UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
Facultad de Ingeniería

Computer Engineering
Compilers
C to Z80 Assembler Compiler

Students:

320317460
423112645
423002205
320262614

Team: 09
Professor:
Rene Adrián Dávila Pérez
Group 5, Semester 2025-2

México, CDMX
Deadline:
June 8 2025

Contents

1 Introduction

1.1 Problem Statement

A compiler is a fundamental tool in computer science that translates source code written in a language into another language (usually into a lower-level one). The compilation process consists of several stages, with the analysis phase being one of the most crucial. Within this phase, the lexical analyzer, or scanner, is responsible for processing the input code and breaking it down into a stream of tokens. These tokens serve as the fundamental building blocks for later stages, such as syntax and semantic analysis.

Once we have understood the main function of all components of a compiler, it is time to assemble it.

First, we define the grammar of our language. It must be a context-free grammar (CFG) that represents all valid constructions of the language. After that, we have to make sure that our grammar is compatible with a top-down recursive descent parser, which means it should be an LL(1) grammar. If the grammar is not LL(1), we must transform it to make it compatible, even if some rules look more complex or ugly than before.

Then, we write the lexer. This part of the compiler is in charge of dividing the input text into tokens, such as identifiers, numbers, operators, keywords, and literals. The lexer removes white spaces and comments and gives a clean sequence of tokens for the parser.

Next, we implement the parser using recursive descent. This parser uses functions that follow the rules of the grammar and tries to match the token stream. If the input follows the grammar, the parser accepts it; if not, it gives an error.

After the parser, we add the syntax tree generation. The syntax tree is a representation of the structure of the input code, and it helps us understand the meaning and the operations that we need to do.

Finally, we write the code generator. It takes the syntax tree and creates assembly code that can be executed by a machine. In our case, we initially planned to generate NASM x86 assembly code, but later decided to adapt the compiler to generate Z80 assembly code instead, due to the opportunity to reuse a Z80 assembler we had previously developed. This assembler takes the generated Z80 code and produces the final object code that can be executed by a Z80-based machine or emulator.

In this project, we are not considering code optimization. The goal is to make a compiler that works and is functional, even if the generated code is not optimal. Once the compiler works, we can think about improving it later.

1.2 Motivation

Since we started learning about compilers, we realized that they are not just a tool used by big companies, but also a great way to understand how programming languages work internally. Building a compiler from scratch helps us to really understand all the parts that are involved when we write and run a program. It's not only about translating code; it's also about learning how languages are designed, how errors are detected, and how computers understand instructions.

This is excellent for us, who decided to continue directly from studying Formal Languages and Automata, and also because we are somewhat experienced in replicating an assembler in a high-level language. Given that we had already developed an assembler for the Z80 architecture, we saw a great opportunity to integrate both efforts—modifying our compiler so that it produces Z80 assembly code as output, which is then processed by our assembler to obtain the final machine code.

For us, doing this project is a challenge, but also an opportunity to apply many things we have seen in class like grammars, parsing, recursion, trees, and code generation. Even if it looks hard at first (especially in classroom), it's very satisfying to see how each part works and connects with the next one.

In the end, our motivation is to learn, to build something that actually works, something that can guarantee us a good grade.

1.3 Objectives

The main objective of this project is to design and implement a functional compiler for a small programming language, using a top-down recursive descent parser. We want to show that we understand how each phase of a compiler works and how they are connected.

To achieve this, we define the following specific objectives:

- Define a context-free grammar for our chosen language and make sure it is LL(1) compatible.
- Develop a lexer that can correctly identify and classify tokens such as identifiers, keywords, numbers, operators, and symbols.
- Implement a top-down recursive descent parser based on the grammar rules.

- Integrate syntax tree generation into the parser for better representation of the program structure.
- Create a code generator that transforms the syntax tree into Z80 assembly code.
- Ensure the compiler is functional and can translate input programs without syntax errors into Z80 assembly code, which is then assembled using our pre-existing Z80 assembler to produce the final executable machine code.

2 Theoretical Framework

To build a basic compiler, we must understand the main concepts and components involved in the compilation process. These concepts come from both academic theory and practical experience. Below we explain the key ideas that support our implementation, based on the content learned in class and supported by reliable sources like the Dragon Book [1], community experience [2] and also we have been influenced by the Let's Build a Compiler guide from Jack Creenshaw [3].

2.1 Compiler and Its Phases

At the introduction of this text we already stated what a compiler is, so now we have to define all the extensive process of building one. This process typically consists of multiple phases that systematically transform the source program into executable instructions. These phases include lexical analysis, parsing, semantic analysis, intermediate code generation, optimization, and Target code generation. Each phase produces an intermediate representation of the program that is closer to machine execution. In our project, we focus on building a functional compiler pipeline without incorporating optimization strategies, allowing us to understand the essential compilation stages thoroughly [1].

2.2 Context-Free Grammar

Programming languages are formally described using context-free grammars (CFGs), which define the syntactic structure of valid programs through production rules. CFGs enable the expression of nested (like nested if's or for's) and recursive language constructs (like recursive functions), which are common in programming languages. They serve as the blueprint for syntax analysis and are fundamental to parsing. Designing a correct and unambiguous CFG is critical for the successful implementation of the parser and the overall compiler. [4]

2.3 LL(1) Grammar and Recursive Descent Parsing

LL(1) grammars are an important class of context-free grammars that are particularly suitable for compiler construction due to their simplicity in syntactic analysis and also for top-down parsers because it is always possible to correctly predict the expansion of any non-terminal [5]. The notation LL(1) means that the parser reads the input from **Left to right** (L), produces a **Leftmost derivation** (L), and uses **1 token of lookahead** (1). This means the parser decides which production rule to apply by looking only at the next token without needing backtracking. [1]

This type of grammar enables the implementation of top-down predictive parsers such as the **recursive descent parser**, where each non-terminal symbol in the language is translated into a recursive function. This method is intuitive, easy to implement, and produces readable and maintainable parser code.

However, many natural grammars are not LL(1) because of issues like **left recursion** or **ambiguities in choosing productions based on a single lookahead token**. To convert a grammar into LL(1), transformations such as eliminating left recursion and performing left factoring are required. These transformations allow the parser to correctly decide which rule to apply simply by examining the next token.

Although the rewriting process can be tedious and sometimes complex, it is essential for building a functional and efficient recursive descent parser. Therefore, understanding and manipulating LL(1) grammars is fundamental in compiler design theory [5].

2.4 Lexical Analysis

Lexical analysis is the first phase of compilation and involves processing the source code as a stream of individual characters, grouping them into higher-level structures called *tokens*. These tokens represent the fundamental syntactic units of a programming language, such as keywords, identifiers, numeric literals, operators, and punctuation. During this phase, irrelevant elements such as whitespace and comments are discarded to simplify the parser's job.

Lexical analyzers are typically based on regular expressions and implemented using finite-state automata. According to the Chomsky's hierarchy, regular grammars (Type 3) can be parsed using finite-state machines, making them suitable for token recognition. Since many low-level language components (like identifiers and numbers) fall into this category, it's both common and practical to delegate token recognition to a separate lexical scanner module.

However, as Crenshaw says, many compilers can be developed effectively without investing heavily in traditional scanner construction techniques. By carefully specifying the syntax of the language, it is possible to design simple and efficient lexical analyzers without the complexity usually presented in textbooks. His approach is based on the "Keep It Simple, Sidney" (KISS) principle, and demonstrates that practical compilers can be built with simple, hand-written scanners [3].

Although lexical scanning could theoretically be integrated with the parser, separating it provides significant practical benefits. For instance, distinguishing whether a sequence of characters like `IF` is a keyword or an identifier often requires reading ahead until a delimiter is encountered. This necessity introduces limited backtracking or lookahead, which is more easily managed when scanning is performed independently.

In our project, the lexer operates as a distinct module from the parser. It tokenizes the input using a simple, deterministic approach and assumes the input adheres to the language's lexical rules. This division of labor between scanner and parser enhances clarity and maintainability, and follows both theoretical reasoning and practical experience.

2.5 Abstract Syntax Tree and Parsing

Once the input is tokenized, the parser takes the resulting sequence of tokens and verifies whether it conforms to the grammatical rules of the programming language. If the input is valid, the parser constructs an Abstract Syntax Tree (AST), which reflects the hierarchical syntactic structure of the source program.

Unlike a concrete syntax tree, the AST abstracts away syntactic sugar (A prettier way to write something more complex) and ignore irrelevant tokens that are not semantically meaningful such as parenthesis or delimiters. This is possible due to the hierarchical relationship between the elements already encoded in the structure of the tree. As a result, the AST focuses solely on the essential language constructs.

This tree structure is essential for subsequent compiler phases, such as semantic analysis and code generation. During semantic analysis, the AST is traversed to perform tasks like type checking, symbol resolution or scope management. Later, it provides the structural basis to generating intermediate code generation. The AST is central to the subsequent phases as it organizes the program logic in a structured and traversable form. [1].

2.6 Code Generation

The final step in our compilation pipeline is code generation, where the abstract syntax tree (AST) is transformed into low-level target code. In our project, this means generating **Z80 assembly instructions** from high-level constructs such as expressions, control flow, and variable operations. Each node in the AST is traversed, and corresponding assembly instructions are emitted to perform the intended operations. This phase marks the point where the abstract logic of the program is finally connected to hardware-level execution, producing assembly code that can be assembled and run on a Z80-based machine or emulator.

According to the dragonbook [1], the code generation phase is critical because it must ensure correctness, adhere to calling conventions, and make efficient use of machine resources like registers and memory. While advanced compilers implement optimizations such as instruction selection, register allocation, and scheduling, even a minimal code generator focuses on translating semantic constructs into a correct and executable form. This transformation involves important decisions, such as how expressions are evaluated (stack-based or register-based), how conditional jumps are implemented, and how local variables are laid out in memory. In the context of the Z80, this includes knowledge of limited general-purpose registers, explicit memory addressing, and the use of flags for conditional branching. The authors emphasize that the generation of intermediate or final machine code is the culmination of all previous phases, tying together syntax, semantics, and program intent.

The Stack Exchange Community [2] highlights that even in the simplest compiler projects, implementing a working code generator offers invaluable insight into how source code maps to machine instructions. Beginners quickly learn that while parsing and AST construction deal with structure and meaning, code generation brings the abstract into the physical world, making the program runnable. This phase tests not only understanding of the source language but also knowledge of the target architecture, including its instruction set, stack behavior, and runtime conventions. In our case, adapting to the Z80 required us to understand the architecture's accumulator-based operations, conditional jump instructions (such as `JP Z`, `JP NZ`), and memory-mapped data access. Successfully completing code generation — even without optimizations — represents a key milestone in compiler construction, demonstrating a full traversal from high-level language down to machine-executable code.

2.6.1 Integration with the Z80 Assembler

To complete the translation from high-level code to executable form, our project incorporates a final step beyond traditional code generation: the use of a separate **Z80 assembler**. This assembler takes the Z80 assembly code produced by our compiler and transforms it into object code that can be directly executed on a Z80-based system or loaded into an emulator.

Unlike the compiler's backend, which focuses on converting the AST into syntactically correct Z80 instructions, the assembler is responsible for resolving labels, calculating memory addresses, handling symbolic constants, and producing a binary or hexadecimal output in a format suitable for execution. In our case, this component was built previously in another project, and integrating it with the current compiler allowed us to create a full toolchain from source code to executable machine code.

The assembler reads files with the `.asm` extension, which follow a structure starting with headers like:

```
CPU "Z80.tbl"
HOF "INT8"
```

These directives are inserted by the compiler during code generation, and the remaining lines consist of actual Z80 instructions derived from the AST traversal. The assembler processes this file and emits the corresponding binary code, taking into account Z80-specific instruction formats and addressing modes.

By integrating both the compiler and assembler, we achieved a fully automated pipeline capable of compiling C-like programs into Z80 machine code. This not only demonstrates our understanding of compiler construction but also our familiarity with low-level architecture and the requirements for real execution on hardware or simulation environments.

3 Development

3.1 Lexical Analysis

First for our lexical scanner, we need to choose the tokens that we are going to identify. For this purpose, we selected a subset of the C programming language:

- **Keywords:** `int`, `while`, `pragma addr`, ...
- **Identifiers:** variable names following standard naming rules.
- **Literals:** numeric constants such as integers.
- **Operators:** arithmetic operators like `+`, `-`, assignment operator `=`, comparison operators such as `>`, `<`, `==`, etc.
- **Punctuation:** parentheses `(`, `)`, braces {, } and semicolons `;`.

By tokenizing the input source code in this way, we prepare a clean and simplified stream of tokens that can be easily processed by the parser in the next phase.

We made it a OOP implementation of this so we manage the tokens as an objects, this objects are classify into different types of tokens that we expect to encounter in the source code. These token types correspond to the smallest meaningful units that the lexer will recognize and classify.

In our implementation, the token types are enumerated as follows:

1. `#PRAGMA_ADDR`, `INT`, `MAIN`, `WHILE`: Keywords of the language.
2. `ID`: Identifiers, which represent variable names.
3. `NUM`: Numeric literals, such as integers.
4. `PLUS`, `MINUS`: Arithmetic operators `+`, and `-`.
5. `ASSIGN`: Assignment operator `=`.
6. `LPAREN`, `RPAREN`: Left and right parentheses `(` and `)`.
7. `LBRACE`, `RBRACE`: Left and right braces { and }.
8. `SEMICOLON`: Semicolon `;` used to terminate statements.
9. `GT`, `LT`, `GE`, `LE`, `EQ`, `NEQ`: Comparison operators `>`, `<`, `>=`, `<=`, `==`, and `!=`.
10. `EOF`: Special token indicating the end of the input stream.

So this token class has his type and the string that contains the token itself.

Now the lexer is designed as a class that processes the input string character by character to produce tokens one at a time. It maintains a current position in the input and reads the characters sequentially.

The lexer provides utility methods to classify characters, such as checking if a character is whitespace, a letter, or a digit. It also includes a method to skip over any whitespace characters, ensuring that irrelevant spacing does not interfere with token recognition.

The core method of the lexer is `nextToken()`, which identifies and returns the next token from the input stream. The method works as follows:

- It first skips any whitespace characters.
- If the end of the input is reached, it returns an `EOF` token to signal the end of the source code.
- If the current character is a letter, it starts collecting characters to form either a keyword or an identifier. It continues reading letters and digits until it encounters a non-identifier character. It then checks if the resulting string matches any reserved keywords (e.g., `int`, `while`, `pragma add`, `main`). If it does, it returns the corresponding keyword token; otherwise, it returns an identifier token.
- If the current character is a digit, it collects the entire numeric literal by reading consecutive digits, then returns a numeric token.
- If the current character matches any operator or punctuation symbol (such as `+`, `-`, `=`, `==`, `>`, `>=`, parentheses, braces, or semicolons), the lexer returns the corresponding token. In the case of multi-character operators like `==`, `>=`, `<=`, and `!=`, it checks the next character to distinguish between the single and double character operators.
- If an unexpected character is encountered, the lexer throws an error, indicating invalid input.

This approach ensures that the lexer converts the raw input string into a well-defined stream of token objects, each representing the smallest unit of meaning in the language. The parser can then consume these tokens to perform syntactic and semantic analysis.

By structuring the lexer as a class that returns `Token` objects with both a type and the actual text, the design follows object-oriented principles, which helps in maintainability and extensibility of the compiler.

Notice that the process of classifying tokens is quite straightforward and heavily inspired by the Crenshaw guide [3]. Instead of using regular expressions, which are available in Java, we identify each token by matching the exact word using a `switch`-case structure. This approach is suitable given the small subset of the language that we are handling.

3.2 Grammar

The grammar represents our first major step in initiating the development of our compiler. It can be considered the most crucial phase, as all subsequent components and behaviors of the compiler will be based upon it.

The target language we chose to compile and what already mentioned in the lexical analysis is a simplified subset of the `C` programming language. This subset includes support for variable declarations, basic arithmetic operations such as addition and subtraction, and one selected control structure. For our implementation, we decided to include the `while` loop as the representative control flow mechanism. This choice allows us to demonstrate the handling of iteration constructs within our compiler while keeping the complexity manageable at this stage.

The grammar defined was the following:

$$
\begin{aligned}
\textit{program} &\to \textit{pragma}\ \texttt{int main ( )}\ \{\ \textit{decls stmts}\ \} \\
\textit{pragma} &\to \texttt{\#pragma addr}\ \textit{NUM} \mid \varepsilon \\
\textit{decls} &\to \textit{decl decls} \mid \varepsilon \\
\textit{decl} &\to \texttt{int}\ \textit{ID}\ \texttt{;} \\
\textit{stmts} &\to \textit{stmt stmts} \mid \varepsilon \\
\textit{stmt} &\to \textit{ID}\ \texttt{=}\ \textit{expr}\ \texttt{;} \\
&\mid \texttt{while (}\textit{cond}\texttt{)}\ \{\ \textit{stmts}\ \} \\
\textit{expr} &\to \textit{expr}\ \texttt{+}\ \textit{term} \\
&\mid \textit{expr}\ \texttt{-}\ \textit{term} \\
&\mid \textit{term} \\
\textit{term} &\to \textit{ID} \mid \textit{NUM} \\
\textit{cond} &\to \textit{expr comp expr} \\
\textit{comp} &\to \texttt{>} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{<=} \mid \texttt{==} \mid \texttt{!=}
\end{aligned}
$$

The grammar, as is the case with most programming languages, is a Context-Free Grammar (CFG), which allows us to define the syntactic structure of valid programs in a precise and hierarchical manner. This formalism enables the separation of syntax from semantics and facilitates the implementation of a recursive descent parser, which is well-suited for our compiler design. But now that it is defined, we need to corroborate that it is an LL(1) grammar. For this, we have to:

- Eliminate any left recursion that may be present in the grammar.
- Perform left factoring to ensure that each parsing decision can be made using a single lookahead token.
- Compute the *FIRST* and *FOLLOW* sets for each non-terminal in the grammar.
- Construct the LL(1) parsing table based on these sets.
- Verify that there are no conflicts (i.e., each cell in the table contains at most one production), which confirms that the grammar is LL(1).

3.2.1 Eliminate Left Recursion

Now that we know all the necessary steps to obtain a suitable grammar for our compiler, it is time to determine whether the proposed grammar satisfies the conditions to be parsed by an LL(1) parser. One of the key requirements is the absence of left recursion, which can lead to infinite recursion in top-down parsers like recursive descent.

We begin by analyzing each production rule of our grammar to detect potential left recursion:

- **program** → `pragma int main ( ) { decls stmts }`
  This rule does not reference itself. **No left recursion**.

- **pragma** → `#pragma addr NUM`
  This rule begins with terminals, and does not reference itself. **No left recursion**.

- **decls** → **decl decls** | ε
  This production refers to itself but only after a different non-terminal (`decl`). Since the recursion is not immediate (not in the first position), **no direct left recursion**.

- **decl** → `int ID ;`
  Simple terminal-based declaration. **No recursion**.

- **stmts** → **stmt stmts** | ε
  Similar to `decls`, the self-reference occurs after another symbol. **No direct left recursion**.

- **stmt** → **ID = expr ;** | **print(expr);** | **while(cond){stmts}**
  All alternatives begin with terminals or other non-terminals; none reference `stmt`. **No left recursion**.

- **expr** → **expr + term** | **expr - term** | **term**
  This rule clearly exhibits **direct left recursion** in the first two alternatives (`expr` appears as the first symbol on the right-hand side).

- **term** → **ID** | **NUM**
  Only terminals. **No recursion**.

- **cond** → **expr comp expr**
  No self-reference to `cond`. **No recursion**.

- **comp** → **>** | **<** | **==** | **!=** | **>=** | **<=**
  Contains only terminals. **No recursion**.

From this analysis, we conclude that the only production with direct left recursion is:

$$expr \to expr + term \mid expr - term \mid term$$

This must be transformed to make the grammar suitable for LL(1) parsing.

We eliminate the left recursion using the standard technique, introducing a new non-terminal `expr'`:

$$expr \to term\ expr'$$
$$expr' \to +\ term\ expr' \mid -\ term\ expr' \mid \varepsilon$$

This transformation ensures that all productions now start with a terminal or a different non-terminal, making the grammar compatible with a top-down LL(1) parser. In the next sections, we proceed with left factoring (if needed), and compute the `FIRST` and `FOLLOW` sets to confirm LL(1) compliance.

3.2.2 Left Factoring

After eliminating left recursion, the next step is to verify whether the grammar requires left factoring. Left factoring is necessary when two or more productions for the same non-terminal begin with the same prefix, making it ambiguous for a predictive parser to choose which rule to apply based solely on the next input token.

We now analyze each non-terminal in the grammar to detect such ambiguity:

- **program → pragma int main ( ) { decls stmts }**
  This rule starts with the optional non-terminal `pragma`, followed by a fixed sequence of terminals and non-terminals. Since there is only one production and no common prefixes to factor, **no left factoring is needed**.

- **pragma →** `#pragma addr NUM` | ε
  Although this rule has two alternatives, one is ε and the other starts with the terminal `#pragma`, so there is no ambiguity or shared prefix to factor out. **No left factoring needed**.

- **decls → decl decls** | ε
  The first alternative begins with `decl`, while the second is ε. Since they are distinct (and ε is only selected when there is no input matching `decl`), **no ambiguity arises**. **No factoring needed**.

- **decl →** `int ID ;`
  Single production. **No factoring needed**.

- **stmts → stmt stmts** | ε
  Same pattern as `decls`. **No factoring needed**.

- **stmt → ID = expr ; | while (cond) { stmts }**
  Each alternative begins with a distinct terminal: `ID`, `print`, and `while`. Thus, **no ambiguity** in prediction. **No factoring needed**.

- **expr** and **expr'**: Already refactored in the previous section.

- **term → ID | NUM**
  The alternatives begin with different terminal symbols. **No factoring needed**.

- **cond → expr comp expr**
  Only one production. **No factoring needed**.

- **comp → > | < | == | != | >= | <=**
  These all start with different terminal symbols, and are distinguishable lexically. **No factoring needed**.

After thorough inspection, we conclude that **no additional left factoring is necessary**. The grammar, as it stands (after eliminating left recursion), is appropriately structured for LL(1) parsing in terms of deterministic rule selection.

3.2.3 Calculation of FIRST and FOLLOW Sets

Once the grammar has been transformed to eliminate left recursion and left factoring has been applied, the next step is to compute the **FIRST** and **FOLLOW** sets for each non-terminal. These sets are essential for constructing the LL(1) parsing table and ensuring that the grammar is suitable for predictive parsing.

Definition:

- **FIRST(X)**: The set of terminals that begin the strings derivable from the symbol X.
- **FOLLOW(A)**: The set of terminals that can appear immediately to the right of the non-terminal A in some sentential form.

We compute FIRST and FOLLOW sets for the non-terminals of our grammar:

Grammar (after eliminating left recursion in `expr`):

$$\begin{aligned}
program &\rightarrow \text{ pragma int main ( ) \{ decls stmts \}} \\
pragma &\rightarrow \text{\#pragma addr NUM} \mid \varepsilon \\
decls &\rightarrow decl\ decls \mid \varepsilon \\
decl &\rightarrow \text{int ID ;} \\
stmts &\rightarrow stmt\ stmts \mid \varepsilon \\
stmt &\rightarrow ID = expr; \mid while(cond)\{stmts\} \\
expr &\rightarrow term\ expr' \\
expr' &\rightarrow +\ term\ expr' \mid -\ term\ expr' \mid \varepsilon \\
term &\rightarrow ID \mid NUM \\
cond &\rightarrow expr\ comp\ expr \\
comp &\rightarrow > | < | == | != | >= | <=
\end{aligned}$$

Computing FIRST sets:

- **FIRST(program)** = { `#pragma addr`, `int` }
  Since `program` starts with the non-terminal `pragma`, and `FIRST(pragma)` = { `#pragma addr`, ε }, we include both `#pragma addr` and `int` (because of the possibility that `pragma` $\Rightarrow \varepsilon$).
- **FIRST(pragma)** = { `#pragma addr`, ε }
  Because `pragma` can either produce the directive starting with `#pragma` or be empty.
- **FIRST(decls)** = FIRST(decl) $\cup$ {ε}
  FIRST(decl) = { `int` }, and since ε is possible, include ε.
  Therefore, FIRST(decls) = { `int`, ε }.
- **FIRST(decl)** = { `int` }.
- **FIRST(stmts)** = FIRST(stmt) $\cup$ {ε}
  We need FIRST(stmt):
- **FIRST(stmt)** = { ID, `while` }
  Because the alternatives start with these terminals.

  Thus, FIRST(stmts) = { ID, `while`, ε }.
- **FIRST(expr)** = FIRST(term)
- **FIRST(expr')** = { +, -, ε }
- **FIRST(term)** = { ID, NUM }

- **FIRST(cond)** = FIRST(expr) = { ID, NUM }

- **FIRST(comp)** = { >, <, ==, !=, >=, <= }

Computing FOLLOW sets:

The FOLLOW set of the start symbol (program) always includes the end-of-input marker `$`.

- **FOLLOW(program)** = { \$ }
  Since `program` is the start symbol, its FOLLOW set contains only the end-of-input marker.

- **FOLLOW(pragma)** = { `int` }
  Because in the production `program → pragma int ...`, the token `int` immediately follows `pragma`.

- **FOLLOW(decls)** = FIRST(stmts) without ε
  FOLLOW(decls) = { ID, `while` }, and if stmts its ε then FOLLOW(decls) = {}}.

- **FOLLOW(decl)** = FOLLOW(decls) ∪ FIRST(decls) if ε in FIRST(decls)
  decls → decl decls, so decl is followed by decls. Thus,
  FOLLOW(decl) = FIRST(decls) = { `int`, ε }, but since ε is in FIRST(decls), we also add FOLLOW(decls).
  Therefore, FOLLOW(decl) = { `int`, ID, `while` }.

- **FOLLOW(stmts)** = { } (look at program rule)
  program → 'int' 'main' '(' ')' '' decls stmts ''
  After stmts is '}', so FOLLOW(stmts) = { '}' }

- **FOLLOW(stmt)** = FOLLOW(stmts) ∪ FIRST(stmts) if ε in FIRST(stmts)
  stmts → stmt stmts
  So stmt is followed by stmts. Because FIRST(stmts) includes ε, also add FOLLOW(stmts).
  FOLLOW(stmt) = FIRST(stmts) without ε ∪ FOLLOW(stmts)
  = { ID, `while` } ∪ { '}' }
  = { ID, `while`, '}' }

- **FOLLOW(expr)**: expr appears in `stmt`, `cond`, etc.
  After expr, in most cases comes either ';', ')' or operators. For example:
  - stmt → ID = expr ; FOLLOW(expr) includes `;`
  - FOLLOW(expr) includes `)`
  - cond → expr comp expr FOLLOW(expr) includes FIRST(comp) and also what follows cond in the grammar.

  Thus, FOLLOW(expr) = { ;,), comp operators }

- **FOLLOW(expr')** = FOLLOW(expr), since expr' follows expr.

- **FOLLOW(term)**: appears in expr and expr', so FOLLOW(term) = FIRST(expr') without ε union FOLLOW(expr').
  FIRST(expr') = { +, -, ε }, so FOLLOW(term) = { +, -, } ∪ FOLLOW(expr').

- **FOLLOW(cond)**: appears in stmt → while (cond) stmts
  After cond comes ')', so FOLLOW(cond) = {) }

- **FOLLOW(comp)**: appears in cond → expr comp expr
  Followed by expr, so FOLLOW(comp) = FIRST(expr) = { ID, NUM }

In summary we have the next table:

No-terminal	FIRST	FOLLOW
program	{ `#pragma addr`, `int` }	{ $ }
pragma	{ `#pragma addr`, ϵ }	{ int }
decls	{ `int`, ε }	{ ID, `while`, } }
decl	{ `int` }	{ `int`, ID, `while` }
stmts	{ ID, `while`, ε }	{ } }
stmt	{ ID, `while` }	{ ID, `while`, } }
expr	{ ID, NUM }	{ ;,), >, <, ==, !=, >=, <= }
expr'	{ +, -, ε }	{ ;,), >, <, ==, !=, >=, <= }
term	{ ID, NUM }	{ +, -, ;,), >, <, ==, !=, >=, <= }
cond	{ ID, NUM }	{) }
comp	{>,<, ==, !=, >=, <= }	{ ID, NUM }

Table 3.1: First and Follow sets

And still having in mind the production rules:

$$
\begin{aligned}
program &\rightarrow \text{ pragma int main ( ) \{ decls stmts \}} \\
pragma &\rightarrow \text{\#pragma addr NUM} \mid \varepsilon \\
decls &\rightarrow decl\ decls \mid \varepsilon \\
decl &\rightarrow \text{int ID ;} \\
stmts &\rightarrow stmt\ stmts \mid \varepsilon \\
stmt &\rightarrow ID = expr; \mid while(cond)\{stmts\} \\
expr &\rightarrow term\ expr' \\
expr' &\rightarrow +\ term\ expr' \mid -\ term\ expr' \mid \varepsilon \\
term &\rightarrow ID \mid NUM \\
cond &\rightarrow expr\ comp\ expr \\
comp &\rightarrow > \mid < \mid == \mid != \mid >= \mid <=
\end{aligned}
$$

With these sets, we can construct the predictive parsing table and verify that for each non-terminal and input token, there is at most one production rule to apply, ensuring the grammar is LL(1).

Non-terminal	#pragma addr	int	ID	while	}	;	NUM	+	-	(	)	> || >=	< || <=	==	!=	$
program	`program → pragma int main ( ) { decls stmts }`	`program → pragma int main ( ) { decls stmts }`														
pragma	`pragma → #pragma addr NUM`	`pragma →` ε														
decls		`decls → decl decls`	`decls →` ε	`decls →` ε	`decls →` ε											
decl		`decl → int ID ;`														
stmts			`stmts → stmt stmts`	`stmts → stmt stmts`	`stmts →` ε											
stmt			`stmt → ID = expr ;`	`stmt → while ( cond ) { stmts }`												
expr			`expr → term expr'`				`expr → term expr'`									
expr'						`expr' →` ε		`expr' → + term expr'`	`expr' → - term expr'`		`expr' →` ε	`expr' →` ε	`expr' →` ε	`expr' →` ε	`expr' →` ε	
term			`term → ID`				`term → NUM`									
cond			`cond → expr comp expr`				`cond → expr comp expr`									
comp												**comp** → > | < | == | != | >= | <=	**comp** → > | < | == | != | >= | <=	**comp** → > | < | == | != | >= | <=	**comp** → > | < | == | != | >= | <=	

Table 3.2: LL(1) Parsing Table for MiniC Grammar — Horizontal Format (Corrected)

As we can see, the parsing table does not contain multiple entries per cell, confirming that the grammar is LL(1).

3.3 Parser

Now that we know all the implications of the grammar, we can describe how this is implemented through a Parser LL(1).

The parser is a recursive-descent LL(1) parser, which means that it processes the input tokens from left to right (L), constructs a leftmost derivation (L), and uses 1-token lookahead (1) to make parsing decisions. The parser expects the source program to follow a specific structure: declarations followed by statements, all enclosed within a `main` function.

The top-level production handled by the parser is the `Program`, which matches the pattern:

```
pragma
int main() {
    decls
    stmts
}
```

Note that the parser makes certain assumptions regarding the `pragma` directive. According to the grammar, this directive is used to define the base memory address where variables will be stored sequentially in the Z80 architecture. However, the directive is optional. If it resolves to ε, the parser will default the base address to `6000h`.
As we see the parser methods follow the grammar rules closely. For instance, `parseDecls` repeatedly parses variable declarations as long as it sees the `int` keyword. Similarly, `parseStmts` parses a list of statements, which can be assignments or while-loops.

Assignments are parsed by detecting an identifier, followed by the assignment symbol `=`, then an expression, and finally a semicolon:

```
ID = expr;
```

While-loops are recognized with the pattern:

```
while (cond) {
    stmts
}
```

Expressions support binary addition and subtraction, and are parsed recursively using left-associative rules:

```
expr -> term expr'
expr' -> + term expr' | - term expr' |
term -> ID | NUM
```

This structure ensures compatibility with LL(1) parsing by avoiding direct left recursion. Although the grammar is now right-recursive, it still allows for correct left-associative interpretation during AST construction.
Conditions for `while` statements use binary comparisons between expressions, such as `==`, `!=`, `<`, `>`, `<=`, and `>=`.

Each syntactic structure maps directly to a class in the Abstract Syntax Tree (AST), ensuring that the parsed program can be represented in a well-structured and navigable object model. This object model can later be used for code generation.

The LL(1) structure of the parser ensures that parsing decisions can be made deterministically using only one token of lookahead, resulting in a simple and efficient parser implementation.

AST Construction Example

To illustrate the parsing process and AST construction, consider the following source code:

```
#pragma addr 4000;
int main() {
    int x;
    int y;
    int z;
    int m;
    m = 0;
    x = 5;
    y = 7;
    z = x + y;
    while (z > 6) {
        m = m + 5;
        z = z - 2;
    }
}
```

The parser proceeds as follows:

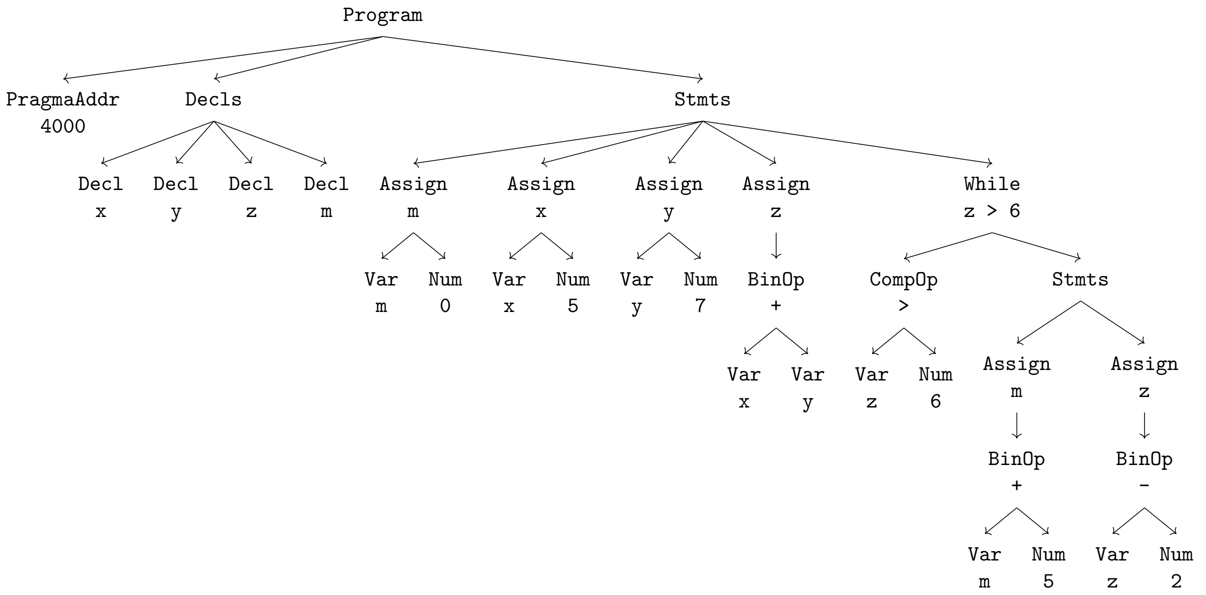
- It detects the `#pragma addr 4000;` directive and creates a `PragmaAddr` node with the address `0x4000`.
- It recognizes the `int main()` function declaration and enters the main program block.
- It parses the variable declarations: `x`, `y`, `z`, and `m`, creating a list of `Decl` nodes.
- It processes the assignment statements:
  - `m = 0;` creates an `Assign` node with `Var(m)` and `Num(0)`.
  - `x = 5;` creates an `Assign` node with `Var(x)` and `Num(5)`.
  - `y = 7;` creates an `Assign` node with `Var(y)` and `Num(7)`.
  - `z = x + y;` creates an `Assign` node with `Var(z)` and a `BinOp` node representing the addition of `Var(x)` and `Var(y)`.
- It parses the `while` loop with condition `z > 6`:
  - The condition is represented by a `Cond` node with the relational operator `'>'`, left expression `Var(z)`, and right expression `Num(6)`.
  - The body of the loop contains two assignment statements:
    * `m = m + 5;` is an `Assign` node with `Var(m)` and a `BinOp` addition of `Var(m)` and `Num(5)`.

* `z = z - 2;` is an `Assign` node with `Var(z)` and a `BinOp` subtraction of `Var(z)` and `Num(2)`.

- Finally, all these components are assembled into the `Program` node representing the entire source.

AST Visualization

This is easily to see in the next representation.



3.4 Semantic Analysis

Once the Abstract Syntax Tree (AST) has been constructed by the parser, we perform semantic analysis to ensure the program is logically correct and adheres to the language rules beyond syntax. This phase verifies that variables are declared before use, expressions involve compatible types, and constructs like loops and assignments are semantically valid.

Symbol Table

The semantic analyzer maintains a **symbol table**, which maps variable identifiers to their memory address. The symbol table is populated during the analysis of declarations.

```
SymbolTable = {
    "x": "Memory pointer"
}
```

Checks Performed

The semantic analysis performs several checks by walking through the AST:

- **Variable declaration**: Every variable used in an assignment, expression, or statement must have been declared previously.

- **Type correctness**: All expressions and assignments must use compatible types. Since this language supports only integers, any operation must involve integer operands.

- **Valid conditions in loops**: The condition in a `while` loop must be a valid comparison between two expressions of compatible types.

Example Analysis

Consider the following program:

```
int main() {
    int x;
    x = 5 + 3;
}
```

Semantic analysis proceeds as follows:

1. Add `x` to the symbol table as an `int`.
2. Verify that `x = 5 + 3;` uses only integers. Both `5` and `3` are valid integer literals.

No semantic errors are found, so the program passes semantic analysis.

Error Example

```
int main() {
    x = 4;
}
```

This program results in a semantic error:

```
Error: Variable 'x' used before declaration.
```

AST Traversal

To perform semantic checks, we implement a recursive traversal of the AST. This recursive traversal ensures that complex expressions and control structures are decomposed into simpler units, which are translated into low-level instructions in a structured and predictable way. Each node type (`Decl`, `Assign`, `While`, etc.) defines a method that performs semantic validation, possibly updating or consulting the symbol table.

Code generation

In the implementation of the compiler, the AST plays a central role in the code generation phase. Each node of the AST corresponds to a syntactic construct in the source language, such as variable declarations, expressions, statements, and control flow structures.

During the code generation, the compiler traverses the AST recursively, using a visitor-like pattern to emit the assembly instructions that implement the semantics of each node.

The Program node is the root of the AST, which includes a list of declarations and statements. The tasks performed in the generation phase are:

- Emit the headers needed to define the CPU (Z80) and the format type.
- Emit a comment to indicate from which memory address the variables are stored.
- Emit code to initialize variables from the defined memory address.
- Traverse and generate code for each statement in the program body.

AST nodes

- Decl Node
  Each Decl node declares a variables identifier, introducing a new entry on the symbols table per variable, managing each entry by incrementing the variable memPointer which starts with the initial direction for variables.

- Assign Node
  It evaluates the right-hand side expression into the A register to then move the result from there to the memory address corresponding to the id according with the symbols table.

- BinOp Node
  When parsing this Expr Node, as it can contain multiple BinOp in each operand, it recursively generate code for the left sub-expression, storing the result in the B register. Then recursively generate the code for the right sub-expression and storing the result in the C register. Finally it moves the right operand to the A register and apply the operator according to the instruction (SUB or ADD) with the A and C registers.

- Num Node
  For the literal integer values a LD instruction is emitted to load the constant into A register directly.

- Var Node
  Similar to Num Node, this Node emits a LD instruction to load a value into A register, but it denotes a reference to a previously declared variable.

- While Node
  The while includes a loop with a condition and a body of statements. First it generates a start and end label, avoiding repetitions with a label counter.
  The start label is emitted, and at the start label the condition is evaluated, and if the condition is false, jumps to the end label. After the conditional jump the body of statements is emitted, to conclude with an unconditional jump to the start label and after that the end label.

- Cond Node
  To represent binary conditions within control structures, the code generation process initially mirrors that of binary arithmetic operations. Both the left-hand and right-hand expressions of the condition are evaluated: the result of the left-hand side is stored in the B register. Then, the right-hand side is evaluated and stored in the A register, which is subsequently compared with the B register to trigger the processor flags based on the result of the comparison. This conditional jumps generated may seem inverted at first glance, but it is to work in junction with the While node and it way to handle the conditional jumps, that only occurs when the condition is not satisfied and it exits the cycle.

  Following this comparison, the compiler determines which jump instruction to emit depending on the specific relational operator (==, !=, >, <). For the == operator, it only checks if the *No Zero* flag is raised; alternatively, for !=, it only checks if the *Zero* flag is raised. Then, for >, the *No Carry and Zero* flags are checked, being replaced the *No Carry* for the *Carry* flag in the case of the < operator. In the case of >=, both *Zero* and *Carry* flags are examined: if either is set, execution continues to a generated label; otherwise, a jump to the given label is performed. This generated label allows branching to bypass the jump when the condition is satisfied. Conversely, for <=, only the *Carry* flag is evaluated indicating a jump when the left operand is strictly greater than the right one.

4 Code Integration with Z80 Assembler

4.1 Motivation for Z80 Assembly Integration

We developed our Z80 assembler as the final project for the "Estructura y Programación de Computadoras" course one year ago. While working on this compiler project, we couldn't help but notice the similarities between certain components of the compiler and the assembler, which is expected considering that assembling is a stage that follows or is integrated into the compilation process.

When the opportunity to include an assembler as an optional extension for this project was presented, we decided to incorporate our previous work and adapt our compiler to generate code compatible with it. This integration was feasible since our compiler was already capable of producing assembly-like output, requiring only specific adjustments to align it with the Z80 instruction set and format.

4.2 Modifications to the Compiler Backend

To integrate our compiler with the Z80 assembler, we had to make several adjustments to the backend, primarily focused on simplifying the output and adapting it to the instruction set and conventions of the Z80 architecture.

One of the first changes involved removing support for the `printf` function. Unlike modern architectures, the Z80 does not include standard output capabilities by default, and in our use case, variable values are inspected directly in memory. Therefore, the compiler no longer generates any output-related instructions.

We also introduced one new keyword: `#pragma addr`, which allow the user to specify the memory addresses where variables will be stored. This addition provides fine control over memory layout, which is crucial in systems with limited and fixed memory regions like the Z80. For example, at the beginning of a program, one may write:

```
#pragma addr 5000;
```

This directive tells the compiler to begin assigning variables starting at address `5000H`. The backend keeps track of this base address and ensures that each declared variable is allocated in a unique memory location within the available memory space.

Finally, we adapted the code generation step to produce valid Z80 assembly instructions. Although similar in structure to the NASM x86 assembly we initially used as a reference, the Z80 has a different syntax and instruction set. For instance, instructions like `MOV` or `INT` from x86 are replaced by Z80-specific equivalents such as `LD` and `JP`. We revised all code templates in the backend to reflect these changes and ensure compatibility with our existing Z80 assembler.

4.3 Structure of the Generated Assembly Code

The assembly code generated by our compiler follows a well-defined structure, ensuring consistency and compatibility with our existing Z80 assembler.

At the beginning of the output file, the compiler includes configuration directives required by the assembler, such as specifying the CPU description file and the header output format. This section is followed by a comment indicating the starting address for variable storage, as shown below:

```
CPU "Z80.tbl"
HOF "INT8"

; Variables assigned starting at 6000h
```

The address shown in the comment is determined by the `#pragma addr` directive in the original C code. The compiler uses this address as the starting point to assign memory to each declared variable, storing them in contiguous memory locations.

After this initial configuration, the file contains the translated Z80 assembly instructions that correspond to the operations defined in the input C program. Memory addresses used in these instructions are written in hexadecimal format followed by the letter `h`, which is standard notation in Z80 assembly.

Finally, the compiler appends a `HALT` instruction at the end of the program to indicate termination. This instruction ensures that once the final operation has been executed, the CPU halts execution in a controlled manner.

4.4 Linking with the Assembler

The integration between the compiler and the assembler is straightforward and relies on a simple workflow. The process begins by executing the compiler and providing it with a valid input file written in a restricted subset of C, as defined by our grammar and supported constructs.

If the input code is syntactically and semantically correct, the compiler generates a corresponding Z80 assembly file with the same base name as the input, but with the `.asm` extension. If the code contains any errors, the compiler outputs an error message and halts without generating the assembly file.

Once the assembly file is available, we proceed to run the assembler, named `TCI`, which is packaged as a `.jar` application developed in a previous course. The assembler takes the `.asm` file as input and produces two output files: `.HEX`, which contains the machine code in hexadecimal format, and `.LST`, which includes a detailed listing of the assembled code with memory addresses.

It is important to note that the assembler performs no syntactic or semantic analysis on the input file. It assumes the assembly code is valid and simply translates the instructions into machine code. This makes the correctness of the generated assembly entirely dependent on the compiler.

4.5 Test Cases and Examples

To demonstrate the functionality and correctness of our compiler and its integration with the assembler, we present two representative test cases. Each example includes the input C code, the generated Z80 assembly code, and a brief explanation of the behavior and expected result.

4.5.1 Example 1:

This example tests arithmetic operations and control flow through a `while` loop. Four variables are declared and stored contiguously starting at address `6000h`. The loop increments variable `m` and decrements `z` until `z < 6`.

C Input Code:

```
#pragma addr 5000;
int x;
int y;
int z;
int m;
m = 0;
x = 5;
y = 7;
z = x + y;
while (z >= 6) {
    m = m + 5;
    z = z - 2;
}
```

Generated Assembly Code:

```
CPU "Z80.tbl"
HOF "INT8"

; Variables asignadas desde 5000h

START:
    LD A, 0
    LD (5003h), A
    LD A, 5
    LD (5000h), A
    LD A, 7
    LD (5001h), A
    LD A, (5000h)
    LD B, A
    LD A, (5001h)
    LD C, A
    LD A, B
    ADD A, C
    LD (5002h), A
L0:
    LD A, (5002h)
    LD B, A
    LD A, 6
```

```
    CP B
    JP Z, _ok2
    JP C, _ok2
    JP L1
_ok2:
    LD A, (5003h)
    LD B, A
    LD A, 5
    LD C, A
    LD A, B
    ADD A, C
    LD (5003h), A
    LD A, (5002h)
    LD B, A
    LD A, 2
    LD C, A
    LD A, B
    SUB C
    LD (5002h), A
    JP L0
L1:
HALT
```

Once the assembly code is generated, it is passed to the `TCI` assembler application. This tool processes the `.asm` file and produces the corresponding `.HEX` and `.LST` files, which contain the machine code and the listing of instructions with memory addresses, respectively.

HEX File:

```
:100000003E003203503E053200503E073201503A66
:100010000050473A01504F78813202503A0250471F
:100020003E06B8CA2C00DA2C00C347003A035047FA
:100030003E054F78813203503A0250473E024F78D6
:0800400091320250C31C00764E
:00000001FF
```

LST File:

```
0000                  CPU "Z80.TBL"
0000                  HOF "INT8"
0000
0000
0000
0000               START:
0000 3E00             LD A,0
0002 320350           LD (5003H),A
0005 3E05             LD A,5
0007 320050           LD (5000H),A
000A 3E07             LD A,7
000C 320150           LD (5001H),A
```

```
000F 3A0050          LD A,(5000H)
0012 47              LD B,A
0013 3A0150          LD A,(5001H)
0016 4F              LD C,A
0017 78              LD A,B
0018 81              ADD A,C
0019 320250          LD (5002H),A
001C              L0:
001C 3A0250          LD A,(5002H)
001F 47              LD B,A
0020 3E06            LD A,6
0022 B8              CP B
0023 CA2C00          JP Z,_OK2
0026 DA2C00          JP C,_OK2
0029 C34700          JP L1
002C              _ok2:
002C 3A0350          LD A,(5003H)
002F 47              LD B,A
0030 3E05            LD A,5
0032 4F              LD C,A
0033 78              LD A,B
0034 81              ADD A,C
0035 320350          LD (5003H),A
0038 3A0250          LD A,(5002H)
003B 47              LD B,A
003C 3E02            LD A,2
003E 4F              LD C,A
003F 78              LD A,B
0040 91              SUB C
0041 320250          LD (5002H),A
0044 C31C00          JP L0
0047              L1:
0047 76              HALT 0000   END

001C  L0                 0047  L1                      0000  START
002C  _OK2
```

Then we load the `.hex` file onto a simulator, and after execution, we observe the following:

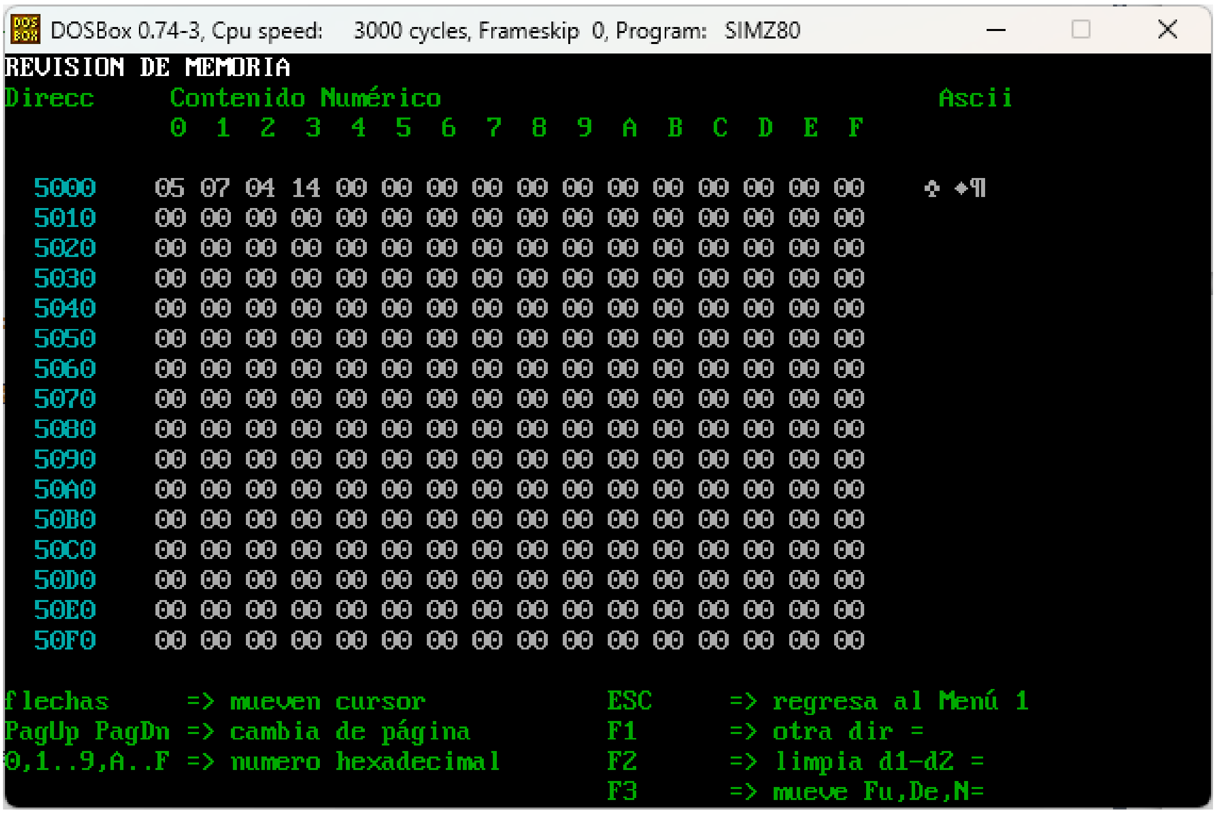


Figure 4.1: Memory in the simulator after executing the program of the example 1.

As expected from the C code, address `5000h` corresponds to variable `x`, which holds the value 5. Variable `y` is located at address `5001h` with the value 7. At address `5002h`, we find variable `z`, storing the result of `x + y` that was also decreased by 2 in each cycle (12-(2*4 iterations) = 4). Finally, variable `m` is located at address `5003h`, holding the value 20 (14 in hexadecimal), which is the result of four iterations of the `while` loop, each adding 5.

4.5.2 Example 2:

For the next example, we compile an almost identical program. However, this time the condition is changed to `z > 6` in order to test that both `>` and `>=` comparisons work as intended. In this case, the loop should execute one less iteration. Additionally, we change the starting address for the variables to `6000h`.

C Input Code:

```
#pragma addr 6000;
int main() {
    int x;
    int y;
    int z;
    int m;
    m = 0;
    x = 5;
```

```
    y = 7;
    z = x + y;
    while (z > 6) {
        m = m + 5;
        z = z - 2;
    }
}
```

Generated Assembly Code:

```
CPU "Z80.tbl"
HOF "INT8"

; Variables asignadas desde 6000h

START:
    LD A, 0
    LD (6003h), A
    LD A, 5
    LD (6000h), A
    LD A, 7
    LD (6001h), A
    LD A, (6000h)
    LD B, A
    LD A, (6001h)
    LD C, A
    LD A, B
    ADD A, C
    LD (6002h), A
L0:
    LD A, (6002h)
    LD B, A
    LD A, 6
    CP B
    JP NC, L1
    JP Z, L1
    LD A, (6003h)
    LD B, A
    LD A, 5
    LD C, A
    LD A, B
    ADD A, C
    LD (6003h), A
    LD A, (6002h)
    LD B, A
    LD A, 2
    LD C, A
    LD A, B
    SUB C
    LD (6002h), A
    JP L0
L1:
HALT
```

The resulting `.asm` file is passed to the `TCI` assembler, which generates the corresponding `.HEX` and `.LST` files.

HEX File:

```
:100000003E003203603E053200603E073201603A36
:10001000060473A01604F78813202603A026047DF
:100020003E06B8D24400CA44003A0360473E054F3A
:1000300078813203603A0260473E024F7891320283
:0500400060C31C007606
:00000001FF
```

LST File:

```
0000                CPU "Z80.TBL"
0000                HOF "INT8"
0000
0000
0000
0000             START:
0000 3E00           LD A,0
0002 320360         LD (6003H),A
0005 3E05           LD A,5
0007 320060         LD (6000H),A
000A 3E07           LD A,7
000C 320160         LD (6001H),A
000F 3A0060         LD A,(6000H)
0012 47             LD B,A
0013 3A0160         LD A,(6001H)
0016 4F             LD C,A
0017 78             LD A,B
0018 81             ADD A,C
0019 320260         LD (6002H),A
001C             L0:
001C 3A0260         LD A,(6002H)
001F 47             LD B,A
0020 3E06           LD A,6
0022 B8             CP B
0023 D24400         JP NC,L1
0026 CA4400         JP Z,L1
0029 3A0360         LD A,(6003H)
002C 47             LD B,A
002D 3E05           LD A,5
002F 4F             LD C,A
0030 78             LD A,B
0031 81             ADD A,C
0032 320360         LD (6003H),A
0035 3A0260         LD A,(6002H)
0038 47             LD B,A
0039 3E02           LD A,2
003B 4F             LD C,A
003C 78             LD A,B
```

```
003D 91             SUB C
003E 320260         LD (6002H),A
0041 C31C00         JP L0
0044              L1:
0044 76             HALT 0000    END

001C   L0                    0044   L1                         0000   START
```

After loading the `.hex` file into a simulator and running the program, we observe the following:

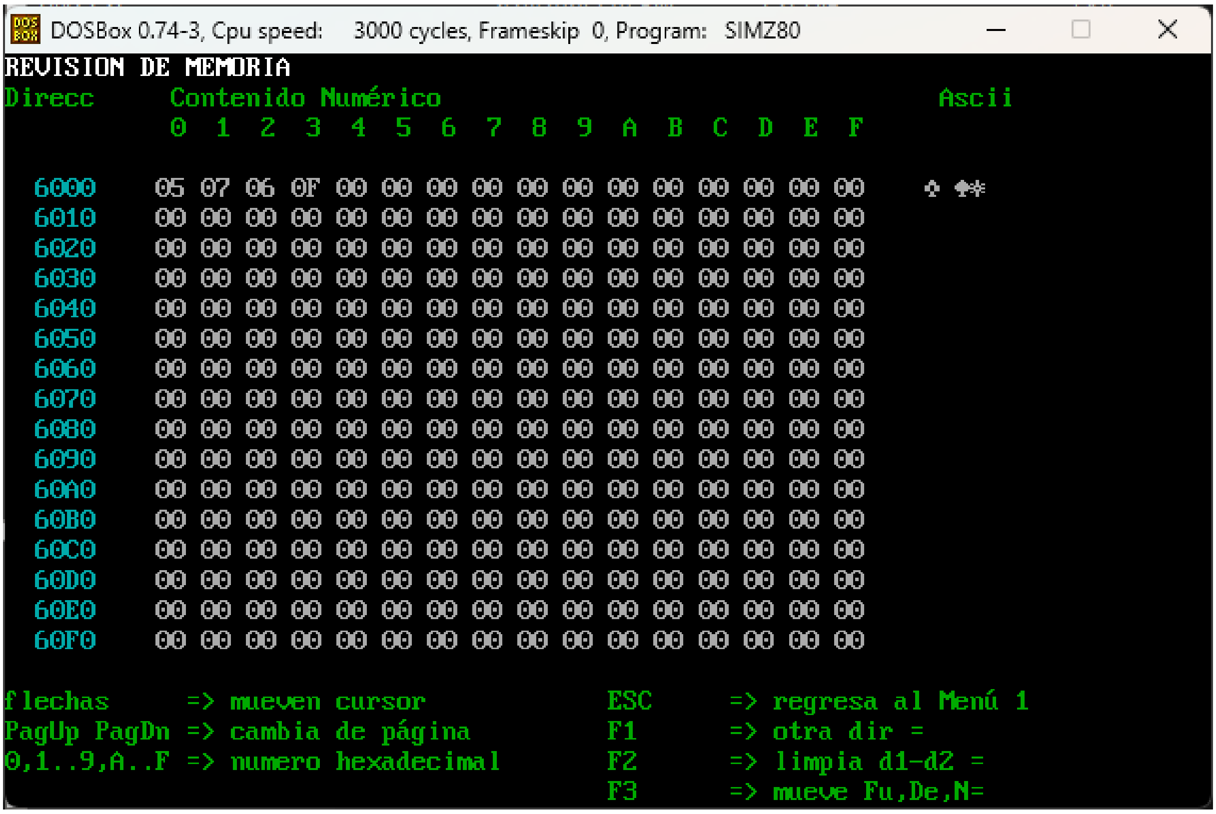


Figure 4.2: Memory in the simulator after executing the program of the example 2.

Address `6000h` holds variable `x` with value 5, and address `6001h` holds variable `y` with value 7. Variable `z`, stored at `6002h`, contains the value `x + y` that was also decreased by 2 in each cycle (12-(2*3 iterations) = 6). Variable `m` is located at `6003h`, now holding the value 15 (0Fh in hexadecimal), since the loop was executed only three times due to the change in condition.

Notes:

- This example is used to verify the correct behavior of the `>` comparison in a `while` loop.
- It is expected that the loop executes one fewer iteration compared to the previous example using `>=`.
- This time variable storage begins at address `6000h`.

4.6 Limitations and Future Work

Our current implementation only supports a limited subset of the C language. Specifically, the compiler is capable of handling variable declarations, basic arithmetic operations such as addition and subtraction, and `while` loops with standard comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`). The syntax supported follows the basic C style, but many features of the language—such as functions, arrays, pointers, or more complex expressions—are not yet implemented.

It is important to note that although our assembler is capable of processing any syntactically correct Z80 assembly code, the compiler's grammar restricts the type of C code that can be translated. This means that any unsupported structure must be written directly in assembly if required.

For future work, we aim to extend the grammar to include additional control structures (such as `if-else` statements), support for functions, and a broader range of operations. Improving error handling and diagnostic messages is also an area of potential enhancement. Additionally, integration with debugging tools or simulation environments could further increase the utility and usability of the project.

5 Results

In this section, we present a series of test cases using the developed compiler and assembler. For each example, we show the input C code, the generated assembly file, the assembling process, and the final simulation result.

5.1 Example 1: Less than case

Input C Code

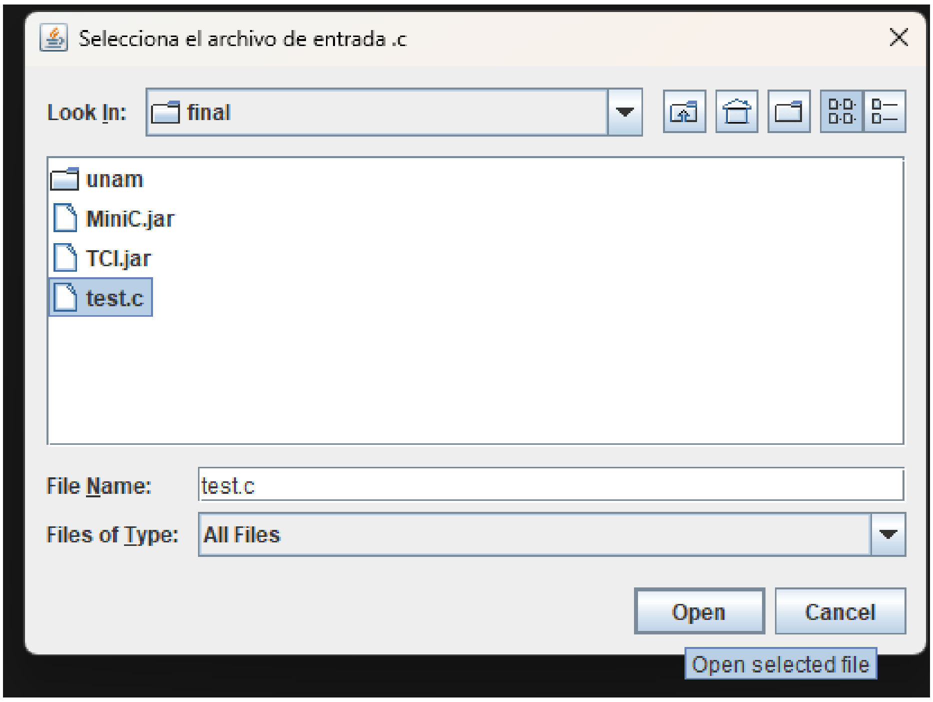
Here we show the input program written in the restricted C-like syntax supported by our compiler.

```c
#pragma addr 3000;
int main() {
    int x;
    int y;
    int z;
    int m;
    m = 0;
    x = 3;
    y = 10;
    z = x + y;
    while (z < 20) {
        m = m + 3;
        z = z + 1;
    }
}
```

Figure 5.1: In this example we test a less than conditional while.

Generated Assembly Code

We use the `.jar` application named "MiniC" to compile the C file.



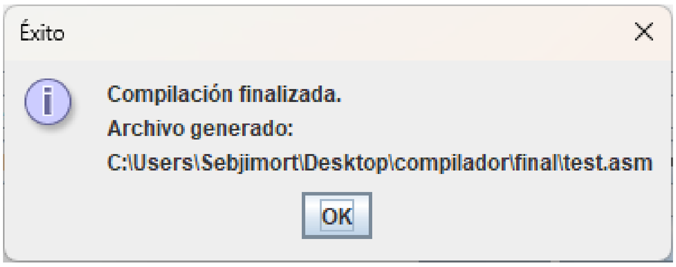


Figure 5.2: Compiling the C file with the MiniC compiler.

After compiling, the corresponding `.asm` file is created.

```
CPU "Z80.tbl"
HOF "INT8"

; Variables asignadas desde 3000h

START:
    LD A, 0
    LD (3003h), A
    LD A, 3
    LD (3000h), A
    LD A, 10
    LD (3001h), A
    LD A, (3000h)
    LD B, A
    LD A, (3001h)
    LD C, A
    LD A, B
    ADD A, C
    LD (3002h), A
L0:
    LD A, (3002h)
    LD B, A
    LD A, 20
    CP B
    JP C, L1
    JP Z, L1
    LD A, (3003h)
    LD B, A
    LD A, 3
    LD C, A
    LD A, B
    ADD A, C
    LD (3003h), A
    LD A, (3002h)
    LD B, A
    LD A, 1
    LD C, A
    LD A, B
    ADD A, C
    LD (3002h), A
    JP L0
L1:
HALT
```

Figure 5.3: .asm file

Assembler Output

We use the `.jar` application named "TCI" to assemble the `.asm` file.

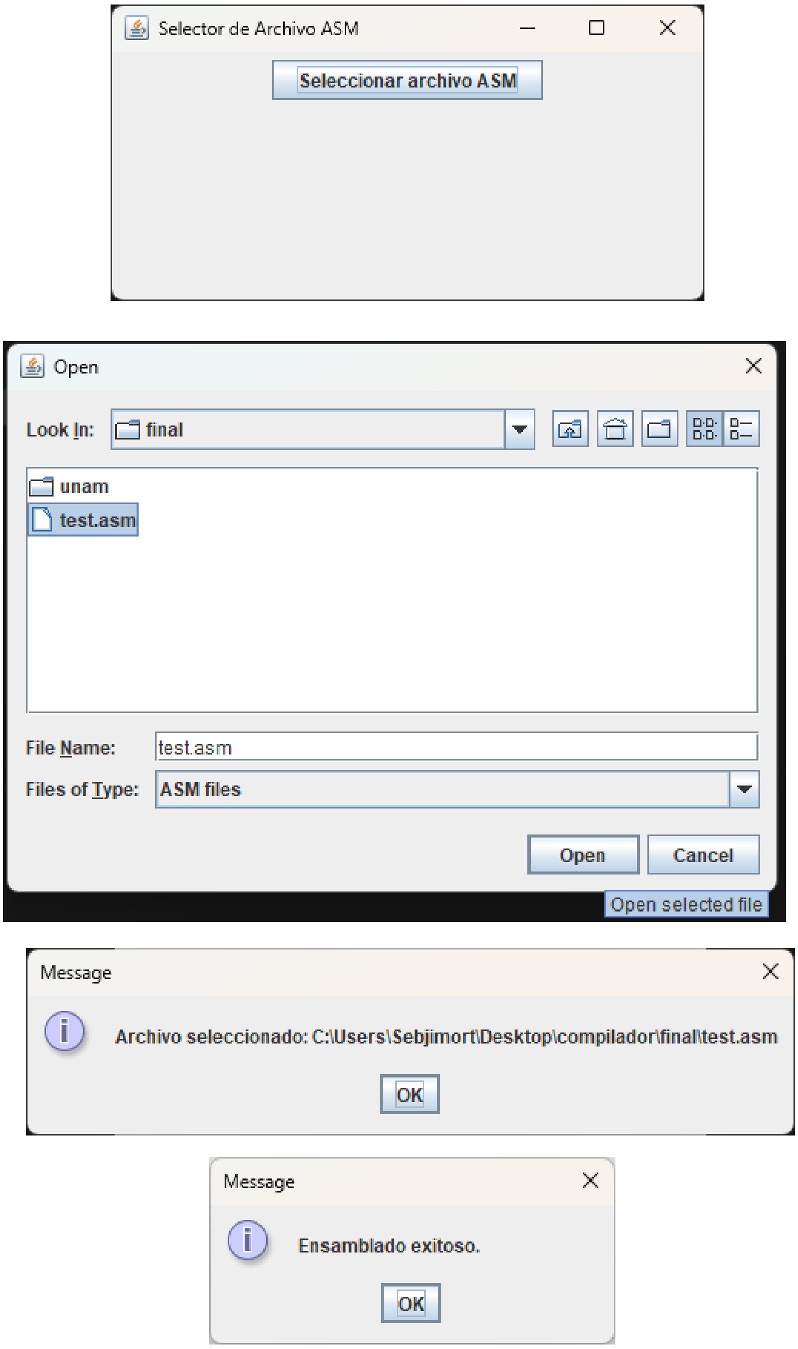


Figure 5.4: Proccess using the TCI assembler.

The assembly file is passed to the assembler, which generates the `.hex` and `.lst` files.

```
:100000003E003203303E033200303E0A3201303AC5
:1000100000030473A01304F78813202303A0230479F
:100020003E14B8DA4400CA44003A0330473E034F56
:1000300007881320330 3A0230473E014F78813202F4
:0500400030C31C007636
:00000001FF
```

Figure 5.5: .hex file

```
0000                 CPU "Z80.TBL"
0000                 HOF "INT8"
0000
0000
0000
0000              START:
0000 3E00            LD A,0
0002 320330          LD (3003H),A
0005 3E03            LD A,3
0007 320030          LD (3000H),A
000A 3E0A            LD A,10
000C 320130          LD (3001H),A
000F 3A0030          LD A,(3000H)
0012 47              LD B,A
0013 3A0130          LD A,(3001H)
0016 4F              LD C,A
0017 78              LD A,B
0018 81              ADD A,C
0019 320230          LD (3002H),A
001C              L0:
001C 3A0230          LD A,(3002H)
001F 47              LD B,A
0020 3E14            LD A,20
0022 B8              CP B
0023 DA4400          JP C,L1
0026 CA4400          JP Z,L1
0029 3A0330          LD A,(3003H)
002C 47              LD B,A
002D 3E03            LD A,3
002F 4F              LD C,A
0030 78              LD A,B
0031 81              ADD A,C
0032 320330          LD (3003H),A
0035 3A0230          LD A,(3002H)
0038 47              LD B,A
0039 3E01            LD A,1
003B 4F              LD C,A
003C 78              LD A,B
003D 81              ADD A,C
003E 320230          LD (3002H),A
0041 C31C00          JP L0
0044              L1:
0044 76              HALT 0000          END
001C  L0                    0044  L1                         0000  START
```

Figure 5.6: .lst file

Simulation Result

The resulting hex file is loaded into the simulator, and we analyze the memory content after execution.

Figure 5.7: The process consists of opening the Z80 simulator, clearing the memory, loading the .hex file, executing the program, and finally navigating to the memory address where the variables were declared.

In the subsequent examples, we include only the essential screenshots to illustrate the results.

5.2 Example 2: Less than or equal case

Input C Code

```
#pragma addr 6000;
int main() {
    int x;
    int y;
    int z;
    int m;
    m = 0;
    x = 3;
    y = 10;
    z = x + y;
    while (z <= 20) {
        m = m + 3;
        z = z + 1;
    }
}
```

Figure 5.8: C like code

Generated Assembly Code

```
CPU "Z80.tbl"
HOF "INT8"

; Variables asignadas desde 6000h

START:
    LD A, 0
    LD (6003h), A
    LD A, 3
    LD (6000h), A
    LD A, 10
    LD (6001h), A
    LD A, (6000h)
    LD B, A
    LD A, (6001h)
    LD C, A
    LD A, B
    ADD A, C
    LD (6002h), A
L0:
    LD A, (6002h)
    LD B, A
    LD A, 20
    CP B
    JP C, L1
    LD A, (6003h)
    LD B, A
    LD A, 3
    LD C, A
    LD A, B
    ADD A, C
    LD (6003h), A
    LD A, (6002h)
    LD B, A
    LD A, 1
    LD C, A
    LD A, B
    ADD A, C
    LD (6002h), A
    JP L0
L1:
HALT

```

Figure 5.9: .asm file

Assembler Output

```
:100000003E003203603E033200603E0A3201603A35
:100010000060473A01604F78813202603A026047DF
:100020003E14B8DA41003A0360473E034F7881320C
:10003000036003A0260473E014F7881320260C31C80
:02004000007648
:00000001FF
```

Figure 5.10: .hex file

```
0000                    CPU "Z80.TBL"
0000                    HOF "INT8"
0000
0000
0000
0000                 START:
0000 3E00               LD A,0
0002 320360             LD (6003H),A
0005 3E03               LD A,3
0007 320060             LD (6000H),A
000A 3E0A               LD A,10
000C 320160             LD (6001H),A
000F 3A0060             LD A,(6000H)
0012 47                 LD B,A
0013 3A0160             LD A,(6001H)
0016 4F                 LD C,A
0017 78                 LD A,B
0018 81                 ADD A,C
0019 320260             LD (6002H),A
001C                 L0:
001C 3A0260             LD A,(6002H)
001F 47                 LD B,A
0020 3E14               LD A,20
0022 B8                 CP B
0023 DA4100             JP C,L1
0026 3A0360             LD A,(6003H)
0029 47                 LD B,A
002A 3E03               LD A,3
002C 4F                 LD C,A
002D 78                 LD A,B
002E 81                 ADD A,C
002F 320360             LD (6003H),A
0032 3A0260             LD A,(6002H)
0035 47                 LD B,A
0036 3E01               LD A,1
0038 4F                 LD C,A
0039 78                 LD A,B
003A 81                 ADD A,C
003B 320260             LD (6002H),A
003E C31C00             JP L0
0041                 L1:
0041 76                 HALT 0000         END
001C  L0                       0041  L1                               0000  START
```

Figure 5.11: .lst file

Simulation Result

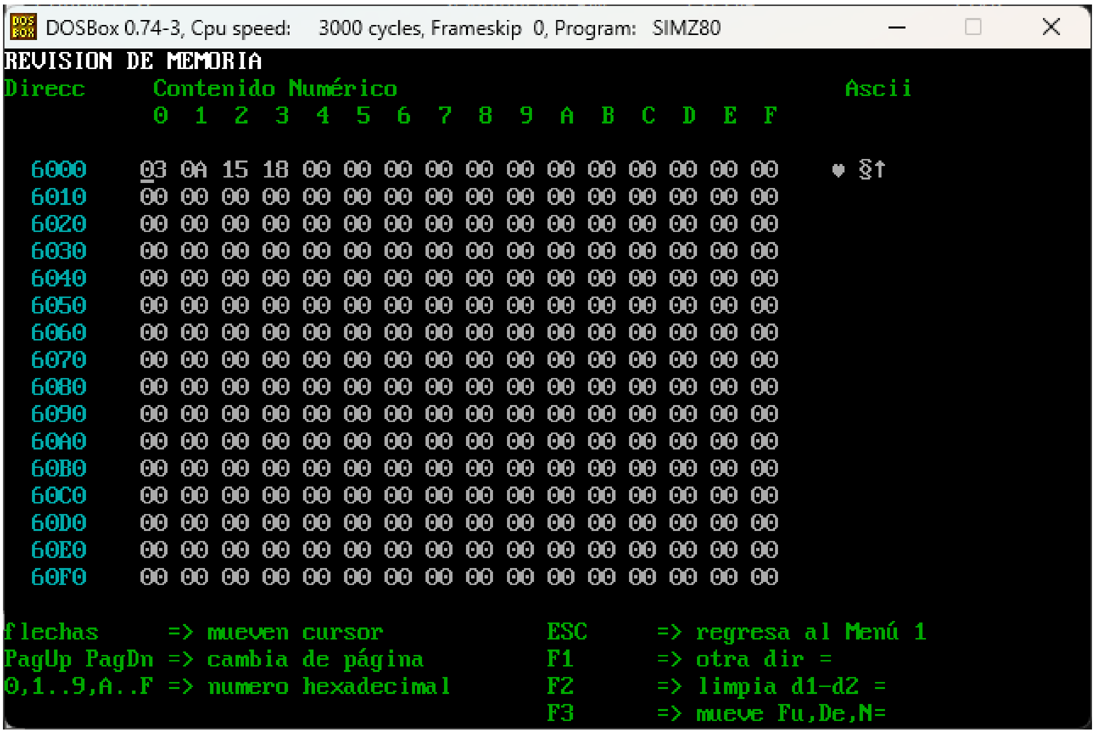


Figure 5.12: Simulator after executing the program.

5.3 Example 3: Equal case

Input C Code

```c
#pragma addr 5000;
int main() {
    int x;
    int y;
    int z;
    int m;
    m = 2;
    x = 20;
    y = 7;
    z = x - y;
    while (z == 13) {
        m = m + 3;
        z = z + 1;
    }
}
```

Figure 5.13: C file

Generated Assembly Code

```
CPU "Z80.tbl"
HOF "INT8"

; Variables asignadas desde 5000h

START:
    LD A, 2
    LD (5003h), A
    LD A, 20
    LD (5000h), A
    LD A, 7
    LD (5001h), A
    LD A, (5000h)
    LD B, A
    LD A, (5001h)
    LD C, A
    LD A, B
    SUB C
    LD (5002h), A
L0:
    LD A, (5002h)
    LD B, A
    LD A, 13
    CP B
    JP NZ, L1
    LD A, (5003h)
    LD B, A
    LD A, 3
    LD C, A
    LD A, B
    ADD A, C
    LD (5003h), A
    LD A, (5002h)
    LD B, A
    LD A, 1
    LD C, A
    LD A, B
    ADD A, C
    LD (5002h), A
    JP L0
L1:
HALT
```

Figure 5.14: .asm file

Assembler Output

```
:100000003E023203503E143200503E073201503A55
:10001000000050473A01504F78913202503A0250470F
:100020003E0DB8C241003A0350473E034F7881323B
:1000300003503A0250473E014F7881320250C31CB0
:02004000007648
:00000001FF
```

Figure 5.15: .hex file

```
0000                    CPU "Z80.TBL"
0000                    HOF "INT8"
0000
0000
0000
0000                START:
0000 3E02               LD A,2
0002 320350             LD (5003H),A
0005 3E14               LD A,20
0007 320050             LD (5000H),A
000A 3E07               LD A,7
000C 320150             LD (5001H),A
000F 3A0050             LD A,(5000H)
0012 47                 LD B,A
0013 3A0150             LD A,(5001H)
0016 4F                 LD C,A
0017 78                 LD A,B
0018 91                 SUB C
0019 320250             LD (5002H),A
001C                L0:
001C 3A0250             LD A,(5002H)
001F 47                 LD B,A
0020 3E0D               LD A,13
0022 B8                 CP B
0023 C24100             JP NZ,L1
0026 3A0350             LD A,(5003H)
0029 47                 LD B,A
002A 3E03               LD A,3
002C 4F                 LD C,A
002D 78                 LD A,B
002E 81                 ADD A,C
002F 320350             LD (5003H),A
0032 3A0250             LD A,(5002H)
0035 47                 LD B,A
0036 3E01               LD A,1
0038 4F                 LD C,A
0039 78                 LD A,B
003A 81                 ADD A,C
003B 320250             LD (5002H),A
003E C31C00             JP L0
0041                L1:
0041 76                 HALT 0000          END
001C  L0                         0041  L1                         0000  START
```

Figure 5.16: .lst file

Simulation Result

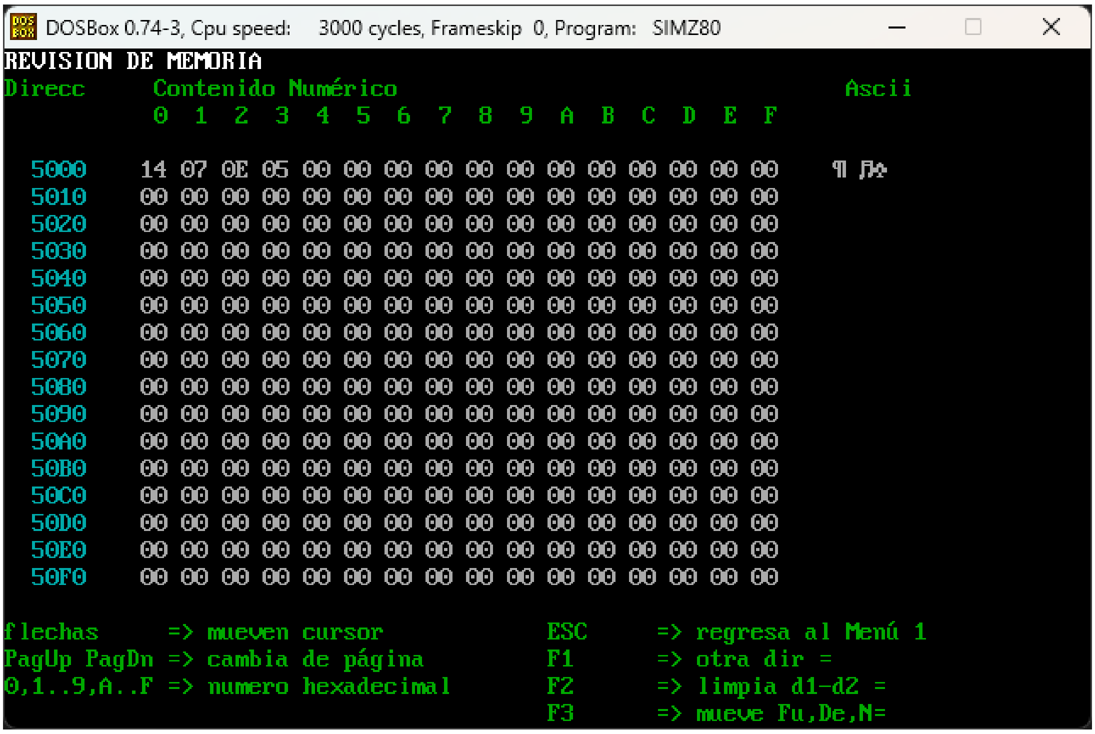


Figure 5.17: Simulator after executing the program.

6 How to Use

To test the compiler and the assembler, there are some programs and files that you will need. These can be found in the GitHub repository of this project: `https://github.com/DOCTT/Compiler-G5-EQ9/tree/main/Z80%20Simulator`

The compiler and the assembler are already packaged in a .jar file, so you can simply double-click it and it will work. It is important to know that it only works with the most recent version of Java (currently Java 24). The compiler lets you choose a file using a simple file chooser, and it will return a .asm file in the same directory. The assembler works the same way, except it returns the binary code (.hex) and the listing file (.LST).

The binary code provided by the assembler is fully executable (if correct), so to test it, you need an x86 environment. We achieve this by using the DOSBox emulator, and we also need a Z80 Simulator (everything can be found in the project repository).

To begin setting up the x86 environment, we have to mount the disk where the simulator and files will be stored. To do this, follow these steps (note that the emulator uses the US keyboard layout):

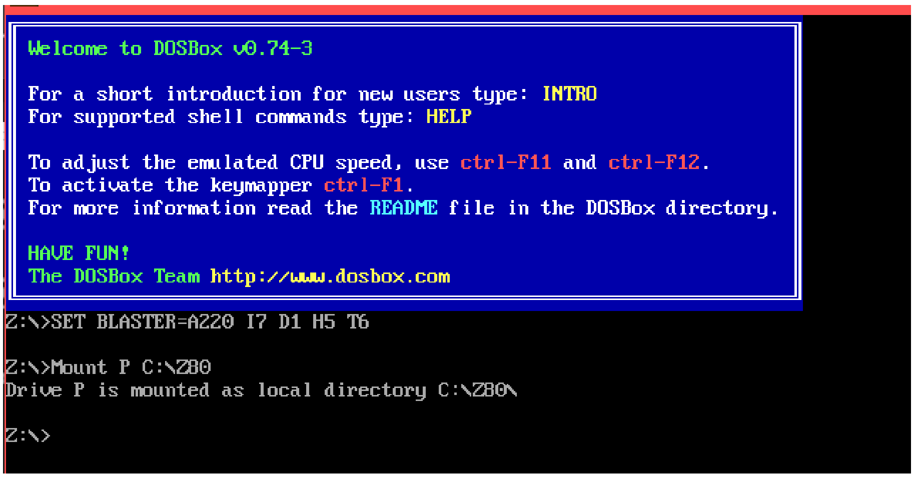


Figure 6.1: Mounting the x86 environment disk

Then we set the mounted disk:

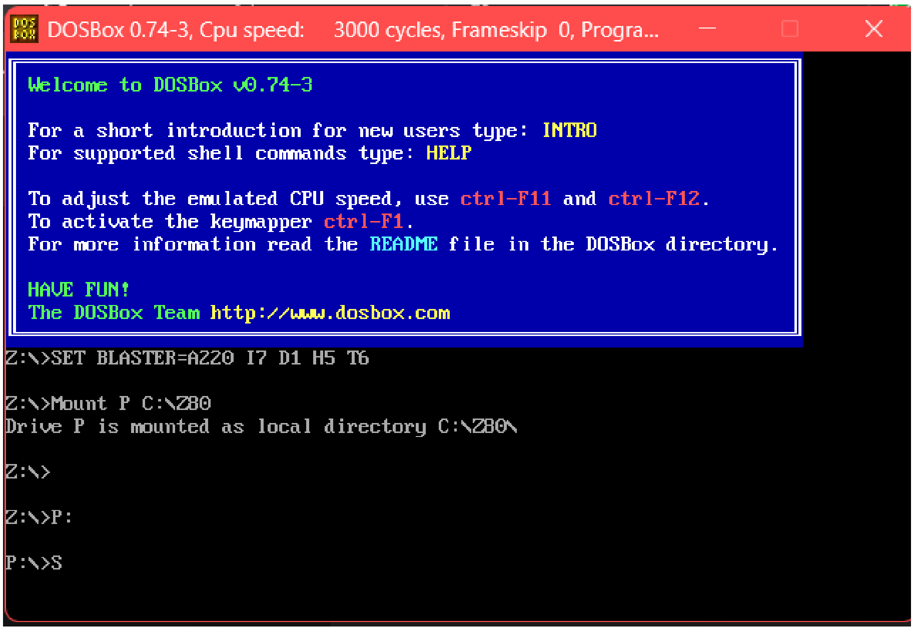


Figure 6.2: Selected disk

Next, move all the necessary files to the mounted disk. These files include the Z80 bytecode, the simulator (SIMZ80.exe), and the Z80 tables (Z80.tbl).

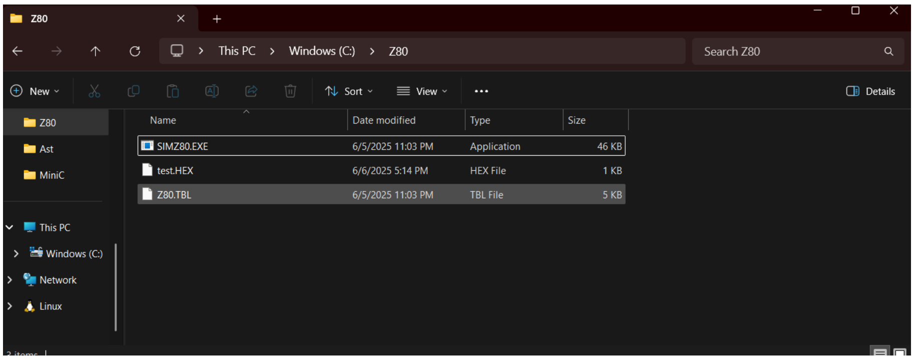


Figure 6.3: Files in the mounted disk

Finally, we can execute the Z80 simulator:

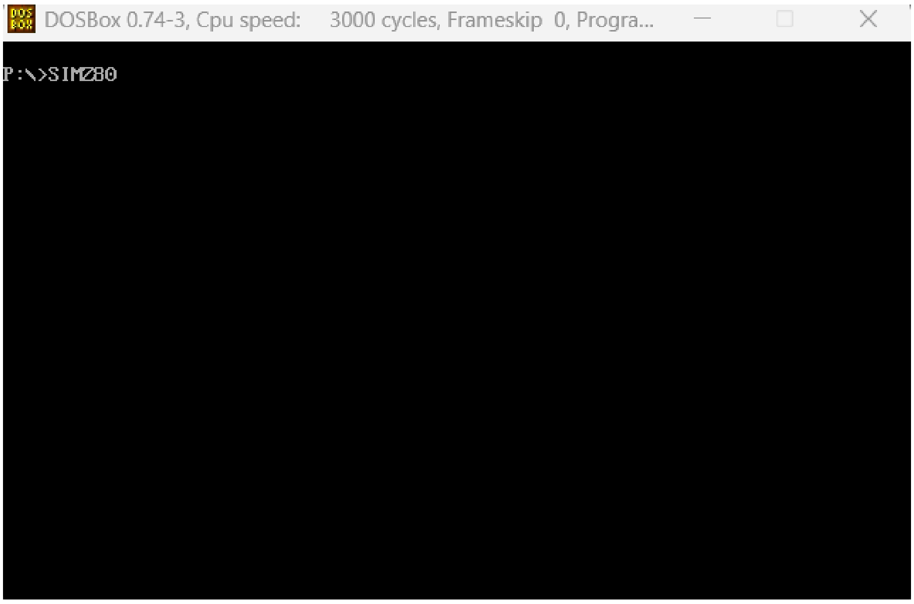


Figure 6.4: Accessing SimZ80 command mode

Figure 6.5: Z80 Simulator main interface

This is a very simple, yet effective simulator. First, we clear all the memory to avoid unexpected results or behaviors.
To do this, we press the letter 'R', which will display the following screen:

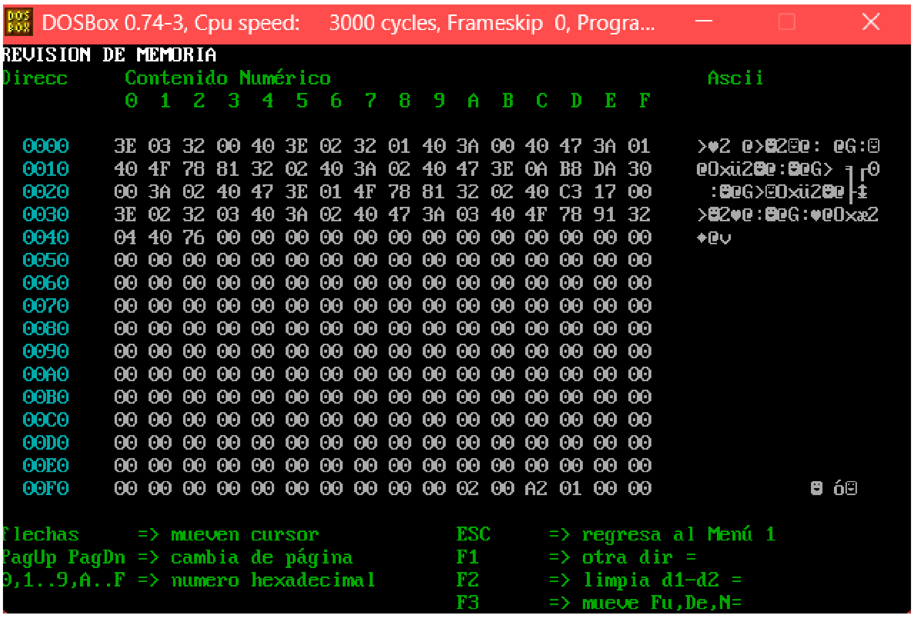


Figure 6.6: Z80 Simulator memory view

It shows the memory and its contents. To clear it, we press F2 and enter the values 0000 and FFFF, which will wipe all the available memory.

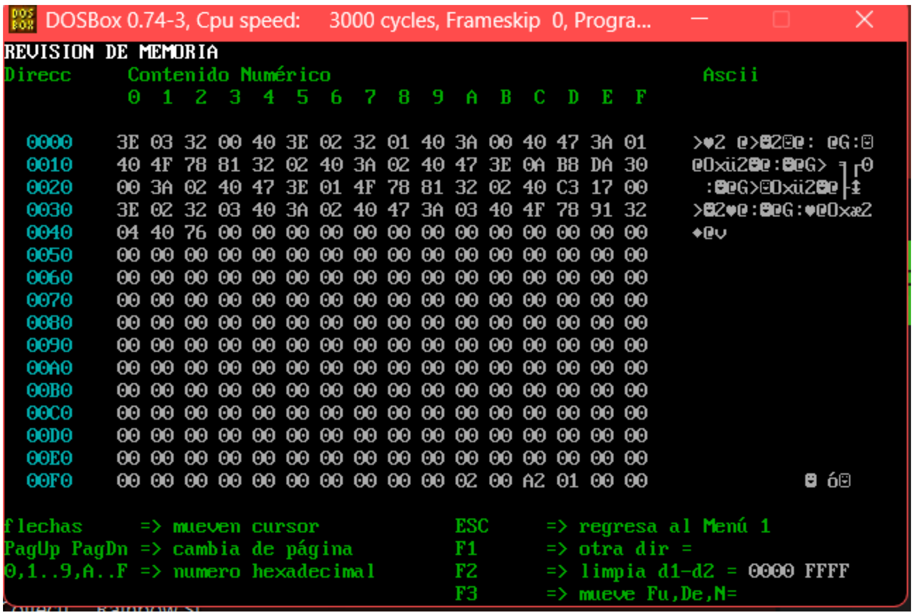


Figure 6.7: Example execution screen

Now the memory is clean: To return to the main menu, simply press ESC.

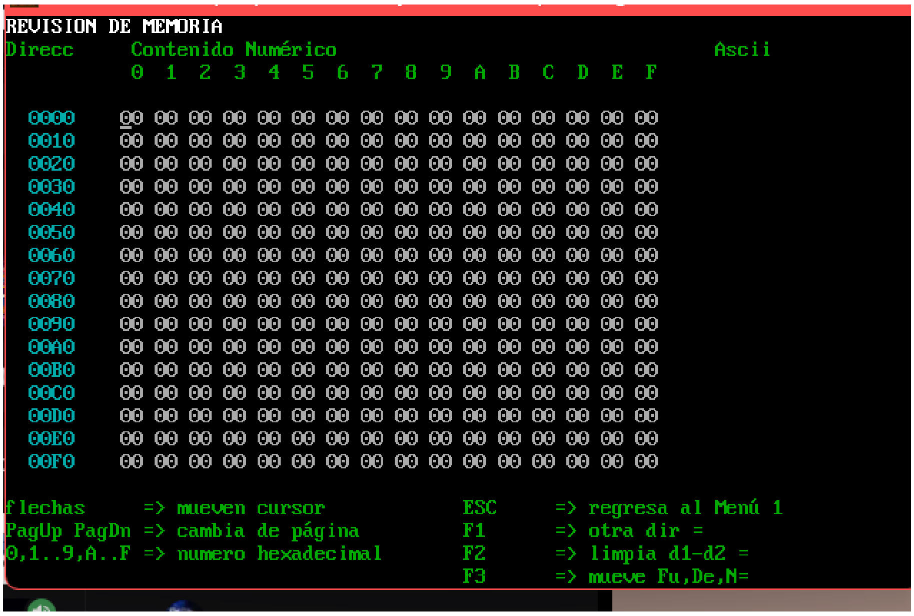


Figure 6.8: Cleared Z80 memory

To load the binary code into memory, press the letter ‘C’, which will display the following screen:

Figure 6.9: Loading the binary code

Then, enter the full name of the binary file (including the file extension), as well as the memory address where you want to load the binary code. Make sure not to load it at the same memory address where your variables will be stored, to avoid unexpected behavior.

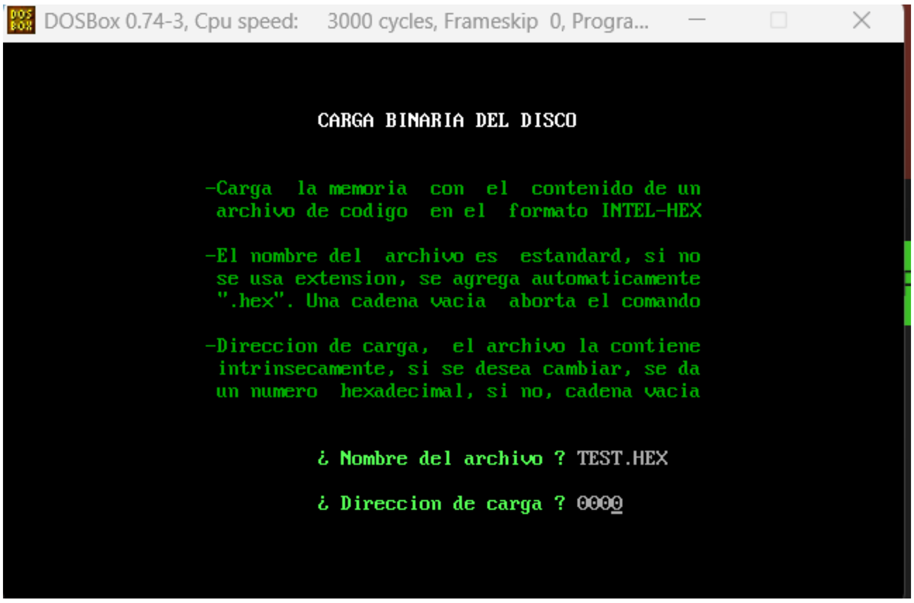


Figure 6.10: Example screen

Now, to execute the code, press 'E'. The following screen will appear:

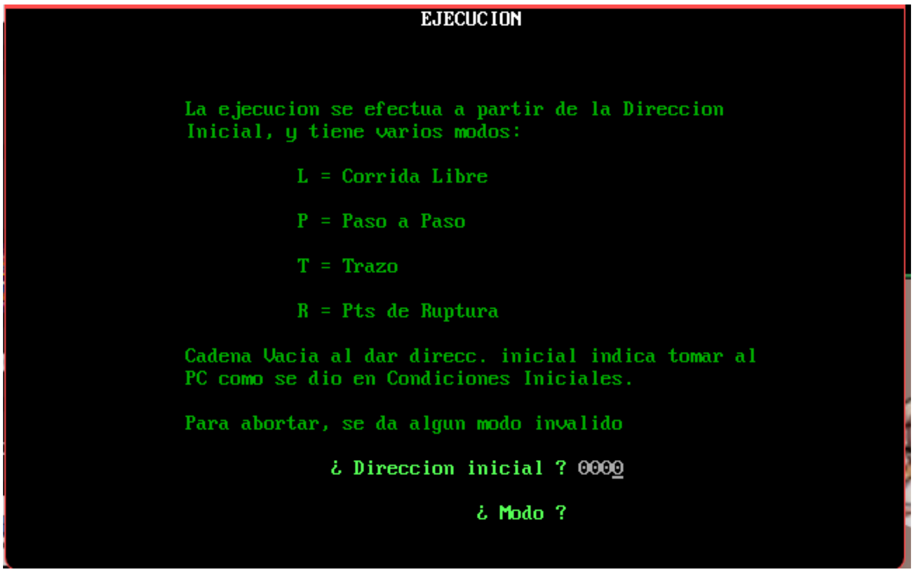


Figure 6.11: Execution screen

Simply enter the memory address where the binary code was loaded, and select T Mode. This mode will display the step-by-step execution of the code and will stop automatically when the halt instruction is encountered.

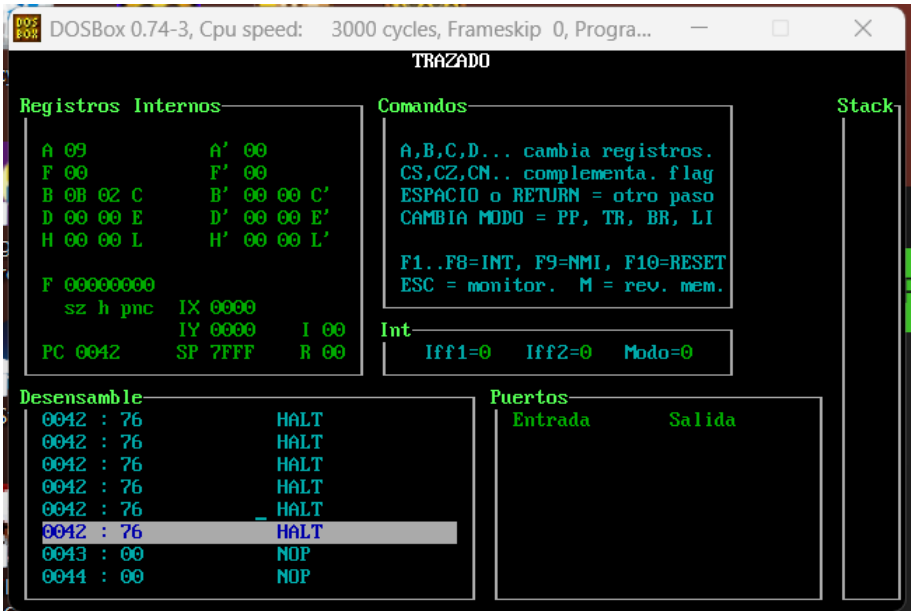


Figure 6.12: Execution process screen

Now, press ‘M’ and then Enter to view the Z80 memory. Next, press F1 and enter the memory address where the variables are stored (by default, this is 6000h, unless a different address was specified using the pragma directive).

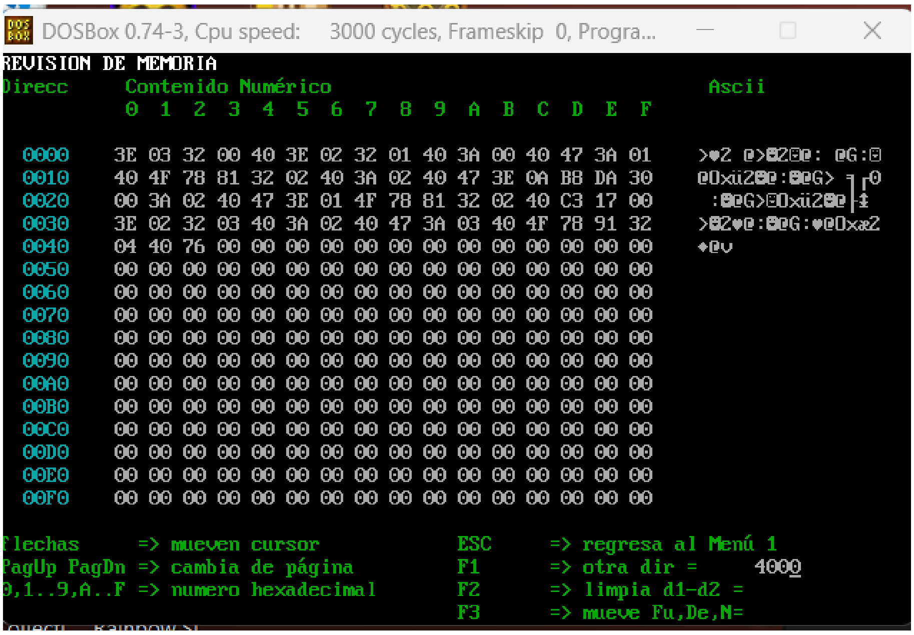


Figure 6.13: Z80 memory view

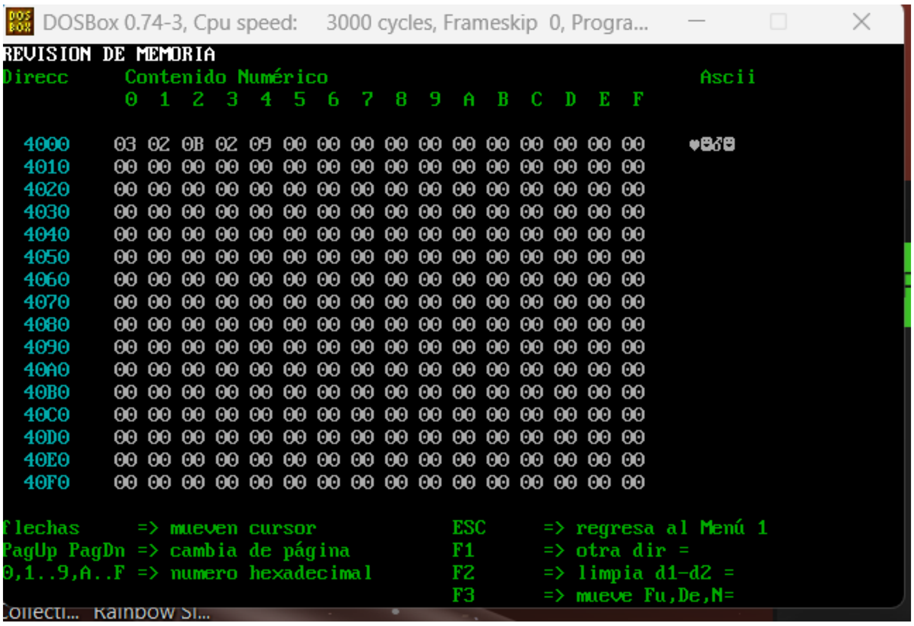


Figure 6.14: Program results

7 Conclusion

The development of the compiler and its integration with a Z80 assembler allowed us to verify the practical application of fundamental compiler construction concepts. Throughout this process, each stage—from lexical analysis to code generation—was directly guided by the theoretical frameworks discussed, particularly context-free grammars, LL(1) parsing, and abstract syntax tree traversal.

The main objective of transforming a subset of C code into valid Z80 assembly instructions was successfully achieved. This was accomplished by defining a grammar capable of handling arithmetic expressions and control flow structures, and by implementing a recursive descent parser that guarantees syntactic validity and facilitates structured code generation.

Furthermore, the implementation of backend features such as memory address mapping using `#pragma addr` and correct translation of control structures (e.g., `while` loops with various relational conditions) demonstrates a clear alignment with the project's objectives related to the generation of assembly code and the preservation of program logic.

Simulation of the output confirmed that the generated programs behave according to their C counterparts, highlighting the correctness of the translation process and the compiler's ability to generate semantically faithful low-level code. This validation step reinforces the importance of theoretical analysis in building reliable software tools.

The results support the notion that even a constrained subset of a high-level language can be systematically compiled into efficient, hardware-specific code when theory is applied methodically and consistently.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd. Boston, MA: Addison-Wesley, 2006, Also known as "The Dragon Book", ISBN: 978-0321486813.

[2] Stack Exchange Community, *How to write a very basic compiler*, `https://softwareengineering.stackexchange.com/questions/165543/how-to-write-a-very-basic-compiler`, Accessed: 2025-06-01, 2017.

[3] J. Crenshaw, *Let's build a compiler*, `https://www.compilers.iecc.com/crenshaw/`, Accessed: 2025-06-01, 1998.

[4] Facultad Regional Córdoba - UTN, *Gramáticas*, `https://www.institucional.frc.utn.edu.ar/sistemas/ghd/T-Gramaticas.htm`, Accessed: 2025-06-01, 2025.

[5] M. J. Fischer, *Lecture 22: Top-down parsing*, `https://pages.cs.wisc.edu/~fischer/cs536.f13/lectures/f12/Lecture22.4up.pdf`, Accessed: 2025-06-01, 2013.