CS 352 Homework

Bleak

1 Overview

Your responsibility in this homework is to learn about assembly programming, the mechanics of a CPU, and the primitive instruction sets upon which complex computation is built. You will do this in the context of creating a virtual machine for an assembly language called Bleak.

2 Bleak ISA

Bleak's *instruction set architecture* (ISA) is simpler than x86 and ARM. To be perfectly honest, the architecture is largely inspired by the video game Human Resource Machine. In summary, Bleak has the following:

1. An input list of integers.
2. An output list of integers.
3. A mapping from human-readable string labels to instruction numbers.
4. Ten general purpose registers, named `r0`, `r1`, ..., `r9`.
5. A register named `pc`—short for *program counter*—for recording which instruction is next to be executed.
6. A register named `nc`—short for *iNput counter*—for recording which integer input is next to be read.
7. A register named `ra`—short for *return address*—for recording which instruction should be jumped to after a function call.
8. Instructions `inc`, `dec`, `add`, and `sub` for performing arithmetic.
9. Instructions `input` and `output` for performing I/O.
10. Instructions `jmp`, `jpos`, `jneg`, and `jzilch` for jumping to labeled instructions.
11. Instruction `store` for assigning a value to a register.

Before reading on, try using your intuition to manually execute this example Bleak program given the input 1 2 3 4:

```
input r0 <- read
input r1
add r0 r1
output r0
jmp read
```

2.1 Execution

When the Bleak VM starts up, all registers—general purpose or otherwise—have value 0. All instructions and input are gathered into an indexable collection for use during execution. It traverses the code to identify which labels map to which instructions.

On each step of execution, it executes the instruction currently pointed to by `pc`, triggering any side effects associated with the instruction. Then it updates `pc` to prepare for the next instruction. Often this means incrementing `pc` by 1, but jump instructions may change `pc` in less predictable ways.

The virtual machine halts when it encounters one of the following situations regarding the instruction about to be executed:

1. The instruction doesn't exist—that is, `pc` is out-of-bounds.
2. The instruction is `input`, but no more input is available to be read—that is, `nc` is out-of-bounds.
3. The instruction is unknown.

2.2 Instructions

Following is the complete grammar specification for Bleak's 13 instructions. The placeholders `lval` and `rval` are discussed in Section 2.3.

`add lval rval` Adds the values together and stores the result in `lval`.

`sub lval rval` Subtracts `rval` from `lval` and stores the result `lval`.

`inc lval` Adds one to `lval` and stores the result in `lval`.

`dec lval` Subtracts one from `lval` and stores the result in `lval`.

`input lval` Reads the next input from the input list (as indicated by `nc`) and stores the result in `lval`. Increments `nc`

`output rval` Emits `rval` to the output list.

`store lval rval` Copies `rval` into `lval`.

`jmp label` Alters `pc` so that the next instruction executed is the one with the given label.

`jpos rval label` If `rval` is positive, alters `pc` so that the next instruction executed is the one with the given label.

`jneg rval label` If `rval` is negative, alters `pc` so that the next instruction executed is the one with the given label.

`jzilch rval label` If `rval` is 0, alters `pc` so that the next instruction executed is the one with the given label.

`call label` Alters `pc` so that the next instruction executed is the one with the given label, but also records in register `ra` the address of the instruction that follows the `call` instruction. This instruction effectively produces a function call. Unlike other jumps, we expect `call` to be paired with a `return` to restore execution to the instruction after the `call`.

`return` Alters `pc` so that the next instruction executed is the one identified by register `ra`.

Any instruction may be followed by `<- label`, where `label` is an alphanumeric string that can be used to assign a symbolic name to an instruction. Jump instructions reference these labels rather than an instruction's less meaningful line number. Consider this example program, which for each number `N` in the input, outputs all numbers [0, `N`]:

```
input r0 <- getN
store r1 0
jneg r0 getN <- checkN
output r1
inc r1
dec r0
jmp checkN
```

2.3 Operands

The operands of an instruction come in two flavors: *lvalues* and *rvalues*. Lvalues are values associated with an identifiable location in the virtual machine—in our case, a register. Any instructions that involve an assignment need a location to store the value and will therefore have an lvalue operand[1].

We can also have values that are more temporary in nature and not necessarily associated with a location. Literals, for instance, are part of an instruction and don't reside in an accessible location. Literals are rvalues—values that appear in an expression but whose location is not important. Registers too are rvalues when we are merely referring to their value but aren't assigning anything to them.

2.4 Direct vs. Indirect Addressing

Registers can be accessed directly through their names `r0`...`r9`. However, they can also be accessed indirectly through another register. In the expression `[r0]`, `r0` is assumed to hold a value in the range 0 through 9. Suppose `r0` is 5. We say that `[r0]` refers to or points to `r5`.

This indirect addressing scheme can simplify certain algorithms. Suppose one wants to assign the value 100 to registers `r1` through `r9`. This code accomplishes the task:

```
store r1 100
store r2 100
store r3 100
store r4 100
```

[1] You may be tempted to define *l*values as those expressions that can appear on the *left* side of assignment. However, constants that reside in a particular place in memory are also lvalues in many high-level languages, but these don't appear on the left-hand side of any assignment. Therefore, think of *l* as short for *locatable*.

```
store r5 100
store r6 100
store r7 100
store r8 100
store r9 100
```

But indirect addressing allows us to write this with less grunting. We set up `r0` as a counter and have it point to each of the registers. Then we indirectly store 100 through the `r0` counter:

```
store r0 9
store [r0] 100 <- fill
dec r0
jpos r0 fill
```

3 Requirements

Implement the files, classes, or routines described below.

3.1 Makefile

Create a `Makefile` to compile your code. Note the capital M. Its default (topmost) rule should build an executable named `main` that runs your `main` function. To be most useful, this target should depend on `BleakVirtualMachine.o`, `main.o`, and `Makefile`. If any of these dependencies are newer than `main`, it rebuilds the executable.

Provide two general rules for compiling `*.o` files from their source, one that includes a header and one that does not:

```
%.o: %.cpp Makefile
  $(CPP) $(CFLAGS) -c -o $@ $<

%.o: %.cpp %.h Makefile
  $(CPP) $(CFLAGS) -c -o $@ $<
```

`$@` is a builtin variable that refers to the rule's target (the `*.o` file). `$<` refers to the leftmost dependency (the `*.cpp` file). Define `CPP` to be `g++` and `CFLAGS` to include debugging information (`-g`) and invoke a modern C++ standard (`-std=c++11`).

Also provide a `clean` rule like the following to dispose of any derived files:

```
rm -f *.o main
```

3.2 Main

Write a C++ file `main.cpp` with a `main` function, which you are encouraged to use to test your code. We will not actively test it, but it must exist.

3.3 BleakVirtualMachine

Write class `BleakVirtualMachine` to model a virtual machine running a single program on a single input file. Place its declaration in file `BleakVirtualMachine.h` and its method definitions in file `BleakVirtualMachine.cpp`. Do not include a `main` function in these files. Since C++ allows only one `main` function to exist within a scope and since the SpecChecker defines one to test `BleakVirtualMachine`, you cannot define one except in `main.cpp`.

This class has the following:

1. Constant `INSTRUCTION_OUT_OF_BOUNDS` defined to be 1, constant `INPUT_OUT_OF_BOUNDS` defined to be 2, and constant `UNKNOWN_INSTRUCTION` defined to be 3. All are `static`, `const`, and `int`s.

2. A constructor that accepts two parameters in this order:

   - A `const` reference to a source file path of type `string`
   - A `const` reference to an input file path of type `string`

   It loads in the instructions and input and stores them for later retrieval. It initializes the output list to be empty and each register to be 0. It also calculates the address (line number) that each label maps to.

3. Method `Reset`. It clears the output list and reverts each register to 0, so that it can be run again with no traces of earlier executions influencing its behavior.

4. Method `GetInstructions`, which is `const`. It returns a `const` reference to a `vector<string>` containing the instructions that the VM is executing.

5. Method `GetInput`, which is `const`. It returns a `const` reference to a `vector<int>` containing the inputs to the VM.

6. Method `GetOutput`, which is `const`. It returns a `const` reference to a `vector<int>` containing the outputs of the VM.

7. Method `GetRegisters`, which is not `const`. It returns a non-`const` reference to a `map<string, int>` that maps register names to their values. This method probably should be `const`, but we open it up so that testing code can tweak register values directly.

8. Method `GetLabels`, which is `const`. It returns a `const` reference to a `map<string, int>` that maps labels to their addresses.

9. Method `ResolveLValue`, which accepts one parameter: a `const` reference to an expression of type `string`. It returns a register name as a `string`. If the expression is itself a register name (a *direct* address), just return the expression as is. If the expression is a register name in square brackets (an *indirect* address), look up the number n stored in the specified register, and return the name of register n. For example:

   - `vm.ResolveLValue("r8")` → `"r8"`.
   - Suppose `r5` is 9. Then `vm.ResolveLValue("[r5]")` → `"r9"`.

If the expression is neither a direct nor an indirect address, or if a specified register does not exist, the results are undefined.

10. Method `ResolveRValue`, which accepts one parameter: a `const` reference to an expression of type `string`. It returns as an `int` the given expression's value. If the expression is a register name (a *direct* address), return the register's value. If the expression is a register name in square brackets (an *indirect* address), look up the number n stored in the specified register, and return the value of register n. Otherwise, assume the expression is an integer literal and return its value. For example:

    - `vm.ResolveRValue("37")` $\rightarrow$ 37.
    - Suppose `r8` is 1023. Then `vm.ResolveRValue("r8")` $\rightarrow$ 1023.
    - Suppose `r5` is 9 and `r9` is 20. Then `vm.ResolveRValue("[r5]")` $\rightarrow$ 20.

    If a specified register does not exist, the results are undefined.

11. Method `Tokenize`, which is `static` and which accepts one parameter: a `const` reference to whitespace-separated text of type `string`. It returns the individual tokens of the text as a `vector` of `string`s. We recommend you use a `stringstream` and pull off each token with the extraction (`>>`) operator.

12. Method `Step`, which executes the instruction indicated by register `pc`. If `pc` doesn't point to a legal instruction address, throw `INSTRUCTION_OUT_OF_BOUNDS`. Examine the first token of the instruction to determine its type and process it accordingly. If the instruction is not in the ISA, throw `UNKNOWN_INSTRUCTION`. If the instruction is `input` but there are no inputs left, throw `INPUT_OUT_OF_BOUNDS`. Otherwise, assume the instruction is well-formed, with whitespace separating each of the instruction's operands. After executing the instruction, update `pc` to be the address of the next instruction to be executed.

4 Debugger

We have provided a simple terminal-based debugger to help you test your solution and visualize the execution of a virtual machine. It uses the `curses` library to draw a textual user interface (TUI). Use the cursor keys to navigate between the *Step*, *Reset*, and *Quit* buttons at the bottom of the screen.

The debugger calls upon your `BleakVirtualMachine` to do all the work. Therefore, it must be compiled and linked with your code. Use the `debug` utility in the `specs` directory to compile, link, and run the debugger on a source file and an input file:

```
../specs/grade_bleak/debug source-file input-file
```

5 Later-week Submission

To qualify for later-week submission, you must provide `Makefile`, `main.cpp`, and define the constants, the constructor, and methods `GetInput`, `GetInstructions`, `GetRegisters`, `GetOutput`, `GetLabels`, `ResolveLValue`, and `ResolveRValue` for `BleakVirtualMachine`.

6 Submission

To submit your work for grading:

1. Run the grading script from your homework directory using `../specs/grade`.
2. Commit and push your work to your repository.
3. Verify that your solution is on Bitbucket by viewing your repository in a web browser.

A passing grading script does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The grading script checks some of them, but not all.
- You must successfully submit your code to your repository. Expect to have issues with Git.
- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don't want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.

The grading script allows you to signal your instructor when requirements are met. You only need to send an email if you qualified for later week submission and are resubmitting after the original deadline.