CPSC 213 – Assignment 6

Static Control Flow and the Stack

Due: Wednesday, July 24, 2019 at 11:59pm
You may use one of your two late days on this assignment to make it due Thursday.

Learning Objectives

Here are the learning objectives of this assignment, which will be examined in the corresponding quiz. They are a subset of the unit learning objectives listed on Slide 2 of Units 1c and 1d.

After completing this assignment you should be able to:

1. translate C code containing for loops into SM213 assembly language;
2. translate C code containing if-then-else statements into SM213 assembly language;
3. identify procedure calls and returns in SM213 assembly language and describe their semantics by writing equivalent C procedure call and return statements; and
4. explain how to use the stack pointer to access local variables and arguments.

Goal

This assignment has four parts. Start early - the first half of the assignment is highly relevant to the midterm, and Q3 can be quite challenging.

First, you will extend the SM213 implementation to add the instructions we have discussed in class that support static control flow, including static procedure calls and procedure return (which is dynamic, actually).

Second, you'll examine the assembly code for for-loops and if-statements using snippet files in the simulator and then translate a simple C file containing these control-flow statements into assembly.

Third, you will write a fairly substantial assembly-language program that uses all of the language concepts we have discussed so far.

Finally, you will examine how programs use the runtime stack to store local variables, arguments and the return address. You will do this using a set of snippets and you will answer questions about their execution. Then, you will examine two SM213 programs that contain procedure calls to determine what they do.

Question 1: Extending the ISA [10%]

Implement the following six control-flow instructions in CPU.java. The code provided with this assignment in www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a6/code.zip includes the file CPU.java that you can use as the starting point for this assignment. Alternatively you can use the version you implemented yourself for Assignment 2.

Note that in the "**Format**" column below, capital letters are hex digit literals and lower-case letters represent values referenced in the "**Assembly**" and "**Semantics**" columns.

Instruction	Assembly	Format	Semantics
branch	`br A`	`8-pp`	pc ← (A = pc + pp*2)
branch if equal	`beq rc, A`	`9cpp`	pc ← (A = pc + pp*2) if r[c]==0
branch if greater	`bgt rc, A`	`Acpp`	pc ← (A = pc + pp*2) if r[c]>0
jump	`j A`	`B--- aaaaaaaa`	pc ← (A = aaaaaaaa)
get pc	`gpc $o, rd`	`6Fpd`	r[d] ← pc + (o = p*2)
indirect jump	`j o(rt)`	`Ctpp`	pc ← r[t] + (o = pp*2)

Note that the indirect-jump offset is unsigned and so for indirect jump `j o(rt)`, `pp` ranges from `0` to `255` (so o ranges from 0 to 510). On the other hand, the branch pc-relative value is signed and so for branches, `pp` ranges from `-128` to `127`.

Question 2: Assembly code of if and loops [20%]

Example the Execution of Assembly Snippets

The file www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a6/code.zip contains the following files:

- `S5-loop.{java,c,s}`
- `S5a-loop-unrolled.{c,s}`
- `S6-if.{java,c,s}`

Run each of these snippets through the simulator – your version if you completed Question 1 correctly, or the reference implementation if you didn't. Carefully observe what happens as they execute so that you could explain this code to someone if they asked. This step is not for marks.

Q2a: Translate while loop to assembly

Translate `q2a.c` into commented assembly code, placing your code in a file named `q2a.s`. Labels for variables should be the name of the variable. Run your code in the simulator and examine every variable to be sure they have the correct value when execution halts.

In the file `q2a.txt`, explain what this while loop does: express the output `c` in terms of the inputs `a`, `b` and `c`.

Q2b: Translate for loop to assembly

Translate `q2b.c` into commented assembly code, placing your code in a file named `q2b.s`. Labels for variables should be the name of the variable. Run your code in the simulator and examine every variable to be sure they have the correct value when execution halts.

Question 3: Write a program in assembly [50%]

Implement an assembly-language program that examines a list of student grades and finds the student with the *median* average grade; this student's id should be placed in the variable `m`.

Your program must correspond to a C program where the input list of students, and the output student id are given as follows.

```
struct Student {
   int sid;
   int grade[4];    // grades for four assignments
   int average;     // this is computed by your program
};

int n;              // number of students
int m;              // you store the median student's id here
struct Student *s;  // a dynamic array of n students
```

Your assembly file must format student records exactly as C would; *i.e.*, a struct with 6 integers. You must use the following assembly declarations for this input and output data. Note: this example is for an array with one student. You should obviously test your code on larger arrays.

```
n:    .long 1     # just one student
m:    .long 0     # put the answer here
s:    .long base  # address of the array
base: .long 1234  # student ID
      .long 80    # grade 0
      .long 60    # grade 1
      .long 78    # grade 2
      .long 90    # grade 3
      .long 0     # computed average
```

Put your solution in a file named `q3.s`.

Work Incrementally – Do each of these steps separately!

This is a very challenging programming problem. It will stretch (and strengthen) your ability to manage a large number of programming details, as is required when writing assembly. To be successful with this you must be very disciplined to program and test incrementally.

Observe that the program breaks down into several subproblems. You should tackle them one at a time and thoroughly test each step before moving to the next.

Here's a list of the key subproblems. You might want to further sub-divide things, but do not try to combine steps.

1. Compute the average grade for a single student and store it in the struct. You must round this grade (when dividing, add half the divisor before dividing to round the result).
2. Iterate through the list of students, computing each student's average grade and storing it in the structure.
3. Swap the position of two adjacent students in the list.
4. Compare the average grades of two adjacent students and swap their position conditionally, using your code from Step 3.
5. Now consider creating a procedure to encapsulate either Step 3 or Step 4 as described in the *Use Procedures* section below.
6. Sort the array by average grade in ascending order. You are free to use any sort algorithm you like, but Bubble Sort is the simplest. Here's a version of bubble (sinking) sort in C that you might consider.

```
void sort(int *a, int n) {
  for(int i=n-1; i>0; i–) {
    for(int j=1; j<=i; j++) {
      if(a[j-1] > a[j]) {
        int t  = a[j];
        a[j]   = a[j-1];
        a[j-1] = t;
      }
    }
  }
}
```

   NOTE: You **must** sort the array; it is not sufficient to simply find the median. The autograder will check for a sorted array in addition to checking the median output in `m`.

   Take this step by step and incorporate your work from Step 3-5 above. For Bubble Sort, here are two subproblems.

   a. First, write a loop that iterates through the list once, bubbling the student with the lowest average to the top (or sinking the lowest student to the bottom). If you created a procedure in Step 5, then the inner part of this loop is a call to this procedure; otherwise it is the code from Step 4.
   b. Then, add the outer loop that repeats the inner loop on the unsorted sublist repeatedly until the entire list is sorted.

7. Find the median entry in the sorted list and store that student's sid in `m`. For simplicity you can assume the list contains an odd number of students.

Strategy

Notice that it is not necessary to do these steps in the order listed above. Steps 1 and 2, for example, are completely independent from Steps 3-7. Step 7 is independent from the other steps. You could do Step 6 before Steps 3-5 and then incorporate the conditional swap once the other parts of Step 6 are working. Similarly, you can do Step 5 before Steps 3 or 4 and incorporate them into the procedure later. The key thing here is work on each piece independently and then carefully combine steps to eventually produce the program. The program itself will be on the order of 70 to 100 assembly-language instructions. That sounds like a lot … and it is. But if you think of this as seven steps each with 15 or so instructions (some will have less), it's not so bad.

Be sure to comment every line of your code and to separate subproblems with comments and labels so that it is easy for you to see what each part of the code does without having to read the code.

Multiplying by 24

You will have noticed that the variable `s` is a dynamic array of type "`struct Student`" and that "`sizeof(struct Student)`" is 24. Unfortunately, our indexed-load instruction only multiplies indices by 4, not 24, so you'll need to find a way to multiply by 24. Our ISA lacks a multiply instruction. However, using the observation that $x*24 = x*16 + x*8$, you might be able to do this multiplication using fewer instructions.

Register Allocation

Keeping track of registers is going to be a challenge. Note that this program is too big for you to use a distinct register for every value in the program. You are going to have to re-use registers to store different things in different parts of the program.

This will add complexity, for example, when combining two steps or when adding a step to the rest of the program. For example, if your code for Step 4 uses register `r0` and one of the loops you write in Step 6 also uses `r0`, you're going to have to change one of them to use a different register (or use a procedure as described in the next section). So, for parts of the code that connect like this, some pre-planning will help.

Another useful strategy is to group sections of code (one or maybe a few steps) and treat them as independent stages with respect to registers. At the beginning of a stage you assume nothing about the current value of registers and you are free to use any registers you like within that stage. At the end of the stage, any register values that are needed by subsequent stages should be written to memory. You may want to create some additional variables to store these temporary values.

Using Procedures

One way to divide code into stages is to use procedures. Step 5 suggests that you do this to encapsulate the code that conditionally swaps two adjacent elements of the list (i.e., Steps 3 and 4).

You can use this approach to simplify register management by having the procedure save registers to memory before it starts and restore them from memory before it returns (e.g., using temporary variables). Any register the procedure saves in this way is a register it is free to use without interfering with the registers used in the two loops in Step 6 that call it.

This swap procedure needs one argument / parameter: i.e., the index of one of the array elements to swap. The index of the other element is this value plus or minus one, depending on how you do it. The caller should pass this value in a register; for example if the loop has the index in `r4` then the procedure should just use `r4` to get this value.

Use `gpc` and `j` to call the procedure and use the indirect jump to return. Do not worry about creating a stack.

All of this is optional, and you can do this for other parts of your program as well.

Partial Marks — Work Incrementally

Finally, partial marks will be awarded based on the number of the steps listed above that you've written correctly. Submitting a big blob of assembly that doesn't do any of the steps right won't be worth anything. So, work incrementally. Test each step separately. Once you have a step working, save it in an auxiliary file just in case.

Question 4 – The Stack [20%]

The provided code package contains the following files.

- `S7-static-call.`{`java`,`c`} and `S7-static-call-`{`stack`,`reg`}`.s`
- `S8-locals-args.`{`java`,`c`,`s`}
- `S9-args.`{`java`,`c`} and `S9-args-`{`stack`,`regs`}`.s`

Evaluating Snippets using Simulator

Familiarize yourself with snippets 7-9 by running them in the simulator and asking yourself the following questions. This part is not for marks.

- Carefully examine the execution of `S7-static-call-stack` in the simulator and compare it to `S7-static-call-regs`. Run the snippets and look carefully at what happens. Ask yourself these questions.
  a) What is the difference between the two approaches?
  b) What is one benefit the approach followed in `stack`?
  c) What is one benefit of the approach followed in `reg`?
- Carefully examine the execution of `S8-locals-args.s` in the simulator. Ask yourself these questions.

d) What lines of `foo` and `b` allocate `b`'s stack frame?

e) What the lines of `foo` and `b` de-allocate `b`'s stack frame?

The next two questions ask you to consider changing `b` so that it has 3 arguments and 4 local variables. Ask yourself these questions.

f) What changes required in `b` to add the arguments and locals; note that you do not actually use these new variables in any way?

g) What changes required in `foo` to call `b(0,1,2)`?

- Carefully examine both versions of `S9-args-stack.s` and `S9-args-regs` in the simulator. Ask yourself these questions.

h) What memory accesses does `stack` makes that `regs` doesn't make?

i) How many more memory accesses does `stack` make, compared to `regs`?

Questions 4a and 4b: Reverse Engineering [20%]

The next two question use these files found in code.zip.

- `q4a.s`
- `q4b.s`

Answer the next two questions by modifying the `.s` files and by writing `.c` files. The `.c` files must compile and execute. When they execute they must perform the same computation as the `.s` file and print out the value of its static variables, one per line as a decimal number (nothing other than that number on each line). This means that `q4a.c` must print 10 lines of numbers and `q4b.c` must print 16.

1. [10%] Examine `q4a.s` and its execution in the simulator. Add a comment to every line that explains what that line does in as high-level a way as possible.

   Then, write an equivalent C program called `q4a.c` that is the most likely candidate for the file that was compiled into `q4a.s`. Use the same variable and procedure names in this program that you used in the assembly-file comments. Ensure that there is a correspondence between lines in the assembly file and lines of the C program, but do not include the `start` procedure in `q4a.c`; this procedure is added automatically by the c compiler to initialize the stack and call `main`. The end of your `main` procedure should print the value of the program's ten static variables as described above.

   You may notice a register that is used in a way that is best explained by saying that it is a local variable, even if its value is never read from or written to the stack. Avoiding these memory accesses is a common optimization compilers make for local variables.

2. [10%] Do the same for `q4b.s`.

What to Hand In

Use the `handin` program.

The assignment directory is **`~/cs213/a6`**, it should contain the following ***plain-text*** files.

1. (optional) `PARTNER.txt` containing your partner's CWL login id and nothing else. Your partner should not submit anything.
2. `CPU.java` that contains your implementation of the new instructions listed in Step 3 as well as a correct implementation of the part of the ISA you implemented previously.
3. `q2a.s`, `q2a.txt` and `q2b.s` containing your answers for Question 2.
4. `q3.s` that contains your answer to Question 3.
5. `q4a.s`, `q4a.c`, `q4b.s`, and `q4b.c` that contain your answers to Question 4.