Please read through the entire examination first! We designed this exam so that it can be completed in 50 minutes and, hopefully, this estimate will prove to be reasonable.

There are 5 problems for a total of 100 points, and one 10 point extra credit problem. The point value of each problem is indicated in the table below. Write your answer neatly in the spaces provided. If you need more space, you can write on the back of the sheet where the question is posed, but please make sure that you indicate clearly the problem to which the comments apply, and that you write your name on all pages. If you have difficulty with part of a problem, move on to the next one. They are independent of each other.

The exam is CLOSED book and CLOSED notes (no summary sheets, no mobile phones, no laptops, and simple calculators only). Please do not ask or provide anything to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers.

Good luck and have fun!

Name: Solution Guide

Student ID: 

<table>
<thead>
<tr>
<th>Problem</th>
<th>Max Score</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>EC</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
1 Number Representation (10 points)

Let $x=0x\text{E}$ and $y=0x\text{7}$ be integers stored on a machine with a word size of 4 bits. Show your work with the following math operations. The answers—including truncation—should match those given by our hypothetical machine with 4-bit registers.

A. (2pt) What hex value is the result of adding these two numbers?

In hex: $0x\text{E} + 0x\text{7} = 0x\text{15} \rightarrow 0x\text{5}

In binary converted back to hex: $0x\text{E} + 0x\text{7} = 1110 + 0111 = 10101 \rightarrow 0101 = 0x\text{5}$

Half credit for not truncating to the appropriate value.

B. (2pt) Interpreting these numbers as unsigned ints, what is the decimal result of adding $x + y$?

In unsigned decimal: $0x\text{E} + 0x\text{7} = 14 + 7 = 21 \% 16 = 5$

Half credit for not truncating to the appropriate value or incorrect conversion.

No credit for computing in signed decimal.

C. (2pt) Interpreting $x$ and $y$ as two's complement integers, what is the decimal result of computing $x - y$?

In signed decimal: $0x\text{E} - 0x\text{7} = -2 - 7 = -9 \rightarrow 7$

Half credit for not truncating to the appropriate value, or incorrect conversion.

No credit for computing in unsigned decimal.

D. (2pt) In one word, what is the phenomenon happening in 1B?

Overflow.

E. (2pt) Circle all statements below that are TRUE on a 32-bit architecture:

Half point each.

- It is possible to lose precision when converting from an int to a float.  True
- It is possible to lose precision when converting from a float to an int.  True
- It is possible to lose precision when converting from an int into a double.  False
- It is possible to lose precision when converting from a double into an int.  True
2 IA32 ASM to C (20 points)

A function 'mystery' has the following overall structure:

```c
int mystery (int x, int y){
    int result;
    for (____________; __________; result++){
        ______________;
        ______________;
    }
    ______________;
    return result;
}
```

The GCC C compiler generates the following x86 (IA32) assembly code (x is at %ebp+8, y at %ebp+12)

```
01    pushl  %ebp
02    movl  %esp, %ebp
03    movl  8(%ebp), %ecx
04    movl  12(%ebp), %edx
05    movl  $0, %eax
06    test  %ecx, %ecx
07    jz     .L3
08    .L6
09    addl  %ecx, %edx
10    subl  $1, %ecx
11    addl  $1, %eax
12    cmpl  $0, %ecx
13    jg     .L6
14    .L3
15    addl  %edx, %eax
16    popl  %ebp
17    ret
```

Fill in the blanks in mystery based on the assembly code above. You may only use the symbolic variables x, y, and result in your expressions. Do not use register names.

Answers to blanks, in order:

- `result = 0;
- `x > 0;`  // Also accept x != 0
- `y += x;
- `x--;`
- `result += y`
3  C to ASM (30 points)

Write x86-64 assembly instructions (see the reference sheet for the list of instructions that you can use on this exam) that might be generated by the following function foo. It may be a good idea to consult the register chart provided on the reference sheet.

```c
int foo (int a, int b){
    int c, d;
    c = a / 16;
    d = b * 64;
    if (c > d)
        return a;
    else
        return b;
}
```

Place the assembly code for function foo here (you should need fewer than 15 instructions), and a comment for each line of your code. **You may only use the instructions that are on reference sheet!**

```
.L00
    movl %edi, %e10  # ( may use another register, but must be 32 bit)
    sar $4, %e10     # ( no credit for anything other than shift)
    movl %esi, %e11  # ( may use another register, but must be 32 bit)
    shl 6, %e11      # ( no credit for anything other than shift)
    cmpl %e11, %e10  # ( accept opposite order, if next line matches)
    jle $.L1         # ( two for instruction, 2 for useful label OR arrow OR address)
    movl %edi, %eax  #
    jmp $.END        # ( also accept ret here instead of jump to end)
.L1
    movl %esi, %eax  #
.END
    ret              # (must be present: all control flow must go through a ret)
```
4 Stack Discipline (30 points)

Given the C function

```c
int proc ( void ){
    int a[3];
    scanf("%x %x %x", &a[1], &a[0], &a[2]);
    return a[2];
}
```

GCC generates the following code:

```
01  pushl  %ebp
02  movl  %esp, %ebp
03  pushl  %ebx
04  pushl  %esi
05  subl  $0x20, %esp
06  leal  -20(%ebp), %eax
07  movl  $0, %esi
08  leal  (%eax,%esi, 4), %ebx
09  movl  %ebx, 8(%esp)
10  addl  $1, %esi
11  leal  (%eax,%esi, 4), %ebx
12  movl  %ebx, 4(%esp)
13  addl  $1, %esi
14  leal  (%eax,%esi, 4), %ebx
15  movl  %ebx, 12(%esp)
16  movl  $.LC0, (%esp)  #Pointer to string "%x %x %x"
17  call  scanf  <= here
18  movl  (%ebx), %eax
19  addl  $0x20, %esp
20  popl  %esi
21  popl  %ebx
22  movl  %ebp, %esp
23  popl  %ebp
24  ret
```

Draw a picture depicting the stack frame of `proc` immediately before the call to `scanf` (labeled "here" above). Draw labeled arrows indicating where the stack and frame pointers are. If needed, you can assume that `%esp = 0x800040` and `%ebp = 0x800060` just before `proc` is called. The next page is left blank to give you more room.

Note: though not necessary to solve the problem, `scanf` is much like the `sscanf` you saw in Lab 2 (matching an input string to some format), except it reads the input string from `stdin` (the terminal).
### 4 STACK DISCIPLINE (30 POINTS)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x800040</td>
<td>??</td>
<td>where %esp used to point</td>
</tr>
<tr>
<td>0x80003C</td>
<td>??</td>
<td>ret addr</td>
</tr>
<tr>
<td>0x800038</td>
<td>0x800060</td>
<td>old ebp</td>
</tr>
<tr>
<td>0x800034</td>
<td>??</td>
<td>saved ebx</td>
</tr>
<tr>
<td>0x800030</td>
<td>??</td>
<td>saved esi</td>
</tr>
<tr>
<td>0x80002C</td>
<td>??</td>
<td>a[2]</td>
</tr>
<tr>
<td>0x80002B</td>
<td>??</td>
<td>a[1]</td>
</tr>
<tr>
<td>0x800024</td>
<td>??</td>
<td>a[0]</td>
</tr>
<tr>
<td>0x800023</td>
<td>--</td>
<td>wasted space</td>
</tr>
<tr>
<td>0x80001C</td>
<td>0x80002C</td>
<td>&amp;a[2]</td>
</tr>
<tr>
<td>0x800018</td>
<td>0x800024</td>
<td>&amp;a[0]</td>
</tr>
<tr>
<td>0x800014</td>
<td>0x800028</td>
<td>&amp;a[1]</td>
</tr>
<tr>
<td>0x800010</td>
<td>??</td>
<td>$.LC0 (pointer to format string)</td>
</tr>
</tbody>
</table>

---

**Grading Notes:**

- First two lines in table are optional. Need to have the other 11.
- Comment column and pointer columns are required. Address and value are optional.
- If addresses are used, they must increment by the correct values.
- Any values provided &a[0], &a[1], &a[2], old ebp, must be correct.
5 Structs (10 points)

Suppose you are given the following struct definition for an x86-64 architecture which is used to implement a linked list of all tweets in Katelin’s SuperTwitter implementation.

typedef struct Super_Tweet{
    char super_tweeter[21];
    int num_retweets;
    int num_favorites;
    long id;
    tweet* next;
    int datetime_encoded; //seconds since SuperTwitter was launched
} tweet

A. (1/2pt each) Given the above definition, fill in the following table:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Offset</th>
<th>Size of Field (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>super tweeter</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>(wasted space)</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>num retweets</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td>num favorites</td>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td>id</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>next</td>
<td>40</td>
<td>8</td>
</tr>
<tr>
<td>datetime encoded</td>
<td>48</td>
<td>4</td>
</tr>
<tr>
<td>(wasted space)</td>
<td>52</td>
<td>4</td>
</tr>
</tbody>
</table>

B. (1pt) What is the size of the struct?  56 bytes

C. (1/2pt) How much internal fragmentation does this struct have?  3 bytes.

D. (1/2pt) How much external fragmentation does this struct have?  4 bytes.
6 Arrays (10 points, extra credit)

In the space below, draw the memory layout on a 32-bit machine for:

```plaintext
char a[2][3] = {{'a', 'b', 'c'}, {'d', 'e', 'f'}}
```

Half point, each box, +1 for correct ordering

<table>
<thead>
<tr>
<th>0x00</th>
<th>'a'</th>
<th>'b'</th>
<th>'c'</th>
<th>'d'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x04</td>
<td>'e'</td>
<td>'f'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
char *b[2] = {"foo", "bar"};
```

Hint: you may place "foo" and "bar" somewhere in memory, to get an address.
Half point, each character box, 1 point each pointer. Solution assumes little endian, big endian also okay.

<table>
<thead>
<tr>
<th>0x00</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x04</td>
<td>'f'</td>
<td>'o'</td>
<td>'o'</td>
<td>'\0'</td>
</tr>
<tr>
<td>0x08</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>'b'</td>
<td>'a'</td>
<td>'r'</td>
<td>'\0'</td>
</tr>
<tr>
<td>0x14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>04</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1C</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
Powers of 2:

\[
\begin{align*}
2^0 &= 1 & 0x00 &= 0 \\
2^1 &= 2 & 2^{-1} &= 0.5 & 0x0A &= 10 \\
2^2 &= 4 & 2^{-2} &= 0.25 & 0x0F &= 15 \\
2^3 &= 8 & 2^{-3} &= 0.125 & 0x20 &= 32 \\
2^4 &= 16 & 2^{-4} &= 0.0625 & 0x28 &= 40 \\
2^5 &= 32 & 2^{-5} &= 0.03125 & 0x2A &= 42 \\
2^6 &= 64 & 2^{-6} &= 0.015625 & 0x2F &= 47 \\
2^7 &= 128 & 2^{-7} &= 0.0078125 \\
2^8 &= 256 & 2^{-8} &= 0.00390625 \\
2^9 &= 512 & 2^{-9} &= 0.001953125 \\
2^{10} &= 1024 & 2^{-10} &= 0.0009765625 \\
\end{align*}
\]

Hex help:

\[
\begin{align*}
0x0 &= 0 \\
0x0A &= 10 \\
0x0F &= 15 \\
0x20 &= 32 \\
0x28 &= 40 \\
0x2A &= 42 \\
0x2F &= 47 \\
\end{align*}
\]

Assembly Code Instructions:

- **push**: push a value onto the stack and decrement the stack pointer
- **pop**: pop a value from the stack and increment the stack pointer
- **call**: jump to a procedure after first pushing a return address onto the stack
- **ret**: pop return address from stack and jump there
- **mov**: move a value between registers and memory
- **lea**: compute effective address and store in a register
- **add**: add src (1st operand) to dst (2nd) with result stored in dst (2nd)
- **sub**: subtract src (1st operand) from dst (2nd) with result stored in dst (2nd)
- **and**: bit-wise AND of src and dst with result stored in dst
- **or**: bit-wise OR of src and dst with result stored in dst
- **sar**: shift data in the dst to the right (arithmetic) by the number in 1st operand
- **shl**: shift data in the dst to the left by the number of bits specified in 1st operand
- **jmp**: jump to address
- **jg**: conditional jump to address if not zero flag and not sign flag
- **jle**: conditional jump to address if zero flag or sign flag
- **jne**: conditional jump to address if zero flag is not set
- **jns**: conditional jump to address if sign flag is not set
- **cmp**: subtract src (1st operand) from dst (2nd) and set flags
- **test**: bit-wise AND src and dst and set flags
Register map for x86-64:
Note: all registers are caller-saved except those explicitly marked as callee-saved, namely, rbx, rbp, r12, r13, r14, and r15. rsp is a special register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return Value</td>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee Saved</td>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
<td>%r10</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
<td>%r11</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
<td>%r12</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
<td>%r13</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack Pointer</td>
<td>%r14</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee Saved</td>
<td>%r15</td>
<td>Callee Saved</td>
</tr>
</tbody>
</table>