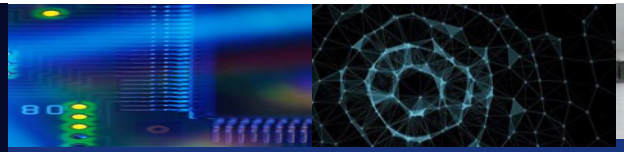
操作系统实验

Operating Systems Principles

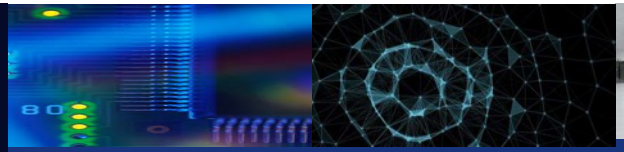
陈鹏飞
计算机学院



实验2——汇编语言练习

目标：

- 熟悉32位 Intel 汇编语言的基本语法；
- 熟悉汇编语言的编译、链接过程；
- 掌握简单编程；
- 掌握磁盘、显示I/O操作；
- 掌握汇编程序调用；
- 掌握C语言与汇编语言之间的调用；



汇编语言编程

汇编语言是机器指令的助记符号；

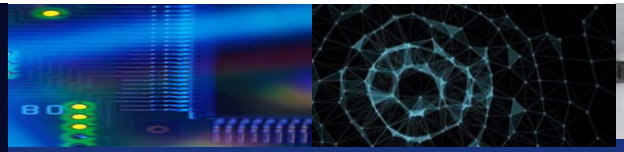
指令具有助记符号（文本形式）；

指令解码产生机器操作码；

汇编指令有助于学习计算机的内部结构；

使用汇编语言的程序具有较小的空间；

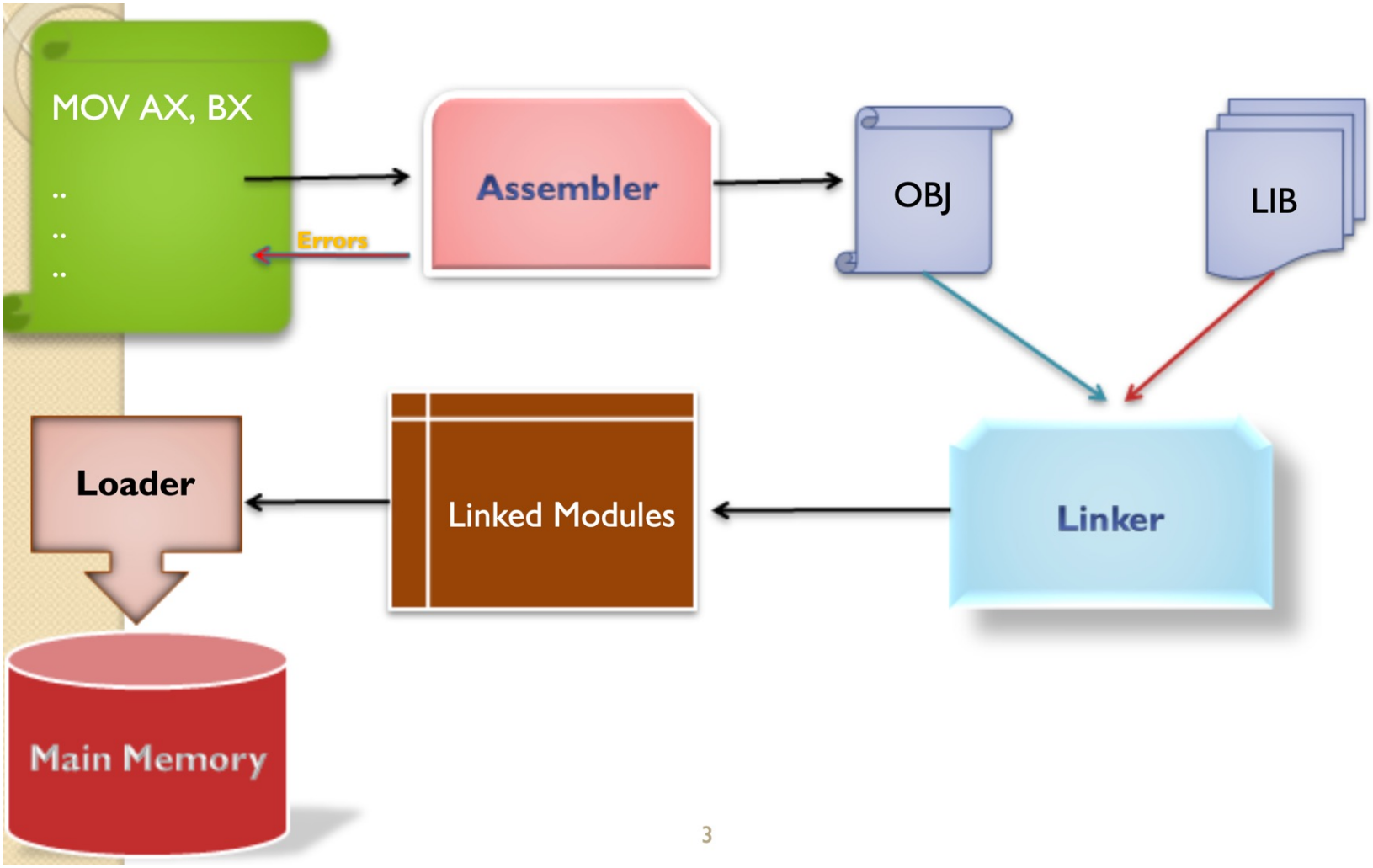
参考：**A Tiny Guide to Programming in 32-bit x86 Assembly Language**



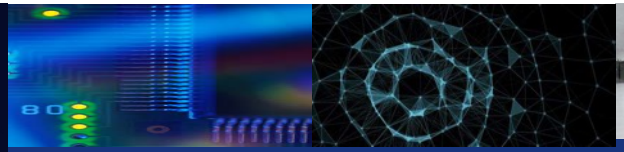
主要的汇编编译器

- MASM(Microsofts micro assembler)
- TASM (Borland Turbo assembler)
- NASM(Netwide assembler)
- 将指令转成二进制代码；
- 产生obj对象文件；
- Obj对象文件包含了指令的二进制代码以及指令的地址；

汇编语言编译过程


MOV AX, BX
..
..
..
Errors
Assembler
OBJ
LIB
Loader
Linked Modules
Linker
Main Memory

3



Linker链接器（Linux ld）

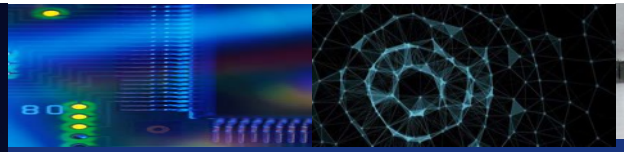
- 用于将多个对象文件合并成一个文件；
- 大型程序被划分成多个小文件；
- 链接文件包含了所有的组合起来的二进制代码；
- 产生可执行文件；

```
Ld  -m  elf_i386 –Ttext 0x1000  -o test   --oformat binary   test.o
```

Linker链接器（Linux ld）

参考： The GNU linker

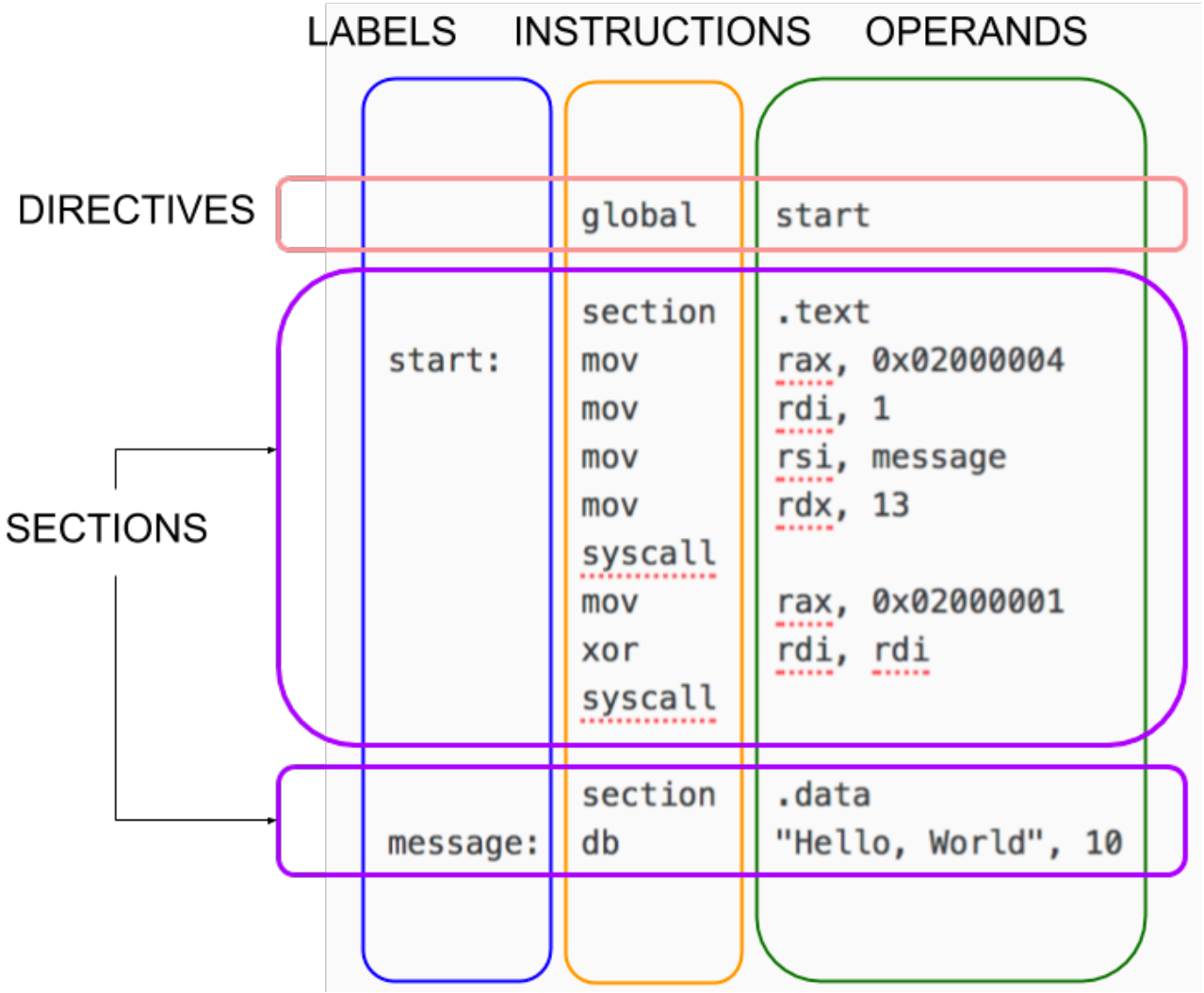
```
/* Simple linker script for the JOS kernel.
   See the GNU ld 'info' manual ("info ld") to learn the syntax. */

OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)

SECTIONS
{
        /* Load the kernel at this address: "." means the current address
        . = 0xF0100000;

        .text : AT(0x100000) {
                *(.text .stub .text.* .gnu.linkonce.t.*)
        }

        PROVIDE(etext = .);     /* Define the 'etext' symbol to this valu

        .rodata : {
                *(.rodata .rodata.* .gnu.linkonce.r.*)
        }

        /* Include debugging information in kernel memory */
        .stab : {
                PROVIDE(__STAB_BEGIN__ = .);
                *(.stab);
                PROVIDE(__STAB_END__ = .);
                BYTE(0)         /* Force the linker to allocate space
                                   for this section */
        }

        .stabstr : {
                PROVIDE(__STABSTR_BEGIN__ = .);
                *(.stabstr);
                PROVIDE(__STABSTR_END__ = .);
                BYTE(0)         /* Force the linker to allocate space
                                   for this section */
        }

        /* Adjust the address for the data segment to the next page */
        . = ALIGN(0x1000);

        /* The data segment */
```



Loader装载器

- 用于将文件装载到内存中；
- 分配特定的地址；
- 将.exe/elf文件转换成.bin文件，并附带物理地址；
- EXE2Bin;

NASM 汇编语言格式


LABELS
INSTRUCTIONS
OPERANDS
DIRECTIVES
SECTIONS
global start
section .text
start: mov rax, 0x02000004
mov rdi, 1
mov rsi, message
mov rdx, 13
syscall
mov rax, 0x02000001
xor rdi, rdi
syscall
section .data
message: db "Hello, World", 10

寄存器

Intel i386处理器含有8个 32位的通用寄存器

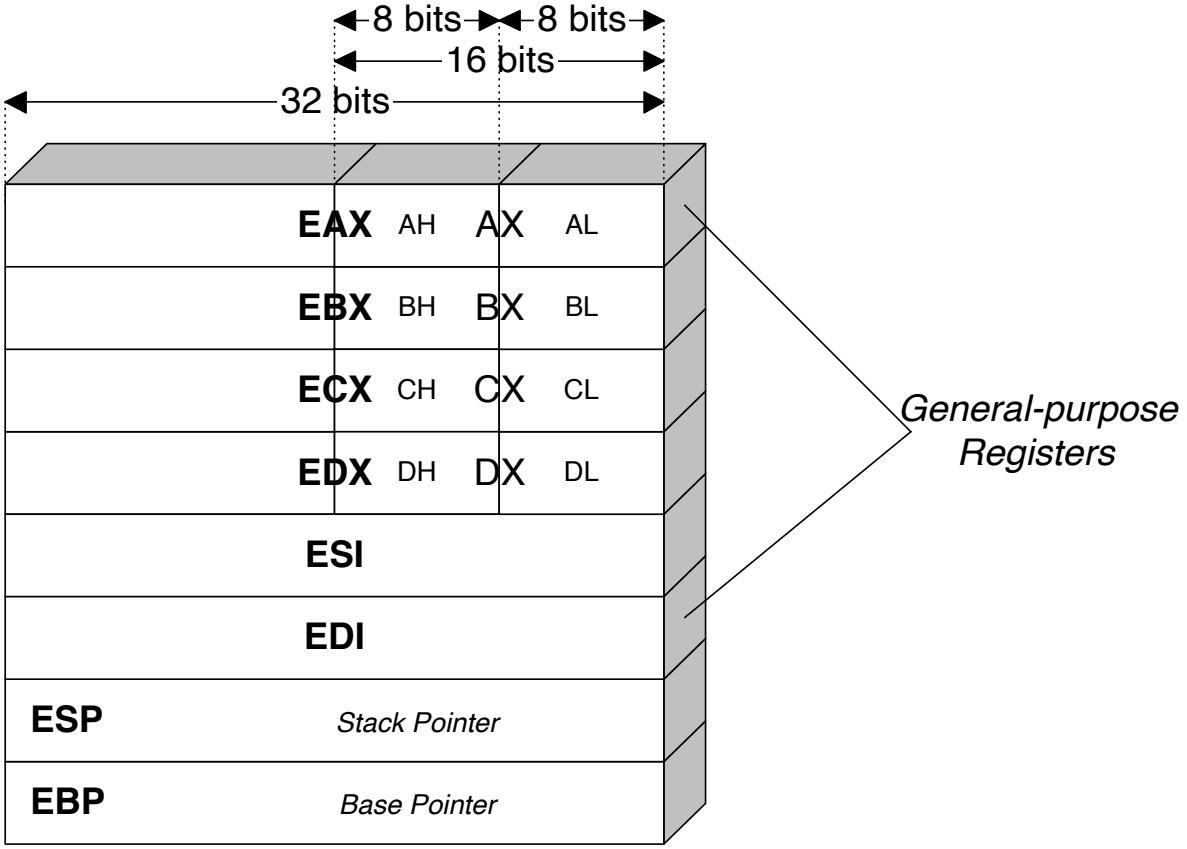


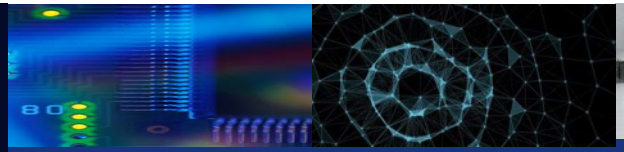
Figure 1. The x86 register set.

汇编指令分类

- Segment Simplification
- Data Allocation
- Segment Related
- Macros Declarations
- Code label
- Scope Declaration
- Listing Control
- Miscellaneous

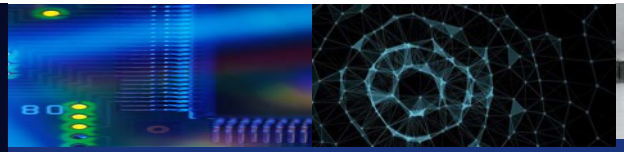
段指令分类

- .CODE :beginning of code segment
  - Ex .CODE [name]
- .DATA : beginning of code segment
- .STACK :used for defining stack
  - Ex .STACK [size]
- .EXIT
- .MODEL :used for selecting standard memory model
  - Ex .MODEL [Memory model]



声明静态数据区

```
.DATA
var     DB 64     ; Declare a byte containing the value 64. Label the
                  ; memory location “var”.
var2    DB ?      ; Declare an uninitialized byte labeled “var2”.
        DB 10     ; Declare an unlabeled byte initialized to 10. This
                  ; byte will reside at the memory address var2+1.
X       DW ?      ; Declare an uninitialized two-byte word labeled “X”.
Y       DD 3000   ; Declare 32 bits of memory starting at address “Y”
                  ; initialized to contain 3000.
Z       DD 1,2,3  ; Declare three 4-byte words of memory starting at
                  ; address “Z”, and initialized to 1, 2, and 3,
                  ; respectively. E.g. 3 will be stored at address Z+8.
```



内存地址

32位指令编码的内存地址空间：2^32， 可以通过寄存器、内存变量寻址

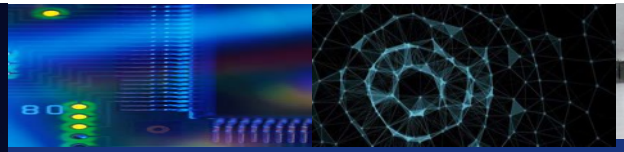
- `mov eax, [ebx]` *; Move the 4 bytes in memory at the address contained in EBX into EAX*
- `mov [var], ebx` *; Move the contents of EBX into the 4 bytes at memory address "var"*
  ; (Note, "var" is a 32-bit constant).
- `mov eax, [esi-4]` *; Move 4 bytes at memory address ESI+(-4) into EAX*
- `mov [esi+eax], cl` *; Move the contents of CL into the byte at address ESI+EAX*
- `mov edx, [esi+4*ebx]` *; Move the 4 bytes of data at address ESI+4*EBX into EDX*

Some examples of incorrect address calculations include:

- `mov eax, [ebx-ecx]` *; Can only **add** register values*
- `mov [eax+esi+edi], ebx` *; At most **2** registers in address computation*

控制操作数据的大小：

- `mov BYTE PTR [ebx], 2` *; Move 2 into the single byte at memory location EBX*
- `mov WORD PTR [ebx], 2` *; Move the 16-bit integer representation of 2 into the 2 bytes starting at*
  ; address EBX
- `mov DWORD PTR [ebx], 2` *; Move the 32-bit integer representation of 2 into the 4 bytes starting at*
  ; address EBX



指令

典型的汇编指令：

Instruction: **mov**

Syntax:
```
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<const>
mov <mem>,<const>
```

Instruction: **push**

Syntax:
```
push <reg32>
push <mem>
push <con32>
```

Instruction: **pop**

Syntax:
```
pop <reg32>
pop <mem>
```

Instruction: **lea**

Syntax:
```
lea <reg32>,<mem>
```

Instruction: **add, sub**

Syntax:
```
add <reg>,<reg>     sub <reg>,<reg>
add <reg>,<mem>     sub <reg>,<mem>
add <mem>,<reg>     sub <mem>,<reg>
add <reg>,<con>     sub <reg>,<con>
add <mem>,<con>     sub <mem>,<con>
```

Instruction: **jmp**

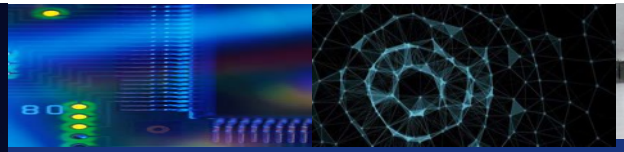
Syntax:
```
jmp <label>
```

Instruction: **cmp**

Syntax:
```
cmp <reg>,<reg>
```

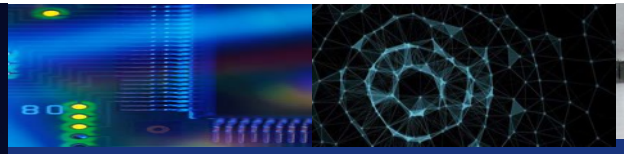
Instruction: **ret**

Syntax:
```
ret
```



内存模型

Model	No. of code segments	No. of data segments
Small	One CS <=64KB	One DS<=64KB
Medium	Any no. and any size	One DS<=64KB
Compact	One CS <=64KB	Any no. and any size
Large	Any no. and any size	Any no. and any size
Huge	Any no. and any size	Any no. and any size

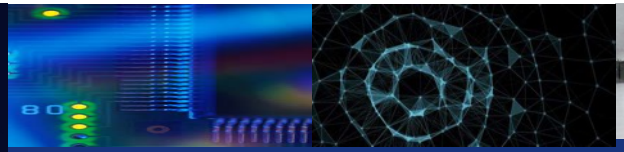


代码作用域

- **PUBLIC**
  - e.g. PUBLIC VAR1,VAR2
  - Var1 and var2 is to be referred from other module
- **EXTRN**
  - e.g. EXTRN SQRT : FAR
  - Labels are in other module

代码作用域

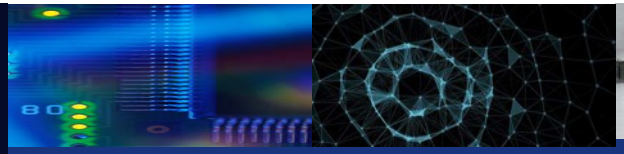
- ALIGN
  - ALIGN number
  - e.g.ALIGN 16
- EVEN
  - e.g. EVEN NAME DB 5 DUP(0)
- LABEL
  - e.g. NEXT LABEL FAR
- PROC
  - procedure-name PROC type



其他指令

- INCLUDE
  - INCLUDE path:file name
- NAME

  Assign name to each assembly module
- GLOBAL
  - GLOBAL variable-name

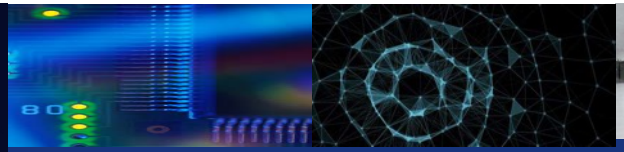


汇编案例

```
1  SECTION .data              ; data section
2  msg:    db "Hello World",10      ; the string to print, 10=cr
3  len:    equ $-msg                ; "$" means "here"
4                                   ; len is a value, not an address
5
6          SECTION .text            ; code section
7          global main              ; make label available to linker
8  main:                            ; standard  gcc  entry point
9
10         mov     edx,len          ; arg3, length of string to print
11         mov     ecx,msg          ; arg2, pointer to string
12         mov     ebx,1            ; arg1, where to write, screen
13         mov     eax,4            ; write sysout command to int 80 hex
14         int     0x80             ; interrupt 80 hex, call kernel
15
16         mov     ebx,0            ; exit code, 0=normal
17         mov     eax,1            ; exit command to kernel
18         int     0x80             ; interrupt 80 hex, call kernel
```

nasm -f elf32 –o test.o test.s;

ld –Ttext 0x10000 –m elf_i386 –o test test.o; (也可以使用gcc进行链接，但是需要以main为程序入口函数)



汇编程序调用

1. At the beginning of the subroutine, the function should push the value of EBP onto the stack, and then copy the value of ESP into EBP using the following instructions:

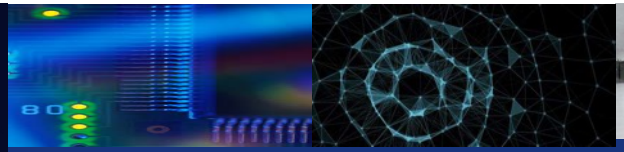
```
push   ebp
mov    ebp, esp
```

2. Next, allocate local variables by making space on the stack. Recall, the stack grows down, so to make space on the top of the stack, the stack pointer should be decremented. The amount by which the stack pointer is decremented depends on the number of local variables needed. For example, if 3 local integers (4 bytes each) were required, the stack pointer would need to be decremented by 12 to make space for these local variables. I.e:

```
sub esp, 12
```

3. Next, the values of any registers that are designated *callee-saved* that will be used by the function must be saved. To save registers, push them onto the stack. The callee-saved registers are EDI and ESI (ESP and EBP will also be preserved by the call convention, but need not be pushed on the stack during this step).

4. When the function is done, the return value for the function should be placed in EAX if it is not already there.



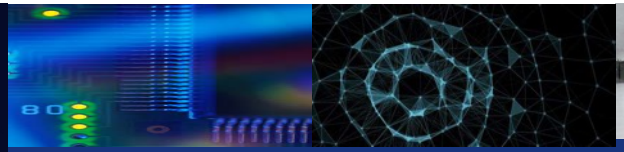
汇编程序调用

5. The function must restore the old values of any callee-saved registers (EDI and ESI) that were modified. The register contents are restored by popping them from the stack. Note, the registers should be popped in the inverse order that they were pushed.

6. Next, we deallocate local variables.

7. Immediately before returning, we must restore the caller's base pointer value by popping EBP off the stack. Remember, the first thing we did on entry to the subroutine was to push the base pointer to save its old value.

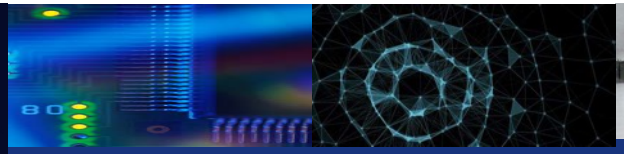
8. Finally, we return to the caller by executing a `ret` instruction. This instruction will find and remove the appropriate return address from the stack.



汇编程序调用

```
; Want to call a function “myFunc” that takes three
; integer parameters. First parameter is in EAX.
; Second parameter is the constant 123. Third
; parameter is in memory location “var”

   push  [var]  ; Push last parameter first
   push  123
   push  eax    ; Push first parameter last

   call _myFunc ; Call the function (assume C naming)

; On return, clean up the stack. We have 12 bytes
; (3 parameters * 4 bytes each) on the stack, and the
; stack grows down. Thus, to get rid of the parameters,
; we can simply add 12 to the stack pointer

   add   esp, 12

; The result produced by “myFunc” is now available for
; use in the register EAX. No other register values
; have changed
```

调用程序



汇编程序调用

```
.486
MODEL FLAT
.CODE
PUBLIC _myFunc
_myFunc PROC
   ; *** Standard subroutine prologue ***
   push ebp     ; Save the old base pointer value.
   mov ebp, esp ; Set the new base pointer value.
   sub esp, 4   ; Make room for one 4-byte local variable.
   push edi     ; Save the values of registers that the function
   push esi     ; will modify. This function uses EDI and ESI.
                ; (no need to save EAX, EBP, or ESP)

   ; *** Subroutine Body ***
   mov eax, [ebp+8] ; Put value of parameter 1 into EAX
   mov esi, [ebp+12]; Put value of parameter 2 into ESI
   mov edi, [ebp+16]; Put value of parameter 3 into EDI

   mov [ebp-4], edi ; Put EDI into the local variable
   add [ebp-4], esi ; Add ESI into the local variable
   add eax, [ebp-4] ; Add the contents of the local variable
                    ; into EAX (final result)

   ; *** Standard subroutine epilogue ***
   pop esi      ; Recover register values
   pop edi
   mov esp, ebp ; Deallocate local variables
   pop ebp      ; Restore the caller's base pointer value
   ret
ENDP _myFunc
END
```

被调用程序

汇编程序调用

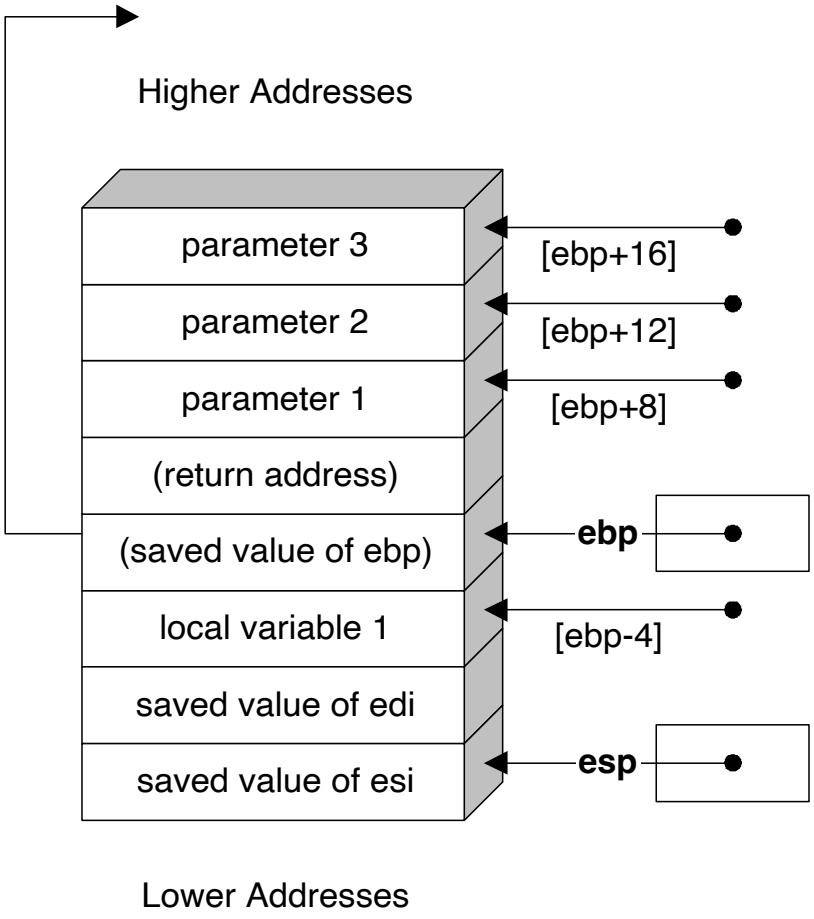


Figure 9. A picture of the stack in memory during the execution of the body of `myFunc`.

汇编系统调用

```
 1 section .data
 2     msg db "hello world!",'\n'
 3
 4 section .text
 5         global  _start
 6
 7 _start:
 8         ;; write syscall
 9         mov     rax, 1
10         ;; file descriptor, standard output
11         mov     rdi, 1
12         ;; message address
13         mov     rsi, msg
14         ;; length of message
15         mov     rdx, 14
16         ;; call write syscall
17         syscall
18
19         ;; exit
20         mov     rax, 60
21         mov     rdi, 0
22
23         syscall
```

汇编系统调用

```
1  [org 0x7c00] ; tell the assembler that our offse
2
3  ; The main routine makes sure the parameters are
4  mov bx, HELLO
5  call print
6
7  call print_nl
8
9  mov bx, GOODBYE
10 call print
11
12 call print_nl
13
14 mov dx, 0x12fe
15 call print_hex
16
17 ; that's it! we can hang now
18 jmp $
19
20 ; remember to include subroutines below the hang
21 %include "boot_sect_print.asm"
22 %include "boot_sect_print_hex.asm"
23
24
25 ; data
26 HELLO:
27     db 'Hello, World', 0
28
29 GOODBYE:
30     db 'Goodbye', 0
31
32 ; padding and magic number
33 times 510-($-$$) db 0
34 dw 0xaa55
```

```
1  print:
2      pusha
3
4  ; keep this in mind:
5  ; while (string[i] != 0) { print string[i]; i++ }
6
7  ; the comparison for string end (null byte)
8  start:
9      mov al, [bx] ; 'bx' is the base address for t
10     cmp al, 0
11     je done
12
13     ; the part where we print with the BIOS help
14     mov ah, 0x0e
15     int 0x10 ; 'al' already contains the char
16
17     ; increment pointer and do next loop
18     add bx, 1
19     jmp start
20
21 done:
22     popa
23     ret
24
25
26
27 print_nl:
28     pusha
29
30     mov ah, 0x0e
31     mov al, 0x0a ; newline char
32     int 0x10
33     mov al, 0x0d ; carriage return
34     int 0x10
35
36     popa
37     ret
```

汇编调用C

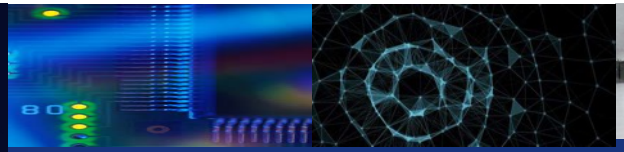
```
1  global _start
2
3  extern print
4
5  section .text
6
7  _start:
8      call print
9
10     mov rax, 60
11     mov rdi, 0
12     syscall
13
14
15
```

```
1  #include<stdio.h>
2
3  extern int print();
4
5  int print() {
6      printf("Hello Wolrd\n");
7      return 0;
8
9  }
```

C也可以调用汇编

C语言嵌入汇编语言

```
 1 #include <string.h>
 2
 3 int main(){
 4
 5     char * str = "Hello World\n";
 6     long len = strlen(str);
 7
 8     int ret = 0;
 9
10     __asm__("movq $1, %%rax \n\t"
11             "movq $1, %%rdi \n\t"
12             "movq %1, %%rsi \n\t"
13             "movl %2, %%edx \n\t"
14             "syscall"
15             : "=g" (ret)
16             : "g" (str),"g" (len));
17
18
19     return 0;
20
21 }
```



项目练习

- 利用32位汇编程序时间从一个数组中找出最大的数（算法不限）；
- 利用32位汇编计算前20个Fibonacci数列；
- 利用汇编清理屏幕的显示内容并定位光标；
- 利用C语言和汇编语言实现磁盘文件的拷贝以及显示，体现C语言与汇编语言之间的调用关系
- 利用不同颜色显式字符串（可选）；