# NSASM MANUAL

NyaSama Assembly Script Module Manual

Introduction (Java, C#, C++)

## 注意: 操作符不区分大小写

### 代码风格类似于 Intel 8086 宏汇编语言

目前实现了动态类型，反射，多态，闭包等高等特性

---

## 1 代码示例

```
.print<reg> {
    out "reg is "
    prt reg
}

run <main>

<main> {
    mov r0, "test"
    code c = (
        prt r0
        prt r0, r0
        prt r0
    )
    eval r0, c
    print<r0>
    prt c, "prt \"hello, gensokyo\""
    eval r0, c
    print<r0>
    end
}
```

---

## 2 注释格式

```
rem "注释内容"
```

对这是一条指令, 还不能跨行(

---

## 3 变量声明

`var 变量名 = 初始值`

初始值可以为这些格式: `1, 0x1, 1h, 1.0, 1.0F, '1', "1", "1" * 重复次数`

其中重复次数除了可以是立即数, 还可以是其他整数型变量或寄存器

以下为详细说明

| 数据类型 | 对应立即数 | 示例 |
| --- | --- | --- |
| 整数 | 1, 0x1, 1h | var num = 0x32 |
| 浮点数 | 1.0, 1.0F | var f = 2.333 |
| 字符 | 'c', '' | var c = ' |
| 字符串 | "hello", "gensokyo" * 2 | var str = "HO!" * num |

需要注意的是，字符串型声明后本身是只读的，需要传送至一个通用寄存器

同时这个寄存器初始类型不能是字符型或字符串型

### 3.1 特殊变量声明

#### *3.1.1 函数型变量*

NSASM 拥有类似于函数型变量的变量, 即把一段代码作为变量, 可以执行和追加

未来还会加入更多的特性(比如插入, 替换等)

多行代码声明如下, 此时等号右侧小括号范围内依然属于立即数

```
var c = (
    prt "hello, world!"
    prt "hello, gensokyo!"
)
```

单行代码声明如下

```
var c = ( prt "single" )
```

#### *3.1.2 映射型变量*

这种变量和 C++的 map, C#的 Dictionary, Java 的 LinkedHashMap 类似

其实 NSASM 在不同语言上的实现就是用的以上三种数据结构

其声明和函数型变量类似, NSASM 就是基于函数型变量实现的映射型变量初始化

在小括号左侧使用 `M` 或者 `m` 修饰符即为映射型变量立即数形式

括号内使用 put key, value 即存入当前的映射, 当然也可加入循环程序结构等普通程序.

使用 use map 来选中要操作的映射型变量, map 是映射型变量或寄存器

取值使用 get var, key , 其中 var 需要是可写的变量或寄存器

要注意一个问题, 由于 NSASM 的设计, 目标操作数不能是字符型, 字符串型, 函数型和映射型立即数. 因此要将这些立即数用为键的话, 需要使用通用寄存器做次替代. 或者说, 这里的目标操作数仅支持寄存器寻址和数值型立即数寻址.

多行声明如下

```
map m1 = M(
    mov r0, 20
    mov r1, 0
[head]
    inc r1
    mov r2, r1
    mul r2, r1
    put r1, r2
    cmp r1, r0
    jnz [head]
)
```

单行声明如下 (此时声明的是空的映射型变量)

```
var map1 = m()
```

### 3.2 显式变量声明

使用这些关键字: int, char, float, str, code, map

要注意的是这样声明的话需要写出初始值

这样可以作为参数在代码中使用

```
int a = 123
float b = 1.23
char c = 'c'
str s = "CHina"
str t = "t" * 10
code c = (
    int x = 3
    int y = 2
)
map m = M(
    put 0, 'a'
    put 1, 'b'
)
```

## 4 代码段声明

这里的代码段类似于 8086 汇编中的概念, 也类似于一般语言的函数的概念

同时, 函数型变量则类似于 Java 的 Runnable 接口以及 C#的 delegate

函数型变量的立即数则类似于 Java 和 C#中的 Lambda 表达式

有基本格式(其中修饰符可省略)

```
修饰符<代码段名> {
    代码块
    ...
}
```

以下为示例

```
<func> {
    prt "hello"
    prt "this is a sample"
}

.<conf> {
    heap 64
    stack 32
    reg 16
}
```

特别地, 当代码段名左侧三角括号的左侧存在如下修饰符时

| 修饰符 | 作用 |
| --- | --- |
| . (小数点) | 当段名为 conf 时, 为解释器配置段, 否则整个段不被解释器加载 |
| @ | 当解释器中已经存在同名段时, 会覆盖掉已有的段, 否则解释器抛出错误 |
| .[NAME] | 这里的 [NAME] 为合法的标识符, 此时这个段为 宏展开声明 , 此段不被解释器加载 |

### 4.1 当修饰符为 . 小数点时

```
.<conf> {
    heap 64
    stack 32
    reg 16
}
```

heap 为堆大小配置(变量空间), stack 是栈大小配置, reg 是寄存器数目配置

这里仅可跟正的整数型立即数, 因为这部分代码不送入解释器执行

### 4.2 当修饰符为 @ 时

首先声明了如下段

```
<seg> {
    prt "hello"
}
```

然后声明如下段(或存在另外一个文件内加载)

```
@<seg> {
    prt "hello, gensokyo!"
}
```

此时执行 seg 段输出是 hello, gensokyo! 而非 hello

### 4.3 当此段为 宏展开声明 时

有一般结构

```
.marcoName<marcoArgs> {
    ...
}
```

此时三角括号内的 marcoArgs 是宏展开的参数表, 而非段名

左侧的 marcoName 为宏展开段名, 后续调用宏展开使用此名称

需要注意的是, 左三角括号与宏展开段名之间不能有分隔符, 宏展开段名需要是合法的标识符

以下为一个具体示例

```
rem "宏展开声明"
.printInfo<arg1, arg2, arg3> {
    out "arg1 is "
    prt arg1
    out "arg2 is "
    prt arg2
    out "arg3 is "
    prt arg3
}

rem "宏展开调用"
printInfo<"hello", 'a', r0>
```

程序运行结果是

```
arg1 is hello
arg2 is a
arg3 is 0
```

这里的宏展开处理是在程序运行之前的预处理阶段, 而非实时的调用, 这和 C/C++里面的宏展开类似

为了避免递归展开, 解释器的宏展开处理仅进行一次, 因此宏展开内的宏展开不会被处理

宏展开段体内调用其它宏展开或者自身是非法的，解释器不会处理这里的宏展开调用

---

## 5 程序标号

这里的标号单独占一行, 使用 [] 包括, 而非传统汇编在指令头部

```
[head]
...
  jnz [head]
[tag]
  prt "..."
...
```

---

## 6 指令格式

指令名 目标操作数，源操作数
指令名 操作数
指令名

需要注意的是, 在绝大部分双操作数指令中目标操作数不能是立即数

而在单操作数指令中, 其操作数要求视不同指令而不同

特别地，函数型变量的立即数仅能作为双操作数指令的源操作数
而在绝大部分双操作数指令中，函数型变量通常作为源操作数，此时其代码会被执行
并返回一个值，在未指定的情况下返回的是最后所使用的目标操作数
若指定则是返回所指定的操作数，如立即数或寄存器

需要注意的是，在函数型变量中的代码里声明的变量不和函数型变量以外的冲突

同时将这些变量作为返回值是返回的拷贝，原变量的对象在函数型变量执行完后就销毁了

因为函数型变量里的代码其实是在一个新的解释器实例下运行，这个实例继承了来自父实例的寄存器组的拷贝

而子实例的堆栈大小和父实例相同，但并未获取父实例的拷贝，因此子实例的变量声明不和父实例冲突

## 7 寄存器组

寄存器是 ARM 的风格, 并非是 8086 的 AX, BX 之类, 而是 r0, r1

这寄存器更像是静态变量, 而非硬件意义上的寄存器

以下为介绍

| 寄存器名 | 作用 |
| --- | --- |
| r# | 通用寄存器组, 其中#为编号, r 不区分大小写, 可直接在程序内使用 |
| 映射寄存器 | 位于通用寄存器组的最后, 默认是 r16 , 建议由指令进行修改访问 |
| 状态寄存器 | 用于程序流程控制, 不可在程序内直接使用, 需要由指令进行修改访问 |
| 段寄存器 | 用于程序流程控制, 不可在程序内直接使用, 需要由指令进行修改访问 |
| 程序计数器 | 用于程序流程控制, 不可在程序内直接使用, 需要由指令进行修改访问 |

## 8 指令一览

| 指令名 | 作用 | 示例 | 备注 |
| --- | --- | --- | --- |
| rem | 行注释 | rem “注释” | 不作为实际代码运行 |
| var | 变量声明 | var a = 0 | 不建议夹杂在普通指令内 |
| int | 整数型变量声明 | int a = 0 | 不建议夹杂在普通指令内 |
| char | 字符型变量声明 | char c = ‘c’ | 不建议夹杂在普通指令内 |
| float | 浮点型变量声明 | float f = 0.1 | 不建议夹杂在普通指令内 |
| str | 字符串型常量声明 | str s = “string” | 不建议夹杂在普通指令内 |
| code | 函数型变量声明 | code c = ( ret 0 ) | 不建议夹杂在普通指令内 |
| map | 映射型变量声明 | map m = M( put 0, 1 ) | 不建议夹杂在普通指令内 |
| mov | 数据传送指令 | mov r0, r1 | 存在重载指令 |
| push | 压栈指令 | push r0 | - |
| pop | 出栈指令 | pop r1 | - |
| in | IO 输入指令 | in 0x00, r0 | 存在重载指令 |

| | | | |
|---|---|---|---|
| out | IO 输出指令 | out “12345” | 存在重载指令 |
| prt | 屏幕打印指令 | prt “hello” | 存在重载指令 |
| add | 算数加法指令 | add r0, 2 | 源操作数为函数型变量时会执行代码 |
| inc | 算数加一指令 | inc r0 | - |
| sub | 算数减法指令 | sub r2, 1.3 | 源操作数为函数型变量时会执行代码 |
| dec | 算数减一指令 | dec r1 | - |
| mul | 算数乘法指令 | mul r1, 2 | 源操作数为函数型变量时会执行代码 |
| div | 算数除法指令 | div r2, 1.5F | 源操作数为函数型变量时会执行代码 |
| and | 逻辑与指令 | and r0, 0x7F | 源操作数为函数型变量时会执行代码 |
| or | 逻辑或指令 | or r0, 0x40 | 源操作数为函数型变量时会执行代码 |
| xor | 逻辑异或指令 | xor r0, r0 | 源操作数为函数型变量时会执行代码 |
| not | 逻辑非指令 | not r12 | - |
| shl | 逻辑左移指令 | shl r0, 4 | 源操作数为函数型变量时会执行代码 |
| shr | 逻辑右移指令 | shr r0, 2 | 源操作数为函数型变量时会执行代码 |
| cmp | 比较指令 | cmp r0, 0 | 源操作数为函数型变量时会执行代码,会修改状态寄存器 |
| test | 测试指令 | test r1 | 会修改状态寄存器 |
| jmp | 无条件转移指令 | jmp [tag] | - |
| jz | 条件转移指令 | jz [tag] | 状态寄存器为零时跳转 |
| jnz | 条件转移指令 | jnz [head] | 状态寄存器非零时跳转 |
| jg | 条件转移指令 | jg [tag] | 状态寄存器大于零时跳转 |
| jl | 条件转移指令 | jl [head] | 状态寄存器小于零时跳转 |
| end | 程序结束指令 | end | 不存在操作的寄存器 |
| ret | 结果返回指令 | ret 0 | 存在重载指令 |
| nop | 空操作指令 | nop | - |
| rst | 复位指令 | rst | 会修改段寄存器和程序计数器 |
| run | 段执行(跳转)指令 | run <seg> | 会修改段寄存器和程序计数器 |
| call | 段调用指令 | call <seg> | 会修改段寄存器和程序计数器 |
| ld | 程序加载指令 | ld “test.ns” | 会修改程序缓存区 |
| eval | 函数型变量执行指令 | eval r0, c | 存在重载指令 |
| use | 映射型变量选 | use map1 | 会修改映射寄存器 |

| | | | |
|---|---|---|---|
| | 中指令 | | |
| put | 映射型变量存入指令 | put 0, 1 | 源操作数为函数型时不会被执行 |
| get | 映射型变量读取指令 | get r0, 0 | 源操作数为函数型变量时会执行代码 |
| cat | 字符串连接指令 | cat r0, “123” | 存在映射型变量重载 |
| dog | 字符串移除指令 | dog r0, “AB” | 存在映射型变量重载 |
| type | 类型测试指令 | type r0, a | 目标操作数会变为字符串类型 |
| len | 长度测试指令 | len r0, str | 存在映射型变量重载 |
| ctn | 字符串查找指令 | ctn str, “abc” | 存在映射型变量重载 |
| equ | 字符串比较指令 | equ r0, str | 相同时状态寄存器为 0 |

这里的指令仅包含原生的 NSASM 指令集, 不含重载指令集

### 8.1 行注释指令

*rem “注释内容”*

这其实是一个合法的指令, 也就是说调用格式可以是 `rem reg` 或者 `rem reg1, reg2`

而指令本身不会修改传入的寄存器, 也不会检查寄存器是不是只读(或者是不是立即数)

### 8.2 变量声明指令

*var 变量名 = 立即数*

这里的变量其实也是一个寄存器类的实例, 其实际的数据类型由立即数确定

`"字符串" * 重复次数` 这种立即数仅在这里有效

*int/char/float/str 变量名 = 立即数*

这里要求立即数须与变量类型匹配, 否则解释器会抛出错误

`str 字符串 = "字符串"` 声明得到的是字符串常量, 需要传送至其他可写寄存器进行修改

*code/map 变量名 = 立即数*

这里的立即数其实比较相似, 映射型变量立即数其实是特殊的函数型立即数

只不过在小括号开头有 M 标记, 括号内部的代码在运行前会先选中默认的映射型变量

代码执行结束后会把这个默认的映射型变量复制到用户声明的变量中

### 8.3 数据传送指令

*mov dst, src*

当 dst 为可写字符型变量, src 为字符串型时, 此指令将字符串的对应指针位置的字符传送到目标变量, 类似于 `dst = src[ptr]`

当 dst 为可写字符串型, src 为字符型变量时, 此指令将源变量传送到字符串的对应指针位置, 类似于 `dst[ptr] = src`

*push reg*

通常意义上的压栈指令

*pop reg*

出栈指令, 这里的 reg 需要是可写变量

*in reg, addr*

当省略 addr 时, 即 `in reg` 等效于 `in reg, 0x00`

这里的 reg 需要是可写变量, 或者是字符串常量(即直接声明的字符串 变量)

addr 可取值为 0x00 或 0xFF, 取值为后者时会在输出流输出调试标记

这条指令是从输入流读取键盘输入, 带回显, 回车结束

输入的数据按立即数处理后传递给 reg

需要注意的是输入的数据应当和 reg 的原有类型匹配, 否则 reg 不会被修改

若 reg 类型为整形, 输入应当是合法的整形立即数

若 reg 类型为浮点形, 输入应当是合法的浮点形立即数

若 reg 类型为字符形, 输入不需要外加引号, 若键入了多个字符则取第一个字符

若 reg 类型为字符串形, 输入不需要外加引号

同时, 如果不键入任何字符就回车结束, reg 将不会被修改

*out addr, reg*

当省略 addr 时, 即 `out reg` 等效于 `out 0x00, reg`

这里的 reg 可以是任何变量, reg 为函数型变量时, 函数型变量将会被执行

addr 可取值为 0x00 或 0xFF, 取值为后者时会在输出流输出调试标记

这条指令是将变量输出到输出流, 不带换行符

reg 为整形, 浮点型, 字符型, 字符串型的情况下按变量实际数据输出

reg 为函数型的情况下, 会先执行这函数型变量, 并输出它的返回值

reg 为映射型的情况下, 输出的是完整的映射关系结构, 如

```
M(
1->2
3->4
hello->world
)
```

*prt dst, src*

当省略 src 时, 此指令类似于 out dst, 将变量输出到输出流, 但是这条指令带换行符

当不省略 src, 且 dst 为可写字符串型或函数型时, 此指令为行编辑指令

若 src 为字符串型, 则按原数据操作, 并按字符串指针所指位置为起始点

src 的数据将连接至 dst 的末尾, 并以换行符分隔, 类似于 dst += ("\n" + src)

若 src 为函数型变量, 同时 dst 为字符串型时, 函数型变量将被执行, 实际用到的是此函数型变量运行的返回值

若 src 为函数型变量, dst 也为函数型时, 函数型变量不会执行, src 的代码将被追加到 dst 后面

若 src 为字符型变量, 同时其值为 '\b' 退格符时, dst 中的最后一行字符串或代码将被删除

### 8.4 算术运算指令

*add/sub dst, src*

dst 需为可写变量, 并且计算过程中的类型转换也以 dst 为准

dst 可以是字符型, 整数型, 浮点型, 字符串型

src 可以是字符型, 整数型, 浮点型, 函数型

若 src 为函数型变量, 函数型变量将被执行, 实际用到的是此函数型变量运行的返回值

特别地, 当 dst 为字符串型, 此时 src 只能是字符型或整数型

这种情况下是在操作 dst 的字符串指针, 若需要复位指针, 只需要重新从原字符串重新获取拷贝

*inc/dec reg*

reg 需为可写变量, 其等效于 add reg, 1 和 sub reg, 1

reg 可以是字符型, 整数型, 浮点型, 字符串型

当 reg 为字符串型时, 对应的是对字符串指针进行操作

*mul/div dst, src*

dst 需为可写变量, 并且计算过程中的类型转换也以 dst 为准

dst 可以是字符型, 整数型, 浮点型

src 可以是字符型, 整数型, 浮点型, 函数型

若 src 为函数型变量, 函数型变量将被执行, 实际用到的是此函数型变量运行的返回值

*mod dst, src*

dst 需为可写变量, 并且计算过程中的类型转换也以 dst 为准

因为取余数操作仅对整数有意义, 因此这里的操作数不能是浮点型

dst 可以是字符型, 整数型

src 可以是字符型, 整数型

若 src 为函数型变量, 函数型变量将被执行, 实际用到的是此函数型变量运行的返回值

## 8.5 逻辑/位运算指令

*and/or/xor dst, src*

dst 需为可写变量, 并且计算过程中的数据宽度也以 dst 为准

因为逻辑运算操作仅对整数有意义, 因此这里的操作数不能是浮点型

dst 可以是字符型, 整数型

src 可以是字符型, 整数型

若 src 为函数型变量, 函数型变量将被执行, 实际用到的是此函数型变量运行的返回值

*not reg*

reg 需为可写变量, 并且计算过程中的数据宽度也以 reg 为准

reg 可以是字符型, 整数型, 并根据具体类型按位取反

*shl/shr reg, val*

reg 需为可写变量, 并且计算过程中的数据宽度也以 reg 为准

reg 可以是字符型, 整数型, 为欲进行位移操作的数据

val 可以是字符型, 整数型, 为位移操作的位数

若 val 为函数型变量, 函数型变量将被执行, 实际用到的是此函数型变量运行的返回值

这里的位移操作会将位移后的空位补零, 并非循环位移操作

### 8.6 程序流程控制指令

*cmp reg, val*

reg 需为数值型变量, 即字符型, 整数型, 浮点型三者之一

val 也需为数值型变量, 或者为函数型变量

若 val 为函数型变量, 函数型变量将被执行, 实际用到的是此函数型变量运行的返回值

此指令等效于以下伪代码

```
mov stateReg, reg
sub stateReg, val
```

其中 stateReg 为无法直接访问的状态寄存器

*test reg*

reg 需为数值型变量, 或者为函数型变量

若 reg 为函数型变量, 函数型变量将被执行, 实际用到的是此函数型变量运行的返回值

此指令等效于 cmp reg, 0, 或等效于以下代码

```
mov stateReg, reg
sub stateReg, 0
```

*jmp tag*

无条件跳转到 `tag` 标号处, 此处仅可使用 `[TAG]` 而非其他寄存器或变量

*jz/jnz/jg/jl tag*

`jz` 和 `jnz` 为零跳转和非零跳转, `jg` 和 `jl` 为大于零和小于零跳转

这里的状态由状态寄存器确定, 所有的比较按带符号浮点数值进行, 即精度最高的类型

*end*

程序终止指令, 无操作数

在函数型变量中执行为终止其单独的解释器实例, 并非终止父解释器

*ret reg*

函数型变量内代码返回指令, `reg` 为想要返回的值

此指令执行后函数型变量的解释器实例便被终止, 并返回 `reg` 的拷贝

父解释器执行此指令等效于 `end`, 返回值并无意义

若 `reg` 被省略, 则返回的值来自于函数型变量解释器的 `r0` 寄存器

*nop*

空指令, 无操作数, 一般由扩展指令集进行覆盖

*rst*

复位指令, 无操作数, 指令执行后将复位解释器的段寄存器和程序计数器

一般也由扩展指令集进行覆盖, 进而添加硬件级复位操作

*run seg*

程序段执行指令, `seg` 须为合法的段名 `<segName>`

程序段执行结束后不返回原调用处

*call seg*

程序段调用指令, `seg` 须为合法的段名 `<segName>`

程序段执行结束后返回原调用处

*ld reg*

程序加载指令, `reg` 须为字符串型或函数型变量

若 reg 为函数型变量, 函数型变量将被执行, 实际用到的是此函数型变量运行的返回值

此指令执行后将读取 reg 中所存储的路径字符串指向的代码文件, 预处理并加载代码到当前的解释器

reg 所存储的路径可以是相对于解释器的相对路径, 也可以是绝对路径

*eval dst, src*

函数型变量执行指令, 函数型变量中的代码会被置入独立的解释器实例执行

当 src 被省略时, dst 须是函数型变量, 函数型变量可能的返回值将被丢弃

当 src 不省略时, dst 须是可写的变量, src 须为函数型变量, 函数型变量可能的返回值将被拷贝到 dst

### 8.7 映射型变量操作指令

*use reg*

映射型变量选中指令, reg 须是映射型变量

后续的指令将对此选中的变量进行操作, 等效于 `mov useReg, reg`

*put key, val*

映射型变量置入指令, key 为键, val 为值

若 key 为函数型变量, 函数型变量将被执行, 实际用到的是此函数型变量运行的返回值

因此映射型变量的键不能是函数型变量

若 val 为函数型变量, 函数型变量 不会被执行, 此变量会原样存入映射型变量中

此处的键和值存入的都是原值的拷贝, 而非引用

若置入的映射对中, 键和映射型变量中已有的重复, 则此指令会覆盖原有的值

*get reg, key*

映射型变量取值指令, key 为键, reg 须为可写的变量

若 key 为函数型变量, 函数型变量将被执行, 实际用到的是此函数型变量运行的返回值

因此映射型变量的键不能是函数型变量

若 key 并不存在于已有的函数型变量中, 此指令会抛出错误

此处的值取出的是映射型变量中的拷贝, 而非引用

### 8.8 其他特殊指令

*cat dst, src*

集合求和指令, 用于字符串型变量的连接和映射型变量的组合

`dst` 须是可写的变量, 二者都须同时是字符串型或映射型

当二者为字符串型时, `src` 存储的字符串将连接至 `dst` 存储的字符串的尾部

当二者为映射型时, `src` 存储的映射对将添加至 `dst` 内部, 有相同键的映射对会被覆盖

*dog dst, src*

集合求差指令, 用于字符串型变量中匹配子串的删除和映射型变量中匹配映射对的删除

`dst` 须是可写的变量, 二者都须同时是字符串型或映射型

当二者为字符串型时, 在 `dst` 中与 `src` 匹配的子串将被删除

当二者为映射型时, 在 `dst` 中与 `src` 中有相同键的映射对将被删除

*type dst, src*

变量类型获取指令, `dst` 须是可写的变量, 指令执行后 `dst` 将变成只读的字符串型

`dst` 的内容取决于 `src` 的变量类型, 如 `int/char/float/str/code/map`

*len dst, src*

集合大小检测指令, 用于检测字符串长度和映射型变量中映射对的数目

`dst` 须是可写的变量, `src` 可被省略

若 `src` 被省略时, 须调用 `use` 指令以选中要操作的映射型变量, `dst` 为对应的映射对数目

若 `src` 未被省略, `src` 则须是字符串型, `dst` 为 `src` 中存储的字符串长度

*ctn dst, src*

集合元素存在指令, 用于检测字符串中是否存在给定子串以及映射型变量中是否存在给定键

若 `src` 被省略时, 须调用 `use` 指令以选中要操作的映射型变量, `dst` 为欲查询的键

若 `src` 未被省略, `src` 则须是字符串型, 且为欲查找的子串, `dst` 为 原字符串

若查询的元素存在, 状态寄存器 `stateReg` 将被置 `1`, 否则置 `0`

*equ dst, src*

字符串比较指令, 两个操作数都须为字符串型

若两个操作数存储的字符串相匹配, 状态寄存器 `stateReg` 将被置 `0`, 否则置 `1`

---

## 9 运行原理

解释器加载程序后, 会将段外的代码集中到一起, 作为公共段加载, 并优先运行

公共段程序运行完成后且未终止解释器, 解释器会随机运行各个段的程序

需要指出的是， Java 和 C#版是按文件顺序执行， 而 C++是随机的， 因此不建议使用这一特性

程序是按文件内的顺序逐行执行, 报错会给出段名和行号

运行过程中若用程序加载指令加载新的程序, 并且新程序中有同名的带有覆盖修饰符的段时

原段会被新段覆盖, 加载指令之后对这个段的调用是调用的新段的程序

---

## 10 高阶用法

这里有部分代码参考

https://github.com/NSDN/NyaSamaRailway/blob/master/src/main/java/club/nsdn/nyasamarailway/Util/NSASM.java

---

## 11 其他

目前不建议在函数型变量内进行段声明, 这里会有 bug, 等待后续修复

---