3. Structs and Instance Variables [9 marks]

Consider the following code, where `f` is a global variable, assuming whatever assembly labels make sense, and where the **size of a pointer is 8-bytes**. Use `r0` for the variable `i`. No comments needed.

```
struct X {
    int* a;
    int  b[4];
    struct Y c;
}

struct X* f;
```

```
struct Y {
    int d;
    struct X* e;
}
```

a) Give assembly code for `i = f->c.e->a[1];` *(i.e., store the result in `r0`)*

b) Give assembly code for `f->b[1] = i;` *(i.e., just use `r0` for `i`)*

c) What is the minimum number of memory references required to compute the value of the following expression (not including fetching the instructions themselves from memory)?

```
f->c.e->c.e[0].a[0]
```

○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7

3 **[10 marks]** **Structs and Instance Variables.** Answer the following questions about the global variables `a` and `b`, declared as follows. Assume that `int`'s and pointers are **4-bytes** long.

```
struct A {
    char x;
    int y[3];
    struct B* z;
};

struct A* a;
struct B  b;
```

```
struct B {
    struct A j;
    int k;
};
```

Treat each sub-question separately (i.e., do not use values computed in prior sub-questions). Comments are not required, but they will probably help you.

3a Give assembly code for the C statement: `b.k = a->z[2].k;`.

3b How many memory reads are required to execute the statement? Fill in a single multiple choice option below.

```
b.k = a->z->j.z->j.y[2];
```

0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8 ○

3 (6 marks) Instance and Local Variables. Give the SM213 assembly code for the following statements that access elements of global and local structs. Consider each statement as if it were the last line of `foo()` at the location indicated by the comment. Use labels such as `a`, `b` etc. for statically known values. Assume that `r5` stores the value of the stack pointer. Comments are not required. You can treat your answers as a sequence and thus use register values loaded in one subquestion in subsequent ones to avoid duplication.

```
struct S {
    int       w;
    int       x;
    struct S* y;
    struct T  z;
};

struct T {
    int w;
    int x;
};

int       i;
struct S* a;

void foo() {
    struct S* b;
    // QUESTION CODE IS HERE
}
```

3a `i = a->y->x;`

3b `i = a->z.x;`

3c `i = b->x;`

2 (6 marks) Static Scalars and Arrays. Consider the following C code containing global variables `s` and `d`.

```
int  s[2];
int* d;
```

Use `r0` for the variable `i` (i.e., do not read or write memory for `i`) and use labels for static values. Give assembly code that is equivalent to each of the following C statements. Treat each question separately; i.e., do not use register values assigned in a previous answer.

2a `i = s[i];`

2b `i = d[i];`

2c `d = &s[10];`

3 (6 marks) Structs and Instance Variables Consider the following C code containing the global variable `a`.

```
struct S {
    int       x;
    int       y;
    struct S* z;
};

struct S* a;
```

Once again use `r0` for the variable `i` (i.e., do not read or write memory for `i`) and use labels for static values. Give assembly code that is equivalent to each of the following C statements. Treat each question separately; i.e., do not use register values assigned in a previous answer.

3a `i = a->y`

3b `i = a->z->y;`

3c `a->z->z = a;`

4 **(3 marks)** **Instance Variables**. Consider the following C global variable declarations.

```
struct S {
    int   a;
    void* b;
    int   c;
};

struct S* s;
```

Give the SM213 assembly code the compiler might generate for the statement:

```
s->b = &s->c;
```

You may use the label `s`. You may not assume anything about the value of registers. **Comment every line.**

5 **(6 marks)** **Count Memory References.** Consider the following C global variable declarations.

```
struct S {
    int a[10];
};

struct S s;
```

```
struct T {
    int* x;
};

struct T* t;
```

For each question, count the number of memory **reads and writes** occur when the statement executes. Do not count the memory reads that fetch instructions. Justify your answer carefully by describing the reads and writes that occur.

5a `s.a[2] = s.a[3];`

5b `t->x[2] = t->x[3];`

4 **(3 marks)** **Instance Variables**. Consider the following C global variable declarations.

```
struct S {
    int   a;
    void* b;
    int   c;
};

struct S* s;
```

Give the SM213 assembly code the compiler might generate for the statement:

```
s->b = &s->c;
```

You may use the label `s`. You may not assume anything about the value of registers. **Comment every line.**

5 **(6 marks)** **Count Memory References.** Consider the following C global variable declarations.

```
struct S {
    int a[10];
};

struct S s;
```

```
struct T {
    int* x;
};

struct T* t;
```

For each question, count the number of memory **reads and writes** occur when the statement executes. Do not count the memory reads that fetch instructions. Justify your answer carefully by describing the reads and writes that occur.

5a `s.a[2] = s.a[3];`

5b `t->x[2] = t->x[3];`

4b `&a[3]`

4c `b[3]`

5 **(6 marks)** **Instance Variables.** In the context of the following C declarations:

```
struct S {                              struct S  a;
    int i,j;                            struct S* b;
};
```

Indicate what, if anything, about the following expressions can not be computed statically. Be precise.

5a `&a.i`

5b `&b->i`

5c `(&b->j)-(&b->i)`

6 **(3 marks)** **Memory Endianness** Examine this C code.

```
char a[4];
*((int*) (&a[0])) = 1;
```

Carefully explain how this code can be used to determine the *endianness* of the machine on which it runs, recalling that on a *Big Endian* machine, the high-order (most-significant) byte of a number has the lowest address.

7 **(8 marks)** Consider the following C declarations.

```
struct S {                              int       a[10];
  int       i;                          int*      b;
  int       j[10];                      int       c;
  struct S* k;                          struct S  d;
};                                      struct S* e;
```

For each of the following questions indicate the total number of memory references (i.e., distinct loads and/or stores) required to execute the listed statement. Justify your answers carefully.

7a `a[3] = 0;`

7b `a[c] = 0;`

7c `b[c] = 0;`

7d `d.i = 0;`

7e `d.j[3] = 0;`

7f `e->i = 0;`

7g `d.k->i = 0;`

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

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

4 **(4 marks)** **Instance Variables**. Consider the following C global variable declarations.

```
struct S {
    int  a;
    int* b;
    int  c;
};

struct S  s0;
struct S* s1;
```

Give the SM213 assembly code the compiler might generate for the following statements that access these variables. Assume the value `0` is in `r0`. Use labels `s0` and `s1` for variable addresses.

4a `s0.c = 0;`

4b `s1->c = 0;`

5 (8 marks) Count Memory References. Consider the following C global variable declarations.

```
struct S {
    struct S* s;
    int*      a;
    int       b[10];
};

struct S s0;
```

And this statement found in some procedure.

```
int v = s0.s->a[s0.s->b[2]];
```

List every memory read that will occur when this statement executes in the SM213 machine. List only memory reads and list each read separately. For each read give the name of the variable being read and an expression that computes the target memory address. This expression may only contain the label `s0`, numbers, the name of any variable previously read and arithmetic operators such as for addition and multiplication. Numbers must be immediately followed by a description in parentheses (e.g., "...4 (size of int)"). There are more lines listed below than you need:

1. Variable:

   Address:

2. Variable:

   Address:

3. Variable:

   Address:

4. Variable:

   Address:

5. Variable:

   Address:

6. Variable:

   Address:

3 **(8 marks)** **Instance Variables and Local Variables.** Consider the following C declaration of global variables.

```
struct X {
    int a;
    int b;
};
struct X c;
struct X* d;
```

Which of the following can be computed statically? Justify your answers.

3a `&c.a`

3b `&d->a`

3c `(&d->a) - (&d->b)`

Now answer this question.

3d Give the assembly code the compiler would generate for "`d->b = c.b;`".

5 **(5 marks)** **Instance Variables.** In the context of the following C declarations:

```
struct S {
    int i[3];
    int j[4];
    int k;
};
struct S  a;
struct S* b;
```

5a Indicate which of the following the compiler knows statically and which is determined dynamically.

- `&(a.i[2])`
- `&(b->j[2])`
- `&(b->k)`
- `(&(b->j[1]) - &(b->j[2]))`

5b Give SM213 assembly code that reads the value of `b->k` into `r0`. Comment your code.

3. Structs and Instance Variables [9 marks]

Consider the following code, where `f` is a global variable, assuming whatever assembly labels make sense, and where the **size of a pointer is 8-bytes**. Use `r0` for the variable `i`. No comments needed.

```
struct X {
    int* a;
    int  b[4];
    struct Y c;
}

struct X* f;
```

```
struct Y {
    int d;
    struct X* e;
}
```

a) Give assembly code for `i = f->c.e->a[1];` *(i.e., store the result in `r0`)*

```
ld $f, r0       # r0 = &f
ld (r0), r0     # r0 = f
ld 32(r0), r0   # r0 = f->c.e
ld (r0), r0     # r0 = f->c.e->a
ld 4(r0), r0    # r0 = i'= f->c.e->a[1]
```

b) Give assembly code for `f->b[1] = i;` *(i.e., just use `r0` for `i`)*

```
ld $f, r1       # r1 = &f
ld (r1), r1     # r1 = f
st r0, 12(r1)   # f->b[1] = i'
```

c) What is the minimum number of memory references required to compute the value of the following expression (not including fetching the instructions themselves from memory)?

```
f->c.e->c.e[0].a[0]
```

○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ● 5 ○ 6 ○ 7

```
int    i;
int a[5];
int   *b;
```

2a Give assembly code for the C statement: `i = a[2]`.

```
ld $2, r0                # r0 = 2
ld $a, r1                # r1 = &a
ld (r2, r0, 4), r1       # r1 = a[2]
ld $i, r2                # r2 = &i
st r1, (r2)              # i = a[2]
```

2b Give assembly code for the C statement: `b[0] = a[b[i]];`.

```
ld $i, r0                # r0 = &i
ld (r0), r0              # r0 = i
ld $b, r1                # r1 = &b
ld (r1), r1              # r1 = b
ld (r1, r0, 4), r2       # r2 = b[i]
ld $a, r3                # r3 = &a
ld (r3, r2, 4), r3       # r3 = a[b[i]]
st r3, (r1)              # b[0] = a[b[i]]
```

2c How many memory reads are required to execute the statement? Fill in a single multiple choice option below.

```
b[a[3]] = a[3] + a[i];
```

```
4: i, b, a[3], a[i]
```

0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8 ○

3 **[10 marks]** **Structs and Instance Variables.** Answer the following questions about the global variables `a` and `b`, declared as follows. Assume that `int`'s and pointers are **4-bytes** long.

```
struct A {
    char x;
    int y[3];
    struct B* z;
};

struct A* a;
struct B  b;
```

```
struct B {
    struct A j;
    int k;
};
```

Treat each sub-question separately (i.e., do not use values computed in prior sub-questions). Comments are not required, but they will probably help you.

3a Give assembly code for the C statement: `b.k = a->z[2].k;`.

```
ld $a, r0            # r0 = &a
ld (r0), r0          # r0 = a
ld 16(r0), r0        # r0 = a->z
ld 68(r0), r0        # r0 = a->z[2].k
ld $b, r1            # r1 = &b
st r0, 20(r1)        # b.k = a->z[2].k
```

3b How many memory reads are required to execute the statement? Fill in a single multiple choice option below.

```
b.k = a->z->j.z->j.y[2];
```

```
4: a, a->z, a->z->j.z, a->z->j.z->j.y[2]
```

0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8 ○

4 **[10 marks]** **Static Control Flow.** Answer the following questions on static control flow.

```
struct S {
    int       w;
    int       x;
    struct S* y;
    struct T  z;
};

struct T {
    int w;
    int x;
};
```

```
int       i;
struct S* a;

void foo() {
    struct S* b;
    // QUESTION CODE IS HERE
}
```

3a `i = a->y->x;`

```
ld   $i, r0          # r0 = &i
ld   $a, r1          # r0 = &a
ld   0(r1), r1       # r1 = a
ld   8(r1),r1        # r1=a->y
ld   4(r1), r2       # r2 = a->y>x
st   r2,(r0)         # i=a->y->x
```

3b `i = a->z.x;`

```
ld   $i, r0          # r0 = &i
ld   $a, r1          # r0 = &a
ld   0(r1),r1        # r1 = a
## new part
ld   16(r1), r2      # r1 = a->z.x
st   r2,(r0)         # i=a->z.x
```

3c `i = b->x;`

```
ld   $i, r0          # r0 = &i
## new part
ld   0(r5), r1       # r1 = b (from stack)
ld   4(r1),r2        # r2 = b->x
st   r2,(r0)         # i=b->x
```

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.

Once again use `r0` for the variable `i` (i.e., do not read or write memory for `i`) and use labels for static values. Give assembly code that is equivalent to each of the following C statements. Treat each question separately; i.e., do not use register values assigned in a previous answer.

3a `i = a->y`

```
ld $a, r1      # r1 = &a
ld (r1), r1    # r1 = a
ld 4(r1), r0   # i = s->y
```

3b `i = a->z->y;`

```
ld $a, r1      # r1 = &a
ld (r1), r1    # r1 = a
ld 8(r1), r2   # r2 = a->z
ld 4(r2), r0   # i = a->z->y
```

3c `a->z->z = a;`

```
ld $a, r1      # r1 = &a
ld (r1), r1    # r1 = a
ld 8(r1), r2   # r2 = a->z
st r1, 8(r2)   # a->z->z = a
```

4 **(6 marks)** **Static Control Flow.** Answer these questions using the register `r0` for `x` and `r1` for `y`.

4a Write commented assembly code equivalent to the following.

```
if (x <= 0)
    x = x + 1;
else
    x = x - 1;
```

```
      bgt r0, else  # goto else if x > 0
      inc r0        # x++ if a <= 0
      br    done
else: dec r0        # x-- if a > 0
done:
```

4b Write commented assembly code equivalent to the following.

```
for (x=0; x<y; x++)
   y--;
```

```
      ld $0, r0      # x = 0
loop: mov r1, r2     # r2 = y
      not r2
      inc r2         # r2 = -y
      add r0, r2     # r2 = x-y
      bgt r2, done   # goto done if x > y
      beq r2, done   # goto done if x == y
      dec r1         # y--
      inc r0         # x++
      br  loop       # goto loop
done:
```

5 **(6 marks)** **C Pointers.** Consider the following C procedure `copy()` and global variable `a`.

```
void copy (char* s, char* d, int n) {
    for (int i=0; i<n; i++)
        d[i] = s[i];
}

char a[9] = {1,2,3,4,5,6,7,8,9};
```

```
b - 2                          = a + 4 - 2
                               = a + 2
a + (b - a) + (&a[7] - &a[6]) = a + ((a+4) - a) + ((a+7) - (a+6))
                               = a + 4 + 1
                               = a + 5
```

So the call to `foo` simplifies to `foo(a+2, a+5, a+2)`. Thus when `foo()` runs we have:

```
*(a+2) = *(a+2) + *(a+5);
       = 2 + 5
       = 7
*(a+2) = *(a+2) + *(a+2)
       = 7 + 7
       = 14
```

Thus `foo()` returns 14.

3 **(6 marks)** **Global Arrays**. Consider the following C global variable declarations.

```
int  a[10];
int* b;
int  i;
```

Give the SM213 assembly code the compiler might generate for the following statements that access these variables. You may use labels `a`, `b`, and `c` for addresses. You may not assume anything about the value of registers. **Comment every line.**

3a `b = a;`

```
ld $a, r0    # r0 = &a
ld $b, r1    # r2 = &b
st r0, (r1)  # b = a
```

3b `a[i] = i;`

```
ld $i, r0           # r0 = &i
ld (r0), r1         # r1 = i
ld $a, r2           # r2 = &a = &a[0]
st r1, (r2, r1, 4)  # a[i] = i
```

4 **(3 marks)** **Instance Variables**. Consider the following C global variable declarations.

```
struct S {
    int   a;
    void* b;
    int   c;
};

struct S* s;
```

Give the SM213 assembly code the compiler might generate for the statement:

```
s->b = &s->c;
```

You may use the label `s`. You may not assume anything about the value of registers. **Comment every line.**

```
ld $s, r0     # r0 = &s
ld (r0), r1   # r1 = s = &s->a
ld $8, r2     # r2 = 8
add r1, r2    # r2 = &s1->c
st r2, 4(r1)  # s1->b = &s1->c
```

5 **(6 marks)** **Count Memory References.** Consider the following C global variable declarations.

```
struct S {
    int a[10];
};

struct S s;
```

```
struct T {
    int* x;
};

struct T* t;
```

For each question, count the number of memory **reads and writes** occur when the statement executes. Do not count the memory reads that fetch instructions. Justify your answer carefully by describing the reads and writes that occur.

5a `s.a[2] = s.a[3];`

1 read: `s.a[3]`; 1 write: `s.a[2]`

5b `t->x[2] = t->x[3];`

3 reads: `t`, `t->x`, `t->x[3]`; 1 write: `t->x[2]`

6 (8 marks) **Loops and If.** The following assembly code computes `s = a[0]` where `a` is a global, static array of integers. Modify this code so that it computes the sum of all positive elements of the array where the size of the array is stored in a global int named `n`. Your solution should avoid unnecessary memory accesses where possible (e.g., inside of the loop). You may modify the code in place. Comment every line you add. Hint: notice that you have to add four things: (1) read the value of `n`, (2) turn part of this code into a loop, (3) exit the loop at the right time, and (4) only sum positive numbers; you might want to take these one at a time.

```
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
```

Added lines are numbered

```
            ld $a, r0              # r0 = &a = &[0]
            ld $0, r1              # r1 = temp_i = 0
            ld $0, r2              # r2 = temp_s = 0
[1]         ld $n, r5              # r5 = &n
[2]         ld (r5), r5            # r5 = n = temp_n
[3]     loop:
[4]         bgt r5, cont           # continue if temp_n > 0
[5]         br done                # exit look if temp_n <= 0
[6]     cont:
            ld (r0, r1, 4), r3     # r3 = a[temp_i]
[7]         dec r5                 # temp_n --
[8]         inc r1                 # temp_i ++
[9]         bgt r3, add            # goto add if a[temp_i] > 0
[10]        br  loop               # skip add & goto loop if a[temp_i] <= 0
[11]    add:
            add r3, r2             # temp_s += a[temp_i] if a[temp_i] < 0
[12]        br loop                # start next iteration of loop
[13]    done:
            ld $s, r4              # r4 = &s
            st r2, (r4)            # s = temp_s
```

7 (7 marks) **Procedure Calls** Implement the following C code in assembly. Pass arguments on the stack. Assume that `r5` has already been initialized as the stack pointer and assume that some other procedure (not shown) calls `doit()`. You do not have to show the allocation of `x`; just use the label `x` to refer to its address. Comment every line.

```
  b - 2                          = a + 4 - 2
                                 = a + 2
  a + (b - a) + (&a[7] - &a[6]) = a + ((a+4) - a) + ((a+7) - (a+6))
                                 = a + 4 + 1
                                 = a + 5
```

So the call to `foo` simplifies to `foo(a+2, a+5, a+2)`. Thus when `foo()` runs we have:

```
  *(a+2) = *(a+2) + *(a+5);
         = 2 + 5
         = 7
  *(a+2) = *(a+2) + *(a+2)
         = 7 + 7
         = 14
```

Thus `foo()` returns 14.

3 **(6 marks)** **Global Arrays**. Consider the following C global variable declarations.

```
int  a[10];
int* b;
int  i;
```

Give the SM213 assembly code the compiler might generate for the following statements that access these variables. You may use labels `a`, `b`, and `c` for addresses. You may not assume anything about the value of registers. **Comment every line.**

3a `b = a;`

```
  ld $a, r0    # r0 = &a
  ld $b, r1    # r2 = &b
  st r0, (r1)  # b = a
```

3b `a[i] = i;`

```
  ld $i, r0           # r0 = &i
  ld (r0), r1         # r1 = i
  ld $a, r2           # r2 = &a = &a[0]
  st r1, (r2, r1, 4)  # a[i] = i
```

4 **(3 marks)** **Instance Variables**. Consider the following C global variable declarations.

```
struct S {
    int   a;
    void* b;
    int   c;
};

struct S* s;
```

Give the SM213 assembly code the compiler might generate for the statement:

```
s->b = &s->c;
```

You may use the label `s`. You may not assume anything about the value of registers. **Comment every line.**

```
  ld $s, r0     # r0 = &s
  ld (r0), r1   # r1 = s = &s->a
  ld $8, r2     # r2 = 8
  add r1, r2    # r2 = &s1->c
  st r2, 4(r1)  # s1->b = &s1->c
```

5 **(6 marks)** **Count Memory References.** Consider the following C global variable declarations.

```
struct S {
    int a[10];
};

struct S s;
```

```
struct T {
    int* x;
};

struct T* t;
```

For each question, count the number of memory **reads and writes** occur when the statement executes. Do not count the memory reads that fetch instructions. Justify your answer carefully by describing the reads and writes that occur.

5a `s.a[2] = s.a[3];`

1 read: `s.a[3]`; 1 write: `s.a[2]`

5b `t->x[2] = t->x[3];`

3 reads: `t`, `t->x`, `t->x[3]`; 1 write: `t->x[2]`

6 (8 marks) **Loops and If.** The following assembly code computes `s = a[0]` where `a` is a global, static array of integers. Modify this code so that it computes the sum of all positive elements of the array where the size of the array is stored in a global int named `n`. Your solution should avoid unnecessary memory accesses where possible (e.g., inside of the loop). You may modify the code in place. Comment every line you add. Hint: notice that you have to add four things: (1) read the value of `n`, (2) turn part of this code into a loop, (3) exit the loop at the right time, and (4) only sum positive numbers; you might want to take these one at a time.

```
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
```

Added lines are numbered

```
             ld $a, r0              # r0 = &a = &[0]
             ld $0, r1              # r1 = temp_i = 0
             ld $0, r2              # r2 = temp_s = 0
[1]          ld $n, r5              # r5 = &n
[2]          ld (r5), r5            # r5 = n = temp_n
[3]      loop:
             bgt r5, cont           # continue if temp_n > 0
[5]          br done                # exit look if temp_n <= 0
[6]      cont:
             ld (r0, r1, 4), r3     # r3 = a[temp_i]
[7]          dec r5                 # temp_n --
[8]          inc r1                 # temp_i ++
[9]          bgt r3, add            # goto add if a[temp_i] > 0
[10]         br  loop               # skip add & goto loop if a[temp_i] <= 0
[11]     add:
             add r3, r2             # temp_s += a[temp_i] if a[temp_i] < 0
[12]         br loop                # start next iteration of loop
[13]     done:
             ld $s, r4              # r4 = &s
             st r2, (r4)            # s = temp_s
```

7 (7 marks) **Procedure Calls** Implement the following C code in assembly. Pass arguments on the stack. Assume that `r5` has already been initialized as the stack pointer and assume that some other procedure (not shown) calls `doit()`. You do not have to show the allocation of `x`; just use the label `x` to refer to its address. Comment every line.

```
struct S {
    int i,j;
};
```

```
struct S  a;
struct S* b;
```

Indicate what, if anything, about the following expressions can not be computed statically. Be precise.

5a `&a.i`

Nothing.

5b `&b->i`

The value of the expression; i.e., the address of the variable.

5c `(&b->j)-(&b->i)`

Nothing.

6 (3 marks) **Memory Endianness** Examine this C code.

```
char a[4];
*((int*) (&a[0])) = 1;
```

Carefully explain how this code can be used to determine the *endianness* of the machine on which it runs, recalling that on a *Big Endian* machine, the high-order (most-significant) byte of a number has the lowest address.

The assignment statement stores the integer `0x00000001` in the array `a`. By testing which entry of `a` stores 1 and which store 0, the program can determine whether the least significant byte (i.e., 1) has the lowest or highest address. Thus if `a[0]==1` this is a Little Endian machine and if `a[3]==1` it is Big Endian.

7 (8 marks) Consider the following C declarations.

```
struct S {
  int       i;
  int       j[10];
  struct S* k;
};
```

```
int       a[10];
int*      b;
int       c;
struct S  d;
struct S* e;
```

For each of the following questions indicate the total number of memory references (i.e., distinct loads and/or stores) required to execute the listed statement. Justify your answers carefully.

7a `a[3] = 0;`

One: store value in variable.

7b `a[c] = 0;`

Two: read value of c; store value in variable.

7c `b[c] = 0;`

Three: read value of b; read value of c; store value in variable

7d `d.i = 0;`

One: store value in variable.

7e `d.j[3] = 0;`

One: store value in variable.

7f `e->i = 0;`

Two: read value of e; store value in variable.

7g `d.k->i = 0;`

Two: read value of `d.k`; store value in variable.

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.

Give the SM213 assembly code the compiler might generate for the following statements that access these variables. Assume the value `0` is in `r0` and the value of `i` is in `r1`. Use labels `a` and `b` for variable addresses.

3a `a[i] = 0;`

```
  ld $a, r2             # r2 = &a
  st r0, (r2, r1, 4)    # a[i] = 0
```

3b `b[i] = 0;`

```
  ld $b, r2             # r2 = &b
  ld (r2), r2           # r2 = b
  st r0, (r2, r1, 4)    # b[i] = 0
```

4 **(4 marks)** **Instance Variables**. Consider the following C global variable declarations.

```
struct S {
    int  a;
    int* b;
    int  c;
};

struct S  s0;
struct S* s1;
```

Give the SM213 assembly code the compiler might generate for the following statements that access these variables. Assume the value `0` is in `r0`. Use labels `s0` and `s1` for variable addresses.

4a `s0.c = 0;`

```
  ld $s0, r2      # r2 = &s0
  st r0, 8(r2)    # s0.c = 0
```

4b `s1->c = 0;`

```
  ld $s1, r2      # r2 = &s1
  ld (r2), r2     # r2 = s1
  st r0, 8(r2)    # s1->c = 0
```

5 **(8 marks)** **Count Memory References.** Consider the following C global variable declarations.

```
struct S {
    struct S* s;
    int*      a;
    int       b[10];
};

struct S s0;
```

And this statement found in some procedure.

```
int v = s0.s->a[s0.s->b[2]];
```

List every memory read that will occur when this statement executes in the SM213 machine. List only memory reads and list each read separately. For each read give the name of the variable being read and an expression that computes the target memory address. This expression may only contain the label `s0`, numbers, the name of any variable previously read and arithmetic operators such as for addition and multiplication. Numbers must be immediately followed by a description in parentheses (e.g., "...4 (size of int)"). There are more lines listed below than you need:

1. Variable: `s0.s`
   Address: `s0 + 0 (offset to s)`
2. Variable: `s0.s->b[2]`
   Address: `s0.s + 8 (offset to b) + 2 (index) * 4 (size of int)`
3. Variable: `s0.s->a`

Address: `s0.s + 4 (offset to a)`

4. Variable: `s0.s->a[s0.s->b[2]]`

   Address: `s0.s->a + s0.s->b[2] * 4 (size of int)`

6 **(4 marks)** **Branch and Jump Instructions**.

6a What is one important benefit that *PC-relative* branches have over *absolute-address* jumps.

smaller instructions

6b What is the value of the program-counter register (i.e., the PC) following the execution of this unconditional branch instruction at address `0x500`. Justify your answer.

```
0x500: 8005
```

`0x502 + 5 * 2 == 0x50c`

7 **(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.

If-then statements whose test condition is a dynamic value and loops that execute a bounded and dynamically determined number of times.

8 **(4 marks)** **Dynamic Allocation**. The following code contains a procedure that creates a copy of a null-terminated string (the standard representation for strings in C). It contains a serious bug related to dynamic memory allocation.

```
char* copy (char* s) {
  int   i, len = 0;
  char* cpy;

  while (s [len] != 0)
    len++;
  cpy = (char*) malloc (len+1);
  for (i=0, i<len; i++)
    cpy [i] = s [i];
  cpy [len] = 0;
  return cpy;
}
```

```
void doSomething () {
  char* x;
  x = copy ("Hello World");
  printf ("%s", x);
}
```

Explain in plan English what the bug is and how you would fix it (without changing the semantics of `copy`).

2 **(8 marks)** **Global Variables.** Consider the following C declaration of global variables.

```
int a;
int *b;
int c[10];
```

Recalling that `&` is the C *get address* operator, which of the following can be computed statically? Justify your answers.

2a `&b`

Statically known. The compiler decides the address of global variables.

2b `&b[4]`

Not statically known. The compiler doesn't know the address of an element in a dynamic array.

`&b[4] = &*(b + 4) = b + 4`. The compiler doesn't know the value of `b`

2c `&c[4]`

Statically known. The compiler chooses where a globally declared array is located and thus knows the address of any of its elements.

Now answer this question.

2f Give the assembly code the compiler would generate for "`c[a] = *b;`". Use labels for static values.

```
ld $c, r0        # r0 = &c[0]
ld $a, r1        # r1 = &a
ld 0(r1), r1     # r1 = a
ld $b, r2        # r2 = &b
ld 0(r2), r2     # r2 = b
ld 0(r2), r2     # r2 = *b
st r2, (r0, r1, 4) # c[a] = *b

# static data area (not required in answer)
.pos 0x1000
a: .long 0
b: .long 0
c: .long 0 # c[0]
   .long 0 # c[1]
   .long 0 # c[2]
   .long 0 # c[3]
   .long 0 # c[4]
   .long 0 # c[5]
   .long 0 # c[6]
   .long 0 # c[7]
   .long 0 # c[8]
   .long 0 # c[9]
```

3 **(8 marks)** **Instance Variables and Local Variables.** Consider the following C declaration of global variables.

```
struct X {
    int a;
    int b;
};
struct X c;
struct X* d;
```

Which of the following can be computed statically? Justify your answers.

3a `&c.a`

Address of member of a statically allocated is statically known.

3b `&d->a`

Not statically known. Need to get the value of the pointer which is only known at runtime to compute that address.

3c `(&d->a) - (&d->b)`

The difference in offset between two struct members is always statically known regardless of where the struct is stored.

Now answer this question.

3d Give the assembly code the compiler would generate for "`d->b = c.b;`".

```
ld $d, r0        # r0 = &d
ld 0(r0), r0     # r0 = d
ld $c, r1        # r1 = &c
ld 4(r1), r1     # r1 = c.b
st r1, 4(r0)     # d->b = c.b

# data areas (not required in answer)
# static data area
.pos 0x1000
c: .long 0       # c.a
   .long 0       # c.b
d: .long 0x2000

# heap
.pos 0x2000
d_data: .long 0 # d->a
        .long 0 # d->b
```

- Garbage collection is a partial solution to the dangling pointer problem. False
- Garbage collection is a partial solution to the memory leak problem. True
- Garbage collection is a full solution to the dangling pointer problem. True
- Garbage collection is a full solution to the memory leak problem. False
- The stack is a partial solution to the dangling pointer problem. False
- Having memory allocation and deallocation in separate functions is a partial solution to the dangling pointer problem. False

4 **(12 marks)** **Global Arrays and Writing Assembly Code.** In the context of the following C declarations:

```
int *b;
int a[10];
int i = 3;
a[i] = 2;
b = &a[5];
b[i] = 4;
```

Provide the SM213 assembly code for the C code above, with comments. Be as concise as possible. You may assume labels `$i`, `$a`, `$b` have been created pointing to appropriate memory locations for storage, so there is no need to write any assembly for the first two lines of C code.

```
      ld $i, r0            # r0 = &i
      ld $0x3, r1          # r1 = 3
      st r1, 0x0(r0)       # i = 3
      ld $a, r2            # r2 = &a
      ld $0x2, r3          # r3 = 2
      st r3, (r2, r1, 4)   # a[i] = 2
      ld $b, r4            # r4 = &b
      ld $20, r6           # r6 = 20 (5*4)
      add r2, r6       # r6 = &a[5]
      st r6, 0x0(r4)       # b = &a[5]
      ld $0x4, r7          # r7 = 4
      st r7, (r6, r1, 4)   # b[i] = 4
```

(12 marks, one per line. 1/2 mark off for verbose/unnecessary instructions. 1/2 mark off for manual add instead of using an indexed instruction. No penalty for leaving off the 0 offset when using base/displacement load or store.)

5 **(5 marks)** **Instance Variables.** In the context of the following C declarations:

```
struct S {
    int i[3];
    int j[4];
    int k;
};
struct S  a;
struct S* b;
```

5a Indicate which of the following the compiler knows statically and which is determined dynamically.

- `&(a.i[2])` static
- `&(b->j[2])` dynamic
- `&(b->k)` dynamic
- `(&(b->j[1]) - &(b->j[2]))` static

5b Give SM213 assembly code that reads the value of `b->k` into `r0`. Comment your code.

```
  ld   $b, r0              # r0 = &b (pointer to struct)
  ld   0x0(r0), r0         # r0 = b (address of struct itself)
  ld   0x1c(r0), r0        # r0 = b->k (content of address of field in struct)
```

3 marks. Full credit also given for alternate answer using indexed load. Subtle point: while the offset has to fit into one hex digit of machine language, what's actually stored is in the machine language instruction is the number from the assembly language instruction divided by 4. Hex 1c is decimal 28. $28/4 = 7$, and 0x7 will indeed fit into that digit. But full credit also given for slightly more verbose solution:

```
  ld  $b, r0       # r0 = &b (pointer to struct)
  ld  0x0(r0), r0  # r0 = b (address of struct itself)
  ld $0x1c, r1     # r1 = decimal 28 (7 integers * 4 bytes each)
  add r1, r0       # r0 = address of struct field b->k
  ld 0x0(r0), r0   # r0 = value of b->k field
```

6 **(12 marks)** **Procedures and Writing Assembly Code.** Consider the following procedure in SM213 assembly code. Assume that arguments have been passed in on the stack, not in registers.

```
int sum (int *a, int i, int* b, int j) {
    return a[i] + b[j];
}
```

6a Why would we use the stack to store arguments instead of just using registers? Explain carefully.

There are a limited number of registers, so procedures with many arguments would not be able to store all arguments on the stack. Also, the convention in SM213 is to use specific registers to store information like the return address and the stack pointer, so the number of registers available for free use inside a procedure is even more limited. Using the stack in the first place avoids the need to save registers to the stack in subsequent procedure calls.

2 marks. The answer "the stack is better than registers because it's too hard to keep track of which register is used for which parameter" is not correct: the compiler can indeed figure that information out at compile time. The answer "the stack is more efficient than using registers" is not correct: the stack is slower since it requires memory accesses. Most answers involving dangling pointers or polymorphism were incorrect.

6b What is known statically vs dynamically?

- The offset between the stack pointer for sum and the address of a? static
- The offset between the stack pointer for sum and the address of i? static
- The address of a? dynamic
- The address of i? dynamic

6c Implement the C procedure above in SM213 assembly code. Assume that arguments have been passed in on the stack, not in registers. Comment your code.

```
struct S {
    int i,j;
};
```

```
struct S  a;
struct S* b;
```

Indicate what, if anything, about the following expressions can not be computed statically. Be precise.

5a `&a.i`

Nothing.

5b `&b->i`

The value of the expression; i.e., the address of the variable.

5c `(&b->j)-(&b->i)`

Nothing.

6 (3 marks) **Memory Endianness** Examine this C code.

```
char a[4];
*((int*) (&a[0])) = 1;
```

Carefully explain how this code can be used to determine the *endianness* of the machine on which it runs, recalling that on a *Big Endian* machine, the high-order (most-significant) byte of a number has the lowest address.

The assignment statement stores the integer `0x00000001` in the array `a`. By testing which entry of `a` stores 1 and which store 0, the program can determine whether the least significant byte (i.e., 1) has the lowest or highest address. Thus if `a[0]==1` this is a Little Endian machine and if `a[3]==1` it is Big Endian.

7 (8 marks) Consider the following C declarations.

```
struct S {
  int       i;
  int       j[10];
  struct S* k;
};
```

```
int       a[10];
int*      b;
int       c;
struct S  d;
struct S* e;
```

For each of the following questions indicate the total number of memory references (i.e., distinct loads and/or stores) required to execute the listed statement. Justify your answers carefully.

7a `a[3] = 0;`

One: store value in variable.

7b `a[c] = 0;`

Two: read value of c; store value in variable.

7c `b[c] = 0;`

Three: read value of b; read value of c; store value in variable

7d `d.i = 0;`

One: store value in variable.

7e `d.j[3] = 0;`

One: store value in variable.

7f `e->i = 0;`

Two: read value of e; store value in variable.

7g `d.k->i = 0;`

Two: read value of `d.k`; store value in variable.