计算机系统原理 Review

RandomStar

非常天空

0. 计算机基本知识

- 计算机系统原理 (feat Mr.Lou) 荣获感动软工十大垃圾课前三名
- 软件的分类
  - System software 系统软件
    - OS 操作系统
    - Database 数据库
    - Server 服务器
    - Compiler：以 C 语言为例，编译器将 C 语言转换成汇编语言，再由汇编器转换成 machine language 二进制串
  - Application software 应用软件
    - 其他各种用途的东西
- 计算机硬件基本知识
  - Motherboard **主板**包含 I/O devices，memory，processor 三个部分
    - Memory：用于存储 running programs and data needed，每块内存包含 8 个集成电路
    - Processor：用于处理运算等操作，CPU(central processor unit)

      内部结构
- 计算机的 5 个基本组成部分
  - Input，Output，Memory，Data path，Control
- 计算机性能评价参数

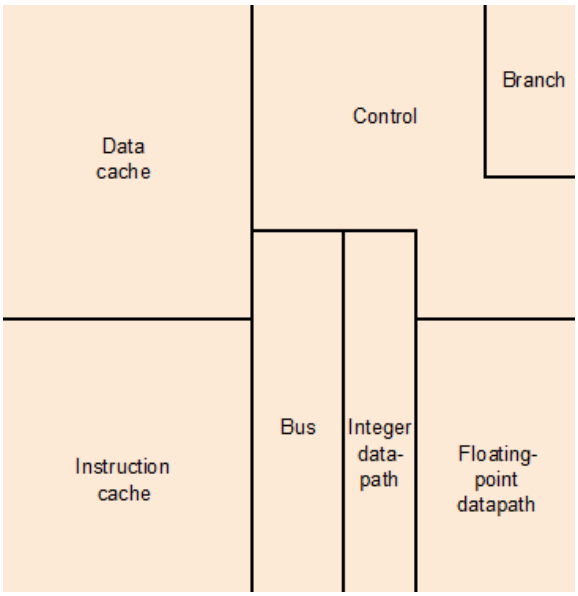


图 1: image-20200520152830447

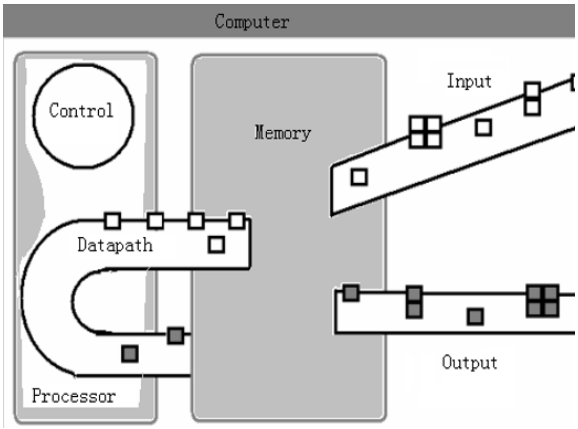


图 2: image-20200807165642385

- 主频：CPU 内数字脉冲信号震荡的频率
- CPU 时钟周期：节拍脉冲或者 T 周期，是处理操作的最基本的单位
- 机器字长：计算机可以直接处理的二进制数据的位数，位数大则运算精度高
- 响应时间
- 吞吐量：系统在一定时间内能够处理的请求数量 (对于总线和存储器有不同的含义)

- 计算机中的信息表示
  - bit 是最小的单位，1byte=8bits，1kb=1024bytes，依次往上
  - 1 个字节就是 8 位二进制，而汉字在计算机中占 2 个字节

1. MIPS 指令系统

1.1 基本的 MIPS 指令

- Operands 操作数
  - 算术指令的操作数一定存放在在寄存器 (registers) 中
    - MIPS 体系中有 32 个寄存器，每个寄存器有 32bits
  - 寄存器中可以用 `$s0-$s7` 来存储变量，其**编号为 16-23**
    - 这和 C 语言中的 `register int i;` 进行的操作相同
  - 可以用 `$t0,$t1,…`来存储一些临时变量，编号为 8-15，各寄存器的功能和对应编号如下
- Data transfer instructions 数据传输指令
  - 需要获取数据在内存的地址
  - 读取数据用 lw，存储数据用 sw，**寄存器中存储了数据在内存中的地址**
  - Offset 偏移量：数据传输指令中的常数，表示读取多少位
  - Base Register：用于存储**基地址**的寄存器 (比如数组的基地址)

Name	Register number	Usage
$zero	0	the constant value 0
$at	1	Reserve for assmbler
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	Reserve for Operating
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

图 3: image-20200228102620662

    * MIPS 中的地址偏移量和数组下标之间的关系是 **4 倍**，比如数组 A 的基地址在 `$s3` 中，要获取 A[8]，则需要指令 `lw $t0,32($s3)`
    * 原因是内存中一次只能读出 **4 字节内存中的一行**，也就是 1word 的长度，事实上一条 MIPS 指令的长度就是 1word

- MIPS 指令对应的机器码格式
  - R 型指令：`OP+RS+RT+RD+shamt+funct`
  - I 型指令：`OP+RS+RT+address`
  - OP 是对应的**操作码**，RS,RT 分别是第一个和第二个操作寄存器，RD 是**目标寄存器**，shamt 是偏移量，func 是函数对应的编码
    * R 型指令的操作码没记错的话都是 `00000`
  - 所有的 **MIPS 指令都是 32 位**的，其中 OP6 位，寄存器都是 5 位，shamt 是 5 位，funct 是 6 位，address 是 16 位
    * 这是设计的原则：good design demands good compromises
    * 计算机的两条原则
      · 指令用数字表示
      · 程序可以存储在内存中，像数字一样读写

1.2 条件判断指令

- 选择判断型指令
  - `beq register1,register2,L1` 和 `bne register1,register2,L1`
    - L1 是一个行标号，可以在每行 MIPS 指令前写，跳转的时候就按照标号所在的位置进行跳转
    - 相当于 C 语言中的 goto
- 例子：实现一个简单的 loop，其中变量 g,h,i,j 存储在 s1-s4 而数组 A 的基地址存储在 s5

```
Loop:        g=g+A[i];
                i=i+j;
                if (i!=h) goto Loop;
```

```
Loop:add $t1,$s3,$s3 #t1=2*i
   add $t1,$t1,$t1
   add $t1,$t1,$s5 #address if A[i]
   lw $t0,0($s1)
   add $s1,$s2,$t0 #g=g+A[i]
   add $s3,$s3,$s4 #i=i+j
   bne $s2,$s3,Loop
```

- 跳转指令 jr，用法是 `jr $r` r 的计算方式是当前所在地址 +4* 要跳转的行数
  - 一个函数结束的时候一定要写 `jr $ra` 来返回主函数

1.3 Procedure Instructions 过程调用指令

- 用于过程调用的寄存器和指令
  - `$a0-$a3` 是 4 个传递参数到函数的寄存器
  - `$v0-$v1` 是 2 个用于存储返回值的寄存器
  - `$ra` 用于返回地址的寄存器
  - `jal` (jump and link) 用于跳转到一个函数中，后面的参数为要跳转的地址，使用 `jar $ra` 返回主函数

- `$sp` 一个栈指针，MIPS 汇编中的栈从高地址往低地址扩展，支持 Push 和 Pop 两种操作

1.3.1 caller-saved 和 callee-saved 寄存器

- Caller-saved register 易失性寄存器：用于保存每个调用过程中**不需要在各个调用之间保留的**临时变量
- Callee-saved register 非易失性寄存器，用于**保存**需要在每个调用过程中保留的临时变量
  - MIPS 在函数调用的过程中需要保留 `$s0-$s7` 和全局变量 `$gp` 和栈指针 `$sp` 还有 `$fp`

1.3.2 案例 1：编写一个简单的函数调用

```
int  leaf_example ( int  g,  int  h,  int  i,  int  j )
{
    int f;
    f=(g+h)-(i+j);
    return f;
}
```

- 其对应的 MIPS 汇编代码如下
  - Assume: g ~ j ---- r0 ~ r3 f ---- r4

```
Leaf: addi $sp,$sp,-12  #adjust stack to make room for 3 items
        sw $t1,8($sp)
        sw $t0,4($sp)
        sw $s0,0($sp)
        add $t0,$a0,$a1 #g+h
        add $t1,$a2,$a3 #i+j
        sub $s0,$t0,$t1 #f=(g+h)-(i+j)
        add $v0,$s0,$zero #return value in v0
        lw * 3 #restore register for caller
        add $sp,$sp,12
        jr $ra
```

- 往往在函数调用的时候把下一行的地址写在寄存器 ra 中，因此函数运行结束之后可以用 `jr $ra` 指令返回到函数调用的下一句继续执行程序
- 函数编写的时候，s0-s7 的寄存器需要被**保护**，不能在函数中使用使用的，t0-t9 存储临时变量，可以视情况调用

1.3.3 案例 2：循环调用 Nested Procedure

- 简单的递归函数

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

- 对应的 MIPS 汇编如下：

```
fact: add $sp,$sp,-8
sw $ra,4($sp)
sw $a0,0($sp)
slti $t0,$a0,1 #test for n<1
beq $t0,$zero,L1
add $v0,$zero,1 #return 1
add $sp,$sp,8
jr $ra #return to after j
L1: addi $a0,$a0,-1
jal fact #call fact with n-1
2000: lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp,$sp,8
mul $v0,$a0,$v0
jr $ra
```

- `$fp` 提供了一了一个 stable base register
- `$gp` 指向静态变量 static variables

1.4 Character Instruction 字符指令

- 操作字节的 MIPS 指令
  - Load byte `lb $t0,0($sp)` 读取字节
  - Store byte `sb $t0,0($sp)` 存储字节
  - MIPS 中表示字符串的三种方法
    - 在 string 的开头先写好长度
    - 维护一个表示 string 长度的变量
    - 在末尾使用结束标志表示一个字符串的末尾 (C 语言的 choice)、
  - 同时还有

案例 strcpy 的 MIPS 实现

```
void strcpy(char x[], char y[])
{
    int i=0;
    while((x[i]=y[i])!=0)
        i+=1;
}
```

- 其对应的 MIPS 代码如下，其中 x 和 y 的首地址位于 a0 和 a1

```
strcpy: addi $sp,$sp,-4
sw $s0,0(sp)
        add $s0,$zero,$zero #i=0
L1: add $t1,$a1,$s0 # address of y[i] in $t1
lb $t2,0($t1)
add $t3,$a0,$s0
sb $t2,0($t3)
add $s0,$s0,1
bne $t2,$zero,L1
lw $s0,0($sp) # end of the string
add $sp,$sp,4 # pop 1 word off the stack
```

```
jr $ra #return
```

1.5 寻址模式

- Immediate addressing 立即数寻址
  - 在 I 型指令中出现，I 型指令中的后 16 位作为一个二进制数字来使用
  - 立即数有时候是地址，有时候是运算数
- **跳转**寻址
  - 比如 J 指令前六位是 000010，后面 26 位就表示要跳转到的地址
    - 这里的 26 位需要乘以 4 变成 PC form
- 分支语句的寻址——相对寻址
  - 对于 bne 之类的分支语句的寻址，计算机采用**相对寻址**
    - PC(Program Counter) 计算方式是 `PC+4+offset*4`，offset 是分支指令后 16 位
  - 原因是每条 MIPS 指令都是 4 字节长
- MIPS 中的寻址方式的总结
  - 寄存器寻址：通过寄存器的编号来找到对应的寄存器，常见于 R 型指令
  - 基地址寻址：用寄存器中存储的地址到内存中去寻址，比如 lw 和 sw
  - 立即数寻址：通过 I 型指令中的**立即数**来寻址
  - PC 相对寻址：通过 PC+4 寻址
  - 伪直接寻址：J 型指令中经常出现

图 4:

1.6 MIPS 指令总结

- MIPS 指令的形式
- MIPS 指令的 Operands

Name	Fields					Comments
Field size	6 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer,branch,imm. format
J-format	op	target address					Jump instruction format

图 5:

  - 32 registers
  - Memory words

- MIPS 编程中的注意点
  - 汇编程序可以不遵循编程原则
  - 函数可以直接跳出
  - 形参超过四个可以用栈来存储参数，在函数中弹出栈，返回值超过 2 个也要用堆栈，当然可以打破规则用闲置的寄存器传递参数
  - 传递参数和获取返回值的寄存器 a0-a3 和 v0-v1 一般不直接参与运算
- C 语言程序在计算机中的编译过程
  - 编译 compiling：将 C 程序转化为汇编语言程序
  - 汇编 assembling：将汇编语言程序转化为机器码
  - 链接 linking：将 object-modules(含各个库) 转化成可执行程序

Extra：文件的一些背景知识

- 文件存储在硬盘的逻辑分区中，具体来说是扇区，一般为 32KB
  - 100 个 1 字节的文件占据了 100_{32KB} 的空间，而 100_{KB} 的内容占据了 $_{4}$\32KB
  - 文件的起始扇区存在目录中，多个扇区的顺序存在 FAT 表中，文件删除的时候需要在目录中先清空文件名，再在 FAT 表中设置各扇区为空闲
  - 目录也是一个文件，每个文件在目录中是一条记录
  - 内存/磁盘碎片整理

    * 文件分配的原理：从 0 开始，找到一块整块的能存下则放置文件，否则就进行分散分配
    * 内存用于执行某个程序，则从 0 开始，找到一块整块的能存下，则分配邮与文件删除和内存释放造成碎片

- 链接库
  - 分为静态链接库和动态链接库，静态链接库可以直接链接，动态链接库作为外部调用
  - Windows 系统中动态链接库的后缀是 dll，静态链接库的后缀是 lib
  - Linux 系统中动态链接库的后缀是 so，静态是 a
- MIPS 体系下的内存动态分配：从低位到高位依次是
  - 保留区，Text 区
  - static data 区
  - dynamic data 区
  - stack

1.7 MIPS 需要补充的内容

1.7.1 伪指令

- 伪指令：没有基于硬件实现，而是用现有的指令组合而成的一些功能
  - `move $rd, $rs`：值的拷贝，实现的功能是 rd=rs
  - `blt $rs, $rt, RR` 比较地址的大小，如果 rt 的地址比较大则进行跳转

1.7.2 系统调用

- MIPS 中的系统调用相当于一个内部中断，调用系统程序
  - 指令的格式是 `syscall` 进行系统调用
  - 系统调用会读取 `$v0` 寄存器中的数字，根据不同的数字来执行不同的系统调用，将结果保存在 a 系列或者 f 系列的寄存器中

Service	SystemCallCode($v0)	Arguments	Result
print_int	1	$a0=integer	--
print_float	2	$f12=float	--
print_double	3	$f12=double	--
print_string	4	$a0=string	--
read_int	5	--	$v0=integer
read_float	6	--	$f0=float
read_double	7	--	$f0=double
read_string	8	$a0=buffer $a1=length	--
sbrk	9	$a0=amount	--
exit	10	--	--
print_char	11	$a0=char	--
read_char	12	--	$a0=char
open	13	$a0=filename $a1=flags $a2=mode	$a0=file descriptor
read	14	$a0=file descriptor $a1=buffer $a2=length	$a0=num chars written
write	15	$a0=file descriptor $a1=buffer $a2=length	$a0=num chars written
close	16	$a0=file descriptor	--
exit2	17	$a0=result	--

图 6:

- 系统调用的一个实例
  - 这个系统调用的作用就是把寄存器 t0 中的数字打印出来

```
li  $v0, 1           # service 1 is print integer
add $a0, $t0, $zero  # load desired value into argument register $a0, using pseudo-op
syscall
```

案例 1：实现一个 swap 函数

```
void swap(int v[],int k)
{
    int temp;
    temp=v[k];
    v[k]=v[k+1];
    v[k+1]=temp;
}
```

- 对应的 MIPS 代码如下，其中 v 的基地址在 a0，k 在 a1，temp 是 t0

```
swap: add $t1,$a1,$a1
add $t1,$t1,$t1
add $t1,$a0,$t1 # t1 has the address of v[k]
lw $t0,0($t1)
lw $t2,4($t1)
```

```
sw $t2,0($t1)
sw $t0,4($t1)
jr $ra
```

案例 2：指针和数组的区别

```
void clear1(int a[], int size)
{
    for(int i=0;i<size;i++)
        a[i]=0;
}

void clear2(int *a,int size)
{
    int *p;
    for(p=&a[0];p<&a[size];p+=1)
        *p=0;
}
```

- 第一种函数的 MIPS 实现
  - 只要 size 是正数就可以工作

```
move $t0,$zero
loop1: add $t1,$t0,$t0
add $t1,$t1,$t1 # i*4
add $t2,$a0,$t1 # address of a[i]
sw $zero,0($t2)
addi $t0,$t0,1
slt $t3,$t0,$a1
bne $t3,$zero,loop1
```

- 第二种函数的 MIPS 实现

```
move $t0,$a0 #p=&a[0]
add $t1,$a1,$a1
add $t1,$t1,$t1
```

```
add $t2,$0,$t1 #t2=&a[size]
loop2: sw $zero,0($t0) #*p=0
addi $t0,$t0,4 #p=p+4
slt $t3,$t0,$t2 # p<&array[size] ?
bne $t3,$zero,loop
```

2. Arithmetic for computer 计算机运算

2.1 数据表示：浮点数/无浮点数

2.1.1 二进制数的表示

- K 进制的数的表示方式 $N = \sum_{i=m}^{n-1} b_i \times K^i$ 其中 0-n-1 是整数位，-1 到 m 是小数位
  - 无符号数：n 位二进制无符号数可以表示的值的范围是：0 到 $2^n - 1$
  - 有符号数：把最高位用来表示数的正负，0 表示正数，1 表负数
    - 存在的问题：存在正 0(0000 0000) 和负 0(1000 0000) 两种 0
- 二进制数的一些操作
  - 对应的十进制数的计算方式是：对于 n 位有符号的二进制数，最高位的权重是 -2^{n-1}，其余位的权重是 2^{n-1}，将权重相加就得到了对应的十进制数
  - 二进制的位扩展 (sign extension)：把符号位扩展到高位的每一位
    - 比如 4 位的 1010 扩展成 8 位就是 11111010
  - 大小比较 (MIPS 指令)
    - 有符号数之间的比较用 slt 和 slti(和立即数比较)
    - 无符号数之间的比较用 sltu 和 sltiu

2.1.2 各种二进制码

- **有符号数**的三种 □：原码，反码，补码和移码
  - 能用这些码表示的都按照有符号数来处理

- 原码：由符号位 + 绝对值组成，最高位是符号位 0 或 1，其余位是原本数字的**绝对值**表示
- 反码：由原码到补码的一种中间形式
  - 正数的反码就是其本身
  - 负数的反码是**符号位不变**，其他位依次取反
- **Two's Complement** 补码
  - 正数的补码还是本身，负数的补码**按位取反之后加 1**，对于负的整数 $[X]_c = 2^{n+1} - |X|$
- overflow
  - 对于 N 位二进制数，如果数值小于 -2^{N-1} 或者大于 $2^{N-1} - 1$ 即为溢出
  - 无符号数不考虑溢出的情况
- 移码 (biased-code) 的表示
  - 由符号位 + 绝对值组成，计算的方法是**补码的最高位取反**
    - 比如-128 的补码是 1000 0000，其对应的移码就是 0000 0000
    - 127 的补码是 0111 1111，其移码就是 1111 1111

- 三种 □ 的比较 (8 位数字的情况下)

类型	原码	补码	移码
范围	-127-127	-128-127	-128-127
最小数	1111 1111	1000 0000	1000 0000
最大数 (+127)	0111 1111	0000 0000	1111 1111
0	0000 0000 1000 0000	0000 0000	1000 0000
优点	直观	加减运算方便	大小上数码完全一致
缺点	同号异号，运算麻烦	大小比较需要单独处理	符号位和别的码不同

2.2 汉字系统

- ASCII 码：美国信息交换标准码
  - 标准的 ASCII 码只有 7 位，但是为了方便计算机处理，扩展成了 1 字节 (8 位)
  - ASCII 码中有 128 个字符，其中可打印字符 96 个，控制字符 32 个
  - 后来因为 128 个字符不够又扩展了新的 128 个
  - 常见的 ASCII 码：A 位于 65，a 位于 97
- 汉字的表示
  - GB2313 区位码
  - 输入码：五笔，拼音等等
  - 字模码：
    - 用 8x8 的点阵表示 ASCII 码
    - 用 16x16 的点阵表示汉字，可以用 16 个 16 位的二进制数来表示一个汉字

2.3 Addition & Subtraction

- 加减法
  - 加法：原码直接相加，进位给下一位
  - 减法：直接相减，或者将两个数的补码相加 (此时得到的结果是补码)
  - overflow，比如 1111 1111+1111 1010 = 1 1111 1001，1000 0001 + 1111 1110 = 0111 1111

Operation	Operand A	Operand B	Result indicating overflow
$A+B$	≥ 0	≥ 0	< 0
$A+B$	< 0	< 0	≥ 0
$A-B$	≥ 0	< 0	< 0
$A-B$	< 0	≥ 0	≥ 0

图 7:

  - MIPS 中处理溢出的方式是 **interrupt**，溢出指令的地址被**保存在寄存器中**，计算机跳到预定地址以为该异常调用适当的例程。中断的地址将被保存，以便在某些情况下程序可以在执行纠正代码后继续执行

* 其他的处理方式还有：在 ALU 中进行硬件检查；interrupt 和 EPC 存储指令地址

– 符号数加法的 MIPS 代码

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor  $t3, $t1, $t2 # Check if signs differ
slt  $t3, $t3, $zero # $t3 = 1 if signs differ
bne  $t3, $zero, No_overflow # $t1, $t2 signs ≠,
                             # so no overflow
xor $t3, $t0, $t1 # signs =; sign of sum match too?
                  # $t3 negative if sum sign different
slt $t3, $t3, $zero # $t3 = 1 if sum sign different
bne $t3, $zero, Overflow # All 3 signs ≠; goto overflow
```

图 8: image-20200527154035453

– 无符号数加法的 MIPS 代码

* 取相反数的运算方式：将数字和 0 进行 nor 运算

```
addu $t0, $t1, $t2     # $t0 = sum
nor $t3, $t1, $zero    # $t3 = NOT $t1
                       # (2's comp - 1: 2^32 - $t1 - 1)
sltu $t3, $t3, $t2     # (2^32 - $t1 - 1) < $t2
                       # ⇒ 2^32 - 1 < $t1 + $t2
bne $t3,$zero,Overflow # if(2^32-1<$t1+$t2) goto overflow
```

图 9: image-20200527154102007

- 加法器的设计

– 最简单的一位 ALU：进行 and 或者 or 操作

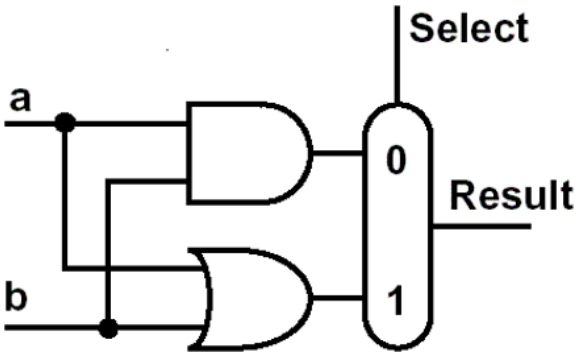


图 10:

– 一些名词的简称

* OF: overflow 判断是否溢出，最高进位和次高进位的**异或**
* CF: CarryOut 进位的值
* ZF：zero，当结果为 0 的时候 ZF=1，否则是 0
* SF：符号位的判断，正 0 负 1
* PF：奇偶校验

– 半加器 half adder(不能和进位 Carryout 进行运算)

* `sum = a xor b`, `carry = a or b` 其设计如下

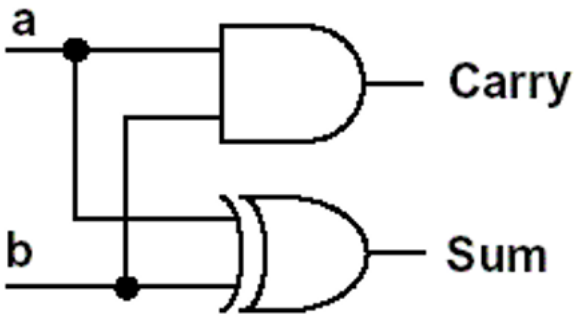


图 11:

– 全加器 full adder

* 运算规则, 本位和 $Sum = A \oplus B \oplus Carry$, 进位 $Carry = AB + BCarry + CarryA$
* 二阶全加器的设计

– 1 bit 的 ALU 的设计：包含 AND, OR , ADD 三种指令

* 可以通过输入的值 operation 来控制输出结果，比如 operation=0 输出的就是 and 运算的结果
* 第一个 Carry in 的值是 0

– 32-bit 的 ALU：实际上是 32 个 1bit 的 ALU 连接起来进行运算

* 如果是逻辑运算，就是 32 个 1bit 的 ALU 分别输出对应位上逻辑运算的结果，然后输出

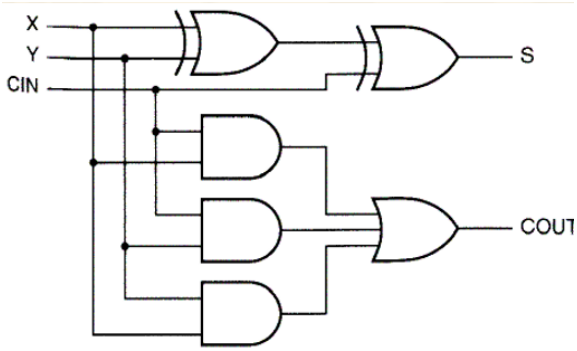


图 12: image-20200527153732006

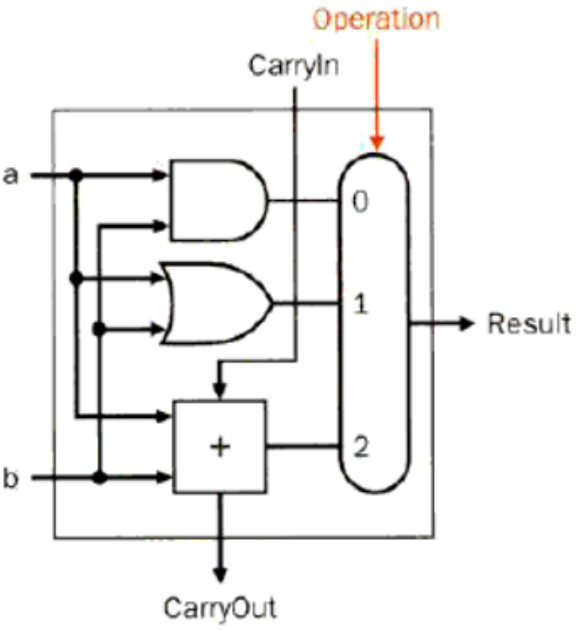


图 13: image-20200527154253716

* 如果是算术运算，Carry In 和 Carry Out 会在 ALU 之间按顺序传递下去，实现加法的进位

– 1bit ALU 的扩展：支持减法

* 此时操作码还是 3 种，参数 Binvert 控制第二个运算数是否取反

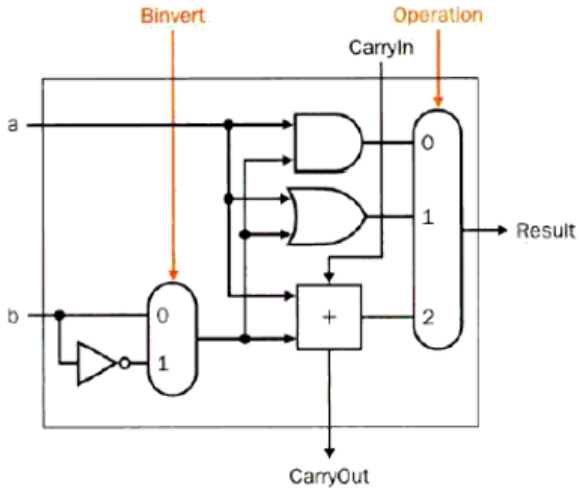


图 14:

* 由于支持减法，因此 ALU 也可以支持大小的比较，比较大小的操作码为 2

  · 加入一个 Less 输入作为大小判断时的输出

– 32-bit 的完全 ALU

* 还可以对 32 个 result 添加一个 zero 检查器

– 加法器的基本原理

* 完全串行的进位方式，一次运算需要 3 个与门，2 个或门，消耗 2 份的时间，完成 32 位全加法需要的时间是一次加法运算的的 32 倍

* 并行计算的进位方式，进位的 ci 计算全部转换成第一位的 a0,b0,c0 的 =，依次进行迭代

* 先行进位 (并行) 用 $g_i = a_i b_i, p_i = a_i + b_i$ 来简化运算

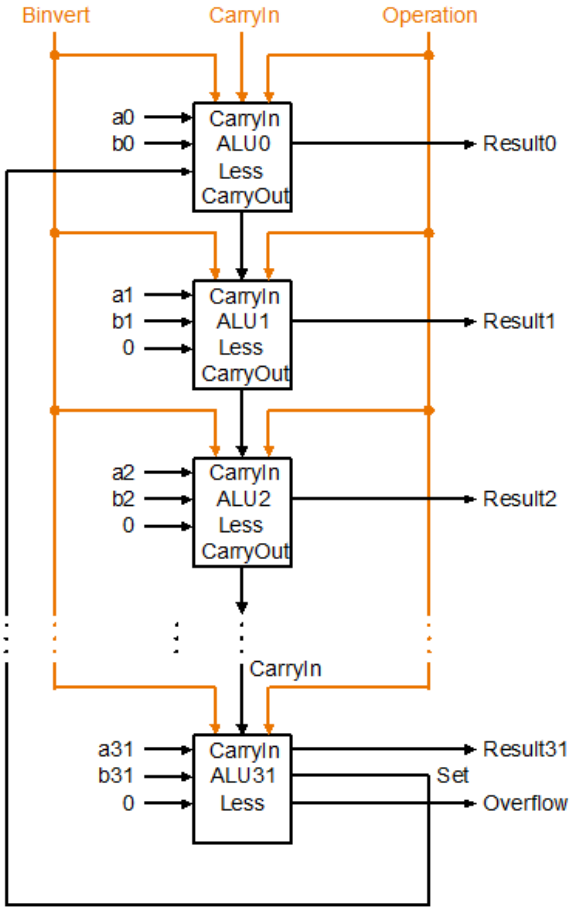


图 15: image-20200808183650715

    * 组内并行，组之间串行：4 位一组

- ALU：算术逻辑单元
  - 一个最基本的 ALU 的结构图如下图所示

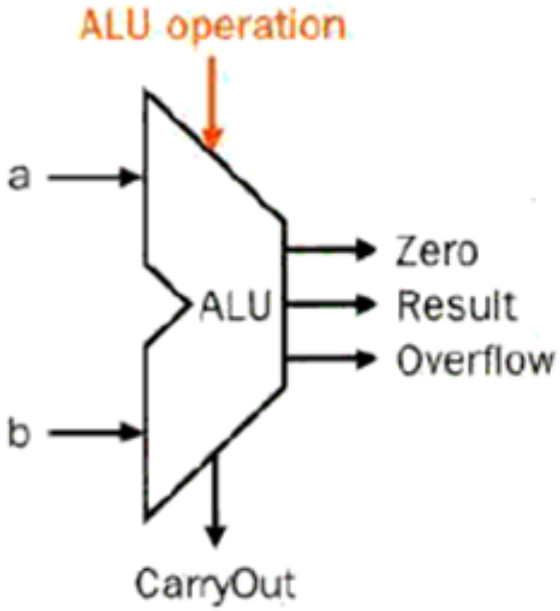


图 16: image-20200808183928282

  - ALU 控制线路：000 对应 and，001 对应 or，010 对应 add，110 对应 sub，111 对应 less than

2.4 Multiplication && Division

- 和十进制乘法一样，可以列竖式计算
  - 一一乘法口诀表：00 得 0，11 得 1，01 得 0，10 得 0
  - 乘法得两个因数分别叫做 multiplicand 和 multiplier
- 乘法算法 1
  - Binary multiplication 列竖式计算，具体过程就和十进制一样

– 逻辑图：乘数是 32 位，被乘数，积和 ALU 是 64 位

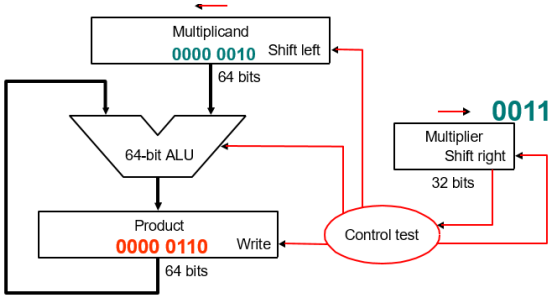


图 17: image-20200529102007327

*

* 需要 32 次迭代，计算非常缓慢，需要 64 位加法器 1 个，64 位寄存器 2 个，32 位寄存器 1 个，移位 2 次

- 乘法算法 2
  - Don't shift the multiplicand，Instead,**shift the product**，Shift the multiplier
  - 逻辑图，积的寄存器的左半边才会进行 change
  - 2 个 64 位的寄存器其中一个变成了 32 位
- 乘法算法 3
  - 1 个 32 位加法器，1 个 64 位寄存器，1 个 32 位寄存器，移位操作 1 次

2.4.1 有符号数的乘法

- 补码乘法
  - $(A \times B) = A \times B$ 已知 AB 的补码，可以把 A 转换成原码和 B 的补码进行计算
  - 无符号数的补码之积等于积的补码
- **Booth 算法** □□□
  - 基本原理：用于**二进制补码的相乘**的运算，从最低位开始，只要这串数字为 0 就不执行任何操作，当遇到第一个 1 时执行一次减法，也就是减去被乘数和该位权值的积，对于后面的 1 不进行操作，再碰上 0 就加权值，如此往复

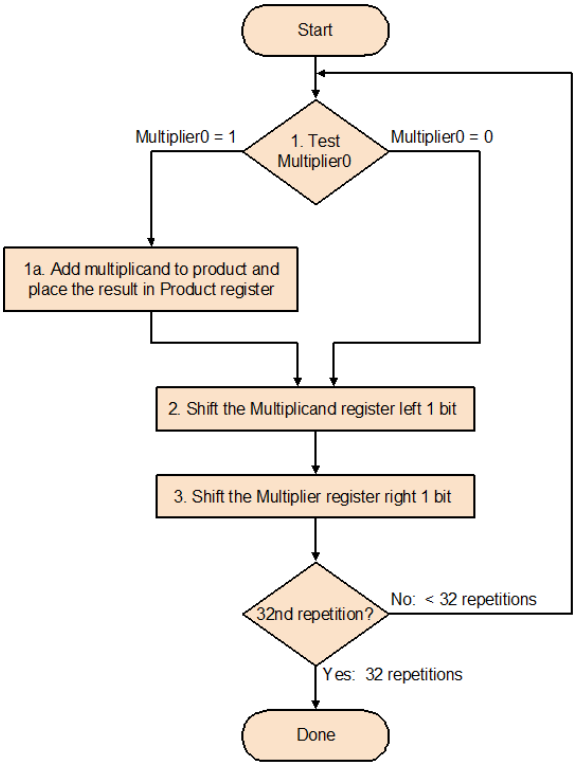


图 18: image-20200808191934189

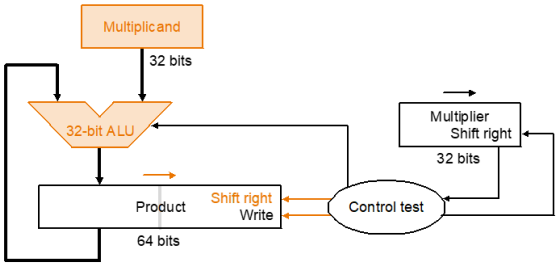


图 19: image-20200529102827830

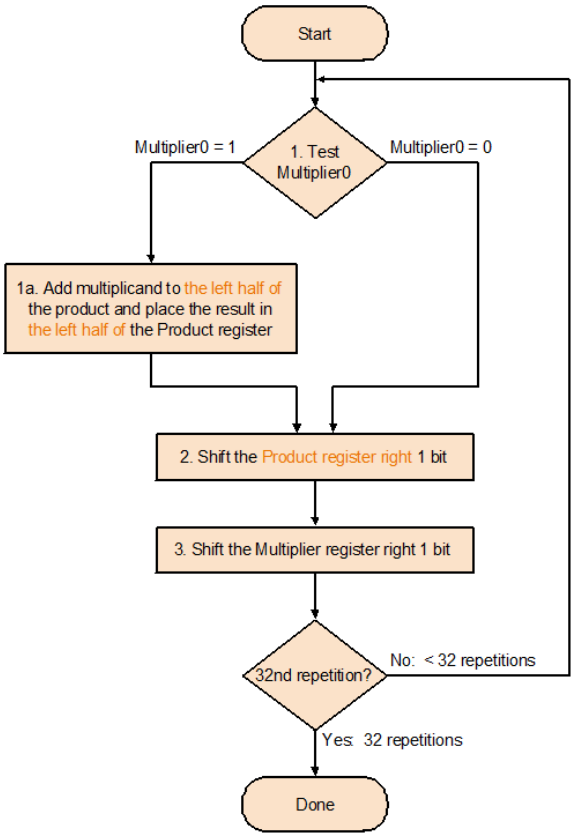


图 20: image-20200808192241507

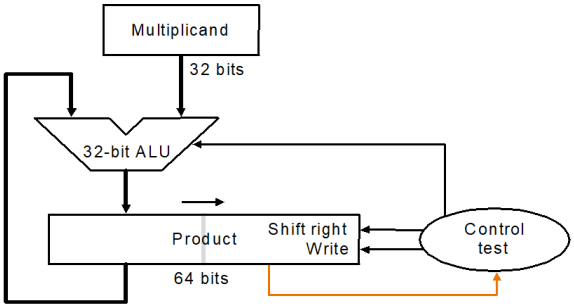


图 21: image-20200808193055418

- 两个因数都用补码的形式参加乘法运算，**结果是积的补码**
- 加速的原理
  - Booth 算法会导致 addition 减少而 shift 增加，如果 shift 的效率更高就会使得运算加快
- 也就是需要判断连续的两位
  - 10 减去 1 所在位置的权重
  - 00 不进行操作
  - 11 不进行操作
  - 01 加上 0 所在位置的权重
  - 为了方便，把第-1 位当作 0 来使用

2.4.2 二进制数的除法

- Dividend = quotient x divisor + remainder 被除数 = 商 x 除数 + 余数
- 算法 1
  - 需要 64 位加法器 1 个，64 位寄存器 2 个，32 位寄存器 1 个，移位操作两次

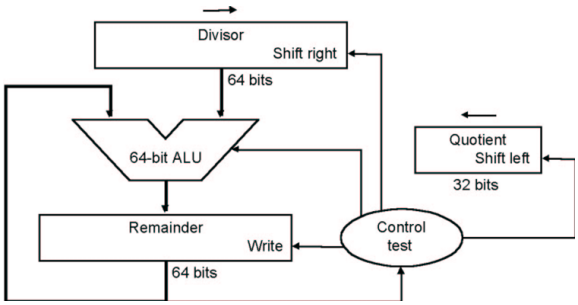


图 22: image-20200603141843168

- 算法 2
  - 需要 32 位加法器 1 个，64 位寄存器 1 个，32 位寄存器 2 个，移位操作两次
- 算法 3

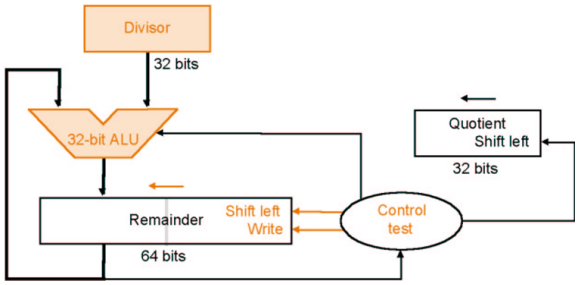


图 23: image-20200603142121548

- 需要 32 位加法器 1 个，64 位寄存器 1 个，32 位寄存器 1 个，位移操作 2 次

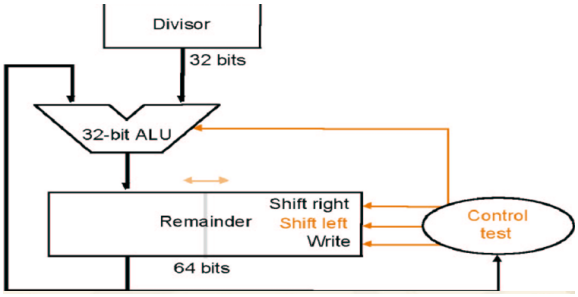


图 24: image-20200603142319359

- 有符号数的除法
  - 基本原则：计算商的时候把除数和被除数都当作正数，得出一个非负数的商，然后根据除数和被除数的符号确定商的符号，在根据关系计算余数
    - 比如 7/2=3 余 1，而-7/2=-3 余-1，7/-2=-3 余 1，-7/-2=3 余 1
  - 除数不能为 0，负责会 overflow

2.5 Float 浮点数的表示——IEEE754 标准

- IEEE 制定的舍入规则
  - ceil 向上取整
  - floor 向下取整

  - int 抹去小数部分
  - 三个特殊的数字 `NaN(Not a Number)`, 正负无穷大
- 二进制的浮点数的表示方式是 `1.xxxxxxxx * 2^yyyyyy`，类似于十进制中的科学计数法，用公式表示为 $x = (-1)^s \times 1.M \times 2^{E-127}$
  - E 是阶码，一定是一个非负数，表示的范围是 0-255(float 中)，0-1023(double 中)
  - M 也叫 significand，E 叫 exponent
  - 浮点数的精度：用 S+E+M 的顺序来表示一个浮点数
    - 单精度浮点数 (float) 中 s 占 1 位，E 占 8 位，M 占 23 位，一共 32 位
    - 双精度浮点数 (double) 中 s 占 1 位，E 占 11 位，M 占 52 位，一共 64 位
  - 浮点数进位的计算方式：
    - 先确定符号位 S，将数字取绝对值
    - 将数字转换到 1. 多的范围里，然后确定 E，获取尾数 M 的十进制表示
    - 将 M 转换成 23 位二进制，具体方法是不断 *2，如果结果超过，取整数部分作为每一位上的结果
    - **注意尾数需要四舍五入，也就是计算 24 位，如果 24 位是 1 就在前 23 位向上进 1**
- 浮点数的加减法
  - 把十进制的全部转换成二进制浮点数的原码
  - 将小数点对齐，进行加法运算
  - 说了半天其实真的算起来加减乘除都要靠转化成十进制来计算

3. CPU 结构

3.1 计算机的基本结构

- 计算机的组成结构
  - CPU

    - control unit 控制单元
    - datapath 数据通道
        - path
        - ALU 算术逻辑单元
        - registers 寄存器
  - Memory 内存
  - I/O interface 输入/输出接口
- 简单的 MIPS 实现原理

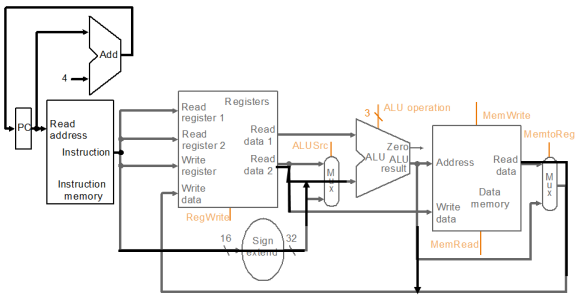


图 25: image-20200809135614210

  - 不言而喻，一目了然

3.2 基本的功能组件

这一部分貌似是数逻学的，没修过所以这里先学一学

- 逻辑门电路：由于非常天空上的图非常难看，所以不贴图了
  - 与门电路：符号是半个椭圆型

图 26:

  - 或门电路：符号是月牙形
  - 非门电路：符号是三角形 + 圆点

图 27:

图 28:

- 异或门电路：$(A \cap \neg B) \cup (\neg A \cap B)$ 由两个与门和一个或门组合而成

图 29:

- 组合逻辑 combinational logic
  - 译码器 Decoder
    - 给出一组输入，所有输出端的输出只有一个与其他所有的输出都不同。译码器可以用于地址选择，如果译码器由 n 个输入端，有 2^n 个输出端，则是完全译码器
    - 由若干个非门电路和与门电路组成
      - 2-4 译码器
      - 3-8 译码器
  - 多路选择器 MUX
    - 有多个数据输入，只能有一个数据输出，通过控制信号决定哪个输入被连接到输出端
    - 通过控制信号，对输出进行筛选，信号是若干个 0 或 1，通过**非门和与门的组合**使得恰好只有一个输入值会被原样输出，n 位的信号刚好可以控制 2^n 个输入数据
      - 四个选择的多路选择器 MUX 实现原理
  - 移位器 Shift
    - 使用**移位寄存器**或者**多路选择器**实现移位的操作，多路选择器的本质是**空间换时间**，利用复杂的设计提高位移速度
    - 固定的移位可以直接通过输入输出线的错位来实现
  - 符号扩展 Sign Extend

图 30:

图 31:

    * 对于有符号的补码进行扩展，并且正数和负数的扩展方式不同，对于补码的扩展可以通过重复符号位的方式实现，对于无符号数的扩展

- 状态元件
  - 状态元件和组合逻辑的区别
    * 组合逻辑的输出只和输入值有关
    * 状态原件的输出还**和状态元件自身的状态有关**，一般输入输出有一个时钟的延时
  - 寄存器单元
    * 输入的值有：clk 时钟，**RegWrite 写入控制**，idat **32 位**的输入数据
    * 输出的值有：odat 32 位的输出数据
- 复杂模块
  - ALU 算术逻辑单元
    * 算术逻辑单元 ALU 可以执行算术逻辑运算，是 CPU 中最重要的运算部件
    * 支持的运算包括加减法，逻辑 and 和 or，以及 slt，采用 3 位控制信号
      · 控制信号中 000 表示 and，001 表示 or，010 表示加法，110 表示减法，111 表示 slt
    * 32 位 ALU 简化的示意图如下
    * 输入：两个 32 位的数据，ALUop **3 位的操作码 ALUoper**
    * 输出：

图 32:

图 33:

图 34:

图 35:

图 36:

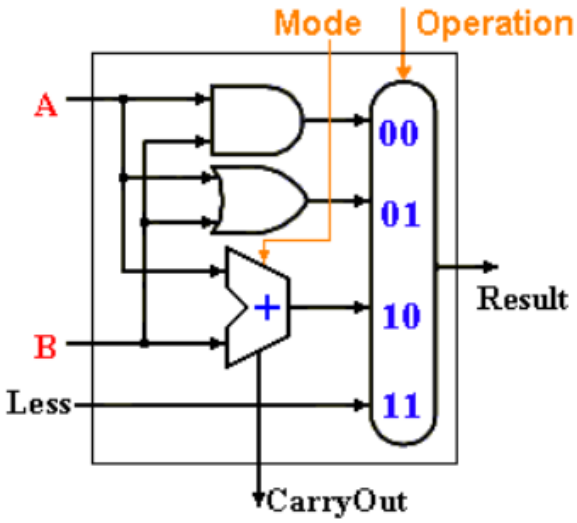


图 37:

图 38:

· Result 32 位的输出

· zero 零标志，这个值为 1 的时候表示结果为 0，为 0 的时候表示结果不是 0

· carryout **进位**标志

· overflow 溢出标志

– ALU 控制器：为了让常见的运算 (加减法) 更加容易控制

* 输入的内容：

· ALUop 两位的**控制信号**

· Func 指令的功能字段

* 输出：ALUoper ——ALU 的操作码，有 3 位

ALUop	Function Field(F5-F0)	ALU
00	XXXXXX	010——add
01	XXXXXX	110——sub
1X	XX0000	010——add
1X	XX0010	110——sub
1X	XX0100	000——and
1X	XX0101	001——or
1X	XX1010	111——slt

* ALU 控制器的示意图：

图 39:

– 算术逻辑单元的多级控制

* 用 ALU 控制器控制 ALU，示意图如下：

图 40:

- 存储器

– 存储器是多个寄存器和集合，由**译码器**选择指定的寄存器单元 (通过地址)

– 存储器分为只读存储器和读写存储器

  * 指令存储器是**只读**存储器 (Read Only Memory, ROM)

  * 数据存储器是读写存储器 (RAM) 可以根据读写控制信号，一次读/写一个 32 位的数据

    · 读写**共用一个地址输入端**，读写数据分别和存储器连接

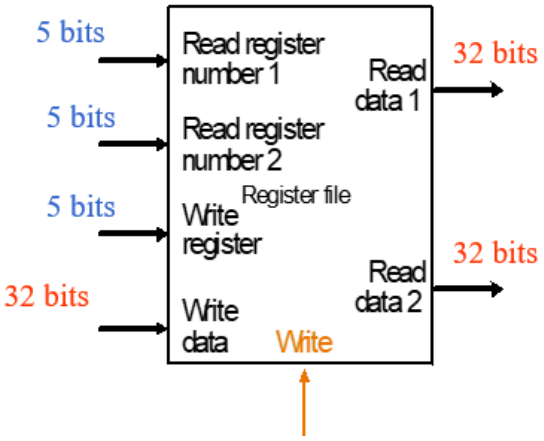


图 41:

  * 存储器按照字节编址，MIPS 采用了 32 位的**地址线**，可寻址空间为 $2^{32} = 4GB$，数据线也是 32 位

– **寄存器组：**

  * 寄存器组是 CPU 的主要部件

  * 因为很多指令都有 2-3 个操作数，因此需要用寄存器组来实现多路同时读写

  * 寄存器组的读

    · 32 个 32 位的存储单元，由两路 5 位寄存器地址选择读出寄存器中的内容

图 42:

  * 寄存器组的写

- 由 5 位地址选择器选择写入的寄存器，数据端位 32 位
- 信号 RehWrite 作为允许写入操作的控制信号

图 43:

* 寄存器组的整体结构图

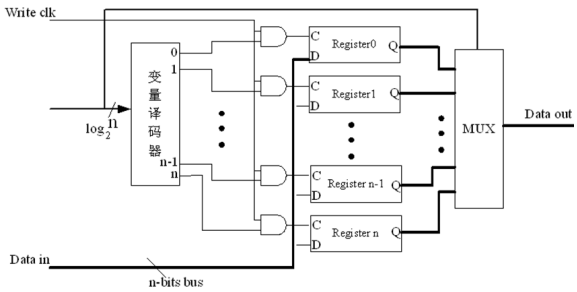


图 44:

* MIPS 的一个寄存器组一共有 32 个 32 位的寄存器，编号为 0-31，采用 5 位二进制寻址
  - 各个寄存器的功能之前已经提到过了，在第一部分 MIPS 指令系统里面

图 45:

- 输入数据
- clk 时钟
- rst 种植控制
- regA，regB 两路 5 位读的寄存器号，regW 一路写的寄存器号
- wdat 写入的数据
- regWrite：1 的时候表示允许写入
- 输出：Adat，Bdat 二路 32 位读出的数据

3.3 单指令数据通道

- PC 组件
  - 在 MIPS 中，所有的指令都是等长的，每条指令 4 字节 (32 位)，在 PC 中，保存着当前执行指令的地址
  - PC 的输出接到指令存储器的地址端，每条指令执行之后，PC+4，在下一个时钟的时候就自动转到执行下一条指令

图 46:

- R 指令的数据通道
  - 回忆：MIPS 中 R 指令的格式为 6 位操作码 +3 个五位的操作数 (rs,rt,rd 其中 rd 是目标寄存器)+5 位 shamt+6 位 func

图 47:

- LW/SW 指令的数据通道
  - lw 指令的格式：`100011+rs+rd+offset`, 表示 `$rd=Memory[$rs + offset]`

图 48:

  - sw 指令的格式 `101011+rs+rt+offset`，表示 `Memory[$rs + offset] = $rt`
- BEQ 指令的数据通道
  - BEQ 指令：`pc+=4; if(r1==r2) PC+= address*4;`
- J 指令：0-25 位是一个 address，`PC={PC[31-28], IR[25-0], {00}}`
- addi 指令:`rd = rs + Data`

3.4 单时钟 CPU 数据通道

- 接入和接出的实现原理
  - 分支接出：对于电路，从一组线接出或者部分接出都可以

图 49:

图 50:

  - 多路接入：使用多路选择器 MUX
    - 当有多个信号接入同一个输入端，不能直接合并，必须通过 MUX 选择性接入
    - 通过控制信号来决定**哪一个输入连接到输出**端

- 指令的组合：可以通过多路选择器的设计将指令合并到一个数据通道中
  - 多路选择器对于不同的指令 (R 型，I 型，J 型) 会选择不同的路径
  - 需要刻在 DNA 里的单时钟 CPU 实现原理图：
  - 新增 control unit 控制单元，可以发出一系列的**控制信号**，对不同的指令取不同的值
    - 控制信号控制了数据的流向和操作类型
    - 可以将指令码各位进行逻辑 and 运算，保证一条线只有当操作码为对应指令的时候才是 1，否则为 0
    - 控制单元内部原理示意图：
    - 控制信号对应表

控制信号	R	LW	SW	BEQ
RegDst	1	0	x	x
ALUsrc	0	1	1	0
ALUop	10	00	00	01
MemtoReg	0	1	x	x
RegWrite	1	1	0	0
MemRead	0	1	0	0
MemWrite	0	0	1	0
Branch	0	0	0	1

    - 该数据通道的若干假设
      - 等长的指令结构，每条指令都是 32-bit，其中最高的 6 位是操作码
      - 内存以字节编址，32 位的寻址范围
      - 寄存器组公有 32 个 32 位的寄存器 (5-bit 寻址)，其中

图 51:

图 52:

图 53:

- 0 号寄存器的值恒为 0
- 31 号寄存器是指令的地址寄存器，用于存放**子程序的地址**
- 29 号是堆栈指针寄存器
- Memory，Register File，ALU 的操作分别需要 2，1，2 个单位的时间，其他的时间可以忽略不计

– 各项指令执行所需要的时间统计表

指令	取指令	寄存器	运算	存储器	回写	合计	使用率
R	2	1	2		1	6	44%
LW	2	1	2	2	1	8	24%
SW	2	1	2	2		7	12%
BEQ	2	1	2			5	18%
J	2					2	2%

* 指令执行的平均时间为 6.34
* 在单数据通道中，每条指令只能设置一次各种控制信号，所以只能以需要时间最长的指令为一个时钟，也就是 8
  - 执行效率：6.34/8=79.25% 比较低

- 单时钟数据通道新增指令设计
  – 设计机器码的格式：操作码 6 位是固定的，其他的东西安排在剩下的 26 位里面
  – 在原设计图上增加必要的组件和通路
    * 寄存器组和存储器，ALU 等一般不变，可以实现新的指令功能
  – 添加新的控制信号：对于添加的新控制信号，一般原指令的默认值为 0
  – 举例：新增 `push $r` 指令
    * 指令的执行过程


```
$sp=$sp-4
Memory[$sp]=$r
```

图 54:

图 55:

图 56:

* 方案 1：由于指令里只有一个操作数 r，因此可以将 sp 直接涉及到指令里面，sp 有读写操作，而 r 只读，需要增加额外的信号 4，为了连接简便，将 ALUsrc 改为 4 选 1
  - 指令格式是 `000000+sp+r+sp+shamt+push`
  - 控制信号中，ALUsrc 设置为 10
* 方案 2：4 可以写在立即数中，但此时第三个寄存器不能放 sp，需要 R1 连接到 R3，作为写 sp
  - 指令格式是 `push+sp+r+0004`
  - 信号设计：RegDst 设置为 10

3.5 多时钟 CPU 设计

3.5.1 DNA 里的图

- 由于单时钟只能以耗时最长的指令为时钟，为了提高效率，可以考虑**缩短时钟**，以主要部件的执行时间为时钟，重新设置一遍控制信号
  - 每个部件在每个时钟**分别执行各自的任务**，在各个主要部件之间用寄存器存放临时的结果
  - IR：指令寄存器，MDR：内存数据寄存器
  - 本数据通道基于如下假设——和单时钟基本一样
    * 等长的指令结构，每条指令都是 32-bit，最高的六位是操作码
    * 内存以字节编址，32 位的寻址空间是 4GB，有 32 个 32 位的寄存器

图 57:

图 58:

  * Memory，Register File 和 ALU 的操作分别需要 2，1，2 个时间，其他的忽略不计
  * 微指令有七个字段，分别是 ALU，Reg，Mem，PC 等等

– 多周期 CPU 的数据通道 (第二张需要**刻在 DNA 里的图**)

图 59:

3.5.2 基本指令的多周期运行过程

- 多时钟周期的五个执行阶段：每个指令需要 **3-5 个周期**
  – IF 阶段：Instruction Fetch 获取指令, 具体的步骤如下
    * 使用 PC 来获取需要执行的指令，放到指令寄存器中
    * PC+4 然后把结果返回给 PC(上面两步基本上是同步进行的
    * 第一阶段所有的指令基本都是一样的
  – 其 Register-Transfer Language(RTL) 是

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

- ID：指令得而**译码**和 register fetch
  – 在需要的时候，读取寄存器 rs 和 rt 中的内容
  – 如果是分支指令就需要计算分支的地址

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

- EX(BC)：执行，内存地址计算或者分支选择完成
  – ALU 根据指令类型执行一种指令

- 对于内存调用 `ALUOut = A + sign-extend(IR[15-0]);`
- 对于 R 型指令，`ALUOut = A op B;`
- 对于分支指令：`if (A == B) PC = ALUOut;`
- 对于 jump 指令，`PC = PC[31-28] + IR[25-0] << 2;`

- MEM(WB)：**内存访问**或者 R 型指令的完成
  - 对于 lw，`MDR = Memory[ALUOut];`
  - 对于 sw，`Memory[ALUOut] = B;`
  - 对于 R 型的指令，`Reg[rd] = Reg[IR[15-11]] = ALUOut;`
- WB：write-back 步骤
  - 对于 lw 指令，`Reg[rt]=Reg[IR[20-16]]= MDR;`
- 总结：

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for 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] II (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		

图 60:

- 所以 R 型指令 4 割周期，J 型指令和分支指令都是 3 个周期，内存读写都是 5 个周期

3.5.3 控制信号和有限状态机

- 重要的控制信号和其作用

信号	设置为 0 时的作用	设置为 1 时的作用
RegDst	设置目标寄存器为 rt(20:16)	WB 阶段设置目标寄存器为 rd(15:11)
RegWrite	None	目标寄存器的输入时 Write data input 的值
ALUScrA	第一个 ALU 的操作数是 PC	ALU 的操作数来自 A 寄存器

信号	设置为 0 时的作用	设置为 1 时的作用
MemRead	None	内存中输入地址指定的内容作为输出
MemWrite	None	内存中输入地址指定的位置的值变为输入的值
MemtoReg	写入寄存器中的输入数据来自 ALUout	写入寄存器的数据来自 MDA
IorD	PC 向内存单元提供地址	ALUOut 提供地址
IRWrite	None	内存的输出被写入 IR
PCWrite	None	PC 被改变，被 PCSource 控制
PCWriteCond	None	如果 ALU 的零输出 active，PC 被改变

信号	值和对应的作用
ALUOp	00：加法运算，01 减法运算，10 按照 funct 来决定 ALU 操作
ALUScrB	00：第二个输入 ALU 的是寄存器 B 01：第二个输入是常数 4 10：第二个输入是符号扩展的
PCSource	00：ALU 的输出 (PC+4) 被写回 PC 01：分支目标地址被写回 PC 10：J 指令被写回 PC

- 有限状态机 (Finte State Machine)
  - 表示各个阶段控制信号需要设置的值
  - 第三张需要刻在 DNA 里的图
- 第一个阶段都一样

图 61:

3.6 微指令

- 把 FSM 所有的控制信号预先存储在存储器中，每次只要从存储器中读到控制寄存器中，由控制存储器作为控制信号，接到相应的组件中

图 62:

  - 存储控制信号的存储器叫做控制存储器 control memory
  - 核心是空间换时间，用一个存储器存储各类信号来缩短工作时间

– 有寄存控制器的多周期 CPU 数据通道

图 63:

– 微程序控制器：

  * 控制寄存器：每一位接相应的控制信号
  * 控制存储器：存放所有的控制信号
  * 地址转移逻辑：决定下一条微指令的地址

– 微程序控制逻辑

图 64:

Dispatch ROM 1		
Op	Opcode name	Value
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	Opcode name	Value
100011	lw	0011
101011	sw	0101

图 65:

- 微指令编程
  – 为了便于微程序的编写，把控制信号分成了若干组：存储器，寄存器组，ALU，PC 等七个字段
  – 为了实现微程序的选择转移，增加标号和下一步指令两个字段

3.7 流水线 (pipline) 和冒险 (hazard)

- 这一部分虽然上课没讲过但是历年卷里出现了选择题

3.7.1 流水线的基本原理

- CPU 的流水线技术本质上利用了时间的并行性，将原本需要串行的处理一定程度上并行化了，使得 CPU 的效率提高，原理如下图
  – 转换成指令的运行的原理如下

图 66:

图 67:

3.7.2 冒险 hazard

- 在流水线中我们希望当前每个时钟周期都有一条指令进入流水线可以执行，但是某些情况下下一条指令无法按照预期的开始执行，这种情况就是冒险 hazard
- 冒险有三种：
  - 结构冒险：如果一条指令需要的**硬件**还在为之前的指令工作，而无法为这条指令提供服务
  - 数据冒险：如果一条指令需要的数据正在被之前的指令操作，那么这条指令就无法执行
    - 解决方法：Forward data if possible
  - 控制冒险：如果需要执行的指令是由之前的运行结果决定的，而运行的结果还没有产生，就是控制冒险
    - 解决方法：stall the pipeline，Predict branch outcome，Delayed branch
  - 总结：硬件——结构冒险，数据——数据冒险，控制指令——控制冒险

4. 存储器

4.0 存储器的分类

- 按照存储介质来分
  - 半导体: SRAM,DRAM,ROM,FLASH
  - 磁介质: 磁带和磁盘
  - 光介质: CDROM

图 68:

- 按照读写功能
  - 只读内存：ROM/CDROM
  - 读写：FLASH, 磁盘和磁带,RAM(随机内存，就是平时说的内存)
- 按存储的易失性
  - 易失：RAM
  - 非易失：ROM, 磁介质, CDROM
- 按照在计算机系统中的作用分类
  - 主存储器：计算机运行的主要存储器，存放运行的程序和数据
  - 辅助存储器：如外存
  - 缓冲存储器：作为一些 I/O 的数据缓存
  - 控制存储器：存放控制程序和数据
- 存储器的**分级结构**
  - 快村：多级 Cache
  - 主存：RAM,ROM
  - 外存：磁盘，磁带，CDROM

图 69:

4.1 存储器基础 & 存储器管理

- 这一部分感觉就是了解一下，考试看起来不太会来考
- 存储器的存储结构
  - 位 bit--位线--位面--字 word--字扩展 (条)--存储器 (多条)
  - 存储器的基本原理，通过 X,Y 地址译码在电容中存储，由于没有电源供电，反向漏电会导致保存的信息丢失，所以必须不断刷新

- 动态存储单元
    - 由于没有电源供电，存在的反向泄漏电流会使得保存的信息丢失，必须不断刷新

图 70:

- **动态存储器 (DRAM)**：需要定时刷新
    - 刷新周期：连续两次对整个存储器全部刷新的间隔时间
    - 刷新的方式分为：集中刷新，分散刷新，异步刷新

4.1.1 芯片和存储器

- 存储器的地址
    - 二四译码器：用两位的 0-1 信号控制四个数据的选择，对应每一组输入只有一个输出和别的输出不同，则可以将该输出选中

图 71:

- 地址译码：
    - 利用译码器将多个**一位的存储单元**组成一个整体
    - 所有的存储单元接在同一条数据线上，但是由于译码器的控制，只有一个单元没选中可以进行数据的读写，这时输入数据即为地址

图 72:

- 二维地址译码：以二维存储单元阵列组成一个整体，可以减少驱动
- 位扩展:
    - 在二位地址译码的基础上，将若干个存储单元阵列以相同的地址译码器选中
    - 每个二维的阵列对应不同的数据线，则相同地址译码在不同二维阵列上对应的单元，合并为存储器，从位---> 字
    - 这图有点抽象，给我看晕了

图 73:

图 74:

    * 存储芯片：
        · 地址线：决定可寻址的范围
        · 数据线：决定数据的位数
        · 控制线：包括电源、读写控制、片选
  - 字扩展
    * 从一个最基本的一位存储单元通过双译码器得到一个一位的存储位阵列面，将同样地址线的若干相同的位平面，接不同的数据线，封装得到一个存储芯片
    * 地址高位交叉：
        · 高位的地址作片选，同一芯片的地址**连续**
        · 但是读取连续的地址单元需要多次读写
    * 地址低位交叉，同一芯片的地址不连续，但读取连续地址单元只需要一次
    * DOS 系统的芯片
        · 只有 1MB，由 RAM,BIOS,ROM 组成

- 一个关于芯片计算的题目

  ```
  Assume an 8Kx8bits memory is composed of 2Kx4bit chips. The first address of the chip whic
  A: 0700h
  B: 0800h
  C: 0600h
  D: 0000h
  ```

  - 这道题应该这样分析，首先可以计算出需要 **8 块芯片**，而 8K*8bits 的内存需要 13 位

图 75:

图 76:

图 77:

地址，其中**最高的三位是片选地址**，低的十位是片内的地址，而 0B1F 的最高三位是 010，对应第二片，因此其起始地址就是 0000100000000000，也就是 0x0800

- 这种题目的一般方法：先算需要几片，再算地址一共有几位，取最高的几位来看就行
- 字扩展和位扩展的分别：
  - 位扩展是指需要多少位来扩展成一个字，字扩展是指扩展成字之后扩展出更大的存储空间
  - 上面这题中 2K*4bit* 的芯片需要 *2* 片一起才能作为一个字，所以位扩展是 *2*，又需要 *4* 个这样的组成 *8K*\8bits 的，所以字扩展是 4

4.1.2 对齐和不对齐

- 计算机一般以字节作为寻址单位，即每一个字节一个地址
  - 对**字 (4 字节)** 或者半字 (2 字节，short) 寻址的时候，最低位字节地址为整个字或者半字的地址
    - 比如一个字的 4 个字节地址分别是 10，11，12，13，则这个字的地址是 10
  - 大端: 一个字或半个字的高位字节位于地址小的字节为大端,BigEndian
  - 小端: 一个字或半字的高位字节位于地址大的字节为小端,LittleEndian
  - 即高位在前为大端，低位在前为小端, 例如，一个字 0x12345678 的地址为 10，即要将这个字写到地址 10，11，12，13 中
    - 大端 0x12,0x34,0x56,0x78
    - 小端 0x78,0x56,0x34,0x12

图 78:

图 79:

- 对齐：当系统要访问存储器的一个字，刚好在硬件存储中是一个字，就是对齐访问。
  - 不对齐就是访问的时候不是一个硬件字，需要硬件多次读写
- 16 位 zjie 对齐与不对齐
  - 为了避免读写一个数据，不得不多次访问存储器，可以将存储器按低位交叉方式编址，也就是译码时用低位地址线来做片选信号
  - 这样一来，地址连续的存储器单元，将对应于不同的存储器芯片 (bank)，当系统要求读写连续地址数据时，就可以从不铜存储器同时读出到寄存器，然后由总线传送
- 内存条：从一个最基本的一位存储单元，通过双译码器得到一个一位的存储位阵列面，将接同样地址线的若干相同的位平面，接不同的数据线封装得到一个存储芯片

4.2 Cache 的基本原理

- 存储器的层次结构 memory hierarchy
  - 最顶层是 register 直接和 CPU 进行通信
  - 下面是各级的 memory

图 80: memhiera

- 程序运行的两个局部性原理
  - 时间局部性：如果每个数据或者指令被引用了，那么不久之后可能将被再次引用
  - 空间局部性：如果一个数据被引用了，那么邻近的单元也可能被引用
- **Cache(高速缓存) 的原理**
  - 计算机只能按照既定的地址来确定一个数据的意义，而地址在转换存储器的时候要进行对应的转换
    - 计算机中的运算都是通过逻辑运算实现的，逻辑运算速度快，因此应该多使用逻辑运算

* 在地址转换中我们都把 2 的幂作为数量单位，比如块的大小，Cache 的容量，Cache 分组和页

图 81: cache

- cache 技术的基础：SRAM 和 DRAM
- cache 根据两个局部性的原理，将程序中**正在运行的部分**调入 cache 中，来提高指令和数据的访问速度，物理存储和 cache 都被分成相同大小的块 (block) 作为最小的管理单元，从物理地址到 cache 地址的转换, 有这样几种方法：(考试必考点)
  * **直接映射**：物理地址按照一定的方式固定映射到 cache 的某一块上，但是物理地址比 cache 中的地址要大，所以会有**很多物理地址映射到同一个 cache 块上**，为了标记是哪个物理块，假设 cache 一共有 M 块，则：

图 82:

    · cache 中有一位**有效位**，1 表明数据有效，0 表名数据无效

    · 物理块号：j= 物理地址/块的大小

    · cache 块号 (索引号)：i=j mod M

    · TAG = j/M 要来记录区分不同的内存块

    · 当 cache 中恰好有 2^n 个 block 的时候，对应的 cache **索引**就是内存地址的最低 n 位

    · cache 中地址的形式：`TAG+Index+ByteOffset`

    · 其中 TAG+index 构成 cache 块的地址

    · TAG 就是物理地址的最高几位，cache 块序位中间几位，最低的几位是块内的地址

    · 主存地址映射到 cache 中时，最高的几位是 TAG，中间的几位地址是索引，最后几位地址是块内存储的信息

    例题 1：对于能存储 16KB 数据的**直接映射** cache，块的大小为 4word，地址为 32 位，需要多少大的存储空间？

- 数据的总大小是 2^{14} 字节，而一个块的大小是 2^4 字节，因此一共有 2^{10} 个块
- 一个块的数据量是 128bits(一次 4 个字，一个字是 32bits，需要 128 位)
- Tag 需要的 bits 数量 =address - index -block size = 32 - 10 -4 = 18bits
- 其中 4word 需要 16bits，因此 block size 是 4
- 有效位是 1bit，一共需要 19bits
- 所以 cache 的总的 size 就是 $2^{10} \times (128 + 18 + 1) = 147000$ bits = 18.4 KB

例题 2：一个 64 块的 cache，块的大小是 16bytes，则字节地址为 1200 应该直接映射到哪里？

- 这种情况的答案 (block address) modulo (number of cache blocks)
- 而 block 的地址计算方式是字节地址/每个块的大小 = 75
- 所以答案就是 11

这里记录例 1 类型的题目的**通用解法**：

- 条件是 32 位的字节地址，直接映射的 cache，有 2^n 个 block，一个 block 的大小是 2^m word，因此需要 n 位作为 index，m 位用于查找块中的字，2 位是字节的偏移信息
- 则 TAG 的大小是 32-(n+m+2)
- cache 需要的总位数是 block 数量 *(block 的大小 + 有效位的大小 +TAG 大小) 也就是
- $2^n(2^{m+2} \times 8 + (32 - m - n - 2) + 1)$
- 关于 block size 的计算：假设 block 是 2^m 个 word 的大小
- 常识：word 是计算机中存取的基本单位

- 为了定位是哪一个 word，需要 m 位
- 而在 word 里面，为了定位是哪个字节，需要 2 位，所以一共需要 m+2 位，block size 就是 m+2 bits

* 全相联：

图 83: cachefull

- 直接映射每个物理块有固定的 cache 块很容易判断命中与否 (比较 TAG)，但造成 cache 块的闲置
- 考虑**不固定的映射方式**，当需要替换的时候就**遍历 cache 中所有的块**，覆盖今后**最不可能用到的块**，这样可以提高 cache 的**命中率**
- 类似于 ads 中装箱问题的 best fit 算法
- 理论上存在的替换方法：LRU,FIFO,RAND
- 虽然提高了命中率，但是时间效率降低了
- 大量 TAG 判断是否命中会使得硬件的设计非常复杂，如果采用顺序判断，由于在 cache 中，Cache 与主存存取速度的差别在 1-2 个数量级，确定是否命中的时间太长，cache 就失去意义了
- 块地址：按照块确定的地址，把原地址除以块的大小即可，块内部分为块内地址
- 主存地址/块大小 =TAG 余数作为 block-offset

* 组相联：组相连把 cache 中的块分成若干组，每一块和主存直接进行映射，当主存快映射到组里之后，具体替换组里的哪一块，通过全相联的方法来选择
  - 缝合怪

图 84: cache4way

- 组相联是全相联和直接映射的折中，直接映射是一种 n 个组的组相联，全相联是 1 个组的组相联
- 主存块号/cache 组数的商是 TAG，余数是 index(组的位置)

        - 组相联和直接映射的区别：
        - 组相联分成了若干组直接映射，因此在每一组里，Index 的位数减少了，而 TAG 的位数增加了
        - 同样大小的 cache，如果分成了 2^n 路组相联，那么相比于直接映射，index 减少了 n 位，TAG 增加了 n 位，实际上原理就是同样多的 m 个 block 分成了若干组，在每个组里面进行直接映射，所以块数相比于直接映射减少了，所以 index 的位数也变少了
    - 楼教授有言曰：**直接映射是个人承包，全相联是共产主义，组相联是家庭联产承包责任制**
  - 不同映射方式的比较
    - 全相联的 cache **效率最好**，但是块数多了就**难以判断是否命中**
    - 直接映射的 cache 效率低，但是命中率高，组相联的各项水平处于二者之间
    - 其实就是因为直接映射和全相联是两种组相联的特殊情况，直接映射是 1 路组相联，全相联是 m 路组相联
  - 多级 cache
    - L1: 一级 cache，在**处理器的内部**，以核心频率工作
    - L2: 以**处理器一半的频率工作**，和 CPU 一起封装
    - L3: 在**主板**上，以总线频率工作
- cache 的命中和失配
  - cache 的命中 (hit)：指 CPU 要访问的数据存储在了 cache 中
  - cache 的失配 (miss)：指 CPU 要访问的数据不在 cache 中，此时需要将这个数据所在的整个 block 放入 cache 中
    - 主要的失配种类
      - compulsory misses 强制失配，第一次启动时总是失配
      - 可以通过加大块的大小降低失配率

        - capacity misses 容量失配，cache 的容量比主存小引起失配
        - 可以通过**多级 cache** 降低失配率
        - conflict misses 相联失配，由于映射关系影响命中率
        - 可以把组相联改成全映射
    * **块越大则命中率越高**，但是当 cache 容量较小的时候，块增大可能使得块数太少而减小命中率
        - 在 **cache 总容量不变**的时候，块越大则命中率更高，块的大小取决于**硬件**
    * 不同的替换策略
        - cache 失配需要读入新的数据到 cache 时，可以在几个块之间选择进行数据替换，选择的依据就是程序访问的局部性原理，有这样一些替换策略
        - LRU 最近最少使用的优先替换
        - LFU 使用频率最低的优先替换
        - FIFO 先进先出
        - Random 随机替换
  - 命中率和失配率 (miss rate)：命中和失配的信息所占的比例
- 保持 cache-主存数据**一致性**的方式
  - 写通 Write-Through，在改写存储器内容时，主存和 cache 要一起改写
    * 优点是简单，缺点是写的速度很慢
  - 写回 Write-back 写数据的时候只写在 cache 中，在块替换的时候再将数据写道主存中
    * 优点是对同一个块多次读写的时候，最后一次才写入主存，速度快
    * 缺点是**控制比较复杂**，可以设置 dirty 位改进
  - 写缓冲 Write-buffer 改写的时候先写到缓冲区中，避免速度过慢
- cache 效率的衡量

- 存取时间的计算：通常使用 AMAT(平均内存访问时间) 来检测 cache 的效率
  - AMAT = (time for a hit) + (miss rate x miss penalty)
  - 计算方法 1：
    - Tc 是 cache 访问的时间，Tm 是主存的访问时间，H 是命中率
    - T=Tc+(1-H)Tm
  - 计算方法 2：指令 + 数据
    - Tc 是 cache 访问的时间，Tm 是主存的访问时间，Hi 是指令的命中率，Hd 是 data 的命中率，P 是程序中 LW/SW 指令的比例
    - T=Ti+Td*p=(Tc+(1-Hi)Tm)+(Tc+(1-Hd)Tm)*p
- CPU 时间的计算
  - CPU 时间 = (CPU 执行的周期数 + Memory-stall 时钟周期) * 一个时钟周期的时间
    - $CPU_{time} = I \times CPI \times Clock$
  - Memory-stall 时钟周期 = 指令的数量 * 失配率 * 失配的 penalty = read-stall cycles + write-stall cycles
    - 而 read-stall cycles = program 中 read 的次数 * 失配率 * 失配的 penalty
    - write-stall cycles = program 中 write 的次数 * 失配率 * 失配的 penalty + write buffer stalls
    - 因此其实 Memory-stall 的时钟周期 = 存储访问的次数 * 失配率 * penalty
    - 这一部分感觉公式比较多，直接抄在 A4 纸上好了

4.3 虚拟存储

- 实际上就是把主存当作磁盘的 cache 来使用
- 虚拟存储器是根据存储器访问的**局部性原理**，将内存中正在使用的调入真正的内存，而暂时不用的存入磁盘

  - 内存也叫**物理存储器**
  - 假设对于 32 位的 CPU，寻址空间为 4G，设 lw 指令为 `LW srd dat(srs)` 则直接寻址的模拟为 `srd=Memory[srs+Dat]`
  - 设内存的大小为 16K，如果地址范围限定在 16K 内则可以直接寻址，而当访问的地址超过物理存储器时，理论上会出错，访问应该改成 `srd=getMemory(srs+Dat)`,即为虚拟存储器的模拟函数
    * page fault：内存中没有找到数据，需要从 disk 中检索
    * 会产生巨大的 miss penalty，一般选用 write-back 策略维持一致性
- 为了尽可能提高命中率，虚拟存储采取的是**全相联**的策略
- 虚拟存储器的几种实现方式
  - 单页映射：整个物理存储器为一页，则需要有标志表明物理存储器的 4k 为 4G 寻址中的哪一部分
  - 反向页表：整个**物理存储器为一页**
    * 页数太少会使得命中率降低，为了提高命中率可以增加页数，相应减小每页的大小
  - 设每一页为 1K，在物理存储器端建立页表，记录物理页和虚拟页的对应关系
    * 页表的数量为物理存储器的大小/每页的大小 =4K/1K=4
  - 正向页表：页表也可以建立在虚拟地址段
    * 页表也可以建立在虚拟地址端，则页表的规模为为 4G/1k
  - 页表的优化
    * 页表实际上也存储在存储器中，设有**页表地址寄存器 PTR** 来保存页表的基地址
    * 为了提高页表的访问速度，可以设置**页表的 cache：TLB**
    * 为了减小页表的的大小，可以：设置段界、使用分级页表
- 虚拟存储的原理

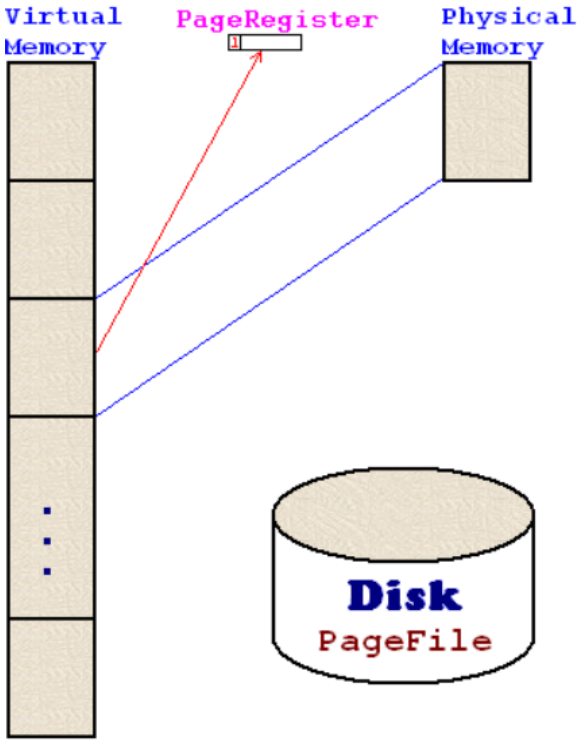


图 85: image-20200515113627571

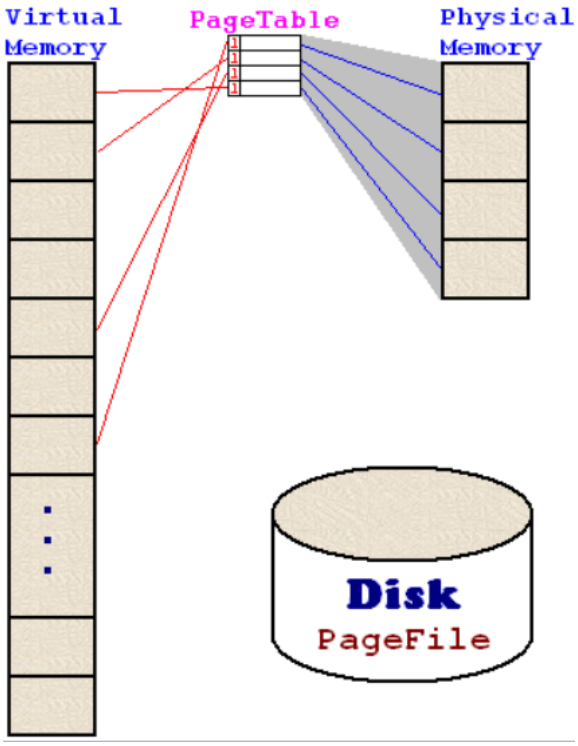


图 86: image-20200515120013915

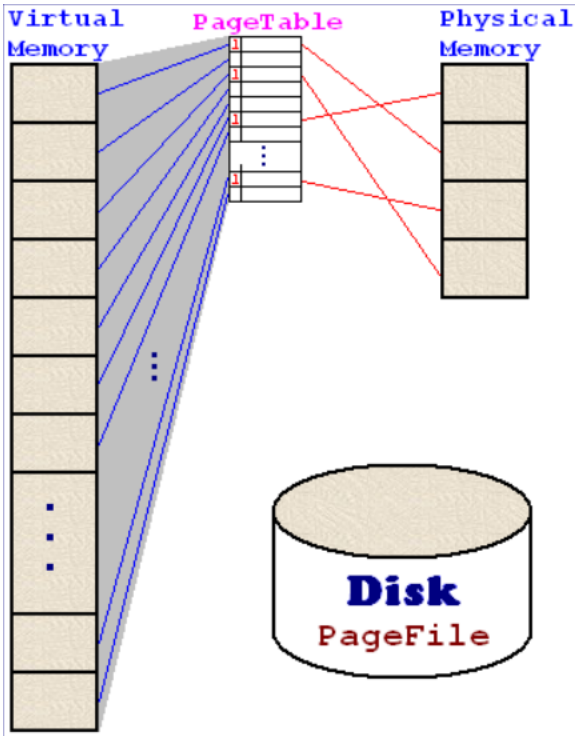


图 87: image-20200515120120252

- 根据局部性原理，可以把程序需要用到的整个存储空间划分成一个个大小相同的页，把其中一些页放在主存中，而其他的页在需要的时候再建立或者放在磁盘中
  - 分页管理是因为**硬件上容易实现**
  - 建立页表来管理页，如果该页在主存中，就记录下地址，如果不在，则在页表上标记不存在
  - 当程序需要调用某个存储单元的内容时，先根据线性地址，算出所在的页，然后查询页表，如果在主存中就直接存取，如果不在就是一个 page fault，要把主存中的某一页存入磁盘，再把要访问的页调入主存中使用 (该过程使用 LRU 算法)
- 几个重要的计算公式
  - 页表的大小 = 表的项数 * 每个表项需要的位数
  - 表的项数 = 虚拟地址的空间 / 每页的大小
  - 每个表项的位数 = log(物理地址空间 / 每页的大小) + 适当控制位数 (很多时候是 4 位)，但是一般凑成整字 (32bits)

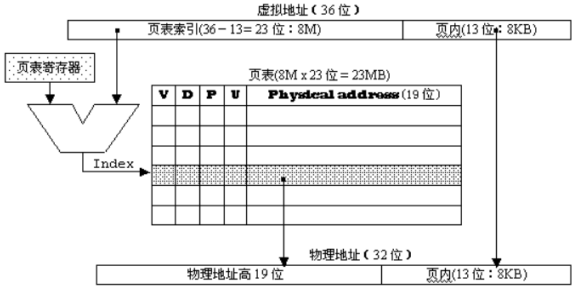


图 88:

- 虚拟地址的换算

图 89:

- TLB 原理

图 90:

    - 在 4G 的存储空间中，如果按照 4K 来分页，则可以分出 1M 页，每个页表项为 4B，则一个页表的大小为 4MB

  - 但是由于局部性原理，页表在某个时刻使用的也只有仅仅几页，因此可以把经常用的一些虚拟页号和对应的物理地址写到 cache 中，即为 TLB

    - 则程序在调用某个存储单元的同时，根据线性地址算出所在的页，先查 TLB，如果有就直接获取物理地址进行存取，如果没有则说明没有写在 TLB 中，需要去查页表，按照 page fault 的情况进行

- 虚拟存储的整体逻辑

图 91: viradd

4.4 保护模式

- cache 和虚拟存储的比较

	cache	虚拟存储
实现	硬件	操作系统 + 少量硬件支持
写策略	write-back/write-through	write back
替换策略	LRU/FIFO/LFU/Random	LRU
交换单元	block	page
映射方式	直接映射，全相联，组相联	全相联
寻址方式	全硬件，相联存储器	Page-Table 页表/TLB 快表/分级页表

  - 原理上讲二者差不多，差距主要在存储存取的速度上

    - 虚拟存储中存储器和磁盘的差距在 6 个数量级以上

    - cache 和主存储器的存取速度相差 1-2 个数量级

- 实模式和保护模式下的地址转换：

  - 物理地址：机器中真正具有的存储空间，程序只有在物理存储器中才能被执行

  - 页表项：32 位，高 20 位是页基地址的最高 20 位，由于按照页来交换，因此页的基地址的低 12 位都是 0，所以低的 12 位都被用作属性控制

    * 第 11-9 位是 AVL，记录页的使用情况，比如 LRU
    * 第 7 位是 D，表示是否该写过
    * 第 1 位是 R 的时候表示读写，0 的时候表示只读
    * 第 0 位是 P，第 6 位是 A
- 逻辑地址：
    * 在程序中使用的地址，在 intel 中由段和偏移组成，段和便宜可以转换成线性地址
    * 没有虚拟存储的话，线性地址就是物理地址，转换方式是
        · 物理地 (20 位) = 段 (16 位) * 16 + 偏移 16 位
    * 286 以后段寄存器中的高 13 位就是表钟的索引，低的三位中 TI 一位描述所在的表，RPL 两位描述特权级
        · 段在线性地址中的基地址在 8 字节的段描述里给出，根据段寄存器的值，在段描述表中找到对应的段描述
        · 线性地址 32 位 = 段基地址 32 位 + 偏移 32 位
        · 每个 8 字节段描述的：分成 4 条，每条 2 字节
        · 第 1 条第 1 个字节和第 2 条的第 2 字节，第 3 条的 2 个字节加起来是 32 位段基地址
        · 其他的都是一些控制位数，具体的太细了
- 线性地址 (虚拟地址)
    * 在 MIPS 结构中，基本为程序中的地址 (lw, sw) ，在 PC 中就是存储器的物理地址
    * 在 286 以后，线性地址为虚拟地址，也就是程序中可以寻址的存储空间，一个程序的运行，并不一定要所有的内容同时调入存储器才能执行，往往在一个时间内，只需要其中的一部分在存储器中，而其他的部分可以存在磁盘上，需要时再调用

        * 虚拟存储技术使得程序员**不必考虑实际存储器的大小**，而**由 OS 来负责计题的存储器的使用**

5. I/O 系统

5.1 I/O 系统的基本介绍

- I/O 设备的三个要素
  - behavior 表现：输入/输出/存储
  - partner 参与者：是人还是机器
  - data rate 数据速度：数据能在 I/O 设备和主存/处理器之间传输的**最大**速度
- I/O 设备的评价标准
  - 带宽 banwidth——最重要的评价指标
  - 带宽可以从以下两个方面来评价
    * 一次可以 move 多少数据
    * 每个单位时间可以作多少次 I/O 操作
- 常见的 I/O 设备
  - 键盘，鼠标，音响，Scanner，打印机，显示屏，磁盘，磁带

5.2 磁盘

- 这一部分 DBS 里也有涉及到，不过 DBS 好像不太考这一部分的内容
- 磁盘的两种类型
  - floppy disk 软盘
  - hard disk 硬盘：容量更大，访问效率更高，有多个盘，由如下几个部分构成
    * platter 盘：每个盘的两个表面都是可以记录的
    * tracks 磁道：每个磁盘的表面被划分成若干磁道

        * sectors 扇区：每个磁道被划分为若干扇区
            · 扇区是磁盘读写的**基本单位**
    - 磁盘的基本参数
        * 引导扇区：
            · 扇区 1：BOOT 区
            · 扇区 2-10：FAT-1，11-19：FAT-2
            · 扇区 20-31：目录区
            · 扇区 32 及以后是数据区
        * FAT 表：每个磁盘有两个相同的 FAT 表，硬盘中一般为 16 位，现在往往是 32 位
            · 0000 未用簇
            · 0001-FFEF 下一个簇号
            · FFF0-FFF6 保留位
            · FFF7 坏簇
            · FFF8-FFFF 最后簇，表示文件结尾
- 磁盘的访问
    - 访问的过程
        * seek：把读写头放置到正确的磁道上面 (寻道时间)
            · 相关的数据有最小访问时间，最大访问时间，平均访问时间
        * rotation latency：等待找到目标扇区的时间
            · 平均延迟是 disk 周期的一半
            · 计算公式：$latency = \frac{0.5}{RPM}$

        - RPM：每分钟的转数 (rounds per minutes)，计算的时候需要将 RPM 转换成 round per 毫秒
    - transfer：传输一个扇区的时间 (1KB/sector) 和转速相关
    - disk controller：控制磁盘和内存之间的传输
  - 访问时间的计算：
    - seek+latency+transfer+controller 一般是毫秒的级别
  - 几个重要的评估标准
    - MTTF 平均 failure 时间
    - MTTR 平均 repair 时间
    - MTBF = MTTF + MTTR
    - 可用性 = MTTF / MTBF
  - 提高 MTTF 的方法
    - 避免错误的访问
    - 容忍错误：使用冗余来允许服务在出现故障的情况下遵守服务规范
    - 错误预报
- RAID：**廉价磁盘的冗余序列**
  - 用一系列便宜的磁盘代替一个大磁盘，来提高磁盘的效率

5.3 总线 bus

5.3.1 基本概念

- 总线是构成计算机系统的互联机构，是多个系统功能部件之间进行数据传输的公共通路，分为
  - 控制线：用于传递各类**信号**和存储数据线中信息的信息
  - 数据线：传递信息 (数据，地址，复杂的指令)

    * 细分的话可以分为地址线和数据线

- 总线和内存，I/O 设备的数据交换方式有两种
  - 内存映射 I/O 方式：内存与 I/O 系统采用共用的控制、地址和数据线
    * 内存和接口部件共享一个地址空间，各自拥有不同的地址端，读写的指令和内存读写的指令也是一样的
  - 独立 I/O 编址：共享地址线和数据线，但是用不同的控制线
- 总线的工作原理：
  - input 操作：
    * 控制线传输一个写的请求到内存，数据线传递地址

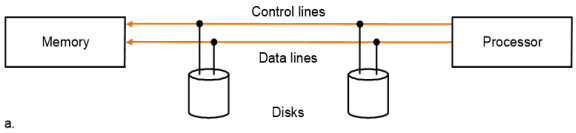


图 92:

    * 内存主备好之后会给设备发送信号，然后在内存中写入数据，设备不需要等待数据的存储完成

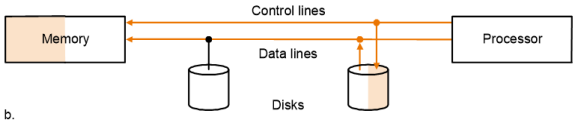


图 93:

  - output 操作：
    * 处理器通过控制线发送 read 请求，数据线传递地址
    * 内存进行数据的访问
    * 通过控制线将数据传输出去，目标设备会存储总线上传递过来的数据
- 总线的分类：

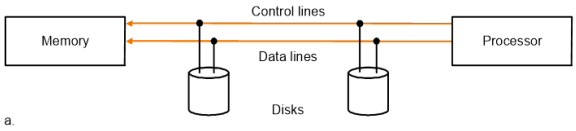


图 94:

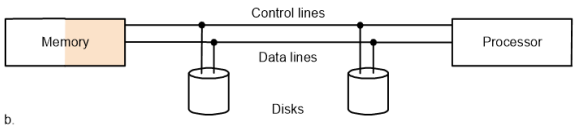


图 95:

  - 一个单处理器的总线分为如下三类
    - 内部总线：CPU 内部连接各寄存器和运算部件之间的总线
    - 系统总线：CPU 同计算机系统的其他**高速功能部件** (如存储器) 等互相连接的总线
    - I/O 总线：中低速 I/O 设备之间互相连接的总线，比如 SCSI
  - backplane 总线 (高速，标准化的总线，我也不知道属于哪一类)，比如 PCI
  - processor-memory 总线 (短而高速，自定义设计的)

- 总线的特性：
  - 物理特性：包括总线的根数，插头插座的形状，引脚线的排列方式
  - 功能特性：描述总线中每一根线的功能
  - 电气特性：定义每一根线上信号的传递方向和有效电平范围，输入 CPU 的信号是 IN，CPU 发出的是 OUT

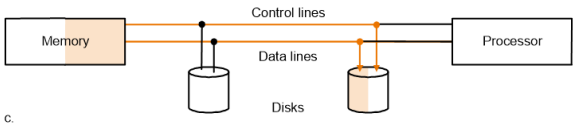


图 96:

  - 时间特性：定义了每根线在什么时候有效，规定了总线上个信号的有效的时序关系

- 总线的标准化：下面是总线的一系列标准
  - SCSI(small computer system interface,，小型计算机系统接口)
  - PCI(peripheral component interconnect，外部组件互联)
  - IPI(intelligent peripheral interface，智能外部接口)
  - IBMPC-AT IBMPC-XT
  - ISA EISA
- 同步传输和异步传输
  - 同步传输：使用时钟和同步协议，每个设备必须在同一个速度和时钟下操作
  - 异步传输：使用握手协议 (handshaking) 或者选通 (stubing) 的策略
    - 此时 CPU，接口和 I/O 设备有不同的时钟周期

5.3.2 总线的仲裁 **Arbitration**

- 当多个主设备同时争夺总线的控制权的时候，由总线仲裁部件授权给其中一个主设备
  - bus master 主设备：处理器总会是一个 bus master(~~bus master 的任务罢子~~)
- CAN 采用优先级方式，TCP/IP 采用 CSMA/CD 级制，而 RS-485 采用**主机轮询**的机制
  - 链式查询：所有的设备共用一条总线请求与相应，当总线控制器接到总线的请求之后，总线授权信号串行地在 I/O 接口之间传递，将控制权交给第一个有总线请求的 I/O 接口
    - 离中央仲裁器近的设备优先级高
    - 容易扩充设备
    - 对电路故障非常敏感
    - 优先级是固定的，如果优先级高的设备请求非常频繁，优先级低的可能一直不能使用总线

- 计数器定时查询：总线控制器按顺序查询各个接口，如果有请求就响应请求并记录设备的地址
    - 每次查询可以从固定的地址开始 (优先级固定)，也可以从上一次相应的设备开始 (循环优先级)，每个设备使用总线有相同的优先级
    - 为传送设备地址需要增加的总线的数量是 **log2 设备数**
- 独立请求：每一个共享总线的设备均有一对请求线和授权线，当需要使用的时候就发出请求信息，控制器独立决定响应哪个设备
    - 响应时间快，不用逐个查询
    - 优先级控制比较灵活
    - 可以屏蔽某些设备的请求

- 分布仲裁方式：
    - 不需要中央仲裁器，每个潜在的主方功能模块**都有自己的仲裁号和仲裁器**，当它们有总线请求的时候会把仲裁号发到仲裁总线上，每个仲裁器将得到的信号和自己的比较，如果仲裁总线上的号大，则不会响应这个仲裁器的请求，并撤销仲裁号，最后获胜的保留在总线上

5.4 数据传输和控制

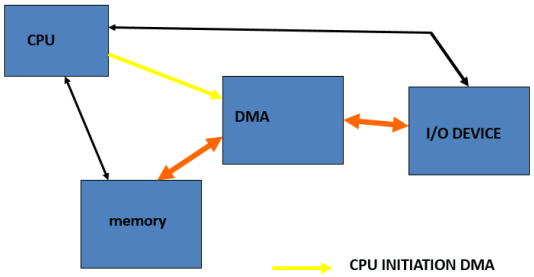
- 程序查询的方式
    - 先检查外设的状态，允许时再进行数据的 I/O 传送
    - 缺点是外设一般比较慢，CPU 必须不断检查外设状态，效率很低
- I/O 设备的特性
    - 通过处理器，被多个程序共用
    - 使用中断来交流 I/O 操作的信息
    - 需要三种不同类型的交流
        - OS 必须能够给 I/O 设备发送命令

- 通过内存映射的 I/O：发送地址给 I/O 设备
- 特殊的 I/O 指令
- 命令端口/数据端口：静态寄存器，指令

* 当 I/O 设备完成一个操作或者抛出异常的时候，设备必须通知 OS
* 数据必须在内存和 I/O 设备之间传输

5.4.0 数据传输的控制

- I/O 系统的数据传输的控制：
  - 三种模式
    * polling 轮询：处理器定期检查设备的**状态位**，来确定是否到了下一次 I/O 操作的时间
    * interrupt 中断：当一个设备想要告知处理器需要完成一些操作时，让处理器中断
      .
    * DMA：设备**直接从内存**中进行数据传输，绕过处理器



  - 三种模式的比较：
    * 轮询的缺点：**耗时**，当 CPU 周期性地轮询 I/O 设备的时候，I/O 设备可能没有请求或者还没有准备好
    * 如果 I/O 操作是中断驱动的，OS 可以在数据被读写的时候进行别的 task

        - 中断驱动的优点就是可以进行**并行**
    - DMA 不需要处理器的控制，所以比较省时间
    - 轮询，中断和 DMA 中，对**硬件支持要求最低的是轮询**

5.4.1 中断

- 中断 Interrupt
  - 中断的类型：
    - 内部中断
      - 软件中断：指令中断，是按照中断的级制进行的此程序调用
      - 硬件异常：内部出错产生的异常中断
    - 外部中断：I/O 设备请求产生的中断
  - 屏蔽中断和非屏蔽中断 NMI：外设中断可以屏蔽
  - 中断服务程序：中断发生的时候，系统中处理中断事件的程序
  - 中断向量：中断服务程序在内存中的入口地址，实模式下分段：偏移，共四个字节，保护模式下是 8 字节终端描述
  - 中断向量表：PC 系统共有 256 个中断，中断向量表就是所有中断向量的集合
    - 实模式下一共有 4*256=1KB
    - 保护模式下是中断描述表的形式
- 中断的优先级
  - 多个 I/O 设备发起中断请求的时候，需要确定设备的优先级，优先级高的设备的中断请求先响应
  - 一般来说数据传输率高的设备 (比如磁盘) 的优先级比较高，数据传输率比较低的设备的优先级比较低 (比如键盘)
  - 判断优先级的方法：
    - 硬件方法：中断优先级判别电路

· 菊花链 (daisy chain) 方法：将所有的终端设备串行连接构成优先级电路

* 软件方法：采用轮询的方式来确定优先级，中断源的

5.4.2 DMA

- DMA 方式
  - DMA：直接存储器存取 Direct Memory Access，不需要经过处理器，由 DMA 控制器控制，在外设和主存储器之间进行数据传送，CPU 通过 DMA 控制器对传输方式进行设置，不直接控制传输的过程
  - DMA 传送过程：
    * 预处理：由 CPU 对 DMA 控制器进行设置，确定数据传送的方式，存储器地址和传送的数据量
  - DMA：由 DMA 控制器接管总线和存储器的读写控制，进行外设和存储器之间的直接数据传送
  - DMA 结束：DMA 控制通过中断通知 CPU DMA 结束了，通过执行中断服务程序对 DMA 结果进行检查，确定是否继续 DMA
- DMA 时期的 CPU 工作方式：
  - 停止总线和存储器的访问：DMA 控制器接管了总线和存储器的控制，CPU 停止对于总线和存储器的访问
    * 控制比较简单，但是 CPU 效率很低
  - 周期挪用：DMA 大部分时间用在外部设备的读写中，总线和存储器有相当的时间处于等待状态，CPU 利用这段时间访问总线和存储器
    * 效率提高，控制复杂
  - 轮流访问：两者的折中，将 DMA 过程划分成较小的周期，由 CPU 和 DMA 控制器交替进行总线和存储器的访问

图 97:

楼教授有言曰：计算机总是有一种方法，是最简单和最理想方式的妥协