SVC16: A Simple Virtual Computer

1 Motivation and Goals

The goal is to recreate the feeling of writing games for a system with very tight hardware constraints without dealing with the complicated reality of real retro systems. Understanding every instruction, writing machine code that runs on it, and writing a compiler for it should be simple. The instruction set is in no way meant to resemble something that would make sense in real hardware. This is primarily because this is aimed only at emulation but also because it encourages different designs when compiling to this architecture. It is also not intended to be as simple and elegant as it could possibly be. This might make it easier to emulate but harder to develop for. Since learning about assemblers and compilers is the point, we provide no guidelines on how to build complex programs.

1.1 Reproducibility

The biggest secondary goal is to design a system that behaves the same everywhere. The question of how the emulation is run should never matter to the person writing the program or game. This means there can be no features that might only be available in one implementation. It also means that the performance characteristics must be the same. An emulator either runs the system at the intended speed or it does not.

2 General Principles

Every value is represented as an unsigned 16-bit integer. That includes numbers, addresses, colors, the instruction pointer, and the input. Booleans are represented as `u16` values as well: 0 for `false` and >0 for `true`. Whenever an instruction writes out a boolean explicitly, it is guaranteed to be represented as the number 1.

There are no registers, no built-in stack, or special sections of memory. Operations can be performed directly on arbitrary addresses. There is no separation between data and instructions.

All numerical operations that will appear in the instructions are wrapping operations. This includes manipulations of the instruction pointer. Division by zero crashes the program.

3 The Simulated System

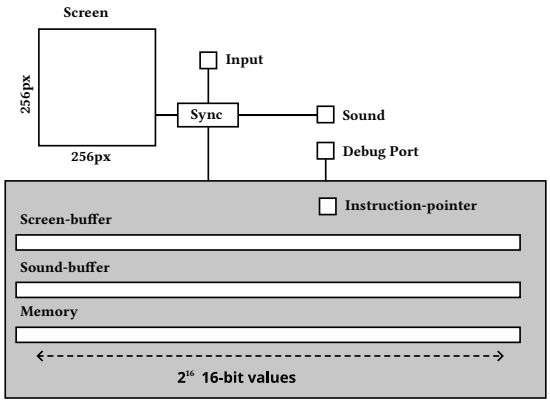

Figure 1: A sketch of all components of the virtual computer. The shaded area indicates what is visible to the simulation.

As seen in Figure 1, the addressable memory contains one value for each address. There is a separate screen buffer of the same size as the main memory and a sound buffer that is of the same size as well. The screen itself has a fixed resolution (256×256). The instruction pointer is stored separately. It always starts at zero.

Information is only transferred in and out of the simulated system when it is synchronized (see Section 3.4).

3.1 Screen and Colors

The color of each pixel is represented with 16-bits using `RGB565`. This means that a color code is given as $2^{11} * \text{red} + 2^5 * \text{green} + \text{blue}$, where the color channels go from zero to 31 for red and blue and 63 for green.

The coordinate (x, y) of the screen maps to the index $256y + x$ in the screen buffer. The coordinate $(0, 0)$ is in the upper left-hand corner.

Not specified

- Colors do not have to be represented accurately (accessibility options, artistic shaders).
- There is no rule for how scaling or filtering might be handled.
- It is not fixed what the screen shows before it is first synchronized.
- A cursor can be shown on the window, as long as its position matches the mouse position passed to the system. The default assumption should be that there is no external cursor, but it might not always be possible to hide it.
- There might be a delay before the updated frame is shown on screen. For example, one might need to wait for *vsync*, or the window takes time to update.

3.2 Sound

The sound buffer is only exposed to the outside world when it is triggered to play. That means that it can be used as additional memory when no sound needs to be dispatched.

When a sound is played, the data in the sound buffer is interpreted as a list of amplitude samples. The sampling rate is 16 kHz, so the total length is about four seconds. Each value in the buffer represents a sample (mono) and it is interpreted as a (two's complement) signed integer (i16). A new sound can be started every frame without stopping the previous ones. The sound buffer is left filled with zeros.

Not specified

There is no guarantee that the audio playback is perfectly synchronized with the rest of the system once a sound has been dispatched.

3.3 Input

The only supported inputs are the mouse position and a list of eight keys. These keys are supposed to represent the face buttons of an NES controller. The codes for the **A** and **B** keys also represent the left and right mouse buttons.

On synchronization, the new input is written to a specified memory address. This means that the input is inaccessible before the system is first synchronized. (See Table 2, Section 3.4)

The **position code** is the index of the pixel the mouse is currently on. It follows the same convention as the screen index explained in Section 3.1.

Bit	Controller Key	Mouse Key	Suggested Mapping
0	🅰	Left	**Space** / Mouse Left
1	🅱	Right	**B** / Mouse Right
2	⬆	-	**Up** / **W**
3	⬇	-	**Down** / **S**

Bit	Controller Key	Mouse Key	Suggested Mapping
4	⬅️	-	**Left / A**
5	➡️	-	**Right / D**
6	select	-	**N**
7	start	-	**M**

Table 1: The available input codes. We count from the least significant bit.

The **key code** uses bitflags. The bits in Table 1 are supposed to indicate if a key is currently pressed (and not if it was just pressed or released). As an example, if only ⬆️ and ➡️ are pressed, the key code is equal to the number $2^2 + 2^5 = 36$.

Not specified

- It is not guaranteed on which frame the virtual machine sees an input activate or deactivate.

3.4 Synchronization

When the console executes the **Sync** instruction, the screen buffer is drawn to the screen. It is not cleared. The system will be put to sleep until the beginning of the next frame. The targeted timing is 30fps. There is a hard limit of 3000000 (three million) instructions per frame. This means that if the Sync command has not been called for 3000000 instructions, it will be performed automatically. This can mean that an event (like a mouse click) is never handled. An alternative way to describe it is that the syncing happens automatically every frame and the instructions each take $\frac{1}{30*3000000}$ seconds. Then the **Sync** command just sleeps until the next frame starts.

4 Instruction Set

All instructions are 4 values long. A value is, of course, a `u16`.

The instructions have the form `opcode arg1 arg2 arg3`.

All instructions are listed in Table 2. `@arg1` refers to the value at the memory address `arg1`. If the opcode is greater than 15, the system will abort.

Opcode	Name	Effect
0	**Set**	`if arg3{ @arg1=inst_ptr }else{ @arg1=arg2 }`
1	**GoTo**	`if(not @arg3){ inst_ptr=@arg1+arg2 }`
2	**Skip**	`if(not @arg3){ inst_ptr=inst_ptr+4*arg1-4*arg2 }`
3	**Add**	`@arg3=(@arg1+@arg2)`
4	**Sub**	`@arg3=(@arg1-@arg2)`
5	**Mul**	`@arg3=(@arg1*@arg2)`
6	**Div**	`@arg3=(@arg1/@arg2)`
7	**Cmp**	`@arg3=(@arg1<@arg2)` (as unsigned)
8	**Deref**	`@arg2=@(@arg1+arg3)`
9	**Ref**	`@(@arg1+arg3)=@arg2`
10	**Debug**	Provides `arg1,@arg2,@arg3` as debug information
11	**Print**	Writes `value=@arg1` to `index=@arg2` of buffer `arg3`
12	**Read**	Copies `index=@arg1` of buffer `arg3` to `@arg2`.
13	**Band**	`@arg3=@arg1&@arg2` (binary and)
14	**Xor**	`@arg3=@arg1^@arg2` (binary exclusive or)
15	**Sync**	Puts `@arg1=position_code`, `@arg2=key_code` and synchronizes (in that order). If arg3!=0, it also triggers the sound buffer to be played.

Table 2: The instruction set.

Every instruction shown in Table 2 advances the instruction pointer by four positions *after* it is completed. The exceptions to this are the **GoTo** and **Skip** instructions. They only do this, if the condition is *not* met.

When an argument refers to the name of a buffer, it means the screen buffer if it is 0 and the sound buffer otherwise.

4.1 The Debug Instruction

The **Debug** instruction is special, as it does not change anything about the state of the system. It still counts as an instruction for the maximum instruction count. It is up to the implementation if, when, and in what way the information is provided to the user. This means that it is valid to not do anything when the instruction is triggered. This might be necessary to run the emulator at the intended speed.

The way to think about the signature of the instruction is that the first argument is a label and the other arguments are the values of variables/addresses.

The instruction is meant only for debugging. For the programmer that means that the output should not be needed for the use of the program or game as it might be shown in different ways or not at all. For the emulator that means that there should be no functionality that depends on the **Debug** instruction.

5 Constructing the Program

A program is just the initial state of the main memory. There is no distinction between memory that contains instructions and memory that contains some other asset. The initial state is loaded from a binary file that is read as containing the (little-endian) u16 values in order. The maximum size is $2 * 2^{16}$ bytes ≈ 131.1 kB. It can be shorter, in which case the end is padded with zeroes. The computer will begin by executing the instruction at index 0.

6 Handling Exceptions

There are only two ways the program can fail (for internal reasons).

- It tries to divide by zero.
- It tries to execute an instruction with an opcode greater than 15.

In both cases, the execution of the program is stopped. It is not restarted automatically. (So you cannot cause an error to restart a game.) There is intentionally no way of restarting or even quitting a program from within.

Not specified

- There is no rule for how (or even if) the cause of the exception is reported.
- It is not guaranteed that the emulator itself closes if an exception occurs. (So you cannot use it to quit a program.)

7 Example Program

Our goal could be to print all 2^{16} possible colors to the screen. We make our lives easier by mapping each index of the screen buffer to the color which is encoded with the index. Here, we use the names of the opcodes instead of their numbers.

```
// Write the value 1 to address 501
Set 501 1 0
// Write the largest possible value to 502
Set 502 65535 0
// Display color=@500 at screen-index=@500
Print 500 500 0
// Increment the color/screen-index
Add 500 501 500
// See if we are not at the max number and negate it.
Cmp 500 502 503
Xor 503 501 503
// Unless we are at the max number,
// go back 4 instructions.
Skip 0 4 503
// Sync and repeat.
Sync 0 0 0
GoTo 0 0 0
```

We could rely on the fact that the value at index 500 starts at zero and we did not have to initialize it.

To build a program that we can execute, we could use python 🐍:

```
import struct
code = [
    0, 501, 1, 0, #Opcodes replaced with numbers
    0, 502, 65535, 0,
    11, 500, 500, 0,
    # ...
]
with open("all_colors.svc16", "wb") as f:
    for value in code:
        f.write(struct.pack("<H", value))
```

Inspecting the file, we should see:

```
➜ hexyl examples/all_colors.svc16 -pv --panels 1

  00 00 f5 01 01 00 00 00
  00 00 f6 01 ff ff 00 00
  0b 00 f4 01 f4 01 00 00
  03 00 f4 01 f5 01 f4 01
  07 00 f4 01 f6 01 f7 01
  0e 00 f7 01 f5 01 f7 01
  02 00 00 00 04 00 f7 01
  0f 00 00 00 00 00 00 00
  01 00 00 00 00 00 00 00
```

Every line represents one instruction. The second column is zero because it is the most significant byte of the opcode.

When we run this, we should see the output shown in Figure 2.

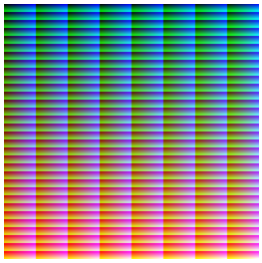

Figure 2: Output of the color example.

Can you figure out why the program crashes if a button is pressed? How could this be fixed?