******华中科技大学计算机科学与技术学院2023~2024第一学期**

解答内容不得超过装订线

**“ 计算机系统基础 ”模拟**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **考试方式** | **闭卷** | **考试日期** | **2023-12-03** | **考试时长** | **150 分钟** |
| **专业班级** |  | **学 号** |  | **姓 名** |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **题号** | **一** | **二** | **三** | **四** | **五** | **总分** | **核对人** |
| **分值** | 20 | 30 | 20 | 20 | 10 | 100 |  |
| **得分** |  |  |  |  |  |  |  |

|  |  |
| --- | --- |
| **分 数** |  |
| **评卷人** |  |

1. **计算机工作基本原理填空（共20分，每空1分）**

1、计算机采用存储程序的工作方式，自动逐条取出和执行指令。指令在内存中的存储地址由程序计数器指明（Intel CPU中为指令指示器EIP 或者RIP，下面以 EIP为例）。EIP自动变化的规则和变化方式如下：

① 在取出指令、对指令进行译码后，EIP会自动加上\_\_该指令的字节长度\_\_\_\_\_\_\_；若该指令不涉及转移，则可以实现程序的顺序执行；

② 若指令是条件转移指令，如\_ JZ\_（或 JE）\_\_\_\_LP (LP为一个标号)，在ZF=1时，会执行（EIP）+位移量→ EIP , 其实质操作是将\_\_\_ LP的偏移（或者有效）地址 送到EIP中；

③ 若指令是无条件转移指令JMP， 除了直接跳转外，还可以进行间接跳转。借助于寄存器EAX，通过间接跳转实现与“JMP LP”相同的功能，可以使用语句：

【使用Intel 格式(At&T格式均可)】

MOV EAX，OFFSET LP 或者 LEA EAX，LP

JMP EAX

【使用At&T格式】

MOV $LP %EAX 或者 LEA LP, %EAX

JMP \*%EAX

④ 借助堆栈段和子程序返回指令，实现与“JMP LP”相同的功能，可以使用语句：

PUSH OFFSET LP 或者 MOV [ESP], OFFSET LP

RET

【使用At&T格式】

PUSHL $LP 或者 MOV $LP, (%ESP)

RET

⑤ 执行 CALL 指令时，CPU 会\_将EIP（断点地址）压栈，然后子程序的首地址送给EIP \_\_\_；

⑥在程序运行过程中，出现中断信号时，CPU会根据中断类型号，利用\_\_中断矢量表\_\_\_\_\_\_\_\_\_\_找到中断处理程序的入口地址。中断处理程序中有\_\_\_ IRET \_\_\_\_\_\_\_指令，从栈中取出进入中断处理程序前保存的断点地址送给 EIP，从而实现被中断程序的继续执行。

2、机器指令一般都有操作对象，指令中可以直接存放常量操作数或者操作数的地址。操作数的地址有多种表达方式。 下面一段代码给出了访问同一单元的多种C 语句。请回答C语句访问方式与机器指令的对应关系。

int global[5] = { 10,20,30,40,50 }; // 全局变量

void func ( )

{ int local[5] = { 10,20,30,40,50 };

int i = 2;

int \*p;

global[i] = 25;

\*(global + 2) = 25;

p = &global[2];

\*p = 25;

local[i] = 25;

}

① 在反汇编窗口看到global[i]=25对应如下两条指令：

mov -0x18(%ebp), %eax # 显示符号名时的语句为 mov i, %eax

movl $0x19, 0x419000(, %eax, 4)

第一条指令中变量i的地址表达式为 \_\_-0x18(%ebp)\_\_\_\_ ；第二条指令中目的操作数的寻址方式是\_\_\_\_\_变址寻址\_\_\_\_\_\_\_，数组global的起始地址是 \_\_\_\_\_0x419000\_\_\_\_\_\_，目的操作数寻址中采用比例因子4的原因是\_\_数组中每个元素的字节长度为4\_\_\_\_\_\_\_。

② \*(global + 2) = 25; 仅对应一条机器指令，指令为 \_\_\_ mov dword ptr[00419008h], 19h

或 movl $0x19, 0x00419008 \_\_\_\_

其目的操作数的寻址方式是 \_\_\_直接寻址\_\_\_\_\_\_ 。

③ \*p = 25; 如果变量p 的地址表达式为 -0x1c(%ebp)，访问存储单元(\*p)时，使用的是寄存器间接寻址方式（使用寄存器 eax），则对应的两条汇编指令为：

\_\_ mov eax, dword ptr [ebp-1ch]\_\_或者\_ mov eax, [ebp-1ch]\_

\_\_\_mov dword ptr [eax], 19h\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_或\_\_mov -0x1c(%ebp), %eax\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_mov $0x19, (%eax) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

④ local[i]=25；访问local[i] 采用的是基址加变址寻址方式，对应的汇编指令如下：

movl -0x18(%ebp), \_\_%eax\_\_\_\_\_\_

movl $0x19, -0x14(%ebp, %eax, 4)

与全局变量global空间分配位置不同，local的空间分配在栈上。

数组local的起始地址的表达式为：\_\_\_\_-0x14(%ebp)\_\_\_\_\_\_\_\_\_。

|  |  |
| --- | --- |
| **分 数** |  |
| **评卷人** |  |

1. **数据存储及C语句转换填空（共30分）**

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 min(int x, int y)

{

int t;

if (x > y) t= y;

else t = x;

return t;

}

void main( )

{

char buf[6] = "12345"; // '1'的ASCII是 0x31

int u = 0x20221204; // movl $0x20221204, -0x20(%ebp)

int v = -32; // movl $0xffffffe0, -0x1c(%ebp)

int w = 0; // movl $0, -0x18(%ebp)

w = min(u, v);

w = \*(int \*)(buf+1);

}

在调试时，在监视窗口中看到数组 buf 的起始地址为 0xffffb516，变量 u 的地址（即&u）为 0xffffd508； v 的地址(即&v) 为 0xffffd50c。

1. 填写执行到“w=min(u, v)；”时，内存中数据存放的结果。要求以字节为单位、用16进制数的形式填空，最左边是内存窗口显示的内存地址。（6分）

0xffffd508 \_04\_\_ \_\_12\_\_\_ \_\_22\_\_\_ \_20\_\_\_ \_\_e0\_\_\_ \_\_ff\_\_\_ \_\_ff\_\_\_ \_\_ff\_\_\_

0xffffd510 XX XX XX XX XX XX \_\_31\_\_\_ \_\_32\_\_\_

0xffffd518 \_\_33\_\_\_ \_34\_\_\_\_ \_35\_\_\_\_ \_\_00\_\_\_ XX XX XX XX

1. 数据传送指令解读 （6分，每空 2分）

语句 “int u = 0x20221204;”的反汇编指令为： **m**ovl $0x20221204, -0x20(%ebp)

根据观察到的 u 的地址， 推断执行该语句时， ebp= 0x \_ ffffd528\_\_\_\_\_\_\_, 该指令的机器码为 c7 45 e0 04 12 22 20，其中 e0 对应的是指令中\_\_\_-0x20\_\_\_\_\_\_\_\_的编码。

执行 “w = \*(int \*)(buf+1);”后， w中的值为 0x\_35343332\_\_\_\_\_\_ 。

1. 函数调用语句解读 （10分，每空2分）

对于语句“w=min(u, v);”对应的反汇编代码（最左边的是机器指令的地址）如下。

|  |  |
| --- | --- |
|  | 0xffffd4ec |
|  | 0xffffd4f0 |
| 0x5655621b | 0xffffd4f4 |
| 0x20221204 | 0xffffd4f8 |
| 0xffffffe0 (或-32) | 0xffffd4fc |
| XXXXXXXX | 0xffffd500 |

0x56556210 <+72>: push -0xlc(%ebp)

0x56556213 <+75>: push -0x20(%ebp)

0x56556216 <+78>: call 0x5655619d <min>

0x5655621b <+83>: add $0x8, %esp

0x565562le <+86>: mov %eax, -0x18(%ebp)

设执行“w=min(u ,v);”之前，esp的值为0xffffd500

① 在表格的适当位置填写 刚进入函数min内部时，堆栈中存放的相关数据。

② 刚进入函数min内部时，esp = 0x\_\_ ffffd4f4\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

③ 执行 “ add $0x8, %esp ” 之后，esp = 0x\_\_ ffffd500\_\_\_\_\_\_\_\_\_\_\_。

4. 函数min的指令解读 (8分，每小题2分)

函数体对应的反汇编代码有： [与 主函数的地址不一样]

0x5655619d <+0>: push %ebp

0x5655619e <+1>: mov %esp, %ebp

0x565561a0 <+3>: sub $0x10, %esp

int t;

if(x>y)

0x565561a3 <+6>: mov 0x8(%ebp), %eax

0x565561a6 <+9>: cmp 0xc(%ebp), %eax // %eax 是目的操作数

0x565561a9 <+12>: jle 0x565561b3 <min+22>

t=y

0x565561ab <+14>: mov 0xc(%ebp), %eax

0x565561ae <+17>: mov %eax, -0x4(%ebp)

0x565561b1 <+20>: jmp 0x565561b9 <min+28>

else

t =x

0x565561b3 <+22>: mov 0x8(%ebp), %eax

0x565561b6 <+25>: mov %eax, -0x4(%ebp)

0x565561b9 <+29>:

① 函数参数 x的地址（即&x）是 0x\_\_ ffffd4f8\_\_\_\_\_\_\_\_\_\_\_。

② 局部变量 t的地址（即&t）是 0x\_\_\_ ffffd4f8\_\_\_\_\_\_\_\_\_\_。

③ 执行 “cmp 0xc(%ebp), %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。

|  |  |
| --- | --- |
| **分 数** |  |
| **评卷人** |  |

1. **程序阅读与理解 （共20分）**

**1、阅读下面的程序，回答问题 （10分）。**

section .data

buf: long -100, 2022, -32, 90, -30

len = (-buf)/4

fmt: ascii"%d\n\0"

.section .text

.global \_start

\_start:

mov $0, %eax

mov $len, %ecx # ①

lea buf, %edi # ②

LP\_START:

cmpl $0, (%edi)

jle LP\_NEXT # ③

inc %eax

LP\_NEXT:

add $4, %edi

sub $1, %ecx

jne LP\_START

push %eax

push $fmt

call printf

mov $l, %eax # 程序正常退出

mov $0, %ebx

int $0x80

1. 上述程序的功能是什么？运行后，屏幕上显示的是什么？（2分）

统计正数并显示。 2

1. 若将 ③ 处的语句改为 “jbe LP\_NEXT”,程序运行的结果是什么？(2分)

显示 5 ，（所有的无符号数都不会低于等于 0）

1. 若标号 LP\_START 写到 ②处语句前，程序运行的结果是什么？为什么？（3分）

显示 0；循环时，每次 edi 都要指向 buf 的开头元素，该元素 <0。

1. 若标号 ①处语句写到标号LP\_START之后，程序运行会出现什么现象？为什么？（3分）

LP\_START: mov $len, %ecx

cmpl $0, (%edi) ……

表面上看是死循环，循环计数器每循环一次都复原了。但是程序运行结果是出现访问异常，因为 操作数的地址指针 edi 在不断增加，循环一定次数后，会超出程序的空间范围，执行 cmpl $0, (%edi) 就会引发异常。

**2. 程序填空（10分，每空 1分）**

将一个以0结束的字符串中所有的小写字母转换为对应的大写字母，并将转换后的字符串、以及修改的字母个数输出。

.section .data

fmt: .ascii "%s %d\n\0"

buf: .ascii "Hello, 123, Good\n\_\0\_\_\_"

.section .text

.global \_start

\_start:

mov \_\_$0\_\_\_\_\_\_\_, %esi # esi 存放buf 数组元素的下标

mov \_\_$0\_\_\_\_\_\_\_, %ecx # ecx 存放修改字母的个数

begin:

mov buf(%esi), %al

cmp $0, %al

jz over\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

cmp $‘a’, %al

jb next\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

cmp $’z’, %al

ja next \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

add $-0x20\_\_\_\_\_\_\_\_\_\_\_, %al

mov %al, buf(%esi)\_\_\_\_\_\_\_\_\_\_\_

inc %ecx\_\_\_\_\_或 add $1, %ecx\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

next:

inc %esi 或 add $1, %esi \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

jmp begin

over:

push %ecx

pushl $buf

pushl $fmt

call printf

mov $1, %eax # 程序正常退出

mov $0, %ebx

int $0x80

|  |  |
| --- | --- |
| **分 数** |  |
| **评卷人** |  |

1. **程序优化 （共20分）。**
2. 对C程序进行编译时，编译器可以做优化工作。请举出5个不同的优化场景，说明做了什么优化，以及能加快执行速度的原因。 (10 分)

① 访问二维数组时，将列序优先调整为行序优先。二维数组是按行序存放的，在访问数据时，数据会成块地调入 CPU的cache中，行序访问能够提高cache的命中率，避免不停地在内存和cache之间交换数据，从而提高执行速度。

② 用循环访问一维数组时，将循环展开，即变成无循环的语句，或者循环次数少的语句。因为CPU中采用指令流水线的方式，转移指令会导致流水线上的一些指令加工工作被废弃。若是顺序执行指令，则会充分发挥指令流水线的作用，提高速度。

1. 分支语句向无分支语句转换，例如使用条件传送指令代替转移指令。原理同②。
2. 用执行快的指令代替执行慢的指令。例如 x\*2， 用 x<<1 来代替。
3. CPU中的串操作指令、SIMD 指令，都可以用来加快程序的执行速度。
4. 执行算法的优化， 例如 3\*x, 可以变成 2\*x +x；而2\*x又可以使用逻辑左移1位指令来实现。虽然，指令条数变多，但速度会更快；原理就是 CPU执行指令的速度不一样，一条慢的指令比几条快的指令耗时更多。
5. 调整指令的执行顺序，避免访存时出现冲突，导致阻塞，提高流水线的使用效率。

上述优化是与计算机硬件设备相关的。其他优化包括软件层面的优化，如编译器就可以计算出值的表达式，可以直接得到结果，而不需要生成相应的机器指令序列。

2、如下的 C语言程序段的功能，其编译后调试版本的汇编语言代码如下(注：在反汇编时，勾选显示符号名)。(10分)

**for(i=0;i<5;i++) sum+=array[i]:**

0x080491b8 <+66>: mov1 $0x0, -0x28(%ebp)

0x080491bf <+73>: jmp 491cf<sm+89>

0x080491c1 <+75>: mov -0x28(%ebp), %eax

0x080491c4 <+78>: mov -0x20(%ebp,%eax,4), %eax

0x080491c8 <+82>: add %eax, -0x24(%ebp)

0x080491cb <+85>: addl $0x1, -0x28(%ebp)

0x080491cf <+89>: cmp1 $0x4, -0x28(%ebp)

0x080491d3 <+93>: jle 0x80491c1 <sum+75>

(1) 指出该段程序执行效率不高的原因。（4分）

重复访问变量时，如变量 i，sum.，每次都从内存取数据，效率不高。

使用循环的方法，涉及到转移，效率也会低。

(2) 编写相应的优化汇编语言程序段，以提高程序的执行效率。(6分)

要求优化时保留循环结构，在优化程序段中可以用标号来代替转移指令的目的地址，要求写出寄存器的分配（即与变量的绑定关系）。

可用寄存器 edx 与 i 绑定， 用 eax 与 sum绑定

mov1 $0x0, %edx

movl -0x24(%ebp), %eax

jmp L2

L1: mov -0x20(%ebp,%edx,4), %ebx

add %ebx, eax

addl $0x1, %edx

L2: cmp1 $0x4, %edx

jle L1

movl %eax, -0x24(%ebp)

消除循环的写法

lea -0x20(%ebp), %esi ； 存放数组元素的地址

mov $0, %edx ; 存放结果

addl (%esi), %edx

add $4, %esi

addl (%esi), %edx

add $4, %esi

addl (%esi), %edx

add $4, %esi

addl (%esi), %edx

add $4, %esi

addl (%esi), %edx

addl %edx, -0x24(%ebp)

|  |  |
| --- | --- |
| **分 数** |  |
| **评卷人** |  |

1. **程序的链接问答（10分）**
2. 在对一个C程序文件进行编译和汇编后，生成的可重定位目标文件(obj文件，或者 .o文件)，一般由哪些节组成（至少说出6个组成部分的名字）？（3分）

文件头、节头表、代码节.text, 已初始化数据节.data, 未初始化数据节.bss,

只读数据节 .rodata，符号表节 .symtab, 字符串表节 .strtab、代码节的重定位节(.rela.text)、数据节的重定位节(.rela.data) 等等。

1. C程序中的什么符号需要重定位？什么符号不需要重定位？（2分）

函数参数、非静态的局部变量、本模块的静态函数不需要重定位；  
静态的局部变量、全局变量、非静态函数需要重定位。

1. 设有一个函数中，有语句 int x=g; 其中g是一个初值为20的int 类型全局变量。编译器对g的定义(int g=20;)和x=g编译时，在可重定位目标文件中会生成哪些信息？（5分）

对于 int g =20; 在**字符串节**中，要存放全局变量g的名字(ASCII串)；

在**符号表节**要存放与g相关的信息，如g的值占存储空间的大小<4>，定义g的节号<数据节>，g的地址（对当前文件而言，一个节的起始地址都是0，g在该节的什么位置）；绑定方式<global>等等；

在**数据节**要存放 g的初值20。

对于 int x=g；在**代码节**要生成相应的指令，其中 对于g 的引用，先使用占位符（00 00 00 00）来填充。在**代码节的重定位节**，要产生一条相应的信息，表明代码节的什么位置要被替换成一个什么样的值（即用什么重定位方式来修改占位符），该值来源于哪一个变量（符号表的索引项）、以及修改时的附加量等。