I. [Total 20pts, 1pts per each] True or False Questions

1. It is not necessary to have the same number of bits when adding or subtracting signed binary numbers in the 2's-complement system. (T/F)
2. The range of negative numbers when using an eight-bit two's complement system is -1 to -128. (T/F)
3. Floating-point addition is associative but not commutative. (T/F)
4. The single-precision floating point representation of zero contains zeros in the mantissa and ones in the exponent. (T/F)
5. The main reason to use the round-to-even method is to avoid systematic bias when calculating with rounded numbers. (T/F)
6. Floating point rounding may create overflow and the postnormalization step resolves it by shifting right once and incrementing the exponent. (T/F)
7. Machine language is independent of the type of microprocessor in a computer system. (T/F)
8. In x64 all arguments to functions are passed via the stack. (T/F)
9. x86 instructions for integer arithmetic operations do not distinguish signed and unsigned numbers. (T/F)
10. %rip register always holds the address of current instruction that the processor is running. (T/F)
11. While the condition code registers can be set implicitly by arithmetic operations, they can also be set explicitly by writing a 1-bit value (i.e., either '1' or '0') on the condition code registers. (T/F)
12. Conditional move instructions eliminate the necessity of control transfer (i.e., jump) by performing computations on both branches and picking up one afterward depending on the condition. (T/F)
13. When converting "for" loops to do-while loops, the initial condition test can be always optimized away since the initial condition is known at compile time. (T/F)
14. At compile time, switch statements are translated into a series of if-else statements to optimize for performance. (T/F)
15. Caller stores the current address of the call instruction on stack before it jumps, since when the callee's computation is done, the control needs to come back to where the caller called. (T/F)
16. Calling conventions are specific to certain micro-architectures of processors. (T/F)
17. Recursions do not require special treats since they are handled by normal calling conventions. (T/F)
18. All elements in an array are allocated in a contiguous region of memory. (T/F)
19. In x86-64 machine, the initial address of structure and its length must be multiple of 8, which is the word size. (T/F)
20. Access to any field in a union references the same memory address. (T/F)

II. [Total 8pts, 2 pts per line] Bit Manipulation

Fill the empty lines with the appropriate lines of code.

```
/* howManyBits - return the minimum number of bits required to represent x in
 *             two's complement
 *  Examples: howManyBits(12) = 5
 *            howManyBits(298) = 10
 *            howManyBits(-5) = 4
 *            howManyBits(0)  = 1
 *            howManyBits(-1) = 1
 *            howManyBits(0x80000000) = 32
 *  Legal ops: ! ~ & ^ | + << >>
 *  Max ops: 90
 *  Rating: 4
 */
int howManyBits(int x) {
    int sign, pos, bias;
    sign = x>>31;
    /* if negative, don't negate, just flip bits */
    x = (sign & (~x)) | (~sign & x);
    /* bias=1 if x==0 */
    bias = !(x^0);
    pos = _________________
    pos |=_________________
    pos |=_________________
    pos |=_________________
    pos |=_ _______________
    return (pos + 2 + (~bias + 1));
}
```

III. [Total 9pts, 3pts per line] Floating Point Operation

Fill the empty lines with the appropriate lines of code.

```
/*
 * float_half - Return bit-level equivalent of expression 0.5*f for
 *   floating point argument f.
 *   Both the argument and result are passed as unsigned int's, but
 *   they are to be interpreted as the bit-level representation of
 *   single-precision floating point values.
 *   When argument is NaN, return argument
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 30
 *   Rating: 4
 */
unsigned float_half(unsigned uf) {
  unsigned sign = uf>>31;
  unsigned exp = uf>>23 & 0xFF;
  unsigned frac = uf & 0x7FFFFF;
  /* Only roundup case will be when rounding to even */
  unsigned roundup = ____________________;
  if (exp == 0) {
    /* Denormalized.  Must halve fraction */
    frac = ____________________;
  } else if (exp < 0xFF) {
    /* Normalized.  Decrease exponent */
    exp--;
    if (exp == 0) {
      /* Denormalize, adding back leading one */
      frac = ____________________;
    }
  }
  /* NaN Infinity do not require any changes */
  return (sign << 31) | (exp << 23) | frac;
}
```

(frac & 0x3) == 0x3

frac>>1 + roundup

((uf & 0x7FFFFFFF) >> 1 + roundup) & 0x7FFFFF

IV. [Total 8pts, 1pts per each row] 8-bit Floating Point

Suppose there is an 8-bit floating point representation, which constitutes 1 sign bit, 3 exponent bits, and 4 mantissa bits. This representation has the same general form as IEEE format. Fill the 16 cells in the following table.

ID	Description	Binary Encoding	Value
1	Zero		
2	Smallest positive (nonzero)		
3	Largest denormalized		
4	Smallest positive normalized		
5	Largest finite number		
6	NaN		
7	Infinity		
8	One		

V. [Total 10pts, 1pts per each] Switch Statement

Consider the following x86-64 assembly code for a procedure duck():

```
duck:
   cmpq        $7, %rdi
   ja          .L2
   jmp         *.L4(,%rdi,8)
   .section    .rodata
   .align 8
.L4:
   .quad       .L3
   .quad       .L2
   .quad       .L5
   .quad       .L2
   .quad .L6
   .quad       .L7
   .quad       .L2
   .quad       .L5
   .text
.L7:
   xorq        $15, %rsi
   movq        %rsi, %rdx
.L3:
   leaq        112(%rdx), %rdi
   jmp         .L6
.L5:
   leaq        (%rdx,%rsi), %rdi
   salq        $2, %rdi
   jmp         .L6
.L2:
   movq        %rsi, %rdi
.L6:
   movq        %rdi, (%rcx)
   ret
```

Based on the assembly code, fill in the blanks below in its corresponding C source code. (Note: you may only use symbolic variables a, b, and result, from the source code in your expressions below — do not use register names.)

```
void duck(long a, long b, long c, long *dest)
{
   long val;
   switch(a)
   {
         case ____:
               c = ____;
               /* Fall through */
         case ____:
               val = ____;
               break;
         case ____:
         case ____:
```

```
            val = ____;
            break;
      case ____:
            val = ____;
            break;

      default:
            val = ____;
   }
   *dest = val;
}
```

VI. [Total 8pts, 2 pts per each] Calling Conventions

Consider the following C code and corresponding assembly. Fill in the missing instructions (one instruction per a blank line).

```
int global;
int bear(int i, int j, int k)
{
   for( ; i < j; i++)
   {
         global += k*i;
   }
   return global;
}
```

```
bear:
   movq   global, %rax
   _____________________
   _____________________
   movq   %rdi, %rcx
   imulq  %rdx, %rcx
.L5
   incq   %rdi
   addq   %rcx, %rax
   addq   %rdx, %rcx
   _____________________
   _____________________
   movq   %rax, global
.L7
   retq
```

VII. [10 pts] Solve the following problems with the given C code and its assembly code.

C Code

```
void phase(char *input)
{
      int i;
   int numbers[6];
   read_six_numbers(input, numbers);
   if ( numbers[0] <= 13 )  ---------- ( 1 )
      explode_bomb();
   for(i = 1; i < 6; i++) {
      if (numbers[i-1] != numbers[i] + i*2)
         explode_bomb();
      }
      }
}
```

Assembly Code

```
0x0000000000400f69 <+0>:    push   %rbp
0x0000000000400f6a <+1>:    push   %rbx
0x0000000000400f6b <+2>:    sub    $0x28,%rsp
0x0000000000400f6f <+6>:    mov    %rsp,%rsi
0x0000000000400f72 <+9>:    callq  0x401715 <read_six_numbers>
0x0000000000400f77 <+14>:   cmpl   $0xd,(%rsp)
0x0000000000400f7b <+18>:   jg     0x400f82 <phase+25>
0x0000000000400f7d <+20>:   callq  0x4016df <explode_bomb>
0x0000000000400f82 <+25>:   mov    %rsp,%rbp
0x0000000000400f85 <+28>:   mov    $0x2,%ebx
0x0000000000400f8a <+33>:   mov    %ebx,%eax          ---------- ( 2 )   "mov" was also accepted
0x0000000000400f8c <+35>:   add    0x4(%rbp),%eax     ---------- ( 3 )   "add"
0x0000000000400f8f <+38>:   cmp    %eax,0x0(%rbp)
0x0000000000400f92 <+41>:   je     0x400f99 <phase+48>
0x0000000000400f94 <+43>:   callq  0x4016df <explode_bomb>
0x0000000000400f99 <+48>:   add    $0x4,%rbp
0x0000000000400f9d <+52>:   add    $0x2, %rbx         ---------- ( 4 )   "%rbx" was also added
0x0000000000400fa0 <+55>:   cmp    $0xc,%ebx
0x0000000000400fa3 <+58>:   jne    0x400f8a <phase+33>
0x0000000000400fa5 <+60>:   add    $0x28,%rsp
0x0000000000400fa9 <+64>:   pop    %rbx
0x0000000000400faa <+65>:   pop    %rbp
0x0000000000400fab <+66>:   retq
```

1. [4 pts] Fill the C statement in the blank **(1)**, based on the corresponding assembly codes.
2. [2 pts per each] Fill the assembly code in the blanks **(2,3,4)**, based on the corresponding c statements.

VIII. [10 pts] Solve the following problems with the given C code and its assembly code.

C Code

```
int func4(int n, int val)
{
   if (n == 0) {
      return 1;
   }
   if (n == 1) {
      return val;
   }
   return func4(___(1)____) + func4(___(2)___);
}

void phase_4(char *input) {
   int user_val, user_sum, result, target_val, numScanned;

   numScanned = sscanf(input, "%d %d", &user_val, &user_sum);
   if ((numScanned != 2) || user_val < 1 || user_val > 10) {
   explode_bomb();
   }

   target_val = 15;
   result = func4(user_val, target_val);

   if (result != user_sum) {
      explode_bomb();
   }
}
```

n-1, val n-2, val

Assembly Code

```
(gdb) disas func4
Dump of assembler code for function func4:
  0x000000000040105b <+0>:    mov    $0x1,%eax
  0x0000000000401060 <+5>:    test   %edi,%edi
  0x0000000000401062 <+7>:    je     0x40108f <func4+52>
  0x0000000000401064 <+9>:    mov    %esi,%eax
  0x0000000000401066 <+11>:   cmp    $0x1,%edi
  0x0000000000401069 <+14>:   je     0x40108f <func4+52>
  0x000000000040106b <+16>:   push   %r12
  0x000000000040106d <+18>:   push   %rbp
  0x000000000040106e <+19>:   _____(3)_____
  0x000000000040106f <+20>:   mov    %esi,%ebp
  0x0000000000401071 <+22>:   mov    %edi,%ebx
  0x0000000000401073 <+24>:   lea    -0x1(%rdi),%edi
  0x0000000000401076 <+27>:   callq  0x40105b <func4>
  0x000000000040107b <+32>:   mov    %eax,%r12d
  0x000000000040107e <+35>:   lea    -0x2(%rbx),%edi
```

push %rbx

```
   0x0000000000401081 <+38>:    mov    %ebp,%esi
   0x0000000000401083 <+40>:    callq  0x40105b <func4>
   0x0000000000401088 <+45>:    add    %r12d,%eax
   0x000000000040108b <+48>:    pop    %rbx
   0x000000000040108c <+49>:    pop    %rbp
   0x000000000040108d <+50>:    pop    %r12
   0x000000000040108f <+52>:    repz retq
End of assembler dump.
(gdb) disas phase_4
Dump of assembler code for function phase_4:
   0x0000000000401091 <+0>:     sub    $0x18,%rsp
   0x0000000000401095 <+4>:     lea    0xc(%rsp),%rcx
   0x000000000040109a <+9>:     lea    0x8(%rsp),%rdx
   0x000000000040109f <+14>:    mov    $0x40290d,%esi
   0x00000000004010a4 <+19>:    mov    $0x0,%eax
   0x00000000004010a9 <+24>:    callq  0x400c70 <__isoc99_sscanf@plt>
   0x00000000004010ae <+29>:    cmp    $0x2,%eax
   0x00000000004010b1 <+32>:    jne    0x4010bf <phase_4+46>
   0x00000000004010b3 <+34>:    mov    0x8(%rsp),%eax
   0x00000000004010b7 <+38>:    sub    $0x1,%eax
   0x00000000004010ba <+41>:    cmp    $0x9,%eax
   0x00000000004010bd <+44>:    jbe    0x4010c4 <phase_4+51>
   0x00000000004010bf <+46>:    callq  0x401634 <explode_bomb>
   0x00000000004010c4 <+51>:    mov    $0xf,%esi
   0x00000000004010c9 <+56>:    mov    0x8(%rsp),%edi
   0x00000000004010cd <+60>:    callq  0x40105b <func4>
   0x00000000004010d2 <+65>:    cmp    0xc(%rsp),%eax
   0x00000000004010d6 <+69>:    je     0x4010dd <phase_4+76>
   0x00000000004010d8 <+71>:    callq  0x401634 <explode_bomb>
   0x00000000004010dd <+76>:    add    $0x18,%rsp
   0x00000000004010e1 <+80>:    retq
End of assembler dump.
```

1. [4 pts] Fill the C statement in the blanks **(1, 2)** based on the corresponding assembly codes.

2. [2 pts per each] Fill the assembly code in the blanks **(3,4,5)**, based on the corresponding c statements.

IX. [8pts] The following two C code fragments show two different ways of supporting 2D data structures.

A. [4pts] Complete the following assembly codes for the two access functions. Discuss the pros and cons of the two approaches.

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};

int get_pgh_digit
  (int index, int dig)
{
  return pgh[index][dig];
}
```

Nested Array

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

#define UCOUNT 3

int *univ[UCOUNT] = {mit, cmu, ucb};

int get_univ_digit (size_t index,
size_t digit){

  return univ[index][digit];

}
```

```
1) leaq (%rdi,%rdi,4), %rax
addl  %rax, %rsi
movl  2) pgh(,%rsi,4), %eax
```

```
salq    $2, %rsi
addq    3) univ(,%rdi,8), %rsi
4) movl (%rsi), %eax
```

%rax = 5*index
%rsi = 5*index+dig

B. [4pts] Complete the following assembly codes

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```

```
.L11:                          # loop:
  movslq  16(%rdi), %rax       #   i = M[r+16]
  5)_____________________
  6)_____________________
  testq   %rdi, %rdi           #   test r
  jne     .L11                 #   if !=0 goto loop
```

X. [9pts] Answer the questions related to the following C and assembly codes.

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

```
void call_echo()
{
          echo();
}
void echo()
{
        char buf[8];
        gets(buf);
        puts(buf);
}
```

```
00000000004006cf <echo>:
  4006cf:     48 83 ec 18               sub    $0x18,%rsp
  4006d3:     48 89 e7                  mov    %rsp,%rdi
  4006d6:     e8 a5 ff ff ff            callq  400680 <gets>
  4006db:     48 89 e7                  mov    %rsp,%rdi
  4006de:     e8 3d fe ff ff            callq  400520 <puts@plt>
  4006e3:     48 83 c4 18               add    $0x18,%rsp
  4006e7:     c3                        retq
00000000004006e8 <call_echo>:
  4006e8:     48 83 ec 08               sub    $0x8,%rsp
  4006ec:     b8 00 00 00 00            mov    $0x0,%eax
  4006f1:     e8 d9 ff ff ff            callq  4006cf <echo>
  4006f6:     48 83 c4 08               add    $0x8,%rsp
  4006fa:     c3                        retq
```

A. [5pts] Find an input for “gets” to change the return address set by “call <echo>” instruction in the stack to 0x0 (8B).

B. [4pts] 1) Fill the following assembly which includes the Stack Canary support. 2) Explain how the Canary code can call __stack_chk_fail@plt if the attacker attempts to insert and execute the malicious code from Question B.

```
40072f: sub    $0x18,%rsp
400733: mov    %fs:0x28,%rax
40073c: mov    %rax,0x8(%rsp)
400741: xor    %eax,%eax
400743: mov    %rsp,%rdi
400746: callq  4006e0 <gets>
40074b: mov    %rsp,%rdi
40074e: callq  400570 <puts@plt>
400753: 1) _____________________
400758: 2) _____________________
400761: je     400768 <echo+0x39>
400763: callq  400580 <__stack_chk_fail@plt>
400768: add    $0x18,%rsp
40076c: retq
```