

# 计算机系统结构

## 实验指导书

殷晓峰 编 著

山东大学计算机科学与技术学院

2004 年 3 月

# 目 录

- 一.       **WinDLX 简介**
- 二.       **WinDLX 寄存器结构及指令集**
- 三.       **WinDLX 教程**
- 四.       **实验注意事项及要求**
- 五.       **实验一 熟悉 WinDLX 的使用**
- 六.       **实验二 用 WinDLX 执行程序求最大公约数**
- 七.       **实验三 用 WinDLX 模拟器完成求素数程序**
- 八.       **实验四 结构相关**
- 九.       **实验五 数据相关**
- 十.       **实验六 指令调度**
- 十一.     **实验七 多处理机并行计算**

## 一. WinDLX 简介


### 1. DLX 流水线处理器

DLX 是贯穿本课程的一个流水线处理器实例，许多讨论、模拟结果和例题都是基于 DLX 的。它不仅体现了当今多种机器(AMD29K、DEC station 3100、HP850、IBM 801、Intel i860、MIPS M/120A、MIPS M/1000、Motorola 88k、RISC I,SGI4D/60, SPARC station 1、Sun 4/110、Sun 4/260 等)指令集结构的共同特点，而且它还将会体现未来一些机器的指令集结构的特点。这些机器的指

令集结构设计思想都和 DLX 指令集结构的设计思想十分相似，它们都强调：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。

## 2. DLX 模拟器-WinDLX

WinDLX 是一个基于 Windows 的 DLX 模拟器，用于模拟 DLX 流水线的工作过程。你可以灵活、方便地设置参数、控制执行、统计数据等。WinDLX 提供了直观的窗口显示。

我们将 WinDLX 模拟器及有关程序已放在实验用计算机上，同学们打开 XTJG 文件夹通过双击 WinDLX  图标，启动 WinDLX 即可。

## 二. WinDLX 寄存器结构及指令集

### 1. DLX 中的寄存器

## DLX中的寄存器

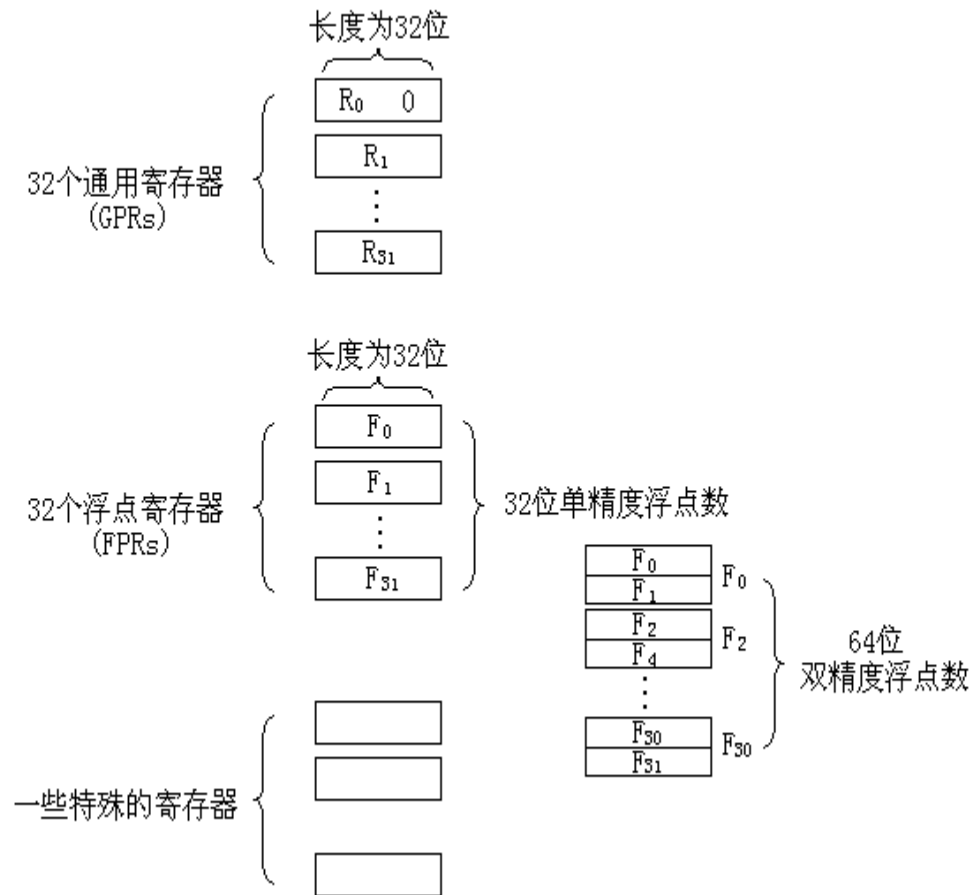


图 1

## 2. DLX 的数据类型

## DLX的数据类型

DLX提供了多种长度的整型数据和浮点数据。

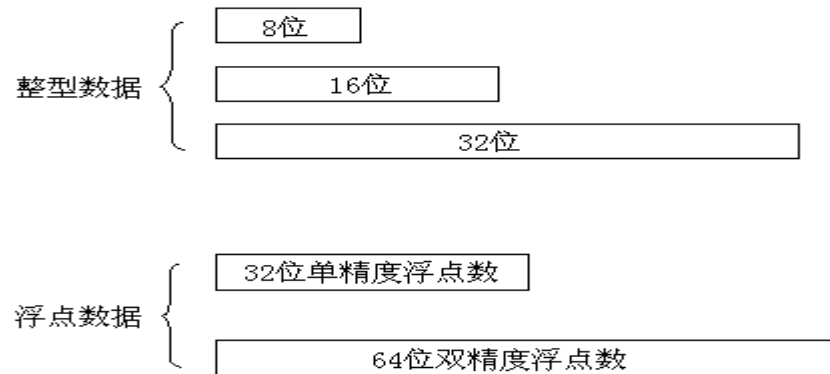


图 2

### 3. DLX 的寻址方式和数据传送

#### DLX的寻址方式和数据传送

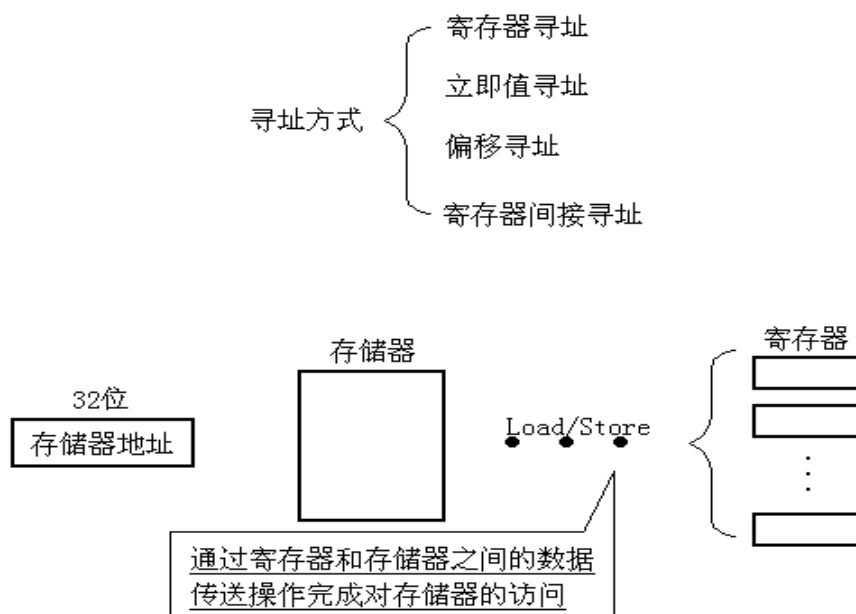


图 3

由于 DLX 支持上述数据类型，所以对通用寄存器而言，相应的存储器访问数据大小有 8 位、16 位和 32 位；而对浮点寄存器而言，相应的存储器访问数据大小有 32 位（单精度浮点数）和 64 位（双精度浮点数）。值得注意的是，DLX 的所有存储器访问均需对齐。

#### 4. DLX 的指令格式

### DLX的指令格式

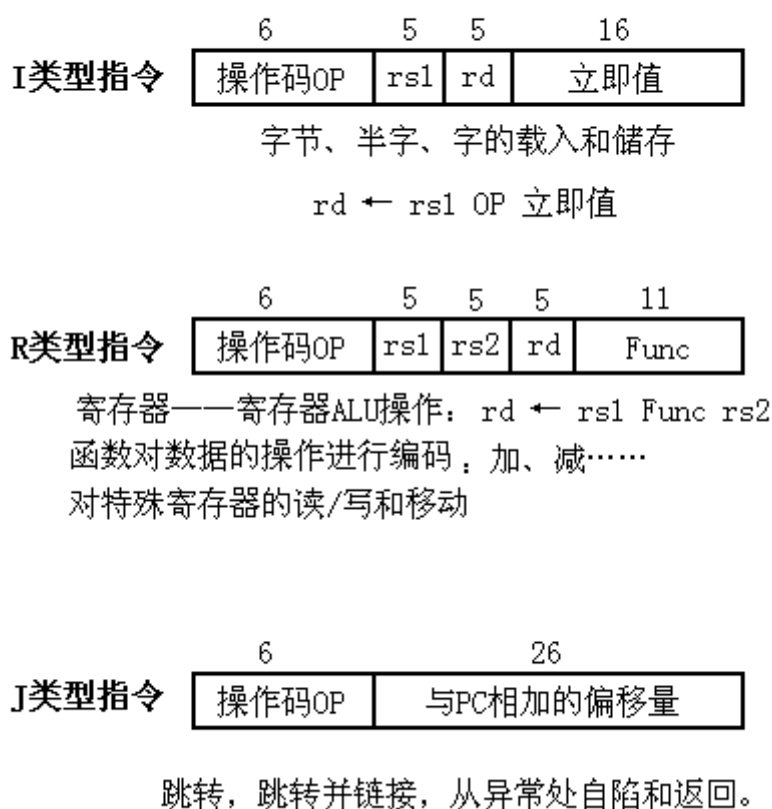


图 4

#### 5. DLX 中的操作

DLX 除了支持上面提到的一些简单操作之外，还支持其它一些操作。DLX 指令中的操作可以分为四种类型，即：Load 和 Store 操作、ALU 操作、分支和跳转操作、浮点操作。在分别讨论这四种操作类型之前，请先阅读有关本课程中所采用的一些符号的约定。

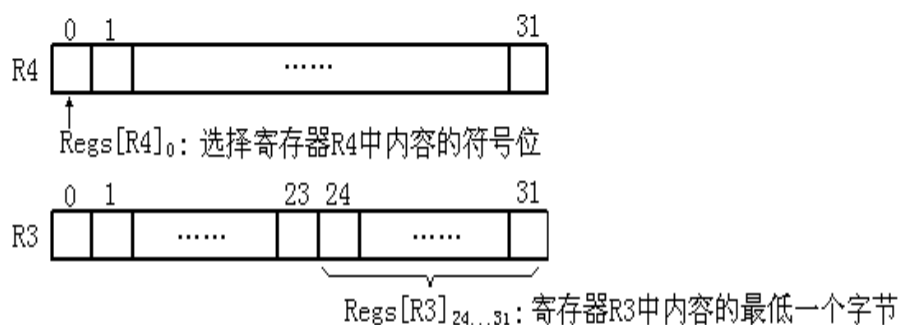
## 一些约定

← : 数据传送操作

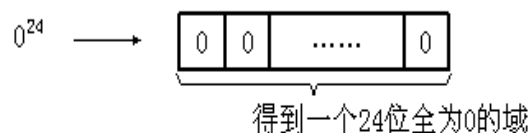
←<sub>n</sub> : 传送一个n位的数据

## : 表示两个域的串联操作

**域的下标**: 表明从该域中选择某一位



**上标**: 表示复制一个域



**变量Mem**: 表示存储器中的一个数组

存储器



图 5

为了进一步说明上述约定表示方法的用途，现设 R8 和 R10 均为 32 位寄存器

举例： $\text{Regs}[\text{R10}]_{16..31} \leftarrow_{16} (\text{Mem}[\text{Regs}[\text{R8}]]_0)^8 \## \text{Mem}[\text{Regs}[\text{R8}]]$  的含义是：以寄存器 R8 的内容为地址，取出存储器单元内容，将该内容的第“0”位（最高位）复制为八个“0”作为高八位再并上该存储单元内容，形成十六位数送寄存器 R10 的 16~31 位。

### 6. WinDLX 指令综述

#### (1) Load 和 Store 操作:

可以对 DLX 的所有通用寄存器和浮点寄存器进行 Load（载入）和 Store（储存）操作，但是对通用寄存器 R0 的 Load 操作没有任何效果。表 1 给出了载入和储存指令的一些实例。

表 1 DLX 中 Load 和 Store 指令实例

指令实例	指令名称	含 义
LW R1 , 30 (R2)	载入整型字	$\text{Regs}[\text{R1}] \leftarrow {}_{32} \text{Mem}[30+\text{Regs}[\text{R2}]]$
LW R1 , 1000 (R0)	载入整型字	$\text{Regs}[\text{R1}] \leftarrow {}_{32} \text{Mem}[1000+0]$
LB R1 , 40 (R3)	载入字节	$\text{Regs}[\text{R1}] \leftarrow {}_{32} (\text{Mem}[40+\text{Regs}[\text{R3}]]_0)^{24}$ ## Mem[40+Regs[R3]]
LBU R1 , 40 (R3)	载入无符号字节	$\text{Regs}[\text{R1}] \leftarrow {}_{32} 0^{24} \text{ ## Mem}[40+\text{Regs}[\text{R3}]]$
LH R1 , 40 (R3)	载入整型半字	$\text{Regs}[\text{R1}] \leftarrow {}_{32} (\text{Mem}[40+\text{Regs}[\text{R3}]]_0)^{16}$ ## Mem[40+Regs[R3]] ## Mem[41+Regs[R3]]
LF F0 , 50 (R3)	载入单精度浮点	$\text{Regs}[\text{F0}] \leftarrow {}_{32} \text{Mem}[50+\text{Regs}[\text{R3}]]$
LD F0 , 50 (R2)	载入双精度浮点	$\text{Regs}[\text{F0}] \text{ ## Regs}[\text{F1}] \leftarrow {}_{64} \text{Mem}[50+\text{Regs}[\text{R2}]]$
SW 500 (R4) , R3	储存整型字	$\text{Mem}[500+\text{Regs}[\text{R4}]] \leftarrow {}_{32} \text{Regs}[\text{R3}]$
SF 40 (R3) , F0	储存单精度浮点	$\text{Mem}[40+\text{Regs}[\text{R3}]] \leftarrow {}_{32} \text{Regs}[\text{F0}]$
SD 40 (R3) , F0	储存双精度浮点	$\text{Mem}[40+\text{Regs}[\text{R3}]] \leftarrow {}_{32} \text{Regs}[\text{F0}]$ $\text{Mem}[44+\text{Regs}[\text{R3}]] \leftarrow {}_{32} \text{Regs}[\text{F1}]$
SH 502 (R2) , R31	储存整型半字	$\text{Mem}[502+\text{Regs}[\text{R2}]] \leftarrow {}_{16} \text{Regs}[\text{R31}]_{16..31}$
SB 41 (R3) , R2	储存整型字节	$\text{Mem}[41+\text{Regs}[\text{R3}]] \leftarrow {}_8 \text{Regs}[\text{R2}]_{24..31}$

## (2) ALU 操作:

在 DLX 中，所有的 ALU 指令都是寄存器—寄存器型指令，其运算包含了简单的算术和逻辑运算，如加、减、AND、OR、XOR 和移位。另外，DLX 还允许所有这些指令对立即值进行操作，立即值以 16 位符号扩展形式出现。LHI（Load 高位立即值）操作将立即值载入到一个寄存器的高半部分，而该寄存器的低半部分则设置为 0。这样就可以通过两条 Load 指令构造一个 32 位的常数。

正如上面所提到的，R0 主要用来合成一些有用的操作。比如，Load 一个常数就可以看作是一次简单的立即值加操作，其中一个源操作数是 R0；寄存器—寄存器间的数据移动也可以看作是一次简单的加，其中一个源操作数是 R0。这两个操作可以分别用 LI 和 MOV 表示。

在 DLX 指令集中，还有一些寄存器比较指令（=, ≠, <, >, ≤, ≥），如果比较结果为真，这些指令就在目标寄存器中填入 1（表示真），否则填入 0（表示假）。因为这些比较操作指令要对目标寄存器进行“设置”，所以也称它们为设置相等、设置不等、设置小于等指令。DLX 同样也提供了这些比较指令的立即值形式，表 2 给出了 ALU 操作指令的一些实例。



表 2 ALU 指令实例

指令实例	指令名称	含 义
Add R1 , R2 , R3	加	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R2}] + \text{Regs}[\text{R3}]$
ADDI R1 , R2 , #3	和立即值相加	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R2}] + 3$
LHI R1 , #42	载入高位立即值	$\text{Regs}[\text{R1}] \leftarrow 42 \text{ \# } 0^{16}$
SLLI R1 , R2 , #5	逻辑左移的立即值形式	$\text{Regs}[\text{R1}] \leftarrow \text{Regs}[\text{R2}] \ll 5$
SLT R1 , R2 , R3	设置小于	if ( $\text{Regs}[\text{R2}] < \text{Regs}[\text{R3}]$ ) $\text{Regs}[\text{R1}] \leftarrow 1$ else $\text{Regs}[\text{R1}] \leftarrow 0$

## (3) 分支和跳转操作:

在 DLX 中，对程序流程的控制是通过一些跳转和分支指令来实现的。根据描述目标地址的方法和是否链接可以将跳转操作指令分为四种类型。其中两种类型的跳转指令用带符号位的 26 位偏移量加上程序计数器的值来确定跳转的目标地址，另外两种类型的跳转指令则指定一个寄存器，由寄存器中的内容决定跳转的目标地址。跳转有两种类型，一种是简单跳转，另一种是跳转并链接（用于过程调用），后者将返回一个地址，即将下一条顺序指令地址（返回地址）保存在寄存器 R31 中。

DLX 中的所有分支指令均是条件分支指令，其源操作数寄存器中包含了一个数值或某个比较结果。分支指令测试该源操作数寄存器中的值是 0 还是非 0，决定分支是否成功。分支目标地址由一个带符号的 26 位偏移量加上程序计数器的值来确定，分支目的地址指向下一条要执行的指令。表 3 给出了一些典型的分支和跳转指令。

表 3 典型的分支和跳转指令

指令实例	指令名称	含 义
J name	跳转	$\text{PC} \leftarrow \text{name};$ $((\text{PC}+4)-2^{25}) \leq \text{name} \leq ((\text{PC}+4)+2^{25})$
JAL name	跳转并链接	$\text{Regs}[\text{R31}] \leftarrow \text{PC}+4; \text{PC} \leftarrow \text{name};$

		$((PC+4)-2^{25}) \leq name \leq ((PC+4)+2^{25})$
JALR R2	寄存器型跳转并链接	$Regs[R31] \leftarrow PC+4; PC \leftarrow Regs[R2];$
JR R3	寄存器型跳转	$PC \leftarrow Regs[R3];$
BEQZ R4, name	“等于 0” 分支	if ( $Regs[R4]==0$ ) $PC \leftarrow name;$ $((PC+4)-2^{15}) \leq name \leq ((PC+4)+2^{15})$
BNEZ R4, name	“不等于 0” 分支	if ( $Regs[R4]!=0$ ) $PC \leftarrow name;$ $((PC+4)-2^{15}) \leq name \leq ((PC+4)+2^{15})$

#### (4) 浮点操作:

在 DLX 中，浮点指令的操作数来源于浮点寄存器，同时该浮点指令还指明了相应的操作是单精度浮点操作还是双精度浮点操作。

DLX 的浮点操作有：加、减、乘、除。后缀 D 代表双精度浮点操作，而后缀 F 代表单精度浮点操作（如：ADDD、ADDF、SUBD、SUBF、MULTD、MULTF、DIVD、DIVF）。值得提出的是，DLX 的浮点比较操作设置浮点状态寄存器中的位，如果比较结果为真，则将该位设置为 1；如果比较结果为假，则将该位设置为 0。浮点分支指令 BFPT 和 BFTF 则测试该寄存器的值来决定分支是否成功。

另外，操作 MOVF 将一个单精度浮点寄存器的内容拷贝至另一个单精度浮点寄存器；MOVD 则将一个双精度浮点寄存器的内容拷贝至另一双精度寄存器；MOVFP2I 和 MOVI2FP 操作则是在一个浮点寄存器和通用寄存器之间移动数据，如果要将一个双精度浮点数移入两个通用寄存器则需要两条指令，另外 DLX 还提供了在 32 位浮点寄存器中进行整数乘除操作的指令。

表 4 列出了 DLX 所有指令及其含义。为了明确哪些指令是最常用的，分别用 SPECint92 和 SPECfp92 基准程序集对 DLX 进行统计，可以分别得到表 5 和表 6 的指令使用频率测试统计结果。对于测试统计结果中使用频率大于 1% 的指令，以直方图的形式分别表示在图 6 和图 7 中。

## 指令使用频率的整型平均

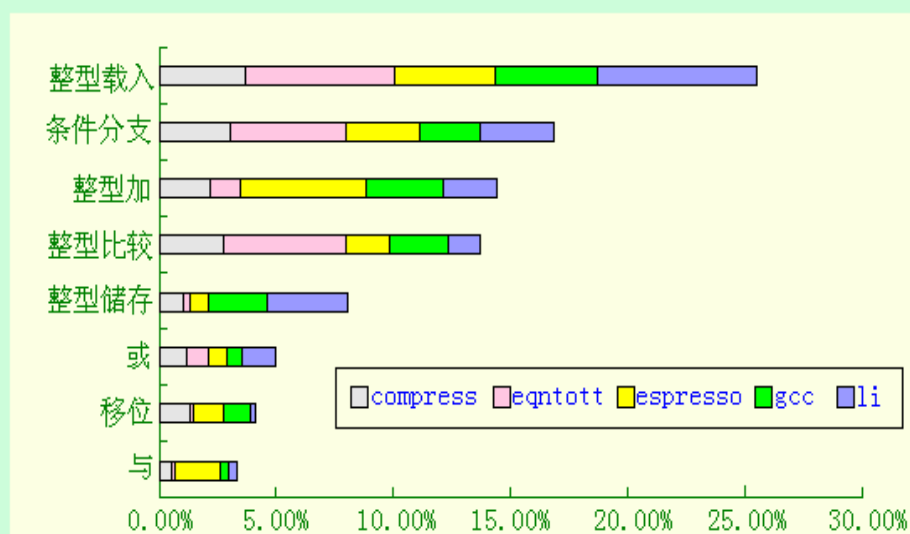


图 6

## 指令使用频率的浮点平均

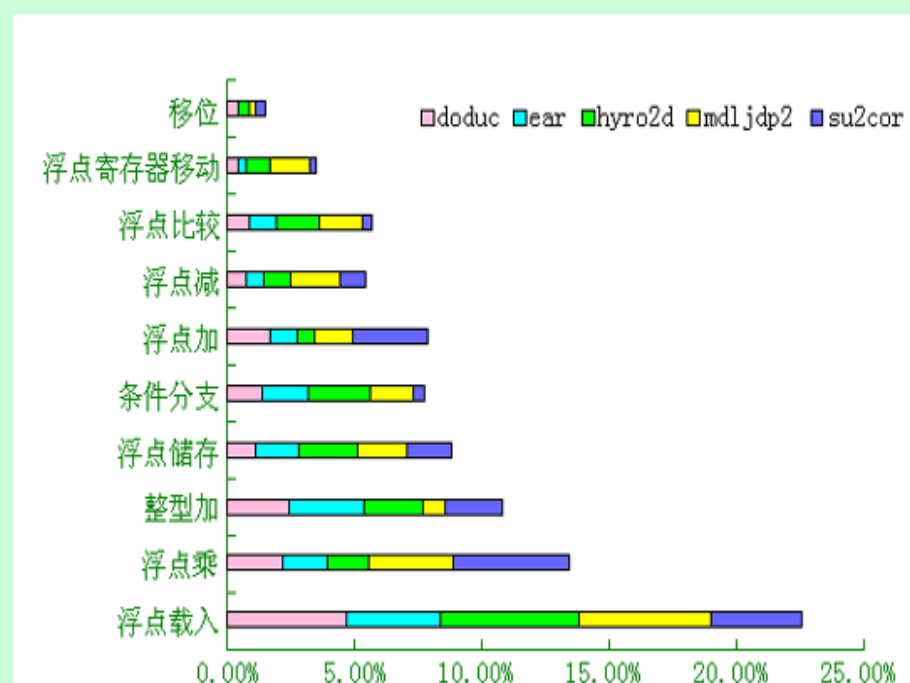


图 7

表 4 DLX 中的所有指令及其含义

指令实例	指令名称	含 义
数据传送	LB, LBU, SB	载入字节, 载入无符号字节, 储存字节
	LH, LHU, SH	载入半字, 载入无符号半字, 储存半字
	LW, SW	载入字, 储存字
	LF, LD, SF, SD	载入单精度浮点, 载入双精度浮点, 储存单精度浮点, 储存双精度浮点
	MOVI2S, MOVS2I	将通用寄存器中的内容移入特殊寄存器, 将特殊寄存器中的内容移入通用寄存器
	MOVF, MOVD	将一个单精度/双精度浮点寄存器的内容拷贝到另一个单精度/双精度浮点寄存器
	MOVFP2I, MOVI2FP	将 32 位浮点寄存器中的内容移入整型寄存器, 将 32 位整型寄存器中的内容移入浮点寄存器
	ADD,ADDI,ADDU,ADDUI	将 32 位浮点寄存器中的内容移入整型寄存器, 将 32 位整型寄存器中的内容移入浮点寄存器
	SUB,SUBI,SUBU,SUBUI	带符号加, 带符号立即值加, 无符号加, 无符号立即值加
	MULT,MULTU,DIV,DIVU	带符号减, 带符号立即值减, 无符号减, 无符号立即值减
算术/逻辑	AND,ANDI	带符号乘, 无符号乘, 带符号除, 无符号除 与, 和立即值与
	OR,ORI,XOR,XORI	或, 和立即值或, 异或, 和立即值异或
	LHI	载入高位立即值
	SLL,SRL,SRA,	包含了立即值 (S_I) 和变量 (S_) 的移位操作, 移位
	SLLI,SRLI,SRAI	有: 逻辑左移, 逻辑右移和算术右移
	S_,S_I	设置条件, "_"可以是 LT,GT,LE,GE,EQ,NE
	BEQZ,BNEZ	根据指定通用寄存器的内容等于/不等于 0 分支
	BFPT,BFPF	测试浮点状态寄存器中的比较位为真/假进行分支
	J,JR	跳转, 基于寄存器的跳转
	JAL,JALR	跳转并链接, 基于寄存器的跳转并链接
控 制	TRAP	转换到操作系统
	RFE	从异常恢复用户模式

	ADDD,ADDF	双精度浮点加，单精度浮点加
	SUBD,SUBF	双精度浮点减，单精度浮点减
	MULTD,MULTF	双精度浮点乘，单精度浮点乘
	DIVD,DIVF	双精度浮点除，单精度浮点除
浮 点	CVTF2D,CVTF2I,CVTD2F,CTD2I,CVTI2F,CVTI2D	转换指令，CVTx2y 表示从类型 x 转换到类型 y，其中 x 和 y 可以是 I（整型）、D（双精度浮点）、F（单精度浮点）
	_D,_F	双精度浮点和单精度浮点比较，"_"可以是 LT、GT、LE、GE、EQ、NE，根据比较结果设置浮点状态寄存器中的位

表 5 基于 SPECint92 基准程序集的指令使用频率测量统计结果

指 令	compress	eqntott	Espresso	gcc(cc1)	li	整型平均
载入	19.8%	30.6%	20.9%	22.8%	31.3%	26%
存储	5.6%	0.6%	5.1%	14.3%	16.7%	9%
加	14.4%	8.5%	23.8%	14.6%	11.1%	14%
减	1.8%	0.3%		0.5%		0%
乘				0.1%		0%
除						0%
比较	15.4%	26.5%	8.3%	12.4%	5.4%	13%
载入立即值	8.1%	1.5%	1.3%	6.8%	2.4%	3%
条件分支	17.4%	24.0%	15.0%	11.5%	14.6%	16%
无条件分支	1.5%	0.9%	0.5%	1.3%	1.8%	1%
调用	0.1%	0.5%	0.4%	1.1%	3.1%	1%
返回，跳转	0.1%	0.5%	0.5%	1.5%	3.5%	1%
移位	6.5%	0.3%	7.0%	6.2%	0.7%	4%
与	2.1%	0.1%	9.4%	1.6%	2.1%	3%
或	6.0%	5.5%	4.8%	4.2%	6.2%	5%
其它						
（异或，非）	1.0%		2.0%	0.5%	0.1%	1%
载入浮点数						0%

储存浮点数	0%
浮点加	0%
浮点减	0%
浮点乘	0%
浮点除	0%
浮点比较	0%
浮点寄存器移动	0%
其它浮点操作	0%

表 6 基于 SPECfp92 基准程序集的指令使用频率测量统计结果

指 令	doduc	ear	hydro2d	mdljdp2	su2cor	整型平均
载入	1.4%	0.2%	0.1%	1.1%	3.6%	1%
存储	1.3%	0.1%		0.1%	1.3%	1%
加	13.6%	13.6%	10.9%	4.7%	9.7%	11%
减	0.3%		0.2%	0.7%		0%
乘						0%
除						0%
比较	3.2%	3.1%	1.2%	0.3%	1.3%	2%
载入立即值	2.2%		0.2%	2.2%	0.9%	1%
条件分支	8.0%	10.1%	11.7%	9.3%	2.6%	8%
无条件分支	0.9%	0.4%		0.4%	0.1%	0%
调用	0.5%	1.9%			0.3%	1%
返回, 跳转	0.6%	1.9%			0.3%	1%
移位	2.0%	0.2%	2.4%	1.3%	2.3%	2%
与	0.4%	0.1%			0.3%	0%
或		0.2%	0.1%	0.1%	0.1%	0%
其它 (异或, 非)						0%
载入浮点数	23.3%	19.8%	24.1%	25.9%	21.6%	23%
储存浮点数	5.7%	11.4%	9.9%	10.0%	9.8%	9%
浮点加	8.8%	7.3%	3.6%	8.5%	12.4%	8%

浮点减	3.8%	3.2%	7.9%	10.4%	5.9%	6%
浮点乘	12.0%	9.6%	9.4%	13.9%	21.6%	13%
浮点除	2.3%		1.6%	0.9%	0.7%	1%
浮点比较	4.2%	6.4%	10.4%	9.3%	0.8%	6%
浮点寄存器移动	2.1%	1.8%	5.2%	0.9%	1.9%	2%
其它浮点操作	2.4%	8.4%	0.2%	0.2%	1.2%	2%

## 7. DLX 指令

add

Ex: add r1,r2,r3

$R[regc] \leftarrow R[rega] + R[regb]$

All are signed integers.

addd

Ex: addd f4,f4,f6

$D[dregc] \leftarrow D[drega] + D[dregb]$

All are double precision floating point numbers.

addf

Ex: addf f3,f4,f5

$F[fregc] \leftarrow F[frega] + F[fregb]$

All are single precision floating point numbers.

addi

Ex: addi r5,r2,#5

$R[regb] \leftarrow R[rega] + \text{imm16}$

All are signed integers.

addu

Ex: addu r2,r3,r4

$R[regc] \leftarrow R[rega] + R[regb]$

All are unsigned integers.

addui

Ex: addui r2,r3,#28

$R[regb] \leftarrow R[rega] + \text{uimm16}$

All are unsigned integers.

and

Ex: and r2,r3,r4

$R[regc] \leftarrow R[rega] \& R[regb]$

All are unsigned integers. Logical 'and' is performed on a bitwise basis.

andi

Ex: andi r3,r4,#5

$R[regb] \leftarrow R[rega] \& \text{uimm16}$

All are unsigned integers. Logical 'and' is performed on a bitwise basis.

beqz

Ex: beqz r1,label

if (R[rega] == 0) PC  $\leftarrow$  PC + imm16 + 4

bfpf

Ex: bfpf label

if (fps == 0) PC  $\leftarrow$  PC + imm16 + 4

fps is the floating point status bit.

bfpt

Ex: bfpt label

if (fps == 1) PC  $\leftarrow$  PC + imm16 + 4

fps is the floating point status bit.

bnez

Ex: bnez r1,label

if (R[rega] != 0) PC  $\leftarrow$  PC + imm16 + 4

cvtd2f

Ex: cvtd2f f1,f4

F[fregc]  $\leftarrow$  (float) D[drega]

Converts double precision floating point value to single precision floating point value.

cvtd2i

Ex: cvtd2i f1,f0

F[fregc]  $\leftarrow$  (int) D[drega]

Converts double precision floating point value to integer.

cvtf2d

Ex: cvtf2d f4,f9

D[dregc]  $\leftarrow$  (double) F[frega]

Converts single precision float to double.

cvtf2i

Ex: cvtf2i f3,f4

F[fregc]  $\leftarrow$  (int) F[frega]

Converts single precision float to integer.

cvti2d

Ex: cvti2d f2,f9

D[dregc]  $\leftarrow$  (double) F[frega]

Converts a signed integer to double precision float.

cvti2f

Ex: cvti2f f2,f5

F[fregc]  $\leftarrow$  (float) F[frega]

Converts a signed integer to single precision float.

div

Ex: div f2,f2,f3

F[fregc]  $\leftarrow$  F[frega] / F[fregb]

All are signed integers.

divd

Ex: divd f4,f4,f6

D[dregc]  $\leftarrow$  D[drega] / D[dregb]

All are double precision floats.



divf

Ex: divf f2,f3,f6

$F[\text{fregc}] \leftarrow F[\text{frega}] / F[\text{fregb}]$

All are single precision floats.

divu

Ex: divu f2,f3,f4

$F[\text{fregc}] \leftarrow F[\text{frega}] / F[\text{fregb}]$

All are unsigned integers.

eqd

Ex: eqd f2,f4

if ( $D[\text{drega}] == D[\text{dregb}]$ ) fps = 1 else fps = 0

Both are double precision floats.

eqf

Ex: eqf f3,f5

if ( $F[\text{frega}] == F[\text{fregb}]$ ) fps = 1 else fps = 0

Both are single precision floats.

ged

Ex: ged f8,f6

if ( $D[\text{drega}] \geq D[\text{dregb}]$ ) fps = 1 else fps = 0

Both are double precision floats.

gef

Ex: gef f3,f6

if ( $F[\text{frega}] \geq F[\text{fregb}]$ ) fps = 1 else fps = 0

Both are single precision floats.

gtd

Ex: gtd f8,f6

if ( $D[\text{drega}] > D[\text{dregb}]$ ) fps = 1 else fps = 0

Both are double precision floats.

gtf

Ex: gtf f3,f6

if ( $F[\text{frega}] > F[\text{fregb}]$ ) fps = 1 else fps = 0

Both are single precision floats.

j

Ex: j label

$PC \leftarrow PC + \text{imm26} + 4$

Unconditionally jumps relative to the PC of the next instruction. imm26 is a 26-bit signed integer.

jal

Ex: jal label

$R31 \leftarrow PC + 8; PC \leftarrow PC + \text{imm26} + 4$

Saves a return address in register 31 and jumps relative to the PC of the next instruction. imm26 is a 26-bit signed integer.

jalr

Ex: jalr r2

$R31 \leftarrow PC + 8; PC \leftarrow R[\text{rega}]$

Saves a return address in register 31 and does an absolute jump to the target address contained in R[rega].

jr

Ex: jr r3

$PC \leftarrow R[rega]$

R[rega] is treated as an unsigned integer. Does an absolute jump to the target address contained in R[rega].

lb

Ex: lb r1,40-4(r2)

$R[regb] \leftarrow (\text{sign extended}) M[\text{imm16} + R[rega]]$

One byte of data is read from the effective address computed by adding signed integer imm16 and signed integer R[rega]. The byte from memory is then sign extended to 32-bits and stored in register R[regb].

lbu

Ex: lbu r2,label-786+4(r3)

$R[regb] \leftarrow 0^{24} \# M[\text{imm16} + R[rega]]$

One byte of data is read from the effective address computed by adding signed integer imm16 and signed integer R[rega]. The byte from memory is then zero extended to 32 bits and stored in register R[regb].

ld

Ex: ld f2,240(r1)

$D[dregb] \leftarrow 64 M[\text{imm16} + R[rega]]$

Two words of data are read from the effective address computed by adding signed integer imm16 and unsigned integer R[rega] and stored in double register D[dregb]. This is equivalent to two lf instructions:

$F[fregb] \leftarrow M[\text{imm16} + R[rega]]$

$F[freg(b+1)] \leftarrow M[\text{imm16} + R[rega] + 4]$

where F[freg(b+1)] is the next fp register after F[fregb] in sequence, and all values are simply copied and not converted.)

led

Ex: led f8,f6

if ( $D[drega] \leq D[dregb]$ ) fps = 1 else fps = 0

Both are double precision floats.

lef

Ex: lef f3,f6

if ( $F[frega] \leq F[fregb]$ ) fps = 1 else fps = 0

Both are single precision floats.

lf

Ex: lf f6,76(r4)

$F[fregb] \leftarrow M[\text{imm16} + R[rega]]$

One word of data is read from the effective address computed by adding signed integer imm16 and signed integer R[rega] and stored in fp register F[fregb].

lh

Ex: lh r1,32(r3)

$R[\text{regb}] \leftarrow (\text{sign extended}) M[\text{imm16} + R[\text{rega}]]$

Two bytes of data are read from the effective address computed by adding signed integer imm16 and signed integer R[rega]. The address must be half-word aligned. The half-word from memory is then sign extended to 32 bits and stored in register R[regb].

lhi

Ex: lhi r3, #-40

$R[\text{regb}] \leftarrow \text{imm16} \ll 0^{16}$

Loads the 16 bit immediate value imm16 into the most significant half of an integer register and clears the least significant half.

lhu

Ex: lhu r2, -40+4(r3)

$R[\text{regb}] \leftarrow 0^{16} \ll M[\text{imm16} + R[\text{rega}]]$

Two bytes of data are read from the effective address computed by adding signed integer imm16 and signed integer R[rega]. The address must be half-word aligned. The half-word from memory is then zero extended to 32 bits and stored in register R[regb].

ltd

Ex: ltd f8, f6

if (D[drega] < D[dregb]) fps = 1 else fps = 0

Both are double precision floats.

ltf

Ex: ltf f3, f6

if (F[frega] < F[fregb]) fps = 1 else fps = 0

Both are single precision floats.

lw

Ex: lw r19, label+63(r8)

$R[\text{regb}] \leftarrow M[\text{imm16} + R[\text{rega}]]$

One word is read from the effective address computed by adding signed integer imm16 and unsigned integer R[rega] and is stored in R[regb].

movd

Ex: movd f2, f4

$D[\text{dregc}] \leftarrow D[\text{drega}]$

Copies two words from double register D[drega] to double register D[dregc].

movf

Ex: movf f1, f2

$F[\text{fregc}] \leftarrow F[\text{frega}]$

Copies one word from fp register F[frega] to fp register F[fregc].

movfp2i

Ex: movfp2i r3, f0

$R[\text{regc}] \leftarrow F[\text{frega}]$

Copies one word from fp register F[frega] to integer register R[regc].

movi2fp

Ex: movi2fp f0, r3

$F[\text{fregc}] \leftarrow R[\text{rega}]$

Copies one word from integer register R[rega] to fp register F[fregc].

movi2s

Ex: movi2s r1

*Unspecified*

Copies one word from integer register R[rega] to a special register.

movs2i

Ex: movs2i r2

*Unspecified*

Copies one word from a special register to integer register R[rega].

mult

Ex: mult f2,f3,f4

$F[\text{fregc}] \leftarrow F[\text{frega}] * F[\text{fregb}]$

All are signed integers.

multd

Ex: multd f2,f4,f6

$D[\text{dregc}] \leftarrow D[\text{drega}] * D[\text{dregb}]$

All are double precision floats.

multf

Ex: multf f3,f4,f5

$F[\text{fregc}] \leftarrow F[\text{frega}] * F[\text{fregb}]$

All are single precision floats.

multu

Ex: multu f2,f3,f4

$F[\text{fregc}] \leftarrow F[\text{frega}] * F[\text{fregb}]$

All are unsigned integers.

ned

Ex: ned f8,f6

if ( $D[\text{drega}] \neq D[\text{dregb}]$ ) fps = 1 else fps = 0

Both are double precision floats.

nef

Ex: nef f3,f6

if ( $F[\text{frega}] \neq F[\text{fregb}]$ ) fps = 1 else fps = 0

Both are single precision floats.

nop

Ex: nop

Idles one cycle.

or

Ex: or r2,r3,r4

$R[\text{regc}] \leftarrow R[\text{rega}] | R[\text{regb}]$

All are unsigned integers. Logical 'or' is performed on a bitwise basis.

ori

Ex: ori r3,r4,#5

$R[\text{regb}] \leftarrow R[\text{rega}] | \text{uimm16}$

All are unsigned integers. Logical 'or' is performed on a bitwise basis.

rfe

Ex: rfe

*Unspecified*

Return from exception.

sb

Ex: sb label-41(r3),r2

$M[\text{imm16} + R[\text{rega}]] \leftarrow 8 R[\text{regb}]_{24..31}$

One byte of data from the least significant byte of register R[regb] is written to the effective address computed by adding signed integer imm16 and signed integer R[rega].

sd

Ex: sd 200(r4),f6

$M[\text{imm16} + R[\text{rega}]] \leftarrow 64 D[\text{dregb}]$

Two words from double register D[dregb] are written to the effective address computed by adding signed integer imm16 and signed integer R[rega].

seq

Ex: seq r1,r2,r3

if (R[rega] == R[regb]) R[regc]  $\leftarrow$  1 else R[regc]  $\leftarrow$  0

All are signed integers.

seqi

Ex: seqi r14,r3,#3

if (R[rega] == imm16) R[regb]  $\leftarrow$  1 else R[regb]  $\leftarrow$  0

All are signed integers.

sf

Ex: sf 121(r3),f1

$M[\text{imm16} + R[\text{rega}]] \leftarrow F[\text{fregb}]$

One word from fp register F[fregb] is written to the effective address computed by adding signed integer imm16 and signed integer R[rega].

sge

Ex: sge r1,r3,r4

if (R[rega] >= R[regb]) R[regc]  $\leftarrow$  1 else R[regc]  $\leftarrow$  0

All are signed integers.

sgei

Ex: sgei r2,r1,#6

if (R[rega] >= imm16) R[regb]  $\leftarrow$  1 else R[regb]  $\leftarrow$  0

All are signed integers.

sgt

Ex: sgt r4,r5,r6

if (R[rega] > R[regb]) R[regc]  $\leftarrow$  1 else R[regc]  $\leftarrow$  0

All are signed integers.

sgti

Ex: sgti r1,r2,#-3000

if (R[rega] > imm16) R[regb]  $\leftarrow$  1 else R[regb]  $\leftarrow$  0

All are signed integers.

sh

Ex: sh 421(r3),r5

$M[\text{imm16} + R[\text{rega}]] \leftarrow 16 R[\text{regb}]_{16..31}$

Two bytes of data from the least significant half of register  $R[\text{regb}]$  are written to the effective address computed by adding signed integer  $\text{imm16}$  and unsigned integer  $R[\text{rega}]$ . The effective address must be halfword aligned.

sle

Ex: sle r1,r2,r3

if ( $R[\text{rega}] \leq R[\text{regb}]$ )  $R[\text{regc}] \leftarrow 1$  else  $R[\text{regc}] \leftarrow 0$

All are signed integers.

slei

Ex: slei r8,r5,#345

if ( $R[\text{rega}] \leq \text{imm16}$ )  $R[\text{regb}] \leftarrow 1$  else  $R[\text{regb}] \leftarrow 0$

All are signed integers.

sll

Ex: sll r6,r7,r11

$R[\text{regc}] \leftarrow R[\text{rega}] \ll R[\text{regb}]_{27..31}$

All are unsigned integers.  $R[\text{rega}]$  is logically shifted left by the low five bits of  $R[\text{regb}]$ . Zeros are shifted into the least-significant bit.

slli

Ex: slli r1,r2,#3

$R[\text{regb}] \leftarrow R[\text{rega}] \ll \text{uimm16}_{27..31}$

All are unsigned integers.  $R[\text{rega}]$  is logically shifted left by the low five bits of  $\text{uimm16}$ . Zeros are shifted into the least-significant bit. (Actually only the bottom five bits of  $R[\text{regb}]$  are used.)

slt

Ex: slt r3,r4,r5

if ( $R[\text{rega}] < R[\text{regb}]$ )  $R[\text{regc}] \leftarrow 1$  else  $R[\text{regc}] \leftarrow 0$

All are signed integers.

slti

Ex: slti r1,r2,#22

if ( $R[\text{rega}] < \text{imm16}$ )  $R[\text{regb}] \leftarrow 1$  else  $R[\text{regb}] \leftarrow 0$

All are signed integers.

sne

Ex: sne r1,r2,r3

if ( $R[\text{rega}] \neq R[\text{regb}]$ )  $R[\text{regc}] \leftarrow 1$  else  $R[\text{regc}] \leftarrow 0$

All are signed integers.

snei

Ex: snei r4,r5,#89

if ( $R[\text{rega}] \neq \text{imm16}$ )  $R[\text{regb}] \leftarrow 1$  else  $R[\text{regb}] \leftarrow 0$

All are signed integers.

sra

Ex: sra r1,r2,r3

$R[\text{regc}] \leftarrow (R[\text{rega}]_0 \wedge R[\text{regb}]) \# (R[\text{rega}] \gg R[\text{regb}])_{R[\text{regb}]..31}$

$R[\text{rega}]$  and  $R[\text{regc}]$  are signed integers.  $R[\text{regb}]$  is an unsigned integer.  $R[\text{rega}]$  is arithmetically shifted right by  $R[\text{regb}]$ . The sign bit is shifted into the most-significant bit. (Actually uses only the five low order bits of  $R[\text{regb}]$ .)

srai

Ex: srai r2,r3,#5

$R[\text{regb}] \leftarrow (R[\text{rega}]_{31})^{\wedge} \text{uimm16} \# \# (R[\text{rega}] \gg \text{uimm16})_{\text{uimm16}..31}$

$R[\text{rega}]$  and  $R[\text{regc}]$  are signed integers.  $\text{uimm16}$  is an unsigned integer.  $R[\text{rega}]$  is arithmetically shifted right by  $R[\text{regb}]$ . The sign bit is shifted into the most-significant bit. (Actually uses only the five low order bits of  $\text{uimm16}$ .)

srl

Ex: srl r15,r2,r3

$R[\text{regc}] \leftarrow R[\text{rega}] \gg R[\text{regb}]_{27..31}$

All are unsigned integers.  $R[\text{rega}]$  is arithmetically shifted right by  $R[\text{regb}]$ . Zeros are shifted into the most significant bit.

srli

Ex: srli r1,r2,#5

$R[\text{regb}] \leftarrow R[\text{rega}] \gg \text{uimm16}_{27..31}$

All are unsigned integers.  $R[\text{rega}]$  is arithmetically shifted right by  $\text{uimm16}$ . Zeros are shifted into the most significant bit.

sub

Ex: sub r3,r2,r1

Ex:  $R[\text{regc}] \leftarrow R[\text{rega}] - R[\text{regb}]$

All are signed integers.

subd

Ex: subd f2,f4,f6

$D[\text{dregc}] \leftarrow D[\text{drega}] - D[\text{dregb}]$

All are double precision floats.

subf

Ex: subf f3,f4,f6

$F[\text{fregc}] \leftarrow F[\text{frega}] - F[\text{fregb}]$

All are single precision floats.

subi

Ex: subi r15,r16,#964

$R[\text{regb}] \leftarrow R[\text{rega}] - \text{imm16}$

All are signed integers.

subu

Ex: subu r3,r2,r1

$R[\text{regc}] \leftarrow R[\text{rega}] - R[\text{regb}]$

All are unsigned integers.

subui

Ex: subui r1,r2,#53

$R[\text{regb}] \leftarrow R[\text{rega}] - \text{uimm16}$

All are unsigned integers.

sw

Ex: sw 21(r13),r6

$M[\text{imm16} + R[\text{rega}]] \leftarrow R[\text{regb}]$

One word from integer register  $R[\text{regb}]$  is written to the effective address computed by adding

signed integer imm16 and unsigned integer R[rega].

trap

Ex: trap #3

Execute trap with number in immediate field

Saves state and jumps to an operating system procedure located at an address in the interrupt vector table. In our systems, this is simulated by calling the procedure corresponding to the trap number.

xor

Ex: xor r2,r3,r4

$R[regc] \leftarrow F[rega] \text{ XOR } R[regb]$

All are unsigned integers. Logical 'xor' is performed on a bitwise basis.

xori

Ex: xori r3,r4,#5

$R[regb] \leftarrow R[rega] \text{ XOR } \text{uimm16}$

All are unsigned integers. Logical 'xor' is performed on a bitwise basis.

### 符号说明

符 号	意 义
x_y	x 的第 y 位
x_y..z	x 的第 y 到第 z 位
x^y	xx...x (x 重复 y 次)
x##y	xy (x 与 y 拼接)
IR	指令寄存器
IAR	中断地址寄存器
PC	程序计数器
R[rega]	整数寄存器[IR_6..10]
R[regb]	整数寄存器[IR_11..15]
R[regc]	整数寄存器[IR_16..20]
F[frega]	浮点寄存器[IR_6..10]
F[fregb]	浮点寄存器[IR_11..15]
F[fregc]	浮点寄存器[IR_16..20]
D[drega]	双精度浮点寄存器[IR_6..10]
D[dregb]	双精度浮点寄存器[IR_11..15]
D[dregc]	双精度浮点寄存器[IR_16..20]
imm16	$(IR\_16)^{16} \# IR\_16..31$
uimm16	$0^{16} \# IR\_16..31$



---

imm26	(IR_6)^6 ## IR_6..31
fps	浮点状态位
←	32 位传送
←n	n 位传送

---

### 三. WinDLX 教程

DLX 处理器 (发音为 “DeLuXe”) 是 Hennessy 和 Patterson 合著一书《Computer Architecture - A Quantitative Approach》中流水线处理器的例子。WinDLX 是一个基于 Windows 的模拟器。本教程通过一个实例介绍 WinDLX 的使用方法。WinDLX 模拟器能够演示 DLX 流水线是如何工作的。

本教程使用的例子非常简单，它并没有囊括 WinDLX 的各个方面，仅仅作为使用 WinDLX 的入门级介绍。当你阅读完本教程后，请参考帮助文件。通过按 F1 键，你可以在任何时候获得相关的帮助信息。

虽然我们将详细讨论例子中的各个阶段，但你应具备基本的使用 Windows 的知识。现假定你知道如何启动 Windows，使用滚动条滚动，双击执行以及激活窗口。

#### 1、安 装

WinDLX 包含 windlx.exe 和 windlx.hlp 文件。同时, 还需要一些扩展名为.s 的汇编代码文件。在本实验中将使用 fact.s 和 input.s 等多个汇编代码文件。

如果你熟悉 Windows 应用程序的安装, 那么将 fact.s 和 input.s 拷贝到 WinDLX 文件夹后, 你可以直接阅读下一部分。


如果你要在自己的计算机上安装 WinDLX, 请按以下步骤在 Windows 下安装:

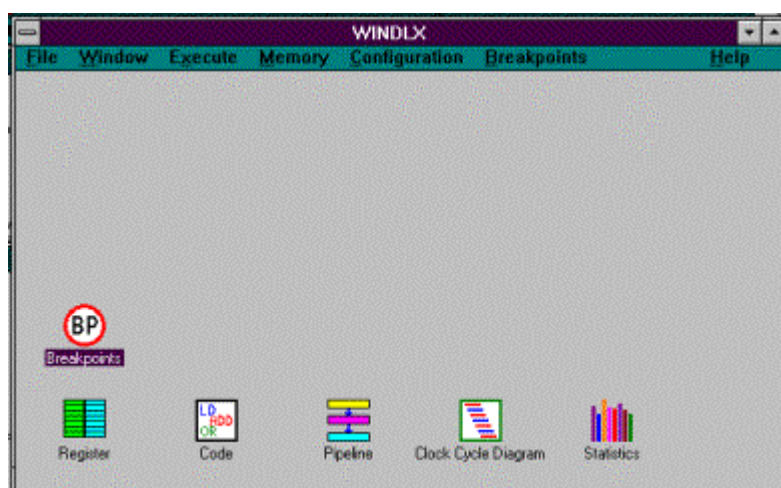
1. 为 WinDLX 创建目录, 例如 D:\WINDLX
2. 解压 WinDLX 软件包或拷贝所有的 WinDLX 文件(至少包含 windlx.exe, windlx.hlp, fact.s 和 input.s)到这个 WinDLX 目录。

## 2、一个完整的例子

我们使用 WinDLX 汇编器中的汇编文件 fact.s, 这个程序计算数(通过键盘输入)的阶乘。这需要用到文件 input.s, 它用于接收从键盘输入的数。

### 2.1 开始和配置 WinDLX

象启动任何 Windows 应用程序一样, 通过双击 WinDLX 图标  启动 WinDLX, 将出现一个带有六个图标的主窗口, 双击这些图标将弹出子窗口。后面将解释和介绍如何使用每一个窗口。



为了初始化模拟器, 点击 File 菜单中的 Reset all 菜单项, 弹出一个“Reset DLX”对话框。然后点击窗口中的“确认”按钮即可。

WinDLX 可以在多种配置下工作。你可以改变流水线的结构和时间要求、存储器大小和其他

几个控制模拟的参数。点击 Configuration / Floating Point Stages（点击 Configuration 打开菜单，然后点击 Floating Point Stages 菜单项），选择如下标准配置：

	Count	Delay
Addition Units:	1	2
Multiplication Units:	1	5
Division Units:	1	19

如果需要，可以通过点击相应区域来改变设置。然后，点击 OK 返回主窗口。

点击 Configuration / Memory Size，可以设置模拟处理器的存储器大小。应设置为 0x8000，然后，点击 OK 返回主窗口。

在 Configuration 菜单中的其他三个配置也可以设置，它们是：Symbolic addresses, Absolute Cycle Count 和 Enable Forwarding。点击相应菜单项后，在它的旁边将显示一个小钩。

## 2.2 装载测试程序

在开始模拟之前，至少应装入一个程序到主存。为此，选择 File / Load Code or Data，窗口中会列出目录中所有汇编程序。

我们在前面已经提到，fact.s 计算一个整型值的阶乘；input.s 中包含一个子程序，它读标准输入（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

选择文件的顺序很关键，它决定了文件在存储器中出现的顺序。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。

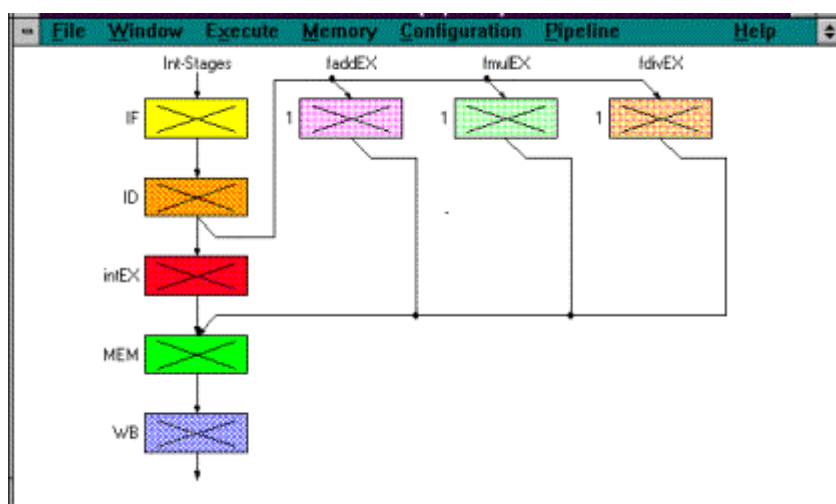
现在可以开始模拟工作了。

## 2.3 模拟

在主窗口中，你可以看见六个图标，它们分别为“Register”，“Code”，“Pipeline”，“Clock Cycle Diagram”，“Statistics”和“Breakpoints”。点击其中任何一个将弹出一个新窗口（子窗口）。在模拟过程中将介绍每一个窗口的特性和用法。

### 2.3.1 Pipeline 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，出现一个子窗口，窗口中用图表形式显示了 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。



此图显示 DLX 处理器的五个流水段和浮点操作 (加 / 减, 乘和除)的单元。

### 2.3.2 Code 窗口

我们来看一下 Code 窗口。双击图标，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。

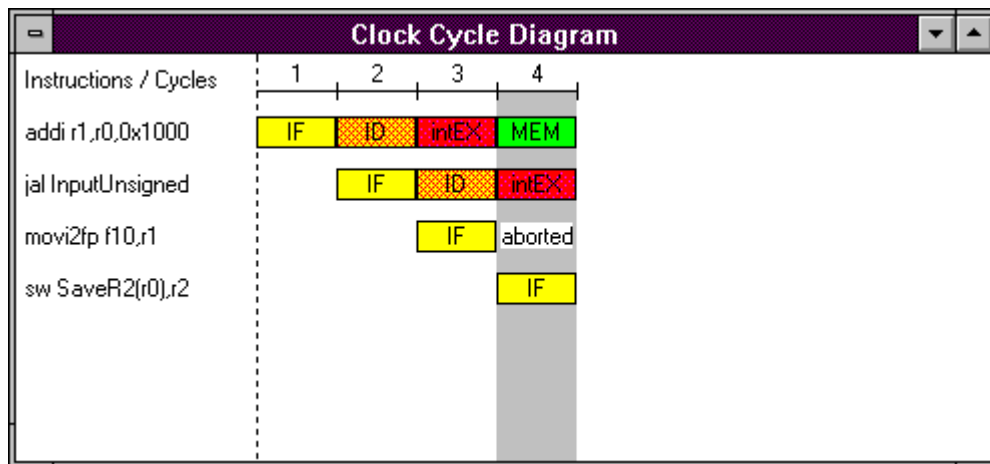
现在，点击主窗口中的 Execution 开始模拟。在出现的下拉式菜单中，点击 Single Cycle 或按 F7 键。

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，第一行的颜色变成橘黄色，下一行变成黄色。这些不同颜色指明命令处于流水线的哪一段。如果 Pipeline 窗口已经关闭，请双击相应图标重新打开它。如果窗口足够大，你能够看到命令“jal InputUnsigned”在 IF 段，“addi r1, r0, 0x1000”在第二段 ID。其他方框中带有“X”标志，表明没有处理有效信息。

再次按下 F7 键，代码窗口中的颜色会再改变，红色表明命令处入第三段“intEX”。再按下 F7，图形显示将变为：在代码窗口中，黄色出现在更下面的位置，并且可能是唯一彩色行。查看一下 Pipeline 窗口，你会发现 IF, intEX 和 MEM 段正在使用而 ID 段没有。为什么？

### 2.3.3 Clock Cycle Diagram 窗口

使所有子窗口图标化，然后打开 Clock Cycle Diagram 窗口。它显示流水线的时空图。



在窗口中，你将看到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 和 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 按钮关闭窗口。

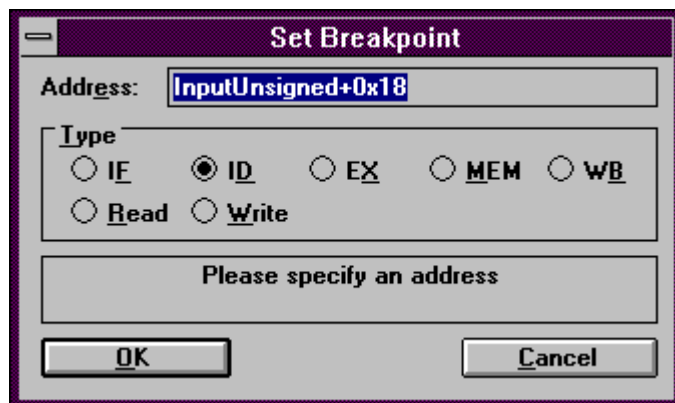
再一次点击 F7，第一条命令（addi）到达流水线的最后一段。如果了解某条命令执行后处理器内部会发生什么？你只要对准 Clock cycle diagram 窗口中相应命令所在行，然后双击它，弹出一个新窗口。窗口中会详细显示每一个流水段处理器内部的执行动作。这个窗口“Information about ...”作为将来的 Information 窗口。观察完后，点击 OK 按钮关闭窗口。双击第三行（movi2fp），你会看到它只执行了第一段（IF），这是因为出现跳转而被取消。（双击 Code 窗口中的某一行或者 Pipeline 窗口中的某一段，同样可以 Information 窗口。）

### 2.3.4 Breakpoint 窗口

当通过 Code 窗口观察代码时（如果未打开，双击图标 Code），你会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，请指向 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，然后单击主窗口菜单 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。通过此窗口，你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。



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，程序中中止执行。

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

现在你只要点击 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”，按“确认”按钮关闭。  
点击 Clock cycle diagram 窗口中的 trap 0x5 行，你将看到模拟正处于时钟周期 14。trap 0x5 行如下所示：



原因是：无论何时遇到一条 trap 指令时，DLX 处理器中的流水线将被清空。在 Information 窗口（双击 trap 行弹出）中，在 IF 段显示消息“3 stall(s) because of Trap-Pipeline-Clearing!”。（不要忘了按 OK 关闭窗口）。

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

### 2.3.5 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。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。

如果你希望不设置断点，而使模拟继续进行。办法是：点击 Execute / Multiple Cycles 或者按 F8 键，在新出现的窗口中输入 17，然后按 Enter 键，模拟程序将继续运行 17 个时钟周期。向上滚动 Clock cycle diagram 窗口，直到看到指令周期 72 到 78。在 EX 段，两个浮点操作 (multd and subd) 分别在不同的部件上运行，它们都需要多个周期才能结束。因而在它们之后的下一条指令能取指，译码和执行，然后暂停一个周期以允许 subd 完成 MEM 段。

### 2.3.6 Statistics 窗口

最后我们来看一下 Statistics 窗口。

按 F5 使程序完成执行，出现消息 “Trap #0 occurred” 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，如 RAW stalls: 17(7.91 % of all Cycles)。

在静态窗口中我们可以比较一下不同配置对模拟的影响。

现在我们看一看定向的作用。在前面的模拟过程中，我们采用了定向。如果不采用定向，执行时间将会怎样呢？

我们先看一下 Statistics 窗口中的各种统计数字：总的周期数(215) 和暂停数 (17 RAW, 25 Control, 12 Trap; 54 Total)，然后关闭窗口。点击 Configuration 中的 Enable Forwarding 使定向无效（去掉小钩），打开断点 Breakpoints 图标并点击 Breakpoints 菜单，删除所有断点，然后按 F5，键入 20 后，按 Enter，模拟程序一直运行到结束。重新查看静态窗口，你会看到控制暂停和 Trap 暂停仍然是同样的值，而 RAW 暂停从 17 变成了 53，总的模拟周期数增加到 236。利用这些值，你能够计算定向技术带来的加速比：

$$236 / 215 = 1.098$$

$DLX_{\text{forwarded}}$  比  $DLX_{\text{not forwarded}}$  快 9.8%。

## 3.实验有关知识的补充

流水线中的相关

流水线中的相关是指相邻或相近的指令因存在某种关联，后面的指令不能

在原指定的时钟周期开始执行。

一般来说，流水线中的相关主要分为如下三种类型：

(1) 结构相关：当硬件资源满足不了指令重叠执行的要求，而发生资源冲突时，就发生了结构相关。

(2) 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。

(3) 控制相关：当流水线遇到分支指令和其它能够改变 PC 值的指令时，就会发生控制相关。

一旦流水线中出现相关，必然会给指令在流水线中的顺利执行带来许多问题，如果不能很好地解决相关问题，轻则影响流水线的性能，重则导致错误的执行结果。消除相关的基本方法是让流水线暂停执行某些指令，而继续执行其它一些指令。

在后面的讨论中,我们约定:当一条指令被暂停时，在该暂停指令之后发射的所有指令都要被暂停，而在该暂停之前发射的指令则可继续进行，在暂停期间，流水线不会取新的指令。

### 3.3.1 流水线的结构相关

如果某些指令组合在流水线中重叠执行时，产生资源冲突，则称该流水线有结构相关。为了能够在流水线中顺利执行指令的所有可能组合，而不发生结构相关，通常需要采用流水化功能单元的方法或资源重复的方法。

许多流水线机器都是将数据和指令保存在同一存储器中。如果在某个时钟周期内，流水线既要完成某条指令对数据的存储器访问操作，又要完成取指令的操作，那么将会发生存储器访问冲突问题（如图 3.3.1 所示），产生结构相关。为了解决这个问题，可以让流水线完成前一条指令对数据的存储器访问时，暂停取后一条指令的操作（如图 3.3.2 所示）。该周期称为流水线的一个暂停周期。暂停周期一般也称为流水线气泡，或简称为气泡。从图 3.3.2 可以看出，在流水线



中插入暂停周期可以消除这种结构相关。

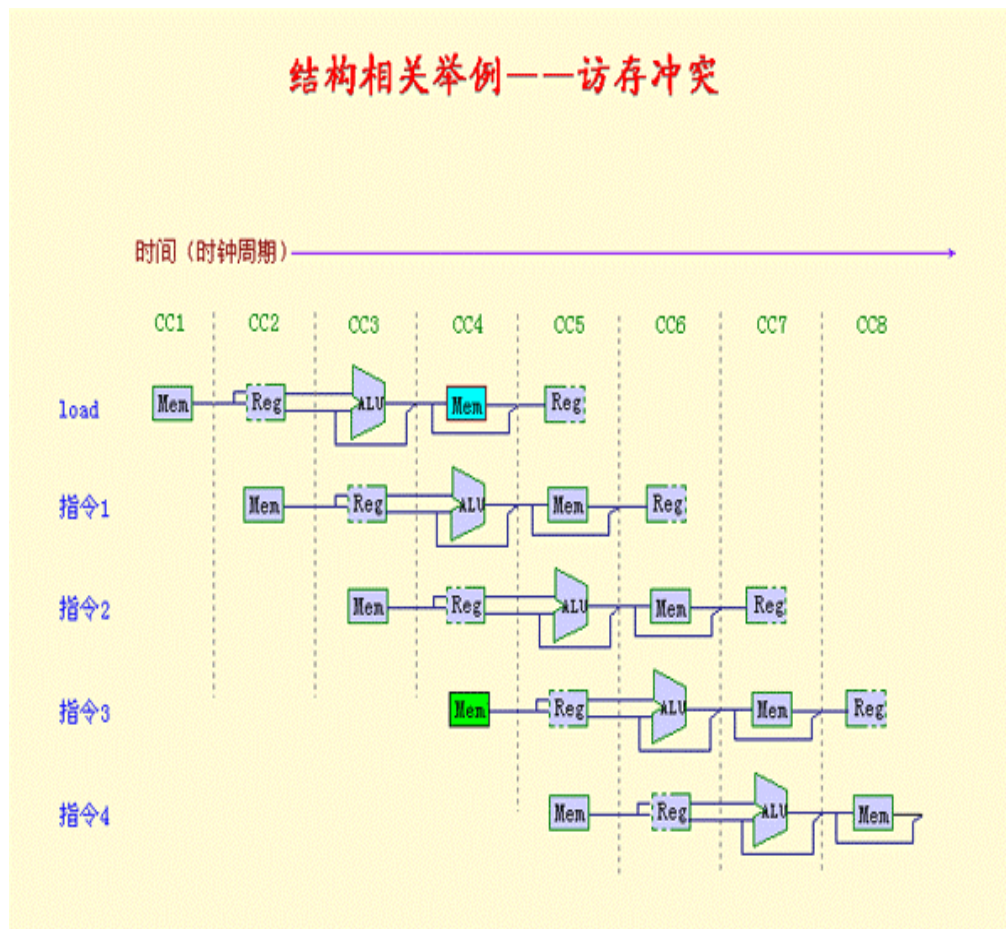


图 3.3.1 由于存储器访问冲突而带来的流水线结构相关

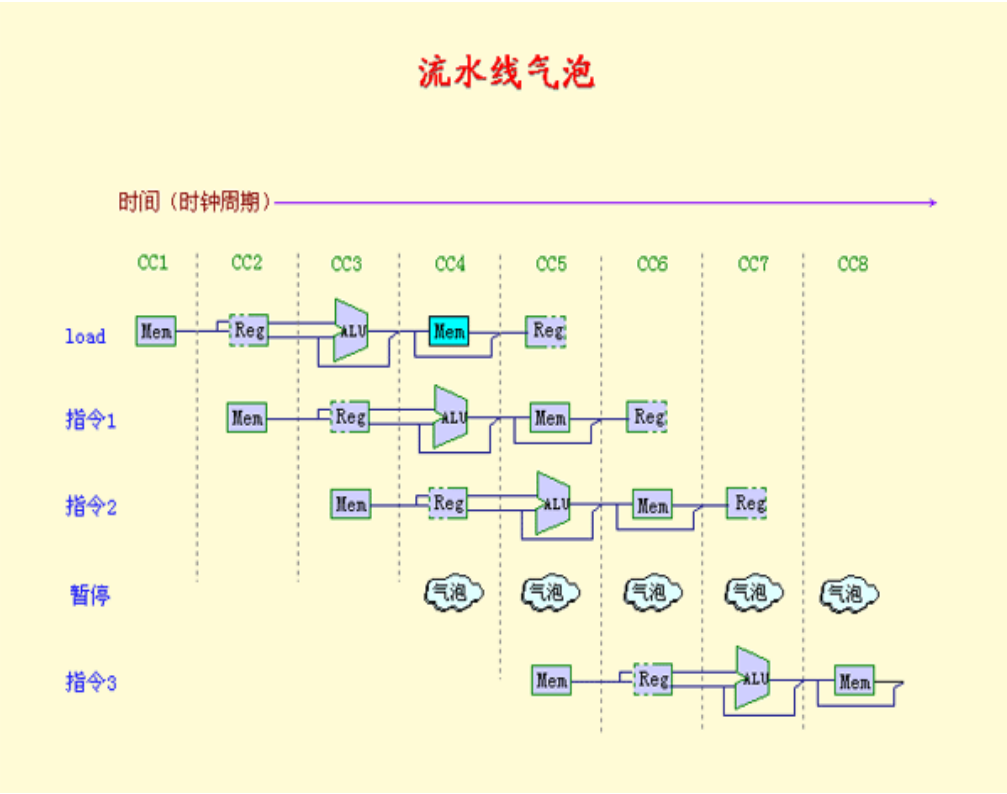


图 3.3.2 为消除结构相关而插入的流水线气泡

也可以用如图 3.3.3 所示的时空图来表示上述暂停情况。



图 3.3.3

由上可知，为消除结构相关而引入的暂停将影响流水线的性能。为了避免结构相关，可以考虑采用资源重复的方法。比如，在流水线机器中设置相互独立的指令存储器和数据存储器；也可以将 Cache 分割成指令 Cache 和数据 Cache。

假设不考虑流水线其它因素对流水线性能的影响，显然如果流水线机器没有结构相关，那么其 CPI 也较小。然而，为什么有时流水线设计者却允许结构相关的存在呢？主要有两个原因：

一是为了减少硬件代价，二是为了减少功能单元的延迟。如果为了避免结构相关而将流水线中的所有功能单元完全流水化，或者设置足够的硬件资源，那么所带来的硬件代价必定很大。

### 3.3.2 流水线的数据相关

#### (1) 数据相关简介

当指令在流水线中重叠执行时，流水线有可能改变指令读/写操作数的顺序，使得读/写操作顺序不同于它们非流水实现的顺序，这将导致数据相关。首先让我们考虑下列指令在流水线中的执行情况：

```
ADD  R1,  R2,  R3
SUB  R4,  R5,  R1
AND  R6,  R1,  R7
OR   R8,  R1,  R9
XOR  R10, R1,  R11
```

ADD 指令后的所有指令都要用到 ADD 指令的计算结果，如图 3.3.4 所示，ADD 指令在 WB 段才将计算结果写入寄存器 R1 中，但是 SUB 指令在其 ID 段就要从寄存器 R1 中读取该计算结果，这种情况就叫做数据相关。除非有措施防止这一情况出现，否则 SUB 指令读到的是错误的值。所以，为了保证上述指令序列的正确执行，流水线只好暂停 ADD 指令之后的所有指令，

直到 ADD 指令将计算结果写入寄存器 R1 之后,再启动 ADD 指令之后的指令继续执行。

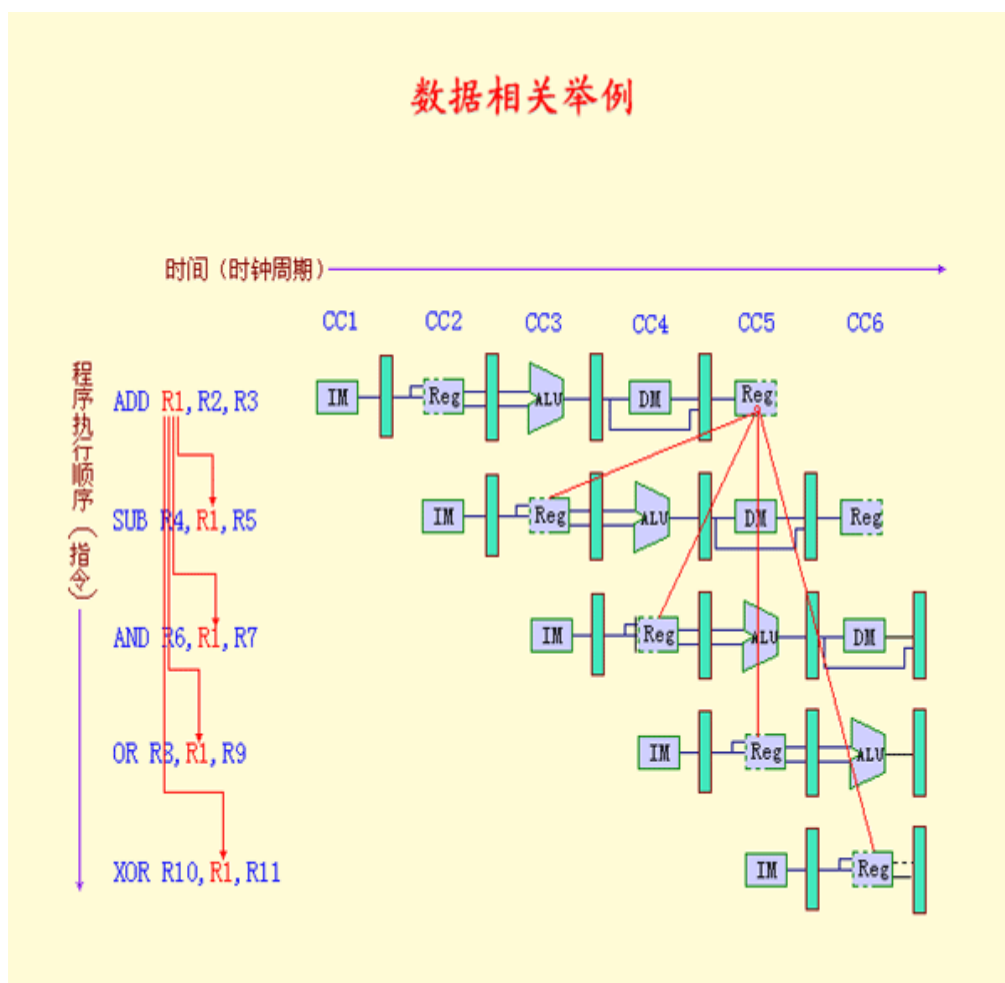


图 3.3.4 流水线的数据相关

从图 3.3.4 还可以看到, AND 指令同样也将受到这种相关关系的影响。ADD 指令只有到第五个时钟周期末尾才能结束对寄存器 R1 的写操作,所以 AND 指令在第四个时钟周期从寄存器 R1 中读出的值也是错误的。而 XOR 指令则可以正常操作,因为它是在第六个时钟周期读寄存器 R1 的内容。

另外,利用 DLX 流水线的一种简单技术,可以使流水线顺利执行 OR 指令。这种技术就是:在 DLX 流水线中,约定在时钟周期的后半部分进行寄存器文件的读操作,而在时钟周期的前半部分进行寄存器文件的写操作。在本章的图中,我们将寄存器文件的边框适当地画成虚线来表示这种技术。

## (2) 通过定向技术减少数据相关带来的暂停

图 3.3.4 中的数据相关问题可以采用一种称为定向(也称为旁路或短路)的简单技术来解决。定向技术的主要思想是:在某条指令(如图 3.3.4 中的 ADD 指令)产生一个计算结果之前,其它指令(如图 3.3.4 中的 SUB 和 AND 指令)并不真正需要该计算结果,如果能够将该计算结

果从其产生的地方（寄存器文件 EX/MEM）直接送到其它指令需要它的地方（ALU 的输入寄存器），那么就可以避免暂停。基于这种考虑，定向技术的要点可以归纳为：

- (1) 寄存器文件 EX/MEM 中的 ALU 的运算结果总是回送到 ALU 的输入寄存器。
- (2) 当定向硬件检测到前一个 ALU 运算结果的写入寄存器就是当前 ALU 操作的源寄存器时，那么控制逻辑将前一个 ALU 运算结果定向到 ALU 的输入端，后一个 ALU 操作就不必从源寄存器中读取操作数。

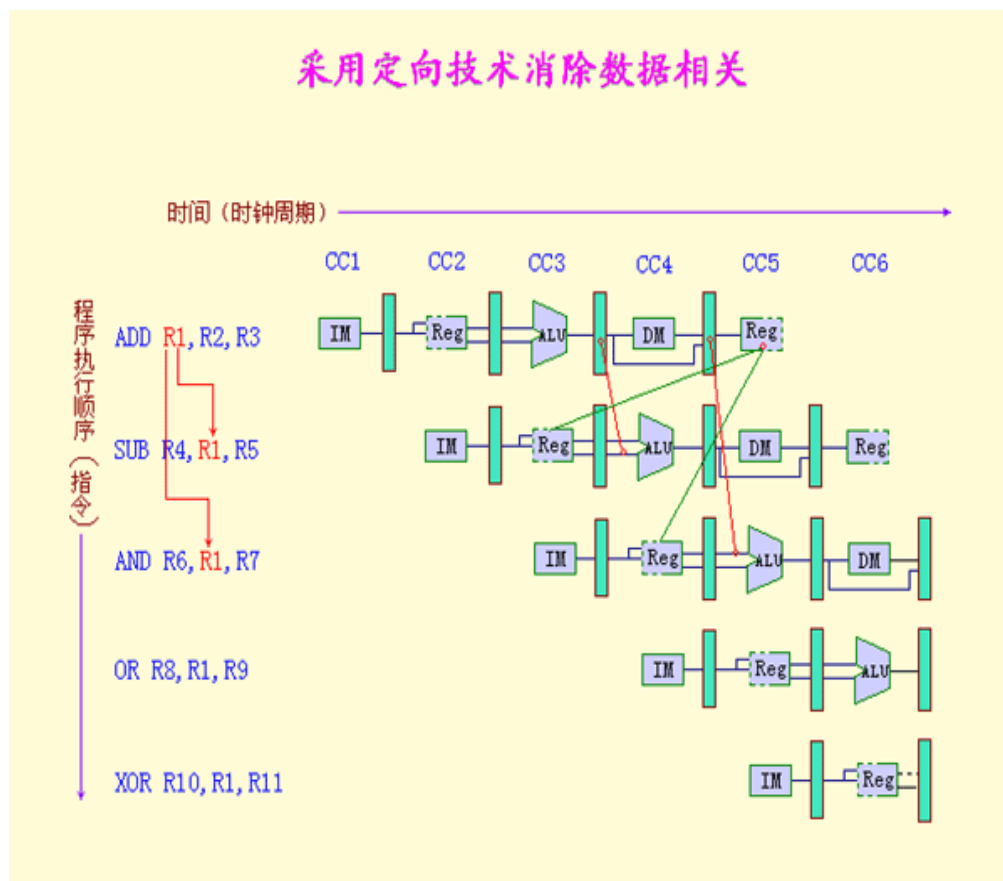


图 3.3.5

图 3.3.4 还表明，流水线中的指令所需要的定向结果可能并不仅仅是前一条指令的计算结果，而且还有可能是前面与其不相邻指令的计算结果，图 3.3.5 是采用了定向技术后上述例子的执行情况，其中寄存器文件和功能单元之间的箭头表示定向路径。上述指令序列可以在图 3.3.5 中顺利执行而无需暂停。

### 采用定向技术后的工作过程

- 指令1    ADD R1, R2, R3
- 指令2    SUB R4, R1, R5
- 指令3    AND R6, R1, R7
- 指令4    OR R8, R1, R9
- 指令5    XOR R10, R1, R11

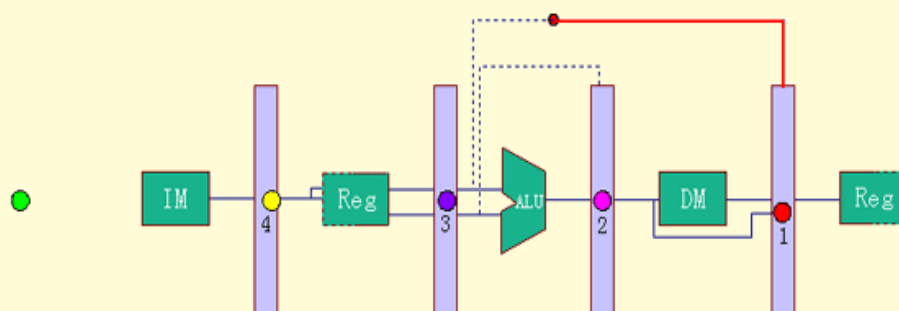


图 3. 3. 6

上述定向技术可以推广到更一般的情况，可以将一个结果直接传送到所有需要它的功能单元。也就是说，一个结果不仅可以从某一功能单元的输出生定向到其自身的输入，而且还可以从某一功能单元的输出生定向到其它功能单元的输入。举例：

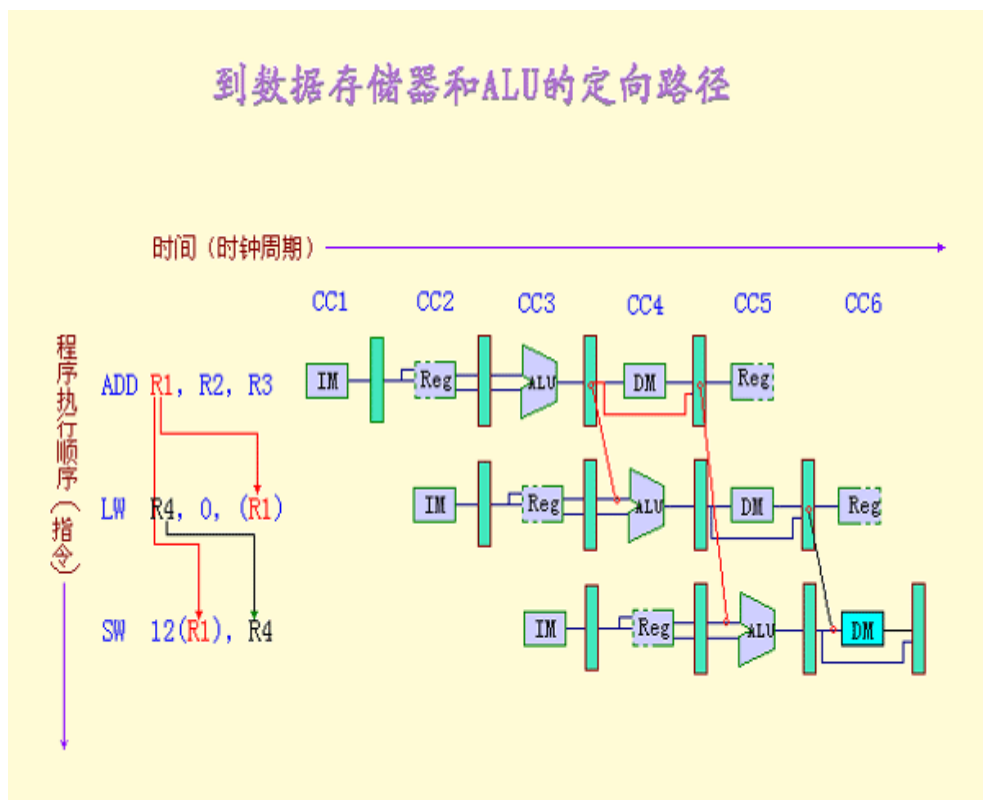


图 3.3.7 到数据存储器和 ALU 单元的定向路径

在 DLX 中，任何流水线寄存器到任何功能单元的输入都可能需要定向路径。

前面的一些数据相关的实例均是有关寄存器操作数的，但是数据相关也有可能发生在一对指令对存储器同一单元进行读写的时候。不过，本节仅讨论有关寄存器的数据相关。

#### 四. 实验注意事项及要求:

1. 计算机系统结构共七个实验，每个实验要求在两小时中完成。
2. 由于实验比较复杂，仅靠实验过程中的两小时完成比较困难，希望同学们实验前一定要按照实验指导书的要求，提前理解认真预习。
3. 同学们上机实验前应提前写出实验报告，带着实验报告参加实验，在实验过程中填写实验数据及结果，实验完成后将实验报告交给实验教师，实验教师根据每个同学的实验过程及实验报告情况评定成绩并记入考试总成绩中。
4. 实验报告的内容应包括：实验目的、实验内容、实验程序、程序流程图、实验步骤、实验中间结果、实验最终结果。
5. 由于特殊原因，未能参加实验者，经指导教师同意方可补做。无故不参加实验，以旷课论处。
6. 实验中要严格遵守实验纪律及实验室管理条例。

## 五. 实验一 熟悉 WinDLX 的使用

### 1.实验目的:

通过本实验，熟悉 WinDLX 模拟器的操作和使用，了解 DLX 指令集结构及其特点。

### 2.实验内容:

- (1) 用 WinDLX 模拟器执行求阶乘程序 facts 。执行步骤详见 “WinDLX 教程”。 这个程序说明浮点指令的使用。该程序从标准输入读入一个整数，求其阶乘，然后将结果输出。该程序中调用了 input.s 中的输入子程序，这个子程序用于读入正整数。
- (2) 输入数据 “3” 采用单步执行方法，完成程序并通过上述使用 WinDLX，总结 WinDLX 的特点。
- (3) 注意观察变量说明语句所建立的数据区，理解 WinDLX 指令系统。

### 3.实验程序:

求阶乘程序 fact.s

```
***** WINDLX Ex.2: Factorial *****
;-----
; Program begin at symbol main
; requires module INPUT
; read a number from stdin and calculate the factorial (type: double)
; the result is written to stdout
;-----

.data
Prompt: .asciiz "An integer value >1 : "

PrintfFormat: .asciiz "Factorial = %g\n\n"
.align 2
PrintfPar: .word PrintfFormat
PrintfValue: .space 8

.text
.global main
main:
;*** Read value from stdin into R1
addi r1,r0,Prompt
```



```

jal InputUnsigned
;*** init values
movi2fp f10,r1 ;R1 -> D0 D0..Count register
cvti2d f0,f10
addi r2,r0,1 ;1 -> D2 D2..result
movi2fp f11,r2
cvti2d f2,f11
movd f4,f2 ;1-> D4 D4..Constant 1
;*** Break loop if D0 = 1
Loop: led f0,f4 ;D0<=1 ?
bfpt Finish
;*** Multiplication and next loop
multd f2,f2,f0
subd f0,f0,f4
j Loop

Finish: ;*** write result to stdout
sd PrintfValue,f2
addi r14,r0,PrintfPar
trap 5
;*** end
trap 0

```

该程序中调用了 input.s 中的输入子程序

```

;***** WINDLX Ex.1: Read a positive integer number *****
;-----
;Subprogram call by symbol "InputUnsigned"
;expect the address of a zero-terminated prompt string in R1
;returns the read value in R1
;changes the contents of registers R1,R13,R14
;-----

.data

```

```
*** Data for Read-Trap
ReadBuffer: .space 80
ReadPar: .word 0,ReadBuffer,80
```

```
*** Data for Printf-Trap
PrintfPar: .space 4
```

```
SaveR2: .space 4
SaveR3: .space 4
SaveR4: .space 4
SaveR5: .space 4
```

```
.text
```

```
.global InputUnsigned
```

```
InputUnsigned:
```

```
*** save register contents
```

```
sw SaveR2,r2
```

```
sw SaveR3,r3
```

```
sw SaveR4,r4
```

```
sw SaveR5,r5
```

```
*** Prompt
```

```
sw PrintfPar,r1
```

```
addi r14,r0,PrintfPar
```

```
trap 5
```

```
*** call Trap-3 to read line
```

```
addi r14,r0,ReadPar
```

```
trap 3
```

```
*** determine value
```

```
addi r2,r0,ReadBuffer
```

```
addi r1,r0,0
```

```
addi r4,r0,10 ;Decimal system
```

```

Loop: ;*** reads digits to end of line
lbu r3,0(r2)
seqi r5,r3,10 ;LF -> Exit
bnez r5,Finish
subi r3,r3,48 ;??
multu r1,r1,r4 ;Shift decimal
add r1,r1,r3
addi r2,r2,1 ;increment pointer
j Loop
Finish: ;*** restore old register contents
lw r2,SaveR2
lw r3,SaveR3
lw r4,SaveR4
lw r5,SaveR5
jr r31 ; Retur

```

## 六. 实验二. 用 WinDLX 模拟器执行程序 求最大公约数

### 1. 实验目的:

通过本实验, 熟练掌握 WinDLX 模拟器的操作和使用, 清楚 WinDLX 五段流水线在执行具体程序时的流水情况, 熟悉 DLX 指令集结构及其特点。

### 2.实验内容:

(1) 用 WinDLX 模拟器执行程序 gcm.s 。

该程序从标准输入读入两个整数, 求他们的 greatest common measure, 然后将结果写到标准输出。 该程序中调用了 input.s 中的输入子程序。

(2).给出两组数 6、3 和 6、1, 分别在 main+0x8(add r2,r1,r0)、gcm.loop(seg r3,r1,r2)和 result+0xc(trap 0x0)设断点, 采用单步和连续混合执行的方法完成程序, 注意中间过程和寄存器的

变化情况，然后单击主菜单 execute/display dlx-i/0,观察结果。

### 3.实验程序

求最大公约数程序：**gcm.s**

```
***** WINDLX Ex.1: Greatest common measure *****
;-----
; Program begins at symbol main
; requires module INPUT
; Read two positive integer numbers from stdin, calculate the gcm
; and write the result to stdout
;-----

.data

;*** Prompts for input
Prompt1: .asciiz  "First Number:"
Prompt2: .asciiz  "Second Number:  "

;*** Data for printf-Trip
PrintfFormat: .asciiz  "gcM=%d\n\n"
.align 2
PrintfPar: .word PrintfFormat
PrintfValue: .space 4

.text

.global main
main:
;*** Read two positive integer numbers into R1 and R2
addi r1,r0,Prompt1
jal InputUnsigned ;read uns.-integer into R1
add r2,r1,r0 ;R2 <- R1
addi r1,r0,Prompt2
jal InputUnsigned ;read uns.-integer into R1

Loop: ;*** Compare R1 and R2
seq r3,r1,r2 ;R1 == R2 ?
```

```

bnez r3,Result
sgt r3,r1,r2 ;R1 > R2 ?
bnez r3,r1Greater
r2Greater: ;*** subtract r1 from r2
sub r2,r2,r1
j Loop

r1Greater: ;*** subtract r2 from r1
sub r1,r1,r2
j Loop

Result: ;*** Write the result (R1)
sw PrintfValue,r1
addi r14,r0,PrintfPar
trap 5

;*** end
trap 0

```

该程序中调用了 input.s 中的输入子程序。

```

;***** WINDLX Ex.1: Read a positive integer number *****
;-----
;Subprogram call by symbol  “InputUnsigned”
;expect the address of a zero-terminated prompt string in R1
;returns the read value in R1
;changes the contents of registers R1,R13,R14
;-----

.data

;*** Data for Read-Trap
ReadBuffer: .space 80
ReadPar: .word 0,ReadBuffer,80

```

```

;*** Data for Printf-Trap
PrintfPar: .space 4

SaveR2: .space 4
SaveR3: .space 4
SaveR4: .space 4
SaveR5: .space 4

.text

.global InputUnsigned
InputUnsigned:
;*** save register contents
sw SaveR2,r2
sw SaveR3,r3
sw SaveR4,r4
sw SaveR5,r5

;*** Prompt
sw PrintfPar,r1
addi r14,r0,PrintfPar
trap 5

;*** call Trap-3 to read line
addi r14,r0,ReadPar
trap 3

;*** determine value
addi r2,r0,ReadBuffer
addi r1,r0,0
addi r4,r0,10 ;Decimal system

Loop: ;*** reads digits to end of line
lbu r3,0(r2)
seqi r5,r3,10 ;LF -> Exit
bnez r5,Finish
subi r3,r3,48 ;??

```

```

multu r1,r1,r4 ;Shift decimal
add r1,r1,r3
addi r2,r2,1 ;increment pointer
j Loop
Finish: ;*** restore old register contents
lw r2,SaveR2
lw r3,SaveR3
lw r4,SaveR4
lw r5,SaveR5
jr r31 ; Retur

```

## 七. 实验三 用 WinDLX 模拟器完成求素数程序

### 1. 实验目的:

通过实验, 熟练掌握 WINDLX 的操作方法, 特别注意在单步执行 WinDLX 程序中, 流水线中指令的节拍数。

### 2. 实验内容:

- (1) 用 WinDLX 模拟器执行求素数程序 prim.s。这个程序计算若干个整数的素数。
- (2) 单步执行两轮程序, 求出素数 2 和 3。
- (3) 在执行程序过程中, 注意体验单步执行除法和乘法指令的节拍数, 并和主菜单 configuration/floating point slages 中的各指令执行拍数进行比较。

### 3. 实验程序

求素数程序 prim.s。

```

;***** WINDLX Exp.2: Generate prime number table *****
;-----
; Program begins at symbol main
; generates a table with the first 'Count' prime numbers from 'Table'
;-----

```

```

.data

;*** size of table
.global Count
Count: .word 10
.global Table
Table: .space Count*4

.text

.global main
main:
;*** Initialization
addi r1,r0,0 ;Index in Table
addi r2,r0,2 ;Current value

;*** Determine, if R2 can be divided by a value in table
NextValue: addi r3,r0,0 ;Helpindex in Table
Loop: seq r4,r1,r3 ;End of Table?
bnez r4,IsPrim ;R2 is a prime number
lw r5,Table(R3)
divu r6,r2,r5
multu r7,r6,r5
subu r8,r2,r7
beqz r8,IsNoPrim
addi r3,r3,4
j Loop

IsPrim: ;*** Write value into Table and increment index
sw Table(r1),r2
addi r1,r1,4

;*** 'Count' reached?
lw r9,Count
srli r10,r1,2

```



```

sge r11,r10,r9
bnez r11,Finish

IsNoPrim: ;*** Check next value
addi r2,r2,1 ;increment R2
j NextValue
Finish: ;*** end
trap 0

```

## 八. 实验四: 结构相关

### 1. 实验目的:

通过本实验,加深对结构相关的理解,了解结构相关对 CPU 性能的影响。

### 2.实验内容:

- (1). 用 WinDLX 模拟器运行程序 structure\_d.s 。
- (2). 通过模拟,找出存在结构相关的指令对以及导致结构相关的部件。
- (3). 记录由结构相关引起的暂停时钟周期数,计算暂停时钟周期数占总执行周期数的百分比。
- (4). 论述结构相关对 CPU 性能的影响,讨论解决结构相关的方法。

### 3.实验程序:

```

程序 structure_d.s 。
LHI R2, (A>>16)&0xFFFF
ADDUI R2, R2, A&0xFFFF
LHI R3, (B>>16)&0xFFFF
ADDUI R3, R3, B&0xFFFF
ADDU R4, R0, R3
loop:
LD F0, 0(R2)
LD F4, 0(R3)
ADDD F0, F0, F4
ADDD F2, F0, F2 ; <- A stall is found (an example of how to answer
your questions)
ADDI R2, R2, #8
ADDI R3, R3, #8
SUB R5, R4, R2
BNEZ R5, loop
TRAP #0 ;; Exit <- this is a comment !!
A: .double 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
B: .double 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

```

## 九. 实验五: 数据相关

### 1. 实验目的:

通过本实验,加深对数据相关的理解,掌握如何使用定向技术来减少数据相关带来的暂停。

### 2. 实验内容:

- (1) 在不采用定向技术的情况下(通过 *Configuration* 菜单中的 *Enable Forwarding* 选项设置),用 WinDLX 模拟器运行程序 `data_d.s`。
- (2) 记录数据相关引起的暂停时钟周期数以及程序执行的总时钟周期数,计算暂停时钟周期数占总执行周期数的百分比。
- (3) 在采用定向技术的情况下,用 WinDLX 模拟器再次运行程序 `data_d.s`。
- (4) 记录数据相关引起的暂停时钟周期数以及程序执行的总时钟周期数,计算暂停时钟周期数占总执行周期数的百分比。
- (5) 根据上面记录的数据,计算采用定向技术后性能提高的倍数。

### 3.实验程序:

程序 `data_d.s`。

```
LHI R2, (A>>16) & 0xFFFF
ADDUI R2, R2, A & 0xFFFF
LHI R3, (B>>16)&0xFFFF
ADDUI R3, R3, B&0xFFFF
loop:
LW R1, 0(R2)
ADD R1, R1, R3
SW 0(R2), R1
LW R5, 0(R1)
ADDI R5, R5, #10
ADDI R2, R2, #4
SUB R4, R3, R2
BNEZ R4, loop
TRAP #0
A: .word 0, 4, 8, 12, 16, 20, 24, 28, 32, 36
B: .word 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
```

## 十. 实验六 指令调度

### 1. 实验目的:

通过本实验,加深对指令调度的理解,了解指令调度技术对 CPU 性能改进的好处。

### 2. 实验内容:

- (1) 通过 *Configuration* 菜单中的“*Floating point stages*”选项,把除法单元数设置为 3,把加法、乘法、除法的延迟设置为 3 个时钟周期。
- (2) 用 WinDLX 模拟器运行调度前的程序 `sch-before.s`。记录程序执行

过程中各种相关发生的次数以及程序执行的总时钟周期数。

- (3) 用 WinDLX 模拟器运行调度后的程序 sch-after.s，记录程序执行过程中各种相关发生的次数以及程序执行的总时钟周期数。
- (4) 根据记录结果，比较调度前和调度后的性能。
- (5) 论述指令调度对于提高 CPU 性能的意义。

### 3. 实验程序：

程序 sch-before.s

```
;-----  
; Example to illustrate instruction scheduling  
;-----  
.data  
.global ONE  
ONE: .word 1  
.text  
.global main  
main:  
lf f1,ONE ;turn divf into a move  
cvti2f f7,f1 ;by storing in f7 1 in  
nop ;floating-point format  
divf f1,f8,f7 ;move Y=(f8) into f1  
divf f2,f9,f7 ;move Z=(f9) into f2  
addf f3,f1,f2  
divf f10,f3,f7 ;move f3 into X=(f10)  
divf f4,f11,f7 ;move B=(f11) into f4  
divf f5,f12,f7 ;move C=(f12) into f5  
multf f6,f4,f5  
divf f13,f6,f7 ;move f6 into A=(f13)  
Finish:  
trap 0  
调度后的程序 sch-after.s
```

```
;-----  
; Example to illustrate instruction scheduling - reordered instructions  
;-----  
.data  
.global ONE  
ONE: .word 1  
.text  
.global main  
main:
```

```

lf f1,ONE ;turn divf into a move
cvti2f f7,f1 ;by storing in f7 1 in
nop ;floating-point format
divf f1,f8,f7 ;move Y=(f8) into f1
divf f2,f9,f7 ;move Z=(f9) into f2
divf f4,f11,f7 ;move B=(f11) into f4
divf f5,f12,f7 ;move C=(f12) into f5
addf f3,f1,f2
multf f6,f4,f5
divf f10,f3,f7 ;move f3 into X=(f10)
divf f13,f6,f7 ;move f6 into A=(f13)
Finish:
trap 0

```

## 十一. 实验七: 多处理机并行计算

### 1. 实验目的:

通过完成“ $\pi$  的并行计算”了解并行算法的原理及执行过程, 观察比较串行算法与并行算法在时间与效率方面的差异, 加深对并行计算的理解。

### 2. 实验原理:

$\pi$  的计算可通过积分

$$\int_0^1 \frac{4.0}{1.0 + x^2} dx$$

得到。由微积分知识知此积分可通过求极限

$\lim_{n \rightarrow \infty} (1/n) * \sum_{i=1}^{n+1} [4.0 / (1.0 + ((1/n) * (i-0.5))^2)]$  得到,  $n$  取不同值则得到的  $\pi$  具有不同的精度, 本程序中  $n$  内定为 20000000。

并行程序中每个结点计算 **for** 循环中的一部分。

串行程序核心如下：

```
h = 1.0/(double) n;

sum = 0.0;

for (i = 1; i <= n; i ++)

{    x = h*(i-0.5);

        sum = sum + f(x);

}

pi = h*sum;
```

并行程序核心如下：

```
h = 1.0/(double) n;

sum = 0.0;

for (i = my_rank+1; i <= n; i+=group_size)

{

        x = h*(i-0.5);

        sum = sum + f(x);

}

pi = h*sum;
```

将每个进程计算得到的 **pi** 相加求得  $\pi$ 。

其中  $f(x)=4.0/(1.0+(x)^2)$

### 3. 实验程序

并行算法源程序：

```
#include <stdio.h>

#include <mpi.h>

#include "math.h"

long        n,    /*    number of slices        */

            i;    /* slice counter        */

double      sum,  /* running sum          */

            pi,   /* approximate value of pi */

            mypi,

            x,    /* independent var.      */

            h;    /* base of slice         */
```

```

int group_size, my_rank;

main(argc, argv)

int argc;

char* argv[];

{

    int group_size, my_rank;

    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank( MPI_COMM_WORLD, &my_rank);

    MPI_Comm_size( MPI_COMM_WORLD, &group_size);


    n=20000000;

        /* Broadcast n to all other nodes */

    MPI_Bcast(&n, 1, MPI_LONG, 0, MPI_COMM_WORLD);

    /*计算 Pi*/

    h = 1.0/(double) n;

    sum = 0.0;

    for (i = my_rank+1; i <= n; i += group_size)

    {

        x = h*(i-0.5);

        sum = sum +4.0/(1.0+x*x);

    }

    mypi = h*sum;

```

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
/* Node 0 handles output */
```

```
if(my_rank==0)
```

```
{
```

```
printf("pi is approximately : %.16lf\n", pi);
```

```
}
```

```
MPI_Finalize();}
```

#### 4. 实验过程:

(1) 安装远程登陆软件支持 SSH 协议的 SecureCRT 等

(2) 登陆: 登陆 IP 211.87.224.140

登陆帐户: yxf

密码: pppppppp

(3) 登陆后在界面上键入以下命令:

看文件列表: ls 或 ls -l (相当于 DOS 命令 dir)

看源程序: vi pi.c (串行程序)

或 vi mpipi.c (并行程序)

退出: :q

编译: 禁止

运行:

串行算法运行:

**./pi**

并行算法运行

mpirun\_rsh -np 1 cu0701-ib ./mpipi )

(在一台服务器上用一个进程运行并行程序, 相当于串行执行)

mpirun\_rsh -np 2 cu0701-ib cu0701-ib ./mpipi

(在一台服务器上两个进程运行并行程序)

mpirun\_rsh -np 4 cu0701-ib cu0701-ib cu0703-ib cu0703-ib ./mpipi

(在两台服务器上四个进程运行并行程序)

mpirun\_rsh -np 4 cu0701-ib cu0701-ib cu0701-ib cu0701-ib ./mpipi

(在一台服务器上四个进程运行并行程序)

记录每个程序运行的时间, 比较并行算法与串行算法执行的时间, 以及各种情况下并行算法执行的效率。