LANGUAGE PROCESSORS

ASSESSED PRACTICAL

C TO ARMV6 COMPILER

Fernandez Buitrago,
Andrea

CID: 00732133
AF1312@ic.ac.uk

INDEX

1 INTRODUCTION

1.1 Overview

This assignment consists in creating a program that compiles from C code to the ARMv6 code that the Raspberry Pi utilises. This compiler will be referred to as GSISI from this point on.

The files that make up GSISI are the following:

c.cpp : Main file. Contains the main function, which calls the parser.
Util.h : Contains support functions.
c.l : Lexer file. Contains all the tokens, generated using regular expressions.
c.y : Grammar file. Contains all grammar, and translation into ARM code by calling member functions of Parser.
Parser.h : Class file for the parser. Contains all functions called in 'c.y'.
README.txt : Contains special instructions needed for GSISI to run.

This report will explain how all these files work individually and together, explaining all design choices, and showing example code when relevant.

1.2 Tools used

In order to facilitate the development of GSISI, tools where used to turn the rules of C into compilable C++ files. The tool used in order to generate the lexis was **flex++**, and the one to generate the grammar was **bisonc++**. To compile both files into an executable, **g++** was used.

The versions and operating system used are told in detail in the file **README.txt**.

1.3 Scope of the project

Due to the flexibility that C offers as a language, and time constraints, the whole of C is not implemented into GSISI. The main aspects that are included are the following:

- Variable assignment.
- Function calls.
- Function declaration and definition.
- Control statements (if/elseif/else, while/for/do while)
- Pointers and arrays
- Some characteristic C functions, such as printf, free, malloc...

A full explanation of these and of any limitations are provided in this report.

2 GRAMMAR AND LEXIS

2.1 Lexis

The lexis for GSISI is contained in the file **c.l**, generated using **flex++**.

This file recognises and generates the tokens used by the grammar file. These tokens are defined in the grammar, and the lexer returns them when their regular expressions are met.

This is a short file since C does not have a great number of different tokens, preferring to reutilise those that have already been defined. An example of this is the '*', which is used for both multiplication and declaration of pointers.

Ne of the only design choices taken for the lexis of GSISI was the avoidance of declaring known types and C codewords as tokens, such as 'int', 'char', 'printf'... The reason for this is to provide a much greater flexibility in the grammar. Since types and functions and not restricted to these known codewords, it is much simpler to recognise none of them as tokens, but to recognise a certain pattern as a variable declaration, or a certain pattern as a function call.

These entails, for example, not defining a variable declaration as 'type id', but as 'id id', since any id can be declared as a type. While this makes for a more difficult implementation, it allows greater flexibility for type definition and structs.

Another design choice was to define the semi-colon ';' as the character to signify the end of line. This was done in consideration to how C is defined. C ignores all new lines, but keeps a strict tab on semicolons, as they mark the end of a statement. Seeing as how a statement is a line of executable code, and how a semicolon marks the end of it, having the semicolon as an end of line makes sense.

2.2 Grammar

2.2.1 Overview

The grammar for GSISI is contained in the file **c.y**, generated using **bisonc++**.

The grammar file divided all C code into two categories: main and everything else. The second category includes but is not limited to functions definitions, function declarations, variable declarations and type definitions.

All of these except the last share the same characteristic: they take arguments in some way or another. These arguments have to recognised using recursion, since their amount is not fixed – a function may have no arguments, or only one argument, or as many arguments as it needs. A special case is applied to main, which, if it takes arguments, have to be of a specific type.

Another characteristic that most of them share is that they have a body. This is called 'function body' throughout the grammar file and is always the same for every type of structure: it consists of one or multiple lines, where a line is an executable statement in C. A line itself can be this function body, since the only thing it indicates is a change in scope. This is reflected in the
C++ code for GSISI.

A line can be many things in GSISI. It can be an expression, which is just a mathematical expression operating on two 'types'. These types are an important element of the grammar.

They basically consist of anything that has a value, may this be an integer, a character, a string, a variable or a function call. A line can also be any control statement, the most notable of which is the 'if'.

An 'if' statement can have an unlimited amount of 'else ifs', which means it has to be defined by recursion. It does not share this with 'for' and 'while' loops, which always have the same fixed structure (aside from function body, which can be anything).

Another important element are function calls. They are defined in almost the same way as function declarations, in the sense that their structure is composed by an 'id' and a set of arguments, which are recognised using recursion. A main difference between them is what they take as arguments. A function declaration takes only variable declarations as arguments, while a function call takes any 'type', anything with a value, as an argument, which makes it trickier to handle.

An essential design choice that was taken later on in the development of GSISI was the separation between variables used in the LHS (left hand side) and variables used in the RHS (right hand side) of an expression. The syntax recognition is almost the same, with the exception that no element of the LHS of an expression can be incremented or decremented with "++"/"--". The difference is in the action to be taken for each side of the equation, which will be further explained in the next section of the report. While on the RHS the value of the variable itself is needed, in order to operate on it, the LHS, or lvalue, as it is called in C, needs only the address of the variable to store the RHS in.

This is why, even though their syntax is almost exactly the same, it makes sense in the design to separate both of them, since different things are needed of each.

A similar things happens with variable declaration, which differs if inside a function body or if as argument of a function. While the syntax is exactly the same, 'id id', the context calls for a separation, since they need to be stored in different ways. This will also be further explained and justified in the next section.

Please note that in the context of this report, an 'id' refers to a variable called by its name, and also to any other reference to it, be it by using it as an array (a[0]) or de-referencing it as a pointer (*a), as well as getting its address (&a). Since this is not the variable itself, which has to be loaded first so that these operations may be done, these operations have been defined in the grammar as 'extended_id', since they provide an extension to what a variable can be called by. In this report, though, to simplify the matter, 'id' will be used to reference both the variable itself and any extended operations that may be performed on it, but not on any other temporary variable (such as an 'int 1', which you may not get the address of).

Multiplication has not been implemented because of syntax conflicts that could not be resolved due to time constraints. The reason for this is that a multiplication and a variable declaration can have the exact same structure in terms of tokens:

```
id ASTERISK id ;
```

A way to resolve this would be to not use a separate non-terminal for pointer variable declaration, but to set up a special case of multiplication for it. This special case could be that if the first id is the name of a type, consider the multiplication to be instead a variable declaration. This could be implemented in future versions.

2.2.1 Return type

The default return type of a non-terminal in bisonc++ is an integer, but a more elaborate type is needed for GSISI. Because of this, the return type of non-terminals was redefined to be a custom class, class Return, which can be found in file **Util.h** .

This class has the following members:

- A description field, d, for any description needed to describe the non-terminal.
- A register field, reg, to specify the register the non-terminal is stored in.
- A label field, lbl, used for variables and functions, to return the label with which they are stored in memory.
- A value field, val, for the value of the non-terminal, when known.

These fields are all optional, and have defaults set for them. The description field, while it is meant for general use, mostly returns the type of the non-terminal, for easier type comparing. The label field is mostly unused, but useful when dealing with variables that have been declared but are not currently in a register, or whose register we do not know at the moment. Returning the label is more accurate than returning the variable name, since variables in different scopes may have the same names, but label names are always different from each other.

The value field was a late addition, since most of the time the compiler needs not know the values of the variables it is working with, only where they are stored. This presents a problem when declaring arrays, though, since the exact value is needed, to be able to know how much space is saved into memory. This is why this field was added to the return type, and will be further explained in the next section.

The main element, though, is the field that holds the register name. Knowing the register, all information can be known about a variable: their type, name, label, arguments... This is why this field is so important. Still, other fields are needed for when the register is not known at the present (like, for example, when calling variables that have already been declared).

2.2.2 Recognised syntax

A full list of the sections of the grammar for GSISI follows:

- Main
- Not main, which can be:
  - Structures (recognised and partially implemented, cannot be operated on)
  - Function definition
  - Function declaration
  - Variable declaration
  - Type declaration

Most of which have:

- Arguments (special case for main)
- Function body

A function body is a series of 'lines', which can be the following:

- Function call
- Expression with no LHS value, with the priority of operators implemented
    - sum, subtract, bitwise or
    - bitwise exclusive or
    - multiply (not implemented), division(not implemented), modulo, bitwise and
    - size of, bitwise not

- Comparison of two expressions
- Variable declaration (which can be a list), with optional storing
- Storing of an expression in a variable
- A control statement
    - if/elseif/else
    - do while
    - while
    - for
    - switch (recognised but not implemented)

- A return, which can return an expression or not
- A break statement (recognised but not implemented)

An expression operates on types, which can be:

- Function call
- extended_id (RHS variable)
- Float (recognised but not implemented)
- Char
- String
- Int (signed and unsigned)

A new variable can be:

- new variable (declared inside function body)
    - pointer
    - array (is pointer too)
    - none of the above
- function variable (declared as argument of a function)
    - pointer
    - array (is pointer too)
    - none of the above

An already declared variable can be:

- a_id (when found in LHS of expression)
- extended_id (when found in RHS of expression)
    - de-referencing ('* id')
    - array indexing (' id [expr] ')
    - address getting ('& id')

Both of which can be a pointer or an array.

3 C TO ARMv6

3.1 Variable storing

Variables are a key element in C. The three types that are implemented in GSISI are the following:

- Variables declared in a scope
- Functions
- Structures

A data structure is needed to store any variables that have been declared. A key element of this structure is that it must be able to differentiate between the different scopes the variables are created in. Because of this, a simple vector does not suffice, since a key must be kept, which has to be used to signify the scope that the variable was created in.

Another option is a map, which has a key and a value, but the main issue with a map is that it can only have one value per key, which is not what is needed in this case. There is more than one variable per scope, so the ideal structure is a multimap, which is the same as a map but allows more than one element with the same key.

This is the reason why a multimap was used as the main container for the declared variables, since it keeps both the data about the variables and the scope they were declared in. While a more efficient option could be to define our own container structure, the main advantage of using an already defined container is that a great amount of functions have already been defined for them, since there are whole libraries for vector, map, multimap... Aside from this, algorithm libraries can be used for this container types, instead of having to define such functions.

So, the main structure where variables are saved is a multimap. The key is the scope, but the value that is stored has yet to be defined. A new class has to be defined, a class which contains all the necessary information to identify a variable. Still, there are different types of data that need to be stored. Variables need to be stored, as well as declared functions and structures. Because of this, the most efficient solution is to create an interface class, which has been called DataInt (from data interface), from which all data types can inherit.

This DataInt class contains the basic information that all data types have in common, which is the type of the data, the name of the data and whether the data is a pointer or not. A variable class inherits from it, adding a member which contains the number of the register the variable has allocated, and whether the variable is currently in that register. A function class has a vector of arguments, and the same is true for a struct class, which also has the added member 'size', for the size of the structure in memory.

In order to be able to reference these three classes we use the interface DataInt. So, what is saved in the value of the multimap is a pointer to DataInt*, so that a variable, a function and a struct may be added in the same multimap.

After declaring a variable, it is stored into memory with an unique label. In order to guarantee it is unique, a counter is kept, and it is increased every time a variable is stored into memory. Since the memory can only be declared at the beginning or at the end of an ARM Assembly file, ARM code for memory is temporarily saved into a vector of string called 'data', which is appended to the rest of the code after compiling has finished.

The default value saved into memory for all variables is 0. If no initial value wants to be set, the instruction '.skip' could instead be used, which just assigns some memory for the variable. The size to be assigned to the variable is gotten from the size which corresponds to the name of the type. Types and their sizes will be explained further on, in section 3.4.2.

This variable map stores only declared variables, but not temporary ones, which are needed for any operations. Another container is needed for temporary variables, but for this one a vector will suffice, since there is no need to keep track of the scope, as these variables will be deleted upon use.

Whenever an operation is needed on a value, this value needs to be temporarily assigned a register. It is then pushed into the vector of temp variables, and taken out when used, and also by default at the end of the statement. This vector contains DataInt*, same as the multimap.

3.2 Register assignment

There is not a great amount of registers to store data in, so efficient register assignment is key. Registers r13, r14 and r15 are reserved by ARM for the stack pointer, the link register and the program counter, respectively, so that cannot be used. As for registers r0 to r3, they are used as arguments for function calls, with r0 being reused as return.

One option would be to use registers r1 to r3 normally, and when calling a function, to store their values, loading them again when the function is done. This would be quite inefficient, though, since operations to memory take 4 cycles per storing/loading. Because of this, I have opted for ignoring registers r0 to r3, and only storing values in r4 to r11, since r12 is overwritten by some functions, such as printf.

In order to represent the registers, an array of DataInt* was created, were the register number is represented by the index. This decision was taken with the idea in mind that knowing the register a variable is in, all information about that variable is available. This does cause some problems, though, since there have to be cautions in place so that a null pointer isn't dereferenced by mistake, which would cause a Segmentation Fault.

As well as the array of registers, a variable regn, member of Parser, was also created, which is used to signify the next register that is empty and free to use. So, when a variable needs a register, it calls the function save_reg(), which stores the variable in the next free register (which has index regn), and finds the next free register, storing its index into regn. A free register is represented by one that has a value of '0' in it.

When there are no more registers, some must be freed. The algorithm used takes into account the following:

- A variable currently in the vector tempvar cannot be overwritten, since it is currently in use.
- A variable currently in scope cannot be overwritten, since it is currently in use.
- And a variable in the vector stack, to be covered latter, may also not be overwritten.

Any variable that is not in any of these three places may be overwritten, and the register freed. In the case that no registers can be freed, all registers are emptied, and a flag is set that the variables are not currently in register.

3.3 Control statements

Control statements add comparisons and branches into the code. The comparison is done at the beginning of the statement, and if it does not hold true, a branch the end of the statement has to occur.

To achieve this, we need to add a label at the end of the statement. Since there may be multiple control statements, and labels have to be unique, a counter is kept. Each control statement creates the same labels every time, distinguished only by this counter, which is added to the label.

So, the code needs to branch to this end label if the condition is not met. That is, if the condition is 1 == 1, the code needs to branch on the condition that they are <u>not equal</u>. Since this could lead to error if reversed manually, a function was created to provide the correct branch condition, as well as one that just checks the condition, without reversing (for normal comparisons).

So, in the case of 'while' and 'do while', the condition is checked, the code branches to the end of the loop if it's not met, or else it continues on with the code. At the end of the code another branch is added that branches back to the beginning, to loop the code over again. Because of this, another label needs to be put at the beginning of the loop. For 'do while', only one branch is needed, since the code is never skipped.

In the case of a 'for', there is one added complication, in the sense that code cannot be outputted as soon as a certain grammatical rule is met. A 'for' loop has an increment or assignment as part of its condition, but this increment is not done as soon as it is seen, but at the end of the loop. Previously to adding the for loop, ARM code was output to the standard output as soon as it was seen. But with this addition, this could not be done, as the increment had to be done later on.

Two vectors of strings were added, 'code' and 'tcode'. All ARM code is pushed into 'code', and then printed out at the end of parsing, or when an error is found. If any code has to be moved from place, the position of the first element to be moved is saved in a member of Parser. The code is popped out of 'code' and moved into the vector 'tcode' as temporary code. When the code code is ready to be implemented, it is then appended to the end of 'code'. This allows for 'for' loops.

'If's present an added complication, since they can have an infinite number of 'else if's. More than one counter is required then, one to keep track of the current 'if', and another to keep track of the current 'else'. In an 'if', the condition is checked, and if it does not hold, a branch is done to the next 'else', comparison done again, goes to the next 'else', etc., until there are no more 'else's left. If a condition is true, the code inside the current if/else is executed, but after that, there must be branch to the end of the code, or else all the other 'else's will be executed afterwards. So, at the end of each 'else', and before the next, a branch to the end of the control statement is needed.

This algorithm does not allow for nested control statements. When a control statement is entered, the label counter is increased, but it is never decreased. So, if there was another control statement inside, the counter would increase, and the outer control statement would create labels with this same value, even though it is not the one that corresponds to it. On the other hand, it is not an option to decreases the counter when going out of a control statement, since this would not allow these statements one after the other, as they would all have the label counter of 1. This is because they increase when they enter, and they decrease when they go out.

Another more elaborate solution would have to be implemented.

Not fully implemented are multiple conditions of the type 'a||b' and 'a&&b'. Precedence is not currently checked, and no brackets are allowed. Because of time constraints, this could not be finished. Currently, double conditions function only if a full comparison is done ('a==1'), and have a limit of one 'else if'. For a single condition, there is no such limit.

For multiple conditions, the comparison is done, and a register set to 1 if it the condition holds. The same thing is done for the second condition, and both registers are compared (either with 'AND' or 'ORR'). If the result of this comparison is '1', then the condition holds true.

3.4 Functions, structs and typedef

3.4.1 Functions

A function is defined by its type, its name, and its arguments. The function str(), as with all classes that inherit from DataInt, produces a string that identifies the function. Any function two functions that produce two different strings may be declared. That is to say, GSISI allows overloading of functions, since functions with different arguments and/or type are considered different functions.

Arguments are identified only by their type, since names may differ between function declaration and function definition. Currently, a function may be defined and later declared, giving only a warning that it has been done. An implementation of declaring and later defined has been put in place, but is not in working order at the moment.

The basic idea for the algorithm is the same as with the 'for' loop: save the position where the definition started, pop the code out and push into a temporary vector 'tcode', and then later push back where appropriate.

A function may also have one or multiple returns. If a function has not been declared as void and there is no return, a warning will be issued. Each function has a member hasReturned, which indicates whether a return has happened. There is also a member currentFunction in Parser, which keeps track of the current function. If NULL, the current function is main. When a return is detected, the link register is loaded into the PC, and a variable is saved into the return register r0 if a variable is being returned.

The type of the return parameter is compared with the type of the function, and a warning is issued if they do not match. Currently, recursion is partially operational, with the limitation if the value of one of the arguments is required after the recursive call, it will have been overwritten. A possible solution would be to store the arguments before the function call, and load them back when the call has ended, but it would be inefficient.

When entering a function that is not main, all registers are saved in the stack, so as to be able to use them freely, and they are loaded back when the function is finished.

Functions may have more than three arguments. If this occurs, these extra arguments need to be stored in the stack, and then loaded into registers inside the function itself. This is currently operational in GSISI.

Aside from the vector of temporary variables, there is another vector of DataInt* that keeps track of the variables currently being used, and that is 'stack'. Stack is used only for function arguments. In a statement, every 'type' found is saved into the stack. At the end of the statement, everything in the stack is considered an argument of the function. Stack and the vector of temporary variables, tempvar, could be fused into only one vector of DataInt*, but this is to be added as a future improvement.

As a last note, when a function is called, it can be a function defined in a C library. Since there is no linker available to us in this practical, it is not an option for us to link the libraries to the functions being called. Because of this, special cases have to be set up for library functions such as 'printf'. No checking is done on these functions, except an argument check if the function requires it.

Since these functions require the library to be included before using them, a flag is set if the proper include was added. If the function is called, but the proper library was not included, a warning will be thrown.

3.4.2 Type definition

Type definition is implemented in GSISI. GSISI has a list of all valid types, stored in a vector called 'typel'. This vector contains elements of the class Type, which has all essential information about a type.

A type may have two elements: a name and a size. The size is relevant for storing in memory, since it is necessary to know the amount of memory to reserve for a variable. As for the name, typedef creates the possibility of there not being one unique name for a type. A type may be referenced by many different names, which is why the element name of a type is implemented as a vector of strings, with each element being one of the possible names that may be used to reference that class.

When typedef is called, a name is added to the list of possible names for a type. When the defined type is a pointer, a flag is set for the type, represented by the boolean isPointer. When a variable is declared, this value isPointer from the type is searched for, and if the type has been defined as a pointer type, the variable itself inherits its characteristic as a pointer.

When two types are compared, all their possible names are gone through, and if any two of them match, the types are considered compatible. If not, a warning is issued.

Currently, there is no measure implemented to ascertain whether the name of the type that is being defined already exists.

3.4.3 Structures

Structures are currently not fully implemented in GSISI due to time constraints. They can be defined, but not used or declared within the code.

Structures are added as a new member of the list of types, unlike typedef which adds a new name to a pre-existing type. The size of the type is calculated from the size in memory of its components.

So, the following structure:

```
struct Foo {
        int a;
        int b;
};
```

Has size 8.

3.5 Pointers and arrays

Adding pointers into GSISI required an overhaul of the whole storing method. When pointers can be used, it is not possible for the value of the variable to be loaded only once, and then operated on – since at any time, a pointer could have changed its value in memory.

Because of this, every time a variable is reference, its value has to be loaded from memory, to make sure that the correct value is obtained, and not an outdated one. This means that is much more efficient to assign a register for the address of a variable, rather than to the value of the variable itself.

So, when a variable is assigned a register, only its memory address is stored in that register, since it is what will be used the most. This is because, in order to store a value into memory, only the address is needed, and if referencing a variable, there is a need to load the variable from memory anyway, since a pointer may have changed its value.

Currently, in order to signify that the variable is a pointer, GSISI sets a member isPointer to true, which is by default false. This makes type checking with pointers tricky, since it's not only the name that has to be checked, but whether a variable is a pointer or not. It also causes problems when a variable is a pointer to a pointer, since the flag can only be set once, and is set back to false when de-referencing happens. A pointer to a pointer cannot currently be attempted in GSISI.

As for arrays, they are an added complication for the compiler. Up until now, the compiler did not have to keep track of the value of the variables, as it does not have to perform the operations itself. The compiler normally needs only know the type and name, and which register they are assigned to, in order to generate the code.

But, for arrays, the space saved in memory depends on the value present between the square brackets. This is the only cause in which the compiler has the need to know the value of an expression. Currently, GSISI can only calculate the value of simple expressions, such as add, multiply... It cannot deal with a variable inside the square brackets, since there currently is no way of tracking where changes have been made to the variable. It is also not able to deal with a function call being inside the square brackets, since it would have to execute the code of the function in order to calculate the value, and this is not kept track of.

Aside from these differences, pointer and array declaration work in the same way as non-pointer variable declaration. In arrays, an offset is applied to the address of the variable, and in pointer de-referencing, instead of loading the value of the variable, the value of the variable is treated as an address and loading is performed twice.

3.6 Error checking

GSISI raises two types of errors: an error and a warning. Errors cause the program to stop compilation and print all the current code, while warnings allows the compilation to finish.

Errors are only raised for operations that are not allowed in C, such as treating a non-pointer as a pointer. Errors are also caused by errors within GSISI itself and not the code that is being compiled. If GSISI has an unexpected error from which it cannot recover, it will raise an error.

An example of such errors are a pointer unexpectedly being empty, or doing operations on a register that does not exist (-1 being the default for a not found register).

Warnings are raised for not completely correct statements that can still be compiled by C. Some examples of this are a function with a return type that doesn't return, or adding different types.

GSISI does type checking and type comparing for all operations, as well as checking whether a variable has been defined in the current scope.

GSISI also checks whether function arguments are correct, for user-defined functions. The number of arguments is checked, as well as the type of the arguments, comparing them with the ones stored in the variable map.

3.7 Performance

Performance of the generated ARM code is average, since there are not many useless instructions performed. Still, the use of pointers slows the code a great amount, since a variable has to be loaded every time it is to be used, and that takes 4 cycles instead of the normal 1.

An option to improve the performance would be to detect whether a pointer has operated on the variable, and to only load if this has happened. This would require a way to keep track of which variable a pointer points to, which cannot currently be achieved using GSISI.

Another improvement on performance would be to use constants instead of registers for the second operand of an instruction. This would save registers as well as improve performance, but it would require accurate knowledge of the value of the second operand, which cannot currently be achieved with GSISI.

Another option would be to detect whether one of the operands is a temporary variable such as an integer with value 1, and replacing the register number with the value itself. This wouldn't remove any instructions though, since a 'move' instruction would still have to be performed every time an integer was seen, even if the register didn't have to be used later. A different option would be use a different non-terminal for the first operand and the second operand, such that the second one would not be saved in a register but would just return a constant. This still causes a problem if the second operand is a variable, since its value cannot be known without doubt in the current structure of GSISI.

GSISI tries to keep any stack operations to a minimum, which is why registers r1 to r3 are never used, in order to avoid having to store their values into the stack. For LHS values, no load is performed, since it is not necessary, which improves performance greatly.

In conclusion, GSISI's performance is adequate, but could be improved upon.

4 CONCLUSION

4.1 Future improvements

There are many characteristics of C still to be implemented in GSISI, some of which have been mentioned during the body of this report. A number of different types still have to be implemented, such as floats and doubles, and variables can be declared in different ways, such as 'static' or 'global'.

While all of these are worthwhile improvements, the main improvement to be done in GSISI is to reduce the number of corner cases that cause it to fail at compilation. Register saving is a main cause of this, and an invalid use of pointers is another. Before making any further additions to GSISI, I believe it would be better to improve the reliability of the compiler, rather than the extent of the language that it can compile.

Reliability, however, is much more difficult to implement that new grammar rules or new structures, and it could not be done within the time constraints. It is still something to be taken into account in future projects.

4.2 Conclusion

GSISI is a C compiler that accomplishes its job of generating ARM code for the common cases that happen during a C program, but still has a long way to go to make it reliable. It is particularly vulnerable to Segmentation Faults because of its extensive use of pointers.

Still, GSISI can compile simple C programs without problem, and passes all eight samples provided, but might not pass any corner cases or have random errors.