University of Washington – Computer Science & Engineering
Autumn 2018     Instructor: Justin Hsia     2018-10-29

CSE351 MIDTERM

Last Name: ____________________________
First Name: ___________________________
Student ID Number: ____________________

Name of person to your Left | Right
____________________________________________________________________________________

All work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CSE351 who haven’t taken it yet. Violation of these terms could result in a failing grade. (please sign)

Do not turn the page until 5:10.

Instructions

- This exam contains 5 pages, including this cover page. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- The last page is a reference sheet. Please detach it from the rest of the exam.
- The exam is closed book (no laptops, tablets, wearable devices, or calculators). You are allowed one page (US letter, double-sided) of handwritten notes.
- Please silence and put away all cell phones and other mobile or noise-making devices. Remove all hats, headphones, and watches.
- You have 70 minutes to complete this exam.

Advice

- Read questions carefully before starting. Skip questions that are taking a long time.
- Read all questions first and start where you feel the most confident.
- Relax. You are here to learn.

<table>
<thead>
<tr>
<th>Question</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possible Points</td>
<td>20</td>
<td>20</td>
<td>12</td>
<td>24</td>
<td>24</td>
<td>100</td>
</tr>
</tbody>
</table>
Question 1: Number Representation  [20 pts]

(A) What is the value of the signed char \texttt{0b 1000 0100} in decimal? [2 pt]

(B) If \( a = 0x2C \), complete the bitwise C statement so that \( b = 0x1F \). [4 pt]

\[
\begin{align*}
b &= a \quad 0x__ \\
\end{align*}
\]

(C) Find the smallest 8-bit numeral \( c \) (answer in hex) such that \( c + 0x71 \) causes signed overflow, but NOT unsigned overflow in 8 bits. [4 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 7 bits split into the following fields:

- Sign (1)
- Exponent (3)
- Mantissa (3)

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

(E) What is the decimal value encoded by \texttt{0b1110101} in this representation? [4 pt]

(F) What value will be read after we try to store -18 in this representation? (Circle one) [4 pt]

\[
\begin{align*}
-16 & \quad -\text{NaN} & \quad -\infty & \quad -18
\end{align*}
\]
Question 2: Pointers & Memory  [20 pts]

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

```
<table>
<thead>
<tr>
<th>Word Addr</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>AC</td>
<td>AB</td>
<td>03</td>
<td>01</td>
<td>BA</td>
<td>5E</td>
<td>BA</td>
<td>11</td>
</tr>
<tr>
<td>0x08</td>
<td>5E</td>
<td>00</td>
<td>68</td>
<td>0C</td>
<td>BE</td>
<td>A7</td>
<td>CE</td>
<td>FA</td>
</tr>
<tr>
<td>0x10</td>
<td>1D</td>
<td>B0</td>
<td>99</td>
<td>DE</td>
<td>AD</td>
<td>60</td>
<td>BB</td>
<td>40</td>
</tr>
<tr>
<td>0x18</td>
<td>14</td>
<td>1D</td>
<td>EC</td>
<td>AF</td>
<td>EE</td>
<td>FF</td>
<td>CO</td>
<td>70</td>
</tr>
<tr>
<td>0x20</td>
<td>BA</td>
<td>B0</td>
<td>41</td>
<td>20</td>
<td>80</td>
<td>DD</td>
<td>BE</td>
<td>EF</td>
</tr>
</tbody>
</table>
```

char* charP = 0x1B
short* shortP = 0xE

(A) Using the values shown above, complete the C code below to fulfill the behaviors described in the comments using pointer arithmetic. [8 pt]

```
char v1 = charP[________];     // set v1 = 0xEC
int* v2 = ((int*)shortP) + ______;     // set v2 = 0x1A
```

(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. Remember to use the appropriate bit widths. [12 pt]

```
<table>
<thead>
<tr>
<th>Register</th>
<th>Data (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x 0000 0000 0000 0000 C</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x 0000 0000 0000 0008</td>
</tr>
</tbody>
</table>
leaw       | (%rsi,2), %r15w |
| %r15w     | 0x |
movswl     | (%rdi,%rsi), %ebp |
| %rbp      | 0x |
addb       | 5(%rdi), %dil |
| %dil      | 0x |
```
Question 3: Design Questions  [12 pts]

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

(A) What values can \( S \) take in an x86-64 memory operand? Briefly describe why these choices are useful/important. [4 pt]

<table>
<thead>
<tr>
<th>Values:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Importance:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

(B) Until very recently (Java 8/9), Java did not support unsigned integer data types. Name one advantage and one disadvantage to this decision to omit unsigned. [4 pt]

<table>
<thead>
<tr>
<th>Advantage:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disadvantage:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

(C) Condition codes are part of the processor/CPU state. Would our instruction set architecture (ISA) still work if we got rid of the condition codes? Briefly explain. [4 pt]

Circle one: Yes No

Explanation:
Question 4: C & Assembly  [24 pts]

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

```
mystery:
  movl $0, %eax # Line 1
  jmp .L2 # Line 2
.L3:  movslq %eax, %rdx # Line 3
  leaq (%rdi,%rdx,8), %rcx # Line 4
  movq (%rcx), %rdx # Line 5
  xorq $-1, %rdx # Line 6
  addq $1, %rdx # Line 7
  movq %rdx, (%rcx) # Line 8
  addl $2, %eax # Line 9
.L2:  movzwl %si, %edx # Line 10
  cmpl %eax, %edx # Line 11
  jg .L3 # Line 12
  retq # Line 13
```

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

(B) What variable type would the 2nd argument be in the corresponding C program? [4 pt]

(C) This function uses a for loop. Fill in the corresponding parts below, using register names as variable names (no declarations necessary). None should be blank. [8 pt]

```
  for ( ___________ ; ___________ ; ___________ )
```

(D) If we call this function with the value 1 as the second argument, how many jump instructions are executed (taken or untaken) in this function? [4 pt]

(E) Describe at a high level what you think this function accomplishes (not line-by-line). [4 pt]
Question 5: Procedures & The Stack  [24 pts]

The recursive function `sum_r()` calculates the sum of the elements of an `int` array and its x86-64 disassembly is shown below:

```c
int sum_r(int *ar, unsigned int len) {
    if (!len) {
        return 0;
    } else
        return *ar + sum_r(ar+1, len-1);
}
```

```
 0000000000400507 <sum_r>:
 400507:  41 53    pushq  %r12
 400509:  85 f6    testl  %esi,%esi
 40050b:  75 07    jne   400514 <sum_r+0xd>
 40050d:  b8 00 00 00 00 movl   $0x0,%eax
 400512:  eb 12    jmp   400526 <sum_r+0x1f>
 400514:  44 8b 1f  movl   (%rdi),%r12d
 400517:  83 ee 01  subl   $0x1,%esi
 40051a:  48 83 c7 04  addq   $0x4,%rdi
 40051e:  e8 e4 ff ff ff callq  400507 <sum_r>
 400523:  44 01 d8  addl   %r12d,%eax
 400526:  41 5b    popq   %r12
 400528:  c3    retq
```

(A) The addresses shown in the disassembly are all part of which section of memory?  [2 pt]

(B) Disassembly (as shown here) is different from assembly (as would be found in an assembly file). Name two major differences:  [4 pt]

Difference 1:

Difference 2:
(C) What is the return address to \texttt{sum\_r} that gets stored on the stack? Answer in hex. [2 pt]

0x

(D) What value is saved across each recursive call? Answer using a \textit{C expression}. [2 pt]

(E) Assume \texttt{main} calls \texttt{sum\_r}(\texttt{ar},3) with \texttt{int \ ar[]} = \{3, 5, 1\}. Fill in the snapshot of memory below the top of the stack \textbf{in hex} as this call to \texttt{sum\_r} returns to \texttt{main}. For unknown words, write “0x unknown”. [6 pt]

\begin{tabular}{|c|c|}
\hline
0x7fffffffde20 & <ret addr to main> \\
0x7fffffffde18 & <original r12> \\
0x7fffffffde10 & 0x \\
0x7fffffffde08 & 0x \\
0x7fffffffde00 & 0x \\
0x7fffffffddf8 & 0x \\
0x7fffffffddf0 & 0x \\
0x7fffffffddfde8 & 0x \\
\hline
\end{tabular}

(F) Assembly code sometimes uses \textit{relative addressing}. The last 4 bytes of the \texttt{callq} instruction encode an integer (in \textit{little endian}). This value represents the difference between which two addresses? \textbf{Hint}: both addresses are important to this \texttt{callq}. [4 pt]

\begin{tabular}{|c|c|}
\hline
value (decimal): & \\
address 1: & 0x \\
address 2: & 0x \\
\hline
\end{tabular}

(G) What could we change in the assembly code of this function to \textbf{reduce the amount of Stack memory used} while keeping it \textit{recursive} and \textit{functioning properly}? [4 pt]
THIS PAGE PURPOSELY LEFT BLANK
CSE 351 Reference Sheet (Midterm)

Binary | Decimal | Hex  
---|---|---
0000 | 0 | 0  
0001 | 1 | 1  
0010 | 2 | 2  
0011 | 3 | 3  
0100 | 4 | 4  
0101 | 5 | 5  
0110 | 6 | 6  
0111 | 7 | 7  
1000 | 8 | 8  
1001 | 9 | 9  
1010 | 10 | A  
1011 | 11 | B  
1100 | 12 | C  
1101 | 13 | D  
1110 | 14 | E  
1111 | 15 | F  

<table>
<thead>
<tr>
<th>2^0</th>
<th>2^1</th>
<th>2^2</th>
<th>2^3</th>
<th>2^4</th>
<th>2^5</th>
<th>2^6</th>
<th>2^7</th>
<th>2^8</th>
<th>2^9</th>
<th>2^10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1024</td>
</tr>
</tbody>
</table>

**IEEE 754 FLOATING-POINT STANDARD**
Value: \( \pm 1 \times \text{Mantissa} \times 2^\text{Exponent} \)
Bit fields: \((-1)^S \times 1.M \times 2^{(E-\text{bias})}\)
where Single Precision Bias = 127,
Double Precision Bias = 1023.

**IEEE Single Precision and Double Precision Formats:**

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>23 bits</td>
<td>52 bits</td>
</tr>
</tbody>
</table>

**IEEE 754 Symbols**

<table>
<thead>
<tr>
<th>E</th>
<th>M</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>all zeros</td>
<td>all zeros</td>
<td>±0</td>
</tr>
<tr>
<td>all zeros</td>
<td>non-zero</td>
<td>± denorm num</td>
</tr>
<tr>
<td>1 to MAX-1</td>
<td>anything</td>
<td>± norm num</td>
</tr>
<tr>
<td>all ones</td>
<td>all zeros</td>
<td>±∞</td>
</tr>
<tr>
<td>all ones</td>
<td>non-zero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**Assembly Instructions**

- **mov a, b** Copy from a to b.
- **movs a, b** Copy from a to b with sign extension. Needs two width specifiers.
- **movz a, b** Copy from a to b with zero extension. Needs two width specifiers.
- **lea a, b** Compute address and store in b.
  
  *Note: the scaling parameter of memory operands can only be 1, 2, 4, or 8.*
- **push src** Push src onto the stack and decrement stack pointer.
- **pop dst** Pop from the stack into dst and increment stack pointer.
- **call <func>** Push return address onto stack and jump to a procedure.
- **ret** Pop return address and jump there.
- **add a, b** Add from a to b and store in b (and sets flags).
- **sub a, b** Subtract a from b (compute b-a) and store in b (and sets flags).
- **imul a, b** Multiply a and b and store in b (and sets flags).
- **and a, b** Bitwise AND of a and b, store in b (and sets flags).
- **sar a, b** Shift value of b right (arithmetic) by a bits, store in b (and sets flags).
- **shr a, b** Shift value of b right (logical) by a bits, store in b (and sets flags).
- **shl a, b** Shift value of b left by a bits, store in b (and sets flags).
- **cmp a, b** Compare b with a (compute b-a and set condition codes based on result).
- **test a, b** Bitwise AND of a and b and set condition codes based on result.
- **jmp <label>** Unconditional jump to address.
- **j* <label>** Conditional jump based on condition codes (*more on next page*).
- **set* a** Set byte a to 0 or 1 based on condition codes.
### Conditionals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>(op) s, d</th>
<th>test a, b</th>
<th>cmp a, b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>d (op) s == 0</td>
<td>b &amp; a == 0</td>
<td>b == a</td>
</tr>
<tr>
<td>jne</td>
<td>d (op) s != 0</td>
<td>b &amp; a != 0</td>
<td>b != a</td>
</tr>
<tr>
<td>js</td>
<td>d (op) s &lt; 0</td>
<td>b &amp; a &lt; 0</td>
<td>b-a &lt; 0</td>
</tr>
<tr>
<td>jns</td>
<td>d (op) s &gt;= 0</td>
<td>b &amp; a &gt;= 0</td>
<td>b-a &gt;= 0</td>
</tr>
<tr>
<td>jg</td>
<td>d (op) s &gt; 0</td>
<td>b &amp; a &gt; 0</td>
<td>b &gt; a</td>
</tr>
<tr>
<td>jge</td>
<td>d (op) s &gt;= 0</td>
<td>b &amp; a &gt;= 0</td>
<td>b &gt;= a</td>
</tr>
<tr>
<td>jl</td>
<td>d (op) s &lt; 0</td>
<td>b &amp; a &lt; 0</td>
<td>b &lt; a</td>
</tr>
<tr>
<td>jle</td>
<td>d (op) s &lt;= 0</td>
<td>b &amp; a &lt;= 0</td>
<td>b &lt;= a</td>
</tr>
<tr>
<td>ja</td>
<td>d (op) s &gt; 0U</td>
<td>b &amp; a &gt; 0U</td>
<td>b-a &gt; 0U</td>
</tr>
<tr>
<td>jb</td>
<td>d (op) s &lt; 0U</td>
<td>b &amp; a &lt; 0U</td>
<td>b-a &lt; 0U</td>
</tr>
</tbody>
</table>

### Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Convention</th>
<th>Name of “virtual” register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lowest 4 bytes</td>
<td>Lowest 2 bytes</td>
</tr>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%ax</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%cx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%dx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%si</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%di</td>
</tr>
<tr>
<td>%esp</td>
<td>%esp</td>
<td>%sp</td>
</tr>
<tr>
<td>%ebp</td>
<td>%ebp</td>
<td>%bp</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
<td>%r8w</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
<td>%r9w</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
<td>%r10w</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
<td>%r11w</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
<td>%r12w</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
<td>%r13w</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
<td>%r14w</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
<td>%r15w</td>
</tr>
</tbody>
</table>

### Sizes

<table>
<thead>
<tr>
<th>C type</th>
<th>x86-64 suffix</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>q</td>
<td>8</td>
</tr>
</tbody>
</table>