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

(a) **F** The x86 registers (eax, ebx, etc.) are part of main memory (RAM).
(b) **T** In C, a formal parameter that is an array of strings could have the type `char **`.
(c) **T** In x86 assembly, `call f` pushes the return address on the stack and jumps to the code at the label f.
(d) **F** In x86 assembly, global variables are allocated on the stack and local variables reside in the data segment.
(e) **T** CPU and main memory are connected by a bus.
(f) **T** A procedure is free to overwrite a caller-saved register without saving it first.
(g) **T** In C, the expression `p->x` is equivalent to `(*p).x`.
(h) **F** In x86 assembly using the calling convention discussed in class, the first parameter to a procedure can be found at the address computed by adding the value of the ebp register and 12 (assuming a 32-bit machine)
(i) **F** In x86 assembly, if f is a recursive procedure that takes two parameters, then when it calls itself, it doesn’t need to push any more parameters on the stack.
(j) **T** In x86 assembly, `pop eax` (in AT&T syntax) is equivalent to `mov %eax,-4(%esp)` followed by `add %esp,4` (in AT&T syntax, `mov (%esp),%eax` followed by `add $4,%esp`).

2. The assembly code that I wrote corresponding to the C procedure

```c
void foo(int x, int y) { int z = x + y; z = z - 7; return z; }
```

is

```assembly
#Intel Syntax               #AT&T Syntax
_foo:                      _foo:
push ebp                    push %ebp
mov ebp,esp                 mov %esp,%ebp     #DRAW THIS
sub esp,4                    sub $4,%esp
mov eax,[ebp+8]             mov 8(%ebp),%eax
add eax,[ebp+12]           add 12(%ebp),%eax
mov [ebp-4],eax            mov %eax,-4(%ebp) #DRAW THIS
sub DWORD PTR [ebp-4],7    subl $7,-4(%ebp)
mov eax,[ebp-4]            mov -4(%ebp),%eax
pop ebp                     pop %ebp         #DRAW THIS
ret                         ret
```

(a) Assume `foo(13,14)` is called from `main()`. Draw the state of the stack, showing the values stored in the stack frame for `foo` and where ESP and EBP point, after the execution of each instruction that has the comment “DRAW THIS”. Since there are three such instructions, you should draw the stack three times.
(b) My program crashes during the execution of foo. Why is that? How should I fix it?

Answer: I forgot to add 4 to esp to remove the local variable z from the stack before executing pop ebp and ret. Thus, esp does not actually point to the return address when the ret instruction is executed, causing the processor to try to jump to an illegal address.

3. In C, suppose you have a binary tree whose node type is defined by the following:

```c
typedef struct node {
    int value;
    struct node *left;
    struct node *right;
} NODE;
```

(a) Below is my version of the procedure that creates a new node and returns a pointer to it, so that the new node can be inserted into the binary tree. This code compiles without errors, but doesn’t execute correctly.

```c
NODE *new_node(int val) {
    NODE n;
    n.left = NULL;
    n.right = NULL;
    n.value = val;
    return &n;
}
```

Explain what I am doing wrong.

Answer: Because n is declared to be a local variable in new_node, n will be allocated within the stack frame for new_node and will be removed from the stack when new_node returns (more precisely, it will still be in memory, but will be subsequently overwritten when other data is pushed onto the stack). new_node should return a pointer to a NODE that is permanent (i.e. until the program ends or is explicitly deleted).

(b) Write the correct code for new_node.

Answer:

```c
NODE *new_node(int val) {
    NODE *n = (NODE *) malloc(sizeof(NODE));
```
n->left = NULL;
n->right = NULL;
n->value = val;
return n;
}

(c) Write a C procedure
void insert_left(NODE *p);

that inserts the node pointed to by p as the leftmost leaf of the tree. Assume that the
global variable root points to the root of the tree (but don’t assume that root is not NULL).
Answer:

void insert_left(NODE *p)
{
    if (root == NULL)
        root = p;
    else {
        NODE *n = root;
        while(n->left != NULL)
            n = n->left;
        n->left = p;
    }
}

4. Write a C procedure with the signature

int intlog(int x);

that computes \( \lfloor \log x \rfloor \) without using * (multiply) or / (divide). That is, it returns the largest
integer no greater than \( \log x \) (i.e. the log, base 2, of \( x \)). For example, \( \text{intlog}(56) \) returns 5,
since \( 5 \leq \log 56 < 6 \). You can assume that \( x > 0 \).
Answer:

//Here, log x is computed by counting the number of times that x can be
//divided by 2 (i.e. shifted right by 1) to arrive at the number 1. This
//code will only work for positive values of x.

int intlog(int x)
{
    int count=0;
    while (x != 1) {
        x = x >> 1;
        count++;
    }
    return count;
}
5. Write, in x86 assembly, a procedure which would have the C signature

    int max(int a[], int size);

and returns the value of the largest element of the array a, whose size is the second parameter to max.

**Answer:** The assembly below corresponds to the following C code:

```c
int max(int a[], int size)
{
    int i;
    int the_max = a[0];
    for(i=1;i<size;i++)
        if (a[i] > the_max)
            the_max = a[i];
    return the_max;
}
```

```assembly
#Intel Syntax
_max:
    push ebp
    mov ebp,esp
    mov ecx,[ebp+8] #ecx holds a
    mov eax,[ecx] #eax holds the_max, initially a[0]
    mov edx,1
LOOP:
    cmp edx,[ebp+12] #compare i to size
    jge DONE
    cmp [ecx+edx*4],eax #compare a[i] to the_max
    jle L
    mov eax,[ecx+edx*4] #otherwise, the_max = a[i]
L:
    inc edx
    jmp LOOP
DONE:
    pop ebp
    ret
#result is already in %eax
```

```assembly
#AT&T Syntax
_max:
    push %ebp
    mov %esp,%ebp
    mov 8(%ebp),%ecx #ecx holds a
    mov (%ecx),%eax #eax holds the_max, initially a[0]
    mov $1,%edx #edx holds i
LOOP:
    cmp 12(%ebp),%edx #compare i to size
    jge DONE #if i >= size, jump out of loop
    cmp %eax,(%ecx,%edx,4) #compare a[i] to the_max
    jle L #if a[i] <= the_max, jump to L
    mov (%ecx,%edx,4),%eax #otherwise, the_max = a[i]
L:
    inc %edx #i++
    jmp edx # go to top of loop
DONE:
    pop %ebp
    ret
```