计算机组成原理

第一章

1.1计算机的基本分类，容量表示

1. **个人计算机（Personal Computer, PC）**
2. **服务器（server）**用于为多用户运行大型程序的计算机，通常由多个用户并行使用，一般通过网络访问。高端服务器称为**超级计算机**
3. **嵌入式计算机（embedded computer）**嵌入到其他设备中的计算机，一般运行预定义的一个或一组应用程序。
4. **后PC时代**
    - **个人移动设备（PMD）**连接到网络上 的小型无线设备，由电池供电，通过下载App的方式安装并运行软件。典型例子：智能手机和平板电脑。
    - **云计算** 在网络上提供服务的大服务器机群，一些运营商根据应用需求出租不同数量的服务器。替代了传统的服务器，依赖于称为仓储规模计算机（Warehouse Scale Computer，WSC）的巨型数据中心。
5. $KB = 10^3byte, Kib = 2^{10}byte$

1.2 掌握7个伟大思想及其各自的功能

1. 使用抽象简化设计 **提高产量 提高硬件和软件生产率**

    计算机架构师和程序员必须发明能够提高产量的技术，否则设计时间也将会向资源规模一样按照摩尔定律增长。提高硬件和软件生产率的主要技术之一是使用**抽象** （ abstraction) 来表示不同的设计层次，在高层次中看不到低层次的细节，只能看到一个简化的模型。

2. 加速大概率事件 **提高性能**
3. 通过并行**提高性能**
4. 通过流水线**提高性能**
5. 通过预测**提高性能**

    在某些情况下，如果假定从误预测恢复执行代价不高并且预测的准确率相对较高，则通过猜测的方式提前开始某些操作，要比等到确切知道这些操作应该启动时才开始要快一些。

6. 存储器层次

    由于存储器的速度通常影响性能 、存储器的容量限制了解题的规模 、当今计算系统中存储器的代价占了主要部分，因此程序员希望存储器速度更快 、容量更大 、价格更便宜。

7. 通过冗余**提高可靠性**

    由于任何一个物理器件都有可能失效，因此可以通过使用冗余部件的方式提高系统的可靠性（dependable) , 冗余部件可以替代失效部件并可以帮助检测错误。

1.3 基本概念及相互关系

- **指令（机器指令）**：计算机硬件所能理解并服从的命令。
- **机器语言**：以二进制元形式表示的机器指令。
- **汇编语言**：以助记符形式表示的机器指令。
- **汇编器（assembler）**：将指令由助记符形式翻译成二进制形式的程序。
- **高级语言**：表示更接近自然语言（C、C++、Java、VB等）
    - **与汇编语言相比具有的优势：**
        1. 程序员能够更好地按人类的思维方式编程
        2. 提高了编程效率程序相对于硬件独立
- **编译器（compiler）**：将高级语言翻译为计算机所能识别的机器语言的程序。
- 汇编语言与机器语言之间的关系：汇编语言是机器语言的符号版。通常一条汇编指令对应一条机器指令。
- 高级语言 → 编译器 → 汇编语言 → 汇编器 → 机器语言

1.4 掌握计算机的五大基本组成部件，还有它各自的功能。

计算机硬件由五大部分组成，分别是：数据通路（也称为运算器）、控制器、存储器、输入设备、输出设备。

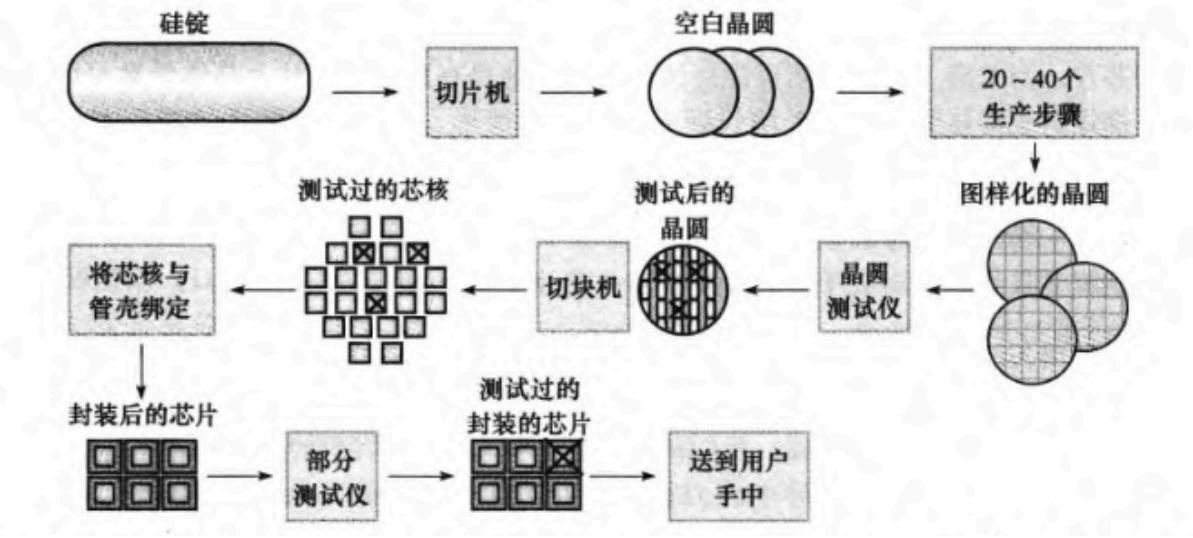
1. **数据通路（也称为运算器）**：在处理器中执行算术操作的部分
2. **控制器**：处理器中根据程序中的指令指挥数据通路、存储器和I/O设备工作的部分
3. **存储器**：程序运行时的存储空间，同时还存储程序运行时所需的数据。
4. **输入设备**：为计算机提供信息的装置
5. **输出设备**：将计算结果输出给用户或其它计算机的装置

- 数据通路和控制器是信息处理的中心部件，所以它们合称为“**中央处理单元**”（CPU：Central Processing Unit）。
- 存储器、数据通路和控制器在信息处理操作中起主要作用，是计算机硬件的主体部分，通常被称为“**主机**”。
- 输入（Input）设备和输出（Output）设备统称为“**外部设备**”，简称为外设或I/O设备。

1.5 晶体管和集成电路的基本概念

晶体管 （ transistor ）仅仅是一种受电流控制的开关。集成电路 （IC) 是由成千上万个晶体管组成的芯片。当戈登・摩尔预测资源持续翻番时，他是在预测单芯片上晶体管数量的增长速度。为了描述这些晶体管从几百个增长到成千上万的情形，形容词 “**超大规模**” 被添加到术语中，简写为 **VLSI**, 即**超大规模集成电路**（ very large-scale integrated circuit)。

硅锭：一块由硅晶体组成的棒。直径大约在 8～12 英寸，长度约 12～24 英寸。
晶圆：厚度不超过 0.1 英寸的硅锭片，用来制造芯片。



1.6 响应时间和吞吐率及改进方法、性能的度量（CPU时间、定义、平均CPI）

- **响应时间（response time）** 也叫**执行时间**，是**计算机完成某任务所需要的总时间**，包括硬盘访问、内存访问、I/O活动和CPU执行时间等。 个人计算机用户对降低响应时间感兴趣。
- **吞吐率（throughput）** 也叫**带宽（bandwidth），表示单位时间内完成的任务数量**。 数据中心比较感兴趣

> **例题**．吞吐率和响应时间 下面两种改进计算机系统的方式能否增加其吞吐率或减少其响应时间，或既增加其吞吐率又减少其响应时间？ 1．将计算机中的处理器更换为更高速的型号。 2．增加多个处理器来分别处理独立的任务，如搜索万维网。 答案 一般来说，降低响应时间几乎都可以增加吞吐率。因此，方式1同时改进了响应时间和吞吐率。方式2不会使任务完成得更快，只会增加其吞吐率。

- **CPU性能**
  - **时钟周期：**指计算机时钟间隔的时间，通常是处理器时钟，一般为常数。习惯上总是以主时钟（时钟周期）T_c来表示一个CPU的速度
  - **CPU执行时间：**简称CPU时间，执行某一任务在CPU上所花费的时间。
  - 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（为执行程序而花费在操作系统上的时间）
  - 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

    一个程序的 **CPU执行时间** = 一个程序的CPU时钟周期数/时钟频率
- **指令性能**一个程序的CPU时钟周期数 = 程序的指令数×每条指令的平均时钟周期数
- **CPI**：每条指令的平均周期数，表示执行某个程序或程序片断时每条指令所需时钟周期平均数。
- 经典的CPU性能公式：一个程序在CPU上运行所需的时间TCPU表示为$T_{CPU} = I_N \times CPI \times I_C$或$T_{CPU} = I_N \times CPI / f_c$
- T_C 表示时钟周期时间。
- CPI 表示执行每条指令所需的平均时钟周期数。
- I_N 表示要执行程序中的指令总数。

- 其中， fc为时钟频率。

第二章

RISC与CISC指令集的区别

> RISC（精简指令集计算机）和CISC（复杂指令集计算机）是两种不同的CPU架构，它们的主要区别在于指令集的复杂性和执行方式。
>
> **CISC**（Complex Instruction Set Computer，复杂指令集计算机）的特点是：
>
> - 指令系统丰富，有专用指令来完成特定的功能。
> - 在CISC微处理器中，程序的各条指令是按顺序串行执行的。
> - CISC CPU包含有丰富的电路单元，因而功能强、面积大、功耗大。
> - CISC微处理器结构复杂，功能强大，实现特殊功能容易。
>
> **RISC**（Reduced Instruction Set Computer，精简指令集计算机）的特点是：
>
> - RISC设计者把主要精力放在那些经常使用的指令上，尽量使它们具有简单高效的特色。
> - RISC对存储器操作有限制，使控制简单化。
> - RISC CPU包含有较少的单元电路，因而面积小、功耗低。
> - RISC微处理器结构简单，指令规整，性能容易把握，易学易用。
>
> 总的来说，CISC的设计目标是通过硬件来减少软件的复杂性，而RISC则试图简化硬件以提高性能。目前，许多现代处理器，如Intel的Pentium系列，都采用了CISC和RISC的混合架构。这些处理器在接收到CISC指令后，会将其分解为RISC指令，以便在同一时间内执行多条指令。这种混合架构的目标是结合CISC和RISC的优点，从软件和硬件方面取长补短。

主要指令

```
add    $t0, $s1, $s2   ;r
addi   $s3, $s3, 4     ;i
addiu  $s3, $s3, 4     ;i
sub    $s0, $t0, $t1   ;r
lw     $t0, 32($s3)    ;i
sw     $t0, 48($s3)    ;i
sll    $t2, $s0, 4     ;r
and    $t0, $t1, $t2   ;r
or     $t0, $t1, $t2   ;r
andi   $s1, $s2, 100   ;i
ori    $s1, $s2, 100   ;i
beq    $s3, $s4, L1    ;i
bne    $t0, $s5, Exit  ;i
j      Loop            ;j
slt    $t0, $s3, $s4   ;r
slti   $t0, $s2, 10    ;i
```

```
17  sltu   $t0, $s3, $s4   ;r
18  jr     $r             ;r   $r= $t4+4 * k
19  jal    ProcedureAddress;j
20  jr     $ra            ;r
21  lui    $t0, 255
```

伪指令

```
1  move $t0,$t1=add $t0,$zero, $t1
2  blt  $s0,$s1=slt $t0,$s0,$s1   bne $t0,$zero L
3  li   rdest,imm=lui $at,imm高   ori rdest,$at,imm低
```

MIPS irj三种指令格式的机器码与汇编码的转换

Field size	6bits	5bits	5bits	5bits	5bits	6bits	All MIPS instruction 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	Imm/Word address			Data transfer ,branch format
J-format	op	target address (word)					Uncondition-al jump

3. 指令格式 [R型]

例： sll $t2, $s0, 4　　#reg $t2=reg $s0<<4 bit

对应的机器语言：

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

MIPS assembly code:

```
lw   $t0, 1200($t1)   # temporary reg $t0 gets A[300]
add  $t0, $s2, $t0    # temporary reg $t0 gets  h  +  A[300]
sw   $t0, 1200($t1)   # stores h  +  A[300]  back into A[300]
```

MIPS machine language code:

Decimal version：

op	rs	rt	rd	address/ shamt	funct
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

■ 寄存器跳转指令[R型]

jr $r

op	rs	rt	rd	shamt	funct
0	rs	0	0	0	8

Name	Format	Example						Comment
Add	R	0	18	19	17	0	32	add $s1, $s2, $s3
Sub	R	0	18	19	17	0	34	sub $s1, $s2, $s3
Lw	I	35	18	17	100			lw $s1, 100($s2)
Sw	I	43	18	17	100			sw $s1, 100($s2)
And	R	0	18	19	17	0	36	and $s1, $s2, $s3
Or	R	0	18	19	17	0	37	or $s1, $s2, $s3
Nor	R	0	18	19	17	0	39	nor $s1, $s2, $s3
Addi	I	8	18	17	100			addi $s1, $s2,100
Ori	I	13	18	17	100			ori $s1, $s2,100
sll	R	0	0	18	17	10	0	sll $s1, $s2,10
srl	R	0	0	18	17	10	2	srl $s1, $s2,10
beq	I	4	17	18	25			beq $s1, $s2,100
bne	I	5	17	18	25			bne $s1, $s2,100
slt	R	0	18	19	17	0	42	slt $s1, $s2,$s3
j	J	2	2500					j 10000(see section 2.9)
jr	R	0	31	0	0	0	8	j Sra
jal	J	3	2500					jar 10000(see section 2.9)
Field size		6bits	5bits	5bits	5bits	5bits	6bits	All MIPS instruction 32 bits
R-format	R	op	rs	rt	rd	Shamt	funct	Arithmetic instruction format
i-format	I	op	rs	rt	Address			Data transfer ,branch format

使用指令编译数组访问、if、for、while、过程调用（嵌套调用、调用指令、返回指令、参数寄存器、数组起始地址给a0、返回寄存器、返回地址寄存器、栈顶指针寄存器）

- 数组访问

```
lw     $t0, 1200($t1)    # temporary reg $t0 gets A[300]
add    $t0, $s2, $t0      # temporary reg $t0 gets  h  +  A[300]
sw     $t0, 1200($t1)    # stores h  +  A[300]  back into A[300]
```

- if

```
      bne     $s3, $s4, Else   # go to Else if  i  ≠  j
      add     $s0, $s1, $s2    # f  =  g  +  h  ( Executed if  i  = =  j  if)
      j       Exit             # go to Exit
Else:sub      $s0, $s1, $s2    # f  =  g  -  h  ( Executed if i ≠ j   else)
Exit:                          # the first instruction of the next C
```

- while

```
Loop:add    $t1, $s3, $s3       # temp reg $t1 = 2 * i
     add    $t1, $t1, $t1       # temp reg $t1 = 4 * i
     add    $t1, $t1, $s6       # $t1 = address of save[i]
     lw     $t0, 0($t1)         # temp reg $t0 = save[i]
     bne    $t0, $s5, Exit      # go to Exit if save[i] ≠ k
     add    $s3, $s3, $s4       # i = i + j
     j      Loop                # go to Loop
Exit:
```

- 过程调用

```
addi    $sp, $sp,-12          # adjust stack to make room for 3 items
sw      $t1, 8($sp)           #These three instructions save three
sw      $t0, 4($sp)           # register $t1,$t0,$s0
sw      $s0, 0($sp)           #  Let's consider why it need to be done.
add     $t0, $a0, $a1         # register $t0 contains   g + h
add     $t1, $a2, $a3         # register $t1 contains   i + j
sub     $s0, $t0, $t1         # f = $t0 - $t1, which is ( g + h ) - (
  i + j )

add     $v0, $s0, $zero       # returns f ( $v0 = $s0 + 0)

lw      $s0, 0($sp)           # restore register $s0 for caller
lw      $t0, 4($sp)           # restore register $t0 for caller
lw      $t1, 8($sp)           # restore register $t1 for caller
addi    $sp, $sp, 12          # adjust stack to delete 3 items
jr      $ra                   # jump back to calling routine
```

- 嵌套调用

```
fact:addi   $sp, $sp, -8      # adjust stack for 2 items
     sw     $ra, 4($sp)       # save the return address
     sw     $a0, 0($sp)       # save the argument n
     slti   $t0, $a0, 1       # test for n < 1
     beq    $t0, $zero, L1    # if n ≥ 1, go to L1(else)
     addi   $v0, $zero, 1     # return if n <1
     addi   $sp, $sp, 8       # Recover $sp (Why not recover $ra and $a0
  ?)
     jr     $ra               # return to after jal
L1:  addi   $a0, $a0, -1      # n ≥ 1: argument gets ( n - 1 )
     jal    fact              # call fact with ( n - 1 )
     lw     $a0, 0($sp)       # return from jal: restore argument n
     lw     $ra, 4($sp)       # restore the return address
     addi   $sp, $sp, 8       # adjust stack pointer to pop 2 items
     mul    $v0, $a0, $v0     # return n*fact ( n - 1 )
```

```
15      jr      $ra             # return to the  caller
```

寄存器的分类和功能并使用在编译中

- MIPS算术运算指令的操作数必须来自寄存器。
- 寄存器大小：32位。
- 寄存器个数：32个。

寄存器编号	助记符	用途
0	zero	不管往里写入什么，总是返回0
1	at	由汇编器使用
2~3	v0,v1	用来存放子程序的返回值（非浮点）
4~7	a0~a3	用来传递子程序的前四个参数（非浮点）
8~15	t0~t7	暂存器，用来放子程序计算过程中的临时变量
16~23	s0~s7	存放子程序调用过程中需要保持不变的值
24,25	t8,t9	和t0~t8一样，据说t9可以存跳转地址
26,27	k0,k1	给异常使用
28	gp	全局指针
29	sp	堆栈指针
30	s8/fp	帧指针。据说是保存gp的上一个gp
31	ra	子程序返回地址

保留	不保留
保存寄存器：**$s0 ~ $s7**	临时寄存器：**$t0 ~ $t9**
栈指针寄存器：**$sp**	参数寄存器：**$a0 ~ $a3**
返回地址寄存器：**$ra**	返回值寄存器：**$v0 ~ $v1**
栈指针以上的栈	栈指针以下的栈

■ 相关术语：

- C 语言存储方式
  - 动态的（ automatic）
  - 静态的（static ）
- 过程帧和帧指针($fp) （动态的 ）
  - 过程帧/活动记录： 栈中包含过程所保存的寄存器和局部变量的片段。
  - $fp：指向给定过程中保存的寄存器和局部变量的值。
- 全局指针($gp)（静态的）

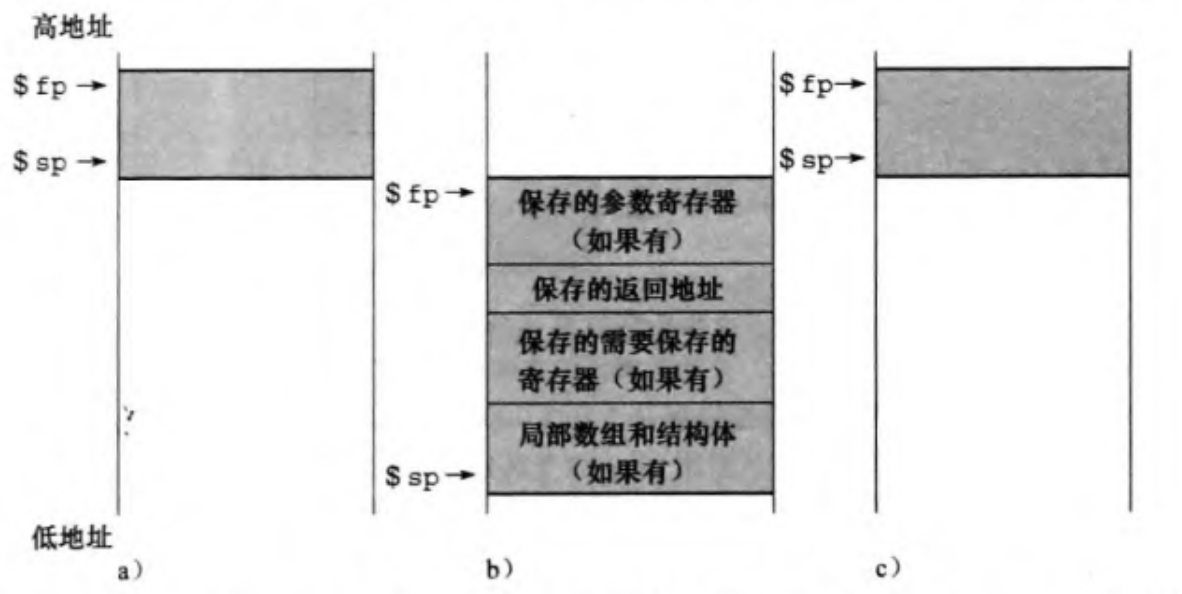
  指向静态数据区的保留寄存器

参数寄存器a0-a3(可以超过四个，通过栈的方式)

过程针的使用 fp、sp 全局指针 gp

- **过程帧**：也称作活动记录，栈中包含过程所保存的寄存器以及局部变量的片段。
- **帧指针**：指向给定过程中保存的寄存器和局部变量的值。

栈的最后一点复杂性是栈还需要存储过程的局部变量，但这些变量不适用于寄存器，例如局部的数组或结构体，栈中包含过程所保存的寄存器和局部变量的片段称为过程帧 (procedure frame) 或活动记录（activation record)。图 2-12 显示了过程调用之前、之中和之后栈的状态。



过程调用之前 (a)、之中 (b)、之后(c) 栈的分配情况。帧指针 ($fp) 指向该帧的第一个字（一般是保存的参数寄存器），而栈指针($sp) 指向栈顶。栈可调整为有足够的空间来容纳所有的保存寄存器和驻留内存的局部变量。因为在程序运行期栈指针可能会改变，所以对于程序员而言，虽然使用栈指针和少量的地址运算就可能完成对变量的引用，但使用固定的帧指针引用变量会更为简单。如果在一个过程中栈内没有局部变量 , 编译器将可以不设置和不恢复帧指针以节省时间。当使用帧指针时，在调用中使用 $sp 的地址进行初始化，而 $sp 可以使用来恢复。相关内容可以在 MIPS 参考数据卡的第 4 列找到

某些 MIPS 软件使用帧指针（frame pointer， $fp) 指向过程帧的第一个字。在过程中栈指针可能会发生改变，因此存储器中对局部变量的引用在过程中的不同位置可能具有不同的偏移量，这使得过程更加难以理解。另一种方案，帧指针在一个过程中为局部存储器引用提供一个固定的基址寄存器。注意，无论是否使用显式的帧指针，活动记录都出现在栈中。我们通过避免在过程中修改 $sp 来避免使用$fp，在我们的例子中，栈H在过程的人口和出口需要调整。

- gp全局指针

除了动态变量对过程是局部有效之外，C 程序员还需要在内存中为静态变量和动态数据结构提供空间。图 2-13 给出了 MIPS 分配内存的约定。栈由内存高端开始并向下增长。内存低端的第一部分是保留的，之后是 MIPS 机器代码的第一部分，通常称为**代码段**（text segment）。代码段之上的代码为静态数据段（static data segment），是存储常量和其他静态变量的空间。尽管数组通常具有固定长度因而能与静态数据段很好地匹配，但类似链表这样的数据结构通常会在生命期内增长或缩短。这类数据结构对应的段习惯上称为堆（heap），一般在存储器中放在静态数据段之后。注意，这种分配允许栈和堆相互增长，从而在两个段此消彼长的过程中达到内存的高效使用。

➲ 代码段：UNIX 目标文件中的段，包含源文件中例程对应的机器语言代码。

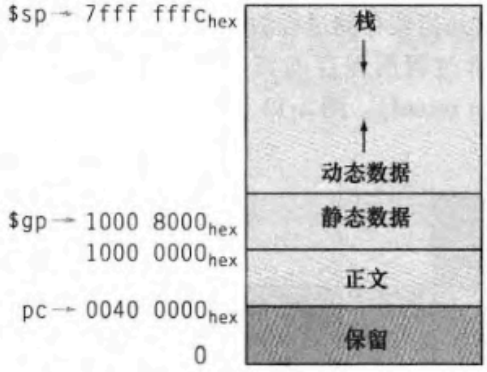


图 2-13 程序和数据的 MIPS 内存分配。这些地址只是一种软件规定，并非 MIPS 体系结构的一部分。栈指针初始化为 $7fff\ fffc_{16}$，并朝数据段的方向向下增长。在另一端，程序代码（代码段）从地址 $0040\ 0000_{16}$ 开始。静态数据从 $1000\ 0000_{16}$ 开始。然后是动态数据，在 C 中使用 malloc 命令分配，在 Java 中使用 new 命令来分配。动态数据在某一区域中朝着栈的方向向上生长，该区域称为堆。全局指针 $gp 应设置为适当地址以便于访问数据。它初始化为 $1000\ 8000_{16}$，这样通过相对 $gp 的正负 16 位的偏移量就可以访问从 $1000\ 0000_{16}$ 到 $1000\ ffff_{16}$ 之间的内存空间。关于这点可参见 MIPS 参考数据卡的第 4 列

C 语言通过显式的函数调用在堆上分配和释放空间。malloc() 在堆上分配空间并返回指向它的指针，free() 释放指针指向的堆空间。内存分配由 C 程序控制，这是很多错误产生的根源。忘记释放空间会导致“内存泄漏”，它会逐渐耗尽大量内存以至于操作系统可能崩溃。过早释放空间会导致“悬摆指针”（dangling pointer），会造成指针指向程序不想访问的位置。Java 使用自动的内存分配和无用单元回收机制来防止类似的错误发生。

32位长数的加载lui，oi，伪指令li

```
1  lui  $s0, 61          # 61 decimal  =  0000 0000 0011 1101 binary
2  ori  $s0, $s0, 2304   # 2304 decimal  = 0000 1001 0000 0000 binary
3  ----------------------------------------------------------------
4  li rdest, imm         # (imm可以为32位常数)
```

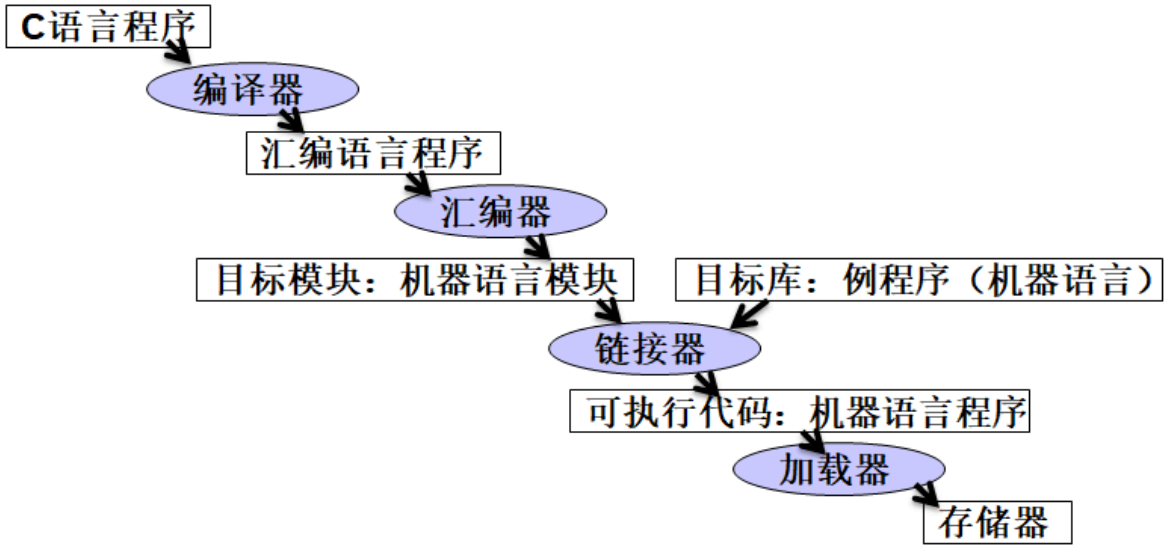
5种寻址方式

- 立即数寻址： addi $s0,$s0,4
- 寄存器寻址： add $s0,$s1,$s2
- 基址寻址： lw $s1,0($s0)
- PC相对寻址：beq $s0,$s1,L1 *PC = (PC + 4) + Branch address 16位偏移字地址左移2位形成18位分支偏移地址*
- 伪直接寻址： j Address *26位的字地址左移两位后直接拼接（PC+4）中的高4位最终形成32位跳转地址。*

条件分支的地址范围：分支前后地址范围各大约128K——2^{18}

跳转和跳转链接指令的地址范围：由PC提供高4位地址的256M大小的块中任意地址——2^{28}

编译器，汇编器，链接器，加载器



编译器 C程序-->汇编语言程序

汇编器 汇编语言程序-->目标文件（机器语言模块）伪指令：汇编语言指令的一个变种，通常被看作一条汇编指令。目标文件包括机器语言指令、数据和指令正确放入内存所需要的信息。

链接器 目标模块（包括目标库）-->可执行文件（机器语言程序）

1. 将代码和数据模块象征性地放入内存
2. 决定数据和指令标签的地址
3. 修补内部和外部引用

加载器 把目标程序装载到内存中以准备运行。

1. 读取可执行文件头来确定代码段和数据段的大小
2. 为正文和数据创建一个足够大的地址空间
3. 将可执行文件中的指令和数据复制到内存中
4. 把主程序的参数（如果存在）复制到栈顶
5. 初始化机器寄存器，将栈指针指向第一个空位置

6. 跳转到启动例程，它将参数复制到参数寄存器并且调用程序的main函数。当main函数返回时，启动例程通过系统调用exit终止程序。

数据表示，进制转换

- 二进制、八进制、十六进制相互转换

源码，补码的表示与转换，表示范围

原码表示法（符号和幅值表示法）

- 规定：最高位为符号位（0为正，1为负），其余有效数值部分用二进制的绝对值表示。
- 注意：
- 1）0可分+0和-0。 +0 为： 00…0 -0为： 10…0
- 2）符号位和数值无关，不能作为数值的一部分直接参与运算，在运算中要额外增加一步处理。
- 3）原码表示的数取不到端点 小数：|X|<1 整数：$|X|<2^n$

补码定义 [X]补 = M + X （mod M）

- 补码的性质
- 符号位是数值的一部分，可以与尾数一起直接参与运算，不需要单独处理。
- 所有负数的最高有效位都是1，硬件只需检测该位就可判断是正数还是负数。
- 数0只有一种表示，即00……0。
- 补码可以取到负方向最值。

反码表示法

- 反码表示规则：
- 正数的反码：与原码相同
- 负数的反码：符号位为1，尾数由原码按位取反。

原码与补码的互换

- 正数：补码表示与原码表示相同
- 负数：原码的符号位保持不变，其余各位取反，末位再加1。

二进制补码的相关操作：

1. 对二进制补码数取反-x = x(反) + 1
2. 符号扩展
    - 用于将一个n位表示的二进制数转化成一个多于n位表示的数。
    - 将最高有效位（符号位）以复制的方式填满高位部分。

第三章

溢出的基本概念

所谓溢出就是指运算结果大于机器所能表示的最大正数或小于机器所能表示的最小负数。

溢出的判别 定义：两操作数的数符分别为S_A、S_B，结果的数符为S_f。符号位直接参与运算，产生的符号位进位为C_f。最高有效数位产生的进位为C。

1. 溢出=$\overline{S_A S_B} S_f + S_A S_B \overline{S_f}$只有同号数相加才能产生溢出，溢出的标志是结果数符与操作数数符相反。
2. 溢出=$C_f \oplus C$　C_f和C不同时表明溢出
3. 操作数采用双符号位（变形补码），通过运算结果的符号位进行判断：

   00 — 结果为正，无溢出　　01— 结果正溢

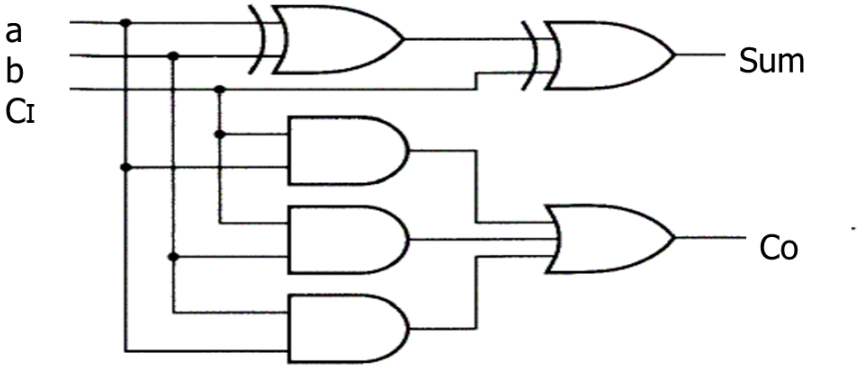
   10 —结果负溢　　11—结果为负，无溢出

ALU的基本结构与设计（并行计算可以不看）

2.一位加法器的逻辑实现

$Sum = a \oplus b \oplus C_I$

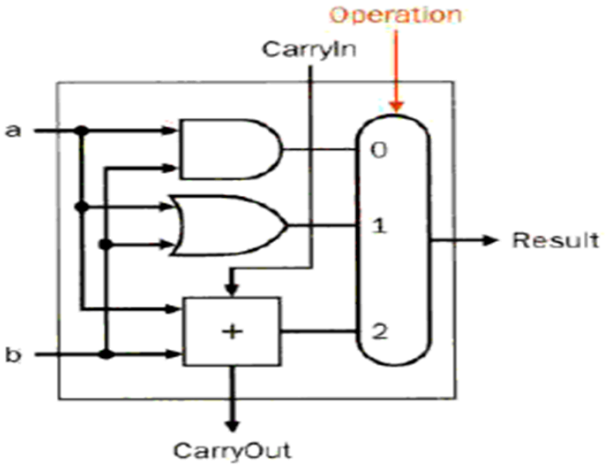
$C_O = b C_I + a C_I + ab$



二. 一位ALU

AND、OR、ADD

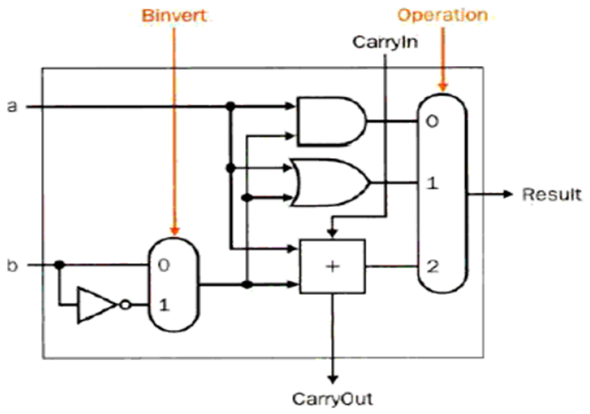
- **AND:** **Operation=00**
- **OR:** **Operation=01**
- **ADD:** **Operation=10**



三.扩展一位ALU

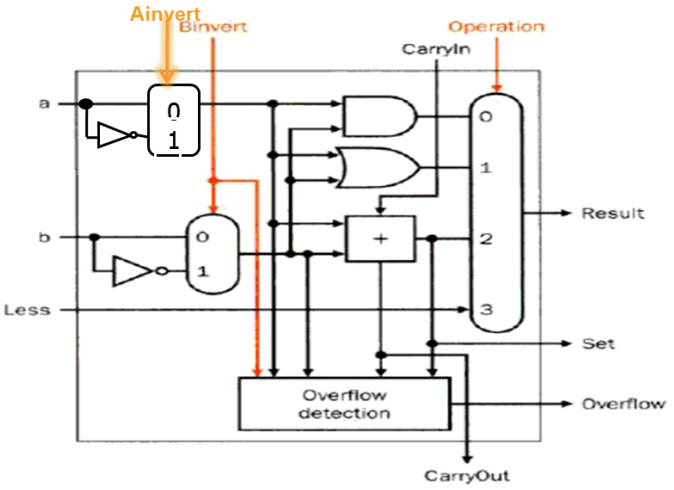
——减法

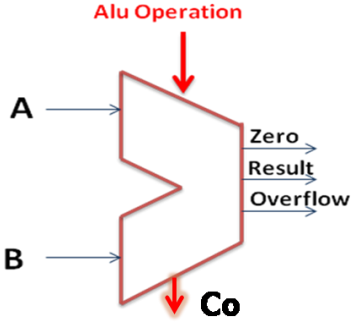
- 实现a-b
  - 将b取反（Binvert=1）
  - 初始C_1=1



四.扩展一位ALU

- 实现b-a
- 实现小于则置位（slt)
- 实现溢出检测



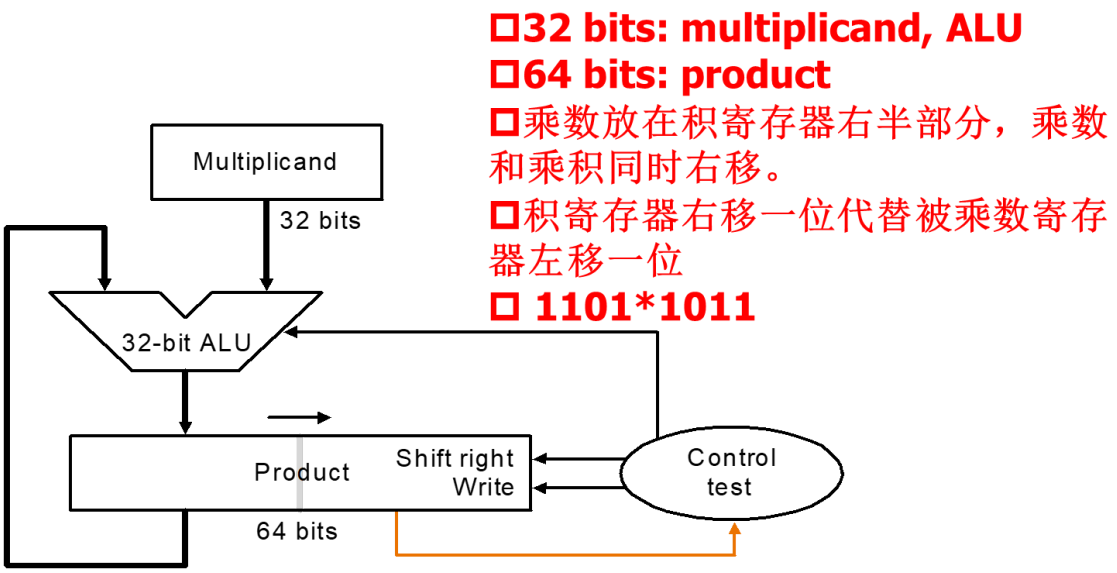


ALU功能控制表

ALU 控制线 (Ainvert Bnegate Operation) 1位 1位 2位	功能
0000	与（AND）
0001	或（OR）
0010	加（ADD）
0110	减（SUB）
0111	小于则置位（slt）
1100	或非（NOR）

改进版的乘法器的基本结构和运算

- 改进版乘法器硬件结构



  - 加法操作只在积寄存器左半部分进行
  - 整个积寄存器进行右移
- 运算步骤

```
Multiplicand:    0001
Multiplier:×     0111
                 00000111                 #Initial value for the product
1                00010111                 #After adding 0001, Multiplier=1
              ⟶  00001011   1             #After shifting right the product one bit
                 0001
2                00011011                 #After adding 0001, Multiplier=1
              ⟶  00001101   1             #After shifting right the product one bit
                 0001                     #After adding 0001, Multiplier=1
3                00011101
              ⟶  00001110   1             #After shifting right the product one bit
                 0000
4                00001110                 #After adding 0001, Multiplier=0
              ⟶  00000111   0             #After shifting right the product one bit
```

- MIPS中的乘法

  两个32位寄存器存放64位的乘积：Hi，Lo

  两条乘法指令（R型，rd=0）：

  - 乘法：mult $S2,$S3
  - 无符号乘法：multu $S2, $S3

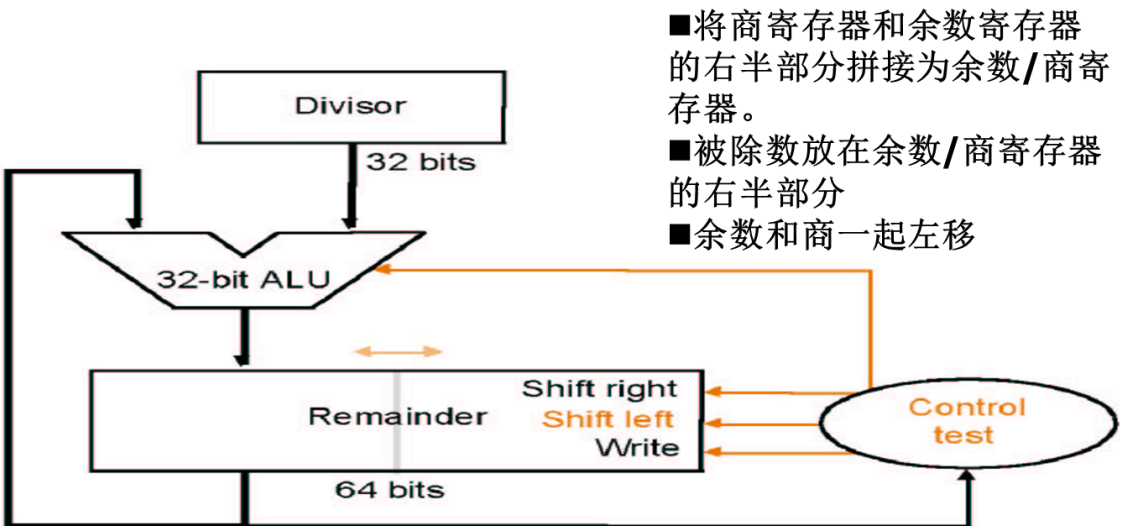
  （64位乘积存放在Hi，Lo中）

  两条取乘积指令（R型，rs=0,rt=0）：

  - 从Lo寄存器取数：mflo $S1
  - 从Hi寄存器取数：mfhi $S1

改进版的除法器的基本结构和运算

- 改进版的除法器的基本结构



- 运算步骤

迭代次数	步骤	除数	余数/商
0	初始值	0010	0000 0111
1	余数左移	0010	0000 1110
	余数=余数-除数	0010	1110 1110
	余数<0, 恢复余数,R0=0	0010	0000 1110
2	余数左移	0010	0001 1100
	余数=余数-除数	0010	1111 1100
	余数<0, 恢复余数, R0=0	0010	0001 1100
3	余数左移	0010	0011 1000
	余数=余数-除数	0010	0001 1000
	余数>0, R0=1	0010	0001 1001
4	余数左移	0010	0011 0010
	余数=余数-除数	0010	0001 0010
	余数>0, R0=1	0010	0001 0011

- MIPS中的除法

  两个32位寄存器：

  - Hi 存放余数
  - Lo 存放商

  两条除法指令：

  - 除法：div $S2, $S3
  - 无符号除法：divu $S2, $S3

单双精度浮点数的表示和十进制真值的互换

- 浮点表示法$N = (-1)^S \times (1 + Fraction) \times 2^{Exponet-Bias}$
- 其中S表示浮点数的符号（1表示负数），F称为数N的尾数，表示数N的全部有效数据，其值在0和1之间。E是一个二进制整数，称为数N的指数（或称为阶码），指明该数的小数点位置，表示数据的大小范围。
- 单精度浮点数的偏阶为127，即：*Bias*=127
- 双精度浮点数的偏阶为1023，即：*Bias*=1023
- MIPS中单精度浮点数：32位

31	30 …… 23	22 …… 0
S	exponent	fraction
1 bit	8 bits	23 bits

- MIPS中双精度浮点数：64位

31	30 …… 20	19 …… 0
S	exponent	fraction
1bit	11 bits	20 bits
31	fraction (continued)	0

- 科学记数法：十进制小数点左边只有一位整数的记数法。
- 规格化数：一个采用科学记数法表示的数，若没有前导零且小数点左边只有一位整数，则可称为规格化数。

  如： 1.210×10-9 √　0.1210×10-8 ×　12.010×10-10 ×
- 二进制规格化浮点数：尾数部分的小数点左边只保留一位非零数。
- 采用规格化科学记数法的优点：
  - 简化了浮点数的数据交换
  - 简化了浮点算术算法
  - 提高了用一个字存储的数的精度
- 偏移（移码）表示法
  - 在浮点数加减中，将两数的指数调整为相同。
  - 若指数用补码表示，不易比较其大小。
- 移码　$[E]_{移} = E + Bias$
- E为指数的真值，$Bias$为偏阶，通常 $Bias = 2^n$ ，其中，n为指数的数码位位数， 2^n是符号位的位权。

指数值范围：-126~127

单精度		双精度		表示对象
指数域	尾数域	指数域	尾数域	
0	0	0	0	0
0	非0	0	非0	±非规格化数
1~254	任何值	1~2046	任何值	±浮点数
255	0	2047	0	±无穷
255	非0	2047	非0	NaN（非数，即不是数）

浮点加法和乘法的运算步骤

- 浮点数的加法
    1. **对阶**

       使两数阶码（指数）相等(小数点实际位置对齐，尾数对应权值相同)。

       $$2^2 \times 1.1001 \rightarrow 2^3 \times 0.11001$$
       $$2^3 \times 1.1101 \rightarrow 2^3 \times 1.1101$$

    2. **尾数（有效数）相加（减）**

       $$A_M \pm B_M \rightarrow A_M$$

    3. **结果规格化，检查上溢和下溢**
- 浮点数的乘法
    1. **指数相加**
        - 若指数不带偏阶，按补码加法进行指数相加。
        - 若指数带偏阶，相加后结果减去一个偏阶
    2. **有效数相乘**
    3. **结果规格化，判断有无溢出**
    4. **进行舍入**
    5. **确定符号位值**

第四章

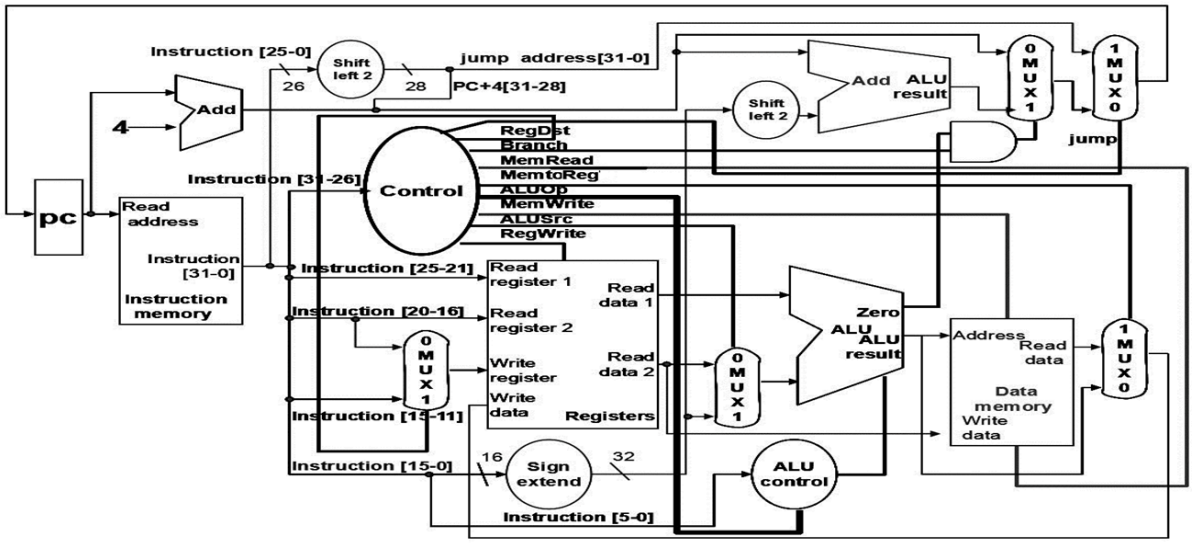
MIPS指令中各指令的操作过程

- 存储器访问指令 lw 和 sw
- 算术逻辑指令 add, sub, and, or, slt
- 分支指令 beq j

状态单元与组合逻辑单元的功能和部件

- 组合单元：一个操作单元，如与门或ALU。
    - 处理数据值
    - 输出是输入的函数
- 状态单元：一个存储单元（元件），如寄存器或存储器。
    - 存放信息
    - 至少有两个输入（要写入的数据值和决定何时写入的时钟信号）
- 主要数据通路部件：指令存储器、数据存储器、寄存器堆、ALU、加法器、程序计数器（PC）等

单周期各主要指令在数据通路的执行过程、部件、功能



ALU控制单元的设计方法和控制信号

ALU 控制信号 (Ainvert Bnegate Operation)	功能
0000	与
0001	或
0010	加
0110	减
0111	小于则置位
1100	或非

指令操作码	ALUOp	指令操作	funct字段	ALU 动作	ALU 控制信号
lw	00	取字	XXXXXX	add	0010
sw	00	存字	XXXXXX	add	0010
beq	01	相等则分支	XXXXXX	subtract	0110
R-type	10	加	100000	add	0010
		减	100010	subtract	0110
		与	100100	AND	0000
		或	100101	OR	0001
		小于则置位	101010	set-on-less-than	0111

当ALUOp为00或01时，ALU动作不依赖于funct字段。

建立真值表，最终形成门电路。$Operation_0 = ALUOp_1(F_0 + F_3)$

单周期下主控制单元各类指令控制信号的功能和值

单周期下时钟周期长度如何确定

时钟周期要由执行时间最长的那条指令决定

单周期实现也称为单时钟周期实现，即一个时钟周期执行一条指令的实现机制

每一条指令从一个时钟周期的上升沿（或下降沿）开始，在下一个时钟周期的上升沿（或下降沿）结束。

简单，容易理解，太慢不实用，效率低

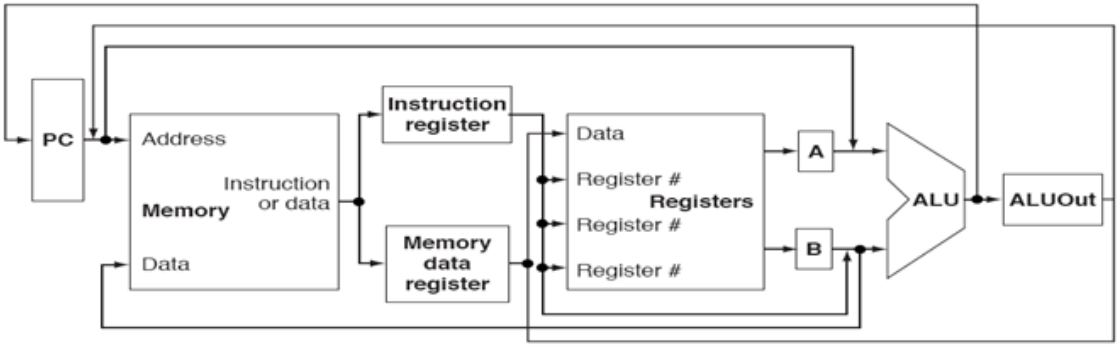
多周期的基本思想

- 将一条指令的执行过程划分成多个阶段，每个阶段占用一个时钟周期
  - 每个阶段的工作量尽量均等
  - 每个周期只使用一个主功能单元
- 在每个周期末尾：
  - 增加额外的内部寄存器
  - 暂存后续周期会使用的数据
- 不同类型的指令占用不同的时钟周期数

多周期下时钟周期长度如何确定

多周期CPU将整个指令周期分成几个时钟周期来完成，时钟周期宽度主要根据最慢的功能部件的时间来确定。

单多周期数据通路的区别减少了什么增加了什么各自的功能



- 将指令存储器和数据存储器合为一个存储器
- 将ALU和两个加法器合为一个ALU
- 在每个功能单元后都增加相应的寄存器用于暂存该单元的输出结果直到在下一个周期将其内容送往下一个功能单元

> 与多周期数据通路中只有一个存储器相比，单周期数据通路中存储器分为了独立的指令存储器和数据存储器，请分析其原因。
>
> 答：在单周期中，处理器在一个周期内只能操作每个部件一次，而在一个周期内不可能对一个单端口存储器进行两次存取。

多周期下各类指令的周期划分及RTL语言描述

- MIPS指令执行过程划分为五个周期：
    1. IF：取指
    2. ID：译码和读寄存器
    3. EX（BC）：执行/地址计算（分支完成）
    4. MEM（WB）：存储器访问（R型指令写回）
    5. WB：写回
- 取指周期（IF）

```
IR = Memory[PC];
PC = PC + 4;
```

- 译码和读寄存器周期（ID）

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut =PC+(sign-extend(IR[15-0]) << 2);
```

- 执行/地址计算周期（EX）

```
访存指令( lw / sw ):
ALUOut = A + sign-extend(IR[15-0]);
R型指令:
ALUOut = A op B;
条件分支:
if (A==B) PC = ALUOut;
跳转指令:
PC = {PC[31-28],(IR[25-0] << 2)};
```

- 访存周期（ MEM ） /R型指令写回周期（WB）

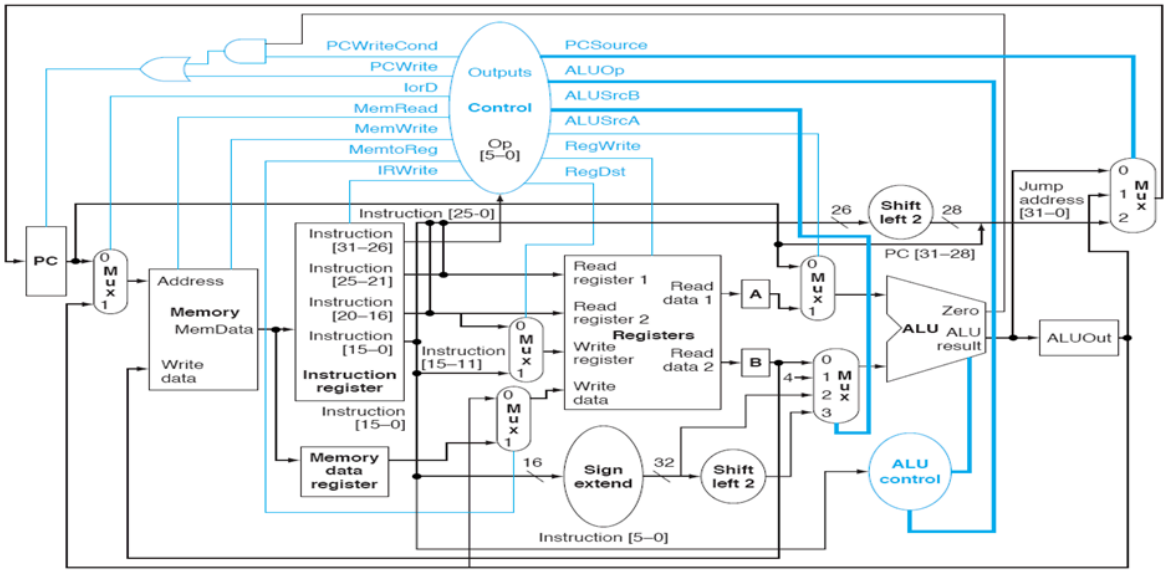
```
Load/stores指令:
MDR = Memory[ALUOut]; # for lw
Memory[ALUOut] = B;   # for sw
R型指令完成:
Reg[IR[15-11]] = ALUOut;
```

- 写回周期（WB）

```
Reg[IR[20-16]]= MDR;
```

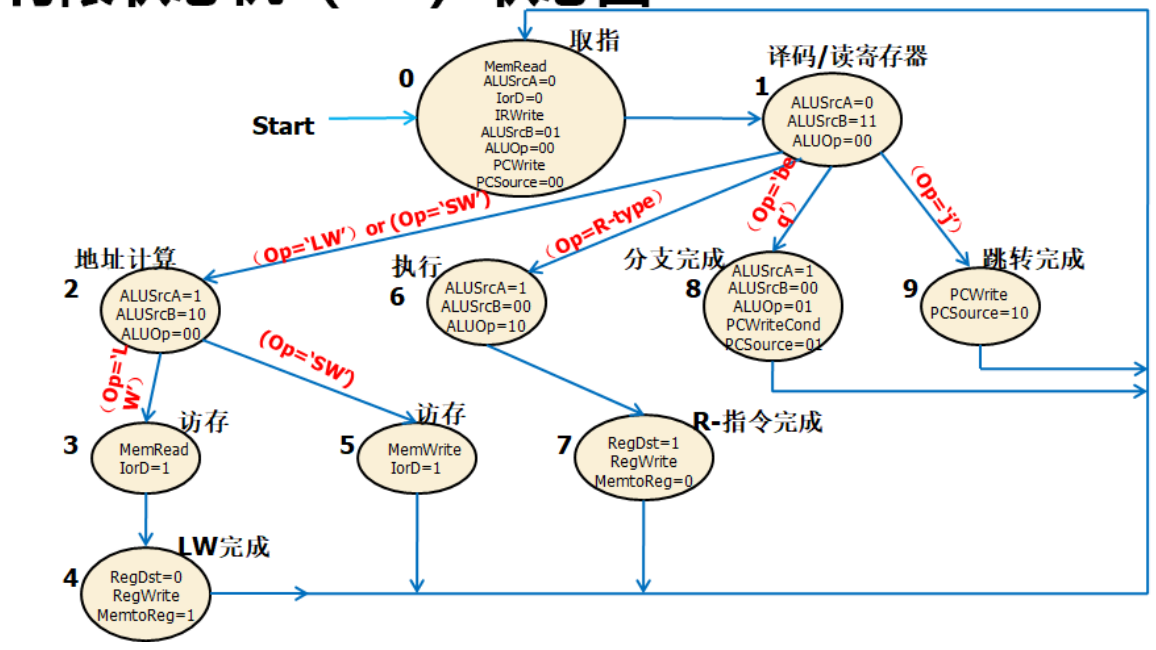
Step name	R-type	memory-reference	branches	jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend(IR[15-0])	if (A ==B) then PC = ALUOut	PC = {PC [31-28], (IR[25-0]<<2)}
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

各类指令在多周期这个数据通路下，它的执行过程中所使用到的功能部件以及相应的控制信号值



信号名	无效时(=0)的含义	有效时(=1)的含义
RegDst	写寄存器的目标寄存器号来自rt字段(20:16)	写寄存器的目标寄存器号来自rd字段(15:11)
RegWrite	无	寄存器堆写使能有效
ALUScrA	ALU 第一个操作数来源于PC	ALU 第一个操作数来源于A 寄存器
MemRead	无	数据存储器读使能有效
MemWrite	无	将写数据端（Write data）的数据写入到指定地址的存储单元中
MemtoReg	写入寄存器的数据来源于ALUOut	写入寄存器的数据来源于MDR
IorD	PC 提供存储器访问地址	ALUOut提供存储器访问地址
IRWrite	无	将存储器中取出的指令写入IR
PCWrite	无	PC写使能有效
PCWriteCond	无	带条件的PC写使能有效（Zero输出有效）
ALUOp	00	ALU执行加法运算
	01	ALU执行减法运算
	10	由 funct 字段决定ALU的具体操作
ALUScrB	00	ALU 第二个操作数来源于B 寄存器
	01	ALU 第二个操作数是常数4
	10	ALU 第二个操作数是由IR中低16位立即数字段经扩展后得到
	11	ALU 第二个操作数是由IR中低16位立即数字段经扩展后并左移两位得到
PCSource	00	ALU的运算结果(PC+4) 被送往PC待写
	01	ALUOut 中的内容(分支目标地址)被送往PC待写
	10	跳转目标地址(IR[25:0]左移 2 位与PC+4[31:28]拼接)被送往PC待写

有限状态机（FSM）状态图



能够进行灵活的扩展

> 假设要在图 2 所示的数据通路上实现一条新指令 bne $s1,$s2，L1。
> 1）是否需要增加新的硬件和控制信号？若需要，如何增加？（3 分）
> 2）该 bne 指令在第三个周期（执行周期）时，主控制单元所产生的控制位（ALUop）的值是多少？与实现 beq 指令时相同吗？为什么？ （3 分）
>
> 1）答：需要。主控制单元根据bne指令产生一个新控制信号PCWriteCondne，将ALU输出端zero信号分支经过一个非门后与PCWriteCondne信号一起进入一个与门，之后同相等则PC写使能信号（用于实现beq指令），以及PCWrite信号一起进入或门，最后送往PC。
>
> 2）答：ALUop的值为01。同beq指令相同，因为都需要控制ALU进行减法操作。

第五章

存储器的三个性能指标

容量、速度、价格

局部性定义和结构层次中的应用

提高存储性能的基本思路：利用局部性原理构建存储器层次结构

存储器层次利用了时间局部性、高速缓存中以“块”装入是利用了空间局部性

- 局部性原理
  - 时间局部性：如果某个数据项被访问，那么在不久的将来它可能再次被访问。
  - 空间局部性：如果某个数据项被访问，与它地址相邻的数据项可能很快也被访问。
- 存储器层次结构

  一种由多存储器层次组成的结构，存储器容量和访问时间随着离处理器距离的增加而增加。
- 三级存储体系结构

  高速缓存 — 内存 — 外存
  - 内存-外存层次：增大容量、构成虚拟存储器
  - Cache-内存层次：提高速度、构成主存储器

平均访存时间 下降因素及方法

相关概念：

- 命中率：在高层存储器中找到目标数据的存储访问比例。
- 缺失率（失效率）：在高层存储器中没有找到目标数据的存储访问比例。
- 命中时间：访问高层存储器所需要的时间，包括判断是否命中所需时间。
- 缺失代价（开销）：将相应的块从低层存储器替换到高层存储器所需的时间。

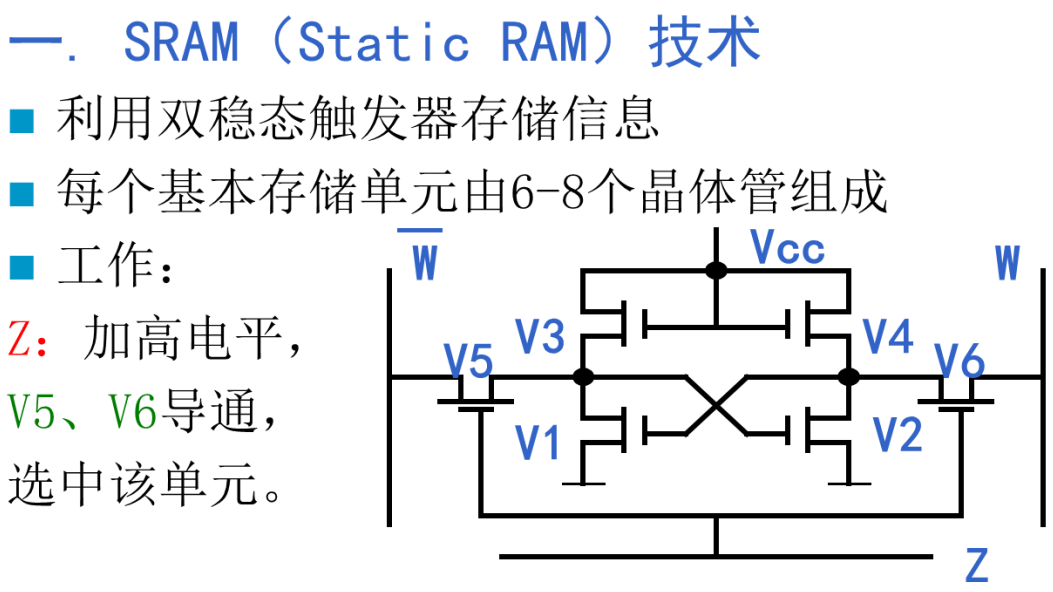
平均访存时间（AMAT）= 命中时间 + 缺失率 × 缺失代价

- 命中时间：Cache小且结构简单

- 缺失率：选择合适的块大小，提高相联度
- 缺失代价：采用两级Cache
- 存储技术，SRAM，DRAM，闪存，磁盘

存储器基本原理、相关特性（易失性，破坏性读取，刷新，最小单位，访问速度）

- SRAM（Static RAM）技术
  - 基本原理



写入：在W、W上分别加高、低电平，写1。在W、W上分别加低、高电平，写0。

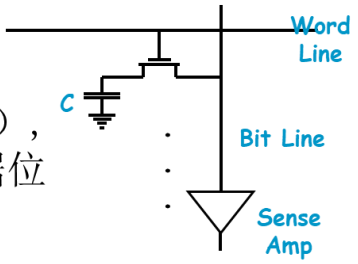
读出：根据W上有电流，读0；W上有电流，读1。

保持：Z加低电平，V5、V6截止，该单元未选中，保持原状态。

  - 相关特性
    - 最小寻址单位：1bit
    - 速度快
    - 只需最小功率即可保持电荷，无需刷新
    - 价格贵
    - 具有易失性
    - 主要用于二级Cache
- DRAM（Dynamic RAM）技术
  - 基本原理

二. DRAM（Dynamic RAM）技术

- 依靠电容存储电荷的原理存储信息
- 写
  - 字线（wordline）设为高电平，设置位线（bitline）为高（写“1”），或为低（写“0”）
- 读
  - 位线先预充电（在高低电平之间），字线设为高电平，Sense Amp根据位线电位的变化，读1/0。

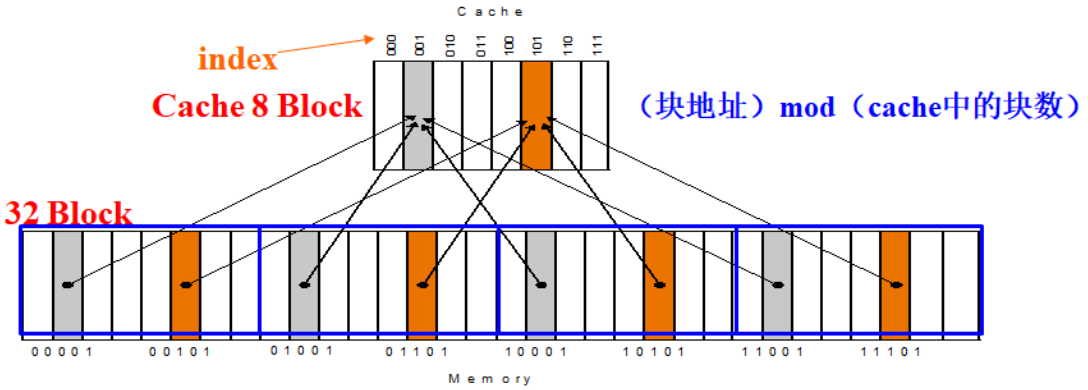


- 相关特性
  - 最小寻址单位：**1字节**
  - 速度低于SRAM
  - 价格低于SRAM
  - 具有易失性，需要刷新

    DRAM是依靠电容上存储电荷来暂存信息。平时无电源供电，时间一长电容上存储的电荷会逐渐泄露。需定期向电容补充电荷，以保持信息不变，即为刷新。
    按行刷新
  - 用作内存
- 闪存
  - 最小寻址单位：**1页**
  - 是一种电可擦除可编程只读存储器（EEPROM）
  - 具有非易失性，可以在线擦除和重写
  - 集成度高、高可靠性、抗振动
  - 单位价格在DRAM和磁盘之间
- 磁盘存储器
  - 最小寻址单位：**扇区**
  - 利用磁层上不同方向的磁化区域表示信息。
  - 容量大，记录信息可以长期保存，具有非易失性。
  - 非破坏性读出，记录介质可以重复使用
  - 顺序存取方式，速度慢

三种映射方式及各自优缺点

1. 直接映射:一种Cache结构，其中每个存储器地址仅仅对应到Cache中的一个位置。



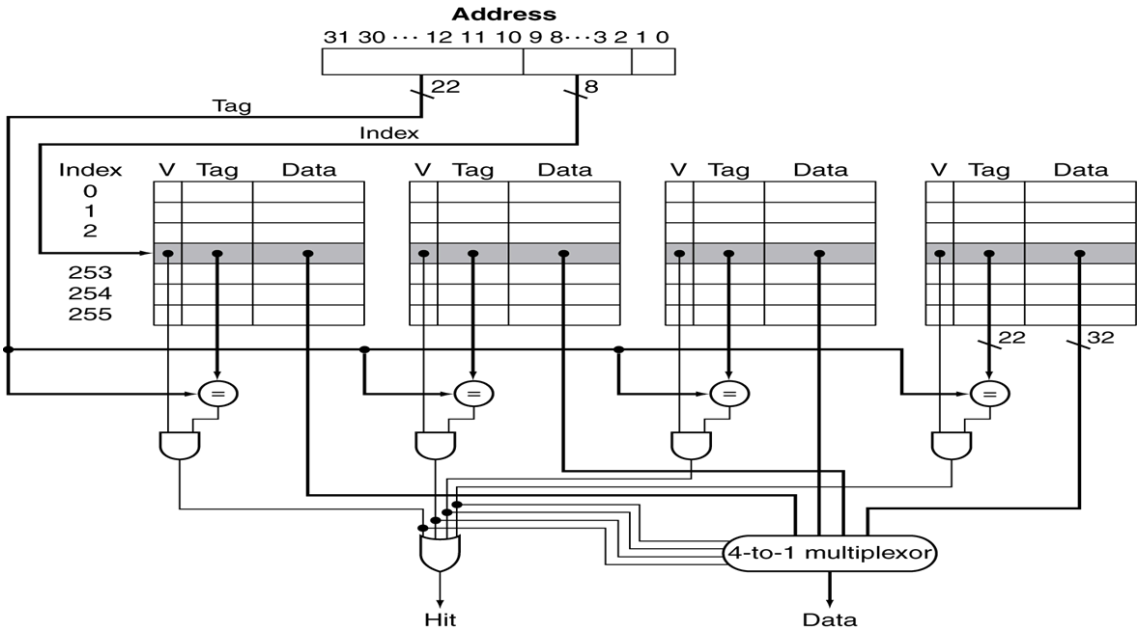
- 优点：硬件实现简单，只需比较标记位 ，速度较快。
- 缺点：Cache 空间利用率较低，块的冲突率较高。

解决办法：全相联映射和组相联映射

> 例1：假设一个直接映射的cache，可以装入16KB的数据，块大小定义为4个字，内存地址为32位，那么该cache总共需要多少位?
>
> 解：
>
> 16KB=16K个字节 1个块=16个字节
>
> 块数= $16K \div 16 = 2^{10}$ 个块，索引位(index bits)=10 bit
>
> 标记位（Tag bits）= 地址位 -索引位-字节偏移位
>
> =32-10-4 =18 bit
>
> 块的大小=4×32=128 bit 有效位（Valid bit） = 1 bit
>
> Cache总位数 =（块的大小+标记位+有效位）$\times 2^{10}$
>
> = (128+18+1) ×210 = 147×2^{10} = 147Kb
>
> =18.4KB
>
> 所以Cache容量18.4KB，是数据量的1.15倍。

> 例2：假设一个cache中有64个块，每块大小为16 字节，采用直接映射。那么主存1200号单元所在主存块将被映射到cache中的哪一块?
>
> 解：
>
> 块地址= 字节地址 ÷ 每块字节数
>
> = 1200 ÷ 16
>
> =75
>
> Cache块号=（块地址）mod（cache中的块数）
>
> = 75 mod 64 = 11
>
> 所以映射到cache中第11块。
>
> u注意：块地址= 字节地址 ÷ 每块字节数 为向下取整，如字节地址为1211的数据，块地址仍为75。

2. 全相联映射
3. 组相联映射



例1：对于具有2GB主存，128KB高速缓存的32位MIPS机器，块大小为64B，当CPU访问内存地址为01000000 00010001 00001011 10000101时，如果高速缓存采用直接映像的话，此地址映射到Cache的行号是多少？如果高速缓存采用2-Way(每行2块)组相联映像的话，此地址映射到Cache的行号又是多少？

解：

- 直接映像：
  块大小为64B =2^6个字节，字节偏移位=6bit，
  即：01000000 00010001 00001011 10000101
  Cache的块数： 128KB ÷64B = 2K = 2^{11}
  索引位占11bit， 即：01000000 00010001 00001011 10000101
  所以：此地址映射到Cache的行号是10000101110
- 2-Way(每行2块)组相联映像：
  块大小为64B =26个字节，字节偏移位=6bit，
  即：01000000 00010001 00001011 10000101
  Cache的组（set）数： 128KB ÷（64B*2 ）= 1K = 210
  索引位占10bit，
  即：01000000 00010001 00001011 10000101
  所以：此地址映射到Cache的行号是0000101110

例2：某计算机M具有4GB内存，采用两级Cache，块大小均为64B 。一级Cache的容量为32KB，采用4-way组相联映像；二级Cache的容量为512 KB，采用8-way组相联映像。请分别计算一级Cache和二级Cache的Tag，Index, Offset的位数？

解：

内存容量为：4GB = 2^{32}B，内存地址为32位。

块大小为：64B =2^6个字节， Offset位=6bit。

二级Cache ：

Cache的组数： 512KB ÷（64B*8 ）= 1K = 2^{10}

Index位=10bit，Tag位=内存地址位- Index位- Offset位

=32-10-6=16bit

- 直接映射、组相联映射和全相联映射的比较：
  - 直接映射：查找速度最快，命中时间最短，但缺失率最高。
  - 全相联映射：缺失率最低，但查找速度最慢，命中时间最长。
  - 组相联映射：介于两者之间。

主存地址的划分、相关位数的计算

例1：假设一个直接映射的cache，可以装入16KB的数据，块大小定义为4个字，内存地址为32位，那么该cache总共需要多少位?

解：

16KB=16K个字节 1个块=16个字节

块数= $16K \div 16 = 2^{10}$个块，索引位(index bits)=10 bit

标记位（Tag bits）= 地址位 -索引位-字节偏移位

=32-10-4 =18 bit

块的大小=4×32=128 bit 有效位（Valid bit） = 1 bit

Cache总位数 =（块的大小+标记位+有效位）$\times 2^{10}$

= (128+18+1) ×210 = 147×2^{10}= 147Kb

=18.4KB

所以Cache容量18.4KB，是数据量的1.15倍。

例2：假设一个cache中有64个块，每块大小为16 字节，采用直接映射。那么主存1200号单元所在主存块将被映射到cache中的哪一块?

解：

块地址= 字节地址 ÷ 每块字节数

= 1200 ÷ 16

=75

Cache块号=（块地址）mod（cache中的块数）

= 75 mod 64 = 11

所以映射到cache中第11块。

u注意：块地址= 字节地址 ÷ 每块字节数 为向下取整，如字节地址为1211的数据，块地址仍为75。

例1：对于具有2GB主存，128KB高速缓存的32位MIPS机器，块大小为64B，当CPU访问内存地址为01000000 00010001 00001011 10000101时，如果高速缓存采用直接映像的话，此地址映射到Cache的行号是多少？如果高速缓存采用2-Way(每行2块)组相联映像的话，此地址映射到Cache的行号又是多少？

解：

- 直接映像：
  块大小为64B =2^6个字节，字节偏移位=6bit，
  即：01000000 00010001 00001011 10000101
  Cache的块数： 128KB ÷64B = 2K = 2^{11}
  索引位占11bit， 即：01000000 00010001 00001011 10000101
  所以：此地址映射到Cache的行号是10000101110
- 2-Way(每行2块)组相联映像：
  块大小为64B =26个字节，字节偏移位=6bit，
  即：01000000 00010001 00001011 10000101
  Cache的组（set）数： 128KB ÷ （64B*2 ）= 1K = 210
  索引位占10bit，
  即：01000000 00010001 00001011 10000101
  所以：此地址映射到Cache的行号是0000101110

例2：某计算机M具有4GB内存，采用两级Cache，块大小均为64B 。一级Cache的容量为32KB，采用4-way组相联映像；二级Cache的容量为512 KB，采用8-way组相联映像。请分别计算一级Cache和二级Cache的Tag，Index, Offset的位数？

解：

内存容量为：4GB = 2^{32}B，内存地址为32位。

块大小为：64B =2^6个字节， Offset位=6bit。

二级Cache ：

Cache的组数： 512KB ÷ （ 64B*8 ） = 1K = 2^{10}

Index位=10bit，Tag位=内存地址位- Index位- Offset位

=32-10-6=16bit

LRU替换算法

最近最少使用法（LRU）：这种算法又称为最久未使用法。选择近期最久没有被访问的块作为被替换的块。

例：2路组相联缓存的容量为16B，块大小为2Bytes，缓存替换采用LRU策略，当前缓存状态见下表。

index	Way 0				Way 1			
	Tag	Off 1	Off 0	LRU	Tag	Off 1	Off 0	LRU
00	00	42	AC	0	11	45	A1	1
01	10	1E	33	0	00	8C	12	1
10	11	32	34	0	01	31	41	1
11	10	76	72	1	00	34	63	0

假设内存中的存放数据如下，若按内存地址00011读取后，缓存的状态是什么？

注意：LRU=1，表示最近被访问过。

地址：00011

Tag Index Offset

Address	Data
00000	AB
00001	3F
00010	12
00011	8C

Cache性能评估

在评价计算机性能时用 响应时间表示计算机完成某任务所需时间，用吞吐量表示计算机单位时间完成任务的数量。

CPU时间 =（CPU执行时钟周期数+存储器阻塞的时钟周期数）×时钟周期

存储器阻塞的时钟周期数=存储器访问次数×缺失率×缺失代价

或

存储器阻塞的时钟周期数=指令数×缺失率×缺失代价

奇偶校验、汉明校验构造和检错和纠错

设校验码为N位，其中有效信息为k位，校验位为r位，分成r组作奇偶校验，产生r位检错信息。这r位检错信息构成一个指误字，可指出2r种状态，其中一种状态表示无错，剩下的2r－1种状态可指出2r－1位中某位出错。

所以 $N = k + r <= 2^r - 1$

码距的定义和计算对应校验功能

码距（汉明距离）：码距指任何一种编码的任两组不同二进制代码中，其对应位置的代码最少有几个二进制位不相同。

- 若码距d为奇数，则能发现d-1位错，或能纠正(d-1)/2位错。
- 若码距d为偶数，则能发现d/2位错，或能纠正d/2-1 位错。