NAME: _____

Part 1: Numbers, Arrays, and Functions

1. (10 pts) For each of the following sections of code, explain why the specified assertion may fail:

a.
```
char x;
for(x=120; x<256; x++) { assert(x>0); }
```
The x variable is of character type, whose largest positive value is 127. When you try to exceed 127, you overflow x, and it becomes negative (-128).

b.
```
int x=7; int y=9; int z=10*(x/y);
assert(z>0);
```
Since x and y are integers, (x/y) will be integer division, which rounds down to the nearest integer. Since 7/9 = 0.777.., this rounds down to 0, and z will become 0.

c.
```
float frac; float sum=0.0; const float sth=1.0/7.0;
for(frac=sth; frac<1.0; frac+=sth) { sum +=frac*17.6; }
assert(sum==17.6);
```
Because floating point numbers are approximations, frac*17.6 is not exactly 17.6/7, so 7*17.6/7 is not exactly 17.6. Furthermore, if sth>=(1.0/7.0), then the loop will only execute 6 times. Furthermore, sum is really 1/7*17.6 + 2/7*17.6+… + 6/7*17.6 = 52.8

d.
```
int x=foo(); char y=x+7;
assert(y==(x+7));
```
If x>255, then x+7 will get truncated when it is assigned to y. For instance, if x=260, x+7=267=x010B, but when this is assigned to y, it will get truncated to 0x0B=11.

e.
```
unsigned int x=7; int y=foo();
if (y<0 && (7*y)<0) { assert(x*y<0); }
```
Since x is unsigned, the expression "x*y" is treated as unsigned, and unsigned numbers are always >= 0.

2. (10 pts) Given the following C code:

```
union { int wn; char cn[4]; } num;
num.wn=foo();
if (num.cn[3] & 0x80) printf("Sign bit on\n");
```

a. If foo() returns the value -129 (0xFFFF FF7F) on a big-endian machine, will the "Sign bit on" message get printed? No – num.cn[3] & 0x80=0x7F & 0x80=0x00 (2 pts)

b. If foo() returns the value -129 (0xFFFF FF7F) on a little-endian machine, will the "Sign bit on" message get printed? Yes – num.cn[3] & 0x80 = 0xFF & 0x80 = 0x10 (2 pts)

c. Give an example of a number returned by foo() which would print "Sign bit on" on both a big-endian and a little-endian machine. -1=0xFFFF FFFF (3 pts)

d. How would you check for the sign bit on a little-endian machine? (There are lots of correct answers… any correct answer gets full credit.) (num.cn[0] &0x80); (num.wn<0); (3 pts)

Sorry about the extra parens

3. (10 pts) Given the following C code:

```
float dist[4][5]; int r; int c;
void setDist(int r,int c,float v) { (*(dist+r*4+c)=v; }
float getDist(int r,int c) { return *(dist+r*4+c); }
for(r=0;r<4;r++) {
    for(c=0;c<5;c++) { setDist(r,c,r*10.0+c); }
}
```

a. After the code executes, what is the value of getDist(2,2)? 22.0 C(2 pts)

b. After the code executes, what is the value of dist[2][2]? 30.0 (dist[2][2] is at offset 2*5+2=12 from the beginning of dist, which is set when r=2, c=4 and again (finally) when r=3, c=0.) (2 pts)

c. If &dist[0]==0x0801 c000, (assuming sizeof(float)==4) what is the value of dist+2*4+2? 0x0801 c028 (dist+8+2 = dist + 10 (floats) = dist + 40 (bytes) =dist + 0x28) (3 pts)

d. Would the answers to a, b, and c be the same if "float dist[4][5]" were replaced by "float *dist=malloc(20*sizeof(float));"? If not, what answer would be different? Same (3 pts)

4. (20 pts) Put an "X" in front of each true statement below:

___ In the x86 ISA, a single instruction can access up to three different memory locations.

_X_ The difference between a computer and an adding machine is that you can program a computer… you can't program an adding machine.

___ If a computer does not support the x86 architecture, you cannot run UNIX on that computer.

___ As an ISA evolves, it stays downwardly compatible (supports everything the previous version supports) in order to make the resulting ISA simpler than the previous version.

_X_ In the x86 ISA, the name of each integer register implies the length (number of bits) of that register.

_X_ In x86, the condition code registers, CF, ZF, SF, and OF, are set either explicitly by a cmp or test instruction, or implicitly by any instruction which performs an arithmetic operation.

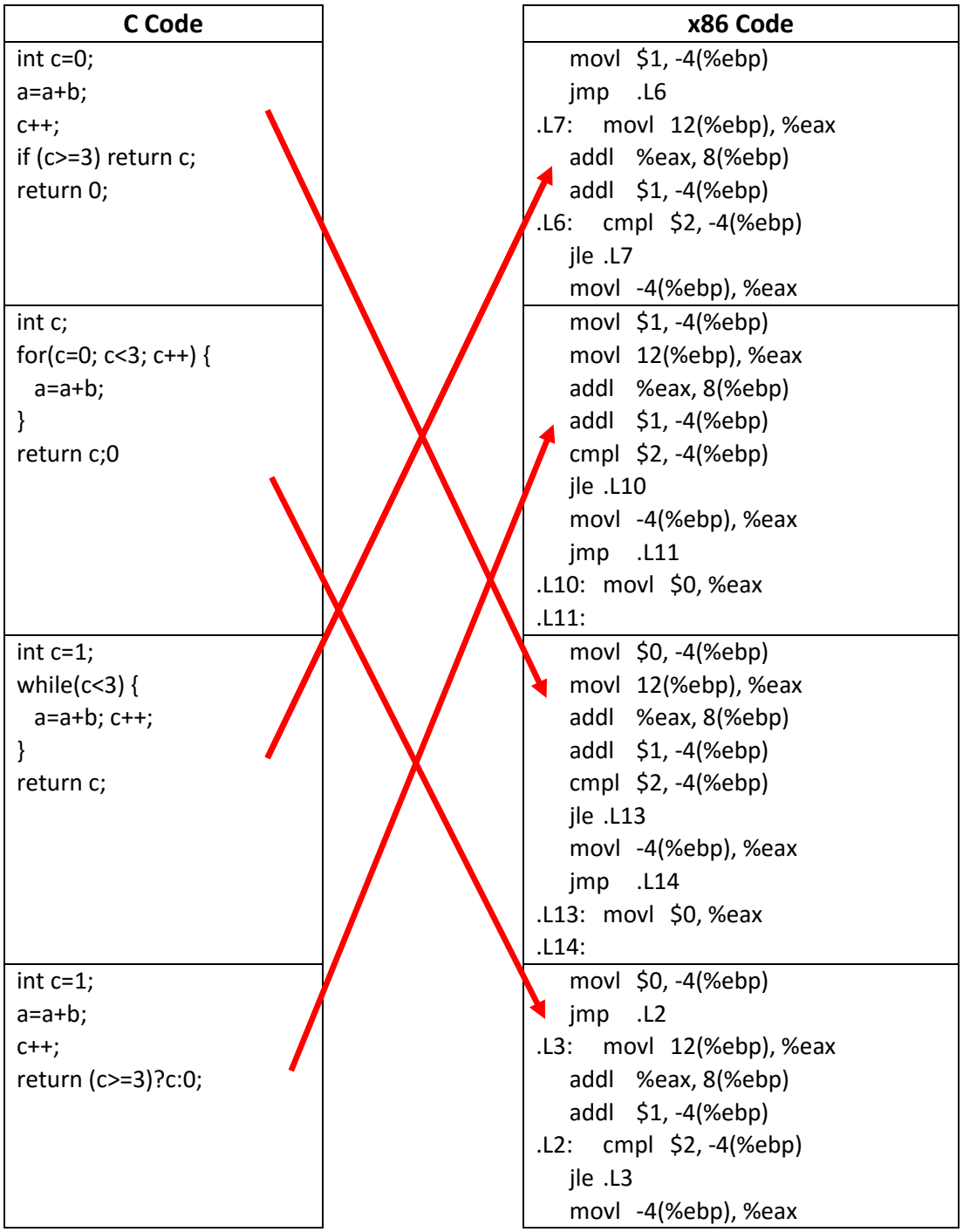
_X_ In x86 IA-32, each function creates its own stack frame below its caller's stack frame on entry, and restores its callers stack frame when it exits.

___ In x86 IA-32, the %eax register always contains the return address when a function exits.

_X_ In x86 IA-32, the responsibility to save and restore register values while a lower level function is called is shared between the caller of a function, and the function being called (the callee).

___ In x86 IA-32, the 4 byte word in the current stack frame with the greatest (highest) address is in the %ebp register, and the 4 byte word in the stack frame with the least (lowest) address is in the %esp register. (The address is in the register, but this implies that the register contains the value… not the address…. but since this was unclear, allowed either X or no X.)

(20 pts) Draw a line between the C code, and the x86 code generated from that C code.



C Code	x86 Code

int c=0;
a=a+b;
c++;
if (c>=3) return c;
return 0;
	movl $1, -4(%ebp)
	jmp .L6
	.L7: movl 12(%ebp), %eax
	addl %eax, 8(%ebp)
	addl $1, -4(%ebp)
	.L6: cmpl $2, -4(%ebp)
	jle .L7
	movl -4(%ebp), %eax

int c;
for(c=0; c<3; c++) {
a=a+b;
}
return c;0
	movl $1, -4(%ebp)
	movl 12(%ebp), %eax
	addl %eax, 8(%ebp)
	addl $1, -4(%ebp)
	cmpl $2, -4(%ebp)
	jle .L10
	movl -4(%ebp), %eax
	jmp .L11
	.L10: movl $0, %eax
	.L11:

int c=1;
while(c<3) {
a=a+b; c++;
}
return c;
	movl $0, -4(%ebp)
	movl 12(%ebp), %eax
	addl %eax, 8(%ebp)
	addl $1, -4(%ebp)
	cmpl $2, -4(%ebp)
	jle .L13
	movl -4(%ebp), %eax
	jmp .L14
	.L13: movl $0, %eax
	.L14:

int c=1;
a=a+b;
c++;
return (c>=3)?c:0;
	movl $0, -4(%ebp)
	jmp .L2
	.L3: movl 12(%ebp), %eax
	addl %eax, 8(%ebp)
	addl $1, -4(%ebp)
	.L2: cmpl $2, -4(%ebp)
	jle .L3
	movl -4(%ebp), %eax

5. (15 pts) Using the following Swap Space Table and Page Table [The LRU column in the page table identifies the head and tail of the LRU pointers, and contains a pointer to the slot of the next recently used. Thus, the most recently used page is in slot 0003, and the least recently used page is in slot 0002.]

Swap Space Table			
Free	PID	Page ID	DiskAddr
0	1111	0801 0	S:01000
0	1111	0801 1	S:02000
0	1111	FFFF C	S:03000
1	2222	0801 0	S:04000
1	2222	FFFF D	S:05000
0	1111	0804 C	S:06000
0	1111	FFFF B	S:07000
1	2222	0801 2	S:08000

Page Table				
PID	Page ID	Slot	Dirty	LRU
1111	0801 0	0001	0	->0006
1111	FFFF C	0002	1	TAIL->NULL
2222	0801 0	0003	0	HEAD->0001
1111	FFFF B	0004	1	->0002
2222	FFFF D	0005	0	->0004
1111	0804 C	0006	1	->0005

If the Memory management unit gets a request to fetch an instruction from memory for PID=1111, at address 0x0801 102C, identify all the actions and table updates the MMU must perform to translate the virtual address 0x0801 102C to a real memory address. You may assume the MMU uses a true least-recently used algorithm.

a. Divide the requested address into Page 0801 1, and offset 02C.
b. Look up PID=1111, Page=0801 1 in the page table… not found.
c. Find least recently used page at slot 0002 in the page table
d. Since slot 0002 dirty bit is on, find unused slot in swap space table @ S:04000
e. Write 4K page in real slot 0002 to disk at S:04000
f. Update LRU “TAIL” pointer to slot 0004 in the page table
g. Update Swap Space Table: Free=0, PID=1111, Page ID=FFFF C at DiskAddr=S:04000
h. Update Swap Space Table: Free=1 at DiskAddr=S:03000
i. Find PID=1111, Page ID=0801 1 in swap space table at DiskAddr S:02000
j. Copy 4K page in Swap at DiskAddr S:02000 to Slot 002 in memory
k. Update page table at Slot 0002: PID=1111 Page ID=0801 1 Dirty=0 LRU: HEAD->0003
l. Return Real Address 0002 02C

Part 4: Reading and Writing C Code

6. (15 pts) Consider the following code used to implement a heap management routine using a free list…

```
struct { void * loc; int len; } freeList[100];
int fUsed=0;

void addFree(void *loc,int len) {
    assert(fUsed<100);
    freeList[fUsed].loc=loc;
    freeList[fUsed].len=len;
    fUsed++;
}
int findFree(int len) {
    int i;
    for (i=0; i<fUsed; i++)
        if (len<=freeList[i].len) return i;
    return -1;
}

addFree(HEAPSTART,HEAPSIZE);
```

```
void * malloc(int request) {
    request=4*((request+7)/4); // Round up + prefix
    int r=findFree(request);
    if (r==-1) return NULL; // Request cannot be satisfied
    void *bptr=freeList[r].loc;
    *(int *)bptr=request;
    freeList[r].loc+=request;
    freeList[r].len-=request;
    return bptr+4;
}
```

Using the conventions above, write the "free" function. The free function takes a void * pointer as an argument, where the pointer is some value returned by the malloc routine. Your free routine may assume that the heap memory is correct… that is, you are given a pointer that was originally the value returned by malloc, and there is a prefix 4 bytes before the pointer which contains the length of the block given by malloc. The free routine should create a new free list entry for the entire block, including the prefix, which contains both the length and location of the newly freed block. The free function does not need to return any value.

```
void free(void *payload) {
    void *bptr=payload-4;
    int len=*(int *)bptr;
    addFree(bptr,len);
}
```

Notes on Problem 3…

int dist[4,5] looks like:

[0][0] +0	[0][1] +1	[0][2] +2	[0][3] +3	[0][4] +4
[1][0] +5	[1][1] +6	[1][2] +7	[1][3] +8	[1][4] +9
[2][0] +10	[2][1] +11	[2][2] +12	[2][3] +13	[2][4] +14
[3][0] +15	[3][1] +16	[3][2] +17	[3][3] +18	[3][4] +19

After setDist…

[0][0] =0	[0][1] =1	[0][2] =2	[0][3] =3	[0][4] =4,10
[1][0] =11	[1][1] =12	[1][2] =13	[1][3] =14,20	[1][4] =21
[2][0] =22	[2][1] =23	[2][2] =24,30	[2][3] =31	[2][4] =32
[3][0] =33	[3][1] =34	[3][2] =??	[3][3] =??	[3][4] =??