WACC Group 46 Project Report

Benson Zhou, Miles Nash, Sipan Petrosyan, Vincent Lee

1 The Final Product

Our WACC compiler successfully meets the functional specification for the project. The `compile` script takes a WACC file as an argument and, if valid, produces functionally correct assembly code for the ARM11 architecture:

- Source code first undergoes Lexical and Syntactical Analysis via ANTLR
- Semantic Analysis is then performed on the resulting tokens. This generates an abstract syntax tree (AST) to give us a succinct internal representation of the code
- Optimisations (if flagged) are then performed by traversing this AST
- Then finally, Code Generation takes place, producing an assembly file of the same name as the provided WACC file, the `.wacc` extension replaced with a `.s`

If instead the file is invalid, detailed error messages are produced. These notify the user of any syntactic, semantic and run-time errors detected during the process, including the line and column number of each error. If a syntax error occurs, the compiling process will halt before it reaches semantic analysis.

We are confident that our extensive use of object-oriented paradigms in our implementation, such as effective inheritance and interfaces, will allow us to add to our compiler with ease. We found this to be the case during the extension, where our abstract class for the AST allowed us to easily optimise and cross-compile the WACC code.

In terms of the performance of our compiler, we use a tree structure for both the AST and Symbol Table, to ensure quick look ups for variable types and the position they are stored on the stack. We have a series of methods in our Symbol Table class to lookup or update variables during optimisation and code generation. We also have a list of stack utility helper functions which help us easily calculate the required stack offset, and manipulate the stack pointer.

During code generation, we extended the existing Symbol Table and AST from the front-end to store additional useful metrics, like the total declared variable size and the total parameter size within each scope, as well as the stack offset position to store the next variable. This ensured efficient lookup and avoided recomputing the size of the scope every time, thus making our compiler much more efficient.

2 Project Management

What went well: Our continuous integration pipeline uses a GitLab runner on a virtual machine. Every time a commit is pushed to the remote repository, jobs are generated for the runner to build and then test the compiler automatically. This was useful to ensure that the project remained stable- if the pipeline broke, we would diagnose the problem and fix it (if possible) before moving on.

We used Git to great effect in this lab project. We created a new branch for every major feature, and merged them back to master when they have passed the test suite. This meant the master branch always has a stable version which passes all of the tests. The test suite was updated regularly to reflect newly added features. We also created branches for debugging and documentation before major milestones, allowing us to pair-program and develop on different branches concurrently, whilst preventing conflicts from arising between each other's work.

Our commit message standard is a header that describes which category the commit falls under (feature, fix, test, continuous integration, documentation, etc.), followed by a detailed description of the commit. For example:

```
feat: Implemented check method for ProgramAST
```

This fastidiousness ensured other group members knew exactly what had been implemented. When combined with daily communication throughout the project on WhatsApp, Teams or in person, this kept everyone up to date, and prevented time being wasted on implementing features that were already complete.
At the beginning of every milestone, we discussed what had to be implemented, the methods we could use to do so, and assigned every person a task. The majority of work was done in person, as we found pair programming especially conducive to understanding and debugging code, and we hosted meetings over Microsoft Teams whenever a member had to work remotely. This flexibility around each others' commitments is just one example of how tremendously well the group got on throughout the project, which certainly had a positive impact on the end product.

What we'd do differently: Setting up the CI pipeline was far from a smooth process, and we ran into issues with Maven dependencies and version compatibility issues. We endeavoured in resolving these before making any real progress with the code, as we recognised it would save us a lot of time throughout the project having a functional pipeline from the start.

Additionally, there were some issues surrounding DoC outages and lengthy LabTS test times. We worked around this by essentially replicating the LabTS test suite in our repository, allowing for us to test locally as well as in our pipeline. This was further streamlined by use of parameterized tests and a singleton caching of the reference compiler outputs, ensuring development could be promptly validated.

3 Design Choices and Implementation Details

Our WACC compiler is written in Kotlin, with ANTLR generating the lexer and parser. ANTLR is a powerful parser generator, and can generate an AST visitor class in Java. Kotlin combines object-oriented and functional paradigms, making it ideal for compiler development as it allows for pattern matching.

One of the interesting issues we faced during the implementation of our compiler was storing the state of the program throughout our AST. For example the stacks storing the free and used callee saved registers had to be passed down through the visitor. We did this by creating a class for the program state and passing it as an argument to GenerateASTVisitor. This made the table a field of the class, so all the visitor functions were able to call it's functions.

We initially thought to implement the data directives, runtime errors and the C library in a similar way. We then faced an issue when trying to call C library functions in the runtime errors class. To prevent this we created a companion object in the ProgramState class. Companion objects are singleton objects, however at runtime they are still instance members of real objects, so our companion object can call functions in the program state such as 'recentlyUsedCalleeReg' and can implement interfaces, which we anticipated may be used for extensions. This is very Kotlin specific code and we faced challenges trying to implement singletons in Kotlin, until we discovered companion objects.

Another implementation issue we faced when implementing the x86_64 cross-compiler was that register and addresses in x86_64 are all 8 bytes long compared to 4 bytes in ARM. We therefore had to compensate for this by carefully adjusting our code generation visitor to produce assembly code compatible with x86_64 and the GCC compiler. In addition to this issue, the x86_64 calling convention requires us to maintain a

16-byte alignment of the stack pointer when calling other functions. Thanks to our abstraction of stack utility functions, we could adjust for this neatly in the calculation and retrieval of stack offset helper functions, padding each variable on the stack with 16 bytes.

Throughout our development, we have ensured that we maintained the use of tree visitors and listeners in our front-end, back-end and extensions. This allowed us to clearly set out which methods and visitors we are overriding, and which visitors we are inheriting from the default behaviour. This helped us to not miss any implementations for the various visitors, and presented each walk-through of our AST tree in a neat and tidy manner.

We've used the template pattern in our implementation of AST nodes. There is an abstract class that contains methods such as check and accept, and acts as a supertype for all the AST nodes, such as If AST and expression ASTs. We also have classes such as ExprAST and StatAST, to further distinguish between expressions and statements.

We were able to reduce time spent writing complicated makefiles by using Maven for compiling our Kotlin code. It also took care of running our tests and was used in our pipeline.

4 Beyond the Specification

We have implemented several optimisations to our compiler, which can be toggled in the `./compile` command either separately using their individual flags, or all at once using the "-o" flag. Where possible, this functionality has been extracted to new classes in the `kotlin/optimisation` package, with tests to ensure that more efficient assembly code is generated.

Optimisation: Constant Evaluation

For all binary or unary expressions containing only literals, we evaluate them and return a single literal AST node, instead of the entire expression. For example, the expression "2 + 3 + 5" would be evaluated as the integer literal 10. This avoids having to push all the constants on to the stack and ing them all together, significantly reducing the number of instructions in the generated assembly file.

Optimisation: Constant Propagation

When we declare a new variable that is a Boolean, integer, character or string, we recursively evaluate the expression assigned to it and return the literal AST the expression represents. This is built on top of the constant evaluation optimisation. If the expression contains a variable, then we look up the variable in the symbol table and find the expression assigned to it, and evaluate that as well. We update the symbol table every time a variable has been re-assigned, to keep it up to date.

Optimisation: Control Flow Analysis

For control flow analysis, we analysed our code for "if true" and "if false" conditional branch statements, and simplified our code to just the branch that would execute. This was also done to "while false" loops to skip the body of the while loop if the condition is false. Since both if and while statements creates a new scope within the body of the branch, we made sure that a new scope was allocated on the stack to ensure that local variables declared within the scope would not affect a similarly named variable outside that scope.

Optimisation: Instruction Evaluation

Instruction evaluation inspects the generated assembly code, and gets rid of redundant instructions. For example, if there is a store instruction followed by a load instruction, both targeting the same register and from the same address, then the load instruction can be omitted. Furthermore, we eliminate any instructions that adds 0 to the same register. We also remove any instructions that moves the content from a register to itself.

Language Extension: Pointer Types

We extended our WACC with an additional C-style pointer type language feature. We included the '&' and '*' operator for referencing and dereferencing variables and array elements just like in C. We achieved this by modifying the ANTLR g4 files to include the operators in our lexer, and the relevant grammatical rules for pointer operations in our parser. We then modified our front end to check for common syntactic and semantic errors for pointer operations like dereferencing a non-pointer type or assigning a non-pointer type to a pointer variable. We created new pointer AST nodes and pointer types in our symbol table for code generation. We also included the ability to do pointer arithmetic which would add 4 bytes to an int pointer increment and 1 byte to a char or bool pointer increment.

Cross Compiler: x86 Architecture

We designed and developed a cross-compiler to also support compilation of WACC to 64 bit X86 instruction set architecture. The gcc compiler was used with the architecture flag set to x86-64 for 'a generic CPU with 64-bit extensions'. This option is activated in out compiler by use of the '-x86' flag.

Compiling into X86 had some benefits, one of which being the simplicity of the 'mov' instruction, in ARM we had different instructions for store, load and move, and a larger range of addressing modes. X86 simplified this translation, however since we had not considered this when designing the ARM compiler section, and did not have enough time to refactor our work, we were not able to implement a generic interface for the move instructions.

One of the big challenges we were able to overcome was that the ARM architecture was for 32 bit architectures, however the X86 was 64 bit. We set the length of data types based on the type of architecture and ensured our stack pointer moved the amount specific to the architecture. Our X86 compiler was also closely aligned with the X86 conventions.

If we had more time: we would have liked to have added a structure type, as we feel this would provide a great deal of utility whilst being relatively straightforward thanks to the structure of our AST.

Additionally, we would have liked to extend the scenarios in which constant propagation is useful, beyond when a new variable is being declared.

Furthermore, the optimisations currently run sequentially, thus when there are several optimisations set, the compilation time is reduced. Instead we could combine the traversals of the tree when there is more than one optimisation set.

Finally, we would like to allow for the optimisations to work with the x86 architecture.