CPSC 213, Winter 2013, Term 2 — Sample Questions for Quiz 1

Date: January 24, 2014; Instructor: Mike Feeley

NAME: _____ **STUDENT NUMBER:** _____

1 What is the smallest number greater than or equal to `0x1246` that is aligned for access to 4-byte integers?

0x1248

2 Answer the following questions about the two statements below that read an element of different arrays in C.

```
int  i;
int  a[10];
int* b;
...
foo() {
   i = a[j];  // first statement
   i = b[j];  // second statement
}
```

2a Are both statements legal in C? If so, what, if anything, is the difference between them?

yes, they r legal.
a[j] is the mem at m[r0+4*j], where r0 = add of a[0]

b[j] is m[m[r0]+4*j], where r0 is the address b points to

2b How many memory reads are required to occur when the first statement executes, assuming every register stores the value 0? Carefully explain.

1: ld $a, r0; ld (r0, r1, 4), r2
2: ld $b, r0; ld (r0), r0 // add of b[0]
ld (r0,r1,4), r2

2c How many memory reads are required to occur when the second statement executes, assuming every register stores the value 0? Carefully explain.

3 Answer the following questions about the two statements below that read an element of different structs in C.

```
struct R {
    int x;
    int y;
    int z;
};
int    i;
struct R  s;
struct R* t;
...
foo () {
    i = s.z;   // first statement
    i = t->z;  // second statement
}
```

3a Are both statements legal in C? If so, what is the difference between them?

3b Give a complete SM213 assembly code implementation of the second statement (i.e., `i = t->z;`). Use labels (or arbitrary numbers) to refer the values the compiler knows. Carefully comment every line of your code.

4 Consider the following C code. Does the execution of this code generate a runtime error? If not, what is the value of i after the code executes? **Show your work to justify your answer.**

```
int a[10] = { 0, 2, 4, 6, 8, 10, 12, 14, 16, 18 };  // A[i] = 2*i
int i     = *(*((&a[3])-1) + (a+2) + (&a[7])-(&a[5]));
```

5 Compare the following two alternative implementing a similar computation. Recall that "`strncpy (s1, s2, n)`" copies up to n bytes of string s2 into strings s1.

```
char* foo () {
    char* x = (char*) malloc (8);
    strncpy (x, "Hello.", 8);
    return x;
}

void bar () {
    char* y = foo ();
}
```

```
char* foo () {
    char* x = (char*) malloc (8);
    strncpy (x, "Hello.", 8);
    free (x);
    return x;
}
void bar () {
    char* y = foo ();
}
```

Both versions of this code have a different bug. Carefully describe each bug (give its name if you can) and then write one new version that fixes both bugs.

5a Describe the bug on left-hand side.

5b Describe the bug on right-hand side.

free(x) is dangeous, b/c in bar(), y is a pointer that points to the same mem as x does. So free x may change the data in that mem while y still use that mem

5c Re-write the code so that neither bug is present.

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,r1`
load base+dis	`1psd`	$r[d] \leftarrow m[(o = p \times 4) + r[s]]$	`ld o(rs),rd`
load indexed	`2sid`	$r[d] \leftarrow m[r[s] + r[i] \times 4]$	`ld (rs,ri,4),rd`
store base+dis	`3spd`	$m[(o = p \times 4) + r[d]] \leftarrow r[s]$	`st rs,o(rd)`
store indexed	`4sdi`	$m[r[d] + r[i] \times 4] \leftarrow r[s]$	`st rs,(rd,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	`7doo`	$r[d] \leftarrow r[d] << oo$	`shl oo, rd`
		(if oo is negative)	`shr -oo, 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 $\leftarrow aaaaaaaa$	`jmp aaaaaaaa`
get program counter	`6fpd`	$r[d] \leftarrow$ pc $+ (o == 2 \times p)$	`gpc $o, rd`
jump indirect	`cdpp`	$pc \leftarrow r[r] + (o = 2 \times pp)$	`jmp o(rd)`
jump double ind, b+disp	`ddpp`	$pc \leftarrow m[(o = 4 \times pp) + r[r]]$	`jmp *o(rd)`
jump double ind, index	`edi-`	$pc \leftarrow m[4 \times r[i] + r[r]]$	`jmp *(rd,ri,4)`

Operation	Machine Language Example	Assembly Language Example
load immediate	`0100 00001000`	`ld $0x1000,r1`
load base+dis	`1123`	`ld 4(r2),r3`
load indexed	`2123`	`ld (r1,r2,4),r3`
store base+dis	`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:  8004`	`br 0x1008`
branch if equal	`1000:  9104`	`beq r1, 0x1008`
branch if greater	`1000:  a104`	`bgt r1, 0x1008`
jump	`b000 00001000`	`jmp 0x1000`
get program counter	`6f31`	`gpc $6, r1`
jump indirect	`c102`	`jmp 8(r1)`
jump double ind, b+disp	`d102`	`jmp *8(r1)`
jump double ind, index	`e120`	`jmp *(r1,r2,4)`