# Lab 11

X86-64

Questions from the Book

### 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.

## 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:

```
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.

**3.65** ◆

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

```
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 $M$?

**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`.

## 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`:

```
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;
}
```

Continued on next page . . . .

3.67 Continued

Gcc generates the following code for these two functions:

```
strB process(strA s)
1   process:
2     movq    %rdi, %rax
3     movq    24(%rsp), %rdx
4     movq    (%rdx), %rdx
5     movq    16(%rsp), %rcx
6     movq    %rcx, (%rdi)
7     movq    8(%rsp), %rcx
8     movq    %rcx, 8(%rdi)
9     movq    %rdx, 16(%rdi)
10    ret
```

```
long eval(long x, long y, long z)
x in %rdi, y in %rsi, z in %rdx
1   eval:
2     subq    $104, %rsp
3     movq    %rdx, 24(%rsp)
4     leaq    24(%rsp), %rax
5     movq    %rdi, (%rsp)
6     movq    %rsi, 8(%rsp)
7     movq    %rax, 16(%rsp)
8     leaq    64(%rsp), %rdi
9     call    process
10    movq    72(%rsp), %rax
11    addq    64(%rsp), %rax
12    addq    80(%rsp), %rax
13    addq    $104, %rsp
14    ret
```

A. We can see on line 2 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?

## 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;

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.)

## EXTRA CREDIT

**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.