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

Date: March 2014; Instructor: Mike Feeley

NAME: _____ **STUDENT NUMBER:** _____

1 **(15 marks)** The following SM213 assembly code implements a simple procedure. Carefully comment every line of the code, give an equivalent C program, and then describe in plain, high-level English what it does.

```
Y:   ld   0x0(r5), r1          # r1 = a
     ld   0x4(r5), r2          # r2 = b
     ld   0x8(r5), r3          #
     ld   $0x0, r0             #
L0:  bgt  r1, L1               #
     br   L2                   #
L1:  dec  r1                   #
     ld   (r2, r1, 4), r4      #
     st   r4, (r3, r1, 4)      #
     add  r4, r0               #
     br   L0                   #
L2:  j    0x0(r6)              #
```

```
Y(int a,int* b,int* c){
        int m = 0;
        int i = a;
        int j[] = b;
        int k[] = c;
        while ( i>0 ){
                i--;
                k[i] = j[i];
                m+= j[i];
        }
return m;
```

1a Equivalent C program:

1b Plain, high-level English description of what it does:

it copies an array to another array

2 (5 marks) PC-relative addressing branch instructions are normally used to implement if statements and loops, but absolute addressing jump instructions are normally used to implement procedure calls. Carefully explain why.

this is because relative addressing branch jumps to procedures that are close to each other, it is faster than abs jump. So we use relative add to implements if and loops, because they both jump to add nearby. Since there is no limits in abs jump, so we use it to perform procedure call, then it can jump to anywhere it want. In addition, it would be a waste if we use abs jump to perform if and loop, b/c it reads more instruction.

3 (10 marks) In the lab you looked at a form of a *stack smash* virus attack in which an attacker exploits a array overflow bug to get your program to run the attackers program. Carefully explain how this attack works, paying careful attention to the role the format of the stack plays in enabling this attack. Then, explain one change you would make to the organization of the stack that would eliminate this particular attack.

4 **(10 marks)** Consider the following C code.

```
void x () { printf ("x"); }
void y () { printf ("y"); }
void z () { printf ("z"); }
void (*proc)();

void foo () {
  proc = z;
  proc ();
}
```

4a Explain in a single, simple plain-English sentence what the user sees when `foo()` executes.

user will see "z"

4b Give commented SM213 assembly code that implements the statement `proc()`.

5 **(7 marks)** A `switch` statements can be implemented using a jump table or a sequence of `if` statements.

5a Explain the benefit of using jump tables (just the benefit; not how they work).

it is faster, it implements linear time.

5b Are jump tables always the best implementation choice for every switch statement? Explain carefully.

no, jump table uses memory, so if the variable and size of switch statement is big, then its better to use switch statement

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 $\leftarrow aaaaaaaa$	`j aaaaaaaa`
get program counter	`6fpd`	$r[d] \leftarrow$ pc $+ (o == 2 \times p)$	`gpc $o, rd`
jump indirect	`cdpp`	$pc \leftarrow r[r] + (o = 2 \times pp)$	`j o(rd)`
jump double ind, b+disp	`ddpp`	$pc \leftarrow m[(o = 4 \times pp) + r[r]]$	`j *o(rd)`
jump double ind, index	`edi-`	$pc \leftarrow m[4 \times r[i] + r[r]]$	`j *(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:  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+disp	`d102`	`j *8(r1)`
jump double ind, index	`e120`	`j *(r1,r2,4)`