《计算机体系结构》实验指导手册

（第二版）

计算机科学与技术学院

目 录

[1 WinDLX模拟器介绍 1](#_Toc399433428)

[1.1 WinDLX简介 1](#_Toc399433429)

[1.2 WinDLX模拟器安装方法 1](#_Toc399433430)

[1.3 WinDLX使用说明 2](#_Toc399433431)

[1.3.1 WinDLX的启动和装载程序 2](#_Toc399433432)

[1.3.2 WinDLX子窗口讲解 4](#_Toc399433433)

[2 WINDLX汇编语言简介 10](#_Toc399433434)

[2.1伪指令 10](#_Toc399433435)

[2.2 WinDLX指令分类 10](#_Toc399433436)

[2.3Trap机制 12](#_Toc399433437)

[2.4常用指令详细讲解 13](#_Toc399433438)

[3 实例分析 20](#_Toc399433439)

[3.1冒泡排序主程序 21](#_Toc399433440)

[3.2输入函数 23](#_Toc399433441)

[3.3运行结果说明 25](#_Toc399433442)

# 1 WinDLX模拟器介绍

## WinDLX简介

美国斯坦福大学计算机系Hennessy教授和加州伯克利分校计算机系Patterson教授是计算机系统结构领域非常受人尊敬的学者和开拓者。John Hennessy 精通软硬件两个方面，是具有传奇色彩的MIPS编译器和几代MIPS硬件产品技术上的领导者。David Paterson 是RISC的最初提倡者之一，他首创了RISC一词，定义了RISC的含义，是Sun公司SPARC处理器的主要顾问。他提出了RAID（独立/廉价磁盘冗余阵列）的概念，推动了为海量数据服务器提供磁盘存储解决方案的工业革命，并且还提出了NOW（工作站网络）的概念。DLX指令集结构是两位教授对当前大多数指令集结构研究的结果，DLX指令集是一种多元未饱和型指令集结构，它体现了当今多种机器指令集结构的共同特点：具有一个简单的Load/Store指令集；注重指令流水效率；简化指令的译码；高效支持[编译器](http://baike.baidu.com/view/487018.htm)等。同时DLX指令集是一种适合于学习和研究的指令集结构模型。

WinDLX模拟器是一个基于Windows操作系统的图形化、交互式DLX指令集模拟器，能够演示DLX指令集流水线是如何工作的。该模拟器可以装载DLX汇编语言程序（后缀为“.s”的文件），然后单步、设断点或是连续执行该程序。CPU的寄存器、流水线、I/O和存储器都用图形进行表示，以形象生动的方式描述DLX流水线的工作过程。模拟器还提供了对流水线操作的统计功能，便于对流水线进行性能分析。

## 1.2 WinDLX模拟器安装方法

WinDLX 包含windlx.exe和windlx.hlp文件。同时，还需要一些扩展名为**.s**的汇编源代码文件。在window操作系统中按以下步骤安装WinDLX：

（1）为WinDLX创建目录，例如D:\WINDLX，注意受编译的约束，路径名中不可以包含汉字。

（2）解压WinDLX软件包或拷贝所有的WinDLX文件（至少包含 windlx.exe，windlx.hlp，fact.s 和input.s ）到这个WinDLX 目录。

## 1.3 WinDLX使用说明

接下来以计算数的阶乘为例介绍WinDLX的使用，我们将用到两个.s汇编源码文件：fact.s和input.s，input.s用来从键盘中读数，fact.s用来计算所读数的阶乘。

### 1.3.1 WinDLX的启动和装载程序

在解压后的文件夹中选中并双击windlx.exe图标启动WinDLX，将会看到如图1.1所示winDLX启动界面。

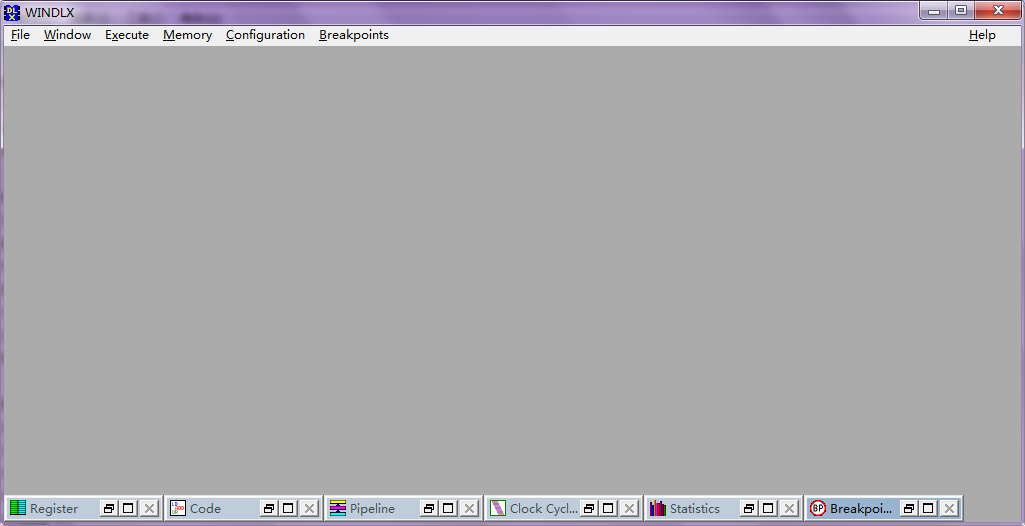


图1.1 WinDLX启动界面

为了初始化模拟器, 点击图1.1中File 菜单中的“Reset all”菜单项，弹出如图1.2所示“Reset DLX”对话框。然后点击窗口中的“确认”按钮即可。

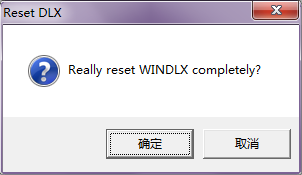


图1.2 Reset DLX对话框

装载阶乘程序到主存。点击图1.1中File/Load Code or Data，窗口中会列出目录中所有汇编源代码文件，如图1.3所示。

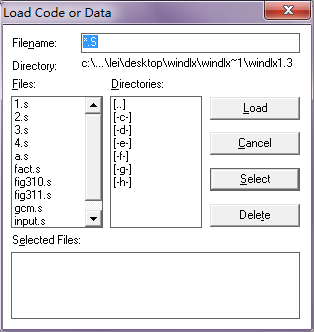


图1.3 装载程序界面

我们在前面已经提到，fact.s计算一个正整型值的阶乘，input.s是一个源码子程序，它的主要功能是读标准输入（键盘）并将值存入DLX处理器的通用寄存器R1中。按如下步骤操作，可将这两个文件装入主存。

点击fact.s

点击select 按钮

点击input.s

点击select按钮

点击load按钮

装载文件的顺序很关键，它决定了文件在存储器中出现的顺序。主程序应该首先装载，再按照子程序在主程序中调用的先后顺序进行装载，此程序只有一个子程序，主程序装载完成后进行装载即可。装载完成后会弹出如图1.4所示提示装载成功对话框，对话框中会显示信息“File(s) loaded successfully. Reset DLX?”，点击“是（Y）” 按钮确认。这样，文件就已被装入到存储器中了。

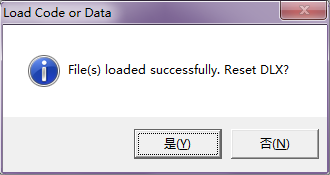


图1.4 提示装载成功对话框

### 1.3.2 WinDLX子窗口讲解

在主窗口中，你可以看见六个图标，它们分别为“Register（寄存器子窗口）”，“Code（代码子窗口）”，“Pipeline（流水线子窗口）”，“Clock Cycle Diagram（时钟周期子窗口）”，“Statistics（统计数据子窗口）” 和“Breakpoints（断点子窗口）”。双击其中任何一个将弹出与其对应的子窗口。接下来详细介绍每一个子窗口的特性和用法。

1. **Pipeline子窗口**

我们首先来看一下DLX处理器的内部结构。为此，双击图标Pipeline，出现如图1.5所示Pipeline子窗口，窗口中用图表形示显示了DLX的五段流水线。你应尽可能地扩大此窗口，以便处于不同流水段的指令都能够在图表中显示。黄色表示取指阶段，棕色表示译码阶段，绿色表示访存阶段，蓝色表示写回阶段，红色表示整型加执行阶段，粉色表示浮点加执行阶段，淡绿色表示浮点乘执行阶段，淡棕色表示浮点除执行阶段。

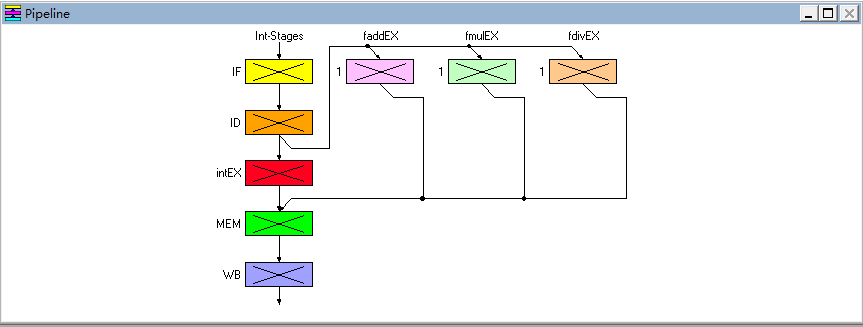


图1.5 Pipeline子窗口

1. **Code子窗口**

双击图标Code，出现如图1.6所示Code子窗口。

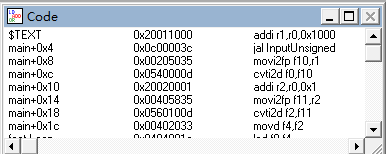


图1.6 Code子窗口

你将看到代表存储器内容的三栏信息，从左到右依次为：地址 (符号或数字)、命令的十六进制机器代码和汇编命令。

现在，点击主窗口中的Execution开始模拟。在出现的下拉式菜单中，点击Single Cycle或按 F7键。

这时，窗口中带有地址“$TEXT ”的第一行变成黄色。按下 F7 键，模拟就向前执行一步，第一行的颜色变成黄色，下一行变成棕色，如图1.7所示。这些不同颜色指明命令处于流水线的哪一段，此颜色和Pipeline窗口颜色一致。如果Pipeline窗口已经关闭，可以双击相应图标重新打开它。

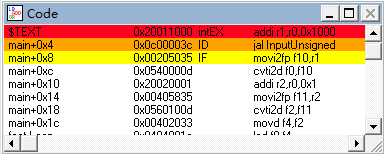


图1.7 Code窗口执行指令

1. **Clock Cycle Diagram子窗口**

双击Clock Cycle Diagram图标打开Clock Cycle Diagram窗口，它显示流水线的时空图，如图1.8所示，此窗口流水线的不同颜色和Pipeline窗口的颜色相对应。

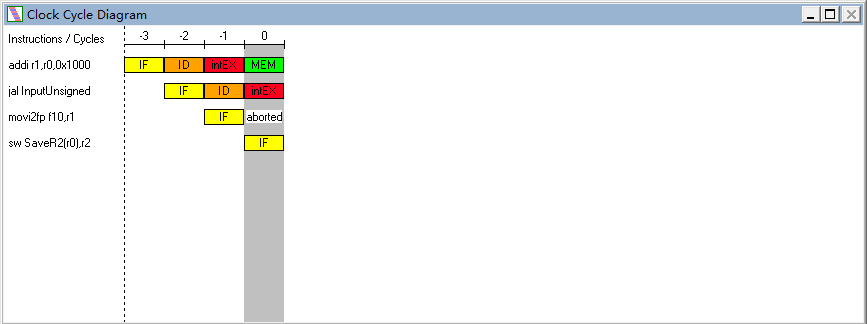


图1.8 Clock Cycle Diagram子窗口

在窗口上方的横坐标代表时钟周期，0代表第一时钟周期，1代表第二时钟周期，以此类推，n代表第n+1的时钟周期，图1.8表示模拟正在第四时钟周期，第一条命令正在MEM段，第二条命令在intEX段，第四条命令在IF段。而第三条命令指示为"aborted"。其原因是：第二条命令（jal）是无条件分支指令， 但只有在第三个时钟周期， jal 指令被译码后才知道，这时，下一条命令movi2fp已经取出，但需执行的下一条命令在另一个地址处，因而，movi2fp的执行应被取消，在流水线中留下气泡。

jal的分支地址命名为"InputUnsigned"。为找到此符号地址的实际值，点击主

窗口中的Memory/Symbols，出现的子窗口中显示相应的符号和对应的实际值。在"Sort："区域选定"name"，使它们按名称排序，而不是按数值排序。数字后的"G"代表全局符号，"L"代表局部符号。"input"中的"InputUnsigned"是一个全局符号，它的实际值为0x144 ，用作地址。点击OK按纽关闭窗口。

再一次点击F7 ，第一条命令（addi）到达流水线的最后一段。如果想了解某条命令执行后处理器内部会发生什么，只要对准Clock cycle diagram窗口中相应命令所在行，然后双击它，弹出一个新窗口。窗口中会详细显示每一个流水段处理器内部的执行动作。这个窗口"Information about ..." 作为将来的Information 窗口。观察完后，点击OK按钮关闭窗口。双击第三行（movi2fp），你会看到它只执行了第一段（IF）， 这是因为出现跳转而被取消。(双击Code窗口中的某一行或者Pipeline 窗口中的某一段，同样可以Information 窗口)。

1. **Breakpoint子窗口**

当通过Code 窗口观察代码时 (如果未打开，双击图标 Code)，看到接下来的几条指令几近一样，它们都是sw-操作：将寄存器中的数写入存储器中。重复按F7 将很枯燥，因此，我们使用断点加快此过程。

现在，指向Code 窗口中包含命令trap 0x5的0x0000015c行，此命令是写屏幕的系统调用。单击命令行，然后点击主窗口菜单Code，单击Set Breakpoint (确保命令行仍被标记！)，将弹出一个新的"Set Breakpoint" 窗口，如图1.9所示。通过此窗口，可以选择命令运行到流水线的哪一阶段时，程序停止执行。缺省为ID段。点击OK 关闭窗口。

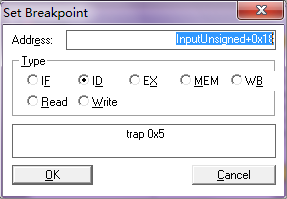


图1.9 设置断点窗口

在Code 窗口中，trap 0x5行上出现 了"BID"，它表示当本指令在译码段时，程序中止执行。

如果想查看已定义的断点，你只要单击图标Reakpoints，将弹出一个小窗口，其中显示了所有断点。重新使窗口图标化。

现在只要点击Execution/Run 或按F5，模拟就继续运行，会出现一个对话框提示你"ID-Stage: reached at Breakpoint #1"，如图1.10所示，按“确认”按钮关闭。

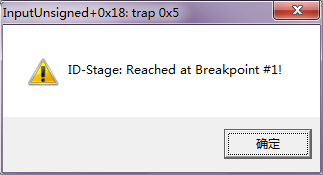


图1.10 运行到断点处提示

指令trap 0x5 已经写到屏幕上，可以通过点击主窗口菜单条上的Execute / Display DLX-I/O来查看。

1. **Register窗口**

为进一步模拟，点击Code 窗口，用箭头键或鼠标向下滚动到地址为0x00000194的那一行（指令是lw r2, SaveR2(r0)），点击此行，然后按Ins键，或点击Code / Set Breakpoint / OK，在这一行上设置一个断点。采用同样的方法，在地址0x000001a4（指令jar r31）处设置断点。现在按F5继续运行。这时，会弹出

DLX-Standard-I/O 窗口，在信息"An integer value >1: "后鼠标闪烁，键入20 然后按 Enter，模拟继续运行到断点#2处。

在Clock cycle diagram 窗口中，在指令之间出现了红和绿的箭头。红色箭头表示需要一个暂停，箭头指向处显示了暂停的原因。R-Stall（R-暂停）表示引起暂停的原因是RAW。绿色箭头表示定向技术的使用。

现在我们来看一下寄存器中的内容。为此，双击主窗口中的Register 图标。Register 窗口会显示各个寄存器中的内容，如图1.11所示。看一下R1到 R5的值。按F5使模拟继续运行到下一个断点处，有些值将发生改变，指令lw从主存中取数到寄存器中。

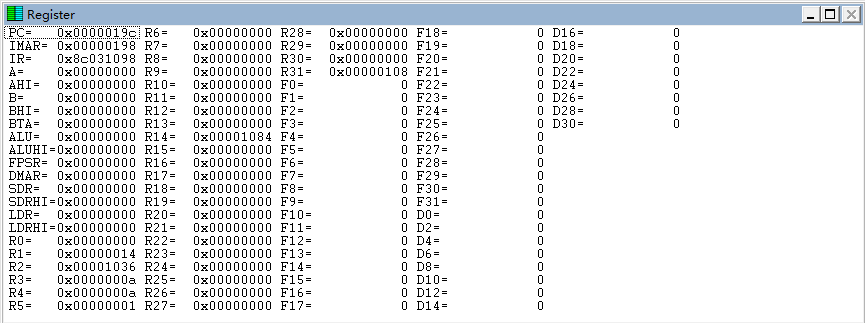


图1.11 Register子窗口

1. **Statistics子窗口**

按F5使程序完成执行，出现消息"Trap#0occurred"表明最后一条指令 trap 0 已经执行， Trap指令中编号“0”没有定义，只是用来终止程序。双击图标Statistics。Statistics 窗口提供各个方面的信息：模拟中硬件配置情况、暂停及原因、条件分支、 Load/Store指令、浮点指令和traps。窗口中给出事件发生的次数和百分比Statistics子窗口如图1.12所示，Total项中59 Cycle(s) executed表示程序执行总共用了59个时钟周期，ID Executed by 34 Instruction(s)表示总共进行了34次取指操作，3 Instruction(s) currently in Pipeline表示流水线中还剩余3个指令没有执行完，Hardware configuration表示硬件配置情况，具体有内存大小、浮点加法器数量、浮点乘法器数量和浮点除法器数量以及有没有使用重定向技术。Stalls表示相关发生的情况，RAW stall为数据相关，Structral stall为结构相关，Control stall为控制相关，Trap stall为Trap相关，Total为总相关数。Conditional Branches为条件分支。Load-/-Store-Instruction为加载/存储指令条数，Total为总数。Load为加载指令条数，Store为存储指令条数。Floating point stage instruction为浮点指令条数，Total为总数，Addition为浮点加指令条数，Multiplication为浮点乘指令条数，Division为浮点除指令条数。Traps为Trap指令条数。

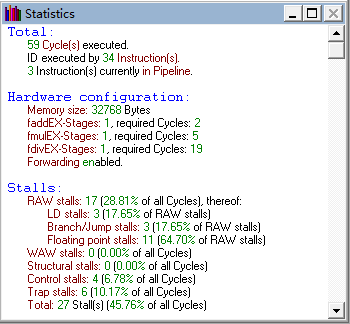


图1.12 Statistics子窗口

# 2 WINDLX汇编语言简介

WINDLX汇编语言编写的程序由两部分组成，第一部分由伪指令构成，用来指导编译器对程序进行编译；第二部分由DLX指令构成，是实际运行时执行的指令。

## 2.1伪指令

伪指令主要有：

.data [address] 标识下面的数据存放在数据区，address指示数据区的起始地址；

.text [address] 标识下面的代码存放在代码区，address指示代码区的起始地址；

.global label 使得标有label的代码可以被全局访问；

.word word1,word2*,…* 在存储器中顺序存放列出的字；

.byte byte1,byte2,*…* 在存储器中顺序存放列出的字节；

.double number1*,…* 在存储器中顺序存放列出的双精度数；

.ascii *“*string1*”, …* 在存储器中顺序存放列出的字符串，每个字符串均不会被自动加零结尾；

.asciiz *“*string1*”, …* 在存储器中顺序存放列出的字符串，每个字符串会被自动加零结尾；

.space size 在存储器中空出size大小的区域；

.align n 使得后面的数据/代码地址低n位为0对齐。

## 2.2 WinDLX指令分类

WinDLX指令大致可以分为4大类：

1.Load/Store指令：除R0之外，所有通用寄存器与浮点寄存器都可以作为加载或存储之用。

2．ALU操作指令：所有的ALU操作都是寄存器–寄存器指令。加，减，与，或，异或，移位，比较指令比较两个寄存器(=, !=, <, >, =<, =>)，如果条件

为真，则在目标寄存器置1，否则置0。

3．分支/跳转指令：所有分支都是条件分支，分支条件由指令测试寄存器为零或非零来指定。

4．浮点运算指令：浮点加，减，乘，除(浮点格式为IEEE754)。

表2-1 DLX的加载和存储指令

|  |  |  |
| --- | --- | --- |
| 指令举例 | 指令名称 | 含义 |
| LW R1，30（R2） | 加载字 | Regs[R1] ←32 Mem[30+Regs[R2]] |
| LW R1，1000（R0） | 加载字 | Regs[R1] ←32 Mem[1000+0] |
| LB R1，40（R3） | 加载字节 | Regs[R1]←32(Mem[40+Regs[R3]]0)24##Mem[40+Regs[R3]] |
| LBU R1，40（R3） | 加载无符号字节 | Regs[R1]←32024##Men[40+Regs[R3]] |
| LH R1，40（R3） | 加载半字 | Regs[R1]←32(Mem[40+Regs[R3]]0)16##Mem[40+Regs[R3]]##  Mem[41+Regs[R3]] |
| LF F0，50（R3） | 加载浮点数 | Regs[F0] ←32 Mem[50+Regs[R3]] |
| LD F0，50（R2） | 加载双精度浮点数 | Regs[F0]##Regs[F1]←64Mem[50+Regs[R2]] |
| SW R3，500（R4） | 存储字 | Mem[500+Regs[R4]] ←32 Regs[R3] |
| SF F0，40（R3） | 存储浮点数 | Mem[40+Regs[R3]] ←32 Regs[F0] |
| SD F0，40（R3） | 存储双精度浮点数 | Mem[40+Regs[R3]] ←32 Regs[F0]  Mem[44+Regs[R3]] ←32 Regs[F1] |
| SH R3，502（R2） | 存储半字 | Mem[502+Regs[R2]] ←16 Regs[R3]16…31 |
| SB R2，41（R3） | 存储字节 | Mem[41+Regs[R3]] ←8 Regs[R2]24…31 |

所有的ALU指令都是寄存器-寄存器指令，包括简单的算术和逻辑操作：加、减、与、或、异或和移位，所有这些指令都支持立即数寻址方式，它带有一个16位的符号扩展立即数。LHI（Load High Immediate）操作将立即数加载到寄存器的高半字，而将低半字设为0，这使得一个32位的常数可以用两条指令来建立。如上所述，R0经常被用来合成通用操作，加载一个常数的操作可以由一个立即数和一个源操作数是R0的加法来实现，寄存器-寄存器传送可以通过其中一个源操作数是R0的加法来完成（DLX有时用助记符LI代表加载来指前者，而用MOV来指后者）。

还有比较两个寄存器的比较指令（＝，≠，＜，＞，≤，≥），如果条件为真，则比较指令将在目的寄存器中放入一个1（代表真）；否则放入一个0。由于这些操作都设置寄存器，因此它们被叫做set-equal，set-not-equal，set-less-than等，同时这些比较指令也具有立即数寻址方式。表2-2给出了一些ALU指令的例子。

**表2-2 DLX中算术/逻辑运算指令，带立即数或不带立即数**

|  |  |  |
| --- | --- | --- |
| 指令举例 | 指令名称 | 含义 |
| ADD R1，R2，R3 | 加 | Regs[R1] ←Regs[R2]+Regs[R3] |
| ADDI R1，R2，#3 | 加立即数 | Regs[R1] ←Regs[R2]+3 |
| LHI R1，#42 | 加载立即数到高半字 | Regs[R1] ←42##016 |
| SLLI R1，R2，#5 | 逻辑左移立即数 | Regs[R1] ←Regs[R2]<<5（R2内容左移5位） |
| SLT R1，R2，R3 | 置小于 | if （Regs[R2]< Regs[R3]）  than Regs[R1] ←1 else Regs[R1] ←0 |

控制由一组跳转指令和一组分支指令来处理。4种跳转指令由指定目的地址的两种方式和是否进行链接来区分。有两种跳转指令把26位带符号的偏移量加到PC中来确定跳转的目的地址；另外两种跳转指令通过指定包含目的地址的寄存器来确定目的地址。有两种跳转：简单跳转，跳转并链接（用于过程调用），后者把返回地址（下一个顺序指令的地址）放入寄存器R31。

所有的分支都是有条件的。分支条件由指令确定，可能是测试源寄存器是否为0；寄存器可能含有一个数据或者比较结果。分支的目标地址由16位带符号偏移量和PC值相加的结果来确定。表1-3给出了几个典型的分支和跳转指令。 所有的控制流指令，除了以寄存器中的地址为目标进行的跳转以外，都是相对于PC的。如果R0作为寄存器操作数，那么BEQZ总是会无条件分支，但是编译器通常会选择一个带有较长偏移量的跳转而不是选择这个“无条件分支”。

**表2-3 典型的DLX控制流指令**

|  |  |  |
| --- | --- | --- |
| 指令举例 | 指令名称 | 含义 |
| J name | 跳转 | PC ← name ；((PC＋4)－225)≤name＜((PC＋4)＋225) |
| JAL name | 跳转并链接 | Regs[R31] ←PC＋4； PC ← name ；  ((PC＋4)－225)≤name＜((PC＋4)＋225) |
| JALR R2 | 寄存器跳转并链接 | Regs[R31] ←PC＋4； PC ← Regs[R2] |
| JR R3 | 寄存器跳转 | PC ← Regs[R3] |
| BEQZ R4，name | 等于0时分支 | if (Regs[R4]＝＝0) PC ← name ；  ((PC＋4)－215)≤name＜((PC＋4)＋215) |
| BNEZ R4，name | 不等于0时分支 | if （Regs[R4] !＝0） PC ← name ；  ((PC＋4)－215)≤name＜((PC＋4)＋215) |

## 2.3Trap机制

Trap在DLX程序和系统I/O之间建立了接口。在WinDLX中共定义了5种Trap。0对于Trap指令来说是无效参数，Trap 0用来结束程序。Trap 1可以为读写打开一个文件，打开的文件在DLX重置或结束之后自动关闭。Trap 2关闭由Trap 1打开的文件。Trap 3从文件块读，读入一个文件块或者标准输入的一行。Trap 4向文件块写，向文存储器或标准输出块写。

## 2.4常用指令详细讲解

**1、add  
格式: add r1,r2,r3  
操作：R[regc] ← R[rega] + R[regb]**

**功能：r2寄存器中整型值加上r3寄存器中整型值放到r1寄存器中**

**2、addd**  
格式: addd f4,f4,f6  
操作：D[dregc] <-- D[drega] + D[dregb]  
功能：f4**寄存器中双精度浮点数加上**f6**寄存器中双精度浮点数放到**f4**寄存器中**

**3、addf**  
格式: addf f3,f4,f5  
操作：F[fregc] <-- F[frega] + F[fregb]  
功能：f4**寄存器中单精度浮点数加上**f5**寄存器中单精度浮点数放到**f3**寄存器中**

**4、addi**  
格式: addi r5,r2,#5  
操作：R[regb] <-- R[rega] + imm16  
**功能：r2寄存器中整型值加上立即数5放到r5寄存器中**

**5、addu**  
格式: addu r2,r3,r4  
操作：R[regc] <-- R[rega] + R[regb]  
**功能：r3寄存器中无符号整型值加上r4寄存器中无符号整型值放到r2寄存器中**

**6、and**  
格式: and r2,r3,r4  
操作：R[regc] <-- R[rega] & R[regb]  
**功能：r3寄存器中无符号整型值与r4寄存器中无符号整型值按位与操作**

**7、andi**  
格式: andi r3,r4,#5  
操作：R[regb] <-- R[rega] & uimm16  
**功能：r3寄存器中无符号整型值与立即数5按位与操作，结果放到r3中**

**8、beqz**  
格式: beqz r1,label  
功能：如果(R[rega] == 0) 则PC <-- PC + imm16 + 4

**9、bfpf**  
格式: bfpf label  
功能：如果(fps == 0)则PC <-- PC + imm16 + 4，fps 是浮点状态位.

**10、bfpt**  
格式: bfpt label  
功能：如果(fps == 1) 则PC <-- PC + imm16 + 4，fps 是浮点状态位.

**11、bnez**  
格式: bnez r1,label  
功能：如果(R[rega] != 0)则PC <-- PC + imm16 + 4

**12、cvtd2f**  
格式: cvtd2f f1,f4  
操作：F[fregc] <-- (float) D[drega]

功能：双精度浮点数转换为单精度浮点数

**13、cvtd2i**  
格式: cvtd2i f1,f0  
操作：F[fregc] <-- (int) D[drega]

功能：双精度浮点数转换为整型数

**14、cvtf2d**  
格式: cvtf2d f4,f9  
操作：D[dregc] <-- (double) F[frega]  
功能：单精度浮点数转换为双精度浮点数

**15、cvtf2i**  
格式: cvtf2i f3,f4  
操作：F[fregc] <-- (int) F[frega]  
功能：单精度浮点数转换为整型数

**16、cvti2d**  
格式: cvti2d f2,f9  
操作：D[dregc] <-- (double) F[frega]  
功能：整型数转换为双精度浮点数

**17、cvti2f**  
格式: cvti2f f2,f5  
操作：F[fregc] <-- (float) F[frega]  
功能：整型数转换为单精度浮点数

**18、div**  
格式: div f2,f2,f3  
操作：F[fregc] <-- F[frega] / F[fregb]  
功能：f2**寄存器中浮点数除以**f3**寄存器中浮点数放到**f2**寄存器中**

**19、divd**  
格式: divd f4,f4,f6  
操作：D[dregc] <-- D[drega] / D[dregb]  
功能;f4**寄存器中双精度浮点数除以**f6**寄存器中双精度浮点数放到**f4**寄存器中**

**20、divf**  
格式: divf f2,f3,f6  
操作：F[fregc] <-- F[frega] / F[fregb]  
功能：f2**寄存器中单精度浮点数除以**f6**寄存器中单精度浮点数放到**f2**寄存器中**

**21、divu**  
格式: divu f2,f3,f4  
操作：F[fregc] <-- F[frega] / F[fregb]  
功能：f3**寄存器中无符号数除以**f4**寄存器中无符号数放到**f2**寄存器中**

**22、eqd**  
格式: eqd f2,f4  
功能：如果(D[drega] == D[dregb])则fps = 1否则fps = 0，f2和f4都是双精度浮点寄存器

**23、eqf**  
格式: eqf f3,f5  
功能：如果(F[frega] == F[fregb]) 则fps = 1否则fps = 0，f3和f5都是单精度浮点寄存器

**24、ged**  
格式: ged f8,f6  
功能：如果(D[drega] >= D[dregb])则fps = 1否则fps = 0，f8和f6都是双精度浮点寄存器

**25、gef**  
格式: gef f3,f6  
功能：如果(F[frega] >= F[fregb])则fps = 1否则fps = 0，f3和f6都是单精度浮点寄存器

**26、gtd**  
格式: gtd f8,f6  
功能：如果(D[drega] > D[dregb]) 则fps = 1否则fps = 0，f8和f6都是双精度浮点寄存器

**27、gtf**  
格式: gtf f3,f6  
功能：如果(F[frega] > F[fregb]) 则fps = 1否则fps = 0，f3和f6都是单精度浮点寄存器

**28、j**  
格式: j label  
操作：PC <-- PC + imm26 + 4  
功能：无条件跳转指令，跳转向PC寄存器中所存数的地址，imm26 是一个26 bit有符号整型数

**29、jal**  
格式: jal label  
操作：R31 <-- PC + 8;PC <-- PC + imm26 + 4  
功能：在R31中保存一个返回地址，跳转向PC寄存器中所存数的地址，imm26 是一个26 bit有符号整型数

**30、jalr**  
格式: jalr r2  
操作：R31 <-- PC + 8;PC <-- R[rega]  
功能：在R31中保存一个返回地址，无条件跳转向r2寄存器中所存数的地址

**31、jr**  
格式: jr r3  
操作：PC <-- R[rega]  
功能：r3中存一个无符号整型数，无条件跳转到r3中所存数的地址

**32、lb**  
格式: lb r1,40-4(r2)  
操作：R[regb] <-- (sign extended) M[imm16 + R[rega]]  
功能：r2寄存器中的数加上立即数，计算出存储器地址，取出此地址中的数并按无符号扩展为32位放到r1寄存器中

**33、lbu**  
格式: lbu r2,label-786+4(r3)  
操作：R[regb] <-- 0^24 ## M[imm16 + R[rega]]  
功能：r2寄存器中的数加上立即数，计算出存储器地址，取出此地址中的数并按按符号扩展为32位放到r1寄存器中

**34、ld**  
格式: ld f2,240(r1)  
操作：D[dregb] <--64 M[imm16 + R[rega]]  
功能：r1寄存器中的数加上立即数，计算出存储器地址，取出此地址中的数并按无符号扩展为64位放到f1双精度浮点数寄存器中

**35、led**  
格式: led f8,f6  
功能：如果(D[drega] <= D[dregb])则fps = 1否则fps = 0，f8和f6为双精度浮点寄存器

**36、lef**  
格式: lef f3,f6  
功能：如果(F[frega] <= F[fregb]) 则fps = 1否则fps = 0，f3和f6为单精度浮点寄存器

**37、lf**  
格式: lf f6,76(r4)  
操作：F[fregb] <-- M[imm16 + R[rega]]  
功能：r4寄存器中的数加上立即数，计算出存储器地址，取出此地址中的数放到f6单精度浮点数寄存器中

**38、lh**  
格式: lh r1,32(r3)  
操作：R[regb] <-- (sign extended) M[imm16 + R[rega]]  
功能：r3寄存器中的数加上立即数，计算出存储器地址，取出此地址中的数并按无符号扩展为32位放到r1寄存器中

**39、lhi**  
格式: lhi r3,#-40  
操作：R[regb] <-- imm16 ## 0^16  
功能：立即数低16位按零扩展为32位放到r3寄存器中

**40、lhu**  
格式: lhu r2,-40+4(r3)  
操作R[regb] <-- 0^16 ## M[imm16 + R[rega]]  
功能：r3寄存器中的数加上立即数，计算出存储器地址，取出此地址中的数并按高16位按符号扩展为32位放到r2寄存器中

**41、ltd**  
格式: ltd f8,f6  
功能：如果(D[drega] < D[dregb])则fps = 1否则fps = 0，f8和f6都是双精度浮点寄存器

**42、ltf**  
格式: ltf f3,f6  
功能：如果(F[frega] < F[fregb]) 则fps = 1否则fps = 0，f3和f6都是单精度浮点寄存器

**43、lw**  
格式: lw r19,label+63(r8)  
操作：R[regb] <-- M[imm16 + R[rega]]  
功能：r8寄存器中的数加上立即数，计算出存储器地址，取出此地址中的数放到r19寄存器中

**44、movd**  
格式: movd f2,f4  
操作：D[dregc] <-- D[drega]  
功能：拷贝双精度浮点寄存器f4中的数放到f2寄存器中

**45、movf**  
格式: movf f1,f2  
操作：F[fregc] <-- F[frega]  
功能：拷贝单精度浮点寄存器f2中的数放到f1寄存器中

**46、movfp2i**  
格式: movfp2i r3,f0  
操作：R[regc] <-- F[frega]  
功能：拷贝浮点寄存器f2中一个字长的数放到r3寄存器中

**47、movi2fp**  
格式: movi2fp f0,r3  
操作：F[fregc] <-- R[rega]  
功能：拷贝r3寄存器中一个字长的数放到f0寄存器中

**48、mult**  
格式: mult f2,f3,f4  
功能：F[fregc] <-- F[frega] \* F[fregb]，f2、f3和f4都是整型

**49、multd**  
格式: multd f2,f4,f6  
功能：D[dregc] <-- D[drega] \* D[dregb]，f2、f6和f4都是双精度浮点数

**50、multf**  
格式: multf f3,f4,f5  
功能：F[fregc] <-- F[frega] \* F[fregb]，f2、f5和f4都是单精度浮点数

**51、multu**  
格式: multu f2,f3,f4  
功能：F[fregc] <-- F[frega] \* F[fregb]，f2、f3和f4都是无符号整型

**52、ned**  
格式: ned f8,f6  
功能：如果(D[drega] != D[dregb])则fps = 1否则fps = 0，f8和f6都是双精度浮点数

**53、nef**  
格式: nef f3,f6  
功能：如果(F[frega] != F[fregb])则fps = 1否则fps = 0，f3和f6都是单精度浮点数

**54、nop**  
格式: nop  
功能：空闲一个周期时间

**or**  
格式: or r2,r3,r4  
操作：R[regc] <-- R[rega] | R[regb]  
功能：r2、r3和r4都是无符号整型 r3和r4按位与结果放在r2中

# 3 实例分析

此程序以冒泡排序为例讲解DLX指令的编写，程序主要包括两部分，一部分为冒泡排序主程序，一部分为输入子函数。输入函数主要接收从键盘标准输入的数，主程序对输入的数进行排序。如图3.1为主程序流程图和子程序流程图。

图3.1 冒泡排序流程图

## 3.1冒泡排序主程序

伪指令代码部分

；.data标识下面的数据放在数据区中

.data

；.asciiz表示在存储区中顺序存放列出的字符串，字符串自动加0结束

Prompt: .asciiz "input An integer which is array's size value >1 : "

PromptLast: .asciiz "input an integer :"

PrintfFormat: .asciiz "Number : %g "

；.align 使后面的数据/代码地址低2位为0对齐(.align 2表示下一个子开始)

.align 2

；.word在存储器中顺序存放列出的字

PrintfPar: .word PrintfFormat

；.space size在存储区中空出size大小的区域

Printf: .space 8

PrintfValue: .space 1024

；.text标识下面的代码存放在代码区中

.text

；.global使带有.global的标识可以被全局访问

.global main

main:

;将Prompt字符串首地址放入r1寄存器中

addi r1,r0,Prompt

;跳转向InputUnsigned标识的指令地址，调用input子函数读取数组大小的值

jal InputUnsigned

;将input函数读取的数放入寄存器中

add r2,r0,r1

add r8,r0,r1

add r6,r0,r1

add r7,r0,r1

addi r3,r0,0

;循环调用input函数读数

InputArray:

;r2寄存器中的值为0则跳转向ProcessPart标识的地址

beqz r2, ProcessPart

addi r1,r0,PromptLast

jal InputUnsigned

;将r1寄存器中的数放入r3寄存区中所存数地址的存储器中

sw PrintfValue(r3),r1

addi r3,r3,4

subi r2,r2,1

;无条件跳转向InputArray标识的指令地址

j InputArray

ProcessPart:

;r3寄存器中数清0

addi r3,r0,0

;外层循环

OutLoop:

;r6中所存数为0则跳转向End所标识的指令地址

beqz r6,End

addi r3,r0,0

subi r7,r6,1

add r6,r0,r7

；内层循环

InnerLop:

;r7中所存数为0则跳转向OutLoop所标识的指令地址

beqz r7, OutLoop

lf f10,PrintfValue(r3)

addi r3,r3,4

;将以寄存器r3所存数为地址的存储区中的数放入f11寄存器中

lf f11,PrintfValue(r3)

;如果f10小于f11，fps为1，否则fps为0

lef f10 ,f11

;如果fps为1，则跳转向HuHuan标识符的地址指令

bfpt HuHuan

；将寄存器中的数写回到存储区中，保存交换后的值

back:

;将寄存器f11中的数存放到以r3寄存器中所存数为地址的存储区中

sf PrintfValue(r3),f11

subi r3,r3,4

sf PrintfValue(r3),f10

addi r3,r3,4

subi r7,r7,1

j InnerLop

HuHuan:

movi2fp f1,r0

addf f2,f10,f1

addf f10,f1,f11

addf f11,f1,f2

j back

End:

addi r3,r0,0

;输出结果

output:

subi r8,r8,1

lf f2, PrintfValue(r3)

;单精度转换为双精度

cvti2d f0,f2

sd Printf,f0

addi r14,r0,PrintfPar

;标准输出

trap 5

addi r3,r3,4

beqz r8,over

j output

;程序结束

over:

trap 0

## 3.2输入函数

.data

ReadBuffer: .space 80

ReadPar: .word 0,ReadBuffer,80

PrintfPar: .space 4

SaveR2: .space 4

SaveR3: .space 4

SaveR4: .space 4

SaveR5: .space 4

.text

.global InputUnsigned

InputUnsigned:

;保存现场，将寄存器中的数放入存储区中

sw SaveR2,r2

sw SaveR3,r3

sw SaveR4,r4

sw SaveR5,r5

sw PrintfPar,r1

addi r14,r0,PrintfPar

;标准输出

trap 5

addi r14,r0,ReadPar

;标准输入

trap 3

addi r2,r0,ReadBuffer

addi r1,r0,0

addi r4,r0,10 ;Decimal system

;计算标准输入值

Loop:

;将以r2为地址的存储区中的数扩展为32位存放在r3寄存器中

lbu r3,0(r2)

;判断r3寄存器中的数是否等于回车，相等则fps为1

seqi r5,r3,10

; fps为1，跳转向finish

bnez r5,Finish

;减去48将assic码转换为十进制数

subi r3,r3,48

multu r1,r1,r4

add r1,r1,r3

addi r2,r2,1

j Loop

Finish:

;恢复现场，将存储区中数放入寄存器中

lw r2,SaveR2

lw r3,SaveR3

lw r4,SaveR4

lw r5,SaveR5

;跳转向寄存器r31所存数的地址

jr r31

## 3.3运行结果说明

运行程序，排序1、3、5、4、2这五个数字，查看实验结果如图3.1所示。

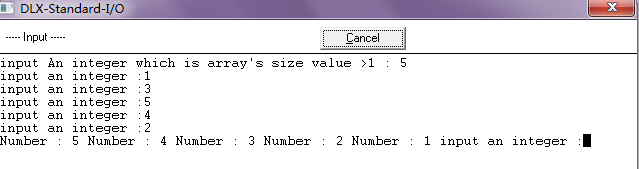


图3.1 排序结果

从图中可以看到，排序结果正确。程序中出现数据相关、结构相关和控制相关三种相关，三中相关的统计结果如图3.2所示。

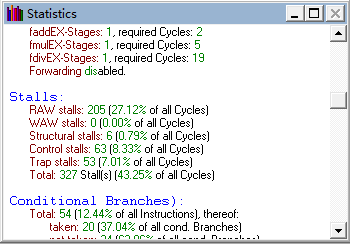


图3.2 相关统计结果

接下来就三种相关进行详细说明。

1）数据相关：当一条指令需要用到前面指令的执行结果，而这些指令均在流水线中重叠执行时，就可能引起“数据相关”。如图3.3中R-Stall发生数据相关，Seqi r5,r3,0xa在译码阶段需要r3中的数据，而lbu r3,0x0(r2)还在执行阶段，r3中的数据还未写回，所以需要等待，发生数据相关。

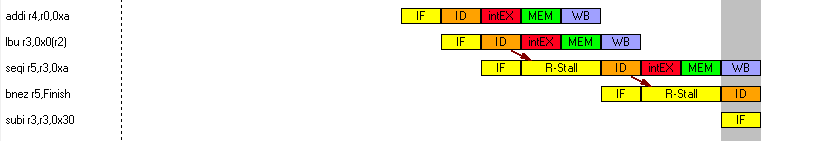


图3.3 数据相关

2）结构相关：当指令在重叠执行过程中，硬件资源满足不了指令重叠执行的要求，发生资源冲突时，将产生“结构相关”，此程序代码段

addf f2,f10,f1

addf f10,f1,f11

addf f11,f1,f2

因为浮点加法器发生结构相关，但是同时发生了数据相关，在Clock Cyle Diagram子窗口观察不到，但是在Statistics子窗口可以观察到发生了结构相关。

3）控制相关：遇到条件分支指令或者其他改变PC值的跳转指令，通常发生“控制相关”。如图3.4所示beqz跳转指令致使发生控制相关。



图3.4 控制相关

接下来讲解三种相关的减少消除技术。

1）数据相关：可以采用重定向技术来减少数据相关，重定向技术是将计算结果直接送到所需要的地方。将Windlx平台的Configuration菜单里的“Enable Forwarding”勾选，即可以采用重定向技术。采用重定向技术后运行程序，结果如图3.5所示。

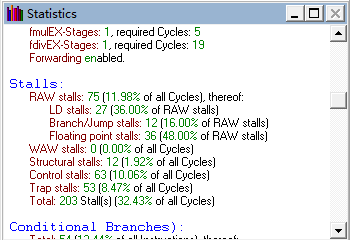


图3.5 采用重定向技术

从图中可以看出，采用重定向技术数据相关由205减少为75，有效的减少了数据相关的发生。再次分析图3.3中的数据相关，可以看到lbu r3,0x0(r2)在写回阶段，同时将数据送给Seqi r5,r3,0xa，原来数据相关占用两个时钟周期变为现在的一个，如图图3.6所示。

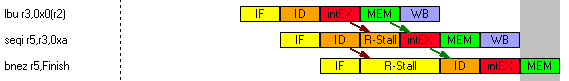


图3.6 重定向技术

2）结构相关：可以添加浮点加法器数量来消除结构相关。可在Windlx平台的Configuration菜单里“Floating Point Stage Configuration”选项对浮点加法器数目进行修改。可以将结构相关消除为0.将浮点加法器的个数设置为3，设置图 如图3.7所示

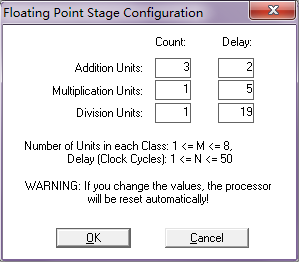


图3.7 调整浮点加法器数量

重新运行程序，查看Statistics子窗口，可以看到结构相关数量变为0。

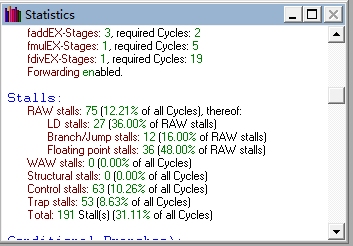


图3.8 添加浮点加法器运行结果

3）控制相关：可以采用循环展开技术减少控制相关。此程序在结果输出阶段，需要多次循环执行以下代码

output: subi r8,r8,1

lf f2, PrintfValue(r3)

cvti2d f0,f2

sd Printf,f0

addi r14,r0,PrintfPar

trap 5

addi r3,r3,4

beqz r8,over

j output

我们可以将循环体内的代码复制2次，减少跳转指令的执行，由此可以有效地减少控制相关发生的次数。循环展开后程序的执行结果如图3.9所示。

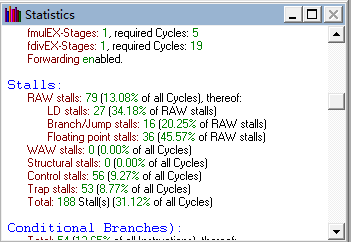


图3.9 采用循环展开技术

从图中可以看到控制相关发生次数由63减少为56。