Documentation - Sourcery

Sanjay Sankaran

Sourcery

Sourcery is an Assembly inspired Interpreted language, where program memory is the course code itself. for example,

```
02 ff << $01
```

Prints out the value at address `01` (the first two bytes are ignored here, more on that later). since the source code itself is the program memory, the value at `00` is `01`, since it is the `0`th byte.

The Sourcery Interpreter can be invoked through the command line: `java sourcery.main yourfile.sc`

Memory

While Sourcery is an Assembly inspired language, it also takes advantage of being an interpreter.

Each "byte" of a program can hold a value from `-0x3ff` to `0x3ff`. All of the source code is typed in hexadecimal notation, except for a few instructions which have aliases, such as `<<`, `++` and `0?`.

The `$` acts as a `"valueof"` operator, similar to unary `*` in C/C++. Thus, `$x` represents the value that is stored at the location `x`. In the example form the previous section, we can see that `$00` is printed to the screen, with a value of `1`.

Also, the program counter is stored in the `0`th byte. Upon loading a program, the program counter is set to the first byte of the program - for example, if the first byte of a program was `A7`, execution will start from address `0xA7`. The program counter is not read-only, so writing a value to `$00` will cause a jump to the written value.

Since Sourcery does not use variables with variable names, specific memory addresses are used to store and access values in a program. for example, instead of declaring a variable `i`, you could just use `$05` or any other address of your choosing.

Instructions

So far, the instructions are as follows:

`<< x` , Prints out the value at `$x` in hex. (the opcode for `<<` is `D0`)

`<- x` , Prints out the value at `$x` as an ascii character. (the opcode for `<-` is `D1`)

`++ x y` , Increments the value at `$x` by `y`. (the opcode for `++` is `D2`)

`-- x y` , Decrements the value at `$x` by `y`. (the opcode for `--` is `D4`)

`== x y` , Sets the value at `$x` to `y`. (the opcode for `==` is `D3`)

`0? x y` , if the value at `$x` is `0`, set it to `y`. else set it to `0`. (the opcode for `0?` is `D5`)

More instructions, including user input, will be added in the future. When user input is included, Sourcery can team up with other unix-cli applications, such as `sh`, `aplay` and `netcat` through pipes, to provide more functionality such as audio output and networking.

Macros

The macro system Currently uses the following:

`{LABEL something}` sets a label wherever it is declared. the position of this label can be acessed using `{$something}`. for example, a `{LABEL some_variable} 05` will replace all `{$some_variable}` with the address of the aforementioned `05`. this is useful for setting keypoints in memory for pointer-like access or for clear jump locations.

A `{DEF somename value}` can also be used, and will replace all `{somename}` with the value. here, the `value` can be anything, and even contain more code.

The `{TEXT sometext}` will replace the region with the `sometext` in hexadecimal form. for example, `{TEXT SUS}` will result in `53 55 53`.

The `{! something here}` macro can be used as a comment.

Hello World

A simple and easy equivalent of hello world would be

```
01 << ff
```

This should print out `"ff"` to the console.

A bad implementation of Hello World can be done as:

```
01
<- 48
<- 65
<- 6c
<- 6c
<- 6f
<- 20
<- 57
<- 6f
<- 72
<- 6c
<- 64
<- 0a
```

This just prints out the Ascii characters in "Hello World\n"

A Better implementation would be:

```
03 {$START} 00

<- $$01
++ 01 01

== 02 {$END}
-- 02 $01

0? 02 -4
++ 02 03
== 00 $02

{LABEL START} {TEXT Hello World} 0a {LABEL END}
```

This implementation uses a loop to print out the characters in `{TEXT Hello World\n}`, and exits once it ends.

Stacks

The implemtation of a stack can be useful for memory allocation, and for subroutines. A basic stack is implemented as follows:

```
{$prog}

01 {! let $01 hold the stack pointer, also let its initial value be 01.}

00 00 00 00 00 {! an array of 5 slots/spaces, to be used as a stack}

{LABEL prog}

{! pushing to the stack}
++ 01 01  {! increment the stack pointer ($01) by 1}
== $01 ff {! set the value at the stack pointer ($$01) to ff}

++ 01 01  {! pushing `aa` next}
== $01 aa

{! popping from the stack}
<< $$01  {! print out the value pointed by the stack pointer}
-- 01 01 {! decrement the stack pointer ($01) by 1}

<< $$01  {! pop a second time}
-- 01 01
```

Output:

```
user@host:~/path$ java sourcery.main path/to/file/program.sc
AA
FF
```

This implementation can be used to push the program counter ($00) onto the stack and pop it back, allowing jumps to and from subroutines.