CPSC 213 – Assignment 6

Static Control Flow

Due: Monday, February 20, 2017 at 11:59pm (with grace period, Tuesday 9am)
After an 9-hour, no-penalty, grace period, no late assignments accepted.

Goal

This assignment has three parts.

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-follow statements into assembly.

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

Question 1: Extending the ISA [20%]

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 hex 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
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, `pp` ranges from `0` to `255`. On the other hand, the branch pc-relative value is signed and so for branches, `pp` ranges from `-128` to `127`.

Test your implementation by creating a file named `test.s` that contains tests for each of the new instructions.

Question 2: Assembly code of if and loops [10%]

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.

Translate C to assembly [20%]

Translate `q2.c` into commented assembly code, placing your code in a file named `q2.s`. Labels for variables should be the name of the variable. Be sure that this code runs in the simulator.

Note that the array in `a2.c` contains some negative numbers. The version of the simulator you have does not allow negative numbers in `.long` directives. Of course, you can convert the negative numbers into unsigned, two's-complement hex to load them (e.g., "`.long 0xfffffffe`" is -30). But, for your convenience, new version of the simulator that accepts negative numbers in `.long`'s is include in *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a6/code.zip*. We will use this version for marking and so you can say either "`.long 0xfffffffe`" or "`.long -30`" in your code.

Question 3: Write a program in assembly [60%]

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];
    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 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. For simplicity, you can ignore the fractional part of the average; i.e., you do not need to round.
2. Iterate through the list of students and compute their average grades and store them.
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 list by average grade. 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;
          }
    }
```

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.

Ordering and Combining Steps

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 careful 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), its not so bad. Or maybe it will be bad, but at least not impossible.

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 and Dividing 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. And so you might find that you'd like to multiple or divide by 24 (i.e., to convert between an array index and a byte-offset), but you'll recall that you can't do this with a single instruction in our ISA. Then you'll notice that *x*24 = x*16 + x*8* and you'll see that you can multiple by 24 (or divide) using 3 instructions (or 4 depending on how you count).

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 (if you need them … probably not!) 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.

What to Hand In

Use the `handin` program.

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

1. `README.txt` that contains the name and student number of you and your partner
2. `PARTNER.txt` containing your partner's CS login id and nothing else (i.e., the 4- or 5-digit id in the form `a0z1`). Your partner should not submit anything.
3. `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.
4. `test.s` that contains your test code for the new instructions.
5. `q2.s` that contain your answer to Question 2.
6. `q3.s` that contains your answer to Question 3.