Design and Implementation of Lexical and Syntax Analysis for an Assembly Language Compiler

Gontla Venkat Sujan, Cheekati Mani Shankar, Kavitha C.R.*
Department of Computer Science & Engineering
Amrita School of Computing, Bengaluru
Amrita Vishwa Vidyapeetham, India
bl.en.u4cse21061@bl.students.amrita.edu , bl.en.u4cse21046@bl.students.amrita.edu , cr_kavitha@blr.amrita.edu

***Abstract*— The Design and Implementation of Lexical and Syntax Analysis for an Assembly Language Compiler recognizes the problem of translating the raw instructions of an assembly language into valid structured instructions Error-free execution of assembly instructions requires very careful interpretation of Instructions, Registers, Labels, and Operands. The goal of this work is to design a compiler that can perform lexical analysis to tokenize a code, syntax analysis to verify and parse instructions and execution to mimic the role of a processor. In the results, it is shown that the compiler can interpret difficult instructions, offer constructive information about errors that occur, and graphically display flowcharts to illustrate the flow of execution, all of which make it a useful tool for studying assembly language programs. The outcome is an Abstract Syntax Tree for well-formed code and informative error messages for ill-formed code structure to foster deep exploratory exercises in compiler functionality during lower realms of compilation. The compilation process is demonstrated comprehensively without delving into semantic analysis or code generation. This structured approach helps one to get acquainted with the first steps of compilation which is of high value when analyzing low-level code, checking instructions, and handling errors in 'assembly language'.**

Keywords*— *Assembly Language, Compiler Design, Syntax Analysis, Lexical Analysis, Error Handling, Visualization.

.

I. INTRODUCTION

Assembly language is an intermediate language between high-level languages and low-level codes providing developers the facility to manage the hardware resources on their own. However, a very close relationship with machine code feature poses great problems because it has a highly structured language, it is implicitly concerned with semantics and relies on low level constructs. Everybody will need to work with assembly language since it personal computer's internal instructions read directly by computer's hardware .It is important to work with assembly language by using tools like the compilers in the correct ways. The optimizer for the high level language is likely to be designed with considerations given to abstraction while that of assembly language is likely to be designed with the given priorities considering accurate physical execution and different from a traditional compiler in the sense that it requires localized design methodologies when developed.

One of the most difficult tasks while designing a compiler for assembly language is to convert direct assembly code enunciated by the programmer directly into a certified format. The instruction set of the assembly has its own general and specific syntax rules governing instructions, operands, registers or labels. To alter them in even the slightest manner can result in execution errors, or create conditions that are undefined. For instance, the branch instruction would require the labels to be defined correctly and arithmetic would require registers to be initialized correctly. Absence or incorrect analysis of these components can hinder program flow and as such the compiler must accurately interpret and validate at each of these levels. The proposed methodology seeks to solve these challenges through developing a compiler that will take signatures that perform lexical and syntax analysis for assembly language. They called it a decoding process, estimating the syntactic correctness of instructions, modelling code hierarchies via Abstract Syntax Trees, and processing instructions on an emulated register set. Moreover, the compiler incorporates methods to check for and report problems, and provides graphics to display instruction dependencies and program flow. The central goal is to develop a strong, speedy, and scale-out compiler that makes ASM more manageable for the consumer and at the same time give him/her deeper insights of the assembly language.

The main focus is on the lexical and syntax analysis of assembly language and the act of coming up with a compiler for it. The lexical analysis component transports raw text and turns it into structured tokens named token namely instructions, registers, labels and literals out of assembly code. The syntax analysis then uses these tokens with the help of a parser to confirm admissible sequences of instructions and operands arrangement with reference to predefined syntactic rules. These two fundamental principles are thus implemented.

II. LITERATURE SURVEY

A. Barve et al.[1] explains about the technique to parallelize the lexical analysis phase of the compiler on multi-cores namely For loops. By partitioning lexical analysis tasks, the proposed approach has been shown to produce substantial speedup by using all the available cores with the use of processor affinity. The outcome shows that this parallelism enhances this method's performance, other enhancements should include reaching out to other forms of control. Joldzic Băicoianu et al.[2] developed practical guidelines and recommendations, especially to graduate students and programmers, in constructing an efficient scanner (lexer), considering also the proper tools currently available.

Sanju, V et al.[3] raises the significance of lexical analysis to identify the syntax error earlier in the compilation process to serve the compilation process and to minimize errors. Moreover, it shows different approaches and the problems associated with creating good lexical analysers, in order to apply efforts to enhance one of the most crucial preliminary steps in compiling. H. Luo et al.[4] develops a new algorithm known as LL (1) grammar-syntax parsing algorithm is presented by the authors of the paper. In the algorithm, the so-called parsing table for prediction within the syntax is used, which will noticeably facilitate this process. It is built from the grammar, then once it has been constructed it can be used in a table driven manner to complete filling in the table perfectly by the parser. The method provides the examples of its

practical applicability in the exposition of how the Predictive Parsing Table will be highly important in the field of NLP. According to Mulik et al.[5] a comparative view of some techniques the author elaborated for parsing of a formal language, concerning to another advantage or a disadvantage of other methods that can be top-down or bottom-up parsing.om the grammar, and having constructed the table then enables the parser to complete filling it in perfectly using a table-driven algorithm. The method shows examples for practical use in illustrating how the predictive parsing table will be very useful in the domain of natural language processing..

Mulik et al.[5] provides a comparative analysis of several techniques developed to parse formal languages is given with respect to other strengths and weaknesses of different methods like top-down or bottom-up parsing. The presentation of an algorithm of the solution, like LL and LR parsing, is given and a consideration made on the efficiency and possibility of such algorithms used in compiler design and syntactic analysis. As in the case of the preceding papers, this paper also stresses on the specific relevance of different parsing techniques as regards the types of grammars that are appropriate for them.

Asăvoae, M et al.[6] deals with usages of the K framework for describing the semantics of assembly languages, and an object of analysis here is SSRISC assembly language. As the research work also finds, in K it becomes possible to make models of the semantics which can be specified in assembly language to be modular and executable without getting entangled in the matters of structure and computation read imbrication. Salloum, S.A et al.[7] discusses several semantic analysis categories of methods used in programming language design, and the techniques used in each category. The paper also supports the idea that performing semantic analysis is vital for language interpretation, structuring and enhancing the correctness of programs.

Leicai Xiao et al.[8] highlights the aspect of the development of assembly language for the reconfigurable flexible assembly lines. It analyses the problems and the prospects of assembly language programming for automated manufacturing systems with examples of industrial robotization. José Meseguer et al. [9] highlights the specific goals of the RLS project in terms of rewriting logic, which constitute the framework of non-ILM rewriting logics. The authors are able to demonstrate how RLS can be used successfully to characterize languages as well as systems and the outstanding performance of RSL in modelling dynamic behaviours is well exemplified.

Vidhate et al. [10] details the data about lexical analysis and the Lex analyzer generator. The work concerns the automatic component of lexical analysis and reveals that Lex does not destroy the idea of the compiler construction but on the contrary helps to embrace it in a more efficient manner as well as to provide pin-point input code parsed with better accuracy.

III. METHODOLOGY

The implementation of the compiler is structured into four major phases: The functional elements include: lexical analysis, syntax analysis and data retrieval, execution, and visualization. As it has been explained above, every phase has a great importance in the process of converting the valuable assembly code into more meaningful and feasible form for execution.

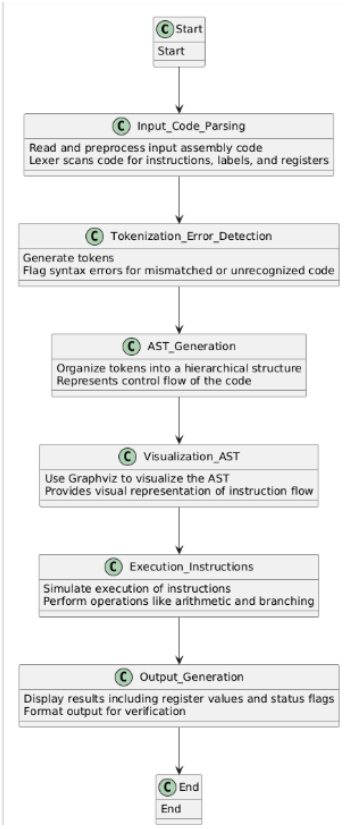

Fig.1. Flowchart describing the implementation

1. Input Code Parsing:

The initial phase starts with reading assembly language code file as input. The input is read line by line and classified into tokens like instruction, register, labels and intermediates among others. This also involves addition of labels that represent them mapped to their respective memory addresses all done to identify and mark syntax errors. To convert the input assembly code to tokens. Tokens are the symbols that are in the instruction set, Mnemonics (MOV, ADD), operands (R1, #5), labels are the tokens. In this phase, there is the utilization of regular expression to match predefined instruction's patterns and check up on the syntax of the code. The lexer associates labels and converts them to memory WINDOW locations for instructions. He noted that any code that is not written in the correct syntax set off a syntax error and informs the user of a line that has the error.

2. Tokenization and Error Detection:

An instruction with their corresponding operands is formatted into tokens, using predefined string patterns. In the case that the lexer comes across an unfamiliar instruction or problematic syntax, error prompt, and cease operation. This

means only syntactically correct instructions pass through to the next level translation.

3. AST Generation (Parser Phase)

The tokens parsed are then used by the parser to form an Abstract Syntax Tree (AST). Every instruction goes to a node and these are connected by the edge to its operands. In case of branching instruction, the whole structure of this branching is interpreted by the parser and the address of the corresponding label is added to the structure of the abstract syntax tree, AST. The ACT stands for Abstract Syntax Tree which is an organized representation of the structure of a program. Facilitates organize the token into Abstract Syntax Tree (AST) the level hierarchy code. The parser checks all operands of instructions for their correct format and looks up labels for branching operations. With respect to the subsequent execution, the syntax of AST may be considered as more structured than that of the source code. This phase also elaborates the initialization of a simulated register set used throughout the execution phase.

4. Visualization of AST

The AST is represented to enable easy visual interpretation of the instructions flow and dependencies, as well as the conditional branching. The visualization – the kind of layout shown here – emphasizes control flow, illustrating where the program goes and how it makes jumps for branching instructions. It improves the readability of the code by the user through the rendering of. In efficient an instruction graph with each node corresponds to an instruction and the edge represents the operand and the control flow. It is used whenever a user needs to understand the dependency scheme within their assembly codes so that they can analyze as well as debug it effectively.

5. Execution of Instructions

The execution phase imitates the behavior of the specified assembly code. Data arithmetic operations such like ADD and SUB modify the value of one register through the other register as operand. Arithmetic and/or logical operations such as AND, ORR, etc, branching conditions such as BGT, BEQ, etc. are performed and the program counter is adjusted accordingly while the Z, N status flags are also set.

6. Output Generation

The results of the computation are shown in tabular form, of the values being in the registers and status flags. Furthermore, Should any branch conditions that have been executed, their correctness of jumps to program is also ensured.

7. Execution Phase:

The AST and executes the instructions in a sequential manner. ‘Register transfer level’ type operations such as ADD, SUB, AND are performed using simulated registers. Conditional branch instructions such as BEQ and BNE update the program counter (PC) based on condition outcomes with a view of allowing a change of control flow. The programs also introduce the ability to compare the register values (CMP) and Adjust/level condition flags (Z/N) for supporting branching decisions. Proper error control checks ensure that if there are analyzed uninitialized registers or invalid operations are not handled causing undefined reactions.

IV. RESULTS

The implementation of the proposed methodology has achieved its aim of realising the compiler capable of correctly tokennizing, parsing and executing the assembly code objectives. The lexical analysis phase recognized and efficiently segmented instruction, operand, and labels to guarantee that each successfully parsed line of code follows a set format. Specific gross syntactic mistakes were identified and flagged with reference to the lines where the mistake was made, thus allowing the users to make the necessary changes quickly. There was a considerable improvement on the quality of the parser in generating an organized AST that well showcased the entire logic of the program.

The execution phase gave a proper imitation of the operation of a processor as it carried out mathematics and logical operations on the register, and condition checking and branch control. Shifting based on the condition flags (Z, N) provided precise control flow transition and made the emulation of actual assembly execution environment possible. Single-byte operations with significant error checking capabilities were employed so as not to execute uninitialized registers, further improving the stability of the system.

The AST was visualized where detailed instruction relationships plus the control flow where made available. I benefited greatly from this feature to debug complex assembly programs and to find out how the components within a program depend on each other.

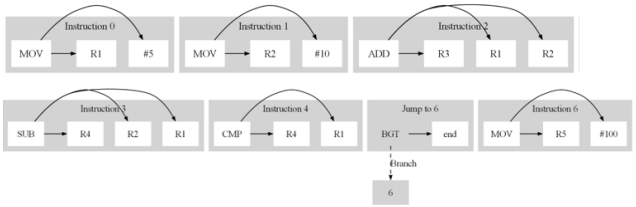

Figure 2 Abstract Syntax Tree

Figure 2. represents Abstract Syntax Tree generated after compilation of Assembly level code.

The above result displays a general representation of how an AST looks like in relation to a sample assembly code line; Instructions that have a relationship in the assembly language are formatted hierarchically as well as graphically. Every shape in the picture corresponds to a single assembly instruction; the latter is designated by a number starting from 0 up to n (for example, Instruction 0, Instruction 1, Instruction n). These nodes represent the operation type, for looping operations or moving information, for instance, MOV, ADD, SUB or control flow like BGT, then the nodes link to the operands that are shown as sub nodes. Operand can be a register such as R1, R2 and or immediate such as 5, 10 making the structure all rounded.

The layout of the AST is in hierarchical format and highlights the dependencies between the instructions and operands. For instance, Instruction 0 is the MOV that loads the value #5 into the register R1; Instruction 1 is also the MOV

that loads the immediate value #10 into register R2. The graphical structure also affects the way data is displayed so as to allow efficient examination of how it courses through the instructions.

An attribute of this AST is to the control flow instructions well. For example, Instruction 5 describes the BGT (branch if greater than) function which is based on the comparison made in the Instruction 4 (CMP function). If the condition given by BGT is met then control goes to Instruction 6 as shown by the dotted line connecting two nodes. It is essential to understand the form of conditional jumps and program's flow by defining the control flow graphical representation. On balance, the above structures that makeup the AST are effective in representing both the linear progression of the instructions/enactment of the type as well as the logic branches of the type plainly and intuitively. This representation makes processing and adoption easier especially in debug and optimization together with the general structuring of the program.

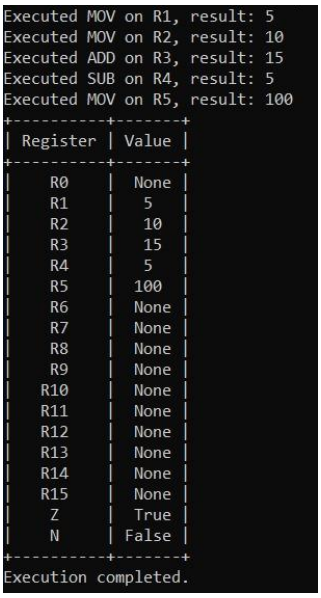

Figure 3 Output of Code Execution

Figure 3. Represents the generated output and represents the Memory allocation of registers.

When looking at the content of each register, we notice that register 1 contains 5, register 2 contains 10, register 3 contains 15 (looks like the sum of R1 and R2), register 4 has 5 which must be the difference between R 3 and R 2, and finally, register 5 contains 100. R6 to R15 are the remaining registers, which are not at all used in the current computing mode. The flag register implies that the Zero flag (Z) is ON suggesting that the last arithmetic operation produced zero. Negative flag is False meaning that the result is not negative and rather positive or less severe.

In the case of an assembly language compiler, this output would be created after the assembly compiler has a 'working' machine code implementation and runs it on a simulator, or a hardware system. Before generating code the compiler would have to undergo lexical analysis (breaking it down to tokens), syntactical analysis (analyzing the syntax of the code), and semantic analysis (checking for semantic errors).

V. CONCLUSION

The results of the implemented methodology have highlighted the need for structured parsing and validation during the construction of assembly language compilers. Apart from what is in this implementation, the approach can be extended to accommodate more instruction level assembler instructions or other optimizations if need arises. Additionally, the graphical representation of the defined AST considerable facilitates the educational process and can help the developers who faced a problem in comprehend or to debug the low-level code. Altogether, the work serves as a link between the low-level assembly language which is almost impervious to structural adaptations and the contemporary computing systems' requirements. Through providing an error-free code and observing the execution of procedures, the compiler design establishes the basis for the future evolutions of low-level programming tools. This work is relevant to the general field of compiler and provides the foundation for furthers advances in assembling language processing and optimization. Lexical and syntax analysis for an AL assembly language compiler discussed in this report contributes fundamental progress towards systematic low level program check and run. According to the research aim and the main implementation goal, the transformation of raw assembly code into structurally more manageable codex has been achieved. Hence, with the focus to ease the complexity in working with assembly language, as seen by a strict syntax and complicated branching mechanisms, the proposed solution shows how parsing of such code and hierarchical representation through the use of an Abstract Syntax Tree (AST) is beneficial.

The AST representation with a focus on hierarchy successfully captures the control flow and data flow so that the program organization is transparent. The technique of visualization is very useful during debugging and optimization because it emphasizes the dependencies between operations, values, operands, and conditions if-else. Further, how arithmetic and logical instructions are implemented, and the high-level handling of conditions and branching means that complex control flow is preserved and accurate.

The execution engine of the system helps to check the correct operating mode of the system by emulating the register operations and the branches of conditions. Just like MOVE, ADD, SUB and condition statements are also well managed in the method enhancing the endurance of the procedure. As explained in the Results section, the AST provides a view of the program with Operand Dependence as well as Condition Flow clearly while the execution results assure the correct spend of the system.

REFERENCES

[1] A. Barve and B. K. Joshi, "A parallel lexical analyzer for multi-core machines," *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, Indore, India, 2012, pp. 1-3, doi: 10.1109/CONSEG.2012.6349505.

[2] Băicoianu, Alexandra & Plajer, Ioana. (2023). Considerations on efficient lexical analysis in the context of compiler design. Bulletin of the Transilvania University of Brasov. Series III: Mathematics and Computer Science. 159-168. 10.31926/but.mif.2023.3.65.2.14.

[3] Sanju, V., 2016, March. An exploration on lexical analysis. In 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT) (pp. 253-258). IEEE.

[4] H. Luo, "A Kind of Syntax Parsing Algorithm Based on The Predictive Parsing Table," *2022 IEEE 5th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, Chongqing, China, 2022, pp. 1122-1126, doi: 10.1109/IMCEC55388.2022.10020088.

[5] Mulik, Sunanda, Sheetal Shinde, and Smita Kapase. "Comparison of Parsing Techniques For Formal Languages." *International Journal on Computer Science and Engineering* 3, no. 4 (2011): 1611-1615.

[6] Asăvoae, M., 2014. K semantics for assembly languages: A case study. *Electronic Notes in Theoretical Computer Science*, *304*, pp.111-125

[7] Salloum, S.A., Khan, R., Shaalan, K. (2020). A Survey of Semantic Analysis Approaches. In: Hassanien, AE., Azar, A., Gaber, T., Oliva, D., Tolba, F. (eds) Proceedings of the International Conference on Artificial Intelligence and Computer Vision (AICV2020). AICV 2020. Advances in Intelligent Systems and Computing, vol 1153

[8] Leicai Xiao, Long Zeng, Zhaobo Xu, Xueping Liu,Assembly language design and development for reconfigurable flexible assembly line,Robotics and Computer-Integrated Manufacturing,Volume 80,2023,102467,ISSN 0736-5845

[9] José Meseguer, Grigore Roşu,The rewriting logic semantics project,Theoretical Computer Science,Volume 373, Issue 3,2007,Pages 213-237,ISSN 0304-3975

[10] Vidhate, Maneesh. (2020), AN EXPLORATION OF LEXICAL ANALYSIS AND LEX ANALYZER GENERATOR. 10.13140/RG.2.2.33941.76003