Homework 3

Mayer Goldberg

April 29, 2018

Contents

1 Goals

The goal of this assignment is to get you practiced with writing self-modifying code in a "safe" environment. The use of self-modifying code is strongly discouraged in modern architectures, because:

- It results in complex, and self-referential code.
- It is difficult to debug, because after self-modification, the code no longer corresponds to the source code (!).

- It suffers performance penalties, because it causes the instruction cache to flush the affected lines: Modifying data is something that happens in the *data cache*. A situation where the same region in memory appears both in the data cache and in the instruction cache, and changes in the data cache, forces the affected lines in the instruction cache to be flushed and re-fetched, at great performance-penalty.

Nevertheless, self-modifying code is an important technique that is *essential* on some computer architectures (e.g., IBM Mainframe/HLASM/BAL assembly language), and that is interesting from and challenging from a programming perspective.

We would like to offer you the opportunity to become acquainted with self-modifying code in a controlled context, where the architecture is exceedingly simple, self-modifying code is essential, and no cache penalties are involved. This is the motivation for introducing the following Weird and *Wonderful* architecture developed by *Willem Louis van der Poel* in his PhD thesis.

2 Overview

The SIC computer architecture has only one instruction: `SBN A, B, C`. `SBN` stands for *Subtract & Branch if Negative*. This is an example of a *one-instruction computer*, of which many are known. SIC is not a realistic computer architecture, in the sense that it would be worth implementing it in hardware, but a theoretical one, in the sense that it is Turing-complete.

For this assignment, you will write the following tools:

1. A virtual machine for SIC, emulating the `SBN` instruction. This shall be written in x86 assembly language.

2. A second virtual machine for SIC, emulating the `SBN` instruction. This shall be written in SIC assembly language, shall only use the instruction `SBN`, and shall run on top of the virtual machine that you wrote in x86 assembly language. This program is essentially a bunch of numbers that we too could run on our own virtual machine for SIC.

You are given the code for computing Fibonacci numbers.

The finish-line for this assignment is to be able to run "a tower" consisting of Fibonacci running on top of (2) running on top of (2) running on top of (2), and so on, as many times as you like, running on top of (1). The ability to run such a "tower" of virtual machines will be our demonstration that your implementation of the virtual machine is correct.

3 A virtual machine for SIC

Below is a virtual machine for SIC, written in C:

```
#include <stdio.h>
#include <stdlib.h>

/* change to whatever you like: */
#define MAX_PHYSICAL_MEMORY 4096

static long M[MAX_PHYSICAL_MEMORY] = {
  ...SIC program encoding goes here...
};

int main()
{
  int i = 0;
  char *format;

  while (M[i] || M[i + 1] || M[i + 2])
    if ((M[M[i]] -= M[M[i + 1]]) < 0) i = M[i + 2];
    else i += 3;

  for (i = 0; i < MAX_PHYSICAL_MEMORY; ++i) {
    printf("%d ", M[i]);
  }

  printf("\n");

  return 0;
}
```

The program contains an array `M`, which models the *memory* of the machine. This is where your SIC code and data go. This code is just a bunch of integer values, separated by commas.

The `while`-loop is the entire virtual machine (VM):

- The index `i` is the *instruction pointer*. We decided *arbitrarily* that it always starts at address 0.
- We decided *arbitrarily* that the instruction `SBN 0, 0, 0`, which would not be very useful in any SIC program, would denote the *termination*

of the virtual machine. This means that `SBN 0, 0, 0` can be thought of as a *halt* instruction or `HLT` in the Intel x86, and terminates SIC.

- Regarding the instruction `SBN A, B, C`, the expression `M[i]` in the `while`-loop is the *address* of `A`. So `M[M[i]]` is the *value* of `A`. The same goes for the *value* of `B` being `M[M[i + 1]]`, and the *value* of `C` being `M[M[i + 2]]`.

- The semantics of the instruction `SBN A, B, C` is given in the C programming language as: `if ((A -= B) < 0) goto C`. If the test *fails*, execution proceeds with the subsequent instruction. Since each instruction is 3 words long, the next instruction will be at `i + 3`.

After execution of the SIC VM terminates, the contents of memory is printed out.

- Your implementation of the SIC VM in x86/64 assembly language should print the contents of memory after the VM has terminated.

- Your implementation of SIC VM in SIC should not (cannot) print the contents of memory, because there are no printing facilities in SIC, so execution just terminates. **This is the only difference between the SIC VM written in x86 assembly and the SIC VM written in SIC**.

4 How to write the SIC VM in x86

- Your executable should be named `sic`.

- You may use the following standard functions from the C libraries: `printf`, `scanf`, `malloc`, `calloc`, `free`, `exit`.

- The program should *read* from `stdin` as many integers as possible, and after the *end-of-file* has been reached, and n integers have been read, the program should *allocate* an array of n *quad-words* using `malloc`, and deposit the n words it read into this array, which shall serve as the *memory* for the SIC VM.

  - Your program will need some memory within which to operate. The number n is the total number of words your program needs, both for code and data.

- The VM should then start with the instruction at address 0.

- Upon termination of the VM, your program should print a *memory dump* of the SIC VM, meaning, the array of n numbers should be printed, with a single space char after each number, as in the above C code.

Writing the SIC VM in x86 is actually as easy and as straightforward as writing it in C. This won't be a very long program.

5 How to write the SIC VM in SIC

- The SIC VM in SIC does no input or output. It assumes that the program it is interpreting appears *immediately following* its own code. For example, my own implementation of the SIC VM in SIC takes up 258 words. I don't know if this is much of little, but this means that my SIC VM starts executing code at word 258 (since its own code occupies words 0..257).

- **Tricky:** Every SIC program is written under the assumption that it starts at address 0. This includes both the SIC VM written in SIC as well as the program that is running on top of it! The SIC VM should be written in a way that permits the program it is interpreting to make this assumption. This assumption is **essential** for us to be able to run any number of virtual machines on top of each other, forming a *tower of VMs*.

6 Computing Fibonacci numbers in SIC

The following code computes the n-th Fibonacci number (where `n` is initialized to 10), and writes the answer in address 0. The listing presents the address, starting at 0, the instructions, as I'd written them symbolically, and the *resolved instructions* after the addresses of all the labels & variables have been computed and replaced in the original program:

Address	**Label**	**Source Instruction**	**Resolved Instruction**
0	lstart:		
0		SBN n, w, lexit	SBN 56, 54, 36
3		SBN t, t, 6	SBN 55, 55, 6
6		SBN t, a, 9	SBN 55, 52, 9
9		SBN t, b, 12	SBN 55, 53, 12
12		SBN a, a, 15	SBN 52, 52, 15

15		SBN r, r, 18	SBN 51, 51, 18
18		SBN r, b, 21	SBN 51, 53, 21
21		SBN a, r, 24	SBN 52, 51, 24
24		SBN b, b, 27	SBN 53, 53, 27
27		SBN b, t, 30	SBN 53, 55, 30
30		SBN t, t, 33	SBN 55, 55, 33
33		SBN t, w, lstart	SBN 55, 54, 0
36	lexit:		
36		SBN 0, 0, 39	SBN 0, 0, 39
39		SBN r, r, 42	SBN 51, 51, 42
42		SBN r, a, 45	SBN 51, 52, 45
45		SBN 0, r, 48	SBN 0, 51, 48
48		SBN 0, 0, 0	SBN 0, 0, 0
51	r:		
51		*.data-word* 0	*.data-word* 0
52	a:		
52		*.data-word* 0	*.data-word* 0
53	b:		
53		*.data-word* 1	*.data-word* 1
54	w:		
54		*.data-word* 1	*.data-word* 1
55	t:		
55		*.data-word* 0	*.data-word* 0
56	n:		
56		*.data-word* 10	*.data-word* 10

The following numbers form the initial memory configuration of a SIC program that computes the 10th Fibonacci number (55). You can obtain these numbers by reading them off of the column for the *resolved instructions*:

```
56, 54, 36, 55, 55,  6, 55, 52,  9, 55,
53, 12, 52, 52, 15, 51, 51, 18, 51, 53,
21, 52, 51, 24, 53, 53, 27, 53, 55, 30,
55, 55, 33, 55, 54,  0,  0,  0, 39, 51,
51, 42, 51, 52, 45,  0, 51, 48,  0,  0,
 0,  0,  0,  1,  1,  0, 10
```

To change the argument to the Fibonacci function, you need to change the **last** number in the list.

7 So what should you do

1. Save the numbers that correspond to the Fibonacci function in the file `fib.sic`. You will have to remove the commas that separate the numbers.

2. Try to trace them out by hand, preferably for a smaller argument, to make sure you understand how the code works. This should give you a head-start in thinking about how to program in SIC.

3. Implement the SIC architecture in assembly language. Name your file `sic.s`, and provide a *makefile* to build the executable `sic`.

4. Verify that your SIC VM works correctly. For example:

   (a) Set the last number in the `fib.sic` file back to 10, and at the shell prompt type `cat fib.sic | ./sic` and verify that the first number printed is 55, which is the 10th Fibonacci number.

   (b) Set the last number in the `fib.sic` file to 20, and re-type `cat fib.sic | ./sic`. You should now get a list of numbers the first of which is 6765, which is the 20th Fibonacci number.

   You may now assume that your SIC VM in x86 works correctly.

5. Implement the SIC VM in SIC. This is the crux of the assignment, and it is here that things should become difficult. Once you are done, you should have a bunch of numbers that you will place in the file `sic.sic`. Let us assume that the size of `sic.sic` is Σ words.

6. To test your code, change the number in `fib.sic` back to 10. We already know that the expected value is 55. You can therefore do type at the shell prompt `cat sic.sic fib.sic | ./sic`. As before, you should get a memory dump that has the number 55 at address $\Sigma + 55$, because address Σ is address 0 from the standpoint of the programming running on top of the VM implemented in `sic.sic`! Change the last number in `fib.sic`, and verify again that you get the correct answer where you expect it.

7. To torture-test your code, try running a tower of SIC VMs using:

   ```
   cat sic.sic sic.sic sic.sic fib.sic | ./sic
   ```

and make sure you get the expected result as the first digit. As the tower grows taller, you should see *a slowdown that is exponential to the number of VMs*. This is normal and expected. The answer should appear at address $\Sigma \cdot n$, where n is the height of the tower, i.e., the number of virtual machines running on top of each other. If you have reached this point, congratulations! Your work is a success!

8 What you need to submit

- A *zip* file containing the following files:
  - `sic.s` containing the SIC VM in x86/64 assembly language
  - `makefile` to build the `sic` executable on linux
  - `sic.sic`, which is a file of integers separated by a single space, and which implements the SIC VM in SIC
  - `README.txt`, which is a text file that contains the following information:
    1. The names and IDs of all the people who worked on this assignment. There should be either your own name, or your name and that of your partner. You may only have one partner for this assignment.
    2. The following statement:

       I (We) assert that the work we submitted is 100% our own. We have not received any part from any other student in the class, nor have we give parts of it for use to others. Nor have we used code from other sources: Courses taught previously at this university, courses taught at other universities, various bits of code found on the internet, etc. We realize that should our code be found to contain code from other sources, that a formal case shall be opened against us with *va'adat mishma'at*, in pursuit of disciplinary action.

    We shall not grade your homework assignment without having received the above statement.

9 How we shall test your assignment

We shall be testing your `sic` program pretty much the same way we suggested that you test it yourself:

- We shall make sure it runs `fib.sic`
- We shall try it on other, more complex SIC code
- We shall then try to run code on VM towers of varying heights