Universidade do Minho
Escola de Engenharia

Master degree in Industrial Electronic Engineering and Computers
Specialization in Embedded Systems and Computers

RocketC

C Compiler for the VeSPA Architecture

Professor:
Adriano Tavares

June 2024

Agenda

List of Acronyms

ABI	Application Binary Interface
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
CFG	Context-Free Grammar
CLI	Command Line Interface
CPU	Central Processing Unit
DFA	Deterministic Finite Automaton
EBNF	Extended Backus-Naur Form
FPGA	Field-Programmable Gate Array
IR	Intermediate Representation
ISA	Instruction Set Architecture
LALR	Look-Ahead LR
LC	Location Counter
LR	Left-to-Right
NFA	Non-Deterministic Finite Automaton
RAW	Read After Write
SLR	Simple Left-to-Right
Soc	System on Chip

List of Figures

List of Algorithms

1 | Introduction

Nowadays, compilers are a critical component in every piece of software we use. Although one might not notice it at first glance, in some way or another, almost all of these were built using high-level programming languages, and had to be translated into machine executable code.

In general, a compiler is responsible for performing some kind of "translation" between different languages, the most common use case is the before mentioned one, that consists of translating high-level programming languages into binary code that CPUs are able to process.

The goal of this project is to design and implement a compiler that is capable of generating machine code for the designed VeSPA SoC, starting from an high-level programming language, in this case, the C programming language.

As for this report, its goal is to elucidate the reader on the full development process, starting with a brief analysis of the problem, the design phase, important decisions and trade-offs performed during it, and finally the obtained results.

2 | Analysis

A compiler is a specialized software tool that translates code written in a high-level programming language into machine code, assembly language, or another executable form that a computer's processor can understand. The translation process consists of several phases: lexical analysis, syntactic analysis, semantic analysis, optimization, and code generation. A well-developed compiler can detect and report errors of lexical, syntactic, and semantic nature, while also enhancing its own performance through code optimization.

In the compiler's front-end, code analysis is performed comprehensively to identify and rectify possible errors through different analysis stages. Once the code has been analyzed and validated, the compiler moves to the back-end, where optimizations are performed, and the internal elements of the system are arranged for the efficient execution of the program. This process culminates in generating machine code specific to the target processor architecture. Figure 2.1 provides a brief overview of the compiler architecture.

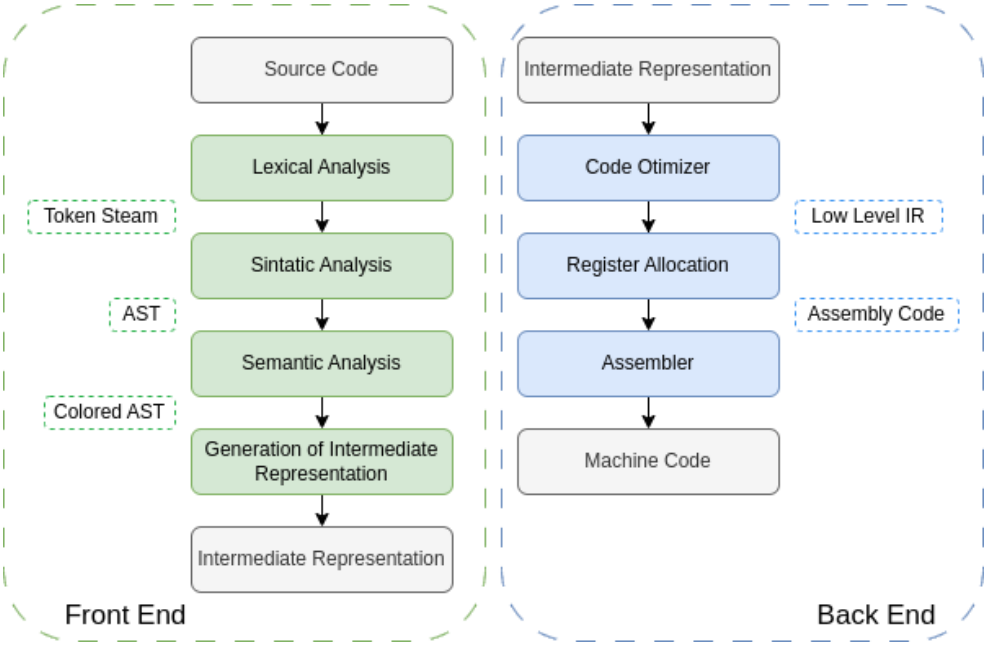


Figure 2.1: Compiler phases schematic.

The compiler to develop includes a set of requirements and constraints:

Requirements

- Generate machine code executable by the target architecture (VeSPA), starting from C source code.
- Produce machine code that is logically correct when compared to the original C source code.
- Generate detailed debug information about potential issues in the code;
- Provide a maintainable and extendable structure, so it can easily be upgraded in the future.

Constraints

- Use the lexical analyzer generator GNU Flex;
- Use the syntactic analyzer generator GNU Bison;
- Compiler source code must be written in C;
- The project must be completed by the end of the semester.

2.1 Lexical Analysis

In the compilation process, lexical analysis is the first phase, where the source code is examined to identify and classify the basic elements of the programming language. This phase converts the source code into a sequence of tokens, which are the smallest units of meaning, such as keywords, identifiers, literals, and operators. Lexical analysis simplifies code by removing unnecessary spaces and comments, and it also detects errors such as invalid characters.

The scanner uses regular expressions to match patterns within the code, with meta characters defining these patterns. By categorizing the tokens the scanner provides a structured representation of the source code, which is essential for the subsequent phases of the compiler to process the code correctly.

Lexical analyzer generator

This generator is responsible for translating the regular expressions mentioned in the scanner into state machines that will return an IR for the lexical analyzer program. These state machines can employ two main approaches:

- DFA (Deterministic Finite Automaton);
- NFA (Non-Deterministic Finite Automaton).

These two approaches differ primarily in the state transition process. In the case of NFA, there is the possibility of transitioning from one state to multiple other states based on the input symbol, as opposed to DFA, where each state is restricted to transitioning to only one unique state for each input symbol.

NFA allow for non-deterministic state transitions, meaning that when the automaton is in a certain state and receives a particular input symbol, it can transition to one or more states simultaneously or even to no state at all. This flexibility in transition behavior allows NFA to represent certain language patterns more succinctly.

On the other hand, DFA are deterministic, meaning that for every state and input symbol, there is exactly one defined transition to another state. This determinism simplifies the behavior of DFA, making them easier to understand and implement, but it also limits their expressive power compared to NFA.

Despite their differences, NFA and DFA are equivalent in terms of computational power, meaning that any language recognized by an NFA can also be recognized by a DFA, and vice versa. However, NFA may require fewer states to represent certain languages, leading to more compact representations and potentially more efficient processing.

Generators like Flex perform the conversion of regular expressions to NFA and NFA to DFA. This behavior allows for greater performance as it enables more optimizations and complexity reduction, resulting in a final algorithm more compact and simplified.

2.2 Syntactic Analysis

Also known as parsing, the second phase of the compilation process involves analyzing the sequence of tokens while considering the grammatical rules of the language. This phase reorganizes the tokens into a tree-like hierarchical structure called the Abstract Syntax Tree (AST). The AST represents the syntactic structure of the source code, making it easier to perform further analysis and transformations during the compilation process.

2.2.1 Parser

In the process of syntactic analysis, the parser is the component responsible for analyzing the stream of tokens resulting from the lexical analysis stage and converting it into an AST that represents the code in hierarchical levels through a set of Context-Free Grammar (CFG) rules. Parsers can be described in two main forms: top-down and bottom-up.

Grammar rules of a CFG are normally described using a formal notation technique, being two of the most well-known BNF and EBNF. Both of these notations use symbols to represent elements like terminal symbols (actual characters) and non terminal symbols (placeholder for larger structures, for example), and both use rules to define how can these elements be combined to form valid expressions, but since EBNF is an entension of BNF, BNF has a stricter syntax and EBNF offers additional features for more flexible grammar descriptions (for example, including options for repetition and optional elements).

In this project, a bottom-up parser will be used due to the constraint of utilizing Bison, which inherently operates as a bottom-up parser. Bison doesn't natively support EBNF, so the grammar description rules to develop will follow the BNF notation, as it is easier to parse and interpret.

2.2.2 Top-Down Parser

In a top-down parser, the analysis begins from the most general rule (root), traversing the tree until it reaches the most specific nodes (leaves). This type of parser fundamentally follows two types of algorithms:

- Backtracking algorithms;
- Predictive algorithms.

In the first mentioned, the parser goes through attempting to derive the input tokens multiple times until it either contains a successful or error response. This feature provides flexibility in handling grammar ambiguities, as well as simplifying its implementation, but at the cost of being slower than predictive parsers.

On the other hand, predictive parsers offer greater speed since they attempt to predict which production to apply based on the arrangement of tokens in advance, proving to be a faster and more efficient alternative to backtracking. However, this approach is more prone to grammar errors.

2.2.3 Bottom-Up Parser

The Bottom-Up parser performs the analysis in an order opposite to that of the Top-Down, starting the traversal from the most specific nodes (leaves) up to the most global nodes (root). There are several types of algorithms for this approach:

- LR (Left-to-right);
- SLR (Simple left-to-right);
- LALR (Look-Ahead LR).

Left recursion is a problem that occurs when a non-terminal symbol can directly derive to itself or to a cycle of non-terminal symbols that eventually leads back to the same non-terminal symbol. With this type of algorithms, the problem is eliminated, thus avoiding infinite loops or incorrect results during the parsing process.

Algorithms of this specific type apply the shift-reduce method for the desired purpose (Fig. 2.2). The shift involves moving the terminal symbol at the current position to the top of the parser stack. Meanwhile, the reduce involves replacing a phrase at the top of the stack with a non-terminal symbol.

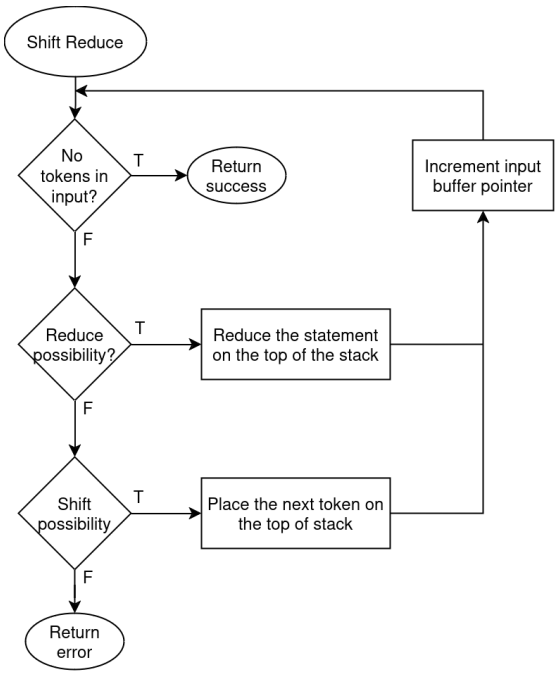


Figure 2.2: Shift-reduce algorithm flowchart.

2.2.4 Abstract Syntax Trees

Abstract Syntax Trees (AST) are a fundamental intermediate representation in the compilation process. An AST is a tree-shaped data structure that captures the syntactic structure of source code in a hierarchical and abstract way.

The AST consists of nodes, where each node represents a significant syntactic construct of the program, such as expressions, declarations, and control flow statements. The relationships between these nodes are represented as parent-child relationships. In an AST for the C language, we can find nodes representing variable declarations, arithmetic and logical expressions, and control flow statements.

The construction of the AST is performed during the parsing phase of the compilation process. The parser uses the grammar of the C language to recognize syntactic constructs and build the corresponding tree. During parsing, each syntactic construct is converted into a node in the AST. For example, the expression $x + y$ would be represented by an operator $+$ node with two child nodes, x and y. The representation of this example can be seen in the Figure 2.3.

Once constructed, the AST serves as the foundation for several other phases of the compiler, such as semantic analysis, optimization, and code generation.

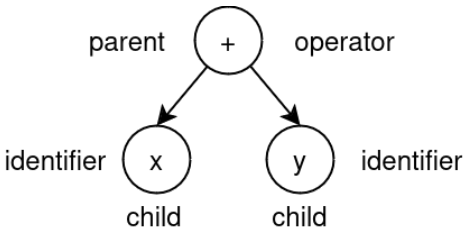


Figure 2.3: Representation of the AST for the expression $x + y$.

2.3 Semantic Analysis

Semantic analysis, the third phase of the compilation process, ensures that the code is logically sound and adheres to the rules of the programming language in the source code. This crucial step validates that the program's operations and expressions are semantically correct, enabling error-free and meaningful execution.

2.3.1 AST Traversal

For both symbol table creation and type checking, understanding tree traversal methods is essential. Tree traversal involves systematically visiting each node of a tree data structure. There are several common types of tree traversal methods such as:

- **Preorder Traversal:** the algorithm visits the current node first, then recursively traverses its left child in preorder, followed by its right child in pre-order;
- **Postorder Traversal:** the algorithm recursively traverses the left child in postorder, then the right child in postorder, and finally visits the current node.

A representation of these crossings can be seen in the Figure 2.4.

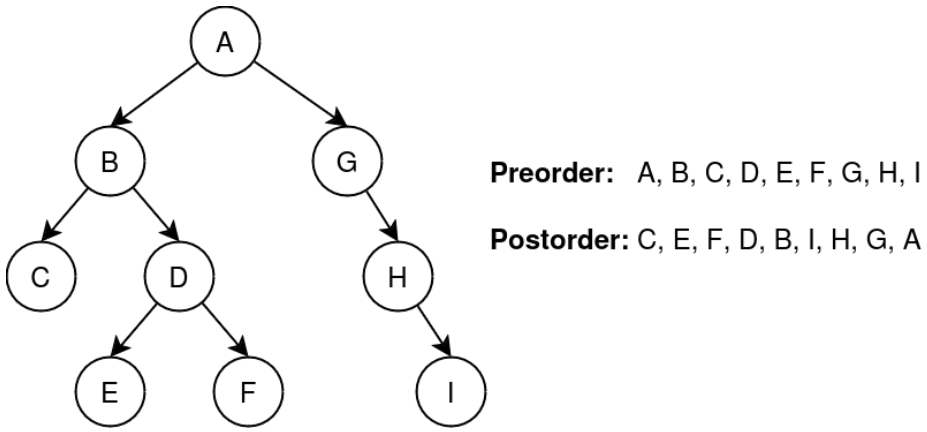


Figure 2.4: Postorder and preorder traversals

2.3.2 Symbol Table

In the semantic analysis phase of a compiler, it is important to ensure that syntactically correct constructs also have meaning within the context of the programming language. To achieve this validation, compilers often use data structures called symbol tables.

The symbol table is used to store information about the identifiers found in the source code. This includes their types, scopes, values, among other relevant details.

In languages like C, the symbol table plays a crucial role in type checking, ensuring that operations performed on variables and expressions are compatible with their type definitions.

Additionally, the symbol table helps detect semantic errors, such as using a variable before it is declared, or assigning incompatible values to a variable.

2.3.3 Type Checking

In semantic analysis, a critical and often complex process is type checking.

Type checking ensures that every instruction in the source code is performed on compatible data types. The compiler verifies the types of variables, functions, and expressions, ensuring they adhere to the language's type rules and preventing type errors. After semantic analysis, it is said that the AST is "colored," indicating that it has been annotated with type information and other semantic details.

2.4 Optimization

2.4.1 Constant Folding

Constant folding is an optimization technique used in programming language compilers to improve the performance of generated code. The compiler evaluates constant expressions at compile time, instead of run time, and replaces these expressions with their calculated result, reducing the program's execution time, as it avoids unnecessary calculations during program execution.

For example, considering the expression: $x = 8 - 2 - 3$, as all values are constant, the compiler can calculate the result of this expression at compile time. Instead of generating code to perform subtractions at run time, the compiler replaces the expression with the calculated value: $x = 3$.

2.4.2 Instruction Scheduling

An instruction scheduler is a crucial tool in the software development toolchain, strategically positioned as middle-end between the generation of code and the translation of this code into machine language by the assembler as shown in Figure 3.30. The main goal of an instruction scheduler is to refine and enhance the code automatically generated by compilers, ensuring that it is not only correct but also more efficient in terms of runtime execution.

Figure 2.5: Instruction scheduling integration.

Stall Ocurrence

In this instruction scheduler for the VesPA ISA, we will demonstrate how to specifically address the Read After Write (RAW) instruction hazard that occurs whenever a load (LD or LDX) instruction is followed by an operation that uses the same register as an operand. We will delve into solving the stall generated by a LD followed by an ALU operation or a store (ST or STX) that uses the loaded register as an operand.

However, the compiler can optimize the code by rearranging the order of instructions and inserting other instructions, as shown in Figure 2.6 . This prevents the problematic sequence from occurring, effectively solving the stall and conserving valuable computer resources.

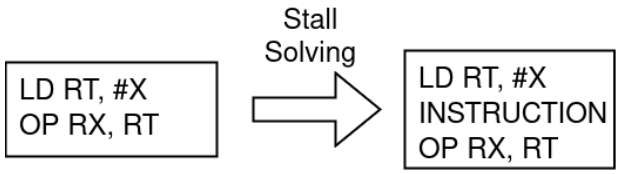


Figure 2.6: Stall generic solving.

Dependencies

While instruction scheduling is extremely efficient in reducing hazards and thus enhancing performance, there are critical key points that need to be taken into consideration to maintain the main program flow. Changing the order of instructions can also alter the execution flow, so it is necessary to verify each instruction to determine if reordering is possible and, if so, in what order it can occur. Dependencies arise when registers are utilized in subsequent instructions, such as in Figure 2.7.

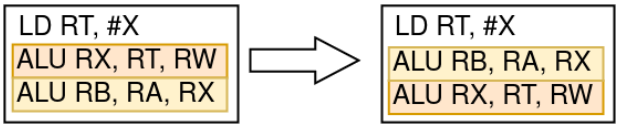


Figure 2.7: Generic dependency fault.

The figure 2.8 demonstrates a conversion example that cannot be performed due to it's obvious impact on the final result.

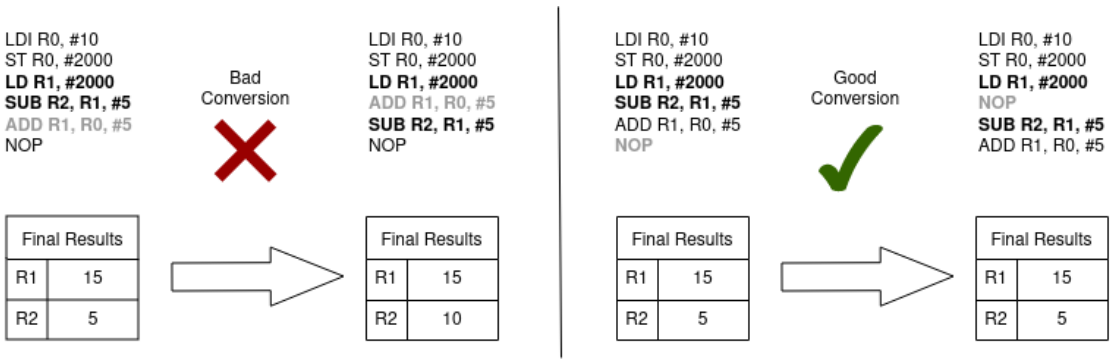


Figure 2.8: Dependency conversion case.

As demonstrated, the value of R2 will be different if this change is made, being 5 on the initial version and 10 after the wrong conversion. This happens because the ADD instruction used the same R1 register as the SUB instruction next to the stall position. The table below demonstrates what instructions can be replaced, taking into account the registers used as destination and operand registers before. In order to prevent the happening of this situation, table 2.1 was made to accommodate all possible dependency situations on the VeSPA ISA.

Instruction	None	Rdst	Rs1	Rs2	Rdst, Rs1	Rdst, Rs2	Rs1, Rs2	Rdst, Rs1, Rs2
NOP	✓	✓	✓	✓	✓	✓	✓	✓
ADD	✓	✗	✗	✗	✗	✗	✗	✗
SUB	✓	✗	✗	✗	✗	✗	✗	✗
OR	✓	✗	✗	✗	✗	✗	✗	✗
AND	✓	✗	✗	✗	✗	✗	✗	✗
NOT	✓	✗	✗	-	✗	-	-	-
XOR	✓	✗	✗	✗	✗	✗	✗	✗
CMP	✓	-	✗	✗	-	-	✗	-
RR	✓	✗	✗	✗	✗	✗	✗	✗
RL	✓	✗	✗	✗	✗	✗	✗	✗
ADD (imm)	✓	✗	✗	-	✗	-	-	-
SUB (imm)	✓	✗	✗	-	✗	-	-	-
OR (imm)	✓	✗	✗	-	✗	-	-	-
AND (imm)	✓	✗	✗	-	✗	-	-	-
XOR (imm)	✓	✗	✗	-	✗	-	-	-
CMP (imm)	✓	-	✗	-	-	-	-	-
RR (imm)	✓	✗	✗	-	✗	-	-	-
RL (imm)	✓	✗	✗	-	✗	-	-	-
MOV	✓	✗	✗	-	✗	-	-	-
LD	✓	✗	✗	-	✗	-	-	-
LDI	✓	✗	-	-	-	-	-	-
LDX	✓	✗	✗	-	✗	-	-	-
ST	✓	✗	✓	-	✗	-	-	-
STX	✓	✗	✓	-	✗	-	-	-

Table 2.1: Intructions register dependency

Control Instructions

With the most common dependencies already solved, it is however important to note that there are several other cases that can change the program execution, and for that reason, they need to be taken in consideration. On control instructions, it is essential to assure that a JUMP, JUMP AND LINK, BRANCH, RET or RETI can not be swapped to solve any stall and that there is a clear separation of the code before and after the control instruction. As shown in Figure 2.9, when the optimizer encounters a control instruction, it cannot swap the control instruction itself, nor can it alter the following code. Doing so would result in the incorrect execution order and disrupt the program's flow.

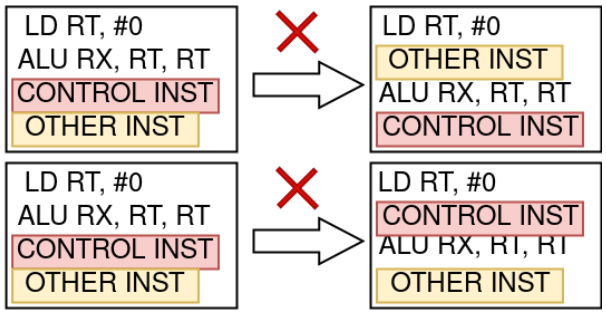


Figure 2.9: Incorrect control conversion.

In order to ensure the correct program execution, instructions need to be partitioned between instruction that can or can not be swapped. In Figure 2.10 the example shown in Figure 2.9 is solved according to this directives. As displayed, splitting the instructions sometimes leaves hazards unaddressed. This occurs because it is impossible to prevent all hazards, thereby leaving the hardware responsible for stalling the CPU when necessary.

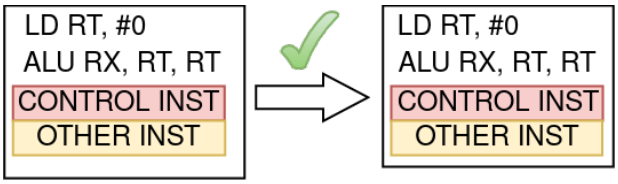


Figure 2.10: Correct control optimization.

In Figure 2.11, the LDI instruction cannot be replaced, because in normal flow without optimizations, that instruction will not be executed.

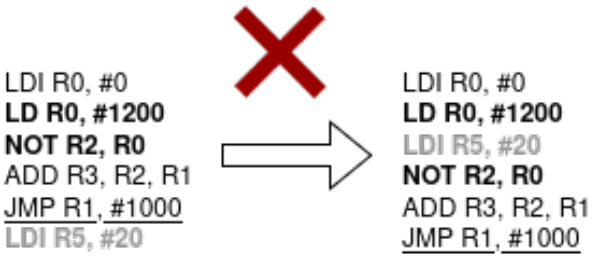


Figure 2.11: Example of a bad conversion for control instruction.

Corner Cases

In the realm of instruction scheduler, corner cases present unique challenges that can significantly impact program execution. These scenarios, often rare and unexpected, can disrupt the intended flow of instructions and lead to errors if not properly addressed. Identifying and handling corner cases is crucial for ensuring robust and reliable code performance.

Branches: More than being a control expression that needs extra care for that, branches also utilize Control Flags in order to function, and therefore imply direct control over all the instructions that have the ability to modify this same Flags. As Figure 2.12 displays, the ADD instructions can not be swapped because there is a possibility of flag changing that could change the BXX being taken or not, which would incorrectly change the program flow.

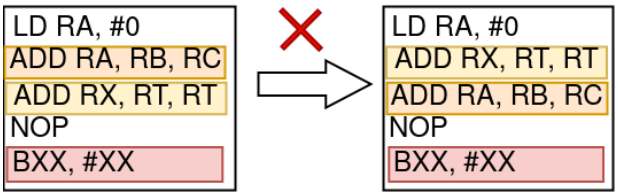


Figure 2.12: Incorrect branch optimization

In this situation, the optimizer should continue looking for a case where it would be able to swap instruction order while ensuring the correct functioning and flow of the program. As Figure 2.13 shows, in this exact case the following NOP would be the correct instruction to swap.

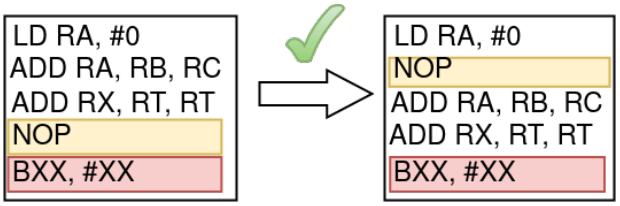


Figure 2.13: Correct branch optimization

In Figure 2.14 is possible to see a real case of why is needed to avoid ALU operations to be a replace instruction in this case.

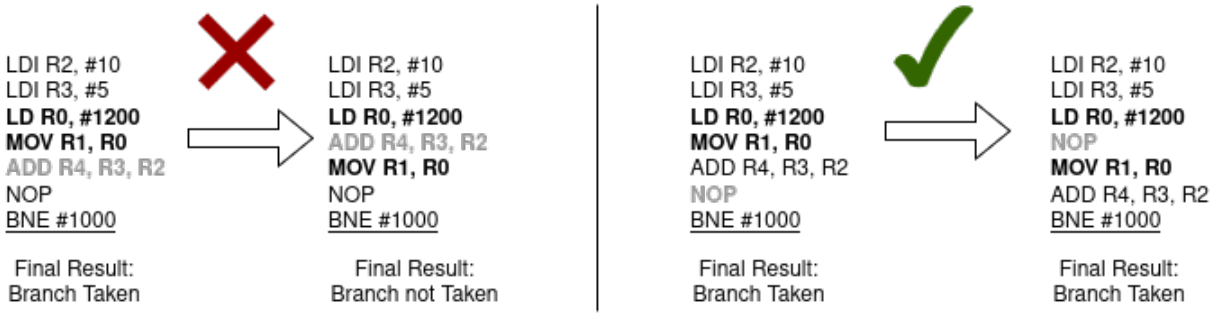


Figure 2.14: Incorrect vs correct branch conversion.

Is possible to verify that, reordering the NOP instruction, which not interfere with the final values of the flags, will not change the condition for the branch, unlike using the ADD instruction.

Double Stall: The double stall corner case refers to whenever there are two stalls that can be solved by each other. This happens, whenever the two stalls are right in front of the each other. What turn this into a corner case is the specifity of, unlike all the order possible instruction combination, two stalls can be solved by only one replacement, as shown on Figure 2.15.

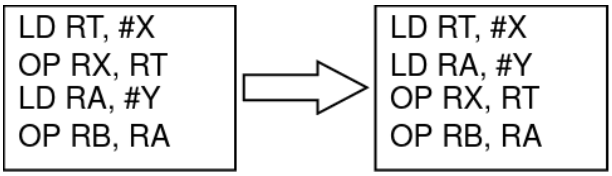


Figure 2.15: Double stall corner case.

2.5 Code Generation

The final phase of the compilation process is code generation, where the intermediate representation (IR) is transformed into executable machine code or assembly language. This phase is divided into two key steps: translating the IR into assembly language and then translating the assembly code into machine code; each phase will be executed by an assembly code generator and an assembler, respectively.

This phase is crucial and highly detailed, as it is tailored to a specific target machine and must address instruction selection, register allocation, and memory management.

2.5.1 Assembly Code Generation

The assembly code generator is one of the major players when it comes to designing a compiler, as it will be responsible for the translation of an abstract structure such as the AST into assembly code. Its main objective, as the name suggests, is to generate machine-oriented assembly code from the previously provided optimised AST.

The figure 2.16 represents where this major component fits into the compiler workflow, specifically in code generation.

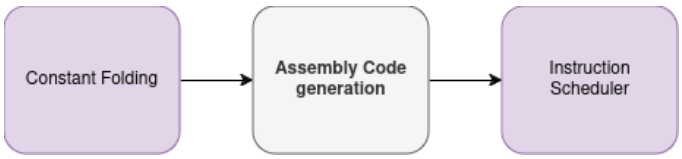


Figure 2.16: Assembly code generation overview.

2.5.2 Assembler

An assembler is an essential software tool in compiler development, especially in the context of a compiler for the C language. In the workflow of a compiler, the assembler performs the task of translating the assembly code generated by the compiler into code machine, which can be directly executed by the processor.

Using an assembler allows you to work at a more manageable level of abstraction than machine code, while maintaining a high level of control over the specific features of the architecture for which the assembler was designed. In the case of this project, an assembler will be developed to generate specific code for the VESPA architecture, ensuring that the final code is efficient and compatible with the FPGA.

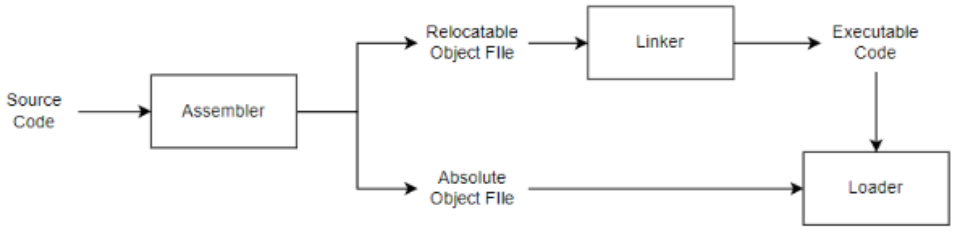


Figure 2.17: Assembler process.

The assembler process is detailed in Figure 2.18, where it is possible to observe the two steps in which the assembler process is divided. Initially, step 1 begins by interpreting code written in assembly language, receiving an opcode table and using it to generate a symbols table. Furthermore in this step, a lexical and syntactic analysis is performed and the product is the Intermediate Representation (IR). The second step of assembler execution receives the IR, the symbols table and the opcode table and uses it to generate the object code. To do this it needs to form instructions, combining various parameters related to each one, such as the opcode and the destination register. During these stages, directives can alter the Location Counter (*LC*) and need to be dealt with accordingly. The *LC* is an internal variable used by the assembler to track the current memory position during the process, acting as an offset used as a reference in the file. It assists the assembler in determining where each instruction is to be stored in memory, ensuring that the program is correctly mapped to the target memory.

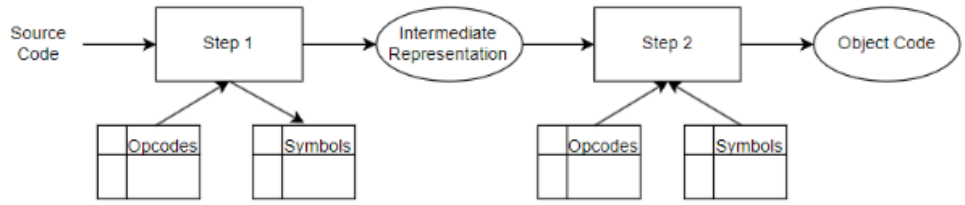


Figure 2.18: Assembler steps.

3 | Design

As mentioned in the previous chapter, a compiler is a complex piece of software, composed by several different components. Considering this, it is of great importance to ensure that each of these components, as well as the overall structure of the compiler, is well defined before implementing said design. The following chapter aims to provide an in-depth analysis of the proposed compiler design.

3.1 Lexical Analysis

Considering the constraints defined during the analysis phase, the tool that shall be used to perform the lexical analysis step of the compilation process, is GNU Flex. This tool is a widely used, open source scanner generator known for its efficiency, flexibility, ease of integration and cross-platform compatibility.

By analysing the C programming language, it was possible to identify several requirements and constraints defined by its syntax. Considering this, it was possible to obtain the tables shown bellow. In Figure 3.1 it is possible to find all the different keywords reserved by the C programming language syntax. These words can not be used to define user specific parameters, as they are used to represent language specific operations. Besides this reserved keywords, there are also some special symbols defined by the C programming language syntax, said symbols can be found on Figure 3.2.

Reserved Words				
char	short	if	goto	switch
float	long	else	do	case
int	union	return	typedef	break
double	sizeof	extern	define	continue
void	static	register	default	
const	volatile	enum	while	
struct	unsigned	signed	for	

Figure 3.1: Reserved words table.

Special Symbols				
+	--	||	/=	>
-	&	=	&=	>=
/	|	==	%=	<=
*	~	!=	+=	>>
%	^	|=	-=	<<
/	!	~=	*=	:
++	&&	^=	<	;
,	.	(	)	{
}	[	]	?	#
->	\0	\n	\t	

Figure 3.2: Special symbols table.

Regular expressions, depicted in Figure 3.3, are patterns used to match character combinations in strings. They define search patterns, crucial for identifying token types like identifiers, keywords, operators, and delimiters, facilitating accurate lexical analysis in compilation processes.

Regular Expressions	
[0-9]	Digit
{digit}+	Number
Digit*"."Digit + (eE][+-] ? Digit+)?	Float Number
[a-zA-Z_][a-zA-Z0-9_]*	Identifier
\n	Newline
([-+])	Sign
[eE]{Sign}?	Expo
[\t]+	Whitespace

Figure 3.3: Regular expressions table.

With these tokens identified, it is possible to construct and high level DFA, as it is possible to observe in Figure 3.4. This DFA demonstrates how the tokens will be divided in order to identify them, as mentioned previously.

When the DFA receives a delimiter, it transitions to the "Done" state, representing the successful identification of a token. If a delimiter is found, and one of the required conditions is not met, the DFA enters an error state.

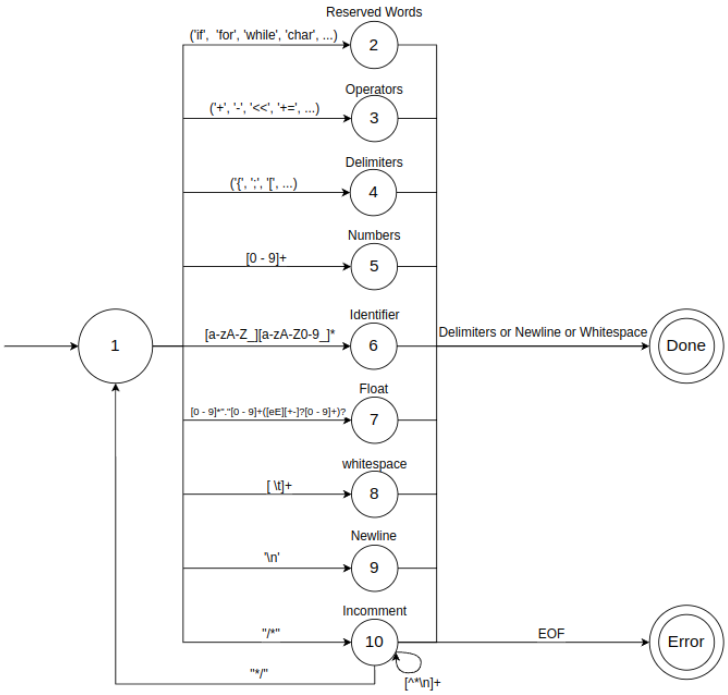


Figure 3.4: DFA representation.

It is important to state that even though the scanner recognizes all the mentioned reserved words and special symbols, not all of them will be supported by the compiler as a whole.

For educational purposes, concentrating on a simplified subset of the language can be more beneficial, allowing a better understanding of compiler design principles without being overwhelmed by the several complications of the entire C language. So, given the time constraint applied, the following reserved words and special symbols will not be supported by the compiler: **"struct"**, **"typedef"**, **"enum"**, **"union"**, **"sizeof"**, **"->"** and **"."**. These constructs introduce additional complexity in both syntactic and semantic analysis, which require a more complex symbol table management.

By excluding these elements, the compiler's architecture is simplified, but a more stable and functional compiler can be developed and serve as a solid foundation for future enhancements.

3.2 Syntatic Analysis

In this section, we'll explore the design of syntatic analysis, a pivotal stage in compilation, ensuring the grammatical accuracy of source code. Syntatic analysis, also known as parsing, involves analyzing the code to determine its grammatical structure based on language rules.

The designated syntactic analyzer generator is GNU Bison, a widely used, open source, bottom up parser generator that transforms grammar descriptions into C or C++ code for syntax analysis. It is also compatible with Flex, which allows a seamless integration of lexical and syntactic analysis.

In the design of syntactic analysis, it is essential to select a suitable grammar for the target language, in this case: C. This process entails establishing production rules that delineate the permissible syntactic structure within the source code. For this project, the Backus-Naur Form (BNF) was adopted as the grammar of choice, given its widespread usage in defining rules for various language constructs, including expressions, declarations, and commands.

3.2.1 Local Statements

Within this context, local statements denote specific instructions confined within a code structure's local scope. These statements play a crucial role in understanding the program's structure and in determining how the code is transformed into an IR or the final machine code. The illustration in Figure 3.5 provides visual clarity on the supported local statements.

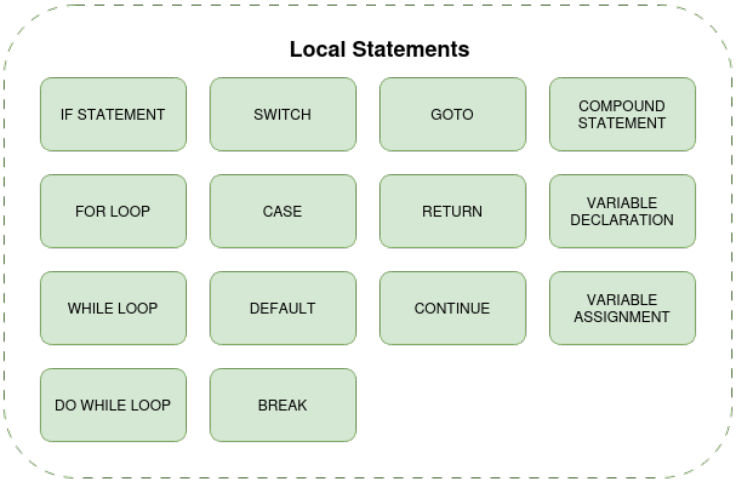


Figure 3.5: Supported local statements.

The "Compound Statement" refers to a block of code enclosed within curly braces '{ }', capable of containing multiple statements and treated as a single unit by the compiler. They are fundamental for organizing code into logical units or different scopes.

As depicted in Figure 3.5, the "Flow Control Statements" include:

- **If Statement** - It executes a block of code conditionally based on a boolean expression. This structure is crucial for creating conditional logic.
- **Switch Statement**- It allows the execution of different blocks based on a selection, unlike the if statement. It provides a convenient way to perform various actions based on the value of an expression. The Switch Statement consists of three types of commands: Case, Default, and Break.
- **Case** - Used within a switch block to define different branches based on the value of an expression.
- **Default** - It serves as a catch-all branch in a switch block, executing its associated code block when none of the preceding case statements match the value of the expression being evaluated.
- **Break** - Used to exit from loops or switch statements prematurely. When encountered, it causes the immediate termination of the innermost loop or switch statement enclosing it, and control resumes at the statement immediately following the terminated loop or switch block.
- **Loop Statements** (For, While, and Do While) - These execute a block of code repeatedly as long as a specific condition, typically a boolean expression, remains true.
- **Goto** - This statement allows unconditional transfer of control from one part of the code to another.
- **Return** - When encountered during the execution of a function, the function is exited immediately and control is transferred back to the point in the program where the function was called.
- **Continue** - Used within loop constructs to skip the current iteration and proceed to the next iteration of the loop.

The "Declarations Statements", also in Figure 3.5, are:

- **Variable Declaration** - Defines a new variable along with its data type within a program.
- **Variable Assignment** - Assigns a value to a variable.

3.2.2 Production Rules

Production rules are a fundamental aspect of syntax analysis in compiler design, defining the grammatical structure of a programming language. Expressed in formal notation like BNF, they outline how syntactic constructs are formed from basic elements. These rules establish a hierarchy in source code structure, enabling precise and efficient syntax analysis. The grammatical rules specify how syntactic variables can be expanded to form phrases.

The following example (Figure 3.6) demonstrates that a local statement can encompass any of the statements discussed in the preceding subsection.

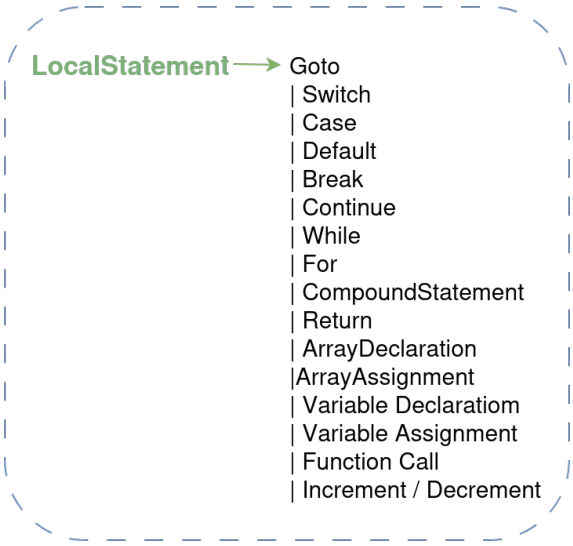


Figure 3.6: Local statements grammatical rule.

If Statement

As previously explained, the If Statement relies on a boolean expression, consisting of a token "if", an expression inside '()' followed by a local statement and may also include an 'else' token followed by another local statement, as can be seen below in the Figure 3.7.

Figure 3.7: If statement grammatical rule.

Switch Statement

The switch is composed by the token "switch" and the switch body, represented in Figure 3.8.

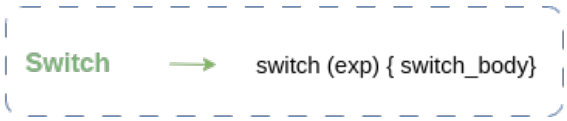


Figure 3.8: Switch statement grammatical rule.

The grammar rule presented in Figure 3.9 outlines the structure of the switch body, accommodating sequences of cases and a default case. This rule provides flexibility in defining the behavior of switch statements within the language syntax.

Figure 3.9: Switch body statement grammatical rule.

Loop Statements

In loop statements, there are three types of loops: While, Do-While, and For. The grammar rules for each type are depicted in Figures 3.10, 3.11, and 3.12, respectively. These rules delineate the syntactic structure for implementing iterative constructs in the language.

The While statement expects the "while" token followed by a boolean expression and a local statement, or just a semicolon.

The "do while" statement expects a "do" token followed by a local statement and a "while" with the boolean expression.

The "for" statement requires a "for" token, an initialization field, an expression, and an assignment field separated by semicolons, followed by a local statement.

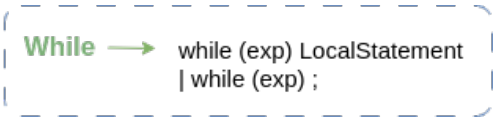


Figure 3.10: While statement grammatical rule.

Figure 3.11: Do while statement grammatical rule.

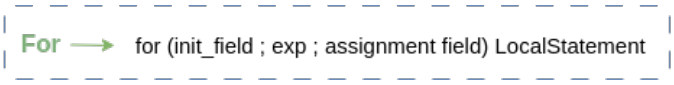


Figure 3.12: For statement grammatical rule.

Variable Declaration

When declaring a variable, the process involves determining whether it's a simple variable or an array, followed by parsing its preamble. In Figure 3.13, it's represented the respective grammar rule.

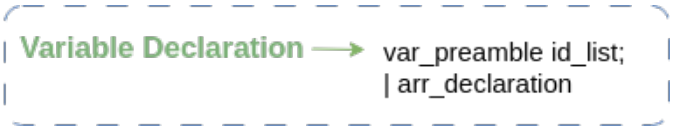


Figure 3.13: Variable declaration statement grammatical rule.

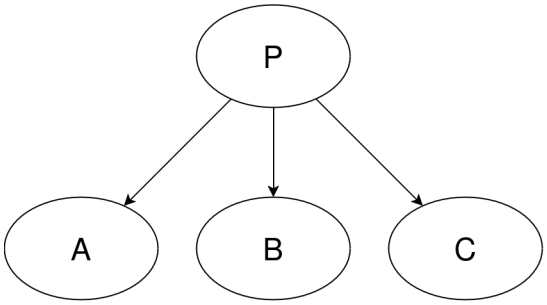
3.2.3 Abstract Syntax Tree

An Abstract Syntax Tree is used to represent the syntactic structure of the source code in a hierarchical and abstract way.

Figure 3.14 illustrates the hierarchical structure of nodes and their children within an abstract syntax tree (*AST*) used in a compiler. The diagram is divided into two parts: the left side shows a generic node (*P*) with its child nodes (*A, B, C*), while the right side provides a detailed description of various node types and their respective children.

The table on the right details different types of nodes commonly found in an *AST*, along with their specific child nodes. Each row represents a type of node, and the columns indicate the different types of child nodes. For example, a *While* node has two children: a condition and a while body, representing the loop condition and the block of code to be executed repeatedly . While an *If* node has three children: a condition, an if body and an else body.

This structure allows the *AST* to comprehensively represent the syntactic structure of the source code, facilitating several compilation steps, such as semantic analysis and code generation.



P	A	B	C
If	condition	if body	else body
Ternary	condition	true body	false body
While	condition	while body	-
While (wihout body)	condition	-	-
Switch	expression	statements list	-
Var Declaration	type	-	-
Array Declaration	type	size	-
Operator	exp	exp	-
Identifier	type	-	-
Function Signature	type	args list	function body
Function Call	args list	-	-
Pointer	type	-	-
TypeCast	type	-	-

Figure 3.14: AST nodes and child relationships.

The *TreeNode* struct represents the nodes in the Abstract Syntax Tree. It contains a pointer to the first child (*pChilds*), a pointer to the next sibling (*pSibling*), the number of children (*childNumber*), the line number in the source code (*lineNumber*), the node type (nodeType), data node-specific variables (*nodeData*), the associated variable type (*nodeVarType*), a pointer to the entry in the symbol table (*pSymbol*) and a pointer to the scope where it is located (*pScope*). This structure allows the creation and navigation of the AST, storing essential information about the nodes (Figure 3.15).

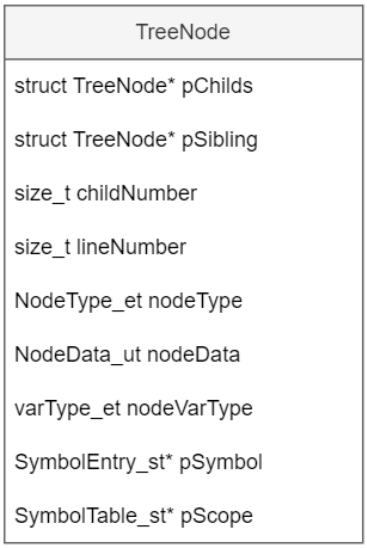


Figure 3.15: TreeNode struct diagram.

To support the generation of the AST, several functions were created:

- **NodeCreate:** Allocates memory for the new node, assigns the node's type and line number, and initializes the symbol pointer;
- **NodeAddChild:** Reallocates memory, copies the child node to the array of children, and frees the memory of the child node. This method is used when non-terminal symbols represent intermediate steps of deriving the rule where they are inserted. Non-terminal symbols have their own derivation rules;
- **NodeAddNewChild:** Reallocates memory, increments the child number, and assigns the new child's type and line number. This method is used when in a rule, there are terminal symbols, such as identifiers;
- **NodeAddChildCopy:** Reallocates memory and copies the child to the array of children. This method is used when multiple variables of the same type are declared in the same line ($intx, y, z;$), ensuring that the number of children created equals the number of variables. Since all variables are of the same type, all children are a "copy" of the first one;

The flowcharts of these functions can be seen in the Figure 3.16.

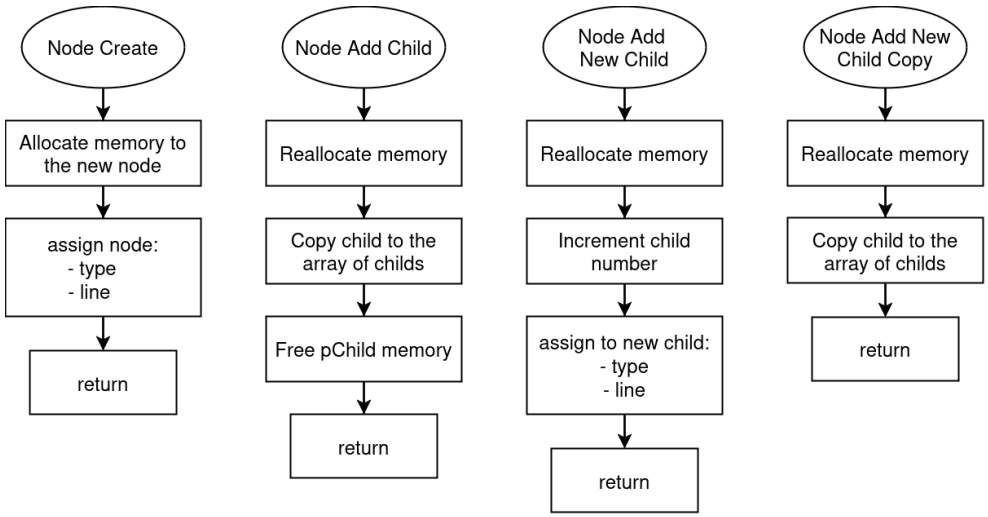


Figure 3.16: AST functions flowcharts.

3.3 Semantic Analysis

The semantic analysis phase is a critical stage in the development of a compiler, responsible for ensuring the validity and consistency of the source code.

During this phase, the compiler verifies that the code sticks to the language's semantic rules, which go beyond the syntactic structure established during parsing. The semantic analysis typically involves several checks, such as type checking and scope resolution. A typical semantic analysis is structured into three main components: symbol table management, type checking, and semantic checks or optimizations.

3.3.1 Symbol Table

The symbol table uses hash tables for efficiency in searching and inserting elements. Each program scope has its own symbol table, allowing for efficient management of variables and identifiers across various contexts.

An approach of scoped symbol tables is adopted, where each symbol is associated with the scope of its declaration. This facilitates scope resolution during semantic analysis, enabling the compiler to prioritize variables in the nearest scope to the referencing point, known as "nearest scope resolution".

Figure 3.17 illustrates this concept, showing the hierarchical organization of symbol tables across different scopes within a program. It demonstrates how different scopes, such as the global scope and nested scopes within functions, each maintain their own symbol tables. This segregation isolates identifiers within their respective scopes, preventing conflicts and ensuring proper resolution of variable declarations. Linked lists are necessary to handle collisions within each hash table, ensuring each identifier is correctly stored and accessible, even when multiple identifiers hash to the same index.

Therefore, variables are always visible if the scope in which they were declared is above the current scope, but not vice versa. This means that an identifier declared in an outer scope can be accessed in inner scopes, but identifiers declared in inner scopes are not visible to outer scopes.

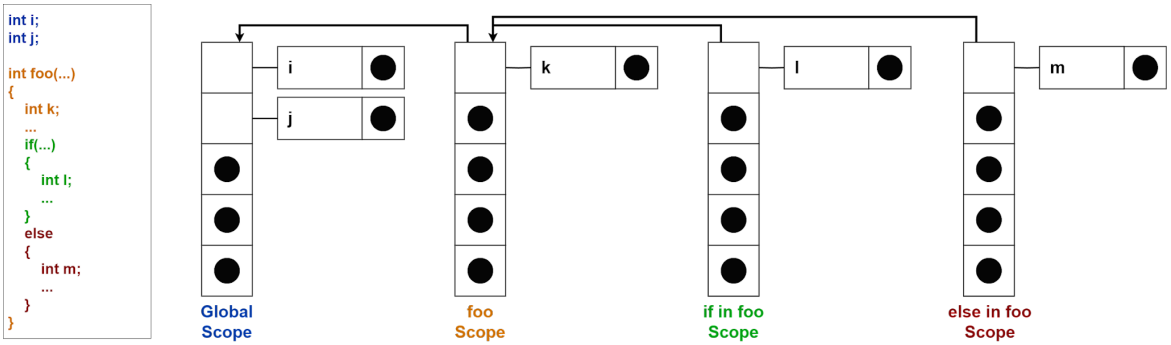


Figure 3.17: Scoped symbol table representation.

In the Figure 3.18, code in C is presented on the left and on the right a schematic representation which illustrates the generated symbol tables and the interrelations among scopes.

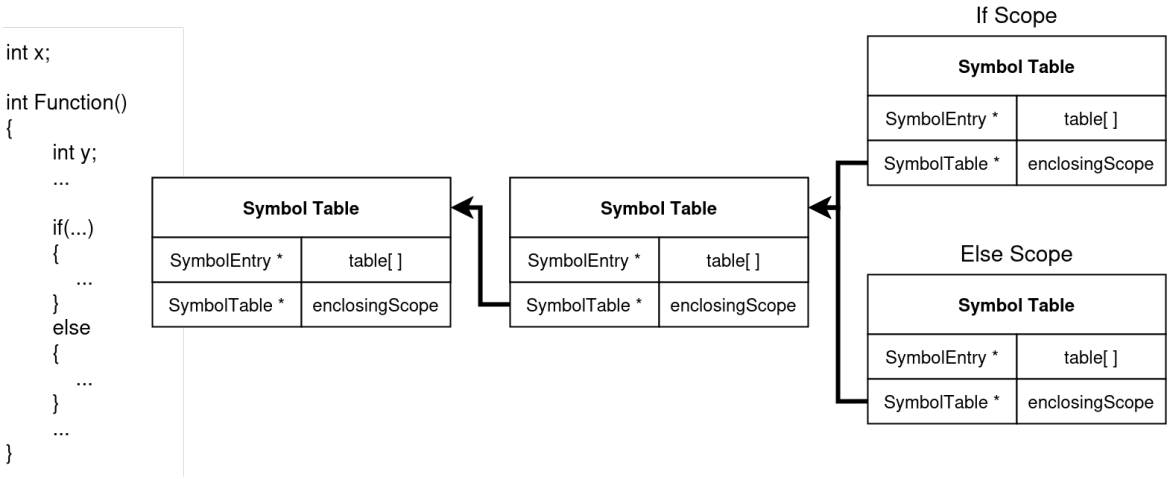


Figure 3.18: Symbol table example.

The *SymbolEntry* structure represents an entry in the symbol table, containing a union called *SymbolContent*. This union encompasses three distinct structs, each containing attributes specific to symbol types, such as: variables, pointer, functions or arrays. (Figure 3.19).

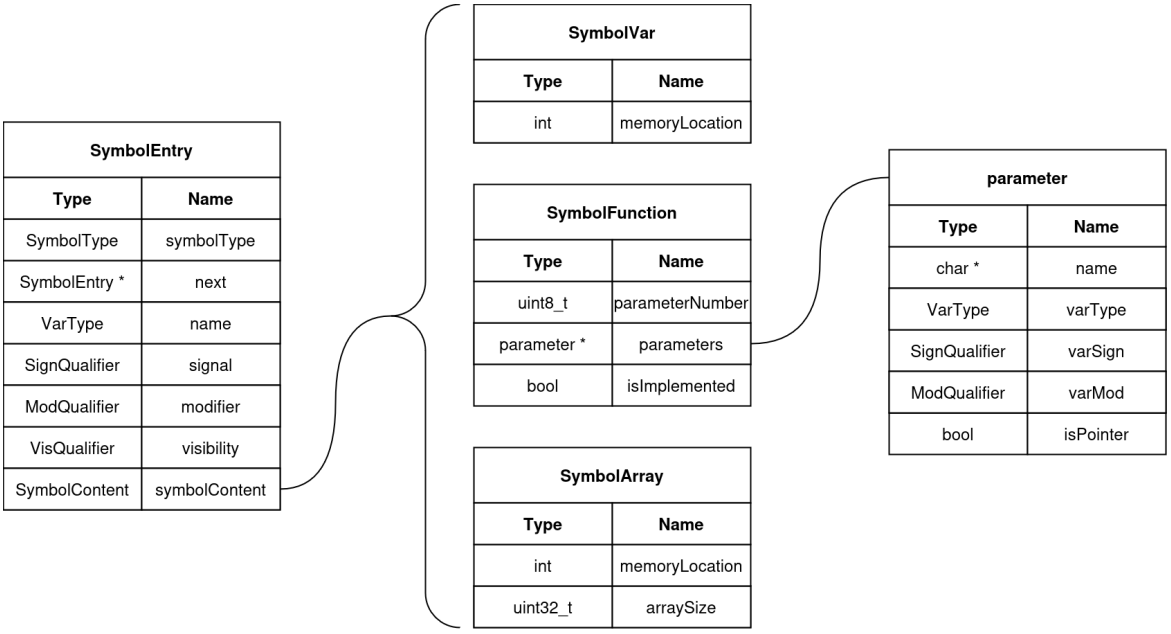


Figure 3.19: Symbol entry schematic representation.

Furthermore, to deal with hash collisions, which are caused when the hash index is the same, each hash table range implements a linked list.

Hash

The hash function, depicted in Figure 3.20, given a key, iterates through each character of the key using a loop until it found the null terminator. During each iteration, the hash value is updated, and once all characters are processed, the hash value which corresponds to the index of the key in the hash table is returned.

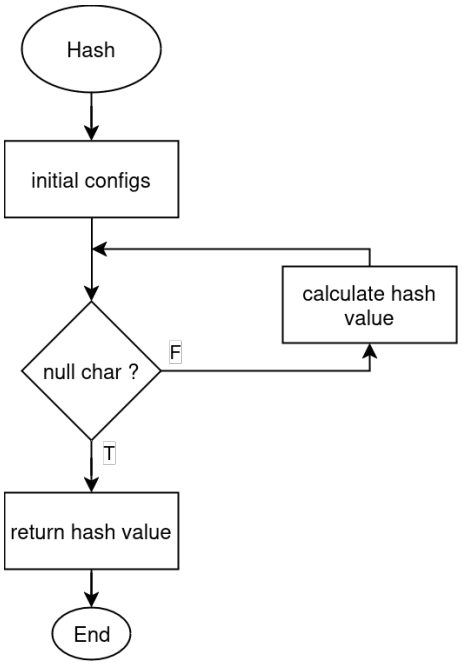


Figure 3.20: Hash algorithm flowchart.

Fetch Symbol

The fetch Symbol function searches for a symbol in a symbol table by its name. It starts by calculating the index for the symbol name (Figure 3.21). Then it goes through a linked list at the calculated index, checking each symbol entry for a name match. If a match is found, it updates the pointer to the symbol entry and returns a flag indicating success. If no match is found in the current scope and an enclosing scope exists, it repeats the process in the enclosing scope. If still, no match is found, it returns a flag indicating symbol not found.

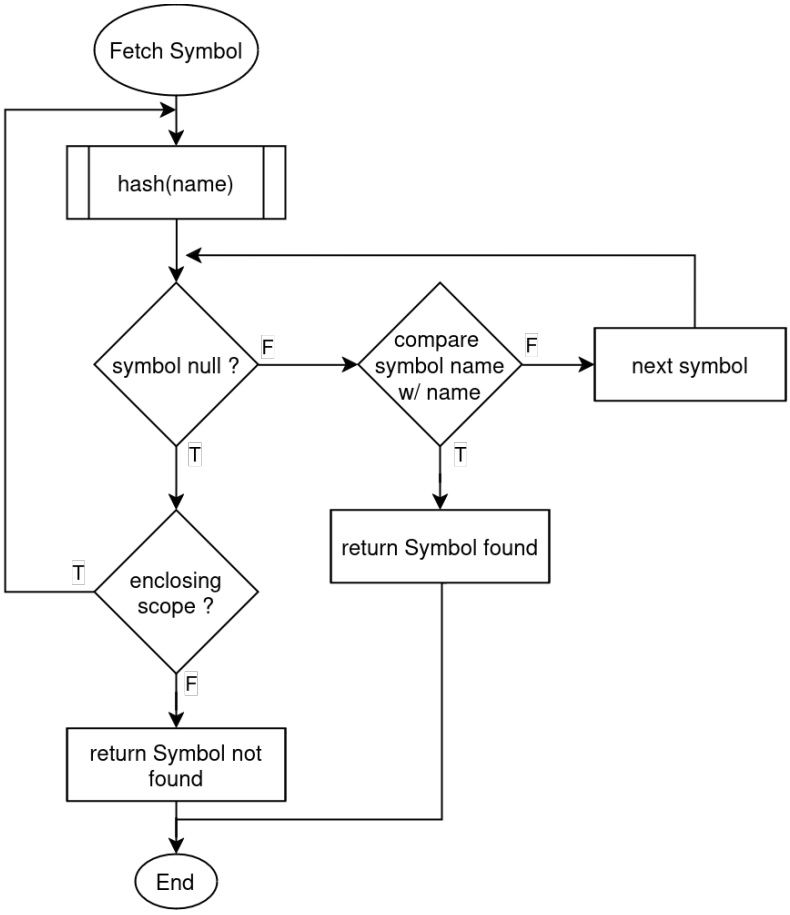


Figure 3.21: Fetch symbol algorithm flowchart.

Insert Symbol

The insert Symbol function (Figure 3.22) starts by calling the function hash to obtain the symbol hash value. Then the fetch function is called to verify if a symbol with the same name exists in the current table. If a symbol is found, the function returns symbol error. If not, a symbol with the given attributes (name and symbol type) will be created and inserted into the table. Finally the function updates the pointer to the symbol entry and returns a symbol added flag.

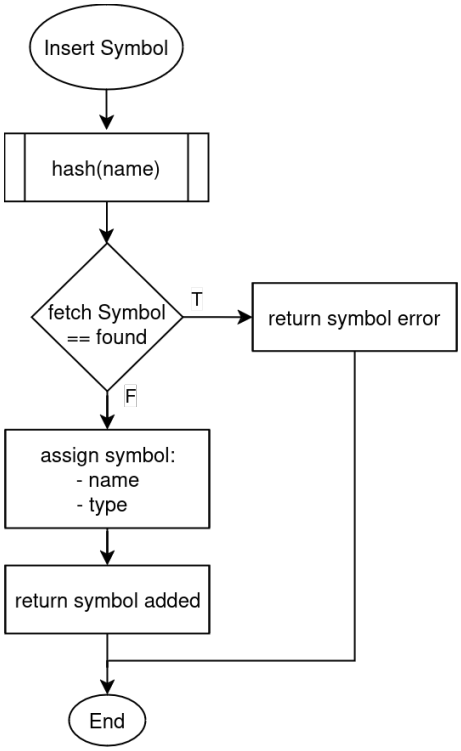


Figure 3.22: Insert symbol algorithm flowchart.

Symbol Table Traversal

This section details the design of the symbol table traversal mechanism employed during the AST traversal for symbol table construction (Figure 3.23).

The traversal follows a **pre-order** approach, ensuring each node is visited before its children. Following this visit, sibling nodes are traversed sequentially.

The flow chart below illustrates how this traversal works.

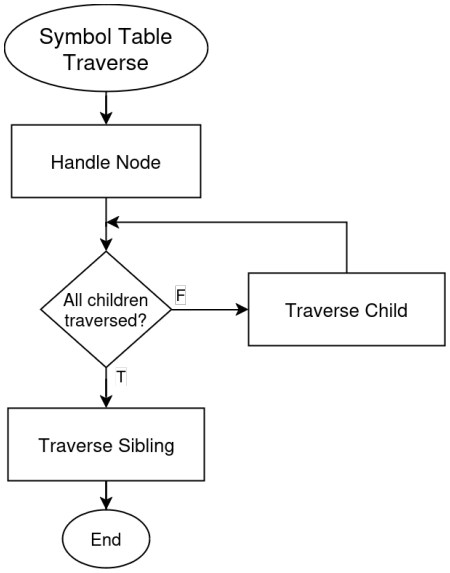


Figure 3.23: AST traversal for symbol table creation flowchart.

3.3.2 Node Processing and Symbol Table Management

A function handles each encountered node during the traversal. This function is responsible for upholding symbol table integrity through various verifications, including for example:

- Duplicate Variable Declarations;
- Missing Variable Declarations;
- Undefined Function Calls;
- Parameter Mismatch;

The processing function handles nodes based on the encountered node type.

Variable Declaration

For instance, upon encountering a node that implies a variable/function declaration, the function checks the current scope for a variable with the same name. If no conflict exists, a new symbol entry is established within the symbol table, encapsulating all information pertaining to the declared variable. If a symbol already exists, then a *Duplicate Variable Declaration* semantic error will be raised.

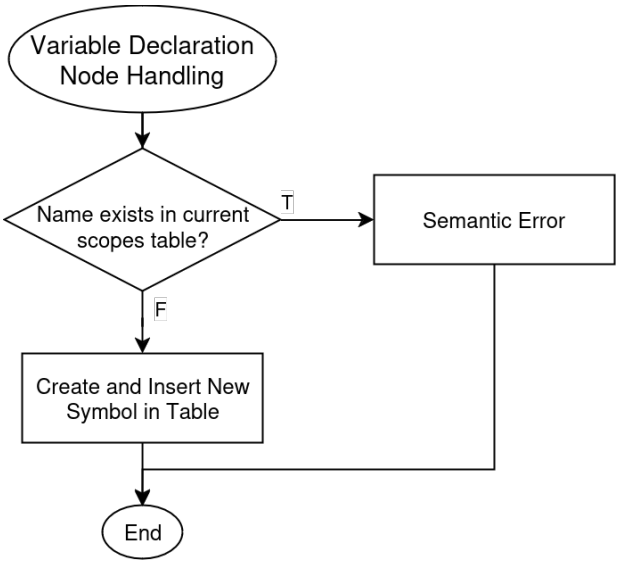


Figure 3.24: Variable declaration node handling flowchart.

Variable Reference

Upon encountering a node in which a variable is referenced, the function checks the current and enclosing scopes for a variable with the same name. If one exists nothing will happen, if not then a *Missing Variable Declaration* semantic error will be raised.

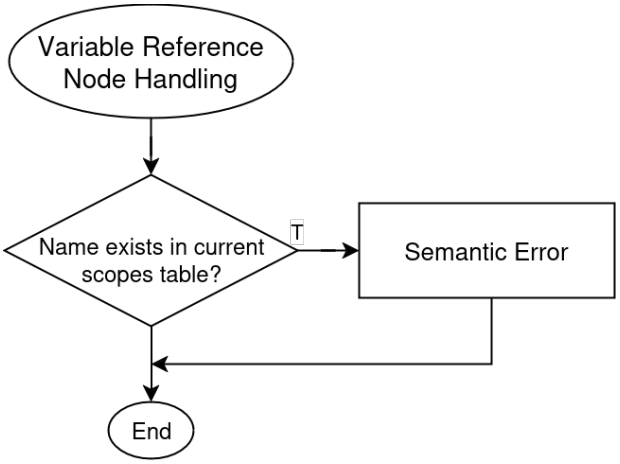


Figure 3.25: Variable reference node handling flowchart.

Function Call

Upon finding function calls, the symbol table is consulted to verify the function's existence and ensure the number of arguments passed matches the declared parameter count. This protects against *Undefined Function Call* and *Parameter Mismatch* semantic errors.

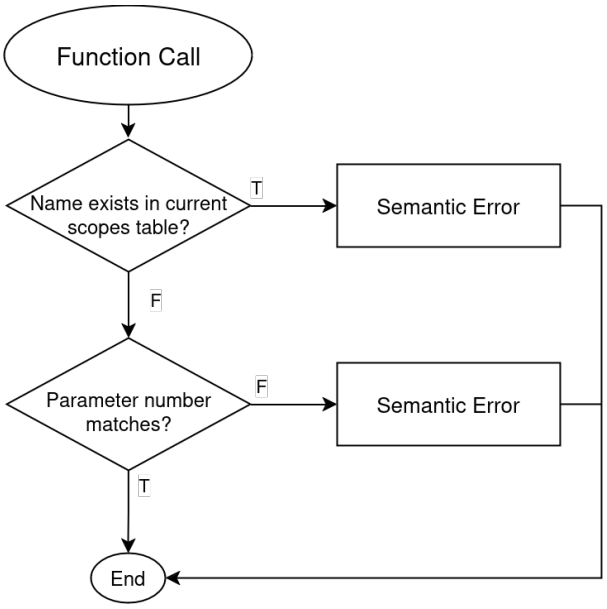


Figure 3.26: Function call handling flowchart.

Scope Management

Maintaining proper scope management is essential for constructing an accurate symbol table. The traversal achieves this by leveraging specialized nodes designated as *NODE_START_SCOPE* and *NODE_END_SCOPE*. These nodes act as markers within the AST, instructing the traversal on how to manage the concept of scopes within the symbol table.

Encountering a *NODE_START_SCOPE* node, the current scope will be updated to newly created/entered scope. Conversely, *NODE_END_SCOPE* signals a return to the enclosing scope.

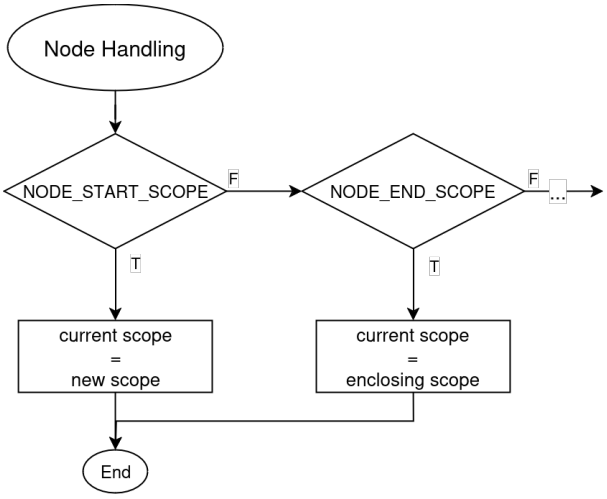


Figure 3.27: Scope handling flowchart.

3.4 Type Checking

For a program to be semantically correct, it is necessary to respect the type system. Thus, the compiler employs type checking to guarantee that the program adheres to the language principles.

Type checking is a critical process that involves examining the types of various entities within a program, such as variables, functions, and expressions, to verify that they align with the rules defined by the programming language's type system. Moreover, the type checker catches errors such as type mismatches, undeclared variables, and incompatible function arguments.

3.4.1 Strongly Typed Language

A strongly typed language employs a type system that enforces strict rules regarding interactions between different data types. This means that operations can only be performed between compatible types, which helps prevent errors or unexpected behavior. Such strictness contributes to making the language more robust and maintainable.

Since a strongly typed language will be used, some rules must be implemented:

- Array index must be INT, LONG or SHORT;
- Increment and Decrement operations can be executed on variables and pointers;
- Case condition must be INT or CHAR;
- Conditions of ternary operator, while, do while, if and switch can't be STRING;
- Operands of bitwise operations must be INT, CHAR, SHORT or LONG;
- Division by zero is not allowed;
- Assigning a pointer to a pointer is allowed;
- Assigning an address of a variable to a pointer is allowed;
- Assigning a STRING to a CHAR pointer is allowed;
- Assigning an INT variable to a CHAR variable is allowed.

3.4.2 Type Checking Methods

So the type checking process can be performed smoothly and efficiently, two different methods are necessary: a wider function that checks the node types of the AST and its compatibility and other rules (Appendix A), and a specific function to check operators and operands types and other rules (Appendix B).

The *checkNode* method ensures that each node follows the rules of the strongly typed C language. For function call nodes, it checks if the function is implemented and validates the arguments. For operator nodes, it calls a specialized function (detailed in Appendix B). Increment and decrement operations are validated for variables or pointers that are not constant. Control flow nodes (like IF, WHILE, SWITCH) are checked to ensure conditions are not of the STRING type.

The *checkOperator* method details type-checking for operators and operands, ensuring compliance with the strongly typed C language. Assignment operators are validated in a more complex way, but in short, it verifies the following cases: array indexes and pointer contents can't be assigned with an address of a variable or a pointer; assignments to a function are not allowed; string literals can't be assigned to non CHAR pointer variables; among others. The divide operator makes sure there is no division by 0. Plus to logical OR operators confirm variables aren't STRING, VOID, or mismatched.

3.4.3 Type Checking Traversal

The process of type checking begins with traversing the AST from the semantic step. The type checker examines each node of the AST and analyzes the types of the corresponding entities. During the traversal, the type checker applies a set of rules defined by the language's type system to verify the validity of type assignments and operations.

For this project, the implementation of post-order traversal was conceived (Fig. 2.4). The traversal starts from the left-bottom node and progresses to the right sibling node. Then, it moves to the parent node following the same algorithm as the child nodes until it reaches the root node.

3.5 Optimization

3.5.1 Constant Folding

After generating the AST and proceeding with the semantic analysis of the source code, before carrying out the code generation, there is an optimization step using the constant folding technique to reduce the number of instructions required in expressions with constants.

This method consists of going through the entire AST and evaluating whether it is possible to optimize operations. For this, a postorder traversal is used, in which, each time an operation has constant children, this operation is replaced by the result of the same and the child nodes that are not being used are free from the memory. This process is represented in the Appendix C.

In Appendix D, some important considerations for this optimization are presented. For example, in the case of the NOT operator, the node has only one child, and only addition, subtraction, multiplication and division operations can be of type float, while the other operations must be of type integer. These type checks are initially performed by type checking during semantic analysis, however as optimization can significantly reduce the number of operations, it is prudent to check types during the constant folding process as well. On the right side of the figure, there are algorithms for operations with floats. These algorithms calculate the value of the operation depending on the type of operation, replace the operation node with a float type node, assign the value of the operation, free the memory of the children, and finally set the number of children to zero.

3.5.2 Instruction Scheduling

Instruction scheduling is an optimization technique used in compilers to reorder the instructions of a program in order to improve the performance of the generated code. For this task, the following functions were utilized to offer support:

Instruction Parsing

Since it was necessary to read the opcode and operands for each instruction to check for dependencies, a very simplistic ad-hoc parser was implemented. Each instruction is assigned several attributes, such as mnemonic, operand register number, type, and line. These attributes are used to check dependencies and detect potential **stalls** and **replacement positions**.

To address this, the **stall positions** were identified along with the **replacement positions** for instructions that could be placed between the LOAD instruction and the instruction causing the stall. Figure 3.28 demonstrates this concept.

Figure 3.28: Stall and Replace position explanation.

Code Blocks Division

Previously, it was determined that some instructions can never be used to replace a stall. To address this issue, a separate block has been established for each group of instructions, segregated by the same instructions and directives. Since this structure enhances hazard isolation, the method is beneficial as it provides designated spots for addressing stalls that might have occurred before this specific instruction setup, with each block possessing its unique set of positions for managing stalls and replacements. This setup not only improves hazard management but also allows for more efficient replacement and stall recovery within the instruction flow.

There are two types of instruction that will break the code in blocks:

- Conditional branches;
- Jump instructions, Labels or Halt.

In the first Conditional branches case, these instructions belong to the current block, leading to the creation of a new block. This approach ensures that within the new block, no arithmetic instructions can be substituted, thereby preserving the consistency of the flag flow for the conditions.

For the second case, a new block is created, and the instruction becomes the first of the new block. This prevents any instructions from being replaced behind it, maintaining the integrity of the instruction sequence.

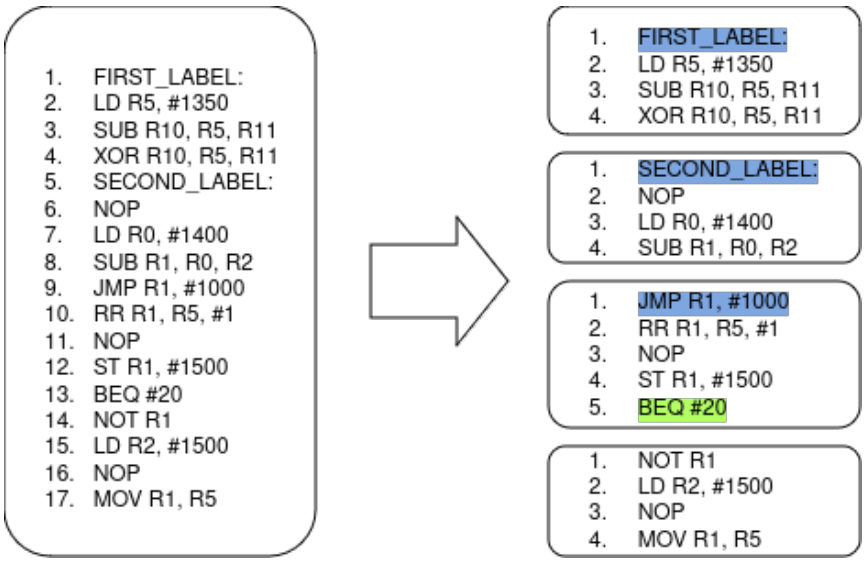


Figure 3.29: Code block division.

In the figure 3.29, it is possible to see different colors representing specific actions: blue indicates the instructions that led to the creation of a new block, being the first of that block, while green highlights the instruction that concluded the current block, allowing a new one to begin.

Stall Positions Verification

To verify the instructions that are causing stalls in the code, this function will analyze all the instructions in each block and identify the specific lines where the issues occur.

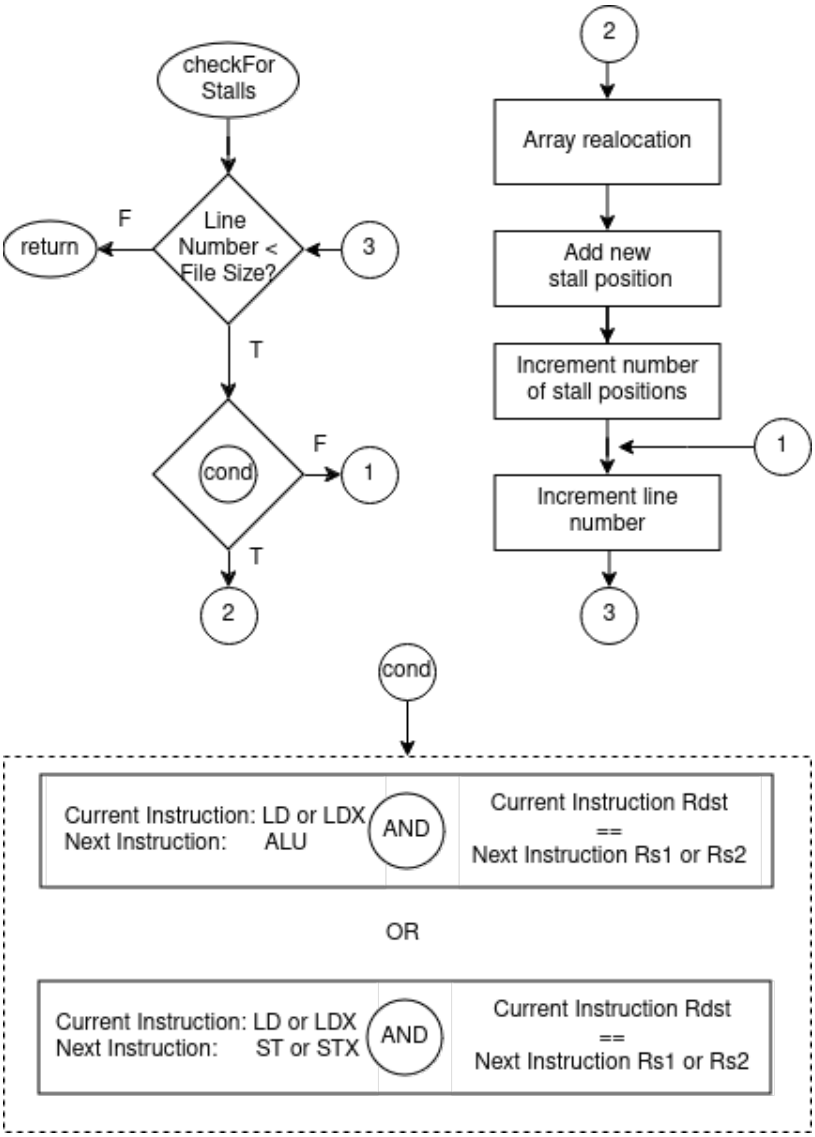


Figure 3.30: Stall searcher flowchart.

Replace positions Checker

After identifying all the stall positions, it becomes possible to determine the appropriate replacement position for each one. Initially, the sequence of instructions following a stall is analyzed. If a solution is found, the analysis resumes from the next stall. However, if a new stall is encountered before determining a replacement position, the analysis will restart from that new stall. Figure 3.31 illustrates this process.

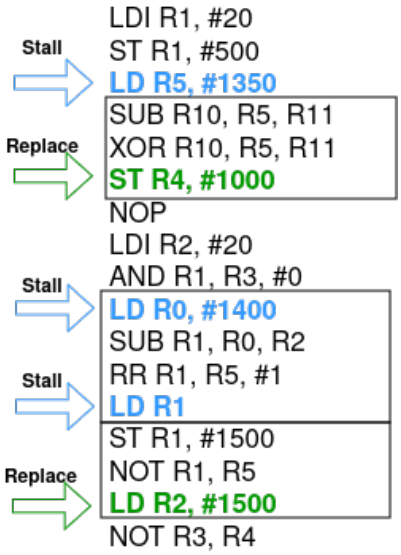


Figure 3.31: Limits for checking replace instructions.

To determine the appropriate replacement position, an approach involving two tables was used. One table contains information about the registers used as destinations (rdst) after the stall, and the other tracks the registers used as operands (rs1 and rs2). Once a suitable instruction for replacement is found, these tables are reset and ready for use after the next stall occurs.

In Figure 3.32, the table filling process begins with the instruction following the LOAD. From there, a check is conducted for the type of instruction and the type of registers on which it depends. If successful, it is identified as a replace position. Otherwise, the table is filled with the respective registers and the reading of the next instruction is carried out.

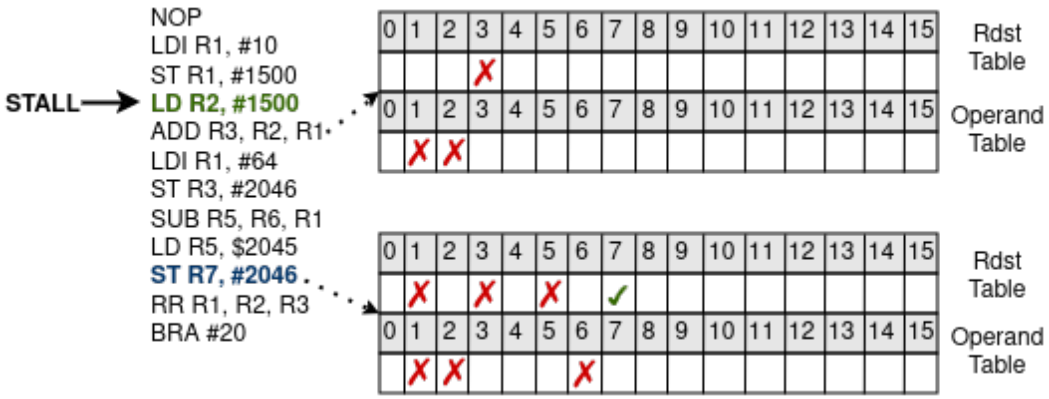


Figure 3.32: Register table for dependencies.

In figure 3.33 there is a flowchart demonstrate in a abstract way the behaviour of this function.

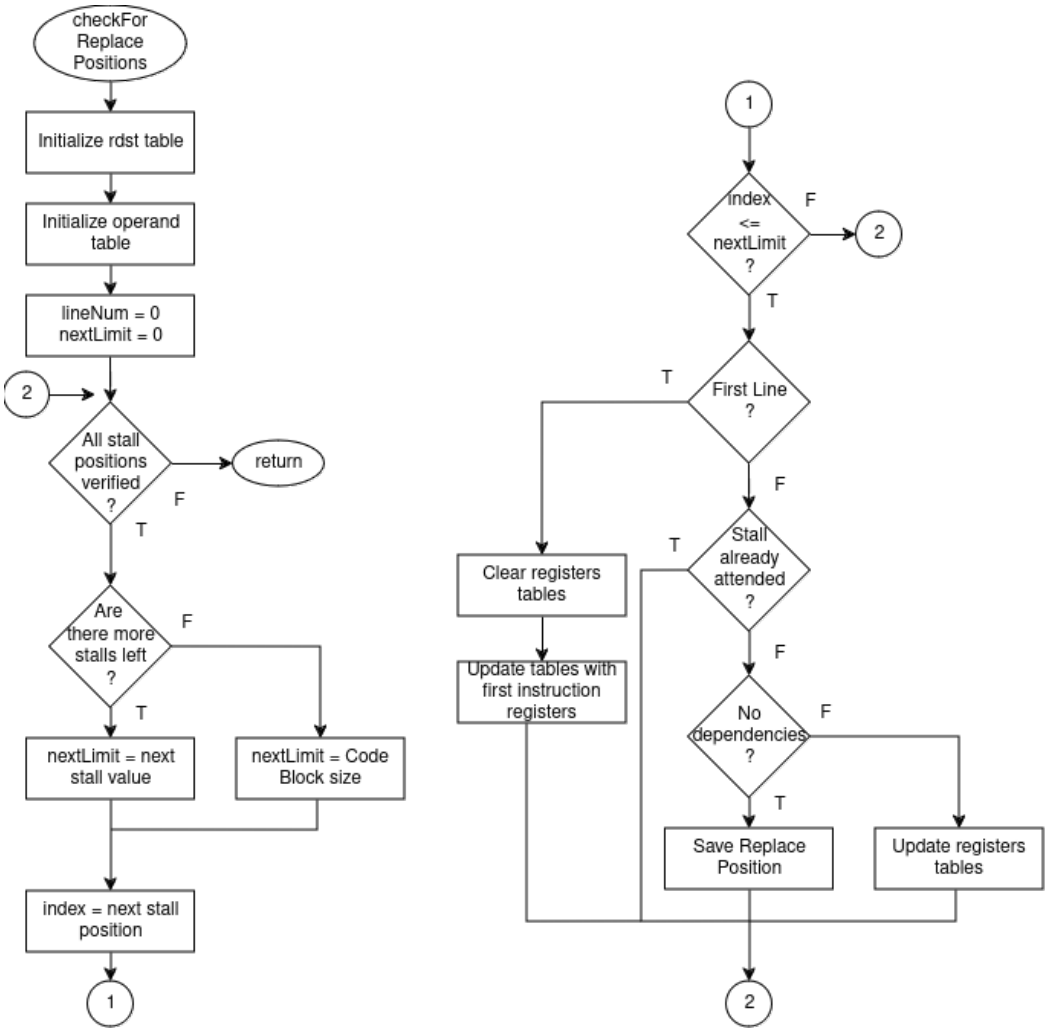


Figure 3.33: Replace position seacher flowchart.

Generated file

Finally, after all stall and replacement positions have been identified, the file with the properly altered code will be generated. Thus, whenever there is a stall + replace position set, it is necessary to first place the stall instruction, followed by the replace instruction, and finally, the set of instructions between the two positions. In the Figure 3.34, it is possible to see a diagram that represents the abstraction of a code, where the stall instructions are in blue, the replace instructions in yellow, and those between the two are in purple. In figure 3.35 there is a diagram with this function flowchart.

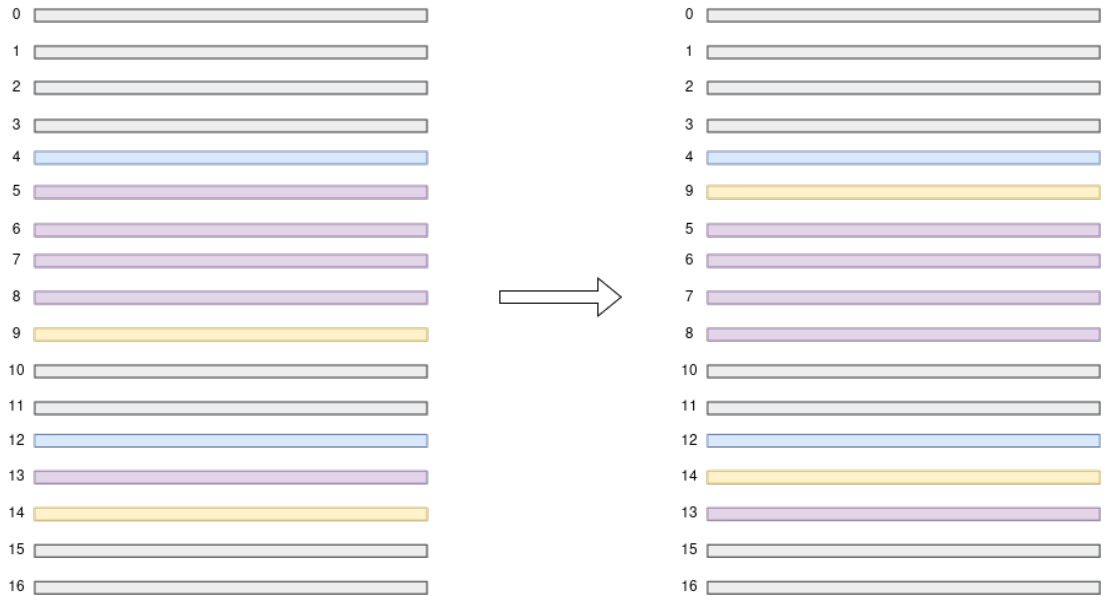


Figure 3.34: Instruction replacement.

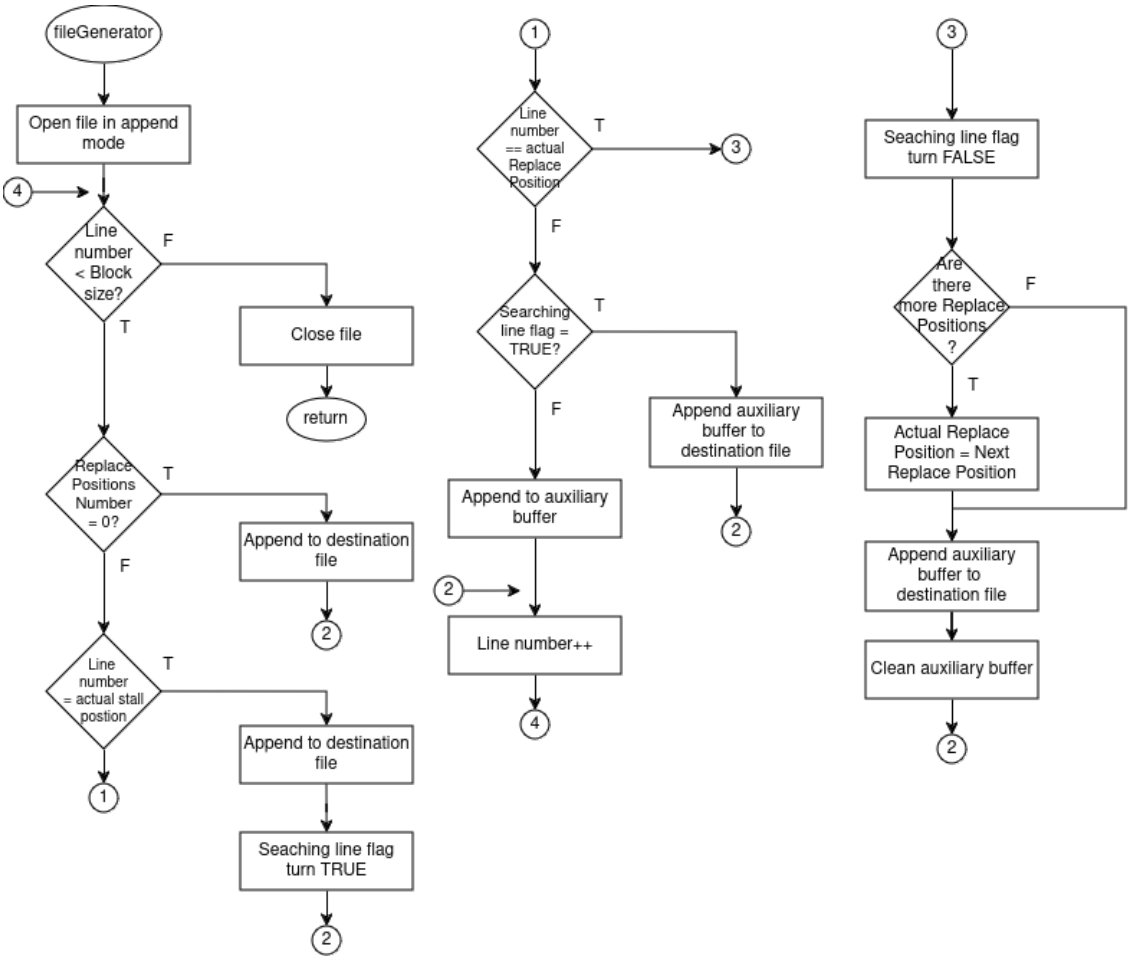


Figure 3.35: File generation flowchart.

3.6 Code Generation

Code generation is the final step of the compiler and is responsible for, as the name implies, generate code that can be ran on the target device from the intermediate representation. In the case of the compiler at hand, this is done in two distinct phases: first, the intermediate representation is converted into assembly code; then, the assembly code serves as an input to the integrated assembler which then converts it to binary code that the VeSPA SoC will be able to execute.

3.6.1 Assembly Code Generation

As previously referred, this is the first phase of the compiler's code generation step. The abstract syntax tree, after undergoing semantic analysis, serves as an input and it's corresponding code is generated in a recursive way. In order to do this, some functions were developed to generate code for each node by filtering the type of operation and were specialised according to the templates designed.

The main actors in code generation are the *executeCodeGen*, *generateCode*, *parseNode* and *parseOperatorNode* functions, the inner-workings of the functions listed will be explained in more detail below.

Execute Code Generation

This function starts code generation and is called after the optimisations on the AST have been completed. It receives as a parameter the root of the AST and the file where it will write all the assembly instructions before this file is delivered to the assembler.

The main purpose of the *executeCodeGeneration* function is to call the *generateCode* function, which is responsible for traversing the tree and generating code recursively.

The following figure 3.36 shows the function behaviour:

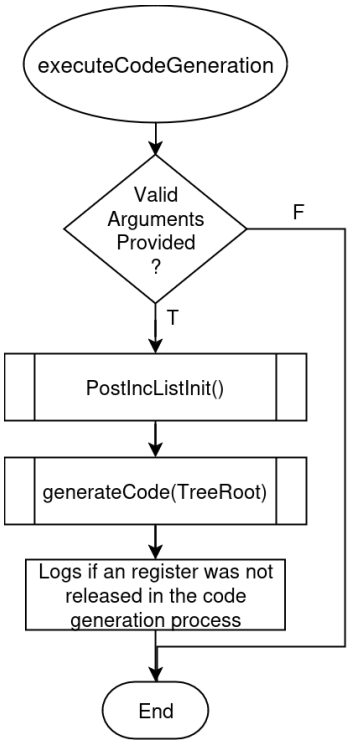


Figure 3.36: Execute code generation function behaviour flowchart.

Generate Code

As previously stated, this function is one of the main elements in code generation. It will initially receive the root of the AST and recursively call it to go through all the nodes and process them correctly.

This function receives a node from the AST as a parameter and then invokes the *parseNode* function, which will filter the node and its type, assess whether it is a terminal or non-terminal node, as well as the type of operation involved in the node being processed. It will go through the AST until it finds no more siblings of the node that has been passed to it.

The following figure 3.37 shows the function behaviour:

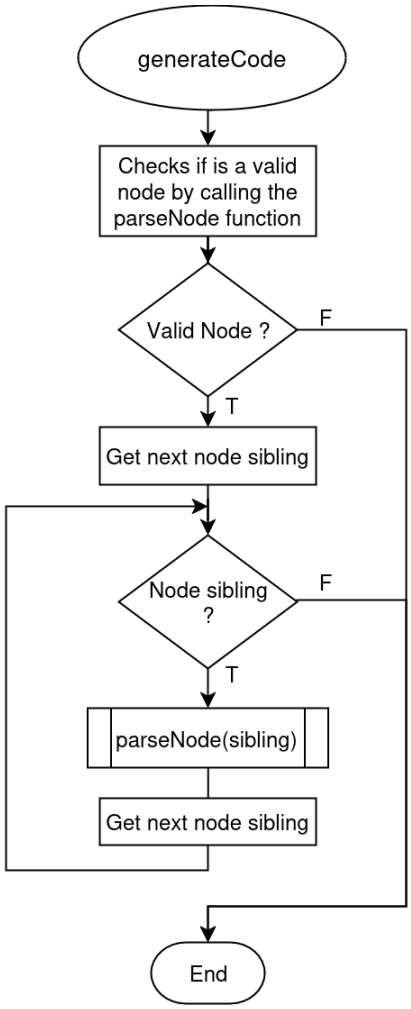


Figure 3.37: Generate code function behaviour flowchart.

Parse Node

The different nodes corresponding code is generated through the use of node-specific templates. The Parse Node function associates the current node to the corresponding template depending on it's type.

The following figure 3.38 shows the function behaviour:

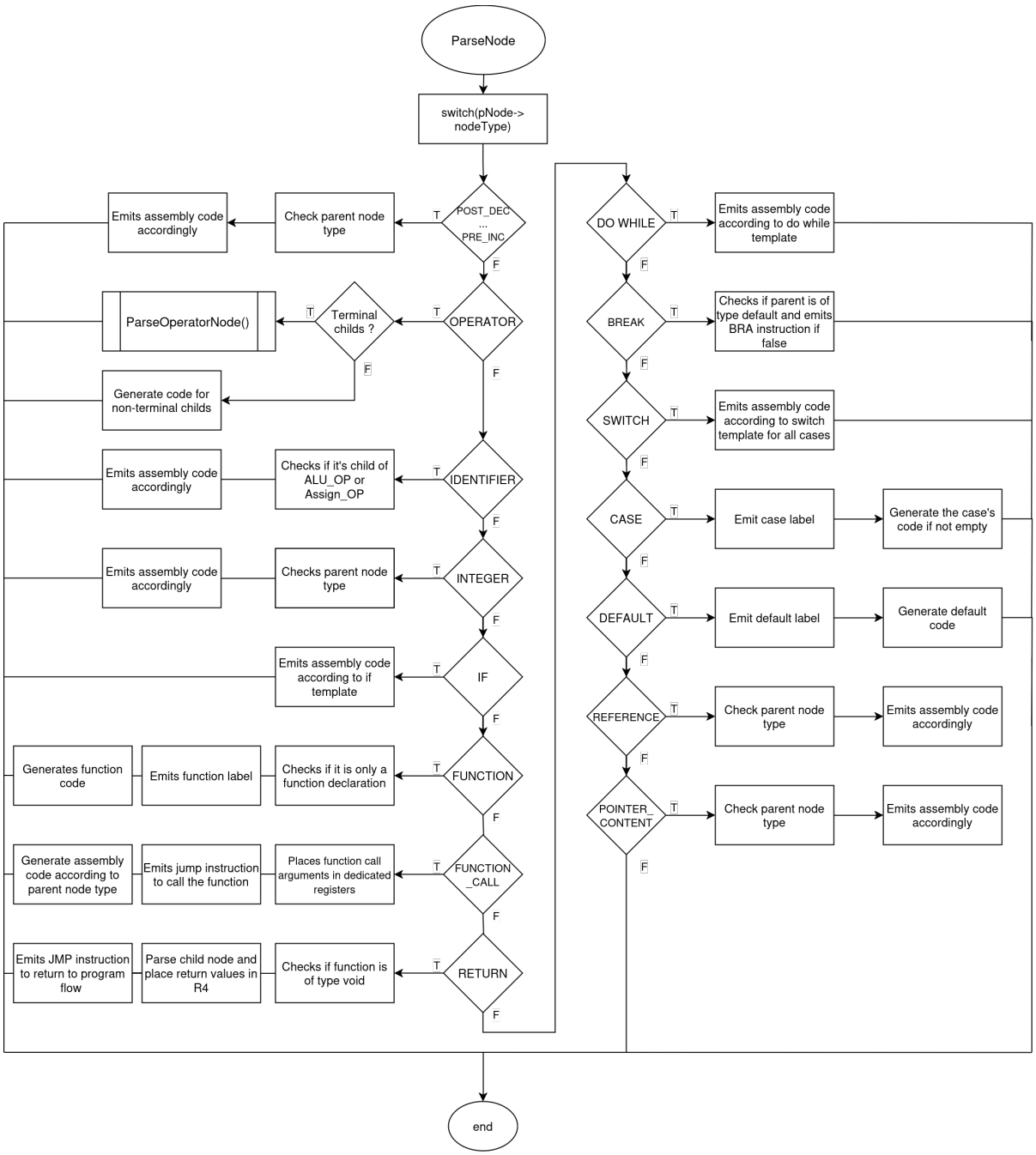


Figure 3.38: Parse node function behaviour flowchart.

Regarding this function, it is also worth noting how the if, while and do...while statements are generated. The figures below depict the statement's diagrams as well as the corresponding assembly code generation templates for each one.

- IF:

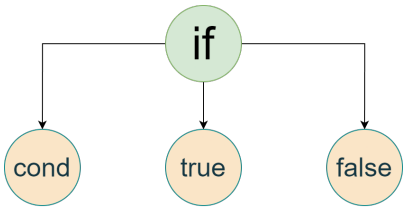


Figure 3.39: IF statement diagram.

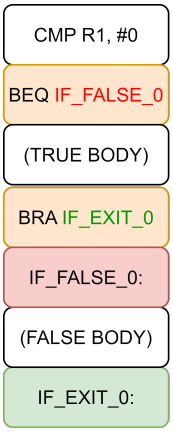


Figure 3.40: IF statement template.

- WHILE:

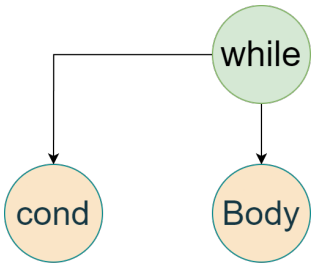


Figure 3.41: WHILE statement diagram.

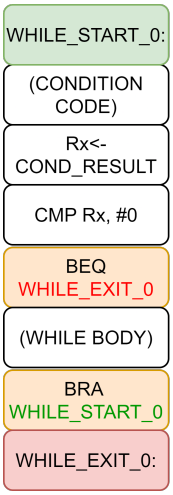


Figure 3.42: WHILE statement template.

- DO WHILE:

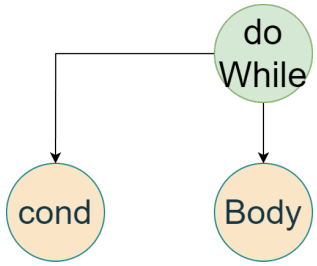


Figure 3.43: DO WHILE statement diagram.

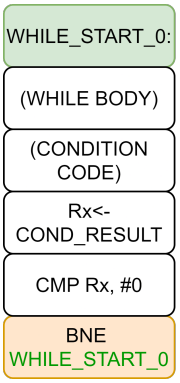


Figure 3.44: DO WHILE statement template.

- SWITCH:

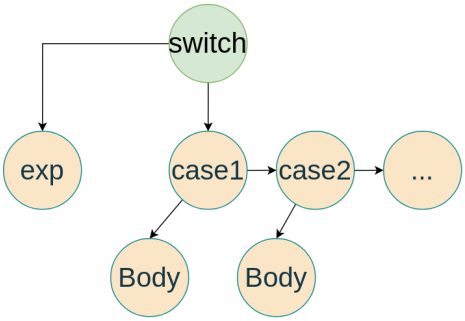


Figure 3.45: SWITCH statement diagram.

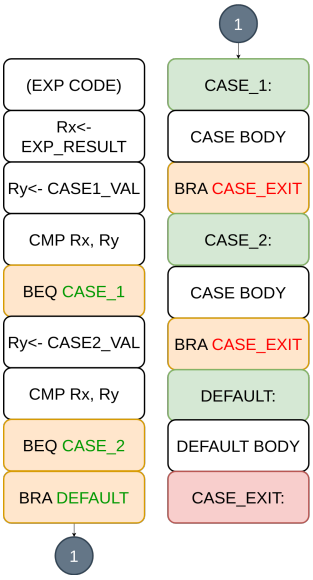


Figure 3.46: SWITCH statement template

Parse Operator Node

The Parse Operator Node Function is complementary to the Parse Node function. As the name implies, it has the same purpose of the latter, but handles the specific case of the operator node, this being one of the most complex nodes in terms of code generation. The following figure 3.47 shows the function behaviour. In the referred diagram, it is possible to check that the function in question makes use of three additional functions, to handle the generation of three different types of operations: ALU related, boolean and assign operations. The referred auxiliary function's behaviour will be further exposed in the following subsections.

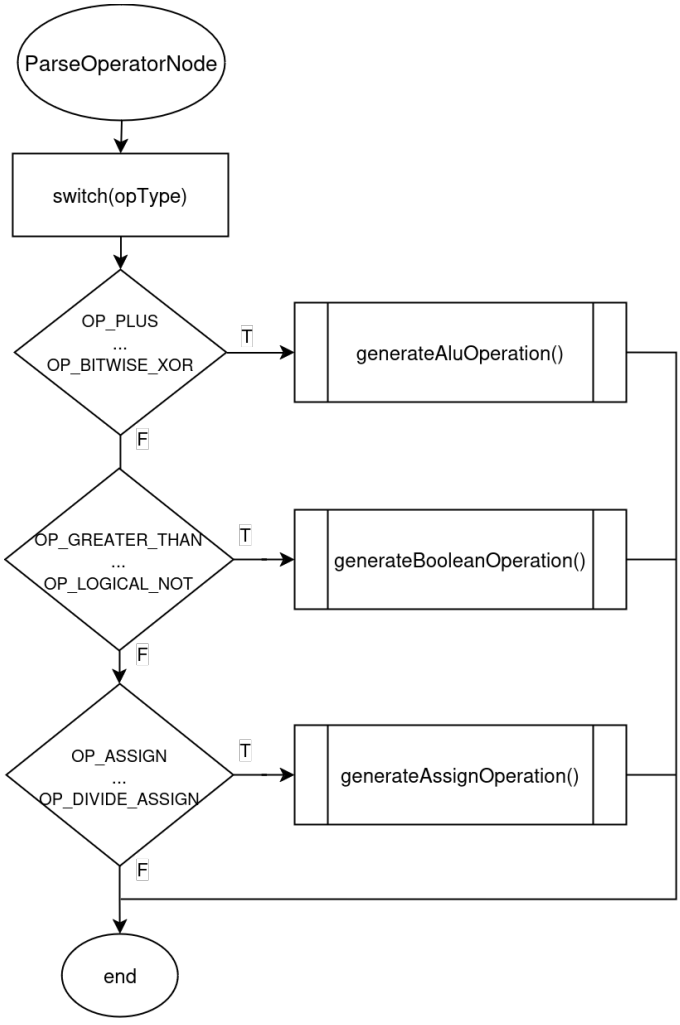


Figure 3.47: Parse operator node function behaviour flowchart.

Generate ALU Operation

As the name implies, this function is responsible for generating code for ALU operations, simplifying the operator nodes code generation process given that these are implemented in a related way. The ALU related operations that this function handles are: plus, minus, right and left shifts, multiply, divide, remain and bit-wise AND, NOT, OR and XOR.

The following figure 3.48 shows the function behaviour:

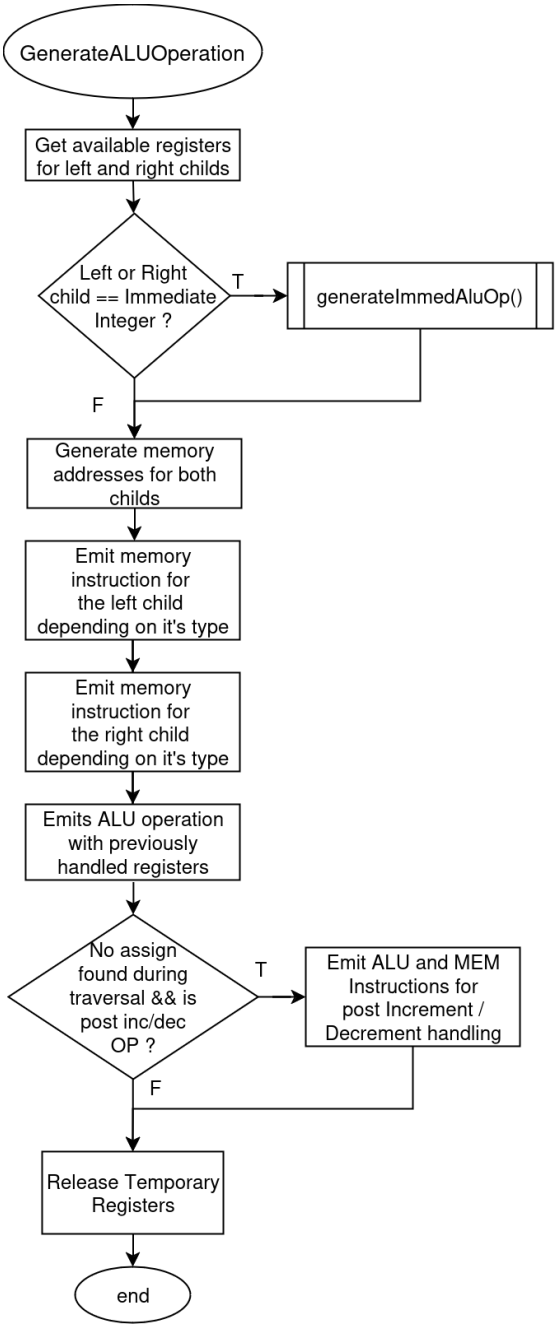


Figure 3.48: Generate ALU operation function behaviour flowchart.

Generate Boolean Operation

In similar fashion to the previous function, the Generate Boolean Operation function is responsible for emiting the code associated with operations such as greater than, less than, equal, not equal, and logical AND, OR and NOT, among some others. The following figures depict said operators and the VeSPA assembly code templates associated with them:

- **Equal, not equal, greater than, greater than or equal, less than, less than or equal:**

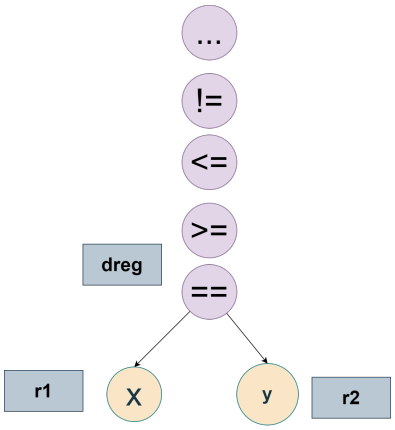


Figure 3.49: Boolean operators diagram.

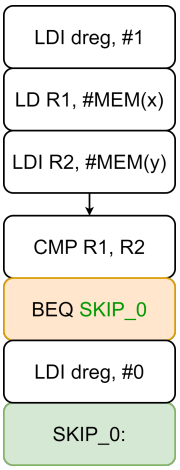


Figure 3.50: Boolean operators template.

- **Logical AND:**

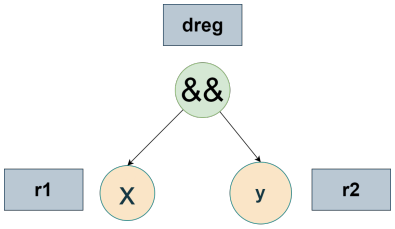


Figure 3.51: Logical AND operator diagram.

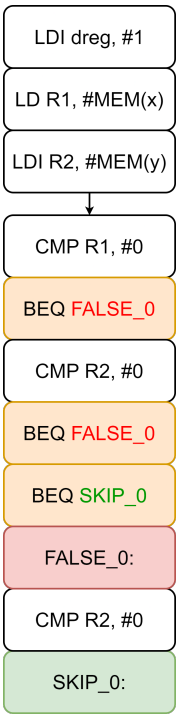


Figure 3.52: Logical AND operator template.

- **Logical OR:**

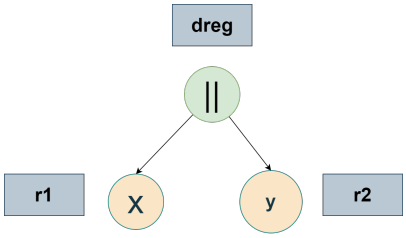


Figure 3.53: Logical OR operator diagram.

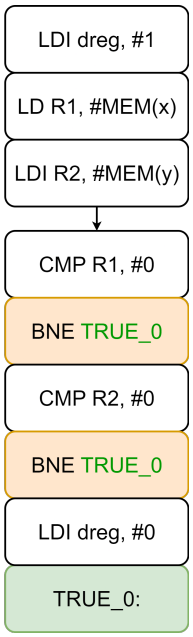


Figure 3.54: Logical OR operator template.

- **Logical NOT:**

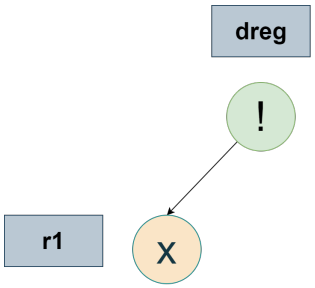


Figure 3.55: Logical NOT operator diagram.

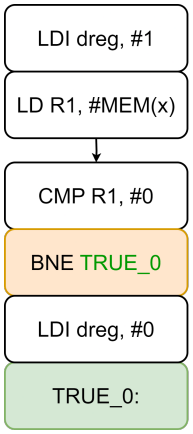


Figure 3.56: Logical NOT operator template.

Generate Assign Operation

There is a set of different assign related nodes and each has a specific template for code generation. The Generate Assign Operation function is responsible for the generation of the code associated with each specific referred assign node.

The following figure 3.57 shows the function behaviour:

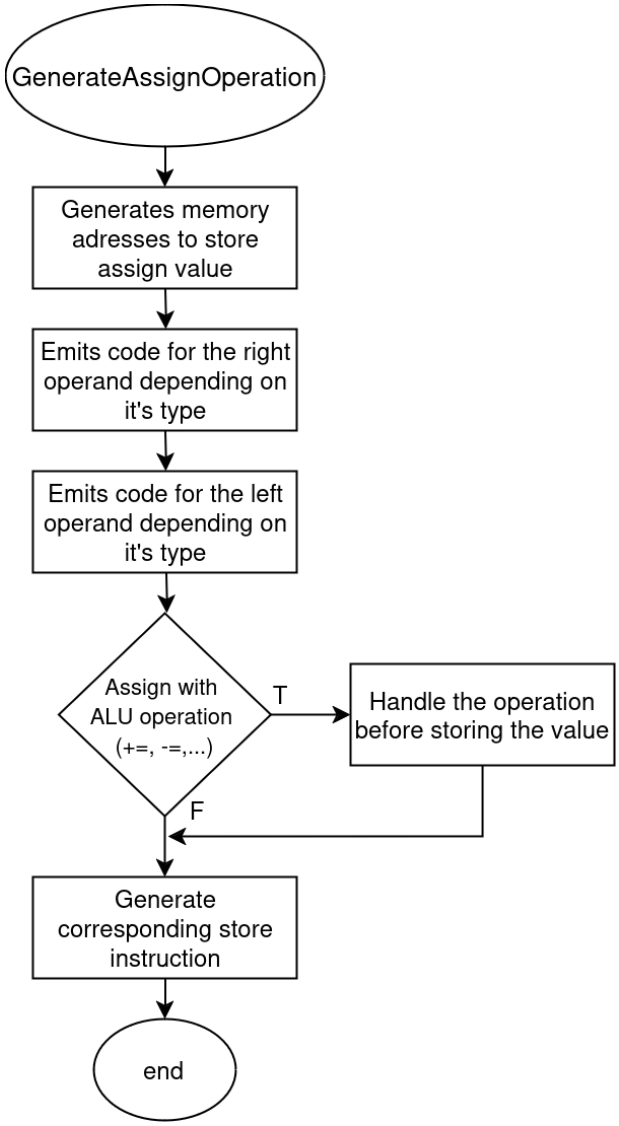


Figure 3.57: Generate assign operation function behaviour flowchart.

Register attribution

For the dynamic allocation of registers, methods were developed to deal with the limited number of registers and how they would be allocated and deallocated according to needs, following the ABI.

Firstly, the list of registers was developed following the ABI, which indicates that R12 to R31 are temporary registers and these will be used throughout the code generation. In this list of registers, each register has a name and a status that will indicate whether it is fit for use or not.

Next, the *getNextAvailableReg* function was developed, which goes through the list of registers and returns the first free register to be used from the list, thus allowing a dynamic allocation of registers depending on their use throughout the generation and execution of code.

In order to reuse registers and obtain an efficient allocation of registers, it was also necessary to implement the *releaseReg* function that releases previously allocated registers, which receives a register as a parameter, traverses the list of register states until it finds the desired register and releases it if it is allocated.

Dynamic execution environment

Since this is a C compiler, the class had to design a dynamic execution environment in order to support local and global variable declarations, as well as recursive calls to functions. In order to accomplish that, the stack handling had to be designed. The diagram in Figure 3.58 contains the stack handling details.

Position	Contents	Frame
	...	**Previous**
R2 + n	Argument 0	
	...	
R2 + 1	Argument n	
R2	Previous frame pointer	**Current**
R2 - 1	Return address	
R2 - 2	Local Variable 0	
	...	
R2 - m	Local Variable m	

Figure 3.58: Stack frame design.

The R2 and R3 registers are respectively specified in the ABI as the frame pointer and the stack pointer. As it can be seen, when calling a new function, its arguments are pushed onto the stack, then the frame pointer is stored as well as a copy of the stack pointer into the frame pointer. After that we save the return address and execute a jump to the function, where we will push its local variables later on.

Caller and callee function templates

In order to generate the function properly, the group designed the function templates for the caller and the callee, which are represented in Figures 3.59.

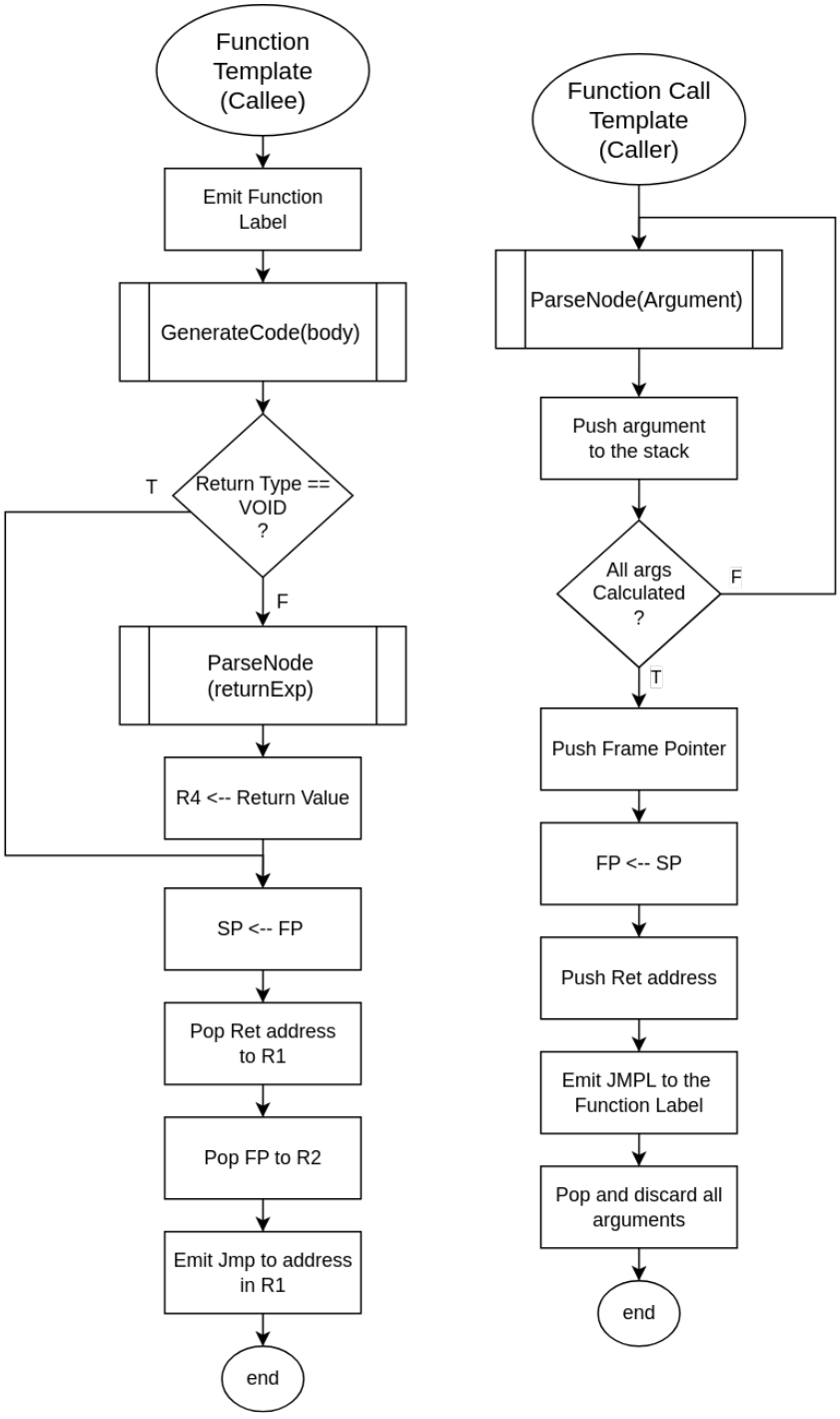


Figure 3.59: Flowchart of function callee and caller template.

3.6.2 Assembler

Lexical Analysis

Lexical analysis is the first step in the compilation process, responsible for converting a sequence of characters into a sequence of tokens. This assembler was developed specifically for the *VeSPA* architecture, and therefore includes a specification that reflects the requirements and conventions of that architecture.

Figure 3.60 includes all the words that represent instructions and operations recognized by the as-

sembler. These words are crucial for defining the operations that the code will perform and are translated directly into machine instructions.

Reserved Words				
ADD	SUB	OR	AND	NOT
XOR	CMP	RR	RL	MOV
JMP	JMPL	LD	LDI	LDX
ST	STX	RETI	HALT	NOP
BRA	BCC	BVC	BEQ	BGE
BGT	BPL	BNV	BCS	BVS
BNE	BLT	BLE	BMI	

Figure 3.60: Assembler reserved words.

Figure 3.61 presents the directives used by the assembler to manage the structure and content of the code. Directives are not directly translated into machine instructions but influence how the code is assembled and organized.

Directives	Description
.org	Sets LC to specified value
.byte	Reserve and initialise memory space
.word	Reserve and initialise memory space
.alloc	Reserve memory
.equ	Assigns value to specific name

Figure 3.61: Assembler directives.

The Figure 3.62 contains symbols that the assembler recognizes and uses for various formatting and structuring functions within the code. These include separators and delimiters like the comma (,), comment symbols like the hash (#), operation symbols such as the plus (+), among others.

Special Symbols				
,	#	:	+	(
)	-	\0	$	;

Figure 3.62: Assembler special symbols.

The *add_symbol* function is used to add symbols to the symbol table when an identifier is recognized during lexical analysis (Figure 3.63).

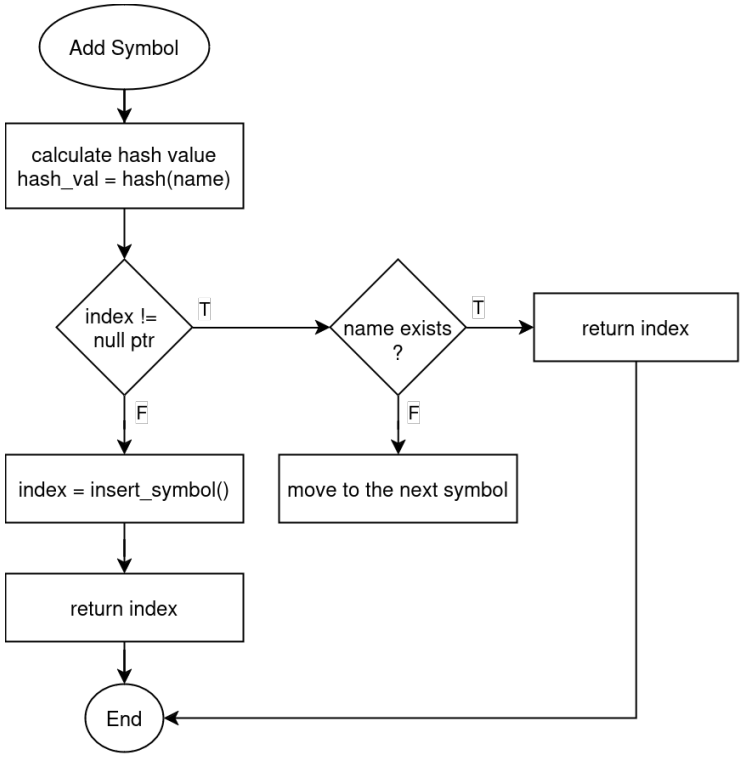


Figure 3.63: Add symbol algorithm flowchart.

The *hash* and *insert_symbol* functions are essential components that support the *add_symbol* function, responsible for adding symbols to the symbol table.

The *hash* function is used to calculate a hash value from a symbol name. This hash value is a numeric representation that makes it easier to store and search for symbols in the symbol table (Figure 3.64).

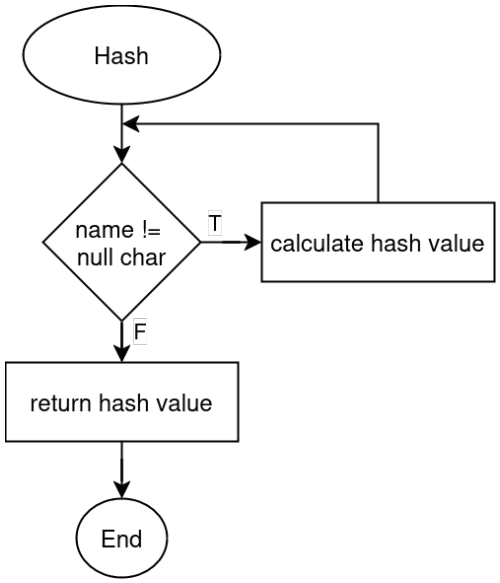


Figure 3.64: Hash algorithm flowchart.

The *insert_symbol* function is called by *add_symbol* when a symbol is not found in the symbol table. It inserts the new symbol into the table and updates the hash table (Figure 3.65).

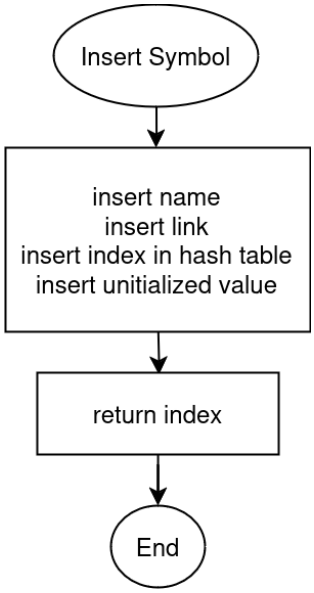


Figure 3.65: Insert symbol algorithm flowchart.

Syntactic Analysis

Syntactic analysis is the second step in the compilation process, where the sequence of tokens generated by the lexical analyzer is checked with the language grammar. In the context of an assembler for the *VeSPA* architecture, parsing is simplified compared to that of a C compiler, due to the direct correspondence between lines of assembly code and machine instructions. Because of this, there is no need to construct an AST as the focus is on checking the structure of instructions and directives, ensuring they are correct and can be translated directly into machine code.

To store the statements effectively, it was necessary to create a struct that includes all the parameters required by the statements, ensuring that each statement has all the relevant information encapsulated for easy access. These structs are then connected in a dynamically allocated list of statements, allowing for efficient management and access throughout the compilation process.

Figure 3.66 provides a visual representation of this struct.

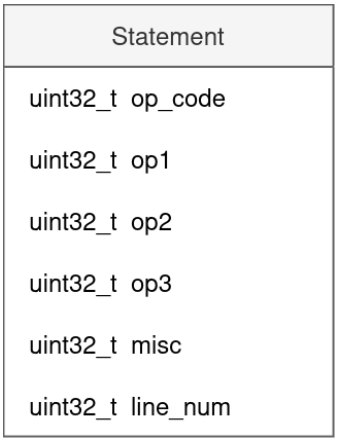


Figure 3.66: Assembler statement structure.

The *add_statement* function, presented in Figure 3.67, is responsible for adding statements to the statements list and it ensures that all necessary information for each statement is stored correctly.

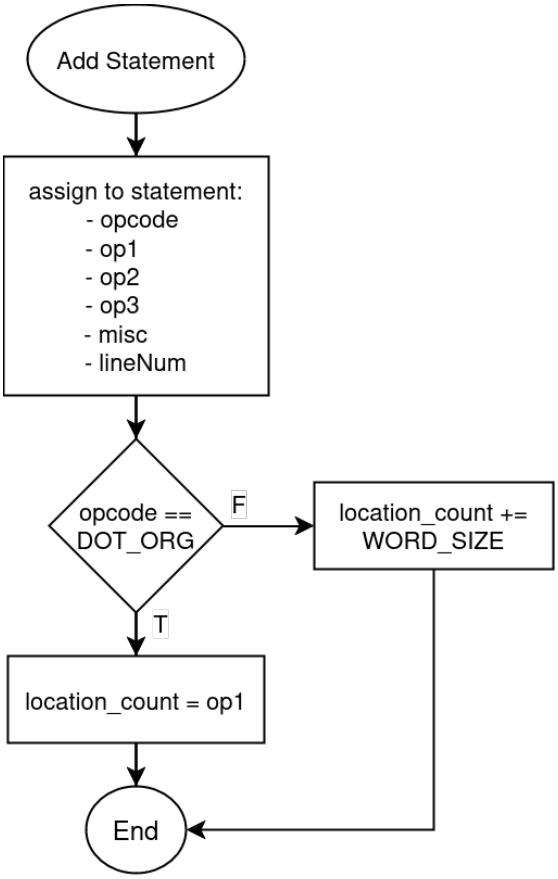


Figure 3.67: Add statement algorithm flowchart.

3.6.3 Assembler Statements

Figure 3.68 shows all statements accepted by the assembler, classified into three distinct categories, represented by different colors.

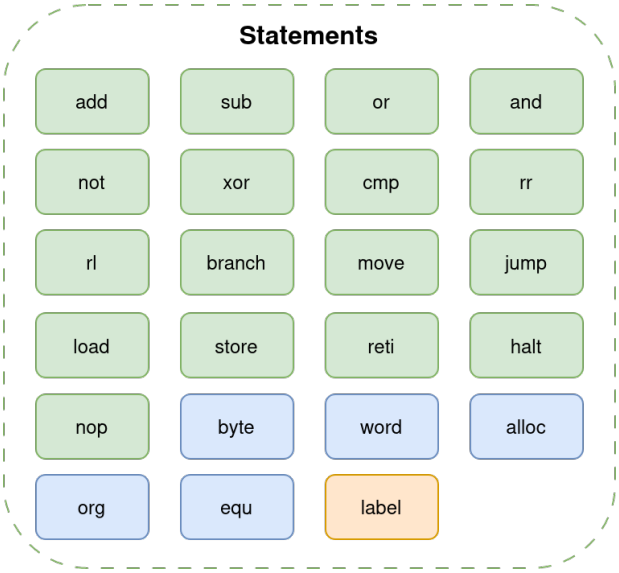


Figure 3.68: Assembler supported statements.

In green, there are statements that correspond to operations directly supported by the *VeSPA* architecture, including arithmetic, logical, flow control and data manipulation instructions. However, some instructions presented do not exist in the *VeSPA* architecture. The *MOV* instruction, for example, does not exist in the architecture and is treated as a zero-sum *ADD*. The *RR*, *RL* and *RETI* instructions, despite not being originally part of the *VeSPA* architecture, have support added. Each of these statements is translated directly into machine instructions that the processor can execute.

In blue, there are special directives that the assembler interprets and executes, but which do not correspond directly to machine instructions. For example, the *org* directive defines the memory location of the following code, while the *equ* directive is responsible for defining constant variables.

Lastly, in orange, the *label* is used to identify a specific position. Labels are not executable instructions, but are often used by other instructions to determine jump points.

Machine Code Generation

The second step of the assembler is responsible for generating binary code from the assembly instructions obtained during Step 1. However, this developed solutions takes a different approach by generating an output file in hexadecimal format, which includes the address of each instruction line. This approach simplifies the conversion process to binary in a COE format, which is the format used by Vivado to initialize memory instances. During this phase, the list of statements and the symbol table, created and filled in previously, are used to translate the assembly instructions into executable machine code, according to the VeSPA specification.

The *generate_code* function goes through the list of statements, translating each one into its binary representation. This process involves the use of a switch that handles the various opcodes specific to the VeSPA architecture. The final output of Step 2 is binary code that can be loaded and executed by the VeSPA architecture. The algorithm can be seen in the Figure 3.69.

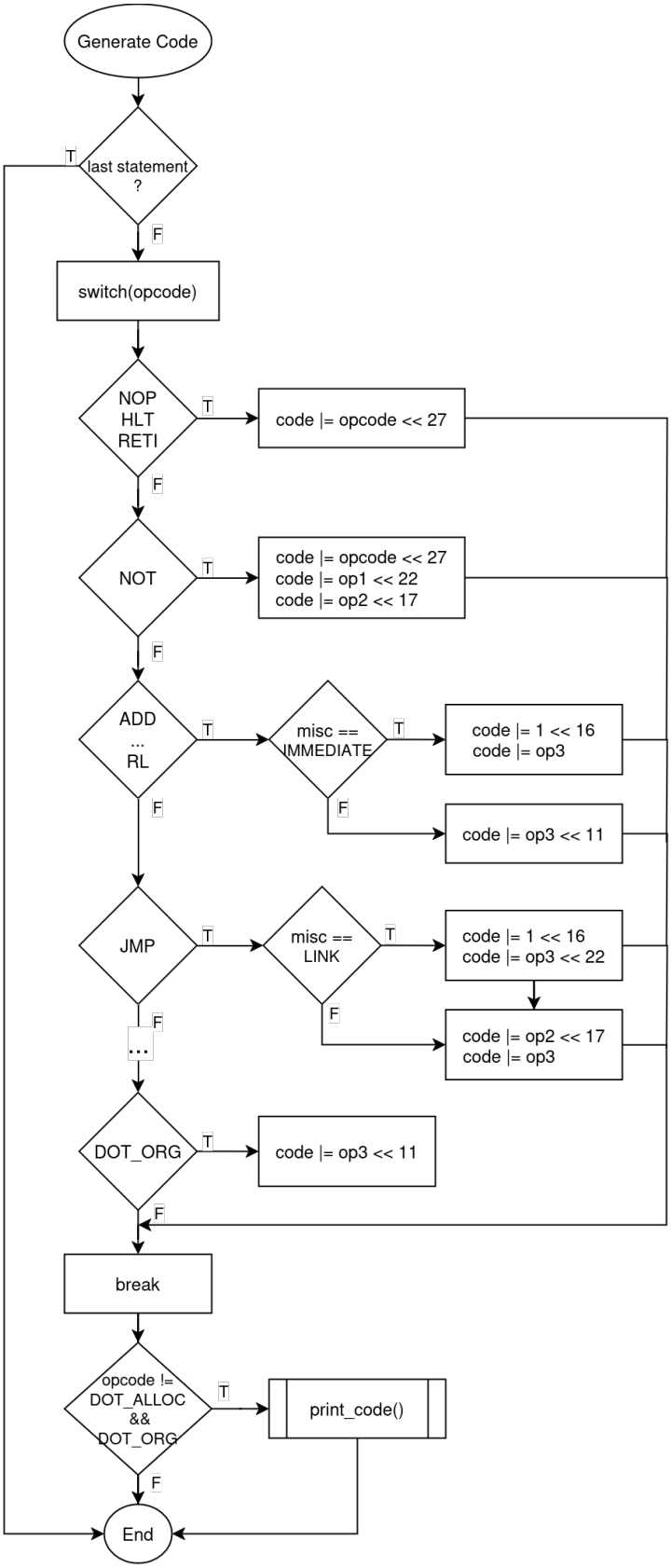


Figure 3.69: Generate code algorithm flowchart.

4 | Implementation

With the design phase complete, the next critical step is to implement the design described during the previous phase. This chapter delves into the detailed process of transforming the theoretical design into a working compiler, exploring the development stages, from parsing and lexical analysis to optimization and code generation.

4.1 Lexical Analysis

Considering the imposed restriction to use the GNU Flex scanner generator, the construction of a .l file is of utmost importance as it is responsible for defining the rules that dictate how tokens are recognized and processed within the system. The .l file has a specific format divided in 4 parts: C global variables, function prototypes and comments; variable definitions; rules; C auxiliary routines.

Thus, as depicted in the following code snippet, we encounter the initial code section. This section facilitates the inclusion of C or C++ code directly within the .l file, which will subsequently be integrated into the lexer generated by Flex.

As previously mentioned, the source code is read character by character, analyzing each character as it is read rather than reading and analyzing the next one immediately. This solution would bring advantages in terms of efficiency. To achieve this and increase efficiency, a buffer with space for 32 characters was implemented. Consequently, each character that is read will be automatically inserted into this buffer until it is completely filled. After that, Flex will proceed to analyze the characters one by one. With this solution, only when Flex finishes analyzing them, will another 32 characters be read and analyzed. The code implemented for this buffer is presented below in Alg. 4.1.

```
#define YY_BUFFER_SIZE 32

static char yy_buffer[YY_BUFFER_SIZE];
static char* yy_buf_pos = yy_buffer;
static size_t yy_buf_size = 0;
static int endOfFileReached = 0;

#define YY_INPUT(buf,result,max_size) \
{ \
    if (!endOfFileReached && yy_buf_pos >= yy_buffer + yy_buf_size) { \
        memset(yy_buffer, 0, YY_BUFFER_SIZE); \
        yy_buf_size = fread(yy_buffer, 1, YY_BUFFER_SIZE, yyin); \
        yy_buf_pos = yy_buffer; \
        if (yy_buf_size <= 0) { \
            endOfFileReached = 1; \
            result = YY_NULL; \
        } \
    } \
    size_t n = YY_BUFFER_SIZE - (yy_buf_pos - yy_buffer); \
    if (n > max_size) n = max_size; \
    memcpy(buf, yy_buf_pos, n); \
    yy_buf_pos += n; \
    result = n; \
}
```

Algorithm 4.1: Streaming of the source code to the buffer input.

Within the definitions section, responsible for defining all names for regular expressions, declaring constants and defining substitution rules, the regular expressions, depicted in the Alg. 4.2, are represented.

```
digit        [0-9]
number       {digit}+
hexNumber    0[xX][0-9a-fA-F]+
binNumber    0[bB][01]+
floatNumber {digit}*"."{digit}+({expo}{digit}+)?

identifier   [a-zA-Z_][a-zA-Z0-9_]*
string       ["]([^"\\\n]|\\.|\\\n)*["]
char         '([^'\\\n]|\\.)'
newline      \n
sign         ([-+])
expo         [eE]{sign}?
whitespace   [ \t]+
```

Algorithm 4.2: Definitions section of the .l file.

In the rules section, the definitions of how to parse the input character stream, as presented in the Figure 3.1, were placed. Below, in Alg. 4.3, some of these rules are presented.

```
"break"      { LOG_DEBUG_SHORT("BREAK ");    return TOKEN_BREAK;    }
"case"       { LOG_DEBUG_SHORT("CASE ");     return TOKEN_CASE;     }
"char"       { LOG_DEBUG_SHORT("CHAR ");     return TOKEN_CHAR;     }
"const"      { LOG_DEBUG_SHORT("CONST ");    return TOKEN_CONSTANT; }
"continue"   { LOG_DEBUG_SHORT("CONTINUE "); return TOKEN_CONTINUE; }
"default"    { LOG_DEBUG_SHORT("DEFAULT ");  return TOKEN_DEFAULT;  }
"do"         { LOG_DEBUG_SHORT("DO ");       return TOKEN_DO;       }
"double"     { LOG_DEBUG_SHORT("DOUBLE ");   return TOKEN_DOUBLE;   }
"if"         { LOG_DEBUG_SHORT("IF ");       return TOKEN_IF;       }
"else"       { LOG_DEBUG_SHORT("ELSE ");     return TOKEN_ELSE;     }
"enum"       { LOG_DEBUG_SHORT("ENUM ");     return TOKEN_ENUM;     }
"extern"     { LOG_DEBUG_SHORT("EXTERN ");   return TOKEN_EXTERN;   }
"float"      { LOG_DEBUG_SHORT("FLOAT ");    return TOKEN_FLOAT;    }
"for"        { LOG_DEBUG_SHORT("FOR ");      return TOKEN_FOR;      }
"goto"       { LOG_DEBUG_SHORT("GOTO ");     return TOKEN_INT;      }
"long"       { LOG_DEBUG_SHORT("LONG ");     return TOKEN_GOTO;     }
"int"        { LOG_DEBUG_SHORT("INT ");      return TOKEN_LONG;     }
"register"   { LOG_DEBUG_SHORT("REGISTER "); return TOKEN_REGISTER; }
"return"     { LOG_DEBUG_SHORT("RETURN ");   return TOKEN_RETURN;   }
"short"      { LOG_DEBUG_SHORT("SHORT ");    return TOKEN_SHORT;    }
    ...
```

Algorithm 4.3: Rules section of the .l file - definitions.

Still within the rules section, it is possible to see in Alg. 4.4 the definition of patterns that match specific tokens in the input buffer. That is, when it detects a regular expression defined in the definitions section, it executes the corresponding action.

```
{char}          { LOG_DEBUG_SHORT("CHAR ");
    yylval.nodeData.dVal = (long int) yytext[1];
    return TOKEN_CNUM;}
{binNumber}     { LOG_DEBUG_SHORT("BIN_NUM ");
    yylval.nodeData.dVal = strtol(yytext, NULL, 2);
    return TOKEN_NUM; }
{hexNumber}     { LOG_DEBUG_SHORT("HEX_NUM ");
    yylval.nodeData.dVal = strtol(yytext, NULL, 16);
    return TOKEN_NUM; }
{number}        { LOG_DEBUG_SHORT("NUM ");
    yylval.nodeData.dVal = strtol(yytext, NULL, 10);
    return TOKEN_NUM; }
{string}        { LOG_DEBUG_SHORT("STR ");
    StringCreateAndCopy(&(yylval.nodeData.sVal),yytext,0);
    return TOKEN_STR; }
{identifier}    { LOG_DEBUG_SHORT("ID ");
    StringCreateAndCopy(&(yylval.nodeData.sVal),yytext,0);
    return TOKEN_ID;  }
{floatNumber}   { LOG_DEBUG_SHORT("FNUM ");
    yylval.nodeData.fVal = strtod(yytext, NULL);
    return TOKEN_FNUM;}
{newline}       { LOG_DEBUG_SHORT(" \n");   incrementLineNumber(1);}
{whitespace}    { /*  ignore */}
```

Algorithm 4.4: Rules section of the .l file - patterns.

Lastly, lexical analysis states were defined. Each state, depicted in Alg. 4.5, has its own rules to handle the input character flow.

In the INITIAL state, the lexer is outside of any comment and is ready to recognize the usual token patterns. On the other hand, in the INCOMMENT state, the lexer is within a comment block. It handles end-of-file conditions, marking an error if the comment is not properly closed. Upon encountering the end of the comment ("*/"), it transitions back to the INITIAL state. Additionally, it consumes newline characters and characters within the comment block. Lastly, it handles end-of-file conditions, marking an error if the comment is not properly closed. Upon encountering the end of the file («EOF»), it terminates the lexer.

```
"//".*          { /* When "//" is found, zero or more occurrences of
    any character are ignored */}

<INITIAL>{      // INITIAL Lexical state
  "/*"          { LOG_DEBUG_SHORT(" [In comment]");} BEGIN(INCOMMENT);
}

<INCOMMENT>{    // INCOMMENT Lexical state
  <<EOF>>       { LOG_DEBUG_SHORT(" Error: Unterminated comment\n");
                  return TOKEN_ERROR;
                }
```

```
11   "*/"          { LOG_DEBUG_SHORT(" [Out comment]");}BEGIN(INITIAL);
12   "\n"          { incrementLineNumber(1);};
13   [^*\n]+       ; // Consume characters within the comment block
14 }
15
16 <<EOF>>         {yyterminate(); return TOKEN_EOF; }
17
18 . {LOG_ERROR("[%s] is an invalid token\n", yytext);   YY_FATAL_ERROR("
     Unrecoverable error in lexer"); return TOKEN_ERROR;}
```

Algorithm 4.5: Rules section of the .l file - lexical states.

4.2 Syntactic Analysis

Considering GNU Bison is the parser generator that shall be used, the creation of a .y file is crucial, as it defines the grammar rules that guides the syntactic analysis of the source code. The .y file is structured into three main sections: definitions; rules; and user subroutines. Each of these sections plays a vital role in the construction of the parser.

The efficiency of the parsing process is enhanced by Bison's ability to handle conflicts and ambiguities within the grammar, providing mechanisms such as precedence and associativity declarations to resolve them. Bison reads the input tokens generated by Flex and applies the grammar rules to construct the AST.

4.2.1 AST

In Alg. 4.6 the AST node struct is presented. This struct saves all attributes relative to a tree node. It has three pointers, one to the node's childs, other to the node's siblings and one to a symbol in the symbol table. It also saves the type of the node and the number of childs that a node has, among other important data. The attributes of a node will also be used later, in the semantic analysis, to build symbol tables and perform typechecking.

```
1  typedef struct TreeNode
2  {
3      struct TreeNode* pChilds;
4      struct TreeNode* pSibling;
5      size_t childNumber;
6      size_t lineNumber;
7
8      NodeType_et nodeType;
9      NodeData_ut nodeData;
10
11     VarType_et nodeVarType;
12     SymbolEntry_st* pSymbol;
13 }TreeNode_st;
```

Algorithm 4.6: AST node struct.

NodeCreate, *NodeAddChild*, *NodeAddNewChild* and *NodeAddChildCopy* are APIs provided to generate an AST with a proper hierarchical structure. By implementing efficient memory allocation and node management, it ensures that the AST can accurately represent the syntactic structure of the source code. This capability is fundamental for the subsequent phases of the compiler, such as semantic analysis and code generation, as the AST serves as the primary intermediate representation of the program being compiled.

In Alg. 4.7, the *NodeCreate* function is depicted, a function responsible for creating new nodes in the AST. It allocates memory for a new node and initializes it with a specified node type and line number.

```
int NodeCreate(TreeNode_st** ppNewNode, NodeType_et nodeType)
{
    TreeNode_st* pNode;

    if (!ppNewNode)
        return -EINVAL;

    *ppNewNode = (TreeNode_st*) calloc(1, sizeof(TreeNode_st));

    if (!(*ppNewNode))
    {
        LOG_ERROR("Failed to allocate memory!\n");
        return -ENOMEM;
    }

    pNode = (TreeNode_st*) (*(ppNewNode));
    pNode->nodeType = nodeType;
    pNode->lineNumber = getLineNumber();
    pNode->pSymbol = NULL;
    return 0;
}
```

Algorithm 4.7: Create AST node function.

4.2.2 Parser

In the definitions section of the .y file, we declare tokens and data types to be used by the generated parser. This section lays the foundation for how the parser will interpret the tokens provided by the lexical analyzer and the type of the variable that will be used to store the values of the tokens (terminal and non-terminal) in the parser. The data type to be used by the parser is specified with the Alg. 4.8.

```
%define api.value.type {ParserObject_ut}
```

Algorithm 4.8: Definition of the data type of the variables what will store tokens.

The type *ParserObject_ut* is an union, as can be seen below in Alg. 4.9.

```
typedef union
{
    TreeNode_st* treeNode;
    NodeData_ut nodeData;
}ParserObject_ut;
```

Algorithm 4.9: ParserObject_ut definition.

The rules section is the core of the .y file, where the grammar of the language is specified. Each rule defines a pattern and the corresponding action to be taken when the pattern is recognized. These actions can range from simple assignments to complex tree-building operations, ultimately enabling the construction of the AST. A simple example can be analyzed in code snippet Alg. 4.10.

```
R_RETURN    :   TOKEN_RETURN TOKEN_SEMI
                {
                    NodeCreate(&($$.treeNode), NODE_RETURN);
                    $$.treeNode->nodeData.sVal = currentFunction;
                }

            |   TOKEN_RETURN R_EXP TOKEN_SEMI
                {
                    NodeCreate(&($$.treeNode), NODE_RETURN);
                    NodeAddChild($$.treeNode, $2.treeNode);
                    $$.treeNode->nodeData.sVal = currentFunction;
                }
            ;
```

Algorithm 4.10: Rule of the "return" instruction.

The user subroutines section includes additional C/C++ code that supports the actions defined in the rules section. This can include functions for error handling, semantic analysis, or any other utility functions required by the parser.

4.3 Semantic Analysis

The semantic analysis is executed calling the function *executeSemanticAnalysis*, in Alg. 4.11, a function that using the AST, creates all symbol tables, performs the type checking process and report any errors found during the function execution.

```
int executeSemanticAnalisys(TreeNode_st* pTreeRoot, SymbolTable_st**
    ppGlobalTable)
{
    if (!ppGlobalTable || !pTreeRoot)
        return -EINVAL;

    if(!initialized)
    {
        if(createSymbolTable(&pGlobalSymTable, NULL))
        {
            return 1;   /* error creating symbol table */
        }
        initialized = true;
    }

    *ppGlobalTable = pGlobalSymTable;
    pCurrentScope = pGlobalSymTable;
    SymbolTableTraverse(pTreeRoot);
```

```
18
19      if(errorCounter == 0)
20      {
21          TypeCheckTraverse(pTreeRoot);
22      }
23
24      if(errorCounter > 0)
25      {
26          LOG_ERROR(": %d error(s) found during semantic analisys!\n",
        errorCounter);
27          return SEMANTIC_ERROR;
28      }
29
30      LOG_DEBUG("%d error(s) found during semantic analisys!\n",
        errorCounter);
31      return SEMANTIC_OK;
32  }
```

Algorithm 4.11: *executeSemanticAnalysis* function implementation.

4.3.1 Symbol Table

The *SymbolTable* structure serves as the implementation of the symbol table, using a hash table approach with a size determined by the constant *HASH_TABLE_SIZE*, set at 97. Additionally, this structure incorporates a pointer directed towards the symbol table of the enclosing scope.

Each instance of the *SymbolEntry* structure represents a symbol in the symbol table. These symbols contain common information such as type, signal, modifier, and visibility. In the case of a variable symbol, it only holds a memory offset. For an array symbol, it includes both the memory offset and the array size. Lastly, for a function symbol, it stores whether the function is implemented and a the list of parameters, that can have a maximum of 256 parameters.

The *parameter* structure is used to represent the parameters of a function, storing information about each parameter, such as the name, type, sign, modifier, and whether it is a pointer or not.

Each of these sructures are presented below in Alg. 4.12.

```
1  typedef struct parameter {
2      char* name;
3      VarType_et varType;
4      SignQualifier_et varSign;
5      ModQualifier_et varMod;
6      bool isPointer;
7  } parameter_st;
8
9  typedef struct SymbolEntry
10 {
11     SymbolType_et symbolType
12     struct SymbolEntry* next;
13     char* name;
14     VarType_et type;
15     SignQualifier_et signal;
```

```
16      ModQualifier_et modifier;
17      VisQualifier_et visibility;
18
19      union {
20          int memoryLocation;
21          struct {
22              uint8_t parameterNumber;
23              parameter_st* parameters;
24              bool isImplemented;
25          } SymbolFunction_s;
26          struct {
27              int memoryLocation;
28              uint32_t arraySize;
29          } SymbolArray_s;
30      } symbolContent_u;
31  } SymbolEntry_st;
32
33  typedef struct SymbolTable
34  {
35      struct SymbolEntry* table[HASH_TABLE_SIZE];
36      struct SymbolTable* enclosingScope;
37  } SymbolTable_st;
```

Algorithm 4.12: Symbol table structures implementation.

Hash

The hash function, which can be seen below in Alg. 4.13, given a key, iterates through each character of the key using a loop until it found the null terminator. The implemented algorithm uses a method of shifting, addition, and finally a module operation to ensure that the result is within the limits.

```
1  static int hash(const char *key)
2  {
3      int hashValue = 0;
4      int i = 0;
5      while (key[i] != '\0')
6          hashValue = ((hashValue << 4) + key[i++]) % HASH_TABLE_SIZE;
7
8      return hashValue;
9  }
```

Algorithm 4.13: Hash function implementation.

Fetch Symbol

The *fetchSymbol* function, in Alg. 4.14, first calculates the hash index for the provided symbol name to determine its location in the symbol table. Then, it goes through the linked list at the calculated hash index, iterating through the elements comparing the name of each symbol in the list with the provided symbol name. If a match is found, the pointer is updated with the symbol found and returns an indicator that the symbol has been found.

If the symbol is not found in the current scope, if there is an enclosing scope available, and if the search is not limited to the current scope, the function recursively calls itself to search in the enclosing scope. If the symbol is not found in any of the scopes, the function returns an indicator that the symbol was not found.

```
1  int fetchSymbol(SymbolTable_st* pSymTable, SymbolEntry_st** ppSymbol,
       char* name, int onlyCurrentScope)
2  {
3      if (!pSymTable || !ppSymbol)
4          return -EINVAL;
5
6      int index = hash(name);
7
8      /* traverse the linked list at the hash index */
9      SymbolEntry_st* symIterator = pSymTable->table[index];
10
11     while (symIterator != NULL)
12     {
13         if (symIterator->name != NULL && strcmp(symIterator->name,
       name) == 0)
14         {
15             (*ppSymbol) = symIterator;
16             return SYMBOL_FOUND;
17         }
18         symIterator = symIterator->next;
19     }
20
21     /* check enclosing scope (if it exists) */
22     if (pSymTable->enclosingScope != NULL && !onlyCurrentScope)
23     {
24         return fetchSymbol(pSymTable->enclosingScope, ppSymbol, name,
       onlyCurrentScope);
25     }
26     /* symbol not found */
27     return SYMBOL_NOT_FOUND;
28 }
```

Algorithm 4.14: *fetchSymbol* function implementation.

Insert Symbol

The *insertSymbol* function, in (Alg. 4.15), is responsible for adding a new symbol to the symbol table. First, it calculates the hash index for the symbol name provided. Then the function checks if the symbol already exists in the current scope by calling the *fetchSymbol* function with the *onlyCurrentScope* parameter set. If the symbol already exists in the current scope the function returns an error code *(SYMBOL_ERROR)*. However, if the symbol does not exist, the function allocates memory for a new symbol entry and the new symbol's name and type are set according to the arguments.

Lastly the new symbol is added to the symbol table by updating the table's entry at the calculated hash index, and the **ppSymEntry* points to the new symbol and returns *SYMBOL_ADDED* to indicate that the symbol was added with success.

```
1  int insertSymbol(SymbolTable_st* pSymTable, SymbolEntry_st**
       ppSymEntry, char *symName, SymbolType_et symType)
2  {
3      if (!pSymTable || !ppSymEntry)
4          return -EINVAL;
5
6      int index = hash(symName);
7      SymbolEntry_st* pEntryAux;
8
9      /* checks if symbol exists in the curr. scope, adds it if not */
10     if( fetchSymbol(pSymTable, &pEntryAux, symName, true) ==
       SYMBOL_NOT_FOUND)
11     {
12         SymbolEntry_st* pNewSymbol;
13         pNewSymbol=(SymbolEntry_st*) calloc(1,sizeof(SymbolEntry_st));
14
15         if (!pNewSymbol)
16         {
17             LOG_DEBUG("Failed to allocate memory for new symbol!\n");
18             return -ENOMEM;
19         }
20
21         pNewSymbol->name = symName;
22         pNewSymbol->symbolType = symType;
23         pEntryAux = pSymTable->table[index];
24         pSymTable->table[index] = pNewSymbol;
25         pNewSymbol->next = pEntryAux;
26         *ppSymEntry = pNewSymbol;
27
28         return SYMBOL_ADDED;
29     }
30     *ppSymEntry = pEntryAux;
31
32     return SYMBOL_ERROR;
33 }
```

Algorithm 4.15: *insertSymbol* function implementation.

4.3.2 Symbol Table Traversal

The Alg. 4.16 details the implementation of the core function responsible for symbol table construction. The following code is the implementation of the main function for the Symbol Table creation.

```
static void SymbolTableTraverse(TreeNode_st* pSintaxTree)
{
    traverse(pSintaxTree,buildSymbolTables,nullProc);
}
```

Algorithm 4.16: *SymbolTableTraverse* function implementation.

The *traverse* function, in Alg. 4.17, exhibits versatility, offering the capability to perform both pre-order and post-order traversals of the provided AST based on the supplied parameters.

```
static void traverse (TreeNode_st* pNode, void (*preOrder) (
    TreeNode_st* ), void (*postOrder) (TreeNode_st* ))
{
    if (pNode != NULL)
    {
        preOrder(pNode);

        for ( int i = 0 ; i < pNode->childNumber ; i++ )
        {
            traverse (&pNode->pChilds[i], preOrder, postOrder);
        }

        postOrder(pNode);
        traverse (pNode->pSibling,preOrder,postOrder);
    }
}
```

Algorithm 4.17: *Traverse* function implementation.

4.3.3 Symbol Processing

The primary function tasked with managing the symbol table is *buildSymbolTables*, depicted in Alg. 4.18. This function encompasses operations for both inserting new symbol table entries and performing verifications based on the encountered node type within the AST.

```
static void buildSymbolTables(TreeNode_st* pNode)
{
    /*
    Flag to signal that you have entered a new scope (when '{') but
    not created the table
    Tables are only created if there are new declarations in the scope
    */
    static bool tablePending = false;

```

```
9       /*
10      Flag to signal that it is a function. In functions, the tables are
         created when the signature appears to pass the parameters
11      */
12      static bool tableFunction = false;
13      SymbolEntry_st* pNewSymbol;
14
15      switch (pNode->nodeType)
16      {
17          case NODE_VAR_DECLARATION:
18              ...
19          case NODE_ARRAY_DECLARATION:
20              ...
21          case NODE_FUNCTION:
22              ...
23          case NODE_FUNCTION_CALL:
24              ...
25          case NODE_START_SCOPE:
26              ...
27          case NODE_END_SCOPE:
28              ...
29          ...
30          default:
31              break;
32      }
33  }
```

Algorithm 4.18: *buildSymbolTables* function implementation.

Variable Declaration

The Alg. 4.19 presents how variable declaration node handling is implemented. It starts by creating a new symbol table if needed, determining the variable type (pointer or regular), validating that the type is not void, inserting the variable symbol into the symbol table (implicitly verifies if the symbol exists), and setting attributes for the symbol like memory location and visibility.

```
case NODE_VAR_DECLARATION:
    /* create a new symbol is tablePending is set */
    if(tablePending)
    {
        SymbolTable_st* ppsymTable;
        createSymbolTable(&ppsymTable, pCurrentScope);
        pCurrentScope = ppsymTable;
        tablePending = false;
    }

    TreeNode_st* pNodeChild = pNode->pChilds;
    SymbolType_et symType;

    if(pNodeChild->nodeType == NODE_POINTER)
    {
        pNodeChild = pNodeChild->pChilds;
        symType = SYMBOL_POINTER;
    }
    else
    {
        symType = SYMBOL_VARIABLE;
    }

    if(pNodeChild->nodeData.dVal == TYPE_VOID)
    {
        semanticError(pNode, "Can not declare a 'void' variable.");
    }

    /* inserts the new symbol into the table if it doesn't exist */
    if( insertSymbol(pCurrentScope, &pNewSymbol, pNode->nodeData.sVal,
     symType) == SYMBOL_ADDED)
    {
        setMemoryLocation(&pNewSymbol->symbolContent_u.memoryLocation,
     pNodeChild->nodeData.dVal, 1);
        setVariblesType(pNodeChild,
                        &pNewSymbol->type,
                        &pNewSymbol->signal,
                        &pNewSymbol->modifier,
                        &pNewSymbol->visibility);
        pNode->pSymbol = pNewSymbol;
    }

```

```
41      else
42      {
43          semanticError(pNode, "Symbol Redefinition!");
44      }
45      break;
46  ...
```

Algorithm 4.19: Variable declaration node handling implementation.

Variable Reference

The code presented in Alg. 4.20 is how variable reference handling is implemented. In this case is just a simple verification if the symbol exists.

```
1   ...
2   case NODE_IDENTIFIER:
3       /* checks if the symbol already exists */
4       if(fetchSymbol(pCurrentScope, &pNewSymbol, pNode->nodeData.sVal,
        false) == SYMBOL_NOT_FOUND)
5       {
6           semanticError(pNode, "Symbol Not Defined!");
7       }
8       else
9       {
10          pNode->pSymbol = pNewSymbol;
11      }
12      break;
13  ...
```

Algorithm 4.20: Variable reference node handling implementation.

Function Call

In Alg. 4.21 the code presents how function call handling is implemented. It starts by checking if the function symbol is present in the current scope and comparing the number of arguments passed to the expected number of parameters. It generates an error message if there's a mismatch or if the function is not defined. If valid, it associates the function call node with the function symbol.

```
1   ...
2   case NODE_FUNCTION_CALL:
3
4       if(fetchSymbol(pCurrentScope, &pNewSymbol, pNode->nodeData.sVal,
        false) == SYMBOL_FOUND)
5       {
6           TreeNode_st* pNodeParameters  = &pNode->pChilds[0];
7
8           int functionParameterNumber = pNewSymbol->symbolContent_u.
        SymbolFunction_s.parameterNumber;
9
10          int parameterCount = 0;
11
```

```
12          /* checks if the arguments number is correct */
13          while(pNodeParameters != NULL)
14          {
15              parameterCount++;
16              pNodeParameters = pNodeParameters->pSibling;
17          }
18
19          if(parameterCount != functionParameterNumber)
20          {
21              char errorMessage[100];
22              snprintf(errorMessage, sizeof(errorMessage), "%s arguments
                    for function call, expected %d, have %d \n",
                    (parameterCount > functionParameterNumber ? "Too many"
                    :"Too few"), functionParameterNumber, parameterCount);
23              semanticError(pNode, errorMessage);
24          }
25
26          pNode->pSymbol = pNewSymbol;
27      }
28      else
29      {
30          semanticError(pNode, "Function Not Defined!");
31      }
32      break;
33  ...
```

Algorithm 4.21: Function call node handling implementation.

Scope Handling

The code in Alg. 4.22 shows how scope handling is implemented.

```
1  ...
2  case NODE_START_SCOPE:
3      if(!tableFunction)
4          tablePending = true;
5      else
6          tableFunction = false;
7      break;
8
9  case NODE_END_SCOPE:
10     if(!tablePending)
11         pCurrentScope = pCurrentScope->enclosingScope;
12     else
13         tablePending = false;
14     break;
15 ...
```

Algorithm 4.22: Scope nodes handling implementation.

The following functions were created to simplify some processes:

- **setVaribles Type:** goes through all the siblings in the preamble and saves qualifiers (Alg. 4.23);
- **setMemoryLocation:** sets a symbol memory offset accordingly variable type (Alg. 4.24);

```
1  static void setVariblesType(TreeNode_st* pNode, VarType_et* type,
       SignQualifier_et* sign, ModQualifier_et* modifier, VisQualifier_et
       * visibility)
2  {
3      TreeNode_st* pNodeTemp = pNode;
4      *sign = 0;
5      *modifier = 0;
6      *visibility = 0;
7
8      while (pNodeTemp != NULL)
9      {
10         switch(pNodeTemp->nodeType)
11         {
12             case NODE_TYPE:
13                 *type = pNodeTemp->nodeData.dVal;
14                 break;
15
16             case NODE_SIGN:
17                 *sign = pNodeTemp->nodeData.dVal;
18                 break;
19
20             case NODE_MODIFIER:
21                 *modifier = pNodeTemp->nodeData.dVal;
22                 break;
23
24             case NODE_VISIBILITY:
25                 *visibility = pNodeTemp->nodeData.dVal;
26                 break;
27
28             default:
29                 LOG_DEBUG("Invalid node");
30                 break;
31         }
32         pNodeTemp = pNodeTemp->pSibling;
33     }
34 }
```

Algorithm 4.23: *setVariblesType* function implementation.

```
static int setMemoryLocation(int* varLocation, VarType_et varType, int
     multiplier)
{
    static int currentLocation = 0;
    *varLocation = currentLocation;

    switch(varType)
    {
        case TYPE_CHAR:
            currentLocation += 1*multiplier;
            break;

        case TYPE_SHORT:
            currentLocation += 2*multiplier;
            break;

        case TYPE_LONG:
        case TYPE_INT:
        case TYPE_FLOAT:
            currentLocation += 4*multiplier;
            break;

        case TYPE_DOUBLE:
            currentLocation += 8*multiplier;
            break;

        case TYPE_LONG_DOUBLE:
            currentLocation += 16*multiplier;
            break;

        default:
            LOG_DEBUG("Invalid variable type");
            break;
    }
}
```

Algorithm 4.24: *setMemoryLocation* function implementation.

4.3.4 Type Checking

As previously designed, the type checking will make use of 2 methods, a method to verify each AST node, and a method to be called in the case of a *NODE_OPERATOR*.

Since it's a post order traversal, it becomes important to set the variable *nodeVarType* of each AST node to its child or children type, so there is no need to perform the same verification more than one time.

Below, in Alg. 4.25, is displayed how the type checking is done when a *NODE_OPERATOR* or a *NODE_REFERENCE* is found in the *checkNode* function.

```
static void checkNode(TreeNode_st * pNode)
{
```

```
3       TreeNode_st* pChild1;
4       TreeNode_st* pChild2;
5       switch(pNode->nodeType)
6       {
7           case NODE_OPERATOR:
8               checkOperator(pNode);
9               break;
10          ...
11          case NODE_REFERENCE:    /* ex: &variable */
12              TreeNode_st* pNodeAux;
13
14              pNodeAux = (pNode->childNumber != 0) ? &pNode->pChilds[0]
                                                     : pNode;
15              if(pNodeAux->pSymbol != NULL)
16              {
17                  if(pNodeAux->pSymbol->symbolType == SYMBOL_ARRAY ||
                       pNodeAux->pSymbol->symbolType == SYMBOL_VARIABLE)
18                  {
19                      pNode->nodeVarType = pNode->pSymbol->type;
20                  }
21                  else
22                  {
23                      semanticError(pNode, "Invalid use of '&'!\n");
24                      pNode->nodeVarType = TYPE_VOID;
25                  }
26              }
27              break;
28          ...
29      }
30  }
```

Algorithm 4.25: NODE_OPERATOR and NODE_REFERENCE handling by *checkNode* function implementation.

Inside the *checkNode*, it's not the current node type that is verified, but its child or children nodes, as can be analyzed below in Alg. 4.26, where is depicted the handling of a division operation.

```
1   static int checkOperator(TreeNode_st * pNode)
2   {
3       SymbolEntry_st* pEntryChild1;
4       SymbolEntry_st* pEntryChild2;
5       SymbolEntry_st* pEntryAux;
6       TreeNode_st* pChild1;
7       TreeNode_st* pChild2;
8
9       int varType1;
10      int varType2;
11
12      switch(pNode->nodeData.dVal)
13      {
```

```
14          ...
15          case OP_DIVIDE:
16              pChild1 = &pNode->pChilds[0];
17              pChild2 = &pNode->pChilds[1];
18
19              /* can't divide by zero */
20              if(((pChild2->nodeType == NODE_INTEGER) &&
                  (pChild2->nodeData.dVal == 0)) || ((pChild2->nodeType ==
                   NODE_FLOAT) && (pChild2->nodeData.fVal == 0.0)))
21              {
22                  semanticError(pNode, "Division by zero is not
                                  supported\n");
23                  return SEMANTIC_ERROR;
24              }
25              else
26              {
27                  varType1 = pChild1->nodeVarType;
28                  varType2 = pChild2->nodeVarType;
29
30                  if((varType1==TYPE_STRING || varType1==TYPE_VOID ) ||
                      (varType2 == TYPE_STRING || varType2 == TYPE_VOID ))
31                  {
32                      semanticError(pNode, "Invalid operand type!\n");
33                      return SEMANTIC_ERROR;
34                  }
35                  else if (varType1 != varType2)
36                  {
37                      semanticError(pNode,"Operand types don't match!");
38                      return SEMANTIC_ERROR;
39                  }
40                  else
41                  {
42                      pNode->nodeVarType = varType1;
43                  }
44              }
45              break;
46          ...
47      }
48  }
```

Algorithm 4.26: OP_DIVIDE handling by *checkOperator* function implementation.

4.3.5 Type Checking Traversal

The post order traversal of the AST to perform type checking is done calling the function in Alg. 4.27, which also calls the function depicted in Alg. 4.17.

```
1  static void TypeCheckTraverse(TreeNode_st* pSintaxTree)
2  {
```

```
    traverse(pSintaxTree,nullProc,checkNode);
}
```

Algorithm 4.27: *TypeCheckTraverse* function implementation.

4.4 Optimization

4.4.1 Constant Folding

The *constFolding* function (Alg. 4.28) is responsible for applying constant folding optimization to an AST. The function receives a pointer to the tree node, and if the type of the node is an operator, the function checks the specific type of operator through a switch statement.

In the *OP_BITWISE_NOT* case, if the type of the first child is an integer, the function calls opIntType to optimize the operation. In the case of arithmetic operations (plus, minus, multiply, divide, remain), if both children are integers, the function calls opIntType and if both children are of type float, the function calls opFloatType.

For the remaining operations (r_shift, l_shift, and, or, xor) it is checked whether both children are of the integer type and the opIntType function is called.

If the operator does not match any of the specified cases, the function takes no action.

```
static void constFolding(TreeNode_st* pTreeRoot)
{
    TreeNode_st* pChild1;
    TreeNode_st* pChild2;
    pChild1 = &pTreeRoot->pChilds[0];
    pChild2 = &pTreeRoot->pChilds[1];

    if(pTreeRoot->nodeType == NODE_OPERATOR) {
        switch(pTreeRoot->nodeData.dVal)
        {
            case OP_BITWISE_NOT:
                if(pChild1->nodeType == NODE_INTEGER)
                {
                    opIntType(pTreeRoot);
                }
            break;

            case OP_PLUS:
            case OP_MINUS:
            case OP_MULTIPLY:
            case OP_DIVIDE:
            case OP_REMAIN:

                if(pChild1->nodeType == NODE_INTEGER &&
                   pChild2->nodeType == NODE_INTEGER)
                {
                    opIntType(pTreeRoot);
                }
```

```
28                  else if(pChild1->nodeType == NODE_FLOAT &&
                            pChild2->nodeType == NODE_FLOAT)
29                  {
30                      opFloatType(pTreeRoot);
31                  }
32              break;
33
34              case OP_RIGHT_SHIFT:
35              case OP_LEFT_SHIFT:
36              case OP_BITWISE_AND:
37              case OP_BITWISE_OR:
38              case OP_BITWISE_XOR:
39
40                  if(pChild1->nodeType == NODE_INTEGER &&
                       pChild2->nodeType == NODE_INTEGER)
41                  {
42                      opIntType(pTreeRoot);
43                  }
44              break;
45
46              default:
47                  break;
48          }
49      }
50  }
```

Algorithm 4.28: *constFolding* function implementation.

The *opIntType* or *opFloatType* function, when called, performs the optimization by replacing the operation node with the resulting value, freeing the memory of the child nodes and setting the number of children to zero. The Alg. 4.29 presents the implementation of the *opFloatType* function.

```
1  static void opFloatType(TreeNode_st* pTreeRoot)
2  {
3      pTreeRoot->nodeType = NODE_FLOAT;
4      pTreeRoot->nodeData.fVal = operationFloat(pTreeRoot->pChilds[0].
       nodeData.fVal, pTreeRoot->pChilds[1].nodeData.fVal, pTreeRoot->
       nodeData.dVal);
5      free(pTreeRoot->pChilds);
6      pTreeRoot->childNumber = 0;
7  }
```

Algorithm 4.29: *opFloatType* function implementation.

The *operationFloat* (Alg. 4.30) function performs arithmetic operations between two numbers based on the operation. Depending on the value of op, it returns the result of the corresponding operation, taking into account that in the case of division the function checks the divisor to avoid division by zero.

```
1  static double operationFloat(double a, double b, long op)
2  {
```

```
    switch(op) {
        case OP_PLUS:
            return a + b;
        case OP_MINUS:
            return a - b;
        case OP_MULTIPLY:
            return a * b;
        case OP_DIVIDE:
            if (b != 0)
            {
                return a / b;
            } else
            {
                LOG_ERROR_SHORT("Cant divide by 0\n");
                return 0.0;
            }
        default:
            LOG_ERROR_SHORT("Invalid Operation\n");
            return 0.0;
    }
}
```

Algorithm 4.30: *operationFloat* function implementation.

4.4.2 Instruction Scheduling

The implementation of the assembly code optimizer was straightforward, involving a clear and systematic approach. This process not only simplified the development but also ensured that the final design was accurate and functionally correct. The optimizer effectively enhanced the assembly code, making sure that the resulting output adhered to the intended specifications and performance criteria. Through meticulous planning and execution, the optimizer was able to achieve its goals without introducing errors or compromising the integrity of the code.

The initial phase to be highlighted is the parsing process, during which the original source assembly code is meticulously analyzed and divided into distinct code blocks. This step is crucial as it lays the foundation for the entire optimization process. During the parsing process, the source code is read line by line, and each line is interpreted. The code is then segmented into manageable code blocks, where each block represents a logical grouping of instructions that can be independently analyzed and optimized. This segmentation is based on the flow of control in the program, as designed on Figure 3.29.

```
    int codeBlockDivision(const char *filename, codeBlock_st **
    ppCodeBlock_st, int *totalBlocks){
    ...
    while (fgets(line, sizeof(line), file)){

        pInstruction_st = realloc(pCurrentBlock->Instructions, (
    lineCount + 1) * sizeof(Instruction_st));
        if(pCurrentBlock->Instructions[lineCount].type ==
    TYPE_DIRECTIVE || pCurrentBlock->Instructions[lineCount].type ==
    TYPE_END_BLOCK){
            (*totalBlocks)++;
```

```
8               *ppCodeBlock_st = realloc(*ppCodeBlock_st, (*totalBlocks)
    * sizeof(codeBlock_st));
9               pCurrentBlock = &(*ppCodeBlock_st)[*totalBlocks-1];
10              pCurrentBlock->Instructions = malloc(sizeof(Instruction_st
    ));
11              lineCount = 0;
12              pCurrentBlock->Instructions[lineCount].lineNum = lineCount
    ;
13              parseInstruction(line, &pCurrentBlock->Instructions[
    lineCount]);
14              pCurrentBlock->size = 1;
15              lineCount++;
16          }
17          else if(pCurrentBlock->Instructions[lineCount].type ==
    TYPE_LAST){
18              lineCount++;
19              pCurrentBlock->size = lineCount;
20              (*totalBlocks)++;
21              *ppCodeBlock_st = realloc(*ppCodeBlock_st, (*totalBlocks)
    * sizeof(codeBlock_st));
22              pCurrentBlock = &(*ppCodeBlock_st)[*totalBlocks-1];
23              pCurrentBlock->Instructions = NULL;
24              pCurrentBlock->size = 0;
25              lineCount = 0;
26          }
27          else{
28              lineCount++;
29              pCurrentBlock->size = lineCount;
30          }
31      }
32  }
33  ...
```

Algorithm 4.31: *codeBlockDivision* function implementation.

Within the parsing process it is to be highlighted the instruction parsing, during which the original source assembly code is parsed into distinct instructions. During this step, the instruction attributes of the are also assigned, setting the groundwork for subsequent stages of optimization.

```
1  void parseInstruction(const char *line, Instruction_st *inst){
2
3      if(sscanf(line, "%3s R%d, #%d", mnemonic,
4      &inst->operands[0], &inst->operands[1]) == 3){
5          inst->type = TYPE_IMMEDIATE;
6          inst->mnemonicValue = getMnemonicValue(mnemonic);
7          inst->operands[1] = NO_OPERAND;
8          inst->operands[2] = NO_OPERAND;
9
10     }
11
12     else if(sscanf(line, "%3s R%d, R%d, #%d", mnemonic,
```

```
    &inst->operands[0], &inst->operands[1], &inst->operands[2]) == 4){
        inst->type = TYPE_IMMEDIATE;
        inst->mnemonicValue = getMnemonicValue(mnemonic);
        inst->operands[2] = NO_OPERAND;
        if(inst->mnemonicValue == OP_JMP)
            inst->type = TYPE_END_BLOCK;
    }
    else{
        inst->type = TYPE_SINGLE;
    }
}
```

Algorithm 4.32: *parseInstruction* function implementation.

The Alg. 4.33 includes all the attributes that each instruction can possess. These attributes will be used within the code blocks for code optimization and flow analysis.

```
    typedef struct
{
    InstructionType_et type;
    int mnemonic[4];
    int operands[3];
    int mnemonicValue;
    int lineNum;
    char line[MAX_LINE_LENGTH];
}Instruction_st;
```

Algorithm 4.33: *Instruction_st* struct implementation.

Each code block is then stored according to a specific structure that facilitates both the parsing and optimization processes. This structure is designed to ensure that all relevant information is systematically captured and easily accessible for subsequent stages of code analysis and transformation-

```
typedef struct codeBlock
{
    char * label;
    Instruction_st * Instructions;

    int * stallPositions;
    size_t stallPositionsNum;

    int * replacePositions;
    size_t replacePositionsNum;
    bool branchFound;
    int size;
}codeBlock_st;
```

Algorithm 4.34: *codeBlock_st* struct implementation.

After completing the parsing process, each code block undergoes a thorough analysis to identify potential hazards that could compromise the efficiency of the code. These hazards, which were outlined in earlier sections, may include issues such as data dependencies and control flow anomalies.

```
int checkForStalls(codeBlock_st *pCodeBlock_st){
```

```
pCodeBlock_st->stallPositionsNum = 0;
pCodeBlock_st->stallPositions = malloc(sizeof(int));

if (pCodeBlock_st->stallPositions == NULL) {
    printf("Failed to allocate memory for stall positions");
    return -ENOENT;
}

for(int i = 0; i < pCodeBlock_st->size - 1; i++){
    if (i + 1 < pCodeBlock_st->size &&
    (pCodeBlock_st->Instructions[i].mnemonicValue == OP_LD ||
     pCodeBlock_st->Instructions[i].mnemonicValue == OP_LDX)
     && ( (isAluOperation(pCodeBlock_st->Instructions[i + 1].
    mnemonicValue)
     && (pCodeBlock_st->Instructions[i].operands[0]
     == pCodeBlock_st->Instructions[i + 1].operands[1]
     || pCodeBlock_st->Instructions[i].operands[0]
     == pCodeBlock_st->Instructions[i + 1].operands[2]))
     || ((pCodeBlock_st->Instructions[i+1].mnemonicValue == OP_ST ||
         pCodeBlock_st->Instructions[i+1].mnemonicValue == OP_STX)
         && pCodeBlock_st->Instructions[i].operands[0]
         == pCodeBlock_st->Instructions[i + 1].operands[0]) ) ){
            pCodeBlock_st->stallPositions = pAux;
    }
}
```

Algorithm 4.35: *checkForStalls* function implementation.

After identifying potential stalls in the code, the next step involves locating positions where replacements are needed to ensure smooth execution. This critical task is managed by the *checkForReplacePositions* function, which meticulously analyzes each instruction within the code block to determine the appropriate points for replacement. The function performs this as exemplified in the Alg. 4.36 with some of the ALU operations.

```
...
switch (pCodeBlock_st->Instructions[i].mnemonicValue){
case OP_ADD:
case OP_SUB:
case OP_OR:
case OP_XOR:
case OP_AND:
case OP_RR:
case OP_RL:

if(pCodeBlock_st->Instructions[i].type == TYPE_REGISTERS){
    if(dst_registers[pCodeBlock_st->Instructions[i].operands[0]]
    || dst_registers[pCodeBlock_st->Instructions[i].operands[1]]
    || dst_registers[pCodeBlock_st->Instructions[i].operands[2]]
    || ope_registers[pCodeBlock_st->Instructions[i].operands[0]]
    || ope_registers[pCodeBlock_st->Instructions[i].operands[1]]
    || ope_registers[pCodeBlock_st->Instructions[i].operands[2]]
```

```
18      || pCodeBlock_st->branchFound){
19
20          dst_registers[pCodeBlock_st->Instructions[i].operands[0]] =
        true;
21          ope_registers[pCodeBlock_st->Instructions[i].operands[1]] =
        true;
22          ope_registers[pCodeBlock_st->Instructions[i].operands[2]] =
        true;
23
24      }
25      else{
26          pCodeBlock_st->replacePositions[pCodeBlock_st->
        replacePositionsNum++]
27          = pCodeBlock_st->Instructions[i].lineNum;
28          stallAtended = true;
29      }
30  }
31  ...
```

Algorithm 4.36: *checkForReplacePositions* function implementation.

Finally, after all the replacement positions are discovered, the code optimizer proceeds to the file generation phase. During this phase, the optimizer iterates through each code block, carefully processing the instructions within them. This iterative process ensures that all identified replacements are correctly applied and that any potential stalls are addressed.

As the optimizer navigates through each code block, it incorporates the necessary modifications and optimizations based on the previously identified positions. This includes reordering instructions while inserting optimized replacements. The goal is to refine the code to enhance performance and efficiency.

```
1   int fileGenerator(const char *destinationFilename, codeBlock_st *
        pCodeBlock_st){
2
3   for(int i = 0; i < pCodeBlock_st->size; i++){
4       if(pCodeBlock_st->Instructions[i].line != NULL){
5       // Just place each instruction in order
6           if(pCodeBlock_st->replacePositionsNum == 0){
7               fputs(pCodeBlock_st->Instructions[i].line, destinationFile
        );
8           }
9       // When the position of the stall instruction is found, add it to
        the file
10          else if(i == pCodeBlock_st->stallPositions[index]){
11              fputs(pCodeBlock_st->Instructions[i].line, destinationFile
        );
12              searchingLine = true;
13          }
14      // As soon as a replace instruction is found, add it to the file
15      //, then add the content of the auxBuffer and clear it
16          else if(i == pCodeBlock_st->replacePositions[index]){
17              searchingLine = false;
```

```
18              fputs(pCodeBlock_st->Instructions[i].line, destinationFile
    );
19              if(index < pCodeBlock_st->replacePositionsNum - 1)
20                  index++;
21              fputs(auxBuffer,destinationFile);
22              memset(auxBuffer, '\0', sizeof(auxBuffer));
23          }
24      // While a replace instruction is not found,
25      //add the content to the auxBuffer
26          else if(searchingLine){
27              strncat(auxBuffer, pCodeBlock_st->Instructions[i].line
28              , sizeof(auxBuffer) - 1);
29          }
30      // If there are no more stalls to solver,
31      //the instructions are appended normally
32          else{
33              fputs(pCodeBlock_st->Instructions[i].line, destinationFile
    );
34          }
35      }
36  }
```

Algorithm 4.37: *fileGenerator* function implementation.

4.5 Code Generation

4.5.1 Assembly Code Generation

During the implementation of Code Generation, all AST nodes were analyzed to develop templates for each type of node and their respective children.

Execute Code Generation

This process begins with the *executeCodeGeneration* function (Alg. 4.38), which is responsible initializing the code generation process.

The function starts with the initialization of a list that is used to handle post incrementations (eg: i++). Then it generates the code for the multiplication and division functions. Since VeSPA doesn't have MUL and DIV instructions, this functions will be called to perform any multiplication, division and remainder operations. After that, the code generation is performed, starting at the root node of the AST.

```
1  int executeCodeGeneration(TreeNode_st *pTreeRoot, FILE* pDestStream)
2  {
3      if (!pTreeRoot || !pDestStream)
4          return -EINVAL;
5
6      pAsmFile = pDestStream;
7
8      PostIncListInit();
9
10     generateInitCode();
11
```

```
12      generateMultiplication();
13
14      generateDivision();
15
16      generateCode(pTreeRoot);
17
18      return 0;
19  }
```

Algorithm 4.38: *executeCodeGeneration* function implementation.

Generate Code

The *generateCode* function (Alg. 4.39), is designed to ensure that all assembly code is generated. Accordingly, whenever a node is fully processed, this function checks if the node has a sibling. If a sibling is present, it generates the respective code for that sibling as well.

```
1  static int generateCode(TreeNode_st *pTreeNode)
2  {
3      TreeNode_st *pSib;
4      int ret = parseNode(pTreeNode, NODE_TYPE_NOT_DEFINED,
       OP_NOT_DEFINED, 0);
5      if (ret < 0)
6          return ret;
7
8      pSib = pTreeNode->pSibling;
9      while (pSib)
10     {
11         parseNode(pSib, NODE_TYPE_NOT_DEFINED, OP_NOT_DEFINED, 0);
12         pSib = pSib->pSibling;
13     }
14
15     return 0;
16 }
```

Algorithm 4.39: *generateCode* function implementation.

Parse Node

At the beginning of the *parseNode* function (Alg. 4.40), registers were allocated for use in certain types of nodes. This function is also responsible for generating the code for each node, making it possible to verify in the algorithm that a switch is made, depending on the respective node type.

```
1  static int parseNode(TreeNode_st *pCurrentNode, NodeType_et
       parentNodeType, OperatorType_et parentOperatorType, bool
       isLeftInherited)
2  {
3      reg_et lReg;
4      reg_et rReg;
5      reg_et dReg;
6
7      OperatorType_et CurrentNodeOpType = (OperatorType_et) pCurrentNode
       ->nodeData.dVal;
```

```
8
9       if (!IS_TERMINAL_NODE(pCurrentNode->nodeType) && NODE_TYPE(
        pCurrentNode) == NODE_OPERATOR)
10      {
11          //If we enter a non terminal node we allocate a new dReg for
        that operation
12          dReg = getNextAvailableReg();
13          lReg = REG_NONE;
14          rReg = REG_NONE;
15      }
16
17      if(IS_TERMINAL_NODE(pCurrentNode->nodeType) && IS_CONDITIONAL_NODE
        (parentNodeType))
18          dReg = getNextAvailableReg();
19
20      if(parentNodeType == NODE_ARRAY_INDEX)
21          arrayIndexRegSave = dReg;
22
23      if (!pCurrentNode)
24          return -EINVAL;
25
26      switch (pCurrentNode->nodeType)
27      {
28          case NODE_OPERATOR:
29          ...
30          case NODE_IDENTIFIER:
31          ...
32          case NODE_INTEGER:
33          ...
34          case NODE_IF:
35          ...
36          NODE_WHILE:
37          ...;
38      }
39      return 0;
40  }
```

Algorithm 4.40: *parseNode* function implementation.

Emit Alu Instructions

The function presented in Alg. 4.41 is responsible for generating the necessary assembly code for all ALU instructions, based on the provided values. This function checks whether it receives two registers, one register and an immediate value, or, in the case of unary instructions like NOT, either a single register or a single immediate value.

```
1  static int emitAluInstruction(asm_instr_et instructionType,
2                                uint8_t isImed,
3                                uint32_t imedValue,
4                                reg_et resultReg,
5                                reg_et leftOperand,
```

```
6                                   reg_et rightOperand)
7  {
8
9      //NOT instruction can't use immediate as a parameter
10     if ((instructionType == INST_NOT) && (isImed))
11         return -EPERM;
12
13     if ((isImed) && (rightOperand != REG_NONE))
14         return -EPERM;
15
16     if (isImed)
17     {
18         if (instructionType == INST_CMP)
19         {
20             fprintf(pAsmFile, "\t%s %s,#%d\n",
21                     INSTRUCTION(instructionType),
22                     REGISTER(leftOperand),
23                     imedValue);
24         }
25         else
26         {
27             fprintf(pAsmFile, "\t%s %s,%s,#%d\n",
28                     INSTRUCTION(instructionType),
29                     REGISTER(resultReg),
30                     REGISTER(leftOperand),
31                     imedValue);
32         }
33     }
34     else
35     {
36         if (instructionType == INST_CMP)
37         {
38             fprintf(pAsmFile, "\t%s %s,%s\n",
39                     INSTRUCTION(instructionType),
40                     REGISTER(leftOperand),
41                     REGISTER(rightOperand));
42         }
43         else
44         {
45             if (rightOperand == REG_NONE)
46             {
47                 fprintf(pAsmFile, "\t%s %s,%s\n",
48                         INSTRUCTION(instructionType),
49                         REGISTER(resultReg),
50                         REGISTER(leftOperand));
51             }
52             else
53             {
54                 fprintf(pAsmFile, "\t%s %s,%s,%s\n",
```

```
                        INSTRUCTION(instructionType),
                        REGISTER(resultReg),
                        REGISTER(leftOperand),
                        REGISTER(rightOperand));
            }
        }
    }

    return 0;
}
```

Algorithm 4.41: *emitAluInstructions* function implementation.

Emit Label Instructions

The code generation for labels was implemented as shown in Alg. 4.43. This function aims to generate all necessary labels, particularly for loops and functions. To ensure that these labels are not repeated and do not corrupt the generated code, a counter (labelCounters) was created. This counter is responsible for differentiating the labels used by assigning them unique numbers.

```
static uint32_t labelCounters[LABEL_MAX] = {0};
static int emitLabelInstruction(label_et labelType, uint32_t count,
    char* nameLabel)
{
    if (labelType < WHILE_START || labelType >= LABEL_MAX)
        return -EINVAL;
    if(labelType != FUNCTION_NAME)
        fprintf(pAsmFile, "%s%u:\n",
                LABEL(labelType),
                count);
    else
        fprintf(pAsmFile, "\n\n\n%s%s:\n",
                LABEL(labelType),
                nameLabel);

    return 0;
}

static uint32_t getPostIncLabelCounter(label_et labeltype)
{
    if (labeltype < WHILE_START || labeltype >= LABEL_MAX)
        return -EINVAL;

    return labelCounters[labeltype]++;
}

static uint32_t getLabelCounter(label_et labelType)
{
    if (labelType < WHILE_START || labelType >= LABEL_MAX)
        return -EINVAL;

```

```
31      return labelCounters[labelType];
32  }
```

Algorithm 4.42: *emitLabelInstructions* and label counters functions implementation.

Emit Branches and Jumps Instructions

For code generation in the case of jumps, the following functions were implemented as shown in Alg. 4.43. The branch function receives the branch instruction to be used, the label to jump to, and its respective counter as parameters. In the case of unconditional jumps, it only checks the type of jump received, the register, and the immediate value to generate the necessary assembly code.

```
1   static int emitBranchInstruction(asm_instr_et instructionType,
        label_et labelType, uint32_t counter)
2   {
3      //CheckLabel type
4       if (labelType < WHILE_START || labelType >= LABEL_MAX)
5           return -EINVAL;
6
7       if (instructionType < INST_BGT || instructionType > INST_BMI)
8           return -EINVAL;
9
10      if(counter < 0)
11          return -EINVAL;
12
13      fprintf(pAsmFile, "\t%s %s%u\n",
14              INSTRUCTION(instructionType),
15              LABEL(labelType),
16              counter);
17
18      return 0;
19  }
20
21  static int emitJumpInstruction(asm_instr_et instructionType, reg_et
        dReg, reg_et lReg, uint32_t dVal)
22  {
23      if (dReg >= REG_NONE)
24          return -EINVAL;
25
26      if(instructionType == INST_JMP)
27          fprintf(pAsmFile, "\t%s %s,#%u\n",
28              INSTRUCTION(instructionType),
29              REGISTER(dReg),
30              dVal
31              );
32
33      else if (instructionType == INST_JMPL)
34          fprintf(pAsmFile, "\t%s %s,%s,#%u\n",
35              INSTRUCTION(instructionType),
36              REGISTER(dReg),
37              REGISTER(lReg),
```

```
38              dVal
39              );
40      else
41          return -EINVAL;
42
43      return 0;
44
45  }
```

Algorithm 4.43: *emitBranchInstructions* and *emitJumpInstructions* functions implementation.

4.5.2 Assembler

This section covers the assembler implementation for the VESPA architecture. The main objective was to develop an assembler that translates the Assembly code generated by the compiler into specific machine code to be executed on an FPGA with the VESPA architecture. To achieve this, several essential functions that will be presented below were implemented.

Add Symbol

The *add_symbol* function presented in Alg. 4.44 is responsible for adding a symbol to the symbol table. It starts by checking whether the symbol table is initialized, returning an error otherwise. It then calculates the hash value for the symbol name to perform a hash table search. If an index corresponding to the hash value is found in the table, the function checks whether the symbol already exists in the list associated with that index. If the symbol is already in the table, returns the index corresponding to it. Otherwise, it inserts the symbol into the table and returns the index of the inserted symbol.

```
1  uint32_t add_symbol(char *name)
2  {
3      uint32_t hash_value;
4      uint32_t index;
5
6      if(!ready){
7          LOG_DEBUG("Symbol table uninitialized\n");
8          exit(EXIT_FAILURE);
9      }
10     else{
11         // Calculate the hash value and the index for the symbol name
12         hash_value = hash(name);
13         index = hash_table[hash_value];
14
15         while((index != NULL_PTR)){
16             // If the symbol with the same name already exists, return
       its index
17             if(!strcmp(sym_table[index].name, name))
18                 return index;
19
20             else
21                 //Move to the next symbol
22                 index = sym_table[index].link;
23         }
```

```
24          // If the symbol doesn't exist, insert it into the symbol
        table and return its index
25          index = insert_symbol(name, hash_value);
26
27          return index;
28      }
29  }
```

Algorithm 4.44: *add_symbol* function implementation.

Add Statement

The *add_statement* function is used to add a new statement to the list of statements. It starts by checking whether the allocated space is sufficient and does not reallocate the necessary memory.

Subsequently, the function stores the values of the parameters: *op_code*, *op1*, *op2*, *op3* and *misc* in the corresponding fields of the instruction structure and the index of the current statement is incremented.

Finally, the function checks whether the operation is *DOT_ORG_OP*, where the location counter is changed to the value of the *op1* field, otherwise the location counter is increased with the size of the instruction.

```
1   uint8_t add_statement(uint32_t op_code, uint32_t op1, uint32_t op2,
        uint32_t op3, uint32_t misc)
2   {
3       if(curr_stmt >= stmt_list_size){
4           stmt_list_size += INC_STMT_LIST;
5           stmt_list = (statement_t*) realloc(stmt_list, sizeof(
        statement_t) * stmt_list_size);
6
7           if(stmt_list == NULL){
8               LOG_DEBUG("Error reallocating memory\n");
9               exit(EXIT_FAILURE);
10          }
11      }
12      stmt_list[curr_stmt].op_code    = op_code;
13      stmt_list[curr_stmt].op1        = op1;
14      stmt_list[curr_stmt].op2        = op2;
15      stmt_list[curr_stmt].op3        = op3;
16      stmt_list[curr_stmt].misc       = misc;
17      stmt_list[curr_stmt].line_num   = curr_stmt + 1;
18
19      curr_stmt++;
20
21      if(op_code == DOT_ORG_OP){
22          location_counter = op1;
23      }
24      else{
25          location_counter += WORD_SIZE;
26      }
27  }
```

Algorithm 4.45: *add_statement* function implementation.

Generate Code

The *generate_code* function is the final step of the assembler, responsible for converting assembly instructions into binary code. Throughout the function, each instruction is processed using a switch-case that encodes different types of instructions according to their opcodes. The code variable is assembled bit by bit, and lc is updated to reflect the memory location of the next instruction. The function performs checks to ensure that values are within expected limits and that symbols have been correctly initialized. In case of errors, the process stops and an error message is displayed.

```
void generate_code()
{
    uint32_t code = 0;
    statement_t current_statement;
    uint32_t lc = 0;
    uint8_t inst_size = 0;
    uint8_t error = 0;
    int32_t bxx_value = 0;

    get_output_file(&fp_hex);

    for (uint32_t i = 0; i < get_current_stmt_index(); i++)
    {
        inst_size = 4;
        code = 0;
        current_statement = get_statement(i);

        switch (current_statement.op_code){
            case NOP_OPCODE:
            case HLT_OPCODE:
            case RETI_OPCODE:
                code |= (0x1f & current_statement.op_code)  << 27;
                break;

            case ADD_OPCODE:
            case SUB_OPCODE:
            case OR_OPCODE:
            case AND_OPCODE:
            case XOR_OPCODE:
            case RR_OPCODE:
            case RL_OPCODE:
                code |= (0x1f & current_statement.op_code)  << 27;
                code |= (0x1f & current_statement.op1)      << 22;
                code |= (0x1f & current_statement.op2)      << 17;

                if(current_statement.misc == IMMEDIATE)
                {
                    error |= check_immed(current_statement.op3,
    IMMED16, i);
                    code |= 1 << 16;
                    code |= (0xffff & current_statement.op3);
```

```
41                  }
42                  else
43                  {
44                      code |= (0x1f & current_statement.op3)  << 11;
45                  }
46                  break;
47
48              case JMP_OPCODE:
49                  code |= (0x1f & current_statement.op_code)  << 27;
50
51                  if(current_statement.misc == LINK)
52                  {
53                      code |= 1 << 16;
54                      code |= (0x1f & current_statement.op1)  << 22;
55                  }
56
57                  code  |= (0x1f & current_statement.op2)     << 17;
58                  error |= check_immed((current_statement.op3), IMMED16,
      i);
59                  code  |= (0xffff & current_statement.op3);
60                  break;
61
62              /* ... */
63
64              case DOT_ORG_OP:
65                  lc = current_statement.op1;
66                  break;
67
68
69              default:
70                  LOG_ERROR("Invalid operation in statements list in %d
     line\n");
71                  break;
72          }
73
74          if (current_statement.op_code != DOT_ALLOC_OP &&
     current_statement.op_code != DOT_ORG_OP)
75          {
76              print_code_hex(code, inst_size, &lc);
77          }
78      }
79
80      if(error)
81      {
82          int fd = fileno(fp_hex);
83          ftruncate(fd, 0);    //clear output file
84      }
85  }
```

Algorithm 4.46: *generate_code* function implementation.

5 | Verification

Throughout the project, several tests were conducted to verify the proper functioning of the system. In the next section, some of these tests will be presented, accompanied by illustrative images.

5.1 Lexical Analysis

Starting with the lexical analysis phase, the developed logger module was used to output a readable name, each time a token was found. In Figure 5.1, it is possible to observe an example C source code, and the tokens identified by the lexer.

```
int main()
{
    float z = 3.3, y = 5.8;
    unsigned int x;

    z = 3.3;
    y = 5.8;
    x = 4 + (-3 + 4);

    x = x / 1;

    int counter = 250;

    while(counter > 0)
    {
        counter --;
    }

    return 0;
}
```

(a) Scanner example code.

```
INT ID LEFT_PAREN RIGHT_PAREN
LEFT_BRACE
FLOAT ID ASSIGN FNUM COMMA ID ASSIGN FNUM SEMI
UNSIGNED INT ID SEMI

ID ASSIGN FNUM SEMI
ID ASSIGN FNUM SEMI
ID ASSIGN NUM PLUS LEFT_PAREN MINUS NUM PLUS NUM RIGHT_PAREN SEMI

ID ASSIGN ID OVER NUM SEMI

INT ID ASSIGN NUM SEMI

WHILE LEFT_PAREN ID GREATER_THAN NUM RIGHT_PAREN
LEFT_BRACE
ID DECREMENT SEMI
RIGHT_BRACE

RETURN NUM SEMI
RIGHT_BRACE
```

(b) Scanner tokens processed.

Figure 5.1: Scanner example.

5.2 Syntatic Analysis

5.2.1 Abstract Syntax Tree

In order to verify the construction of the Abstract Syntax Tree (AST), a print function was created. As can be seen in Figure 5.2 a sample C code and the corresponding AST generated from it are presented.

The example code includes variable declarations where types, visibility qualifiers, modifiers, and other attributes are visible. For instance, *complexVar* is declared as a static, constant, unsigned integer.

Additionally, the example includes implemented functions. The main function and another function named *function* are defined. In the AST, the main function call to *function(24, 3.5)* is represented, showing the integer and float parameters passed to it. The return statement in main is also captured in the AST, displaying the integer value *0*.

Furthermore, the example contains if statements to demonstrate the creation of three child nodes for the if statement: the first node evaluates the condition, the second node represents the body of the if block and the third node represents the else block.

```c
static const unsigned int complexVar;
extern float fVar;
float function (int x, float y);

//main function
int main(){
    function(24, 3.5);
    return 0;
}

//function
float function(int x, float y)
{
    if (x > y) {
        y++;
    }
    else {
        if (x < y){
            y += 2;
        }
        else {
            y = (3 / x);
        }
    }
    return y;
}
```

(a) Example1 code.

```
----------------AST-----------------

VAR DECLARATION: complexVar
    TYPE: int
    VISIBILITY QUALIFIER: static
    MOD QUALIFIER: const
    SIGN QUALIFIER: unsigned
VAR DECLARATION: fVar
    TYPE: float
    VISIBILITY QUALIFIER: extern
Function: function
    TYPE: float
    Identifier: x
        TYPE: int
    Identifier: y
        TYPE: float
Function: main
    TYPE: int
    Function Call: function
        Integer: 24
        Float: 3.5
    RETURN
        Integer: 0
Function: function
    TYPE: float
    Identifier: x
        TYPE: int
    Identifier: y
        TYPE: float
    IF
        OPERATOR: >
            Identifier: x
            Identifier: y
        Post incrementation: y++
        IF
            OPERATOR: <
                Identifier: x
                Identifier: y
            OPERATOR: +=
                Identifier: y
                Integer: 2
            OPERATOR: =
                Identifier: y
                OPERATOR: /
                    Integer: 3
                    Identifier: x
        RETURN
            Identifier: y


---------------AST END---------------
```

(b) Example1 AST generated (terminal).

Figure 5.2: Example1 and AST generated.

This report utilizes a tree representation that employs nodes and lines to illustrate parent-child and sibling relationships within the structure.

Nodes represent individual elements, while lines indicate connections between them. Vertical lines signify child nodes, representing elements that are directly connected below a parent node. Horizontal lines denote sibling nodes, indicating elements that share the same parent node.

The Figure 5.3 presents a tree representation of the tree generated from Figure 5.2.

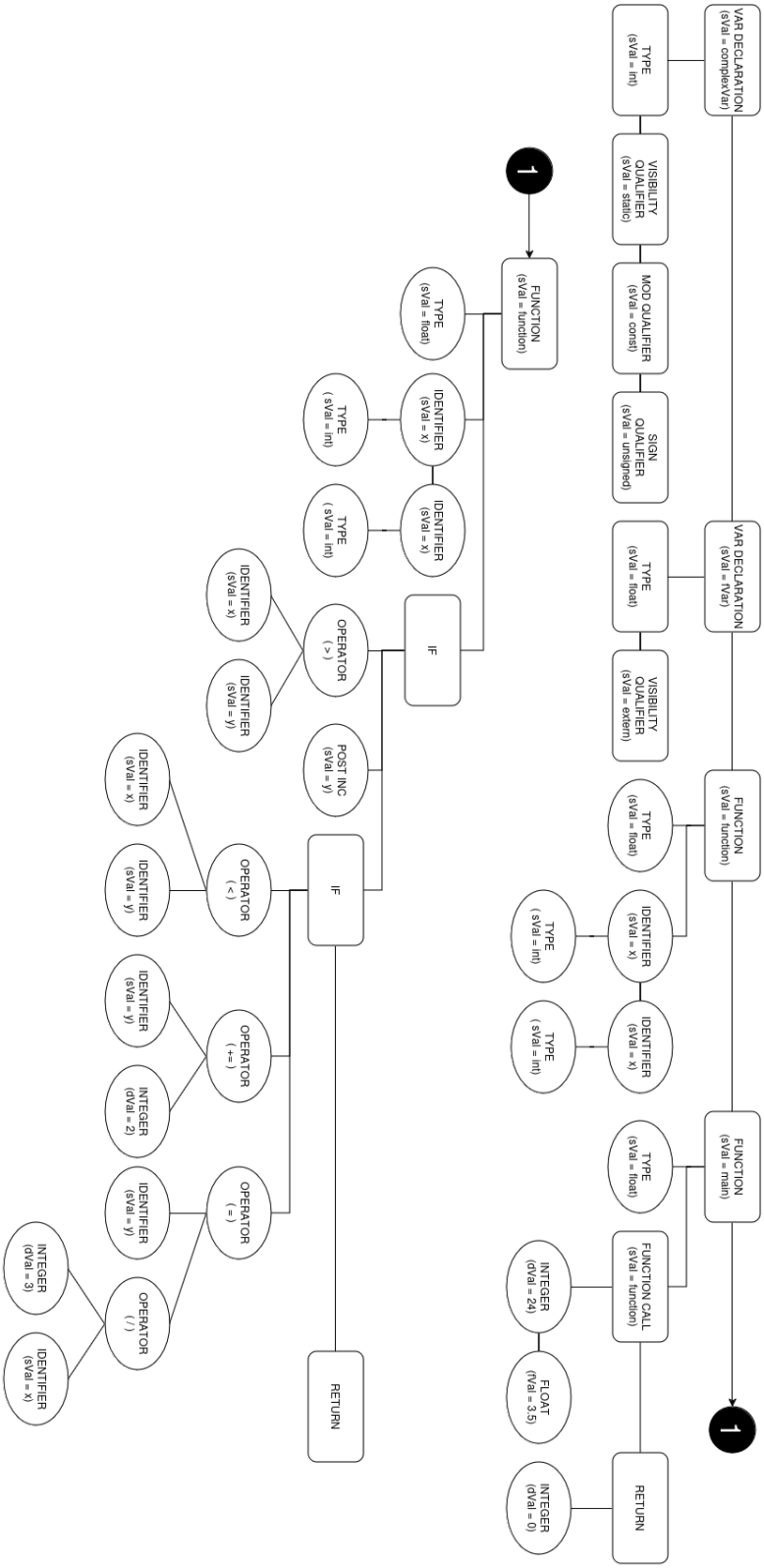


Figure 5.3: Representation of the generated Example1 AST.

5.3 Semantic Analysis

Regarding semantics, several tests were carried out in order to analyze the creation of symbol tables and verify the correct execution of the type checking algorithm.

5.3.1 Symbol Table

Alg. 5.1 presents an example code to demonstrate the creation of symbol tables. In Figure 5.4 it can be seen that for the example code, three symbol tables are created. Symbol table 0 refers to the global scope, which in this case has a global variable x, the *func1* function and the main function. For symbol table 1, this is created for the f1 function and symbol table 2 refers to the main function. In all symbol tables created, all information relevant to the symbol is observed, such as: the type of symbol, the name, the sign, among other data.

```
1  int x = 0;
2
3  void func1(const int a, float* b){
4      const int x = 9;
5  }
6
7  int main()
8  {
9      const int x = 4;
10     static float y = 3.0;
11     unsigned int array[4];
12
13     return 0;
14 }
```

Algorithm 5.1: Code example for symbol table creation.

```
------------------------------------------------------------ SYMBOL TABLE 0 ------------------------------------------------------------
|     SYMBOL |                NAME |        TYPE |       SIGN |   MODIFIER | VISIBILITY |    MEM_LOC | ARRAY SIZE |    IS_IMPL | PARAM_NUM |
|   FUNCTION |                main |         int |     signed |            |            |        n/a |        n/a |       True |         0 |
|   VARIABLE |                   x |         int |     signed |            |            |          0 |        n/a |        n/a |       n/a |
|   FUNCTION |               func1 |      signed |     signed |            |            |        n/a |        n/a |       True |         2 |
|  PARAMETERS:                   a |         int |     signed |      const | Pointer: no
                                   b |       float |     signed |            | Pointer: yes

------------------------------------------------------------ SYMBOL TABLE 1 ------------------------------------------------------------
|     SYMBOL |                NAME |        TYPE |       SIGN |   MODIFIER | VISIBILITY |    MEM_LOC | ARRAY SIZE |    IS_IMPL | PARAM_NUM |
|   VARIABLE |                   a |         int |     signed |            |            |        n/a |        n/a |        n/a |       n/a |
|    POINTER |                   b |       float |     signed |            |            |        n/a |        n/a |        n/a |       n/a |
|   VARIABLE |                   x |         int |     signed |      const |            |          0 |        n/a |        n/a |       n/a |
----------------------------------------------------------------------------------------------------------------------------------------


------------------------------------------------------------ SYMBOL TABLE 2 ------------------------------------------------------------
|     SYMBOL |                NAME |        TYPE |       SIGN |   MODIFIER | VISIBILITY |    MEM_LOC | ARRAY SIZE |    IS_IMPL | PARAM_NUM |
|   VARIABLE |                   x |         int |     signed |      const |            |          0 |        n/a |        n/a |       n/a |
|   VARIABLE |                   y |       float |     signed |            |     static |          4 |        n/a |        n/a |       n/a |
|      ARRAY |               array |         int |   unsigned |            |            |          8 |          4 |        n/a |       n/a |
----------------------------------------------------------------------------------------------------------------------------------------
```

Figure 5.4: Symbol table creation.

5.3.2 Type Checking

The example presented in 5.5 presents 4 errors to prove the correct functioning of type checking. Two errors are related to operations with operands of different types since due to the option of using strong

typed, floats cannot be used with integers. The third error occurs because you cannot pass a float as an integer parameter and the last error occurs because it is not possible to access a position in the array greater than its size.

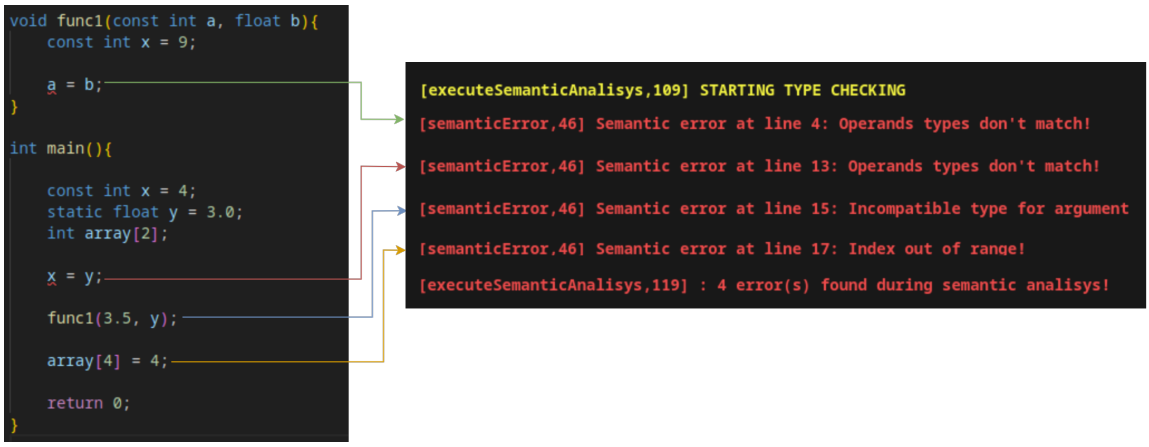


Figure 5.5: Type checking errors.

If no errors exist, a message as shown in Figure 5.6 will appear and the compiler steps will follow.

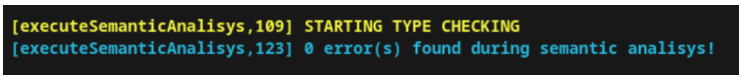


Figure 5.6: Type checking with no errors.

5.4 Optimization

5.4.1 Constant Folding

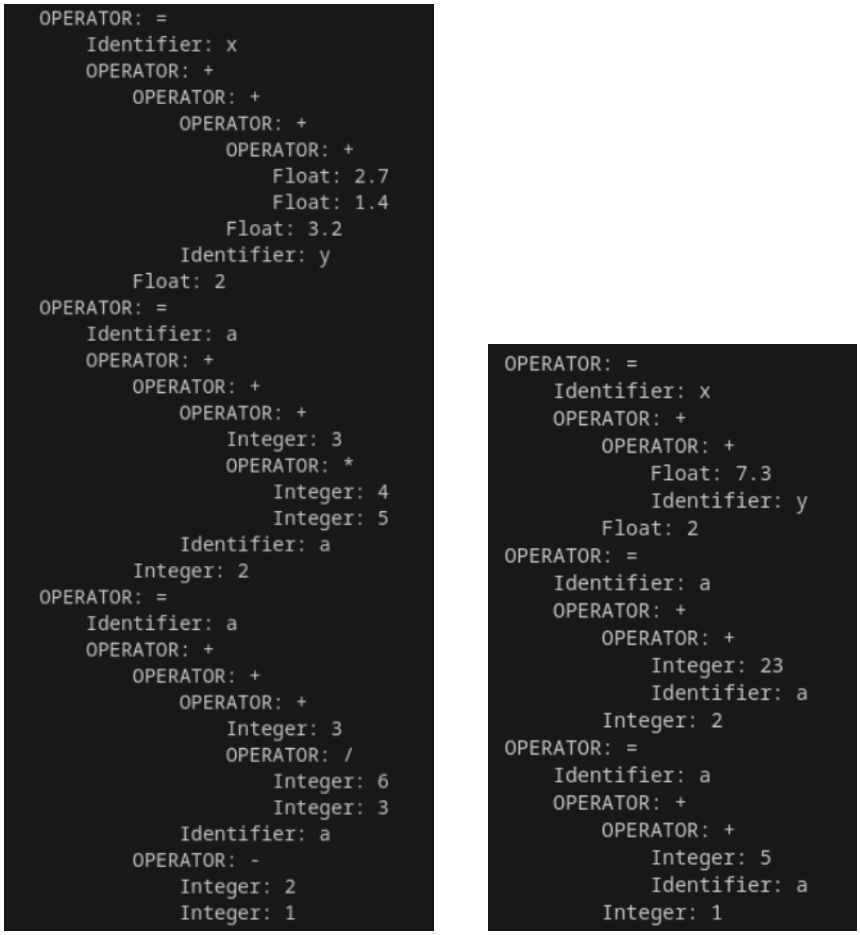
In Figure 5.7 it is possible to observe that the constant folding optimization is being executed correctly. In the examples, all operations involving constant numbers are reduced to their calculated values.

The examples used in the test were:

- $x = 2.7 + 1.4 + 3.2 + y + 2.0;$
- $a = 3 + 4 \times 5 + a + 2;$
- $a = 3 + 6/3 + a + (2 - 1);$

In the first expression of the example, the optimization process begins by simplifying the sum operation between the float values 2.7 and 1.4 is identified. This sum operation is replaced with a single float node representing the result, 4.1. Next, the expression involving the addition of 4.1 and 3.2 is encountered. This operation is also optimized by directly computing the sum, resulting in a float node with the value 7.3.

Therefore, the initial series of sum operations involving constants is reduced step-by-step to a single float node with the value 7.3.



(a) AST before optimization.

(b) AST after optimization.

Figure 5.7: Constant folding optimization.

5.4.2 Instruction Scheduling

Generic Stall Solving

In the best-case scenario, as described in Figure 5.8, the task of the instruction scheduler is significantly simplified. In this optimal situation, the scheduler's responsibility is to make straightforward adjustments to the code positions. This involves reordering instructions to eliminate stalls and ensure efficient execution flow.

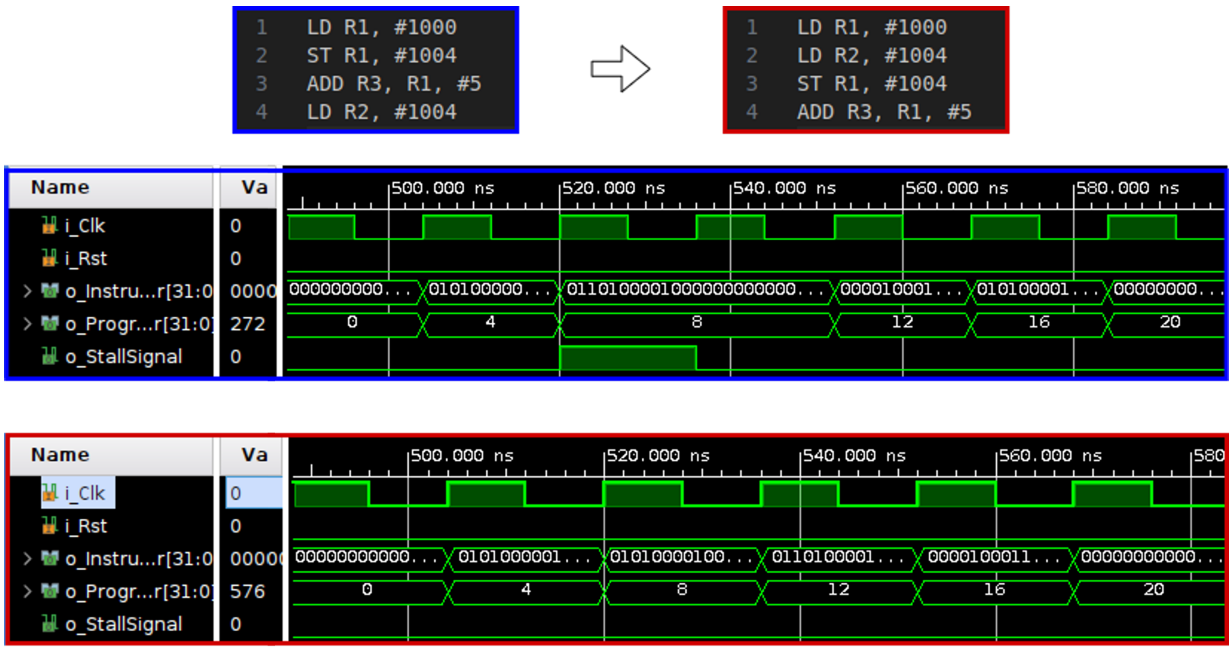


Figure 5.8: Verification for generic stall solving.

Dependency Solving

To verify the dependency presented in Figure 5.9, we conducted a series of targeted tests designed to simulate real-world scenarios and stress the dependency-solving mechanism. These tests involved varying the order of operations, deliberately introducing dependencies, and observing the system's response to these changes.

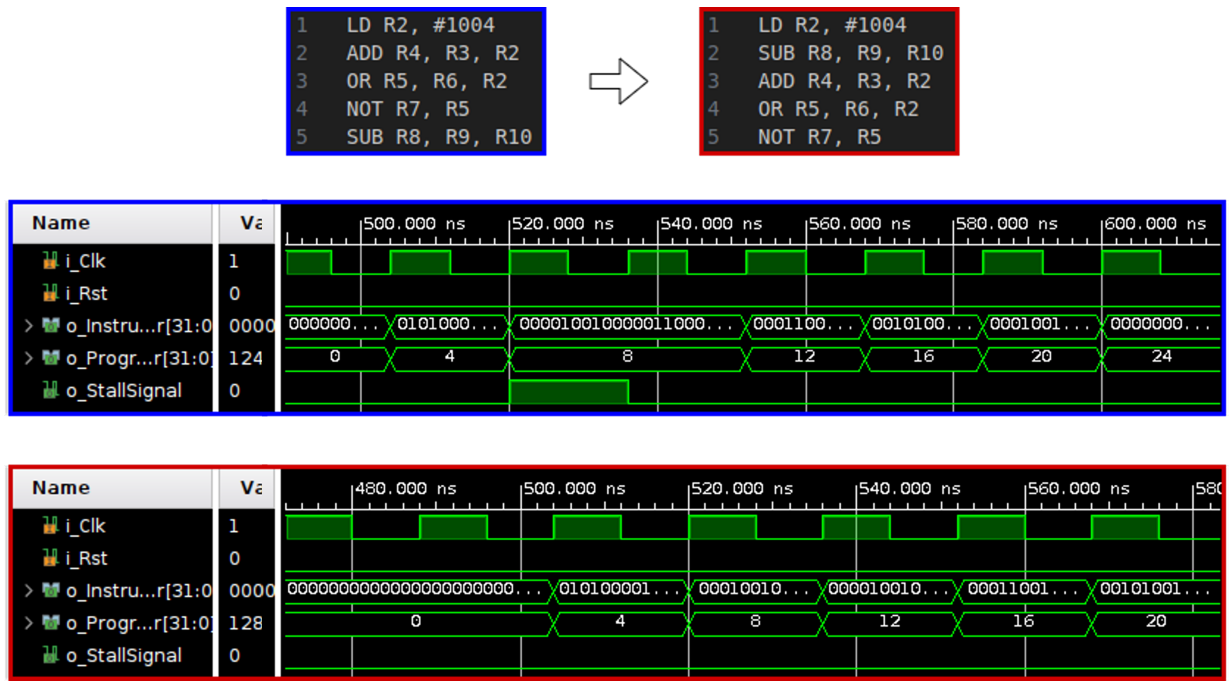


Figure 5.9: Verification for dependency solving.

To further verify the dependency-solving method, additional instructions were tested. This time, a variety of different instructions were used, and they were placed in multiple positions throughout the code. This expanded testing aimed to thoroughly evaluate the robustness and flexibility of the dependency-solving mechanism across a range of scenarios.

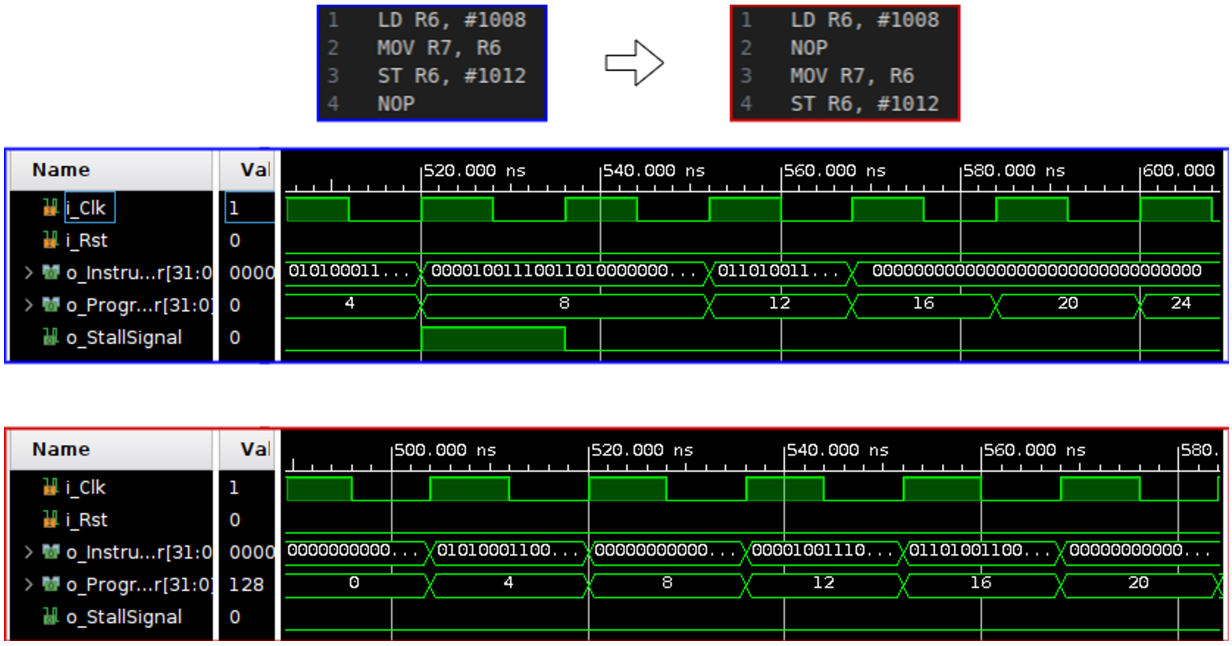


Figure 5.10: Verification for dependency solving.

These verifications provide comprehensive assurance that dependencies are being correctly managed and that the source code flow remains accurate. By systematically testing a wide variety of instructions in multiple positions, we have confirmed that the dependency-solving method effectively handles potential conflicts and maintains the intended execution order.

Stall with conditional branch replacement

For code with branch conditions, it is necessary to take into account that arithmetic instructions can never be swapped to resolve the stall. In the code Alg. 5.2 a stall occurs between lines 4 and 5 due to the load operation ($LDR0, 1200$) followed by the move operation $MOVR1, R0$). When the processor executes a load instruction, it might take several cycles to fetch the data from memory to the register, and if another instruction depends on the result of this load, it must wait until the data is available, causing a stall in the pipeline.

```
1 INIT:
2     LD R2, #10
3     LDI R3, #5
4     LD R0, #1200
5     MOV R1, R0
6     ADD R4, R3, R2
7     NOP
8     BNE INIT
```

Algorithm 5.2: Prevention of solving a stall with a new stall example.

To solve the stall in the Alg. 5.2, there are two options for replacement: reordering the ADD instruction ($ADDR4, R3, R2$) or by inserting another instruction between them, in this case the *NOP* instruction, in the line 7 can be used for that purpose.

However, the *ADD* instruction cannot be used as a replacement because it directly impacts the branch condition. Therefore, only the *NOP* remains. The Figure 5.11 demonstrates the behavior of the execution flow for the different cases.

In the first case, the initial behavior of the code is demonstrated, with the *PC* marked in red. In the second case, the *ADD* instruction is used to resolve the stall. However, as shown, the *PC* will have a different value, altering the execution flow. The flag zero visibly changes the behavior. In the final simulation, using the *NOP* as a replacement successfully resolves the hazard without altering the *PC* in the normal flow.

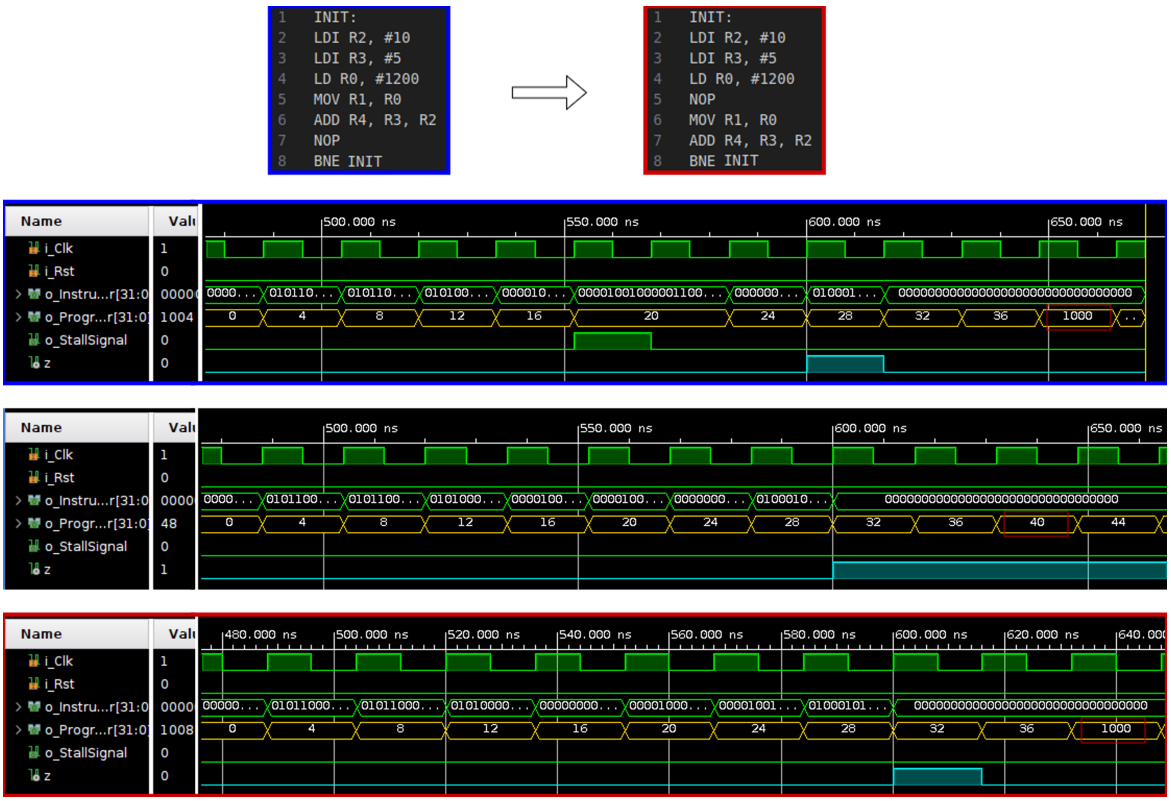


Figure 5.11: Stall with conditional branch demonstration.

Eliminating two stalls with one replace position

In the event that there are two stalls and the position of the second stall serves as the replace position for the second, the scheduler will carry out this action, resulting in the natural elimination of the second stall as can be seen in Figure 5.12.

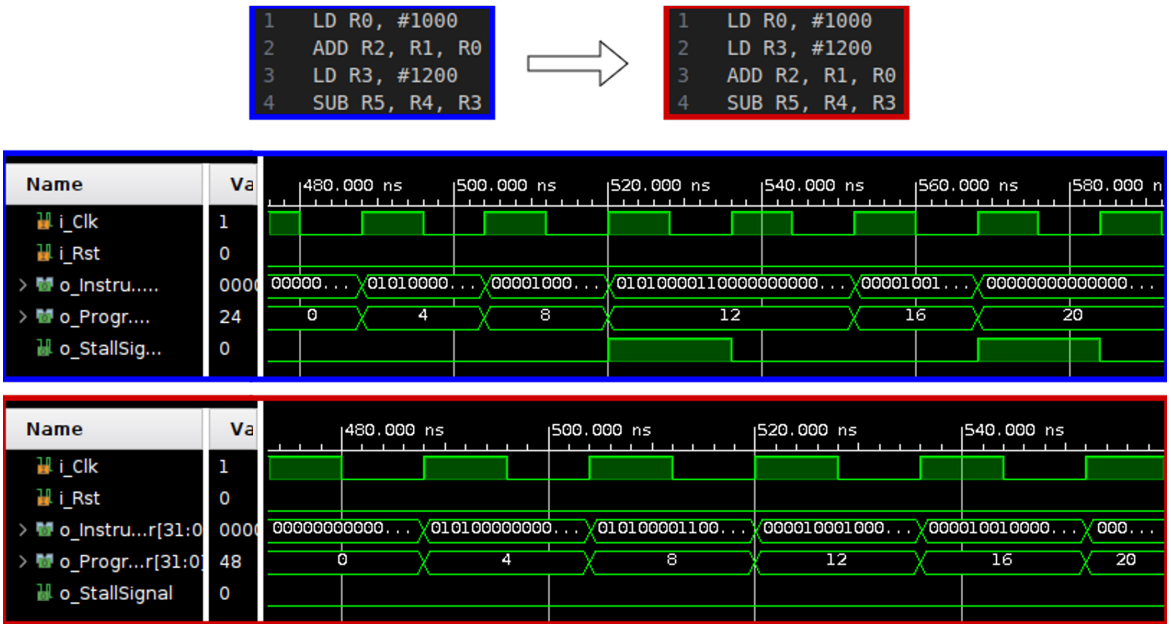


Figure 5.12: Eliminating two stalls with one replace position demonstration.

Prevention of solving a stall with a new stall

Finally, the instruction scheduling is implemented to ensure that replacements do not create new stalls. This is demonstrated in the following example:

```
1 LD R4, #1008
2 OR R6, R4, R5
3 ST R4, #1000
4 ST R2, #1004
```

Algorithm 5.3: Prevention of solving a stall with a new stall example.

For the code presented in Alg. 5.3, it is possible to see that under normal conditions, modifying the ST instruction on line 3 is plausible since it does not depend on other instructions.

However, making this change will create a new stall, as can be seen in the second case in the figure. Therefore, the scheduler will replace it with the following instruction (line 4), successfully removing the stall, as the simulation of Figure 5.13 show in the third case.

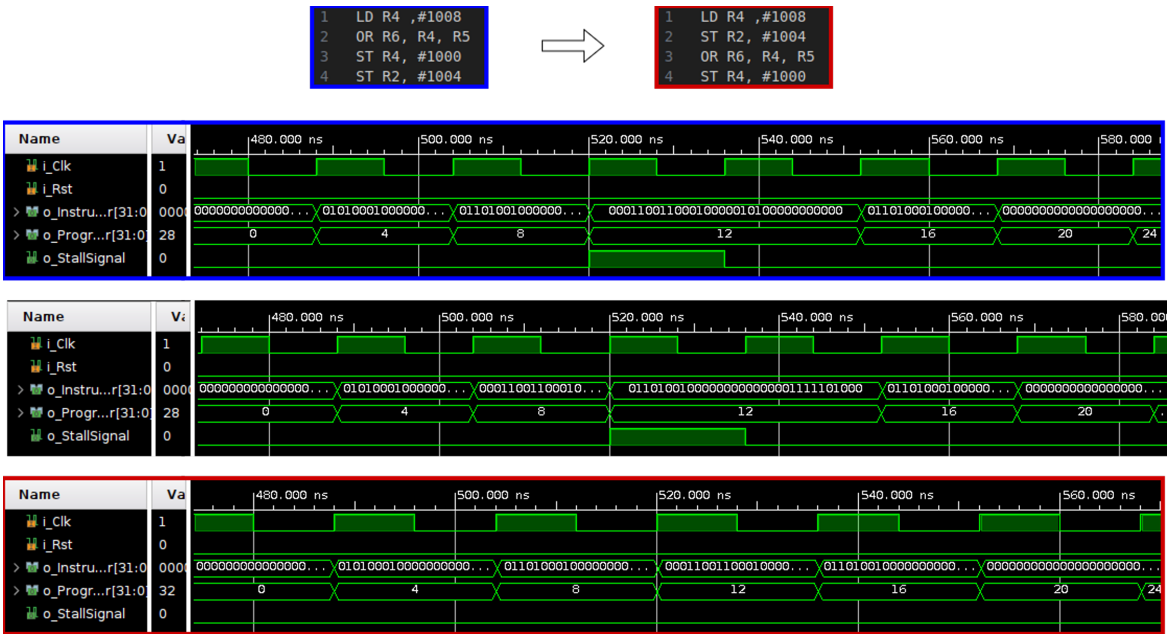


Figure 5.13: Simulation of a stall being solved and creating another stall (second simulation) and avoiding it (third simulation).

5.5 Code Generation

5.5.1 Assembler

Similar to the lexical analysis of the compiler, the assembler also performs one. Figure 5.14 presents an example code in assembly taking into account the VeSPA architecture and the tokens processed in the example.

```
.org 0x00
.equ VAR1, 100

INIT :      ;Init variables
LDI R0, #VAR1
LDI R1, #010b
LDI R2, #0x0B
LDI R3, #0AH
ADD R2, R0, R1

.equ VAR2, 11 + 4 - 5 + VAR1

LOOP :      //Computing Loop
MOV R1, R0
MOV R0, R2
ADD R2, R0, R1
SUB R3, R3, #VAR2
BNE LOOP

BRA $
```

(a) Example code in .asm format.

```
.org [HEX_NUMBER]
.equ [IDENTIFIER] COMMA [NUMBER]

[IDENTIFIER] COLON Comment
LOAD_IMMEDIATE [REGISTER] COMMA CARDINAL [IDENTIFIER]
LOAD_IMMEDIATE [REGISTER] COMMA CARDINAL [BIN_NUMBER]
LOAD_IMMEDIATE [REGISTER] COMMA CARDINAL [HEX_NUMBER]
LOAD_IMMEDIATE [REGISTER] COMMA CARDINAL [HEX_NUMBER]
ADD [REGISTER] COMMA [REGISTER] COMMA [REGISTER]

.equ [IDENTIFIER] COMMA [NUMBER] PLUS [NUMBER] MINUS [NUMBER] PLUS [IDENTIFIER]

[IDENTIFIER] COLON Comment
MOVE [REGISTER] COMMA [REGISTER]
MOVE [REGISTER] COMMA [REGISTER]
ADD [REGISTER] COMMA [REGISTER] COMMA [REGISTER]
SUB [REGISTER] COMMA [REGISTER] COMMA CARDINAL [IDENTIFIER]
BRANCH_NOT_EQUAL [IDENTIFIER]

BRANCH_ALWAYS [DOLLAR]
```

(b) Processed tokens.

Figure 5.14: Assembler example and tokens.

Figure 5.15 shows a symbol table generated from the example code presented in Figure 5.14 and an intermediate representation with the opcodes and operands of each instruction.

```
------------SYMBOL TABLE------------
index: 001  |  value: 00000  |  name: 0
index: 002  |  value: 00100  |  name: VAR1
index: 003  |  value: 00000  |  name: INIT
index: 004  |  value: 00110  |  name: VAR2
index: 005  |  value: 00020  |  name: LOOP
-----------------------------------
```

(a) Symbol table.

```
************INTERMEDIATE REPRESENTATION************
LINE    OPCODE      OP1     OP2     OP3     MISC
-------------------------------------------------------
01       43         255     000     000     0
02       11         000     100     000     0
03       11         001     002     000     0
04       11         002     011     000     0
05       11         003     010     000     0
06       01         002     000     001     0
07       01         001     000     000     0
08       01         000     002     000     0
09       01         002     000     001     0
10       02         003     003     110     1
11       08         005     011     000     0
12       08         040     000     000     1
*******************************************************
```

(b) Intermediate representation.

Figure 5.15: Assembler symbol table and intermediate representation.

Considering the decision to generate an intermediate file in hexadecimal and with the format shown in Figure 5.16 (a), an extra step must be performed, in order to convert this to a valid COE file. The binary code shown in 5.16 (b), reflects the assembly code shown in Figure 5.14.

```
@0000 00 00 00 00
@0004 58 00 00 64
@0008 58 40 00 02
@000c 58 80 00 0b
@0010 58 c0 00 0a
@0014 08 80 08 00
@0018 08 40 00 00
@001c 08 04 00 00
@0020 08 80 08 00
@0024 10 c7 00 6e
@0028 45 ff ff f0
@002c 40 00 00 00
```

(a) Generated code in hexadecimal format.

```
memory_initialization_radix=2;
memory_initialization_vector=
00000000000000000000000000000000,
01011000000000000000000001100100,
01011000010000000000000000000010,
01011000100000000000000000001011,
01011000110000000000000000001010,
00001000100000000000100000000000,
00001000010000000000000000000000,
00001000000001000000000000000000,
00001000100000000000100000000000,
00010000110001110000000001101110,
01000101111111111111111111110000,
01000000000000000000000000000000;
```

(b) Binary code in coe file.

Figure 5.16: Hexadecimal and binary assembler code.

5.5.2 Assembly Code Generation

In the following images, the assembly code generation for the example shown in Alg. 5.4 is presented. This example demonstrates a simple code that calls a function foo and, depending on an if condition, either adds or multiplies the values passed as arguments to the main function, returning that value and storing it in the result variable.

```
1 int y; //this is placed at 0x00 (global scope)
2 int foo(int a, int b);
```

```
3
4  int main()
5  {
6      int result;
7      int x = 6;
8      y = 4;
9
10     result = foo(x, y);
11     return 0;
12 }
13
14
15 int foo(int a, int b)
16 {
17     int sum;
18
19     if(a > 5)
20         sum = a + b;
21     else
22         sum = a * b;
23
24     return sum;
25 }
```

Algorithm 5.4: Example code in C language

At the beginning of the assembly code generation, the code shown in Alg. 5.5 is always generated. This code is essential for initializing the parameters and directing the processor to the start of the program, specifically to the main function.

```
1  .org 0x00
2      JMP R0,#64
3
4  .org 0x4000
5      HALT
6
7  .org 0x40
8      LDI R2,#1023
9      LDI R3,#1023
10     LDI R1,#16384
11     SUB R3,R3,#1
12     STX R1,R3,#0
13     LDI R1,:FUNCTION_main
14     JMP R1,#0
```

Algorithm 5.5: Generated code in .asm format - .org

In Alg. 5.6, the complete assembly code generated from the .c file represented in Alg. 5.5 can be examined. This generated code ensures that all requested values are correctly stored in their respective variables and confirms the presence of a JMPL instruction for calling the foo function. In the code snippet provided below, detailed comments accompany each line, allowing for a comprehensive understanding of the code generation process.

```
FUNCTION_main:
    ; Reserve Stack space for local "result" and "x"
    SUB R3,R3,#1
    SUB R3,R3,#1
    ; Store 6 to x
    LDI R12,#6
    STX R12,R2,#-3
    ; Store 4 tp y
    LDI R12,#4
    ST R12,#0
    ; Load x to push to stack to pass as arg
    LDX R14,R2,#-3
    SUB R3,R3,#1
    STX R14,R3,#0
    ; Load y to push to stack to pass as arg
    LD R14,#0
    SUB R3,R3,#1
    STX R14,R3,#0
    ; Push Frame Pointer to Stack
    SUB R3,R3,#1
    STX R2,R3,#0
    ; Copy current Stack Pointer to Frame Pointer
    MOV R2,R3
    ; Save return address Register R1
    SUB R3,R3,#1
    STX R1,R3,#0
    ; Load function address and jump to it
    LDI R14,:FUNCTION_foo
    JMPL R1,R14,#0
    ; Upon return to main flow, increment Stack Pointer
    ; to free space occupied by arguments
    ADD R3,R3,#2
    ; Copy contents of return regster R4 to variable result
    MOV R12,R4
    STX R12,R2,#-2
    LDI R12,#0
    ; Main Return
    MOV R4,R12
    ; Copy FP to SP
    MOV R3,R2
    ; Restore R1 (that holds the return address)
    SUB R3,R3,#1
    LDX R1,R3,#0
    ADD R3,R3,#1
    ; Restore R2
    LDX R2,R3,#0
    ADD R3,R3,#1
    ; Jump to address in R1
    ; R1 before main was executed is preloaded with an address
```

```
50      ; that contains an halt instruction to stop execution
51      JMP R1,#0
```

Algorithm 5.6: Generated code in .asm format - Function Main

The Alg. 5.7 illustrates the entire assembly code generation process for constructing an if statement. It generates an auxiliary label (SKIP) to determine the truth of the condition and the labels of the if statement itself (IF_FALSE, IF_EXIT), facilitating the correct program flow.

```
1  FUNCTION_foo:
2      ; Reserve stack space for local "sum"
3      SUB R3,R3,#1
4        ; Get the value in "a"
5      LDX R13,R2,#2
6      LDI R14,#5
7      ; Preload the register containing the result from condition with 1
8      LDI R12,#1
9      ; Compare "a" with 5
10     CMP R13,R14
11     BGT SKIP_0
12     ; Assign 0 to R12 if cond is false, else it keeps the value 1
13     LDI R12,#0
14 SKIP_0:
15     ; Check if the condition is true or false
16     CMP R12,#0
17     BEQ IF_FALSE_0
18     ; True Block starts here
19     ; Get values of "a" and "b", add them and store them in a
       temporary block
20     LDX R13,R2,#2
21     LDX R14,R2,#1
22     ADD R12,R13,R14
23     STX R12,R2,#-2
24     ; Skip the "else" body
25     BRA IF_EXIT_0
26 IF_FALSE_0:
27     ; Load values of "a" and "b"
28     LDX R13,R2,#2
29     LDX R14,R2,#1
30     ; Pass "a" and "b" as arguments for multiplication through R4 and
       R5
31     MOV R4,R13
32     MOV R5,R14
33     ; Call the multiplication function
34     LDI R15,:FUNCTION_FUNCTION_MUL
35     JMPL R1,R15,#0
36     ; Get the return value from the function
37     MOV R12,R4
38     ; Store the value in "sum"
39     STX R12,R2,#-2
40 IF_EXIT_0:
```

```
41      ; Load the value in "sum"
42      LDX R12,R2,#-2
43      ; Place the value in the return register
44      MOV R4,R12
45      ; Copy FP into SP
46      MOV R3,R2
47      ; As the return address is on a offset of minus 1 from stack
        pointer
48      ; decrement 1 from stack pointer and then do the restore
49      SUB R3,R3,#1
50      LDX R1,R3,#0
51      ADD R3,R3,#1
52      ; Restore previous Frame Pointer
53      LDX R2,R3,#0
54      ADD R3,R3,#1
55      ; Jump to the return address
56      JMP R1,#0
```

Algorithm 5.7: Generated code in .asm format - Function foo

5.6 RocketSim

In order to ease the software development process, for the developed VeSPA SoC, and considering that deploying said software to the target hardware, in this case the Zybo Z7 FPGA, can be a time consuming process, and that debug tools are overall restricted in the target hardware, a virtual runtime environment was developed.

RocketSim, is able to execute machine code targeted for the VeSPA architecture, while running on any x86 machine. Besides this, the developed tool also provides different debugging tools, such as, setting breakpoints, reading the current state of the CPU registers, as well as dumping a memory region.

Considering the target hardware also implements different memory mapped peripherals, it is of great interest to also be able to simulate the behaviour of these. With this in mind, the RocketSim tool also implements peripheral mocking, which allows the user to define how the simulation should behave when this virtual hardware devices are accessed.

Bellow, in Figure 5.17, it is possible to find the simple Command Line Interface (CLI), that allows the user to interact with the simulator.

```
1. Free Run
2. Goto next breakpoint
3. Set breakpoint
4. Dump registers
5. Dump memory
6. Exit
```

Figure 5.17: RocketSim CLI.

When the first entry on the menu (Free Run) is selected, the simulation will run freely, until further user

input is needed, which can occur, for example, when an hardware mock requires interaction from the user.

The next menu entry (Goto next breakpoint) will execute code the same way as the previously mentioned option, except it will halt execution if a breakpoint is hit.

In order for the previous entry to work correctly, it is first necessary to set breakpoints. This menu option allows to do so, by selecting the option with the following syntax: 3 <Address>, where 3 is the menu option and <Address> should be replaced by the target address in hexadecimal representation.

When the fourth entry (Dump registers) is selected, all the CPU registers are printed out to the console. This option can be of great importance when trying to debug code.

The fifth menu entry is very similar to the previous one, the only difference is that it prints a range of memory values to the console. It can be used with the following syntax: 5 <StartAddress> <EndAddress>, where 5 is the menu option and <StartAddress> is the address where the memory dump should start, and <EndAddress> is the address where it should end.

The last menu entry allows the user to exit the tool, even during execution.

For demonstrating the capabilities of the developed tool, the machine code seen bellow, in Figure 5.18, was used. The example code calculates the Fibonacci Sequence up to N iterations, where N is given by a value read from the UART peripheral.

The RocketSim tool uses the intermediate hex file format, which can be seen on the Figure 5.18, as an input format for machine code.

```
INIT:
    LDI R0, #0
    LDI R1, #0
    LDI R2, #0
    LD R3,  #1035
    ADD R2, R0, R1

LOOP:
    MOV R1, R0
    MOV R0, R2
    ADD R2, R0, R1
    SUB R3, R3, #1
    CMP R3, #1
    BGT LOOP
```

(a) Machine code for the Fibonacci sequence test.

```
@0004 58 40 00 00
@0008 58 80 00 00
@000c 50 c0 04 0B
@0010 08 80 08 00
@0014 08 41 00 00
@0018 08 05 00 00
@001c 08 80 08 00
@0020 10 c7 00 01
@0024 38 07 00 01
@0028 42 ff ff e8
@002C ff ff ff ff
```

(b) Intermediate file in hex format of the Fibonacci sequence test.

Figure 5.18: Testing Fibonacci sequence with RocketSim.

After executing the tool and passing the target machine code file as an argument, the user greeted with the previously described CLI menu (Figure 5.17).

After selecting one of the possible run options ("Free Run" or "Goto next breakpoint"), the code will execute until either an HALT instruction is found, or further user input is needed. As mentioned previously, the test code in use reads the number of iterations from the UART peripheral. Considering this, the implemented peripheral mock, requires the user to input a value in the CLI which simulates what the real

hardware would read, as can be seen in Figure 5.19.

```
1. Free Run
2. Goto next breakpoint
3. Set breakpoint
4. Dump registers
5. Dump memory
6. Exit
1
Waiting on UART RX value...
A
```

Figure 5.19: RocketSim simulation waiting for an input value.

When an HALT instruction is found, the simulation is stopped, here the user is again presented with the main menu, which allows to perform some further debugging of the code (Figure 5.20). Considering the example under study, and considering an user input of 0x0A (10 in decimal format) the expected Fibonacci sequence for this number of iterations is 55 in decimal format. By dumping the registers to the console, it is possible to observe that the value present in R0, does in fact match with the expected value, and with this, it is possible to confirm the correct behavior of the program.

```
HALT found! Stopping CPU!
1. Free Run
2. Goto next breakpoint
3. Set breakpoint
4. Dump registers
5. Dump memory
6. Exit
4
R0 -> 55
R1 -> 34
R2 -> 89
R3 -> 1
```

Figure 5.20: Registers values after the completion of the Fibonacci sequence test.

6 | Conclusion

To conclude the report, the project development was a success, with all of its objectives met, also with some additional work being performed. Besides the successful implementation of the lexical, syntactic and semantic analysis and the associated code generation according to the VeSPA ISA, some additional optimizations were performed in the generated code, such as constant folding and stall avoidance. Additionally, an emulator was developed to streamline the testing process, allowing for a faster verification of the performed implementation, particularly regarding the integration tests developed to test the compiler as a whole.

As in every project, it went through some ups and downs, with times where it seemed that with every forward step taken, two or three steps back taken as well. Several challenges arose during the development of the compiler, such as correctly integrating the different phases of the compiler and ensuring the correctness of the generated code. These difficulties were addressed through rigorous testing and constant improvements, which in itself helped to increase the class capabilities and a deeper understanding of the concepts learned throughout the semester.

In summary, the project delivered a functional compiler with reliable performance, laying a solid foundation for future developments and acting as a testament to the amazing heights that can be achieved when a good educational foundation is paired with hard work.

Bibliography

[1] Prof. Dr. Adriano Tavares. *Compilador(overview)*. Slide.

[2] Prof. Dr. Adriano Tavares. *FasesDeCompilação(Scanner)*. Slide.

[3] Prof. Dr. Adriano Tavares. *Lex(Scanner)*. Slide.

[4] Prof. Dr. Adriano Tavares. *FasesDeCompilação(Parser)*. Slide.

[5] Prof. Dr. Adriano Tavares. *FasesDeCompilação(BottomUpParser)*. Slide.

[6] Prof. Dr. Adriano Tavares. *FasesDeCompilação(analise semantica)*. Slide.

[7] Prof. Dr. Adriano Tavares. *Analise semantica(tabela de simbolos)*. Slide.

[8] Prof. Dr. Adriano Tavares. *Analise semantica(typechecking)*. Slide.

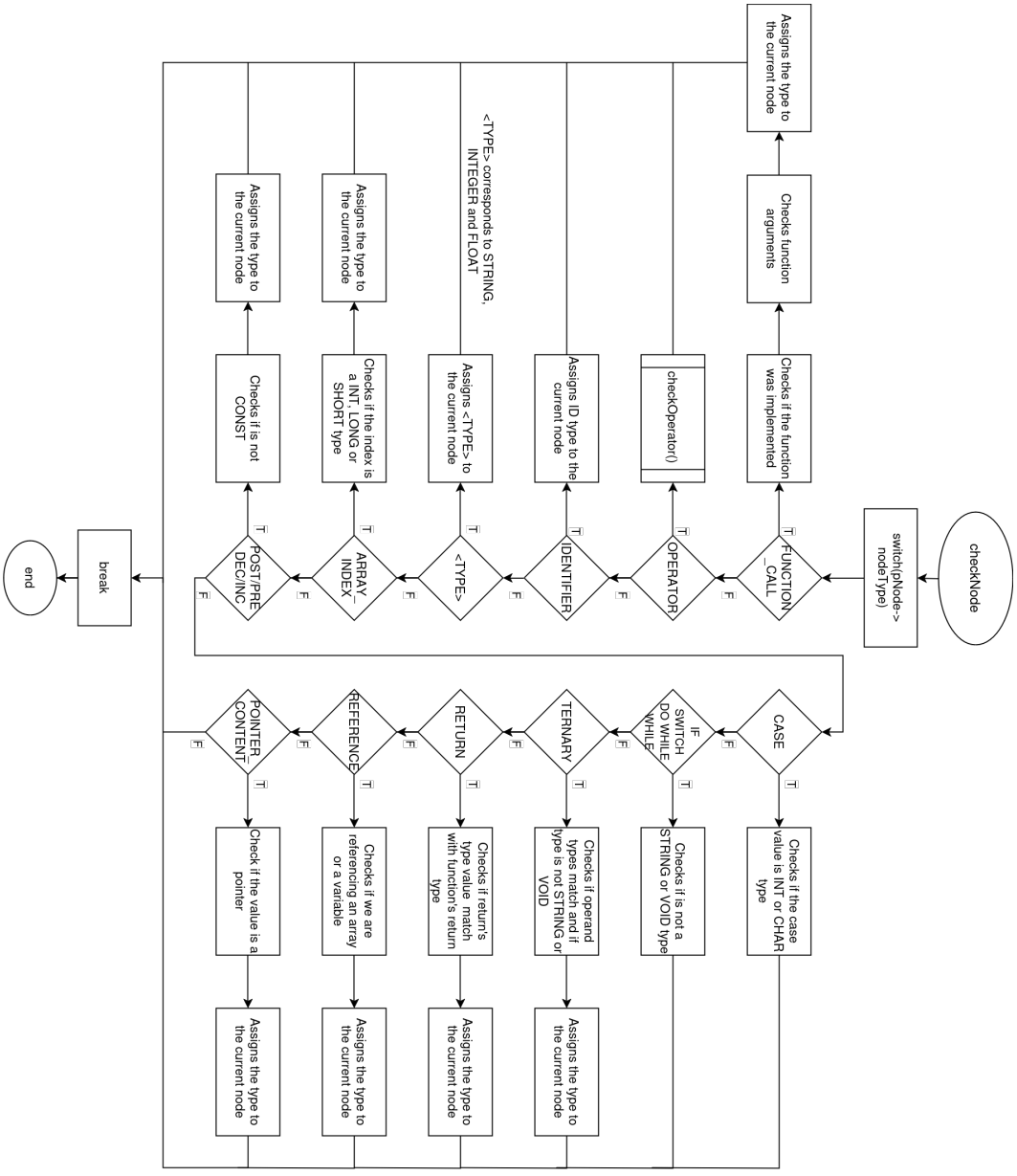
[9] Prof. Dr. Adriano Tavares. *Geração de código (ambiente de execução)*. Slide.

[10] Prof. Dr. Adriano Tavares. *Geração de código(Code Generation)*. Slide.

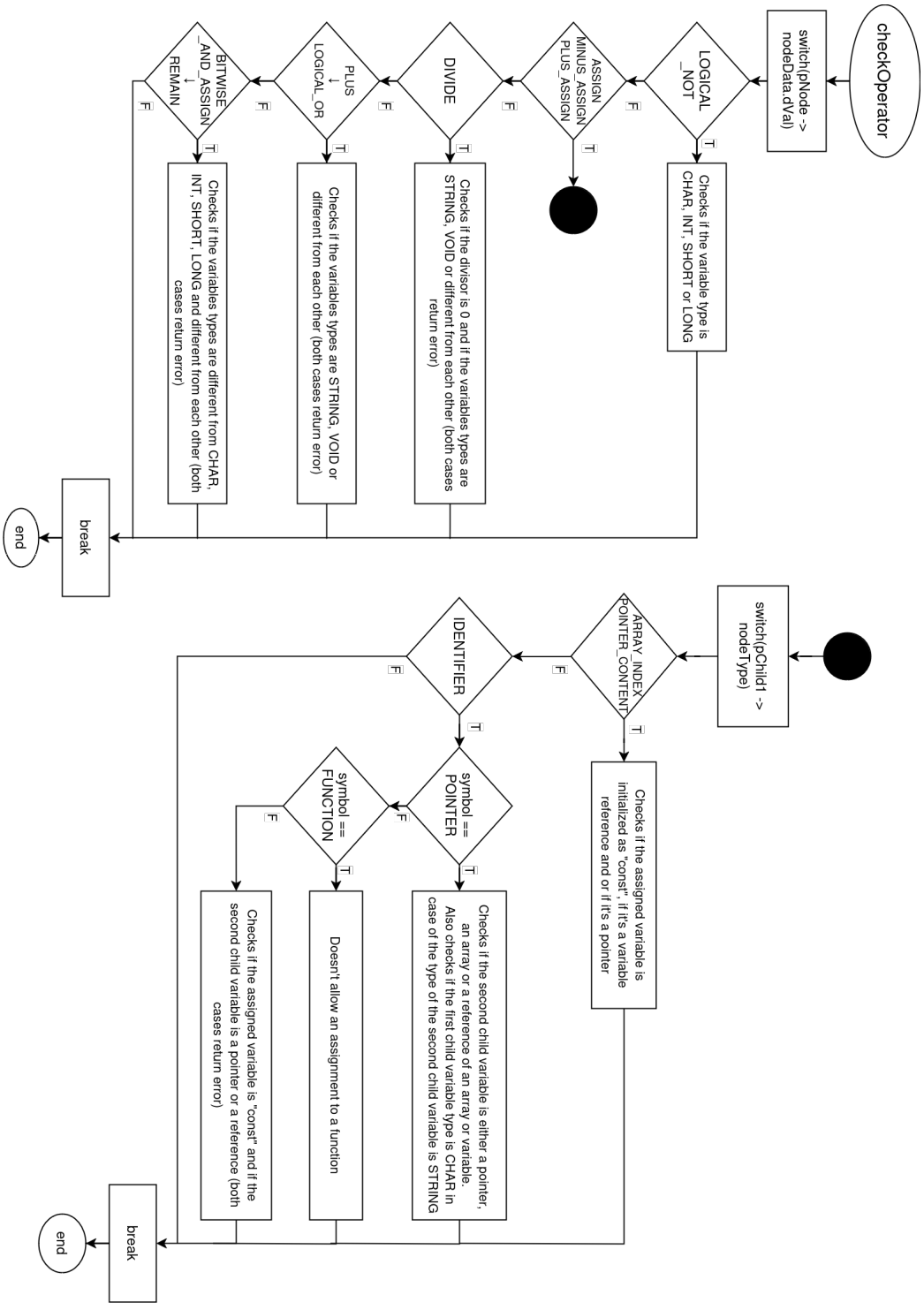
[11] Sandler, Nora. Writing a C Compiler. No Starch Press, 2022.

[12] Lilja, David J., and Sachin S. Sapatnekar. Designing Digital Computer Systems with Verilog. Cambridge University Press, 2005.

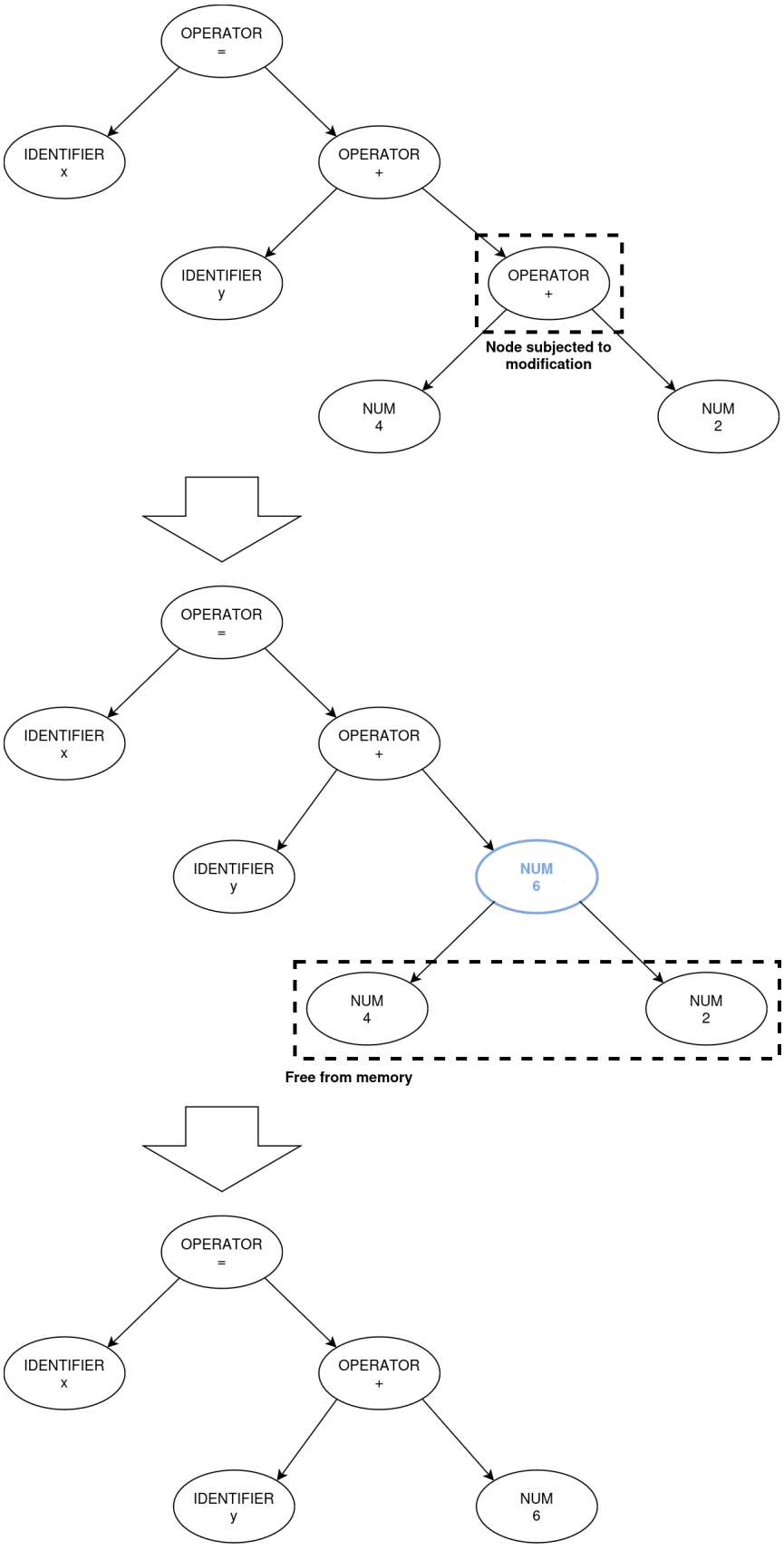
A Type Checking and Rules Checking of AST Nodes Flowchart



B Type Checking and Rules Checking of Operators and Operands Flowchart



C | Constant Folding Process



D | Constant Folding Handling