第 5 次作业

作业文件命名格式： 专业班级_学号_姓名_第几次作业.pdf

例如：CS2201_U202212345_张三_5.pdf

提交到 qq 邮箱：**2112745268@qq.com**

提交截止时间：2024.04.16

1. 编写汇编语言程序段（Intel）去优化下面的 C 函数，使得汇编程序中没有分支转移语句。假设变量 p 的地址为 (ebp) –10h。

```
int getv(int *p)
{
    return p ? *p : 0;
}
```

答案：

```
lea     ebx, [ebp-10h]
cmp     [ebx], 0
mov     eax, 0
cmovnz  eax, [ebx]
```

2. 编写汇编语言程序段（Intel）实现 C 语言的！运算：(eax) = !(ebx)，要求没有分支转移语句。

答案：

```
and     ebx, ebx
mov     eax, 0
mov     ecx, 1
cmovz   eax, ecx
```

3. 假设 (ax) = 0D000h, (bx) = 2000h，执行

```
cmp   ax, bx
ja    L1
```

程序可以转到 L1 吗？将 ja 换成 jg、js，结果如何？

答案：

(1) ja，无符号比较，条件成立，转到 L1；

(2) jg，有符号比较，条件不成立（(ax)是负数, (bx)是正数），不会转到 L1；

(3) js，cmp 实际上做减法（结果不保存），结果的最高位为 1，js 成立，转到 L1；

4. 在分支转移指令的机器码中，如何表示转移的目标地址？

答案：机器码中存贮的是：目标地址– (EIP) 的补码

5. 编写与“loope lp”等效的程序段（Intel）。

答案：

```
lp:     …
        jnz   next
        dec   ecx
        jnz   lp
next:   …
```

6. 根据下列程序段（Intel）

```
        … …
        MOV   ESI, OFFSET BUF1
        MOV   EDI, OFFSET BUF2
        MOV   ECX, n
LOOLA:  MOV   AL, [ESI]
        MOV   [EDI], AL
        INC   ESI
        INC   EDI
        LOOP  LOOPA      ;使用 ECX 计数
        … …
```

绘制流程图，并回答如下(1)至(5)问题。其中 BUF1、BUF2 均为串长度为 n 的字节存储区首址。

(1) 该程序段完成了什么工作？

(2) 若将 “MOV ECX, n”误写成“MOV ECX, 0”，则循环体被执行多少次？

(3) 若漏掉了 “MOV ECX, n”，则循环体执行的次数能确定吗？为什么？

(4) 若漏掉了指令“INC ESI”，则程序运行结果如何？

(5) 若不小心将标号 LOOPA 上移了一行，即将标号写在指令“MOV ECX, n”之前，则程序运行情况如何？

答案：

(1) 将 BUF1 的 n 个字节拷贝到 BUF2 处。

(2) 程序可能会崩溃，如果不崩溃则循环体被执行 10000H 次。
(3) 不能确定，因为（ECX）不能确定。
(4) 将 BUF1 的第 1 个字节拷贝到 BUF2 的 n 个字节。
(5) 程序死循环，可能崩溃。

7. 编写子程序 PROG（不能用带变量说明的高级伪指令去定义 PROG），实现如下功能：(X - Y) / Z（X、Y、Z 是有符号数，不需考虑 X-Y 的溢出），子程序将商保存到 AX、余数保存到 DL。主程序（Intel）的调用格式如下：

```
.model flat, stdcall
… …
X       DW      ?   ;X 为 2 个字节的有符号整数
Y       DB      ?   ;Y 为 1 个字节的有符号整数
Z       DB      ?   ;Z 为 1 个字节的有符号整数
… …
MOV         AL, Y
MOV         AH, Z
PUSH        AX
PUSH        X
CALL        PROG
ADD         ESP, 4
… …
```

答案：

```
PROG        PROC        C
            PUSH        EBP
            MOV         EBP, ESP
            MOV         AX, [EBP+8]                 ;X
            MOVSX       BX, BYTE PTR [EBP+0AH]      ;Y
            SUB         AX, BX                      ;X-Y => (AX)
            CWD                                     ;(AX) => (DX, AX)
            MOVSX       BX, BYTE PTR [EBP+0BH]      ;Z
            IDIV        BX
            POP         EBP
            RET
PROG        ENDP
```

或者

```
PROG        PROC        C
            MOV         AX, [ESP+4]                 ;X
```

```
        MOVSX   BX, BYTE PTR [ESP+6]    ;Y
        SUB     AX, BX                  ;X-Y => (AX)
        CWD                             ;(AX) => (DX, AX)
        MOVSX   BX, BYTE PTR [ESP+7]    ;Z
        IDIV    BX
        POP     EBP
        RET
PROG    ENDP
```

8. 阅读下面的（C 和 Intel 汇编）程序，回答问题

```
void func ( )
{   unsigned short us = 65535;
    unsigned int ui;
    short   s = -1;
    int     i;
    int     x = 0, y = 0;       ; ①
    ui = us;
    i = s;                      ; ②
    if (ui > 0) x = 1;
    if (i > 0) y = 1;           ; ③
    ui = ui >> 1;               ; >>1  右移一个二进制位
    i = i >> 1;                 ; ④
    ui = ~ui;                   ; 按位取反
    i = -i;                     ; ⑤
}
```

⑴ **执行完** ① 处语句后，

us = 0x _FFFF__　　s = 0x _FFFF___（2 个字节的 16 进制数）

⑵ **执行完** ② 处语句后，

ui = 0x _0000FFFF_　　i = 0x _FFFFFFFF_（4 个字节的 16 进制数）

ui = us; 对应的机器指令：　_movzx_　eax, us、 mov ui, eax

i = s;　对应的机器指令：　_movsx_　eax, s、 mov i, eax

⑶ **执行完** ③ 处语句后，

x =　__1__　　y =　__0__

if (ui>0) x=1; 对应的机器指令：　cmp ui, 0、 __jbe__ lp1、 mov x, 1、 lp1: …

if (i >0) y=1; 对应的机器指令：　cmp i, 0、 __jle__ lp2、 mov y, 1、 lp2: …

⑷ **执行完** ④ 处语句后，

ui = 0x _00007FFF __　　i = 0x _FFFFFFFF_ （4 个字节的 16 进制数）

ui = ui >> 1; 对应的机器指令：　__shr__　ui, 1

i = i >> 1;　对应的机器指令：　__sar__　 i, 1

⑸ 执行完 ⑤ 处语句后，

ui = 0x _ffff8000_　　i = 0x __1__ （4 个字节的 16 进制数）

ui = ~ui;　对应的机器指令：　__not__　ui

i = -i;　　对应的机器指令：　__neg__　i

9. 在 VS2019（或其他 VS 版本）、x86 平台、Debug 模式，观察下面 C 程序的反汇编程序、内存等，分析 switch … case 的执行过程，并回答如下问题：

(1) 写出 break 对应的汇编指令；

(2) 若少写 break，导致的后果是什么？

(3) 各个分支摆放顺序对结果有无影响？default 分支能不能写在最开头？

(4) 程序是如何判断并进行相应分支转移的？

```
#include <stdio.h>
int main( )
{int x = 3;
  int y = -1;
  int z;
  char c;
  c = getchar();      //从键盘输入一个字符
  switch (c) {
  case '9':
  case '7':
        z = x + y;
        break;
  case '1':
  case '0':
        z = x - y;
        break;
  default:
        z = 0;
  }
  printf("%d %c %d = %d \n", x, c, y, z);
  return 0;
}
```

答案：

switch … case 的执行过程：编译器分析 case 分支的范围（'0'~'9'），建立一个表格（**case 分支程序地址表**），用于记录每个 case 分支语句体的首地址；同时建立另外一个表（**索引表**），记录 case 所有可能的分支（'0'~'9'共 10 个分支）的处理程序在“**case 分支程序地址表**”的位置信息（即每个分支的处理程序的地址是“**case 分支程序地址表**”的第几个地址，用 1 个字节

表示该索引值），在 case 分支范围（'0'~'9'）中不存在的分支，其索引值为 default 分支的索引值。执行时，cpu 判断 switch 中表达式 c 的值是否超出 case 分支的范围（'0'~'9'）否，若超出则执行 default 语句。若没有超出，则在索引表中查找该分支的索引值，然后根据索引值在“**case 分支程序地址表**”中取出该分支处理程序的地址，并转跳到该地址执行。

(1) break 对应的汇编指令：mov dword ptr [z], 0

(2) 若少写 break，则当输入的字符 c 与 case 语句不匹配时，变量 z 不能赋值。

(3) 各个分支摆放顺序对结果没有影响，default 分支可以放在最开头。

(4) 程序根据 c-'0'的值在索引表中取出索引值，然后根据索引值在“**case 分支程序地址表**”中取出该分支处理程序的地址，并转跳到该地址执行。

10. 在Linux环境下，对一个C语言程序进行编译、链接、调试运行，程序片段如下。

```
int fadd(int a, int b)
{   int temp;
    temp = a + b;
    return temp;
}

void main( )
{   int x = 0x1234;
    int y = -32;
    int result = 0;
    char msg[6] = "abc12";
    result = fadd(x, y);
    result = *(int *)(msg + 1);
}
```

调试时，设变量 x 的地址（&x）为 0xffffd508，y 的地址（&y）为 0xffffd50c，result 的地址为 0xffffd510，数组 msg 的起始地址为 0xffffb516。

(1) 执行到 result = fadd(x, y) 时，写出以字节为写出内存的内容（用 16 进制数的形式填空，最左边是内存地址）。

0xffffd508	34	12	00	00	E0	FF	FF	FF
0xffffd510	00	00	00	00	XX	XX	61	62
0xffffd518	63	31	32	00	XX	XX	XX	XX

(2) 数据传送指令解读

int x = 0x1234 对应的机器指令为：movl $0x1234, -0x20(%ebp)，执行该语句时， ebp = 0x __ffffd528__。

int result = 0 对应的反汇编指令为 __ movl $0, -0x18(%ebp) __。

执行 “result = *(int *)(msg+1);”后， result 中的值为 0x__32316362__ 。

(3) 函数调用语句解读

语句 result = fadd(x, y) 对应的反汇编代码（最左边的是机器指令的地址）如下。

```
0x56556210 <+72>:    push   -0xlc(%ebp)
0x56556213 <+75>:    push   -0x20(%ebp)
0x56556216 <+78>:    call    0x5655619d <fadd>
0x5655621b <+83>:    add     $0x8, %esp
0x565562le <+86>:    mov    %eax, -0x18(%ebp)
```

设执行 result = fadd(x, y) 之前，esp 的值为 0xffffd500。

① 在表格的适当位置填写刚进入函数 fadd 内部时，堆栈中存放的相关数据；

② 刚进入函数 fadd 内部时，esp = 0x__ ff ff d4 f4__；

③ 执行 “ add $0x8, %esp ” 之后，esp = 0x__ff ff d5 00__。

	0xffffd4ec
	0xffffd4f0
56 55 62 1B	0xffffd4f4
00 00 12 34	0xffffd4f8
FF FF FF E0	0xffffd4fc
XXXXXXXX	0xffffd500

(4) fadd 函数的反汇编代码：

```
0x5655619d <+0>:     push   %ebp
0x5655619e <+1>:     mov    %esp, %ebp
0x565561a0 <+3>:     sub    $0x10, %esp
0x565561a3 <+6>:     mov    0x8(%ebp), %edx
0x565561a6 <+9>:     mov    0xc(%ebp), %eax
0x565561a9 <+12>:    add    %edx, %eax
0x565561ab <+14>:    mov    %eax, -0x4(%ebp)
0x565561ae <+17>:    mov    -0x4(%ebp), %eax
0x565561b1 <+20>:    ……
```

① 函数参数 a 的地址（即&a）是 0x__ff ff d4 f8__。

② 局部变量 temp 的地址（即&temp）是 0x__ff ff d4 ec__。

③ 执行 “add %edx, %eax” 后， CF=__1__, SF=__0__, ZF=__0__, OF= __0__。

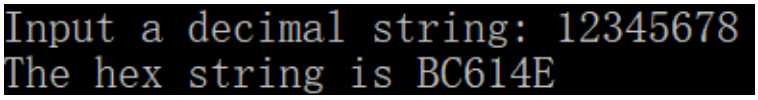
④ 在函数的结束处，有程序段

```
__mov   %ebp, %esp__
__pop   ebp__________
ret
```

执行后可返回到调用函数的断点处。

11. 编写完整汇编程序（Intel）：主程序调用 C 函数 scanf(“%s”)从键盘接受一个无符号 10 进制数字符串，然后调用子程序 str10_to_num（需要编写）将这个 10 进制字符串转换为一个长整数，接着调用子程序 num_to_str16（需要编写）将这个长整数转换为 16 进制字符

串，最后主程序调用 C 函数 printf(“%s”)在显示器上输出这个 16 进制字符串。例如：



要求：(1) 不能使用子程序定义和调用的伪指令（如 f1 proc v1:dword，invoke 100）；

(2) 调用子程序时，只能采用堆栈传递参数。

答案：

```
.686P
.model flat, c
  ExitProcess  proto stdcall :dword
  printf       proto c :vararg
  scanf        proto c :vararg
  includelib   user32.lib                    ;; MessageBoxA()
  includelib   kernel32.lib                  ;; ExitProcess()
  includelib   libcmt.lib                    ;;_mainCRTStartup => main
  includelib   legacy_stdio_definitions.lib  ;;printf

.data
  inMsg   db  "Input a decimal string: ", 0
  inFmt   db  "%s",0
  outFmt  db  "The hex string is %s",0
  decBuf  db  16 dup(0)
  hexBuf  db  16 dup(0)
  num     dd  0

.stack   200

.code
main    proc
start:  push    offset inMsg
        call    printf              ;printf(inMsg)
        add     esp, 4
        push    offset decBuf
        push    offset inFmt
        call    scanf               ;scanf("%s", decBuf)
        add     esp, 8
        ;;
        push    offset decBuf
        call    str10_to_num        ;num = str_to_num(decBuf)
        add     esp, 4
        mov     num, eax
        ;;
        push    offset hexBuf
        push    num
```

```
        call    num_to_str16        ;num_to_str16(num, hexBuf)
        add     esp, 8
        ;;
        push    offset hexBuf
        push    offset outFmt
        call    printf              ;printf(outMsg, hexBuf)
        add     esp, 8
        ;;
        push    0
        call    ExitProcess
main    endp

str10_to_num        proc
            mov     esi, [esp+4] ;decBuf
            mov     eax, 0
toNum1:     mov     bl, [esi]
            cmp     bl, 0
            jnz     toNum2
            ret                     ;num = eax
toNum2:     sub     bl, '0'
            imul    eax, 10
            movzx   ebx, bl
            add     eax, ebx
            inc     esi
            jmp     toNum1
str10_to_num    endp

num_to_str16    proc
        mov     eax, [esp+4]    ;num
        mov     esi, [esp+8]    ;hexBuf
        mov     ebx,16
        mov     ecx, 0
toStr1: mov     edx, 0
        div     ebx
        push    edx
        inc     ecx
        cmp     eax, 0
        jnz     toStr1
toStr2: pop     edx
        cmp     dl, 9
        ja      toStr3
        add     dl, '0'
        jmp     toStr4
toStr3: sub     dl, 10
        add     dl, 'A'
```

```
toStr4:  mov     [esi], dl
         inc     esi
         loop    toStr2
         mov     [esi], cl  ;(cl)=0
         ret
num_to_str16     endp
         end
```