4. Static Control Flow [7 marks]

Assuming `x`, `y` and `z` are global variables that have already been declared and initialized. No comments needed.

a) Give assembly code for the loop (you must implement a loop)

```
for (int x=0; x<y; x++) {
    z = z + 1;
}
```

b) Give assembly code for the procedure call (ignore anything having to do with the stack).

```
foo();
```

c) Give assembly code for the return statement (ignore anything having to do with the stack).

```
return;
```

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

4a Give assembly code for the following. Assume `x` and `y` are global integer variables.

```
if (x > 0 && x < 10) {
    y = x;
} else {
    y = 0;
}
```

4b Consider a procedure with the following signature: `int foo(int a, int b);`

Give assembly code for the following.

```
x = foo(x, y);
```

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;
```

4b Write commented assembly code equivalent to the following.

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

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};
```

And this procedure call:

```
copy (a, a+3, 6);
```

List the value of the elements of the array `a` (in order), following the execution of this procedure call.

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
```

8 **(3 marks)** **Programming in C.** Consider the following C code.

```
int* b;

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

There is a dangerous bug in this code. Carefully describe what it is. Assume that `b` was assigned a value somewhere else in the program.

9 **(3 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();
}
```

There is a dangerous bug in this code. Carefully describe what it is.

Hint: what is the value of `*ret` just before and just after `two()` is called? Look carefully at the implementation of `one()`, what it returns, and when variables are allocated and deallocated.

10 **(4 marks)** **Branch and Jump Instructions**.

10a What is one important benefit that *PC-relative* branches have over *absolute-address* jumps.

10b 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
```

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() {
    for (i=0; i<size; i++)
        if (a[i]==searchFor)
            a[i]=replaceWith;
}
```

```
void callReplace() {
    replace();
    // halt; do not return
}
```

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.

CPSC 213, Winter 2015, Term 2 — Extra Questions

Date: March 3, 2015; Instructor: Mike Feeley

Q1	/ 8
Q2	/ 6
Q3	/ 8
Q4	/ 8
Q5	/ 12
Q6	/ 10
Q7	/ 10

Answer in the space provided. Show your work; use the backs of pages if needed. There are **7** questions on **5** pages, totaling **62** marks.

STUDENT NUMBER: ____________________

NAME: ____

SIGNATURE: ____

1 **(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.

2 **(6 marks)** **Static Control Flow.** Give SM213 assembly code for the following C statements. Assume that `i` is a global variable of type `int`.

2a
```
if (i==0)
    i = 1;
else
    i = 2;
```

2b
```
while (i!=0)
    i -= 1;
```

```
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;
}
```

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

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

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
```

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.

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`).

9 **(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() {
    for (i=0; i<size; i++)
        if (a[i]==searchFor)
            a[i]=replaceWith;
}
```

```
void callReplace() {
    replace();
    // halt; do not return
}
```

7 **(4 marks)** **Static Control Flow.** Consider the following procedure in C.

```
if (a > 2) b = 3;
else if (a < -4) b = 5;
```

In SM213 assembly, how many branch/jump statements are needed to implement this code? How many of these are conditional and how many are unconditional? Justify your answer.

You may remove this page. These two tables describe the SM213 ISA. The first gives a template for instruction machine and assembly language and describes instruction semantics. It uses 's' and 'd' to refer to source and destination register numbers and 'p' and 'i' to refer to compressed-offset and index values. Each character of the machine template corresponds to a 4-bit, hexit. Offsets in assembly use 'o' while machine code stores this as 'p' such that 'o' is either 2 or 4 times 'p' as indicated in the semantics column. The second table gives an example of each instruction.

Operation	Machine Language	Semantics / RTL	Assembly
load immediate	`0d-- vvvvvvvv`	$r[d] \leftarrow vvvvvvvv$	`ld $vvvvvvvv,rd`
load base+offset	`1psd`	$r[d] \leftarrow m[(o = p \times 4) + r[s]]$	`ld o(rs),rd`
load indexed	`2bid`	$r[d] \leftarrow m[r[b] + r[i] \times 4]$	`ld (rb,ri,4),rd`
store base+offset	`3spd`	$m[(o = p \times 4) + r[d]] \leftarrow r[s]$	`st rs,o(rd)`
store indexed	`4sdi`	$m[r[b] + r[i] \times 4] \leftarrow r[s]$	`st rs,(rb,ri,4)`
halt	`F000`	(stop execution)	`halt`
nop	`FF00`	(do nothing)	`nop`
rr move	`60sd`	$r[d] \leftarrow r[s]$	`mov rs, rd`
add	`61sd`	$r[d] \leftarrow r[d] + r[s]$	`add rs, rd`
and	`62sd`	$r[d] \leftarrow r[d] \;\&\; r[s]$	`and rs, rd`
inc	`63-d`	$r[d] \leftarrow r[d] + 1$	`inc rd`
inc addr	`64-d`	$r[d] \leftarrow r[d] + 4$	`inca rd`
dec	`65-d`	$r[d] \leftarrow r[d] - 1$	`dec rd`
dec addr	`66-d`	$r[d] \leftarrow r[d] - 4$	`deca rd`
not	`67-d`	$r[d] \leftarrow !r[d]$	`not rd`
shift	`7dss`	$r[d] \leftarrow r[d] << ss$	`shl ss, rd`
		(if ss is negative)	`shr -ss, rd`
branch	`8-pp`	$pc \leftarrow (aaaaaaaa = pc + pp \times 2)$	`br aaaaaaaa`
branch if equal	`9rpp`	if $r[r] == 0: \; pc \leftarrow (aaaaaaaa = pc + pp \times 2)$	`beq rr, aaaaaaaa`
branch if greater	`Arpp`	if $r[r] > 0: \; pc \leftarrow (aaaaaaaa = pc + pp \times 2)$	`bgt rr, aaaaaaaa`
jump	`B--- aaaaaaaa`	pc $\leftarrow aaaaaaaa$	`j aaaaaaaa`
get program counter	`6Fpd`	$r[d] \leftarrow$ pc $+ (o = 2 \times p)$	`gpc $o, rd`
jump indirect	`Cdpp`	$pc \leftarrow r[d] + (o = 2 \times pp)$	`j o(rd)`
jump double ind, b+off	`Cdpp`	$pc \leftarrow m[(o = 4 \times pp) + r[d]]$	`j *o(rd)`
jump double ind, index	`Edi-`	$pc \leftarrow m[4 \times r[i] + r[d]]$	`j *(rd,ri,4)`

Operation	Machine Language Example	Assembly Language Example
load immediate	`0100 00001000`	`ld $0x1000,r1`
load base+offset	`1123`	`ld 4(r2),r3`
load indexed	`2123`	`ld (r1,r2,4),r3`
store base+offset	`3123`	`st r1,8(r3)`
store indexed	`4123`	`st r1,(r2,r3,4)`
halt	`f000`	`halt`
nop	`ff00`	`nop`
rr move	`6012`	`mov r1, r2`
add	`6112`	`add r1, r2`
and	`6212`	`and r1, r2`
inc	`6301`	`inc r1`
inc addr	`6401`	`inca r1`
dec	`6501`	`dec r1`
dec addr	`6601`	`deca r1`
not	`6701`	`not r1`
shift	`7102`	`shl $2, r1`
	`71fe`	`shr $2, r1`
branch	`1000: 8003`	`br 0x1008`
branch if equal	`1000: 9103`	`beq r1, 0x1008`
branch if greater	`1000: a103`	`bgt r1, 0x1008`
jump	`b000 00001000`	`j 0x1000`
get program counter	`6f31`	`gpc $6, r1`
jump indirect	`c104`	`j 8(r1)`
jump double ind, b+off	`d102`	`j *8(r1)`
jump double ind, index	`e120`	`j *(r1,r2,4)`

8 **(3 marks)** **Programming in C.** Consider the following C code.

```
int* b;

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

There is a dangerous bug in this code. Carefully describe what it is. Assume that `b` was assigned a value somewhere else in the program.

9 **(3 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();
}
```

There is a dangerous bug in this code. Carefully describe what it is.

Hint: what is the value of `*ret` just before and just after `two()` is called? Look carefully at the implementation of `one()`, what it returns, and when variables are allocated and deallocated.

10 **(4 marks)** **Branch and Jump Instructions**.

10a What is one important benefit that *PC-relative* branches have over *absolute-address* jumps.

10b 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
```

4. Static Control Flow [7 marks]

Assuming `x`, `y` and `z` are global variables that have already been declared and initialized. No comments needed.

a) Give assembly code for the loop (you must implement a loop)

```
for (int x=0; x<y; x++) {
    z = z + 1;
}
```

```
            ld $0, r0       # r0 = x' = 0
            ld $y, r1       # r1 = &y
            ld (r1), r1     # r1 = y
            not r1          # r1 = ~y
            inc r1          # r1 = -y
            ld $z, r2 # r2 = &z
            ld (r2), r2     # r2 = z
loop:       mov r1, r3      # r2 = -y
            add r0, r3      # r3 = x' - y
            bgt r3, done    # goto done if x' > y
            beq r3, done    # goto done if x' >= y
            # cont if x' < y
            inc r2          # z' = z' + 1
            inc r0          # x' = x' + 1
            br loop         # goto top of loop
done:       ld $z, r1       # r1 = &z
            st r2, (r1)     # z = z'

            # last two lines optional (since x in declared in for statement)
            ld $x r1        # r1 = &x
            st r0, (r1)     # x = x'
```

b) Give assembly code for the procedure call (ignore anything having to do with the stack).

```
foo();
```

```
gpc $6, r6
j foo
```

c) Give assembly code for the return statement (ignore anything having to do with the stack).

```
return;
```

```
j (r6)
```

4a Give assembly code for the following. Assume x and y are global integer variables.

```
if (x > 0 && x < 10) {
    y = x;
} else {
    y = 0;
}
```

```
            ld $x, r0       # r0 = &x
            ld (r0), r0     # r0 = x
            bgt r0, L0      # if (x>0) goto L0
            br L2           # else goto L2
L0:         ld $10, r1      # r1 = 10
            mov r0, r2      # r2 = x
            not r2          #
            inc r2          # r2 = -x
            add r2, r1      # r1 = 10-x
            bgt r1, L1      # if (x>10) goto L1
L2:         ld $0, r0       # r0 = 0
L1:         ld $y, r1       # r1 = &y
            st r0, (r1)     # y = r0 (0 or x)
```

4b Consider a procedure with the following signature: `int foo(int a, int b);`

Give assembly code for the following.

```
x = foo(x, y);
```

```
    ld   $x, r1      # r1 = &x
    ld   (r1), r0    # r0 = x
    ld   $y, r2      # r2 = &y
    ld   (r2), r2    # r2 = y
    deca r5
    deca r5          # sp -= 8
    st   r1, (r5)    # arg0 = x
    st   r2, 4(r5)   # arg1 = y
    gpc  $6, r6      # r6 = ra
    j    foo         # foo(x, y)
    inca r5
    inca r5          # sp += 8
    ld   $x, r1      # r1 = &x
    st   r0, (r1)    # x = foo(x,y)
```

5 **[10 marks]** **C Pointers.** Consider the following global variable declarations.

```
int a[4] = (int[4]){5, 4, 3, 2};
int b[3] = (int[3]){6, 7, 8};
int* c;
```

Assume that the address of a is 0x1000, the address of b is 0x2000, and the address of c is 0x3000. Now, consider the the execution of the following additional code.

```
c = &a[3];
b[*c] = a[1];
c = (b+1);
*c = *c + 1;
*c = *c + 1;
a[3] = c[1];
```

Indicate the value of each of the following expressions (i.e., the values of a, b, and c) below by checking *Unchanged* if the execution does not change the value or by entering its new value in the space provided. Assuming that int's are 4-bytes long. Answer these questions about the value of these variables following the execution of this code.

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};
```

Yes; there's a dangling pointer. The procedure `one()` returns a pointer to a local variable, but that local variable is deallocated when the procedure returns. Just before `three()` calls `two()` the value of `*ret` is 1, but after calling `two()` it changes to `2` because `two()`'s local variable `zot` will be allocated in the same location as `one()`'s `loc`, and `*ret` is a dangling pointer pointing to that location.

10 (4 marks) **Branch and Jump Instructions**.

10a What is one important benefit that *PC-relative* branches have over *absolute-address* jumps.

smaller instructions

10b 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
```

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.

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

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

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

Static. The compiler knows the address of every procedure.

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

Dynamic. A procedure can be called from multiple statements and each of these will have different return addresses. The same return statement must thus be able to jump to many different addresses, depending on which statement called it.

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() {
    for (i=0; i<size; i++)
        if (a[i]==searchFor)
            a[i]=replaceWith;
}
```

```
void callReplace() {
    replace();
    // halt; do not return
}
```

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

```
untZero: 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
op:      bgt r1, cont          # goto cont if len>0
         br  done              # goto done if len<=0
nt:      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
ne:      j   (r6)              # return c
```

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`).

The `copy()` procedure allocates memory that is never freed. The simplest fix is to insert a `free(x)` statement as the last line of `doSomething()`; you get full marks for this answer. A better fix is to move the allocation to `doSomething()` and have it pass the target string to `copy()` as a parameter instead of having `copy()` return it; one bonus mark is available for a good explanation of the better solution.

You didn't have to give the code, but here is the code for the better solution.

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

  while (s [len] != 0 && len+1 < n)
    len++;
  for (i=0, i<len; i++)
    cpy [i] = s [i];
  cpy [len] = 0;
  return cpy;
}

void doSomething () {
  char x[1000];
  copy (x, "Hello World", sizeof (x));
  printf ("%s", x);
}
```

9 **(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() {
    for (i=0; i<size; i++)
        if (a[i]==searchFor)
            a[i]=replaceWith;
}

void callReplace() {
    replace();
    // halt; do not return
}
```

```
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
```

```
sum:    ld   0(r5), r1        # r1 = a
        ld   4(r5), r2        # r2 = i
        ld   8(r5), r3        # r3 = b
        ld   12(r5), r4       # r4 = j
        ld   (r1,r2,4), r0    # r0 = a[i]
        ld   (r3,r4,4), r1    # r1 = b[j]
        add  r1, r0           # r0 = a[i] + b[j]
        j    (r6)             # return
```

8 marks.

- +1 for using parameters on the stack that were placed by the caller, as opposed to pushing parameters explicitly in this code or using registers.
- +1 for having the correct offsets for stack arguments
- +1 for leaving the teardown to the caller (not doing it explicitly here)
- +1 for correct calculation for a/b
- +1 for correct calculation for i/j
- +1 for add
- +1 for having return value in r0
- +1 for jump to r6 value
- -1/2 for moving result to r0 instead of computing it there in the first place (verbose)
- -1/2 for wrong order of arguments
- -1/2 for changing the stack pointer instead of using offsets from it
- -1/2 for jump to r6 instead of (r6)

7 **(4 marks)** **Static Control Flow.** Consider the following procedure in C.

```
if (a > 2) b = 3;
else if (a < -4) b = 5;
```

In SM213 assembly, how many branch/jump statements are needed to implement this code? How many of these are conditional and how many are unconditional? Justify your answer.

Three total, two conditional and one unconditional. One conditional is needed for each of the two tests in the C code. One unconditional is needed to skip past the second code block. A second unconditional is not needed at the end of that block, because control will simply flow through to the next line.

4 marks: 1 for correct answer of 3 total, 1 for its justification. 1 for correct answer of 2 cond + 1 uncond, 1 for its justification.

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.

```
sum:    ld   0(r5), r1       # r1 = a
        ld   4(r5), r2       # r2 = i
        ld   8(r5), r3       # r3 = b
        ld   12(r5), r4      # r4 = j
        ld   (r1,r2,4), r0   # r0 = a[i]
        ld   (r3,r4,4), r1   # r1 = b[j]
        add  r1, r0          # r0 = a[i] + b[j]
        j    (r6)            # return
```

8 marks.

- +1 for using parameters on the stack that were placed by the caller, as opposed to pushing parameters explicitly in this code or using registers.
- +1 for having the correct offsets for stack arguments
- +1 for leaving the teardown to the caller (not doing it explicitly here)
- +1 for correct calculation for a/b
- +1 for correct calculation for i/j
- +1 for add
- +1 for having return value in r0
- +1 for jump to r6 value
- -1/2 for moving result to r0 instead of computing it there in the first place (verbose)
- -1/2 for wrong order of arguments
- -1/2 for changing the stack pointer instead of using offsets from it
- -1/2 for jump to r6 instead of (r6)

7 **(4 marks)** **Static Control Flow.** Consider the following procedure in C.

```
if (a > 2) b = 3;
else if (a < -4) b = 5;
```

In SM213 assembly, how many branch/jump statements are needed to implement this code? How many of these are conditional and how many are unconditional? Justify your answer.

Three total, two conditional and one unconditional. One conditional is needed for each of the two tests in the C code. One unconditional is needed to skip past the second code block. A second unconditional is not needed at the end of that block, because control will simply flow through to the next line.

4 marks: 1 for correct answer of 3 total, 1 for its justification. 1 for correct answer of 2 cond + 1 uncond, 1 for its justification.

CPSC 213, Winter 2015, Term 2 — Extra Questions **Solution**

Date: March 3, 2015; Instructor: Mike Feeley

1 **(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.

2 **(6 marks)** **Static Control Flow.** Give SM213 assembly code for the following C statements. Assume that `i` is a global variable of type `int`.

2a

```
if (i==0)
    i = 1;
else
    i = 2;
```

```
    ld $i, r0      # r0 = &i
    ld (r0), r1    # r1 = i
    beq r1, L0     # goto L0 if i==0
    ld $2, r2      # t_i = 2 if i !=0
    br L1          # goto L1
L0: ld $1, r2      # t_i = 1 if i==0
L1: st r2, (r0)    # i = t_i
```

2b

```
while (i!=0)
    i -= 1;
```

```
    ld $i, r0      # r0 = &i
    ld (r0), r1    # t_i = i
L0: beq r1, L2     # goto L1 if t_i == 0
    dec r1         # t_i--
    br L0          # goto L0
L1: st r1, (r0)    # i = t_i
```

3 **(8 marks)** **Dynamic Control Flow.** Give SM213 assembly code for the following C statements. Assume that `i` is a global variable of type `int`.

3a Using a jump table, the statement:

```
switch (i) {
    case 4:
        i = 0;
        break;
    case 6:
        i = 1;
        break;
    default:
        i = 2;
        break;
}
```

```
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.