CPSC 213: Assignment 6

Due: Sunday, October 24, 2010 at 6:00 pm

Goal

First up in this assignment is to figure out how to mount a buffer overflow attack on an SM213 program running in the simulator. You will need to be up to date on lecture material and spend some time figuring out how the attack works.

Then you will complete the implementation of the SM213 ISA by adding two types of double-indirect jump instructions (base-plus-displacement and indexed). Double-indirect jumps determine their jump-target address by reading it from memory at runtime.

Next you will explore role of dynamic jumps in implementing dynamic procedure calls using Snippet A and switch statements in Snippet B.

Then you will test your understanding of the ISA and your skill at using the simulator to gain program understanding, by examining two SM213 assembly-language programs to determine what they do.

Extending the ISA

You will implement two additional instructions.

Instruction	Assembly	Format	Semantics
dbl ind jmp b+d	j *o(rt)	dtii	pc ← m[r[t] + (o == i*4)]
dbl ind jmp indx	j *(rb,ri,4)	ebi0	pc ← m[r[b] + r[i]*4]

Code Snippets Used this Week

As explained in detail below, you will use the following code snippet this week. There are C, Java and SM213 Assembly versions.

- SA-dynamic-call
- SB-switch

Your Implementation

You are implementing two methods of the CPU class in the Arch.SM213.Machine.Student package of the SM213 Simulator.

1. **fetch ()** loads instructions from memory into the **instruction** register, determines their length and adds this number to the **pc** register so that it points to the *next* instruction, and

then loads the various pieces of the instruction into the registers **insOpCode**, **insOp0**, **insOp1**, **insOp2**, **insOpImm** and **insOpExt** (for 6 byte instructions). The meaning of each of these registers and a primmer on the Java syntax for accessing them was given in class and is part of the online lecture slides and Companion notes. *No changes are required to your fetch stage.*

2. **execute ()** uses the register values stored by the fetch stage to execute the instructions, transforming the register file (i.e,. **reg**) and main memory (i.e., **mem**) appropriately. Update this function with the two additional instructions for this week.

Using the Simulator to Test and Debug Your Code

The simulator displays the current value of the register file, main memory and the internal registers such as the pc and instruction registers.

You will use the simulator GUI to test and debug your ISA implementation and to analyze the code snippets. In each case you should single-step through the machine code and carefully observe the state changes that occur in the process registers, register file and main memory.

Here are a few quick things that you will find helpful.

1. You can edit instructions and data values directly in the simulator (including adding new lines or deleting them).
2. The simulator allows you to place "labels" on code and data lines. This label can then be used as a substitute for the address of those lines. For example, the variable's **a** and **b** are at addresses 0x1000 and 0x2000 respectively, but can just be referred to using the labels **a** and **b**. Keep in mind, however, that this is just an simulator/assembly-code trick, the machine instructions still have the address hardcoded in them. You can see the machine code of each instruction to the left of the instructions in the *memory image* portion of the instruction pane.
3. You can change the program counter (i.e., **pc**) value by double-clicking on an instruction. And so, if you want to execute a particular instruction, double click it and then press the "Step" button. The instruction pointed to by the **pc** is coloured green.
4. Memory locations and registers read by the execution of an instruction are coloured blue and those written are coloured red. With each step of the machine the colours from previous steps fade so that you can see locations read/written by the past few instructions while distinguishing the cycle in which they were accessed.

Requirements

Here are the requirements for this week's assignment.

1. Write a simple SM213 assembly-language program that copies a 0-terminated array of integers (use Snippets 8 or 9 as a guide). 0-terminated means the last element of the array is the number 0, which tells the program to stop copying the array.

   The source array should be stored in a global variable. The destination array should be a local variable (i.e., stored on the stack). You need two procedures: one that copies the array, one that initializes the stack pointer and calls the copy procedure. Ensure that the array-copy procedure saves r6 (the return address) on the stack in its prologue and restores

it from the stack in its epilogue, even though these steps are not strictly required for this procedure since it does not call any other procedures (we'll just pretend it does).

Here is a C template for the program.

```
int src[2] = {1,0};
void copy() {
     int dst[2];
     int i = 0;
     while (src[i] != 0) {
          dst[i] = src[i];
          i++;
     }
}
void main () {
     copy ();
}
```

2. By changing only the values of src and the size of src, construct a buffer overflow attack on this program such that the end result is this: the value of every register is set to -1 and then it halts at the end. You may not change the program or its stack directly when mounting this attack. Recall that what you are doing here is what most virus writers do to exploit buffer-overflow bugs to gain control of programs (i.e., to get those programs to execute their virus code) and that a real virus writer will have the virus do something more sinister than just changing the value of registers.
3. Implement the double indirect jump instructions and extend your test program to test them.
4. Execute the snippets in the simulator, step by step. Carefully examine their behaviour and document the key changes you see to the register-file and memory.
5. Execute the SM213 programs A6-a.s and A6-b.s to determine what they do. Explain their behaviour by giving an equivalent C program and by explaining in plain English what simple computation they perform. Basically you are trying to determine the C code from the given SM213 assembly.

Material Provided

The Simulator code was provided as part of Assignment 1 and modified in Assignments 2-5. Use this code as the starting point for this assignment.

This assignment includes snippets A and B in the file snippet-A-B.zip.

The mystery SM213 programs are located in code.zip.

What to Hand In

Use the handin program. The assignment directory is **a6**. Use one big README.txt text file for all of your answers. Avoid PDF or DOC or other file formats. Make sure you put down your student name and student number on the top, or marks will be deducted. For the SM213 code, only handin the files you've changed – for example, CPU.java for this assignment - and don't include the other .java files.

1. Your buffer-overflow program and the values of src you used to mount the attack.
2. A plain-English description of your buffer overflow virus and how it works.
3. Your modified CPU.java
4. Your test program.
5. A written description of the key things you noted about the machine execution while running snippets A and B.
6. A description of the mystery programs A6-a and A6-b that includes **(a) an equivalent C program** and **(b) a plain-English description**.