VAM
Virtual Assembler Machine
–
An Interpreter for a Simple Processor's Instruction Set

Dr. Jürgen Vollmer
Juergen.Vollmer@informatik-vollmer.de

6th April 2009
Version 1.2

Contents

1 DESCRIPTION

vam.pl is an interpreter of a simple processor's (designed by me and called **VAM**, Virtual Assembler Processor) instruction set. It has the "usual" machine instructions available on "real" processors.

vam.pl may be used e.g. in a compiler construction course to have a "play ground" when generating machine code.

vam.pl read the instructions to be interpreted either from *stdin* or the file(s) *srcfile* given on the command line. The *BIOS* (Basic Input / Output System) of the processor allows to read and write a file during runtime (see –**input** and –**output**).

2 FORMAT OF THE SOURCE FILE

The source file *srcfile* contains the assembler instructions to be interpreted.

A line is either empty, contains a comment or *one* instruction or *one* (string or data) declaration.

The # character starts a comment up to the end of the line.

The instruction and registers names are case insensitive. Labels and string literals are case sensitive.

3 ARCHITECTURE OF THE PROCESSOR VAM

The VAM processor has a usual *von-Neumann* architecture with registers, memory, an ALU (arithmetic logical unit) etc.

The processor knows two data types: (signed) integers and floating point numbers. Memory addresses are positive integer values.

The number of registers may be specified by the –**registers** command line option and are denoted by **R0**, **R1**, ... A register may hold *one* value of a given data type at a time.

The memory is partitioned into *memory cells*, which may hold *one* value of a given data type at a time.

3.1 Special Registers

The registers **R0**, **R1** und **R2** are reserved and should not be used for other purposes (even it is legal to modify them). All other registers may be used without any restrictions.

R0

The register **R0** is used as *instruction counter* and set to 1 before the program starts.

With the exception of the control flow and non-executable statements, after performing the operation specified by an instruction, the instruction counter **R0** will be incremented by 1.

R1

The register **R1** is used as stack pointer pointing to the top of the stack maintained by some instructions (see below). At program start **R1** is initialized to the value of *high-memory*.

R2

The register **R2** holds the pointer to the first unused memory cell. The the free memory may be used manage dynamically allocated memory similar to the *sbrk*(2) call of the C-library.

3.2 Layout of the Memory

```
 Address        | Description
----------------+--------------------------------------------------
    0           | Undefined address
----------------+--------------------------------------------------
    1           | The instruction to be executed at program start
   ...          | More instructions.
  <code-end>    | The textual last instruction
----------------+--------------------------------------------------
 <data-start>   | Global data
  ...           | ........
 <data-end>     | ........
----------------+--------------------------------------------------
 <malloc-start> | Start of free memory.
  ...           | ....
 <high-memory>-1| Last memory cell, initial top of stack
----------------+--------------------------------------------------
```

The free memory "grows" towards larger address, while the stack grows towards small addresses.

3.3 VAM Runtime Errors

The VAM emits under the following conditions an error message and terminates:

* VAM emits an error if the memory at address 0 or an address >= *high-memory* is accessed.

* VAM emits an error, if a memory cell not containing an instruction should be read as instruction and executed.

* Division by 0 causes an error.

VAM does not emit an error if the two memory areas overlap.

4 MACHINE INSTRUCTIONS

In the sequel the instruction are declared, which the processor is able to process. In the description below *r*, *r1*, *r2*, *r3* specify any processor register **R0**, **R1**, ..., **R***max_registers-1*, even the reserved ones. If for an instruction several registers may be specified, the same register may be used several times.

The notion **M**[*address*] specifies the access of the memory cell at the given *address*.

4.1 Labels

There are three kinds of named entities, named by so called *label*:

1. Data declaration:

   label **:** **DATA** <count>

2. String declaration

   label **:** **"**character string**"**

3. Instruction

   label **:** *one-instruction*

A *label* consists of a letter followed by letters, digits or an underscore character. The labels of each entity class (data, string, instruction) reside in a separate namespace and must be unique in that namespace.

The string and data declarations are non-executable instructions, therefore the instruction counter is not modified. They may occur everywhere in the source file.

Memory cells holding global data must be reserved and a label must be given be given. This declaration reserves *count* memory cells. The address of the first memory cell may be accessed using the name *label*.

A string declaration declares a name for a character string used in the **write_s** instruction. The two characters "\n" represent a newline. The two characters "\t" represent a tabulator.

A control flow statement may use an instruction *label* to specify the next statement to be executed.

Some instructions have a suffix in their name. The suffix **_f** indicates floating point and **_i** integer operation. The suffix **_c** indicates that the instruction takes a literal integer argument, **_r** a register is used and **_l** a label.

A prefix **i** indicates some kind of indirection.

4.2 Control Flow Instructions

nop

No Operation, do nothing and continue execution with the next instruction. If a label declaration is not followed by an instruction, a **nop** is assumed.

end

Stop the processor and terminate the interpretation.

goto ***label***

igoto ***r***

Continue execution with the instruction named by *label*, or at the address stored in register *r*.

call ***r1*** **,** ***label***

icall ***r1*** **,** ***r2***

Continue execution with the instruction named by *label* (or the address stored in register *r2*) and push the address of the statement following the **call** onto the stack pointed to by *r2*.

r1 = *r1* - 1; Memory[*r1*] = **R0** + 1; **R0** = *label*

r1 = *r1* - 1; Memory[*r1*] = **R0** + 1; **R0** = *r2*

return ***r***

Get address from the top of the stack *r*, pop it and continue execution at the instruction specified by that address.

R0 = Memory[*r*]; *r* = *r* + 1;

4.3 Conditional Control Flow Instructions

If register *r* holds the shown condition, then continue execution with the instruction specified by *label*, else continue with the textual next instruction.

The registers should contain an integer value.

See also **cmp_i** and **cmp_f**.

iflt ***r*** **,** ***label***

less then: $r < 0$

ifle ***r*** **,** ***label***

less equal: $r <= 0$

ifeq ***r*** **,** ***label***

equal: $r = 0$

ifne ***r*** **,** ***label***

not equal: $r != 0$

ifgt ***r*** **,** ***label***

greater then: $r > 0$

ifge ***r*** **,** ***label***

greater equal: $r >= 0$

iftrue ***r*** **,** ***label***

true-value: $r != 0$

iffalse ***r*** **,** ***label***

false-value: $r == 0$

4.4 Arithmetic instructions

Except when noted, all registers used by an instruction must hold values of the same datatype, which must match the type of the instruction. Otherwise the result may be undefined.

The processing of an instructions read the source registers performs the operation and store the result in the destination register.

There is no test over underflow or overflow of the result. A division by 0 causes an error.

add_i *r1* , *r2* , *r3*

add_f *r1* , *r2* , *r3*

Addition: *r1* = *r2* + *r3*

sub_i *r1* , *r2* , *r3*

sub_f *r1* , *r2* , *r3*

Subtraction: *r1* = *r2* - *r3*

mult_i *r1* , *r2* , *r3*

mult_f *r1* , *r2* , *r3*

Multiplication: *r1* = *r2* * *r3*

div_i *r1* , *r2* , *r3*

div_f *r1* , *r2* , *r3*

Division: *r1* = *r2* / *r3*

If *r3* == 0 an error message is emitted and the program is terminated.

mod_i *r1* , *r2* , *r3*

Modulo: *r1* = *r2* % *r3*

If *r3* == 0 an error message is emitted and the program is terminated.

Note: there is no floating point mod operation!

cmp_i *r1* , *r2* , *r3*

cmp_f *r1* , *r2* , *r3*

comparison: *r1* = *r2* compare *r3*

r1 holds an integer -1, 0, or 1 according to:

-1 if *r2* < *r3*

0 if *r2* == *r3*

1 if *r2* > *r3*

add_c *r1* , *r2* , *value*

sub_c *r1* , *r2* , *value*

mult_c *r1* , *r2* , *value*

div_c ***r1*** **,** ***r2*** **,** ***value***

mod_c ***r1*** **,** ***r2*** **,** ***value***

cmp_c ***r1*** **,** ***r2*** **,** ***value***

Integer arithmetic / comparison with an (signed) integer constant:

r1 = *r2* <operand> *value*

f2i ***r1*** **,** ***r2***

Type cast: *r1* = (int) *r2* (in C speak)

i2f ***r1*** **,** ***r2***

Type cast: *r1* = (float) *r2* (in C speak)

lshift ***r1*** **,** ***r2*** **,** ***r3***

rshift ***r1*** **,** ***r2*** **,** ***r3***

Left shift: *r1* = *r2* << *r3* (in C speak)

Right shift: *r1* = *r2* >> *r3* (in C speak)

All registers must hold integer values.

4.5 Transport Instructions

Transport instructions move values from one place to another.

If a register specifies an address, it must hold an positive integer value. Everything else is undefined.

copy ***r1*** **,** ***r2***

Copies the content of registers: *r1* = *r2*

cload_i ***r*** **,** ***value***

cload_f ***r*** **,** ***value***

Loads the (signed) integer or floating point *value* into the register *r*.

load ***r1*** **,** ***r2***

load_l ***r1*** **,** ***label***

load_c ***r1*** **,** ***r2*** **,** ***value***

Load the content a memory cell to a register. *label* is a data label and *value* is a signed integer.

r1 = Memory[*r2*]

r1 = Memory[*label*]

r1 = Memory[*r2* + *value*]

iload ***r1*** **,** ***r2*** **,** ***r3***

Indirect load with register offset: *r1* = Memory[Memory[*r2*] + *r3*]

iload_c *r1* , *r2* , *value*

Indirect load with constant offset: *r1* = Memory[Memory[*r2*] + *value*]

value is an (signed) integer number.

store *r1* , *r2*

store_l *label* , *r2*

store_c *r1* , *r2* , *value*

Store the content a register in a memory cell. *label* is a data label and *value* is a signed integer.

Memory[*r1*] = *r2*

Memory[*label*] = *r2*

Memory[*r1* + *value*] = *r2*

istore *r1* , *r2* , *r3*

Indirect store with register offset: Memory[Memory[*r1*] + *r2*] = *r3*

istore_c *r1* , *r2* , *value*

Indirect store with constant offset: Memory[Memory[*r1*] + *value*] = *r2*

value is an (signed) integer number.

4.6 Stack Related Instructions

Stack grows from higher to lower addresses.

push *r1* , *r2*

Push the value of *r2* to the stack pointed by *r1*.

r1 = *r1* - 1; Memory[*r1*] = *r2*

r1 should contain an address, while *r2* may hold any value.

pop *r1* , *value*

pop_r *r1* , *r2*

Pop *values* / *r2* many values from the stack.

r1 = *r1* + *r2*

r1 = *r1* + *value*

r1 should contain an address, while *r2* / *value* is an integer.

4.7 Input/Output Instructions

The VAM has a build in BIOS (basic I/O system) which allows to write character strings and numbers to a file (see –**output**). Only numbers may be read from a file (see –**input**).

##

4.7.1 Input

The following instruction read some values from the input file.

read_i *r1* , *r2*

read_f *r1*, *r2*

Read a (signed) integer / floating point number and store it in register *r1* and store an error code in *r2*. If the read value is an integer or floating point constant, then *r2* holds the value 1 (true), else 0 (false)

If *r1* and *r2* denote the same register, no error code is stored. In case of an error the value 0 is stored in the register.

The read number must be terminated by RETURN (newline).

eof *r*

Test if End Of File (EOF) has been reached: $r = \text{EOF}?\ 1 : 0$ (in C speak)

4.7.2 Output

The following instruction write some values to the output file.

write_s *label*

Write the character string denoted by *label*.

write_i *R*

Write the integer value stored in register *R*.

write_f *R*

Write the floating point value stored in register *R*.

5 SYNOPSIS

vam.pl [–help] [–manual | -M] [–verbose] [–Version] [–input infile] [–output outfile] [–registers count] [–memory size] [–dump | -D] [–statistics] [srcfile...]

6 OPTIONS

–input *infile*

Read the input of the interpreted program, from *infile*.

Default: *stdin*.

–output *outfile*

Write the output of the interpreted program, to *outfile*.

Default: *stdout*.

–memory ***size***

Size of the avaliable memory. *size* is a positive number followed by **k** (kilo) or **M** (mega), which specifies the size in kilo or mega memory cells respectively.

Default: 32M

–registers ***count***

Number of the available registers: **R0**, **R1**, ... **R***count-1*.

Default: 32

–help

Show this help and terminate.

–manual -M

Show the entire manual and terminate.

–verbose

Verbose.

–Version

Show program version.

–dump

-D

After parsing the source file, dump the instruction list.

–statistics

Emit some statistical information about the program.

Option names may abbreviated as long as the are unique. Instead of –XX -XX may be used.

7 EXAMPLES

The following program prints the square numbers 1 ... n^2:

```
###############################################################
# Description:  A VAM program computing square numbers
#               Input:   read a number
#               Output:  write square numbers 1 .... n^2

s_in:  "Please input an integer: "
s_out: "^2 = "
NL:    "\n"

start:
      write_s s_in
      read_i  R4, R4             # R4: n (ignore errors)
      write_s NL
```

```
      cload_i R5, 1             # R5: i = 1
loop:
      cmp_i   R6, R5, R4        # if (i > n) goto end
      ifgt    R6, end
      mult_i  R6, R5, R5        # i * i
      write_i R5                # printf ("%d^2 = %d\n", i, i*i);
      write_s s_out
      write_i R6
      write_s NL
      add_c   R5, R5, 1         # i = i + 1
      goto loop                 # and loop
end:
      end
##############################################################
```

More examples will be found in the example files shipped together with this program.

8 REQUIREMENTS

perl(1)

9 AUTHOR

Dr. Jürgen Vollmer <Juergen.Vollmer@informatik-vollmer.de>

Homepage of **VAM**: http://www.informatik-vollmer.de/software/vam.html
If have written "large programs" doing some interesting job, it would be nice to send me your VAM source.
If you find this software useful, I would be glad to receive a postcard from you, showing the place where you're living:
Dr. Jürgen Vollmer, Viktoriastrasse 15, D-76133 Karlsruhe, Germany.

10 LICENSE

11 CHANGELOG

```
$Log: vam.pl,v $
Revision 1.10  2009/04/06 19:51:33  vollmer
fixed for newer Perl versions

Revision 1.9  2009/04/06 19:49:46  vollmer
fixed typoos

Revision 1.6  2005/06/19 10:41:10  vollmer
fixed focu

Revision 1.5  2005/03/11 22:58:51  vollmer
fixed name of vam_ifne

Revision 1.4  2005/03/11 22:51:40  vollmer
sub cm: if a memory cell undefined during an access, assign 0 to it

Revision 1.3  2005/03/11 22:48:05  vollmer
fixed sub_f

Revision 1.2  2005/03/11 14:11:45  vollmer
fixed docu

Revision 1.1  2005/02/24 16:55:52  vollmer
Initial revision
```