CPU控制信号自动编程实验

摘要：运用自编程算法，对CPU内部控制信号进行排列枚举，从而构建出能够完成间接寻址、二级指针寻址、内存单元之间互相传值等任务的复杂指令序列。

关键词：自编程算法、摸索学习过程、☆分层次组合

作者：蔡超联

实验编号：32

编程语言：C++

实验成功日期：实验3年（公元2018年）3月17日

读者若对自编程算法理论不熟悉，请参阅视频：<http://tieba.baidu.com/p/5798422641>

目录

[一、《计算机组成原理》背景知识及实验改动 2](#_Toc509157754)

[1.主机框架图 2](#_Toc509157755)

[2.细节改动 2](#_Toc509157756)

[3.读取内存单元的过程 3](#_Toc509157757)

[二、自编程算法的试错流程 3](#_Toc509157758)

[三、元操作 4](#_Toc509157759)

[四、内层枚举：构建指令 5](#_Toc509157760)

[1.用数码表示寄存器 5](#_Toc509157761)

[2.用数码表示指令函数 5](#_Toc509157762)

[3.用3位十进制数表示一句完整的命令 5](#_Toc509157763)

[五、外层枚举：对指令语句进行排列 6](#_Toc509157764)

[六、程序运行过程 8](#_Toc509157765)

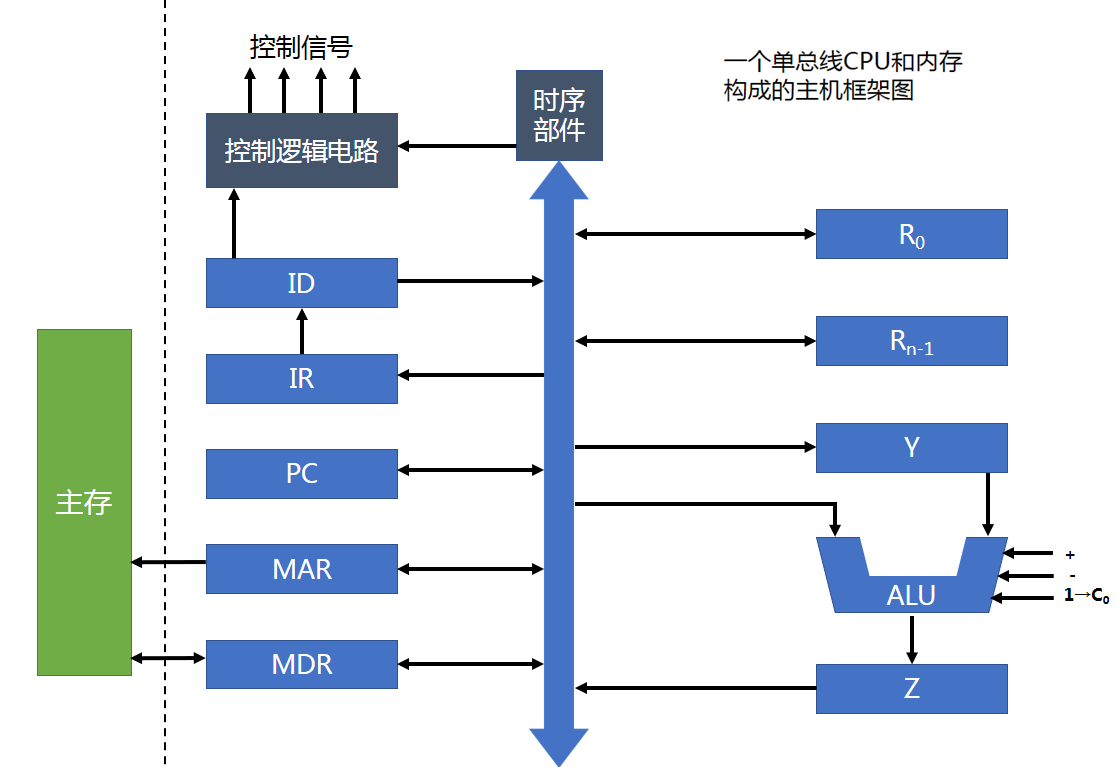
[1.实验一：让计算机通过试错实现间接寻址功能 8](#_Toc509157766)

[2.实验二：让计算机通过试错实现二级指针传值功能（论证“按层次组合对于提高自编程算法收敛性的重要性”） 9](#_Toc509157767)

[3.实验三：内存单元之间传值 12](#_Toc509157768)

# 一、《计算机组成原理》背景知识及实验改动

## 1.主机框架图



上图给出了一个简化的单总线CPU和内存构成的主机框架图。

存储器地址存储器MAR用于保存当前CPU所访问的内存单元地址。存储器数据寄存器MDR是CPU和主存以及外部设备之间信息传送的中转站。

R0——Rn-1 为通用寄存器。

运算器包括算术逻辑单元ALU和暂存器。ALU完成各种算术运算和逻辑运算，暂存器用于存放ALU运算数据和结果。在上图所示的计算机中，Y是ALU的输入暂存器，存放一个需要ALU运算的数据。Z是ALU的输出暂存器，存放ALU运算后的结果。ALU有多种运算，控制信号比较多，上图所示的计算机简化这些控制信号，其中+表示加法信号，-表示减法信号，1→C0表示ALU低位进位置1的控制信号。本次试验只用到前两个信号。

## 2.细节改动

①IR寄存器用于存储指令内容。本实验对此寄存器稍作改动，设有IR1、IR2两个寄存器，IR1存储指令的第一个立即数参数，IR2存储指令的第二个立即数参数。

②本实验再增设一个寄存器指针RegPtr，它用于指令的第一个寄存器参数。

③为了实验的方便，我强制规定：一个指令只能改变一个寄存器或者一个内存单元的值。一条命令的第一个参数，要么是被改变的寄存器（于是RegPtr就指向这个寄存器），要么是被改变的内存单元的地址（于是IR1=该内存单元地址）。指令的第二个参数可以是立即数也可以是寄存器，如果是立即数，那么这个立即数就会被存入IR2寄存器；如果是寄存器，那么就将寄存器的值存入IR2寄存器。

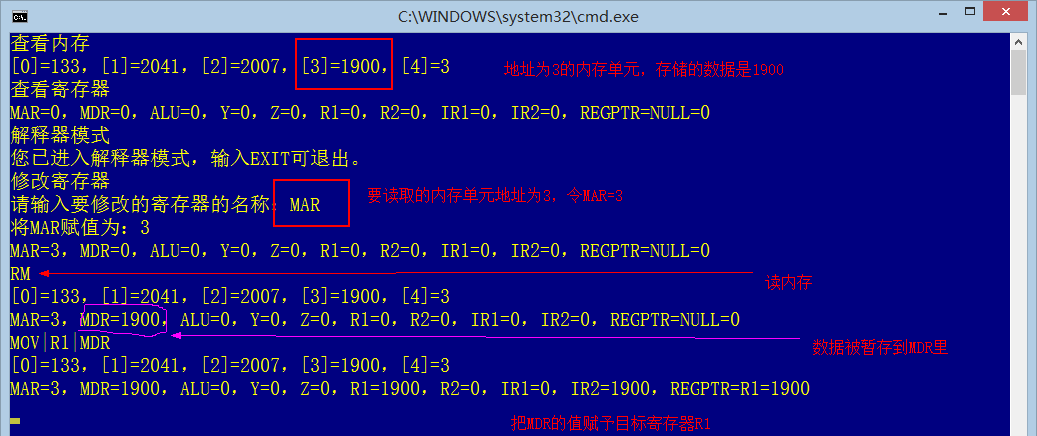
以汇编语言里的传值指令MOV为例，假设用户输入了MOV R1,8 这一条命令。那么程序会令RegPtr指向R1寄存器，由于左边不是立即数，IR1寄存器完全没有用，就清零；同时IR2被赋值为右边的立即数8。

④本实验规定：IR1、IR2不能被随意修改，其余任何两个寄存器之间可以相互传值，命令格式为：MOV 目的寄存器，源寄存器。另外，目的寄存器指针RegPtr不能被用户随意修改。

## 3.读取内存单元的过程

CPU读取内存地址为X的单元处的数据并将数据传给R寄存器（指令写作 **传值\_间接寻址|R|X**），需要经过以下几个步骤：

1. IR2→MAR，此后MAR寄存器的值等于X；
2. 发出RM读内存命令，地址为MAR（X）的内存单元的数据被读到MDR寄存器中；
3. MDR→R，MDR暂存器的值被传给目标寄存器R。

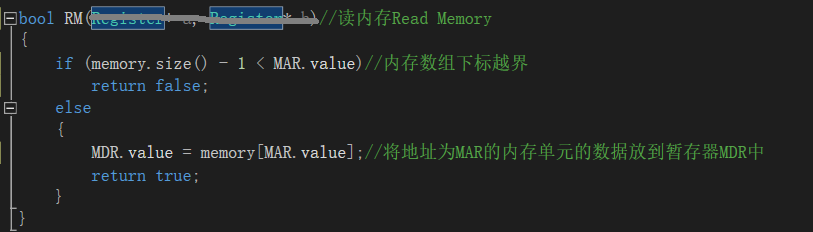


# 二、自编程算法的试错流程

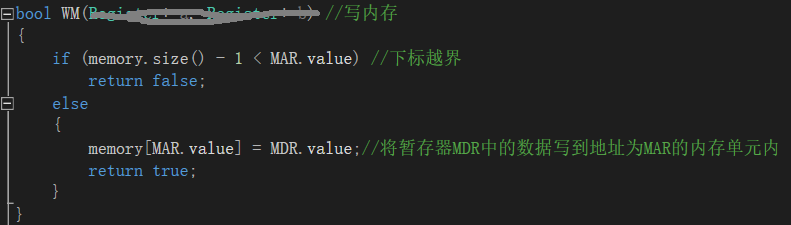
1. 计算机枚举得到所有方案，方案编号i初始化为0
2. 用户指定目标
3. 计算机备份寄存器和内存数据
4. 尝试当前方案（编号为i的方案）
5. 如果成功（达到目标状态），保存到候选方案集合中
6. 方案编号i加一，转第3步；如果穷举完所有方案，就将所有方案设定为候选方案，进行下一步淘汰，转向第2步，让用户修改一下条件，重新设定一个目标。

# 三、元操作

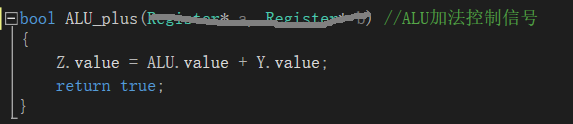
1. 读内存RM



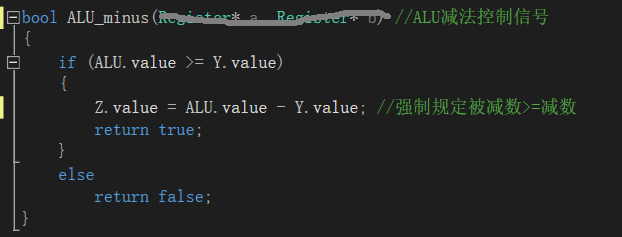
1. 写内存WM



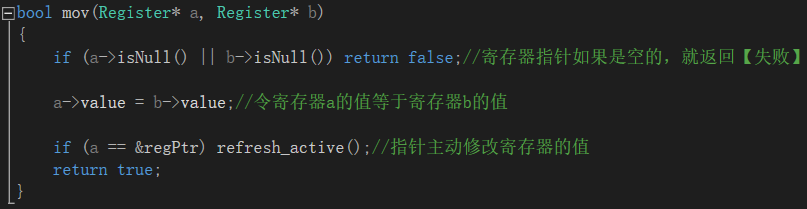
1. ALU加法控制信号：PLUS



1. ALU减法控制信号：MINUS



1. 寄存器之间传值 MOV



# 四、内层枚举：构建指令

## 1.用数码表示寄存器

把所有寄存器放在全局数组里面，0号对应MAR，1号对应MDR，7号对应IR1……8号对应IR2,9号对应目标寄存器指针RegPtr。

代码：Register\* registers[10] = { &MAR,&MDR,&ALU,&Y,&Z,&R1,&R2,&IR1,&IR2,&regPtr};

## 2.用数码表示指令函数

定义函数指针类型FunctionPointer，它只能指向返回值为bool型且接受两个寄存器参数的函数（这就是为什么我强行给WM、RM那些无参函数加上参数）。

用一个全局数组来存放函数指针，0号对应RM，1号对应WM，2号对应PLUS，3号MINUS，4号对应MOV，同时用一个字符串型数组存放它们的名字。

代码：

typedef bool(\*FunctionPointer)(Register\*,Register\*);//函数指针

FunctionPointer basicFunctions[BASICFUNCTION\_LENGTH] = { &RM,&WM,&ALU\_minus,&ALU\_plus,&mov };

string names\_basicFunctions[BASICFUNCTION\_LENGTH] = { "RM","WM","PLUS","MINUS","MOV"};

另外再定义一个全局的变长数组，用于存放大型函数（通过自编程算法建立的函数），

第一个大函数编号为5，第二个大函数编号为6……以此类推

## 3.用3位十进制数表示一句完整的命令

以传值命令“MOV MAR,MDR ”为例。这句命令的指令函数是MOV，对应的编号为4；目标寄存器为MAR，编号为0；源寄存器是MDR，编号为1。所以整句命令被表示为401.

第二个参数可以是立即数，立即数存放于IR2寄存器中。

以命令“MOV MAR，3”为例。立即数3被存放在IR2寄存器内，IR2寄存器对应的编号是8，所以整句命令被表示为408而不是403.

由于禁止对IR1和IR2进行操作，因此三位数的十位数码不能是7或8.

由于写内存、读内存、ALU加法信号、减法信号都是无参函数，因此它们的十位和个位全部只能为0.（000、100、200、300）

另外，目标寄存器不能和源寄存器相同。

# 五、外层枚举：对指令语句进行排列

用全局变长数组存储存储第三小节枚举得到的所有语句。读者应该能够看出，语句的三位十进制数码是不连续的，比如由于十位数字不能是7或8，所以469的下一个数字不是470而是490。

但是，数组的下标是连续的。这样我们就能用单层的for循环来遍历所有的可能性。

举个例子：假设用户要求计算机用穷举法完成某个任务，规定穷举得到的解决方案只能包含3步指令，不能更多。现在全局数组里面存储了75条命令，也就是说计算机就只能对这75条命令进行排列（可重复使用），排列出来的指令序列长度为3.结果如下（数字代表下标）：

【0,0,0】

【0,0,1】

【0,0,2】

……

【74,74,73】

【74,74,74】

那么总共有75^3 = 421875 种方案。

这是一个三重循环，我们怎么把它写成单层循环呢？一种普遍的方法是写个递归函数，但我不喜欢这么做。我的办法是——把数组转化为整数。

比如【2,5,34】就算作：2\*(75^2)+5\*75+34=11659.

需要用的时候，就把11659转化为75进制数，

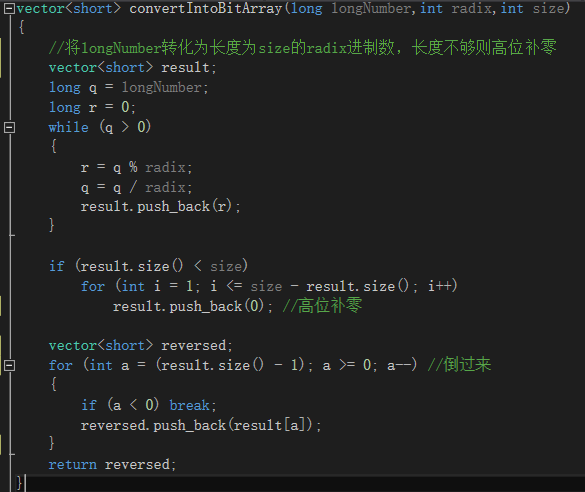
高位=11659/(75\*75)=2，余数为409；

中间位=409/75=5，余数为34；

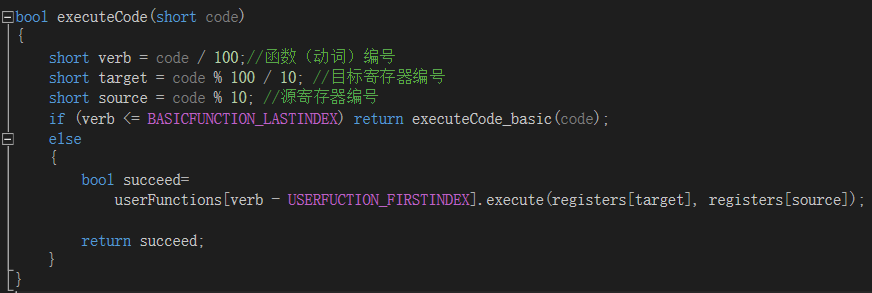
低位=34。

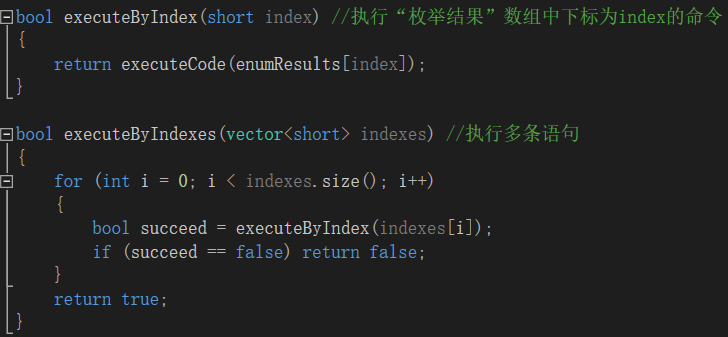
执行命令的时候，先取出高位的命令。高位数值是2，就取出下标为2的命令——ALU\_PLUS; 再取出下标为5（中间位）的命令——MOV MAR,MDR，再取出下标为低位的命令。

进制转换函数的代码：



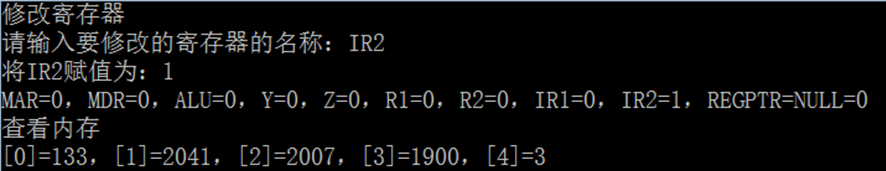
函数executeCode：



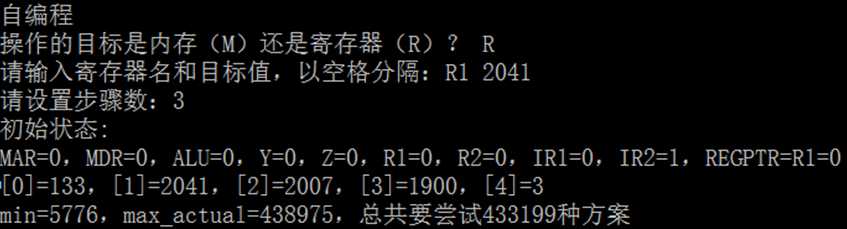


# 六、程序运行过程

## 1.实验一：让计算机通过试错实现间接寻址功能

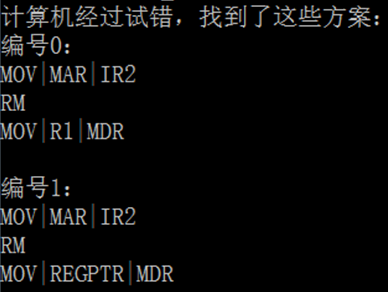


假设我们要把地址为1的内存单元的数据（2041）传递给R1寄存器。立即数（内存地址也看作立即数）是1，于是我们修改寄存器，令IR2=1



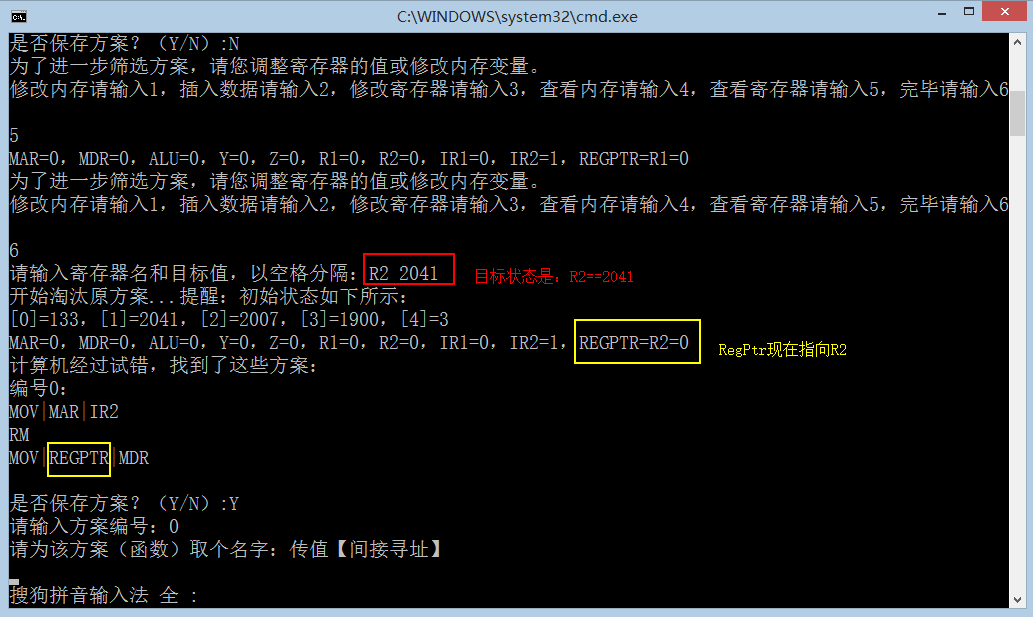
我们向计算机输入需求，要求计算机构建一个解决方案，使得R1寄存器的值等于2041，必须在3步之内解决问题。由于被操作的寄存器是R1，于是RegPtr指针指向R1寄存器。

计算机经过试错得到如下方案



读者不难发现，这两个方案只有最后一步指令是不同的，“MOV|R1|MDR”表示把MDR的值赋予R1寄存器，如果我的目标是把某个内存单元的值赋给R2寄存器或者其他寄存器，那么这个指令就不能起到作用了；后一个方案中的“MOV|REGPTR|MDR”表示把MDR寄存器的值赋予目标寄存器，如果我要操作的寄存器是R2那么RegPtr就指向R2，如果我要操作的寄存器是Y那么RegPtr就指向Y。RegPtr不是固定的。

现在设定新的目标“R2==2041”，也就是：将地址为IR2（即1）的内存单元的数据（即2041）传给R2，效果如截图所示：



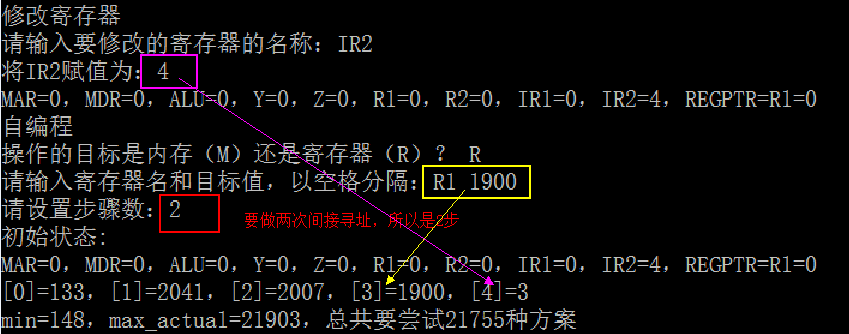
我们给新建的函数取个名字，叫做 传值【间接寻址】。

## 2.实验二：让计算机通过试错实现二级指针传值功能（论证“按层次组合对于提高自编程算法收敛性的重要性”）

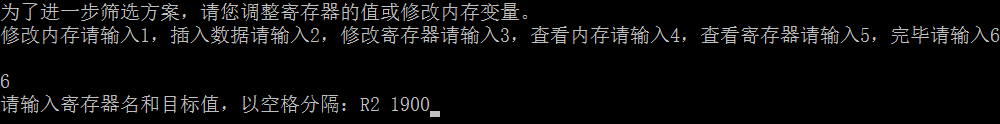
“二级指针”是C语言里的概念，相当于做两次间接寻址，最终取出内存某个单元里的数据。

比如地址为4的内存单元里面存储的数据为3，内存为3的单元里存储的数据为1900.

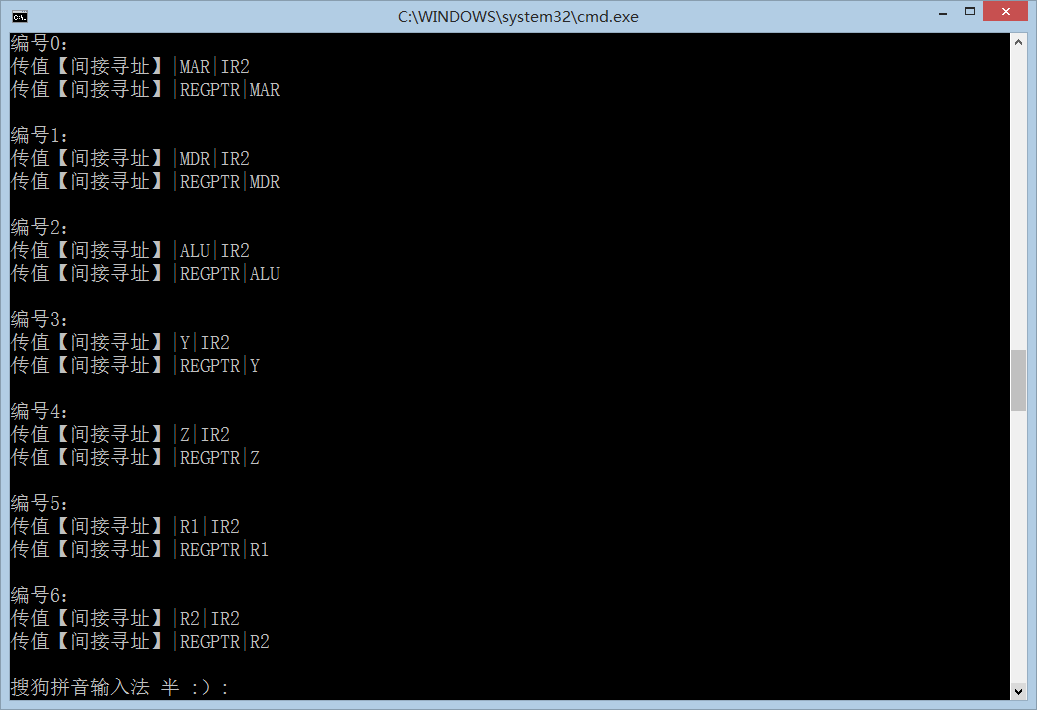
那么。可以看出，对“4”做两次间接寻址，可以得到数据1900.



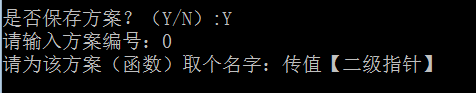
计算机枚举得到若干种方案，为了避免偶然性，我们再修改一下目标：R2==1900，



这次计算机经过筛选得到了6个正确方案



我们随便选择其中的一个（比如编号0），取名为：传值【二级指针】

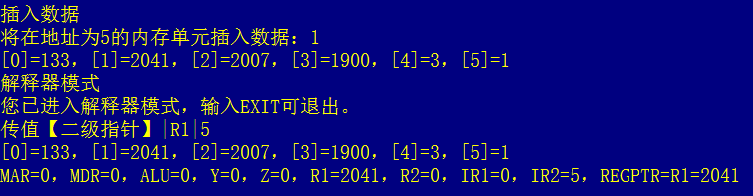


现在我们来手工验证一下效果

在内存地址为5的单元插入数据：1。

这个数字1代表一个一级指针，指向地址为1的内存单元，而这个内存单元存放的数据是2041.

现在我们执行命令：传值【二级指针】|R1|5，最终R1寄存器的值是2041。



☆☆☆☆☆☆☆☆☆☆☆☆☆

计算机通过自编程算法，学会了如何进行间接寻址以后，它只要进行两次间接寻址，就可以模拟出“二级指针”的效果。

但是，假如计算机不知道如何进行间接寻址呢？

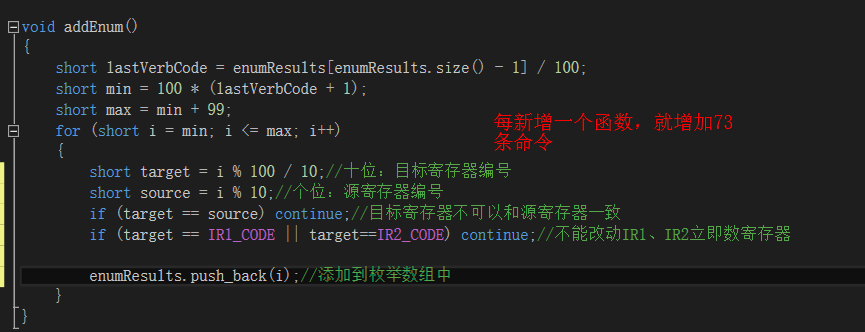
那么，计算机需要5步才能模拟出“二级指针”的效果：

* 1. MOV MAR，IR2
  2. RM //此时地址为IR2的内存单元的数据（暂记作x）被读到MDR暂存器中
  3. MOV MAR,MDR
  4. RM //此时地址为x的内存单元处的数据被读到MDR寄存器中
  5. MOV REGPTR，MDR //将MDR寄存器的值赋予目标寄存器

枚举五步，总共有75^5=23 7304 6875种方案。

而计算机在学会间接寻址的情况下，只需要两步就可以实现二次间接寻址，总共尝试次数为148^2=21094种方案，两者简直不是一个数量级的！

这里不详细说明148这个数是怎么来的，大家看图自己理解一下。



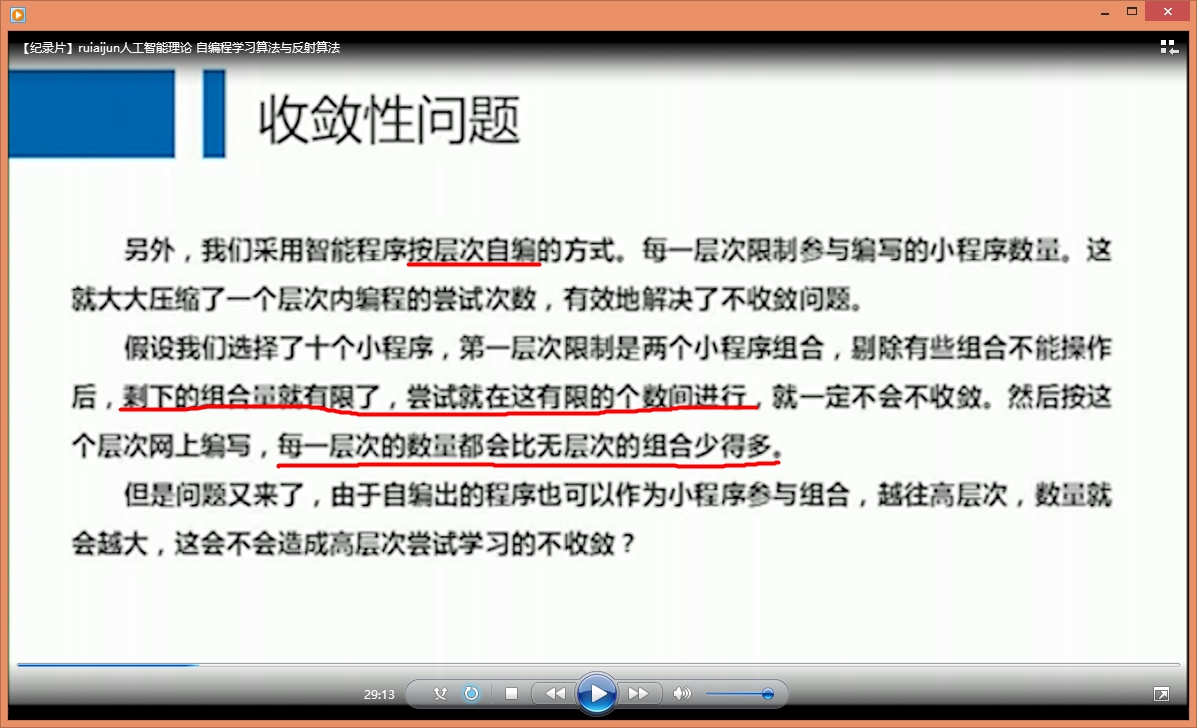
MOV寄存器传值指令、RM读内存指令、WM写内存指令都属于“底层指令”，或称为“元操作”。

相对地，“间接寻址”属于高层指令，或称“后代操作”。它是由若干底层指令排列组合而成的，蕴含的信息量更大。

同样的任务，用高层指令完成只需要2步，而用底层指令完成却需要5步。我们知道，步骤数越大，则枚举的次数越多。

符合以下公式：枚举次数＝总语句数∧步骤数

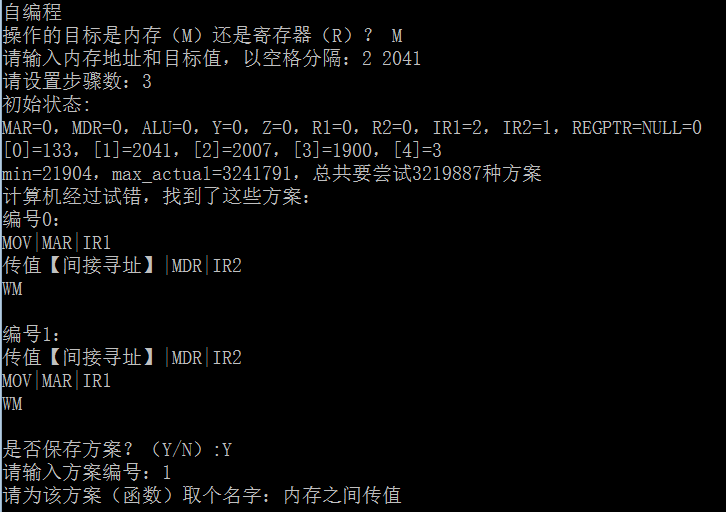
在本实验中，总语句数＝75＋(函数个数－5)×73 。这是个一次函数。计算机每学习得到一个新函数，虽然总语句数增加了，但是完成任务所需的步骤数却下降了，总体上枚举的次数也大大降低了。



纪录片《RUIAIJUN人工智能理论：自编程学习算法与反射算法》截图

（观看链接：<http://tieba.baidu.com/p/4760009643>）

## 3.实验三：内存单元之间传值



验证效果：将地址为2的内存单元的数据（2007） 传递给地址为3的内存单元。

