Please write neatly and clearly. You have 75 minutes to answer all questions. Closed book and closed notes EXCEPT: you are allowed to bring a copy of the Intel x86-64 reference sheets provided in the lab discussion (you should not have any written notes on your copy). No calculators or electronic devices are allowed.

<table>
<thead>
<tr>
<th>Question</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td></td>
</tr>
<tr>
<td>#2</td>
<td></td>
</tr>
<tr>
<td>#3</td>
<td></td>
</tr>
<tr>
<td>#4</td>
<td></td>
</tr>
<tr>
<td>#5</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
</tr>
</tbody>
</table>
NAME:

1) Match each of the assembler routines on the left with the equivalent C function on the right.

```assembly
foo1:
sar  $0x7,%edi
mov  %edi,%eax
and  $0x1,%eax
retq

foo2:
lea  (%rdi,%rdi,1),%eax
retq

foo3:
lea  0x4(%rdi,4),%eax
retq
```

```c
int choice1(int x)
{
    return (x + 1)*4;
}

int choice2(int x)
{
    return 47 / x;
}

int choice3(int x)
{
    return (x >> 7) & 1;
}

unsigned int choice4(unsigned int x)
{
    return (x >> 7) & 1;
}

int choice5(int x)
{
    return (x*2);
}

int choice6(int x)
{
    return 17 / x;
}
```

**Fill in your answers here:**

foo1 corresponds to choice ______.
foo2 corresponds to choice ______.
foo3 corresponds to choice ______.
NAME:
  2) Consider the following assembly representation of a function `foo` containing a `for` loop:

```
loop:
    movslq %esi, %rsi
    leaq -1(%rdi,%rsi), %rax
    jmp .L2
.L3:
    movzbl (%rdi), %edx
    xor %rax, %dl
    mov %dl, (%rdi)
    xor %rax, %dl
    mov %dl, (%rax)
    xor %dl, (%rdi)
    add $1, %rdi
    sub $1, %rax
.L2:
    cmp %rax, %rdi
    jb .L3
    ret
```

Fill in the blanks to provide the functionality of the loop:

```c
int loop(char * h, int len)
{
    char *t;
    for( _______; _______; h++, t-- ) {
        ------------------;
        ------------------;
        ------------------;
    }
}
```
3) Consider the source code below, where M and N are constants declared with `#define`.

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

int copy_element(int i, int j)
{
    mat1[i][j] = mat2[j][i];
}
```

This generates the following assembly code:

```assembly
copy_element:
    movslq  %edi, %rdi
    movslq  %esi, %rsi
    leaq    (%rsi,%rsi,2), %rax
    leaq    (%rsi,%rax,4), %rax
    addq    %rdi, %rax
    movl    mat2(,%rax,4), %edx
    leaq    (%rsi,%rdi,8), %rax
    movl    %edx, mat1(,%rax,4)
    ret
```

(a) What is the value of M:

(b) What is the value of N:
NAME: The next problem concerns the following C code:

```c
void foo(int x)
{
    int a[2];
    a[0] = 0xF0F1F2F3;
    a[1] = x;
    gets((char *)(a));
    printf("a=0x%016x a[0] = 0x%x, a[1] = 0x%x\n", a, a[0], a[1]);
}

int
main(int argc, char **argv)
{
    foo(0xdeadbeef);
    return 0;
}
```

Here is the corresponding machine code on a Linux/x86_64 machine:

**Dump of assembler code for function foo:**

```
0x0000000000400580 <+0>:  sub     $0x18,%rsp
0x0000000000400584 <+4>:  movl    $0xf0f1f2f3,(%rsp)
0x000000000040058b <+11>: mov     %edi,0x4(%rsp)
0x000000000040058f <+15>: mov     %rsp,%rdi
0x0000000000400592 <+18>: callq   0x400480 <gets@plt>
0x0000000000400597 <+23>: mov     0x4(%rsp),%ecx
0x000000000040059d <+29>: mov     (%rsp),%edx
0x00000000004005a1 <+33>: mov     (%rsp,%edi) 0x400660,%edi
0x00000000004005a6 <+38>: mov     0x0,%eax
0x00000000004005aa <+42>: callq   0x400450 <printf@plt>
0x00000000004005b0 <+48>: add     $0x18,%rsp
0x00000000004005b4 <+52>: retq
```

**Dump of assembler code for function main:**

```
0x00000000004005b5 <+0>:  sub     $0x8,%rsp
0x00000000004005b9 <+4>:  mov     $0xdeadbeef,%edi
0x00000000004005bc <+9>:  callq   0x400580 <foo>
0x00000000004005c3 <+14>: mov     $0x0,%eax
0x00000000004005c8 <+19>: add     $0x8,%rsp
0x00000000004005cc <+23>: retq
```
4) This problem tests your understanding of how procedures and the stack work, arrays, and byte ordering. Here are some notes to help you work the problem:

- `gets(char *dst)` copies bytes from the input to the string at address `dest` until a return (newline) is detected (it does not copy the return but does null terminate the string with '\n' character). It does **not** check the size of the destination buffer.
- Note that Linux/x86_64 machines are Little Endian, i.e. the lowest order byte of a multi-byte integer is stored in the lowest memory address.
- You will need to know the hex values of the following characters: 'a' - 'z' are the contiguous values from 0x61 to 0x7a inclusively and null is 0x00.

Now consider what happens on a Linux/x86_64 machine when `main` calls `foo` and the user enters the input string "abcdefghijklmnopqrstuvwxyz" followed by a return.

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

\[
\begin{align*}
a[0] &= 0x____________ \\
a[1] &= 0x____________ \\
a[2] &= 0x____________ \\
a[3] &= 0x____________ \\
a[4] &= 0x____________ \\
a[5] &= 0x____________ \\
a[6] &= 0x____________ \\
\end{align*}
\]

(b) Immediately after the `add` instruction at address 0x00000000004005b0 executes, what is the 8 byte value at the top of the stack?

\[
\text{VALUE AT THE TOP OF THE STACK } = 0x__________________________
\]

(c) Immediately after the `retq` instruction at address 0x00000000004005b4 executes, what is the value of the program counter register `%rip`?

\[
%rip = 0x__________________________
\]
NAME:

5) Consider the following C declaration:

```c
struct Node {
    long value;
    int num;
    char type;
    struct Node* next;
    struct Node* prev;
};

typedef struct Node* pNode;

/* NodeTree is an array of N pointers to Node structs */
pNode NodeTree[N];
```

(a) Using the template below (allowing a maximum of 32 bytes), indicate the allocation of data for a Node struct. 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 (padding for alignment).

Assume the x86_64 Linux alignment rules as follows.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (bytes)</th>
<th>Alignment (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>pointers</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Clearly indicate the right hand boundary of the data structure with a vertical line.
(b) For each of the two C references below, please indicate which assembly code section (labeled A – F) places the value of that C reference into register %eax. If no match is found, please write “NONE” next to the C reference. Don’t forget NodeTree is an array of pointers.

The initial register-to-variable mapping for each assembly code section is:

%rdi = starting address of the NodeTree array
%rsi = i

C References:

1. _____ NodeTree[i]->next->prev->num;

2. _____ NodeTree[i]->prev->next->num;

Linux/x86_64 Assembly:

A: mov (%rdi,%rsi,8),%rax
   mov 0x10(%rax),%rax
   mov 0x18(%rax),%rax
   mov 0x8(%rax),%eax

B: mov (%rdi,%rsi,8),%rax
   mov 0x10(%rax),%rax
   mov 0x10(%rax),%rax
   mov 0x10(%rax),%rax
   mov 0x18(%rax),%rax

C: mov (%rdi,%rsi,8),%rax
   mov 0x18(%rax),%rax
   mov 0x10(%rax),%rax
   mov 0x8(%rax),%eax

D: mov (%rdi,%rsi,8),%rax
   mov (%rax),%rax
   mov (%rdi,%rsi,8),%rax

E: mov (%rdi,%rsi,8),%rax
   mov (%rax),%rax

F: movslq %esi,%rsi
   mov (%rdi,%rsi,8),%rax
   mov 0x18(%rax),%rax
   movsbl 0xc(%rax),%eax