**WACC Project Report**

1. **The Final Product**
2. **Project Management**
   1. **Work Distribution**

The work distribution of the first two milestones in our group is quite different from that of the last. After discussing, we noticed that for the frontend as well as backend, most of the tasks had to be done sequentially as opposed to in parallel. For instance, syntax analysis for the compiler would have to be implemented before semantics analysis. Hence, we did pair programming as opposed to working separately and merging all the code in the end. We would often have productive programming sessions during the week and even over weekends. In each session, two people in the group would meet up and work on new features together, with one person typing and the other observing, and we would constantly switch roles. The advantages this technique brings is evident. Firstly, this led to more optimal design choices as the person that we worked with could often notice flaws or code that could be improved, which would otherwise be neglected if we had programmed alone. Secondly, debugging was much more efficient since logical errors could be found more easily when two people discussed and communicated their ideas.

For the last milestone, we decided to implement each new extension features separately, due to the fact that most features are standalone and independent of others. However, this did lead to an obvious issue, which is that the details of implementation of a new feature are only known to the group member who worked on it. Despite the drawback, working separately allowed us to incoporate more features in a wide range of areas.

* 1. **Use of Tools**

Git was used extensively throughout the project. It primarily served the purpose of creating and keeping track of branches for tentative features that were yet to be implemented correctly. For instance, new branches were initiated for each new extension feature we were planning to incorporate. Testing was mainly performed locally as opposed to on Git since we had some troubles installing the required packages for testing on our Gitlab runner.

Gradle is another tool which facilitated our project apart from Git. In the beginning, we put a lot of effort into configuring Gradle properly, and it was frustrating and quite time-consuming. Once set up correctly, the incremental builds provided by Gradle made building and testing our code much quicker.

* 1. **Adjustments for the Future**

There are a few aspects that can be improved regarding work distribution and communication in the last milestone. Firstly, the overlap between new language features and optimizations for code generation was not dealt with properly, which caused corresponding optimizations not to be implemented for some extension features, such as traits and **newtype**. Therefore, our planning will need to be more thorough in the future. Secondly, the details of implementation in our independent work will have to be communicated to every group member so that less confusion will be caused when trying to modify or debug someone else’s code.

1. **Design Choices and Implementation Details**
2. **Beyond the Specification**
   1. **Various Code Generation Optimizations**

There are three levels of optimizations which were implemented for the extension, which are constant folding, constant propagation and peephole optimization. To run the compiler with optimization, simply add the **-o{n}** flag as an argument to the compile script, where n indicates the desired level of optimization with 0 to 2 corresponding to constant folding, constant propagation and peephole optimization respectively. If optimization is turned on, the program will output the number of lines reduced and the optimization rate at the end of compilation.

We implemented both constant folding and constant propagation on the AST level as opposed to the conventional intermediate representation level. We opted for this approach since our intermediate representation for backend is not in the form of three-address code or SSA and switching from our original implementation to a more convenient IR was proven to be too costly. As a result, most of the optimizations were done by reconstructing the AST, and some of the more sophisticated optimizations such as dead code elimination and loop invariant code motion were not implemented.

* + 1. **Constant Folding**

Constant folding is done by traversing through the AST and evaluating binary operations as well as unary operations that contain only constants. Some examples are shown as follows (**could be moved to appendix if there is no space**):

|  |  |
| --- | --- |
| Before Constant Folding | After Constant Folding |
| **int x = (1 + 2) \* 3** | **int x = 9** |
| **bool b = true && false** | **bool b = false** |
| **int x = 10;**  **if (x < 2 \* 6) then**  **println 1**  **else**  **println 2**  **fi** | **int x = 10;**  **if (x < 12) then**  **println 1**  **else**  **println 2**  **fi** |

* + 1. **Constant Propagation**

Similar to constant folding, our implementation of constant propagation traverses through the AST and simplifies each node. However, instead of only evaluating expressions that consist of only constants, we record the values assigned to each variable, substitute its value into an operation if applicable and propagate through the whole program. However, substitution will not be peformed in a situation where the value of a variable cannot be determined. In addition, loops and conditional statements are also optimized in such a manner so as to guarantee that the post conditions of these statements are defined correctly. Examples are shown as follows (**could be moved to appendix**):

|  |  |
| --- | --- |
| Before Constant Propagation | After Constant Propagation |
| **int x = 1;**  **int y = x + 3** | **int x = 1;**  **int y = 4** |
| **int x = 12;**  **if (x == 12) then**  **println “x is 12”**  **else**  **println “x is not 12”**  **fi** | **int x = 12;**  **println “x is 12”** |
| **int i = 0;**  **while (i < 10) do**  **i = i + 1**  **done;**  **println i** | **int i = 0;**  **println i** |

* + 1. **Peephole Optimization**

Unlike the previous techniques, the peephole optimization is done on the intermediate representation of the ARM assembly, so that certain patterns can be clearly identified and optimized. The algorithm is rather simple as it goes through the instructions line by line and try to determine which optimizable pattern matches the current instructions. There are four types of patterns that are currently simplified, which are demonstrated in the table below (**could be move to appendix**):

|  |  |  |
| --- | --- | --- |
| Pattern | Before Optimization | After Optimization |
| Multiply by a number that is power of two | **MOV rn, #n**  **SMULL rm, rn, rm, rn**  **CMP rn, rm, ASR #31**  **BLNE p\_overflow\_error**  **STR rm, [offset]**  Where is an integer that is the power of two | **LSLS rm, rm, #**  **BLE p\_overflow\_error**  **STR rm, [offset]** |
| Divide by a number that is power of two | **MOV r0, rm**  **MOV r1, #n**  **BL p\_check\_div\_by\_zero**  **BL \_\_aeabi\_idiv**  **MOV rx, r0**  Where is an integer that is the power of two | **LSR rm, rm, #** |
| Load a constant as an argument | **MOV rn #n**  **MOV r0 rn** | **MOV r0 #n** |
| Redundant load after store | STR rn, [offset]  LDR rn, [offset]  where the two offsets are the same | STR rn, [offset] |

* 1. **VS Code Extension**

Since WACC is a language designed purely for academic purposes for the compiler course, we could not find tools to help users code in WACC. Hence, our group decided to build an IDE to help programmers code in WACC more efficiently. Since it would be too costly to create one from scratch and it would not be easy to popularize the tool, we opted for VS Code as a platform to develop and deploy our WACC IDE. To start, we created syntax highlighting for WACC. Then, we utilized the sample language server for VS Code extension development as a basis for our IDE, and gradually customized the server according to the specifications of the WACC language. The core of the IDE is our own WACC compiler, as we parsed the output from the compiler and generated code diagnostics based on those. In addition, the users could also execute the WACC file directly in VS Code. Since it was difficult to append an ARM assembly emulator to the extension, we took advantage of the provided reference emulator script. Thus, every time a file is to be executed, it will first be compiled into assembly by our compiler, then the assembly code is sent to the reference compiler. Lastly, the language server parses the output generated by the emulator and displays the messages in VS Code. The IDE currently has four main features, which are syntax highlighting, syntax and semantic diagnostics, auto completion and code execution. It has to be noted that the IDE only supports the most basic WACC language with no other features added. We decided not to include our own features as we thought it would be better for the IDE to be an opensource project which everyone can customize and distribute according to their own flavour of WACC. To access the VS Code extension, simply search for “WACC language support” in the marketplace and install it.

* 1. **Ideas for future extension**

If more time was given, we would possibly redesign the backend to use SSA as our IR so that more optimization options such as dead code elimination would be easier to implement. Moreover, we would want to make the import feature available so that we could build our own standard libraries for WACC. (Feel free to add your own thoughts)