problem3：2024090904002-胡杨-CS-03

Part 0：相关名词的解释

- 内存代码区（Text 区）
  - 存放编译之后 CPU 可执行的机器指令
  - 只读
- 常量区（Text 区）
  - 储存不会变化的量，包括但不限于 `const int` 、数组名字、结构体名字……
  - 可读可取，但是在程序运行的时候不可以进行更改。
- 全局数据区（data 区）
  - 代码中放在所有函数之外进行定义的相关变量，同时，也存放在函数内存放的静态变量（即用 `static` 进行定义的数据）
  - 可读可取。
  - 内存一旦使用只有在程序结束的时候才会释放。
  - 未初始化的全局变量存储在BSS区
- 堆区（heap 区）
  - ~~一块特别自由的土地。~~ 是一片需要程序员手动维护的内存空间，很大很大（至少相对于栈区而言）
  - 使用 `malloc` 进行申请内存，使用 `free` 进行内存空间的释放。
  - **尤其注意，申请空间，不需要了之后一定要及时释放，否则容易造成内存泄漏。**
  - 实现为链表，并不连续，链表存储的是空闲空间的地址。空间大但是执行效率低。
  - 可读可取
  - 地址由低地址向高地址不连续地延申。
- 动态链接库
  - 区别于静态库，它是“动态的”。
  - 静态库会在程序编译成可执行文件的时候就整个塞进可执行文件，好处是一个人就能跑，但是代价就是文件会非常臃肿。同时，当项目由多个部分组成，这多个部分都有引用静态库的时候，往往会重复塞进去，造成极大的空间浪费。更新时需要重新获取整个包体。
  - 动态库只有在程序载入内存后再去请求使用库，好处是文件特小，坏处就是万一 dll 丢失就完蛋了。更新时只用获取更新的那个动态库即可。不过动态库会存在轻微的性能损失。
- 栈区（stack 区）
  - 一块很智能（）的区域，局部变量存放都在此处，形参、函数返回值也放在此处。
  - 其得名原因就是栈的规则 FILO。
  - 空间由系统进行自动掌控，自动分配自动回收。空间很有限但是执行效率高。
  - 可读可取
  - 地址由高地址向低地址延申：即一开始给定了一个固定的栈底地址，移动栈顶的地址进行存储。
- 整体内存模型，自上而下是栈区、堆区、全局区、常量区、代码区。
- 补充笔记：

    - SRAM 内 包含 栈区、堆区、BSS 、全局区；
    - Flash 内包含 全局区的静态变量、已初始化全局变量、text区
    - 全局变量和静态变量将会在编译时进行初始化，仅一次
    - 在程序编译的时候，程序的已初始化全局变量会与未初始化全局变量分开，前者位于 data 段，后者位于 BSS 区。
    - 寻址的最小单位是一个字节。
- 关于基本单元的构建：锁存器（and - or 锁存器）、寄存器。锁存器（最小的）用于储存一位数据，，寄存器则由数个锁存器组成矩阵放置而产生。SRAM 同样也是由更多的锁存器以矩阵的方式套娃构成更大的空间。

Part 1：Answer Questions

- 什么是“栈溢出”?
    - 根据前置知识：栈区是位于整个结构的顶部，当栈区使用的内存超出了它本能有的大小，就会干涉到其他区域的正常工作，引起一堆莫名其妙的错误。
- 堆区和栈区的区别是什么?
    - 前置知识已叙述
    - 堆区空间大效率低，栈区空间小效率高
    - 堆区由链表维护，栈区由栈维护。
    - 堆区需要自己维护空间，栈区由系统进行维护。
    - 二者地址延伸方向不同：堆区由低向高，栈区由高向低地址。
- 程序运行过程中，内存模型当中的哪些区是只读的，哪些区是可读写的?
    - 只读：代码区、常量区
    - 可读可写：栈区、堆区、全局区
- 如何使用 `malloc()` 、`free()` 函数，它们针对的哪一个区进行操作?
    - 它们针对堆区进行操作。
    - 语法：`typename = malloc(size)` （通常为给指针分配）
        - 特别提醒，当你试图给结构体指针分配空间的时候，需要如下使用：`(struct_name *)malloc(sizeof (struct_name))`
        - ~~当然，如果你闲的没事，也可整整给某 int 指针分配23 byte 这种烂活 ()~~
    - 语法：`free(pointername)` 释放指针指向的空间，而不是释放指针（划重点）
- 为什么要对程序使用的内存进行管理?
    - ~~不管理会爆炸的你真的不看一眼吗~~
    - 内存有限，要及时释放相关无用的内存
    - 内存出问题会导致程序出现各种无法预料的错误，甚至可能影响到其他程序正常运行。

Part2：内存模型的应用

- ~~我是 GPT ()~~
- 依次输出，得到 place.out 文件。输出如下：

    ▶ place.out

- 查询资料得：
  - 0xBFFFFFFF ~ ？ 为栈区，？ 由使用多少决定
  - ？ ~ 0x40000000 为堆区
  - 0x40000000 ~ 0x08048000 为全局区
- 依次进行分析，由观察可知
  - localVar 与 ptr指向的地址十分相近，可以知道二者应在同一个区域内。
  - constValue 与 constString 地址十分相近，可以知道二者位于同一区域内。（位于常量区）
  - globalVar 与 staticVar 地址十分相近，可知二者位于同一区域内。（位于全局区）
  - 再观察，可知 localVarMain 与 localVar 、 ptr 存储的区域大致相近或相同（位于栈区）

Part3：浅谈 Cache

~~一定要学计组吗……~~

- Answer Question
  - 什么是冯诺伊曼体系结构？什么是现代计算机的组织结构？这两者的不同点在哪里？
    - ▶ 冯诺依曼体系结构
    - ▶ 现代计算机的组织结构
      - 但其实还有一个哈佛架构，与冯诺依曼架构相区别，前者可以直接从 flash 执行代码，后者必须将代码存储到 RAM 中再执行。
  - 主存储器是如何工作的？
    - DRAM 是由字线、位线、晶体管、电容、传感放大器、读写驱动程序组成。
    - DRAM 的工作主要分为两类：读取、写入。
      - 读取：当 DRAM 需要进行读取的时候，会接收到一个 31 位的二进制数。前三位将会用于标识阅读这个命令，接着五位会用于在整个庞大的库中寻找特定的库。寻找到库之后，这个库之内的所有的字线将会被关闭，用于保存其储存的内容。此时，会将所有的位线全部预充电到 0.5 V 。在这之后，接着的 16 位的二进制数将会用于寻找需要读取哪一行，并将这一行全数连接到他们的位线上。此时他们原本存储的电荷（0|1）将会与位线相通，并且造成影响：传感放大器将会检测到微小电流的变化&方向，使得它们更保持（刷新）原本的电压。此时最后 8 位将会寻址确定是哪八列，并且将他们的列接到驱动上，将他们存储的内容（电压）进行输出。
      - 写入：大体同上，但是在最后会由驱动反向更新其中的电压值。
    - 补充：存储变量的时候，为了便于寻址，会选择使用牺牲内存来保证 **数据对齐。**
    - 补充：与主存相区别，有一个东西叫做内存。内存 = 主存（DRAM） + 高速缓存（SRAM）（Cache）。
  - 什么是Cache的局部性原理？它包括哪些方面的内容？
    - 局部性原理分两个部分，包括空间局部性和时间局部性。
    - 具体解释为：我们要用的数据在内存上是大致相近的（比如循环访问连续地址），同时在未来可能用到的数据，很可能是现在正在使用的数据（比如对一个数连续进行操作）（循环的指令会被重复使用）。
  - Cache的运用为什么可以加快系统整体性能？
    - CPU 的性能提升远远快于物理内存的性能提升，出现了 CPU 等待 RAM 提供数据的情况（CPU 空闲，RAM 满载）

        - 于是这个时候，Cache 选择了用大小换性能，大幅提高运行速度的同时大幅砍掉了大小，以便于在 CPU 与 RAM 之中构建一处缓冲区域，减少上述情况的发生，尽可能不让 CPU 挂机，以提高整体运行性能。
        - 具体：Cache 利用我们运行程序时具有局部性的特点，提前将可能访问的区域从 RAM 拉取到 Cache 中，在 CPU 申请原本在 RAM 中的数据之前，就带到 Cache中来。
        - 再补充：CPU 访问数据顺序：寄存器、一缓、二缓、三缓、RAM、硬盘。访问速度依次递减，存储大小依次增加。
- 接下来是自己的学习笔记，可以 pass （防止审题人被又臭又长的资料气昏过去（）

  ▶ Cache 分类
- Cache 工作原理
    - 放置
        - 放置方式：全相联、直接映射、组相连
        - 分别为，可以放在 Cache 的任意位置，可以放在 Cache 的某一行、可以放在 Cache 的某几行
        - 相连的组数越大，比较电路越大（越臃肿），但 Cache 的利用率更高，Cache Miss（Cache 中无所需数据）的发生概率小
    - 读取
        - 比对 TAG 确定是否命中（Cache 中包含所需的数据，利用和 TAG 一起的 Index 进行访问读取）
        - 当发生 Cache miss 的时候，我们需要对 Cache 进行数据的替换（或者直接转发地址到主存进行访问，不进行替换）。
    - 替换
        - 随机替换：直接不管三七二十一从 Cache 划出一片空间覆盖掉
        - LRU ： 最近使用的块最后替换（时序）
        - FIFO：先进先出（空间顺序）
    - 写
        - Cache 实际上是一个临时缓存，是主存中一部分数据的拷贝，当 Cache 发生了写操作的时候，就注定了它和主存内的原数据会产生差异，因此需要尤其注意。
        - 有以下三种方案能解决这个问题
            - 通写：把数据写回 Cache 的同时，将数据写回主存
            - 回写：先把数据写回 Cache，在这部分数据被替换掉的时候再更新主存
            - 通写队列：先写回一个队列，然后慢慢往主存写
        - 在三个方案之中，通写极度影响读写效率，一般不采用。
    - 在现代的计算机多核常态化的情形之下，需要面对另一个问题：Cache 一致性。
        - 具体描述为，在多核的状态下，主存内的数据在 Cache 中被某核更改了，却被其他的核在不知情的情况下误用了，为了避免这种情况的出现，我们就需要保证 Cache 的一致性。
        - 解决方案有两种
            - 监听：在 Cache 中数据被写了的时候，将其他 Cache 中的同一数据都写了，或者都置于无效
                - 性能开销极大，一般不采用。
            - 基于目录：在主存处维护一块表，记录哪些数据被写入了 Cache 更新相应的状态。

- SI：一块数据有两种状态：share or invalid。invalid 状态下可以使用，share 下其他 Cache 的对应数据无法使用。
- MSI： 基于 SI ，添加了 modified 状态，表示这个数据仅属于某个 Cache 。
- MESI： 基于 MSI，添加了 exclusive 状态，用于标识和其他 Cache 不依赖。要写的时候只需将它修改为 M

  ▶ 以 MESI 简述过程

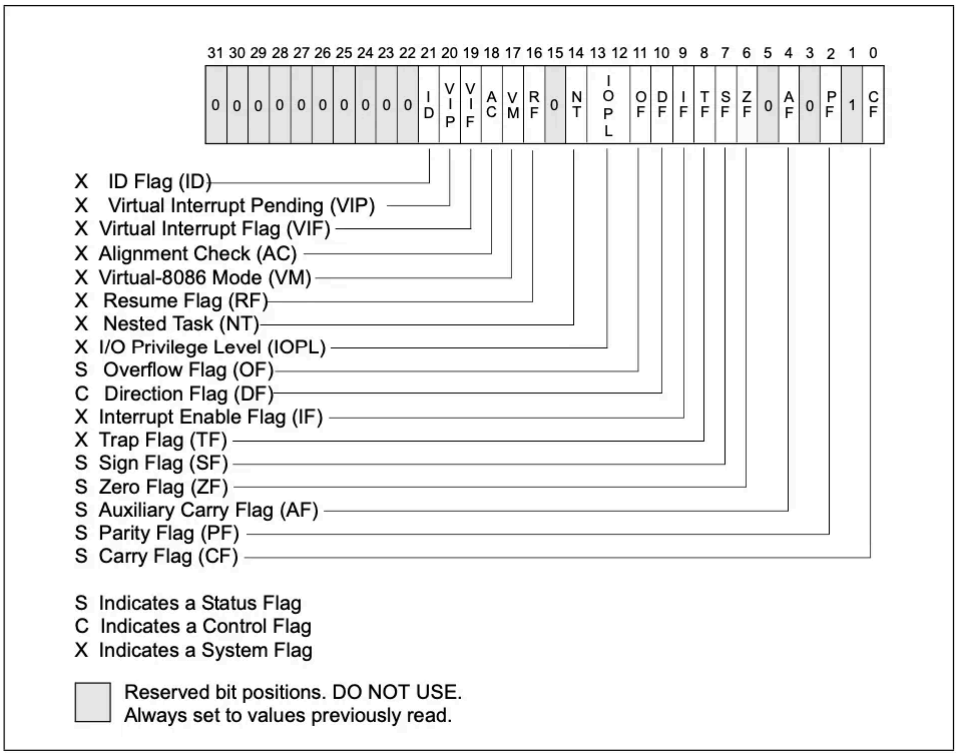
Part end：额外知识

P.S. 没看懂该怎么用（）是纯理论吗（

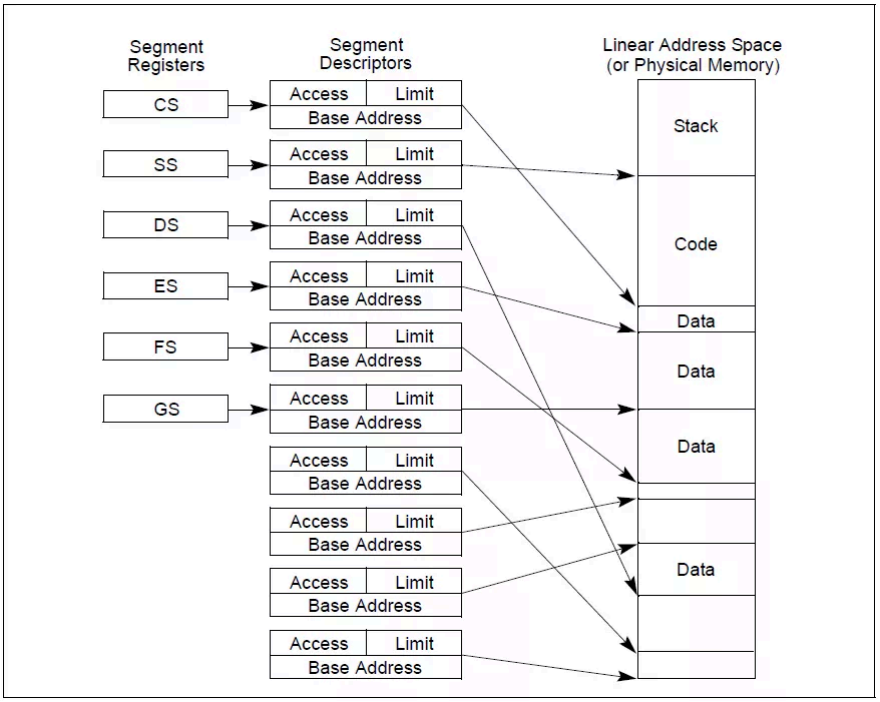
- 在 google 的时候，惊奇的发现函数名字其实也是指针，就想知道为什么函数未定义会报 CE 而不是像野指针那样发电。再 google 了一下，发现原来编译器对函数有着特殊处理。在编译的时候，编译器会对函数进行检查，确定它指向一个有效的函数定义。 ~~所以为什么野指针你不检查一下（）~~
- 大端模式和小段模式：关于低位字节、高位字节的地址放置。
  - 小端模式：低位字节存在低地址上，高位字节存在高地址上。
  - 大端模式：低位字节存在高地址上，高位字节存在低地址上。
- `calloc` 函数：与 `malloc` 基本一致，只不过多一个初始化空间为0。
- 页表 & 页目录：虚拟空间中连续的 4GB（x86）将会被映射到实体内存上并不连续的区段。内存管理时，需要依据 4 GB 的大小建立映射表，但很有可能吃不完 4GB ，但是映射表不能少，造成了空间的浪费。因此将其拆成众多页表（更小一级的内存大小集合），对页表进行维护，再由页表映射到实体物理地址（4 MB）上去。由于我们需要使用单独的页表，就需要知道每个页表的起始空间，页表起始空间的集合即页目录。

  省流：多级映射，需要时再开辟，节约空间。
- 关于寄存器的了解(x86)（说实话，没看懂怎么用，似乎仅仅是理论知识？）
  - x84 对 x86 的兼容是仅读取后16位。
  - 通用寄存器（括号内为 x64）
    - **eax**: 通常用来执行加法，函数调用的返回值一般也放在这里面（rax）
    - **ebx**: 数据存取（rbx）
    - **ecx**: 通常用来作为计数器，比如for循环（rcx）
    - **edx**: 读写I/O端口时，edx用来存放端口号（rdx）
    - **esp**: 栈顶指针，指向栈的顶部（rsp）
    - **ebp**: 栈底指针，指向栈的底部，通常用 `ebp+偏移量` 的形式来定位函数存放在栈中的局部变量（rbp）
    - **esi**: 字符串操作时，用于存放数据源的地址（rsi）
    - **edi**: 字符串操作时，用于存放目的地址的，和esi两个经常搭配一起使用，执行字符串的复制等操作（rdi）
    - **（r8 ~ r15）**： 原本 x86 调用函数常用线程的栈进行传递，在 x64 中改为了更多使用寄存器传参。减少对内存的读写，提高了速度。

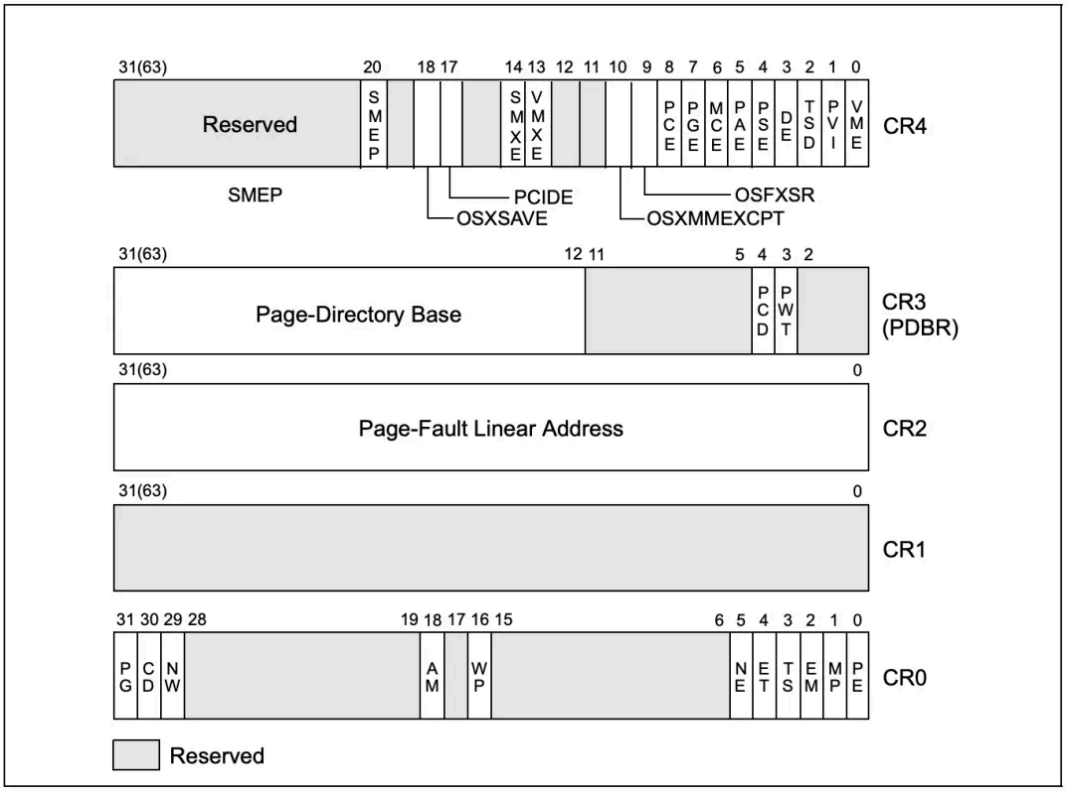
- 标志寄存器



- 指令寄存器：eip（rip）指向下一条命令存放的地址。
- 段寄存器（和内存模型紧密相关）
  - 和计算机的寻址紧密相关。计算机将内存空间进行分段，每一个段的起始地址就存放在段寄存器中。当CPU处于16位实地址模式下时，段寄存器存储段的基地址，寻址时，将段寄存器内容左移4位（乘以16）得到段基地址+段内偏移得到最终的地址。当CPU工作于保护模式下，段寄存器存储的内容不再是段基址了，此时的段寄存器中存放的是**段选择子**（Index : 索引表 + TI : GDT or LDT+ RPL : 特权级），用来指示当前这个段寄存器“指向”的是哪个分段。



- 控制寄存器



  - **cr0**: 存储了CPU控制标记和工作状态：是否启用分页、是否启动内存对齐自动检查、是否开启内存写保护、是否开启保护模式。
  - **cr1**: 保留未使用（reserved），空闲
  - **cr2**: 页错误出现时保存导致出错的地址（便于跳转）
  - **cr3**: 存储了当前进程的虚拟地址空间的重要信息——**页目录**地址（由于页对齐，所以仅高 20 位有效，后 12 位在写入的时候必须全 0，读取的时候忽略后 12 位，后十二位往往留给更高一级的寄存器使用）其中含有两个标志位 PWT & PCD。

    PWT(Page Write Through)，PWT=1时 写Cache的时候也要将数据写入内存中。

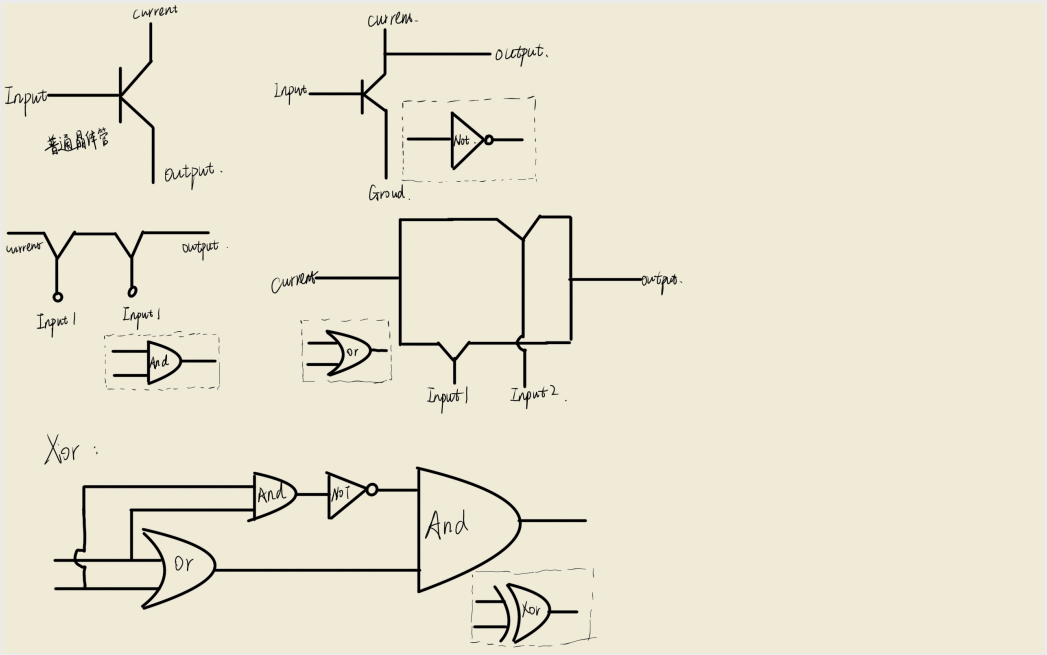
    PCD(Page Cache Disable)，PCD=1时，禁止某个页写入缓存，直接写内存。
  - **cr4**: 保护模式下提供对 Vitrual-8086 的支持，启用 I/O 断电，页面大小扩展和机器检查异常。
  - **（cr8）**: 64位新增扩展使用，提供对任务优先级寄存器的读写访问，允许调整优先级。
- 调试寄存器： CPU 提供对调试的支持。
  - 软中断指令（int 3 0xCC（烫（））执行时，触发中断处理流程（可以理解为一个函数跳转？），CPU 取出 IDTR 寄存器指向的中断描述符表 IDT 的第三项，启用里面的中断处理函数。（与寄存器无关）
  - 通过接口设置硬件断点，能使 CPU 在执行命令的过程中，当满足条件（例如某块内存值如何）就自动停止。
- 描述符寄存器
  - GDTr 指向 GDT
    - 全局的，用于描述内存段集合的表（页表）
  - LDTr 指向 LDT
    - 局部的，用于描述一个内存分段的信息（段选择子？）
  - IDTr 指向 IDT

        - 描述中断、异常时候去哪里执行处理函数
        - 包含中断门、任务门、陷阱门
        - 软中断走的是陷阱门，真正的中断走的中断门。
    - 任务寄存器
        - 用于指向当前运行的任务，切换寄存器指向即切换任务（虽然大家都不用）
    - MSR寄存器
        - 变变变功能不固定
        - IA32_SYSENTER_CS

          IA32_SYSENTER_ESP

          IA32_SYSENTER_EIP

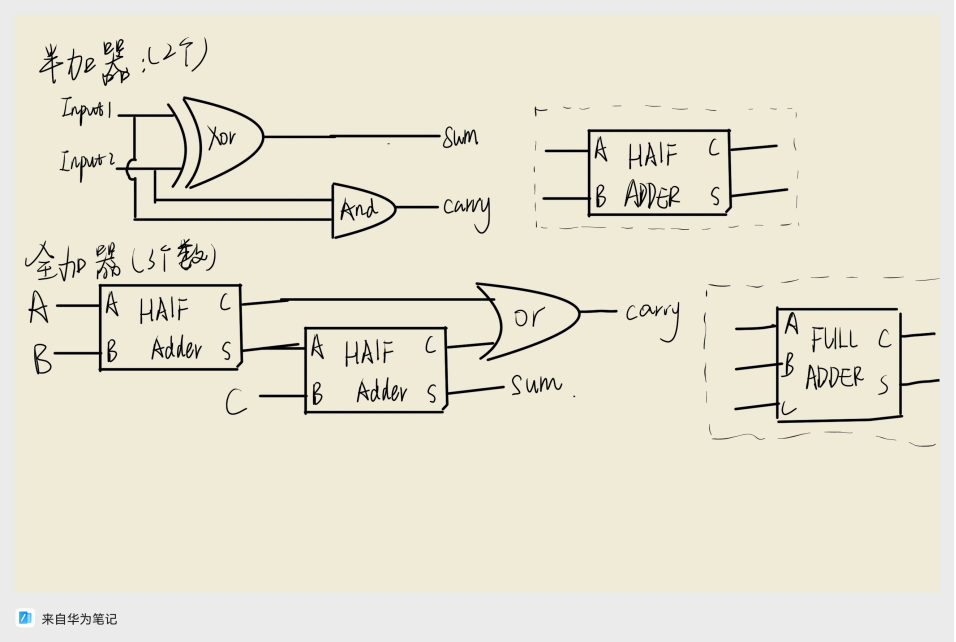
          用于系统快速访问（最高权限指令最优先性能）
- 关于二进制的补充
    - 原码：是什么就是什么
    - 反码：正数不变，负数各位依次取反
    - 补码：正数不变，负数为反码加一
- 让我们从物理重新再开始一遍
    - 逻辑门



来自华为笔记

    - 构建 ALU（算数逻辑单元）
        - 计算：通过半加器、全加器等逻辑门的组合进行（观察到 XOR 的运算法则和加法很相近）

- 举例：半加器和全加器



- 逻辑单元：为计算||判定进行逻辑门组合
  - 举例：判断结果是否为 0 的逻辑单元（把输入的两两 OR 起来，并依次 OR 结果，直到最后一个，再连上一个 NOT 即可）

- （因出题人提示没必要，中途终止（）