vcc Reference

Michael Tobin
mtobin@mac.com

Overview

vcc is a compiler for a simple derivative of the C language to target the VeSPA processor architecture. The compiler itself is very simple, consisting of perhaps around 3,000 lines of code and 10 modules. It is based on the compiler described in [1], though with many enhancements to its functionality, such as arrays, for loops, and more.

About this document

This document has three aims: first, to describe to the user how to invoke the compiler and what options are available; second, to describe the subset of C the compiler uses and idiosyncrasies about the language; and finally how to make simple modifications in order to extend the compiler's functionality.

Part 1: Installation and running

Installation is pretty straightforward. First, unpackage the `vcc-1.0.tar.gz` file by running the command:

```
% gtar –xvzf vcc-1.0.tar.gz
```

This will create several directories and files needed to build and run the compiler. Next, in the `vcc-1.0/src` directory created by unpacking the file is a file called `config.h`, which looks like (with some additional comments at the top):

```
#ifndef __CONFIG_INCLUDED__
#define __CONFIG_INCLUDED__

#define CPP     "/usr/local/bin/cpp"
#define VASM     "../../vasm/vasm"

#endif
```

The file comes with defaults as they were on my computer. You need to replace the paths listed here with the correct paths to `cpp`, the C preprocessor, and `vasm`, the VeSPA assembler. To locate `cpp`, simply run

```
which cpp
```

and copy the file listed to the CPP macro. See your instructor about installing `vasm`. It is easiest to use an absolute path for the VASM macro; it is the least ambiguous. However,

if you want to use relative paths, remember that it is relative to the `vcc-1.0/src/` directory, and not the parent directory `vcc-1.0/.`

Finally, to finish installation, simply run the install script:

```
% ./install
```

This script builds the `vcc` compiler and installs the resulting binary in the `vcc-1.0/` directory.

Additional installation notes

If for some reason compilation doesn't work, you may have to edit the file `Makefile` in `vcc-1.0/src/`. The only changes you should have to make are to the macro definitions at the top. For instance if you don't have bison installed on your system but you have `yacc`, simply change the macro `YACC` to be `yacc`:

```
YACC=yacc
```

Also, feel free to change the `CFLAGS` macro for different compiling options. However, do not remove the "`-include boolean.h`" option unless your compiler supports the `bool` data type (which gcc, the default compiler, doesn't). Add the `-g` option to `CFLAGS` to generate debugging information if you want to debug the compiler. Also, defining the TRACE macro will run the compiler with much more verbose output. This is also more for debugging purposes, so it's only really useful if you are modifying the compiler.

Beyond these notes, feel free to change anything you want, but at your own risk. Of course, it's always a good idea to back up anything before you change it.

Invoking the compiler

After installation is complete, you can test the compiler. In the `vcc-1.0/` directory there is a subdirectory called `testfiles/` which contains several sample C source code files. Lets test the compiler on a file called `ssort.c`. (This is an implementation of the selection sort algorithm.)

```
% ./vcc testfiles/ssort.c
```

This creates a compiled binary file called `v.out`. This file can be used as input for Verilog. This invocation produces an executable binary file, but there are several command line options which control the type of output, from stopping compilation without linking functions, to outputting the finally, ready to run binary file.

How the compiler works

This section is a very high level overview of the stages the compiler goes through in order to compile a source file. It also describes command line options available to the compiler.

Step 1: Preprocessing

The vcc compiler uses the program `cpp` found on most Unix systems to preprocess C files. This means that you can include comments and a few preprocessor directives. Below is a summary of features the cpp program provides.

Command Line options

You can pass the command line options C, D, E, I, P, and U to `vcc` which will then pass them to the preprocessor. Please see the man pages for `cpp` for a description of what each of these options does.

Preprocessor directives

You can use a number of preprocessor directives in your source code which the preprocessor will handle. Table 1 describes the legal directives.

`#define, #undef`	Macro processing
`#if, #ifdef, #ifndef, #elsif`	Conditional compile
`#endif`	End of conditional code

Table 1: Legal preprocessor directives

For more information about what each of these directives do, see the `cpp` documentation.

Step 2: Compilation

After the preprocessor has finished, the compiler proper begins parsing and analyzing the file. It generates the file `v.tmp`, which is an intermediary file containing assembly code and comments. If the file contains any errors or possible errors (warnings) they are printed to the standard output stream. If no error are encountered, the compiler then invokes the linker to finish the compilation stage.

Command Line options

`-t`	Do not delete the `v.tmp` file after compiling. (Useful for debugging.)
`-l`	Don't link. This option implies the `-t` option

Step 3: Linking

After the compiler parses and generates the v.tmp file, the linker is called so that all function invocations are linked to the proper program address. The output from this stage is either the file specified in the –o option (see below) or `v.asm` by default.

Command Line options

`-s`	Don't assemble and save the assembly file generated by the linker (`v.asm` by default). This option implies the `-a` option.
`-a`	Don't delete the file produced by the linker after assembling.
`-o` *filename*	Produce the assembly file *filename* instead of `v.asm`.

Step 4: Assembling

The final step in creating a binary file for VeSPA is actually completely separate from the compiler. The compiler, after all compilation and linking takes place, invokes the VeSPA assembler, detailed elsewhere in this book. See Chapter 5, *Building an Assembler for VeSPA* for information regarding the assembly process.

Part 2: The *vcc-C* language

Note: This section assumes familiarity with the C programming language, generally as defined in the ANSI and ISO standards.

`vcc` uses a very restricted but quite functional subset of the C programming language, which we will call *vcc-C* from this point on. Probably the most noticeable restriction is that there is no support for floating point operations. The reasons for this are two-fold: implementing a typing system would make writing the compiler much harder, and, more importantly, VeSPA only supports integer operations.

As a quick overview, table 2 lists all of the standard C keywords supported by vcc.

`void`	`int`	`while`
`for`	`if`	`else`
`break`	`continue`	`return`

Table 2: vcc C keywords

As can be seen from Table 2, the only data types allowed are void and int.* All of the keywords listed above follow all of the semantics of ANSI and ISO C.

* There is one important note to be made about the `void` type. It is included for the purpose of declaring void functions so as to avoid the compiler complaining about

As a quick note, then length of identifier names (i.e. functions and variables) is limited. However, it's limited to something like 128 characters, so unless you're one of those smart-asses who likes to do things just because he can—like create variables with names exceeding 128 characters—this should not be a limiting factor.

Operators

vcc-C also supports a fairly wide range of operators. Table 3 lists all of the operators in order of precedence, from highest to lowest precedence. The rightmost column indicates whether they are right or left associative.

1	`( ), [ ]`	Left to right
2	`++, --` (prefix only)	Right
3	`*, /`	Left
4	`+, -`	Left
5	`<, >, <=, >=`	Left
6	`==, !=`	Left
7	`&`	Left
8	`^`	Left
9	`|`	Left
10	`=, +=, -=, *=, /=`	Right

Table 3: vcc-C operators and precedence

Again, the operators assume their standard semantics for vcc.

Note: The multiplicative operators (i.e. '*', '/', '%', etc.) *do not* have corresponding instructions in the VeSPA ISA defined in this textbook. I have them for convenience, so if you plan to use these operators or arrays—arrays rely on multiplication to compute offsets—you must use a modified version of the processor and assembler.

Functions

Functions, for the most part, operate like their ANSI C counterparts. There is no specific limit to the number of parameters or local variables you can declare; the only limit is the stack frame size, which is set to be 512k, which translates into 128 4-byte integer variables, probably more than anyone will declare. Recursion is supported. However, functions cannot return arrays.[†]

`return` statements. However, as a consequence, you can also declare `void` variables. These variables will behave exactly like `int` variables—the compiler does not recognize types, so anything is an `int`—and using the void keyword to declare variables is highly discouraged.

[†] This isn't strictly true; it might be possible since you could just return the address, but the compiler doesn't "know" about returning arrays, and I've not tested this and thus make no claims to its feasibility.

The biggest difference is that you don't have to include function prototypes. In fact, the compiler will issue an error if you do. This does have the side effect of reducing the compiler's ability to make sure that function calls are made correctly, but since there are no types in *vcc-C*, this is not a big issue. The only problems can come when passing arrays, since the compiler doesn't check to make sure the arguments match up with the formal parameters. My suggestion to make sure the compiler checks arguments against the function declaration is to declare a function above any point where it's called.

Arrays

A special bonus to the *vcc-C* specification is arrays. *vcc* creates a heap high in memory which it uses to make arrays, and this heap is 1MB, which means you have space for up 256 elements for arrays.

Arrays are declared in *vcc-C* as they are in ANSI C, which only a few changes. First, all arrays are *static*, so you can't declare an array as a pointer and then dynamically allocate memory for it. The size must be a literal constant (or a macro if you prefer). For instance, the following declarations are legal:

```
#define kMyMacro 50
…
int array1[25];
int array2[kMyMacro];
```

While these declarations are not allowed.

```
int myInt = 5;
…
int array3[myInt];
int array4[];
```

However, the last declaration is legal if it is also initialized on the same line. So for instance

```
int array4[] = { 1, 2, 3, 4 };
```

Creates an array with four elements initialized to 1, 2, 3, and 4 respectively. Here, unlike in ANSI C, the compiler will issue an error if you also initialize the array's size. So

```
int array[17] = { 1, 2, 3, 4 };
```

Is not a legal declaration in *vcc-C*.

Arrays as parameters and arguments to functions

When declaring a function which accepts an array as an argument, you simply declare the parameter with the brackets following the parameter name. Here too, you cannot specify the size of the array (which you shouldn't be doing anyway); the brackets must be empty. So for instance, the heading for the quicksort function in `testfiles/qsort.c` looks like this:

```
void quicksort(int a[], int q, int r)
{
      …
```

Calling `quicksort()` is just the same as you would any function in ANSI C. Just use the name of the array without the brackets. As noted earlier, there is no support for returning an array, but since arrays are passed by address, they can be used as reference parameters; i.e., any changes made to an array's elements in a subroutine will be reflected back in the calling function as well.

Arrays and scope

The only issue relating to the scope of arrays is that currently they cannot be declared in the global namespace. This is, needless to say, a bug, and hopefully will get ironed out in the future, but for now, they don't work correctly if declared outside of a function.

Global and local variables

The only issues with global and local variables are with initializers and global arrays—or lack thereof. Variables can be initialized upon declaration, but it can only be to a literal constant. This is more of an issue for keeping the grammar simple. For example, the following initializations are legal:

```
/* Global area */
#define          X     5
int myGlobalInt = X;
int myOtherInt = 7;

int foo()
{
      /* Inside a function */
      int a = 7;
      int q = X;
      …
}
```

ANSI C only allows constant initializers for global variables, but you can also initialize local variables to values of other locals and parameters. But in *vcc-C* this is not legal:

```
int foo(int a, int b)
{
      int x = a; /* Will cause an error */
      …
}
```

Finally, as in ANSI C you can have nested blocks, at which point you can declare more local variables. This can be useful for having variables whose scope only extends for the duration of an if statement, for example.

I/O Operations

vcc-C supports two simple input/output statements: `getc()` and `putc()`. These two functions are actually built into the language, rather than being external functions in a library. They do behave just like their ANSI C counterparts. `getc()` takes no arguments and returns an integer. `putc()` takes one integer argument and outputs it to the output stream.

```
void foo()
{
      int a, b = 7;

      a = getc(); /* Gets an integer */
      putc(b);     /* Writes '7' to the output */
}
```

It is important to note that to use `getc()` and `putc()` requires a modified version of Verilog and the corresponding behavioral file. These are included with the *vcc* distribution.

Constants

The only reason why there is a section on constants is because of this important constraint:

> *There are no negative constants in* vcc-C.

The parser—as was detailed in the textbook this compiler is based on—does not recognize negative constants. This does not mean that negative numbers are not possible in *vcc-C*, it's just that they must be computed, such as by subtraction. e.g., 0 – 1 will compute to be –1. It's a pain, I know, and hopefully will be fixed in the future.

Part 3: Extending the compiler

Adding more functionality to the compiler is not that difficult, provided you don't want to make any major changes to the grammar. In this section I will walk you through the process of adding support for the modulus operator (`%`). Bear in mind that changing the

compiler sometimes entails changing the assembler (which is true in this case), and sometimes the Verilog description, but that is outside the scope of this section. We will assume that our assembler already has support for this operation.

Step 1: Analyzing what needs to be done

Probably the most common extension students will add to the processor is more functionality for the ALU. Adding a new ALU function is easy, because all arithmetic instructions have the same format:

```
      mnemonic    rst,rs1,rs2
```

or

```
      mnemonic    rst,rs1,#immed
```

This means that we can follow the same sequence of operations for say the `ADD` operation, and just substitute our new instruction in place. We will look at the existing code for compiling an `ADD` instruction and use that at a basis for our new instruction.

Step 2: Extending the grammar

Look at the file `parser.y`, which contains the grammar description for *vcc-C*. Line 433 defines a rule called binary which describes the format for a simple expression. Looking down a few lines, we see several productions that have the format

```
binary OP binary
```

where `OP` is an arithmetic operation. For example `binary '+' binary`, or `binary '*' binary`. All of these productions generate a call to a function `gen_alu()`. `gen_alu()` is designed as a template for generating any ALU operation (again because they all follow the same format). All of the calls look exactly the same. For example, the add and multiplication calls look like

```
$$ = gen_alu(ADD, -1, $1, $3, "+");
$$ = gen_alu(MULT, -1, $1, $3, "*");
```

The only difference being the mnemonics `ADD` and `MULT`. (Don't worry about the other parameters for now; you need to know about *yacc* syntax, but it's not necessary for what we're doing.) To add our new rule, we need a new mnemonic. Let's call it `MOD`. All we do then is copy the complete production for say the add operation, and paste it after the division rule (italicized code is existing code):

```
…
| binary '/' binary
      { $$ = gen_alu(DIV, -1, $1, $3, "/"); }
```

```
| binary '+' binary
      { $$ = gen_alu(ADD, -1, $1, $3, "+"); }

| binary '>' binary
…
```

Next, we need to change the symbols so that they describe the modulus operation:

```
…
| binary '/' binary
      { $$ = gen_alu(DIV, -1, $1, $3, "/"); }

| binary '%' binary
      { $$ = gen_alu(MOD, -1, $1, $3, "+"); }

| binary '>' binary
…
```

Step 3: Modify machine.h

The only thing left to do is modify the file `machine.h` so that the program knows about the MOD symbol. Open this file, and you will see a lot of macros defining the mnemonics for all the instructions in the VeSPA ISA. Simply add (anywhere in this file will work) a line like

```
#define          MOD   "MOD"
```

Step 4: Recompile and run

All that needs to be done now is to remake the project. In the `vcc-1.0/src` directory simply run the command make, and the project will rebuild with the new additions. If the assembler and Verilog description file have also been appropriately modified, you should be able to compile and run a program using the modulus operator. If not, you can still compile with either the `-t` or `-s` options, which will save the assembly file generated by the compiler and look to see that your code has been produced correctly.

If you are now feeling extra saucy, you can try to add the modulus-assignment operator (`%=`) as well. Again, just follow the example given by the existing operators (though you will need to create a new symbolic token in `parser.y`).

Summary

Following is a summary of the call syntax for vcc and a table summarizing all the command line options.

Call Syntax

(Items in square brackets ('[' and ']') are optional.)

```
% vcc [-CDEIUP] [-alst] [-o filename] source
```

Command line option summary

`-C,-D,-E, -I,-U,-P`	Options controlling the preprocessor
`-a`	Don't delete the file produced by the linker after assembling.
`-l`	Don't link. This option implies the `-t` option
`-o` *filename*	Produce the assembly file *filename* instead of `v.asm`.
`-s`	Don't assemble and save the assembly file generated by the linker (`v.asm` by default). This option implies the –a option.
`-t`	Do not delete the `v.tmp` file after compiling. (Useful for debugging.)

Known bugs and inconsistencies

This compiler has not been thoroughly tested, so I'm not aware of any bugs. As far as I can tell, it generates correct code, as long as you follow the guidelines in this document. Of course, there are bugs; every program has them. The biggest issue I'm aware of is the inability to use global arrays.

One other issue is with error reporting. Sometimes, because of the behavior of the parser and other parts of the compiler, it may report the same error twice in different ways. This is merely cosmetic, but I'll be working on fixing it.

If you find any bugs, please send me an e-mail, and I'll look into it. Of course, the more information the better. Please include the source file you were trying to compile and which command options you were using. Running the compiler again with the options `-a` and `-t` options and including the `v.tmp` and the `.asm` files is helpful, especially if the program is compiling but running incorrectly. If you find a bug and fix it yourself, *please* let me know so I can include it with the standard distribution.