8. Reading Assembly Code [10 marks]

a) Add high-level, C-like comments to the following assembly program. Be sure to clearly indicate the location of high-level structure such as *loops*, *if-statements*, *reading* from or *writing* to *variables* or *arrays* etc. Partial marks on this question will be determined by the amount of high-level structure you identify.

```
     ld  $a, r0          #
     ld  (r0), r0        #
     ld  $b, r1          #
     ld  (r1), r1        #
     not r1              #
     inc r1              #
     ld  $0, r2          #
L0:  mov r2, r3          #
     add r1, r3          #
     beq r3, L4          #
     mov r2, r3          #
     inc r3              #
L1:  mov r3, r4          #
     add r1, r4          #
     beq r4, L3          #
     ld (r0, r2, 4), r5  #
     ld (r0, r3, 4), r6  #
     mov r5, r7          #
     not r7              #
     inc r7              #
     add r6, r7          #
     bgt r7, L2          #
     st r6, (r0, r2, 4)  #
     st r5, (r0, r3, 4)  #
L2:  inc r3              #
     br L1               #
L3:  inc r2              #
     br L0               #
L4:  halt
```

b) Give C code that does what this assembly code does. For full credit, your code must be straight-forward C, like you would write if you were writing this from scratch and not just translating it from the assembly code.

c) Give a plain-English description of what this code does. Your description must be a clear, simple, high-level description of what the code achieves, not a step by step description of how it does it.

8 **[15 marks]** **Reading Assembly.** Comment the following assembly code and then translate it into C. Assume that the caller prologue was completed as shown in lecture, and that *register 0* is used to return a value. *Use the back of the preceding page for extra space if you need it.*

```
foo:      deca r5              #
          st r6, (r5)          #
          ld $0, r0            #
          ld $0, r1            #
          ld 4(r5), r2         #
          ld 8(r5), r3         #
          ld 12(r5), r4        #
L0:       mov r3, r6           #
          not r6
          inc r6               #
          add r1, r6           #
          beq r6, L3           #
          ld (r2, r1, 4), r6   #
          not r6
          inc r6               #
          add r4, r6           #
          beq r6, L1           #
          br L2                #
L1:       inc r0               #
L2:       inc r1               #
          br L0                #
L3:       ld (r5), r6          #
          inca r5              #
          j (r6)               #
```

8a Translate into C:

8b Explain what the code does in one sentence.

4 **(8 marks)** **Reading Assembly Code.** Consider the following SM213 assembly procedure defined as `int foo(int n)`. The procedure used the call/return conventions described in class where register r5 is the stack pointer. Register r0 is used to store the return value of the function.

```
foo:    deca r5           #
        deca r5           #
        st   r6, 4(r5)    #
        ld   8(r5), r1    #
        bgt  r1, L1       #
        ld $0, r2         #
        br L3             #
L1:     dec r1            #
        bgt r1, L2        #
        ld $1, r2         #
        br L3             #
L2:     ld 8(r5), r1      #
        dec r1            #
        deca r5           #
        st r1, 0(r5)      #
        gpc $0x6, r6      #
        j foo             #
        inca r5           #
        st r0, 0(r5)      #
        ld 8(r5), r1      #
        dec r1            #
        dec r1            #
        deca r5           #
        st r1, 0(r5)      #
        gpc $0x6, r6      #
        j foo             #
        inca r5           #
        ld (r5), r2       #
        add r0, r2        #
L3:     mov r2 ,r0        #
        ld 4(r5), r6      #
        inca r5           #
        inca r5           #
        j (r6)            #
```

4a Carefully comment every line of code above.

4b Convert the assembly into C.

4c The code implements a simple function. What is it? Give the simplest, plain English description you can.

5 **(4 marks)** **Procedure Calls.** Given these global declarations

```
int x;
int (*proc) (int);
```

Give the SM213 assembly for the code below (just for this single statement, not for the procedure itself). Assume that the return value is in `r0`. Comments are not required.

```
x = proc (1);
```

9 **(10 marks)** **Reading Assembly.** Comment the following assembly code and then translate it into C. *Use the back of the preceding page for extra space if you need it.*

```
foo:    ld   $-12, r0          #
        add  r0, r5            #
        st   r6, 8(r5)         #
        ld   $0, r1            #
        st   r1, 0(r5)         #
        st   r1, 4(r5)         #
        ld   20(r5), r2        #
        not  r2                #
        inc  r2                #
L0:     mov  r2, r3            #
        add  r1, r3            #
        beq  r3, L3            #
        bgt  r3, L3            #
        ld   12(r5), r3        #
        ld   (r3, r1, 4), r3   #
        ld   16(r5), r4        #
        ld   (r4, r1, 4), r4   #
        ld   $-8, r0           #
        add  r0, r5            #
        st   r3, 0(r5)         #
        st   r4, 4(r5)         #
        gpc  $6, r6            #
        j    bar               #
        ld   $8, r3            #
        add  r3, r5            #
        beq  r0, L2            #
        ld   0(r5), r3         #
        inc  r3                #
        st   r3, 0(r5)         #
L2:     inc  r1                #
        br   L0                #
L3:     ld   0(r5), r0         #
        ld   8(r5), r6         #
        ld   $12, r1           #
        add  r1, r5            #
        j    (r6)              #
```

Translate into C:

11 **(4 marks)** **Loops**. Consider the consequences of eliminating the two conditional branch instructions from SM213 (and not adding any other instructions). Would compilers still be able to compile C programs to run on this modified machine? If so, explain how. If not, carefully explain what feature or features of C would be impossible to execute on the modified machine.

12 **(4 marks)** **Procedure Call and Return**.

12a Is a procedure call a static or dynamic jump? Justify your answer.

12b Is a procedure return a static or dynamic jump? Justify your answer.

13 **(10 marks)** **Writing Assembly Code**. Write SM213 assembly code that implements the following C program. Use labels for static addresses but do not include variable label declarations (i.e. "`.long`" lines). Show only the code for these two procedures. Do not implement a return from `callReplace()`; simply halt at the end of that procedure. Do not use the stack. Comment every line.

```
int* a;
int  searchFor, replaceWith, size, i;

void replace() {                        void callReplace() {
    for (i=0; i<size; i++)                  replace();
        if (a[i]==searchFor)                // halt; do not return
            a[i]=replaceWith;           }
}
```

14 **(20 marks)** The following SM213 assembly code implements a simple procedure. Carefully comment every line, give an equivalent C program that would compile into this assembly, and explain in plan English what this procedure does.

14c Plain English explanation of what this procedures does.

15 **(10 marks)** **Reading Assembly Code.** Consider the following snippet of SM213 assembly code.

```
foo: ld $s,     r0          # r0 = &s}
     ld 0(r0),  r1          # r1 = s.a}
     ld 4(r0),  r2          # r2 = s.b}
     ld 8(r0),  r3          # r3 = s.c}
     ld $0,     r0          # r0 = 0}
     not        r1          # }
     inc        r1          # r1 = -a}
L0:  bgt        r3, L1      # goto L1 if c>0}
     br         L9          # goto L9 if c<=0}
L1:  ld         (r2), r4    # r4 = *b}
     add        r1, r4      # r4 = *b-a}
     beq        r4, L2      # goto L2 if *b==a}
     br         L3          # goto L3 if *b!=a}
L2:  inc        r0          # r0 = r0 +1 if *b==a}
L3:  dec        r3          # c--}
     inca       r2          # a++}
     br         L0          # goto L0}
L9:  j          (r6)        # return}
```

15a Carefully comment every line of code above.

14c Plain English explanation of what this procedures does.

15 **(10 marks)** **Reading Assembly Code.** Consider the following snippet of SM213 assembly code.

```
foo: ld $s,     r0          # r0 = &s}
     ld 0(r0),  r1          # r1 = s.a}
     ld 4(r0),  r2          # r2 = s.b}
     ld 8(r0),  r3          # r3 = s.c}
     ld $0,     r0          # r0 = 0}
     not        r1          # }
     inc        r1          # r1 = -a}
L0:  bgt        r3, L1      # goto L1 if c>0}
     br         L9          # goto L9 if c<=0}
L1:  ld         (r2), r4    # r4 = *b}
     add        r1, r4      # r4 = *b-a}
     beq        r4, L2      # goto L2 if *b==a}
     br         L3          # goto L3 if *b!=a}
L2:  inc        r0          # r0 = r0 +1 if *b==a}
L3:  dec        r3          # c--}
     inca       r2          # a++}
     br         L0          # goto L0}
L9:  j          (r6)        # return}
```

15a Carefully comment every line of code above.

15b Give precisely-equivalent C code.

15c The code implements a simple function. What is it? Give the simplest, plain English description you can.

16 **(10 marks)** Implement the following in SM213 assembly. You can use a register for `c` instead of a local variable. **Comment every line.**

```
int  len;
int* a;
int countNotZero () {
    int c=0;
    while (len>0) {
        len=len-1;
        if (a[len]!=0)
            c=c+1;
    }
    return c;
}
```

5 **(12 marks)** Consider the following SM213 assembly code that implements a simple C procedure.

```
L0:  deca r5                #
     st   r6, (r5)          #
     ld   4(r5), r1         #
     ld   8(r5), r2         #
     ld   $0, r3            #
L1:  bgt  r2, L2            #
     br   L3                #
L2:  dec  r2                #
     ld   (r1, r2, 4), r4   #
     deca r5                #
     st   r4, (r5)          #
     gpc  $2, r6            #
     j    *16(r5)           #
     inca r5                #
     beq  r0, L1            #
     add  r4, r3            #
     br   L1                #
L3:  mov  r3, r0            #
     ld   (r5), r6          #
     inca r5                #
     j    (r6)              #
```

5a Comment every line in a way that illustrates the connection to corresponding C statements.

5b Give an equivalent C procedure (i.e., a procedure that may have compiled to this assembly code).

```
foo: ld $s,     r0           #
     ld 0(r0),  r1           #
     ld 4(r0),  r2           #
     ld 8(r0),  r3           #
     ld $0,     r0           #
     not        r1           #
     inc        r1           #
L0:  bgt        r3, L1       #
     br         L9           #
L1:  ld         (r2), r4     #
     add        r1, r4       #
     beq        r4, L2       #
     br         L3           #
L2:  inc        r0           #
L3:  dec        r3           #
     inca       r2           #
     br         L0           #
L9:  j          (r6)         #
```

8a Carefully comment every line of code above.

8b Give precisely-equivalent C code.

8c The code implements a simple function. What is it? Give the simplest, plain English description you can.

9 **(5 marks)** **Pointers in C** Consider the follow declarations in C.

```
int  a[10] = 0,2,4,6,8,10,12,14,16,18; // a[i] = 2*i;
int* b     = &a[4];
int* c     = a+4;
```

Answer the following questions. Show your work for the last question.

9a What is the *type* of the variable `a`?

9b What is the value of `b[4]`?

9c What is the value of `c[4]`?

9d What is the value of `*(a+4)`?

9e What is the value of `b-a`?

9f What is the value of `*(&a[3] + *(a+(&a[3]-&a[2])))`?

10 **(3 marks)** **Mystery Variable 1** This code stores 0 in a variable.

```
ld $0, r0
st r0, 8(r5)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

11 **(3 marks)** **Mystery Variable 2** This code stores 0 in a variable.

```
ld $0, r0
ld $3, r1
ld $0x1000, r2
st r0, (r2, r1, 4)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

12 **(3 marks)** **Mystery Variable 3** This code stores 0 in a variable.

```
ld $0, r0
ld $3, r1
ld $0x1000, r2
ld (r2), r2
st r0, (r2, r1, 4)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

13 **(3 marks)** **Mystery Variable 4** This code stores 0 in a variable.

```
    ld $0, r0
    ld $0x1000, r2
    ld (r2), r2
    st r0, 8(r2)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

14 (9 marks) Dynamic Storage

14a Carefully explain how a C program can create a *dangling pointer* and what bad thing might happen if it does.

14b Carefully explain how a C program can create a *memory leak* and what bad thing might happen if it does.

14c Can either or both of these two problems occur in a Java program? Briefly explain.

15 (10 marks)

Implement the following in SM213 assembly. You can use a register for `c` instead of a local variable. **Comment every line.**

```
int  len;
int* a;
int countNotZero () {
    int c=0;
    while (len>0) {
        len=len-1;
        if (a[len]!=0)
            c=c+1;
    }
    return c;
}
```

5 **(12 marks)** **Read Assembly Code.** Consider the following SM213 code.

```
X:  deca  r5               #
    deca  r5               #
    st    r6, 4(r5)        #
    ld    $0, r1           #
    st    r1, 0(r5)        #
    ld    12(r5), r2       #
    ld    16(r5), r3       #
    not   r3               #
    inc   r3               #
L0: mov   r1, r4           #
    add   r3, r4           #
    beq   r4, L2           #
    bgt   r4, L2           #
    ld    (r2,r1,4), r4    #
    deca  r5               #
    st    r4, 0(r5)        #
    gpc   $2, r6           #
    j     *12(r5)          #
    inca  r5               #
    ld    $1, r4           #
    and   r0, r4           #
    beq   r4, L1           #
    ld    0(r5), r4        #
    add   r0, r4           #
    st    r4, 0(r5)        #
L1: inc   r1               #
    br    L0               #
L2: ld    0(r5), r0        #
    ld    4(r5), r6        #
    inca  r5               #
    inca  r5               #
    j     (r6)             #
```

5a Add a comment to every line of code. Where possible use variables names and C pseudo code in your comments to clarify the connection between the assembly code and corresponding C statements.

5b Give an equivalent C procedure (i.e., a procedure that may have compiled to this assembly code).

6 **(3 marks)** **Programming in C.** Consider the following C code.

```
int* b;

void set (int i) {
    b [i] = i;
}
```

Is there a bug in this code? If so, carefully describe what it is.

7 **(6 marks)** **Programming in C.** Consider the following C code.

```
int* one () {
    int loc = 1;
    return &loc;
}

void two () {
    int zot = 2;
}
```

```
void three () {
    int* ret = one();
    two();
}
```

7a Is there a bug in this code? If so, carefully describe what it is.

7b What is the value of "`*ret`" at the end of `three`? Explain carefully.

14a

```
X:  ld    0(r5), r0         # r0 = item}
    ld    4(r5), r1         # r1 = list}
    ld    8(r5), r2         # r2 = i = n}
    dec   r2                # i = i - 1}
L0: bgt   r2, L1            # goto L1(cont) if i > 0}
    beq   r2, L1            # goto L1(cont) if i >= 0}
    br    L2                # goto L2(done) if i < 0}
L1: ld    (r1, r2, 4), r3   # r3 = list [i]}
    mov   r3, r4            # r4 = list [i]}
    not   r4                # r4 = ˜ list [i]}
    inc   r4                # r4 = - list [i]}
    add   r0, r4            # r4 = item - list [i]}
    bgt   r4, L2            # goto L2(done) if (item > list[i])}
    inc   r2                # r2 = i + 1}
    st    r3, (r1, r2, 4)   # list [i + 1] = list [i] if (item <= list [i])}
    dec   r2                # r2 = i}
    dec   r2                # i = i - 1}
    j     L0                # goto L0(loop)}
L2: inc   r2                # r2 = i + 1}
    st    r0, (r1, r2, 4)   # list [i + 1] = item}
    j     (r6)              # return}
```

14b Equivalent C program that would compile into this assembly:

8. Reading Assembly Code [10 marks]

a) Add high-level, C-like comments to the following assembly program. Be sure to clearly indicate the location of high-level structure such as *loops*, *if-statements*, *reading* from or *writing* to *variables* or *arrays* etc. Partial marks on this question will be determined by the amount of high-level structure you identify.

```
     ld  $a, r0            # r0 = &a
     ld  (r0), r0          # r0 = a
     ld  $b, r1            # r1 = &b
     ld  (r1), r1          # r1 = b = b'
     not r1                # r1 = ~b'
     inc r1                # r1 = -b'
     ld  $0, r2            # r2 = i' = 0
L0:  mov r2, r3            # r3 = i'              <= top of outer for loop (i')
     add r1, r3            # r3 = i' - b'
     beq r3, L4            # goto L4 if i' == b'
     mov r2, r3            # r3 = i'
     inc r3                # r3 = j' = i' + 1
L1:  mov r3, r4            # r4 = j'             <= top of inner for loop (j')
     add r1, r4            # r4 = j' - b'
     beq r4, L3            # goto L3 if j' == b'
     ld (r0, r2, 4), r5    # r5 = a[i']          <= load two values from array
     ld (r0, r3, 4), r6    # r6 = a[j']
     mov r5, r7            # r7 = a[i']
     not r7                # a7 = ~a[i']
     inc r7                # a7 = -a[i']
     add r6, r7            # a7 = a[j'] - a[i']
     bgt r7, L2            # goto L2 if a[j'] > a[i']      <= IF statement
     st r6, (r0, r2, 4)    # a[i'] = a[j']'  if a[j'] <= a[i']  <= THEN: swap two
     st r5, (r0, r3, 4)    # a[j'] = a[i']'  if a[j'] <= a[i']  <= array elements
L2:  inc r3                # j'++
     br L1                 # goto top of inner loop    <= end of inner loop
L3:  inc r2                # i'++
     br L0                 # goto top of outer loop    <= end of outer loop
L4:  halt
```

b) Give C code that does what this assembly code does. For full credit, your code must be straight-forward C, like you would write if you were writing this from scratch and not just translating it from the assembly code.

```
for (int i=0; i<b; i++) {
    for (int j=i+1; j<b; j++) {
        if (a[j] < a[i]) {
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
}
```

c) Give a plain-English description of what this code does. Your description must be a clear, simple, high-level description of what the code achieves, not a step by step description of how it does it.

Use Bubble Sort to reorder a list of integers in ascending order.

```
foo:      deca r5               #  allocate stack
          st r6, (r5)           #  save ra
          ld $0, r0             #  result = 0
          ld $0, r1             #  i = 0
          ld 4(r5), r2          #  r2 = arg0
          ld 8(r5), r3          #  r3 = arg1
          ld 12(r5), r4         #  r4 = arg2
L0:       mov r3, r6            #  r6 = arg1
          not r6
          inc r6                #  r6 = -arg1
          add r1, r6            #  r6 = i-arg1
          beq r6, L3            #  if (i==arg1) goto L3
          ld (r2, r1, 4), r6    #  r6 = arg0[i]
          not r6
          inc r6                #  r6 = -arg0[i]
          add r4, r6            #  r6 = arg3-arg0[i]
          beq r6, L1            #  if (arg0[i]==arg3) goto L1
          br L2                 #  else goto L2
L1:       inc r0                #  then result++;
L2:       inc r1                #  i++;
          br L0                 #  goto L0 (loop start)
L3:       ld (r5), r6           #  load ra
          inca r5               #  de-allocate stack
          j (r6)                #  return result
```

8a Translate into C:

```
   int foo(int* arg0, int arg1, int arg2) {
       int result = 0;
       for (int i = 0; i < arg1; i++)
           if (arg0[i] == arg2)
               result++;
       return result;
   }
```

8b Explain what the code does in one sentence.

Counts how many values in the arg0 array are equal to arg2

```
foo:    deca r5           # sp=sp-4
        deca r5           #  int result
        st   r6, 4(r5)    # push return address   M[sp]=ra
        ld   8(r5), r1    #  r1 = n (argument)
        bgt  r1, L1       #  if (r1>0) goto L1  if (n>0) goto L1
        ld $0, r2         # if-part r0=0 or result=0   result=0
        br L3             # goto endif
L1:     dec r1            # r1 = r1-1
        bgt r1, L2        # elseif (r1>0) goto L2  if (n-1 > 0) goto L2
        ld $1, r2         # r0=1   result=1    result=1
        br L3             # goto endif
L2:     ld 8(r5), r1      # else: r1 = n (argument)
        dec r1            # r1=n-1
        deca r5           #  sp=sp-4
        st r1, 0(r5)      #  M[sp] = n-1
        gpc $0x6, r6      # call foo(n-1)
        j foo             # jump foo
        inca r5           # sp=sp+4, pop argument
        st r0, 0(r5)      # result=r0
        ld 8(r5), r1      # else: r1 = n (argument)
        dec r1            # r1=r1-1 (n-1)
        dec r1            # r1=r1-1 (n-2)
        deca r5           # sp=sp-4
        st r1, 0(r5)      # push argument n on stack
        gpc $0x6, r6      # call foo(n-2)
        j foo             # jump foo
        inca r5           # sp=sp+4, pop argument
        ld (r5), r2       # r2=result
        add r0, r2        # r0 = foo(n-1) + foo(n-2)   result= foo(n-1) + foo(n-2)
L3:     mov r2 ,r0        # return r2 (result)
        ld 4(r5), r6      # pop return address
        inca r5           # sp=sp+4, pop local address
        inca r5           # sp=sp+4
        j (r6)            # return
```

4a Carefully comment every line of code above.

4b Convert the assembly into C.

```
int fib(int n) {
    int result;
    if (n==0)
            result = 0;
    else if (n==1)
            result = 1;
    else {
            result = fib(n-1);
            result = result + fib(n-2);
    }
    return result;
}
```

4c The code implements a simple function. What is it? Give the simplest, plain English description you can.

It computes f(n) = f(n-1) + f(n-2) with f(0)=0 and f(1)=1, non-negative n.

Marking Rubic: If (c) is correct just look over (b), otherwise look at (b) and give 2 marks IF, 4 marks for recursive calls, and 2 marks for correct return (sum), otherwise (a) give up to 1/2 the marks for the same concepts if there is some understanding of them given in the commenting of the assembly.

5 **(4 marks)** **Procedure Calls.** Given these global declarations

```
replace:      ld   $size, r0           # r0 = &size
              ld   0x0(r0), r0         # r0 = size = i
              ld   $a, r1              # r1 = &a
              ld   0x0(r1), r1         # r1 = a
              ld   $searchFor, r2      # r2 = &searchFor
              ld   0x0(r2), r2         # r2 = searchFor
              not  r2                  # r2 = !searchFor
              inc  r2                  # r2 = -searchFor
              ld   $replaceWith, r3    # r3 = &replaceWith
              ld   0x0(r3), r3         # r3 = replaceWith
loop:         beq  r0, done            # goto done if i==0
              dec  r0                  # i--
              ld   (r1, r0, 4), r4     # r4 = a[i]
              add  r2, r4              # r4 = a[i] - searchFor
              beq  r4, match           # goto match   if a[i]==searchFor
              br   nomatch             # goto nomatch if a[i]!=searchFor
match:        st   r3, (r1, r0, 4)     # a[i] = replaceWith
nomatch:      br   loop                # goto loop
done:         j    0x0(r6)             # return
callReplace:  gpc  $0x6, r6            # ra = pc + 6
              j    replace             # replace()
              halt
```

14 **(20 marks)** The following SM213 assembly code implements a simple procedure. Carefully comment every line, give an equivalent C program that would compile into this assembly, and explain in plan English what this procedure does.

14a

```
X:  ld    0(r5), r0          # { r0 = item}
    ld    4(r5), r1          # { r1 = list}
    ld    8(r5), r2          # { r2 = i = n}
    dec   r2                 # { i = i - 1}
L0: bgt   r2, L1             # { goto L1(cont) if i > 0}
    beq   r2, L1             # { goto L1(cont) if i >= 0}
    br    L2                 # { goto L2(done) if i < 0}
L1: ld    (r1, r2, 4), r3    # { r3 = list [i]}
    mov   r3, r4             # { r4 = list [i]}
    not   r4                 # { r4 = ˜ list [i]}
    inc   r4                 # { r4 = - list [i]}
    add   r0, r4             # { r4 = item - list [i]}
    bgt   r4, L2             # { goto L2(done) if (item > list[i])}
    inc   r2                 # { r2 = i + 1}
    st    r3, (r1, r2, 4)    # { list [i + 1] = list [i] if (item <= list [i])}
    dec   r2                 # { r2 = i}
    dec   r2                 # { i = i - 1}
    j     L0                 # { goto L0(loop)}
L2: inc   r2                 # { r2 = i + 1}
    st    r0, (r1, r2, 4)    # { list [i + 1] = item}
    j     (r6)               # { return}
```

14b Equivalent C program that would compile into this assembly:

```
void insertIntoSortedList (int item, int* list, int n) {
    for (int i = n - 1; i >= 0 && item <= list [i]; i --)
        list [i + 1] = list [i];
    list [i + 1] = item;
}
```
Or:
```
void insertIntoSortedList (int item, int* list, int n) {
    for (int i = n - 1; i >= 0; i --) {
        if (item > list [i])
            break;
        list [i + 1] = list [i];
    }
    list [i + 1] = item;
}
```

14c Plain English explanation of what this procedures does.

It inserts an integer into a sorted, ascending list of integers, maintaining sort order.

15 (10 marks) **Reading Assembly Code.** Consider the following snippet of SM213 assembly code.

```
foo: ld $s,     r0          # { r0 = &s}
     ld 0(r0),  r1          # { r1 = s.a}
     ld 4(r0),  r2          # { r2 = s.b}
     ld 8(r0),  r3          # { r3 = s.c}
     ld $0,     r0          # { r0 = 0}
     not        r1          # { }
     inc        r1          # { r1 = -a}
L0:  bgt        r3, L1      # { goto L1 if c>0}
     br         L9          # { goto L9 if c<=0}
L1:  ld         (r2), r4    # { r4 = *b}
     add        r1, r4      # { r4 = *b-a}
     beq        r4, L2      # { goto L2 if *b==a}
     br         L3          # { goto L3 if *b!=a}
L2:  inc        r0          # { r0 = r0 +1 if *b==a}
L3:  dec        r3          # { c--}
     inca       r2          # { a++}
     br         L0          # { goto L0}
L9:  j          (r6)        # { return}
```

15a Carefully comment every line of code above.

15b Give precisely-equivalent C code.

```
struct S {
    int  a;
    int* b;
    int  c;
};
S s;
int foo () {
    int  i=0;
    int* b=s.b;

    while (s.c>0) {
        if (s.a==*b)
            i++;
        s.c--;
        b++;
    }
    return i;
}
```

Or

```
int foo () {
    int i=0,j;

    for (j=0; j<s.c; j++)
        if (s.a==s.b[j])
            i++;
    return i;
}
```

15c The code implements a simple function. What is it? Give the simplest, plain English description you can.

It counts the number of elements in the integer array `s.b` whose size is `s.c` that have the value `s.a` and returns this number.

16 **(10 marks)** Implement the following in SM213 assembly. You can use a register for `c` instead of a local variable. **Comment every line.**

```
int  len;
int* a;
int countNotZero () {
    int c=0;
    while (len>0) {
        len=len-1;
        if (a[len]!=0)
            c=c+1;
    }
    return c;
}
```

```
countZero: ld  $len, r1         # r1 = &len
           ld  0(r1), r1        # r1 = len
           ld  $a, r2           # r2 = &a
           ld  0(r2), r2        # r2 = a
           ld  $0, r0           # r0 = c
loop:      bgt r1, cont         # goto cont if len>0
           br  done             # goto done if len<=0
cont:      dec r1               # len = len - 1
           ld  (r2, r1, 4), r3  # r3 = a[len]
           beq r3, loop         # goto skip if a[len]==0
           inc r0               # c=c+1 if a[len]!=0
           br  loop             # goto loop
done:      j   (r6)             # return c
```

```
L0:  deca r5               #  make stack space for saved ra
     st   r6, (r5)         #  store saved ra on stack
     ld   4(r5), r1        #  r1 = a
     ld   8(r5), r2        #  r2 = t_i = n
     ld   $0, r3           #  r3 = t_s = 0
L1:  bgt  r2, L2           #  goto L2 if t_i > 0
     br   L3               #  goto L3 if t_i <= 0
L2:  dec  r2               #  t_i --
     ld   (r1, r2, 4), r4  #  r4 = a[t_i]
     deca r5               #  make stack space for arg
     st   r4, (r5)         #  arg = a[t_i]
     gpc  $2, r6           #  r6 = return address
     j    *16(r5)          #  t_j = f (a[t_i])
     inca r5               #  free stack space for arg
     beq  r0, L1           #  goto L1 if t_j == 0
     add  r4, r3           #  t_s += a[i] if t_j != 0
     br   L1               #  goto L1
L3:  mov  r3, r0           #  r0 = t_s
     ld   (r5), r6         #  r6 = saved return address
     inca r5               #  free stack space for ra
     j    (r6)             #  return t_s
```

5a Comment every line in a way that illustrates the connection to corresponding C statements.

5b Give an equivalent C procedure (i.e., a procedure that may have compiled to this assembly code).

```
int foo (int* a, int n, int (*f)(int)) {
    int s = 0;
    for (int i=n-1; i>=0; i--)
        if f (a [i])
            s += a[i]
    return s;
}
```

```
        ld $a, r0               # r0 = &a = &[0]
        ld $0, r1               # r1 = temp_i = 0
        ld $0, r2               # r2 = temp_s = 0
        ld (r0, r1, 4), r3      # r3 = a[temp_i]
        add r3, r2              # temp_s = temp_s + a[temp_i]
        ld $s, r4               # r4 = &s
        st r2, (r4)             # s = temp_s
```

7h `e->k->i = 0;`

Three: read value of `e`; read value of `k`; store value in variable.

8 **(10 marks)** **Reading Assembly Code.** Consider the following snippet of SM213 assembly code.

```
foo: ld $s,     r0          # r0 = &s
     ld 0(r0),  r1          # r1 = s.a
     ld 4(r0),  r2          # r2 = s.b
     ld 8(r0),  r3          # r3 = s.c
     ld $0,     r0          # r0 = 0
     not        r1          #
     inc        r1          # r1 = -a
L0:  bgt        r3, L1      # goto L1 if c>0
     br         L9          # goto L9 if c<=0
L1:  ld         (r2), r4    # r4 = *b
     add        r1, r4      # r4 = *b-a
     beq        r4, L2      # goto L2 if *b==a
     br         L3          # goto L3 if *b!=a
L2:  inc        r0          # r0 = r0 +1 if *b==a
L3:  dec        r3          # c--
     inca       r2          # a++
     br         L0          # goto L0
L9:  j          (r6)        # return
```

8a Carefully comment every line of code above.

8b Give precisely-equivalent C code.

```
struct S {
    int  a;
    int* b;
    int  c;
};
S s;
int foo () {
    int i=0;

    while (s.c>0) {
        if (s.a==*s.b)
            i++;
        s.c--;
        b++;
    }
    return i;
}
```
Or
```
int foo () {
    int i=0,j;

    for (j=0; j<s.c; j++)
        if (s.a==s.b[j])
            i++;
    return i;
}
```

8c The code implements a simple function. What is it? Give the simplest, plain English description you can.

It counts the number of elements in the integer array `s.b` whose size is `s.c` that have the value `s.a` and returns this number.

9 (5 marks) Pointers in C

Consider the follow declarations in C.

```
int  a[10] = 0,2,4,6,8,10,12,14,16,18; // a[i] = 2*i;
int* b     = &a[4];
int* c     = a+4;
```

Answer the following questions. Show your work for the last question.

9a What is the *type* of the variable `a`?

`int*`

9b What is the value of `b[4]`?

`16`

9c What is the value of `c[4]`?

`16`

9d What is the value of `*(a+4)`?

`8`

9e What is the value of `b-a`?

`4`

9f What is the value of `*(&a[3] + *(a+(&a[3]-&a[2])))`?

```
= *((a+3) + *(a+(a+3)-(a+2)))
= *((a+3) + *(a+1))
= *((a+3) + a[1])
= *((a+3) + 2)
= *(a+5)
= a[5]
= 10
```

10 **(3 marks)** **Mystery Variable 1** This code stores 0 in a variable.

```
ld $0, r0
st r0, 8(r5)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

Local or argument int.

11 **(3 marks)** **Mystery Variable 2** This code stores 0 in a variable.

```
ld $0, r0
ld $3, r1
ld $0x1000, r2
st r0, (r2, r1, 4)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

Static array of ints.

12 **(3 marks)** **Mystery Variable 3** This code stores 0 in a variable.

```
ld $0, r0
ld $3, r1
ld $0x1000, r2
ld (r2), r2
st r0, (r2, r1, 4)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

Dynamic array of ints.

13 **(3 marks)** **Mystery Variable 4** This code stores 0 in a variable.

```
ld $0, r0
ld $0x1000, r2
ld (r2), r2
st r0, 8(r2)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

Entry of type int in dynamic, global struct.

14 **(9 marks)** **Dynamic Storage**

14a Carefully explain how a C program can create a *dangling pointer* and what bad thing might happen if it does.

If it retains a pointer to heap-allocated storage after it has been freed and then dereferences this pointer. The program could write to or read from a part of another, unrelated struct, array or variable that is stored in the freed, but pointed-to memory.

14b Carefully explain how a C program can create a *memory leak* and what bad thing might happen if it does.

If it fails to free heap-allocated storage after it is no longer needed by the program. The program's memory size could grow to the point where it no longer fits in available memory on the machine.

14c Can either or both of these two problems occur in a Java program? Briefly explain.

Dangling pointers can not exist, because memory is only freed by the garbage collector when there are not pointers referring to it. Memory leaks can occur when a program inadvertently retains references to objects that it no longer needs.

15 **(10 marks)** Implement the following in SM213 assembly. You can use a register for `c` instead of a local variable. **Comment every line.**

5 **(12 marks)** **Read Assembly Code.** Consider the following SM213 code.

```
X:  deca   r5                # allocate space for return address on stack
    deca   r5                # allocate local variable on stack (x)
    st     r6, 4(r5)         # store return address on stack
    ld     $0, r1            # r1 = 0 (i)
    st     r1, 0(r5)         # local var x = 0
    ld     12(r5), r2        # r2 = arg 2 (b)
    ld     16(r5), r3        # r3 = arg 3 (c)
    not    r3                # r3 = ~c (bitwise NOT)
    inc    r3                # r3 = -c
L0: mov    r1, r4            # r4 = i
    add    r3, r4            # r4 = i - c
    beq    r4, L2            # goto L2 if i - c == 0, ie i == c
    bgt    r4, L2            # goto L2 if i - c > 0, ie i > c
    ld     (r2,r1,4), r4     # r4 = b[i]
    deca   r5                # allocate space for arg (to pass) on stack
    st     r4, 0(r5)         # arg (to pass) = b[i]
    gpc    $2, r6            # r6 = return address
    j      *12(r5)           # call a(b[i]) where a is arg 1 that we received
    inca   r5                # deallocate arg off stack
    ld     $1, r4            # r4 = 1
    and    r0, r4            # r4 = a(b[i]) & 1
    beq    r4, L1            # goto L1 if a(b[i]) & 1 == 0
    ld     0(r5), r4         # r4 = x
    add    r0, r4            # r4 = x + a(b[i])
    st     r4, 0(r5)         # x = r4, ie x += a(b[i])
L1: inc    r1                # i++
    br     L0                # goto L0
L2: ld     0(r5), r0         # r0 = x
    ld     4(r5), r6         # r6 = return address
    inca   r5                # deallocate local variable
    inca   r5                # deallocate return address
    j      (r6)              # return x
```

5a Add a comment to every line of code. Where possible use variables names and C pseudo code in your comments to clarify the connection between the assembly code and corresponding C statements.

If you do this part of the question well, the next section is trivial. The important thing to do here is to identify every single variable that is in the code and assign it a name so you can figure out what the code is trying to do. Every variable here can be classified as one of the following (the names can be whatever you like, as long as they're consistent):

1. Local variables on the stack (`int x`) and in registers (`int i`)
2. Arguments passed to X (`int (*a)(int)`, `int* b`, `int c`)
3. Arguments passed by X to whatever it calls (`b[i]`)

The most difficult one to identify is the variable `i` stored in r1. It's initialized to 0, but then also saved on the stack *to initialize another variable* which we call `x` here. You know that `x` is a local variable because it's saved to the stack and not near a function call. We know that `i` is its own variable because r1 goes on to have a value other than what's stored in `x`.

It is extremely important to be aware of the layout of the stack throughout the execution of the program. I'd highly recommend drawing out the stack so that when you see load/store offsets, you can figure out exactly which variable it's referring to. If you're not sure how I determined that `b` and `c` are arguments 2 and 3 (instead of 1 and 2), draw the stack. This will also help you figure out that the instruction `j *12(r5)` is reading a function pointer off the stack.

Notice how in the comments, I always try to refer to values by their variable name rather than register name (as the question asks me to do). This will help immensely in the next step. Also notice how I've commented and rearranged the conditional branches - this will also help you when translating the code to C.

5b Give an equivalent C procedure (i.e., a procedure that may have compiled to this assembly code).

```
int X(int (*a)(int), int* b, int c) {
  int x = 0;
  for (int i = 0; i < c; i++) {
    if (a(b[i]) & 1)
        x += a(b[i]);
  }
  return x;
}
```

Pay attention to how the loop guard `i < c` was derived from the assembly code - the assembly code said `goto L2 if i >= c` meaning "end the loop if `i >= c`". The opposite of this is then "only enter the loop if `i < c`". This is one possible way the loop could have been written. If you wanted to translate the assembly code exactly as you saw it, this is what you would get:

```
int i = 0;
while (1) {
  if (i - c == 0) break;
  if (i - c > 0) break;
  if (a(b[i]) & 1)
    x += a(b[i]);
  i++;
}
```

You probably wouldn't lose marks for this, but I think the way it's written in the answer is more likely to be the original C code. Learn to identify common looping patterns like iterating from 0 to some number. At this point, you may want to even rename your variables to show that you really understand what the code is doing. Something like this:

```
int X(int (*fn)(int), int* array, int size) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    if (fn(array[i]) & 1)
      sum += fn(array[i]);
  }
  return sum;
}
```

You don't have to name every variable like this in an exam. I do this here just to show you exactly what the code is doing. However if you do choose to use good variable names, it will help you convince yourself that you've translated the code correctly. It should be clear now that this code is applying the given function to all odd elements in the array and summing the results. (`n & 1 == 1` implies `n` is odd).

6 **(3 marks)** **Programming in C.** Consider the following C code.

```
int* b;

void set (int i) {
    b [i] = i;
}
```

Is there a bug in this code? If so, carefully describe what it is.

Yes there is a bug in this code. The function does not performing bounds checking on the array `b` and we are not given its size, so this could modify anything in memory.

9 **(10 marks)** **Reading Assembly.** Comment the following assembly code and then translate it into C. *Use the back of the preceding page for extra space if you need it.*

```
foo:    ld   $-12, r0           # r0 = stack space for ra and 2 locals
        add  r0, r5             # allocate stack space for ra and 2 locals
        st   r6, 8(r5)          # save ra to stack
        ld   $0, r1             # i = 0
        st   r1, 0(r5)          # loc0 = 0
        st   r1, 4(r5)          # loc1 = 0 (later realized that this is i)
        ld   20(r5), r2         # r2 = arg2
        not  r2                 #
        inc  r2                 # r2 = -arg2
L0:     mov  r2, r3             # r3 = -arg2
        add  r1, r3             # r3 = i-arg2
        beq  r3, L3             # goto L3 if i > arg2
        bgt  r3, L3             # goto L3 if i == arg2
        ld   12(r5), r3         # r3 = arg0
        ld   (r3, r1, 4), r3    # r3 = arg0[i]
        ld   16(r5), r4         # r4 = arg1
        ld   (r4, r1, 4), r4    # r1 = arg1[i]
        ld   $-8, r0            # r0 = space for 2 arguments
        add  r0, r5             # allocate stack space for 2 arguments
        st   r3, 0(r5)          # save bar_arg0 = arg0[i] to stack
        st   r4, 4(r5)          # save bar_arg1 = arg1[i] to stack
        gpc  $6, r6             # r6 = return address for call to bar
        j    bar                # t = bar (arg0[i], arg1[i])
        ld   $8, r3             # r3 = space for 2 arguments
        add  r3, r5             # deallocate stack space for 2 arguments
        beq  r0, L2             # goto L2 if t==0
        ld   0(r5), r3          # r3=loc0 if t
        inc  r3                 # r3++ if t
        st   r3, 0(r5)          # loc0++ if t
L2:     inc  r1                 # i++
        br   L0                 # goto L0 (top of loop)
L3:     ld   0(r5), r0          # r0=loc0
        ld   8(r5), r6          # restore ra from stack
        ld   $12, r1            # r1 = stack space for ra and 2 locals
        add  r1, r5             # deallocate stack space for ra and 2 locals
        j    (r6)               # return loc0
```

Translate into C:

```
int foo (int* arg0, int* arg1, int arg2) {
    int loc0 = 0;
    int i    = 0;
    for (i=0; i<a2; i++)
        if (bar (arg0[i], arg1[i]))
            loc0++;
    return loc0;
}
```