The DoC Lab WACC Compiler: User Manual

Second Year Computing Laboratory
Department of Computing
Imperial College London

1 What is WACC?

WACC (pronounced "whack") is a simple variant on the While family of languages encountered in many program reasoning/verification courses (in particular in the Models of Computation course taught to our 2nd year undergraduates). It features all of the common language constructs you would expect of a While-like language, such as program variables, simple expressions, conditional branching, looping and no-ops. It also features a rich set of extra constructs, such as simple types, functions, arrays and basic tuple creation on the heap.

The WACC language is intended to help unify the material taught in our more theoretical courses (such as Models of Computation) with the material taught in our more practical courses (such as Compilers). The core of the language should be simple enough to reason about and the extensions should pose some interesting challenges and design choices for anyone implementing it.

2 WACC Compiler Reference Implementation

The lab has written a reference implementation of the WACC compiler which will enable you to explore the behaviour of the WACC language. The core of this compiler has been written in Scala, making use of the powerful parser combinator library `parsley` (`https://github.com/j-mie6/parsley`).

There are two ways in which you can access the reference compiler:

1. via a web interface that lets you upload a WACC program and set the compiler options (section 2.1);
2. via a Scala script that provides a programmatic interface to the web-service (section 2.2).

Each of these is explained in more detail below with a brief guide to help get you started with using the reference compiler.

2.1 Web Interface

A web interface to the reference compiler can be found online at:

- `https://wacc-vm.doc.ic.ac.uk`.

This site has a few useful features to help with discovery of the WACC language:

- WACC programs can be written directly in the browser.
- Assembly code from successful compilation can be downloaded directly, and has the expected name of `*filename*.s` if a file `*filename*.wacc` was uploaded.

- All of the compiler flags can be controlled in a way which will not lead to invalid configuration: these can be copied for use with the CLI using the button at the top-right of the options pane.

For those interested The website is implemented completely in Scala, with a Scala.js frontend using the Laminar UI framework connected to a Scala backend running the JDK21 HTTP server via a library called Tapir. Because both sides are written in Scala, they share code for JSON codecs and corresponding datatypes, which means they are guaranteed to always be in sync with each other. For fun, everything you see on the site has been hand-crafted only using the HTML and CSS primitives available – Laminar allows you to do the plumbing of reactive state through the site, but the UI elements themselves are up to you. In other words, no Javascript libraries required! Since the reference compiler itself is developed and deployed separately, there is no harm in open-sourcing this site to serve as an example of full-stack Scala development: the site source can be found at `https://gitlab.doc.ic.ac.uk/jhw419/wacc-new-web-dev` along with further architecture discussion.

2.2 Command-Line Interface

A separate Scala script, `wacc-reference-cli.jar`, can be found at `https://wacc-vm.doc.ic.ac.uk/wacc-reference-cli.jar`. A version is included in the provided Git repository for the 2nd Year Compiler Lab as well, though you should ensure that it is up-to-date by running `java -jar wacc-reference-cli.jar update`. The script can interact with the backend for the website via its API[1], and only requires that Java is installed. As it shares the same API with the website, it has all the same functionality available – note that the website always operates with the `--verbose` flag set, the CLI by default is silent on success. The flags and commands available can be printed by using the `--help` flag (it is also possible to run `check --help` or `update --help` too).

```
> java -jar wacc-reference-cli.jar --help
Usage:
    java -jar wacc-reference-cli.jar [--optimise <0 to 1>] [--target <arch> [--intel]]
                                     [--out <path> | --emulate [--stdin <string>]]
                                     [--print-assembly] [--verbose] <input file>
    java -jar wacc-reference-cli.jar check
    java -jar wacc-reference-cli.jar update

Compile a WACC program.

Options and flags:
    --help
        Display this help text.
    --optimise <0 to 1>, -O <0 to 1>
        Turn on optimisations at given level.
    --Opeephole-simple
        Enables simple peephole optimisation.
    --Ono-peephole-simple
        Disables simple peephole optimisation.
    --Oconstant-division
        Enables improved compilation of division/modulus with a constant divisor.
    --Ono-constant-division
        Disables improved compilation of division/modulus with a constant divisor.
    --Opeephole-subst
        Enables instruction substitution peephole optimisation.
```

[1]Technically, you could also use this directly if you wished, the documentation is found here: `https://wacc-vm.doc.ic.ac.uk/docs`.

```
    --Ono-peephole-subst
        Disables instruction substitution peephole optimisation.
    --Opeephole-mul
        Enables improved compilation of multiplication with a constant multiplicant.
    --Ono-peephole-mul
        Disables improved compilation of multiplication with a constant multiplicant.
    --target <arch>
        The assembly language to target.
    --intel
        Use Intel syntax, not AT&T (not available when --arch is set to ARM)
    --out <path>, -o <path>
        File file to output assembly into.
    --emulate, -x
        Run the program instead of generating assembly.
    --stdin <string>
        The input stream to feed to an emulated/interpreted program.
    --print-assembly
        Dump the assembly to stdout.
    --verbose, -v
        Makes the compiler print out more information during execution

Subcommands:
    check
        Only check if the program is valid.
    update
        Update the CLI.
```

The reference compiler parses the input file and, if this was successful, generates assembly code. The compiler requires the file to have the `.wacc` extension, though it is possible to drop this extension in the command line invocation.

The compiler has three modes of execution (this is broadly applicable to the web interface too):

1. By default, if the provided file is syntactically and semantically valid, it will generate an assembly file with the same name as the input file with the extension `.s`. By passing the `--out` flag, you can specify the desired output file name and location; by passing `--out tmp`, you can skip the generation of the file itself – this is useful in conjunction with `--print-assembly` if you just want to see the assembly printed.

2. By using the `check` command, you can instruct the compiler to stop before code generation and just check the syntactic and semantic validity of the program. The command `check --phase parse` can be used to stop before any typechecking occurs.

3. By passing the `--emulate` flag, you can instruct the compiler to run the program instead of producing an assembly file. If the program you're running requires input, you must specify it using `--stdin "..."` upfront. In future, the CLI may allow for interactive execution of a program on the server. All programs will have a maximum timeout of five seconds imposed – this is to ensure fairness for the server resourcs. While this mode does not produce an assembly file, the `--print-assembly` flag can still be used to see it on the console.

By default the compiler will produce assembly code matching your system architecture (but only for Linux machines), but it can also be configured to produce 64-bit or 32-bit ARM assembly code (`--target aarch64` and `--target arm32`) as well as x86_64 code (`--target x86`, though `x86-64`, `x86_64` and `amd64` are all supported aliases). The syntax of x86_64 code is by default AT&T style, though `--intel` can be used to switch to the (more palatable) Intel syntax.

2.3 Optimisations and Code Generation

Entirely unoptimised assembly can be idiosyncratic and heavily dependent on the higher level structure of the code generator, which can make it hard to follow. Because most students are not too familiar with assembly, the reference compiler has some of its optimisation flags set by default (known as `-Odefault` in the web interface). This has some lightweight *peephole* optimisations to remove a lot of the garbage, making it easier to follow. Other optimisation levels are supported, with `-O0` disabling all optimisations. It is easy to find out what flags a level sets in the web interface, where setting the optimisation level automatically sets the corresponding individual flags for you (it always sends `-O0` with these flags for implementation simplicity).

This section explores what the aim of each optimisation flag is:

- **`--peephole-simple`**: this flag enables a peephole pass which aims to find low-hanging fruit like consecutive moves between "scratch" registers and variables, or pushes immediately followed by pops. These just add unnecessary noise and junk to the output.
- **`--peephole-subst`**: this flag looks for possible instruction substitutions and applies them. For instance, in x86_64, `cmp` *`reg`* `0` is the same as `test` *`reg reg`*.
- **`--peephole-mul`**: multiplication is not *necessarily* a cheap operation on some architectures and can take many cycles to perform. If multiplying by a known constant, it *may* be faster to transform the operation into something else. For instance, `x * 2` can be represented by `x + x` and this is guaranteed to be cheaper. It is good to be careful with this and actually check the cycle counts as opposed to relying on "common wisdom". This flag performs this transformation where an appropriate cycle-saving replacement is known.
- **`--constant-division`**: on many hardwares, division is incredibly slow. Any division or modulus by a constant can be rewritten in terms of multiplication using *fixed-point arithmetic*. The algorithm used by Gcc and Clang for this has been implemented in the reference compiler, and this flag enables that for all constant divisons: this is good fun to play around with, but getting your head around how the algorithm works is very tricky!

Acknowledgements

The Wacc language and original reference compiler (implemented in Java, using the parser generator tool Antlr http://www.antlr.org/) were developed by Tienchai Wirojsaksaree as part of a Urop placement with Mark Wheelhouse over the summer in 2013. That compiler was later extended to perform various optimisations by Luke Cheeseman and Zuhayr Chagpar as part of a UROP placement with Mark Wheelhouse over the summer in 2014.

The web-service and scripting front-ends were developed by Mark Wheelhouse, with additional improvements later added by Luke Cheeseman and Zuhayr Chagpar.

A new (and significantly improved) version of the reference compiler (impelemented in Scala, using the parser combinator library `parsley` https://github.com/j-mie6/parsley) was written by Jamie Willis in 2022 and deployed behind the web-service interface in January 2023.

Over the years a number of students have contributed to the quality of the Wacc Laboratory by suggesting changes to language specification document and reporting bugs in the reference compiler implementation. A list of these students can be found on the "Wacc Bug Hunters" web-page at:

https://wacc-vm.doc.ic.ac.uk/bughunters