

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); }
```

b. 

```
int x=7; int y=9; int z=10*(x/y);
assert(z>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);
```

d. 

```
int x=foo(); char y=x+7;
assert(y==(x+7));
```

e. 

```
unsigned int x=7; int y=foo();
if (y<0 && (7*y)<0) { assert(x*y<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");
```

- If `foo()` returns the value -129 (0xFFFF FF7F) on a big-endian machine, will the "Sign bit on" message get printed?
- If `foo()` returns the value -129 (0xFFFF FF7F) on a little-endian machine, will the "Sign bit on" message get printed?
- 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.
- 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.)

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); }
}
```

- After the code executes, what is the value of `getDist(2,2)`?
- After the code executes, what is the value of `dist[2][2]`?
- If `&dist[0]==0x0801 c000`, (assuming `sizeof(float)==4`) what is the value of `dist+2*4+2`?
- 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?

Part 2: ISA, X86, and Stack Frames

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

5. (20 pts) Draw a line between the C code, and the x86 code generated from that C code.

C Code	x86 Code
<pre>int c=0; a=a+b; c++; if (c&gt;=3) return c; return 0;</pre>	<pre>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</pre>
<pre>int c; for(c=0; c&lt;3; c++) {     a=a+b; } return c;</pre>	<pre>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:</pre>
<pre>int c=1; while(c&lt;3) {     a=a+b; c++; } return c;</pre>	<pre>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:</pre>
<pre>int c=1; a=a+b; c++; return (c&gt;=3)?c:0;</pre>	<pre>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</pre>

Part 3: Linking, Loading, Processes, Virtual Memory, Heap Memory

6. (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.

Part 4: Reading and Writing C Code

7. (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.

Note... we did not study heap memory management in the Spring 2016 semester, so a question about heap memory management will not appear on the final exam. However, I left this question in to show what kinds of questions will be asked, and to give some idea of how long it may take to do the final exam.