

1 General

The VM features a very basic assembler capable of little more than adress resolution. But yet it gives us an ability to create some neat little programs.

2 Syntax

The syntax for the assembly code is pretty straight forward. Each declaration is written on a single line. There are a few reserved identifiers:

Identifier	Name	Description
%<text>	Comment	Will be ignored by the assembler
#<name>	Label	Declares a label called <name>
@<name>	Value	Declares a value called <name>
:<data>	Raw input	Returns <data> as is
\$	Address	Treats a as a address
x	x	The X register
y	y	The Y register
s	s	The stack S
q1	IRQ1	The Q₁ register
q2	IRQ2	The Q₂ register

The names given to labels and values can contain any characters except for whitespace ones.

Operations are declared in a straightforward approach as:

<opcode> <arg1> <arg2>

Wich arguments are allowed are dependent upon the opcode.

3 Usage

3.1 Registers

The usage of the registers is pretty straight forward. One has to remeber that q1 and q2 are write only registers and that **s** cant be used for addressing so **\$s** is not allowed and will generate a **syntax** error. It is also good to keep in mind that all operations reading from the stack will consume what is on top of the stack.

3.2 Pointers

Using pointers is fairly straight forward. Alltough one has to keep in mind how the addresses are resolved. All pointers will be resolved after the tokenazation fo the code. First the labels will be resolved and then the values. This means that the first value will lie after the last line of code. Since the address of a label depends on where in the code their addresses are easy to reason about.

However for values things are bit different. Since values will be given addresses which are “independent” of where in the code they appear it is hard to reason about the address of a value. Although the value pointers are resolved in order the first value declared will lie intermediately after the last line of code and the last value declared will lie “at the end” of the memory used by the program. This can be exploited to use relative addressing. Although great care has to be taken.

It's important to remember that all pointers are referred to throughout the entire program therefore it's not allowed to define two pointers with the same name. If this were to be allowed it would generate unpredictable behaviour so instead the assembler will return a **assembler** error.

Labels and values are interchangeable. Since opcodes take pointers as arguments and has no idea whether or not they are labels or values. From this the need for caution arises. Since one can use value pointers as arguments to jump operation like this:

```
@bad_idea
ADD x y
MOV s x
MUL x y
JMP bad_idea
```

Since it is not known what where **bad_idea** points jumping to it is suicidal.

Since pointers are just numbers under the hood one needs to take into account whether or not one uses them for their address or for their values. Here are some examples

```
@pointer
% This stores x in pointer
MOV x value
% This stores x in the address which is
% stored at pointer
MOV x $value
% This adds one to the value stored at
% pointer
ADD $pointer 1
% This adds one to the address of pointer
ADD pointer 1
```

Pointers are immutable and once they have been declared they can not be changed. One has to do some tricking to achieve relative addressing using labels or values.

3.2.1 Labels

Labels are declared using the **#** identifier. Labels are resolved first and their addresses correspond to location in the code where they are written. For example:

```

MOV x y
#loop
INC x
MOV x s
JMP loop

```

In this code `loop` points to the address where `INC x` is stored. In the tokenization of the assembly code the lines where a pointer is defined will be ignored and the address where the next instruction or raw entry occurs. This can lead to that poorly written code becomes ambiguous. For example:

```

MOV x y
#loop
#silly
INC y

```

Here `loop` and `silly` will both point to the same address which is `silly`.

Because tokenization of the code happens before the address resolving a label will be “in scope” throughout the entire code. So this code is perfectly valid:

```

MOV x y
JMP ahead
INC x
ADD x y
#ahead
ADD s x

```

The `JMP ahead` will jump to `ADD s x` even though the `ahead` flag is defined after the jump. This was not a conscious design choice but it is actually quite useful since one can define subroutines anywhere in the code which can be accessed from anywhere in the code.

One possible pitfall arises due to the fact that the assembler does not know the difference between a label and a value after their addresses have been resolved. So this code is valid assembly code:

```

#loop
ADD s x
MOV s $loop
JMP loop

```

Although what this will do is that it will change what is at the address of `loop`. But there `ADD s x` lies! This is what is known as self-modifying code and it's the spawn of Satan and should be avoided like one avoids Miami Beach during spring break. Although in some cases the interchangeability of value and label can be very useful if one wants to have “arrays” in one's code. This is easily achieved like this:

```
#array
:0
:1
:2
:3
```

Here `array` can be used as a pointer to the array. One can then manipulate the array through using relative addressing of `array` like this:

```
ADD 2 array
MOV s y
MOV 5 $y
#array
:0
:1
:2
:3
```

This code would change the 2 into a 5. But great care needs to be taken since one could easily end up outside of the “array” and corrupt the program.