Vmgen

for Gforth version 0.6.1, March 11, 2003

M. Anton Ertl (anton@mips.complang.tuwien.ac.at)

This manual is for Vmgen (version 0.6.1, March 11, 2003), the virtual machine interpreter generator

Table of Contents

1 Introduction

Vmgen is a tool for writing efficient interpreters. It takes a simple virtual machine description and generates efficient C code for dealing with the virtual machine code in various ways (in particular, executing it). The run-time efficiency of the resulting interpreters is usually within a factor of 10 of machine code produced by an optimizing compiler.

The interpreter design strategy supported by Vmgen is to divide the interpreter into two parts:

- The *front end* takes the source code of the language to be implemented, and translates it into virtual machine code. This is similar to an ordinary compiler front end; typically an interpreter front-end performs no optimization, so it is relatively simple to implement and runs fast.
- The *virtual machine interpreter* executes the virtual machine code.

Such a division is usually used in interpreters, for modularity as well as for efficiency. The virtual machine code is typically passed between front end and virtual machine interpreter in memory, like in a load-and-go compiler; this avoids the complexity and time cost of writing the code to a file and reading it again.

A *virtual machine* (VM) represents the program as a sequence of *VM instructions*, following each other in memory, similar to real machine code. Control flow occurs through VM branch instructions, like in a real machine.

In this setup, Vmgen can generate most of the code dealing with virtual machine instructions from a simple description of the virtual machine instructions (see Chapter 6 [Input File Format], page 9), in particular:

VM instruction execution
VM code generation
: Useful in the front end.

VM code decompiler
: Useful for debugging the front end.

VM code tracing
: Useful for debugging the front end and the VM interpreter. You will typically provide other means for debugging the user's programs at the source level.

VM code profiling
: Useful for optimizing the VM interpreter with superinstructions (see Section 8.6 [VM profiler], page 22).

To create parts of the interpretive system that do not deal with VM instructions, you have to use other tools (e.g., `bison`) and/or hand-code them.

Vmgen supports efficient interpreters though various optimizations, in particular

- Threaded code
- Caching the top-of-stack in a register
- Combining VM instructions into superinstructions
- Replicating VM (super)instructions for better BTB prediction accuracy (not yet in vmgen-ex, but already in Gforth).

As a result, Vmgen-based interpreters are only about an order of magnitude slower than native code from an optimizing C compiler on small benchmarks; on large benchmarks, which spend more time in the run-time system, the slowdown is often less (e.g., the slowdown of a Vmgen-generated JVM interpreter over the best JVM JIT compiler we measured is only a factor of 2-3 for large benchmarks; some other JITs and all other interpreters we looked at were slower than our interpreter).

VMs are usually designed as stack machines (passing data between VM instructions on a stack), and Vmgen supports such designs especially well; however, you can also use Vmgen for implementing a register VM (see Section 6.5 [Register Machines], page 15) and still benefit from most of the advantages offered by Vmgen.

There are many potential uses of the instruction descriptions that are not implemented at the moment, but we are open for feature requests, and we will consider new features if someone asks for them; so the feature list above is not exhaustive.

2 Why interpreters?

Interpreters are a popular language implementation technique because they combine all three of the following advantages:

- Ease of implementation
- Portability
- Fast edit-compile-run cycle

Vmgen makes it even easier to implement interpreters.

The main disadvantage of interpreters is their run-time speed. However, there are huge differences between different interpreters in this area: the slowdown over optimized C code on programs consisting of simple operations is typically a factor of 10 for the more efficient interpreters, and a factor of 1000 for the less efficient ones (the slowdown for programs executing complex operations is less, because the time spent in libraries for executing complex operations is the same in all implementation strategies).

Vmgen supports techniques for building efficient interpreters.

3 Concepts

3.1 Front end and VM interpreter

Interpretive systems are typically divided into a *front end* that parses the input language and produces an intermediate representation for the program, and an interpreter that executes the intermediate representation of the program.

For efficient interpreters the intermediate representation of choice is virtual machine code (rather than, e.g., an abstract syntax tree). *Virtual machine* (VM) code consists of VM instructions arranged sequentially in memory; they are executed in sequence by the VM interpreter, but VM branch instructions can change the control flow and are used for implementing control structures. The conceptual similarity to real machine code results in the name *virtual machine*. Various terms similar to terms for real machines are used; e.g., there are *VM registers* (like the instruction pointer and stack pointer(s)), and the VM instruction consists of an *opcode* and *immediate arguments*.

In this framework, Vmgen supports building the VM interpreter and any other component dealing with VM instructions. It does not have any support for the front end, apart from VM code generation support. The front end can be implemented with classical compiler front-end techniques, supported by tools like `flex` and `bison`.

The intermediate representation is usually just internal to the interpreter, but some systems also support saving it to a file, either as an image file, or in a full-blown linkable file format (e.g., JVM). Vmgen currently has no special support for such features, but the information in the instruction descriptions can be helpful, and we are open to feature requests and suggestions.

3.2 Data handling

Most VMs use one or more stacks for passing temporary data between VM instructions. Another option is to use a register machine architecture for the virtual machine; we believe that using a stack architecture is usually both simpler and faster.

however, this option is slower or significantly more complex to implement than a stack machine architecture.

Vmgen has special support and optimizations for stack VMs, making their implementation easy and efficient.

You can also implement a register VM with Vmgen (see Section 6.5 [Register Machines], page 15), and you will still profit from most Vmgen features.

Stack items all have the same size, so they typically will be as wide as an integer, pointer, or floating-point value. Vmgen supports treating two consecutive stack items as a single value, but anything larger is best kept in some other memory area (e.g., the heap), with pointers to the data on the stack.

Another source of data is immediate arguments VM instructions (in the VM instruction stream). The VM instruction stream is handled similar to a stack in Vmgen.

Vmgen has no built-in support for, nor restrictions against *garbage collection*. If you need garbage collection, you need to provide it in your run-time libraries. Using *reference counting* is probably harder, but might be possible (contact us if you are interested).

3.3 Dispatch

Understanding this section is probably not necessary for using Vmgen, but it may help. You may want to skip it now, and read it if you find statements about dispatch methods confusing.

After executing one VM instruction, the VM interpreter has to dispatch the next VM instruction (Vmgen calls the dispatch routine '`NEXT`'). Vmgen supports two methods of dispatch:

switch dispatch
: In this method the VM interpreter contains a giant `switch` statement, with one `case` for each VM instruction. The VM instruction opcodes are represented by integers (e.g., produced by an `enum`) in the VM code, and dispatch occurs by loading the next opcode, `switch`ing on it, and continuing at the appropriate `case`; after executing the VM instruction, the VM interpreter jumps back to the dispatch code.

threaded code
: This method represents a VM instruction opcode by the address of the start of the machine code fragment for executing the VM instruction. Dispatch consists of loading this address, jumping to it, and incrementing the VM instruction pointer. Typically the threaded-code dispatch code is appended directly to the code for executing the VM instruction. Threaded code cannot be implemented in ANSI C, but it can be implemented using GNU C's labels-as-values extension (see section "Labels as Values" in *GNU C Manual*).

Threaded code can be twice as fast as switch dispatch, depending on the interpreter, the benchmark, and the machine.

4 Invoking Vmgen

The usual way to invoke Vmgen is as follows:

`vmgen` *inputfile*

Here *inputfile* is the VM instruction description file, which usually ends in '`.vmg`'. The output filenames are made by taking the basename of '`inputfile`' (i.e., the output files will be created in the current working directory) and replacing '`.vmg`' with '`-vm.i`', '`-disasm.i`', '`-gen.i`', '`-labels.i`', '`-profile.i`', and '`-peephole.i`'. E.g., `vmgen hack/foo.vmg` will create '`foo-vm.i`', '`foo-disasm.i`', '`foo-gen.i`', '`foo-labels.i`', '`foo-profile.i`' and '`foo-peephole.i`'.

The command-line options supported by Vmgen are

'`--help`'
'`-h`' Print a message about the command-line options

'`--version`'
'`-v`' Print version and exit

5 Example

5.1 Example overview

There are two versions of the same example for using Vmgen: '`vmgen-ex`' and '`vmgen-ex2`' (you can also see Gforth as example, but it uses additional (undocumented) features, and also differs in some other respects). The example implements *mini*, a tiny Modula-2-like language with a small JavaVM-like virtual machine.

The difference between the examples is that '`vmgen-ex`' uses many casts, and '`vmgen-ex2`' tries to avoids most casts and uses unions instead. In the rest of this manual we usually mention just files in '`vmgen-ex`'; if you want to use unions, use the equivalent file in '`vmgen-ex2`'.

The files provided with each example are:

```
Makefile
README
disasm.c           wrapper file
engine.c           wrapper file
peephole.c         wrapper file
profile.c          wrapper file
mini-inst.vmg      simple VM instructions
mini-super.vmg     superinstructions (empty at first)
mini.h             common declarations
mini.l             scanner
mini.y             front end (parser, VM code generator)
support.c          main() and other support functions
fib.mini           example mini program
simple.mini        example mini program
test.mini          example mini program (tests everything)
test.out           test.mini output
stat.awk           script for aggregating profile information
peephole-blacklist list of instructions not allowed in superinstructions
seq2rule.awk       script for creating superinstructions
```

For your own interpreter, you would typically copy the following files and change little, if anything:

```
disasm.c           wrapper file
engine.c           wrapper file
peephole.c         wrapper file
profile.c          wrapper file
stat.awk           script for aggregating profile information
seq2rule.awk       script for creating superinstructions
```

You would typically change much in or replace the following files:

```
Makefile
mini-inst.vmg      simple VM instructions
mini.h             common declarations
mini.l             scanner
mini.y             front end (parser, VM code generator)
```

```
support.c          main() and other support functions
peephole-blacklist list of instructions not allowed in superinstructions
```

You can build the example by cding into the example's directory, and then typing `make`; you can check that it works with `make check`. You can run run mini programs like this:

```
./mini fib.mini
```

To learn about the options, type `./mini -h`.

5.2 Using profiling to create superinstructions

I have not added rules for this in the 'Makefile' (there are many options for selecting superinstructions, and I did not want to hardcode one into the 'Makefile'), but there are some supporting scripts, and here's an example:

Suppose you want to use 'fib.mini' and 'test.mini' as training programs, you get the profiles like this:

```
make fib.prof test.prof #takes a few seconds
```

You can aggregate these profiles with 'stat.awk':

```
awk -f stat.awk fib.prof test.prof
```

The result contains lines like:

```
      2      16        36910041 loadlocal lit
```

This means that the sequence `loadlocal lit` statically occurs a total of 16 times in 2 profiles, with a dynamic execution count of 36910041.

The numbers can be used in various ways to select superinstructions. E.g., if you just want to select all sequences with a dynamic execution count exceeding 10000, you would use the following pipeline:

```
awk -f stat.awk fib.prof test.prof|
awk '$3>=10000'|                #select sequences
fgrep -v -f peephole-blacklist| #eliminate wrong instructions
awk -f seq2rule.awk|  #transform sequences into superinstruction rules
sort -k 3 >mini-super.vmg       #sort sequences
```

The file 'peephole-blacklist' contains all instructions that directly access a stack or stack pointer (for mini: `call`, `return`); the sort step is necessary to ensure that prefixes precede larger superinstructions.

Now you can create a version of mini with superinstructions by just saying 'make'

6 Input File Format

Vmgen takes as input a file containing specifications of virtual machine instructions. This file usually has a name ending in '.vmg'.

Most examples are taken from the example in 'vmgen-ex'.

6.1 Input File Grammar

The grammar is in EBNF format, with *a*|*b* meaning "*a* or *b*", {*c*} meaning 0 or more repetitions of *c* and [*d*] meaning 0 or 1 repetitions of *d*.

Vmgen input is not free-format, so you have to take care where you put newlines (and, in a few cases, white space).

```
description: {instruction|comment|eval-escape|c-escape}

instruction: simple-inst|superinst

simple-inst: ident '(' stack-effect ')' newline c-code newline newline

stack-effect: {ident} '--' {ident}

super-inst: ident '=' ident {ident}

comment:      '\ '  text newline

eval-escape:  '\E ' text newline

c-escape:     '\C ' text newline
```

Note that the \s in this grammar are meant literally, not as C-style encodings for non-printable characters.

There are two ways to delimit the C code in `simple-inst`:

- If you start it with a '{' at the start of a line (i.e., not even white space before it), you have to end it with a '}' at the start of a line (followed by a newline). In this case you may have empty lines within the C code (typically used between variable definitions and statements).
- You do not start it with '{'. Then the C code ends at the first empty line, so you cannot have empty lines within this code.

The text in `comment`, `eval-escape` and `c-escape` must not contain a newline. `Ident` must conform to the usual conventions of C identifiers (otherwise the C compiler would choke on the Vmgen output), except that idents in `stack-effect` may have a stack prefix (for stack prefix syntax, see Section 6.1.1 [Eval escapes], page 10).

The `c-escape` passes the text through to each output file (without the '\C'). This is useful mainly for conditional compilation (i.e., you write '\C #if ...' etc.).

In addition to the syntax given in the grammer, Vmgen also processes sync lines (lines starting with '#line'), as produced by 'm4 -s' (see section "Invoking m4" in *GNU m4*) and similar tools. This allows associating C compiler error messages with the original source of the C code.

Vmgen understands a few extensions beyond the grammar given here, but these extensions are only useful for building Gforth. You can find a description of the format used for Gforth in 'prim'.

6.1.1 Eval escapes

The text in `eval-escape` is Forth code that is evaluated when Vmgen reads the line. You will normally use this feature to define stacks and types.

If you do not know (and do not want to learn) Forth, you can build the text according to the following grammar; these rules are normally all Forth you need for using Vmgen:

```
text: stack-decl|type-prefix-decl|stack-prefix-decl|set-flag

stack-decl: 'stack ' ident ident ident
type-prefix-decl:
    's" ' string '" ' ('single'|'double') ident 'type-prefix' ident
stack-prefix-decl:  ident 'stack-prefix' string
set-flag: ('store-optimization'|'include-skipped-insts') ('on'|'off')
```

Note that the syntax of this code is not checked thoroughly (there are many other Forth program fragments that could be written in an eval-escape).

A stack prefix can contain letters, digits, or ':', and may start with an '#'; e.g., in Gforth the return stack has the stack prefix 'R:'. This restriction is not checked during the stack prefix definition, but it is enforced by the parsing rules for stack items later.

If you know Forth, the stack effects of the non-standard words involved are:

```
stack                 ( "name" "pointer" "type" -- )
                      ( name execution: -- stack )
type-prefix           ( addr u item-size stack "prefix" -- )
single                ( -- item-size )
double                ( -- item-size )
stack-prefix          ( stack "prefix" -- )
store-optimization    ( -- addr )
include-skipped-insts ( -- addr )
```

An *item-size* takes three cells on the stack.

6.2 Simple instructions

We will use the following simple VM instruction description as example:

```
sub ( i1 i2 -- i )
i = i1-i2;
```

The first line specifies the name of the VM instruction (`sub`) and its stack effect (`i1 i2 -- i`). The rest of the description is just plain C code.

The stack effect specifies that `sub` pulls two integers from the data stack and puts them in the C variables `i1` and `i2` (with the rightmost item (`i2`) taken from the top of stack; intuition: if you push `i1`, then `i2` on the stack, the resulting stack picture is `i1 i2`) and later pushes one integer (`i`) on the data stack (the rightmost item is on the top afterwards).

How do we know the type and stack of the stack items? Vmgen uses prefixes, similar to Fortran; in contrast to Fortran, you have to define the prefix first:

```
\E s" Cell"   single data-stack type-prefix i
```

This defines the prefix `i` to refer to the type `Cell` (defined as `long` in '`mini.h`') and, by default, to the `data-stack`. It also specifies that this type takes one stack item (`single`). The type prefix is part of the variable name.

Before we can use `data-stack` in this way, we have to define it:

```
\E stack data-stack sp Cell
```

This line defines the stack `data-stack`, which uses the stack pointer `sp`, and each item has the basic type `Cell`; other types have to fit into one or two `Cells` (depending on whether the type is `single` or `double` wide), and are cast from and to Cells on accessing the `data-stack` with type cast macros (see Section 8.1 [VM engine], page 17). By default, stacks grow towards lower addresses in Vmgen-erated interpreters (see Section 6.2.3 [Stack growth direction], page 13).

We can override the default stack of a stack item by using a stack prefix. E.g., consider the following instruction:

```
lit ( #i -- i )
```

The VM instruction `lit` takes the item `i` from the instruction stream (indicated by the prefix `#`), and pushes it on the (default) data stack. The stack prefix is not part of the variable name. Stack prefixes are defined like this:

```
\E inst-stream stack-prefix #
```

This definition defines that the stack prefix `#` specifies the "stack" `inst-stream`. Since the instruction stream behaves a little differently than an ordinary stack, it is predefined, and you do not need to define it.

The instruction stream contains instructions and their immediate arguments, so specifying that an argument comes from the instruction stream indicates an immediate argument. Of course, instruction stream arguments can only appear to the left of `--` in the stack effect. If there are multiple instruction stream arguments, the leftmost is the first one (just as the intuition suggests).

6.2.1 C Code Macros

Vmgen recognizes the following strings in the C code part of simple instructions:

`SET_IP`
: As far as Vmgen is concerned, a VM instruction containing this ends a VM basic block (used in profiling to delimit profiled sequences). On the C level, this also sets the instruction pointer.

`SUPER_END`
: This ends a basic block (for profiling), even if the instruction contains no `SET_IP`.

`INST_TAIL;`
: Vmgen replaces '`INST_TAIL;`' with code for ending a VM instruction and dispatching the next VM instruction. Even without a '`INST_TAIL;`' this happens automatically when control reaches the end of the C code. If you want to have

this in the middle of the C code, you need to use 'INST_TAIL;'. A typical example is a conditional VM branch:

```
if (branch_condition) {
  SET_IP(target); INST_TAIL;
}
/* implicit tail follows here */
```

In this example, 'INST_TAIL;' is not strictly necessary, because there is another one implicitly after the if-statement, but using it improves branch prediction accuracy slightly and allows other optimizations.

SUPER_CONTINUE
: This indicates that the implicit tail at the end of the VM instruction dispatches the sequentially next VM instruction even if there is a SET_IP in the VM instruction. This enables an optimization that is not yet implemented in the vmgen-ex code (but in Gforth). The typical application is in conditional VM branches:

```
if (branch_condition) {
  SET_IP(target); INST_TAIL; /* now this INST_TAIL is necessary */
}
SUPER_CONTINUE;
```

Note that Vmgen is not smart about C-level tokenization, comments, strings, or conditional compilation, so it will interpret even a commented-out SUPER_END as ending a basic block (or, e.g., 'RESET_IP;' as 'SET_IP;'). Conversely, Vmgen requires the literal presence of these strings; Vmgen will not see them if they are hiding in a C preprocessor macro.

6.2.2 C Code restrictions

Vmgen generates code and performs some optimizations under the assumption that the user-supplied C code does not access the stack pointers or stack items, and that accesses to the instruction pointer only occur through special macros. In general you should heed these restrictions. However, if you need to break these restrictions, read the following.

Accessing a stack or stack pointer directly can be a problem for several reasons:

- Vmgen optionally supports caching the top-of-stack item in a local variable (that is allocated to a register). This is the most frequent source of trouble. You can deal with it either by not using top-of-stack caching (slowdown factor 1-1.4, depending on machine), or by inserting flushing code (e.g., 'IF_spTOS(sp[...] = spTOS);') at the start and reloading code (e.g., 'IF_spTOS(spTOS = sp[0])') at the end of problematic C code. Vmgen inserts a stack pointer update before the start of the user-supplied C code, so the flushing code has to use an index that corrects for that. In the future, this flushing may be done automatically by mentioning a special string in the C code.
- The Vmgen-erated code loads the stack items from stack-pointer-indexed memory into variables before the user-supplied C code, and stores them from variables to stack-pointer-indexed memory afterwards. If you do any writes to the stack through its stack pointer in your C code, it will not affect the variables, and your write may be overwritten by the stores after the C code. Similarly, a read from a stack using a stack pointer will not reflect computations of stack items in the same VM instruction.

- Superinstructions keep stack items in variables across the whole superinstruction. So you should not include VM instructions, that access a stack or stack pointer, as components of superinstructions (see Section 8.6 [VM profiler], page 22).

You should access the instruction pointer only through its special macros (`IP`, `SET_IP`, `IPTOS`); this ensure that these macros can be implemented in several ways for best performance. `IP` points to the next instruction, and `IPTOS` is its contents.

6.2.3 Stack growth direction

By default, the stacks grow towards lower addresses. You can change this for a stack by setting the `stack-access-transform` field of the stack to an xt `( itemnum -- index )` that performs the appropriate index transformation.

E.g., if you want to let `data-stack` grow towards higher addresses, with the stack pointer always pointing just beyond the top-of-stack, use this right after defining `data-stack`:

```
\E : sp-access-transform ( itemnum -- index ) negate 1- ;
\E ' sp-access-transform ' data-stack >body stack-access-transform !
```

This means that `sp-access-transform` will be used to generate indexes for accessing `data-stack`. The definition of `sp-access-transform` above transforms n into -n-1, e.g, 1 into -2. This will access the 0th data-stack element (top-of-stack) at sp[-1], the 1st at sp[-2], etc., which is the typical way upward-growing stacks are used. If you need a different transform and do not know enough Forth to program it, let me know.

6.3 Superinstructions

Note: don't invest too much work in (static) superinstructions; a future version of Vmgen will support dynamic superinstructions (see Ian Piumarta and Fabio Riccardi, *Optimizing Direct Threaded Code by Selective Inlining*, PLDI'98), and static superinstructions have much less benefit in that context (preliminary results indicate only a factor 1.1 speedup).

Here is an example of a superinstruction definition:

```
lit_sub = lit sub
```

`lit_sub` is the name of the superinstruction, and `lit` and `sub` are its components. This superinstruction performs the same action as the sequence `lit` and `sub`. It is generated automatically by the VM code generation functions whenever that sequence occurs, so if you want to use this superinstruction, you just need to add this definition (and even that can be partially automatized, see Section 8.6 [VM profiler], page 22).

Vmgen requires that the component instructions are simple instructions defined before superinstructions using the components. Currently, Vmgen also requires that all the subsequences at the start of a superinstruction (prefixes) must be defined as superinstruction before the superinstruction. I.e., if you want to define a superinstruction

```
foo4 = load add sub mul
```

you first have to define `load`, `add`, `sub` and `mul`, plus

```
foo2 = load add
foo3 = load add sub
```

Here, `sumof4` is the longest prefix of `sumof5`, and `sumof3` is the longest prefix of `sumof4`.

Note that Vmgen assumes that only the code it generates accesses stack pointers, the instruction pointer, and various stack items, and it performs optimizations based on this assumption. Therefore, VM instructions where your C code changes the instruction pointer should only be used as last component; a VM instruction where your C code accesses a stack pointer should not be used as component at all. Vmgen does not check these restrictions, they just result in bugs in your interpreter.

The Vmgen flag `include-skipped-insts` influences superinstruction code generation. Currently there is no support in the peephole optimizer for both variations, so leave this flag alone for now.

6.4 Store Optimization

This minor optimization (0.6\%–0.8\% reduction in executed instructions for Gforth) puts additional requirements on the instruction descriptions and is therefore disabled by default.

What does it do? Consider an instruction like

```
dup ( n -- n n )
```

For simplicity, also assume that we are not caching the top-of-stack in a register. Now, the C code for dup first loads `n` from the stack, and then stores it twice to the stack, one time to the address where it came from; that time is unnecessary, but gcc does not optimize it away, so vmgen can do it instead (if you turn on the store optimization).

Vmgen uses the stack item's name to determine if the stack item contains the same value as it did at the start. Therefore, if you use the store optimization, you have to ensure that stack items that have the same name on input and output also have the same value, and are not changed in the C code you supply. I.e., the following code could fail if you turn on the store optimization:

```
add1 ( n -- n )
n++;
```

Instead, you have to use different names, i.e.:

```
add1 ( n1 -- n1 )
n2=n1+1;
```

Similarly, the store optimization assumes that the stack pointer is only changed by Vmgen-erated code. If your C code changes the stack pointer, use different names in input and output stack items to avoid a (probably wrong) store optimization, or turn the store optimization off for this VM instruction.

To turn on the store optimization, write

```
\E store-optimization on
```

at the start of the file. You can turn this optimization on or off between any two VM instruction descriptions. For turning it off again, you can use

```
\E store-optimization off
```

6.5 Register Machines

If you want to implement a register VM rather than a stack VM with Vmgen, there are two ways to do it: Directly and through superinstructions.

If you use the direct way, you define instructions that take the register numbers as immediate arguments, like this:

```
add3 ( #src1 #src2 #dest -- )
reg[dest] = reg[src1]+reg[src2];
```

A disadvantage of this method is that during tracing you only see the register numbers, but not the register contents. Actually, with an appropriate definition of `printarg_src` (see Section 8.1 [VM engine], page 17), you can print the values of the source registers on entry, but you cannot print the value of the destination register on exit.

If you use superinstructions to define a register VM, you define simple instructions that use a stack, and then define superinstructions that have no overall stack effect, like this:

```
loadreg ( #src -- n )
n = reg[src];

storereg ( n #dest -- )
reg[dest] = n;

adds ( n1 n2 -- n )
n = n1+n2;

add3 = loadreg loadreg adds storereg
```

An advantage of this method is that you see the values and not just the register numbers in tracing. A disadvantage of this method is that currently you cannot generate superinstructions directly, but only through generating a sequence of simple instructions (we might change this in the future if there is demand).

Could the register VM support be improved, apart from the issues mentioned above? It is hard to see how to do it in a general way, because there are a number of different designs that different people mean when they use the term *register machine* in connection with VM interpreters. However, if you have ideas or requests in that direction, please let me know (see Chapter 12 [Contact], page 27).

7 Error messages

These error messages are created by Vmgen:

`# can only be on the input side`
: You have used an instruction-stream prefix (usually '`#`') after the '`--`' (the output side); you can only use it before (the input side).

`the prefix for this superinstruction must be defined earlier`
: You have defined a superinstruction (e.g. `abc = a b c`) without defining its direct prefix (e.g., `ab = a b`), See Section 6.3 [Superinstructions], page 13.

`sync line syntax`
: If you are using a preprocessor (e.g., `m4`) to generate Vmgen input code, you may want to create `#line` directives (aka sync lines). This error indicates that such a line is not in th syntax expected by Vmgen (this should not happen; please report the offending line in a bug report).

`syntax error, wrong char`
: A syntax error. If you do not see right away where the error is, it may be helpful to check the following: Did you put an empty line in a VM instruction where the C code is not delimited by braces (then the empty line ends the VM instruction)? If you used brace-delimited C code, did you put the delimiting braces (and only those) at the start of the line, without preceding white space? Did you forget a delimiting brace?

`too many stacks`
: Vmgen currently supports 3 stacks (plus the instruction stream); if you need more, let us know.

`unknown prefix`
: The stack item does not match any defined type prefix (after stripping away any stack prefix). You should either declare the type prefix you want for that stack item, or use a different type prefix

`unknown primitive`
: You have used the name of a simple VM instruction in a superinstruction definition without defining the simple VM instruction first.

In addition, the C compiler can produce errors due to code produced by Vmgen; e.g., you need to define type cast functions.

8 Using the generated code

The easiest way to create a working VM interpreter with Vmgen is probably to start with '`vmgen-ex`', and modify it for your purposes. This chapter explains what the various wrapper and generated files do. It also contains reference-manual style descriptions of the macros, variables etc. used by the generated code, and you can skip that on first reading.

8.1 VM engine

The VM engine is the VM interpreter that executes the VM code. It is essential for an interpretive system.

Vmgen supports two methods of VM instruction dispatch: *threaded code* (fast, but gcc-specific), and *switch dispatch* (slow, but portable across C compilers); you can use conditional compilation ('`defined(__GNUC__)`') to choose between these methods, and our example does so.

For both methods, the VM engine is contained in a C-level function. Vmgen generates most of the contents of the function for you ('*name*`-vm.i`'), but you have to define this function, and macros and variables used in the engine, and initialize the variables. In our example the engine function also includes '*name*`-labels.i`' (see Section 8.2 [VM instruction table], page 20).

In addition to executing the code, the VM engine can optionally also print out a trace of the executed instructions, their arguments and results. For superinstructions it prints the trace as if only component instructions were executed; this allows to introduce new superinstructions while keeping the traces comparable to old ones (important for regression tests).

It costs significant performance to check in each instruction whether to print tracing code, so we recommend producing two copies of the engine: one for fast execution, and one for tracing. See the rules for '`engine.o`' and '`engine-debug.o`' in '`vmgen-ex/Makefile`' for an example.

The following macros and variables are used in '*name*`-vm.i`':

`LABEL(`*inst_name*`)`
: This is used just before each VM instruction to provide a jump or `switch` label (the '`:`' is provided by Vmgen). For switch dispatch this should expand to '`case` *label*`:`'; for threaded-code dispatch this should just expand to '*label*`:`'. In either case *label* is usually the *inst_name* with some prefix or suffix to avoid naming conflicts.

`LABEL2(`*inst_name*`)`
: This will be used for dynamic superinstructions; at the moment, this should expand to nothing.

`NAME(`*inst_name_string*`)`
: Called on entering a VM instruction with a string containing the name of the VM instruction as parameter. In normal execution this should be expand to nothing, but for tracing this usually prints the name, and possibly other information (several VM registers in our example).

`DEF_CA` Usually empty. Called just inside a new scope at the start of a VM instruction. Can be used to define variables that should be visible during every VM instruction. If you define this macro as non-empty, you have to provide the finishing '`;`' in the macro.

`NEXT_P0 NEXT_P1 NEXT_P2`
: The three parts of instruction dispatch. They can be defined in different ways for best performance on various processors (see '`engine.c`' in the example or '`engine/threaded.h`' in Gforth). '`NEXT_P0`' is invoked right at the start of the VM instruction (but after '`DEF_CA`'), '`NEXT_P1`' right after the user-supplied C code, and '`NEXT_P2`' at the end. The actual jump has to be performed by '`NEXT_P2`' (if you would do it earlier, important parts of the VM instruction would not be executed).

  The simplest variant is if '`NEXT_P2`' does everything and the other macros do nothing. Then also related macros like '`IP`', '`SET_IP`', '`IP`', '`INC_IP`' and '`IPTOS`' are very straightforward to define. For switch dispatch this code consists just of a jump to the dispatch code ('`goto next_inst;`' in our example); for direct threaded code it consists of something like '`({cfa=*ip++; goto *cfa;})`'.

  Pulling code (usually the '`cfa=*ip++;`') up into '`NEXT_P1`' usually does not cause problems, but pulling things up into '`NEXT_P0`' usually requires changing the other macros (and, at least for Gforth on Alpha, it does not buy much, because the compiler often manages to schedule the relevant stuff up by itself). An even more extreme variant is to pull code up even further, into, e.g., NEXT_P1 of the previous VM instruction (prefetching, useful on PowerPCs).

`INC_IP(`*n*`)`
: This increments `IP` by *n*.

`SET_IP(`*target*`)`
: This sets `IP` to *target*.

`vm_`*A*2*B*`(a,b)`
: Type casting macro that assigns '`a`' (of type *A*) to '`b`' (of type *B*). This is mainly used for getting stack items into variables and back. So you need to define macros for every combination of stack basic type (`Cell` in our example) and type-prefix types used with that stack (in both directions). For the type-prefix type, you use the type-prefix (not the C type string) as type name (e.g., '`vm_Cell2i`', not '`vm_Cell2Cell`'). In addition, you have to define a vm_*X*2*X* macro for the stack's basic type *X* (used in superinstructions).

  The stack basic type for the predefined '`inst-stream`' is '`Cell`'. If you want a stack with the same item size, making its basic type '`Cell`' usually reduces the number of macros you have to define.

  Here our examples differ a lot: '`vmgen-ex`' uses casts in these macros, whereas '`vmgen-ex2`' uses union-field selection (or assignment to union fields). Note that casting floats into integers and vice versa changes the bit pattern (and you do not want that). In this case your options are to use a (temporary) union, or to take the address of the value, cast the pointer, and dereference that (not always possible, and sometimes expensive).

`vm_two`*A*2*B*`(a1,a2,b)`
`vm_`*B*`2two`*A*`(b,a1,a2)`
: Type casting between two stack items (`a1`, `a2`) and a variable `b` of a type that takes two stack items. This does not occur in our small examples, but you can look at Gforth for examples (see `vm_twoCell2d` in '`engine/forth.h`').

stackpointer
: For each stack used, the stackpointer name given in the stack declaration is used. For a regular stack this must be an l-expression; typically it is a variable declared as a pointer to the stack's basic type. For '`inst-stream`', the name is '`IP`', and it can be a plain r-value; typically it is a macro that abstracts away the differences between the various implementations of `NEXT_P*`.

`IMM_ARG(access,value)`
: Define this to expland to "(access)". This is just a placeholder for future extensions.

stackpointer`TOS`
: The top-of-stack for the stack pointed to by *stackpointer*. If you are using top-of-stack caching for that stack, this should be defined as variable; if you are not using top-of-stack caching for that stack, this should be a macro expanding to '*stackpointer*`[0]`'. The stack pointer for the predefined '`inst-stream`' is called '`IP`', so the top-of-stack is called '`IPTOS`'.

`IF_`*stackpointer*`TOS(`*expr*`)`
: Macro for executing *expr*, if top-of-stack caching is used for the *stackpointer* stack. I.e., this should do *expr* if there is top-of-stack caching for *stackpointer*; otherwise it should do nothing.

`SUPER_END`
: This is used by the VM profiler (see Section 8.6 [VM profiler], page 22); it should not do anything in normal operation, and call `vm_count_block(IP)` for profiling.

`SUPER_CONTINUE`
: This is just a hint to Vmgen and does nothing at the C level.

`VM_DEBUG`
: If this is defined, the tracing code will be compiled in (slower interpretation, but better debugging). Our example compiles two versions of the engine, a fast-running one that cannot trace, and one with potential tracing and profiling.

`vm_debug`
: Needed only if '`VM_DEBUG`' is defined. If this variable contains true, the VM instructions produce trace output. It can be turned on or off at any time.

`vm_out`
: Needed only if '`VM_DEBUG`' is defined. Specifies the file on which to print the trace output (type '`FILE *`').

`printarg_`*type*`(`*value*`)`
: Needed only if '`VM_DEBUG`' is defined. Macro or function for printing *value* in a way appropriate for the *type*. This is used for printing the values of stack items during tracing. *Type* is normally the type prefix specified in a `type-prefix` definition (e.g., '`printarg_i`'); in superinstructions it is currently the basic type of the stack.

8.2 VM instruction table

For threaded code we also need to produce a table containing the labels of all VM instructions. This is needed for VM code generation (see Section 8.3 [VM code generation], page 20), and it has to be done in the engine function, because the labels are not visible outside. It then has to be passed outside the function (and assigned to '`vm_prim`'), to be used by the VM code generation functions.

This means that the engine function has to be called first to produce the VM instruction table, and later, after generating VM code, it has to be called again to execute the generated VM code (yes, this is ugly). In our example program, these two modes of calling the engine function are differentiated by the value of the parameter ip0 (if it equals 0, then the table is passed out, otherwise the VM code is executed); in our example, we pass the table out by assigning it to '`vm_prim`' and returning from '`engine`'.

In our example ('`vmgen-ex/engine.c`'), we also build such a table for switch dispatch; this is mainly done for uniformity.

For switch dispatch, we also need to define the VM instruction opcodes used as case labels in an `enum`.

For both purposes (VM instruction table, and enum), the file '*name*`-labels.i`' is generated by Vmgen. You have to define the following macro used in this file:

`INST_ADDR(`*inst_name*`)`
: For switch dispatch, this is just the name of the switch label (the same name as used in '`LABEL(`*inst_name*`)`'), for both uses of '*name*`-labels.i`'. For threaded-code dispatch, this is the address of the label defined in '`LABEL(`*inst_name*`)`'); the address is taken with '`&&`' (see section "Labels as Values" in *GNU C Manual*).

8.3 VM code generation

Vmgen generates VM code generation functions in '*name*`-gen.i`' that the front end can call to generate VM code. This is essential for an interpretive system.

For a VM instruction '`x ( #a b #c -- d )`', Vmgen generates a function with the prototype

```
void gen_x(Inst **ctp, a_type a, c_type c)
```

The `ctp` argument points to a pointer to the next instruction. `*ctp` is increased by the generation functions; i.e., you should allocate memory for the code to be generated beforehand, and start with *ctp set at the start of this memory area. Before running out of memory, allocate a new area, and generate a VM-level jump to the new area (this overflow handling is not implemented in our examples).

The other arguments correspond to the immediate arguments of the VM instruction (with their appropriate types as defined in the `type_prefix` declaration.

The following types, variables, and functions are used in '*name*`-gen.i`':

`Inst`
: The type of the VM instruction; if you use threaded code, this is `void *`; for switch dispatch this is an integer type.

`vm_prim` The VM instruction table (type: `Inst *`, see Section 8.2 [VM instruction table], page 20).

`gen_inst(Inst **ctp, Inst i)`
This function compiles the instruction `i`. Take a look at it in '`vmgen-ex/peephole.c`'. It is trivial when you don't want to use superinstructions (just the last two lines of the example function), and slightly more complicated in the example due to its ability to use superinstructions (see Section 8.4 [Peephole optimization], page 21).

`genarg_`*type_prefix*`(Inst **ctp,` *type type_prefix*`)`
This compiles an immediate argument of *type* (as defined in a `type-prefix` definition). These functions are trivial to define (see '`vmgen-ex/support.c`'). You need one of these functions for every type that you use as immediate argument.

In addition to using these functions to generate code, you should call `BB_BOUNDARY` at every basic block entry point if you ever want to use superinstructions (or if you want to use the profiling supported by Vmgen; but this support is also useful mainly for selecting superinstructions). If you use `BB_BOUNDARY`, you should also define it (take a look at its definition in '`vmgen-ex/mini.y`').

You do not need to call `BB_BOUNDARY` after branches, because you will not define superinstructions that contain branches in the middle (and if you did, and it would work, there would be no reason to end the superinstruction at the branch), and because the branches announce themselves to the profiler.

8.4 Peephole optimization

You need peephole optimization only if you want to use superinstructions. But having the code for it does not hurt much if you do not use superinstructions.

A simple greedy peephole optimization algorithm is used for superinstruction selection: every time `gen_inst` compiles a VM instruction, it checks if it can combine it with the last VM instruction (which may also be a superinstruction resulting from a previous peephole optimization); if so, it changes the last instruction to the combined instruction instead of laying down `i` at the current '`*ctp`'.

The code for peephole optimization is in '`vmgen-ex/peephole.c`'. You can use this file almost verbatim. Vmgen generates '*file*`-peephole.i`' which contains data for the peephoile optimizer.

You have to call '`init_peeptable()`' after initializing '`vm_prim`', and before compiling any VM code to initialize data structures for peephole optimization. After that, compiling with the VM code generation functions will automatically combine VM instructions into superinstructions. Since you do not want to combine instructions across VM branch targets (otherwise there will not be a proper VM instruction to branch to), you have to call `BB_BOUNDARY` (see Section 8.3 [VM code generation], page 20) at branch targets.

8.5 VM disassembler

A VM code disassembler is optional for an interpretive system, but highly recommended during its development and maintenance, because it is very useful for detecting bugs in the front end (and for distinguishing them from VM interpreter bugs).

Vmgen supports VM code disassembling by generating '*file*-disasm.i'. This code has to be wrapped into a function, as is done in 'vmgen-ex/disasm.c'. You can use this file almost verbatim. In addition to 'vm_*A2B*(a,b)', 'vm_out', 'printarg_*type*(*value*)', which are explained above, the following macros and variables are used in '*file*-disasm.i' (and you have to define them):

`ip`
: This variable points to the opcode of the current VM instruction.

`IP IPTOS`
: 'IPTOS' is the first argument of the current VM instruction, and 'IP' points to it; this is just as in the engine, but here 'ip' points to the opcode of the VM instruction (in contrast to the engine, where 'ip' points to the next cell, or even one further).

`VM_IS_INST(Inst i, int n)`
: Tests if the opcode 'i' is the same as the 'n'th entry in the VM instruction table.

8.6 VM profiler

The VM profiler is designed for getting execution and occurence counts for VM instruction sequences, and these counts can then be used for selecting sequences as superinstructions. The VM profiler is probably not useful as profiling tool for the interpretive system. I.e., the VM profiler is useful for the developers, but not the users of the interpretive system.

The output of the profiler is: for each basic block (executed at least once), it produces the dynamic execution count of that basic block and all its subsequences; e.g.,

```
9227465  lit storelocal
9227465  storelocal branch
9227465  lit storelocal branch
```

I.e., a basic block consisting of 'lit storelocal branch' is executed 9227465 times.

This output can be combined in various ways. E.g., 'vmgen-ex/stat.awk' adds up the occurences of a given sequence wrt dynamic execution, static occurence, and per-program occurence. E.g.,

```
2       16         36910041 loadlocal lit
```

indicates that the sequence 'loadlocal lit' occurs in 2 programs, in 16 places, and has been executed 36910041 times. Now you can select superinstructions in any way you like (note that compile time and space typically limit the number of superinstructions to 100–1000). After you have done that, 'vmgen/seq2rule.awk' turns lines of the form above into rules for inclusion in a Vmgen input file. Note that this script does not ensure that all prefixes are defined, so you have to do that in other ways. So, an overall script for turning profiles into superinstructions can look like this:

```
awk -f stat.awk fib.prof test.prof|
awk '$3>=10000'|                #select sequences
```

```
fgrep -v -f peephole-blacklist| #eliminate wrong instructions
awk -f seq2rule.awk|            #turn into superinstructions
sort -k 3 >mini-super.vmg       #sort sequences
```

Here the dynamic count is used for selecting sequences (preliminary results indicate that the static count gives better results, though); the third line eliminates sequences containing instructions that must not occur in a superinstruction, because they access a stack directly. The dynamic count selection ensures that all subsequences (including prefixes) of longer sequences occur (because subsequences have at least the same count as the longer sequences); the sort in the last line ensures that longer superinstructions occur after their prefixes.

But before using this, you have to have the profiler. Vmgen supports its creation by generating '*file*-`profile.i`'; you also need the wrapper file '`vmgen-ex/profile.c`' that you can use almost verbatim.

The profiler works by recording the targets of all VM control flow changes (through `SUPER_END` during execution, and through `BB_BOUNDARY` in the front end), and counting (through `SUPER_END`) how often they were targeted. After the program run, the numbers are corrected such that each VM basic block has the correct count (entering a block without executing a branch does not increase the count, and the correction fixes that), then the subsequences of all basic blocks are printed. To get all this, you just have to define `SUPER_END` (and `BB_BOUNDARY`) appropriately, and call `vm_print_profile(FILE *file)` when you want to output the profile on `file`.

The '*file*-`profile.i`' is similar to the disassembler file, and it uses variables and functions defined in '`vmgen-ex/profile.c`', plus `VM_IS_INST` already defined for the VM disassembler (see Section 8.5 [VM disassembler], page 22).

9 Hints

9.1 Floating point

How should you deal with floating point values? Should you use the same stack as for integers/pointers, or a different one? This section discusses this issue with a view on execution speed.

The simpler approach is to use a separate floating-point stack. This allows you to choose FP value size without considering the size of the integers/pointers, and you avoid a number of performance problems. The main downside is that this needs an FP stack pointer (and that may not fit in the register file on the 386 arhitecture, costing some performance, but comparatively little if you take the other option into account). If you use a separate FP stack (with stack pointer `fp`), using an fpTOS is helpful on most machines, but some spill the fpTOS register into memory, and fpTOS should not be used there.

The other approach is to share one stack (pointed to by, say, `sp`) between integer/pointer and floating-point values. This is ok if you do not use `spTOS`. If you do use `spTOS`, the compiler has to decide whether to put that variable into an integer or a floating point register, and the other type of operation becomes quite expensive on most machines (because moving values between integer and FP registers is quite expensive). If a value of one type has to be synthesized out of two values of the other type (`double` types), things are even more interesting.

One way around this problem would be to not use the `spTOS` supported by Vmgen, but to use explicit top-of-stack variables (one for integers, one for FP values), and having a kind of accumulator+stack architecture (e.g., Ocaml bytecode uses this approach); however, this is a major change, and it's ramifications are not completely clear.

10 The future

We have a number of ideas for future versions of Vmgen. However, there are so many possible things to do that we would like some feedback from you. What are you doing with Vmgen, what features are you missing, and why?

One idea we are thinking about is to generate just one '`.c`' file instead of letting you copy and adapt all the wrapper files (you would still have to define stuff like the type-specific macros, and stack pointers etc. somewhere). The advantage would be that, if we change the wrapper files between versions, you would not need to integrate your changes and our changes to them; Vmgen would also be easier to use for beginners. The main disadvantage of that is that it would reduce the flexibility of Vmgen a little (well, those who like flexibility could still patch the resulting '`.c`' file, like they are now doing for the wrapper files). In any case, if you are doing things to the wrapper files that would cause problems in a generated-'`.c`'-file approach, please let us know.

11 Changes

User-visible changes between 0.5.9-20020822 and 0.5.9-20020901:

The store optimization is now disabled by default, but can be enabled by the user (see Section 6.4 [Store Optimization], page 14). Documentation for this optimization is also new.

User-visible changes between 0.5.9-20010501 and 0.5.9-20020822:

There is now a manual (in info, HTML, Postscript, or plain text format).

There is the vmgen-ex2 variant of the vmgen-ex example; the new variant uses a union type instead of lots of casting.

Both variants of the example can now be compiled with an ANSI C compiler (using switch dispatch and losing quite a bit of performance); tested with `lcc`.

Users of the gforth-0.5.9-20010501 version of Vmgen need to change several things in their source code to use the current version. I recommend keeping the gforth-0.5.9-20010501 version until you have completed the change (note that you can have several versions of Gforth installed at the same time). I hope to avoid such incompatible changes in the future.

The required changes are:

`TAIL;`
: has been renamed into `INST_TAIL;` (less chance of an accidental match).

`vm_`*A2B*
: now takes two arguments.

`vm_two`*A2B*`(b,a1,a2);`
: changed to vm_two*A2B*(a1,a2,b) (note the absence of the '`;`').

Also some new macros have to be defined, e.g., `INST_ADDR`, and `LABEL`; some macros have to be defined in new contexts, e.g., `VM_IS_INST` is now also needed in the disassembler.

12 Contact

To report a bug, use `https://savannah.gnu.org/bugs/?func=addbug&group_id=2672`.

For discussion on Vmgen (e.g., how to use it), use the mailing list `bug-vmgen@mail.freesoftware.fsf.org` (use `http://mail.gnu.org/mailman/listinfo/help-vmgen` to subscribe).

You can find vmgen information at `http://www.complang.tuwien.ac.at/anton/vmgen/`.

Appendix A Copying This Manual

A.1 GNU Free Documentation License

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgments" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgments", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.1.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

L

M

N

O

P

R

S

T

U

V

W