[指令系统 2](#_Toc507244868)

[数据表示 2](#_Toc507244869)

[数据寻址 2](#_Toc507244870)

[指令格式 3](#_Toc507244871)

[数据通路 5](#_Toc507244872)

[寄存器 5](#_Toc507244873)

[指令流水线 6](#_Toc507244874)

[运算器 15](#_Toc507244875)

[浮点运算 15](#_Toc507244876)

[存储系统 16](#_Toc507244877)

[Cache 17](#_Toc507244878)

[输入输出系统 18](#_Toc507244879)

[中断管理 19](#_Toc507244880)

[异常 19](#_Toc507244881)

[中断 19](#_Toc507244882)

[初始化 19](#_Toc507244883)

# 指令系统

## 数据表示

整数：字节8bit、半字16bit、字32bit、双子64bit

浮点数：单精度32bit，双精度64bit

设置半字，是因为在类似于C的高级语言中有这种数据类型。而且在操作系统等程序中也很常用，这些程序很重视数据所占的空间大小设置单精度浮点操作书也是基于类似的原因

MIPS的操作就是针对64bit整数以及32bit或64bit浮点数进行的

字节、半字或字装入64位寄存器时，用零扩展或者符号位扩展来填充剩余部分

装入以后都按照64位整数的方式计算

## 数据寻址

|  |  |
| --- | --- |
| 立即数寻址 | 16bit |
| 偏移量寻址 | 16bit |
| 16bit绝对寻址 | 把R0作为基址寄存器，因为R0的值永远是0 |
| 寄存器间接寻址 | 把R0的值作为偏移量 |

## 指令格式

所有指令都是32位的

操作码占6位

|  |  |
| --- | --- |
| I | 0~5：操作码  6~10：rs  11~15：rt  16~31：立即数  load：访存有效地址为[rs]+immediate，从存储器取来的数据放入寄存器rt  store：访存有效地址为[rs]+immediate，要存入存储器的数据放在寄存器rt  立即数指令：[rs] op immediate →[rt]  分支指令：转移目标地址为PC+immediate，rt无用  存器跳转指令、寄存器链接跳转指令：转移目标地址为[rs] |
| R | 0~5：操作码  6~10：rs  11~15：rt  16~20：rd  21~25：shamt  26~31：funct  ALU指令：[rs] funct [rt]→[rd]  funct是具体的运算操作编码 |
| J | 0~5：操作码  6~31：与PC相加的偏移量 |

|  |  |  |  |
| --- | --- | --- | --- |
| load | LD R2,20(R3) | 装入双字 | (20+[R3])→R2 |
| LW R2,40(R3) | 装入字 |  |
| LB R2,30(R3) | 装入字节 |  |
| LBU R2,40(R3) | 装入无符号字节 |  |
| LH R2,30(R3) | 装入半字 |  |
| L.S F2,60(R4) | 装入单精度浮点数 |  |
| L.D F2,40(R3) | 装入双精度浮点数 |  |
| store | SD R4,300(R5) | 保存双字 | [R4]→(300+[R5]) |
| SW R4,300(R5) | 保存字 |  |
| S.S F2,40(R2) | 保存单精度浮点数 |  |
| SH R5,502(R4) | 保存半字 |  |
| ALU | DADDU R1,R2,R3 | 无符号加 | [R2]+[R3]→R1 |
| DADDIU R4,R5,#6 | 加无符号立即数 | [R5]+6→R4 |
| LUI R1,#4 | 把立即数装入到一个字的高16位 |  |
| DSLL R1,R2,#5 | 逻辑左移 |  |
| DSLT R1,R2,R3 | 置小于 |  |
| 控制  指令 | J name | 跳转 |  |
| JAL name | 跳转并链接 |  |
| JALR R3 | 寄存器跳转并链接 |  |
| JR R5 | 寄存器跳转 |  |
| BEQZ R4,name | 等于零时分支 |  |
| BNE R3,R4,name | 不相等时分支 |  |
| MOVZ R1,R2,R3 | 等于零时移动 |  |

浮点指令对浮点寄存器中的数据进行操作，并由操作码指出操作数是SP还是DP，

在指令助记符中，用后缀S表示操作数是SP，D表示DP

如

MOV.S：把一个单精度浮点寄存器中值复制到另一个同类型的寄存器中

MOV.D：把一个双精度浮点寄存器中值复制到另一个同类型的寄存器中

# 数据通路

## 寄存器

MIPS有32个64位的通用寄存器GPR：R0~R31

R0的值永远是0

32个浮点数寄存器FPR：F0~F31

既可以用来存放32个单精度浮点数(32bit)，这时只用到FPR的一般，另一半没有用

也可以用来存放32个双精度浮点数(64bit)

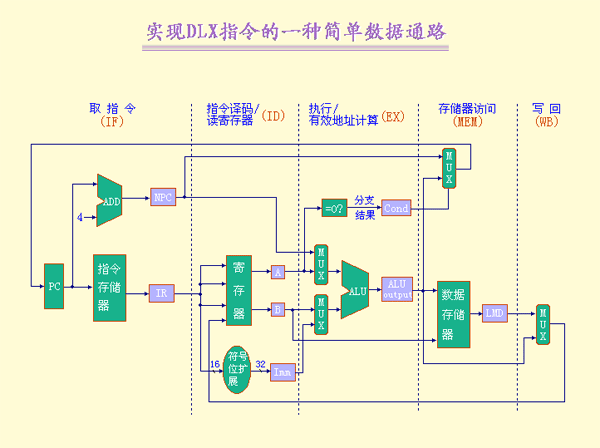
MIPS提供了单精度和双精度操作的指令，而且还提供了在FPR和GPR之间传送数据的指令

另外还有一些特殊寄存器，它们可以与通用寄存器交换数据。

如浮点状态寄存器用来保存有关浮点操作结果的信息

## 指令流水线

CPI=5



临时寄存器：

NPC：存放以下条指令的地址

IR：存放当前执行的指令字

A：第一操作数寄存器，存放从GPR读出的操作数

B：第二操作数寄存器，存放从GPR读出的另一个操作数

Imm：存放符号扩展后的立即数操作数

cond：存放条件判定结果，

ALUo：存放ALU的运算结果

LMD：存放load指令从存储器读出的数据

不同的操作周期之间设置有存储单元，用于保存当前指令的执行结果或者在上一个周期产生且在后面周期要用到的结果。其中，PC、GPR、存储器单元是在指令之间保存结果，而临时寄存器IE、NPC、A、B、Imm、cond、ALUo、LMD则是在单条指令的执行过程中保存中间结果

一条指令的执行过程分为以下5个周期：

|  |  |  |
| --- | --- | --- |
| 取指  IF | 根据PC指示的地址从存储器中取出指令，并放入指令寄存器，同时PC+4(假设每条指令占4B)  IR←Mem[PC]  NPC←PC+4 | |
| 译码  读寄存器  ID | 对指令进行译码，以指令中的rs和rt字段作为地址访问GPR，把读出的操作数放入A和B，同时IR的低16位进行符号位扩展，然后存入Imm  A ← Regs[rs]  B ← Regs[rt]  Imm ← （（IR16）16##IR16..31）  固定字段译码技术：在MIPS指令格式中，操作码字段以及rs、rt字段都是在固定的位置，因此指令的译码操作和读寄存器操作并行进行  这里读出的操作数在后面不一定会用到，但是这样统一处理可以简化硬件  另外，由于立即数在所有MIPS指令中的位置是相同的，因此在这里同意对其进行符号扩展，一边在一下一个周期使用，当然也许有的指令不会用到和这个立即数，但无论如何，提前形成总是没有坏处 | |
| 执行  有效地址计算  EX | ALU对在一码周期准备好的操作数进行运算或处理， | |
| 访存指令 | ALU把所指定的寄存器的内容与偏移量相加，形成用于访存的有效地址，存入临时寄存器  ALUo←A + Imm |
| R-R | ALU按照操作码指定的操作，对从通用寄存器组中读取的数据进行运算  ALUo←A func B |
| R-immediate | ALU按照操作码指定的操作，对从通用寄存器组中读取的第一操作数和立即数进行运算  ALUo←A op Imm |
| 分支指令 | ALU把偏移量与PC值相加，形成转移目标的地址，同时，对在前一个周期读出的操作数进行判断，确定分支是否成功  ALUo←NPC+（Imm<<2）  cond←（A = = 0）  ALU将NPC和Imm中的值(左移两位)相加，得到转移目标的地址存入ALUo，同时，对在前一个周期读入到寄存器A的值进行判断，决定分支是否成功，为简单起见，这里只考虑一种分支，即BEQZ，判断的结果存入寄存器cond，供以后使用。  这里将有效地址计算周期和执行周期合并为一个时钟周期，这时因为MIPS指令集采用load/store结构，没有任何指令需要同时进行数据有效地址计算、转移目标地址计算、数据运算。 |
| 存储器访问  分支完成  MEM | load | 用执行周期计算出的有效地址从存储器找那个读出相应的数据  LMD←Mem[ALUo] |
| store | 把指定数据写入执行周期计算出的有效地址指定的存储单元  Mem[ALUo]←B |
| 分支 | 如果执行周期判定分支成功，就把转移目标地址送入PC  ALUo←NPC+（Imm<<2）  cond←（A = = 0） |
| 其他类型的指令再次皱起不做任何操作，等待 | |
| 高性能方案中，可以吧分支指令的执行放到ID段完成，但需要增加用于计算转移目标地址的加法器  将有效地址计算周期和执行周期合并为一个时钟周期，这是因为MIPS指令集采用load／store结构，没有任何指令需要同时进行数据有效地址的计算、转移目标地址的计算和对数据进行运算 | |
| 写回  WB | ALU | 把结果数据写入GRU，这个结果数据来自ALU  Regs[rd]← ALUo |
| R-immediate | Regs[rt]← ALUo |
| load | 把结果数据写入GRU，这个结果数据来自mem  Regs[rt]← LMD |
| 上述指令都是讲结果写入GPR，这个结果可能是ALU的计算结果(ALUo)，也可能是来自存储器(LMD)，写入目标寄存器由指令中的rd或rt字段指出，具体是哪一个由操作码决定 | |

在上述实现方案中，分支指令需要4个时钟周期，如果把分支指令的执行提前到ID，则需要2个时钟周期

store指令需要4个时候周期

其他指令需要5个时钟周期

采用流水线时，

1.保证不会在同一时钟周期要求同一个功能段做两件不同的工作。

如不能要求ALU同时做有效地址计算和算术运算。

2.为了避免IF段的访问与MEM段的访存发生冲突，使用分离的指令存储器和数据存储器

3.ID段读通用寄存器组，WB写通用寄存器组，因此将写操作安排在时钟周期的前半拍，读操作在后半拍

4.设置专门的加法器，修改PC，这时在IF完成的

构造流水线



1.[流水寄存器](3-2-18%20流水线数据通路.exe)

段与段之间设置流水寄存器：IF/ID、ID/EX、EX/MEM、MEM/WB

[每个流水寄存器是由若干个寄存器构成的](3-2-21%20流水线寄存器的构成.exe)

寄存器的命名形式为：x.y

所包含的字段的命名形式为：x.y[s]

ID/EX.IR：流水寄存器ID/EX中的子寄存器IR

ID/EX.IR[op]：流水寄存器的ID/EX中的子寄存器IR中的op字段

作用：

(1)将各段的工作隔开，使得它们不会互相干扰。

(2)保存相应段的处理结果例如：

EX/MEM.ALUo：保存EX段ALU的运算结果

MEM/WB.LMD：保存MEM段从数据存储器读出的数据

(3)向后传递后面将要用到的数据或者控制信息，所有有用的数据和控制信息每个时钟周期会随着指令在流水线中的流动往后流动一段。

2.增加了向后传递IR和从MEM/WB.IR回送到通用寄存器组的连接

当一条指令从ID流到EX时，新的指令会进入ID段，冲掉IF/ID中的内容

所以指令中的有用信息必须跟着指令流到到ID/EX.IR，一次类推，后面需要用到的指令信息要一次向后传递，知道MEM/WB.IR，其中目标寄存器地址回送到GPR，用于实现结果回写到GPR

3.将对PC的修改移到了IF段，以便PC能及时地加4，为取下一条指令做好准备

为了详细了解流水线工作情况，需要知道各种指令在每一个流水段进行什么样的操作，

各段包含的指令

|  |  |  |
| --- | --- | --- |
| IF | IF/ID.IR ← Mem[PC]  IF/ID.NPC, PC ← （if（（ EX/MEM.IR[op] == branch ）& EX/MEM.cond）{EX/MEM.ALUo} else {PC+4}）； | |
| ID | ID/EX.A ← Regs[IF/ID.IR[rs]]；ID/EX.B ← Regs[IF/ID.IR[rt]]；  ID/EX.NPC ← IF/ID.NPC；ID/EX.IR ←IF/ID.IR；  ID/EX.Imm ← （IF/ID.IR16)16##IF/ID.IR16..31； | |
| EX | ALU | EX/MEM.IR ← ID/EX.IR；  EX/MEM.ALUo ←ID/EX.A func ID/EX.B  或  EX/MEM.ALUo ←ID/EX.A op ID/EX.Imm； |
| load/store | EX/MEM.IR ← ID/EX.IR；  EX/MEM.ALUo ←ID/EX.A + ID/EX.Imm；  EX/MEM.B←ID/EX.B； |
| branch | EX/MEM.IR ← ID/EX.IR；  EX/MEM.ALUo ←ID/EX.NPC+ID/EX.Imm<<2；  EX/MEM.cond ← (ID/EX.A ==0）； |
| MEM | ALU | MEM/WB.IR ←EX/MEM.IR；  MEM/WB.ALUo ←EX/MEM.ALUo； |
| load/store | MEM/WB.IR ← EX/MEM.IR；  MEM/WB.LMD ←Mem[EX/MEM.ALUo]；  或  Mem[EX/MEM.ALUo] ←EX/MEM.B； |
| WB | ALU | Regs[MEM/WB.IR[rd]] ←MEM/WB.ALUo；  或  Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUo； |
| load/store | Regs[MEM/WB.IR[rt]] ←MEM/WB.LMD； |

流水线的控制主要是如何[控制4个多路选择器](3-2-31%20对MUX的控制.exe)。

MUX2：若ID/EX.IR中的指令是分支指令，则选择ID/EX.NPC，否则选ID/EX.A。

MUX3：若ID/EX.IR中的指令是寄存器－寄存器型ALU指令，则选ID/EX.B，否则选ID/EX.Imm。

MUX1：若EX/MEM.IR中的指令是分支指令，而且EX/MEM.cond为真，则选EX/MEM.ALUo，即分支目标地址，否则选PC+4。

MUX4：若MEM/WB.IR中的指令是load指令，则选MEM/WB.LMD，否则选MEM/WB.ALUo。

第5个多路器：从MEM/WB回传至通用寄存器组的写入地址应该是从MEM/WB.IR[rd] 和MEM/WB.IR[rt]中选一个。

寄存器－寄存器型ALU指令：选择MEM/WB.IR[rd] ；

寄存器－立即数型ALU指令和load指令：选择MEM/WB.IR[rt] 。

数据冲突：

所有的数据冲突均可以在ID段检测到，如果存在数据冲突，就在相应的指令流出ID段之前将之暂停。

完成该工作的硬件称为流水线的互锁机制。

在ID段确定需要什么样的定向，并设置相应的控制。

降低流水线的硬件复杂度。不必挂起已经改变了机器状态的指令）

也可以在使用操作数的那个时钟周期的开始检测冲突和确定必需的定向。

检测冲突是通过比较寄存器地址是否相等来实现的。

举例： load互锁

由于使用load的结果而引起的RAW冲突称为load互锁。

[在ID段检测是否存在RAW冲突](3-3-17在ID段检测Load互锁需进行三种比较.exe)（这时load指令在EX段）

|  |  |  |
| --- | --- | --- |
| ID/EX中的操作码  （ID/EX.IR[op]） | IF/ID中的操作码  （IF/ID.IR[op]） | 比较的操作数字段 |
| load | RR ALU | ID/EX.IR[rt]=IF/ID.IR[rs] |
| load | RR ALU | ID/EX.IR[rt]=IF/ID.IR[rt] |
| load | load、store  ALU立即数或分支 | ID/EX.IR[rt]=IF/ID.IR[rs] |

若检测到RAW冲突，流水线互锁机制必须在流水线中插入停顿，并使当前正处于IF段和ID段的指令不再前进。为实现这一点，将ID/EX.IR中的操作码改为全0(全0表示空操作)IF/ID寄存器的内容回送到自己的入口

定向逻辑要考虑的情况更多，通过比较流水寄存器中的寄存器地址来确定

例如：

若(ID/EX.IR.op==RR ALU)&(EX/MEM.IR.op==RR ALU）&(ID/EX.IR[rt]==EX/MEM.IR[rd])，

即EX段和MEM段中的指令都是RR ALU，而且MEM段中指令的目标寄存器地址与EX断种指令的第二源操作数地址相同，则需要把EX/MEM.ALUo定向到ALU的下面一个输入。

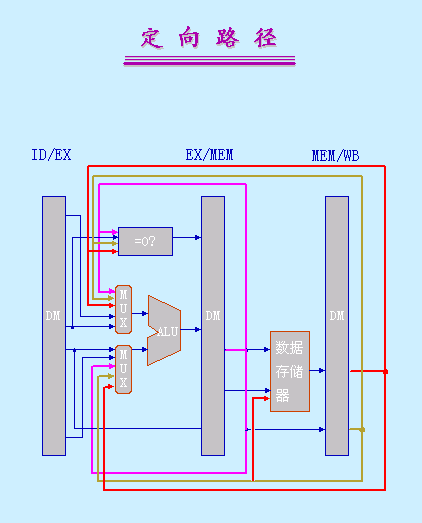
若(ID/EX.IR[op]==RR ALU)&(MEM/WB.IR[op]==load)&(ID/EX.IR[rt]==MEM/WB.IR[rt])

即EX段中的指令是RR ALU，MEM段中的指令是load，而且该load指令的目标寄存器地址与EX断种指令的第二源操作数地址相同，则把MEM/WB.LMD定向到ALU的下面一个输入

类似的可以列出所有的情况。

为了实现定向而增加的连通路：

它们粉笔吧ALU的运算结果和访存得到的数据定向到ALU的两个输入端



分支指令的条件测试和分支目标地址计算在EX段完成，对PC的修改[在MEM段完成](3-3-50转移成功时PC值得改变.exe)。它所带来的分支延迟是3个时钟周期。

需要减少分支延迟，尽早完成这些工作：

如果只考虑BEQZ和BNEZ，就可以把上述工作提前到ID段进行

为此需要在ID段增设一个加法器，用于计算分支目标地址。并把条件测试“=0？”的逻辑电路移到ID段。

这些结果直接回送到IF段的MUX1，如图所示

该影后流水线的分支延迟是1个时钟周期它对分支指令的处理如下表所示



|  |  |
| --- | --- |
| IF | IF/ID.IR ← Mem[PC]；  **IF/ID.NPC, PC ←**  **if(( IF/ID.IR[op]= =branch)&((Regs[IF/ID.IR[rs]] = = 0)**  **IF/ID.NPC + (IF/ID.IR16)16 ## （IF/ID.IR16..31 << 2）**  **else**  **PC+4；** |
|  | ID/EX.A ←Regs[IF/ID.IR[rs]]； ID/EX.B← Regs[IF/ID.IR[rt]]；  ID/EX.IR ← IF/ID.IR；  ID/EX.Imm ← （ IF/ID.IR16 ）16 ## IF/ID. IR16..31； |
|  |  |

# 运算器

## 浮点运算

# 存储系统

## Cache

# 输入输出系统

# 中断管理

## 异常

## 中断

## 初始化