Short Warmup Test2

This “warm-up” test is by no means meant to be comprehensive (see the study guide for a comprehensive list), but it will give you some sample problems on which to work. I recommend that you work on them with a friend from class. Remember no calculators will be permitted when you take the test.

1. The prototype for the C equivalent to the following function is \texttt{int foo(int n1, int n2);}:

\begin{verbatim}
foo:
pushl  %ebp
movl  %esp, %ebp
subl  $4, %esp
movl  8(%ebp), %eax
cmpl  12(%ebp), %eax
jle   .L3
movl $0, -4(%ebp)
jmp   .L2
.L3:
subl  $8, %esp
pushl 12(%ebp)
movl  8(%ebp), %eax
incl  %eax
pushl %eax
call  foo
addl $16, %esp
addl  8(%ebp), %eax
movl  %eax, -4(%ebp)
.L2:
movl -4(%ebp), %eax
leave
ret
\end{verbatim}

(a) What is returned by \texttt{foo} if it is called like this:

\texttt{int foo(0, 4);} 

(b) Give the C code equivalent to the \texttt{foo} function.
2. Consider the following structure and union declarations:

```c
struct s1
{
    union u * ptr;
    int num;
    char let;
};

union u
{
    int anotherNum;
    char anotherLet;
};
```

Using the IA32 assembly code sequence below on the left, fill in the missing portion of the corresponding C source code on the right.

```c
void procedure(struct s1 * s)
{
    s->ptr->anotherNum = _____________________;
    s->let = _________________________________;
}
```

```asm
procedure: void procedure(struct s1 * s)
{
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    movl (%eax), %eax
    movb (%eax), %al
    movb %al, 8(%edx)
    leave
    ret
```
3. Consider the following datatype definitions on a Windows/x86 machine.

```c
typedef struct {
    char c;
    double *p;
    int i;
    double d;
    short s;
} struct1;
```

```c
typedef union {
    char c;
    double *p;
    int i;
    double d;
    short s;
} union1;
```

A. Using the template below (allowing a maximum of 32 bytes), indicate the allocation of data for a structure of type `struct1`. Mark off and label the areas for each individual element (there are 5 of them). Cross hatch the parts that are allocated, but not used (to satisfy alignment).

Assume the alignment rules discussed in lecture: data types of size $x$ must be aligned on $x$-byte boundaries. **Clearly indicate the right hand boundary of the data structure with a vertical line.**

```
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
| |
```

B. How many bytes are allocated for an object of type `struct1`?

C. What alignment is required for an object of type `struct1`? (If an object must be aligned on an $x$-byte boundary, then your answer should be $x$.)

D. If we define the fields of `struct1` in a different order, we can reduce the number of bytes wasted by each variable of type `struct1`. What is the number of unused, allocated bytes in the best case?

E. How many bytes are allocated for an object of type `union1`?

F. What alignment is required for an object of type `union1`? (If an object must be aligned on an $x$-byte boundary, then your answer should be $x$.)
4. Consider the source code below, where M and N are constants declared with \#define.

```c
int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j)
{
    return mat1[i][j] + mat2[i][j];
}
```

Suppose the above code generates the following assembly code:

```assembly
sum_element:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 12(%ebp),%ecx
    sall $2,%ecx
    leal 0(,%eax,8),%edx
    subl %eax,%edx
    leal (%eax,%eax,4),%eax
    movl mat2(%ecx,%eax,4),%eax
    addl mat1(%ecx,%edx,4),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

What are the values of M and N?

M = 

N =
5. The next problem concerns the following C code:

```c
/* copy string x to buf */
void foo(char *x) {
    int buf[1];
    strcpy((char *)buf, x);
}

void callfoo() {
    foo("abcdefghi");
}
```

Here is the corresponding machine code on a Linux/x86 machine:

```
080484f4 <foo>:
080484f4: 55 pushl %ebp
080484f5: 89 e5 movl %esp,%ebp
080484f7: 83 ec 18 subl $0x18,%esp
080484fa: 8b 45 08 movl 0x8(%ebp),%eax
080484fd: 83 c4 f8 addl $0xfffffffa,%esp
08048500: 50 pushl %eax
08048501: 8d 45 fc leal 0xfffffffc(%ebp),%eax
08048504: 50 pushl %eax
08048505: e8 ba fe ff ff call 80483c4 <strcpy>
0804850a: 89 ec movl %ebp,%esp
0804850c: 5d popl %ebp
0804850d: c3 ret

08048510 <callfoo>:
08048510: 55 pushl %ebp
08048511: 89 e5 movl %esp,%ebp
08048513: 83 ec 08 subl $0x8,%esp
08048516: 83 c4 f4 addl $0xfffffffa,%esp
08048519: 68 9c 85 04 08 pushl $0x804859c # push string address
0804851e: e8 d1 ff ff ff call 80484f4 <foo>
08048523: 89 ec movl %ebp,%esp
08048525: 5d popl %ebp
08048526: c3 ret
```
This problem tests your understanding of the stack discipline and byte ordering. Here are some notes to help you work the problem:

- `strcpy(char *dst, char *src)` copies the string at address `src` (including the terminating `\0` character) to address `dst`. It does **not** check the size of the destination buffer.
- Recall that Linux/x86 machines are Little Endian.
- You will need to know the hex values of the following characters:

<table>
<thead>
<tr>
<th>Character</th>
<th>Hex value</th>
<th>Character</th>
<th>Hex value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>0x61</td>
<td>'f'</td>
<td>0x66</td>
</tr>
<tr>
<td>'b'</td>
<td>0x62</td>
<td>'g'</td>
<td>0x67</td>
</tr>
<tr>
<td>'c'</td>
<td>0x63</td>
<td>'h'</td>
<td>0x68</td>
</tr>
<tr>
<td>'d'</td>
<td>0x64</td>
<td>'i'</td>
<td>0x69</td>
</tr>
<tr>
<td>'e'</td>
<td>0x65</td>
<td>'0'</td>
<td>0x00</td>
</tr>
</tbody>
</table>

- If the entire contents of a word in the stack wasn’t overwritten then certain bytes in the word would remain as they were before the `strcpy`. You’ll need to figure out some of the original contents of the stack.

Now consider what happens on a Linux/x86 machine when `call foo` calls `foo` with the input string “abcdefghi”.

(a) List the contents of the following memory locations immediately after `strcpy` returns to `foo`. Each answer should be an unsigned 4-byte integer expressed as 8 hex digits.

```
buf[0] = 0x____________________
buf[1] = 0x____________________
buf[2] = 0x____________________
```

(b) Immediately **before** the `ret` instruction at address `0x0804850d` executes, what is the value of the frame pointer register `%ebp`?

```
%ebp = 0x____________________
```

(c) Immediately **after** the `ret` instruction at address `0x0804850d` executes, what is the value of the program counter register `%eip`?

```
%eip = 0x____________________
```
6. Use the Y86 code below to answer the parts of this problem.

(a) Using figures 4.2, 4.3, and 4.4 to give the encoding for the following Y86 instructions. (Assume
the address of the first instruction is 0.)

I1: irmovl ele1, %ecx
I2: irmovl ele2, %eax
I3: mrmovl (%ecx), %ecx
I4: mrmovl (%eax), %eax
I5: addl %ecx, %eax
I6: halt
.align 4
ele1: .long 3
ele2: .long 4

(b) What values would be in the registers %ecx and %eax when the halt statement is executed.

(c) Fill in the table below, identifying data dependences in the sequence of code above. For source of
dependence, fill in the label of the statement producing the result and for target fill in the label
of the statement using the result.

<table>
<thead>
<tr>
<th>Source of dependence</th>
<th>Target of dependence</th>
<th>Register involved</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(d) Would the code produce the same results (ie. free of hazards) if executed on the PIPE- processor?
If not, rewrite the code so that when executing it on the PIPE- processor will not result in hazards.