Project Report

2. Product: Quality of our WACC Compiler

2.1 Functional Correctness

2.2 Future Development

* Does our compiler allow easy implementation of new WACC language features?

The way we built our Semantic analyser does not allow great extension of certain part of the WACC language. This is the area where we could have done better, especially in type checking, handled at a big cost. We were not creating a good enough internal representation of the mismatched types which could lead to a lot of time spent expanding the types of the WACC language. Nevertheless, the variable table handles very well variable tracking because of it’s structure that was extremely useful in the code generation. This table could therefore save a lot of time in extensions such as class implementation, where new forms of scopes are introduced.

* Does our compiler allow easy implementation of code generation optimisation?

The code generation is the part of the project our group is the proudest of. The generation of the Assembly commands is done in such a way that we are easily able to go over all of them and detect what can be optimized. This design allowed us to achieved some of the code generation optimisations in the extensions in a relatively short amount of time.

2.3 Performance issues

* Any slow, redundant processes?

The way our compiler is build requires 4 visits of the generated syntax tree after parsing the code. The first one is achieved by our syntax analyser to check for return keywords in the branches of the if statements. The second pass is achieved by our semantic analyser which checks all the semantic of the given code. The two last visits are made by the code generator to allow first the construction of an internal representation of the code and in a second attempt, by crawling this internal representation, generate the assembly code. For the sake of clarity and design, we decided to build our compiler this way as every steps were separated form another but this seem like a lot of redundancy in the crawling of the trees and it could have been done differently, with fewer visits of the trees, especially in the code generation where an intermediate representation wasn’t needed but served a lot for the comprehension of the codebase. This is a good example of design that took over efficiency.

3.1 Organisation of our group

* Splitting up the work

The task splitting was most of the time achieved by identifying the 2 separate tasks of every major task, in order to work in pairs. For the Front-End, 2 major task were notable: Lexer/Parser creation and Syntax/Sematic checkers. We decided for each of them to split the group in 2 and one was therefore achieving the Lexer while the other was working on the Parser. The same occurred for the Syntax/Semantic analysers and later in the back-end. The idea was to work with different pairs throughout the project while still working as a group overall for design ideas.

* Group meetings and coding sessions

On of the key aspect of the way we worked was to always try to be altogether to learn from other and merge ideas. We discovered that combining design philosophies and ideas leads to a great overall design, hard to attain on your own.

3.2 Use of Project Management Tools

* Version control with GIT

This project made our knowledge of git far more accurate and precise than before and had a strong impact on our efficiency. We decided to use git not just for what we thought it was for at the beginning of the project, a code storing cloud, but also to seriously keep track of the progress of the overall group through commit messages. This is where we understood the importance of every message we were submitting on git, which if badly written, could have a negative impact on the productivity of the group as people might overwrite each other’s work.

* The use of a communication tool: SLACK

At the start of the project, we decided to use a communication tool used quite a lot in industry nowadays: SLACK. This messaging application allowed us to communicate between each other for issues regarding the project. It is a good filter to make the difference between personal messages send though other applications such as Whatsapp which can stay unseen for a while and messages related to work and in our case, the project, which were important and required a quick response most of the time.

3.3 Reflecting on our project Management

* What went well

The contribution and the open-mindedness of the group in the design field of the project allowed us to achieve a great structure for the overall project. Everyone understood that it could and had to contribute to the brainstorming sessions at the start of every milestone, allowing us to quickly move forward with a clear idea of what we needed to do.

* What we would do differently

The area where we struggled the most and that we tried to overcome throughout the project was the readability of the code produced towards others. Many times, we managed to get stuck on a part of the project because we could not fix the problems of the code produced by one of our teammates, due to lack of comprehension. Nevertheless, everyone worked hard to overcome this difficulty during the project, making us more experienced programmers both individually and as a team.

4. Design Choices

4.1 Syntax Analysis Design Choices

# Corentin / Gregoire

4.2 Semantic Analysis Design Choices

* Using a visitor to check semantic consistency of the program

The way we decided to achieve the semantic analysis was using a tree visitor that crawled our built and syntax-free antlr tree. On every specific node, we performed the required checking, throwing using a special created class ErrorReporter (singleton pattern), which, given some arguments was always printing the same formatted message with different information depending on the type of the semantic error.

* Using an internal type representation

To overcome the problem of the types, we decided to build an internal representation of the types that could be place later in the variable table to be used throughout all the program. This representation especially helped in the code generation.

* Using a symbol table to store all the variables

The symbol table was the most important data structure of the whole project as it was used to link the front-end and the back-end. We decided the do a tree where each node represented a scope (containing variables which had a name, type and various other attributes) and where each child of the node was representing a scope inside the scope the node was representing. The tree was doubly linked as we needed to go back in the parent scope as much as going visiting the child’s scope.

* Using a general table holding the top Symbol Table and a function table

We used a general Table (singleton pattern) to hold throughout all the compilation the top symbol table (a dummy node with its children being the functions scope and the main scope) and a function table which was only a mapping of function names to a function representation (holding the type of the arguments taken, the return type and other information) also helpful for semantic checking and code generation. As this table could be access from anywhere, it helped us to keep track at any moment of what variables were in the specific scope and which functions could be called.

4.3 Code Generation Design Choices

* Command interface with hierarchy of superclasses to be able to reproduce all useful ARM commands

Knowing that we would have to optimise our commands in the extensions, we decided to create a Command interface (with a method generating the string representation of the command) and ARM instruction representations. Each ARM instruction was implementing the interface and therefore the method generating the string representation of the instruction that would be printed on the file. Therefore, all we had to do during our code generation is construct commands giving the right parameters and return a list of commands.

* FileWriter class which take a list of Commands and write them in the file

Once all the commands were created in a list, they were all passed to a FileWriter which was responsible for formatting the output file and writing the commands one by one on the file by calling the interface method generating their string representation.

* Factory Pattern to build the internal representation of the program

Having an idea of what we needed to produce (a list of commands), we decided to build an internal representation of the program (a tree). All we had then do was to call on the main node, a Program node a method generating the command for this program. The program itself was creating some commands and then in-between, when it needed to, was asking to its children to generate their own commands (without knowing what these children were). The key point of the structure was that each node was generating its own commands and was only asked by their parent to generate these commands.

* Expression, AssignRhs, AssignLhs and Statement interfaces which all have common generateCommands() method

The tree internal representation of the program had to be general as some node required an Expression, a Statement, an AssignRhs or an AssignLhs, all of which could take different forms. Therefore, when a node needed an integer expression for example, it was only given a reference to the interface Expression (which concrete type was integer expression) which had a method generateCommands(). This expression could generate its own commands as it knew its concrete type and implementation of generateCommands(). Therefore, the node could generate the commands needed for its own representation and when it needed to add the commands of the expression it had a reference to, it only called the method generateCommands() on this reference.

* Hardware Manager singleton pattern to manage registers and position of variables in stack/memory

Considering our structure, we needed something to communicate between a node and its child as it had no information on how the registers were managed and which one it should use to continue generating its commands. This is why we created a hardware manager that could be accessed from anywhere during the whole code generation to manage registers and tell the parent node which registers it should use to continue generating its commands. The hardware manager was also responsible for keeping track of variable location in the memory/stack in order to compute the right offset when accessing them in a specific command.

5. Extension