1. **True/False.** Circle the appropriate choice (there are no trick questions).

   (a) **T** In C, a pointer variable is used for storing an address.

   (b) **F** In x86 assembly, the eax register can be used to hold integers but not addresses.

   (c) **T** In C, if \(x\) is an integer variable, the expression “\(x \ll 3\)” computes \(x \times 8\) but does not change the value of \(x\).

   (d) **F** If, in C, an array is declared by “\(\text{int } a[10];\)”, then accessing \(a[11]\) during execution will generate an error message.

   (e) **T** A callee-saved register should be saved before it is written to in a procedure.

   (f) **T** A mask is a value used to modify or examine individual bits of a byte or word.

   (g) **T** The bitwise **xor** operation (\(^\) in C) can be used to flip the bits of a word.

   (h) **F** The bitwise **and** operation (\(\&\) in C) can be used to flip the bits of a word.

   (i) **F** Compiling C into x86 assembly comprises translating each C statement into a single assembly instruction.

   (j) **T** In x86 assembly, a label corresponds to an address.

2. **Answer this question on this sheet**

   Consider the following x86 code fragment for computing the sum of an array of 10 32-bit integers, pointed to by ecx.

   ```
   #Intel Syntax               #AT&T Syntax
   mov eax,0
   mov edx,0
   TOP:
   cmp edx,10
   jl OUT
   add eax,[ecx+edx]
   inc edx
   jmp TOP
   OUT:
   ```

   (a) There are two bugs in the code. What are they?

   The conditional jump should be a “jge” instruction and each array element should be accessed as “\([eax+edx*4]\)” or “\([%ecx,%edx,4]\)”.

   (b) Put a comment following each “#” in the above code to describe what that instruction does (or is supposed to do, in the case of a bug).

   **See the code above.**
3. **Put your answer in the blue book.**

Consider the following C procedure, `foo`.

```c
int foo(int x)
{
    int i;
    int y = 0;
    for(i=0;i<32;i++) {
        y = y + (x & 0x1);
        x = x >> 1;
    }
    return y;
}
```

(a) What does `foo` do? That is, what is the relationship between the value that it returns and the value passed into it?

**It counts the number of bits in a word that contain a 1.**

(b) Write an x86 assembly procedure that computes the same thing that the above C procedure does. Be sure that your assembly procedure could be called from a C procedure.

**Intel Syntax:**

```
.intel_syntax
.text
.globl _foo
_foo:
push ebp
mov ebp,esp
push ebx          #save ebx
mov eax,0         #eax contains y
mov ecx,0         #ecx contains i
mov edx,[ebp+8]   #edx contains x

TOP:
cmp   ecx,32
jge   OUT

mov   ebx,edx       #mov x into ebx
and   ebx,1
add   eax,ebx
shr   edx          #shift x
inc   ecx
jmp   TOP

OUT:
pop   ebx           #restore ebx
pop    ebp
ret
```
AT&T Syntax:

```assembly
.text
.globl _foo
_foo:
push %ebp
mov %esp,%ebp
push %ebx #save ebx
mov $0,%eax #eax contains y
mov $0,%ecx #ecx contains i
mov 8(%ebp),%edx #edx contains x

TOP:
cmp $32,%ecx
jge OUT
mov %edx,%ebx #mov x into ebx
and $1,%ebx
add %ebx,%eax
shrl %edx #shift x
inc %ecx
jmp TOP

OUT:
pop %ebx #restore ebx
pop %ebp
ret
```


(a) Define a C struct type `CELL` that contains the following fields: an integer `x`, a string `y`, and a `next` field that points to another structure of type `CELL`.

Answer:
```c
typedef struct cell {
    int x;
    char *y;
    struct cell *next;
} CELL;
```

(b) Write in C a procedure corresponding to the declaration

```c
int list_length(CELL *head);
```

that returns the length of a linked list whose first element is pointed to by `head`.

Answer:
```c
int list_length(CELL *head)
{
    int count = 0;
    CELL *p = head;
```
while (p != NULL) {
    count++;
    p = p->next;
}
}

5. **Put your answer in the blue book.**

A simple C procedure for computing the integer square root of a number (i.e. the greatest integer less than or equal to the square root of the number) is as follows:

```c
int square_rt(int x, int low, int high)
{
    if (low >= high - 1)
        return low;
    int mid = (low + high) >> 1;
    if ((mid * mid) > x)
        return square_rt(x,low,mid);
    else
        return square_rt(x,mid,high);
}
```

where the initial call to `square_rt` would be `square_rt(n,1,n)`. Translate `square_rt` into x86 assembly, so that it could be called from C.

**Intel Syntax:**

```assembly
.globl _square_rt
_square_rt:
push ebp
mov ebp,esp
push ebx #using ebx, so save it

mov eax,[ebp+12] #eax gets low
mov ecx,[ebp+16] #ecx gets high
sub ecx,1 #now ecx has high-1
cmp eax,ecx # if low >= high-1
jge DONE # eax already contains low, just return

mov ecx,[ebp+16] #reload ecx with high
mov edx,eax #edx gets mid, first get low
add edx,ecx #add high
shr edx #shift right by one

mov ebx,edx #ebx will hold mid*mid
imul ebx,ebx

```

```assembly
    cmp ebx,[ebp+8] #if mid*mid > x
    jg GREATER # jump to GREATER
```

```assembly
DONE:
```
```
push ecx #push high
push edx #push mid
push DWORD PTR [ebp+8] #push x
call _square_rt
add esp,12 #adjust stack
jmp DONE #we're done, result is already in eax

GREATER:

push edx #push mid
push eax #push low
push DWORD PTR [ebp+8] #push x
call _square_rt
add esp,12 #adjust stack
#we're done, result is already in eax

DONE:

pop ebx #restore ebx
pop ebp
ret

AT&T syntax:

.globl _square_rt
_square_rt:
push %ebp
mov %esp,%ebp
push %ebx #using ebx, so save it

mov 12(%ebp),%eax #eax gets low
mov 16(%ebp),%ecx #ecx gets high
sub $1,%ecx #now ecx has high-1
cmp %ecx,%eax #if low >= high
jge DONE # eax already contains low, just return

mov 16(%ebp),%ecx #reload ecx with high
mov %eax,%edx #edx gets mid, first get low
add %ecx,%edx #add high
shr %edx #shift right by one

mov %edx,%ebx #ebx will hold mid*mid
imul %ebx,%ebx
cmp 8(%ebp),%ebx #if mid*mid > x
jg GREATER # jump to GREATER

push %ecx #push high
push %edx  #push mid
pushl 8(%ebp)  #push x
call _square_rt
add $12,%esp  #adjust stack
jmp DONE  #were done, result is already in %eax

GREATER:

#calling square_rt(x, low, mid)
push %edx  #push mid
push %eax  #push low
pushl 8(%ebp)  #push x
call _square_rt
add $12,%esp  #adjust stack
jmp DONE  #result is already in %eax

DONE:

pop %ebx  #restore ebx
pop %ebp
ret