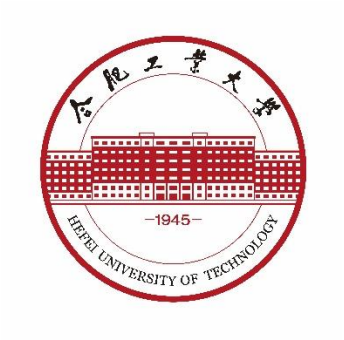
合肥工業大學



2023~2024 学年 第一学期

《计算机体系结构》课程报告

【基于流水线的指令调度技术】

学 号 2021214710

姓 名 杨程锦

2023 年 11 月

目 录

1 概述

随着计算机的广泛应用，流水线的指令调度技术已成为提高计算机性能的重要手段。然而，对于大多数用户来说，计算机并非总是完美的助手。它常常面临诸如硬件问题、软件问题、CPU 性能、主板性能以及存储器限制等让人烦恼的挑战。同时，随着程序规模的增大，计算机的运行速度却变得越来越慢。

为了应对这些问题，流水线的指令调度技术发挥了重要作用。它通过优化指令的执行顺序，最大程度地提高 CPU 的利用率和吞吐量，从而改善计算机的性能。通过合理安排指令的调度顺序，可以减少流水线的停顿和空闲时间，提高指令的执行效率。

类似于虚拟存储技术，流水线的指令调度技术也是一种智能地管理计算机指令执行的方式。它能够充分利用流水线的优势，如并行执行和指令重排序，以提高计算机的整体性能。指令调度技术可以将指令按照优先级和相关性进行排序，以最大程度地减少数据相关性和控制相关性带来的延迟，从而加快程序的执行速度。

因此，流水线的指令调度技术类似于虚拟存储技术，它们都通过智能管理和优化资源的方式，改善计算机系统的性能和效率。流水线的指令调度技术在多机系统中的改进策略和基于网络的指令调度技术也是探索的方向，以进一步提升计算机的性能和可扩展性。

本篇报告主要介绍对流水线指令调度、静态调度、动态调度、Tomasulo 算法等方面深度学习所得体会。

1.1 指令调度技术的背景

流水线是一种通过将计算机指令划分为多个阶段并在不同阶段同时执行来提高计算机性能的技术。然而，指令之间存在数据相关性和控制相关性，这可能导致流水线停顿、资源浪费和性能下降。为了最大程度地利用流水线的潜力，需要一种智能的指令调度技术来优化指令的执行顺序，减少停顿和延迟。

1.2 指令调度的意义

指令调度在处理器设计中的主要目的是优化指令的执行顺序，以最大程度地提高处理器性能。在上述文中，提到了分支预测技术作为一种应对控制冒险的方法，而指令调度是在更广泛的范围内，通过调整指令的执行顺序来最小化不同类型的冒险，包括数据冒险和结构冒险。

具体来说，指令调度可以通过以下方式发挥作用：

解决数据冒险： 数据冒险是由于指令需要等待前一条指令的结果而导致的延迟。指令调度可以通过重排指令的执行顺序，将不依赖于前一条指令结果的指令优先执行，从而减少数据冒险的影响。

充分利用流水线： 处理器通常采用流水线方式执行指令，分为不同的阶段（取指、译码、执行等）。指令调度可以通过优化指令的执行顺序，确保在流水线中的各个阶段都得到充分利用，从而提高整体的吞吐率。

最小化分支冒险影响： 在文中提到了分支指令可能导致的控制冒险，而分支预测技术是一种解决方案。指令调度可以通过调整分支指令的位置，尽量减少分支预测错误时的影响，从而提高整体性能。

减少结构冒险： 结构冒险是由于硬件部件不足而导致指令无法继续执行。指令调度可以通过合理的安排指令的执行顺序，减少对硬件资源的竞争，从而缓解结构冒险的问题。

综上所述，指令调度的意义在于通过合理的规划和调整指令的执行顺序，最大程度地减少各种类型的冒险，提高处理器的性能和效率。这对于确保指令的流畅执行，最大限度地发挥处理器性能至关重要。

2 流水线指令调度

2.1 流水线的基本概念

流水线是一种将指令处理过程划分为多个阶段，使得多条指令可以同时在处理器中执行的组织结构。这种并行执行的方式可以显著提高处理器的吞吐率和性能。

典型的流水线阶段包括指令提取（Fetch）、指令译码（Decode）、执行（Execute）、访存（Memory Access）和写回（Write Back）。

2.2 指令调度在流水线中的作用

指令调度是一种优化技术，旨在最大程度地减小流水线中的空闲周期，使得处理器能够更有效地执行指令。其核心原理在于调整指令的执行顺序，以最小化数据相关性和依赖关系，从而减少数据冒险，提高流水线的吞吐率。

2.3 数据相关性分析

数据相关性分析是指对指令之间的数据依赖关系进行深入研究，以确定哪些指令依赖于其他指令的结果。主要包括以下几种数据相关性：

真依赖（True Dependency）： 后续指令需要前一条指令的结果作为输入。

假依赖（False Dependency）： 后续指令对前一条指令的结果没有实际依赖，但由于寄存器重命名等原因，导致数据相关性的存在。

输出相关性（Output Dependency）： 两条指令都依赖于相同的计算结果。

3 静态调度

静态调度，作为一种在编译阶段对程序进行指令顺序优化的技术，旨在通过重新安排指令的执行顺序，最小化流水线中的空闲周期，提高指令级并行性，从而优化程序性能。对于循环展开，它被广泛应用于提高程序并行性和执行效率。Allen 和 Kennedy 在他们的论文中指出：“在现代高性能处理器上，循环展开是一种改进性能的重要技术”[1]。对于指令调度，通过代码重排和调整以及控制流图的优化，可以最小化数据相关性，减小数据冒险的概率，提高整个系统的吞吐率。Chen 等人在论文中指出：“指令调度可以显著提高指令级并行性，从而提高整体性能”[2]。这些静态调度技术在编译器层面的优化中，对于提高程序执行效率和适应不同硬件架构的需求具有重要作用。

3.1 指令调度

指令调度是静态调度中的关键技术，通过重新安排指令的执行顺序，最小化流水线中的空闲周期，提高指令级并行性，优化程序性能。Kennedy 和 Allen 在其著作中指出：“指令调度是一项关键的编译优化，它可以通过减小数据相关性、降低数据冒险的概率，从而提高整个系统的吞吐率”[3]。通过代码重排、调整和控制流图的优化，指令调度有助于最小化分支带来的性能影响、降低数据冒险的延迟，并最大化指令的并行度。编译器通过对控制流图和数据相关性的分析，采用合适的调度算法，以提高程序在静态阶段的执行效率。

```
for (i=1; i<=1000; i++)
    x[i] = x[i] + s;
```

图 1

```
Loop:  L.D   F0,0  (R1)
       ADD.D         F4,F0,F2
       S.D           F4, 0  (R1)
       DADDIU   R1,R1,  #-8
       BNE           R1,R2,Loop
```

图 2

将图 1 代码改写为 MIPS 汇编代码如图 2 所示。

假设浮点流水线延迟如表 1 所示。

产生结果的指令	使用结果的指令	延迟（时钟周期数）
浮点计算	另一个浮点计算	**3**
浮点计算	浮点 store（S.D）	**2**
浮点 load（L.D）	浮点计算	**1**
浮点 load（L.D）	浮点 store（S.D）	**0**

表 1

不进行指令调度时，程序实际执行情况如图 3 所示。

```
Loop:  L.D   F0,0  (R1)
       (空转)
       ADD.D         F4,F0,F2
       (空转)
       (空转)
       S.D           F4, 0  (R1)
       DADDIU   R1,R1,  #-8
       (空转)
       BNE           R1,R2,Loop
       (空转)
```

图 3

接下来进行指令调度，将 DADDIU 指令调度到 L.D 指令和 ADD.D 指令之间的“空转”拍，将 BNE 指令调度到 ADD.D 指令的第二个“空转”拍中，并且对存储器地址偏移量进行调整。调度之后的结果如图 4 所示。

```
Loop: L.D   F0,0 (R1)
      DADDIU  R1,R1, #-8
      ADD.D       F4,F0,F2
      (空转)
      BNE         R1,R2,Loop
      S.D         F4, 8 (R1)
```

图 4

由于只有 L.D、ADD.D 和 S.D 这 3 条指令是真正有用操作。只占用了 3 个时钟周期，而 DADDIU、空转、BNE 这三个时钟周期都是附加的循环控制开销。如果进行优化就需要引入循环展开技术。

3.2 循环展开

循环展开作为静态调度的一项关键技术，在编译阶段通过增加循环中的迭代次数，旨在减小循环控制开销、提高指令级并行性，从而优化程序性能。在 Merrifield 和 Nicolau 的经典论文中指出：“通过在编译器中应用循环展开技术，可以提高指令级并行性，降低循环控制开销”[4]。循环展开的优化策略包括减小分支开销、增加指令级并行性，以及降低循环控制相关性，有助于提高指令的吞吐率。编译器通过分析代码结构和性能需求，选择合适的展开方式，以优化循环的执行效率。将上述例子中的循环展开 4 次得到 4 个循环体，然后对展开后的指令序列在不调度和调度两种情况下，分析代码的性能。假定 R1 的初值为 32 的倍数，即循环次数为 4 的倍数。消除冗余的指令，并且不要重复使用寄存器。

循环调度则需要进行分配寄存器，使用较大的空间开销来减少时间开销。

F0、F4：用于展开后的第 1 个循环体

F2：保存常数 s

F6、F8：第 2 个循环体

F10、F12：第 3 个循环体

F14、F16：第 4 个循环体

进行循环展开之后的代码如图 5 所示。

```
Loop:   L.D       F0,0(R1)          1
        (空转)                      2
        ADD.D     F4,F0,F2          3
        (空转)                      4
        (空转)                      5
        S.D       F4, 0 (R1)        6
        L.D       F6,-8(R1)         7
        (空转)                      8
        ADD.D     F8,F6,F2          9
        (空转)                      10
        (空转)                      11
        S.D       F8, -8 (R1)       12
        L.D       F10,-16(R1)       13
        (空转)                      14
        ADD.D     F12,F10,F2        15
        (空转)                      16
        (空转)                      17
        S.D       F12,-16(R1)       18
        L.D       F14,-24(R1)       19
        (空转)                      20
        ADD.D     F16,F14,F2        21
        (空转)                      22
        (空转)                      23
        S.D       F16,-24(R1)       24
        DADDIU    R1,R1,#-32        25
        (空转)                      26
        BNE       R1,R2,Loop        27
        (空转)                      28
```

图 5

这个循环每遍共使用了 28 个时钟周期。有 4 个循环体，完成 4 个元素的操作。平均每个元素使用 28/4=7 个时钟周期原始循环的每个元素需要 10 个时钟周期。节省的时间：从减少循环控制的开销中获得的。在整个展开后的循环中，实际指令只有 14 条，其他 14 个周期都是空转。

因此，如果我们将循环展开与指令调度结合起来，则会得到更加高效的代码，减少空转周期。综合后代码如图 6 所示。

```
Loop:   L.D     F0,0(R1)        1
        L.D     F6,-8(R1)       2
        L.D     F10,-16(R1)     3
        L.D     F14,-24(R1)     4
        ADD.D   F4,F0,F2        5
        ADD.D   F8,F6,F2        6
        ADD.D   F12,F10,F2      7
        ADD.D   F16,F14,F2      8
        S.D     F4,0(R1)        9
        S.D     F8,-8(R1)       10
        DADDIU  R1,R1,#-32      12
        S.D     F12,16(R1)      11
        BNE     R1,R2,Loop      13
        S.D     F16,8(R1)       14
```

图 6

4 动态调度

4.1 Tomasulo 算法

动态调度是一项在程序运行时动态调整指令执行顺序的技术，旨在最大程度地利用处理器资源，提高程序性能。与此相关的 Tomasulo 算法，由 Tomasulo 于 1967 年提出[5]，是一种常用的动态调度算法，通过引入 Reservation Stations 和功能单元来解决数据相关性问题。Reservation Stations 存储指令的操作数和状态信息，使得指令可以等待操作数就绪而不需要阻塞流水线。功能单元执行特定类型的操作，多个功能单元可以并行执行不同的指令。该算法的高级特性包括异步执行和有效的数据相关性处理机制，允许指令以异步方式执行，提高了指令级并行性，并通过 Reservation Stations 机制高效处理数据相关性，降低了数据冒险的影响。Tomasulo 算法在实现动态调度时，充分体现了对处理器性能的优化追求，使得指令能够更灵活、高效地在流水线中执行。

图 7 为 Tomasulo 算法架构图。

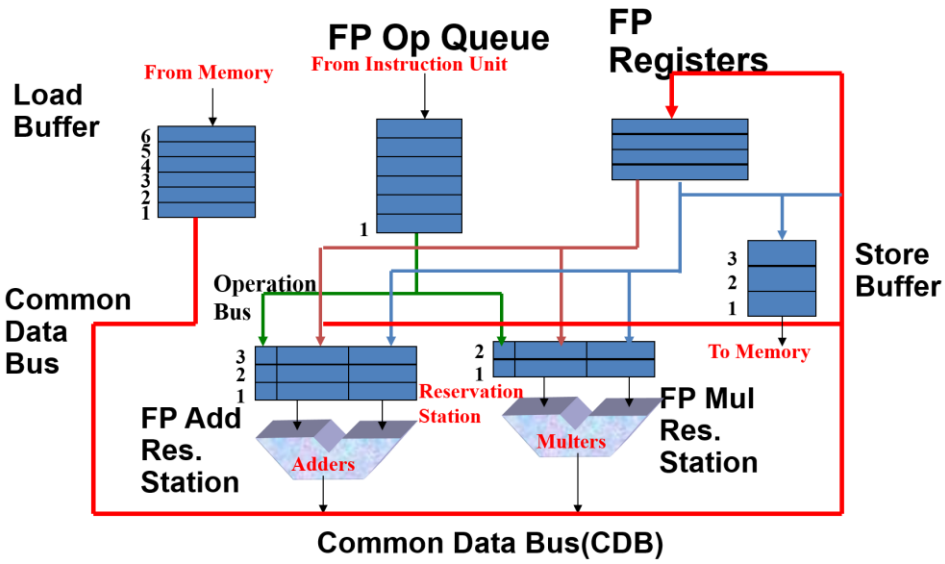


图 7

图 8 为保留站，图 9 为寄存器结果状态表

Reservation Stations		S1	S2	RS for j	RS for k
Time	Name	Bus Op	Vj	Vk	Qj	Qk
0	Add1	No				
0	Add2	No				
0	Add3	No				
0	Mult1	No				
0	Mult2	No				

图 8

Clock	F0	F2	F4	F6	F8	F10	F12 ...	F30
0	FU							

图 9

保留站是 Tomasulo 算法提出的新结构，有点类似记分牌中每一个配置通路前面的译码信息流水段寄存器，但是记分牌中每一条配置通路只能存放一条指令，而 Tomasulo 算法则为每一条通路配置了一组缓冲，其中浮点加法单元拥有能够缓冲三条指令的保留站，乘法单元拥有两个保留站保留站。

在动态调度中的结构和功能与 Cache 相似，它包含多行数据，每行对应一条待发射的指令。每行的关键字段包括 Busy 位、Vj、Vk、Qj、Qk 和 OP。与记分牌不同，

保留站的 Vj 和 Vk 字段直接存储能读取的数据，避免了仅记录源寄存器编号的瓜葛。一旦数据进入保留站，对应寄存器与指令解除关联。Qj 和 Qk 字段与记分牌相似，记录尚不能读取的数据将由哪条指令计算得出，而 OP 字段用于存储指令的地址。

Tomasulo 算法的运行分为三个主要阶段，分别是发射（Issue）、执行（Execute）、写回（Write Back）。在发射阶段，指令被发射到保留站；执行阶段中，指令在功能单元中执行；而写回阶段则将计算结果写回寄存器文件。这三个阶段形成了 Tomasulo 算法的基本执行流程。

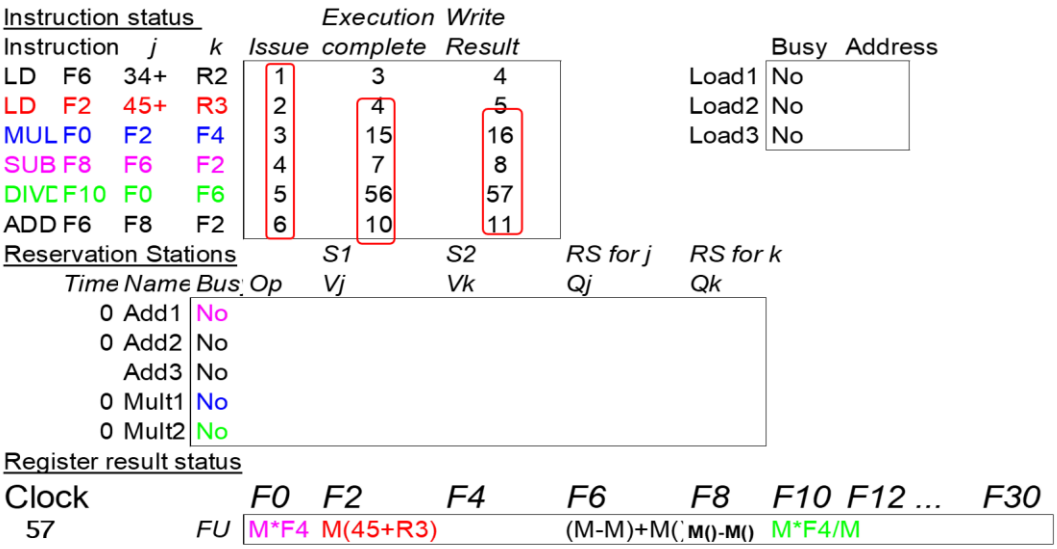


图 10

Tomasulo 算法运行结束之后会将 Execution Write 表填满周期数，将寄存器结果状态表中使用到的寄存器写入输值并且将 tag 位置位。

尽管 Tomasulo 算法是动态调度的经典算法，但是仍然有一些显著的缺点。

Tomasulo 算法中每一个执行单元对应一个保留站，保留站中缓冲多条指令，所以有可能在同一周期有多条指令准备好数据，但是执行单元同时只能执行一条指令，所以就需要从中选择一条指令，简单的解决办法是为保留站的每一行增加一个年龄位，每次出现冲突，就选择最老的指令送到执行单元。

电路里的 CDB 总线只有一组，这意味着每一个周期只能写回一条指令，如果同时有多条指令完成，那就只能选择一条指令进行广播，别的指令等待；第二种办法是增加 CDB 总线，支持多指令广播，但是这会让电路面积大增，包括增加寄存器堆

写口、增加保留站 tag 和 CDB 总线的比较电路、增加保留站写口。

Tomasulo 算法没办法实现精确中断，精确中断是指在指令和指令之间如果出现了中断/异常，那么处理器要确保中断/异常之前的所有指令都执行完毕，而中断/异常之后的所有指令都没有执行，然后处理器把中断发生时的处理器状态给保存下来或是呈现给程序员看。这样的精确中断保证了程序的正确执行，也提供程序员调试程序的手段，是现代处理器必不可少的能力，而 Tomasulo 并不支持。

4.2 记分牌技术

记分牌技术是一种在超标量处理器中广泛使用的动态调度技术。论文《Dynamic Scheduling of Micro-Operations》首次提出了记分牌技术，该技术旨在解决数据相关性和提高指令级并行性的问题。记分牌通过为每个功能单元维护一个记录表，记录了当前功能单元的状态、指令来源、以及操作数的可用性。这种机制允许指令在运行时动态调整执行顺序，避免数据相关性造成的流水线停顿。记分牌技术的优点在于其对数据相关性的敏感性较低，可以有效提高处理器的并行度。虽然记分牌技术在 Tomasulo 算法中首次应用，但它的基本原理仍然在现代处理器设计中得到广泛应用，为处理器的性能优化提供了重要的思路和实践基础。

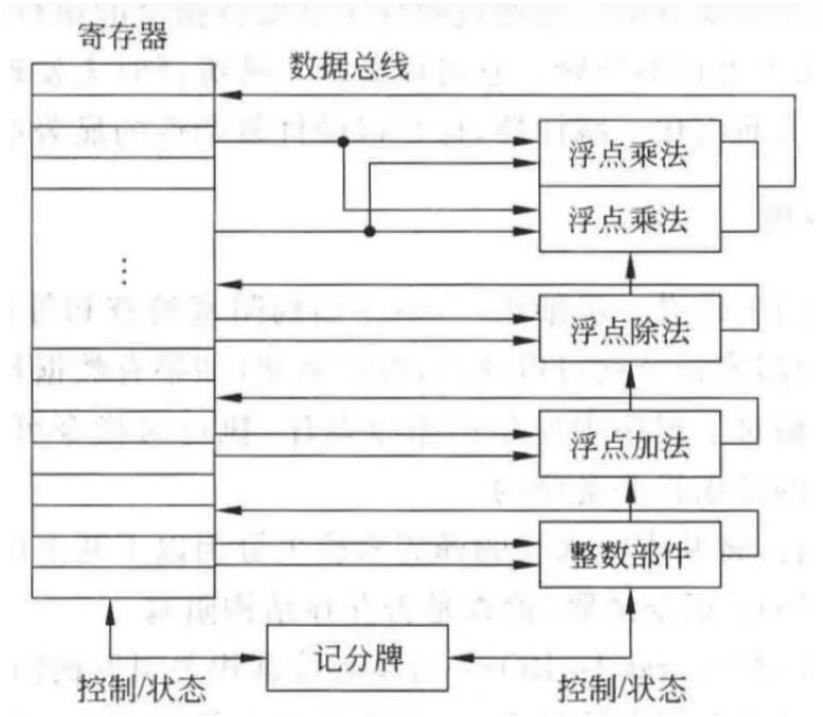


图 11

如图 11 为具有记分牌的 MIPS 处理器基本结构

每条指令均经过记分牌，并记录下各条指令间数据相关的信息，这一步对应于指令流出，部分取代 MIPS 的指令译码（ID）阶段的功能。然后记分牌就需要判断什么时候指令可以读到所需的操作数，开始执行指令。如果记分牌判断出一条指令不能立即执行，它就检测硬件的变化从而决定何时能够执行。记分牌还控制指令写目标寄存器的时机。

每条指令在流水线中的执行过程可分为指令的流出 读操作数、执行和写结果等四段，由于主要考虑浮点操作，因此不涉及存储器访问段。

（1）流出（Issue,IS）。若本指令所需的功能部件有空闲，且其他正在执行的指令使用的目的存储器于本指令的不同，记分牌就向功能部件流出本指令，并修改记分牌内部的数据记录。

（2）读操作数（Read Operation,RO）。记分牌需要监测源操作数寄存器中数据的有效性，如果前面已流出的还在运行的指令不对本指令的源操作数寄存器进行写操作，或一个正在工作的功能部件已经完成了对这个寄存器的写操作，则此操作数有效。解决了 RAW 相关。

（3）执行（Execution,EX）。取到操作数后就开始执行指令。

（4）写结果（Write Result,WR）。记分牌知道指令执行完毕后，如果目标寄存器空闲，就将结果写入到目标寄存器中，然后释放本指令使用的资源。

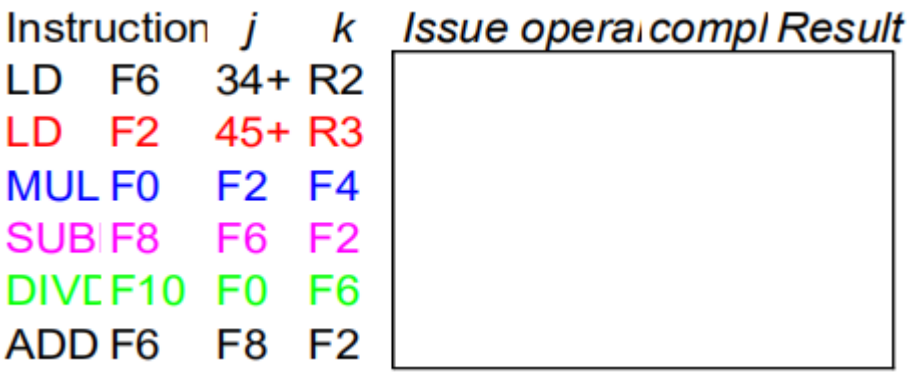


图 12

Time Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Divide	No								

图 13

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU									

图 14

如图 12 为记分牌记录周期数部件。如图 13 为保留站。如图 14 为寄存器结果状态表。

记分牌记录的信息与 Tomasulo 算法类似。每行的关键字段包括 Busy 位、Fi、Fj、Fk、Qj、Qk、Rj、Rk 和 OP。Fi 为目的寄存器编号，Fj、Fk 为源寄存器编号。Qj、Qk 为向 Rj、Rk 写结果的功能部件。Rj、Rk 表示 Fj、Fk 是否就绪，是否以及被使用。busy 指示功能不见是否在工作，而 OP 字段用于存储指令的地址。

大致流程为首先是取指令然后到 ID 段，分为译码与读数两个部分，先检查结构冒险（检查该指令需要使用的运算器是否正在被占用，通过相应功能部件的 Busy 段可得），有就等待没有就进行操作数的检查也就是数据冒险的检查（检查目的寄存器是否被功能单元占用等待写入，通过寄存器的写入状态 FU 表对应）。若无数据冲突则指令流出至读取操作数段。这里解决了相关的 WAW 冲突与结构冲突。流出指令后对于对应的 FU 表中就应有相应的标识了。

读取操作数阶段会通过上面的 Qj，Qk 段与 Rj，Rk 字段来判断是否有功能单元对源寄存器有写入操作，防止 RAW 发生，如果有的话就会等待相关的冲突结束，也就是相关的功能部件将结果写入后再读操作数。读取后就交由运算器处理。这样对于运算器而言有些先流出的指令会因为操作数的读取而后执行，实现了指令的乱序执行。

记分牌技术同样也有一些缺点。对于冒险的处理大多以等待来解决，效率低。存在结构冒险，就暂停发射指令；无法消除 WAR 和 WAW 两种相关，仅通过暂停来

处理。并行性的使用仅在处理 RAW 相关时使用来乱序执行减少停顿周期。

5 指令调度的静态调度以及动态调度的应用

（1）编译器优化： 静态调度主要发生在编译阶段，编译器通过对代码的分析和重排，优化指令的执行顺序以提高程序性能。循环展开、指令调度等技术被广泛应用于生成更有效率的机器代码。

（2）架构特定优化： 静态调度允许编译器在生成目标代码时考虑目标体系结构的特性。通过静态调度，编译器可以优化指令的排列，以适应目标处理器的流水线结构、缓存层次等特征，提高指令级并行性。

（3）代码重排和流水线填充： 静态调度可以通过重新排列指令的执行顺序来减少流水线中的空闲周期。编译器可以通过填充流水线，使得在执行阶段更多的指令可以同时处于不同的流水线阶段，提高整体性能。

（4）超标量和乱序执行处理器： 动态调度在超标量和乱序执行处理器中发挥关键作用。处理器可以在运行时动态地调整指令的执行顺序，以最大程度地利用硬件资源，提高指令级并行性，从而提高性能。

（5）Tomasulo 算法： Tomasulo 算法是一种典型的动态调度算法，广泛应用于超标量流水线处理器中。它通过使用保留站和功能单元来实现对指令的乱序执行，解决了数据相关性的问题，提高了指令级并行性。

（6）分支预测和动态调度的结合： 动态调度在处理分支指令时具有一定的优势。通过动态调度，处理器可以更灵活地调整流水线中的指令，减轻分支带来的性能影响。结合分支预测技术，动态调度可以更好地处理分支指令，提高程序的执行效率。

（7）现代处理器设计： 多核处理器和同时多线程（SMT）处理器通常采用动态调度以提高整体性能。通过动态调度，处理器能够更灵活地适应多任务和多线程的执行需求，充分利用硬件资源。

6 未来发展发方向

（1）深度学习和人工智能： 随着深度学习和人工智能应用的普及，对于大规模数据并行处理的需求日益增加。未来的指令调度可能会更加专注于优化神经网络和深度学习工作负载，通过特定的调度算法提高计算性能。

（2）异构计算： 随着异构计算平台的兴起，包括 CPU、GPU、FPGA 等在同一系统中的集成，未来的指令调度可能需要更加智能的方法，以有效地分配任务到不同的处理单元，并充分利用异构计算平台的优势。

（3）量子计算： 随着量子计算技术的研究和发展，未来的指令调度可能需要重新考虑如何有效地调度量子指令和经典指令，以优化混合量子经典计算的性能。

（4）自适应调度和自学习系统： 未来的指令调度系统可能更加自适应和自学习，能够根据应用程序和运行环境的动态变化，实时调整调度策略，以最大程度地提高性能。

（5）量子指令调度： 随着量子计算的发展，未来可能需要研究和设计专门用于量子计算的指令调度策略，以充分利用量子计算的并行性和量子特性。

（6）量子经典混合系统： 针对量子经典混合系统，未来的指令调度可能需要考虑如何有效地协调经典处理器和量子处理器之间的任务分配和指令调度，以实现更高效的混合计算。

（7）能效和节能： 随着对能效和节能要求的提高，未来的指令调度可能会更加注重在保证性能的同时最小化功耗，通过更智能的调度策略来实现更高的能效。

7 总结

学完计算机体系结构这门课以及书写完指令调度相关的报告，我深刻地认识到计算机体系结构是计算机科学中至关重要的一门基础课程。通过对计算机体系结构的学习，我不仅加深了对计算机硬件组成和运作原理的理解，还掌握了许多关键概念和技术，其中指令调度作为重要的一环，给我留下了深刻的印象。

在指令调度的学习中，我深入了解了静态调度和动态调度的概念、原理及其应用。静态调度在编译阶段通过编译器进行优化，而动态调度则在运行时通过处理器

动态调整指令执行顺序以提高性能。深入研究 Tomasulo 算法，为提高指令级并行性和解决数据相关性问题提供了有效的解决方案。

通过报告的撰写，我总结了指令调度的基本概念和原理。我特别强调了 Tomasulo 算法的关键作用，它不仅在处理器设计中发挥重要作用，也为处理器架构的研究和教育提供了有力的支持。

8 参考文献

[1] Allen, R., & Kennedy, K. (2001). "Optimizing Compilers for Modern Architectures: A Dependence-based Approach." Morgan Kaufmann.
[2]Chen, G., & Baer, J. L. (1995). "Effective Instruction Scheduling for Superscalar Processors." ACM Transactions on Programming Languages and Systems (TOPLAS), 17(1), 1-38.
[3] Kennedy, K., & Allen, R. (2001). "Optimizing Compilers for Modern Architectures: A Dependence-based Approach." Morgan Kaufmann.
[4] Merrifield, R., & Nicolau, A. (1983). "Optimization of Loop Nests." ACM Transactions on Programming Languages and Systems (TOPLAS), 5(4), 668-699.

原创性声明

本人郑重声明：该课程报告的内容，是由作者本人独立完成的。有关观点、方法、论述、文献、图片引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字: 梅程锦

	评价内容	权重	得分
报告格式	报告的格式是否严格按照模板所给予的格式要交进行，图表的制作与引用是否规范。	0.2	
章节组织逻辑	报告章节结构组织是否严谨，段落内容描述是否条理清楚。	0.2	

报告撰写	报告内容是否比较规范，语言流畅，描述正确、详实，有无自己加工消化输出的内容，有无进行独立思考后的总结、体会等。	0.6	
合计			
指导教师（签章）： ___ 2023 年___月___日			