3_homework

3.58 *

For a function with prototype

```
long decode2(long x, long y, long z);
```

gcc generates the following assembly code:

```
decode2:
        subq %rdx, %rsi
        imulq %rsi, %rdi
        movq %rsi, %rax
        salq $63, %rax
        sarq $63, %rax
        xorq %rdi, %rax
        ret
```

Parameters x, y, and z are passed in registers %rdi, %rsi, and %rdx. The code stores the return value in register %rax.
Write C code for decode2 that will have an effect equivalent to the assembly code shown.

Solution:
Analyze assembly code:

```
# long decode2(long x, long y, long z)
# x - rdi
# y - rsi
# z - rdx
decode2:
        subq %rdx, %rsi     # rsi=rsi-rdx: y = y - z
        imulq %rsi, %rdi    # rdi=rdi*rsi: x = x * (y - z)
        movq %rsi, %rax     # rax=rsi: rax = y - z
        salq $63, %rax      # rax=rax<<63: rax = (y - z) << 63
        sarq $63, %rax      # rax=rax>>63: rax = ((y - z) << 63) >> 63
        xorq %rdi, %rax     # rax=rax^rdi: rax = (((y - z) << 63) >> 63) ^ (x * (y - z))
        ret
```

So the code:

```
long decode2(long x, long y, long z)
{
        return (((y - z) << 63) >> 63) ^ (x * (y - z));
}
```

3.59 **

The following code computes the 128-bit product of two 64-bit signed values x and y and stores the result in memory:

```
typedef __int128 int128_t;

void store_prod(int128_t *dest, int64_t x, int64_t y) {
        *dest = x * (int128_t)y;
}
```

Gcc generates the following assembly code implementing the computation:

```
store_prod:
        movq    %rdx, %rax
        cqto
        movq    %rsi, %rcx
        sarq    $63, %rcx
        imulq   %rax, %rcx
        imulq   %rsi, %rdx
        addq    %rdx, %rcx
        mulq    %rsi
        addq    %rcx, %rdx
        movq    %rax, (%rdi)
        movq    %rdx, 8(%rdi)
        ret
```

This code uses three multiplications for the multiprecision arithmetic required to implement 128-bit arithmetic on a 64-bit machine. Describe the algorithm used to compute the product, and annotate the assembly code to show how it realizes your algorithm. Hint: When extending arguments of x and y to 128 bits, they can be rewritten as $x = 2^{64} \times x_h + x_l$ and $y = 2^{64} \times y_h + y_l$, where x_h, x_l, y_h, and y_l are 64-bit values. Similarly, the 128-bit product can be written as
$p = 2^{64} \times p_h + p_l$, where p_h and p_l are 64-bit values. Show how the code computes the values of p_h and p_l in terms of x_h, x_l, y_h, and y_l.

Solution:
Firstly, let's analyze the 'hint':
$x \times y$ = ($2^{64} \times x_h + x_l$) x ($2^{64} \times y_h + y_l$) = $2^{128} \times x_h \times y_h + 2^{64} \times x_h \times y_l + 2^{64} \times x_l \times y_h + x_l \times y_l$
We look into it one by one:

- $2^{128} \times x_h \times y_h$: ignore
- $2^{64} \times x_h \times y_l$: signed multiplication, only need 64 bits
- $2^{64} \times x_l \times y_h$: signed multiplication, only need 64 bits
- $x_l \times y_l$: we need 128 bits. Unsigned multiplication.
  This is our algorithm.
  Secondly, reverse the assembly code:

```
# void store_prod(int128_t *dest, int64_t x, int64_t y)
# rdi - dest
# rsi - x
# rdx - y
store_prod:
        movq    %rdx, %rax      # rax=rdx: rax = y
        cqto                    # signed extend to 128bits
                                # rdx: y>>63 - copy the sign bit of rax to all bits in rdx - yh
                                # rax: y - yl
        movq    %rsi, %rcx      # rcx=rsi: rcx = x - xl
        sarq    $63, %rcx       # rcx=rcx>>63 - get the sign bit of x - xh
        imulq   %rax, %rcx      # rcx=rcx*rax: rcx = xh * yl
        imulq   %rsi, %rdx      # rdx=rdx*rsi: rdx = yh * xl
        addq    %rdx, %rcx      # rcx=rcx+rdx: rcx = xh * yl + yh * xl
        mulq    %rsi            # rdx:rax=rax*rsi: rdx:rax = yl * xl
        addq    %rcx, %rdx      # rdx=rdx+rcx: ph = rdx +  xh * yl + yh * xl
                                               pl = rax
        movq    %rax, (%rdi)    # M(rdi)=rax: *dest = pl
        movq    %rdx, 8(%rdi)   # M(rdi+8)=rdx: *(dest+8) = ph
        ret
```

3.60 **

Consider the following assembly code:

```
# long loop(long x, int n)
# x in %rdi, n in %esi

loop:
        movl %esi, %ecx
        movl $1, %edx
        movl $0, %eax
        jmp .L2

.L3:
        movq %rdi, %r8
        andq %rdx, %r8
        orq %r8, %rax
        salq %cl, %rdx

.L2:
        testq %rdx, %rdx
        jne .L3
        rep; ret
```

The preceding code was generated by compiling C code that had the following overall form:

```c
long loop(long x, long n)
{
        long result = ______;
        long mask;
        for (mask = ______; mask ______; mask = ______) {
                result |= ______;
        }
        return result;
}
```

Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register `%rax`. You will find it helpful to examine the assembly code before, during, and after the loop to form a consistent mapping between the registers and the program variables.

A. Which registers hold program values x, n, result, and mask?

B. What are the initial values of result and mask?

C. What is the test condition for mask?

D. How does mask get updated?

E. How does result get updated?

F. Fill in all the missing parts of the C code.

Solution:
Firstly, analyze the assembly code:

```
# long loop(long x, int n)
# x in %rdi, n in %esi

loop:
        movl %esi, %ecx     # ecx=esi: ecx = n
        movl $1, %edx       # edx=1: edx = 1
        movl $0, %eax       # eax=0: eax = 0
        jmp .L2

.L3:
        movq %rdi, %r8      # r8=rdi: r8 = x
        andq %rdx, %r8      # r8=r8&rdx: r8 = x & rdx
        orq %r8, %rax       # rax=rax|r8: rax = rax | (x & rdx)
        salq %cl, %rdx      # rdx=rdx>>cl

```

```
.L2:
        testq %rdx, %rdx
        jne .L3            # if rdx!=0, goto .L3 - rdx is mask
        rep; ret
```

A. Which registers hold program values x, n, result, and mask?

mask – rdx

result – rax

x – rdi

n – esi

B. What are the initial values of result and mask?

`movl $0, %eax` – The initial value of result is 0.

`movl $1, %edx` – The initial value of mask is 1.

C. What is the test condition for mask?

`testq %rdx, %rdx` – `mask != 0`

D. How does mask get updated?

`salq %cl, %rdx` – `mask << n`

E. How does result get updated?

`andq %rdx, %r8` – `r8 = x & mask`

`orq %r8, %rax` – `result = result | (x & mask)`

F. Fill in all the missing parts of the C code.

```c
long loop(long x, long n)
{
        long result = 0;
        long mask;
        for (mask = 1; mask != 0; mask = mask << n) {
                result |= (x & mask);
        }
        return result;
}
```

3.61 **

In Section 3.6.6, we examined the following code as a candidate for the use of conditional data transfer:

```c
long cread(long *xp) {
        return (xp ? *xp : 0);
}
```

We showed a trial implementation using a conditional move instruction but argued that it was not valid, since it could attempt to read from a null address.

Write a C function cread_alt that has the same behavior as cread, except that it can be compiled to use conditional data transfer. When compiled, the generated code should use a conditional move instruction rather than one of the jump instructions.

Solution:

If compile using a conditional move instruction, it would be like:

```
# long cread(long *xp)
# Invalid implementation of function cread
# xp in register %rdi

cread:
        movq (%rdi), %rax      # v = *xp
        testq %rdi, %rdi       # Test x
        movl $0, %edx          # Set ve = 0
```

```
        cmove %rdx, %rax      # If x==0, v = ve
        ret                   # Return v
```

which is:

```c
long cread(long *xp) {
        v = *xp;
        if x = 0
        ve = 0
        v = ve
        return v;
}
```

Obviously, we need to avoid the expression `*xp`.

If we change the source code to be:

```c
long cread(long *xp) {
        return (!xp ? 0 : *xp);
}
```

Then the pseudo-code would be:

```c
long cread(long *xp) {
        v = 0;
        if !xp = 0
        v = (*xp)
        return v;
}
```

So the assembly code would be:

```
cread:
        movq     $0, %rax             # v = 0
        testq    %rdi, %rdi           # Test x
        cmovne   (%rdi), %rax         # If x==0, v = ve
        ret                           # Return v
```

3.62 **

The code that follows shows an example of branching on an enumerated type value in a switch statement. Recall that enumerated types in C are simply a way to introduce a set of names having associated integer values. By default, the values assigned to the names count from zero upward. In our code, the actions associated with the different case labels have been omitted.

```c
/* Enumerated type creates set of constants numbered 0 and upward */
typedef enum {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;

long switch3(long *p1, long *p2, mode_t action)
{
        long result = 0;
        switch(action) {
                case MODE_A:
                case MODE_B:
                case MODE_C:
                case MODE_D:
                case MODE_E:
                default:
        }
        return result;
}
```

The part of the generated assembly code implementing the different actions is shown below. The annotations indicate the argument locations, the register values, and the case labels for the different jump destinations.

```
# p1 in %rdi, p2 in %rsi, action in %edx

.L8:            # MODE_E
        movl $27, %eax
        ret

.L3:            # MODE_A
        movq (%rsi), %rax
        movq (%rdi), %rdx
        movq %rdx, (%rsi)
        ret

.L5:            # MODE_B
        movq (%rdi), %rax
        addq (%rsi), %rax
        movq %rax, (%rdi)
        ret

.L6:            # MODE_C
        movq $59, (%rdi)
        movq (%rsi), %rax
        ret

.L7:            # MODE_D
        movq (%rsi), %rax
        movq %rax, (%rdi)
        movl $27, %eax
        ret

.L9:            # default
        movl $12, %eax
        ret
```

Fill in the missing parts of the C code. It contained one case that fell through to another —try to reconstruct this.

Solution:

```
# p1 in %rdi, p2 in %rsi, action in %edx

.L8:            # MODE_E
        movl $27, %eax          # eax = 27
        ret

.L3:            # MODE_A
        movq (%rsi), %rax       # rax = M(rsi): rax = *(p2)
        movq (%rdi), %rdx       # rdx = M(rdi): rdx = *(p1)
        movq %rdx, (%rsi)       # M(rsi) = rdx: *(p2) = rdx = *(p1)
        ret

.L5:            # MODE_B
        movq (%rdi), %rax       # rax = M(rdi): rax = *(p1)
        addq (%rsi), %rax       # rax = rax + M(rsi): rax = *(p2) + *(p1)
        movq %rax, (%rdi)       # M(rdi) = rax: *p1 = rax = *(p2) + *(p1)
        ret

.L6:            # MODE_C
        movq $59, (%rdi)        # M(rdi) = 59: (*p1) = 59
        movq (%rsi), %rax       # rax = M(rsi): rax = *p2
        ret

```

```
.L7:             # MODE_D
        movq (%rsi), %rax        # rax = M(rsi): rax = *p2
        movq %rax, (%rdi)        # M(rdi) = rax: (*p1) = rax = (*p2)
        movl $27, %eax           # eax = 27
        ret

.L9:             # default
        movl $12, %eax           # eax = 12
        ret
```

So we can get the C code below:

```c
/* Enumerated type creates set of constants numbered 0 and upward */
typedef enum {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;

long switch3(long *p1, long *p2, mode_t action)
{
        long result = 0;
        switch(action) {
                case MODE_A:
                        result = *p2;
                        *(p2) = *(p1);
                        break;

                case MODE_B:
                        result = *(p2) + *(p1);
                        *p1 = result;
                        break;

                case MODE_C:
                        (*p1) = 59;
                        result = *p2;
                        break;

                case MODE_D:
                        (*p1) = (*p2);
                        result = 27;
                        break;

                case MODE_E:
                        result = 27;
                        break;

                default:
                        result = 12;
        }
        return result;
}
```


3.63 **

This problem will give you a chance to reverse engineer a switch statement from disassembled machine code. In the following procedure, the body of the switch statement has been omitted:

```c
long switch_prob(long x, long n) {
        long result = x;
        switch(n) {
                /* Fill in code here */
        }
        return result;
}
```

Figure below shows the disassembled machine code for the procedure:

```
# long switch_prob(long x, long n)
```

```
2    # x in %rdi, n in %rsi
3
4    0000000000400590 <switch_prob>:
5            400590: 48 83 ee 3c              sub  $0x3c,%rsi
6            400594: 48 83 fe 05              cmp  $0x5,%rsi
7            400598: 77 29                    ja   4005c3 <switch_prob+0x33>
8            40059a: ff 24 f5 f8 06 40 00     jmpq *0x4006f8(,%rsi,8)
9            4005a1: 48 8d 04 fd 00 00 00     lea  0x0(,%rdi,8),%rax
10           4005a8: 00
11           4005a9: c3                       retq
12           4005aa: 48 89 f8                 mov  %rdi,%rax
13           4005ad: 48 c1 f8 03              sar  $0x3,%rax
14           4005b1: c3                       retq
15           4005b2: 48 89 f8                 mov  %rdi,%rax
16           4005b5: 48 c1 e0 04              shl  $0x4,%rax
17           4005b9: 48 29 f8                 sub  %rdi,%rax
18           4005bc: 48 89 c7                 mov  %rax,%rdi
19           4005bf: 48 0f af ff              imul %rdi,%rdi
20           4005c3: 48 8d 47 4b              lea  0x4b(%rdi),%rax
21           4005c7: c3                       retq
```

The jump table resides in a different area of memory. We can see from the indirect jump on line 5 that the jump table begins at address 0x4006f8. Using the gdb debugger, we can examine the six 8-byte words of memory comprising the jump table with the command x/6gx 0x4006f8. Gdb prints the following:

```
1    (gdb) x/6gx 0x4006f8
2
3    0x4006f8: 0x00000000004005a1 0x00000000004005c3
4    0x400708: 0x00000000004005a1 0x00000000004005aa
5    0x400718: 0x00000000004005b2 0x00000000004005bf
```

Fill in the body of the switch statement with C code that will have the same behavior as the machine code.

Solution:
We look into the assembly code:

```
1    # long switch_prob(long x, long n)
2    # x in %rdi, n in %rsi
3
4    0000000000400590 <switch_prob>:
5            400590: 48 83 ee 3c              sub  $0x3c,%rsi                # rsi=rsi-0x3c: n = n - 0x3c
6            400594: 48 83 fe 05              cmp  $0x5,%rsi                 # cmp 5 and rsi: cmp 5 and n-0x3c
7            400598: 77 29                    ja   4005c3 <switch_prob+0x33>  # if n-0x3c > 5, goto 4005c3
8            40059a: ff 24 f5 f8 06 40 00     jmpq *0x4006f8(,%rsi,8)        # goto *(0x4006f8+8*rsi)
9            4005a1: 48 8d 04 fd 00 00 00     lea  0x0(,%rdi,8),%rax         # if rsi=0 or 2, rax=8*rdi:
10                                                                          # if n - 0x3c = 0, result = 8*x
11           4005a8: 00
12           4005a9: c3                       retq
13           4005aa: 48 89 f8                 mov  %rdi,%rax                 # if rsi=3, rax=rdi
14                                                                          # if n - 0x3c = 3, result = x
15           4005ad: 48 c1 f8 03              sar  $0x3,%rax                 # rax=rax>>3: result = x/8
16           4005b1: c3                       retq
17           4005b2: 48 89 f8                 mov  %rdi,%rax                 # if rsi=4, rax=rdi
18                                                                          # if n - 0x3c = 4, result = x
```

```
  4005b5: 48 c1 e0 04        shl  $0x4,%rax          # rax=rax<<4: result = x*16
  4005b9: 48 29 f8           sub  %rdi,%rax          # rax=rax-rdi: result = 15*x
  4005bc: 48 89 c7           mov  %rax,%rdi          # rdi=rax: rdi = 15*x

  4005bf: 48 0f af ff        imul %rdi,%rdi          # sharedcode - rdi=rdi*rdi
  4005c3: 48 8d 47 4b        lea  0x4b(%rdi),%rax    # sharedcode - if rsi=1, rax=rdi+0x4b
                                                   # rax = rdi + 0x4b
                                                     # sharedcode - if n - 0x3c = 1, result = x + 0x4b
  4005c7: c3                 retq
```

Then construct the C code:

```c
long switch_prob(long x, long n) {
        long result = x;
        switch(n) {
                /* Fill in code here */
                case 0x3c:         // rsi = 0
                case 0x3e:         // rsi = 2
                        result = 8*x;
                        break;
                case 0x3f:         // rsi = 3
                        result = x/8;
                        break;
                case 0x40:         // rsi = 4
                        result = 15*x*15*x + 0x4b;
                        break;
                default:         // rsi = 1
                        result = x + 0x4b;
        }
        return result;
}
```

3.64 ***

Consider the following source code, where R, S, and T are constants declared with `#define`:

```c
long A[R][S][T];

long store_ele(long i, long j, long k, long *dest)
{
        *dest = A[i][j][k];
        return sizeof(A);
}
```

In compiling this program, gcc generates the following assembly code:

```
# long store_ele(long i, long j, long k, long *dest)
# i in %rdi, j in %rsi, k in %rdx, dest in %rcx

store_ele:
        leaq (%rsi,%rsi,2), %rax
        leaq (%rsi,%rax,4), %rax
        movq %rdi, %rsi
        salq $6, %rsi
        addq %rsi, %rdi
        addq %rax, %rdi
        addq %rdi, %rdx
        movq A(,%rdx,8), %rax
        movq %rax, (%rcx)
        movl $3640, %eax
```

```
        ret
```

A. Extend Equation 3.1 from two dimensions to three to provide a formula for the location of array element `A[i][j][k]`.

B. Use your reverse engineering skills to determine the values of R, S, and T based on the assembly code.

Solution:

A.

What is euqation 3.1?

T D[R][C];

$\&D[i][j] = x_D + L(C \times i + j)$

This is for 2 dimensions' array. When doing on 3 dimensions' array, it's like:

T A[R][S][T];

$\&A[i][j][k] = x_A + L(S * T * i + T * j + k)$;

B.

```
# long store_ele(long i, long j, long k, long *dest)
# i in %rdi, j in %rsi, k in %rdx, dest in %rcx

store_ele:
        leaq (%rsi,%rsi,2), %rax    # rax=3*rsi:  rax = 3j
        leaq (%rsi,%rax,4), %rax    # rax=rsi+4*rax: rax = j + 12j = 13j
        movq %rdi, %rsi             # rsi=rdi: rsi = i
        salq $6, %rsi               # rsi=rsi<<6: rsi = 64i
        addq %rsi, %rdi             # rdi=rdi+rsi: rdi = i + 64i = 65i
        addq %rax, %rdi             # rdi=rdi+rax: rdi = 65i + 13j
        addq %rdi, %rdx             # rdx=rdx+rdi: rdx = k + 65i + 13j
        movq A(,%rdx,8), %rax       # rax=A+8*rdx: rax = A + 8(k + 65i + 13j)
        movq %rax, (%rcx)           # M(rcx)=rax: *dest = A + 8(k + 65i + 13j)
        movl $3640, %eax            # eax=3640
        ret
```

Thus, we get `*dest = A + 8(k + 65i + 13j)`

From all the information, we get:

$R * S * T * 8 = 3640$

$S * T = 65$

$T = 13$

$S = 5$

$R = 7$

3.65 *

The following code transposes the elements of an M × M array, where M is a constant defined by `#define`:

```c
void transpose(long A[M][M]) {
        long i, j;
        for (i = 0; i < M; i++)
                for (j = 0; j < i; j++) {
                        long t = A[i][j];
                        A[i][j] = A[j][i];
                        A[j][i] = t;
                }
}
```

When compiled with optimization level -O1, gcc generates the following code for the inner loop of the function:

```
.L6:
        movq (%rdx), %rcx
        movq (%rax), %rsi
        movq %rsi, (%rdx)
        movq %rcx, (%rax)
        addq $8, %rdx
        addq $120, %rax
        cmpq %rdi, %rax
        jne .L6
```

We can see that gcc has converted the array indexing to pointer code.

A. Which register holds a pointer to array element A[i][j]?

B. Which register holds a pointer to array element A[j][i]?

C. What is the value of M?

Solution:
Look into the assembly code:

```
# void transpose(long A[M][M])
# rdi - A
.L6:
        movq (%rdx), %rcx     # rcx=M(rdx): rcx = A[i][j]
        movq (%rax), %rsi     # rsi=M(rax): rsi = A[j][i]
        movq %rsi, (%rdx)     # M(rdx)=rsi:
        movq %rcx, (%rax)     # M(rax)=rcx:
        addq $8, %rdx         # rdx=rdx+8: (&A[i][j])++
        addq $120, %rax       # rax=rax+120: &A[j][i] = &A[j][i] + j
        cmpq %rdi, %rax
        jne .L6
```

A.
rdx

B.
rax

C.
15

3.66 *

Consider the following source code, where NR and NC are macro expressions declared with `#define` that compute the dimensions of array A in terms of parameter n. This code computes the sum of the elements of column j of the array.

```
long sum_col(long n, long A[NR(n)][NC(n)], long j) {
        long i;
        long result = 0;
        for (i = 0; i < NR(n); i++)
                result += A[i][j];
        return result;
}
```

In compiling this program, gcc generates the following assembly code:

```
# long sum_col(long n, long A[NR(n)][NC(n)], long j)
# n in %rdi, A in %rsi, j in %rdx

sum_col:
        leaq   1(,%rdi,4), %r8
        leaq   (%rdi,%rdi,2), %rax
```

```
        movq   %rax, %rdi
        testq  %rax, %rax
        jle    .L4
        salq   $3, %r8
        leaq   (%rsi,%rdx,8), %rcx
        movl   $0, %eax
        movl   $0, %edx

.L3:
        addq   (%rcx), %rax
        addq   $1, %rdx
        addq   %r8, %rcx
        cmpq   %rdi, %rdx
        jne    .L3
        rep; ret

.L4:
        movl   $0, %eax
        ret
```

Use your reverse engineering skills to determine the definitions of NR and NC.

Solution:

Look into the assembly code:

```
# long sum_col(long n, long A[NR(n)][NC(n)], long j)
# n in %rdi, A in %rsi, j in %rdx

sum_col:
        leaq   1(,%rdi,4), %r8       # r8=4*rdi+1: r8 = 4n + 1
        leaq   (%rdi,%rdi,2), %rax   # rax=3*rdi: rax = 3n
        movq   %rax, %rdi            # rdi=rax: rdi = 3n
        testq  %rax, %rax
        jle    .L4                   # if 3n <= 0, goto .L4
        salq   $3, %r8               # r8=r8<<3: r8 = 8*r8 = 8*(4n+1)
        leaq   (%rsi,%rdx,8), %rcx   # rcx=rsi+8*rdx: rcx = A + 8j
        movl   $0, %eax              # eax = 0
        movl   $0, %edx              # edx = 0

.L3:
        addq   (%rcx), %rax          # rax+=M(rcx): rax = rax + *(A+8j)
        addq   $1, %rdx              # rdx+=1: i++
        addq   %r8, %rcx             # rcx+=r8: rcx = rcx + 8*(4n+1)
        cmpq   %rdi, %rdx
        jne    .L3                   # if rdx!=rdi, goto .L3
        rep; ret

.L4:
        movl   $0, %eax
        ret
```

Thus we get that: in loop, rcx is A[i][j]

From `addq %r8, %rcx # rcx+=r8: rcx = rcx + 8*(4n+1)`, we get that NC(n) = 4n + 1

From `cmpq %rdi, %rdx`, we get that NR(n) = 3n

3.67 **

For this exercise, we will examine the code generated by gcc for functions that have structures as arguments and return values, and from this see how these language features are typically implemented.

The following C code has a function process having structures as argument and return values, and a function eval that calls process:

```c
typedef struct {
        long a[2];
        long *p;
} strA;

typedef struct {
        long u[2];
        long q;
} strB;

strB process(strA s) {
        strB r;
        r.u[0] = s.a[1];
        r.u[1] = s.a[0];
        r.q = *s.p;
        return r;
}

long eval(long x, long y, long z) {
        strA s;
        s.a[0] = x;
        s.a[1] = y;
        s.p = &z;
        strB r = process(s);
        return r.u[0] + r.u[1] + r.q;
}
```

Gcc generates the following code for these two functions:

```
1   # strB process(strA s)
2
3   process:
4           movq %rdi, %rax
5           movq 24(%rsp), %rdx
6           movq (%rdx), %rdx
7           movq 16(%rsp), %rcx
8           movq %rcx, (%rdi)
9           movq 8(%rsp), %rcx
10          movq %rcx, 8(%rdi)
11          movq %rdx, 16(%rdi)
12          ret
13
14  # long eval(long x, long y, long z)
15  # x in %rdi, y in %rsi, z in %rdx
16
17  eval:
18          subq $104, %rsp
19          movq %rdx, 24(%rsp)
20          leaq 24(%rsp), %rax
21          movq %rdi, (%rsp)
22          movq %rsi, 8(%rsp)
23          movq %rax, 16(%rsp)
24          leaq 64(%rsp), %rdi
25          call process
26          movq 72(%rsp), %rax
27          addq 64(%rsp), %rax
28          addq 80(%rsp), %rax
29          addq $104, %rsp
30          ret
```

A. We can see on line 18 of function eval that it allocates 104 bytes on the stack. Diagram the stack frame for eval, showing the values that it stores on the stack prior to calling process.

B. What value does eval pass in its call to process?

C. How does the code for process access the elements of structure argument s?

D. How does the code for process set the fields of result structure r?

E. Complete your diagram of the stack frame for eval, showing how eval accesses the elements of structure r following the return from process.

F. What general principles can you discern about how structure values are passed as function arguments and how they are returned as function results?

Solution:

A.

Address	Relating Instruction	value	size
%rsp+104			
...			
%rsp+24	movq %rdx, 24(%rsp)	z	sizeof(long)
%rsp+16	leaq 24(%rsp), %rax movq %rax, 16(%rsp)	%rsp+24 / &z / s.p	64 bits
%rsp+8	movq %rsi, 8(%rsp)	y/s.a[1]	sizeof(long)
%rsp	movq %rdi, (%rsp)	x/s.a[0]	sizeof(long)

B.
`%rsp + 64` according to `leaq 64(%rsp), %rdi`.

C.
`%rsp + offset`

D.
When run into process function, the stack is like:

Address	Relating Instruction	value	size
%rsp+112			
...			
%rsp+72			
...			
%rsp+32	movq %rdx, 24(%rsp)	z	sizeof(long)
%rsp+24	leaq 24(%rsp), %rax movq %rax, 16(%rsp)	%rsp+32 / &z	64 bits
%rsp+16	movq %rsi, 8(%rsp)	y	sizeof(long)
%rsp+8	movq %rdi, (%rsp)	x	sizeof(long)
%rsp	call process	return address	64 bits

Copy the process C code here:

```c
strB process(strA s) {
        strB r;
        r.u[0] = s.a[1];
        r.u[1] = s.a[0];
        r.q = *s.p;
        return r;
}
```

Look into the process assembly language:

```
# strB process(strA s)
# the eval pass s=rsp+64 as parameter
process:
```

```
        movq %rdi, %rax         # rax=rdi: rax = rsp+72
        movq 24(%rsp), %rdx     # rdx=M(rsp+24): rdx = &z
        movq (%rdx), %rdx       # rdx=M(rdx): rdx = z
        movq 16(%rsp), %rcx     # rcx=M(rsp+16): rcx = y
        movq %rcx, (%rdi)       # M(rdi)=rcx: M(rsp+72) = r.u[0] = y
        movq 8(%rsp), %rcx      # rcx=M(rsp+8): rcx = x
        movq %rcx, 8(%rdi)      # M(rdi+8)=rcx: M(rsp+80) = r.u[1] = x
        movq %rdx, 16(%rdi)     # M(rdi+16)=rdx: M(rsp+88) = r.q = z
        ret
```

- We find that:
  - even though in C code: `strB r = process(s)`, it's calling process by passing `strA s` as a parameter.
  - gcc actually use rsp+64 as the address of `r`.

E.
After return from function process, the return address is popped from stack then the stack will shrink by 8 bytes: `rsp = rsp + 8`. So we get the stack below:

Address	value	size
%rsp+104		
...		
%rsp+80	z / r.q	sizeof(long)
%rsp+72	x / r.u[1]	sizeof(long)
%rsp+64	y / r.u[0]	sizeof(long)
...		
%rsp+24	z	sizeof(long)
%rsp+16	%rsp+32 / &z	64 bits
%rsp+8	y	sizeof(long)
%rsp+0	x	sizeof(long)

Then look into the assembly code below:

```
        movq 72(%rsp), %rax     # rax = M(rsp+72) = x
        addq 64(%rsp), %rax     # rax += M(rsp+64) = y: rax = x + y
        addq 80(%rsp), %rax     # rax += M(rsp+80) = z: rax = x + y + z
```

F.
caller find space and pass space address to callee, callee store data on this space area and return this address.

3.68 ***

In the following code, A and B are constants defined with `#define`:

```c
typedef struct {
        int x[A][B]; /* Unknown constants A and B */
        long y;
} str1;

typedef struct {
        char array[B];
        int t;
        short s[A];
        long u;
} str2;

```

```c
void setVal(str1 *p, str2 *q) {
        long v1 = q->t;
        long v2 = q->u;
        p->y = v1+v2;
}
```

Gcc generates the following code for setVal:

```
# void setVal(str1 *p, str2 *q)
# p in %rdi, q in %rsi

setVal:
        movslq 8(%rsi), %rax
        addq 32(%rsi), %rax
        movq %rax, 184(%rdi)
        ret
```

What are the values of A and B? (The solution is unique.)

Solution:
Look into the assembly code first:

```
# void setVal(str1 *p, str2 *q)
# p in %rdi, q in %rsi

setVal:
        movslq   8(%rsi), %rax     # rax=M(rsi+8): rax = *(q+8)
        addq     32(%rsi), %rax    # rax=rax+M(rsi+32): rax = rax + *(q+32)
        movq     %rax, 184(%rdi)   # M(rdi+184)=rax: *(p+184) = rax
        ret
```

Compare with the C code:

```c
void setVal(str1 *p, str2 *q) {
        long v1 = q->t;
        long v2 = q->u;
        p->y = v1+v2;
}
```

Easy to find that:

- `q->t` equals `*(q+8)`
- `q->u` equals `*(q+32)`
- `p->y` equals `*(p+184)`
  But we need to consider alignment, which means it's not so exactly.
  For str2 struct:

```c
typedef struct {
        char array[B];
        int t;
        short s[A];
        long u;
} str2;
```

- For `q->t` equals `*(q+8)`, we get that 4 < B <= 8.
- For `q->u` equals `*(q+32)`, we get that 24 < B + 4 + 2\A <=32, thus `20 < B + 2A <=28`.

```c
typedef struct {
        int x[A][B]; /* Unknown constants A and B */
        long y;
} str1;
```

- For `p->y` equals `*(p+184)`, we get that 176 < 4*A*B <=184, thus 44 < A*B <= 46.
- According to the information above, we get that: `A=9, B=5`

3.69 ◆◆◆

You are charged with maintaining a large C program, and you come across the following code:

```c
typedef struct {
        int first;
        a_struct a[CNT];
        int last;
} b_struct;

void test(long i, b_struct *bp)
{
        int n = bp->first + bp->last;
        a_struct *ap = &bp->a[i];
        ap->x[ap->idx] = n;
}
```

The declarations of the compile-time constant CNT and the structure a_struct are in a file for which you do not have the necessary access privilege. Fortunately, you have a copy of the .o version of code, which you are able to disassemble with the objdump program, yielding the following disassembly:

```
0000000000000000 <test>:
        0: 8b 8e 20 01 00 00     mov 0x120(%rsi), %ecx
        6: 03 0e                 add (%rsi), %ecx
        8: 48 8d 04 bf           lea (%rdi, %rdi, 4), %rax
        c: 48 8d 04 c6           lea (%rsi, %rax, 8), %rax
        10: 48 8b 50 08          mov 0x8(%rax), %rdx
        14: 48 63 c9             movslq %ecx, %rcx
        17: 48 89 4c d0 10       mov %rcx, 0x10(%rax, %rdx, 8)
        1c: c3 retq
```

Using your reverse engineering skills, deduce the following:

A. The value of CNT.

B. A complete declaration of structure a_struct. Assume that the only fields in this structure are idx and x, and that both of these contain signed values.

Solution:
Reverse the assembly code and also put the structB here for convenience:

```c
typedef struct {
        int first;
        a_struct a[CNT];
        int last;
} b_struct;
```

```
# void test(long i, b_struct *bp)
# rdi - i
# rsi - bp

0000000000000000 <test>:
        0:  mov 0x120(%rsi), %ecx        # ecx=M(rsi+0x120): ecx = bp->last
        6:  add (%rsi), %ecx             # ecx=ecx+M(rsi): ecx = bp->last + bp->first
        8:  lea (%rdi, %rdi, 4), %rax    # rax=5*rdi: rax = 5i
        c:  lea (%rsi, %rax, 8), %rax    # rax=rsi+8*rax: rax = bp + 8*5i
        10: mov 0x8(%rax), %rdx          # rdx=M(rax+8): rdx = M(bp + 8*5i + 8) = bp-
>a[i].idx = ap->idx
```

```
          14: movslq %ecx, %rcx                # rcx=ecx
          17: mov %rcx, 0x10(%rax, %rdx, 8)    # M(rax+8*rdx+0x10)=rcx
          1c: retq
```

A.
From `lea (%rsi, %rax, 8), %rax # rax=rsi+8*rax: rax = bp + 8*5i`, we get that the size of a_struct is 40 bytes including alignment.

We must know that the alignment is 8 bytes.
From `mov 0x120(%rsi), %ecx # ecx=M(rsi+0x120): ecx = bp->last`, we get that CNT = (0x120 - 8)/40 = 7.

B.
After reversing the assembly code and compare to the original C code, we get the following useful information:

- We easily get: `bp->a[i]` is `M(bp + 8*5i + 8)`, which is also `ap->idx` stored in `rdx`.
- Thus the 1st element is a_struct is `idx`.
- Then let's analyze this one: `mov %rcx, 0x10(%rax, %rdx, 8) # M(rax+8*rdx+0x10)=rcx` step by step.
  `rax+0x8` is `bp+8`, which is `&bp->a[i]`
  `rax+0x8+0x8` is `&bp->a[i] + 0x8`, which is `&bp->a[i].x`. `idx` is 8 bytes.
  `*rax+8*rdx+0x10` is `&bp->a[i].x + 8*rdx`, *which is* `&ap->x + 8*rdx`, then `&ap->x[ap->idx]`. Now `we know that: 1. the size of a_struct is 40 bytes, 2. the 1st element of a_struct is 8 bytes (` idx`), 3. the size of a in a_struct is 8 bytes.
  We easily get the a_struct below:

```c
typedef struct
{
        long idx;
        long x[4];
} a_struct;
```

3.70 ◆◆◆

Consider the following union declaration:

```c
union ele {
        struct {
                long *p;
                long y;
        } e1;

        struct {
                long x;
                union ele *next;
        } e2;
};
```

This declaration illustrates that structures can be embedded within unions.

The following function (with some expressions omitted) operates on a linked list having these unions as list elements:

```c
void proc (union ele *up) {
        up-> ______ = *( ______ ) - ______ ;
}
```

A. What are the offsets (in bytes) of the following fields:

```
e1.p          ______
e1.y          ______
```

```
e2.x          ______
e2.next       ______
```

B. How many total bytes does the structure require?

C. The compiler generates the following assembly code for proc:

```
# void proc (union ele *up)
# up in %rdi

proc:
        movq 8(%rdi), %rax
        movq (%rax), %rdx
        movq (%rdx), %rdx
        subq 8(%rax), %rdx
        movq %rdx, (%rdi)
        ret
```

On the basis of this information, fill in the missing expressions in the code for proc. Hint: Some union references can have ambiguous interpretations. These ambiguities get resolved as you see where the references lead. There is only one answer that does not perform any casting and does not violate any type constraints.

Solution:

A.

We analyze the offsets first:

```c
union ele {
        struct {
                long *p;      // offset - 0
                long y;       // offset - 8
        } e1;

        struct {
                long x;             // offset - 0
                union ele *next;    // offset - 8
        } e2;
};
```

Element	Offset
e1.p	0
e1.y	8
e2.x	0
e2.next	8

B.

16 bytes

C.

Look into the assembly code first:

```
# void proc (union ele *up)
# up in %rdi

proc:
        movq 8(%rdi), %rax     # rax=M(rdi+8): rax = *(up + 8) - up->y or up->next
        movq (%rax), %rdx      # rdx=M(rax): rdx = *(up->next) = up->next->p
        movq (%rdx), %rdx      # rdx=M(rdx): rdx = *(up->next->p)
        subq 8(%rax), %rdx     # rdx=rdx-M(rax+8): rdx = *(up->next->p) - (up->next->y)
        movq %rdx, (%rdi)      # M(rdi)=rdx: up->x = *(up->next->p) - (up->next->y)
        ret
```

Then fill in the code below:

```c
void proc (union ele *up) {
        up->e2.x = *(up->e2.next->e1.p) - up->e2.next->e1.y;
}
```

3.71 ◆

Write a function good_echo that reads a line from standard input and writes it to standard output. Your implementation should work for an input line of arbitrary length. You may use the library function fgets, but you must make sure your function works correctly even when the input line requires more space than you have allocated for your buffer. Your code should also check for error conditions and return when one is encountered. Refer to the definitions of the standard I/O functions for documentation [45, 61].

Solutino:

Get the syntax of `fgets` function:

```c
char* fgets(char *_str_, int _n_, FILE *_stream_);
```

- The `fgets()` reads a line from the specified stream and stores it into the string pointed to by str.
- It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.
- The `fgets()` function returns a pointer to the string where the input is stored.

```c
#include <stdio.h>
#define BUFSIZE 5

void good_echo(void)
{
        char buf[BUFSIZE];
        while (!feof(stdin)) {
                if (fgets(buf, BUFSIZE, stdin) == NULL)  return;
                fputs(buf, stdout);
        }
}

int main()
{
        good_echo();
        return 0;
}
```


3.72 ◆◆

The C code below shows the code for a function that is similar to function vfunct. We used vfunct to illustrate the use of a frame pointer in managing variable-size stack frames. The new function aframe allocates space for local array p by calling library function `alloca`. This function is similar to the more commonly used function `malloc`, except that it allocates space on the run-time stack. The space is automatically deallocated when the executing procedure returns. The assembly code below shows the part of the assembly code that sets up the frame pointer and allocates space for local variables i and p. It is very similar to the corresponding code for vframe. Let us use the same notation as in Problem 3.49: The stack pointer is set to values s_1 at line 7 and s_2 at line 10. The start address of array p is set to value p at line 12. Extra space e_2 may arise between s_2 and p, and extra space e_1 may arise between the end of array p and s_1.

```c
#include <alloca.h>
```

```c

long aframe(long n, long idx, long *q) {
        long i;
        long **p = alloca(n * sizeof(long *));
        p[0] = &i;
        for (i = 1; i < n; i++)
                p[i] = q;
        return *p[idx];
}
```

```
# long aframe(long n, long idx, long *q)
# n in %rdi, idx in %rsi, q in %rdx

aframe:
        pushq %rbp
        movq %rsp, %rbp
        subq $16, %rsp              # Allocate space for i (%rsp = s1)
        leaq 30(,%rdi,8), %rax
        andq $-16, %rax
        subq %rax, %rsp             # Allocate space for array p (%rsp = s2)
        leaq 15(%rsp), %r8
        andq $-16, %r8              # Set %r8 to &p[0]
        ...
```

A. Explain, in mathematical terms, the logic in the computation of s_2.

B. Explain, in mathematical terms, the logic in the computation of p.

C. Find values of n and s_1 that lead to minimum and maximum values of e_1.

D. What alignment properties does this code guarantee for the values of s_2 and p?

Solution:

Reverse the assembly code:

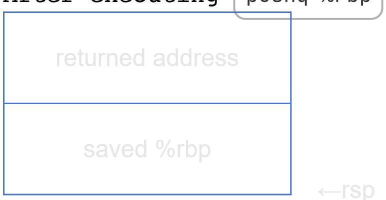
```
# long aframe(long n, long idx, long *q)
# n in %rdi, idx in %rsi, q in %rdx

aframe:
        pushq %rbp
        movq %rsp, %rbp
        subq $16, %rsp              # rsp=rsp-16: Allocate space for i (%rsp = s1)
        leaq 30(,%rdi,8), %rax      # rax=8*rdi+30: rax = 8*n + 0x 1E
        andq $-16, %rax             # rax = rax & 0x FFFF FFF0
        subq %rax, %rsp             # rsp=rsp-rax: Allocate space for array p (%rsp = s2)
        leaq 15(%rsp), %r8          # r8=rsp+15
        andq $-16, %r8              # Set %r8 to &p[0]
        ...
```

Go through all instructions with changing in stack frame step by step:

1. Enter function.

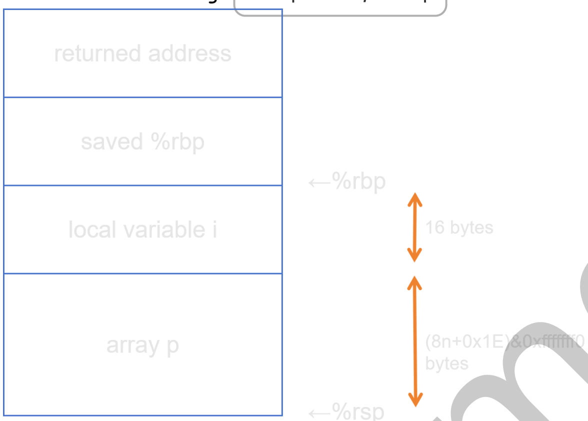
2. After executing `pushq %rbp`



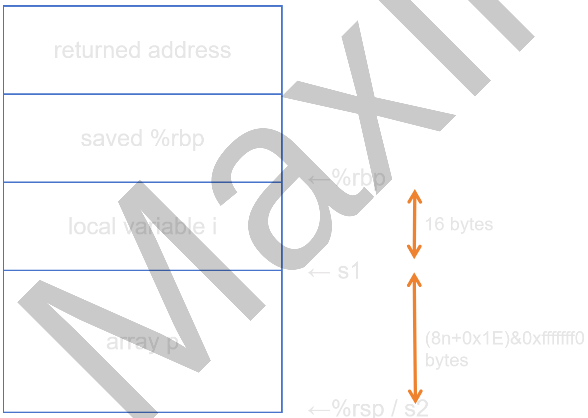
3. After executing `movq %rsp, %rbp`

4. After executing `subq $16, %rsp`

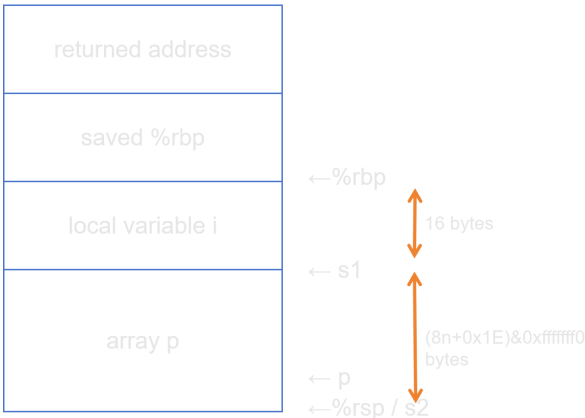
5. After executing `leaq 30(,%rdi,8), %rax`, no change for stack frame.
   We get: `rax = 8*n + 0x 1E`
6. After executing `andq $-16, %rax`, no change for stack frame.
   We get: `rax = (8*n + 0x 1E) & 0xfffffff0`
7. After executing `subq %rax, %rsp`



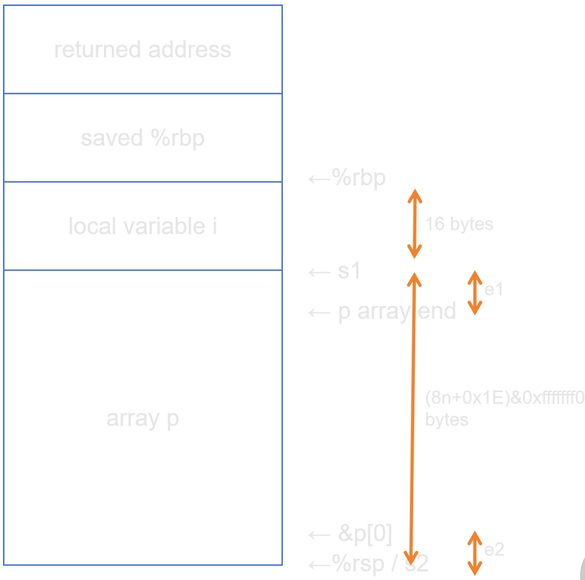
8. Add s_1, s_2 labels



9. After executing `leaq 15(%rsp), %r8`, no change for stack frame.
   We get: `r8=rsp+15`
10. After executing `andq $-16, %r8`, while `-16 = 0x ffff ffff ffff fff0`
    We get: `r8=(rsp+15)&(-16)` for 16 bytes alignment



11. Add e_1, e_2 labels



Consider s1 - s2: ((8n+0x1E)&0xfffffff0)

- If n = 2m + 1 (odd number):
  - s1 - s2 = (8*(2m+1)+30)&0xfffffff0 = (16m+38)&0xfffffff0 = 16m + 32 bytes, while p array roughly takes 16m + 32 bytes.
  - $\&p[0]$ =(s2+15)&(-16) = (s2+15) - (s2+15)%16 for 16 bytes alignment.
  - p_array_end = $\&p[0]$ + 8*n = $\&p[0]$ + 8*(2m+1) + 8 = $\&p[0]$ + 16m + 8.
  - e_2 = $\&p[0]$ - s_2 = 15 - (s2+15)%16 = 15 - (s1 - 16m - 32 + 15)%16 = 15 - (s1 - 16m - 17)%16
  - e_1 = s_1 - p_array_end = s2 + 16m + 32 - ($\&p[0]$ + 16m + 8) = s2 - $\&p[0]$ + 24 = s_2 - (`(s2+15) - (s2+15)%16`) + 24 = 9 + (s2+15)%16
- If n = 2m (even number):
  - s1 - s2 = (8*2m+30)&0xfffffff0 = 16m + 16, while p array takes 16m + 16 bytes
  - $\&p[0]$ = `(s2+15)&0xfffffff0` = `(s2+15) - (s2+15)%16` for 16 bytes alignment
  - p_array_end = ($\&p[0]$ + 8*n) & (-16) = $\&p[0]$ + 16m for 16 bytes alignment
  - e_2 = $\&p[0]$ - s_2 = 15 - (s_2+15)%16
  - e_1 = s_1 - p_array_end = (16m + 16 + s_2) - ($\&p[0]$ + 16m) = s_2 - $\&p[0]$ + 16 = s_2 - (`(s2+15) - (s2+15)%16`) + 16 = 1 + (s_2+15)%16

A.

$s_2 = s_1 - (8 \times n + 30)\&0xFFFFFFF0$

This is to:

- meet 16 bytes' alignment;
- allocate proper stack space for use;

B.

p = (s_2+15) & 0x FFFF FFF0, which is the closest multiples of 16 which is greater than s_2.

C.
The minimum e1: when n is an even number, $e1_{min}$ = 1
The maximum e1: when n is an odd number, $e2_{max} = 24$

D.
16 bytes.

3.73 ◆

Write a function in assembly code that matches the behavior of the function find_range in code below. Your code should contain only one floating-point comparison instruction, and then it should use conditional branches to generate the correct result. Test your code on all 2^{32} possible argument values. Web Aside asm:easm on page 214 describes how to incorporate functions written in assembly code into C programs.

- C code:

```c
typedef enum {NEG, ZERO, POS, OTHER} range_t; // 0 (NEG), 1 (ZERO), 2 (POS), and 3 (OTHER)

range_t find_range(float x)
{
        int result;
        if (x < 0)
                result = NEG;
        else if (x == 0)
                result = ZERO;
        else if (x > 0)
                result = POS;
        else
                result = OTHER;
        return result;
}
```

- Assembly code:

```
# range_t find_range(float x)
# x in %xmm0

find_range:
        vxorps %xmm1, %xmm1, %xmm1          # Set %xmm1 = 0
        vucomiss %xmm0, %xmm1               # Compare 0:x
        ja .L5                              # If >, goto neg
        vucomiss %xmm1, %xmm0               # Compare x:0
        jp .L8                              # If NaN, goto posornan
        movl $1, %eax                       # result = ZERO
        je .L3                              # If =, goto done

.L8:                                    # posornan:
        vucomiss .LC0(%rip), %xmm0          # Compare x:0
        setbe %al                           # Set result = NaN?1:0
        movzbl %al, %eax                    # Zero-extend
        addl $2, %eax                       # result += 2 (POS for > 0, OTHER for NaN)
        ret                                 # Return

.L5:                                    # neg:
        movl $0, %eax                       # result = NEG

.L3:                                    # done:
        rep; ret                            # Return
```

Solution:
First, let's go through the compare instruction `vucomiss` from intel manual:

- UCOMISS — Unordered Compare Scalar Single Precision Floating-Point Values and Set EFLAGS.

```
RESULT := UnorderedCompare(DEST[31:0] <> SRC[31:0]) {
        (* Set EFLAGS *) CASE (RESULT) OF
        UNORDERED: ZF,PF,CF := 111;
        GREATER_THAN: ZF,PF,CF := 000;
        LESS_THAN: ZF,PF,CF := 001;
        EQUAL: ZF,PF,CF := 100;
        ESAC;
        OF, AF, SF := 0; }
```

    - We get that by using `UCOMISS` instructions, only when `PF == 1` means it's unordered.

Secondly, let's go through the conditional jump instruction `jp` from intel manual: `JP - Jump near if parity (PF=1)`
Thirdly, let's write the assembly code below:

```
# range_t find_range(float x)
# x in %xmm0
# typedef enum {NEG, ZERO, POS, OTHER}

find_range:
        vxorps %xmm1, %xmm1, %xmm1          # Set %xmm1 = 0
        vucomiss %xmm1, %xmm0               # Compare x:0
        jp .other                           # if unordered
        ja .pos                             # if x > 0
        je .zero                            # if x == 0
        jb .neg                             # if x < 0

.other:
        movl $3, %eax                       # eax = OTHER
        jmp .done
.pos:
        movl $2, %eax                       # eax = POS
        jmp .done
.zero:
        movl $1, %eax                       # eax = ZERO
        jmp .done
.neg
        xorl %eax, %eax                     # eax = NEG
.done:                                  # done:
        rep; ret                            # Return
```

Fourthly, let's do inline assembly code in C code:
How to do inline assembly code - please refer to my videos about assembly languages.

```c
#include <stdio.h>
typedef enum {NEG, ZERO, POS, OTHER} range_t;

range_t find_range(float x)
{
        __asm__(
        "vxorps %xmm1, %xmm1, %xmm1\n\t"
        "vucomiss %xmm1, %xmm0\n\t"
        "jp .other\n\t"
        "ja .pos\n\t"
        "je .zero\n\t"
        "jb .neg\n\t"
        ".other:\n\t"
        "movl $3, %eax\n\t"
        "jmp .done\n\t"
        ".pos:\n\t"
        "movl $2, %eax\n\t"
```

```
		"jmp .done\n\t"
		".zero:\n\t"
		"movl $1, %eax\n\t"
		"jmp .done\n\t"
		".neg\n\t"
		"xorl %eax, %eax\n\t"
		".done:\n\t"
		"rep; ret\n\t"
		)
}

```

Finally, let's the testing code, including testing all 2^{32} numbers.

```
#include <stdio.h>
#include <limits.h>
#include <assert.h>

typedef enum {NEG, ZERO, POS, OTHER} range_t;

/* Access bit-level representation floating-point number */
typedef unsigned float_bits;

range_t find_range(float x)
{
	__asm__(
		"vxorps %xmm1, %xmm1, %xmm1\n\t"
		"vucomiss %xmm1, %xmm0\n\t"
		"jp .other\n\t"
		"ja .pos\n\t"
		"je .zero\n\t"
		"jb .neg\n\t"
		".other:\n\t"
		"movl $3, %eax\n\t"
		"jmp .done\n\t"
		".pos:\n\t"
		"movl $2, %eax\n\t"
		"jmp .done\n\t"
		".zero:\n\t"
		"movl $1, %eax\n\t"
		"jmp .done\n\t"
		".neg:\n\t"
		"xorl %eax, %eax\n\t"
		".done:\n\t"
		"rep; ret\n\t"
	);
}


float u2f(unsigned x)
{
	unsigned* p_x = &x;
	return *(float*)p_x;
}

int main()
{
	unsigned u = 0;
	while (u <= UINT_MAX)
	{
		float f = u2f(u);
		unsigned range;

		if(f < 0) {
			range = find_range(f);
			assert(NEG == range);
		}
		else if(f == 0) {
			range = find_range(f);
			assert(ZERO == range);
		}
		else if(f > 0){
			range = find_range(f);
```

```c
                        assert(POS == range);
                }
                else {
                        range = find_range(f);
                        assert(OTHER == range);
                }
                u = u + 0x1000;    // to cut down testing cost
                printf("Test passed on 0x%g \n", f);
        }
        return 0;
}
```

- **Compile:** `gcc -Og -m32 3_73.c -o 3_73`

3.74 ◆◆

Write a function in assembly code that matches the behavior of the function find_range in Figure 3.51. Your code should contain only one floating-point comparison instruction, and then it should use conditional moves to generate the correct result. You might want to make use of the instruction `cmovp` (move if even parity). Test your code on all 2^{32} possible argument values. Web Aside asm:easm on page 214 describes how to incorporate functions written in assembly code into C programs.

Solution:
Firstly, let's modify the asm code for homework 3.37

```
# range_t find_range(float x)
# x in %xmm0
# typedef enum {NEG, ZERO, POS, OTHER} range_t

find_range:
        vxorps %xmm1, %xmm1, %xmm1          # Set %xmm1 = 0
        vucomiss %xmm1, %xmm0               # Compare x:0
        cmovpq   other, %eax                # if unordered and neg is local variable
        cmovaq   pos, %eax                  # if x > 0
        cmoveq   zero, %eax                 # if x == 0
        cmovbq   neg, %eax                  # if x < 0
        rep; ret                            # Return
```

Secondly, we make it into C code as inline asm:

```c
#include <stdio.h>
typedef enum {NEG, ZERO, POS, OTHER} range_t;

range_t find_range(float x)
{
        int result;
        int other = OTHER;
        int pos   = POS;
        int zero  = ZERO;
        int neg   = NEG;
        asm(
        "vxorps   %%xmm1, %%xmm1, %%xmm1\n\t"
        "vucomiss %%xmm1, %%xmm0\n\t"
        "cmovpq   %1, %%eax\n\t"
        "cmovaq   %2, %%eax\n\t"
        "cmoveq   %3, %%eax\n\t"
        "cmovbq   %4, %%eax\n\t"
        "movl     %%eax, %0\n\t"
        : "=a"(result)
        : "m"(other), "m"(pos), "m"(zero), "m"(neg)
        );
        return result;
}

```

```c
/* memtest_x64.c - An example of using memory locations as values */

#include <stdio.h>

int main()
{
	long dividend = 20;
	long divisor = 5;
	long result;

	asm("divb %2\n\t"
		"movq %%rax, %0"
		: "=m"(result)       // m - local variable result
		: "a"(dividend), "m"(divisor));  // a - eax, m - local variable divisor

	printf("The result is %ld\n", result);
	return 0;
}
```

Thirdly, let's construct the full code:

```c
#include <stdio.h>
#include <limits.h>
#include <assert.h>

typedef enum {NEG, ZERO, POS, OTHER} range_t;

/* Access bit-level representation floating-point number */
typedef unsigned float_bits;

range_t find_range(float x)
{
	int result;
	int other = OTHER;
	int pos   = POS;
	int zero  = ZERO;
	int neg   = NEG;
	asm(
	"vxorps   %%xmm1, %%xmm1, %%xmm1\n\t"
	"vucomiss %%xmm1, %%xmm0\n\t"
	"cmovpq   %1, %%eax\n\t"
	"cmovaq   %2, %%eax\n\t"
	"cmoveq   %3, %%eax\n\t"
	"cmovbq   %4, %%eax\n\t"
	"movl     %%eax, %0\n\t"
	: "=a"(result)
	: "m"(other), "m"(pos), "m"(zero), "m"(neg)
	);
	return result;
}


float u2f(unsigned x)
{
	unsigned* p_x = &x;
	return *(float*)p_x;
}

int main()
{
	unsigned u = 0;
	while (u <= UINT_MAX)
	{
		float f = u2f(u);
		unsigned range;

		if(f < 0) {
			range = find_range(f);
			assert(NEG == range);
		}
		else if(f == 0) {
			range = find_range(f);
			assert(ZERO == range);
```

```
			}
			else if(f > 0){
				range = find_range(f);
				assert(POS == range);
			}
			else {
				range = find_range(f);
				assert(OTHER == range);
			}
			u = u + 0x1000;    // to cut down testing cost
			printf("Test passed on 0x%g \n", f);
		}
		return 0;
	}
```

- Compile and run: `gcc -Og 3_74.c -o 3_74`

3.75 ◆

ISO C99 includes extensions to support complex numbers. Any floating-point type can be modified with the keyword `complex`. Here are some sample functions that work with complex data and that call some of the associated library functions:

```
#include <complex.h>

double c_imag(double complex x) {
	return cimag(x);
}

double c_real(double complex x) {
	return creal(x);
}

double complex c_sub(double complex x, double complex y) {
	return x - y;
}
```

When compiled, gcc generates the following assembly code for these functions:

```
# double c_imag(double complex x)
c_imag:
	movapd %xmm1, %xmm0
	ret

# double c_real(double complex x)
c_real:
	rep; ret

# double complex c_sub(double complex x, double complex y)
c_sub:
	subsd %xmm2, %xmm0
	subsd %xmm3, %xmm1
	ret
```

Based on these examples, determine the following:
A. How are complex arguments passed to a function?
B. How are complex values returned from a function?

Solution:
Firstly, let's find out what is `complex numbers`.

- "complex numbers" refers to numbers that have both a real part and an imaginary part.
- A complex number is typically written in the form (a + bi)
  (a) is the real part

(b) is the imaginary part

(i) is the imaginary unit with the property that (i^2 = -1).

a and b are both real number

So we get that, as a complex number, when doing parameters transferring, basically it's just passing a and b.

Therefore, according to the assembly code above. We can get the result as below:

A.

%xmm0, %xmm2 are for passing parameter a.

%xmm1, %xmm3 are for passing parameter b.

B.

return %xmm0 for real part (a)

return %xmm1 for img part (b)