

## vov

### Name

**vov** — Emulate a Von-Neumann Machine

### Synopsis

```
vov [-h] [-i] [-m size] [-q] [-t secs] [-v] [-V] [program_file]
```

### Description

**vov** (Vov's Obsessive Von-Neumann) is a tool that emulates the behavior of a Von-Neumann machine. It is basically an interpreter, which reads files in the form of memory assignments and executes the encoded instructions. The **vov**'s instructions make you able to perform simple arithmetic data manipulation. It is a very useful tool to see if your programs work and how.

I am a strong believer in Open Source, so I encourage you to send feedback, updates, patches et cetera. Do not hesitate to inform me of typos or plain old error. If my English sounds somewhat wooden, please realize that I'm not a native speaker. Feel free to send suggestions.

### Options

-h

List all options, with brief explanations.

-i

Run in interactive mode; this option will be ignored if you specify a *program\_file* in the command line. When **-i** is in effect, **vov** starts with a bootstrap program that read opcodes from the input device (usually your keyboard), store them in RAM and, when you type the *halt* instruction, run your code.

**Note:** The bootstrap program is located at the end of the machine's RAM, therefore it don't ask you the starting offset where to place your code (it assumes 0), and you are free to write your code without count the "offset slide" (i.e. when jump). The bootstrap code uses 13 words of memory; it's possible to make it more small without insert a "want to survive" feature that don't allow you to overwrite the bootstrap code (when **-i** is in effect you don't have all RAM available).

-m *size*

Limit the maximum memory used by **vov** to *size* bytes. If you don't set this option explicitly the default value will be used (2000000 bytes). This option may have no effects on your system.

-q

Run in quiet mode; do not print the introductory and copyright messages. These messages are also suppressed in batch mode.

-t *secs*

Limit the maximum cpu time used by **vov** to *secs* seconds. If you don't set this option explicitly the default value will be used (2 seconds). This option may have no effects on your system.

-v

Run in verbose mode; this will print lots of useful information on the output device saying what the machine is actually doing; useful for debugging purposes.

-V

Print version information and then exit.

### The Machine

Von-Neumann machine is a computer system organization actually realized and operated in a Princeton University Project, under control of the mathematician John Von-Neumann. The machine is realized through the interconnection of four complex devices: memory, control unit, input and output.

1. The memory device called RAM (Random Access Memory) realizes the storage of an array of integer numbers. Both the array size (the number of words) and the maximum storable value in each word are given at building or assembling time. As a virtual machine, it supports two basic instructions: stores  $x$  in word  $y$ , fetches the value previously stored in  $y$ .
2. The control unit realizes the functioning of the machine with a sequential working mode using the *PC* (a.k.a. Program Counter) register using the following algorithm:
  - *Initialization*: store the value 0 in the *PC* register.
  - *Fetch*: get the value held in  $RAM[PC]$  and store it in a third register called *IR* (a.k.a. Instruction Register), then increments *PC* by 1.
  - *Decode*: compare the value held in *IR* with the symbols table to find out which instruction is encoded to that number.
  - *Execution*: execute the instruction encoded by the value held in *IR*; at the end go back to fetch unless it was the *halt* instruction.

Applying this simple algorithm, the control unit is able to execute sequential programs made of instructions coded in numeric form in RAM words starting from offset 0. That is,  $RAM[0]$  holds the encoding of the first instruction of the program,  $RAM[1]$  the second and so on. As we can see, the control unit fetches and decodes instructions only basing itself on the value stored in the *PC*, and it cannot know if the value in the *IR* was inserted in  $RAM[PC]$  to be used as code or data. It is the programmer who must make sure that the word from which the value is fetched actually holds the encoded instruction he wants to be executed at the point. A program which first inserts a datum in a memory word and then interprets that word as an instruction is said *self modifying* (like a bootstrap program).

3. The input device gives to the user the ability to interact with the machine (for example through a numeric keyboard) to enter an integer value, one at a time.
4. The output device gives to the machine the ability to print a readable format of an integer value at a time.

In this Von-Neumann's machine implementation, we have a RAM with 1000 words (each word is a 32-bit signed integer) and each word can be indexed by a number between 0 and 999. We use the notation  $RAM[0]$  to identify the first RAM word. The control unit is realized in such a way that it can execute (directly or commanding other parts of the machine) a set of 9 basic instructions. Each instruction may be encoded by a single number between 0 and 8, no matter how complex the operation is. Some instructions are uniquely determined by the number,

while other need to know one more parameter (the  $x$  value for the memory offset) to find out the second operand. Instructions which need to know explicitly an operand must be coded with the explicit operand. A Von-Neumann's machine doesn't officially handle negative numbers therefore in this implementation such a thing as `RAM[x]=-y` (with  $y > 0$ ) gives an error. By the way it is actually possible in **vov** to use negative numbers with a trick: simply using the subtraction and an auxiliary word. It makes thus sense to have programs (or functions) that calculate the absolute value of a number (see `scripts/` directory), and the like.

## Instructions Set

A very simple way to obtain a numeric code for instructions is to list them in a sorted list, and to use the order numbers of the list to find the instructions, therefore a simple algorithm to obtain the numeric encoding of all the instructions, including those having an operand, is to multiply by 1000 the order number of the instruction and sum the possible operand (always between to 0 and 999, due to RAM size).

- `0`: sum of two integers. It adds the value contained in the accumulator with the value stored in `RAM[y]`; the result is stored in the accumulator. The first operand is lost.
- `1`: difference of two integers. It subtracts the value kept in `RAM[y]` to the value stored in the accumulator; the result is stored in itself. The first operand is lost.
- `2`: read of an integer number from the input device. The read value is stored in the accumulator. The previously stored value is lost.
- `3`: write of an integer value on the output device. The value stored in the accumulator is copied on the output device. No value is lost.
- `4`: stores the number held in the accumulator also in `RAM[y]`. Previous content of `RAM[y]` is lost; content of the accumulator is kept.
- `5`: copies the value held in `RAM[y]` in the accumulator. Previous content of the accumulator is lost. Content of `RAM[y]` is kept.
- `6`: usually called a *program jump*, as it inserts an interruption in the linear RAM scan: it makes the next instruction fetched from `RAM[y]` instead of the next instruction in RAM after the jump instruction.
- `7`: checks the value stored in the accumulator. If this value is 0, jump as *instruction 6*, otherwise don't do anything (a.k.a. *conditional jump*).
- `8`: halts the machine.

## vov program

You can run a **vov** program from a file as you would any other shell script, perl(1) program, python(1) program or ruby(1) program. You can simply run **vov** giving the script name as an argument; if no script name was given and `-i` isn't in effect, the standard input will be used.

A **vov** program must be written using the following syntax:

```
RAM[x] = yz;
```

where  $x$  is an address in the machine's RAM,  $y$  is an instruction opcode and  $z$  is the argument to the opcode (if the instruction require one, otherwise 0 should be used). Comments starts with the character `#` and go on up to the end of line. Blanks are ignored except inside tokens, and case is not important.

Since **vov** is actually a full-blown interpreter, you can use the Unix "shebang" notation as the first line of the program file (if your system supports it, you can avoid hard-coding the path to **vov** in the shebang line by using `#!/usr/bin/env vov`, which will search your path for **vov** and then execute it). If you make this source file executable (using, for instance, `chmod +x myprog.vnm`), Unix lets you run the file as a program. You can do something similar under Microsoft Windows using file associations.

## Examples

Here you can find some examples on how to invoke **vov**:

### Example 1. Run a script in verbose mode

```
$ vov -v foo.vnm
```

### Example 2. Run in verbose and interactive mode

```
$ vov -iv
```

### Example 3. Read the program from the standard input being verbose but don't show any banner

```
$ vov -vq
```

## Confirmed Platforms

**vov** should compile (and run) on any platform (it is nearly entire ANSI C) with any version of gcc(1); by the way there is a list of confirmed platforms. If you get it working on other platforms, please contact me.

### GNU/Linux

- Slackware 8.1
- Slackware 10.2
- Slackware 11.0
- KUbuntu 5.10 (breezy)
- KUbuntu 6.06 (dapper)
- Ubuntu 6.10 (edgy)

### BSD

- FreeBSD 6.0-RELEASE



**Note:** **vov**, internally, uses `malloc(3)` only for the RAM words (tree nodes), therefore things are simple: since the nodes are homogeneous data (same size) *if a hunk doesn't fit for an allocation then doesn't fit even for the next one*. Therefore when we need to allocate a new RAM word (tree node) we don't need (again) to scan the entire list but we can just check the last hunk (we have a pointer to it) for available bytes: that has practically no overhead. We don't need any best-fit, first-fit, et cetera algorithm implementations keeping the code (more fast and) **KISS (Keep It Simple, Stupid)**. This is very important because security holes can't show up in features that don't exist©.

For all reasons explained before (as minimize page and cache misses during program execution), hunk size is set to the size of system memory page. Size of memory page is architecture and operating system dependent. You can set it using the configure script. By default memory page size is set to 4096 bytes. This should ensure that no memory will be left unused; on my (32-bit) system, where a memory page is 4096 bytes, compiling the code using `gcc(1)`, the size of a list cell is 16 bytes and size of a tree node is 20 bytes therefore one hunk can hold up to 204 memory words filling itself perfectly. On a 64-bit system, where a memory page is 4096 bytes, compiling the code using `gcc(1)`, both the size of a list cell and a tree node are 32 bytes therefore one hunk can hold up to 127 memory words filling itself perfectly. Thus, on all cases, allocating one hunk is enough for almost all **vov** programs (just one `malloc(3)` for an average **vov** session).

#### Input/Output

- *I've mostly given up on the standard C library. Many of its facilities, particularly `stdio`, seem designed to encourage bugs©*; thus premier causes for bloat are `stdio` and the `printf` family of functions. Since **vov** was written even to be small, it don't use any of these functions. **vov**'s input/output routines are stolen (and modified) from `libowfat` a GPL version of the `djb` library code. Since I don't want any external and non standard dependencies and I modified the used routines I don't link **vov** against `libowfat` directly.

#### Parser

- The parser is realized using `flex(1)` and `bison(1)`. I made some little changes to the scanner generated by `flex(1)` because I don't want to use any `stdio` facilities (as `printf`, `FILE *`, et cetera) keeping the code small and not bug-prone (see the Input/Output section). If you have any problem to apply my patch after you regenerated the scanner, please note that I used an old version of `flex(1)` (2.5.4a) because it works for me and thus more recently versions of `flex(1)` insert some code (as the function `clearerr` that accepts a `FILE *` as parameter) that I can't modify.

**Note:** I also rewrote the `yy_fatal_error` routine because the generated version has some memory and descriptor leaks. The default function do not close the parsed `FILE *` (descriptor leak) and do not destroy the `yy_current_buffer` (memory leak).

#### Resource exhaustion

- **vov** don't want to eat all your shared resources as memory and cpu time (the only shared resources that **vov** will actually use). I believe that include separate code in every application to impose configurable artificial limits on every dynamically allocated data is a wrong solution to make a secure program (you can always fail putting an artificial limit on a dynamically allocated data) therefore I use *rlimits* on systems that support it.

**Note:** These limits are only for your convenience; you can't override system defined limits. If an user on a system sets a limit greater than the relative system limit then the system limit will be used.

If you think that arbitrary memory allocation up to *rlimits* or physical limits is a bug, please wait a second: systems that allows users to allocate any amount of memory are vulnerable before they are running **vov** because they aren't well managed systems and can be damaged by memory exhaustion attacks, whether or not they are running **vov**. Probably, on these systems, **vov** is the last problem of the system administrator. By the way, on UNIX like systems, we have the *rlimits* facilities.

## COPYING

Copyright © 2001-2008 Davide Scola <davide (dot) scola (at) gmail (dot) com>

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be included in translations approved by the Free Software Foundation instead of in the original English.