nasm基本语法

一个单文件的汇编示例

```
;Section to store uninitialized variables
section .data
string: db 'Hello World', 0Ah
length: equ $-string

section .bss
var: resb 1

section .text
global _start:
_start:
mov eax, 4
mov ebx, 1
mov ecx, string
mov edx, length
int 80h

;System Call to exit
mov eax, 1
mov ebx, 0
int 80h
```

Sections in NASM

- *Section .text:* This is the part of a NASM Program which contains the executable code. It is the place from where the execution starts in NASM program, analogous to the main( ) function in CProgramming.

- *section .bss :* This is the part of program used to declare variables without initialization

- *section .data* : This is the part of program used to declare and initialize the variables in the program.

编译

- 生成elf文件
  - nasm –felf32 my_print.asm
- 链接
  - ld my_print.o
- 运行
  - ./my_print.out

注意平台差异

- 32位和64位
- Linux和macOS

```
print:
  pusha
  mov rax, 0x2000004          ; syscall需要用到的参数，表示write
  mov rdi, 1                  ; 表示stdout
  syscall                     ; 一直打印，直到遇到空字符为止
  popa
  ret

read:
  pusha
  mov rax, 0x2000003          ; syscall需要用到的参数，表示read
  mov rdi, 0                  ; 表示stdin
  syscall                     ; 读到回车结束，不抛弃，放到字符串末尾
  popa
  ret
```

```
all:
    nasm -f macho64 main.asm
    ld -e _start main.o -macosx_version_min 10.13 -lSystem
```

多文件

%include "./help.asm"

```
global print

print:
        push rbp
        mov rbp, rsp
        call strlen
        mov rdx, rax
        mov rsi, rdi
        mov rdi, 1
        mov rax, 1
        syscall
        leave
        ret
```

example

```
section .data
     var1: db 10
     str1: db “Hello World!..”
section .bss
     var3: resb 1
     var4: resq 1
```

- **RESx** directive is used to reserve just space in memory for a variable without giving any initial values.
- **Dx** directive is used for declaring space in the memory for any variable and also providing the initial values at that moment

x	Meaning	No: of Bytes
b	BYTE	1
w	WORD	2
d	DOUBLE WORD	4
q	QUAD WORD	8
t	TEN WORD	20

声明变量

- 可能会想每次Dx或者RESx命令只能声明一个变量吗？那不是很麻烦？如果要声明一个字符串呢？每次一个字符？
- 可以这样：
- `var: db 10,5,8,9`
- `string: db “Hello”`
  `string2: db “H”, “e”, “l”, “l”, “o”`
- 上面两种是等价的

访问变量-解引用

- MOV dword[ebx], 1
  INC BYTE[label]
  ADD eax, dword[label]

- []之前可以有的：
- **BYTE, WORD, DWORD, QWORD, TWORD**

可能用到的一些系统调用和小技巧

系统调用

- EXIT SYSTEM CALL
- mov eax, 1 ;System Call Number
  mov ebx, 0 ;Parameter
  int 80h ;Triggering OS Interrupt

- 注意将系统调用号放在eax寄存器里，参数放在其他通用寄存器里，然后使用int指令。

Read System Call

- ```
  mov eax, 3
  mov ebx, 0
  mov ecx, var
  mov edx, dword[size]
  int 80h
  ```
  ```

Write System Call

- mov eax, 4
  mov ebx, 1
  mov ecx, msg1
  mov edx, size1
  int 80h

预处理指令

- %define SIZE 100
- %ifdef DEBUG
- 可以用来输出一些调试信息
- 配合Makefile可以生成不同版本

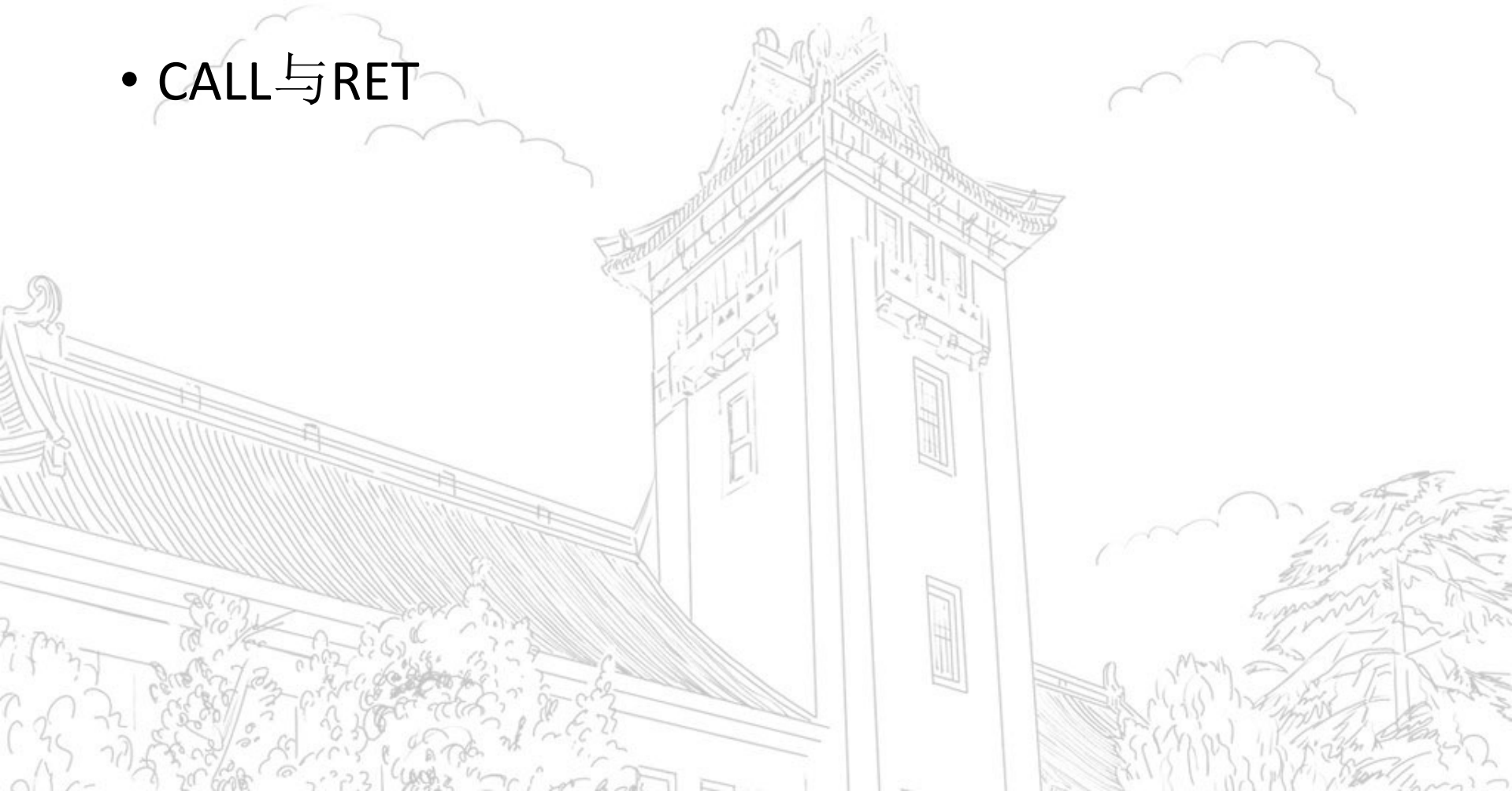
```
debug:
    nasm -f macho64 main.asm -DDEBUG
    ld -e _start main.o -macosx_version_min 10.13 -lSystem
```

```
%ifdef DEBUG
  ; 打印提示信息
  mov rsi, out_string
  mov rdx, out_string_lenth
  call print

  ; 打印输入的字符
  mov rsi, in_char
  mov rdx, input_buffer_length
  call print
%endif
```

函数

- CALL与RET



```
SECTION .data
msg     db      'Hello, brave new world!', 0Ah

SECTION .text
global  _start

_start:
    mov     eax, msg
    call    strlen

    mov     edx, eax
    mov     ecx, msg
    mov     ebx, 1
    mov     eax, 4
    int     80h

    mov     ebx, 0
    mov     eax, 1
    int     80h
```

```
;; strlen(str: eax)->len: eax
strlen:
    push    ebx
    mov     ebx, eax

nextchar:
    cmp     byte [eax], 0
    jz      finished
    inc     eax
    jmp     nextchar

finished:
    sub     eax, ebx
    pop     ebx
    ret
```

宏

- 有些环境可能不支持popa指令
- 可以用宏来自定义一个简单版

```
%macro popa 0

    pop rdx
    pop rcx
    pop rbx
    pop rax

%endmacro
```

注释

- 可读性
- 可以将一些约定写成注释，减轻记忆负担
- 方便检查的时候说明代码含义

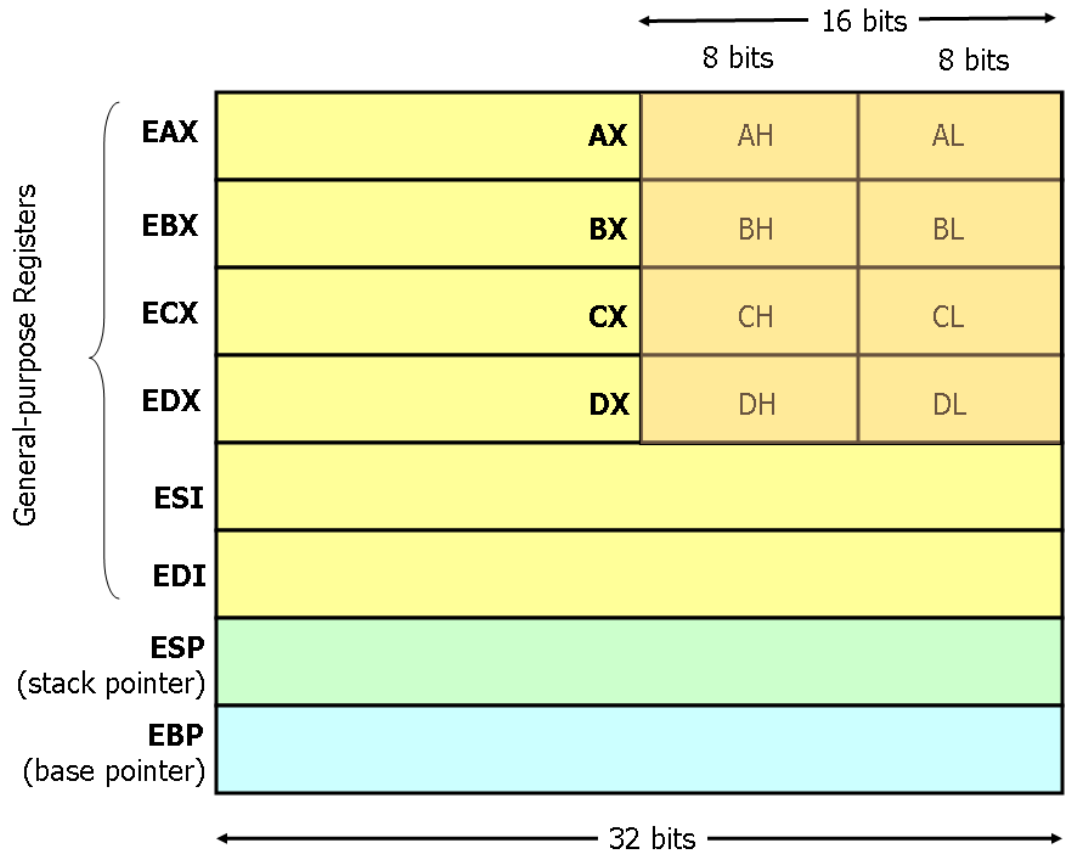
```
reverse:
    ;; 反转一个字符串，存储到指定位置
    ;; rcx存放要存储位置的目的地址
    ;; rbx存放目标字符串的最后一个字符的位置
    ;; rdx存放目标字符串的起始第一个字符位置
```

常用基本指令集简介

- 算术逻辑指令
- 比较类指令
- 跳转指令
- 移位类指令

寄存器长度问题

- eax d
- ax w
- al b
- ah b



MOV

- mov eax, ebx
- mov ecx, 109
- mov al, bl
- mov byte[var1], al
- mov word[var2], 200
- mov eax, dword[var3]

MOVZX(无符号扩展)

- ```
  movzx eax, ah
  movzx cx, al
  ```


  ```

ADD

- ```
  add eax, ecx ; eax = eax + ecx
  add al, ah ; al = al + ah
  add ax, 5
  add edx, 31h
  ```
  ```

SUB

- ```
  sub eax, ecx ; eax = eax - ecx
  sub al, ah ; al = al - ah
  sub ax, 5
  sub edx, 31h
  ```
  ```

MUL

- *mul src*
- 要知道相同位数的两个数字相乘之后位数翻倍

- If src is 1 byte then AX = AL * src
- If src is 1 word (2 bytes) then DX:AX = AX * src (ie. Upper 16 bits of the result will go to DX and the lower 16 bits will go to AX)
- If src is 2 words long(32 bit) then EDX:EAX = EAX * src (ie. Upper 32 bits of the result will go to EDX and the lower 32 bits will go to EAX)

DIV

- div src
- 跟乘法有一点反过来的意思
- If src is 1 byte then AX will be divide by src, remainder will go to AH and quotient will go to AL
- If src is 1 word (2 bytes) then DX:AX will be divide by src, remainder will go to DX and quotient will go to AX
- If src is 2 words long(32 bit) then EDX:EAX will be divide by src, remainder will go to EDX and quotient will go to EAX

条件分支指令

- JMP
- CMP
- …

JMP

- label:

  JMP label

  JMP exit

  ----

  exit:

CMP

- *CMP op1, op2*
- it will affect the CPU FLAGS

Instruction	Working
JZ	Jump If Zero Flag is Set
JNZ	Jump If Zero Flag is Unset
JC	Jump If Carry Flag is Set
JNC	Jump If Carry Flag is Unset
JP	Jump If Parity Flag is Set
JNP	Jump If Parity Flag is Unset
JO	Jump If Overflow Flag is Set
JNO	Jump If Overflow Flag is Unset

i) For Unsigned numbers:

Instruction	**Working**
JE	Jump if op1 == op2
JNE	Jump if op1 ≠ op2
JA (Jump if above)	Jump if op1 > op2
JNA	Jump if op1 <= op2
JB (Jump if below)	Jump if op1 < op2
JNB	Jump if op1 >= op2

ii) For Signed numbers:

Instruction	Working
JE	Jump if op1 == op2
JNE	Jump if op1 ≠ op2
JG (Jump if greater)	Jump if op1 > op2
JNG	Jump if op1 <= op2
JL (Jump if lesser)	Jump if op1 < op2
JNL	Jump if op1 >= op2

条件分支实例

```
cmp eax, ebx
JE  if
    ------
    JMP L1
if:
    INC eax
L1:
    -----------
```

循环

- 循环也没有额外的语法，有了JMP和条件跳转就能组合形成循环。

位运算

- AND
- OR
- XOR
- NOT
- TEST
- SHL
- SHR
- ROL
- ROR
- RCL
- RCR

AND

- *AND op1, op1*
  op1=op1 and op2

- or, xor, 类似

NOT

- *NOT op1*
- *op1=~op1*

TEST

- *TEST op1, op2*

- It performs the bitwise logical AND of op1 and op2 but it won't save the result to any registers. Instead the result of the operation will affect CPU FLAGs.

SHL与SHR

- SHL – Shift Left

- *sy: SHL op1, op2*
  op1 = op1 << op2
- example

  shl eax, 5

op1 should be a reg / memory variable but op2 must be an immediate(constant) value

SHR类似，左边用0补充

ROL与ROR

- 循环左移与循环右移
- *ROL op1, op2*

栈操作

- PUSH
- POP
- PUSHA
- POPA

- PUSHA和POPA用于将所有通用寄存器压栈出栈，当你在函数调用时需要保存现场的时候用会比较方便

PUSH

- PUSH decreases the value of ESP and copies the value of a reg / constant into the system stack

- PUSH ax ;ESP减2
  PUSH eax ;ESP减4
  PUSH ebx
  PUSH dword 5
  PUSH word 258

POP

- POP bx ; ESP= ESP + 2
  POP ebx ; ESP= ESP + 4

- increases the value of ESP

注意事项

- Nasm的不同架构的寄存器和编译方式
- 乘法和除法算术指令会影响多个寄存器
- 编程时注意代码规范，要有注释，函数封装
- 检查作业前要熟悉自己的代码逻辑

- 本PPT旨在介绍一个基本的语法，一些不是必须要用到的语法可能不在讲解之列。课后自行阅读nasm.pdf，参考：https://www.nasm.us/docs.php

谢谢！