Au16 Midterm Q1

**Question 1: Number Representation [12 pts]**

(A) What is the value of the char 0b 1101 1101 in decimal? [1 pt]

(B) What is the value of char \( z = (0xB \ll 7) \) in decimal? [1 pt]

(C) Let char \( x = 0xC0 \). Give one value (in hex) for char \( y \) that results in both signed and unsigned overflow for \( x+y \). [2 pt]

For the rest of this problem we are working with a floating point representation that follows the same conventions as IEEE 754 except using 8 bits split into the following vector widths:

| Sign (1) | Exponent (4) | Mantissa (3) |

(D) What is the magnitude of the bias of this new representation? [2 pt]

(E) Translate the floating point number 0b 1100 1110 into decimal. [3 pt]

(F) What is the smallest positive integer that can’t be represented in this floating point encoding scheme? **Hint:** For what integer will the “one’s digit” get rounded off? [3 pt]
Sp15 Midterm Q1
1 Number Representation (10 points)

Let $x=0x$E and $y=0x7$ 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?

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

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

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

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

- It is possible to lose precision when converting from an int to a float.
- It is possible to lose precision when converting from a float to an int.
- It is possible to lose precision when converting from an int into a double.
- It is possible to lose precision when converting from a double into an int.
Wi18 Midterm Q2

Question 2: Pointers & Memory [14 pts.]

For this problem, assume we are executing on a 64-bit x86-64 machine (little endian). The current state of memory (values in hex) is shown below.

```c
int *x = 0x00;
long *y = 0x10;
unsigned short *z = 0x18;
```

(A) Fill in the type and value (in hex) for each of the following C expressions. Remember to use the appropriate bit widths. [8 pts.]

<table>
<thead>
<tr>
<th>Expression (in C)</th>
<th>Type</th>
<th>Value (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>unsigned short *</td>
<td>0x 0000 0000 0000 0018</td>
</tr>
<tr>
<td>*x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x+3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*(y-1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>z[3]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(B) What are the values (in hex) stored in each register shown after the following x86-64 instructions are executed? We are still using the state of memory shown above in part a. Remember to use the appropriate bit widths. [6 pts.]

<table>
<thead>
<tr>
<th>Register</th>
<th>Value (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x 0000 0000 0000 0008</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x 0000 0000 0000 0018</td>
</tr>
<tr>
<td>%cl</td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td></td>
</tr>
<tr>
<td>%r8d</td>
<td></td>
</tr>
</tbody>
</table>

movb (%rsi), %cl
leaq 16(%rsi, %rsi, 4), %rcx
movswl -10(%rsi, %rax, 4), $r8d
Pointers, Memory & Registers (14 points)

Assuming a 64-bit x86-64 machine (little endian), you are given the following variables and initial state of memory (values in hex) shown below:

<table>
<thead>
<tr>
<th>Address</th>
<th>+0</th>
<th>+1</th>
<th>+2</th>
<th>+3</th>
<th>+4</th>
<th>+5</th>
<th>+6</th>
<th>+7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>AB</td>
<td>EE</td>
<td>1E</td>
<td>AC</td>
<td>D5</td>
<td>8E</td>
<td>10</td>
<td>E7</td>
</tr>
<tr>
<td>0x08</td>
<td>F7</td>
<td>84</td>
<td>32</td>
<td>2D</td>
<td>A5</td>
<td>F2</td>
<td>3A</td>
<td>CA</td>
</tr>
<tr>
<td>0x10</td>
<td>83</td>
<td>14</td>
<td>53</td>
<td>B9</td>
<td>70</td>
<td>03</td>
<td>F4</td>
<td>31</td>
</tr>
<tr>
<td>0x18</td>
<td>01</td>
<td>20</td>
<td>FE</td>
<td>34</td>
<td>46</td>
<td>E4</td>
<td>FC</td>
<td>52</td>
</tr>
<tr>
<td>0x20</td>
<td>4C</td>
<td>A8</td>
<td>B5</td>
<td>C3</td>
<td>D0</td>
<td>ED</td>
<td>53</td>
<td>17</td>
</tr>
</tbody>
</table>

int* ip = 0x00;
short* sp = 0x20;
long* yp = 0x10;

a) Fill in the type and value for each of the following C expressions. If a value cannot be determined from the given information answer UNKNOWN.

<table>
<thead>
<tr>
<th>Expression (in C)</th>
<th>Type</th>
<th>Value (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp + 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*(sp - 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ip[5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;ip</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b) Assuming that all registers start with the value 0, except %rax which is set to 0x4, fill in the values (in hex) stored in each register after the following x86 instructions are executed. Remember to give enough hex digits to fill up the width of the register name listed.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x0000 0000 0000 0004</td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td></td>
</tr>
<tr>
<td>%si</td>
<td></td>
</tr>
</tbody>
</table>

movl 2(%rax), %ebx
leal (%rax,%rax,2), %ecx
movsbl 4(%rax), %edi
subw (,%rax,2), %si
Au17 Midterm Q3

Question 3: Design Questions [6 pts]

Answer the following questions in the boxes provided with a single sentence fragment. Please try to write as legibly as possible.

(A) We have repeatedly stated that Intel is big on legacy and backwards-compatibility. Name one example of this that we have seen in this class. [2 pt]

(B) Name one programming consequence if we decided to assign an address to every 4 bytes of memory (instead of 1 byte). [2 pt]

(C) If we changed the x86-64 architecture to use 24 registers, how might we adjust the register conventions? [2 pt]

One thing that should remain the same:

One thing that should change:
(a) Would this change make it harder or easier to implement hardware that executes instructions as quickly?

(b) Would this change make it harder or easier for software to use less stack space?

(c) Would you expect a revised calling convention to have more caller-save registers or fewer caller-save registers?

(d) Would you expect a revised calling convention to have more callee-save registers or fewer callee-save registers?

(e) Would it be possible to make this change in a way that existing x86-64 executables could still run without modifying them (yes or no)?
Su18 Midterm Q4

Question 4: C & Assembly  [24 pts]

Answer the questions below about the following x86-64 assembly function:

mystery:

```
movl $0, %eax  # Line 1
.L2:
    cmpl %esi, %eax  # Line 2
    jge .L1        # Line 3
    movslq %eax, %rdx  # Line 4
    leaq (%rdi,%rdx,2), %rcx  # Line 5
    movzwl (%rcx), %edx  # Line 6
    andl $1, %edx     # Line 7
    movw %dx, (%rcx)  # Line 8
    addl $1, %eax     # Line 9
    jmp .L2           # Line 10
.L1:
    retq             # Line 11
```

(A) What variable type would %rdi be in the corresponding C program? [4 pt]

(B) Briefly describe why Line 4 is needed before Line 5. [4 pt]

(C) This function uses a for loop. Fill in the corresponding parts below, using register names as variable names. None should be blank. [8 pt]

```
for ( __________ ; __________ ; __________ )
```

(D) If we call this function with the value 3 as the second argument, what value is returned? [4 pt]

(E) Describe at a high level what you think this function accomplishes (not line-by-line). [4 pt]
Wi18 Midterm Q3

Question 3: C Programming & x86-64 Assembly [20 pts.]

Consider the following x86-64 assembly and (mostly blank) C code. The C code is in a file called foo.c and contains a main function and a mystery function, foo. The function foo takes one input and returns a single value. Fill in the missing C code that is equivalent to the x86-64 assembly for the function foo. You can use the names of registers (without the %) for C variables. [18 pts.]

Hint: the function foo contains a for loop. There are more blank lines in the C Code than should be required for your solution.

Follow up: Assume the code in main is correct and has no errors. However, the provided x86-64 code for function foo has a single correctness error. What is the error, and when might this error cause a problem with the execution of foo? Answer in one or two short English sentences. [2 pts.]
Su18 Midterm Q5

Question 5: Procedures & The Stack  [20 pts]

The recursive power function power() calculates base^pow and its x86-64 disassembly is shown below:

```c
int power(int base, unsigned int pow) {
    if (pow) {
        return base * power(base,pow-1);
    }
    return 1;
}
```

00000000004005a0 <power>:

- 4005a0:  85 f6          testl  %esi,%esi
- 4005a2:  74 10          je  4005b4 <power+0x14>
- 4005a4:  53              pushq  %rbx
- 4005a5:  89 fb          movl  %edi,%ebx
- 4005a7:  83 ee 01        subl  $0x1,%esi
- 4005aa:  e8 f1 ff ff ff  call  4005a0 <power>
- 4005af:  0f af c3        imull  %ebx,%eax
- 4005b2:  eb 06          jmp  4005ba <power+0x1a>
- 4005b4:  b8 01 00 00 00  movl  $0x1,%eax
- 4005b9:  c3              ret
- 4005ba:  5b              popq  %rbx
- 4005bb:  c3              ret

(A) How much space (in bytes) does this function take up in our final executable?  [2 pt]

(B) Circle one: The label power will show up in which table(s) in the object file?  [4 pt]

   Symbol Table    Relocation Table    Both Tables    Neither Table

(C) Which register is being saved on the stack?  [2 pt]
(D) What is the return address to `power` that gets stored on the stack? Answer in hex. [2 pt]

(E) Assume `main` calls `power(8,3)`. Fill in the snapshot of memory below the top of the stack in hex as this call to `power` returns to `main`. For unknown words, write “unknown”. [6 pt]

(F) Harry the Husky claims that we could have gotten away with not pushing a register onto the stack in `power`. Is our intrepid school’s mascot correct or not? Briefly explain. [4 pt]
5. Stack Discipline (15 points)

Examine the following recursive function:

```c
long sunny(long a, long *b) {
    long temp;
    if (a < 1) {
        return *b - 8;
    } else {
        temp = a - 1;
        return temp + sunny(temp - 2, &temp);
    }
}
```

Here is the x86_64 assembly for the same function:

```
0000000000400536 <sunny>:
400536:   test   %rdi,%rdi
400539:   jg  400543 <sunny+0xd>
40053b:   mov   (%rsi),%rax
40053e:   sub $0x8,%rax
400542:   retq
400543:   push   %rbx
400544:   sub $0x10,%rsp
400548:   lea   -0x1(%rdi),%rbx
40054c:   mov   %rbx,0x8(%rsp)
400551:   sub $0x3,%rdi
400555:   lea   0x8(%rsp),%rsi
40055a:   callq  400536 <sunny>
40055f:   add   %rbx,%rax
400562:   add $0x10,%rsp
400566:   pop   %rbx
400567:   retq
```

We call `sunny` from `main()`, with registers `%rsi = 0x7ff...ffad8` and `%rdi = 6`. The value stored at address `0x7ff...ffad8` is the long value 32 (0x20). We set a breakpoint at “return *b - 8” (i.e. we are just about to return from `sunny()` without making another recursive call). We have executed the `sub` instruction at `40053e` but have not yet executed the `retq`.

Fill in the register values on the next page and draw what the stack will look like when the program hits that breakpoint. Give both a description of the item stored at that location and the value stored at that location. If a location on the stack is not used, write “unused” in the Description for that address and put “-----” for its Value. You may list the Values in hex or decimal. Unless preceded by 0x we will assume decimal. It is fine to use f…f for sequences of f’s as shown above for `%rsi`. Add more rows to the table as needed. Also, fill in the box on the next page to include the value this call to `sunny` will finally return to `main`. 
<table>
<thead>
<tr>
<th>Register</th>
<th>Original Value</th>
<th>Value at Breakpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsp</td>
<td>0x7fffffff...ffad0</td>
<td></td>
</tr>
<tr>
<td>rdi</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>rsi</td>
<td>0x7fffffff...ffad8</td>
<td></td>
</tr>
<tr>
<td>rbx</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>rax</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

What value is **finally** returned to `main` by this call?

<table>
<thead>
<tr>
<th>Memory address on stack</th>
<th>Name/description of item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffffffffffad8</td>
<td>Local var in <code>main</code></td>
<td>0x20</td>
</tr>
<tr>
<td>0x7fffffffffffffffad0</td>
<td>Return address back to <code>main</code></td>
<td>0x400827</td>
</tr>
<tr>
<td>0x7fffffffffffffffac8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffffac0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffffab8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffffab0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffffaa8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffffaa0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffff98</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffff98</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffff88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffff80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffff78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffff70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffff68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffffffffff60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>