**School of ECE, UNIST**
**CSE251 (System Programming)**
**Spring 2019**

Instructor:
Hyungon Moon

Midterm, 21:00 - 23:00 (120min) Apr 17, 2019

**Start after filling out this.**

| | |
|---|---|
| **Your Name (In English):** | |
| **Your Student ID:** | |

**Instruction:**

1. You will be submitting all the sheets that you are given.
2. You can neither ask questions nor leave the room within the first 45 minutes of the exam.
3. Write your student ID at every page, bottom left.
4. Your answers should be printed at the designated locations, and written with an **inerasable black or blue** pen.
5. You are not allowed to go to the rest room and come back.

| Question | Points | Score |
|---|---|---|
| 1 | 20 | |
| 2 | 8 | |
| 3 | 16 | |
| 4 | 20 | |
| 5 | 5 | |
| 6 | 6 | |
| 7 | 4 | |
| 8 | 16 | |
| 9 | 4 | |
| 10 | 12 | |
| 11 | 8 | |
| 12 | 14 | |
| 13 | 18 | |
| 14 | 32 | |
| 15 | 16 | |
| 16 | 26 | |
| Total: | 225 | |

**Enjoy!**

1. [20 points] (2 points each) **Bit-level representations and implicit casting.**

   Print the bit-level representation of the value `x` after each of the expressions being executed. You should also represent the preceeding zeros. For example, if `x` is 2-byte long and the value is zero, you should answer with `0x00`.

   (a) `int x = 0xC0CAF00D * 1024` Your answer: **0x2BC03400**

   (b) `int x = (int)0xFFFFFFFC & (int)0x8` Your answer: **0x00000008**

   (c) `int x = (int)0xCAFEC0C0 >> 16` Your answer: **0xFFFFCAFE**

   (d) `int x = (unsigned int)0xCAFEC0C0 >> 16` Your answer: **0x0000CAFE**

   (e) `int x = ((short)0x1C0FFEE))` Your answer: **0xFFFFFFEE**

   (f) `unsigned char x = ((unsigned int)-1 > -1)` Your answer: **0x00**

   (g) `unsigned char x = ((unsigned int)-1 > 1)` Your answer: **0x01**

   (h) `unsigned char x = ((unsigned int)-10 > -20)` Your answer: **0x01**

   (i) `unsigned int x = !(unsigned int)0xD0D0C0DE` Your answer: **0x00000000**

   (j) `unsigned short x = ~(unsigned short)0xBAD` Your answer: **0xF452**

   **Solution:** Incorrect number of or no preceeding zeros: 1 point.

2. [8 points] (1 point each) **Representing integers.**

   Fill in the blanks in the following table. Write the value in decimal in the **Value** column and the bit-level representation including the preceeding zeros in **Hex** column. **Assume the word size to be 12 bits.**

| Description | **Value** | **Hex** |
|---|---|---|
| $U_{Max}$ (The largest unsigned integer) | **4095** | **0xFFF** |
| $T_{Max}$ (The largest two's complement integer) | **2047** | **0x7FF** |
| $T_{Min}$ (The smallest two's complement integer) | **-2048** | **0x800** |
| $-1$ in two's complement | -1 | **0xFFF** |
| $-5$ in two's complement | -5 | **0xFFB** |

3. **Vulnerability.**

Answer the questions regarding the following code snippet.

```
1  #define BSIZE 1024
2  char public[BSIZE];
3  char secret[BSIZE];
4  void memcpy(void* dest, void* src, size_t len);
5
6  int copy_to_stranger(void* dest, int maxlen) {
7      int len = maxlen;
8      if (KSIZE < len) len = KSIZE;
9      memcpy(dest,public,len);
10     return len;
11 }
```

(a) [8 points] Why is the function `copy_to_stranger` vulnerable? Why there is a chance for a stranger who can call the function to obtain the contents of the buffer `secret`? Explain within 5 sentences.

**Solution: Example**

The function uses the `signed` integer type to make sure that `len` is smaller then `KSIZE` when it calls `memcpy`. Unfortunately, any negative number thus pass the size check, and will be used to call `memcpy`. As `memcpy` considers the `len` as `unsigned`, any negative number given to the function will be considered as a large positive number, resulting in copying the memory beyond whatever is pointed by `src`.

(b) [8 points] Write a code snippet that triggers the vulnerable behavior of the function. Your code snippet should be shorter than 7 lines. Also explain how in 3 sentences.

**Solution: Example**

```
1  char malbuf[1024];
2  copy_to_stranger(malbuf,-1024);
```

The first line declares a buffer to keep the result and the second line calls the vulnerable function with a negative integer as the length argument. `copy_to_stranger` will end up dumping beyond the `public`.

- Code: 4 points.
- Explaination: 4 points.

**Remark (not part of the expected solution)** Note that, the call will (highly likely) end up crashing the calling program, so we will need a way to obtain the result before then.

4. [20 points] **Floating point.**

Implement the following function.

```
/*
 * floatIsLess - Compute f < g for floating point arguments f and g.
 *   Both the arguments are passed as unsigned int's, but
 *   they are to be interpreted as the bit-level representations of
 *   single-precision floating point values.
 *   If either argument is NaN, return 0.
 *   +0 and -0 are considered equal.
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 30
 *   Rating: 3
 */

int floatIsLess(unsigned uf, unsigned ug) {
  return 0;
}
```

Your answer:

**Solution:** This is just an example.

```
int floatIsLess(unsigned uf, unsigned ug) {
  int uf_a = (uf>>31)&0x1 , ug_a = (ug>>31)&0x1;
  int uf_b = (uf>>23)&0xff , ug_b = (ug>>23)&0xff;
  int uf_c = uf&0x007fffff, ug_c = ug&0x007fffff;
  if(!(uf&0x7fffffff) && !(ug&0x7fffffff)) return 0;
  if((ug_b==0xff&&ug_c) || (uf_b==0xff&&uf_c)) return 0;
  if(uf_a^ug_a) return uf_a>ug_a;
  if(uf_b^ug_b) return (uf_b<ug_b)^(uf_a);
  if(uf_c^ug_c) return (uf_c<ug_c)^(uf_a);
  return 0;
}
```

Criteria:

1. Correctly handling when either one of them is $NaN$: 4 points.
2. Correctly handling when either one of them is denormalized: 4 points.
3. Correctly handling when they have different sign bits: 4 points.
4. Correctly handling when the sign bit matches but the exponents differ: 4points.
5. Correctly handling when they have the same sign bit and the exponent: 4points.
6. Minor syntax error: 0.1 point each.

5. [5 points] **Processor model.**

   A processor has a set of general-purpose registers like `%rsp, %rax`. `x86_64` has 16 of them and some others have about 32. What prevents us from enlarging it to have much more, say, 256? Answer within 2 sentences.

   **Solution: Example**

   The larger the register file is, the slower they become.

6. [6 points] **Disks.**

   Explain why random accesses are slow (long latency) on disks and we should optimize our software to read to/write from it in the unit of large block (sequential access). Use the terms *seek time ($T_s$)*, *rotational latency ($T_r$)*, and *transfer time ($T_t$)*.

   **Solution: Example**

   Each random access of a byte on a disk takes $T_s + T_r + T_t$, where the formal two are dominant (orders of magnitude slower than the latter). When accessing more than one ($n$) bytes from the same block, the total latency becomes roughly $T_s + n \times (T_r + T_t)$, which makes the average latency shorter.

   Criteria: 3 points for each part.

7. [4 points] **Trick.**

   Assume that `%rax` has the value of a variable `x` and `%rcx` has of `y` at a moment. Write a single instruction evaluates the following expression and store the result (`z`) in `%rdi`.

   ```
   z = y + x * 4 + 0x4000
   ```

   **Solution:** `leaq 0x4000(%rcx,%rax,4)`

8. [16 points] (2 points each) **Translation of loops.**

These are an `x86_64` assembly code and its C source function. Fill in the eight blanks. Assume the calling convention to be: `%edi` is the first argument and `%esi` is the second argument.

```
0000000000000000 <uni>:
   0:        85 f6                        test   %esi,%esi
   2:        74 43                        je     47 <uni+0x47>
   4:        41 89 fa                     mov    %edi,%r10d
   7:        44 8d 47 08                  lea    0x8(%rdi),%r8d
   b:        b8 09 00 00 00               mov    $0x9,%eax
  10:        41 b9 00 00 00 00            mov    $0x0,%r9d
  16:        44 89 c1                     mov    %r8d,%ecx
  19:        45 85 c0                     test   %r8d,%r8d
  1c:        7e 1a                        jle    38 <uni+0x38>
  1e:        01 f0                        add    %esi,%eax
  20:        39 c7                        cmp    %eax,%edi
  22:        7f 12                        jg     36 <uni+0x36>
  24:        ba 00 00 00 00               mov    $0x0,%edx
  29:        83 c2 03                     add    $0x3,%edx
  2c:        39 d1                        cmp    %edx,%ecx
  2e:        7e 08                        jle    38 <uni+0x38>
  30:        01 f0                        add    %esi,%eax
  32:        39 c7                        cmp    %eax,%edi
  34:        7e f3                        jle    29 <uni+0x29>
  36:        f3 c3                        repz retq
  38:        41 83 c1 05                  add    $0x5,%r9d
  3c:        45 01 d0                     add    %r10d,%r8d
  3f:        44 39 ce                     cmp    %r9d,%esi
  42:        75 d2                        jne    16 <uni+0x16>
  44:        89 c8                        mov    %ecx,%eax
  46:        c3                           retq
  47:        b8 08 00 00 00               mov    $0x8,%eax
  4c:        eb e8                        jmp    36 <uni+0x36>
```

```
int uni(int a, int b) {
    int i,j,k,m;
    i = 0; j = 1; k = 8; m = 9;
    while(___1___) {
        ___2___
        for(___3___; ___4___; ___5___) {
            ___6___
            ___7___
        }
        ___8___
    }
    return k;
}
```

**Please write your answer in the table on the next page.**

**Your answers for the question 8 here:**

| | | | |
|---|---|---|---|
| 1 | `i != b` | 5 | `j += 3` |
| 2 | `k += a;` | 6 | `m += b;` |
| 3 | `j = 0` | 7 | `if(m < a) return m;` |
| 4 | `j < k` | 8 | `i += 5` |

9. [4 points] **Conditional move.**

Why we should not to translate the following expression using conditional move?

```
int a = (p)? *p : 0;
```

**Solution:** To use `cmov`, we should evaluate both the part (`*p` and `0`) but the evaluation of the former causes a memory error. That makes the meaning of the generated code with the one of the source code, which do not evaluate `*p` if `p` is a null pointer.

10. [12 points] (2 points each) **Switch.**

These are a function using a `switch` statement and its translation with a jump table. Please fill in each the blanks either with an instruction or an immediate value.

```c
int pick(int k, int* a, unsigned int len) {
    int ret = 0;
    switch(k) {
        case 0:
            ret = a[len-1];
            break;
        case 5:
            ret = a[1];
        case 6:
            ret -= a[2];
        case 4:
        case 2:
            ret += a[3];
            break;
        default:
            ret = a[1] + a[2];
            break;
    }
    return ret;
}
```

```
0000000000400497 <pick>:
  400497:       00          cmp    $0x6,%edi
  40049a:       00          ja     4004c4 <pick+0x2d>
  40049c:       00          mov    %edi,%edi
  40049e:       00          ___1___
  4004a5:       00          lea    -0x1(%rdx),%eax
  4004a8:       00          ___2___
  4004ab:       00          retq
  4004ac:       00          mov    0x4(%rsi),%eax
  4004af:       00          jmp    ___3___
  4004b1:       00          mov    $0x0,%eax
  4004b6:       00          sub    ___4___(%rsi),%eax
  4004b9:       00          add    0xc(%rsi),%eax
  4004bc:       00          retq
  4004bd:       00          mov    $0x0,%eax
  4004c2:       00          jmp    ___5___
  4004c4:       00          mov    0x8(%rsi),%eax
  4004c7:       00          add    ___6___(%rsi),%eax
  4004ca:       00          retq
```

**Please write your answer in the table on the next page.**

**Question 10 continues.**

Figure 1: Jump Table.

| Address | Value |
|---|---|
| 0x400568 | 0x4004a5 |
| 0x400570 | 0x4004c4 |
| 0x400578 | 0x4004bd |
| 0x400580 | 0x4004c4 |
| 0x400588 | 0x4004bd |
| 0x400590 | 0x4004ac |
| 0x400598 | 0x4004b1 |

**Your answers for the question 10 here:**

| | | | |
|---|---|---|---|
| 1 | `jmpq *0x400568(,%rdi,8)` | 4 | `0x8` |
| 2 | `mov (%rsi,%rax,4),%eax` | 5 | `4004b9` |
| 3 | `4004b6` | 6 | `0x4` |

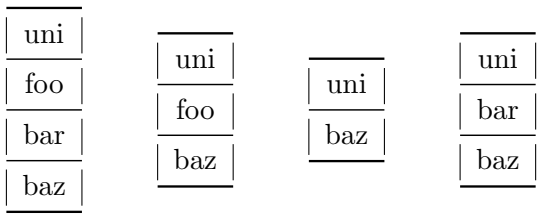
11. [8 points] **Stack frames.**

In the following (pseudo) code, at line 9, the only possible stack frame looks like this.


Draw all (four) possible shapes of the stack frames at line 21. If you present more than four shapes, you will get **no** score.

```
1  void uni() {
2      star();
3      foo();
4      baz();
5      bar();
6      baz();
7  }
8  void star() {
9      return;
10 }
11 void foo() {
12     bar();
13     baz();
14 }
15
16 void bar() {
17     baz();
18 }
19
20 void baz() {
21     return;
22 }
```

**Solution:**




Two points for each frame.

12. **Layout.**

Let's represent the layout of the following `struct`

```
struct {
    char b;
    int a;
} s;
```

as

| b | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | a | a | a | a | a | a | a(end) |

Each slot represents which member the byte belongs to, and left empty if it belongs to a padding. Each row represents an 8-byte word. The last byte of the `struct` is notes with `(end)` as well.

Answer the questions regarding the following definition.

```
struct {
    short *a;
    char b;
    short c;
    int d;
    char e;
    double f;
    char g
    short h;
    char j;
    int k;
} s;
```

(a) [6 points] Show how the `struct` definition would appear on a 64-bit Linux machine.

**Solution:**

| a | a | a | a | a | a | a | a |
|---|---|---|---|---|---|---|---|
| b | | c | c | d | d | d | d |
| e | | | | | | | |
| f | f | f | f | f | f | f | f |
| g | | h | h | j | | | |
| k | k | k | k | | | | (end) |

**This question continues on the next page.**

**Question 12 continues:**

(b) [8 points] Provide an alternative declaration that minimizes the padding size and the corresponding memory layout.

**Solution: Example**

```
struct {
    short *a;
    char b;
    char e;
    short c;
    int d;
    double f;
    char g
    char j;
    short h;
    int k;
} s;
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | a | a | a | a | a | a | a |
| b | e | c | c | d | d | d | d |
| f | f | f | f | f | f | f | f |
| g | j | h | h | k | k | k | k (end) |

- 4 points for the declaration.
- 4 points for the layout diagram.
- If the new declaration is smaller than the original but larger than the model, 2 points each.

13. **Cache Behavior.**

We are given the state of a cache at moment and a sequence of trace, each of which represents either read from or write to a memory location. With the following assumptions, determine if each of the memory access in the trace results in hit or miss.

- The memory is byte addressable.
- All memory access are to a 1-byte word.
- Addresses are 8 bit wide.
- The cache is two-way set associative and each block has two bytes.
- There are eight sets in the cache.
- When we evict a cache block, we evict the least recently used (LRU) one. If none of the lines in a set has been used by the given trace we evict the first line (line 1).

Start from this state.

| | line 1 | | | | line 2 | | | |
|---|---|---|---|---|---|---|---|---|
| Set index | Tag | Valid | Byte 0 | Byte 1 | Tag | Valid | Byte 0 | Byte 1 |
| 0 | - | 0 | - | - | 0 | 1 | 00 | 01 |
| 1 | 3 | 1 | DE | 23 | 2 | 1 | FF | 33 |
| 2 | E | 1 | 14 | FF | D | 1 | 23 | 33 |
| 3 | F | 1 | FE | FF | 1 | 1 | 45 | 11 |
| 4 | 2 | 1 | 23 | 78 | F | 1 | 45 | 23 |
| 5 | 3 | 1 | FE | FF | - | 0 | - | - |
| 6 | 2 | 1 | 01 | 02 | - | 0 | - | - |
| 7 | B | 1 | 02 | 03 | E | 1 | FF | CD |

(a) [10 points] State either miss or hit at each trace. Assume that they are executed in sequence. The first operand of both commands stands for the address, and the second operand of `write` command is the data to be written.

| | |
|---|---|
| 1. write 0x10 0xA | 1. **miss** |
| 2. read 0x01 | 2. **hit** |
| 3. read 0x31 | 3. **miss** |
| 4. write 0x2C 0xD | 4. **hit** |
| 5. read 0x3D | 5. **miss** |
| 6. read 0x2D | 6. **hit** |
| 7. read 0xF6 | 7. **hit** |
| 8. write 0xF5 0xC | 8. **miss** |
| 9. read 0x11 | 9. **miss** |
| 10. read 0x2B | 10. **miss** |

**This question continues on the next page.**

**Question 13 continues.**

(b) [8 points] Show the final state of the cache. Represent all the values in hexadecimal numbers. If the value is unknown, mark is as `XX`.

| | line 1 | | | | line 2 | | | |
|---|---|---|---|---|---|---|---|---|
| Set index | Tag | Valid | Byte 0 | Byte 1 | Tag | Valid | Byte 0 | Byte 1 |
| 0 | | | | | | | | |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 | | | | | | | | |
| 5 | | | | | | | | |
| 6 | | | | | | | | |
| 7 | | | | | | | | |

**Solution:**

| | line 1 | | | | line 2 | | | |
|---|---|---|---|---|---|---|---|---|
| Set index | Tag | Valid | Byte 0 | Byte 1 | Tag | Valid | Byte 0 | Byte 1 |
| 0 | 3 | 1 | XX | XX | 1 | 1 | 0A | XX |
| 1 | 3 | 1 | DE | 23 | 2 | 1 | FF | 33 |
| 2 | F | 1 | XX | 0C | D | 1 | 23 | 33 |
| 3 | F | 1 | FE | FF | 1 | 1 | 45 | 11 |
| 4 | 2 | 1 | 23 | 78 | C | 1 | 45 | 23 |
| 5 | 3 | 1 | FE | FF | 2 | 1 | XX | XX |
| 6 | 2 | 1 | 0D | 02 | 3 | 1 | XX | XX |
| 7 | B | 1 | 02 | 03 | E | 1 | FF | CD |

There are 21 differences. -0.25 point for each incorrect changes, between 0 and 8.

14. **Estimating the miss rate.**

With the these assumptions and the given code snippet, answer the questions.

1. `len` is the length or each array `a` and `b`.
2. The cache is empty initially.
3. The cache is fully associative (that is, there is no conflict miss).
4. Upon a cache miss, the least recently used one is being evicted.
5. `i` and `j` are stored in registers, and all three arrays are kept in the memory.
6. The size of each cache block is 4 bytes (2 `shorts`).
7. The cache size is 16 bytes, which can contain 8 `shorts`.
8. The function is called with `len = 48`.

```
1  void running_sum(short* a, short* b, size_t len) {
2      size_t i,j;
3      if (len % 12 != 0) return; // Make len to be a multiple of 12.
4      for(i = 0; i < len; i+=1) {
5          a[i] = 0;
6          for(j = i; j < (i + 12) && i + j < len ; j+=1) {
7              a[i] += b[j % len];
8          }
9      }
10 }
```

(a) [4 points] What is the total number of memory reads?

**Solution:**

- for $i < 18$, 24 reads per iteration: $24 \times 18 = 432$
- for $18 <= i < 24$, 24, 20, 16, 12, 8, 4 reads at each iteration: $28 \times 3 = 84$
- 516 reads.

Mistake in calculation: 2 point.

(b) [4 points] What is the total number of cache misses?

**Solution:**

- Accesses to `a` cause one miss at every other iteration: 24.
- As `a` occupies one block, 3 blocks are available for `b`.
- for $i < 18$, Accesses to `b` cause 6 misses at every iteration: $18 \times 6 = 108$.
- for $18 <= i < 24$, 6, 5, 4, 3, 2, 1 misses at most, at each iteration: $7 \times 3 = 21$
- On the last iterations, in fact, we will not experience all those misses as the accesses across the iterations will touch overlapping cache blocks.
- roughly 153 misses.

Mistake in calculation: 2 point.


Question 14 continues on the next page...

**This question continues on the next page..**

**Question 14 continues.**

(c) [10 points] Each element in the array `b` is used exactly 12 times, which is causing roughly 6 misses. Can we optimize the code to reduce it to roughly three misses? We could do by unrolling the loop: by filling out two elements in `a` at each iteration. Write the optimized version and calculate the number of total cache misses.

**Solution:**

```
1  void running_sum(short* a, short* b, size_t len) {
2      size_t i,j;
3      if (len % 12 != 0) return; // Make len to be a multiple of 12.
4      for(i = 0; i < len; i+=2) {
5          a[i] = 0;
6          a[i+1] = 0;
7          for(j = i; j < (i + 12) && i + j < len; j+=1) {
8              a[i] += b[j % len];
9              a[i+1] += b[j+1 % len];
10         }
11     }
12     /* Some code blocks to handle the leftovers */
13 }
```

- Accesses to `a` cause one miss at every iteration: 24.
- As `a` occupies one block, 3 blocks are available for `b`.
- for $i < 18$, Accesses to `b` cause 6 misses at every iteration: $9 \times 6 = 54$.
- for $18 <= i < 24$, 6, 4, 2 misses at most, at each iteration: 12
- On the last iterations, in fact, we will not experience all those misses as the accesses across the iterations will touch overlapping cache blocks.
- roughly 90 misses.
- code: 6 points.
- calculation: 4 points. Numeric mistake: 2 points.

(d) [14 points] Further unroll the loop to fill out four elements in `a` at each iteration. Does it still result in the reduction in miss rate? State the optimized version and answer the question with the number of misses.

**Solution:**

```
1  void running_sum(short* a, short* b, size_t len) {
2      size_t i,j;
3      if (len % 12 != 0) return; // Make len to be a multiple of 12.
4      for(i = 0; i < len; i+=4) {
5          a[i] = 0;
6          a[i+1] = 0;
7          a[i+2] = 0;
```

```
        a[i+3] = 0;
        for(j = i; j < (i + 12) && i + j < len; j+=1) {
            a[i] += b[j % len];
            a[i+1] += b[j+1 % len];
            a[i+2] += b[j+2 % len];
            a[i+3] += b[j+3 % len];
        }
    }
    /* Some code blocks to handle the leftovers */
}
```

- Accesses to `a` two miss at every iteration: 24.
- As `a` occupies two blocks, 2 blocks are available for `b`.
- for $i < 16$, Accesses to `b` cause 6 misses at every iteration: $4 \times 6 = 24$.
- for $16 <= i < 24$, 6, 4, 2 misses at most, at each iteration: 12
- On the last iterations, in fact, we will not experience all those misses as the accesses across the iterations will touch overlapping cache blocks.
- roughly 60 misses.
- code: 10 points.
- calculation: 4 points. Numeric mistake: 2 points.

15. **Optimization**

We are given the following code snippet.

```
int sum(int* c, size_t len) {
    size_t i;
    int ret;
    for(i = 0; i < len; i+=1) {
        ret += c[i];
    }
    return ret;
}

void foo(int* a, int* b, size_t len) {
    size_t i = 0;
    while(a[0] != 10 && i < len) {
        if (b[i] > sum(a)) {
            b[i] = sum(a);
        }
        i += 1;
    }
    return;
}
```

(a) [2 points] If a compiler does not optimize the program at all, how many time the function `sum` will be called in the worst case?

**Solution:** `2 * len` times.

(b) [4 points] How can the compiler reduce the number of `sum` calls using a technique related to the *common subexpressions*? Write the optimized version of `foo` function in C.

**Solution:**

```c
void foo(int* a, int* b, size_t len) {
    size_t i = 0;
    while(a[0] != 10 && i < len) {
        int temp = sum(a);
        if (b[i] > temp) {
            b[i] = temp;
        }
        i += 1;
    }
    return;
}
```

**This question continues on the next page.**

**Question 15 continues.**

(c) [10 points] Can a developer further optimize to reduce the number of `sum` function calls? Under what assumption? Provide an example of function arguments that prevents the compiler from making such optimization and the optimized version of `foo` function.

**Solution:**

- Possible.
- Assumption: `a` and `b` are different arrays.
- Example input: `a = b = {1,2,3,4,5}`.
- Optimized version:

```
void foo(int* a, int* b, size_t len) {
    size_t i = 0;
    if(a[0] == 10) return;
    int temp = sum(a);
    while(i < len) {
        if (b[i] > temp) {
            b[i] = temp;
        }
        i += 1;
    }
    return;
}
```

- Code: 4 points.
- Assumption: 4 points.
- Example argument: 2 points.

16. **Linking.**

Answer the questions about the following code snippet of the two files.

```c
/* foo.c */
#include <stdio.h>
#include <stdlib.h>

unsigned long mul_var = 2;
static unsigned long mul_const;
unsigned long foo(int in) {
    static unsigned long sum = 0;
    sum += in;
    mul_var *= in;
    mul_const *= 7;
}
```

```c
/* bar.c */
#include <stdio.h>

extern void foo(int);

unsigned long mul_var;
unsigned long mul_const = 1;
static unsigned long sum;
static void print(void) {
    printf("sum: %lu\n",sum);
    printf("mul_var: %lu\n",mul_var);
    printf("mul_const: %lu\n",mul_const);
}
int main(int argc, char* argv[]) {
    size_t i = 0;
    sum = 1;
    for(i = 0; i < argc; i+= 1) {
        foo(argc);
    }
    print();
}
```

**This question continues on the next page.**

**Question 16 continues.**

(a) [15 points] This table shows if each symbol in the two source files found in the symbol table after linking, and if so, some of the contents. Fill out the blanks. If a symbol is not found from the symbol table, leave the remaining slots empty

| Name | Exists (Yes/No) | Binding (Local/Global) | Section (text/data/bss) |
|---|---|---|---|
| `main` | Yes | **Global** | text |
| `i` | **No** | | |
| `sum` | **Yes** | **Local** | **bss** |
| `argc` | **No** | | |
| `mul_var` | **Yes** | **Global** | **data** |
| `foo` | **Yes** | **Global** | **text** |
| `print` | **Yes** | **Local** | **text** |

**Solution:** There are 15 blanks to fill. +1 for each correct blank, -1 for each blank filled out, which shouldn't have been filled.

(b) [6 points] What will this program print when the value of `argc` is 3?

**Solution:**

```
sum: 1
mul_var: 54
mul_const: 1
```

2 points each.

(c) [5 points] This is the `call` instruction in the `main` function calling `foo`, before linking.

```
 89:        e8 00 00 00 00              callq  8e <main+0x36>
 8e:        48 83 45 f8 01              addq   $0x1,-0x8(%rbp)
```

The latter four out of five bytes store the offset of the function `foo`, which is determined at link time.

In the following code snippet after the linking, state the byte sequence that should replace the `XX`s.

```
6d3:        e8 XX XX XX XX              callq  6f4 <foo>
6d8:        48 83 45 f8 01              addq   $0x1,-0x8(%rbp)
```

```
00000000000006f4 <foo>:
6f4:        55                          push   %rbp
6f5:        48 89 e5                    mov    %rsp,%rbp
```

**Solution:** `1c 00 00 00`

**The end of questions.**