CPSC 213, Winter 2015, Term 2 — Midterm Exam

Date: February 29, 2016; Instructor: Mike Feeley

Q1	/ 8
Q2	/ 6
Q3	/ 6
Q4	/ 6
Q5	/ 6
Q6	/ 6
Q7	/ 8
Q8	/ 8
Q9	/ 10

This is a closed book exam. No notes. No electronic calculators. Note: this is a normal 1-hour-length midterm, but you have 2 hours to write it.

Answer in the space provided. Show your work; use the backs of pages if needed. There are **9** questions on **8** pages, totaling **64** marks. You have **2 hours** to complete the exam.

STUDENT NUMBER: ____________________

NAME: ____

SIGNATURE: ____

1 **(8 marks)** **Memory and Numbers.** Consider the following C code containing global variables `a`, `b`, and `c` that is executing on a *little endian*, 32-bit processor. Assume that the address of `a[0]` is `0x1000` and that the compiler allocates global variables in the order they appear in the program without *unnecessarily* wasting space between them. With this information, you can determine the value of certain bytes of memory following the execution of `foo()`.

```
char a[2];
int  b[2];
int* c;
```

```
void foo() {
    a[1] = 0x10;
    b[1] = 0x30405060;
    c    = b;
}
```

List the address and value of every memory location whose address and value you know. Use the form “`address: value`”. **List every byte on a separate line** and list all numbers in hex.

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

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 **(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 **(6 marks)** **Dynamic Allocation.** The following four pieces of code are identical except for the their use of `free()`. Each of them may be correct or they may have a memory leak, dangling pointer or both. In each case, determine whether these bugs exists and if so, briefly describe the bug(s); do not describe how to fix the bug.

6a

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  return dst;
}
```

```
int foo() {
  int  a = 3;
  int* b = copy (&a);
  return *b;
}
```

6b

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  free (dst);
  return dst;
}
```

```
int foo() {
  int  a = 3;
  int* b = copy (&a);
  return *b;
}
```

6c

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  return dst;
}
```

```
int foo() {
  int  a = 3;
  int* b = copy (&a);
  free (b);
  return *b;
}
```

6d

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  free (dst);
  return dst;
}
```

```
int foo() {
  int  a = 3;
  int* b = copy (&a);
  free (b);
  return *b;
}
```

7 **(8 marks)** **Reference Counting.** The following extends the code from the previous question by adding a procedure `saveIfMax` that is implemented in a separate module. Add calls to `inc_ref` and `dec_ref` to use referencing counting to eliminate all dangling pointers and memory leaks in this code while creating no *coupling* between `saveIfMax` and the rest of the code (i.e., `saveIfMax` can not know about what the rest of the code does and neither can the rest of the code know what `saveIfMax` does). Do not implement reference counting nor worry about storing the reference count itself; just add calls to `inc_ref` and `dec_ref` in the right places, **which may require slightly rewriting portions of the code**.

```
int* copy (int* src) {

  int* dst = malloc (sizeof (int));

  *dst = *src;

  return dst;

}


int foo() {

  int  a = 3;

  int* b = copy (&a);

  saveIfMax (b);

  return *b;

}
```

```
int* max;

void saveIfMax (int* x) {

  if (max==NULL || *x > *max) {

    max = x;

  }

}
```

8 **(8 marks)** **Procedures and the Stack.** Answer the following questions about this assembly code.

```
[01]        ld   $-12, r0
[02]        add  r0, r5
[03]        ld   $2, r0
[04]        st   r0, 0(r5)
[05]        st   r0, 4(r5)
[06]        st   r0, 8(r5)
[07]        gpc  $6, r6
[08]        j    foo
[09]        ld   $12, r1
[10]        add  r1, r5
[11]        ld   $0x1000, r1
[12]        st   r0, (r1)
[13]        halt

[14]  foo:  ld   $-8, r0
[15]        add  r0, r5
[16]        st   r6, 4(r5)
[17]        ld   8(r5), r0
[18]        ld   12(r5), r1
[19]        ld   16(r6), r2
[20]        add  r1, r0
[21]        add  r2, r0
[22]        ld   $8, r1
[23]        add  r1, r5
[24]        j (r6)
```

8a How many arguments, if any, does `foo()` have?

8b How many local variables, if any, does `foo()` have (count them even if they are not used)?

8c Is `foo()`'s return address saved on the stack at any point. If so, which line saves it?

8d If you can determine the integer value in memory at address `0x1000` following the execution of this code, give its value.

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:

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)`