Au16 Midterm Q1

Question 1: Number Representation [12 pts]

(A) What is the value of the char 0b 1101 1101 in decimal? [1 pt]
If x = 0xDD, -x = 0x23 = 2^5+3 = 35
Also accepted unsigned: 0xDD = (16+1)*13 = 221

(B) What is the value of char z = (0xB << 7) in decimal? [1 pt]
0xB << 7 = 0b 1000 0000 = TMin_char = -128
Also accepted unsigned: 0x80 = 128

(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]
x<0, so need large enough (in magnitude) neg num for signed overflow. Unsigned overflow comes naturally along with this. 0x80 ≤ y ≤ 0xBF

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]
Bias = 2^4-1 - 1 = 7

(E) Translate the floating point number 0b 1100 1110 into decimal. [3 pt]
S = 1, E = 10012, M = 1102. Notice that E indicates this is not a special case.
Exp = 9 - 7 = 2, Man = 1.1102.
(-1)^1 × 1.110_2 × 2^2 = −111_2 = −7.

(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]
Look for number such that the 2^0=1 bit is just off the end of the mantissa.
So of the form 1.0001 × 2^Exp, with the underlined bit being 2^0.
Counting to the left, we find that Exp = 4, and 1.0001 × 2^4 = 17.
Sp15 Midterm Q1
1 Number Representation (10 points)

Let \( x = 0xE \) 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?

In hex: \( 0xE + 0x7 = 0x15 \rightarrow 0x5 \)
In binary converted back to hex: \( 0xE + 0x7 = 1110 + 0111 = 10101 \rightarrow 0101 = 0x5 \)
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: \( 0xE + 0x7 = 14 + 7 = 21 \% 16 = 5 \)
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: \( 0xE - 0x7 = -2 - 7 = -9 \rightarrow 7 \)
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**
**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 0000 0018</td>
</tr>
<tr>
<td>*x</td>
<td>int</td>
<td>0x ffdc abac</td>
</tr>
<tr>
<td>x+3</td>
<td>int *</td>
<td>0x 0000 0000 0000 0000 000c</td>
</tr>
<tr>
<td>*(y-1)</td>
<td>long</td>
<td>0x ff42 5a32 aeac adde</td>
</tr>
<tr>
<td>z[3]</td>
<td>unsigned short</td>
<td>0x 3322</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 0000 0008</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x 0000 0000 0000 0000 0018</td>
</tr>
<tr>
<td>%cl</td>
<td>0x bb</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x 0000 0000 0000 0000 0088</td>
</tr>
<tr>
<td>%r8d</td>
<td>0x 0000 1722</td>
</tr>
</tbody>
</table>
Sp17 Midterm Q4
4. 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:

\[
\begin{array}{cccccccc}
\text{Address} & +0 & +1 & +2 & +3 & +4 & +5 & +6 & +7 \\
0x00 & AB & EE & 1E & AC & D5 & 8E & 10 & E7 \\
0x08 & F7 & 84 & 32 & 2D & A5 & F2 & 3A & CA \\
0x10 & 83 & 14 & 53 & B9 & 70 & 03 & F4 & 31 \\
0x18 & 01 & 20 & FE & 34 & 46 & E4 & FC & 52 \\
0x20 & 4C & A8 & B5 & C3 & D0 & ED & 53 & 17 \\
\end{array}
\]

\[\text{int* ip} = 0x00;\]
\[\text{short* sp} = 0x20;\]
\[\text{long* yp} = 0x10;\]

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>long*</td>
<td>0x20</td>
</tr>
<tr>
<td>*(sp - 1)</td>
<td>short</td>
<td>0x52FC</td>
</tr>
<tr>
<td>ip[5]</td>
<td>int</td>
<td>0x31F40370</td>
</tr>
<tr>
<td>&amp;ip</td>
<td>int**</td>
<td>UNKNOWN</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.

\[\text{Remember to give enough hex digits to fill up the width of the register name listed.}\]

<table>
<thead>
<tr>
<th>Register</th>
<th>Value (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x0000 0000 0000 0000 0004</td>
</tr>
<tr>
<td>%ebx</td>
<td>0x84f7 e710</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x0000 0000c</td>
</tr>
<tr>
<td>%rdi</td>
<td>0x0000 0000 ffff fff7</td>
</tr>
<tr>
<td>%si</td>
<td>0x7B09</td>
</tr>
</tbody>
</table>
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.

Many different answers were accepted for these questions, including some not listed here.

(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]

- Naming of first 8 registers (%rax, etc.) comes from IA32.
- Any 32-bit result stored in a register will zero-out the upper 32-bits (so IA32 programs run correctly on 64-bit machines).
- The “word” instruction suffix in x86-64 (e.g. mov_w) still refers to 16 bits.
- Use of CISC design philosophy: keeps old instructions in newer instruction sets.

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

- For the same word size, your address space will be 4 times larger now.
- For same address space, addresses could be 2 bits shorter now.
- Difficult to access data for small datatypes in memory (alternatively, much more padding needed when storing small datatypes).
- Might not be able to use b and w assembly instruction suffixes when accessing memory.

(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:
- Only need 1 stack pointer and 1 return value.
- Still have both callee-saved and caller-saved registers.
- Keep the names of the existing 16 registers.

One thing that should change:
- Probably increase the number of argument registers.
- Anything related to defining which of the new registers are callee-saved or caller-saved was given credit.
6. (7 points) (Instruction-Set Architecture Design) Suppose we decide to change x86-64 to have 100 registers instead of 16. Give one-word answers to the following questions.

(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)?

Solution:
(a) harder
(b) easier
(c) more
(d) more
(e) yes
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]

%rcx is calculated from %rdi with scale 2 (Line 5) and then dereferenced with a `movzwl` instruction (Line 6).

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

Memory operands (Line 5) must take 64-bit register names, since addresses are 8 bytes wide. So the 4-byte value in %eax, must be extended to 8 bytes beforehand.

(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 ( _eax = 0___ ; _eax < esi__ ; _eax++__ )
```

Init is from Line 1, Test is from Lines 2-3, Update is from Line 9.

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

Return value is %rax and we exit the loop when %eax = %esi. **3**

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

Overrides an array of shorts with the parity of the entries (1 for odd, 0 for even – given by the least significant bit).
**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.

<table>
<thead>
<tr>
<th>x86-64 Assembly: function <code>foo</code></th>
<th>C Code: file <code>foo.c</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.text</code></td>
<td><code>#include &lt;stdio.h&gt; // for printf</code></td>
</tr>
<tr>
<td><code>.globl foo</code></td>
<td><code>long foo(int x) {</code></td>
</tr>
<tr>
<td><code>.type foo, @function</code></td>
<td><code>  long sum;</code></td>
</tr>
<tr>
<td><code>foo:</code></td>
<td><code>  for (int i = x; i &gt; 0; i = i-3) {</code></td>
</tr>
<tr>
<td><code>  jmp .L2</code></td>
<td><code>    if (i &amp; 0x1) {</code></td>
</tr>
<tr>
<td><code>.L4:</code></td>
<td><code>      sum += i;</code></td>
</tr>
<tr>
<td><code>  testb $1, %dil</code></td>
<td><code>    }</code></td>
</tr>
<tr>
<td><code>  je .L3</code></td>
<td><code>  return sum;</code></td>
</tr>
<tr>
<td><code>  movslq %edi, %rdx</code></td>
<td><code>} </code></td>
</tr>
<tr>
<td><code>  addq %rdx, %rax</code></td>
<td><code>Note: variable names may be different in students’ answers (e.g., use rax instead of sum).</code></td>
</tr>
<tr>
<td><code>.L3:</code></td>
<td><code>int main(int argc, char **argv) {</code></td>
</tr>
<tr>
<td><code>  subl $3, %edi</code></td>
<td><code>  long r = foo(10);</code></td>
</tr>
<tr>
<td><code>.L2:</code></td>
<td><code>  printf(&quot;r: %ld\n&quot;, r);</code></td>
</tr>
<tr>
<td><code>  testl %edi, %edi</code></td>
<td><code>  return 0;</code></td>
</tr>
<tr>
<td><code>  jg .L4</code></td>
<td><code>}</code></td>
</tr>
<tr>
<td><code>  ret</code></td>
<td></td>
</tr>
</tbody>
</table>

**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.]

The variable “sum” (or the variable we return from `foo`) is never initialized. Thus, it will hold a random value prior to the loop, and the execution of `foo` will always be incorrect (unless the variable happens to have the value 0 prior to loop execution).
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;
}
```

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

Count all bytes (middle columns) or subtract address of next instruction (0x4005bc) from 0x4005a0.  

28 B

(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

power is called in this file (recursively) and can be called by external files, so in both.

(C) Which register is being saved on the stack? [2 pt]

See `pushq` instruction (0x4005a4).

%rbx
(D)  What is the return address to \texttt{power} that gets stored on the stack?  Answer in hex.  [2 pt]

\textbf{The address of the instruction after call.}

\begin{center}
\begin{tabular}{|c|c|}
\hline
0x7fffeca3f748 & \texttt{<ret addr to main>} \tabularnewline
0x7fffeca3f740 & \texttt{<original rbx>} \tabularnewline
0x7fffeca3f738 & \texttt{0x4005af <ret addr>} \tabularnewline
0x7fffeca3f730 & \texttt{0x8 <base>} \tabularnewline
0x7fffeca3f728 & \texttt{0x4005af <ret addr>} \tabularnewline
0x7fffeca3f720 & \texttt{0x8 <base>} \tabularnewline
0x7fffeca3f718 & \texttt{0x4005af <ret addr>} \tabularnewline
0x7fffeca3f710 & \texttt{unknown} \tabularnewline
\hline
\end{tabular}
\end{center}

\textbf{power(8,3)} \underline{power(8,2)} \underline{power(8,1)} \underline{power(8,0)}

The base case doesn’t push \texttt{%rbx} onto the stack, so 0x7fffeca3f710 remains unknown.

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

Harry the Husky claims that we could have gotten away with not pushing a register onto the stack in \texttt{power}.  Is our intrepid school’s mascot correct or not?  Briefly explain.  [4 pt]

\textbf{Harry is correct!  base doesn’t change between recursive calls and power doesn’t call other procedures, so there is no need to save %rdi in %rbx.}

In fact, if you compile the C function with an optimization flag of -O2, it doesn’t push %rbx onto the stack!
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>0x7ff...ffad0</td>
<td>0x7ff...ffa90</td>
</tr>
<tr>
<td>rdi</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>rsi</td>
<td>0x7ff...ffad8</td>
<td>0x7ff...ffaa0</td>
</tr>
<tr>
<td>rbx</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>rax</td>
<td>5</td>
<td>-6</td>
</tr>
</tbody>
</table>

What value is **finally** returned to **main** by this call? **1**

<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 <strong>main</strong></td>
<td>0x20</td>
</tr>
<tr>
<td>0x7fffffffffffffffad0</td>
<td>Return address back to <strong>main</strong></td>
<td>0x400827</td>
</tr>
<tr>
<td>0x7fffffffffffffffac8</td>
<td>Saved <code>%rbx</code></td>
<td>4</td>
</tr>
<tr>
<td>0x7fffffffffffffffac0</td>
<td><code>temp</code></td>
<td>5</td>
</tr>
<tr>
<td>0x7fffffffffffffffab8</td>
<td><code>Unused</code></td>
<td>--------------</td>
</tr>
<tr>
<td>0x7fffffffffffffffab0</td>
<td>Return address to <strong>sunny</strong></td>
<td>0x40055f</td>
</tr>
<tr>
<td>0x7fffffffffffffffaa8</td>
<td>Saved <code>%rbx</code></td>
<td>5</td>
</tr>
<tr>
<td>0x7fffffffffffffffaa0</td>
<td><code>temp</code></td>
<td>2</td>
</tr>
<tr>
<td>0x7fffffffffffffff98</td>
<td><code>Unused</code></td>
<td>--------------</td>
</tr>
<tr>
<td>0x7fffffffffffffff90</td>
<td>Return address to <strong>sunny</strong></td>
<td>0x40055f</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>