

Homework 2

CS Osvita

December 20, 2025

As software engineers, we study computer systems to understand how our programs run. Our immediate reward is to be able to write faster, more memory-efficient, and more secure code. In the long term, the value of understanding computer systems may be even greater. Every abstraction between us and the hardware is leaky to some degree. This homework aims to provide a set of first principles from which to build sturdier mental models and reason more effectively.

Specifically, you will write a program that simulates the behavior of computer hardware. Our primary objective is to understand the [fetch-decode-execute cycle](#) of a physical computer by simulating one.

In addition, we hope you start to appreciate how virtual machines such as the CPython interpreter, Java Virtual Machine, or V8 JavaScript engine work.

The computer

To simulate computer architecture in the time available, we’ve created a fictional instruction set architecture. What follows is a description of the computer architecture that your virtual machine should simulate.

The device we’re simulating is much simpler than a modern CPU; it has:

- **256 bytes of memory**, simulated by an array with length 256
- **Three 8-bit registers**: 2 general purpose registers and 1 for the “program counter”
- **5 basic instructions**: load word, store word, add, subtract, and halt
- As a stretch goal, **4 more instructions**: add immediate, subtract immediate, jump, and branch if equal to zero

Memory

For the computer in this exercise, main memory is 256 “bytes”. We will use a fixed size array to model this. Here is one way to picture our memory:

```

00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f ... ff
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

```

All of the instructions and all of the data for a particular “program” must fit within these 256 bytes “bytes”.

Our computer follows a special convention for organizing memory:

- Data is stored in the first 8 bytes
- Instructions are stored immediately after the data
- Any program output is expected to be written to byte zero

This can be pictured as so, where XX indicates the output location:

```

00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f ... ff
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
XX ^==OTHER DATA=====^ ^==INSTRUCTIONS=====^

```

Instruction Format

Our architecture has a very limited instruction set, with only 5 basic instructions. Each can be mapped to a specific byte value, which is how we’ll specify the instruction in byte code:

```

load    0x01
store   0x02
add      0x03
sub      0x04
halt     0xff

```

Other than halt, each of these instructions has “parameters” that need to be supplied:

```

load    r1 addr    # Load value at given address into given register
store   r2 addr    # Store the value in register at the given memory address
add     r1 r2      # Set r1 = r1 + r2
sub     r1 r2      # Set r1 = r1 - r2
halt

```

For simplicity, we’ve decided to represent each “parameter” with a single byte. This means that all the instructions except halt will take three bytes—and halt, just one—to encode. The reg parameters may only take one of two values, because our architecture only has 2 general purpose registers. We can choose any single-byte value to identify the registers; we’ve chosen the values 0x01 and 0x02 (reserving value 0x00 for the program counter).

Now we have enough information to write a program. The “assembly”:

```

load    r1 0x01    # Load data from byte 1 into register 1
load    r2 0x02    # Load data from byte 2 into register 2
add     r1 r2      # Add the two register values, store the result in r1
store   r1 0x00    # Store the sum into the output location
halt

```

When translated line by line to our “machine code”:

```

0x010101
0x010202
0x030102
0x020100
0xff

```

Remember that machine code values are ultimately binary; we only use a hexadecimal representation here to aid readability.

And, partially, into our visualization of memory:

```

00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f ... ff
__ __ __ __ __ __ __ __ 01 01 01 01 02 02 03 01 ... __
XX ^==OTHER DATA=====^ ^==INSTRUCTIONS=====^

```

By loading two values into locations 0x01 and 0x02, and running the VM with the entirety of memory as input, it should modify memory, writing the sum to location 0x00.

For instance, say the input to our program were the numbers 3 and 5:

```

00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f ... ff
__ 03 05 __ __ __ __ __ 01 01 01 01 02 02 03 01 ... __
XX ^==OTHER DATA=====^ ^==INSTRUCTIONS=====^

```

After running the VM, the state of memory should be:

```

00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f ... ff
08 03 05 __ __ __ __ __ 01 01 01 01 02 02 03 01 ... __
XX ^==OTHER DATA=====^ ^==INSTRUCTIONS=====^

```

The exercise

Write a “virtual computer” function that takes as input a reference to “memory” (an array of 256 bytes), executes the stored program by fetching and decoding each instruction until it reaches *halt*, then returns. This function shouldn’t return anything, but should have the side-effect of mutating “main memory”.

Your VM should follow the fetch-decode-execute format, which you can model in a loop. The program counter should always contain the address of the “next instruction” (and so should start at 8). Fetch the current instruction by getting all the relevant information from memory, decode the instruction to find out what operation should be performed using which registers/memory-

addresses, then execute the instruction and update the program counter appropriately.

Your virtual computer should have one piece of internal state, an array of three registers: two general purpose registers and a program counter. Main memory is considered external state because it is passed in as an argument. The provided test suite will test your program against a number of such sequences of bytes.

Stretch goals

If you have time, attempt to implement four additional instructions with the following opcodes:

addi	0x05
subi	0x06
jump	0x07
beqz	0x08

addi and *subi* are the “immediate” versions of *add* and *sub*. Instead of adding or subtracting the value stored at a memory location, they add/subtract the value stored as the second argument. E.g. *addi* r1 4 will result in $r1 = r1 + 4$

jump has a single argument: the address to which the program counter should be set. It is absolute, i.e. “jump 40” will cause the program to continue from the 40th byte in memory. In practice, *jump* is used for such things as function calls.

beqz has two arguments: a register and a relative offset amount. If the value in that register is zero, then the program counter should be increased or decreased by that offset amount (in addition to any increase due to the *beqz* instruction being processed). Most instruction set architectures support a variety of such “conditional branch” instructions, which are used for such things as conditional statements, switch statements and loops.

Further Resources

There are many additional resources that you might find useful to better understand the execution of a computer at a very high level, as we are aiming to do in the first unit of the course. Two shorter resources in particular that we recommend are [Richard Feynman’s introductory lecture](#)(1:15 hr) and the article [How Computers Work: The CPU and Memory](#)(try VPN if link doesn’t work). The first is very conceptual; the second is more concrete. Both are useful angles.

Several books provide a good high-level introduction to how computers work: a popular one is [Code](#) by Charles Petzold, another is [But How Do It Know](#) by J Clark Scott.

For those looking for an introduction to computer architecture from a more traditional academic perspective, we recommend (Computer Organization and Design)P&H chapters 1.3-1.5, as well as [this 61C lecture](#) from 55:51 onward.

If you'd like other comparable examples of writing a simple VM, see the article [Write your Own Virtual Machine](#) as well as [A Python Interpreter Written in Python](#) from 50 Lines or Less.

The main text for this class is Computer Systems: A Programmer's Perspective ("CS:APP"). Please note that the cheaper "international" version of the book has different exercises and often has incorrect solutions.

For a more hardware-focused supplementary text, we suggest Computer Organization and Design by Patterson and Hennessy ("P&H") — a classic text commonly used in undergraduate computer architecture courses. P&H is one of the most successful textbooks in all of computer science. The authors received the 2017 Turing Award for their pioneering work on RISC (reduced instruction set computing). David Patterson now works as a researcher at Google after 40 years as a Professor at UC Berkeley, and John Hennessy was most recently President of Stanford before becoming Chairman of Alphabet. Chapter references are for the [5th edition](#), but older editions should be close in content.