Module 1: Virtual-Machine State

Introduction

This week you'll build your prototype virtual machine: you'll design and implement a representation for virtual-machine state, and you'll validate your design with a simple `vmrun` function. Your `vmrun` function needs to support only a few instructions.

This week is about learning basic VM concepts and getting immigrated back into C programing; you won't write much new code. The machine state and its functions took me about 30 lines, and a simple `vmrun` with six instructions and no optimizations took another 30 lines.

- *What am I doing?*
  - You'll design and implement header file `<vmstate.h>`, which specifies your chosen representation for the state of a running VM.
  - You'll build the very first prototype of function `vmrun`, which runs VM code until it gets to a halt instruction. Your prototype will support just enough instructions to get the key ideas and run a simple test or two. (This week, I provide the tests.)
- *Why am I doing it?*
  - You'll build the foundation and core for your Simple Virtual Machine, i.e., the rest of the project.
  - You'll use C concepts, including pointers and local variables, to design code that provides fast access to the most frequently used data.
  - You'll start learning the concepts of *embedding* and *projection* and how to use them to take advantage of existing code. You'll apply these concepts to manipulation of values and to instruction decoding.
  - You'll refine (or develop) skills at translating formal inference rules (in our case, operational semantics) into code.
  - You'll get reacquainted with C programming and will look at interfaces for doing basic tasks like print error reports, manipulate bits, or work with hash tables—old stuff from CS 15 and CS 40.
- *How?*
  - In pre-lab you'll review the recommended values and the operational semantics for the VM. You'll also download course code and confirm that you can compile it.
  - In lab you'll define the representation of your VM state, and you'll start your `vmrun` function with just a few instructions.
  - To prepare to implement more VM instructions, you'll study instruction formats and decoding, and you'll find the embedding and projection functions for values.

– At the end of the week you'll deliver a `<vmstate.h>` and a `vmrun` function, which will link with my code in an `svm-test` binary. You'll be running VM code!

The module step by step

Pre-lab reading

(1) Watch the short video on "Embedding and Projection"

Optional: Read about embedding in section 5.4 of *Programming Languages: Build, Prove, and Compare*.

(2) Read the handout on "VM Semantics, part I."

Pre-lab exercises

(3) Clone the student Git repository from https://gitlab.cs.tufts.edu/cs106-staff/student-2023s. Every file is documented, but you don't need to look at the documentation just yet.

(4) In your clone, go to subdirectory `src/svm`, run `make svm-test`, and confirm that the GNU Makefile builds a test binary in the `bin` directory.

(5) Add the `bin` directory to your Unix PATH, run `svm-test`, and confirm that it halts with an assertion failure.

(6) Think about how you to define a C `struct` that represents the state of the virtual machine.

(7) Study the header file `<value.h>`, which is described in detail in the overview of the code. Identify the five tags of values used to represent S-expressions. Pick three of the tags, and for each one, write down at least one machine instruction that you would like to have in the VM that either produces or consumes a value of that tag. Start with the *semantics* of the instruction; here are some examples:[1]

- An instruction that adds two registers and puts the result in a third register
- An instruction that sets a register to zero
- An instruction that examines a value in a register and projects that value to a Boolean—using some notion of "truthiness"—and puts the Boolean in a register
- An instruction that allocates and initializes a `cons` cell

An instruction design works only if it can be coded within the limitations of the 32-bit format. There are several instruction formats available, but you're always safe designing an instruction that operates on one, two, or three VM registers.

(8) Now study the short handout "Instruction Formats" and the accompanying header file `<iformat.h>`. Choose a format for each of your instructions in step 6, and identify the decoding functions you will need to implement each instruction.

[1] Not all of the examples are good ideas, and not all of them are easy to implement, but at least one example is both, and every example is *either* a good idea or easy to implement.

If any of your hoped-for instructions doesn't fit any of the available formats, either redesign the instruction or choose another instruction.

Come to lab prepared with your list of instructions and the decoding function you will need for each one.

(9) Finally, think about how you will answer these questions:

   A. Do you like to start work as soon as it's assigned, or do you prefer to wait until later?

   B. Do you prefer to work before dinner or after dinner?

   C. What attitudes or skills do you bring to a programming partnership?

   D. What do you look for in a programming partner?

Lab

Introductions

(10) When instructed, count off (most likely by fours).

(11) Join the other students who have the same number as you. Introduce yourselves, and exchange answers to these questions:

   A. Do you like to start work as soon as it's assigned, or do you prefer to wait until later?

   B. Do you prefer to work before dinner or after dinner?

   C. What attitudes or skills do you bring to a programming partnership?

   D. What do you look for in a programming partner?

Coding

You may work the coding problems as a single group, or if your group seems to big, you may split into two subgroups.

(12) Using the suggestions in the handout on VM semantics, edit `vmstate.h` to define `struct VMState`.

(13) Implement functions `newstate` and `freestatep`; for now, skip `literal_slot`. A VM state is never garbage-collected, so allocate space using `malloc` or `calloc`; free it with `free`. To confirm that your code compiles, run `make obj`.

(14) Edit the template file `vmrun.c` and alter the `vmrun` function so it runs a sequence of instructions. Implement just one instruction: `Halt`. The halt instruction tells `vmrun` to save any cached VM state onto the heap and to `return`. (The idea of caching is explained in the handout on efficient interpretation, so depending on the choices you make, you may not have anything cached during lab.)

(15) Using the `Makefile`, compile and run the `svm-test` binary. You should get an output suggesting that the code passed one test, something like this:

```
Test: halt
Test: print $r0; halt
Run-time error: Unknown opcode 17 in instruction 0x11000000
```

(16) If time permits, you can add more instructions. (Over the next 17 weeks, you'll be doing that a lot.) To pass my basic tests, implement opcodes `Print`, `Check`, and `Expect`, and go back to `vmstate.c` to implement `literal_slot`.

- `Print` prints the value in register X, followed by a newline. If `rx` is the value, you can call `print("%v\n", rx);`.
- `Check` calls function `check` from header file `<check-expect.h>`. It has a register number in the X field and literal index in the YZ field.
- `Expect` is just like `check`, except it calls `expect`.

Post-lab

(17) If necessary, complete your lab work.

(18) Implement the `literal_slot` function, which adds a `Value` to the VM's literal pool (and returns the index of the added value).

(19) Rebuild `svm-test`, which should now pass all three of my initial tests. Good output looks like this:

```
Test: halt
Test: print $r0; halt
nil
Test: check $r0, ...; expect $r0, ...; halt
The only test passed.
```

(20) Extend your implementation with the three instructions you chose in step 5. A simple ALU instruction generally requires three or four lines of code: a `case` line for the opcode, one or two lines for the δ semantics, and a line for `break;`.

(21) Read "Principles of Efficient Bytecode Interpretation" and if needed, adjust your `vm-run` to make it efficient.

(22) Revisit the learning outcomes and make sure you are in a good position to claim as many as you can.

(23) As time permits, implement more VM instructions. The more instructions you implement now, the fewer you will have to do later on.

- Each species of value can potentially be supported with VM instructions. Instructions that support numbers are the simplest and easiest. Lean on your embedding and projection functions!

- You'll want instructions that move data between VM registers and that move data between VM registers and VM global variables.
- Any vScheme primitive function is a legitimate candidate to be implemented as a VM instruction.
- The semantics defines two control-flow instructions, `if` and `goto`, which are both pretty easy to implement.

Hold off on instructions that support function calls. We'll do functions in module 7.

What and how to submit

(24) On Monday, *submit your work.* In the `src/svm` directory you'll find a file `SUBMIT.01`. That file needs to be edited to answer a number of questions:

- Say who did the work. If the work was done by a pair or group, *one person should submit on behalf of the group*. All group members will be credited with the work.
- Say what work you were proud of, what you'd like help with, and what you'd like the course staff to review.
- Say what code you'd like help with during code review. If there are bugs that code review might help you find, say what they are.
- If there's any code you'd like the course staff to look at, say what it is, and let us know why you'd like us to look at it.
- For the first module, I will present at a plenary code review. If any of you are willing to serve on the review panel, let us know. And if there's code you'd like me to present, let us know that, too.

Run `make clean`.

To submit, you'll need to copy your working tree to the department servers. We recommend using `rsync`, but `scp` also works.

Now log into a department server, change to the `src` directory of your working tree, and submit your entire `svm` directory:

```
provide cs106 hw01 svm
```

(25) On Tuesday, *submit* your reflection. Create a plain text file `REFLECTION`, which will hold your claims for project points and depth points.

For each project point you claim, write the number of the point, plus whatever is called for in the section "How to claim the project points"—usually a few sentences.

For point number 5, where you write a transition rule, please use plain text with Unicode characters. As an example, here's the first transition rule in the operational-semantics handout:

```
                  i ∈ dom δ
-----------------------------------------------
⟨I1 • i · I2, R, L, G, σ⟩ → ⟨I1 · i • I2, R, L, G, δ(i, R, L, σ)
```

Now copy your `REFLECTION` file to a department server and submit it using `provide`:

```
provide cs106 reflection01 REFLECTION
```

Learning outcomes

Learning outcomes for project points:

1. Completeness: The makefile builds `svm-test`, which passes its three tests.
2. Embedding and projection of values: You can analyze 7 instructions (`halt`, `print`, `check`, `expect`, and the three you propose) and say which ones embed values, which project values, which do both, and which do neither.
3. Embedding and projection of languages: a 32-bit word can sometimes be projected into a machine instruction with one 8-bit opcode and up to three operands. You can identify, by file names and function names, where in your code such projection is used. And you can identify, by line number, a decision point (`if` statement, `case` in a `switch` statement, or ternary expression) where it is determined whether a given 32-bit word can be projected into a machine instruction.
4. Reading formalism: In addition to the store σ, which is represented by the C heap, the operational semantics identifies four other components of abstract-machine state: an instruction stream with instruction pointer, a register file, a literal pool, and a global-variable table. For each of these components, you can say how it is represented in your `struct VMState`.
5. Writing formalism: You can write operational semantics for VM states. To claim a project point for this outcome, write a transition rule of operational semantics for an instruction that does one of the following: set a global variable, read a global variable, or copy a value from one VM register to another.
6. Performance: The state of a VM is stored in a `struct` that is allocated on the heap, but during the execution of `vmrun`, and with the help of the C compiler, some components may be stored in real machine registers. You can identify which components these are, you can name the C variables that store them, and you can explain why you chose them.
7. Reuse of values: You can say which types from `value.h` would be useful and not useful for implementing another language not in the Scheme family.
8. Reuse of code: You can say how you would reuse a C module (`.c` file and accompanying `.h` file) to implement another language not in the Scheme family.

Learning outcomes for depth points:

9. Depth (1 point), design: You can say what would constitute a *complete* set of VM instructions suitable for implementing vScheme. *Expires when module 2 is delivered.*
10. Depth (20 points), implementation: Add bignums to the SVM, and change the UFT so that bignums are the default numeric type. These points can be claimed any time before classes end
11. Depth (up to 5 points), tools: You use ChatGPT and/or Copilot to write some of your code, and you demonstrate your understanding of what the tool did well and how to

prompt it effectively. 1 point per module up to a maximum of 5. These points can be claimed any time before classes end

How to claim the project points

Each of the numbered learning outcomes 1 to 8 is worth one point, and the points can be claimed as follows:

1. Submit code that works.
2. List the instructions in each category.
3. Name all the functions *you wrote* that do such projection, with the files in which they appear. Give the file name and line number of one decision point.
4. Name each component of the semantic state and say what parts of the `struct VMState` implement it and how.
5. Write a transition rule. Use Unicode characters.
6. Name the components and variables, and say *briefly* why you chose them. A couple of sentences would be plenty.
7. To claim a project point for this outcome, choose one of the following dynamically typed languages: Awk, Erlang, Icon, JavaScript, Lua, Matlab, Perl, Python, Rexx, Tcl, or VBScript.[2] Then say for each type of tag whether it would be useful, not useful, or possibly useful. Finally, pick *one* tag in each category (useful and not useful) and explain your answer. (Example: "The `Number` tag would be useful in implementing Icon because Icon has arithmetic.")
8. To claim a project point for this outcome, choose one module from the sets "Code you will look at and understand" or "Code you will look at eventually". List three of the languages from the previous outcome that you think the module could help you implement. Pick one of the three and *explain* how you would use the module to help implement that language.

To give you an idea of the recommended scope of your answers, I've written a model reflection that addresses a related set of learning outcomes.

[2] Ideally choose one that you know. If you prefer a language that's not on this list, ask the instructor's approval on Slack.