

Lecture 16: 知识 - 高级语言程序与机器级代码之间的对应

内容概述

分析：考试要求

- 只需关注 `x86` 汇编语言；若考察其他汇编语言题目会详细注释
- 题目给出某段简单程序的 `C` 语言、汇编语言、机器语言表示；
- 能结合 `C` 语言看懂汇编语言的关键语句（看懂常见指令、选择结构、循环结构、函数调用）
- 汇编语言、机器语言一一对应，要能结合汇编语言分析机器语言指令的格式、寻址方式
- 不会考：
 - 将 `C` 语言人工翻译为汇编语言或机器语言

x86 汇编语言基础

概念： `x86` 指令

- 指令格式： `操作码` + `地址码`
- 地址码：
 - 数据在寄存器里；
 - 数据在主存里；
 - 数据在指令里；
- 源/目的操作数：
 - 在 `x86` 中，第一个为目的操作数，第二个为源操作数，方向 **从右到左**；

x86 常用寄存器

概念：寄存器

- 概念：
 - 每个寄存器都有自己的名称，可独立存取；
- 图示：

通用寄存器					16bit	32bit	说明
31	16	15	8	7	0		
						AX	EAX 累加器 (Accumulator)
						BX	EBX 基地址寄存器 (Base Register)
						CX	ECX 计数寄存器 (Count Register)
						DX	EDX 数据寄存器 (Data Register)
						ESI	变址寄存器 (Index Register)
						EDI	
						EBP	堆栈基指针 (Base Pointer)
						ESP	堆栈顶指针 (Stack Pointer)

概念：通用寄存器

- 4 个 16 位寄存器又可分割成 8 个独立的 8 位寄存器；
- AX : AH-AL ;
- BX : BH-BL ;
- CX : CH-CL ;
- DX : DH-DL ;

AH&AL=AX(accumulator): 累加寄存器

BH&BL=BX(base): 基址寄存器

CH&CL=CX(count): 计数寄存器

DH&DL=DX(data): 数据寄存器

概念: 指针寄存器

- 寄存器 EBP、ESP 称为指针寄存器；
- 主要用于存放 **堆栈内存存储单元的偏移量**，用它们可实现多种存储器操作数的寻址方式，为以不同的地址形式访问存储单元提供方便；
- 指针寄存器不可分割成 8 位寄存器；
- 作为通用寄存器，也可存储算术逻辑运算的操作数和运算结果；
- 它们主要用于访问堆栈内的存储单元，并且规定：
 - BP 为基指针 (Base Pointer)寄存器，用它可直接存取堆栈中的数据；
 - SP 为堆栈指针 (Stack Pointer)寄存器，用它只可访问栈顶；

SP (Stack Pointer): 堆栈指针寄存器

BP (Base Pointer): 基址指针寄存器

SI (Source Index): 源变址寄存器

DI (Destination Index): 目的变址寄存器

IP (Instruction Pointer): 指令指针寄存器

概念: 标志寄存器

OF -> overflow flag 溢出标志 操作数超出机器能表示的范围表示溢出,溢出时为1

SF -> sign Flag 符号标志 记录运算结果的符号,结果负时为1.

ZF -> zero flag 零标志 运算结果等于0时为1,否则为0.

CF -> carry flag 进位标志 最高有效位产生进位时为1,否则为0.

AF -> auxiliary carry flag 辅助进位标志 运算时,第3位向第4位产生进位时为1,否则为0.

PF -> parity flag 奇偶标志 运算结果操作数位为1的个数为偶数个时为1,否则为0.

DF -> direcion flag 方向标志 用于串处理.DF=1时,每次操作后使SI和DI减小.DF=0时则增大.

IF -> interrupt flag 中断标志 IF=1时,允许CPU响应可屏蔽中断,否则关闭中断.

TF -> trap flag 陷阱标志 用于调试单步操作.

概念: 段寄存器

CS (Code Segment): 代码段寄存器

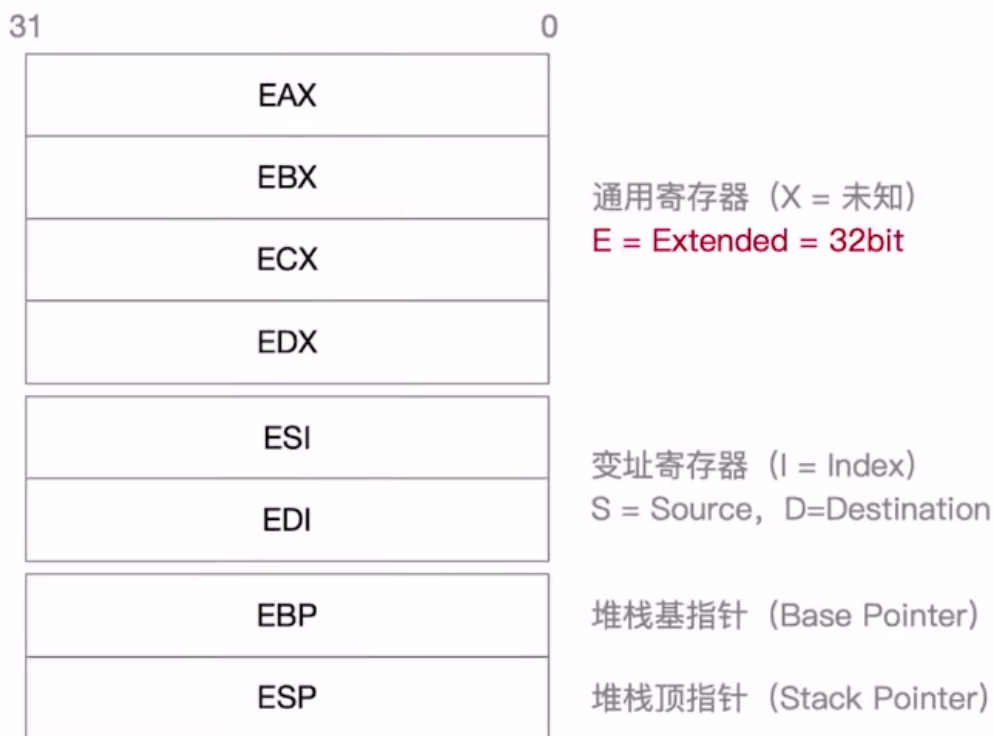
DS (Data Segment): 数据段寄存器

SS (Stack Segment): 堆栈段寄存器

ES (Extra Segment): 附加段寄存器

补充：x86 架构 CPU

- 区别：
 - 在 x86 的其寄存器名称前，会加上一个 E
- 数据寄存器：
 - 以 AX 为例，数据寄存器可以分成：
 - EAX：32bit；
 - AX：16bit；
 - AH/AL：8bit 来独立使用；
- 图示：



AT&T 格式与 Intel 格式

引入

- 基于 x86 格式的汇编代码有 AT&T 格式与 Intel 格式；
- 前面讲的是 Intel CPU 中使用的格式；

总结：AT&T 格式与 Intel 格式

- 图示：

	AT&T 格式	Intel 格式
目的操作数d、源操作数s	op s, d 注：源操作数在左，目的操作数在右	op d, s 注：源操作数在右，目的操作数在左
寄存器的表示	mov %ebx, %eax 注：寄存器名之前必须加%"	mov eax, ebx 注：直接写寄存器名即可
立即数的表示	mov \$985, %eax 注：立即数之前必须加"\$"	mov eax, 985 注：直接写数字即可
主存地址的表示	mov %eax, (af996h) 注：用“小括号”	mov [af996h], eax 注：用“中括号”
读写长度的表示	movb \$5, (af996h) movw \$5, (af996h) movl \$5, (af996h) addb \$4, (af996h) 注：指令后加 b、w、l 分别表示读写长度为 byte、word、dword	mov byte ptr [af996h], 5 mov word ptr [af996h], 5 mov dword ptr [af996h], 5 add byte ptr [af996h], 4 注：在主存地址前说明读写长度 byte、word、dword
主存地址偏移量的表示	movl -8(%ebx), %eax 注：偏移量(基址) movl 4(%ebx, %ecx, 32), %eax 注：偏移量(基址, 变址, 比例因子)	mov eax, [ebx - 8] 注：[基址+偏移量] mov eax, [ebx + ecx*32 + 4] 注：[基址+变址*比例因子+偏移量]

基础指令

mov 指令

分析：mov

- 格式：mov 目的操作数d 源操作数s
- 补充：数据读写长度
 - dword ptr：双字，32bit
 - word ptr：单字，16bit
 - byte ptr：字节，8bit
- 注意：
 - 若未指明主存读写长度，默认 32bit

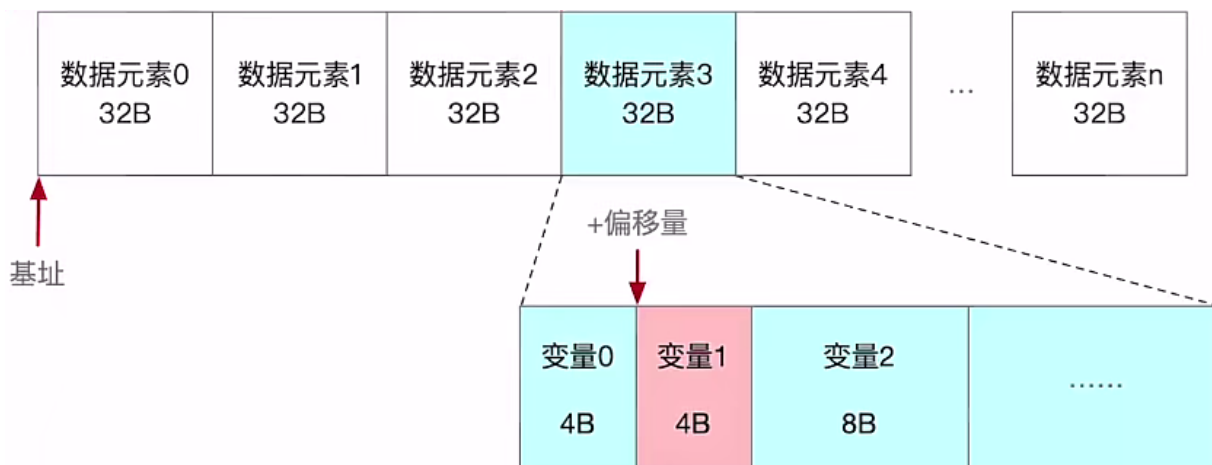
```
# 将寄存器 ebx 的值复制到寄存器 eax
mov eax, ebx
```

```
# 将内存地址 af996h 所指的32bit值复制到寄存器 eax
mov eax, dword ptr [af996h]
```

```
# 寄存器间接寻址：将ebx所指主存地址的32bit复制到eax寄存器中
mov eax, dword ptr [ebx]
```

补充：更复杂的寻址

- mov 指令：
 - mov eax, [ebx+ecx*32+4]
- 图示：



分析：真题：2023年 - mov 指令与数组寻址

- 问题：
 - 描述：给二维数组 `int a[24][64]` 赋值 `0AH`；
 - 指令：`mov [ecx+edx*4+00422000h], 0Ah`
- 分析：
 - 源操作数为 `0AH`，其直接以立即数的形式给出，因此指令中的源操作数采用立即寻址方式；
 - 目的操作数为 `[ecx+edx*4+00422000h]`：
 - 中括号 `[]` 表示对其中的内容进行寻址访问；
 - `00422000h` 是一个地址，在此问题中表示的为数组的基址；
 - `ecx` 的内容未给出，但因为一个 `int` 变量的大小为 `4B`，而一行有 `64` 个，因此用 `64*4B=256B` 可以得到一行的大小为 `256B`，因此 `ecx` 中的内容为 `行号*256B`；
 - `edx*4` 表示列号乘以 `4B`，表示在当前行中的偏移量，因此 `edx` 为 `64`；

算数运算指令

概念：目的、源

- `d`：destination
 - 目的：可以是寄存器、主存，不可以是常量
- `s`：source
 - 源：可以是寄存器、主存、常量
- `x86` 中不允许源和目的的操作数都来自主存；

概念：指令

- 加：`add`
- 减：`sub`
- 乘：
 - 无符号数：`mul d,s`
 - 有符号数：`imul d,s`
- 除：

- `div s` : 无符号数除法 `edx:eax/s` , 商存入 `eax` , 余数存入 `edx`
- `idiv` : 有符号数除法 `edx:eax/s` , 商存入 `eax` , 余数存入 `edx`
- 取负数: `neg d`
- 自增: `inc d`
- 自减: `dec d`

逻辑运算指令

概念: 指令

- 与: `and d,s` $\rightarrow$ 将 `d`、`s` 逐位相与, 结果放回 `d`
- 或: `or d,s`
- 非: `not d`
- 异或: `xor d,s`
- 左移: `shl d,s`
 - 也可用于乘法
- 右移: `shr d,s`

语句实现

跳转指令

概念: 跳转指令

- 概念:
 - 条件转移指令, 依据 `CPU` 状态字中的一系列条件状态转移;
 - `CPU` 状态字中包括指示最后一个算术运算结果是否为 `0` , 运算结果是否为负数等;
- 指令格式:


```
je <label> (jump when equal)
jz <label> (jump when last result was zero)
jne <label> (jump when not equal)
jg <label> (jump when greater than)
jge <label> (jump when greater than or equal to)
jl <label> (jump when less than)
jle <label> (jump when less than or equal to)
```

分析: 代码实现

- 前提:
 - 假设当前 `p1` 和 `p2` 对应的实参已被压入调用函数的栈帧;
 - 它们对应的存储地址分别为 `R[ebp]+8`、`R[ebp]+12` ;
 - 返回结果存放在 `EAX` 中;
- C 语言代码

```
int get_cont(int *p1,int *p2)
{
    if(p1>p2)
    {
```

```

        return *p2;
    }
    else
    {
        return *p1;
    }
}

```

- 汇编代码

```

mov eax,dword ptr [ebp+8]
mov edx,dword ptr [ebp+12]
cmp eax,edx
jbe .L1          # 若p1<p2,则跳转至L1处执行
mov eax,dword ptr [edx]
jmp .L2

.L1
mov eax,dword ptr [eax]

.L2

```

循环语句

概念：循环语句实现 -> 基于跳转指令

- 实现：

- 1. 循环前的初始化;
- 2. 是否跳过循环;
- 3. 循环主体;
- 4. 是否继续循环;

- 图示：

```

int result = 0;
for(int i=1;i<=100;i++) {
    result +=i;
} //求 1+2+3+...+100

```

```

int i=1;
int result = 0;
while(i<=100) {
    result +=i;
    i++;
} //求 1+2+3+...+100

```

```

mov eax,0    #用 eax 保存 result, 初值为0
mov edx,1    #用 edx 保存 i, 初始值为1
cmp edx,100  #比较 i和100
jg L2        #若i>100, 转跳到 L2 执行
L1:          #循环主体
add eax,edx  #实现 result +=i
inc edx      #inc 自增指令, 实现 i++
cmp edx,100  #i和100
jle L1       #若 i<=100, 转跳到 L1 执行
L2:          #跳出循环主体

```

概念：循环语句实现 -> 基于 `loop` 指令

- 实现：
 - `loop = dec ecx;cmp ecx,0;jne L1`
 - `loop` 使用 `ecx` 寄存器作为计数器；
- 补充：
 - `loopnz`：当 `ecx` 不等于零并且 `ZF==0` 时，继续循环；
 - `loopz`：当 `ecx` 不等于零并且 `ZF==1` 时，继续循环；
- 图示：

```
for(int i=500;i>0;i--) {
    做某些处理；
} //循环500轮
```

```
mov ecx,500    #用ecx作为循环计数器
Looptop:       #循环的开始
...
做某些处理
...
loop Looptop   #ecx--, 若ecx!=0, 跳转到Looptop
```

函数调用

概念：函数调用的实现

- 概念：
 - `call <函数名>`：函数调用
 - `ret`：函数返回
 - `IP`：x86 中使用 `IP` 作为指针寄存器；
- 规范：
 - 调用者保存寄存器：
 - `EAX`、`ECX`、`EDX`
 - 被调用者保护寄存器：
 - `EBX`、`ESI`、`EDI`
 - 当外层函数调用内层函数时，内层函数必须将这些内容保存在栈上，并在返回外层函数前恢复它们的值；
 - 栈变化：
 - 每个过程都有自己的栈区，称为栈帧，因此，一个栈由若干栈帧组成；
 - `EBP`：
 - 寄存器 `EBP` 指示**栈帧的起始位置**；
 - `ESP`：
 - 寄存器 `ESP` 指示栈顶，栈从**高地址向低地址增长**；
 - 范围：
 - 过程执行时，`ESP` 会随着数据的入栈而动态变化；
 - `EBP` 固定**不变**；
 - 当前栈帧的范围在 `EBP` 和 `ESP` 指向的区域之间；
- 图示：
 - 实现一：

```

caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125
mov [ebp-8],80
mov eax,[ebp-8]
mov [esp+4],eax
mov eax,[ebp-12]
mov esp,eax
call add
mov [ebp-4],eax
mov eax,[ebp-4]
leave
ret

```

```

add:
push ebp
mov ebp,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax,edx
leave
ret

```

```

int caller(){
    int temp1=125;
    int temp2=80;
    int sum=add(temp1,temp2);
    return sum;
}

```

```

int add(int x, int y){
    return x+y;
}

```



- 实现二：
 - 代码实现：

```

int add(int x, int y){
    return x+y;
}

int caller(){
    int temp1=125;
    int temp2=80;
    int sum=add(temp1,temp2);
    return sum;
}

```

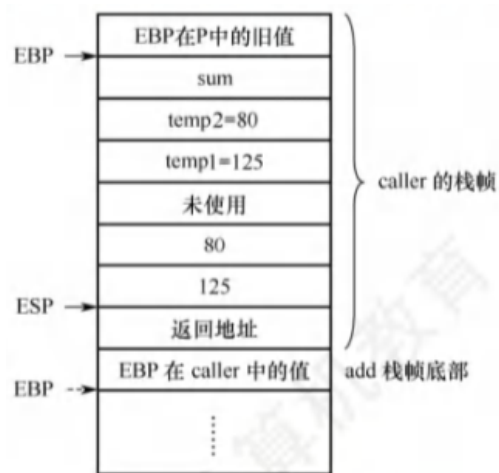
经 GCC 编译后，caller 过程对应的代码如下（#后面的文字是注释）：

```

caller:
push ebp
mov ebp,esp
sub esp,24
mov [ebp-12],125          #M[R[ebp]-12]←125, 即 temp1=125
mov [ebp-8],80            #M[R[ebp]-8]←80, 即 temp2=80
mov eax,dword ptr [ebp-8] #R[eax]←M[R[ebp]-8], 即 R[eax]=temp2
mov [esp+4],eax           #M[R[esp]+4]←R[eax], 即 temp2 入栈
mov eax,dword ptr [ebp-12] #R[eax]←M[R[ebp]-12], 即 R[eax]=temp1
mov [esp],eax             #M[R[esp]]←R[eax], 即 temp1 入栈
call add                  #调用 add, 将返回值保存在 eax 中
mov [ebp-4],eax           #M[R[ebp]-4]←R[eax], 即 add 返回值送 sum
mov eax,dword ptr [ebp-4] #R[eax]←M[R[ebp]-4], 即 sum 作为返回值
leave
ret

```

- 栈帧：



概念： `call` 指令与 `ret` 指令

• 实现： `call`

- 1. 将 `IP` 寄存器的值压入栈 -> 用于 `ret` 时跳回此函数的地址 -> 效果即为 `Push IP`;
- 2. 设置新的 `IP` 值，将 `IP` 指向新的位置 -> 效果相等于 `jmo [目标地址]`;

• 实现： `ret`

- 1. 在函数的栈顶中，找到跳回的函数的 `IP` 地址，将其出栈;
- 2. 将当前 `PC` 指针指向出栈得到的 `IP` 值;

概念： 函数调用的固定代码



概念： 恢复栈帧

```
mov esp, ebp
pop ebp
```

Push 与 Pop

概念： 进程地址空间

• 查看一个进程的区域：

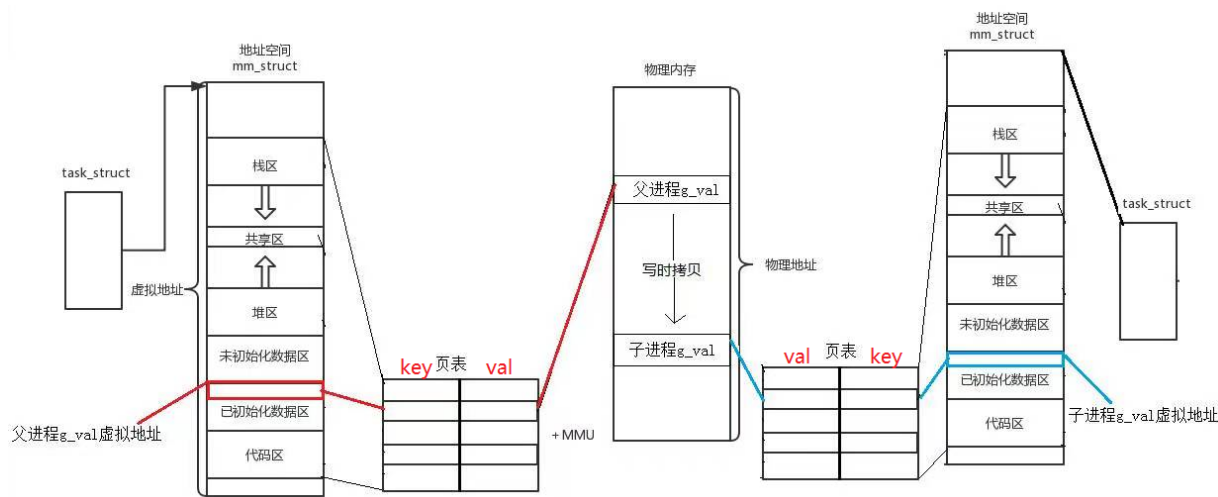
```
37491: /tmp/demo/a.out
0000000000400000      8K r-x-- a.out
00007ffff7ffb000    12K r---- [ anon ]
00007ffff7ffe000     4K r-x-- [ anon ]
00007ffff7ffde00   132K rwx-- [ stack ]
• ffffffff600000     4K --x-- [ anon ]
```

- 更详细的信息:

- 包含栈区、只读代码区、堆区等部分;

```
00400000-00402000 r-xp 00000000 fd:00 525719 /tmp/de
mo/a.out
7ffff7ffb000-7ffff7ffe000 r--p 00000000 00:00 0 [vvar]
7ffff7ffe000-7ffff7fff000 r-xp 00000000 00:00 0 [vdso]
7ffff7fff000-7ffff7fff000 r-xp 00000000 00:00 0 [stack]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsysca
ll]
```

- 完整信息:



概念: 访问栈帧数据 - push、pop 指令

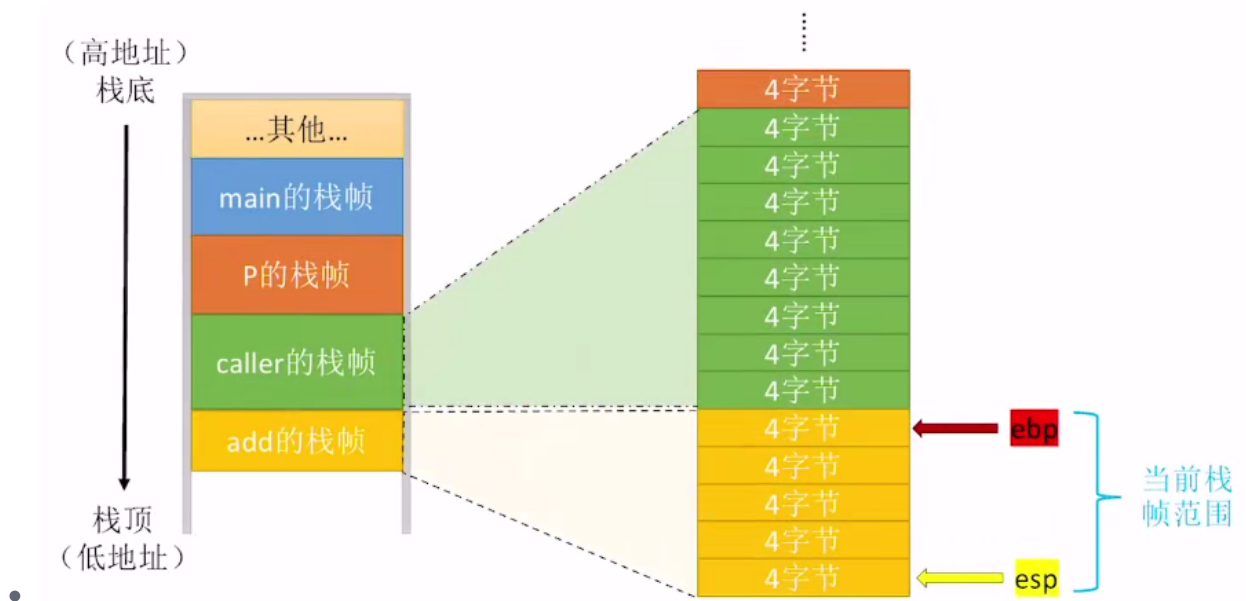
- 概念:

- **ebp**: 指向当前栈的底部;
- **esp**: 指向当前栈的顶部;
- **x86** 系统中, 默认以 **4 字节 (32bit)** 为栈的操作单位
- 图中, 上边是高地址, 下边是低地址;

- 实现:

- **Push <主存 | 寄存器 | 立即数>**: 将元素压入, 可以是立即数、寄存器、主存地址
- **Pop <主存 | 寄存器>**: 将元素弹出, 并将弹出的元素值放入某个寄存器或者主存中, 可以是寄存器, 主存地址;

- 图示:



概念：访问栈帧数据 - `mov` 指令

- 概念：
 - 可以使用 `mov` 指令结合 `esp`、`ebp` 指针访问栈帧数据；
- 注意：
 - 可以用减法/加法指令，即 `sub/add` 修改栈顶指针 `esp` 的值

```
sub esp,12 # 将栈顶向下移动12
mov [esp+8],eax # 将eax寄存器中的数值复制到esp上8个字节的位置
mov eax,[ebp+8] # 将主存中的ebp基址+八个字节的位置的内容，取出放到eax中
mov [esp],eax # eax的值放入esp所指向的主存地址中
add esp,8 # 栈顶指针+8
```

栈帧中包含内容

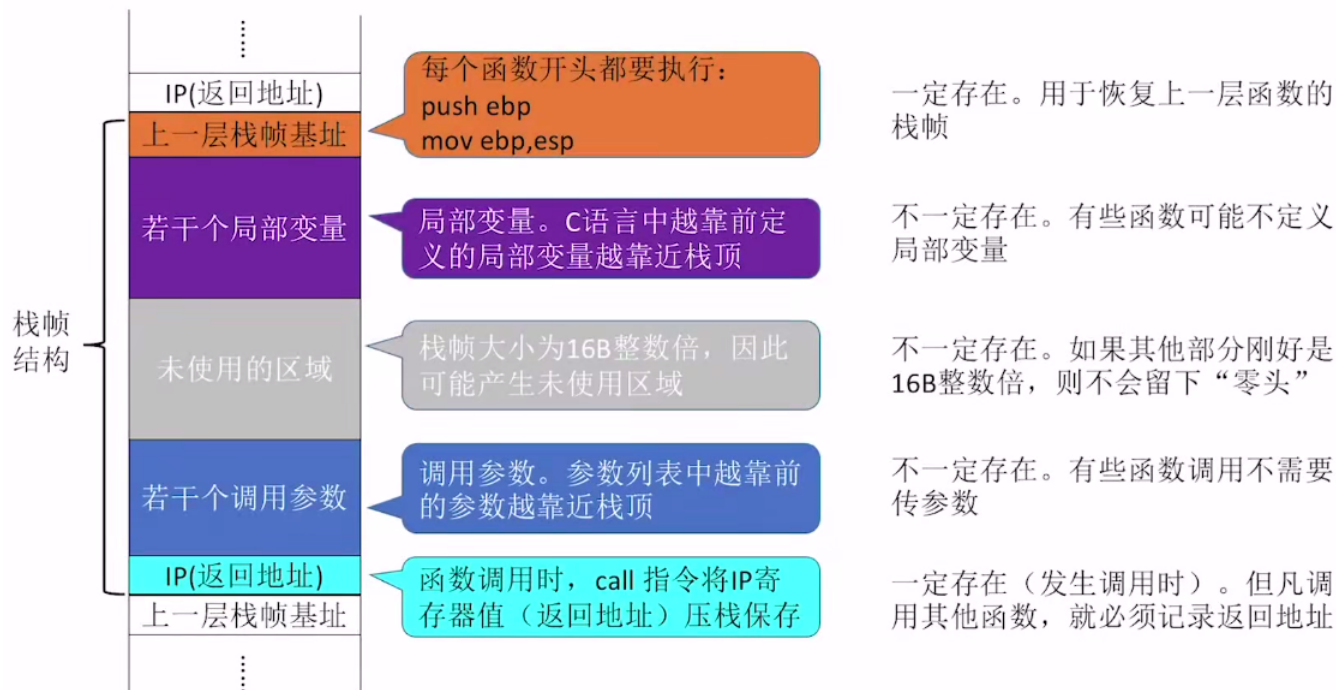
概念：栈帧中包含内容

- 1. 通常将**局部变量**集中存储在栈帧底部区域；
 - C 语言中越靠前定义的局部变量越靠近栈底；
- 2. 通常将调用参数集中存储在栈帧顶部区域；
- 3. 栈帧最底部一定是上一层栈帧基址 (`ebp` 旧值)
 - 调用参数。参数列表中越靠前的参数越靠近栈顶；
 - 比如 `[ebp+b]`
- 4. 栈帧最顶部一定是返回地址 (当前函数的栈帧除外)；

注意：`gcc` 编译器将每个栈帧大小设置为 `16B` 的整数倍(当前函数的栈帧除外)

- 因此栈帧内可能出现空闲未使用的区域；

概念：图示



概念：总结



综合应用

真题：2019

- 代码：

int	f1 (int n) {		
1	00401000	55	push ebp
	...	...	...
	if (n>1)		
11	00401018	83 7D 08 01	cmp dword ptr [ebp + 8], 1
12	0040101C	7E 17	jle f1+35h (00401035)
	return n*f1(n-1);		
13	0040101E	8B 45 08	mov eax, dword ptr [ebp + 8]
14	00401021	83 E8 01	sub eax, 1
15	00401024	50	push eax
16	00401025	E8 D6 FF FF FF	call f1 (00401000)
	...	...	...
19	00401030	0F AF C1	imul eax, ecx
20	00401033	EB 05	jmp f1+3Ah (0040103a)
	else return 1;		
21	00401035	B8 01 00 00 00	mov eax, 1
	}		
	...	...	...
26	00401040	3B EC	cmp ebp, esp
	...	...	...
30	0040104A	C3	ret

• call 指令:

int	f1 (int n) {		
1	00401000	55	push ebp
	...	...	...
	if (n>1)		
11	00401018	83 7D 08 01	cmp dword ptr [ebp + 8], 1
12	0040101C	7E 17	jle f1+35h (00401035)
	return n*f1(n-1);		
13	0040101E	8B 45 08	mov eax, dword ptr [ebp + 8]
14	00401021	83 E8 01	sub eax, 1
15	00401024	50	push eax
16	00401025	E8 D6 FF FF FF	call f1 (00401000)
	...	...	...
19	00401030	0F AF C1	imul eax, ecx
20	00401033	EB 05	jmp f1+3Ah (0040103a)

• 条件转移

```

int f1 (int n) {
1  00401000      55      push ebp
...
if (n>1)
11 00401018      83 7D 08 01 比较  cmp dword ptr [ebp + 8], 1
12 0040101C      7E 17 00 00 小于或等于  jle f1+35h ((00401035))
    return n*f1(n-1);
13 0040101E      8B 45 08      mov eax, dword ptr [ebp + 8]
14 00401021      83 E8 01      sub eax, 1
15 00401024      50           push eax
16 00401025      E8 D6 FF FF  call f1 (00401000) 递归调用f1
...
19 00401030      0F AF C1      imul eax, ecx
20 00401033      EB 05 无条件跳转  jmp f1+3Ah ((0040103a))
    else return 1;
21 00401035      B8 01 00 00 00  mov eax, 1
}
...
26 00401040      3B EC      cmp ebp, esp
...
30 0040104A      C3           ret  返回到本函数调用处

```

2) 左侧代码中，哪条指令是条件转移指令？哪几条指令一定会使程序跳转执行？

解析

第12行的 jle 指令是条件转移指令

第16行的 call 指令是函数调用指令

第20行的 jmp 指令是无条件转移指令

第30行的 ret 指令是函数（或子程序）返回指令

一定会使程序跳转执行

获取笔记持续更新可加微信: WandD221