1. **Number Representation** (20 pts)

Consider the binary value \(110101_2\):

(a) Interpreting this value as an **unsigned 6-bit integer**, what is its value in decimal?

\[2^5 + 2^4 + 2^2 + 2^0 = 32 + 16 + 4 + 1 = 53\]

(b) If we instead interpret it as a **signed (two's complement) 6-bit integer**, what would its value be in decimal?

\[-2^5 + 2^4 + 2^2 + 2^0 = -32 + 16 + 4 + 1 = -11\]

*(most significant bit becomes "negatively weighted")*

(c) Assuming these are all signed two's complement 6-bit integers, compute the result (leaving it in binary is fine) of each of the following additions. For each, indicate if it resulted in **overflow**.

**Note:** \(TMIN = -32\)

\[
\begin{array}{llllll}
9 & 001001 & -15 & 110001 & 011001 & 101111 \\
-10 & +110110 & -5 & +111011 & +001100 & +011111 \\
\end{array}
\]

Result:

\[
\begin{array}{cccc}
111111 & +101100 & 100101 & +001110 \\
\end{array}
\]

Overflow?

\[
\begin{array}{cccc}
\text{No} & \text{No} & \text{Yes} & \text{No} \\
\end{array}
\]

*Overflow only occurs for signed addition if the result comes out wrong. The easiest way to determine this is by looking at the signs: if 2 positive values result in a negative result, or 2 negatives result in a positive, then overflow must have occurred.*
Now assume that our fictional machine with 6-bit integers also has a 6-bit IEEE-like floating point type, with 1 bit for the sign, 3 bits for the exponent (\(\text{exp}\)) with a bias of 3, and 2 bits to represent the mantissa (\(\text{frac}\)), not counting implicit bits.

(d) If we reinterpret the bits of our binary value from above as our 6-bit floating point type, what value, in decimal, do we get?

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>exp</td>
<td>frac</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[-1.01_2 \cdot 2^{(4+1-3)} = -1.01_2 \cdot 2^2 = -101_2 = -5\]

(e) If we treat 110101\(^2\) as a signed integer, as we did in (b), and then cast it to a 6-bit floating point value, do we get the correct value in decimal? (That is, can we represent that value in our 6-bit float?) If yes, what is the binary representation? If not, why not? (and in that case you do not need to determine the rounded bit representation)

No, we cannot represent it exactly because there are not enough bits for the mantissa.

To determine this, we have to find out what the mantissa would be once we are in "sign-and-magnitude" style: 110101 \(-11\) \(\rightarrow\) 001011 \(+11\). In normalized form, this would be: \((-1)^1 \cdot 1.011 \cdot 2^3\), which means \(\text{frac}\) would need to be 011, which doesn’t fit in 2 bits.

(f) Assuming the same rules as standard IEEE floating point, what value (in decimal) does the following represent?

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>exp</td>
<td>frac</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0.0 (it is a denormalized case)
Sp15 Midterm Q1 Solutions

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: \( 0x{\text{E}} - 0x{\text{7}} = _i^{-2} - 7 = -9 \rightarrow 7 \)
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:

- 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
Wi19 Midterm Q2 Solutions

Question 2: Pointers

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>BD</td>
<td>28</td>
<td>ED</td>
<td>02</td>
<td>35</td>
<td>72</td>
<td>3A</td>
<td>AF</td>
</tr>
<tr>
<td>0x08</td>
<td>66</td>
<td>6F</td>
<td>B1</td>
<td>E9</td>
<td>00</td>
<td>FF</td>
<td>5D</td>
<td>4D</td>
</tr>
<tr>
<td>0x10</td>
<td>86</td>
<td>06</td>
<td>04</td>
<td>30</td>
<td>64</td>
<td>31</td>
<td>8C</td>
<td>B3</td>
</tr>
<tr>
<td>0x18</td>
<td>63</td>
<td>78</td>
<td>1E</td>
<td>1C</td>
<td>25</td>
<td>34</td>
<td>EE</td>
<td>93</td>
</tr>
<tr>
<td>0x20</td>
<td>42</td>
<td>6C</td>
<td>65</td>
<td>67</td>
<td>DE</td>
<td>AD</td>
<td>BE</td>
<td>EF</td>
</tr>
<tr>
<td>0x28</td>
<td>CA</td>
<td>FE</td>
<td>D0</td>
<td>0D</td>
<td>1E</td>
<td>93</td>
<td>FA</td>
<td>CE</td>
</tr>
</tbody>
</table>

(a) (16 points) Write the value in hexadecimal of each expression within the commented lines at their respective state in the execution of the given program. Write UNKNOWN in the blank if the value cannot be determined.

```c
int main(int argc, char** argv) {
    char *charP;
    short *shortP;
    int *intP = 0x00;
    long *longP = 0x28;

    // The value of intP is: 0x00 00 00 00 00 00 00 00
    // *intP
    0x02 ED 28 BD
    // &intP
    0xUNKNOWN

    // longP[-2]
    0x93 EE 34 25 1C 1E 78 63

    charP = 0x20;
    shortP = (short *) intP;
    intP++;
    longP--;

    // *shortP
    0x28 BD

    // *intP
    0xAF 3A 72 35

    // *((int*) longP)
    0x67 65 6C 42

    // (short*) (((long*) charP) - 2)
    0x10
}
```
Au16 Midterm Q2 Solutions

**Question 2: Pointers & Memory** [12 pts]

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

```plaintext
char* cp = 0x12
short* sp = 0x0C
unsigned* up = 0x2C
```

<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>AB</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>CD</td>
<td>FA</td>
<td>1D</td>
<td>D0</td>
<td>41</td>
<td>ED</td>
<td>77</td>
</tr>
<tr>
<td>0x20</td>
<td>B0</td>
<td>B0</td>
<td>FF</td>
<td>20</td>
<td>80</td>
<td>AA</td>
<td>BE</td>
<td>EF</td>
</tr>
</tbody>
</table>

(A) What are the values (in hex) stored in each register shown after the following x86 instructions are executed? Remember to use the appropriate bit widths. [6 pt]

- `leaw (%rsi, %rdi), %ax`  
  `%ax` = 0x0004

- `movb 8(%rdi), %bl`  
  `%bl` = 0xBE

- `movswl (,%rdi,8), %ecx`  
  `%rcx` = 0x0000 0000 FFFF BOBA

Moveb instruction pulls byte from memory at address 8+4 = 12 = 0x0C.

Movewl instruction pulls 2 bytes from memory starting at addresses 8*4 = 32 = 0x20.

Remember little-endian! Then sign extended to 32 bits, zero out top 32 bits.

(B) It’s a memory scavenger hunt! Complete the C code below to fulfill the behaviors described in the comments using pointer arithmetic. [6 pt]

```c
long v1 = (long) *(cp + 3);  // set v1 = 0x60
unsigned* v2 = up + 5;       // set v2 = 64
int v3 = *(int *)(sp + 1);   // set v3 = 0xB01DFACE
```

- `v1`: Byte 0x60 at address 0x15. 0x15 - cp = 3.
- `v2`: No dereferencing, just pointer arithmetic (scaled by sizeof(unsigned)=4). up = 0x2C = 44. To get to 64, need to add 20 (5 by pointer arithmetic).
- `v3`: The correct bytes can be found (in little-endian order) in addresses 0x0E-0x11. Want (0x0E - sp)/sizeof(short) = 1.
Wi18 Midterm Q5 Solutions

Question 5: Fun Stuff [10 pts.]

(A) Assume we are executing code on a machine that uses $k$-bit addresses, and each addressable memory location stores $b$-bytes. What is the total size of the addressable memory space on this machine? [2 pts.]

\[
(2^k) \cdot b
\]

(B) In C, who/what determines whether local variables are allocated on the stack or stored in registers? Circle your answer. [2 pts.]

- Programmer
- Compiler
- Language (C) Runtime
- Operating System

(C) Assume procedure P calls procedure Q and P stores a value in register %rbp prior to calling Q. True or False: P can safely use the register %rbp after Q returns control to P. Circle your answer. [2 pts.]

- a. True. %rbp is a callee saved register.
- b. False

(D) Assume we are implementing a new CPU that conforms to the x86-64 instruction set architecture (ISA). Answer the following questions, in one or two English sentences, regarding this new CPU. [4 pts.]

a. In modern x86-64 CPUs, a new add operation can be executed every cycle. However, for our new CPU, we realize that we can save power by implementing the add operation such that we can execute a new add only once every three cycles. Is our new CPU still a valid x86-64 implementation?

Yes. The x86-64 architecture/specification says nothing about how fast any operation must execute in hardware.

b. In our new CPU implementation, we decide to change the width of register %rsp to be 48-bits, since most modern x86-64 CPUs only use 48-bit physical addresses, but we still use the name %rsp. Is our CPU still a valid x86-64 implementation?

No. The x86-64 architecture/specification determines the number and size of registers available to the programmer/compiler. Changing this in our implementation violates the architecture.
Au16 Midterm Q3 Solutions

Question 3: Computer Architecture Design  [8 pts]

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

(A) Why can’t we upgrade to more registers like we can with memory?  [2 pt]

Registers are part of the CPU (and the architecture) and are not modular like RAM.

(B) Why don’t we see new assembly instruction sets as frequently as we see new programming languages?  [2 pt]

Hard to implement/get adopted – need to build new hardware. (by comparison, a new programming language only needs a new compiler – software)

(C) Name one reason why a program written in a CISC language might run slower than the same program written in a RISC language and one reason why the reverse might be true: [4 pt]

| CISC slower: Complicated instructions take longer to execute (fewer instructions, but each is slower). | RISC slower: Need more instructions to do complicated computations (faster instructions, but more numerous). |
You are given the following x86-64 assembly function:

```
mystery:
    movl $0, %edx
    movl $0, %eax
    .L3:
        cmpl %esi, %edx
        jge .L1
        movslq %edx, %rcx
        addl (%rdi,%rcx,4), %eax
        addl $1, %edx
        jmp .L3
    .L1:
        rep ret
```

a) (1 pt) What variable type would `%rdi` be in the corresponding C program?

b) (1 pt) What variable type would `%rsi` be in the corresponding C program?

c) (7 pts) Fill in the missing C code that is equivalent to the x86-64 assembly above:

```c
int mystery(int *rdi, int rsi) {
    int eax = 0;
    for (int edx = 0; edx < rsi; edx++) {
        eax += rdi[edx];
    }
    return eax;
}
```

d) (2 pts) In 1 sentence, describe what this function is doing?

Summing the first rsi elements of the int array starting at rdi
Wi15 Midterm Q2 Solutions

2. Assembly and C (20 points)

Consider the following x86-64 assembly and C code:

```assembly
<do_something>:
    cmp $0x0,%rsi
    jle <end>
xor %rax,%rax
    sub $0x1,%rsi

<loop>:
    lea (%rdi,%rsi, 2),%rdx
    add (%rdx),%ax
    sub $0x1,%rsi
    jns <loop>

<end>:
    retq
```

```c
short do_something(short* a, int len) {
    short result = 0;
    for (int i = len - 1; i >= 0 ; i--) {
        result += a[i];
    }
    return result;
}
```

(a) Both code segments are implementations of the unknown function `do_something`. Fill in the missing blanks in both versions. (Hint: %rax and %rdi are used for result and a respectively. %rsi is used for both len and i)

(b) Briefly describe the value that `do_something` returns and how it is computed. Use only variable names from the C version in your answer.

`do_something` returns the sum of the shorts pointed to by `a`. It does so by traversing the array backwards.
Sp14 Midterm Q4 Solutions

4. Stack Discipline (30 points)

The following function recursively computes the greatest common divisor of the integers \( a, b \):

```c
int gcd(int a, int b) {
    if (b == 0) {
        return a;
    } else {
        return gcd(b, a % b);
    }
}
```

Here is the x86_64 assembly for the same function:

```
4006c6 <gcd>:
4006c6:  sub $0x18, %rsp
4006ca:  mov %edi, 0x10(%rsp)
4006ce:  mov %esi, 0x0 8(%rsp)
4006d2:  cmpl $0x0, %esi
4006d7:  jne  4006df <gcd+0x19>
4006d9:  mov 0x 10(%rsp), %eax
4006dd:  jmp  4006f5 <gcd+0x2f>
4006df:  mov 0x 10(%rsp), %eax
4006e3:  cltd
4006e4:  idivl 0x08(%rsp)
4006e8:  mov 0x08(%rsp), %eax
4006ec:  mov %edx, %esi
4006ee:  mov %eax, %edi
4006f0:  callq 4006c6 <gcd>
4006f5:  add $0x18, %rsp
4006f9:  retq
```

Note: \texttt{cltd} is an instruction that sign extends \%eax into \%edx to form the 64-bit signed value represented by the concatenation of \ [%edx | %eax \].

Note: \texttt{idivl \textless mem\textgreater} is an instruction divides the 64-bit value \ [%edx | %eax \] by the long stored at \textless mem\textgreater, storing the quotient in \%eax and the remainder in \%edx.
A. Suppose we call `gcd(144, 64)` from another function (i.e. `main()`), and set a breakpoint just before the statement “return a”. When the program hits that breakpoint, what will the stack look like, starting at the top of the stack and going all the way down to the saved instruction address in `main()`? Label all return addresses as "ret addr", label local variables, and leave all unused space blank.

<table>
<thead>
<tr>
<th>Memory address on stack</th>
<th>Value (8 bytes per line)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffffffffffad0</td>
<td>Return address back to main</td>
</tr>
<tr>
<td>0x7fffffffffffffffac8</td>
<td>1st of 3 local variables on stack (argument <code>a</code> = 144)</td>
</tr>
<tr>
<td>0x7fffffffffffffffac0</td>
<td>2nd of 3 local variables on stack (argument <code>b</code> = 64)</td>
</tr>
<tr>
<td>0x7fffffffffffffffab8</td>
<td>3rd of 3 local variables on stack (unused)</td>
</tr>
<tr>
<td>0x7fffffffffffffffab0</td>
<td>Return address back to <code>gcd(144, 64)</code></td>
</tr>
<tr>
<td>0x7fffffffffffffffaa8</td>
<td>1st of 3 local variables on stack (argument <code>a</code> = 64)</td>
</tr>
<tr>
<td>0x7fffffffffffffffaa0</td>
<td>2nd of 3 local variables on stack (argument <code>b</code> = 16)</td>
</tr>
<tr>
<td>0x7fffffffffffffff98</td>
<td>3rd of 3 local variables on stack (unused)</td>
</tr>
<tr>
<td>0x7fffffffffffffff90</td>
<td>Return address back to <code>gcd(64,16)</code></td>
</tr>
<tr>
<td>0x7fffffffffffffff88</td>
<td>1st of 3 local variables on stack (argument <code>a</code> = 16)</td>
</tr>
<tr>
<td>0x7fffffffffffffff80</td>
<td>2nd of 3 local variables on stack (argument <code>b</code> = 0)</td>
</tr>
<tr>
<td>0x7fffffffffffffff78</td>
<td>3rd of 3 local variables on stack (unused)</td>
</tr>
<tr>
<td>0x7fffffffffffffff70</td>
<td>&lt;-%rsp at “return a&quot; in 3rd recursive call</td>
</tr>
</tbody>
</table>
B. How many total bytes of local stack space are created in each frame (in decimal)?

\[32\] 24 allocated explicitly and 8 for the return address.

C. When the function begins, where are the arguments (a, b) stored?

They are stored in the registers \%rdi and \%rsi, respectively.

D. From a memory-usage perspective, why are iterative algorithms generally preferred over recursive algorithms?

Recursive algorithm continue to grow the stack for the maximum number of recursions which may be hard to estimate.
4. Stack Discipline (30 pts)

Take a look at the following recursive function written in C:

```c
long sum_asc(long * x, long * y) {
    long sum = 0;
    long v = *x;
    if (v >= *y) {
        sum = sum_asc(x + 1, &v);
    }
    sum += v;
    return sum;
}
```

Here is the x86-64 disassembly for the same function:

```
0000000000400536 <sum_asc>:
  0x400536:  pushq  %rbx
  0x400537:  subq   $0x10,%rsp
  0x40053b:  movq   (%rdi),%rbx
  0x40053e:  movq   %rbx,0x8(%rsp)
  0x400543:  movq   $0x0,%rax
  0x400548:  cmpq   (%rsi),%rbx
  0x40054b:  jl     40055b <sum_asc+0x25>
  0x40054d:  addq   $0x8,%rdi
  0x400551:  leaq   0x8(%rsp),%rsi
  0x400556:  callq  400536 <sum_asc>
  0x40055b:  addq   %rbx,%rax
  0x40055e:  addq   $0x10,%rsp
  0x400562:  popq   %rbx
  0x400563:  ret
```

Suppose that `main` has initialized some memory in its stack frame and then called `sum_asc`. We set a breakpoint at "return sum", which will stop execution right before the first return (from the deepest point of recursion). That is, we will have executed the `popq` at 0x400562, but not the `ret`.

(a) **On the next page: Fill in the state of the registers and the contents of the stack (in memory) when the program hits that breakpoint.** For the contents of the stack, give both a description of the item stored at that location as well as the value. 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 (prefixed by 0x) or decimal. Unless preceded by 0x, we will assume decimal. It is fine to use ff... for sequences of f’s, as we do for some of the initial register values. Add more rows to the table as needed. (20 pts)
Name: _______________________________

### Register Values

<table>
<thead>
<tr>
<th>Register</th>
<th>Original Value</th>
<th>Value at Breakpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
<td>0x7ff..070</td>
<td>0x7ff..050</td>
</tr>
<tr>
<td>%rdi</td>
<td>0x7ff..080</td>
<td>0x7ff..088</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x7ff..078</td>
<td>0x7ff..060</td>
</tr>
<tr>
<td>%rbx</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>%rax</td>
<td>42</td>
<td>2</td>
</tr>
</tbody>
</table>

### Memory Address Details

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Description of Item</th>
<th>Value at Breakpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffff090</td>
<td>Initialized in main to: 1</td>
<td>1</td>
</tr>
<tr>
<td>0x7fffffffff088</td>
<td>Initialized in main to: 2</td>
<td>2</td>
</tr>
<tr>
<td>0x7fffffffff080</td>
<td>Initialized in main to: 7</td>
<td>7</td>
</tr>
<tr>
<td>0x7fffffffff078</td>
<td>Initialized in main to: 3</td>
<td>3</td>
</tr>
<tr>
<td>0x7fffffffff070</td>
<td>Return address back to main</td>
<td>0x400594</td>
</tr>
<tr>
<td>0x7fffffffff068</td>
<td>Original %rbx value</td>
<td>2</td>
</tr>
<tr>
<td>0x7fffffffff060</td>
<td>Temporary variable v or %rbx</td>
<td>7</td>
</tr>
<tr>
<td>0x7fffffffff058</td>
<td>Unused</td>
<td>---</td>
</tr>
<tr>
<td>0x7fffffffff050</td>
<td>Return address back to sum_asc</td>
<td>0x40055b</td>
</tr>
<tr>
<td>0x7fffffffff048</td>
<td>Previous value of %rbx (v from first call)</td>
<td>7</td>
</tr>
<tr>
<td>0x7fffffffff040</td>
<td>Temporary variable v or %rbx</td>
<td>2</td>
</tr>
<tr>
<td>0x7fffffffff038</td>
<td>Unused</td>
<td>---</td>
</tr>
<tr>
<td>0x7fffffffff030</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffff028</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffff020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffff018</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffff010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffff008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7fffffffff000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Grading Rubric

** Registers (6 pts) **
- %rsp: (2) (-1 if only missing last pop)
- %rdi: (1)
- %rsi: (1)
- %rbx: (1)
- %rax: (1)

** Stack (14 pts) **
- Generally, 1 pt for each stack frame where correct desc/value appears.
- saved %rbx: desc (2), value (2)
- temp "v"/"rbx": desc (2), value (2)
- unused space: (2) second unused optional
- return address desc (2), value (2)

Additional questions about this problem on the next page.
Continue to refer to the sum_asc code from the previous 2 pages.

(b) What is the purpose of this line of assembly code: 0x40055e: addq $0x10, %rsp? Explain briefly (at a high level) something bad that could happen if we removed it. (5 pts)

This resets the stack pointer to deallocate temporary storage. If we didn't increment here, we wouldn't pop the correct return address or the right value of %rbx.

Note that this would not lead to slow stack overflow due to leaking memory – the first ret would most likely crash because it got the wrong return address; it is highly unlikely that it could continue to execute successfully long enough for this leak to be a problem.

(c) Why does this function push %rbx at 0x400536 and pop %rbx at 0x400562? (5 pts)

The register %rbx is a callee-saved register, so if we use it, it is our responsibility to set it back to what it was before we return from the function.

We gave some points for people recognizing that the two have to be matched for everything else on the stack to work out (similar to the reasoning for deallocation above), but if that were the only reason, then we could have just left both of the instructions out.