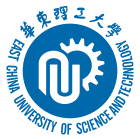


组成原理

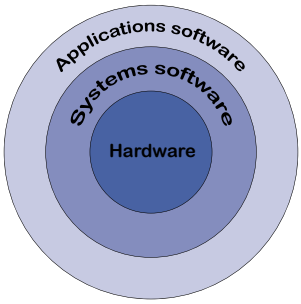
East China University of Science and Technology

目录

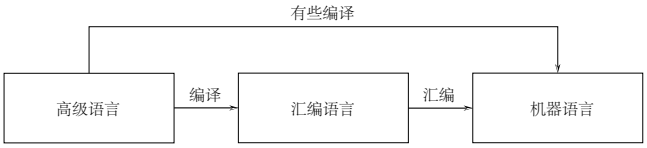
Chapter 1

概要

1.1 程序



(a) 简化的硬软件层次结构



(b) 从高级语言到机器语言

- 应用软件

  用高级语言编写

- 系统软件

  - 操作系统

    处理I/O, 分配内存, 为应用程序提供服务

  - 编译程序

- 硬件

  处理器, 主存, 输入输出系统等

1.2 硬件

计算机的核心硬件是CPU(控制器和ALU), 存储器 (主存和外存), I/O设备. 其中, 较受到关注的I/O设备有:

- 鼠标
  - 电动机械式: 一个球滚动
  - 光电式: LED光源, 每秒1500次采样, 处理器进行照片对比
- 显示器

  图像的像素矩阵用位图表示, 分辨率是x, y轴上点的个数. 彩色显示器每种颜色可以用8位二进制数表示, 每像素RGB三色, 故每像素用24位表示. 图像保存在帧缓存中, 一幅图像为一帧
  - 液晶显示器

    每个像素由一个三极管控制光线是否通过, 三个三极管控制颜色分配, 共四个三极管. 不施加电压是透光, 施加电压是不透光

1.3 性能

- 机器字长

  计算机进行一次整数运算能处理的二进制位数, 通常和CPU的寄存器位数/加法器有关.
- 数据通路带宽

  数据总线 (非CPU内部总线) 一次能传递信息的位数.
- 主存容量
  - MAR: 主存地址, 反映存储单元的个数
  - MDR: 主存数据, 反映存储单元的位数
- 运算速度
  - 响应时间: 计算机完成某任务需要的总的时间
    - CPU执行时间
      - 用户CPU时间: 程序本身花费的时间, 体现CPU性能 (重点研究)
      - 系统CPU时间: 操作系统的时间, 体现系统性能
    - 等待I/O等或多任务时其他程序运行的时间
  - 吞吐量

    单位时间那完成的任务量
  - 时钟周期

    常数. 又称节拍. 时钟频率 =1/时钟周期
  - Average CPI(clock cycles per instruction)

    一个程序或者程序片段的全部指令所有时钟周期数的平均值.

看了上面一些性能衡量的指标, 所以**到底**我们应该怎么衡量性能呢?–下面的公式:

Average CPI = CPU时钟周期数 / 指令数
一个程序的CPU执行时间 = 指令数 * Average CPI * 时钟周期
性能 = 1 / 一个程序的CPU执行时间

1.4 功耗

CMOS集成电路的动态功耗计算:

动态功耗 = 负载电容 * 电压^2 * 开关频率

从上面的公式看出, 我们可以通过下降电压来降低功耗, 但是降低电压会导致 “漏电” 现象. 同时, 电压过高会导致散热问题难以解决. 所以, 我们采用了一种方法: 多核处理器.

1.5 性能测试方法

- SPEC CPU基准测试程序

  SPEC(System Performance Evaluation Coorperative)为CPU, I/O, Web,...等开发了若干用于测试性能的程序, 我们只要选择一种处理器作为基准处理器, 按照下面的公式即可测试性能:

  SPECratio = 参考处理器执行时间 / 被测计算机执行时间
  性能测试结果 = SPECratio的几何平均值

- 功耗基准测试程序

  性能用吞吐率ssj_ops表示, 即每秒钟的操作次数. 现在, 我们将负载设为0, 同时记录ssj_ops, 将负载提高10%, 记录ssj_ops, 再将负载提高10%, 再记录ssj_ops,... 如此循环往复, 直到满负载. 得到的结果用以下公式计算:

$$\text{功耗测试结果} = \frac{\sum_{i=0}^{10} \text{ssj\_ops}_i}{\sum_{i=1}^{10} \text{power}_i}$$

1.6 Amdahl 定律

首先, 要解释以下几个变量的含义:

- T_0: 改进前的总执行时间
- T_improved: 改进后的总执行时间
- T_affected: 受改进影响的执行时间
- T_unaffected: 未受改进影响的执行时间

根据上面的几个变量, 衍生出了下面的几个变量:

- 可改进比`Fe`

  `Fe = T_affected / T_0`

- 加速比`Sn`

  `T_0 / T_improved`

- 部件加速比`Se`(改进量)

  `T_affected / (T_improved - T_unaffected)`

下面就是著名的`Amdahl`定律:

`T_improved = T_affected / Se + T_unaffected`

当然, 这个公式还有其他改进:

$$\frac{1}{\sum_{i=1}^{m} \frac{Fe_i}{Se_i} + (1 - \sum_{i=1}^{m} Fe_i)}, \text{其中 } i \text{ 表示改进的部件的数量}$$

Chapter 2

指令

2.1 引言

指令集 一个给定的计算机体系结构所包含的指令集合

汇编语言和机器语言 前者是编程的书写形式, 后者是计算机所能识别的形式

存储程序 多种类型的指令和数据均以数字形式存储在存储器 (内存) 中

MIPS 是一种汇编语言, 属于精简指令集

2.2 硬件操作

2.2.1 MIPS 汇编指令

- 每条 MIPS 算数运算指令只执行一个操作
- 一行写一条命令
- # 后是注释

2.2.2 高级语言与编译语言之间的关系

高级语言经过编译器编译, 形成汇编语言.

2.3 MIPS 寄存器及常用指令

高级语言的变量数量不受限制, 而汇编语言逻辑运算指令的变量对应寄存器, 而寄存器数量有限, 故变量数量受限.

2.3.1 寄存器

- 共有32个寄存器, 编号为0-31
- 32bit数据称为一个 “字”, 32位为字长, 每个字4字节
- 按字节编址
- 寄存器分类型:
  - $ZERO: 恒为0
  - $v0-$v1: 返回值
  - $a0-$a3: 参数
  - $t0-$t9: 临时变量, 其中$t0-$t7对应编号8-15, $t8-$t9对应编号24-25, 无需压栈
  - $s0-$s7: 保留变量, 对应编号16-23, 必须压栈
  - $gp: 静态数据的全局指针
  - $sp: 栈指针
  - $fp: 帧指针
  - $ra: 返回地址

2.3.2 常见指令

复杂的数据结构 (数组等) 存储于存储器中, 需要用数据传输指令交换数据:

- `lw rt, shamt(rs)`: 取数
- `sw rs, shamt(rt)`: 存数

数据被存储到寄存器后, 可以进行相加减:

- `add rd, rs, rt`: 加法
- `sub rd, rs, rt`: 减法

我们经常要在加减运算的时候用到常数, 这样就会导致计算机会去内存中取出这个常数存储到寄存器这一多余的步骤, 可以通过立即数以除去这一过程:

- `addi rt, rs, constant`: 加立即数
- `-addi rt, rs, constant`: 没有减立即数, 用这个替代

特殊的, 如果我们要进行寄存器间的赋值, 可以通过add或者addi实现:

- `add rd, rs, $ZERO`: 将rs赋值给rd
- `addi rt, rs, 0`: 将rs赋值给rt

2.4 MIPS 指令格式

指令包含操作码和地址码.

2.4.1 R 型指令

op	rs	rt	rd	shamt	funct

他们的作用:

- op: 操作码
- rs: 第一个源寄存器号
- rt: 第二个源寄存器号
- rd: 目标寄存器号
- shamt: 位移量
- funct: 功能码 (与op一起起作用)

他们的位数:

- op: 6位, 因为有64种指令
- rs,rt,rd: 5位, 因为有32个寄存器
- shamt: 5位, 因为 MIPS 是32位指令
- funct: 6位, 因为32-6-5-5-5-5=6

上述操作码可以查询表格, 寄存器号需要记忆, 下面也一样.

2.4.2 I 型指令

op	rs	rt	constant or address

他们的作用:

- op: 操作码
- rs: 源寄存器号
- rt: 目标寄存器号
- constant or address: 偏移量

他们的位数:

- op: 6位, 因为有64种指令
- rs,rt: 5位, 因为有32个寄存器
- constant or address: 16位, 因为32-6-5-5=16

2.4.3 J 型指令

op	address

他们的作用:

- op: 操作码
- address: 地址

他们的位数:

- op: 6位, 因为有64种指令
- address: 26位, 因为32-6=26

2.5 MIPS 逻辑操作

2.5.1 指令

逻辑移动指令:

- sll rd, rt, shamt: 逻辑左移指令, rt中的数左移shamt位, 空出的位补0, 结果存rd
- srl rd, rt, shamt: 逻辑右移指令, rt中的数右移shamt位, 空出的位补0, 结果存rd

上述指令为R型指令, 其中op都为0, func分别为0和6, shamt为位移量, rs不使用为 0.
逻辑运算指令:

- add rd, rs, rt: 逻辑与指令, rs和rt按位与, 结果存rd
- or rd, rs, rt: 逻辑或指令, rs和rt按位或, 结果存rd
- nor rd, rs, rt: 逻辑或非指令, rs和rt按位或非, 结果存rd

上述指令为R型指令, op都为0, func分别为20,25,27, shamt全为0.

2.6 MIPS 决策指令

2.6.1 指令

bne和beq指令:

- beq rs, rt, L1: 如果rs=rt跳转到标签为L1的指令
- bne rs, rt, L1: 如果rs!=rt跳转到标签为L1的指令
- j L1: 无条件转移到标签为L1的指令

前两条为I型指令, op分别为4,5,2. 最后一条指令为J型指令.
slt和slti指令:

- slt $rd, $rs, $rt: 若rs<rt, 则rd=1, 否则rd=0
- slti $rt, $rs, constant: 若rs<constant, 则rd=1, 否则rd=0

2.6.2 条件分支代码转 MIPS

将以下代码:

```
# f,g,h,i,j存储于$s0,$s1,$s2,$s3,$s4
if (i == j) f = g + h;
else f = g - h;
```

转换为 MIPS:

```
      bne $s3, $s4 ELSE
      add $s0, $s1, $s2
      j EXIT
ELSE: $s0, $s1, $s2
EXIT: ...
```

2.6.3 循环代码转 MIPS

将以下代码:

```
while (save[i] == k) i += 1; # i存于$s3, k存于$s5, save的基址存于$s6
```

转为 MIPS:

```
LOOP: sll $t0, $s3, 2 # $t0 = i * 4, 找到地址
      add $t0, $s6, $t0 # $t0 = 基址 + 偏移量
      lw $t1, 0($t0) # 从内存中取出数存到$t1
      bne $t1, $s5, EXIT # 若和k不相等退出
```

```
5       addi $s3, $s3, 1 # 循环体
6       j LOOP # 实现循环
7   EXIT: ...
```

2.7 MIPS 函数

2.7.1 指令

- `jal Address`: 跳转到函数地址, 并将PC+4存储于`$ra`以便返回断点处
- `jr $ra`: 返回断点处

2.7.2 栈

我们使用任何寄存器需要保存它原来的值 (类似于中断保存现场), 用完了再把原来的值放回去, 因为寄存器的数量是有限的.

一般来说, `$s`开头的寄存器必须压栈, `$t`/`$a`开头的寄存器不必压栈.

压栈和出栈

由于栈的增长是按地址从高到低的顺序进行的, 所以出栈和入栈的操作分别为:

1. 入栈 (push): `$sp=$sp-4`
2. 出栈 (pop): `$sp=$sp+4`

2.7.3 函数代码转 MIPS

将以下代码:

```
1   int leaf_example (int g, h, i, j)
2   {
3      int f;
4      f = (g + h) - (i + j);
5      return f;
6   }
```

转换为 MIPS:

```
1   # 入栈
2   addi $sp, $sp, -12
3   sw $t1, 8($sp)
4   sw $t0, 4($sp)
5   sw $s0, 0($sp)
```

```
# 运算
add $t0, $a0, $a1
add $t1, $a2, $a3
add $s0, $t0, $t1
addi $v0, $s0, $ZERO # 将结果$s0放到函数返回值寄存器$v0
# 出栈
lw $s0, 0($sp)
lw $t0, 4($sp)
lw $t1, 8($sp)
addi $sp, $sp, 12
# 返回
jr $ra
```

2.8 MIPS 嵌套

不调用其他过程的过程称为**叶过程**, 嵌套调用就是过程体中调用其他的过程 (甚至包括自己)

首先要知道, 递归分为两个阶段: 递归阶段和返回阶段.

假设主程序将参数3传入寄存器$a0, 然后使用jal A调用过程A. 再假设过程A通过jal B调用过程B, 参数为7, 同样存入$a0. 由于A尚未完成任务, 所以寄存器$a0的使用上存在冲突. 同样, 在寄存器$ra保存的返回地址上也存在冲突, 因为它现在保存的是B的返回地址. 所以我们必须采用压栈的方式对数据进行保存:

Caller 把所有在返回阶段需要用到的参数寄存器 ($a0-$a3) 或临时寄存器$t0-$t9压栈. Callee把将所有在返回阶段要用到的返回地址寄存器$ra和保存寄存器$s0-$s7都压栈. 栈指针$sp会随诊栈中寄存器的个数调整. 到返回的时候, 寄存器就会从存储器中恢复, 栈指针也会重新调整.

2.8.1 递归代码转 MIPS

将以下代码:

```
int fact(int n)
{
   if (n < 1) return (1);
       else return (n * fact(n - 1));
}
```

转换为 MIPS:

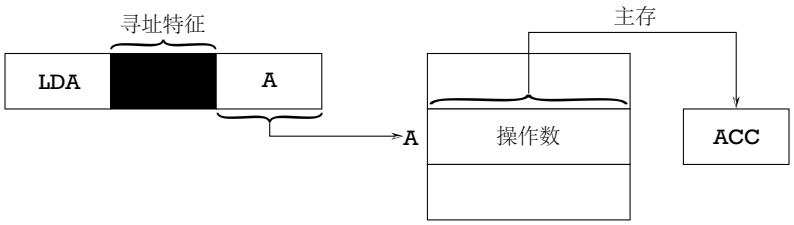
```
fact:
     addi $sp, $sp, -8
     sw $ra, 4($sp)
     sw $a0, 0($sp)
```

```
5       slti $t0, $a0, 1
6       beq $t0, $ZERO, L1
7       addi $v0, $ZERO, 1
8       addi $sp, $sp, 8
9       jr $ra
10      # 申请一块大小为8的空间
11      # 将返回阶段要用到的Caller的返回地址存储到栈
12      # 将返回阶段要用到的Callee的参数存储到栈
13      # 如果$a0也就是n大于等于1，则跳到L1
14      # 如果$a0也就是n小于1，则递归阶段到达最底层，计算0!=1并保存结果
15      # 由于是最底层函数，其$ra和$a0不会被下一层调用，所以可以直接释放栈
16      # 最底层函数返回
17  L1:
18      addi $a0, $a0, -1
19      jal fact
20      lw $a0, 0($sp)
21      lw $ra, 4($sp)
22      addi $sp, $sp, 8
23      mul $v0, $a0, $v0
24      jr $ra
25      # 设置下一层调用函数的参数为n-1
26      # 返回fact，执行fact(n-1)
27      # 开始返回阶段，将递归阶段存储的$a0取出
28      # 开始返回阶段，将递归阶段存储的$ra取出
29      # 释放栈
30      # 根据刚取出的$a0和$v0相乘
31      # 函数返回
```

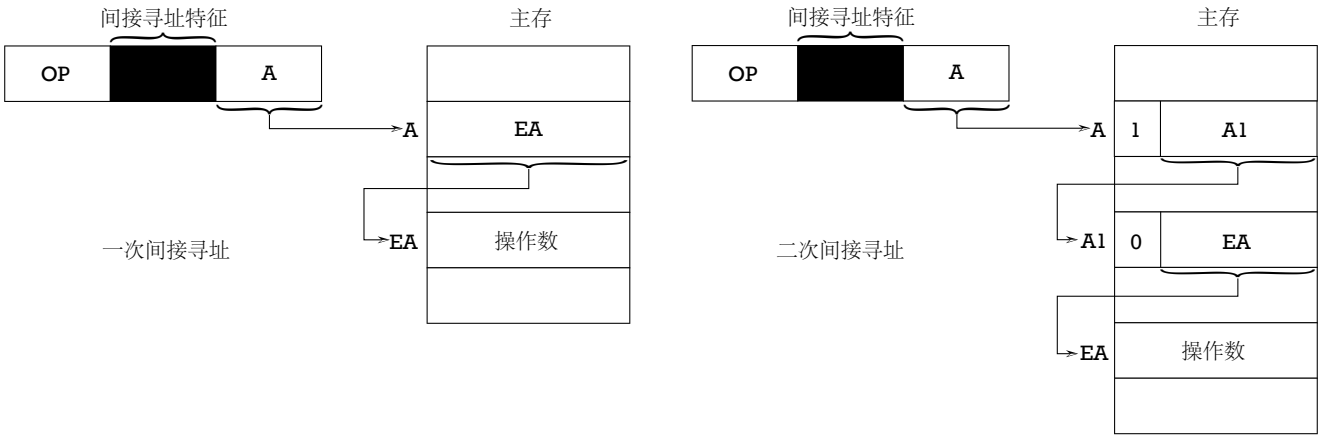
2.9 寻址方式

寻址方式就是根据地址找到指令或者操作数的方法.
假设有数据存储在地址为EA的内存中, 用()表示内存的内容:

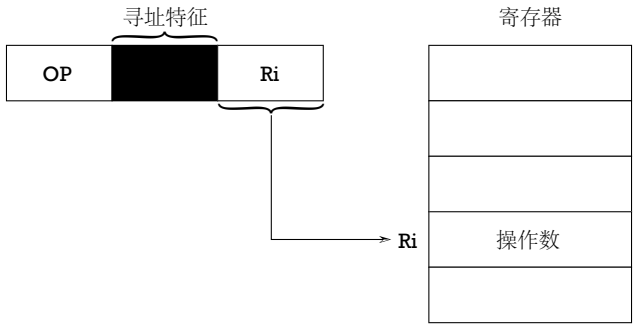
1. 直接寻址: 指令中的形式地址A就是真实地址EA, 即A=EA



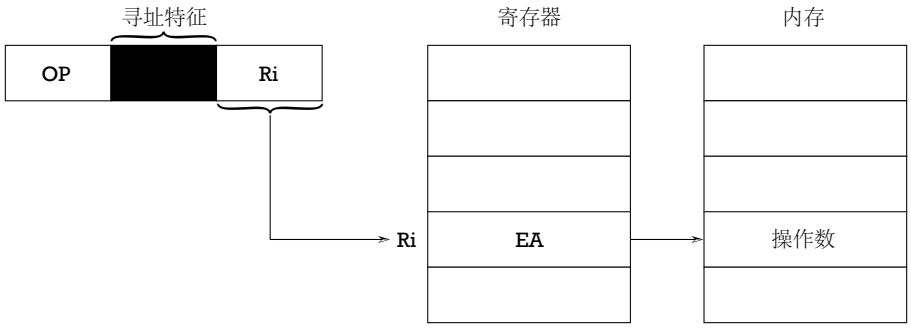
2. 间接寻址: 指令中的形式地址A是真实地址的地址, 即(A)=EA



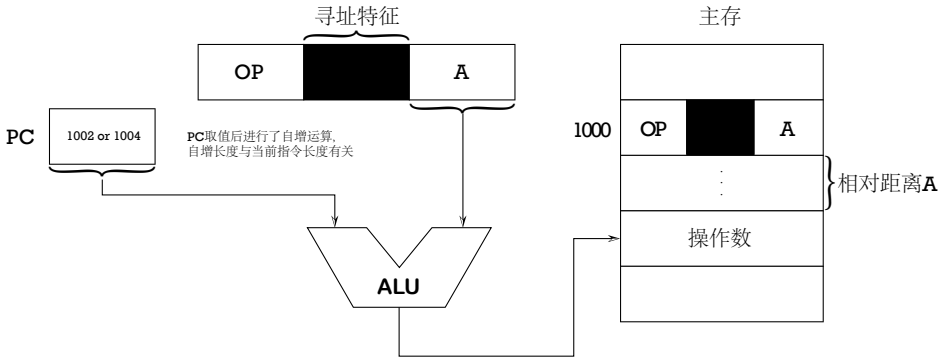
3. 寄存器寻址: 指令中的地址是寄存器号Ri, 寄存器中存储了操作数, 即Ri=EA



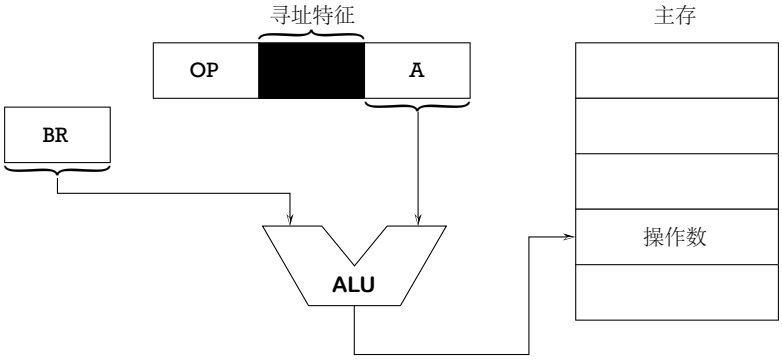
4. 寄存器间接寻址: 指令中的地址是寄存器号Ri, 寄存器中存储了真实地址EA, 即(Ri)=EA



5. 相对寻址: 将程序计数器PC的内容和指令中的形式地址A相加得到真实地址EA, 即(PC)+A=EA



6. 基址寻址: 将基址寄存器BR的内容和指令中的形式地址A相加得到真实地址EA, 即(BR)+A=EA



Chapter 3

运算

3.1 进位计数制

3.1.1 进位计数法

r进制数, 每个数码位可能出现r种字符, 逢r进1.

3.1.2 不同进制数之间的转换

r 进制数 → 十进制

各数码位与位权的乘积之和, 全为1的二进制转十进制: 2^n-1

十进制 → r 进制

- 整数部分: 除基取余法, 先取得的 “余” 是整数的低位 (除r)
- 小数部分: 乘基取整法, 先取得的 “整” 是消暑的高位 (乘r)

二进制 ↔ 八进制

每三个二进制位对应一个八进制位

二进制 ↔ 十六进制

每四个二进制位对应一个十六进制位

3.2 定点数的表示

3.2.1 数的分类

- 有符号数

  - 定点数: 小数点位置固定的数
    * 定点整数: 纯整数
    * 定点小数: 纯小数
  - 浮点数: 小数点位置不定的数 (既有整数又有小数)

- 无符号数 (如地址)

3.2.2 真值和机器数

- 真值: 实际带正负号的数值 (人类习惯的样子)
- 机器数: 把正负号数字化的数 (存到机器里的样子), 包括原码, 反码, 补码, 移码, 他们都是有符号数

3.2.3 原码

1. 含义

   用尾数表示真值的绝对值, 符号位 "0/1" 对应 "正/负". 若机器字长为n+1位, 则尾数就占n位.

2. 范围

   0表示不唯一, ([+0]原=0,0000000,[-0]原=1,000000), 范围为-2^n-1~2^n-1, n为尾数长度

3. 示例

   机器字长为8位, 将真值x=-14转换为原码: [x]原=1,0001110(注意: 不足的机器字长要补0)

3.2.4 反码

1. 含义

   - 正数: 与原码相同
   - 负数: 符号位不变, 数值位取反

2. 范围

   0表示方法不唯一, ([+0]反=0,0000000, [-0]反=1,1111111), 范围为-2^n-1~2^n-1, n为尾数长度

3. 示例

   机器字长为8位, 将真值x=-14转换为反码: [x]反=1,1110001

3.2.5 补码

1. 含义

   先将原码转化为反码, 再将数值位+1.

   或者, 将原码直接转为补码: 从后往前, 遇到的第一个1之前不变, 后面取反 (符号位不变), 反过来补码转原码也是如此.

2. 范围

   0的表示方法唯一, ([0]=0,0000000, 1,0000000用于表示-128), 范围为-2^n~2^n-1, n为尾数长度

3. 示例

   机器字长为8位, 将真值x=-14转换为补码: 原码为[x]原=1,0001110, 从右往左第一个1不变, 左边取反, 得到补码[x]补=1,1110010

比较这两个数字的大小: 1,1111111和1,0000000, 前者大, 因为前者转换为真值是1, 而后者转化为真值为-128

3.2.6 移码

移码与原码, 反码, 补码不同, 他是一种无符号数.

移码 = 真值 + 偏置值, 若机器字长为n+1位, 则偏置值为+2^n, 当偏置值为+128的时候: 补码 = 补码的符号位取反.

偏置值可以取其他值, 如在IEEE 754中, 单精度浮点数的偏移量为+127, 即+01111111.

3.3 定点数的运算

3.3.1 移位运算

左移相当于*2, 右移相当于/2

- 逻辑移位: 无符号数, 当成正数, 补0
- 算术移位: 有符号数, 有正有负
  - 正数: 符号位不参与移位, 原码, 反码, 补码数值位均补0
  - 负数: 符号位不参与移位, 原码补0, 反码补1, 补码如果是左移, 补0; 如果是右移, 补1

左移: 若舍弃的位为1, 将产生严重误差;
右移: 若舍弃的位为1, 将丢失精度.

3.3.2 加减运算

- 原码的加减运算
  - 加法运算
    * 正 + 正: 绝对值做加法, 符号位为0
    * 负 + 负: 绝对值做加法, 符号位为1
    * 正 + 负: 绝对值大的减绝对值小的, 符号位同绝对值大的数
    * 负 + 正: 绝对值大的减绝对值小的, 符号位同绝对值大的数

  - 减法运算: 减数的符号位取反, 转变为加法

- 补码的加减运算
  - 加法运算: [A+B]补=[A]补+[B]补
  - 减法运算: [A-B]补=[A]补+[-B]补

3.4 浮点数的表示和运算

3.4.1 浮点数的表示格式

浮点数N=M*r^E, 其中N为浮点数, M为尾数, E为阶码, r为基数, 二进制的话为2, 上述数字均为十进制.

3.4.2 浮点数的规格化

规格化浮点数: 规定原码尾数的最高位一定要是个 1⇔ |M| 属于[0.5,1], 所以衍生出以下两种规范化的方法:

- 左规: 说明 M<1/2, 太小, 需要扩大, 左移1位将尾数的数值位扩大到原来的2倍, 同时阶码减1
- 右规: 说明 M>1, 太大, 需要缩小, 右移1位将尾数的数值位缩小到原来的1/2倍, 同时阶码加1

尾数可以用原码或者是补码表示, 所以上述左规和右规可以分别用于原码或者补码的规格化:

- 原码: 尾数最高位为1
  - 尾数为正数: 通过左规和右规得到的标准形式应该是0.1XXXXXXXX
  - 尾数为负数: 通过左规和右规得到的标准形式应该是1.1XXXXXXXX

  注意, 上面尾数可以等于0.5或者是-0.5, 只要是0.1000000...和1.1000000...即可. 但是做不到等于1或者-1. 此外, 他们的数值位第一位都是1, 所以在IEEE 754中, 数值位的第一位可以默认省略.
- 补码: 尾数最高位与符号位相反
  - 尾数为正数: 通过左规和右规得到的标准形式应该是0.1XXXXXXXXX
  - 尾数为负数: 通过左规和右规得到的标准形式应该是1.0XXXXXXXXX

3.4.3 IEEE 754 标准

数符	阶码的数值部分-移码	尾数的数值部分-原码

阶码

阶码用移码表示, 单精度浮点数下长度为8位.

移码是一种无符号数, 由于移码 = 真值 + 偏置值, 加完之后大于等于0, 所以阶码没有符号位. 偏置值和数的类型有关, 单精度浮点数的偏置值为+127, 双精度浮点数的偏置值为+1023.

尾数

尾数用原码表示, 单精度浮点数下长度为23位.

尾数是一种有符号数, 他的符号放在开头, 数值部分放在末尾. 通过规格化后, 数值部分默认隐藏首位1, 所以单精度浮点数下实际能表示的数值位有24位.

举例

以-1.75*2^(-3)为例:

- 数符: 由于是负数, 所以是1.
- 尾数的数值部分: 十进制下是1.75>1, 故需要右归, 尾数的数值部分右移1位, 阶码需要加1, 得到尾数的数值位为0.875即1110000...(共24位), 阶码为-2. 由于尾数的数值位的首位1可以省略, 所以最终尾数的数值位为110000...(共23位).
- 阶码: 经过上述规格化后, 阶码为-2, 加上偏置值+127, 得到移码十进制下为+125, 转换为二进制为01111101.

综上所述, 我们得到的IEEE 754下的表示为: 1,01111101,110000...(共23位).

3.4.4 浮点数的加减运算

我们通常在十进制下做浮点数的加减运算.

步骤

1. 对阶: 保持阶数一致, 小阶数向大阶数对齐
2. 尾数相加/减
3. 规格化, 保证尾数M属于[0.5,1]

举例

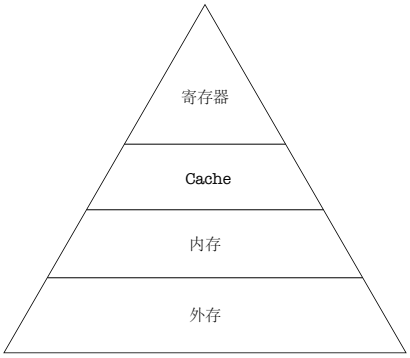
实现`0.75*2^4+1.5*2^3`:

1. 对阶: 小阶 → 大阶, `1.5*2^3`→`0.75*2^4`
2. 加减: `0.75*2^4+0.75*2^4=1.5*2^4`
3. 规格化: `1.5*2^4`→`0.75*2^5`

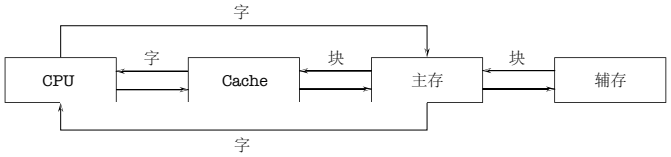
Chapter 4

存储器

4.1 存储器的层次



(a) 多级存储器结构



(b) 存储器之间的数据交换

这就是存储器的层次结构. 在pyramid的越上层, 价格越高, 速度越快, 容量越小. 要注意的是, Cache和主存之间的数据交换是由硬件完成的, 对所有程序员透明. 主存和辅存之间的数据交换是由硬件和操作系统共同完成的. 对应用程序员透明.

4.2 存储器的分类

- 按层次分类
  - 主存储器: 简称主存, 或者内存. 可以直接和Cache交互
  - 辅助存储器: 简称辅存, 或者外存. 是主存的后援存储器
  - 高速缓冲存储器: 简称Cache. 放在CPU中
- 按存取分类
  - 随机存储器: RAM, 可读可写, 存取时间和物理位置无关, 断电丢失数据

    * 静态随机存储器: SRAM, 双稳态触发器, 用于Cache
    * 动态最急存储器: DRAM, 栅极电容, 用于内存
  - 只读存储器: ROM, 只读, 存取时间和物理位置无关, 断电不丢失数据
  - 串行访问存储器: 存储时间和物理位置有关, 如顺序存取存储器 (磁带) 与直接存取存储器 (磁盘, 光盘)

- 按介质分类分为: 磁表面存储器 (磁盘, 磁带), 磁芯存储器, 半导体存储器 (MOS 型存储器, 双极型存储器) 和光存储器 (光盘)

- 按可保存性分类

  - 易失存储器: 断电后, 存储信息消失. 如: RAM
  - 非易失存储器: 断电后, 存储信息依然保持, 如: ROM
  - 破坏性读出: 某个存储单元被读出时, 原存储器信息被破坏
  - 非破坏性读出: 某个存储单元被读出时, 原存储器信息不会被破坏

4.3 Cache

Cache是由SRAM构成的, 在CPU中的一种存储器.

4.3.1 基本原理

操作系统为了缓和CPU和主存之间的速度矛盾, 将某些主存块放到了Cache中, 而这个步骤是基于局部性原理:

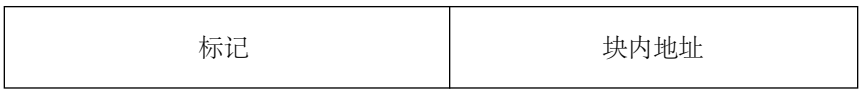
- 时间局部性: 现在访问的地址, 不久之后也很可能再次被访问
- 空间局部性: 现在访问的地址, 其附近的地址也很可能即将被访问

4.3.2 主存到 Cache 的映射

Cache被分成与主存的页框大小一致的Cache块 (或称 Cache 行), 两者之间以块为单位进行数据交换. 下面解释了内存中的块应该放在Cache中的哪个位置:

全相联映射

主存的块可以装入Cache的任何块, 没有公式. 相应的地址结构为:



直接映射

主存中的块必须放入下面公式的指定Cache块:

`j = i mod 2^c，其中j为Cache块号，i为主存块号，2^c为Cache的总块数`

从上面的映射关系可以看出, 主存块号的低c位正是它要装入的Cache行号. 若主存中用m位表示块号, 则标记 (有效位 + 内存地址高位) 共m-c位, Cache块号共c位, 相应的地址结构为:

标记	Cache块号	块内地址

组相联映射

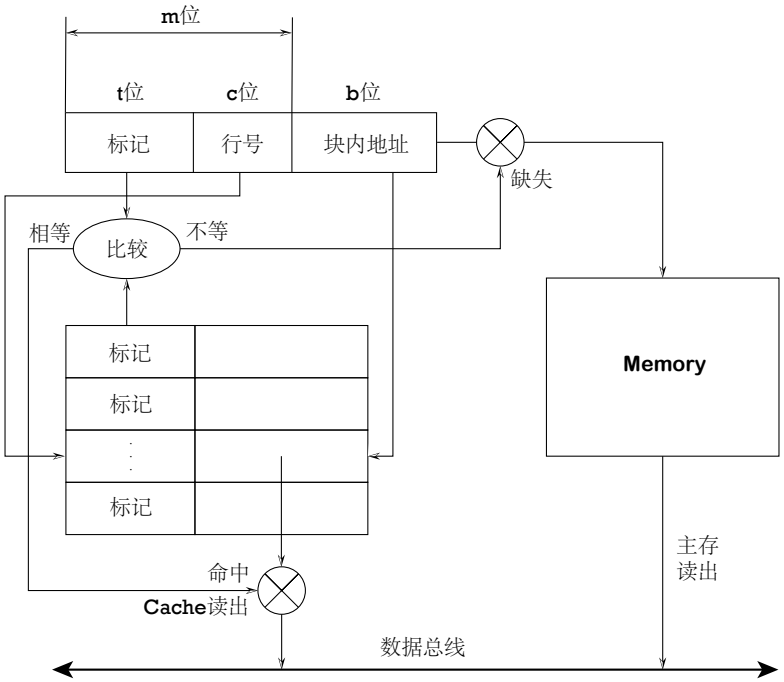
将Cache分成大小相等的组, 主存的一个数据块可以装入一组内的任意位置, 即组间采取直接映射, 组内采取全相联映射, 内存块可以放在下面公式指定的Cache组中:

`j = i mod Q，其中j为Cache块号，i为主存块号，Q为组数`

与直接映射相似, 主存块号的低位用来表示组号, 内存块号的高位和有效位用来表示标记, 相应的地址结构为:

标记	组号	块内地址

根据上述的地址结构, 我们应该可以推断出CPU的访存过程:



直接映射

上图为直接映射方式, 全相联映射中没有组, 行号, 内存地址高位低位的概念, 他的标记就是有效位 + 内存块号. 组相联映射是一种介于全相联映射和直接映射之间的一种映射.

4.3.3 替换算法

由于Cache的空间通常很小, 所以用不到的Cache块就要及时换出, 下面说明上述三种映射中替换算法要解决的问题:

- 全相联映射

  Cache满了才需要替换, 需要在全局选择替换哪一块

- 直接映射

  如果对应位置非空, 则毫无选择地直接替换, 无需考虑替换算法

- 组相联映射

  分组内满了才需要替换, 需要在分组内选择替换哪一块

随机算法 (RAND)

若Cache已满, 则随机选择一块替换.
实现简单, 但是完全没有考虑局部性原理. 命中率低, 实际效果很不稳定.

先进先出算法 (FIFO)

若Cache已满, 则替换最先被调入Cache的块.
实现简单, 也是完全没有考虑局部性原理, 会出现抖动现象: 频繁的换入换出现象

近期最少使用算法 (LRU)

为每个Cache设置一个 “计数器”, 用于记录每个Cache块已经多久没有被访问了, 当Cache满后替换 “计数器” 最大的.

1. 命中

   命中行计数器清零, 比其低的计数器加1, 其余不变

2. 未命中

   (a) 有空闲行

   空闲行计数器清零, 其余非空闲行加1

   (b) 无空闲行

   计数器值最大行的信息块被淘汰, 新装行的块的计数器清零, 其余全加1

该算法基于局部性原理, 运行效果优秀, Cache命中率高.

最不经常使用算法 LFU

为每个Cache设置一个 “计数器”, 用于记录每个Cache块被访问过几次, 当Cache满后替换 “计数器” 最小的.

新调入的块计数器=0, 之后每一次被访问计数器+1, 需要替换的时候, 选择计数器最小的一行 (若相同, 选择行号更小的一行).

最被经常访问的主存块在未来不一定会用得到, 并没有很好的遵循局部性原理, 实际效果不如LRU.

4.3.4 Cache 写策略

由于CPU在访存的时候会优先访问Cache, 假设这次的命令是要到某个块里修改数据, 那么就有两种情况, 即 “写命中”(要写的块在Cache中) 和 “写不命中”(要写的块不在Cache中), 他们均会造成内存和Cache的数据不一致性, 下面的方法能解决这个问题:

- 写命中
  - 写回法
  - 全写法
- 写不命中
  - 写分配法
  - 非写分配法

写回法

适用于写命中. 只修改Cache中的内容, 而不立即写入主存, 只有在此块被换出的时候再写回主存.

减少了访问次数, 但是存在数据不一致的隐患.

全写法 (写直达法)

适用于写命中. 同时修改Cache中的数据和主存中的内容, 一般使用写缓冲的方法, 创建一个队列, 慢慢地写回主存.

访存次数增加, 速度变慢, 但能保证数据的一致性. 如果使用写缓冲, 若写操作很频繁, 会导致写缓冲饱和而阻塞.

写分配法

适用于写不命中. 把主存中的块调入Cache, 在Cache中修改. 搭配写回法使用.

非写分配法

适用于写不命中. 只写入主存, 不调入Cache. 搭配全写法使用.

4.3.5 多级 Cache

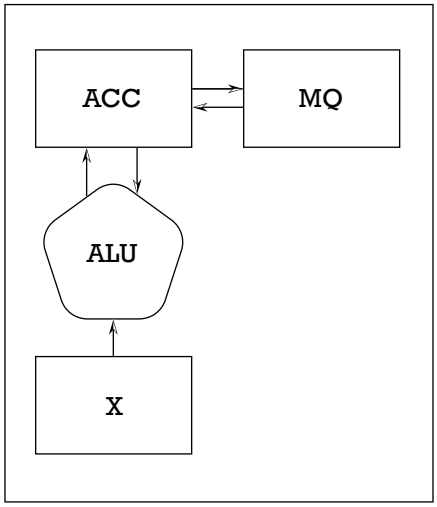
各级Cache之间使用的是 “全写法 + 非写分配法”, Cache-主存之间使用的是 “写回法 + 写分配法”.

Chapter 5

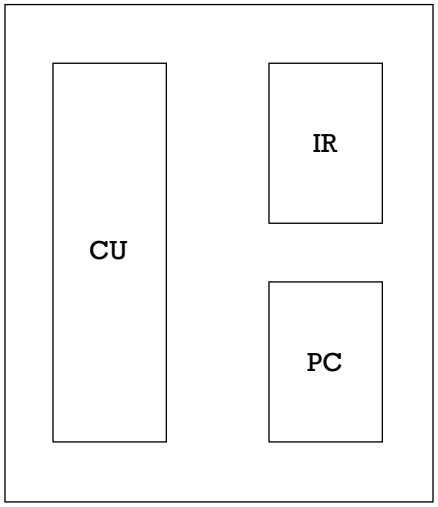
处理器

5.1 处理器的结构

CPU由控制器和运算器组成:



(a) 运算器



(b) 控制器

5.1.1 控制器

控制器内有三个重要的部件:

- CU: Control Unit, 分析指令, 给出控制信号
- IR: Instruction Registor, 存放当前指令
- PC: Program Counter, 存放下一条指令的地址, 有自动加1的功能

那么, 控制器能实现什么功能呢?

- 取指令
- 分析指令
- 执行指令, 发出各种操作命令
- 控制程序输入及结果的输出
- 总线管理
- 处理异常情况和特殊请求

5.1.2 运算器

运算器内有四个重要的部件:

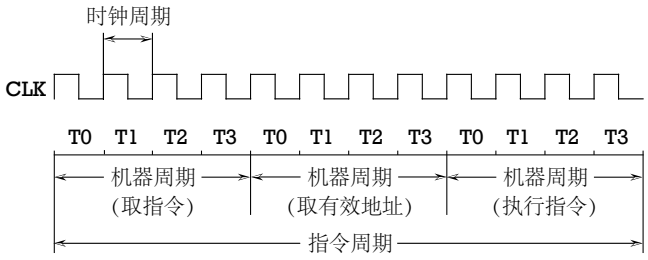
- ACC: 累加器, 用于存放被操作数, 或者运算结果
- MQ: 乘商寄存器, 在乘, 除运算的时候, 用于存放操作数或者运算结果
- X: 通用的操作数寄存器, 用于存放操作数
- ALU: 算术逻辑单元, 通过内部复杂的电路实现算术运算, 逻辑运算

运算器的功能, 顾名思义, 实现算术和逻辑运算.

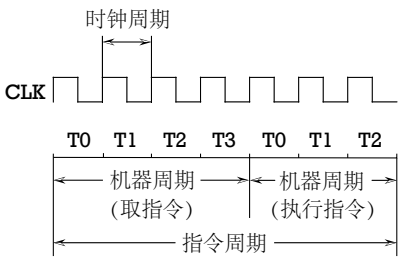
5.2 指令周期

CPU从主存中取出并执行一条指令的时间称为指令周期. 不同指令的指令周期可能不同. 指令周期用若干的机器周期表示, 一个机器周期又包含若干时钟周期 (节拍).

根据机器周期内的节拍数是否相等, 可以拆分为:



(a) 定长机器周期



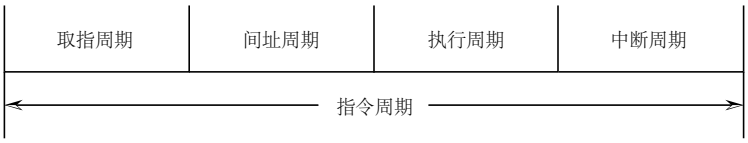
(b) 不定长机器周期

对于无条件转移指令j, 在取值令完成后无需访存, 所以只包含取值周期和执行周期 (都属于机器周期).

对于间接寻址指令, 在取指令完成后仍需访存, 因为取指令阶段取的是指令, 要根据这个指令访存取出有效地址, 所以还包括间址周期.

CPU在每条指令结束之前, 都要发中断查询信号, 若有中断请求, 则CPU进入中断响应阶段, 所以还包括中断周期.

综上所述, 一个一般的指令为:

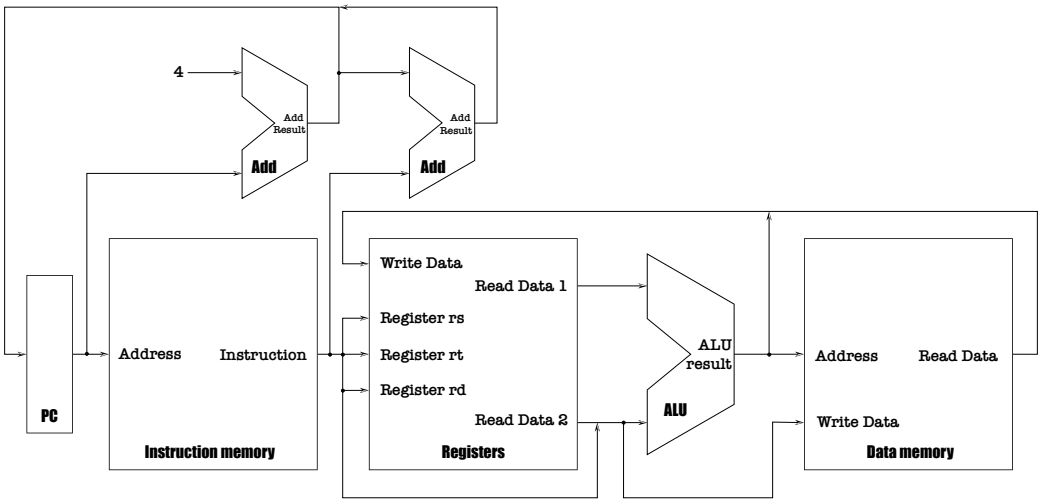


四个周期的工作:

- 取指周期: 取出并分析指令
- 间址周期: 取有效地址
- 执行周期: 取操作数, 并执行指令
- 中断周期: 保存程序断点

5.3 MIPS 指令体系

5.3.1 一个基本的 MIPS 实现



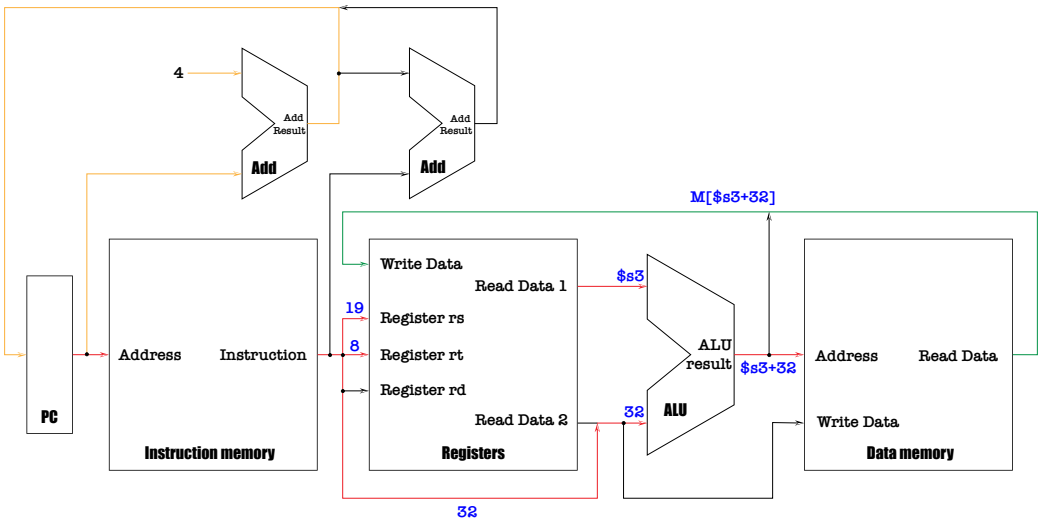
MIPS 子集基本实现 (不包含选择器和控制器)

1. `lw` 操作

   我们以`lw $t0, 32($s3)`为例, 首先, 将上述指令转化为十进制的样式:

35(op)	19(rs)	8(rt)	32(Address)

   然后, 将数据流在电路图中表示:



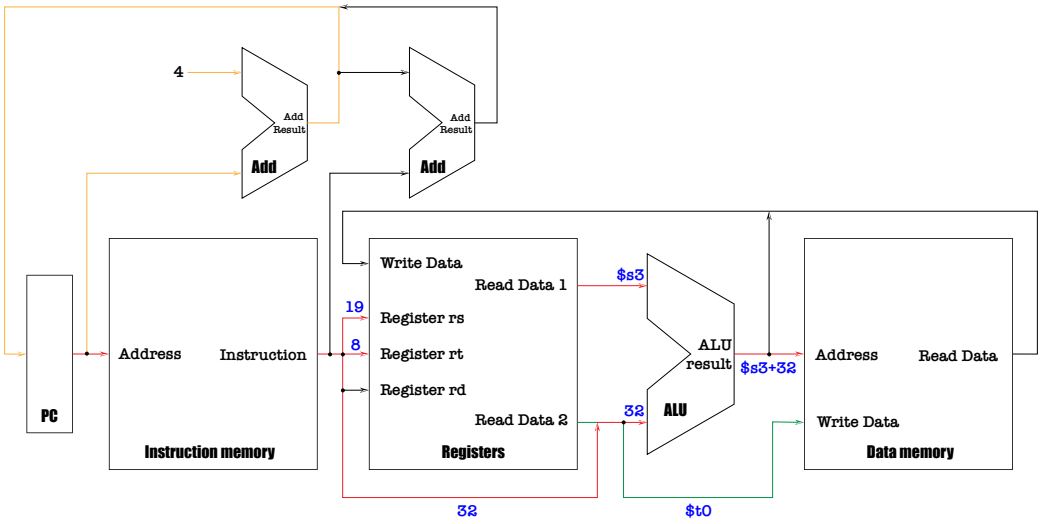
上图表示`lw $t0, 32($s3)`指令执行的源数据流向, 结果流向, PC 改变.

2. `sw` 操作

   我们以`sw $t0, 32($s3)`为例, 首先, 将上述指令转化为十进制的样式:

43(op)	19(rs)	8(rt)	32(Address)

然后, 将数据流在电路图中表示:



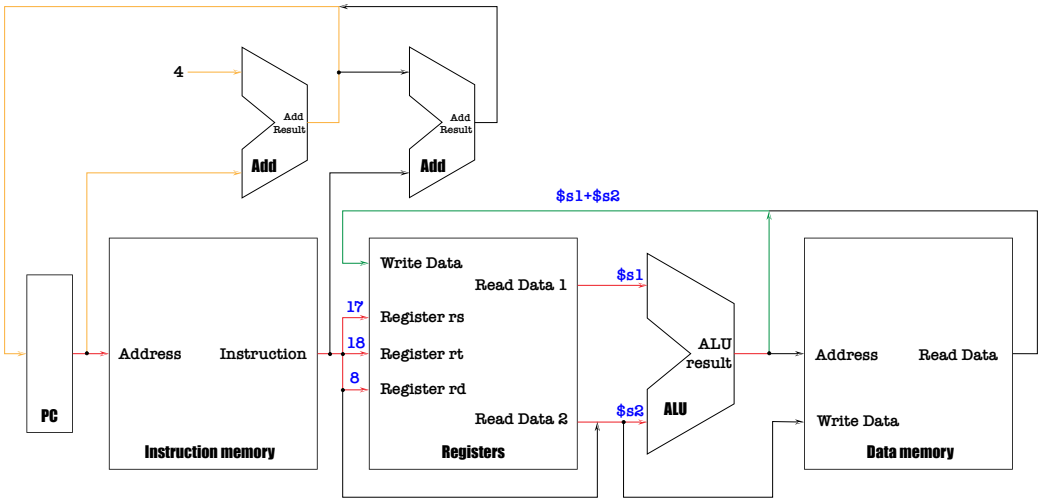
上图表示`sw $t0, 32($s3)`指令执行的源数据流向, 结果流向, PC 改变.

3. `add` 操作

   我们以`add $t0, $s1, $s2`为例, 首先, 将上述指令转化为十进制的样式:

0(op)	17(rs)	18(rt)	8(rd)	0(shamt)	32(funct)

然后, 将数据流在电路图中表示:



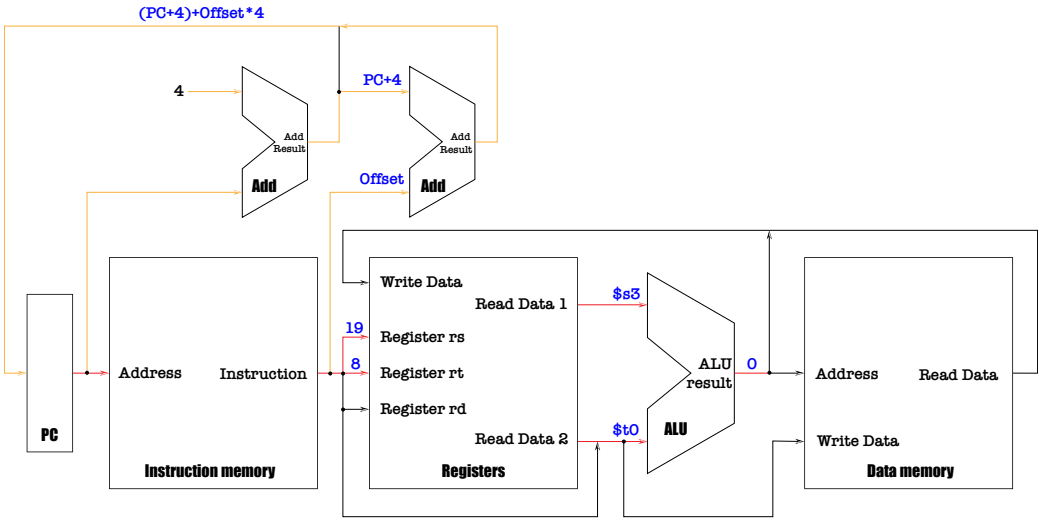
上图表示add $t0, $s1, $s2指令执行的源数据流向, 结果流向, PC 改变.

4. beq 操作

我们以beq $s3, $t0, label为例, 首先, 将上述指令转化为十进制的样式:

4(op)	19(rs)	8(rt)	label(offset)

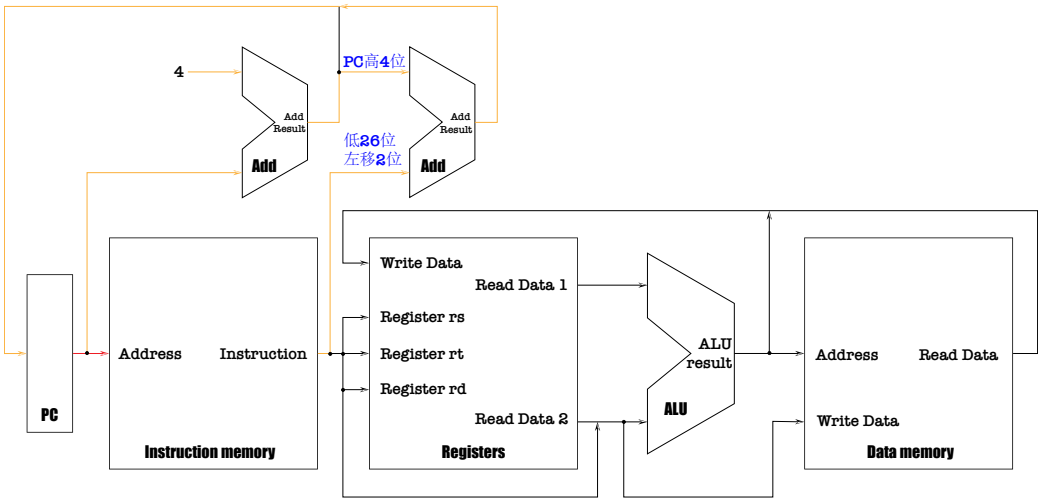
然后, 将数据流在电路图中表示:



上图表示beq $s3, $t0, label指令执行的源数据流向, PC 改变.

5. j 操作

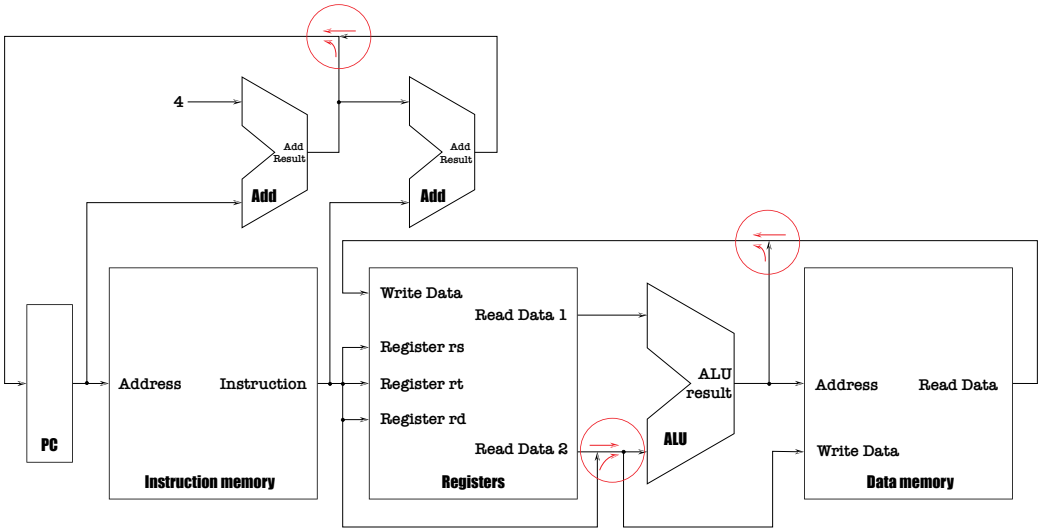
我们以j Addr为例, 在电路图中表示为:



上图表示j Addr指令执行的PC 改变.

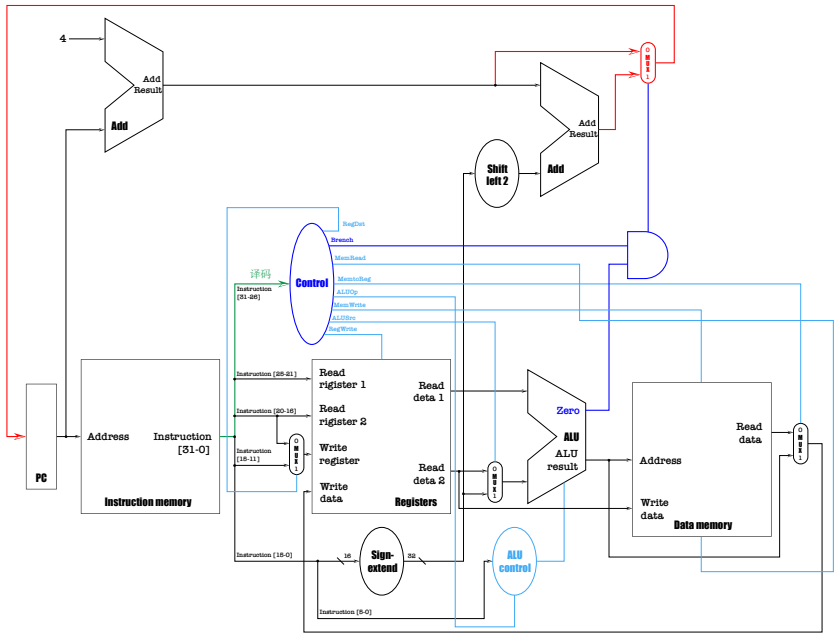
增加选择器和控制器

观察上面的电路图我们会发现, 有很多分支电路



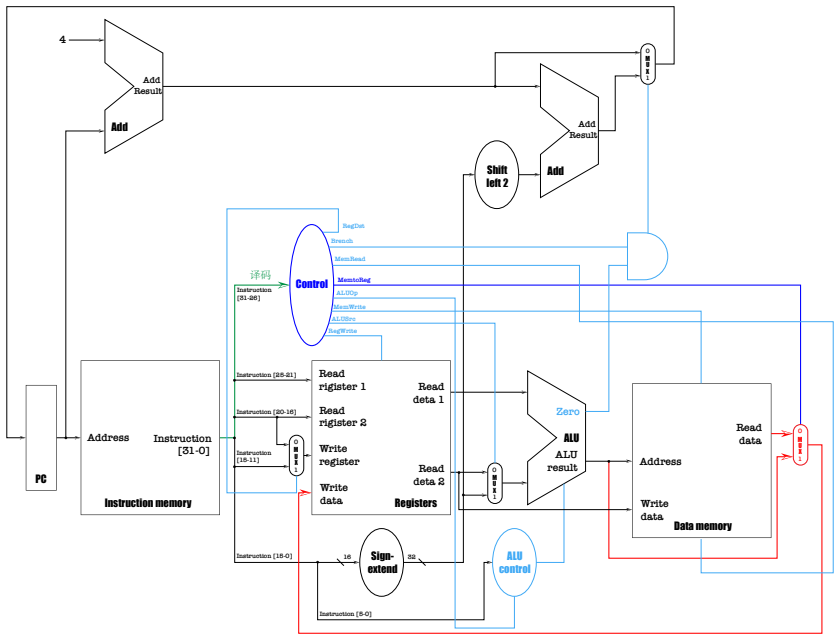
所以数据到底应该走哪一条呢? 我们需要添加选择器和控制器以选择数据:

- 1号选择器 + 控制器



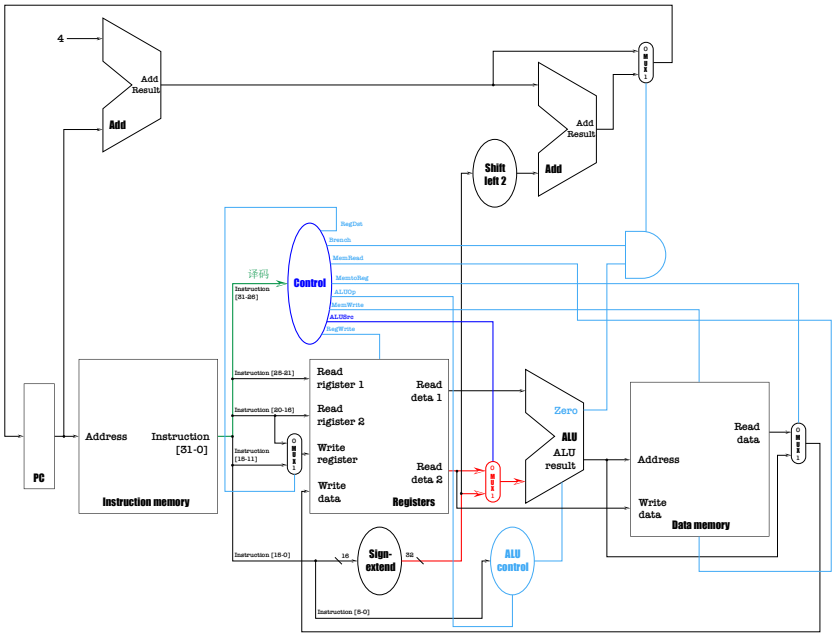
该选择器的作用是选择PC+4还是PC+4+Offset*4，条件由控制器控制，若指令译码结果op为beq指令，则ALU如果相减为0则否，选择器选择PC=PC+4+Offset*4.

- 2号选择器 + 控制器



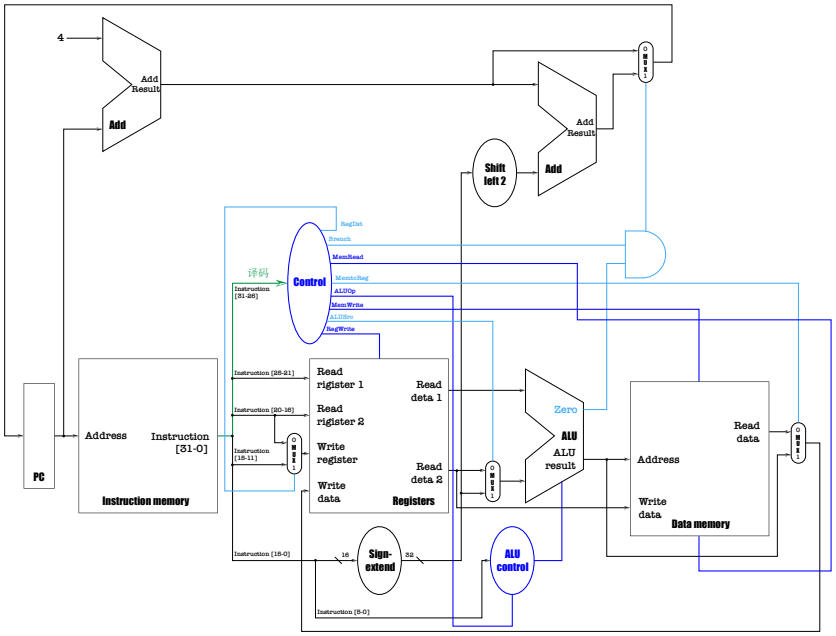
该选择器的作用是选择lw还是逻辑运算指令add,sub...，前者存入寄存器的是从主存中读出的数据，而后者是ALU计算的数据或者是寄存器置0/1.

- 3号选择器 + 控制器



该选择器的作用是选择`lw, sw`指令的`Offset`, 还是算数逻辑命令指明的寄存器的值.

- 其余控制器

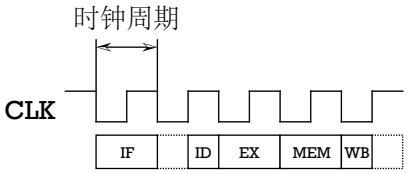


用于控制寄存器, 存储器的读写, `ALU`进行何种运算.

5.4 流水线

流水线是一种实现多条指令重叠执行的计数, 它改善的是系统的吞吐率, 每一个任务本身的执行时间不变.

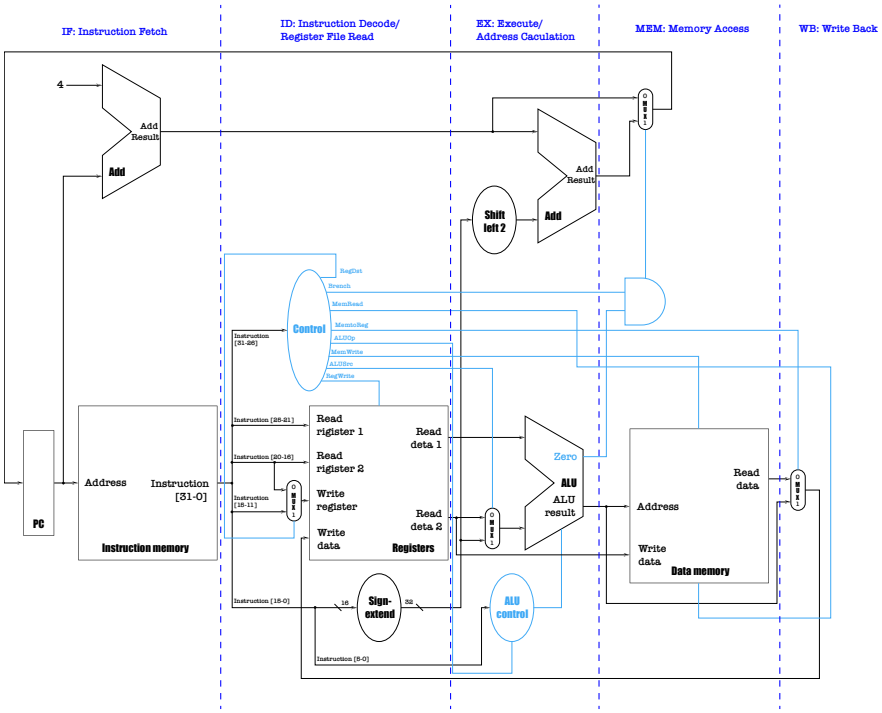
5.4.1 处理步骤



执行一条MIPS指令包含5个步骤:

1. `Instructioin Fetch`: 从指令存储器中读出指令
2. `Instruction Decode`: 指令译码, 并且同时读出寄存器的内容
3. `Execute`: 执行操作或者计算地址
4. `Memory Access`: 从数据存储器中读取操作数
5. `Write Back`: 将结果写回寄存器

从电路图上看:



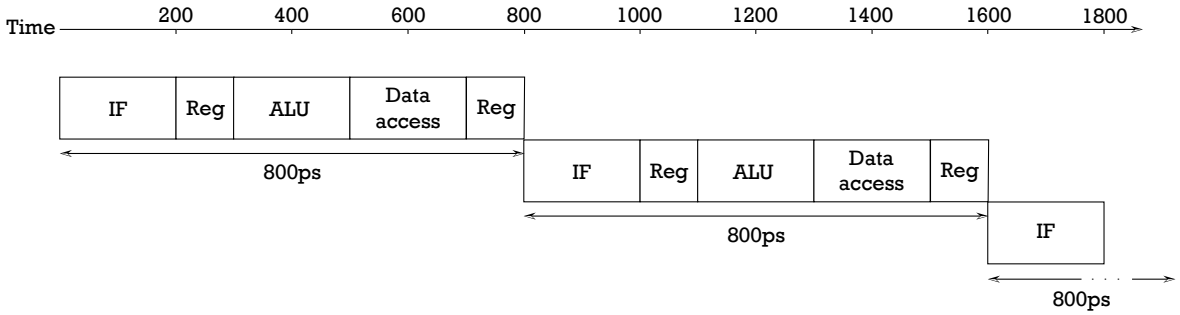
5.4.2 单周期和流水线

单周期指的是指令以串行的方式执行, 而流水线就是指令以并行的方式执行.

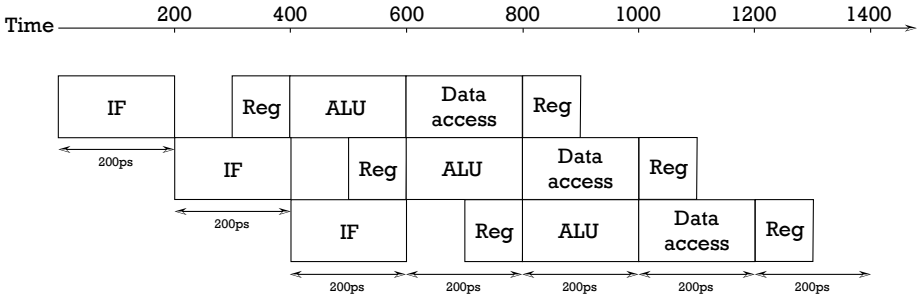
单周期每一个指令的时间相同, 而流水线每一个步骤的时间相同. 这是什么意思呢? 假设我们现在要执行3次`lw`指令, 先看`lw`指令内部每一个步骤花的时间:

Instr	IF	Reg Read	ALU	MEM	Reg Write
lw	200ps	100ps	200ps	200ps	100ps

根据单周期和流水线的不同, 我们画出以下的以时间为轴的过程图:



(a) 单周期



(b) 流水线

假设对寄存器的读操作发生在时钟周期的后半部分, 写操作发生在前半部分. 可以看到, 流水线的时钟周期受限于最慢的处理步骤, 时间为200ps, 所以所有的流水级 (Stage) 都是200ps. 而单周期的每一个指令都是800ps.

加速比=3*800/(800+3*200)≈1.7, 如果指令数量足够多的化, 如再增加1000000条, 则加速比=1000000*800/(800+1000000*200)≈4. 没有达到5, 理想情况下可以达到.

总结

- 流水线可以改善吞吐率
- 有些指令的有些时钟周期是浪费的

- 加速比=单周期指令的执行时间/流水线指令的执行时间≈流水线级数(理想情况)
- 单挑指令的执行时间相比于单周期不减少, 有时反而会增加 (如上述流水线中单条指令的执行时间为1000ps)

5.4.3 冒险

流水线线还有这样一种情况, 在下一个时钟周期中下一条指令不能执行. 这种情况称为冒险. 下面介绍三种冒险:

- 结构冒险

  由于硬件结构导致的冒险, 两条指令没法同时使用一块硬件

  (如两条指令在不同的阶段需要同时访问相同的寄存器)

- 数据冒险

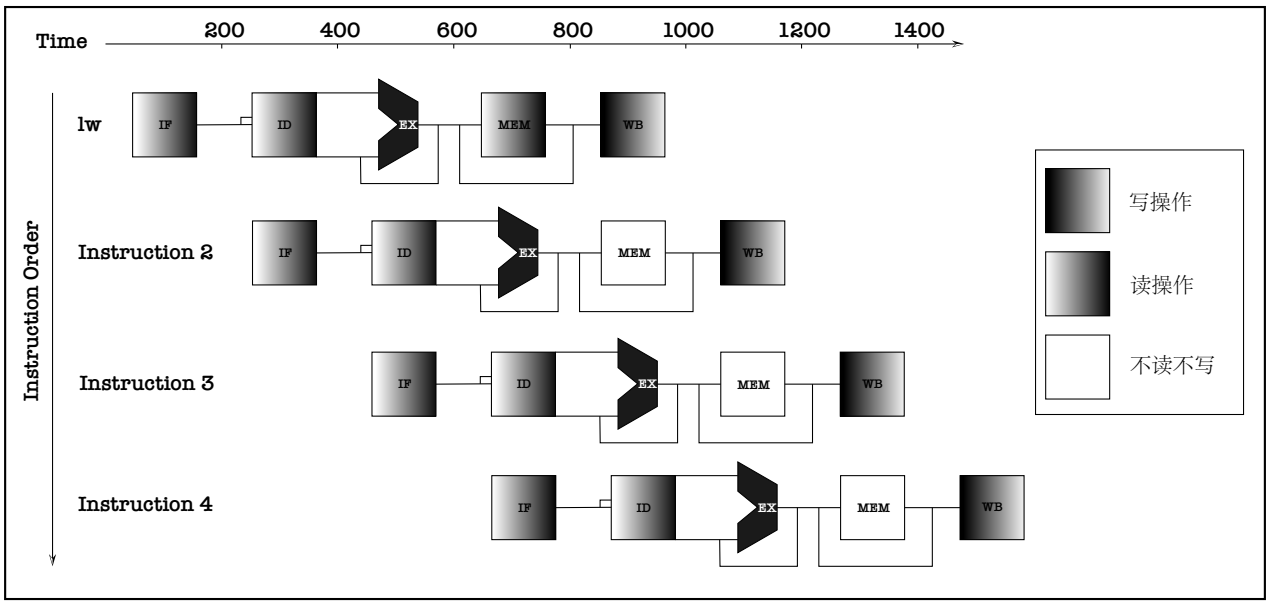
  下一条指令需要数据, 上一条指令还再计算中.

```
1 add $s0, $t0, $t1
2 sub $t2 $s0 $t3
3 sub指令执行到ID的时候, add指令只执行到EX阶段, 需要等到add指令执行完WB才能把结果写回$s0中
```

- 控制冒险 (分支冒险)

  下一条指令取决于上一条分支指令的结果.

流水线的图形表示

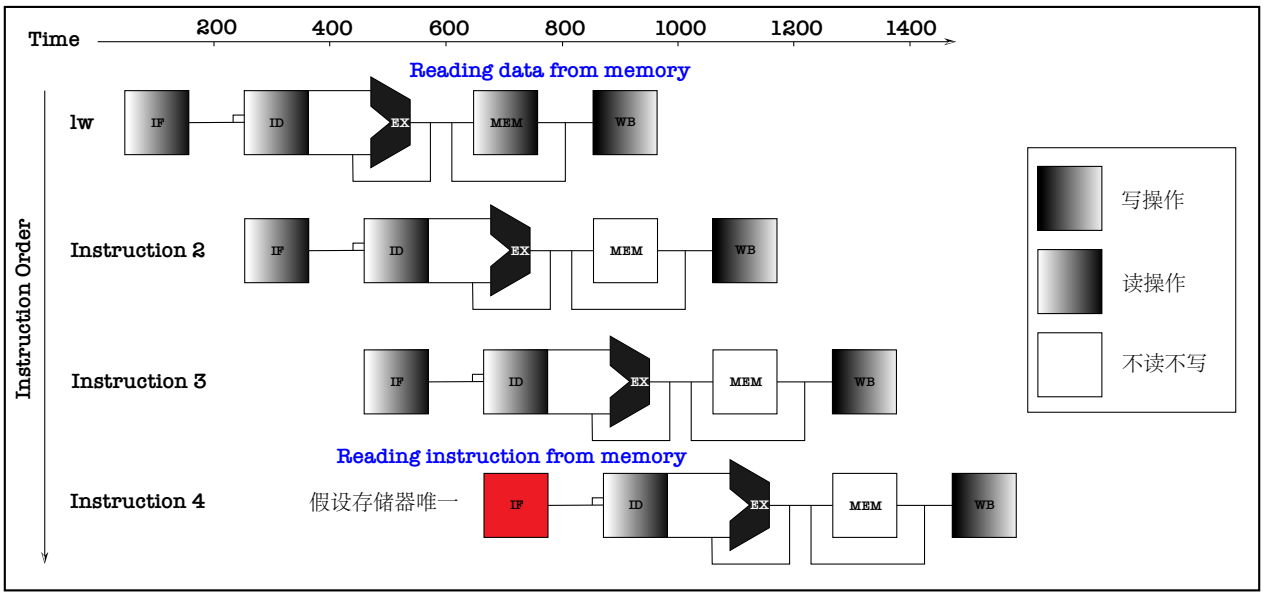


上图的方框含义:

- IF: 表示取指阶段, 其外方框表示存储器, 右边的阴影表示读取存储器
- ID: 表示指令的译码或者寄存器堆的读取阶段, 其外方框表示要读取的寄存器堆, 右边的阴影表示读取寄存器堆
- EX: 表示指令的执行阶段, 其外边的图符表示ALU, 阴影表示使用ALU计算
- MEM: 表示存储器访问阶段, 其外放框表示存储器, 右边的阴影表示读取存储器
- WB: 表示写回阶段, 其外方框表示被写回的寄存器堆, 左边的阴影表示写入寄存器堆

结构冒险

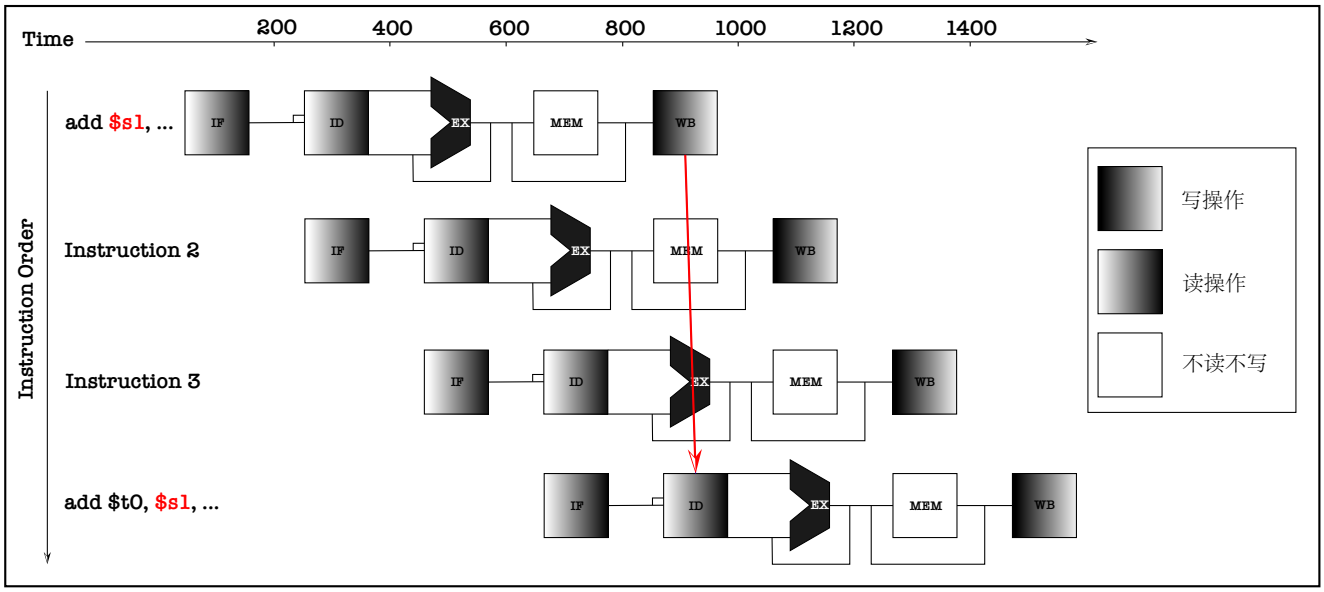
结构冒险是由于两条指令没法使用同一个硬件.



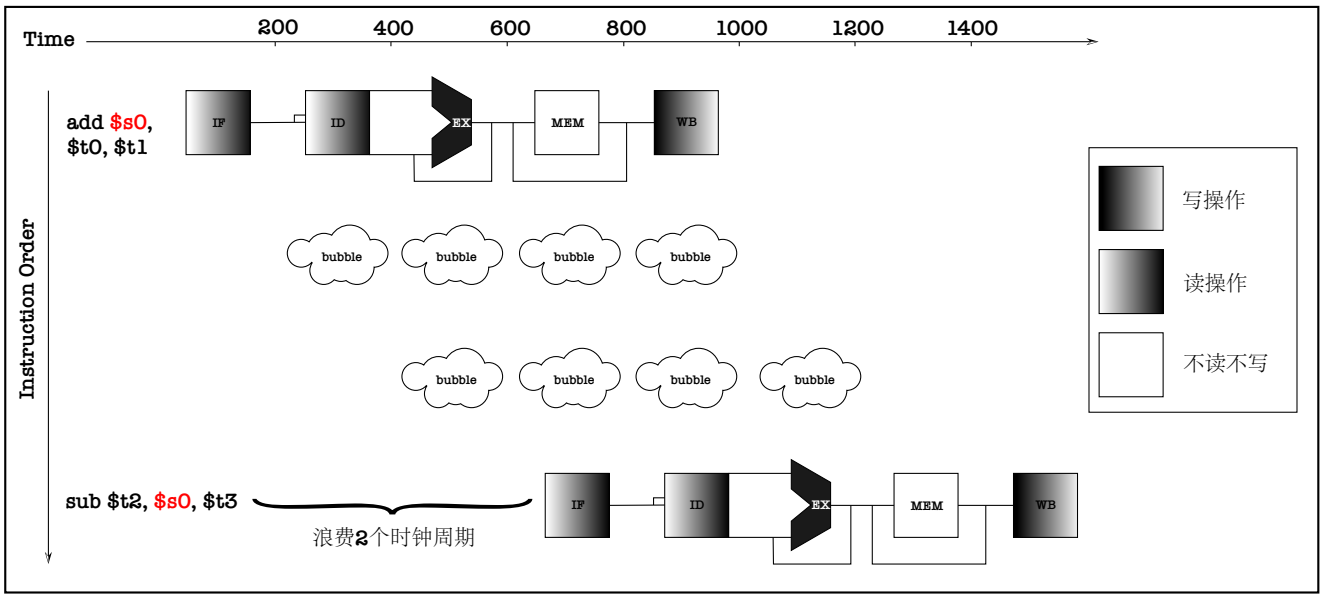
由于使用了同一个存储器, 第1条指令在访问存储器的时候, 第4条指令无法取指令. 进而就产生了结构冒险. 由于MIPS是为流水线而设计的, 所以在设计之初就想到了这种问题, 所以就采取了一系列应对措施, 如将存储器拆分为指令存储器和数据存储器.

数据冒险

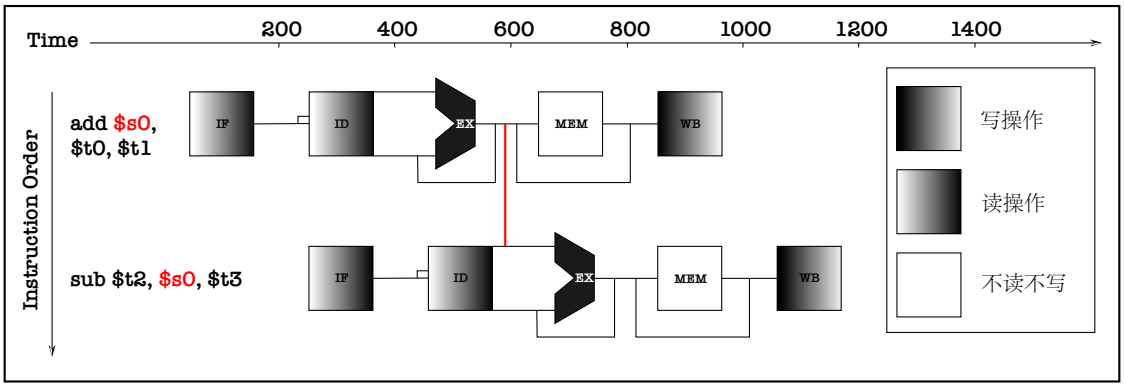
数据冒险是由于一条指令必须等待另一条指令的完成. 比较理想的情况是: 第1条指令的WB步骤在时钟周期的前半部分已经完成写操作, 第4条指令的Reg Read步骤在时钟周期的后半部分紧接着完成读操作, 如下图



然而, 事实上, 这种情况几率较小, 大多数的情况下会造成阻塞:

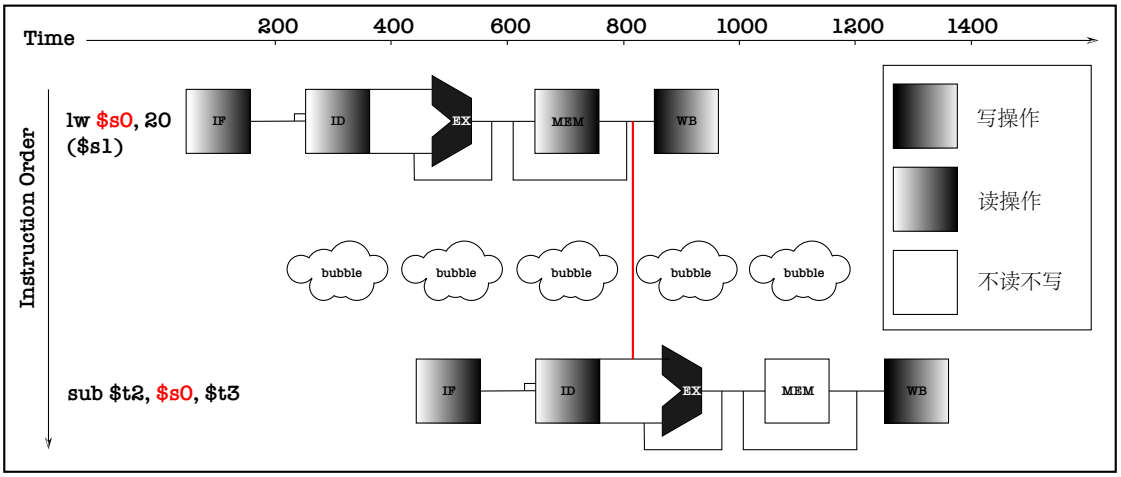


所以, 我们怎么解决这个问题呢?



通过在数据通路中增加额外的硬件支持, 使前一条指令的ALU一旦产生结果, 立即可以给后一条指令使用, 不必等到该数据存入指令中的目标寄存器. 上图中add指令执行后的$s0中的值作为sub指令执行的输入的旁路连接.

转发并不能避免所有的流水线阻塞, 如**取数-使用型数据冒险**, 具体如图:



取数-使用型流水线是一种特殊的数据冒险, 指当装载指令要取的数还没取出的时候其他指令就要使用的情况, 流水线不得不阻塞一个步骤 (正式的叫法是流水线阻塞, 又称气泡或者stall).

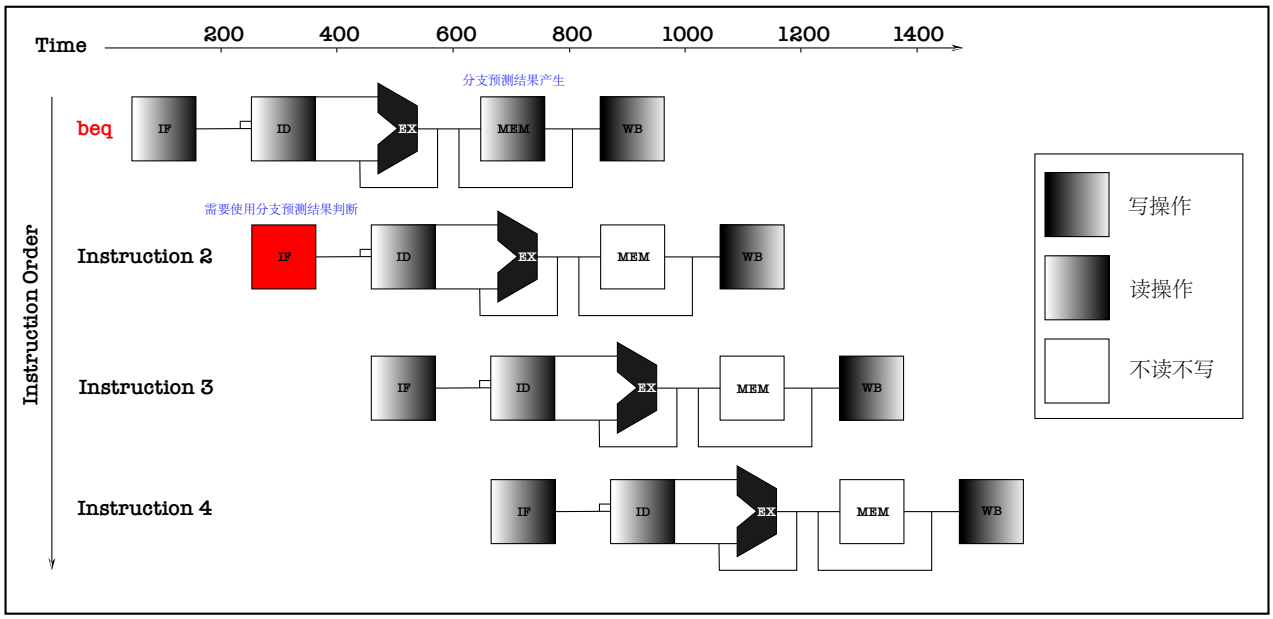
所以, 解决这个问题的唯一方法就是重新构建代码以避免阻塞. 如以下代码:

```
1  # BEFORE
2  lw $t1, 0($t0)
3  lw $t2, 4($t0)
4  add $t3, $t1, $t2 # stall
5  sw $t3, 12($t0)
6  lw $t4, 8($s0)
7  add $t5, $t1, $t4 # stall
8  sw $t5, 16($t0)
9  #AFTER
10 lw $t1, 0($t0)
```

```
11 lw $t2, 4($t0)
12 lw $t4, 8($t0)
13 add $t3, $t1, $t2
14 sw $t3, 12($t0)
15 add $t5, $t1, $t4
16 sw $t5, 16($t0)
```

控制冒险

根据流水线的规则, 当前指令处在ID的时候, 下一条指令在IF阶段. 但若当前指令是分支指令beq, 则下一条指令是哪一条取决于分支指令的执行结果. 如下图就是由分支引起的控制冒险:

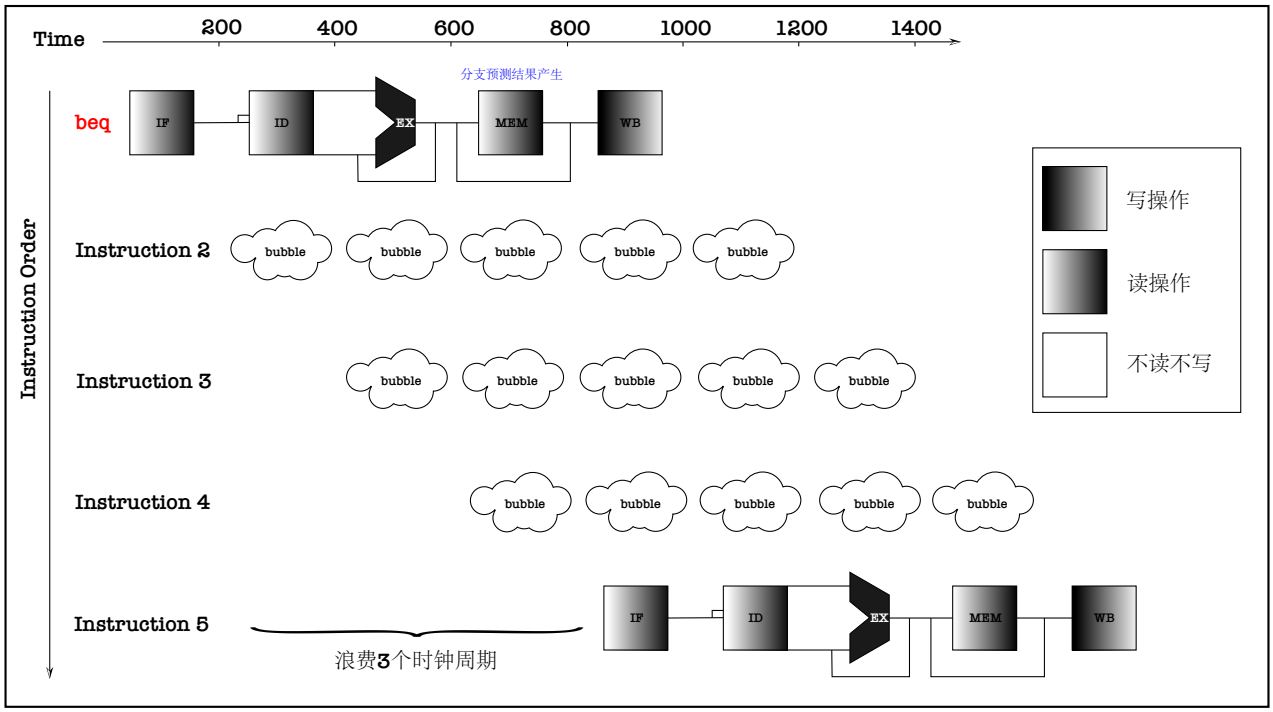


当Instruction2处于IF阶段时, 分支指令beq还处于ID阶段. beq的MEM阶段分支预测结果产生, 所以只有当beq指令执行到WB阶段的时候, 才能知道分支的结果.

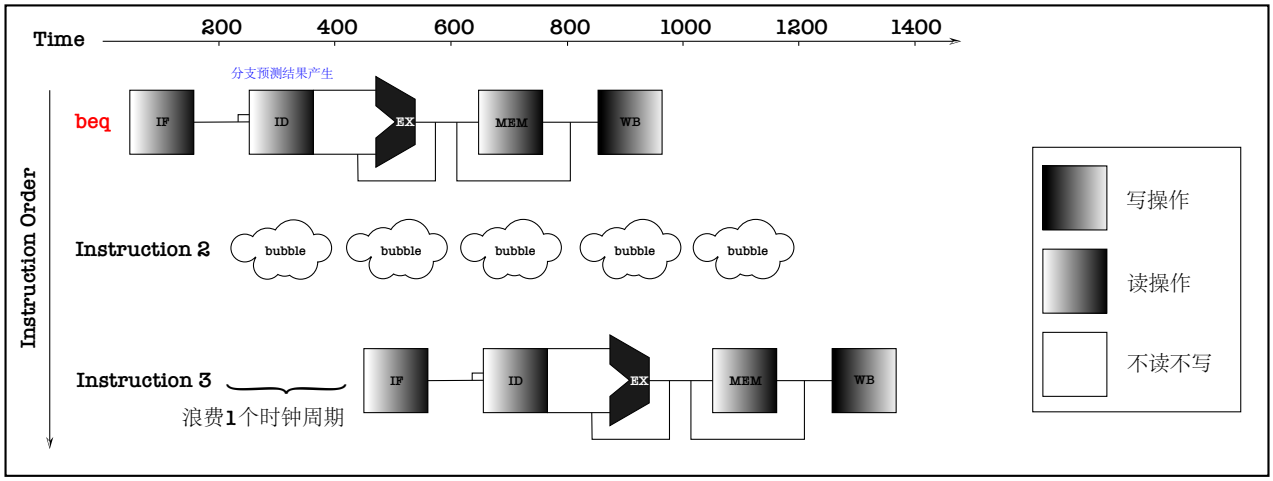
解决上述问题有两种方法:

1. 阻塞

   等待分支指令的输出, 再取指. 正常情况下, 如果分支结果在MEM阶段产生, 则需要阻塞三个周期, 如图:



正常流水线下CPI为1, 即每一个时钟周期内就有一个指令完成, 假设beq指令占所有指令的30%, 则CPI =1+0.3*3=1.9, 大大降低了性能. 为此, 设计者又在ID阶段增加了硬件, 使beq指令在ID阶段就能计算分支地址并更新PC, 如图:



这种情况下, CPI=1+0.3*1=1.3, 能改善性能, 但微乎其微.

2. 预测

总是预测分支未发生, 总是在分支指令beq后立刻取指, 只有当预测错误 (分支发生了) 才阻塞.