Components

2019年2月25日 16:12

电脑由Input device，Output device，Memory，Processor(Central Processor Unit)组成

在CPU中，有Datapath，Control，Cache Memory(暂时的存储器)

Datapath是用来处理数据的路径，可以看作是
ALU（一个对数据做运算的地方，执行与或非运
算）

control负责控制Datapath，输入
和输出还有内存

cache比内存快，比寄存器慢
cache（缓存）由SRAM（静态随机储存）组成，内存由DRAM（动态随机存储）组成

各个存储单元

2019年2月25日 14:01

对于储存器，存的东西越多，速度越慢，价格越便宜 寄存器封装在在CPU中

速度相比：

硬盘 < 内存 < cache < 寄存器

Volatile（main memory）：容易丢失存储数据的：寄存器，cache，内存

Non-volatile（secondary memory）：磁盘，闪存，DVD

Instruction set architecture(计算机结构指令集)：是硬件和底层软件之间的接口

性能的定义

2019年2月25日 14:10

Response time，也叫excution time，elapsed time，wall clock time：计算机完成某个任务所花的总时间
Throughput（吞吐率），也叫带宽：单位时间内完成的任务数量，假设有无穷多的work而且每一个核都要做（时序任务要求只能单核完成）

如果计算机仅仅运行一个程序
对于升级处理器，能同时提升响应时间和吞吐率
对于增加多个处理器，由于每一个处理器只能进行一个任务，不能一个任务交给多个处理器同时进行，所以响应时间不会变，但是吞吐率增加（如果运行多个程序，吞吐率增加同时响应时间减少）

性能 = 1 / 响应时间
X是Y的n倍快=>性能x / 性能y = 响应时间y / 响应时间x = n

CPU执行时间：做某一任务花在CPU上的时间
CPU执行时间 = 用户CPU时间（程序本身所花费的CPU时间） + 系统CPU时间（为执行程序花在操作系统上的时间）
计算机性能就包括了CPU执行，IO处理等等

一个程序的CPU执行时间 = 一个程序的CPU时钟周期数 * 时钟周期时间

由于不同指令所需的时间不同，故可认为
CPU时钟周期数 = 指令数 * 每条指令的平均时钟周期数

CPI：clock cycles per instruction

故 CPU时间 = 指令数 * CPI * 时钟周期时间

指令集体系结构影响了时钟频率，CPI和指令数
其他的因素只影响CPI和指令数（算法，编程语言，编译程序）

SPECratio值越大，代表性能越快

系统软件

2019年2月25日 19:14

系统软件分为编译器和操作系统

编译器是将高级语言转换成汇编语言

操作系统作用有：
处理基本的输入输出操作
分配外存和内存
为多个应用程序提供共享计算机资源的服务

要点

2019年2月25日 20:11

Clock rate = frequency太高，static power就高，pipeline也难做

benchmark：用于比较计算机性能的程序

在计算机内放多个core：用来更好的降温
当密度大时，用来降温的难度越大

功耗墙

2019年2月25日 20:15

现在用的集成电路技术是CMOS，主要功耗来源是动态功耗（开关晶体管所需要的能量）
动态功耗 = 负载电容 * 电压（2）* 开关频率

Dynamic power：用来计算，处理数据的能量 ，降频，降电压来降低能耗
Static power：与晶体管的数量，电压成正相关，指的是晶体管在关闭时会有泄漏电流存在，从而产生了消耗

如果一直减少电压，集体管就难以开关，就不能区分0和1了

输出Hello,world

2019年2月26日 10:18

```
.data
        str: .asciiz "Hello,World"

.text
.globl main
main:
        la $a0,str
        li $v0,4
        syscall
        li $v0,10
        syscall
```

```
.data
        str:  .asciiz "hello,world\n"
```

.data指的是数据域，是声明数据的地方

这里声明了一个asc的字符串，为str

la $a0,str

la指的是load address，即将str这个变量的地址存在a0中

li $v0,4

v0 是一个特殊的寄存器，里面通过赋值相应的编号，通过syscall指令来调用从而形成相应的操作，这里的4代表存储输出一个字符串。值得注意的是，如果代码是4，那么a0就应该存放字符串第一个字符的地址。在读取字符的过程中，系统按照给定的a0地址按照顺序读取，直到遇到结束符\0为止

syscall 系统调用，通过v0中存放的不同的值来调用不同的操作

汇编语言的细节

2019年2月26日 14:01

1byte = 8bit

对于寄存器（CPU中暂时用来存储的单位），位宽是4个byte
而对于内存，每一个储存单元是1个byte，一共有2^32个储存空间，编号为0~2^32 - 1

对于字符串，按照asc码顺序排列，每一个asc码都是1byte，即每一个字符都要占用内存的储存空间。如果寄存器要读取输出字符串，则需要将字符串的第一个字符的存储地址放到寄存器中，再读取

对于整数，存储方式有byte（1byte），word（4byte）

对于存储的地址，大小是4byte

对于每一个操作数也是有相应的地址，大小是4byte，在内存中以0开头的地方存储
而对于数据，在内存中以1开头的地方存储

demo讲解

2019年2月26日 23:26

```
1  .data
2          str:.asciiz "A"
3  .text
4  .globl main
5  main:
6          la $a0,str
7          lb $t0,($a0)
8          addi $t0,$t0,32
9          sb $t0,str
10         li $v0,4
11         syscall
12         li $v0,10
13         syscall
14
```

如图，这个程序是将A转换成小写输出

第六行讲的是将str的地址存到a0中
第七行指的是将a0中存放地址的值放入t0中
这行换成 lb $t0,str 也是可以的，此时就是将str中的第一个存储单元的值放到了t0中（变量名就指代第一个元素的值，与C很像）
这哈给换成 la $t0,str 指的是将str的存放的第一个地址放入t0中

第八行是将t0的值增加32，如果后面的值是一个变量（寄存器中的值，内存中的值），就不能立刻相加，就只能相加

第九行是将t0中的值存放到str中，将寄存器中的值放入内存中就用stroe，由于存放的是一个数（asc码），所以要用stroe byte
注意此时存放的str是放在str的第一个元素的位置
第九行也可以用 sb $t0,($a0) 这句话指的是将t0的值放在内存中，其地址根据a0存放的地址来寻找

单位换算

2019年4月12日 11:54

1字节(b) = 8比特(bit)
1kb = 1024b
1mb = 1024kb
1gb = 1024mb
1tb = 1024gb
1pb = 1024tb

1mm = 1000um
1um = 1000nm
1nm = 1000pm

冯诺伊曼机

2019年3月4日 14:13

任何逻辑，只要写成指令，就能执行
所有的code，data都存在内存中，CPU调用内存中的code和data

简单指令集特点：不能对memory直接操作，必须要先load进来
RISC的常用指令集：MIPS，ARM，PowerPC
多用于手机，平板（ARM ）

简单指令以及技巧

2019年3月4日 20:19

load：将数据从储存器复制到寄存器

相比于add，addi效率更高（相当于立即数，就是相加一个确定的值）

可以通过无符号数来判断数组是否越界（如果真正的索引值是一个负数，那么他肯定比正数大）

例子

2019年3月4日 20:21

编译以下语句：

g = h + A[8];

$s1,$s2存放g和h的值

A数组的基址存放在$s3中

解：

lw $t0,8($s3) // 根据$s3中存放的地址往右偏移8个单元，然后将这个地址里面的值存入寄存器中

add $s1,$s2,$t0 //将t0中的值和s2中的值相加，放入s1中

将C语句代码转换成汇编：

```
while(save[i] == k){
i++;}
```

假设i和k存在$s3和$s5中，save数组的基址存在$s6中

Loop: sll $t1,$s3,2 //由于要计算偏移量，所以要乘4（每四个byte存一个数字）

add $t1,$t1,$s6 //由于计算偏移量时括号左边不能是数字，所以只能先计算出第i个元素的地址，再通过($t1)找到第i个元素

```
bne $s5,0($t1) Exit
addi $s3,$s3,1
j Loop
Exit:
```

执行C语句的代码：

```
if(i == j){
f = g + h;
}
else{
f = g - h;
}
```

转换成汇编，假设f,g,h,i,j都存在于$s0~$s4中

bne $s3,$s4,Else //如果i不等于j则跳转到else语句块

add $s0,$s1,$s2 //代表true的语句块

j Exit //代表跳出这个判断的语句

Else: sub $s0,$s1,$s2 //标签作用，用来跳转语句

Exit:

要点

2019年3月4日 20:36

MIPS是按照字节编址的，在储存器（内存）中，每个字（word）的起始地址必须是4的倍数，这叫字节限制
所以说通过偏移量来寻址时要乘4（内存中一个单元是1个字节（byte））

加括号的只有加载内存里面的东西的时候才用

内存的地址是通过byte存储的

在MIPS指令集中，总共有32个寄存器，所以记录一个寄存器的地址需要5个bit

直接访问内存是很慢的

对于指令，都是用32bit的二进制数字保存，并且存放到内存中
在程序运行的过程中，先在内存中取出一条指令，根据指令再抽取相应的编号的数据，做相应的操作

越好的编译器用的寄存器越多，因为访问到内存的时间是非常慢的

Jump register 只有在调用函数的时候才用

常见伪代码

2019年3月4日 20:39

在进行move操作时，机器不是直接对其进行移动
而是采用了add指令（与一个恒为零的寄存器相加，最终得到的值放进目标寄存器中）

加法运算

2019年3月4日 20:47

将16位的二进制数转换成32位的二进制数：

将这16位放到低位的32位上，对于另外16位则符号位（最高有效位）以复制的方式填满新数

对于无符号数的二进制转换成十进制方法：

$a_{31} \times 2^{31} + a_{30} \times 2^{30} + \ldots + a_1 \times 2^1 + a_0 \times 2^0$

有符号数二进制化十进制方法：

$-a_{31} \times 2^{31} + a_{30} \times 2^{30} + \ldots + a_1 \times 2^1 + a_0 \times 2^0$

机器语言

2019年3月4日 20:57

对于相加操作：
add $t0,$t1,$t2
这是一个R类型的指令，这个指令的机器语言分为五个字段
其机器语言为：
0 17 18 8 0 32
第一段是operation code，告诉机器要做什么类型（这个是R类型）的操作（即接下来的机器码如何去读）
第二段指代$s1的寄存器号
第三段指代$s2的寄存器号
第四个字段指代$s0的寄存器号
第五个字段在这个类型中代表的是移位的多少
第六个字段指代R类型中具体的是哪个操作

R-type instruction add $t0, $s1, $s2

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct
opcode	source	source	dest	shift amt	function

I-type instruction lw $t0, 32($s3)

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	constant

对于装载字的操作 lw $t0,32($s3)
就是一个I类型的操作
值得注意的是对于16位的常数就是规定了偏移量，
由于有16个位宽，所以其偏移的范围是$\pm 2^{15}$

控制逻辑

2019年3月4日 23:03

控制逻辑有if-else,while,switch
对于if-else或者while语句块，都是用bne或者beq来判断是否相等，通过slt来判断是否小于某个值
对于switch语句块，可以转换成if-else语句块。不过更好的解决方案是使用转移表，通过jr（jump register）指令来跳转到转移表

对于如果想要在while语句中产生小于，就用slt（set on less than）
slt $t0,$t3,$t4代表如果$s3 < $s4，$t0的值就是1，否则就是0，然后再通过与$zero来比较是不是0（原本这个可以通过一个指令来完成，但就会提高CPI或延长时钟周期事件，所以，以简洁的方式来考虑就将原本的一条指令拆分成两个）

Lab

2019年3月5日 10:47

对于有符号数，如果31个bit位相加影响到符号位，就认为产生溢出

调用函数的过程

2019年3月11日 14:34

关键点：$ra,$sp,存重要的值，返回值，jal，jr

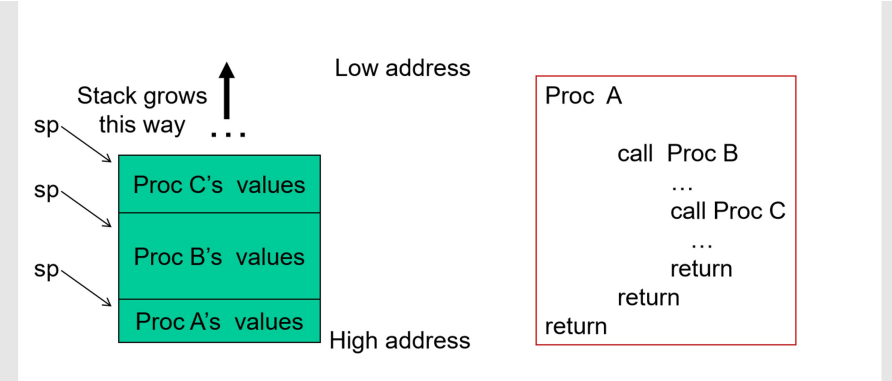
Caller指的是对内部函数的调用者，相当于主函数
Callee指的是被调用的函数
是Caller 调用Callee

Callee是通过某个标签的形式来接受调用
通过jal(jump and link)这个指令来调用目标函数。这个指令的意思是：将程序执行的下一个指令变成内部函数的下一条指令；同时将外部函数的下一条指令的地址存到寄存器$ra（return address）中。当内部函数执行完成，计算机通过$ra里面的值加载出外部函数的下一条指令。
在内部函数中，如果内部函数执行完毕，机器就要在$ra中找到下一步要执行的指令
所以说，jal这个指令是在外部函数中用来调用内部函数的
jr(jump register)这个指令时在内部函数进行完后采取的跳转操作
函数执行完后再调用ra的值，执行ra存储地址的值

在调用函数过程中，不可避免的就是使用参数和返回值
一般的，我们通过实现规定寄存器$a0到$a3的值来完成，因此，在内部函数调用之前，我们都要实现将想要传递的参数传到寄存器中
而对于返回值，一般的，我们都将返回值放在$v0或者$v1中，所以一个函数最多有两个返回值
如果参数个数大于4个，一个办法是将参数放到内存中，另外一个方法是调用现有的寄存器

通过jr(jump register)这个指令加上函数的地址，程序执行函数内的指令（通过一定的算法，将函数的32位地址缩减到26位，对于beq指令，就是缩减到16位）

对于函数调用，我们可以想象是一个间谍在执行任务
每调用一个内部函数，当这个内部函数执行完毕后，除了将正确的结果带回出发地，还必须要将之前所有改变的寄存器的值还原回来（这样做是防止混淆）
而将需要保留的寄存器的值，我们通过$sp(stack pointer)来实现
所有用到过的值都要保存到内存内，改变时重新放回去
由于函数要执行相应的步骤，需要占用到相关的寄存器，所以这些原本寄存器里面的值需要放到内存中，当函数执行完毕后再将原来的内存中保留寄存器的值放到原来对应的寄存器中，通过一个叫$sp（stack pointer）的寄存器的地址来存放参数的个数和值，如果需要存储的个数是3，那么$sp就下移12个byte，每四个byte存储一个数字
对于$sp，都是从上往下存储，所以说，如果在函数运行过程中有重要参数存储，那就先$sp减去所需要的存储空间，然后再将需要的参数一个一个存进去
人为规定，规定$t0,$t1可以不用改回来，$s0,$s1是必须要改回来的（s代指save，t代指暂时的）



函数调用过程和栈存储过程如图

Memory Layout

2019年3月11日 15:37

对于一整个内存，人为规定

Stack pointer 从上往下走（高地址往低地址走）

heap从下往上走（从低地址往高地址走）

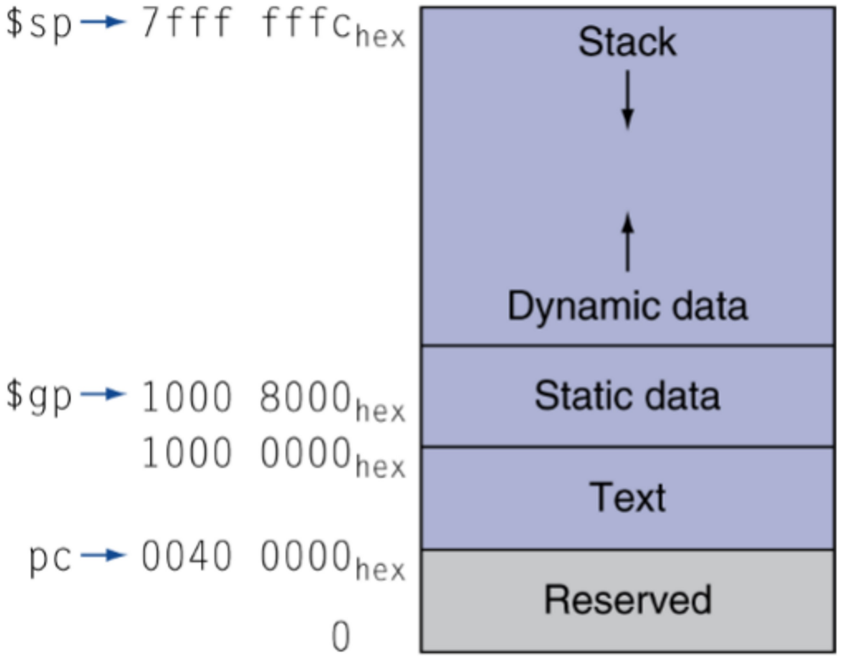
当都碰到了，内存就爆了

Stack存的是函数调用中的需要保留的值

heap存的是动态数据（dynamic data），相当于malloc和new关键字

对于static data，里面存的是汇编中.data定义的数据

Text就存的是各个指令



动态变量位于过程中，过程退出时就失效（相当于使用了delete）

对于静态变量，在进入和推出的整个过程都存在着

对于Reserved，是系统预留的执行空间，pc存的是当前指令的地址

$gp（global pointer）指的是全局指针，指向的是静态变量中的数据

其他细节

2019年3月13日 16:13

在程序运行中，使用一个寄存器保存当前运行的指令地址是十分必要的，我们叫程序计数器（Program counter，简称PC） 负责存储调用哪一个程序，而jal指令就是将PC + 4保存在$ra中，方便指令调用

通过jal(jump and link)这个指令来调用目标函数。这个指令的意思是：将程序执行的下一个指令变成内部函数的下一条指令；同时将外部函数的下一条指令的地址存到寄存器$ra（return address）中。

在函数调用过程中，使用寄存器来传递参数
$a0 - $a3: 传递参数的寄存器
$v0 - $v1: 传递返回值
$ra: 存储的是下一个函数的地址

伪指令

2019年3月13日 16:27

blt $s0, $s1, Label ■ If s0<s1, jump to Label
bgt $s0, $s1, Label ■ If s0<s1, jump to Label
ble $s0, $s1, Label ■ If s0<=s1, jump to Label
beqz $s0, Label ■ If s0==0, jump to Label
li $t0, 5 ■ Load immediate, t0 = 5
Move $t0, $s0 ■ t0 = s0

对于左边的式子，都是伪指令，即在实际操作中是不会设置这么多的指令（对于精简指令集，增加许多复杂的指令会增加CPI，从而影响运行速度）
通过增加伪指令，方便了书写
实际上这些伪指令都是通过几个简单指令组合而成

slt $s2, $s0, $s1
bne $s2, $zero, Label
对于blt,bgt,ble,beqz，都是通过上述的指令组合而成
对于li $t0,5 Move $t0,$s0
是通过与add $t0,$t0,5 和 addi $t0,$s0,$zero来实现的

递归的讲解

2019年3月13日 16:49

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

对于函数的开始，首先要将需要的参数存入栈中，然后才能进行操作

当n<1时，返回1，此时$v0就是1，不过记得要将存储的参数还原，而且，记得返回参数

当n>=1时，返回n*fact(n-1)，由于现在不知道fact(n-1)的具体值，所以再调用一次fact从而产生fact(n-1)，记得此时要将参数调整。当进行了一次调用后，将返回值乘上n即可，记得此时要将储存的参数还原，而且记得返回参数

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

        slti    $t0, $a0, 1
        beq     $t0, $zero, L1
        //和1进行比较

        addi  $v0, $zero, 1
        addi  $sp, $sp, 8
        jr    $ra
        //返回act(1)
L1:
        addi    $a0, $a0, -1
        jal     fact
        //返回fact(n-1)

        lw      $a0, 0($sp)
        lw      $ra, 4($sp)
        addi    $sp, $sp, 8

        mul     $v0, $a0, $v0
        jr      $ra
```

寻址方式

2019年3月18日 14:09

立即寻址
寄存器寻址
Base/Displacement addressing
PC-relative addressing
跳转寻址

移位操作是r-format
Operation code

立即寻址

2019年3月20日 16:34

典型的操作例如：addi，subi，li，被分成4个部分，i-format

如果数字大于2^16，就用lui和ori指令分成两个方式存储

op指的是操作码，immediate指的是立即数

rs指的是目标寄存器，rt指的是初始的寄存器

对于：addi $1,$2,100	翻译成指令就是：rt=$1,rs=$2

所以说rs和rt都是反过来的

根据指令可见立即数只能加16位，如果想要加的立即数大于16位，就只能使用lui（load upper immediate）指令和ori指令（or immediate）

lui指的是将这16位放到较高的16位上相加（也是相当于or命令）

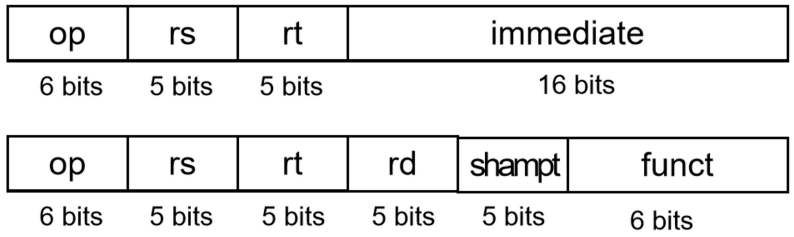
ori指的是将这个低16位相加（由于是和0相加就成了or命令）

通过这样的方式就能产生立即数

寄存器寻址和基址寻址

2019年3月20日 16:52

这两种寻址方式很广泛，指的是：
寄存器寻址是但凡有包含有寄存器的指令就用寄存器寻址，就是说通过寄存器的编号来表达寄存器的地址
基址寻址是寄存器存储字符串（数组）的基址，通过偏移量（或者循环）来完成寻址
下面两种都是寄存器寻址和基址寻址



对于rs，rt，rd就是寄存器寻址
其中，对于三个寄存器的op（操作数）是0，通过function code来细分

跳转寻址

2019年3月20日 16:52

跳转指令如下：对于jump，jump_and_link都是通过以下的表示方式

j Label

op	address
6 bits	26 bits

先找后面28bit，最开始的2个bit，肯定是00（指令的基址都是4的倍数），所以舍去。对于最开始的两个bit基本没啥信息量，就用PC一开始的两位计算即可。但是，如果跳到的标签真的太远了，即用两个标签来跳从而代替一条

而将32位压缩成26位的方法：

取得地址是后面28bit，而末尾的2个bit，肯定是00（指令的基址都是4的倍数），所以舍去。

对于最开始的四个bit基本没啥信息量，就用PC一开始的两位计算即可。

当汇编器检测到：

跳转的标签真的太远了，就会用自动使用两个标签来跳从而代替一条，从而增大了跳转范围

```
        beq $s0,$s1, L1
                ↓
        bne $s0,$s1, L2
        j L1
L2:     …
```

分支寻址（PC相对寻址）

2019年3月20日 16:56

这种指令是i-type，适用于：

bne $s1,$s2,Label

其中，跳转的地址就存储到16bit中，而对于存储的方式很巧妙

对Label标签的存储，实际上存的是相对于PC的差

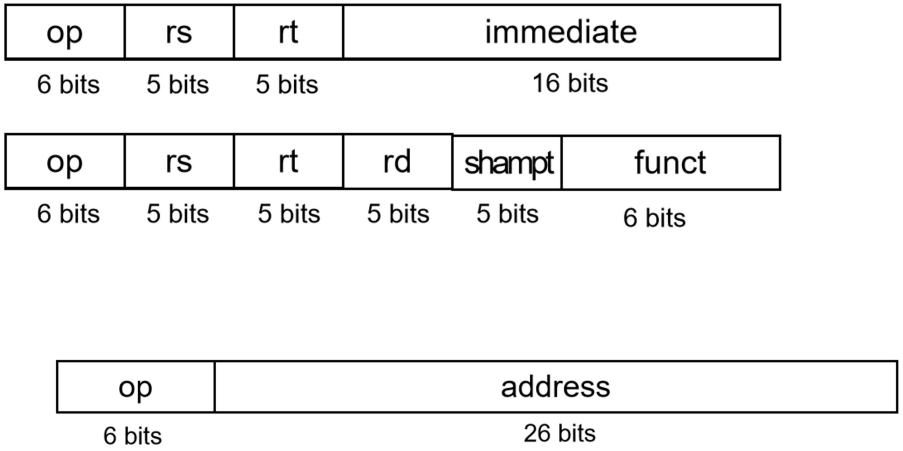
在汇编器处理时，确定了这个label和这条指令在第几行（当然，这个跳转指令必须是在同一个函数内部），而这16bit存的是这两条指令的相对差

当想要跳转时，用当前的指针PC（PC指针一次就是增加4） + address（这个存在指令中） * 4即可

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

指令的表示方式

2019年3月20日 17:18



最上面的那个指的是i-type，一般的涉及到两个寄存器操作还有立即数的就是i-type

中间的指的是r-type，大多应用于左移右移操作和三个寄存器的操作

最下面的是j-type，应用于jump操作还有jal（jump and link）

程序执行步骤

2019年3月18日 14:48

高级程序先经过编译器，转换成汇编语言
然后通过汇编器，转换成机器能看懂的二进制文件
再通过链接器将目标库和源文件链接到一起，变成可执行的文件
再通过加载器储存到储存器中，等待执行

汇编器是将各种指令拆分，将伪代码拆分，转
换成实际的代码
同时，也会将各种指令转换成一个符号表，方
便链接器转换成实际的地址

对于链接器，是将翻译后的两个汇编文件变成一个汇编文件
原来的汇编文件可以看成一个表，上面有数据段（放静态数据的地方），符号表（存各种标记Label的位置），重定位地址信息（记录lw，sw等等指令是从内存中的哪个地方读取的），代码段（存代码的地方）
可执行汇编文件就只有数据段和代码段了
所以说：在链接前都是不知道绝对地址的，只是知道一个相对地址

加载器就是将代码加载到内存中，方便读取

其他细节

2019年3月20日 16:15

$gp(global pointer)是用来指向动态变量区的指针
$fp(frame pointer)过程帧，是只想静态变量区的头，但是一般都使用$sp来避免对$fp的使用

重要的约定

2019年3月20日 16:19

加大大概率事件（设置$zero）

ASC码和unicode

2019年3月20日 16:26

ASC码是一开始的字符集，有128个，占据8个bit，只能表示英文
Unicode是之后进行汉字扩充的字符集，占据16个bit，对应在mips就是一个“half word”可以表示中文
java是用unicode存储，C使用ASC码存储

对于java的字符串变量，程序在运行过程中，会将2个halfword（就是两个short）压缩成一个字（word）存入寄存器中
而C的字符数组，会每4个字节（byte）压缩成一个字（word）

要点

2019年3月25日 14:04

对于饱和操作（saturating operation）指的是当检测到数字要溢出时，令其数字大小等于边界的数字而达到不溢出的目的
具体例子是：
处理图片时整体调亮：提供了addv（addvector）指令，使得图片上每一个像素点的rgb三个值都加上相同的值
整体调亮时，当要超过边界时，令其等于边界，就是指saturating operation

商：先把被除数放到remainder
先相减，看看被除数是否为负，如果为负，商就是0，并且加回来；如果为正，商就是1
移动一位
如果四位的数相除，就移动五次

ALU有and，or，add，什么都不输出
两个都算出结果，然后经过控制来判断是否输出

计算机中加减法及其要点

2019年3月25日 20:18

对于加法，计算机中就直接相加就行了

对于减法，减数先取反，然后加一，再与被减数相加，就变成了减法

当加减法超过边界时就会产生溢出

有一些语言对于加减法出现溢出时不会报错（如C，Java）

处理溢出：

addu：忽略溢出，不报错

add操作就会报错

addui：addui $s0,$s0,-1不会报错

有些语言遇见溢出不会报错（C，java）

处理异常：在EPC寄存器内处理

当报错时，EPC = PC，将ra放到相应的标签，然后跳转

对于负数，通过基于2的补码来表示

异常发生及其处理

2019年3月26日 21:14

异常产生方式有许多种，比如：有符号数越界，以不对齐的方式读取数据等等
越界这个异常，如果是有符号数就会报错，但是对于无符号数就不会报错
对于除数为零，如果是直接使用div $a0,$a0,0是不会报错的，但是如果使用div $a0,$a0,$s0($s0 = 0)就会检测然后报错

和异常相关的语句有：
Conditiaonal trap：通过条件判断来决定是否进入异常（常用语句有tne，teq等等）
break：直接进入异常状态
eret：指的是异常处理完成之后回到相应程序进行的地方，从而继续进行程序
对于eret指令，返回时读取的是epc内的地址，所以说，需要先将epc的地址读进来，增加4的倍数跳转到某个指令，再将希望的地址存进去，才能用eret跳转到期望的地址

和异常相关的寄存器有：
EPC这个寄存器用来存储异常发生的指令地址，status用来检测每一步发生的状态
cause是用来记录异常发生的原因，vaddr用来存储发生异常时该寄存器的值
这四个寄存器都是在Coproc0这个类别中，使用mfc0指令来读取相应的数据
还有$k0,$k1这两个寄存器，在运行异常时用来暂存数据的

对于异常，都存放于内核中
而在内核运行的程序中，其基址是0x80000180，所以当声明异常时都要加上这个基址，代表其起始地址是0x80000180。而且，不管是代码段还是数据段，只要其中一个先出现在编辑的文本中就要将其声明基址

.kdata	.ktext

.kdata和.ktext指的是内核中代码段和程序段的声明方式

对齐方式及其处理

2019年3月26日 21:16

对于byte，如果读取的基址不是2的倍数就会报错
对于word，如果读取的基址不是4的倍数就会报错
而对齐的最简单方式，使用.align 2，代表下面的数据值的基址自动是2的2次方（也就是4）的倍数
另外，如果是用了.align 0就是自动识别下面的数据类型来进行对齐

例如：

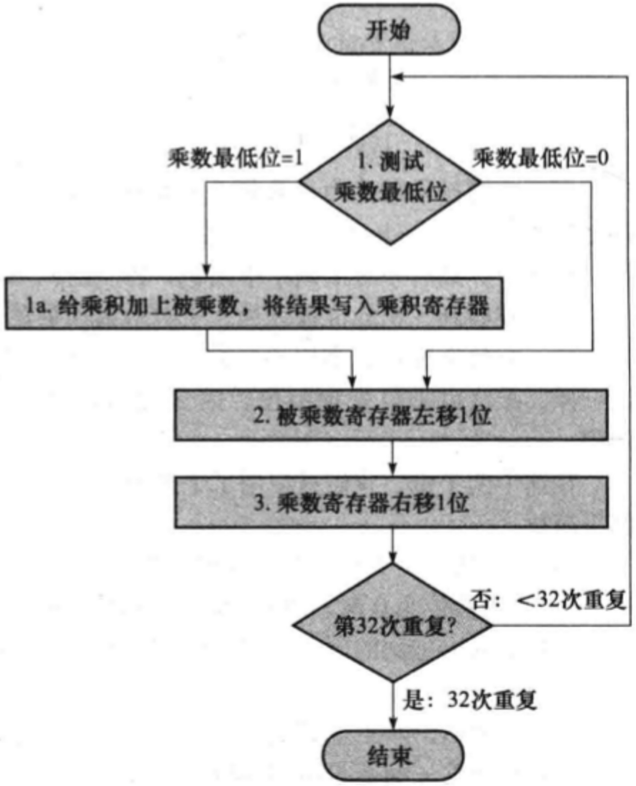
```
.data
str: .asciiz "data is:"
.align 2
bs: .byte 1:10
ws: .word 2:10
```

对于这段声明中align就代表自动对齐（str占用了9个格子，因此如果没有对齐这句话，如果将bs
加载进来就会报错）

乘法表示及其电路图

2019年3月26日 10:37

以下为乘法器的基本原理



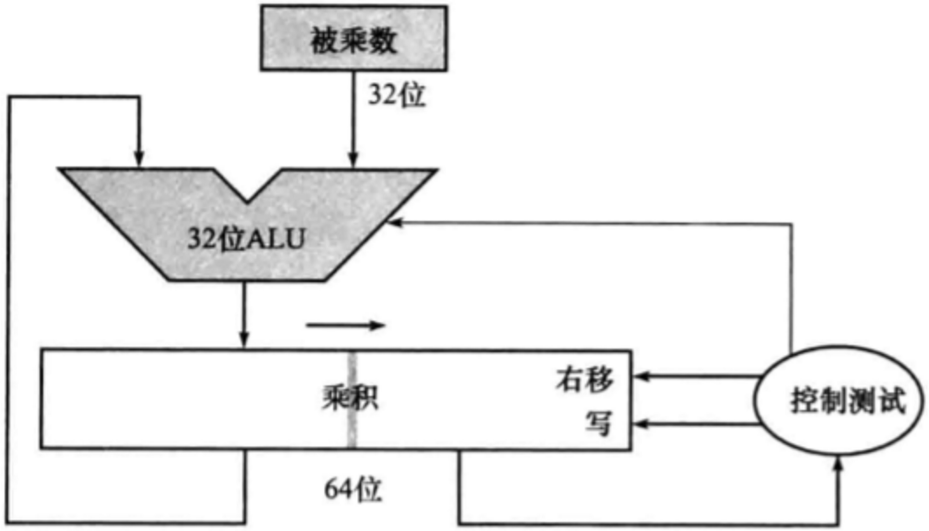
乘法的方式：使用乘法表

优化：相加的时候使用原来寄存器的位数，相加

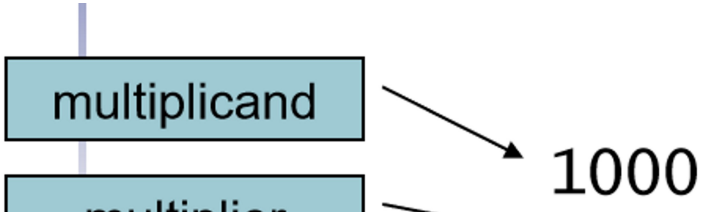
更快的乘法器：使用并行处理

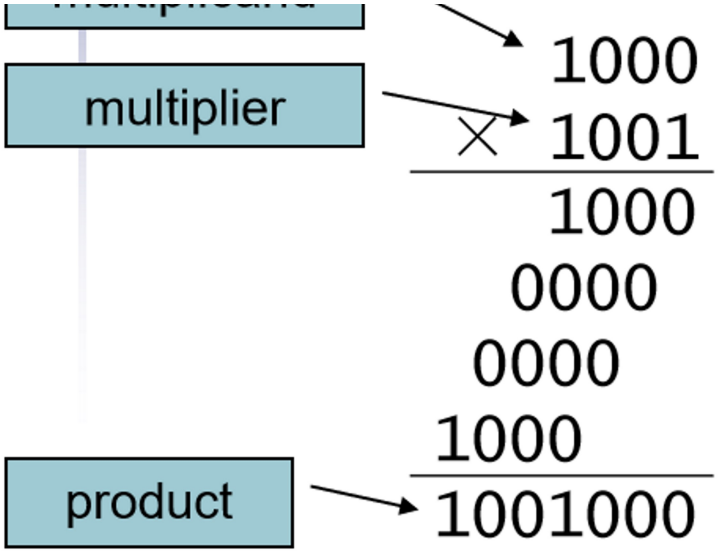
最后一位

可以看到，初始的乘法器中，被乘数其实只有32位，却搞成了64位，说明其他的32位都是0（这32个0仅仅是用来充当移位）；而且，一开始，乘积的最高32位也是被闲置的（在前几步时，被乘数是不会移动这么多步的），造成了空间的浪费。所以，提出了优化的方案

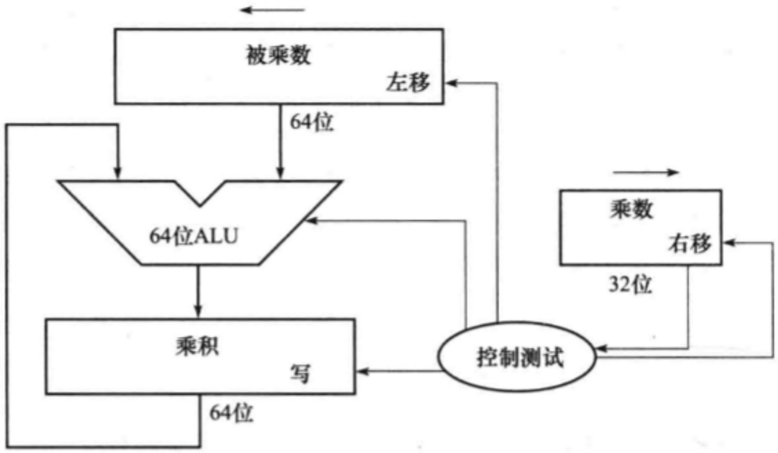


对于乘法的运算，关键是使用乘法表进行运算





可以看到，在二进制中，乘法就是多个数字相加，而数字的数值只有两种，要么是被乘数要么是零，而每一次坐加法都要往前提一位，这个可以通过进位来实现，所以加法的电路如下所示



初始情况，64位的ALU对应乘积，被乘数在最低32位，乘数完整的放到寄存器中

对于每一次循环，

1. 乘数做右移操作，将最低位的数字弹出

在优化的方案中，乘数放到了乘积的低32位，32位ALU永远对应乘积的高32位

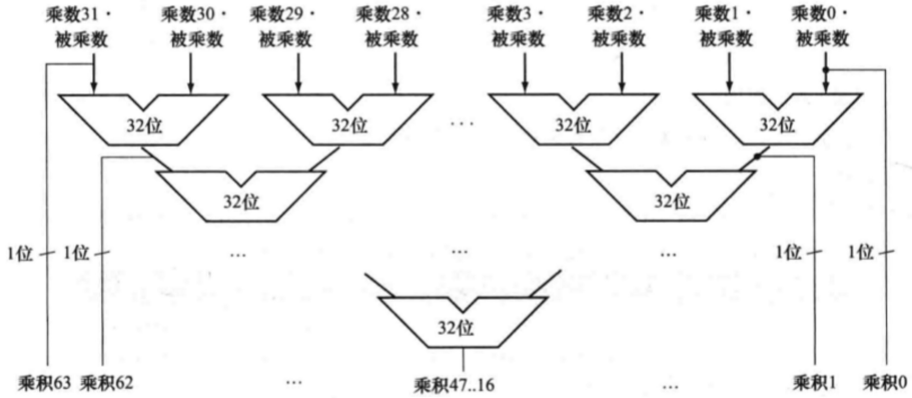
对于每一次循环：

1. 控制电路获取乘积最低位的信号来决定是否相加
2. 如果相加，32位的ALU将结果输入到乘积的高32位
3. 乘积右移（这个步骤代替了之前被乘数左移和乘积右移的操作）
4. 执行以上操作32次，直到结束

更快的乘法器（并行乘法器）

这种乘法器就是直接模拟加法的过程

也就是将32个数字通过移位来相加，从而得到结果



对于除了最后一次加法，高位的加数将最高位取出直接作为结果输出（也就是32位变成31位），低位的加数不变，两者经过ALU，产生结果，并且作为新的加数

对于最后一次加法，左边是高32位，右边是低32位（相对于左边总体只高了一位），此时左边去掉最高位，右边去掉最

的放到高位部中

对于每一次循环，

1. 乘数做右移操作，将最低位的数字弹出
2. 弹出的数字经过控制系统，判断是否相加，如果传入的数字是0，代表相加的数为0，就不加；如果是1，就加上被乘数
3. 被乘数的每一位向左移，代表进高的一位

对于最后一次加法，左边是高32位，右边是低32位（相对于左边总体只高了一位），此时左边去掉最高位，右边去掉最低位（最高位和最低位直接输出），两者相加，得到的结果就是乘积的中间32位（两边的32位在之前并行加法时直接输出了）

这种并行的乘法器可以通过特殊的设计，抵御进位的情况。就是说，如果有进位（专指最高位进位），则之前输出的那一位就要进一，所以先输出的旁边32位不一定就是正确结果，还需要进一步验证

除法表示及其电路图

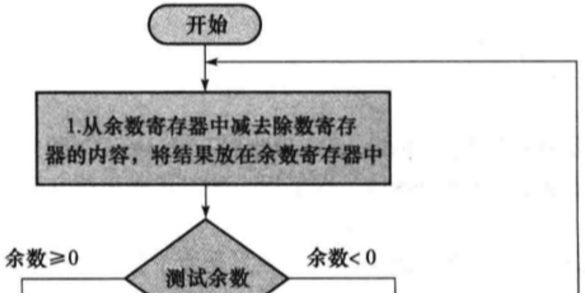
2019年3月27日 14:19

除法的表示也是用除法列出表格的形式来表示

$$
\begin{array}{rl}
 & 1001_{10} \quad \text{商} \\
\text{除数}\ 1000_{10} & \overline{)1001010_{10}} \quad \text{被除数} \\
 & \underline{-1000} \\
 & \quad 10 \\
 & \quad 101 \\
 & \quad 1010 \\
 & \quad \underline{-1000} \\
 & \quad\ \ 10_{10} \quad \text{余数}
\end{array}
$$

通过除法运算，我们发现除数都是一开始和最高位对齐，如果发现这一位不能减（相减结果为负），就换到下一位；如果能减，就相减，然后继续做上述动作直到产生的余数不再大于除数为止

以下为流程图

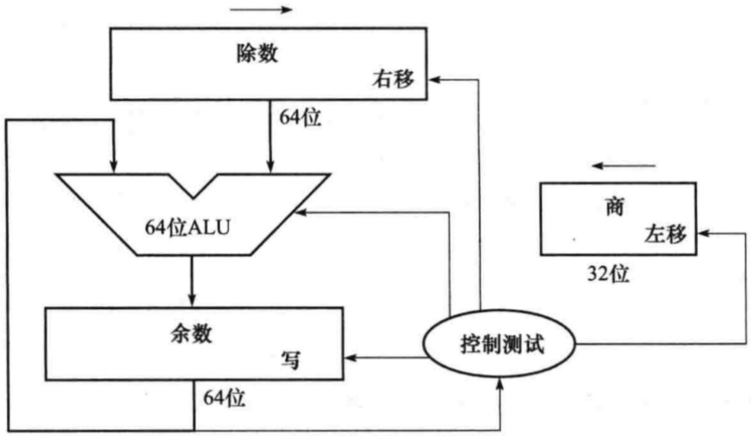


通过上述算法我们就推导出一般的除法算法

由于除法是先和被除数最高位依次比较，所以对于被除数，需要扩展其位数，也就是将32位寄存器扩展成64位寄存器

而对于除数，也是需要通过移位来保证数值大小

所以说，原来保存除数和被除数的寄存器，需要都扩展到64位，下图为除法器电路

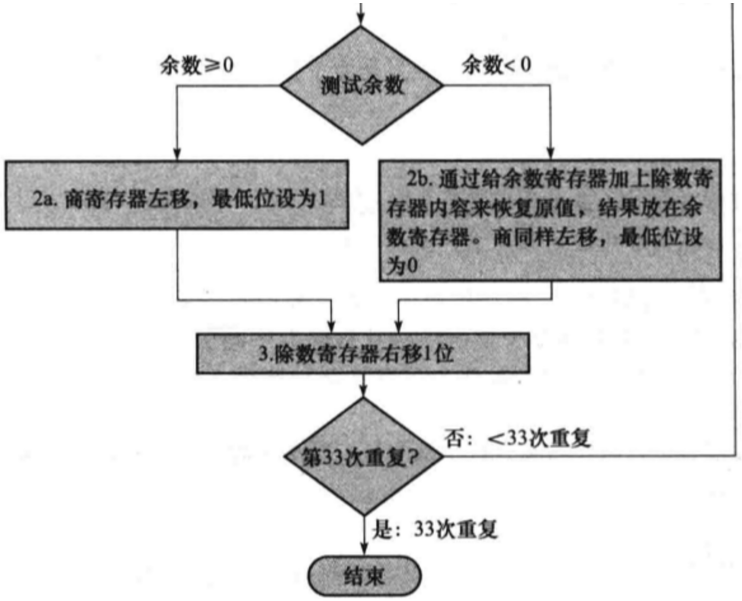


如图所示，一开始，除数放在最高的32个有效位上

被除数放在余数的最低32位上

对于每一次循环

1. 将余数减除数，如果得出来的值是负数，这个位置的商就是0，就要将余数加回来；如果得到的是正数，这个位置的商就是1，不需要加回来



商就是0，就要将余数加回来；如果得到的是正数，这个位置的商就是1，不需要加回来

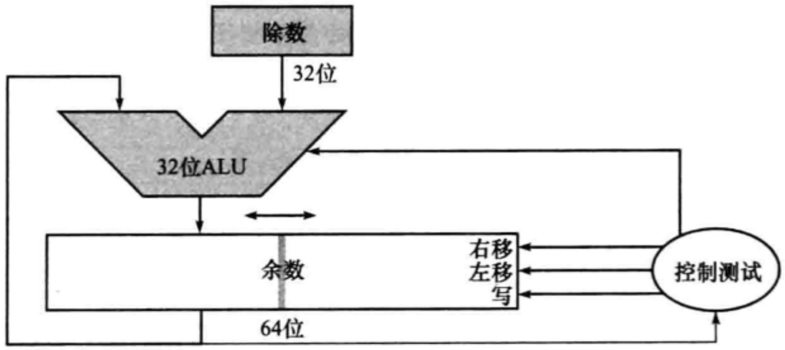
2. 除数右移，代表每次的退位
3. 商左移，给新来的结果腾出空间

对于改进的思路是：

对于除数的移位操作，真正需要移位的值实际上不变，所以变成32位寄存器就好

对余数的移位操作，最后得到的移位结果只有32位，而寄存器用了64位，很不划算，于是打算将商和余数混合在一起

改进的电路如图



一开始，被除数放到余数的低32位，32位ALU和余数的最高位对齐

在每一次循环中，

1. ALU先取余数的最高32位进行计算（余数减除数），如果结果大于等于零，将结果写进余数中（从右边写入）；如果结果小于零，保留结果
2. 整个余数向左移
3. 承接1，如果结果大于等于零，将1写进最低位的余数；如果小于零，将0写进余数中

重复以上步骤32次

则余数最高的32位是余数，最低32位是商

浮点数表示

2019年4月1日 14:26

在计算机中，浮点数的表示方法是：

$xxxx_2 \times 2^{yyyy}$

相当于使用科学记数法来表示

浮点数中float有32bit，double有64bit

分为3个部分，符号位，指数域（exponent），尾数域（fraction）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	指数								尾数																						
1位	8位								23位																						

一般浮点数表示为这样的形式：

$$(-1)^s \times F \times 2^E$$

对于指数域，存的就是指数的大小

指数的大小有正有负，如果超出指数域能表达的上界，叫上溢

超出下界，叫下溢

例如对于单精度float而言，指数位有8个，说明能表示2^8个数字。但是有两个数字不能用（全零和全一），也就是说，能表示的范围是从$-2^7+2 \sim 2^7-1$

对于浮点数，采用偏阶（bias）的方式来储存指数位，就是在实际存储中，增加某个数来达到储存的目的。对于float类型，就是需要加上127，使得最小的指数（-126）为正，从而储存

其中，由于是科学计数法，所以说除了0，前面的整数部分就必定是非零

相对精度

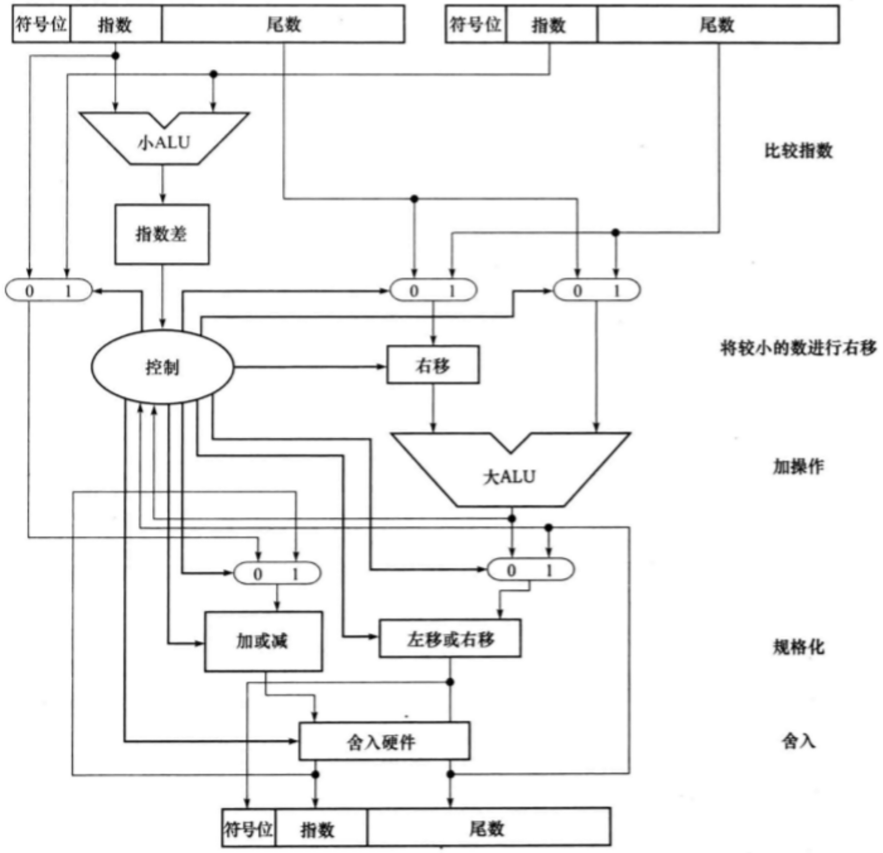
对于尾数域，由于使用科学记数法，可以知道尾数中的整数部分一定不是0（0除外），而在二进制中的整数部分就只有1，所以说整数部分就不计入尾数域中，只是将小数部分放在尾数域中

通过尾数域得知：浮点数计算机不能表示0，计算机不能表示一个很小的数（指的是无限接近于0），也不能表示一个很大的数

浮点数加法

2019年4月1日 14:49

浮点数加法过程有点复杂，电路图如下：



浮点数加法：

通过移位，然后相加，相加的结果再移位

将最大的exp转换成小的exp

首先，对于两个数字相加，先看指数部分是不是相同的，如果相同，进行下一步；如果不相同，将小的指数转换成较大的指数，同时尾数移位

第二步，对于尾数，此时进行相加或者相减操作（电路图有一点不准确的是符号位也是要和其他元件相同）

第三步，根据结果，转化成科学记数法的形式

第四步，转换完成时，判断指数域是否越界，如果越界，报错

第五步，对于尾数域，进行适当的舍入

第六步，判断结果是不是符合科学记数法（舍入的操作可能产生了一个非科学记数法的数字），如果不是进行第三步直到结果正确

精确度

2019年4月3日 16:17

在浮点数运算过程中，为了使结果更加精确，计算时的末尾（右侧）会多加两个位
其中（从左到右）首位叫保护位（guard），用来提高舍入精度
第二位叫舍入位（round），用来得到最精确的数字
粘贴位（sticky bit）用于舍入的位，当舍入位右边有非零的数时粘贴位为1
粘贴位用来区别0.50……1和0.50……0的区别

尾数最低位（unit in the last place），用来判断误差范围的
IEEE 754 保证误差小于半个ulp

eg：将2.56和2.34×10^2相加，假设有3位十进制尾数，使用保护位和舍入位将其舍入到只有三位尾数

$2.56 = 0.0256 \times 10^2$

2.3400
0.0256

2.3656

由于只有3位尾数可以保留，所以5就是保护位，6就是舍入位，根据四舍五入的原则，结果是2.37，更加的靠近答案

基本的逻辑

2019年4月8日 14:00

对于有偏移量的存储，先和偏移量相加，然后再存储

R-type指令
需要的部分：PC，registers，ALU，instruction memory
对于add操作
registers中RegWrite永远是1
registers输出的数据中，放入ALU（其中有operation code，存放哪一个操作类型）直接出结果

对于load，store操作
需要的：PC，registers，ALU（计算偏移量），instruction memory
registers中RegWrite永远是1
对于sw，MemWrite永远是1

ALU中zero是指判断结果是不是zero，用于bne和beq

J-type指令
需要：PC，register，ALU
记得是PC+4
Sign-extend：当是i-format时就将16位扩展成32位

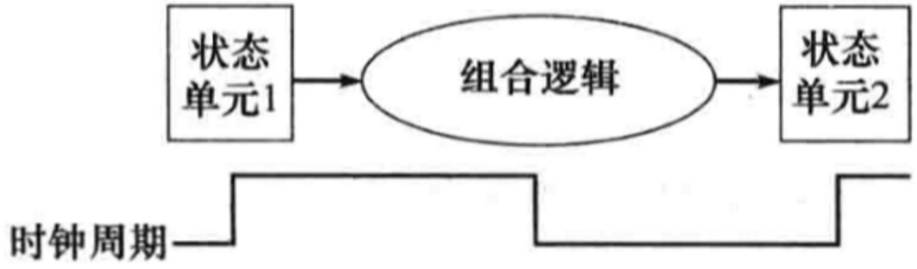
i-type
addi立即数通过instruction memory得来

设计的原则

2019年4月13日 19:10

通过使用组合单元（组合逻辑）和状态单元（时序逻辑）来实现设计
对于时钟频率的设定，要求在有效始终边沿信号到来之前，状态单元的输入信号必须达到稳定

asserted代表逻辑高
deasserted代表逻辑低



Combinational 不取决于之前的值，只是取决于输入值，组合逻辑
State element 输入值取决于之前的值，时序逻辑

在传输数据时，宽度可以看作32位
也就是说，这32位寄存器不是分别传输，而是同时传输

除了ALU之外都是时序逻辑
需要clock的部分：
PC，instruction memory，registers，data memory

建立数据通路

2019年4月13日 19:20

数据通路的作用是读取指令

对于读取指令，首先需要program counter来保存每一条指令的地址

由于需要指令的内容，所以需要一个保存指令的存储器，即指令存储器（instruction memory）

指令存储器实际储存在内存中，内存专门开辟了一段指令用来存储指令内容

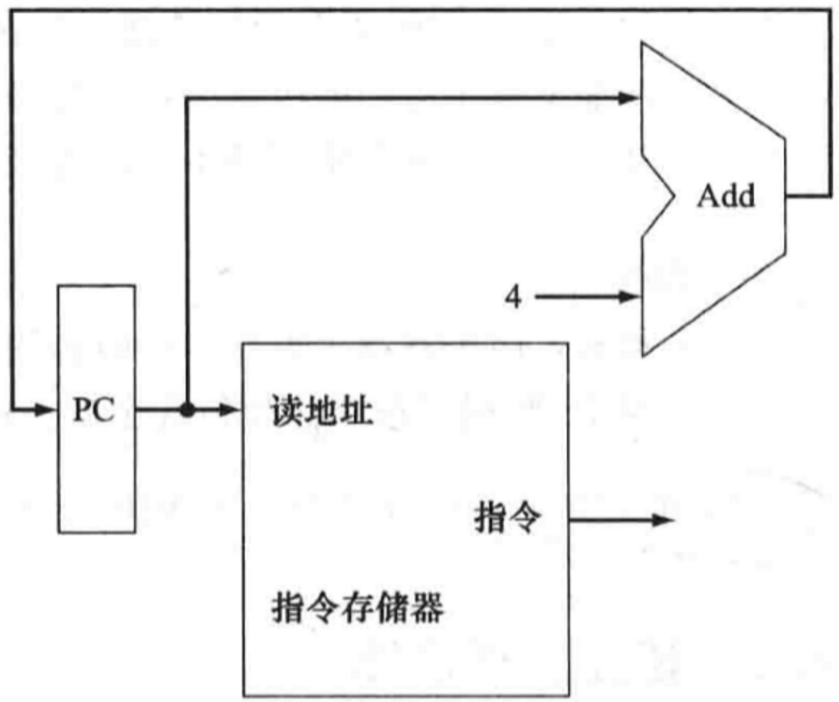
对于很多指令，都涉及到对32位通用寄存器的操作

指令都是通过寄存器堆（register file），总共有5位，

来记录这些寄存器的地址

因此，以下结构为数据通路的电路设计

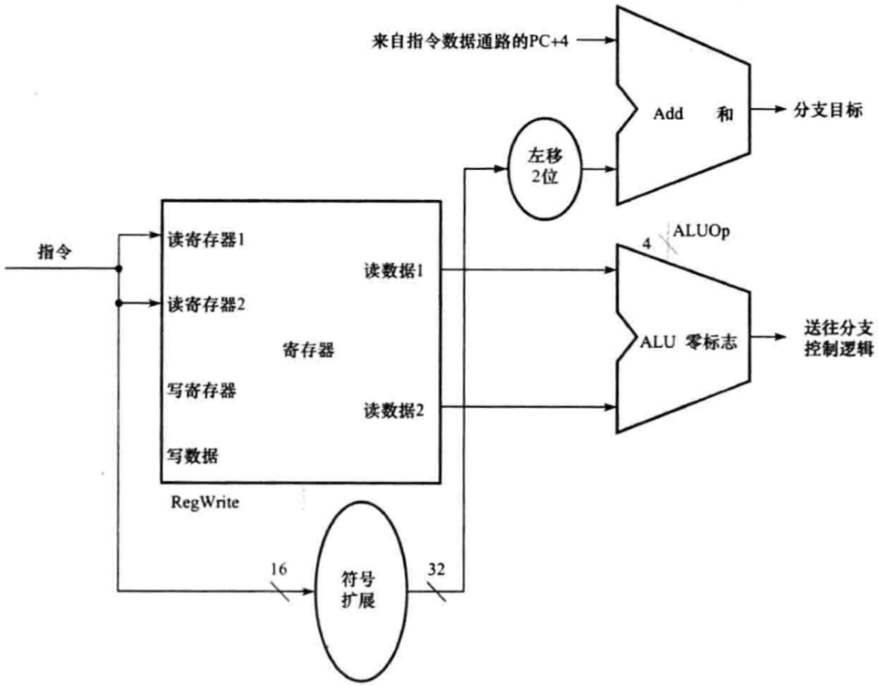
每当进行到时钟上升沿，PC就加4，同时读取相应的指令，放到指令存储器中


Add
4
PC
读地址
指令
指令存储器

处理指令

2019年4月13日 19:46

如图



在此之前，指令先确定类型，从而进行特定的操作

寄存器部分是为了应对add，sub等操作

符号扩展是为了应付分支语句，也就是i-format的16位，通过补零操作变成32位，再往左移变成实际的式子，再和PC+4进行相加，算出分支的目的地

由于

实现指令的步骤

2019年4月13日 18:12

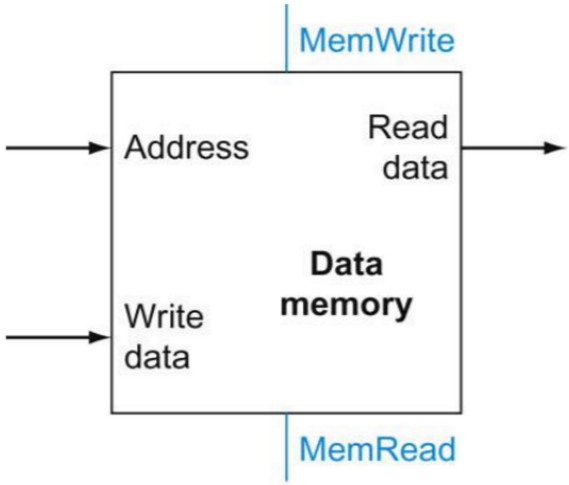
首先，根据program counter内部的寄存器的值在内存中去处指令
根据指令的字段，读取指令，读取指令一般会涉及到两个或三个寄存器

各个电路结构

2019年4月13日 19:33

Data memory

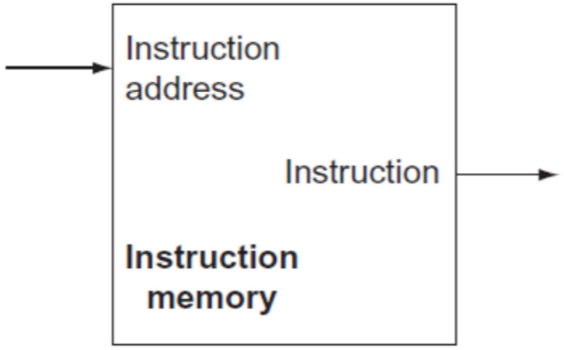
用来存储数据的内存



需要有可读，可写的操作，也要输入地址，得出数据（读字符串）

Instruction memory

用来储存指令的地方



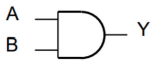
由于每一次都在时钟上升沿时读取指令，所以没有MemWrite和MemRead按钮

Register

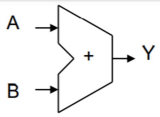
用来存放通用寄存器的地方

对于组合逻辑部分，则是

- And gate
  - Y = A & B

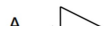


- Adder
  - Y = A + B

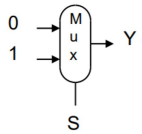


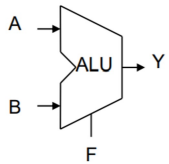
- Multiplexer
  - Y = S ? 1 : 0

- Arithmetic/Logic Unit
  - Y = F(A, B)



由于读入的指令在下一个时钟上升沿之前不会变，所以没有RegRead控制
Register没有RegRead，永远都可以读





对于ALU，F是输入规定的操作数，从而进行加法，或减法等等操作
其中，由于经常使用分支语句，ALU还有一个输出代表结果是不是零

Lab

2019年4月9日 10:41

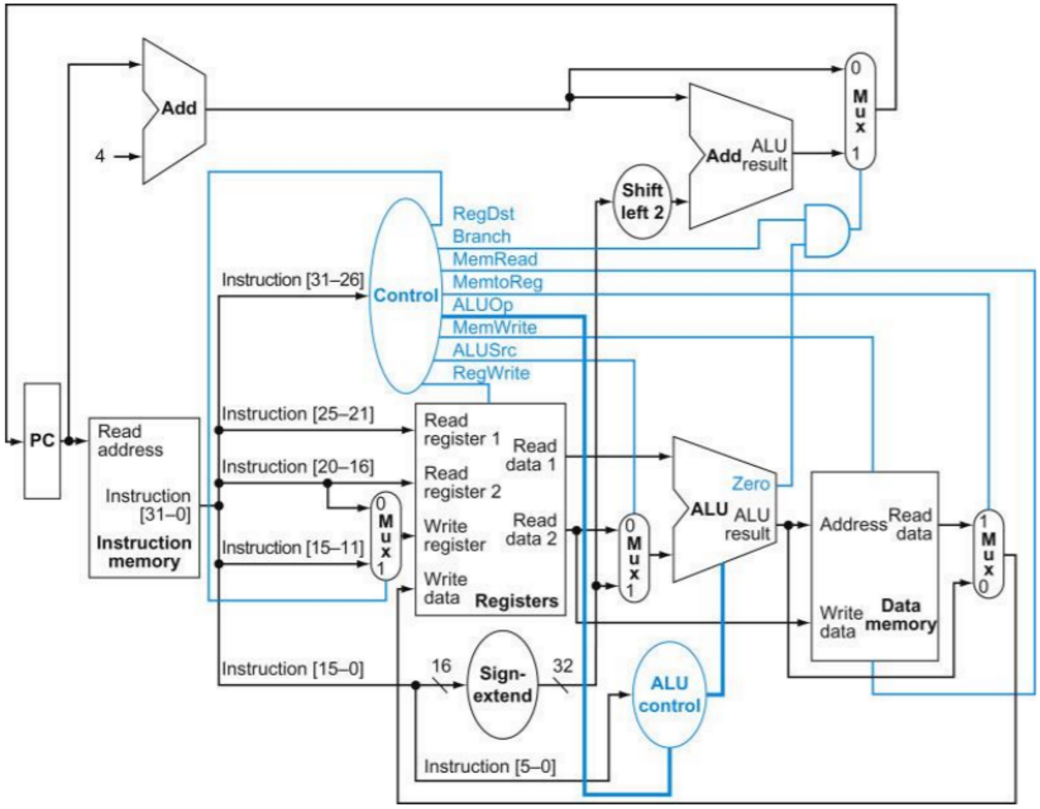
asc不可重复性编程 循环少用，仿真时不支持
FPGA可重复性编程 最好在输入循环用于仿真文件

设计文件有输入和输出
仿真文件没有

完整的操作过程

2019年4月13日 20:13

如图



对于i-format

1. 如果是添加立即数

注意，这个电路图是不支持J-type的

在每一次执行指令的过程中，PC先读取指令所在的地址，并输入给Instruction memory
Instruction memroy将相应的指令输出，从而得到相应的指令内容
同时，PC的当前值和4相加来获得下一次周期的指令地址

处理指令的过程如下
首先，将instruction的前六位放到control中，判断是那种类型的指令，从而发出对应的控制信号

指令为R-format

1. 如果指令是涉及到3个寄存器的操作
   左边的mux为1，同时两个寄存器的地址读入register file中，出来这两个寄存器的值
   同时将下面的instruction【5，0】送到ALU control中，控制ALU进行对应的操作
   令中间的mux为0，使得data2能进入ALU中
   数据通过ALU算出一个新的值
   令右侧的mux为0，使得输出能进入Registers中的Write data，control将RegWrite设置为true，使得数据能写入目标寄存器内（通过instruction【15，11】显示的地址获得）
2. 若指令涉及到两个寄存器的操作（向左移，向右移

对于i-format

1. 如果是添加立即数
   就分成两个操作，先操作前16位，然后是后16位。
   不管是操作哪16位都没有区别（只是ALU执行次序的不同）
   左边的mux设置为0，代表接收两个数
   下面的16位常数通过符号位扩展，变成32位
   中间的mux设置为1，和前面的数值相加，再通过ALU得出结果
   右边的mux设为0，使得结果能进入写的数据流中
   RegWrite变成1，数据写进寄存器内，完成指令
   这不是跳转指令，所以上面的mux设置为0
2. 如果是将数据存入内存，则mux设为0（rt指的是第一个寄存器地址），方便以后进入Data memory中的Write data
   为了防止错误的信息写入寄存器，RegWrite设为0
   偏移常量存在于[15：0]中，通过符号扩展成32位
   中间的mux设为1，并与data1相加，得到真实的偏移量
   如果是储存，则MemWrite设为1，使得能将真实值读入
   如果是读取，则右边的mux的值设为1，RegWrite设为1，方便读入
   上面的mux设为0
3. 如果是分支，则左边的mux值随意，
   下面的16个相对值通过扩展，右移，再与真实值相加，变成了真实的地址值
   同时Branch变成1，代表可以通过
   ALU经过两个寄存器的比较，通过zero来输出，如果结果是0，则上面的mux输出位1，PC切换成目标label，反之，变成0
   中间的mux为0，使得能读出的两个数进行比较，结果通过zero产生
   右边的mux值随意，MemWrite和RegWrite变成0，代表不允许进行写入操作
   instruction【15，11】显示的地址获得）
2. 若指令涉及到两个寄存器的操作（向左移，向右移等），左边的mux就设为1，代表可以允许写入
   下面的16位通过符号扩展，变成32位，通过中间的mux（中间mux设为1）进入ALU。值得注意的是上面的数据是需要修改的数，而下面输入的数据只有[10：6]是可用的数据（代表移位的个数）
   结果通过右边的mux（右侧mux设为0）进入写入的数据，将结果写入目标寄存器内
   而上面的mux设为0，代表不跳转

综上所述，如果是r-format，左边的mux为1
如果为3个寄存器的操作，中间的mux为0，否则为1
右边的mux为0，上边的mux为0

传统处理指令的方式

2019年4月24日 13:53

之前处理指令的方式是单周期实现（single clock cycle implementation）
也就是要求所有指令都在一个周期内完成，优点是实现简单，缺点是效率太低了
对于执行周期比较短的指令来说，会有很多的时间空余
而使用pipeline会很好的解决这个问题

pipeline指的是将时钟周期变成每个单元指令的最大时钟周期
在执行一条指令中，同时读取另外一条指令，使得在一条指令执行的过程中其他单元能尽可能地
利用，即为pipeline

对指令来说，每一条指令都必须且最多会使用到下面五个单元
指令存储器，寄存器堆（在寄存器堆中取出指令），ALU，数据
存储器，寄存器堆（将数据写入寄存器中）

要点

2019年4月22日 13:56

在判断条件分支过程中，加了一个快速比较的过程（快捷模块），而不需要直接使用ALU，这样就不用得到ALU，省了一个时钟周期

根据汇编的形式，直接执行跳或者不跳的部分，好的编译器会预测分支是否跳转

如果预测正确，就直接执行

如果预测错误，就多一步，执行反面的操作

对于寄存器堆读和写操作，可以在同一个时钟周期内进行

如果需要读的是寄存器中需要写入的数字，也可以同时进行（需要读入$t2,而写入的操作是写入寄存器$t2,则这种操作也可以同时进行，读取的数据是将要写入寄存器$t2的数值）

对于非流水线的时钟周期，是各个单元相加

对于pipeline的时钟周期，是各个单元取最大值

如果没有ALU-ALU forwarding，ALU—ALU运算就会相差2个cycle（相对于理想化pipeline来说）

加上了ALU-ALU forwarding，就可以直接进行pipeline，不需要NOP指令

如果没有lw-ALU forwarding，lw-ALU运算就会相差了2个cycle

加上了lw-ALU forwarding，仍然会相差1个cycle

使用了forwarding会使得时钟周期变长

Pipeline定义

2019年4月24日 14:07

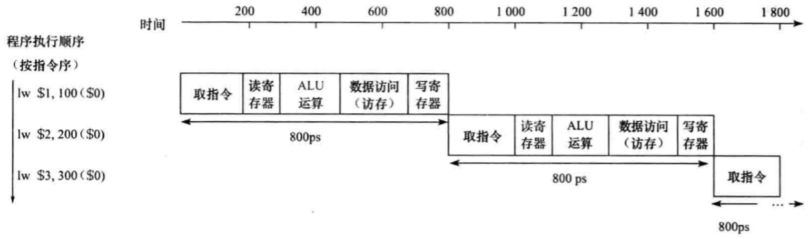
pipeline指的是流水线，指的是将一个具体的任务分成许多个小的步骤
每一个步骤互不干扰，这样就可以在同一时间对两个任务执行不同的步骤
使得运行时间降低
pipeline指的是一个时钟周期内做多个指令
资源是固定的（并没有增加多个处理器），pipeline是将多个任务分解成几个小任务，这几个小任务交给不同的部分来做工作

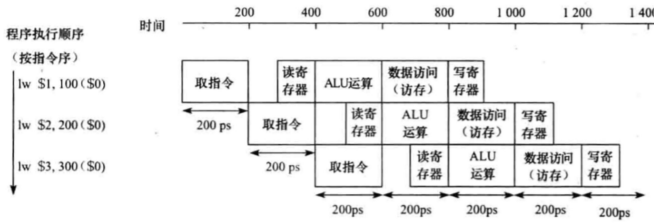
阻塞（stall），指的是因为冒险不能进行正常pipeline（当前时间不能执行下一条指令）
又称作bubble

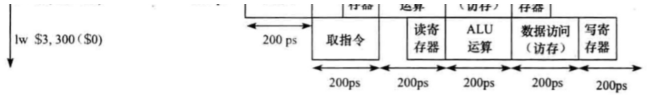
pipeline指的是同一个时刻用一个机器的不同的资源来优化
parilism指的是用不同的机器的不同资源进行优化

pipeline
可以提高throughput（吞吐率），但是latency（执行每一条指令花的时间）升高了（一共进行五个时钟周期，每一个都取五个中的最大值）

理想化的流水线操作







如果都按照这个方式来，则根据公式

优化的倍数 = 原来需要的时间 / 优化后需要的时间

得到优化后需要的时间为原来的1/4（当指令数很多的时候）

做成pipeline的方法

2019年4月24日 14:14

每一条指令都可以简化成5个步骤
取指令，读寄存器（抽取寄存器中的值），ALU操作，数据存取（将数据存到内存中），写寄存器（将内存中的或者已经计算好的值存到寄存器中）

要想做好pipeline的方法

1. 使指令的格式相同，也就是必须要求指令的长度相同
   对于x86（指令格式不相同），则先转换成指令相同的格式再进行pipeline
2. 很少的指令格式，也就是说指令简化的步骤越少越好。
   步骤多了会降低效率
3. 存储器操作数仅出现在指令中
4. 所有操作数（op）在存储器中必须对齐，如果没有对齐就会新添加一个步骤，降低效率

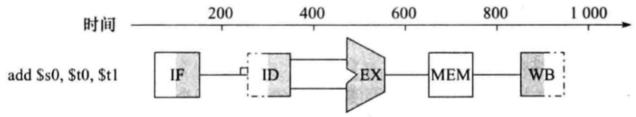
冒险

2019年4月24日 14:26

Hazard 冒险
指的是在pipeline中，由于某些原因，在下一个时钟周期中下一条指令不能执行（有些指令要读，有些要写，
就产生了竞争，因此要停止运行一段指令）
分为三种情况

结构冒险（structural hazard）：由于硬件设计的不好，就会产生。
例如：将instruction memory和data memory合在一起，就使得取指令和取内存中的数据不能同时进行（如果同时写就会产生冲突）
解决方法是设计者在设计流水过程中就避免这样的问题
例如：现实中是用cache来存储数据，而且内存中instruction memory和data memory是可以同时进行的
寄存器堆的读和写是可以同时进行的

处理指令的步骤如下

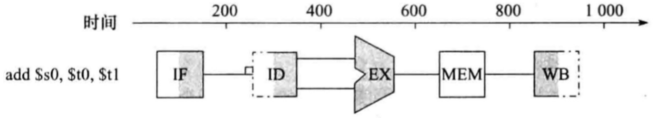


IF指的是instruction fetch，抓取指令
ID指的是instruction decode，从寄存器中读取指令
EX指的是execute opertation，指令ALU运算
MEM指的是access memory operand，将数据读或写入内存中
WB指的是write result back to register，将结果写入寄存器中

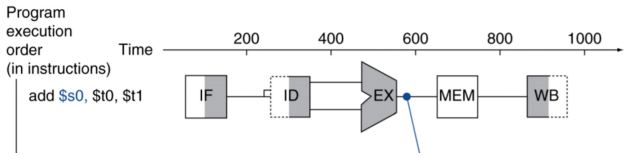
数据冒险（Data hazard）：现在指令的指令需要上一个指令的结果，而上一个指令还没有获取到结果，就不能马上进行pipeline
例如：对于下面两条指令，就不能很好的做pipeline（sub需要用到add的结果）加法指令需要进行到第五步才能开始进行下一条指令

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

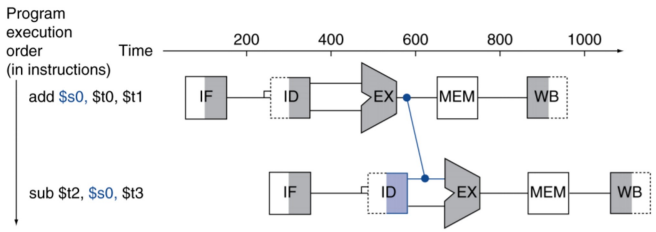
解决方法是：



使用Forwarding（bypassing）：在EX运算的结尾处加一条线，连接到EX开始处，使得指令能连续执行，直接使用结果，不需要读取
旁路（bypassing）：指的是将原件的通路指向之前的原件，从而减少了时间消耗，bypassing（设计桥梁的方式）不止一种



MEM指的是access memory operand，将数据读或写入内存中
WB指的是write result back to register，将结果写入寄存器中



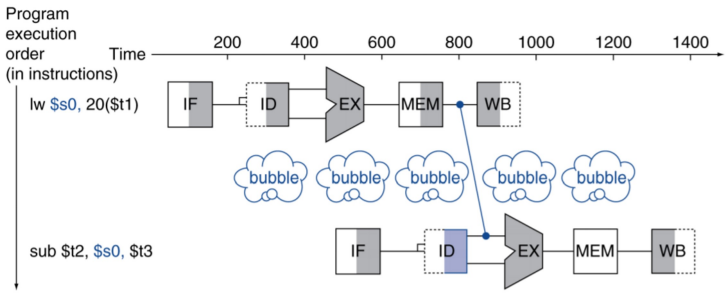
但是即使使用了bypassing也不可避免地产生延迟
取数-使用型数据冒险（load-use data hazard），data hazard的一部分：读取数据和使用数据的竞争（使用by passing）
例如：
对于上面的指令

Add $s0,$t0,$t1	Sub $t2,$s0,$t3

改成了

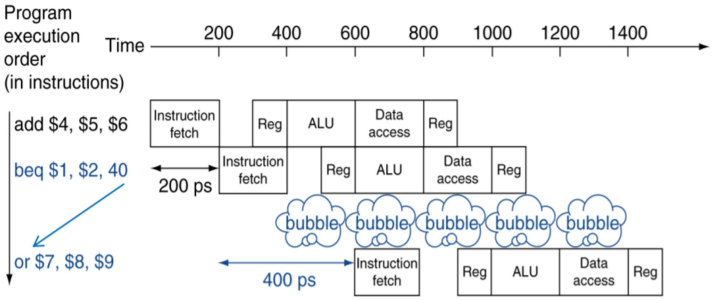
Lw $s0,20($t1)	Sub $t2,$s0,$t3

则就必须要等从内存取数操作完成了之后才能执行下一条指令，这样就不得不耽误一个时钟周期，这种现象叫流水线阻塞（pipeline stall），也叫bubble
下图为修改后的bypassing和流水线方式



解决方法是：通过优化编译器来改变汇编程序的执行顺序，使得lw操作的目标不是下一条需要用到的数值
注意：
对于
Add $a0,$s0,$t0

控制冒险（control hazard）：决策依赖于执行的结果导致的竞争
例如



如果直接等待，中间就会存在一个bubble，导致时间增加
解决方法就是提前预测
就是先执行预测结果，如果是对的，就继续
如果预测是错的，就重新进行下一步

对于预测的方法，有
Static branch prediction：判断是不是循环来进行预测
Dynamic branch prediction：通过硬件来统计之前结果来进行预测，使用分支预测表来预测是否有分支

Sw $a0,($t1)

这两个指令是可以没有bubble的，通过bypassing来实现（将EX结束端连接到EX开始端）

对于不同指令的不同组合方式，应该具体情况具体分析，看看是不是一定会产生bubble

指令级并行

2019年5月6日 20:53

指令级并行（Instruction Level Parallelism）
指的是每一次运行两个指令

为了实现ILP，有两种方法
一种是将指令的步骤分成多个，进行更深度的pipeline
另外一种是使用更多的物理资源，每一个时钟周期吞掉多条指令，这个方法叫multiple issue

在ILP中，使用的单位是instuction per cycle（IPC），CPI的倒数
由于涉及到并行运算

实现并行

2019年5月9日 17:45

实现ILP的方法：
使用multiple issue，也就是复制计算机内部部件的数量
为了达到ILP，有两种方法：
Static muitiple issue和Dynamic multiple issue
Static multiple issue指的是使用编译器来对并行进行优化，由于这个工作在指令进行前完成，所以是静态的
Dynamic multiple issue是在运行过程中对并行进行优化，是通过硬件（处理器）来实现的，所以是动态的

Static multiple issue

2019年5月9日 18:01

Static multiple issue（用编译器来执行避免hazard）
为了实现这个目标，在编译的过程中就要确定issue packet，也就是确定同一时间内需要发射什么类型的指令
还需要一个超长指令字（Very Long Instruction Word）来保证并行的指令之间不会相互影响

实现的过程中会遇到的hazard（只列举部分）：

1. EX data hazard：
   eg：
   ```
   add  $t0, $s0, $s1
   load $s2, 0($t0)
   ```
   这个例子中$t0都被用到了，所以不能进行并行操作
   解决办法是将这两条指令放到不同的packet下
2. load-use hazard
   解法是放到两个不同的packet下，而且隔一个时间段

eg：将下面指令变成两条指令的并行

```
Loop: lw   $t0, 0($s1)       # $t0=array element
      addu $t0, $t0, $s2     # add scalar in $s2
      sw   $t0, 0($s1)       # store result
      addi $s1, $s1,-4       # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

为了简化操作，先将指令
类型的指令或者存储类型
1. 需要data memory
2. 不需要data memo
在每一个周期同时执行这
条件：
1. 如果这两个指令的目
2. load指令有一个周期
   少

分成两部分：是否需要data memory（或者说，ALU
的指令）

ry
两条指令

目标寄存器相同，就不能同时进行
期的使用延迟（use latency），这些都会导致IPC减

答案是：

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

此时IPC只有1.25（5/4，五条指令，四个周期）

加快的方法是：

Loop unrolling：通过循环展开来对代码进行优化（可能每一次对多个循环的过程进行展开，这里发现最优的个数是四个）

解决的问题：数据本身没有dependence，由于有loop，loop的存储和读取造成了数据的dependence

在循环展开的过程中，会存在寄存器重命名，通过增加使用的寄存器，来保证消除一些虚假的数据依赖

反相关：只是和名字相关，而不是和寄存器本身的值相关

Dynamic multiple issue

2019年5月9日 18:40

动态发射处理器简称超标量（superscalar）

流水线被分成3个主要单元：
取值和发射单元，多个功能单元，一个提交单元（commit unit）
每个功能单元都有自己的缓冲区（保留站，resversation station）
先将操作数送到处理单元，然后送到缓冲区
在操作数齐全且功能单元就绪时，将操作数送到提交单元
提交单元缓存这个结果，这里的缓存区叫reorder buffer
在确定安全后，将缓存区的数据写入寄存器堆

执行方式是：顺序发射，乱序执行，顺序提交（为保证代码顺序执行）
防止一些阻塞发生（cache中数据缺失）导致的时间消耗
当数据进入保留区中，原本寄存器中的值不再需要，就可以释放寄存器资源

Cache

2019年5月6日 14:09

miss：发现想要的数据不再cache中
如果发生了miss，就将丢失的东西放到cache中
Miss ratio：misses/accesses
hit：指的是想要的数据在cache中
Miss penalty：发生一次miss所需要的总时间

寄存器中以word存储，内存中以byte为单位存储
内存中的地址是一个虚拟的值，不是真实存在的
block：每一次copy到cache，都是copy附近的数据，而不是一个准确的数据

通过使用direct map来对一个区域的block储存
每一个区域存内存中固定的地址，加上前面的前缀用来区分内存中存储的地址（tag），还要加上前面的前缀代表是否有数（valid）（即是否可以取数来进行运算）

write：通过cache看看是不是在里面
然后将cache进行操作，写了多次再存（当这个cache中的内容要被踢出去时再将数据放到memory中）（设置一个dirty bit，看看是否修改过）
还有一种方法是使用缓冲，等到缓冲满了就放到内存中

如果内存使用byte来存储，而block用word来存储
则需要在block头上加上offset，代表应该读取什么

局部性原理

2019年5月9日 19:02

Spatial locality（空间局部性）：如果想要访问某个数据，就很有可能访问附近的数据
这样将block搬到cache中，就会很少有数据丢失
Temporal locality（时间局部性）：如果这个数据曾经被访问过，那么在不久的将来这个数据极有可能再次被访问
由于数据有这两个特性，所以使用cache就很有必要

存储器层次结构

2019年5月9日 19:04

存储器的价格从高到低为：SRAM > DRAM > 磁盘

通过局部性原理就可以建立memory hierarchy
便于简化，这里只讨论两层存储器结构
block：两级层次结构储存信息的最小单元
注意：block应该不大也不小
如果block太小，则根据spatial locality，附近的信息极有可能不在cache中，造成时间的消耗
如果block太大，则cache存储block的数量就变小，就有可能带来更多的数据丢失，而且，block适度地小可以增加数据写入内存的概率。如果数据丢失，则将单个block搬入cache的时间也变慢了，增加了时间的消耗。（不利于temporal locality，利于Spatial locality）
根据实验，当block太大，数据丢失率是上升的

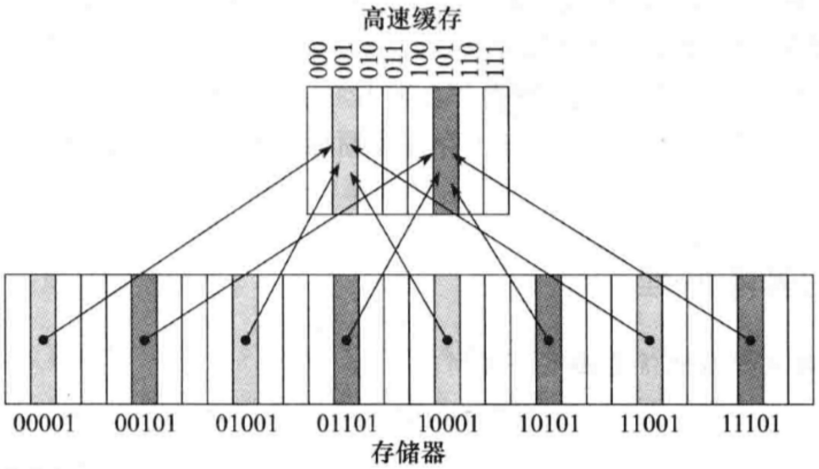
SRAM：cache
DRAM：内存
Flash：U盘

DRAM（内存）访问数据很慢
cache用SRAM技术，是处理器的存储结构，更加的快
cache在CPU内部
DRAM，经常刷新，易失性
SRAM，不经常刷新，具有易失性

cache基本原理

2019年5月9日 19:30

通过直接映射的方式来存储block
内存中的每一个数据都放到cache的固定位置
也就是说，cache中每一个位置的block可以由多个内存中的位置存储
内存中每一个位置只能找到对应的cache来存储
映射方法是：
（内存中block地址）mod（cache中的块数）



判断cache中的是否存在需要的数据：
通过标签来判断
eg：假设内存大小是32word，cache的大小是8word
则32 / 8 = 4，说明每一个cache中的block需要存储4个不同位置
这四个不同位置通过地址前两位来区分（内存大小只是将其分成4分，
说明相同cache的位置的后三（$\log_2 8 = 3$）位必定相同，不同的只有前
两位（$\log_2 32 - 3 = 2$））
也就是说，标记位是前两位，地址低3位的索引域用来选择cache中的
块
这样，当需要找内存相关的数据时，只需先找索引，再找标记位是否对
应即可
另外，还需要设置一个有效位来标识这个块是否含有有效地址（刚开机
时，cache中还没有数据，此时就是无效数据）

标记域（tag）：用来查看想要的数据是否在cache中
cache索引（index）：用来选择cache中的哪一块区域
值得注意的是，cache索引实际上并不存在，也就是说并没有实际
的空间存放cache索引，计算机可以自动找到

MIPS中内存通过一个字（byte）来存储，地址是32位，cache中存的是字，MIPS中规定一个字存的是32位，所以还
要把字换算成位

cache块的大小：看offset
cache块中有多少项：看index
每一个cache块中会存放多少种不同的数据：看tag

MIPS中内存通过一个字（byte）来存储，地址是32位，cache中存的是字，MIPS中规定一个字存的是32位，所以还要把字换算成位

假设一个cache中有2^n个块（就是说标记域为n），每一个块有2^m个字（word）（2^{m+2}个字节（byte）），直接映射到cache

则索引域的大小为32 - n - m - 2

这32位被分成了n（用来存储不同的块，tag），m+2（2表示偏移量）（表示同一个块中不同的数据的个数，另外的2指的是偏移信息，因为查找地址时是通过4的倍数进行查找，所以要加二），还有索引位（区分不同的块）

32位 = tag + index + 偏移量

而且还需要一个有效位

cache总位数 = $2^n \times$ (块大小 + 索引域 + 有效位)

$= 2^n \times (2^m \times 32 + (32 - n - m - 2) + 1)$

eg：

例题 • cache 中的位数

假设一个直接映射的 cache，有 16KiB 的数据，块大小为 4 个字，地址为 32 位，那么该 cache 总共需要多少位？

16KiB = 2^{12}Byte，块大小为4个字（2^2），所以共有2^{10}个块

所以索引域大小为32 - 10 - 2 = 20

所以cache总数为$2^{10} \times (2^2 \times 32 + (32 - 2 - 10 - 2) + 1) = 2^{10} \times 147$

故需要147Kib，为18.4KiB，故是原cache的1.15倍

eg：

考虑一个 cache 中有 64 个块，每块大小为 16 字节，那么字节地址 1 200 将被映射到 cache 中的哪一块？

由于给出的是字节地址，所以先用1200÷16，算出这个字节地址之前被分成多少块

然后再用结果mod64，表示放到第几个cache中

注意：这里的32位指的是每32位（bit）构成了一个word，也就是cache中的基本存储单位，由于偏移量是2，所以每一个块共有2^2个byte，4个byte相当于一个word也就是一个字

偏移量的单位是byte，要除以8换算成word才能和32相乘

者

一

相

成

读操作

2019年5月9日 20:34

如果内容命中，则只需要将cache的内容调入寄存器中即可
当cache内容缺失时，处理器发生阻塞
直到从存储器取回目标数据时，程序才能继续进行

cache内容缺失时需要的时间比直接从内存中读取数据时间要长
cache内容缺失时先要将数据从内存放到cache，再抽取数据

写操作

2019年5月9日 20:37

Write-through：
当将新的内容写入cache中，每写一次就更新内存中的内容
优化：设置缓冲，当数据在等待写入内存时，设置一个缓冲
当处理器将数据写到缓冲和cache中就继续执行
缓冲自行将内容写入内存
当缓冲满了，处理器停止，直到缓冲中有一个空位置

如果需要写的数据不在cache中：
write-allocate
数据块从主存中取回，并在该块的恰当区域重写数据
Not write-allocate
直接在内存中写，不放到cache中
初始化时用not write-allocate
平常情况下用write-allocate

write-back：更复杂
当cache中的内容需要被替换时，再将内容写进内存
不过这个机制需要筛选被替换的内容是否被修改过，不然很浪费时间
很多的write-back机制也需要写缓冲

FastMath处理器

2019年5月16日 18:58

对于该处理器，将cache分为了两部分
Data cache 和instruction cache

首先，将需要的数据地址通过信号传到cache中
cache检测是否命中
如果检测到命中，就将数据放到CPU中
如果检测到未命中，就从内存中找对应的数据

一般的，使用混合cache具有更高的命中率
但是分离的cache可以提高cache的带宽

要点

2019年5月13日 14:02

RAM：随机存储
ROM：只读

增加associativity可以减少miss rate

多个层级的cache可以降低miss panelty和miss rate
L1的容量小于L2的容量

cache性能

2019年5月16日 19:02

改进cache性能有两种方法

1. 减少储存器中不同数据块，而减少cache中同一位置的竞争，从而降低缺失率
2. 额外增加一层cache来提高速度，这个方法被称为多级缓存

如果刻意加大cache容量就会导致命中消耗的时间增加，反而不利于速度的提升

CPU时间 = （CPU执行时钟周期数 + 存储器阻塞时钟周期数）* 时钟周期

存储器阻塞时钟周期数 = （存储器访问次数 / 程序数）* （缺失数 / 指令）* 缺失代价

当写的时候还需要加上缓冲区缓冲带来的阻塞，不过一般情况下可以被忽略

eg：

假设指令 cache 的缺失率为 2%，数据 cache 的缺失率为 4%，处理器的 CPI 为 2，没有存储器阻塞，且每次缺失的代价为 100 个时钟周期，那么配置一个从不发生缺失的理想的 cache，处理器的速度快多少？这里假定全部 load 和 store 的频率为 36%。

首先计算i-cache的

$2\% \times 100 = 2$说明由于写操作带来的阻塞占2个CPI

然后计算d-cache的

$4\% \times 100 \times 36\% = 1.44$这里不需要乘98%，因为可能同时造成指令丢失和数据丢失，由于只有load和store才会发生指令缺失，所以要乘36%

所以总的CPI是$2 + 1.44 + 2 = 5.44$（miss penalty指的是由于存储器阻塞带来的额外时间，而不是总的时间）

是理想状态的$5.44 \div 2 = 2.72$倍

平均存储器访问时间（AMAT）

AMAT = 命中时间 + 缺失率 * 缺失代价

cache定位机制

2019年5月16日 19:32

直接映射（direct mapping）：
存储器中的任何一块都被映射到存储器中的唯一位置
全相连（fully associate）：
存储器中的块可以放到cache中的任何一个位置
组相连（set associate）：
每个块可被放置的位置数是固定的。即一个块只能被放到cache的连续某个区间内

事实证明，set associate具有更高的效率

在set associate cache中
存储块的组索引值 = 块号 mod cache中的组数
由于使用组相连，所以在检索过程中需要检索这个组中的任何一个位置

Set associative：
放同一个block的地方变成set
set里面放多个block
防止多次替换导致的多个miss
缺点：需要查找每一个set中的tag，就很麻烦

使用组相连能降低缺失率，但是提高了命中时间
在cache总容量不变的情况下，相连度（每个set中块的个数）增加一倍，索引位就会减少一位，tag位就会增加一位

组相连中每一块都包含一个地址标记来给出块地址，含义如下

标记	索引	块偏移

在一个set里面程序需要检测每一个块标记（tag），观察是否为想要的标记
索引用来定位寻找的set
块偏移指的是块中被请求数据的地址

不需要有确定的地点告知需要放到哪里
由于set的数量不能太多，所以直接一个一个比较即可
通过连接多个电路即可

替换set中的块

2019年5月16日 19:54

Least-recently used（LRU）
剔除cache的算法
通常是随机生成的，随机踢掉
和LRU的性能相似，而且用了更少的空间

替换掉组中不用的块可以用LRU算法
但是需要记录每一块的使用情况
在关联度很高时，不太可行

使用多级cache减少缺失代价

2019年5月16日 20:00

大多处理器会再增加一级cache，设为二级cache，一般都在处理器内部
当一级cache中发现缺失，先在二级cache中找，那么一级cache发生缺失的时间就是访问二级cache的时间

通常情况下，一级cache的容量很小，访问时间也很短
二级cache容量就比较大，容量也更大，关联度也更高，从而减少缺失率，减少访问内存的几率

使用分块进行软件优化

2019年5月16日 20:10

指的是在处理矩阵的过程中
使用blocksize的变量将矩阵分块
存入cache中，使得在运算过程中减少了缺失率

Dependability

2019年5月22日 16:04

指的是dependable
对于用户的需求，有两种状态：服务实现，服务中断
用失效（failure）代指实现->中断
用恢复（restoration）代指中断->实现
MTTF（mean time to failure）指的是平均多久失效一次
AFR（annual failure rate）指的是给定MTTF的情况下，一年之内器件失效的比率（如果换算成小时，就是MTTF的倒数）
MTTR（mean time to repair）维修平均时间，指的是恢复所需要的时间

MTBF（mean time between failure）
失效间隔平均时间 = MTTF + MTTR
可用性：MTTF / （MTTF + MTTR）
系统正常工作时间在连续两次服务中断间隔时间中所占的比例

提高MTTF的三种方法：
Fault avoidance：合理的建构系统
Fault tolerance：使用冗余措施，在发生失效时确保冗余措施仍然工作
Fault forecasting：对故障进行预测，在失效前替换

Hamming SEC code

2019年5月22日 16:17

避免发生错误的方式有增加冗余，就是将原来的编码扩展三倍，这样会保证更高的精确度，但是很占空间，于是就产生了hamming code

Hamming distance：
两个数对应位置不同位的数量：
011011和001111中就是第2，4个不一样，所以hamming distance就是2
Minium distance：
指的是对的编码之间的最小hamming distance
eg：使用奇偶校验位，假设奇偶校验位在最后一位，1代表奇数个1
假设正确的数据只有两种，即10和11，扩展成奇偶校验位为101和110
则minimum distance为2

计算Hamming error correction code：

1）对数据部分从左到右由 1 开始依次编号，这跟通常采用的从最右边开始由 0 开始编号的做法相反。

2）将所有编号为 2 的整数次幂的位标记为奇偶校验位（1，2，4，8，16，…）。

3）其他剩余位置用作数据位（位置 3，5，6，7，9，10，11，12，13，14，15，…）。

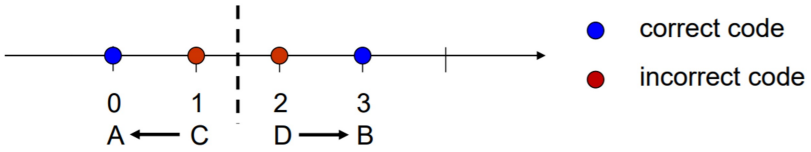
4）奇偶校验位的位置决定了其对应的数据位（图 5-24 用图形的方式进行了说明）如下所示：

位置		1	2	3	4	5	6	7	8	9	10	11	12
编码之后的数据位		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
奇偶校验位覆盖范围	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

图 5-24　用于 8 位数据的汉明纠错码，其中奇偶校验位、数据位及覆盖范围如图所示

- 校验位 1（0001_2）检查第 1，3，5，7，9，11，…位，这些数位编号的二进制形式最右边一位均为 1（0001_2，0011_2，0101_2，0111_2，1001_2，1011_2，…）。

如果minium distance为3，则就有
single error correction（SEC）
Double error detection（DED）



如果数据传输过程中只发生了一个bit错误，假设结果为C，则原始数据必定是A，就可以进行correction
如果发生了两个bit错误，假设结果为C，由于在实际情况中不知道实际发生了一个错误还是两个错误，就不能进行纠正，只能检测出错误
如果发生了三个bit错误，就无法检测出错误
根据以上规律，可以得到：
若minium distance为m
对于correction的个数，为$\left\lfloor\frac{m-1}{2}\right\rfloor$
对于detection的个数，为m-1

- 校验位 2（0010_2）检查第 2，3，6，7，10，11，14，15，…位，这些数位编号的二进制形式右边起第二位均为 1。
- 校验位 4（0100_2）检查第 4 ~ 7，12 ~ 15，20 ~ 23，…位，这些数位编号的二进制形式右边起第三位均为 1。
- 校验位 8（1000_2）检查第 8 ~ 15，24 ~ 31，40 ~ 47，…位，这些数位编号的二进制形式右边起第四位均为 1。

注意到每个数据位都被至少两个奇偶校验位覆盖。

对于每一个奇偶校验位的数值要求，要求奇偶校验位及其负责的位置上各个值之和为2的倍数

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
		0	1	1	1	0	0	1	0	1	0	1	0
Encoded date bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverate	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

此时如果出现了一位错误，则就可以立即纠正

而对于两位错误，就只能发现，而不能纠正

所以hamming distance = 3

如果设置了一个全局的奇偶校验位，就能使hamming distance变成4

就可以纠正一位错误，检测两位错误

通过设置hamming code的方式就可以很好的纠错，而且额外空间比传统的方式小

要点

2019年6月2日 17:05

虚拟内存存放在硬盘中
存放的地址数量比内存大得多
虚拟内存存放的是磁盘上的物理地址
虚拟存储器上面的值有三种状态：
未分配，未缓存，已缓存
未分配指的是在磁盘中还没有分配地址
未缓存指的是磁盘上面的数据还没有放到内存中
已缓存指的是磁盘上面的数据已经放到了内存中

页表存放在内存中
存放的是物理页号或者磁盘地址
    物理页号指的是内存中的实际地址
    磁盘地址指的是虚拟内存中的对应数据的地址
大小和虚拟内存相同
页表中的元素叫PTE（page table entries）
每一个PTE都设置了有效位
PTE有效，说明存储内容放在内存中
PTE无效，说明存储内容对应在虚拟内存中

TLB放在cache这个级别
是page table的cache
记录的是活跃的PTE

虚拟地址由虚页号和页偏移组成
页偏移存放的是精确的数据
虚页号存放的是每一个页表的精确定位

TLB存放的内容和cache毫无关系
    就是说，可能TLB找到，但是cache缺失
    也可能TLB缺失，但是cache找到
页表和内存有着必要的关系
    就是说，页表缺失，数据一定不能在内存中找到
    页表不缺失，数据也可能不会在内存中找到

虚拟存储器

2019年5月22日 15:50

个人电脑中虚拟存储器控制DRAM和闪存，在服务器中则是DRAM和磁盘

用途：
允许云计算在多个虚拟机之间有效而安全的共享存储器
消除一个小而受限主存容量对程序设计造成的影响（自动管理储存结构）
需要对不同虚拟机之间进行保护，确保每个程序只能对划分给他的那部分主存进行读写操作
主存只存放一个程序的活跃部分即可
为了隔离不同虚拟机之间的行为，需要对每一个虚拟机专门设置一个特定的地址空间，虚拟存储器
就是将这个地址空间转换成物理地址，从而实现了虚拟机之间的保护

存储器中，块被称为页（page）
访问缺失被称为page fault
处理器产生一个虚拟地址，在虚拟存储器中转换成实际的物理地址，这个过程叫地址转换
这个物理地址指的是内存中的实际地址

在虚拟存储器中，地址被分成虚页号（virtual page number）和页偏移
实际的物理地址中分成物理页（physical page number）和页偏移
虚页号比物理页号大，用来构造更大的内存的假象
两者的页偏移相同

使用重定位的方式来进行转换，就是将一个虚拟地址转换成多个物理地址
而不是有意的开辟几个连续的块来存放，这样可以减少缺页时的时间损耗
但是增加了访问页时的平均时间

设计思路

2019年5月22日 21:32

为降低缺页率

存储的方式是fully associative

为防止缺页，需要使得页足够大

为了防止缺页，能用算法处理的地方就用算法处理

由于写时间太长，write-through机制会耽误大量时间，所以普遍使用write-back

页的查找和存放

2019年5月22日 17:35

页表寄存器：指向页表首地址的寄存器
页表：定位页（保存虚拟地址和物理地址转换关系的表）
页表寄存器找到的是页表的首地址
页表上面存储的是虚拟地址，通过转换，变成实际的物理地址

如果在页表上面还是没有找到（虚页号上对应的有效位是关闭状态），操作系统就会搜索磁盘从而找到对应的内容

主存优先替换不被使用进程中的页
一般计算机都会近似的使用LRU算法，而不是严格的执行这个算法

交换区：
为进程全部虚拟地址空间预留的磁盘空间
当内存不够用的时候，就用交换区的空间
页表存在主存中，防止页缺失时加载的太慢

只要创建了一个进程，就需要一个页表

TLB

2019年5月22日 18:10

按照传统的方法，页表应该在主存中，就是说要想访问数据就应该访问两次
第一次得到实际的物理地址
第二次得到想要的数据
这样子做就很慢

所以说，就新构造了一个cache，里面装TLB（快表）
TLB里面记录了最近使用地址信息的高速缓存，从而避免每一次都要访问页表
可以说，TLB就是页表的cache，操作和cache与内存之间的操作一样

使用的是write-back机制
由于是write-back，所以要设置一个脏位判断是否需要写的操作

写访问位：判断是否可写

访问过程

2019年5月22日 21:55

首先，CPU发出读取或写入的操作，得到一个虚拟的地址

首先，数据访问TLB，观察是否有TLB中是否有这个虚拟地址

如果有，转换成物理地址，在cache中观察是否有该数据，没有的话，将cache从内存搬到cache中（由于TLB是页表的子集，页表存放内存的地址，所以数据肯定在内存中）

如果没有，访问页表

如果页表含有，先在cache中找，如果没有就将数据从内存中搬到cache

如果页表不包含，即缺失，则直接到硬盘中找，再分配给内存和cache

TLB	页表	cache	这种情况可能发生么？如果可能，在什么情况下发生？
命中	命中	缺失	可能，但若 TLB 命中就不可能检查页表
缺失	命中	命中	TLB 缺失，但在页表中找到表项；重试后在 cache 中找到数据
缺失	命中	缺失	TLB 缺失，但在页表中找到表项；重试后在 cache 中未找到数据
缺失	缺失	缺失	TLB 缺失，随后发生缺页；重试后在 cache 中必找不到数据
命中	缺失	缺失	不可能：如果页不在主存中，TLB 中没有此转换
命中	缺失	命中	不可能：如果页不在主存中，TLB 中没有此转换
缺失	缺失	命中	不可能：如果页不在主存中，数据不允许在 cache 中存在

进程之间的保护

2019年5月22日 22:10

为防止TLB的写访问位被另外一个页改写，需要：

1. 支持用户进程和超级用户管理进程
2. 能让处理器在用户态和操作管理态之间相互切换
3. 提供一部分处理器的状态

令每个进程都有自己的虚拟地址空间，这样就可以映射到不相交的物理页上

从而防止一个进程访问另外一个进程的数据

这也要求一个用户进程不能改变页表的映射

当共享时，需要生成一个额外的虚拟页，还要使用写保护位来防止其他用户改写

在进程切换的时候，保证下一个进程不能访问上一个进程的页表

TLB缺失

2019年5月22日 22:19

如果TLB缺失，有两种情况
数据在主存中，则只需将数据搬到cache中，更新TLB即可
数据不在主存中，需要报错，将任务交给操作系统处理缺页的情况，此时EPC会保存这个报错的值

通过设置使能异常控制处理器是否响应异常
在处理器正确处理异常之前，必须保证不继续发生异常

如果操作系统知道了缺页

1. 用虚拟地址查找页表项，并在磁盘找到页的位置
2. 选择替换一个物理页，如果这个物理页被写过，还要在新的虚拟页装载之前将物理页写入磁盘中
3. 启动读操作，将被访问的页从磁盘上放到被替换的物理页的位置上

异常处理后，必须要使得指令重新执行，好像什么都没发生过

总结

2019年5月22日 22:33

写操作方式及其优缺点：

- 写直达：信息被同时写到 cache 的块和存储器层次结构较低层的块中（对 cache 来说是主存）。5.3 节中的 cache 使用这个机制。
- 写回：信息仅仅写到 cache 中的块。被改写的块只有在它被替换时才写回到存储器层次结构的较低层中。虚拟存储器系统通常采用写回策略，原因在 5.7 节中讨论过。

写回和写直达策略有其各自的优点，写回的主要优点如下：

- 处理器可以以 cache 而不是存储器能接收的速度写单个的字。
- 多次写同一块中的字只需对存储器层次结构较低层进行一次写操作。
- 当块被写回时，由于写一整块，系统可以充分利用高带宽传输。

写直达的优点如下：

- 缺失比较简单，缺失代价也较小，这是因为不需要把整个块写回到较低层存储系统中。
- 尽管为了可行性，写直达的 cache 需要一个写缓冲区，然而写直达还是比写回更易于实现。

缺失类型：

- **强制缺失**（compulsory miss）：对从没有在 cache 中出现的块第一次进行访问引起的缺失。也称为**冷启动缺失**（cold-start miss）。
- **容量缺失**（capacity miss）：由于 cache 容纳不了一个程序执行所需要的所有块而引起的 cache 缺失，当某些块被替换出去，随后再被调入时，将发生容量缺失。
- **冲突缺失**（conflict miss）：在组相联或者直接映射的 cache 中，多个块竞争同一个组时而引起的 cache 缺失。冲突缺失在直接映射或组相联 cache 中存在，而在同样大小的全相联 cache 中不存在。这种 cache 缺失也称为**碰撞缺失**（collision miss）。

设计变化	对缺失率的影响	可能对性能产生的负面影响
增加 cache 容量	减少了容量缺失	可能增加访问时间
提高相联度	由于减少了冲突缺失，因此降低了缺失率	可能增加访问时间
增加块的容量	由于空间局部性，因此对很宽范围内变化的块大小，都能降低缺失率	增加缺失代价，块太大还会增加缺失率

替换

2019年5月27日 14:06

LRU

使用reference bit（发现写过一次的时候就reference bit为1，且）

Miss

2019年5月27日 14:07

分成两种：
Compulsory miss：开始启动的时候不得不miss
Capacity miss：由于内存不够大，miss
Conflict miss：cache中有的地方空了，有的地方是满的，但是满的被踢出去造成的miss（只会发生在direct map和set associative）

避免Compulsory miss：增加block size，代价：放的block少了，access time增加
Capacity miss：Cache size变大，代价：访问时间慢，成本高
Conflict miss：fully assiciative，代价：n个comparator，要么额外加一个lookup table

Full lookup table

2019年5月27日 14:03

用于Fully associative
通过建立page table来查找
Page number：

要点

2019年5月27日 14:30

Parallel processing program：要求硬件上parallel
Amdahl ‘s law
实际的时间 = 可以运行在并行硬件上的时间 / 并行的处理器 + 不能并行的时间

Strong scaling：随着数据量的增大，消耗时间变短
Weak scaling：数据量和处理器数量增加的情况下，消耗的时间相同

Load balancing：
os中不知道哪个要运行的快
平均分配
就算有很好的硬件，软件的指令也很难
分配到中间层中

MIMD：多核

SIMD：（单核）
Single instruction，multiple data
Simplifies synchronization简单
要求parallism很高
MMX：运算多个数（MIPS中是addv，设置了vector processor）

Software as a service
Platfrom as a service
hardware。。。
一部分程序跑在电脑端，一部分跑在云端

Multithread：在一个核中跑很多thread
使用coarse-grain：将多个线程合并（等到需要大量度磁盘的时候就切换线程）
fine-grain：在运行过程中会有bubble，此时运行其他的线程
同时multithreading：多个核中同时填满多个程序，运行
SMP（shared memory multiprocessors）：原来的cache访问不同的memory，现在访问相同的memory，instruction分到不同的processor，就不会对program进行重写，这个是硬件架构

使用多个computer合并成一个
用interconnection network（并不是share同一个memory，但是有信息在云端运行）
从而看成一个大的computer（这些computer共同享用一个computer）
Loosely coupled cluster（每一台电脑为一个独立的电脑，都有自己的os，memory）

1. 一个坏了，可以用另外一个
2. 访问数多时，服务能力高，能处理多个请求
3. 可控制成本

缺点：管理不同电脑的权限会消耗时间

并行处理器

2019年5月30日 17:04

来源：
想将单个处理器变成多个处理器
优点：
提高可用性，一个失效，就可以使用另外一个，而不是一个失效，全部报废
任务级并行（task-level parallelism）：多个处理器运行多个程序，每一个处理器只负责一个程序
并行处理程序（parallel processing program）：顺序或并发的软件运行在并行的硬件上
集群（cluster）：通过互联网连接的多个计算机
多核微处理器（multicore microprocessor）：一个集成电路上包含多个微处理器的处理器
共享内存处理器（shared memroy processor）：共享一个物理地址空间的并行处理器

并行：
分成hardware和software
sequential（顺序），concurrent（并发）：相对于软件来说
serial（串行），parallel（并行）：这些都是相对于硬件来说
相同的程序可以运行在或串行，或并行的硬件上
不同的程序也可以运行在软件上

创建并行处理程序的难点

2019年5月31日 21:03

1. 编写顺序程序较简单，编写并行程序很困难
2. 要想编写一个并行的程序，需要将程序分为等量的若干份，而且面临负载均衡，同步时间，通信开销等问题，就会增加额外的开销

定义加速比为：未采用优化的时间消耗 / 优化后的时间消耗
加速比越大，说明效果越好
问题规模增加，加速比增加
处理器数量增加，加速比增加
假设问题规模为M，处理器数量为P
强比例缩放：在多处理器上不需增加问题规模即可获得的加速比（M / P）
弱比例缩放：在多处理器上增加处理器数量同时增加问题规模所获得的加速比（M）
强比例缩放就是现在给定数据后算出的答案
弱比例缩放就是不同比例下的数据算出来的答案，算法和强比例相同

SIMD和vector

2019年5月31日 21:33

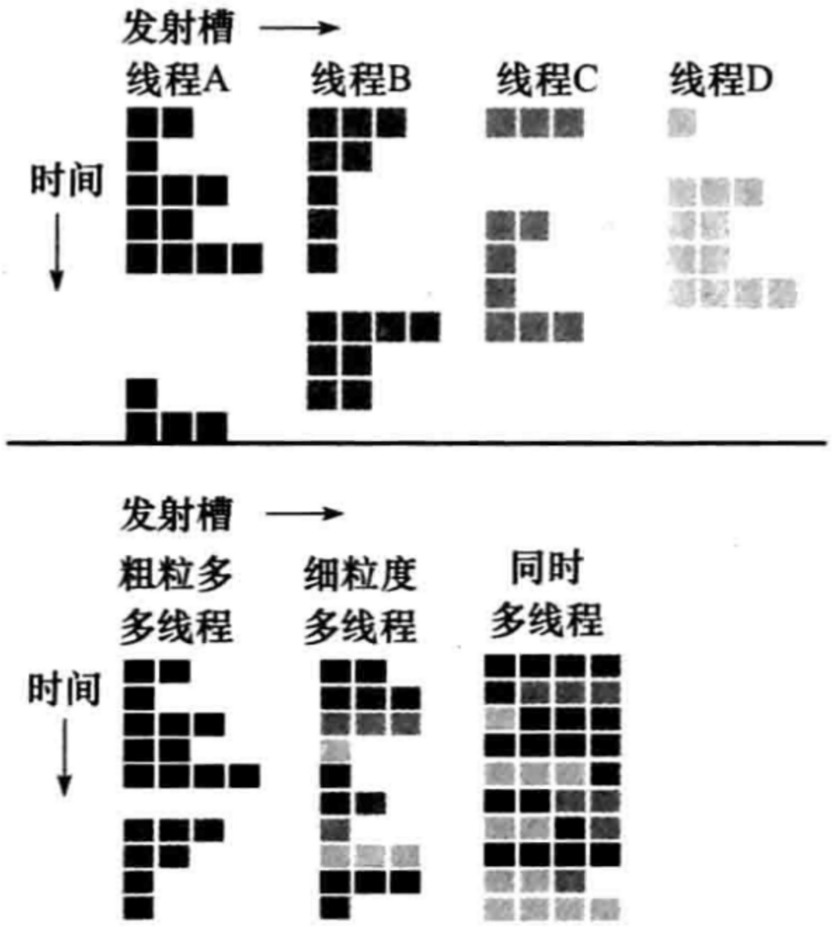
SISD：单指令流单数据流的单处理器
MIMD：多指令流多数据流的多处理器，取数的时候一次取多个
SPMD：单程序，多数据，是MIMD的基本编程方式
SIMD：同样的指令在多个数据流中进行，和向量运算相同，提供了数据级别的并行（data-level parallelism）

Hardware Multithreading

2019年5月31日 21:34

是与MIMD相关概念，允许多个进程或多个线程运行在多个处理器中
多个线程可以运行在一个处理器中，共享一个地址空间

fine-grained multithreading：
每条指令结束后进行线程切换，结果就是在多个线程之间交叉进行
可以隐藏长或者短阻塞引起的吞吐量缺失
coarse-grained multithreading：
仅在高开销阻塞时才进行线程切换
隐藏高开销造成的缺失
simultaneously multithreading：
利用multiple issue，动态调度实现的多线程

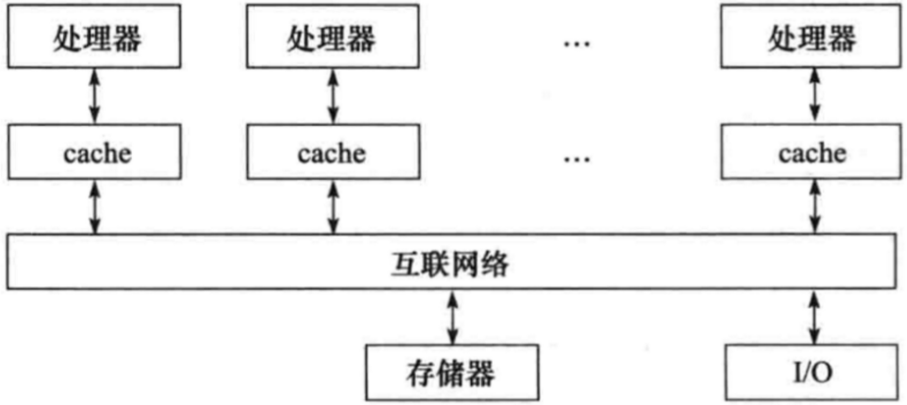

发射槽
线程A
线程B
线程C
线程D
时间
发射槽
粗粒多
多线程
细粒度
多线程
同时
多线程
时间

SMPs和cluster

2019年5月31日 21:34

多线程提升了在单个处理器的运行效率

Shared Memory Multiprocessors 指的是在多个核中共享内存，提高了多核处理器的工作效率



Message parsing multiprocessor

多个独立的小电脑通过网络连接成大的计算机（共用一个操作系统，memory和cache是独立的）

Loosely Coupled Clusters

同上面的MPM，这个是小电脑单独使用一个操作系统

GPU

2019年5月31日 21:34

CPU中的很多工作，GPU不用做
GPU的数据规模是从MB到GB的

GPU有很多的线程
高度的数据并行
用作并行的计算

Virtual machine

2019年5月27日 14:17

应用：操作系统调用时用virtual machine
云计算
来源：用户的os和实际的os不一样
安全性能

Instruction set support
Privileged只有在系统模式下才能用

实现方式：硬件之上有一个virtual machine monitor
Virtual到physical的映射
上层用户不能直接操作底层的用户，只能通过vm来实现操作
如果发生了interrupt，是monitor产生interrupt，然后告诉用户
如果用户希望调用一些东西，就告诉monitor，monitor来调用