시스템프로그램 중간시험 대체과제 (2020학년도 1학기)

학과		학번		학년		이름		

(1) Suppose we are given the task of generating code to multiply integer variable x by various different constant factors K. To be efficient, we want to use only the operations +, -, <<. For the following values of K, write C expressions to perform the multiplication using at most three operations per expression.
① K = 17
② K = -7
③ K = 60
④ K = -112

(2) Write C expressions th generate the following bit patterns, where a^k represents k repretitions of symbol a. Assume a w-bit data type. Your code may contain references to parameters j and k, representing the values of j and k, but not a parameter representing w.
① $1^{w-k}0^k$
② $0^{w-k-j}1^k0^j$

(3) Intel-compatible processors also support an "extended-precision" floating-point format with an 80-bit word, divided into a sign bit, k=15 exponent bits, a single integer bit, and n=63 fraction bits. The integer bit is an explicit copy of the implied bit in the IEEE floating-point representation. That is, it equals 1 for normalized values and 0 for denormalized values. Fill in the following table giving the approximate values of some "interesting" numbers in this format:

Description	Extended precision	
	Value	Decimal
Smallest positive denormalized		
Smallest positive normalized		
Largest normalized		

(4) You have been assigned the task of writing a C function to compute a floating-point representation of 2^x. You decide that the best way to do this is to directly construct the IEEE single-precision representation of the result. When x is too small, your routine will return 0.0. When x is too large, it will return +∞. Fill in the blank positions of the code that follows to compute the correct result. Assume the function u2f returns a floating-point value having an identical bit representation as its unsigned argument.

```
float fpwr2(int x)
{
   /* result exponent and fraction */
   unsigned exp, frac;
   unsigned u;

   if (x < __________) {
      /* too small, return 0.0 */
      exp = __________;
      frac = __________;
   } else if (x < __________) {
      /* denormalized result */
      exp = __________;
      frac = __________;
   } else if (x < __________) {
      /* normalized result */
      exp = __________;
      frac = __________;
   } else {
      /* too big, return +∞ */
      exp = __________;
      frac = __________;
   }

   /* pack exp and frac into 32-bits */
   u = exp << 23 | frac;
   /* return as float */
   return u2f(u);
}
```

(5) 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.

(6) 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 multi-precision 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 $x = 2^{64} \cdot x_h + x_l$ and $y=2^{64} \cdot y_h + y_l$, where x_h, x_l, y_h, and y_l are 64-bit values. Similarly, the 128-bit product can be written as $p = 2^{64} \cdot p_h + p_l$, 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.

(7) 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.

① Which registers hold program values **x**, **n**, **result**, and **mask**?
② What are the initial values of **result** and **mask**?
③ What is the test condition for **mask**?
④ How does **mask** get updated?
⑤ How does **result** get updated?
⑥ Fill in all the missing parts of the C code.

(8) 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.

```
/* 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.

(9) Consider the following source code, shown on the left side of the table, where **R**, **S**, and **T** are constants declared with **#define**. In compiling this program, **gcc** generates the following assembly code on the right side. Answer the following questions.

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

long code7(long i, long j, long k, long *d)
{
   *d = A[i][j][k];
   return sizeof(A);
}
```

```
code7:
   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
```

① Give a formula for the location of array element A[i][j][k], in terms of i, j, k, T, and S.
② Use your reverse engineering skills to determine the values of R, S, and T based on the given assembly code.

(10) In the following code, A and B are constants defined with #define:

```
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?

☆ 수고했습니다~.