汇编语言 速效救心丸

帮助快速上手汇编语言编程，虽然只限于编程

VScode配置调试环境

1. 安装插件TASM/MASM
2. 右键扩展设置，选择Assembler：MASM
3. 右键调试即可开始调试了！

Debug.exe

> R命令：查看、改变CPU寄存器的内容 D命令：查看内存中的内容 T命令：执行一条机器指令 G命令：从停顿的地方运行到底

第一段代码：Hello World

```
DSEG SEGMENT
      MESS DB 'Hello, World!',0DH,0AH,24H
DSEG ENDS

SSEG SEGMENT PARA STACK
         DW 256 DUP(?)
SSEG ENDS

CSEG SEGMENT
          ASSUME CS:CSEG, DS:DSEG
     BEGIN:MOV    AX,DSEG
          MOV    DS,AX
          MOV    DX,OFFSET MESS

          MOV    AH,9
          INT    21H

          MOV    AH,4CH
          INT    21H
CSEG ENDS
END BEGIN
```

基础指令

用以下指令可以写一个基础的程序：

1. 段定义+Assume

```
XXX SEGMENT(XXX:DATA/STACK/CODE)
XXX ENDS
```

```
ASSUME CS:CSEG, DS:DSEG, SS:SSEG
MOV AX,DSEG
MOV DS,AX
MOV AX,SSEG
MOV SS,AX
```

2. 数据定义

```
(ORG 1000)
(NAME) DB ?/...
(NAME) DB N DUP(?/...)
db:12H/dw:1234H
```

3. MOV

```
 MOV AX,Y
 MOV Y,AX
 MOV AX,BX
```

4. +-

```
ADD AX,X;AX+=X
SUB AX,X;
INC AX;AX++
DEC AX;AX--
```

```
NEG AX ;取负
```

5. 程序的终止

```
MOV AH,4CH
INT 21H
```

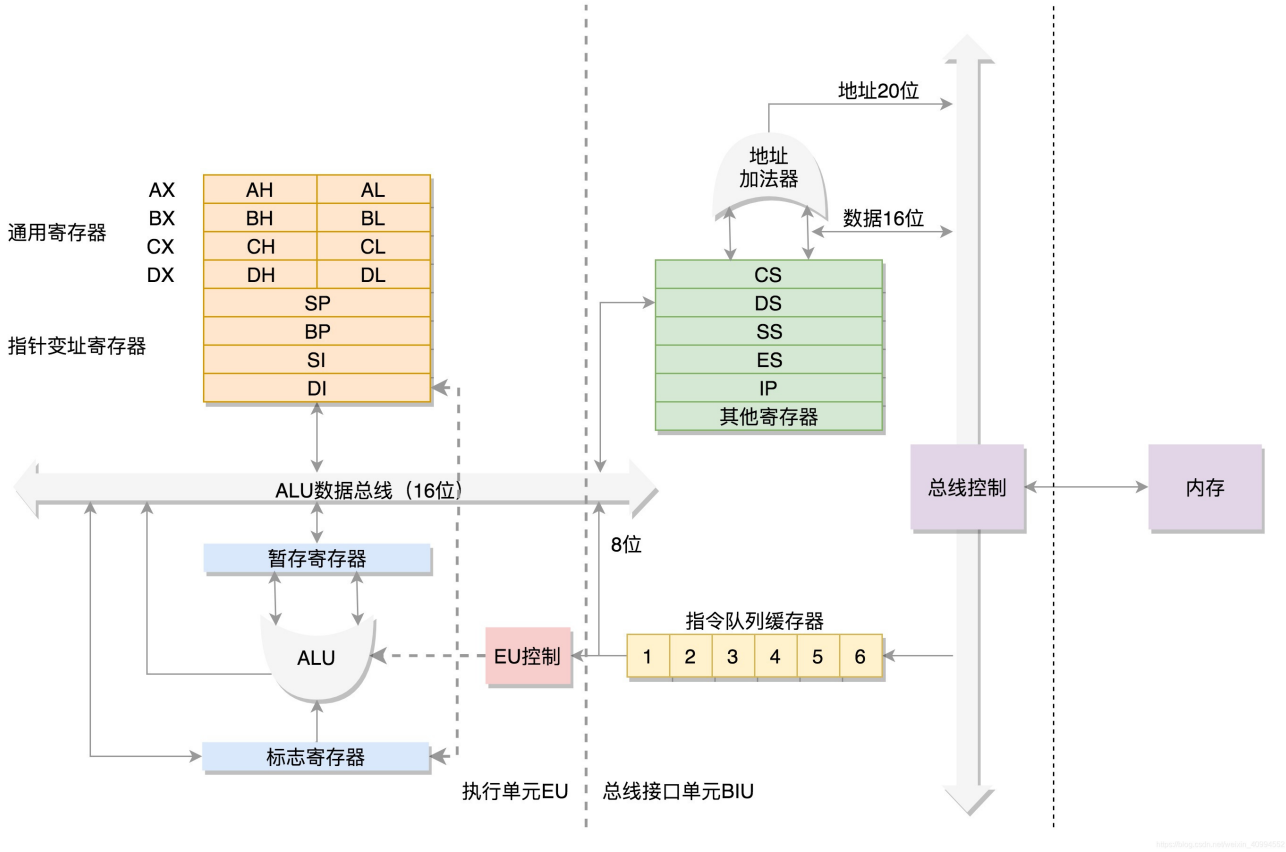
稍后会讲的进阶指令：

1. Label和JUMP：跳转
2. 分支和循环
3. 堆栈的使用
4. <函数>：PROC和MACRO
5. INT 21H指令：输入/输出

寄存器的使用

在汇编语言中，我们不能对内存中的数据进行直接操作，如果要操作，需要把数据先`MOV`到寄存器中再进行处理。 为了写代码的过程更顺滑，最好先简单了解一下这些寄存器的使用，当然，你也可以只了解通用寄存器的使用，别的等用到了再去查询。 我会尽可能简单地表述。

8086 CPU 中有14个16位寄存器 。16位的存储可以用16进制表示，BeLike：`123AH`（H表示16进制)在查看内存情况的时候，由于数据从高位到低位存储，BeLike：`3A 12`



通用寄存器：AX，BX，CX，DX

```
DSEG ...
X DB 12H
Y DB ?
....

CSEG...
...
MOV AH,X
MOV Y,AH ;Y--12H
```

可以拆分为两个寄存器使用（AH和AL），不过各自有各自的独特作用，用到再提，这和它们的名字是关联的

说到底为什么<通用>寄存器会有<独特的作用>？ 这是因为一些内置的指令依靠**固定**的寄存器传递参数，所以这些寄存器也有了独特的作用

一般来说随便用就可以，反正里面的东西不久存，只是用来做中转

AX：Accumlator 累加器

特殊功能和MUL/DIV有关，后面再说

BX：Base 基地址寄存器

可以存储地址并访问 说到地址，就得提一下汇编语言里地址的表示方法 在汇编语言里，内存中的地址BeLike：`204B:1001`（以16进制表示） `204B`是**段地址**，`1001`是**偏移地址**，各需要一个Word进行存储 有两个指令对应的获取内存单元的这两种地址 `SEG`可以获取**段地址**（这个段就是指我们程序对应的段Segment），`OFFSET`可以获取**偏移地址** 使用这两个词只需要在MOV时加在变量前即可，比如`MOV BX offset X` 在“通过地址找内容”这件事方面，一般用BX存储偏移地址 比如：

```
X DW 1234H
Y DW ?
...
MOV BX, OFFSET X;BX中存储了X的偏移地址
MOV Y, [BX];BX存储的偏移地址对应的内容被存放到y
```

一般来说，`[BX]`就是指 `DS:[BX]`，默认段地址为数据段，当然你也可以指定为CS和SS

CX：Count 计数器

和循环指令`LOOP`有关

LOOP指令类似于C语言中的For循环，`loop NAME`近似于`for(cx;;cx--)` 关于LOOP的用法，具体到程序结构再说好了~。

DX：Data 数据寄存器

特殊功能和MUL/DIV有关，后面再说 也有与输入输出的暂存有关的功能（9.10号指令）

指针变址寄存器：SP，BP，SI，DI

都倾向于用来存地址

SP：Stack Pointer

和堆栈段的使用有关，定义堆栈段要记得手动把SP放在栈顶

BP：Base Pointer

和BX有类似的用法，只是一般更倾向于用在堆栈的数据里，`[BP]`默认为`SS:[BP]`

SI：Source Index

DI：Destination Index

和BX有类似的用法，`[SI]`默认为`DS:[SI]` 如果要转移数据，倾向于用SI存原地址，DI存新地址

段寄存器：CS，DS，SS，ES，IP

段的存在方便我们以段地址+偏移地址的方式定位内存单元 刚刚在例子中看到，一般的程序我们定义三个段，Data、Stack和Code，它们的作用和名字是一致的

这些寄存器都和程序段还有程序的运行有关。 在程序启动的时候，操作系统会把IP（Instruction Pointer）指向程序的第一句开始运行，之后IP会一直指向每次要运行的下一条指令（显然我们可以用IP玩一些花活，但是对于简单的程序，我们没有必要操作IP）

在代码段的开始，我们就用Assume语句声明CS、DS、SS的地址 和CS不同，DS和SS寄存器的值需要我们手动指定，而与SS寄存器绑定的SP指针也需要我们手动设置（SS:SP指向的就是栈顶元素）

ES是Extra Segment，程序有附加段落的时候才用，用法和DS SS差不多

标志寄存器 FLAG

只是写代码的话不用管它 16位分开使用，有各自不同的意思，结果会以下面的形式呈现在-R中

标志位名称	值为 1 时的标记	值为 0 时的标记
OF	OV	NV
SF	NG	PL
ZF	ZR	NZ
PF	PE	PO
CF	CY	NC
DF	DN	UP

阶段练习

有以上的知识已经能写很多代码。

练习a: x+y

1. 在数据段（data segment）中定义3个word，其中x=1234H，y=2345H, z=?
2. 将x+y的结果保存在z中

进阶指令

这里开始会有一点复杂，建议一边写一边看

1. Label和Jump：跳转

一段代码可以拥有label，Jump NAME 即可跳转至label位置 比如

```
MAIN：MOV X,AX
JUMP DONE
MOV AX,Y
...
DONE:
   MOV AH, 4CH
   INT 21
```

在这段程序中，`MOV AX,Y`就会直接被跳过

2. 分支和循环

分支 CMP-JGE/...

Mnemonic	Condition Tested	"Jump IF ..."
JA/JNBE	(CF = 0) and (ZF = 0)	above/not below nor zero
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OR) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNO	OF = 0	not overflow
JNP/JPO	PF = 0	not parity/parity odd
JNS	SF = 0	not sign
JO	OF = 1	overflow
JP/JPE	PF = 1	parity/parity equal
JS	SF = 1	sign

15

BeLike:(求abs(AX)保存在AX中)

```
MAIN:
...
CMP AX,0
JGE DONE; Jump if Greater or Equal
NEG AX
DONE:
...
END MAIN
```

练习b: 求最大值

1. 在数据段（data segment）中定义4个word，其中x=1234H，y=2345H，z=-1234H，w=?
2. 求max(x,y,z)储存于w

循环 LOOP

一种简单的循环，类似于`for(cx;;cx--)`。（事实上，你可以用JUMP和分支结构来实现循环）

LOOP NM过程中： 0. CMP CX,0 1. 如果CX>0，继续执行以下语句，否则跳出 2. DEC CX(CX>0) 3. JUMP NM

```
MOV CX,6
NM: ...
LOOP NM;这样写一共执行CX次（声明NM时执行1次，LOOP中执行CX-1次）
```

练习c: 数组初始化

> 在数据段（data segment）中分配100字节，并为每个字节依次赋值0-99。

3. 堆栈的使用

初始化

两个好用的方法

比较直观的（堆栈段中做定义）

1. 在堆栈段划分位置，保存栈顶位置
2. 在程序段开始的时候把堆栈段的位置告诉堆栈寄存器SS，把栈顶的位置告诉指针寄存器SP

```
SSEG SEGMENT
    STACK DW  128 dup(?)
    TOP   DW LENGTH STACK ;划定范围
SSEG ENDS

CSEG SEGMENT
        ASSUME CS:CSEG,DS:DSEG,SS:SSEG
    MAIN:
        MOV   AX,DSEG
        MOV   DS,AX
        MOV   AX,SSEG
        MOV   SS,AX
        MOV   AX,TOP
        MOV   SP,AX                 ;栈顶地址载入
```

稍微没那么直观的（程序段中划空间）

直接给SP赋值

```
SSEG SEGMENT
SSEG ENDS
;ss:0000-ss:1000
CSEG SEGMENT
        ASSUME CS:CSEG, DS:DSEG,SS:SSEG
    BEGIN:MOV   AX,DSEG
        MOV   DS,AX
        MOV   AX,SSEG
        MOV   SS,AX
        MOV   SP,1000H                ;手动规定了1000H的空置空间(OFFSET 0H-1000H)
```

PUSH和POP

注意：只能操作寄存器，不能直接操作内存单元 `PUSH AX`：将AX的值入栈（如果AX两个字节，就会入栈两个字节，SP也相应-2） `POP AX`：出栈，内容保存在AX（如果AX两个字节，就会入栈两个字节，SP也相应+2）

用SP和BP操作堆栈

在主程序只是暂存数据用的话，一般不用操作指针 但是，由于PROC需要使用到堆栈段，所以这是操作指针就是必要的，接下来在PROC中解释

4. <函数>：PROC和MACRO

PROC&CALL（子程序结构）

定义 PROC-RET-ENDP

(Near 属性是默认值)

```
MAIN:
CALL NM

NM PROC
...
RET
NM ENDP
...
END MAIN
```

完整的表达式： 调用：`CALL FAR/NEAR PTR NM` 定义：`NM PROC FAR/NEAR`

子程序属性和调用

段内调用

只需要Main（主Label调用）的话空置即可（默认Near）

```
A:...
   CALL B;调用B

   PROC B:...;默认为near属性子程序
   RET
   B ENDP
...
END A
```

段间调用

```
PROC A:CALL FAR PTR B
RET
ENDP

PROC B FAR:...;写明属性
RET ENDP
```

小心堆栈！

PROC的本质是：入栈程序出口指针，RET时从回到出口指针的位置 所以：

1. 第一个出栈元素会是一个偏移地址
2. 如果最后SP的指针位置不对，就无法正确RET

简单的方法：用寄存器BP保护SP，使用BP进行数据的读取

例子a(1): x+y子程序化

> ...
>
> 1. 在堆栈段push任意两个长度为1word的数据
> 2. 使用子程序，将这两个数据的和存储于AX ...

```
SUM PROC ;取两个栈顶元素求和储存到AX中
        MOV    BP,SP
        MOV    AX,[BP+2]
        ADD    AX,[BP+4]
        RET
SUM ENDP
```

练习b(1) 求最大值 子程序化

> 1. 在数据段（data segment）中定义4个word，其中x=1234H，y=2345H，z=-1234H，w=?
> 2. 在堆栈段push x,y,z
> 3. 求max(x,y,z)储存于w

*Macro（宏定义）

PROC的使用有调用开销（程序的中断 跳转 继续），而MACRO没有 MACRO相当于写代码的人把重复写代码的过程交给了汇编器，相比子程序来说，是通过多占程序的内存来提高运行速度（对机器来说，每调用一次Macro，就是把这段指令重复了一次）

```
NM MACRO R1,R2...(参数)
...
END M

NM MACRO AX,BX...(寄存器取值)
```

5. INT 21H指令：输入/输出

其实查书就可以了

到这里汇编语言编程的<大局>已经描述完全

关于一些语句的细节可以通过搜索引擎和汇编相关的任何书籍进行确认~

键盘输入

1号指令：单个字符输入

```
MOV AH,1
INT 21H
```

(内容会保存在AL)

10号指令：从键盘输入字符串

内存里需要划分三个部分： 1.一个字节存放最大长度（你写，溢出会被裁掉） 2.一个字节存放实际长度（指令运行完CPU会写） 3.一些字节用来存字符串

```
DATA SEGMENT
   MAXLENGTH    DB 100          ;一个字节，用它存最大的长度
   ACTUALLENGTH DB ?            ;一个字节，用它存实际的长度，在指令执行后会被填写
   STRING       DB 100 DUP(?)   ;用来存字符串
DATA ENDS

STACK SEGMENT
STACK ENDS

CODE SEGMENT
       ASSUME DS:DATA,SS:STACK,CS:CODE
   MAIN:
       MOV    AX,DATA
       MOV    DS,AX
       MOV    DX,OFFSET MAXLENGTH       ;把需要用到的内存块（三个部分）的地址存入DX

       MOV    AH,10
       INT    21H

       MOV    AH,4CH
       INT    21H
CODE ENDS
END MAIN
```

显示器输出

2号调用：单个字符输出

```
MOV DL,'A'
MOV AH,2
INT 21H
```

9号调用：字符串输出

你的字符串必须要以'$'结尾！不然输出不会结束！（类似于'\0'，'$'是一种字符串的终止符）
程序会将DS:DX地址开始输出字符到'$'结尾

```
MOV DX,OFFSET STRING
MOV AH,9
INT 21H
```

综合练习

练习d. 大小写转换+输入输出

> （分支/循环/子程序+输入输出） 用户输入一个单词，程序将所有大写转换为小写并输出到显示器 (注：'a'='A'+20H)