3_homework

3.58 *

For a function with prototype

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

gcc generates the following assembly code:

```
1    decode2:
2          subq %rdx, %rsi
3          imulq %rsi, %rdi
4          movq %rsi, %rax
5          salq $63, %rax
6          sarq $63, %rax
7          xorq %rdi, %rax
8          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:

```
1    # long decode2(long x, long y, long z)
2    # rdi - x, rsi - y, rdx - z
3    decode2:
4          subq %rdx, %rsi    # rsi=rsi-rdx: rsi = y - z
5          imulq %rsi, %rdi   # rdi=rdi*rsi: rdi = x * (y - z)
6          movq %rsi, %rax    # rax=rsi: rax = y - z
7          salq $63, %rax     # rax=rax<<63: rax = (y-z)<<63
8          sarq $63, %rax     # rax=rax>>63: rax = ((y-z)<<63)>>63
9          xorq %rdi, %rax    # rax=rax^rdi: rax = ((y-z)<<63)>>63 ^ (x * (y - z))
10         ret
```

Thus the C code:

```
1    long decode2(long x, long y, long z)
2    {
3          return ((y-z)<<63)>>63 ^ (x*(y-z));
4    }
```

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:

```
1    typedef __int128 int128_t;
2
3    void store_prod(int128_t *dest, int64_t x, int64_t y) {
4          *dest = x * (int128_t)y;
5    }
```

Gcc generates the following assembly code implementing the computation:

```
1    store_prod:
2            movq    %rdx, %rax
3            cqto
4            movq    %rsi, %rcx
5            sarq    $63, %rcx
6            imulq   %rax, %rcx
7            imulq   %rsi, %rdx
8            addq    %rdx, %rcx
9            mulq    %rsi
10           addq    %rcx, %rdx
11           movq    %rax, (%rdi)
12           movq    %rdx, 8(%rdi)
13           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 theory:
  - For bit representation $[x_0, x_1 \ldots x_{63}]$, x is its signed value and ux is its unsigned value.
  - Then $ux = x + 2^{64} * x_{63}$ while x_{63} is the sign bit value of x.
  - So we also get same for y: $uy = y + 2^{64} * y_{63}$
  - Combined to get $x * y$: $x * y = (ux - 2^{64} * x_{63}) * (uy - 2^{64} * y_{63}) = ux * uy - (x_{63} * uy + y_{63} * ux)2^{64}$
  - Ok, now we get the formula, let's go through the assembly code.

- Analyze the assembly code:

```
1    # void store_prod(int128_t *dest, int64_t x, int64_t y)
2    # rdi - dest, rsi - x, rdx - y
3    store_prod:
4            movq    %rdx, %rax     # rax=rdx: rax = y
5            cqto                   # convert 64bits in rax to 128bits:
6                                   # rdx = y >> 63
7                                   # rax = y
8            movq    %rsi, %rcx     # rcx=rsi: rcx = x
9            sarq    $63, %rcx      # rcx=rcx>>63: rcx = x>>63 - get the sign bit of x which is x63
10           imulq   %rax, %rcx     # rcx=rcx*rax: rcx = y * (x>>63)
11           imulq   %rsi, %rdx     # rdx=rdx*rsi: rdx = (y>>63) * x
12           addq    %rdx, %rcx     # rcx=rcx+rdx: rcx = (y*(x>>63)) + ((y>>63)*x)
13           mulq    %rsi           # rdx:rax=rax*rsi: rdx:rax = uy * ux
14           addq    %rcx, %rdx     # rdx=rdx+rcx: rdx = rdx + (y*(x>>63)) + ((y>>63)*x) - high64bits
    operation
15           movq    %rax, (%rdi)   # M(rdi)=rax: *dest = rax - set low 64 bits
16           movq    %rdx, 8(%rdi)  # M(rdi+8)=rdx: *(dest+8) = rdx - set high 64 bits
17           ret
```

3.60 **

Consider the following assembly code:

```
1    # long loop(long x, int n)
2    # x in %rdi, n in %esi
3
```

```
 4    loop:
 5            movl %esi, %ecx
 6            movl $1, %edx
 7            movl $0, %eax
 8            jmp .L2
 9
10    .L3:
11            movq %rdi, %r8
12            andq %rdx, %r8
13            orq %r8, %rax
14            salq %cl, %rdx
15
16    .L2:
17            testq %rdx, %rdx
18            jne .L3
19            rep; ret
```

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

```
 1    long loop(long x, long n)
 2    {
 3            long result = _____;
 4            long mask;
 5            for (mask = _____; mask _____; mask = _____) {
 6                    result |= _____;
 7            }
 8            return result;
 9    }
```

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:

```
1    # long loop(long x, int n)
2    # x in %rdi, n in %esi
3
4    loop:
5            movl %esi, %ecx      # ecx=esi: ecx = n
6            movl $1, %edx        # edx=1: edx = 1
7            movl $0, %eax        # eax=0: eax = 0
8            jmp .L2
9
10   .L3:
11           movq %rdi, %r8       # r8=rdi: r8 = x
12           andq %rdx, %r8       # r8=r8&rdx: r8 = x & rdx
13           orq %r8, %rax        # rax=rax|r8: rax = rax | (x & rdx)
14           salq %cl, %rdx       # rdx=rdx>>cl
15
16   .L2:
17           testq %rdx, %rdx
18           jne .L3              # if rdx!=0, goto .L3 - rdx is mask
19           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.

```
1    long loop(long x, long n)
2    {
3            long result = 0;
4            long mask;
5            for (mask = 1; mask != 0; mask = mask << n) {
6                    result |= (x & mask);
7            }
8            return result;
9    }
```

3.61 **

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

```
1    long cread(long *xp) {
2            return (xp ? *xp : 0);
3    }
```

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:

```
1    # long cread(long *xp)
2    # Invalid implementation of function cread
3    # xp in register %rdi
4
5    cread:
6          movq (%rdi), %rax      # v = *xp
7          testq %rdi, %rdi       # Test x
8          movl $0, %edx          # Set ve = 0
9          cmove %rdx, %rax       # If x==0, v = ve
10         ret                    # Return v
```

which is:

```
1    long cread(long *xp) {
2          v = *xp;
3          if x = 0
4          ve = 0
5          v = ve
6          return v;
7    }
```

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

If we change the source code to be:

```
1    long cread(long *xp) {
2          return (!xp ? 0 : *xp);
3    }
```

Then the pseudo-code would be:

```
1    long cread(long *xp) {
2          v = 0;
3          if !xp = 0
4          v = (*xp)
5          return v;
6    }
```

So the assembly code would be:

```
1    cread:
2          movq    $0, %rax        # v = 0
3          testq   %rdi, %rdi      # Test x
4          cmovne  (%rdi), %rax    # If x==0, v = ve
5          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.

```asm
    # 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:

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

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

.L3:            # MODE_A
```

```
 8              movq (%rsi), %rax        # rax = M(rsi): rax = *(p2)
 9              movq (%rdi), %rdx        # rdx = M(rdi): rdx = *(p1)
10              movq %rdx, (%rsi)        # M(rsi) = rdx: *(p2) = rdx = *(p1)
11              ret
12
13      .L5:             # MODE_B
14              movq (%rdi), %rax        # rax = M(rdi): rax = *(p1)
15              addq (%rsi), %rax        # rax = rax + M(rsi): rax = *(p2) + *(p1)
16              movq %rax, (%rdi)        # M(rdi) = rax: *p1 = rax = *(p2) + *(p1)
17              ret
18
19      .L6:             # MODE_C
20              movq $59, (%rdi)         # M(rdi) = 59: (*p1) = 59
21              movq (%rsi), %rax        # rax = M(rsi): rax = *p2
22              ret
23
24      .L7:             # MODE_D
25              movq (%rsi), %rax        # rax = M(rsi): rax = *p2
26              movq %rax, (%rdi)        # M(rdi) = rax: (*p1) = rax = (*p2)
27              movl $27, %eax           # eax = 27
28              ret
29
30      .L9:             # default
31              movl $12, %eax           # eax = 12
32              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:

```
1    long switch_prob(long x, long n) {
2          long result = x;
3          switch(n) {
4                  /* Fill in code here */
5          }
6          return result;
7    }
```

Figure below shows the disassembled machine code for the procedure:

```
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
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
19         4005b5: 48 c1 e0 04              shl  $0x4,%rax               # rax=rax<<4: result =
   x*16
20         4005b9: 48 29 f8                 sub  %rdi,%rax               # rax=rax-rdi: result =
   15*x
21         4005bc: 48 89 c7                 mov  %rax,%rdi               # rdi=rax: rdi = 15*x
22
23         4005bf: 48 0f af ff              imul %rdi,%rdi               # sharedcode -
   rdi=rdi*rdi
24         4005c3: 48 8d 47 4b              lea  0x4b(%rdi),%rax         # sharedcode - if rsi=1,
   rax=rdi+0x4b                                               # rax = rdi + 0x4b
25                                                                      # sharedcode - if n -
   0x3c = 1, result = x + 0x4b
26         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:

9

```
1    # long store_ele(long i, long j, long k, long *dest)
2    # i in %rdi, j in %rsi, k in %rdx, dest in %rcx
3
4    store_ele:
5            leaq (%rsi,%rsi,2), %rax
6            leaq (%rsi,%rax,4), %rax
7            movq %rdi, %rsi
8            salq $6, %rsi
9            addq %rsi, %rdi
10           addq %rax, %rdi
11           addq %rdi, %rdx
12           movq A(,%rdx,8), %rax
13           movq %rax, (%rcx)
14           movl $3640, %eax
15           ret
```

A. Extend Equation 3.1 from two dimensions to three to provide a formula for the location
of array element $\boxed{\texttt{A[i][j][k]}}$.

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.

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

Thus, we get $\boxed{\texttt{*dest = A + 8(k + 65i + 13j)}}$
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 $\boxed{\texttt{#define}}$:

```
1    void transpose(long A[M][M]) {
2          long i, j;
3          for (i = 0; i < M; i++)
4                for (j = 0; j < i; j++) {
5                      long t = A[i][j];
6                      A[i][j] = A[j][i];
7                      A[j][i] = t;
8                }
9    }
```

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

```
1    .L6:
2          movq (%rdx), %rcx
3          movq (%rax), %rsi
4          movq %rsi, (%rdx)
5          movq %rcx, (%rax)
6          addq $8, %rdx
7          addq $120, %rax
8          cmpq %rdi, %rax
9          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:

```
1    # void transpose(long A[M][M])
2    # rdi - A
3    .L6:
4          movq (%rdx), %rcx     # rcx=M(rdx): rcx = A[i][j]
5          movq (%rax), %rsi     # rsi=M(rax): rsi = A[j][i]
6          movq %rsi, (%rdx)     # M(rdx)=rsi:
7          movq %rcx, (%rax)     # M(rax)=rcx:
8          addq $8, %rdx         # rdx=rdx+8: (&A[i][j])++
9          addq $120, %rax       # rax=rax+120: &A[j][i] = &A[j][i] + j
10         cmpq %rdi, %rax
11         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.

```
1    long sum_col(long n, long A[NR(n)][NC(n)], long j) {
2          long i;
3          long result = 0;
```

```
4          for (i = 0; i < NR(n); i++)
5               result += A[i][j];
6          return result;
7    }
```

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

```
1    # long sum_col(long n, long A[NR(n)][NC(n)], long j)
2    # n in %rdi, A in %rsi, j in %rdx
3
4    sum_col:
5            leaq   1(,%rdi,4), %r8
6            leaq   (%rdi,%rdi,2), %rax
7            movq   %rax, %rdi
8            testq  %rax, %rax
9            jle    .L4
10           salq   $3, %r8
11           leaq   (%rsi,%rdx,8), %rcx
12           movl   $0, %eax
13           movl   $0, %edx
14
15   .L3:
16           addq   (%rcx), %rax
17           addq   $1, %rdx
18           addq   %r8, %rcx
19           cmpq   %rdi, %rdx
20           jne    .L3
21           rep; ret
22
23   .L4:
24           movl   $0, %eax
25           ret
```

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

Solution:

Look into the assembly code:

```
1    # long sum_col(long n, long A[NR(n)][NC(n)], long j)
2    # n in %rdi, A in %rsi, j in %rdx
3
4    sum_col:
5            leaq   1(,%rdi,4), %r8        # r8=4*rdi+1: r8 = 4n + 1
6            leaq   (%rdi,%rdi,2), %rax    # rax=3*rdi: rax = 3n
7            movq   %rax, %rdi             # rdi=rax: rdi = 3n
8            testq  %rax, %rax
9            jle    .L4                    # if 3n <= 0, goto .L4
10           salq   $3, %r8                # r8=r8<<3: r8 = 8*r8 = 8*(4n+1)
11           leaq   (%rsi,%rdx,8), %rcx    # rcx=rsi+8*rdx: rcx = A + 8j
12           movl   $0, %eax               # eax = 0
13           movl   $0, %edx               # edx = 0
14
15   .L3:
16           addq   (%rcx), %rax           # rax+=M(rcx): rax = rax + *(A+8j)
17           addq   $1, %rdx               # rdx+=1: i++
18           addq   %r8, %rcx              # rcx+=r8: rcx = rcx + 8*(4n+1)
19           cmpq   %rdi, %rdx
20           jne    .L3                    # if rdx!=rdi, goto .L3
21           rep; ret
22
23   .L4:
24           movl   $0, %eax
25           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:

```
1    typedef struct {
2            long a[2];
3            long *p;
4    } strA;
5
6    typedef struct {
7            long u[2];
8            long q;
9    } strB;
10
11   strB process(strA s) {
12           strB r;
13           r.u[0] = s.a[1];
14           r.u[1] = s.a[0];
15           r.q = *s.p;
16           return r;
17   }
18
19   long eval(long x, long y, long z) {
20           strA s;
21           s.a[0] = x;
22           s.a[1] = y;
23           s.p = &z;
24           strB r = process(s);
25           return r.u[0] + r.u[1] + r.q;
26   }
```

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)
```

13

```
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:

```
1    strB process(strA s) {
```

```
2          strB r;
3          r.u[0] = s.a[1];
4          r.u[1] = s.a[0];
5          r.q = *s.p;
6          return r;
7      }
```

Look into the process assembly language:

```
1    # strB process(strA s)
2    # the eval pass s=rsp+64 as parameter
3    process:
4          movq %rdi, %rax        # rax=rdi: rax = rsp+72
5          movq 24(%rsp), %rdx    # rdx=M(rsp+24): rdx = &z
6          movq (%rdx), %rdx      # rdx=M(rdx): rdx = z
7          movq 16(%rsp), %rcx    # rcx=M(rsp+16): rcx = y
8          movq %rcx, (%rdi)      # M(rdi)=rcx: M(rsp+72) = r.u[0] = y
9          movq 8(%rsp), %rcx     # rcx=M(rsp+8): rcx = x
10         movq %rcx, 8(%rdi)     # M(rdi+8)=rcx: M(rsp+80) = r.u[1] = x
11         movq %rdx, 16(%rdi)    # M(rdi+16)=rdx: M(rsp+88) = r.q = z
12         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:

```
1          movq 72(%rsp), %rax    # rax = M(rsp+72) = x
2          addq 64(%rsp), %rax    # rax += M(rsp+64) = y: rax = x + y
3          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`:

```
1    typedef struct {
2            int x[A][B]; /* Unknown constants A and B */
3            long y;
4    } str1;
5
6    typedef struct {
7            char array[B];
8            int t;
9            short s[A];
10           long u;
11   } str2;
12
13   void setVal(str1 *p, str2 *q) {
14           long v1 = q->t;
15           long v2 = q->u;
16           p->y = v1+v2;
17   }
```

Gcc generates the following code for setVal:

```
1    # void setVal(str1 *p, str2 *q)
2    # p in %rdi, q in %rsi
3
4    setVal:
5            movslq 8(%rsi), %rax
6            addq 32(%rsi), %rax
7            movq %rax, 184(%rdi)
8            ret
```

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

Solution:

Look into the assembly code first:

```
1    # void setVal(str1 *p, str2 *q)
2    # p in %rdi, q in %rsi
3
4    setVal:
5            movslq   8(%rsi), %rax     # rax=M(rsi+8): rax = *(q+8)
6            addq     32(%rsi), %rax    # rax=rax+M(rsi+32): rax = rax + *(q+32)
7            movq     %rax, 184(%rdi)   # M(rdi+184)=rax: *(p+184) = rax
8            ret
```

Compare with the C code:

```
1    void setVal(str1 *p, str2 *q) {
2            long v1 = q->t;
3            long v2 = q->u;
4            p->y = v1+v2;
5    }
```

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:

```
1    typedef struct {
2            char array[B];
3            int t;
```

```
4            short s[A];
5            long u;
6    } 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`.

```
1    typedef struct {
2            int x[A][B]; /* Unknown constants A and B */
3            long y;
4    } 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:

```
1    typedef struct {
2            int first;
3            a_struct a[CNT];
4            int last;
5    } b_struct;
6
7    void test(long i, b_struct *bp)
8    {
9            int n = bp->first + bp->last;
10           a_struct *ap = &bp->a[i];
11           ap->x[ap->idx] = n;
12   }
```

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:

```
1    0000000000000000 <test>:
2           0: 8b 8e 20 01 00 00    mov 0x120(%rsi), %ecx
3           6: 03 0e                add (%rsi), %ecx
4           8: 48 8d 04 bf          lea (%rdi, %rdi, 4), %rax
5           c: 48 8d 04 c6          lea (%rsi, %rax, 8), %rax
6           10: 48 8b 50 08         mov 0x8(%rax), %rdx
7           14: 48 63 c9            movslq %ecx, %rcx
8           17: 48 89 4c d0 10      mov %rcx, 0x10(%rax, %rdx, 8)
9           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:

```
1   typedef struct {
2           int first;
3           a_struct a[CNT];
4           int last;
5   } b_struct;
```

```
1    # void test(long i, b_struct *bp)
2    # rdi - i
3    # rsi - bp
4
5    0000000000000000 <test>:
6            0:  mov 0x120(%rsi), %ecx        # ecx=M(rsi+0x120): ecx = bp->last
7            6:  add (%rsi), %ecx             # ecx=ecx+M(rsi): ecx = bp->last + bp->first
8            8:  lea (%rdi, %rdi, 4), %rax    # rax=5*rdi: rax = 5i
9            c:  lea (%rsi, %rax, 8), %rax    # rax=rsi+8*rax: rax = bp + 8*5i
10           10: mov 0x8(%rax), %rdx          # rdx=M(rax+8): rdx = M(bp + 8*5i + 8) = bp-
     >a[i].idx = ap->idx
11           14: movslq %ecx, %rcx            # rcx=ecx
12           17: mov %rcx, 0x10(%rax, %rdx, 8)   # M(rax+8*rdx+0x10)=rcx
13           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+8rdx+0x10` is `&bp->a[i].x + 8rdx` , which is `&ap->x + 8rdx` , 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:

```
1    typedef struct
2    {
3            long idx;
4            long x[4];
5    } a_struct;
```

3.70 ◆◆◆

Consider the following union declaration:

```
1    union ele {
2            struct {
3                    long *p;
4                    long y;
5            } e1;
6
7            struct {
8                    long x;
```

```
 9              union ele *next;
10          } e2;
11      };
```

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:

```
1    void proc (union ele *up) {
2         up-> _____ = *( _____ ) - _____ ;
3    }
```

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

```
1    e1.p          _____
2    e1.y          _____
3    e2.x          _____
4    e2.next       _____
```

B. How many total bytes does the structure require?

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

```
1    # void proc (union ele *up)
2    # up in %rdi
3
4    proc:
5         movq 8(%rdi), %rax
6         movq (%rax), %rdx
7         movq (%rdx), %rdx
8         subq 8(%rax), %rdx
9         movq %rdx, (%rdi)
10        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:

```
1    union ele {
2         struct {
3              long *p;     // offset - 0
4              long y;      // offset - 8
5         } e1;
6
7         struct {
8              long x;             // offset - 0
9              union ele *next;    // offset - 8
10        } e2;
11   };
```

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

B.

16 bytes

C.

Look into the assembly code first:

```
1    # void proc (union ele *up)
2    # up in %rdi
3
4    proc:
5            movq 8(%rdi), %rax      # rax=M(rdi+8): rax = *(up + 8) - up->y or up->next
6            movq (%rax), %rdx       # rdx=M(rax): rdx = *(up->next) = up->next->p
7            movq (%rdx), %rdx       # rdx=M(rdx): rdx = *(up->next->p)
8            subq 8(%rax), %rdx      # rdx=rdx-M(rax+8): rdx = *(up->next->p) - (up->next->y)
9            movq %rdx, (%rdi)       # M(rdi)=rdx: up->x = *(up->next->p) - (up->next->y)
10           ret
```

Then fill in the code below:

```
1    void proc (union ele *up) {
2            up->e2.x = *(up->e2.next->e1.p) - up->e2.next->e1.y;
3    }
```

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:

```
1    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.

```
1    #include <stdio.h>
2    #define BUFSIZE 5
3
4    void good_echo(void)
5    {
6            char buf[BUFSIZE];
7            while (!feof(stdin)) {
8                    if (fgets(buf, BUFSIZE, stdin) == NULL)  return;
9                    fputs(buf, stdout);
10           }
11   }
12
13   int main()
14   {
15           good_echo();
16           return 0;
17   }
```

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.

```
1    #include <alloca.h>
2
3    long aframe(long n, long idx, long *q) {
4          long i;
5          long **p = alloca(n * sizeof(long *));
6          p[0] = &i;
7          for (i = 1; i < n; i++)
8                  p[i] = q;
9          return *p[idx];
10   }
```

```
1    # long aframe(long n, long idx, long *q)
2    # n in %rdi, idx in %rsi, q in %rdx
3
4    aframe:
5          pushq %rbp
6          movq %rsp, %rbp
7          subq $16, %rsp              # Allocate space for i (%rsp = s1)
8          leaq 30(,%rdi,8), %rax
9          andq $-16, %rax
10         subq %rax, %rsp             # Allocate space for array p (%rsp = s2)
11         leaq 15(%rsp), %r8
12         andq $-16, %r8              # Set %r8 to &p[0]
13         ...
```

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?
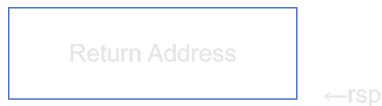
Solution:

Reverse the assembly code:

```
1    # long aframe(long n, long idx, long *q)
2    # n in %rdi, idx in %rsi, q in %rdx
3
4    aframe:
5          pushq %rbp
6          movq %rsp, %rbp
7          subq $16, %rsp              # rsp=rsp-16: Allocate space for i (%rsp = s1)
8          leaq 30(,%rdi,8), %rax      # rax=8*rdi+30: rax = 8*n + 0x 1E
9          andq $-16, %rax             # rax = rax & 0x FFFF FFF0
10         subq %rax, %rsp             # rsp=rsp-rax: Allocate space for array p (%rsp = s2)
11         leaq 15(%rsp), %r8          # r8=rsp+15
12         andq $-16, %r8              # Set %r8 to &p[0]
13         ...
```
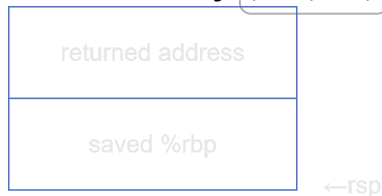
Go through all instructions with changing in stack frame step by step:

1. Enter function.

Return Address
←rsp

2. After executing `pushq %rbp`

returned address
saved %rbp
←rsp

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

returned address
saved %rbp
←%rsp / %rbp

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

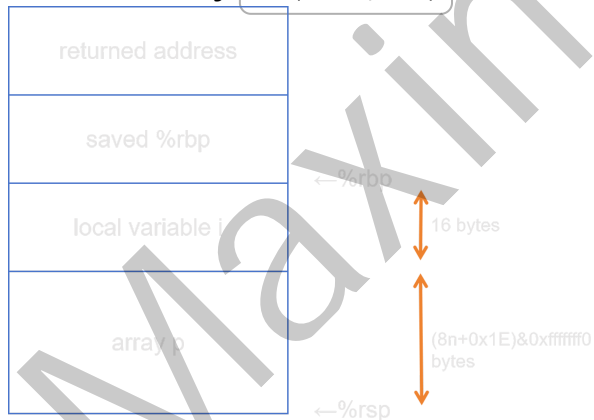| |
|---|
| returned address |
| saved %rbp |
←%rbp
| local variable i |
16 bytes
←%rsp

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`

returned address
saved %rbp
←%rbp
| local variable i |
16 bytes
| array p |
(8n+0x1E)&0xfffffff0 bytes
←%rsp

8. Add s_1, s_2 labels

returned address
saved %rbp
←%rbp
| local variable i |
16 bytes
← s1
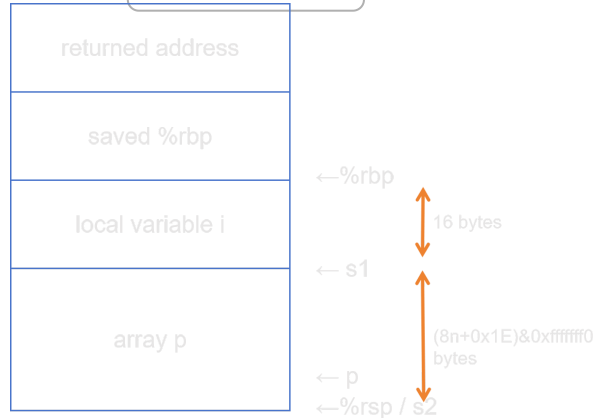| array p |
(8n+0x1E)&0xfffffff0 bytes
←%rsp / s2
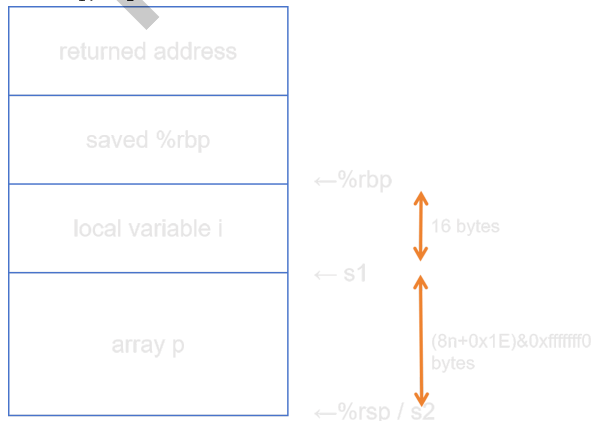
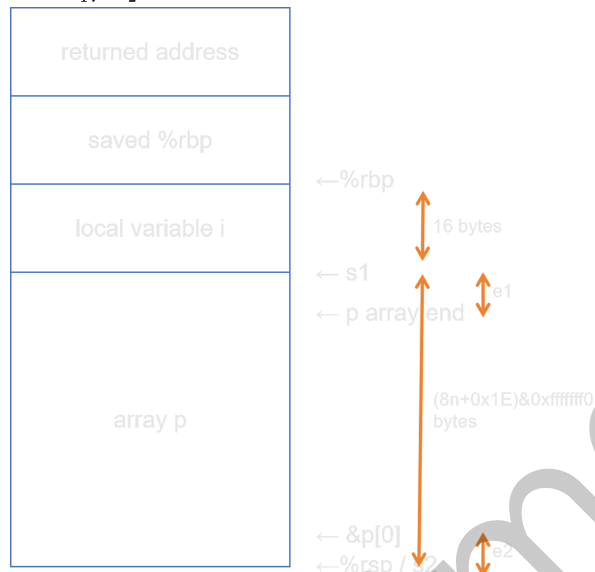9. After executing `leaq 15(%rsp), %r8`, no change for stack frame.
   We get: `r8=rsp+15`

10. After executing `andq $-16, %r8`, while `-16 = 0x ffff ffff ffff fff0`
    We get: `r8=(rsp+15)&(-16)` for 16 bytes alignment

returned address
saved %rbp
local variable i
array p

←%rbp

← s1 — 16 bytes

← p
←%rsp / s2 — (8n+0x1E)&0xfffffff0 bytes

11. Add e_1, e_2 labels

returned address
saved %rbp
local variable i
array p

←%rbp

← s1 — 16 bytes

← p array end — e1

← &p[0]
←%rsp / s2 — e2   (8n+0x1E)&0xfffffff0 bytes

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 – ((s2+15) - (s2+15)%16) + 16 = 1 + (s_2+15)%16

A.
$s_2 = s_1 - (8 \times n + 30) \& 0xFFFFFFFF0$

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:

```
1    typedef enum {NEG, ZERO, POS, OTHER} range_t; // 0 (NEG), 1 (ZERO), 2 (POS), and 3 (OTHER)
2
3    range_t find_range(float x)
4    {
5        int result;
6        if (x < 0)
7            result = NEG;
8        else if (x == 0)
9            result = ZERO;
10       else if (x > 0)
11           result = POS;
12       else
13           result = OTHER;
14       return result;
15   }
```

- Assembly code:

```
1    # range_t find_range(float x)
2    # x in %xmm0
3
4    find_range:
5        vxorps %xmm1, %xmm1, %xmm1        # Set %xmm1 = 0
6        vucomiss %xmm0, %xmm1             # Compare 0:x
7        ja .L5                           # If >, goto neg
8        vucomiss %xmm1, %xmm0            # Compare x:0
9        jp .L8                           # If NaN, goto posornan
10       movl $1, %eax                    # result = ZERO
11       je .L3                           # If =, goto done
12
13   .L8:                                 # posornan:
14       vucomiss .LC0(%rip), %xmm0       # Compare x:0
15       setbe %al                        # Set result = NaN?1:0
16       movzbl %al, %eax                 # Zero-extend
17       addl $2, %eax                    # result += 2 (POS for > 0, OTHER for NaN)
18       ret                              # Return
19
20   .L5:                                 # neg:
21       movl $0, %eax                    # result = NEG
22
23   .L3:                                 # done:
24       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.

```
1    RESULT := UnorderedCompare(DEST[31:0] <> SRC[31:0]) {
2          (* Set EFLAGS *) CASE (RESULT) OF
3          UNORDERED: ZF,PF,CF := 111;
4          GREATER_THAN: ZF,PF,CF := 000;
5          LESS_THAN: ZF,PF,CF := 001;
6          EQUAL: ZF,PF,CF := 100;
7          ESAC;
8          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:

```
1    # range_t find_range(float x)
2    # x in %xmm0
3    # typedef enum {NEG, ZERO, POS, OTHER}
4
5    find_range:
6          vxorps %xmm1, %xmm1, %xmm1          # Set %xmm1 = 0
7          vucomiss %xmm1, %xmm0               # Compare x:0
8          jp .other                          # if unordered
9          ja .pos                            # if x > 0
10         je .zero                           # if x == 0
11         jb .neg                            # if x < 0
12
13   .other:
14         movl $3, %eax                      # eax = OTHER
15         jmp .done
16   .pos:
17         movl $2, %eax                      # eax = POS
18         jmp .done
19   .zero:
20         movl $1, %eax                      # eax = ZERO
21         jmp .done
22   .neg:
23         xorl %eax, %eax                    # eax = NEG
24   .done:                         # done:
25         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
1    #include <stdio.h>
2    typedef enum {NEG, ZERO, POS, OTHER} range_t;
3
4    range_t find_range(float x)
5    {
6          __asm__(
7          "vxorps %xmm1, %xmm1, %xmm1\n\t"
8          "vucomiss %xmm1, %xmm0\n\t"
9          "jp .other\n\t"
10         "ja .pos\n\t"
11         "je .zero\n\t"
12         "jb .neg\n\t"
13         ".other:\n\t"
14         "movl $3, %eax\n\t"
```

```
15          "jmp .done\n\t"
16          ".pos:\n\t"
17          "movl $2, %eax\n\t"
18          "jmp .done\n\t"
19          ".zero:\n\t"
20          "movl $1, %eax\n\t"
21          "jmp .done\n\t"
22          ".neg\n\t"
23          "xorl %eax, %eax\n\t"
24          ".done:\n\t"
25          "rep; ret\n\t"
26          )
27      }
28
```

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

```
1       #include <stdio.h>
2       #include <limits.h>
3       #include <assert.h>
4
5       typedef enum {NEG, ZERO, POS, OTHER} range_t;
6
7       /* Access bit-level representation floating-point number */
8       typedef unsigned float_bits;
9
10      range_t find_range(float x)
11      {
12          __asm__(
13              "vxorps %xmm1, %xmm1, %xmm1\n\t"
14              "vucomiss %xmm1, %xmm0\n\t"
15              "jp .other\n\t"
16              "ja .pos\n\t"
17              "je .zero\n\t"
18              "jb .neg\n\t"
19              ".other:\n\t"
20              "movl $3, %eax\n\t"
21              "jmp .done\n\t"
22              ".pos:\n\t"
23              "movl $2, %eax\n\t"
24              "jmp .done\n\t"
25              ".zero:\n\t"
26              "movl $1, %eax\n\t"
27              "jmp .done\n\t"
28              ".neg:\n\t"
29              "xorl %eax, %eax\n\t"
30              ".done:\n\t"
31              "rep; ret\n\t"
32          );
33      }
34
35
36      float u2f(unsigned x)
37      {
38          unsigned* p_x = &x;
39          return *(float*)p_x;
40      }
41
42      int main()
43      {
44          unsigned u = 0;
45          while (u <= UINT_MAX)
46          {
47              float f = u2f(u);
48              unsigned range;
49
50              if(f < 0) {
51                  range = find_range(f);
52                  assert(NEG == range);
53              }
54              else if(f == 0) {
55                  range = find_range(f);
56                  assert(ZERO == range);
```

```
57                    }
58                    else if(f > 0){
59                            range = find_range(f);
60                            assert(POS == range);
61                    }
62                    else {
63                            range = find_range(f);
64                            assert(OTHER == range);
65                    }
66                    u = u + 0x1000;   // to cut down testing cost
67                    printf("Test passed on 0x%g \n", f);
68            }
69            return 0;
70    }
```

- 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

```
1    # range_t find_range(float x)
2    # x in %xmm0
3    # typedef enum {NEG, ZERO, POS, OTHER} range_t
4
5    find_range:
6            vxorps %xmm1, %xmm1, %xmm1           # Set %xmm1 = 0
7            vucomiss %xmm1, %xmm0                # Compare x:0
8            cmovpq   other, %eax                 # if unordered and neg is local variable
9            cmovaq   pos, %eax                   # if x > 0
10           cmoveq   zero, %eax                  # if x == 0
11           cmovbq   neg, %eax                   # if x < 0
12           rep; ret                             # Return
```

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

```
1    #include <stdio.h>
2    typedef enum {NEG, ZERO, POS, OTHER} range_t;
3
4    range_t find_range(float x)
5    {
6            int result;
7            int other = OTHER;
8            int pos   = POS;
9            int zero  = ZERO;
10           int neg   = NEG;
11           asm(
12           "vxorps   %%xmm1, %%xmm1, %%xmm1\n\t"
13           "vucomiss %%xmm1, %%xmm0\n\t"
14           "cmovpq   %1, %%eax\n\t"
15           "cmovaq   %2, %%eax\n\t"
16           "cmoveq   %3, %%eax\n\t"
17           "cmovbq   %4, %%eax\n\t"
18           "movl     %%eax, %0\n\t"
19           : "=a"(result)
20           : "m"(other), "m"(pos), "m"(zero), "m"(neg)
21           );
22           return result;
```

```
23    }
24
```

```
1    /* memtest_x64.c - An example of using memory locations as values */
2
3    #include <stdio.h>
4
5    int main()
6    {
7            long dividend = 20;
8            long divisor = 5;
9            long result;
10
11           asm("divb %2\n\t"
12                   "movq %%rax, %0"
13                   : "=m"(result)        // m - local variable result
14                   : "a"(dividend), "m"(divisor));  // a - eax, m - local variable divisor
15
16           printf("The result is %ld\n", result);
17           return 0;
18    }
```

Thirdly, let's construct the full code:

```
1    #include <stdio.h>
2    #include <limits.h>
3    #include <assert.h>
4
5    typedef enum {NEG, ZERO, POS, OTHER} range_t;
6
7    /* Access bit-level representation floating-point number */
8    typedef unsigned float_bits;
9
10   range_t find_range(float x)
11   {
12           int result;
13           int other = OTHER;
14           int pos   = POS;
15           int zero  = ZERO;
16           int neg   = NEG;
17           asm(
18           "vxorps    %%xmm1, %%xmm1, %%xmm1\n\t"
19           "vucomiss %%xmm1, %%xmm0\n\t"
20           "cmovpq    %1, %%eax\n\t"
21           "cmovaq    %2, %%eax\n\t"
22           "cmoveq    %3, %%eax\n\t"
23           "cmovbq    %4, %%eax\n\t"
24           "movl      %%eax, %0\n\t"
25           : "=a"(result)
26           : "m"(other), "m"(pos), "m"(zero), "m"(neg)
27           );
28           return result;
29   }
30
31
32   float u2f(unsigned x)
33   {
34           unsigned* p_x = &x;
35           return *(float*)p_x;
36   }
37
38   int main()
39   {
40           unsigned u = 0;
41           while (u <= UINT_MAX)
42           {
43                   float f = u2f(u);
44                   unsigned range;
45
46                   if(f < 0) {
47                           range = find_range(f);
48                           assert(NEG == range);
```

```
49                         }
50                         else if(f == 0) {
51                                 range = find_range(f);
52                                 assert(ZERO == range);
53                         }
54                         else if(f > 0){
55                                 range = find_range(f);
56                                 assert(POS == range);
57                         }
58                         else {
59                                 range = find_range(f);
60                                 assert(OTHER == range);
61                         }
62                         u = u + 0x1000;   // to cut down testing cost
63                         printf("Test passed on 0x%g \n", f);
64                 }
65         return 0;
66     }
```

- **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:

```
1     #include <complex.h>
2
3     double c_imag(double complex x) {
4             return cimag(x);
5     }
6
7     double c_real(double complex x) {
8             return creal(x);
9     }
10
11    double complex c_sub(double complex x, double complex y) {
12            return x - y;
13    }
```

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

```
1     # double c_imag(double complex x)
2     c_imag:
3             movapd %xmm1, %xmm0
4             ret
5
6     # double c_real(double complex x)
7     c_real:
8             rep; ret
9
10    # double complex c_sub(double complex x, double complex y)
11    c_sub:
12            subsd %xmm2, %xmm0
13            subsd %xmm3, %xmm1
14            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)