冲突。

（2）等待数据冲突消失。

只要检测到没有结构冲突，就可以让指令流出。并且流出后的指令一旦其操作数就绪就可以立即执行。这样上面的 SUB.D 指令就不会被阻塞了。修改后的流水线是乱序执行（Out-of-order Execution）的，即指令的执行顺序与程序顺序不相同。同样，指令的完成也是乱序完成（Out-of-order Completion）的，即指令的完成顺序与程序顺序不相同。

为了支持乱序执行，将前述 5 段流水线（图 3.17，下同）的译码（ID）段细分为两个段。

（1）流出（issue）：指令译码，并检查是否存在结构冲突。如果不存在结构冲突，就将指令流出。

（2）读操作数：等待数据冲突消失（如果有的话），然后读操作数。

可以看出，指令的流出仍然是按序流出（In-order Issue）的。但是，它们在读操作数段可能停顿和互相跨越，因而进入执行段时就可能已经是乱序的了。

在前述 5 段流水线中，是不会发生 WAR 冲突和 WAW 冲突的。但现在的乱序执行就使得它们都可能发生了。例如，考虑下面的代码：

```
DIV.D  F10,F0,F2
ADD.D  F10,F4,F6
SUB.D  F6,F8,F14
```

SUB.D 指令与 ADD.D 指令存在反相关（关于 F6），如果流水线在 ADD.D 读出 F6 之前就完成 SUB.D 了，就会出现错误。类似地，ADD.D 与 DIV.D 存在输出相关（关于 F10），流水线必须能检测出该相关，并避免 WAW 冲突。在后面将看到，可以通过使用寄存器重命名来消除它们。

执行段紧跟在读操作数段之后，和前述 5 段流水线一样。在浮点流水线中，执行阶段可能需要多个时钟周期，因不同的运算而不同。

在后面的讨论中，将区分指令何时开始执行、何时结束执行。在开始执行与结束执行之间，指令是处于正在执行当中。采用动态调度的流水线支持多条指令同时处于执行当中，这是动态调度的一大优点。但这要求具有多个功能部件，或者功能部件流水化，或者两者兼而有之。这里假设具有多个功能部件。

指令乱序完成大大增加了异常处理的复杂度。动态调度的处理机是这样来保持正确的异常行为的：对于一条会产生异常的指令来说，只有当处理机确切地知道该指令将被执行时，才允许它产生异常。

即使保持了正确的异常行为，动态调度处理机仍可能发生不精确异常。所谓不精确异常(Imprecise Exception)是指：当执行指令 i 导致发生异常时，处理机的现场(状态)与严格按程序顺序执行时指令 i 的现场不同。反之，如果发生异常时，处理机的现场跟严格按程序顺序执行时指令 i 的现场相同，就称为精确异常(Precise Exception)。不精确异常使得在异常处理后难以接着继续执行程序。

之所以会发生不精确异常，是因为当发生异常(设为指令 i)时：①流水线可能已经执行完按程序顺序是位于指令 i 之后的指令；②流水线可能还没完成按程序顺序是指令 i 之前的指令。

记分牌算法和 Tomasulo 算法是两种比较典型的动态调度算法。下面先介绍记分牌算法，然后详细论述 Tomasulo 算法。Tomasulo 算法比记分牌算法改进了许多，是一种更强的算法。许多开发指令级并行的现代处理机都采用了 Tomasulo 算法或其变形。

5.3.2 记分牌动态调度方法

1. 基本思想

记分牌(Scoreboard)方法这一名称起源于最早采用此功能的 CDC 6600 计算机。该机器用一个称为记分牌的硬件实现了对指令的动态调度。该硬件中维护着三张表，分别用于记录指令的执行状态、功能部件状态、寄存器状态以及数据相关关系等。它把前述 5 段流水线中的译码段 ID 分解成了两个段：流出和读操作数，以避免当某条指令在 ID 段被停顿时挡住后面无关指令的流动。

记分牌的目标是在没有结构冲突时，尽可能早地执行没有数据冲突的指令，实现每个时钟周期执行一条指令。如果某条指令被暂停，而后面的指令与流水线中正在执行或被暂停的指令都不相关，那么这些指令就可以跨越它们，继续流出和执行下去。记分牌全面负责和管理这些指令的流出和执行，当然也包括检测所有的冲突。

由于允许指令相互跨越，所以尽管所有的指令在流出段都是按序流出的，但它们却可能是乱序执行和乱序完成的。这有利于提高处理器的性能，因为乱序执行和完成消除了许多等待。

要发挥指令乱序执行的好处，必须有多条指令同时处于执行阶段。CDC 6600 具有 16 个独立的功能部件：4 个浮点部件，5 个访存部件，7 个整数操作部件。在类似 MIPS 结构的处理器中，记分牌主要用于浮点部件才有意义，这是因为其他部件的操作延迟都很小。假设所考虑的处理器有两个乘法器、一个加法器、一个除法部件和一个整数部件，整数部件用来处理所有的存储器访问、分支处理和整数操作。尽管这个例子比 CDC 6600 简单，但它足以阐明记分牌的基本工作原理。由于 MIPS 和 CDC 6600 都采用 load-store 结构，所以它们采用的技术非常类似。图 5.1 是采用记分牌的 MIPS 处理器的基本结构。

每条指令都要经过记分牌。记分牌负责相关检测并控制指令的流出和执行。指令流出时，记分牌在表中记录相关的信息，并决定什么时候该指令可以读出操作数和开始执行。如

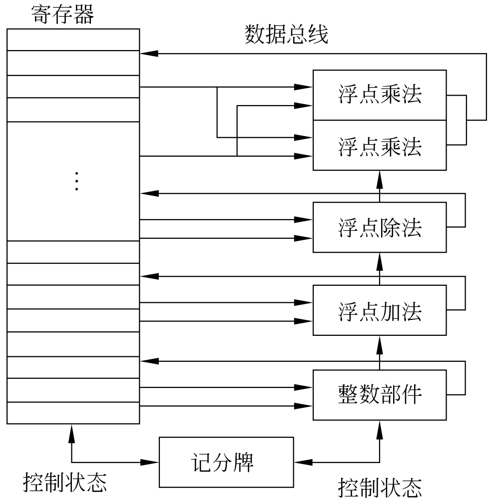

图 5.1　采用记分牌的 MIPS 处理器的基本结构

果确定该指令不能马上执行，记分牌就会监视硬件中信息的每一个变化，一旦所需的操作数就绪，就立即启动该指令的执行。

下面来看看指令在采用记分牌的流水线中的处理步骤。每条指令的执行过程分为 4 段：流出、读操作数、执行和写结果。由于主要考虑浮点操作，运算在浮点寄存器上进行，因此不涉及存储器访问段。

1）流出

如果当前流出指令所需的功能部件空闲，并且所有其他正在执行的指令的目的寄存器与该指令的不同，记分牌就向功能部件流出该指令，并修改记分牌内部的记录表。

可以看出，在记分牌方法中，如果存在结构相关或 WAW 冲突，该指令就不流出。在这里就解决了 WAW 冲突。当然，该指令若不流出，也就阻止了后面指令的流出。

2）读操作数

记分牌监测源操作数的可用性，如果数据可用，它就通知功能部件从寄存器中读出源操作数并开始执行。这一步动态地解决了 RAW 冲突，并导致指令可能乱序开始执行。那么如何判断数据可用呢？对于给定的寄存器来说，如果所有前面已流出且还在执行的指令都不对该寄存器进行写操作，那么该寄存器中的数据就是可用的了。显然，如果有正在执行的指令要对其进行写入，就要等待，直到相应的功能部件完成对这个寄存器的写操作。

上述流出和读操作数段合起来就完成了前述 5 段流水线的译码段（ID）的功能。

3）执行

取到操作数后，功能部件开始执行。当产生出结果后，就通知记分牌它已经完成执行。这一步相当于前述 5 段流水线中的执行段（EX）。在浮点流水线中，这一段可能要占用多个时钟周期。

4）写结果

记分牌一旦知道执行部件完成了执行，就检测是否存在 WAR 冲突。如果不存在，或者原有的 WAR 冲突已消失，记分牌就通知功能部件把结果写入目的寄存器，并释放该指令使用的所有资源。这一步对应于前述 5 段流水线中的写回段（WB）。

如果检测到 WAR 冲突,就不允许该指令将结果写到目的寄存器,这发生在以下情况。

前面的某条指令(按顺序流出)还没有读取操作数,而且其某个源操作数寄存器与本指令的目的寄存器相同。

在这种情况下,记分牌必须等待,直到该冲突消失。

记分牌是通过与功能部件的通信来控制指令的逐步执行。由于功能部件与寄存器组之间只有很有限的几条总线连接,所以这里有可能会发生结构冲突。记分牌要保证在上述第2)和第4)步中,所允许同时执行的功能部件的个数不超过可用总线的条数。CDC 6600 是这样来解决这个问题的:把它所拥有的功能部件分成4组,然后给每一组配备一套总线(两条“入”,一条“出”)。在每个时钟周期,每一组中只有一个设备可以进行读操作数或者写结果。

记分牌中记录的信息由三部分构成。

(1) 指令状态表:记录正在执行的各条指令已经进入到了哪一段。

(2) 功能部件状态表:记录各个功能部件的状态。每个功能部件有一项,每一项由以下9个字段组成。

Busy:忙标志,指出功能部件是否忙。初值为 no。

Op:该功能部件正在执行或将要执行的操作。

F*i*:目的寄存器编号。

F*j*,F*k*:源寄存器编号。

Q*j*,Q*k*:指出向源寄存器 F*j*、F*k* 写数据的功能部件(即 F*j*、F*k* 中的数据将由它们产生)。

R*j*,R*k*:标志位,为 yes 表示 F*j*,F*k* 中的操作数就绪且还未被取走,否则就被置为 no。

(3) 结果寄存器状态表 Result:每个寄存器在该表中有一项,用于指出哪个功能部件(编号)将把结果写入该寄存器。如果当前正在运行的指令都不以它为目的寄存器,则其相应项置为 no。Result 各项的初值为 no(全零)。

2. 举例

下面详细分析如图5.1所示的 MIPS 记分牌所要维护的数据结构。图5.2给出了下列代码运行过程中记分牌保存的信息。其中 Integer 表示整数部件,Mult1 和 Mult2 表示乘法部件,Add 表示加法部件,Divide 表示除法部件。

```
L.D     F6,34(R2)
L.D     F2,45(R3)
MULT.D  F0,F2,F4
SUB.D   F8,F6,F2
DIV.D   F10,F0,F6
ADD.D   F6,F8,F2
```

在图5.2的指令状态表中,第一条 L.D 指令已经完全执行完,其结果已经写入了 F6;第二条 L.D 指令也已经执行完,但是结果还没有写入目的寄存器 F2;由于第二条 L.D 指令与 MULT.D 和 SUB.D 指令之间存在关于寄存器 F2 的先写后读(RAW)相关,因此 MULT.D 和 SUB.D 在流出段等待,不能进入流水线的读操作数段。同样,MULT.D 与 DIV.D 之间存在关于寄存器 F0 的 RAW 相关,因此 DIV.D 也只能在流出段等待。指令 ADD.D 与指令 SUB.D 之间存在关于加法器的结构相关,因此后面的 ADD.D 连流出都不能做,必须等到前面 SUB.D 指令全部执行完毕、释放加法器后才能够流出。

指令		指令状态表			
		流出	读操作数	执行	写结果
L.D	F6,34(R2)	√	√	√	√
L.D	F2, 45(R3)	√	√	√	
MULT.D	F0, F2, F4	√			
SUB.D	F8, F6, F2	√			
DIV.D	F10, F0, F6	√			
ADD.D	F6, F8, F2				

部件名称	功能部件状态表								
	Busy	Op	F*i*	F*j*	F*k*	Q*j*	Q*k*	R*j*	R*k*
Integer	yes	L.D	F2	R3				no	
Mult1	yes	MULT.D	F0	F2	F4	Integer		no	yes
Mult2	no								
Add	yes	SUB.D	F8	F6	F2		Integer	yes	no
Divide	yes	DIV.D	F10	F0	F6	Mult1		no	yes

	结果寄存器状态表							
	F0	F2	F4	F6	F8	F10	…	F30
部件名称	Mult1	Integer			Add	Divide		

图 5.2　MIPS 记分牌中保存的信息

关于图 5.2 中的功能部件状态表,先来看第一行的情况。整数(Integer)部件的 Busy 字段为 yes,表示部件正忙,从其 Op 字段可以知道它在执行 L.D 指令,目的寄存器字段 F*i* 中记录的是 F2。相应地,在结果寄存器状态表的 F2 字段中记录的是 Integer 部件。对于存储器访问类指令来说,第一源操作数寄存器字段 F*j* 中记录的应该是访存地址寄存器。在本例中就是 R3。它的 R*j* 字段为 no,表示 R3 的数据已经被读取过了。

接下来看看功能部件状态表中第二行的各字段。在乘法 1(Mult1)部件的 Busy 字段也是 yes,表示正忙。Op 字段记录的是 MULT.D 指令,其目的寄存器字段 F*i* 的内容为 F0。相应地,在结果寄存器状态表的 F0 字段中记录的是 Mult1,表示 F0 将存放 Mult1 部件的运算结果。在第一源操作数寄存器字段 F*j* 中记录的是 F2,它的 Q*j* 字段非空,为 Integer,表示 F2 的数据将来自 Integer 部件的操作结果,它的 R*j* 字段为 no,表示 F2 的数据还没有就绪,这个过程可以用来判断并解决数据的写后读相关;第二源操作数寄存器字段 F*k* 中记录的是 F4,Q*k* 字段为空,表示 F4 不依赖于当前工作的任何部件,R*k* 字段为 yes,表示 F4 的数据已经就绪。乘法 2(Mult2)部件的 Busy 字段是 no,表示该功能部件当前空闲。

其他部件的状态字段分析与上述描述类似,就不一一介绍了。

结果寄存器状态表中的字段与每个寄存器一一对应,它记录了当前机器状态下将把结果写入该寄存器的功能部件的名称。在图 5.2 中,当前写 F0 的为 Mult1 部件,写 F2 的为 Integer 部件,写 F8 的为 Add 部件,写 F10 的为 Divide 部件。字段为空表示空闲,即对应的寄存器没有被任何当前正在工作的功能部件作为目的寄存器使用。

下面来看一看图 5.2 中的指令序列如何往下执行。

例 5.1　假设浮点流水线中各部件的延迟如下。

加法需 2 个时钟周期;

乘法需 10 个时钟周期;

除法需 40 个时钟周期。

代码段和记分牌信息的起始点状态如图 5.2 所示。分别给出 MULT.D 和 DIV.D 准备写结果之前的记分牌状态。

解 图 5.2 中的代码段存在以下相关性。

(1) 先写后读(RAW)相关:第二条 L.D 指令到 MULT.D 和 SUB.D 之间,MULT.D 到 DIV.D 之间,SUB.D 到 ADD.D 之间。

(2) 先读后写(WAR)相关:DIV.D 和 ADD.D 之间,SUB.D 和 ADD.D 之间。

(3) 结构相关:ADD.D 和 SUB.D 指令关于浮点加法部件。

图 5.3 和图 5.4 分别给出了 MULT.D 指令和 DIV.D 指令将要写结果时记分牌的状态。

指令		指令状态表			
		流出	读操作数	执行	写结果
L.D	F6, 34(R2)	√	√	√	√
L.D	F2, 45(R3)	√	√	√	√
MULT.D	F0, F2, F4	√	√	√	
SUB.D	F8, F6, F2	√	√	√	√
DIV.D	F10, F0, F6	√			
ADD.D	F6, F8, F2	√	√	√	

部件名称	功能部件状态表								
	Busy	Op	F*i*	F*j*	F*k*	Q*j*	Q*k*	R*j*	R*k*
Integer	no								
Mult1	yes	MULT.D	F0	F2	F4			no	no
Mult2	no								
Add	yes	ADD.D	F6	F8	F2			no	no
Divide	yes	DIV.D	F10	F0	F6	Mult1		no	yes

	结果寄存器状态表							
	F0	F2	F4	F6	F8	F10	…	F30
部件名称	Mult1			Add		Divide		

图 5.3 程序段执行到 MULT.D 将要写结果时记分牌的状态

从图 5.3 中可以知道,在 MULT.D 准备写结果之前,由于 DIV.D 指令与 MULT.D 指令之间存在关于寄存器 F0 的 RAW 相关,因此在 MULT.D 指令完成写结果之前,DIV.D 指令被阻塞在流出段而无法进入读操作数段。同时由于 ADD.D 指令与 DIV.D 指令之间存在关于寄存器 F6 的 WAR 相关,因此在 DIV.D 指令从 F6 中读出操作数之前,ADD.D 指令被阻塞在执行段,无法进入写结果段。

在图 5.4 中,DIV.D 准备写结果之前,这条指令前面的指令已经全部执行完毕,由于 DIV.D 指令执行需要 40 个时钟周期,其后的 ADD.D 指令只需要两个时钟周期,而且 DIV.D 和 ADD.D 之间存在的 WAR 相关在 DIV.D 读走源操作数后就已经消失了,因此 ADD.D 有足够的时间完成所有操作,所以就只剩下 DIV.D 指令没有完成写结果操作。

3. 具体算法

现在来看一看记分牌是如何控制指令执行的。指令在流水线中前进是有条件的,而且

指令		指令状态表			
		流出	读操作数	执行	写结果
L.D	F6, 34(R2)	√	√	√	√
L.D	F2, 45(R3)	√	√	√	√
MULT.D	F0, F2, F4	√	√	√	√
SUB.D	F8, F6, F2	√	√	√	√
DIV.D	F10, F0, F6	√	√	√	
ADD.D	F6, F8, F2	√	√	√	√

部件名称	功能部件状态表								
	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	no								
Mult1	no								
Mult2	no								
Add	no								
Divide	yes	DIV.D	F10	F0	F6			no	no

	结果寄存器状态表							
	F0	F2	F4	F6	F8	F10	…	F30
部件名称						Divide		

图 5.4　程序段执行到 DIV.D 将要写结果时记分牌的状态

在前进时记分牌必须记录有关的信息。下面给出每条指令在流水线中进入各段的条件以及所进行的记分牌内容的修改工作。为了区分寄存器的名字和寄存器的值，约定寄存器的内容表示为 Regs[S]的形式，其中的 S 是寄存器的名字。所以，Fj[FU]←S1 表示将寄存器名 S1(而不是寄存器 S1 中的内容)送入 Fj[FU]。

在下面的算法中，约定

FU：当前指令所要用的功能部件。

D：目的寄存器的名称。

S1、S2：源操作数寄存器的名称。

Op：要进行的操作。

Fj[FU]：功能部件 FU 的 Fj 字段(其他字段以此类推)。

Result[D]：结果寄存器状态表中与寄存器 D 相对应的内容。其中存放的是将结果写入寄存器 D 的功能部件的名称。

1）指令流出

```
(1) 进入条件。
    not Busy[FU] & not Result[D];   //功能部件空闲且没有写后写(WAW)冲突
(2) 记分牌内容修改。
    Busy[FU]←yes;                   //把当前指令的相关信息填入功能部件状态表
                                    //功能部件状态表中各字段的含义见前面
    Op[FU]←Op;                      //记录操作码
    Fi[FU]←D;                       //记录目的寄存器编号
    Fj[FU]←S1;                      //记录第一个源寄存器编号
    Fk[FU]←S2;                      //记录第二个源寄存器编号
    Qj[FU]←Result[S1];              //记录将产生第一个源操作数的部件
    Qk[FU]←Result[S2];              //记录将产生第二个源操作数的部件
```