UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Informatics

Implementing high-performance interpreted languages through the use of FPGAs. A proof of concept.

End of Degree Project

Bachelor's Degree in Informatics Engineering

AUTHOR: González Castiñeiras, Daniel

Tutor: Lucas Alba, Salvador

Cotutor: Gutiérrez Gil, Raúl

ACADEMIC YEAR: 2023/2024

Abstract

The exponential growth of hardware resources in recent decades has led to substantial software bloat, fostering the development of high-level programming languages that prioritize development efficiency over execution time and resource optimization. However, as the physical limits of transistor miniaturization are reached, this growth is slowing.

This presents a challenge for the software industry. As writing efficient, resource-conscious code is complex and time-consuming, while modern development paradigms focus on speeding up the process.

This thesis explores the use of reprogrammable hardware to bridge the gap between high-level abstractions and hardware. To this end, we developed LWVCG (Light-Weight Verilog Compiler Generator), a Python framework for specifying and generating hardware interpreters in Verilog. Using this framework, we implemented a hardware-based Just-In-Time (JIT) compiler for the SIMPLER language, a modification of SIMPL. Tested on two low-cost FPGAs (2x Lattice ICE40UP5K), this solution achieved 2x compilation speedup compared to the JavaScript V8 JIT compiler while operating at 1/250th of the clock frequency and consuming 1/16th of the power of an Intel i5-13400 CPU core.

Key words: Compilation, FPGA, Computing, CPU, Interpreted Programming Languages, Computational Efficiency, Energy Efficiency, Performance, Rust, Go, Javascript, Zig, Python, Verilog, Custom Hardware, Distributed Software Systems

Resumen

El crecimiento exponencial de los recursos de hardware en las últimas décadas ha provocado un considerable hinchamiento del software, lo que ha fomentado el desarrollo de lenguajes de programación de alto nivel que priorizan la eficiencia del desarrollo sobre el tiempo de ejecución y la optimización de recursos. Sin embargo, a medida que se alcanzan los límites físicos de la miniaturización de los transistores, este crecimiento se ralentiza.

Esto supone un reto para la industria del software: escribir código eficiente y consciente de los recursos es complejo y requiere mucho tiempo, mientras que los paradigmas de desarrollo modernos se centran en acelerar el proceso.

Esta tesis explora el uso de hardware reprogramable para salvar la distancia entre las abstracciones de alto nivel y el hardware. Para ello, desarrollamos LWVCG (Light-Weight Verilog Compiler Generator), un marco de trabajo en Python para especificar y generar intérpretes de hardware en Verilog. Utilizando este marco, implementamos un compilador Just-In-Time (JIT) basado en hardware para el lenguaje SIMPLER, una modificación de SIMPL. Probada en dos FPGAs de bajo coste (2x Lattice ICE40UP5K), esta solución consiguió una velocidad de compilación 2 veces superior a la del compilador JIT JavaScript V8, a la vez que funcionaba a 1/250 de la frecuencia de reloj y consumía 1/16 de la potencia de un núcleo de CPU Intel i5-13400.

Palabras clave: Compilación, FPGA, Computación, CPU, Lenguajes de Programación Interpretados, Eficiencia computacional, Eficiencia Energética, Rendimiento, Rust, Go, Javascript, Zig, Python, Verilog, Custom Hardware, Sistemas

Resum

El creixement exponencial dels recursos de maquinari en les darreres dècades ha provocat un inflament considerable del programari, cosa que ha fomentat el desenvolupament de llenguatges de programació d'alt nivell que prioritzen l'eficiència del desenvolupament sobre el temps d'execució i l'optimització de recursos. No obstant això, a mesura que s'assoleixen els límits físics de la miniaturització dels transistors, aquest creixement s'alenteix.

Això suposa un repte per a la indústria del programari: escriure codi eficient i conscient dels recursos és complex i requereix molt de temps, mentre que els paradigmes de desenvolupament moderns se centren a accelerar el procés.

Aquesta tesi explora lús de maquinari reprogramable per salvar la distància entre les abstraccions dalt nivell i el maquinari. Per això, desenvolupem LWVCG (Light-Weight Verilog Compiler Generator), un marc de treball a Python per especificar i generar intèrprets de maquinari a Verilog. Utilitzant aquest marc, implementem un compilador Just-In-Time (JIT) basat en maquinari per al llenguatge SIMPLER, una modificació de SIMPL. Provada en dues FPGAs de baix cost (2x Lattice ICE40UP5K), aquesta solució va aconseguir una velocitat de compilació 2 vegades superior a la del compilador JIT JavaScript V8, alhora que funcionava a 1/250 de la freqüència de rellotge i consumia 1/16 de la potència dun nucli de CPU Intel i5-13400.

Paraules clau: Compilació, FPGA, Informàtica, CPU, Llenguatges de programació interpretats, Eficiència computacional, Eficiència energètica, Rendiment, Rust, Go, Javascript, Zig, Python, Verilog, Personalitzat Desenvolupament de maquinari, programari distribuït Sistemes

Dedication

I dedicate this final thesis, as well as all the works that will follow, to all the people this world dares to call misfits.

To those who spend their free time embracing their uniqueness, studying, and learning. To those who fell in love with knowledge and were labeled as nerds for it. Not everyone has the courage to be different, so don't let the words of those who live a life you wouldn't want discourage you.

> *"Being different makes a person unforgettable. History doesn't remember the forgettable; it honors the unique minority that the majority can neither forget nor dare to be."*
>
> – Suzy Kassem

Contents

List of Figures

List of Tables

1 Introduction

In his landmark 1936 paper [15], Alan M. Turing introduced what is now known as the *Turing Machine* (TM). A Turing Machine is a computing device with an infinite tape, where a *reading/writing head* can read and write symbols from a finite alphabet. The machine operates in one of the finitely many *states*, changing according to a finite set of transition rules. Based on the current state and the symbol being read, a transition rule determines the machine's next state, whether the head moves left or right or if a symbol is written on the tape. Turing Machines can be efficiently implemented directly in hardware using logic gates.

Turing demonstrated how his machines could solve various problems. For example, he showed how to generate an infinite list of natural numbers by defining one of his machines. He further proposed that any problem that can be solved by some form of *effective calculation* could be solved by a Turing Machine provided the problem is encoded as a set of transition rules. This idea is commonly called the *Turing Thesis*.

Years later, Turing referred to a specific set of transition rules (i.e., a particular TM) that solves a given problem as a *program* [14]. In [15], Turing also introduced the concept of a *Universal Turing Machine* (UTM), capable of simulating the computation of any other TM by encoding the description of the simulated machine and its input data onto the UTM's tape.

In the 1940s, these ideas laid the foundation for the first prototypes of *general-purpose computers*, which can be viewed as UTMs that simulate any TM (i.e., execute any program). John von Neumann's work led to what is now known as *von Neumann's architecture* [17], which has been implemented in all computers since. Therefore, computer science has historically used inefficient but faster-to-program abstraction layers since the beginning of the computation, resulting in 74 years of accumulated layers of hardware and software on top of each other to execute even a simple statement. Each layer abstracts the previous one but introduces inefficiencies, which are then carried on by all the layers above.

Therefore, von Neumann's architecture (like any other existing abstraction) has limitations, most notably the von Neumann bottleneck, caused by the need to use the bus connecting memory and the Central Processing Unit (CPU) to transfer each instruction and piece of data, as well as any data updates.

Turing's original idea of treating a program as a Turing Machine suggests a potential solution to overcome inefficiencies: by running each program on a specialized or ad-hoc hardware Turing Machine. This involves implementing complex algorithms, which programming languages are based on, directly through state machines rather than on a general-purpose von Neumann computer. This thesis delves into exploring this possibility.

1.1 Motivation

The high-level programming languages currently used in the software industry have become easy to use but bloated and slow with compilers and interpreters that host complex logic. This trend has been driven by dramatic improvements in hardware components predicted by the so-called Moore's law, formulated in the sixties of the last century and claiming that every two years, the number of transistors allocated in integrated circuits would be duplicated [9]. In turn, modern professional software developers have built practices over decades that postpone execution efficiency to promote development time efficiency (i.e., the faster creation of perhaps slower, less efficient code).

Nowadays, physical limits have slowed down these improvements while (in contrast) the technological industry keeps growing faster than ever. Indeed, we are increasingly approaching a Post-Moore era in classical computing, where improvements to computing performance will come mainly from CPU architecture, algorithms, and software instead of from transistor and integrated circuit technologies [8]. Therefore, the miniaturization of hardware components is starting to be exchanged for optimization engineering and specialization of hardware components (GPUs are an example of specialization and optimization).

This situation collides with the programming paradigms and practices that have been driving developers over the past decade. In the case of the software industry, this change of priorities has been increasingly shown over the years through the creation of high-level but high-performance programming languages that abstract away parallelization, concurrency, and efficient memory management. Carbon, Rust, Go, or Zig are some examples of this. However, the adoption of these languages is still very limited, as they offer abstractions just to model common and safer practices without making them invisible to the programmer.

This thesis aims to contribute to the solution of this problem (difficulty of efficiently abstracting high performance, efficient code) by exploring the feasibility of using the ability of *Field-Programmable Gate Arrays* (FPGAs) [51] to create custom CPUs to migrate from software to hardware the abstractions provided by high-level programming languages. In this way, the possibility of creating hardware that adapts to the running programming language appears as an alternative to creating bloated and complex software compilers and interpreters. Sections 2.1 and 2.4 will dive deeper into the source of these problems and previous attempts to solve them.

1.2 Proposal and goals

This thesis proposes creating a custom processor capable of running high-level interpreted languages through a Just-In-Time compiler. In particular, as a proof of concept, an interpreter for a variant of the academic *Simple IMPerative Language* (SIMPL [18]) will be made. Therefore, the goals of this thesis are:

- Build an end-to-end prototype capable of receiving from a serial port the ASCII characters of a SIMPLER (modification of SIMPL programming language) program and printing to a different serial port the results of the computation.

- Build a framework in a high-level language to generate compilers made in a Hardware Description Language (HDL). Giving a general solution that would allow extending this thesis to other programming languages.

- Optimize the analysis phase of SIMPLER, showing an increase in tokens/sec and reductions/sec compared to a state-of-the-art interpreter running in a classical machine (Javascript V8 engine).
- Optimize energy efficiency of the analysis phase of SIMPL, showing a decrease in watts compared to a state-of-the-art interpreter running in a classical machine (Javascript V8 engine).
- Analyze space (hardware consumption) and time efficiency of the proposed solution and implemented algorithms.
- Analyze the price feasibility of replacing classical computing resources in servers/data centers for heterogeneous computing for a state-of-the-art (Javascript V8 engine).

It is important to note that the primary focus of optimization in this thesis will be on the JIT compiler rather than the execution of the compiled instructions. This decision was made to narrow the scope of the thesis to a proof of concept, with the execution of instructions being handled by a conventional but soft-core microprocessor [52]. To further optimize the execution phase, a custom processor and instruction set tailored to the specific language would need to be developed. Additionally, all comparisons in this thesis will be made against the JavaScript V8 engine, as 2.4 indicates that JavaScript is the most widely used language among software professionals. This serves to demonstrate the potential extensibility of the proposed solution to JavaScript.

1.3 Structure of this document

The rest of the document will apply the following structure:

- **Fundamentals:** This section will review all the basic knowledge needed to fully understand this thesis, including FPGAs, Compilers, interpreters, and tools used.
- **Design:** This section will explain in a high-level manner the design of the proposed system without falling into implementation details.
- **Implementation:** This section will go over the details of the implementation of each one of the parts of the system.
- **Validation:** This section will include functional testing results and proofs of each part working and performance tests and analysis of the presented results.
- **Conclusions:** This section will contrast the results obtained with state-of-the-art solutions to draw conclusions.
- **Relation with completed studies:** This section will go over the subjects of the cursed Bachelor's degree related to the topics treated.
- **Bibliography:** This section will host all the references mentioned in this document.

2 Fundamentals

In this section, introductory explanations cover both the historical context and the technical topics necessary to understand this thesis.

2.1 Historical context of the problem

The main focus of this thesis is to explore possible hardware optimizations to improve the time, space, and energy efficiency of high-level interpreted programming languages without harming software development time. This is a common trade-off that is usually assumed to be inevitable.

In order to motivate our argument, we first briefly explore the history of this trade-off and the reasons behind a current shift from prioritizing software development time to prioritizing efficiency.

2.1.1. The Moore era: Bottom optimizations

"There's Plenty of Room at the Bottom: An Invitation to Enter a New Field of Physics" [3] was a lecture given by the Nobel Prize-winning physicist Richard Feynman at the annual American Physical Society meeting at Caltech on December 29, 1959. In this lecture, Feynman foresaw a trend where computer performance would benefit from the miniaturization of computer components.

Later on, in 1965, Intel's founder Gordon Moore predicted the regularity of this miniaturization trend, now called Moore's law [9], which, until recently, doubled the number of transistors on computer chips every 2 years. Figure 2.1 shows the predictions made by Moore in his original paper [9] in a log-2 scale.

In this paper, Moore predicted the yearly double of the minimum number of components that could be placed in an integrated circuit and its impact on electronics and devices. Thus, in a single paper, he predicted the creation of home computers, smart watches, autonomous vehicles, and the general use of integrated circuits in consumer electronics.

In 1971 Intel created the first commercially available microprocessor, the Intel 404 [19], witnessing the accomplishment of the advances predicted by Moore. The abundance of hardware resources led to the creation of General programming languages filled with "wasteful" abstractions. Accordingly, these languages promote the optimization of software development time rather than the optimization of execution time or the use of hardware resources. Figure 2.2 shows the number of programming languages created

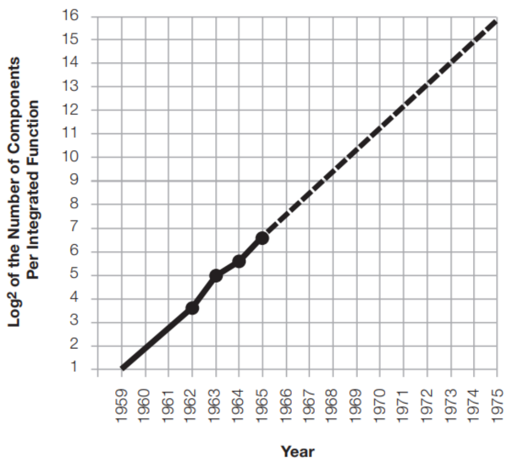


Figure 2.1: Moore prediction number of Components per Integrated circuit

between 1950 and 2000. The graph shows an increasing trend just after the commercialization of microprocessors in the 70s.

In sharp contrast, the number of languages created decreased after the year 2000 due to Standardization and Industry Adoption.

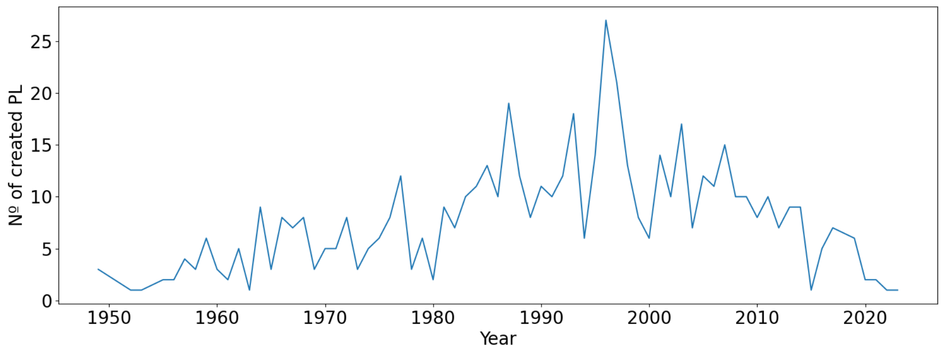


Figure 2.2: Nº of created PLs by year (from Wikipedia scrapped data)

Pioneering this revolution (and based on previous languages like Fortran or Cobol [20]), languages like Smalltalk, C, and SQL were born during the 70s.

These languages inspired a new spike during the 90s when more languages solely focused on improving the programmer's productivity were born. Some examples of such languages include Visual Basic, Ruby (1993), Java (1995), Javascript (1995), and PHP (1995). These are examples of attempts to hide some of the complexity behind lower-level languages such as C in exchange for performance, from abstracting the machine where the code is running through the use of interpreters to improving and adding more features to the already existing Object-Oriented paradigm.

These languages were created with the purpose of rapidly improving business adoption of the newly created technological advances and hardware resources, thus effectively pushing the “dot com” revolution by allowing products to be created easily. Figure 2.3 displays a correlation between the Nasdaq composite index peak in the 2000s and the number of created languages of figure 2.2.

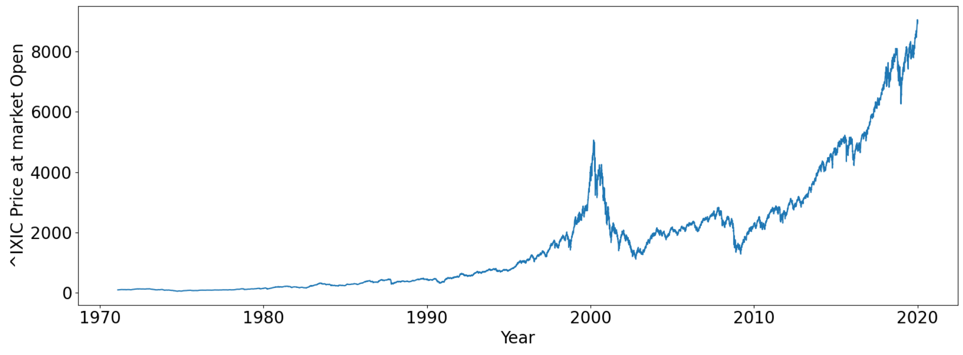


Figure 2.3: Nasdaq Composite index

The idea that "There's plenty of room at the Bottom" [3] has led to the creation of generations of software developers used to trade-off execution efficiency for development time. Indirectly impulsed by Moore's law and the fast development of hardware resources the industry has been relying on "Bottom optimizations" (improvements on the transistor technology and count) to sustain the fast growth of technological companies in the last years.

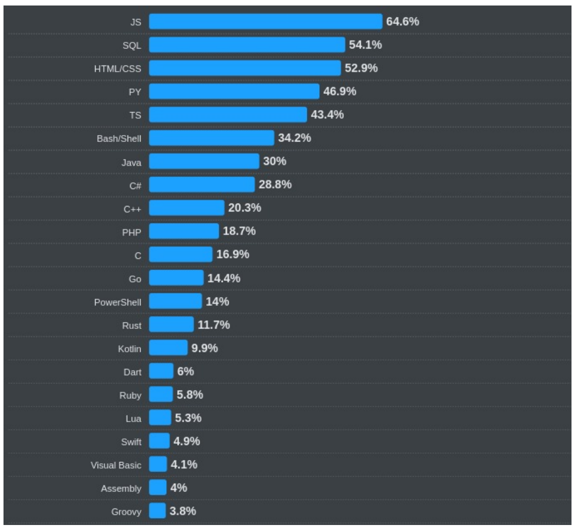
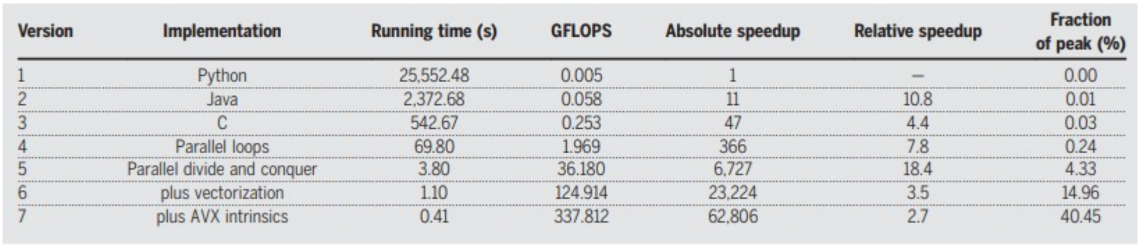


Figure 2.4: StackOverflow language popularity survey 2024

Figure 2.4 shows the 2024 StackOverflow (one of the most popular programming forums among professional software developers) language popularity survey [21]. The question asked for this survey was, "Which programming, scripting, and markup languages have you done extensive development work in over the past year, and which do you want to work in over the next year?". As it can be seen, even though traffic and throughput requirements of applications are at their highest in history, high-level non-compiled languages such as Javascript, Python, Java, and C# are the most used languages, with more than 60% of developers reporting to have extensively used Javascript in the last year.

Other lower-level old and new languages, such as C or Rust, have only been used by 25% or 11% of developers, respectively. As already demonstrated in [8], coding in these languages without any optimization leads to up to x47 performance losses. This, of course, depends on the particular application. Figure 2.5 shows the relative speedup of commonly used languages using a matrix-multiplication algorithm as an example.



Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	–	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

Figure 2.5: Programming language performance comparison (matrix-multiplication)

Figure 2.5 shows how the software development industry is currently sacrificing performance over development time.

2.1.2. The post-Moore era: Top optimizations

Is this miniaturization of semiconductors that power our reliance on abundant hardware resources sustainable? As stated in 2020s paper [8] the International Technology Roadmap for Semiconductors unfortunately foresees an end to miniaturization, and Intel (a leader in microprocessor technology) has acknowledged an end to the Moore cadence.

Figure 2.6 shows the evolution of transistor size in the CPUS of the main processor manufacturers on a logarithmic scale.

Note that Intel transistor sizes stayed unchanged for 5 years before changing to 7nm; AMD spent 4 years on 7nm before changing to 5nm in 2024; and Qualcomm spent 3 years to get from 4nm to 3nm.

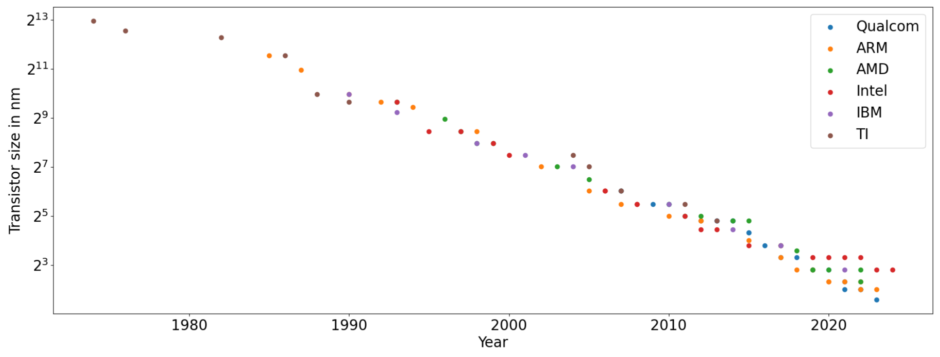


Figure 2.6: Transistor size of main CPU manufacturers

Taking a look at this graph, it is obvious that Moore's law has been slowing since 2016 and that in past years (since 2019/2020) CPU manufacturers have made little progress on transistor technologies (even though there are rumors that different possibilities will revive Moore's law with technological leaps such as 3D placement of transistors in CPUs).

Figure 2.7 shows transistor count on CPUs. As already stated in [8], we can see how, indeed, CPU transistor count has been increasing slower but still in exponential growth due to improvements in CPU architecture.

Therefore, we can conclude that in the incoming post-Moore era, the industry will have to face optimizations in the "Top" (CPU architecture, software, and algorithms) to sustain the still-growing technological sector. As CPU architecture is currently powering the exponential growth of computer performance, software, and algorithm improve-

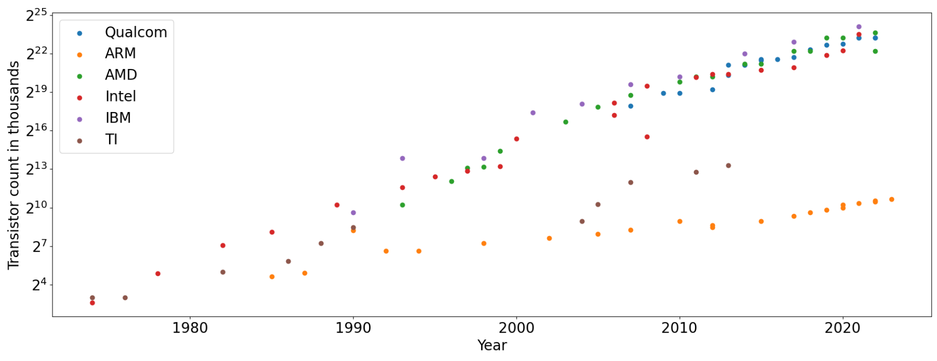


Figure 2.7: Transistor count of main CPU manufacturers

ments will have to power in the next decades, when CPU architecture improvements will not be worth it anymore due to the law of diminishing returns.

This will be especially tough in a context where most software developers and companies have well-established practices and strategies centered around highly abstracted and fast-to-develop systems. This trend of attempting to combine performance with rapid development is already showing up in the industry with the last developed languages, such as Go, Zig, and Rust, which attempt to bring performance to the table (especially for parallel and concurrent systems) without impacting too much development time.

2.2 Programming languages and compilers

Programming languages act as a bridge between the problem humans want to solve and the computer that executes the solution. Compilers play a crucial role in this process by translating programs of high-level programming languages, which are closer to human language, into machine code that computers can understand and execute. This section introduces compilers, interpreters, and the various phases that comprise their operation.

2.2.1. Structure of a compiler

Figure 2.8 shows the basic function of a compiler and its two main phases [2].

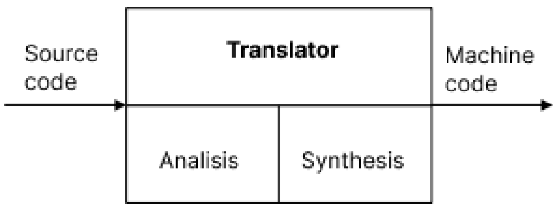


Figure 2.8: Basic compiler diagram

Thus, a programming language is defined by a specification that outlines how the language's structures are analyzed and translated into executable code. These specifications can be divided into the following categories:

- **Lexical specification:** It defines the valid symbols of the language that can appear in valid programs.

- **Syntactic specification:** defines the rules and structure of valid programs, focusing on how symbols and tokens are combined to form valid statements and expressions
- **Semantic specification:** defines both the semantic constraints of the language and the meaning of valid programs. In compilation, the meaning of a source program is considered to be its equivalent object program.

There is a general consensus on using Formal Language Theory to define the lexical and syntactic specifications of programming languages. However, semantic specifications lack such agreement due to the inherent complexity of defining meaning. Various semantic models have been proposed, with attributed semantics frequently employed for static specifications.

2.2.2. Approaches to translate programs into machine code

Programming language translators can be classified based on the type of translation used to convert them into machine code. Following this approach, translators can be categorized as follows: [2]:

- **Compilers:** a high-level source program is directly converter into machine language.
- **Interpreters:** instructions are translated and executed step by step as the source program runs.
- **Assemblers:** source code close to machine language is translated without providing any further abstractions.

Figure 2.9 displays the difference between the process that interpreters and compilers follow [2].

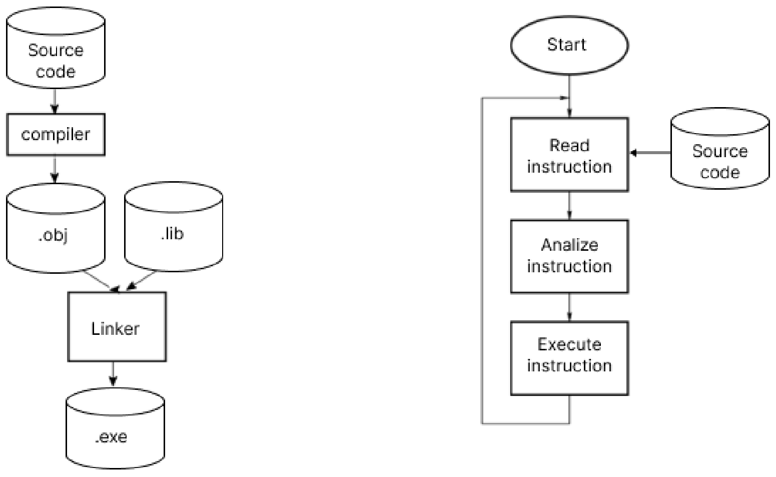


Figure 2.9: Difference between compilers and interpreters

In addition to these variations, there are other types of translators, such as Just-In-Time (JIT) compilers and bytecode interpreters. These translators combine aspects of both interpreters and compilers.

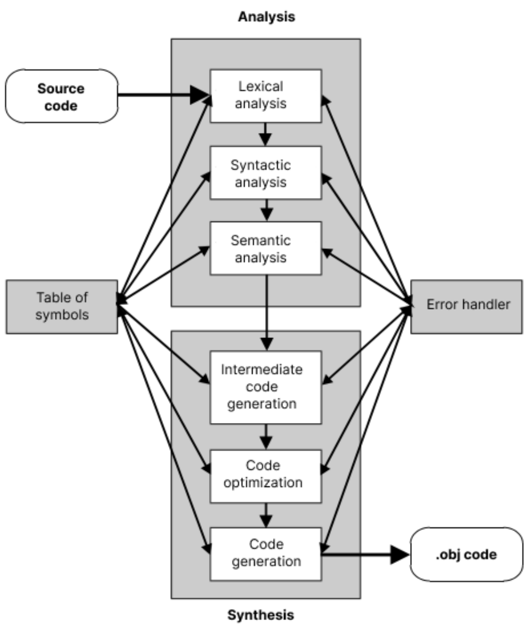


Figure 2.10: Detailed view of the classical compilation process

Figure 2.10 shows a more detailed view of the steps and processes inside a compiler [2]. The table of symbols is an abstraction present in the compiler to store information about the symbols of the programming language during compilation (created variables, functions, stack position...). While the error handler of a compiler is a component responsible for detecting, reporting, and managing errors that occur during the compilation process.

2.2.3. Lexical analysis

The objective of this phase is to read the source program and generate the lexical units ("tokens"). Key tasks include detecting language symbols, performing associated semantic actions, generating tokens, and discarding unnecessary strings. Regular expressions are used as the mathematical model to specify and efficiently perform the parsing in this phase. Figure 2.11 shows the lexical analysis of the string "x = y + z * 4" as an example.

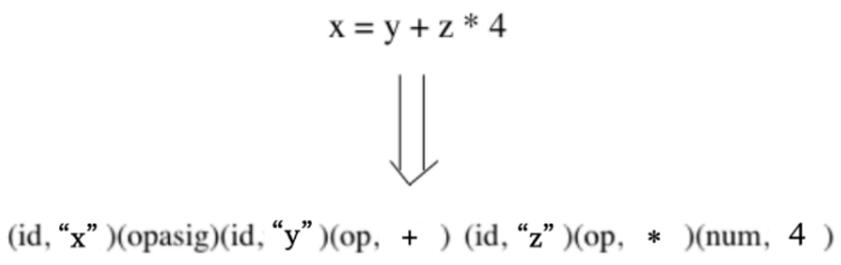

x = y + z * 4
(id, “x”)(opasig)(id, “y”)(op, +) (id, “z”)(op, *)(num, 4)

Figure 2.11: Example of lexical analysis for "x = y + z * 4"

As observed, the outcome of the lexical analysis is a series of tokens.

2.2.4. Syntactic analysis

The syntax analyzer takes the sequence of tokens generated by the lexical analyzer and checks whether it complies with the syntactic rules defined by the language's grammar.

If the sequence is valid, the analyzer produces a hierarchical representation called an abstract syntax tree (AST). This phase ensures that the code structure conforms to the language's syntax, resulting in a structured output as an AST, which can be further processed. Context-free grammars are used as the formalism for the syntactic specification.

Figure 2.12 illustrates the abstract syntax tree resulting from the syntactic analysis of the expression "x = y + z * 4," based on the grammar rules provided.

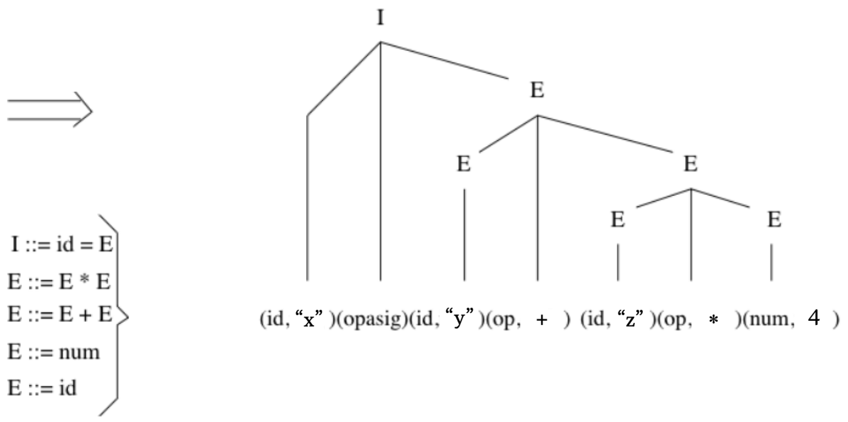

I ::= id = E
E ::= E * E
E ::= E + E
E ::= num
E ::= id
I
E
E
E
E
E
(id, "x")(opasig)(id, "y")(op, +) (id, "z")(op, *)(num, 4)

Figure 2.12: Example of syntactical analysis for 2.11.

2.2.5. Semantic analysis

This module's objective is to verify the language's semantic constraints. One of the primary components of semantic analysis is type checking, which ensures that each operation has operands permissible under the language's semantics.

Figure 2.13 illustrates a potential outcome of semantic checking for the expression "x = y + z * 4,". The annotated tree's nodes represent operations, while their children represent the respective operands. The "integer-to-real" node denotes the semantic action of type conversion, considering that the variables are of type real and the constant is of type integer.

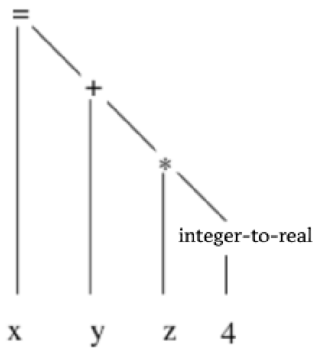
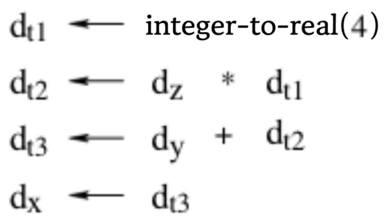


Figure 2.13: Example of semantic analysis for 2.12.

2.2.6. Intermediate code generation

Intermediate code is an abstract representation that is independent of the target machine for which the object code will be generated. This code must fulfill two key requirements:

it should be easy to generate from the syntactic analysis and simple to translate into the target machine language. A basic example is shown in Figure2.14. While this phase is not strictly necessary, it is highly recommended for portability reasons.

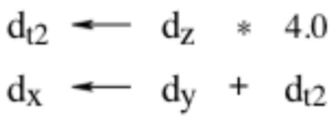


$$
\begin{aligned}
d_{t1} &\longleftarrow \text{integer-to-real}(4)\\
d_{t2} &\longleftarrow d_z * d_{t1}\\
d_{t3} &\longleftarrow d_y + d_{t2}\\
d_x &\longleftarrow d_{t3}
\end{aligned}
$$

Figure 2.14: Example of intermediate code generation for 2.13.

2.2.7. Code optimization

In this module, all available tools for optimizing intermediate machine-independent code are incorporated. Improvements are made to the produced code, similar to those shown in the example in Figure 2.15.

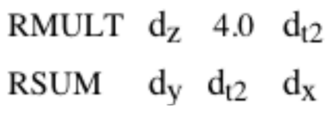


$$
\begin{aligned}
d_{t2} &\longleftarrow d_z * 4.0\\
d_x &\longleftarrow d_y + d_{t2}
\end{aligned}
$$

Figure 2.15: Example of intermediate code optimization for 2.14.

2.2.8. Code generation

In this final stage, machine code is generated while attempting to maximize the performance of the machine's architecture to produce the most optimized code possible. A simple example for a hypothetical machine language is illustrated in 2.16.



$$
\begin{array}{llll}
\text{RMULT} & d_z & 4.0 & d_{t2}\\
\text{RSUM} & d_y & d_{t2} & d_x
\end{array}
$$

Figure 2.16: Example of code generation for 2.15.

2.3 Introduction to FPGAs and HDL

This section discusses the technology underlying FPGAs (Field-Programmable Gate Arrays), their applications, and the differences between FPGAs, ASICs (Application-Specific Integrated Circuits), and general-purpose microprocessors.

As noted in [23], an FPGA is a reconfigurable integrated circuit (IC) that allows the implementation of a wide range of custom digital circuits by configuring and manipulating logic gates. These custom circuits can be optimized for applications such as digital signal processing (DSP), machine learning, and cryptocurrency mining. FPGAs are found in many consumer electronics, as well as specialized equipment like satellites and communication systems.

Thus, an FPGA provides a means to implement custom digital logic circuits in a relatively short time using a Hardware Description Language (HDL). Before the advent of reconfigurable logic ICs, engineers had to wire prototype boards to test their designs manually. Complex digital designs often led to significant wiring challenges, making bug detection and resolution nearly impossible. Reconfigurable logic ICs, such as CPLDs and FPGAs, help reduce space, wiring complexity, and debugging efforts.

An FPGA consists of numerous Programmable Logic Blocks (PLBs), each containing several Logic Cells. Figure 2.17 shows the block diagram for the iCE40HX1K FPGA.

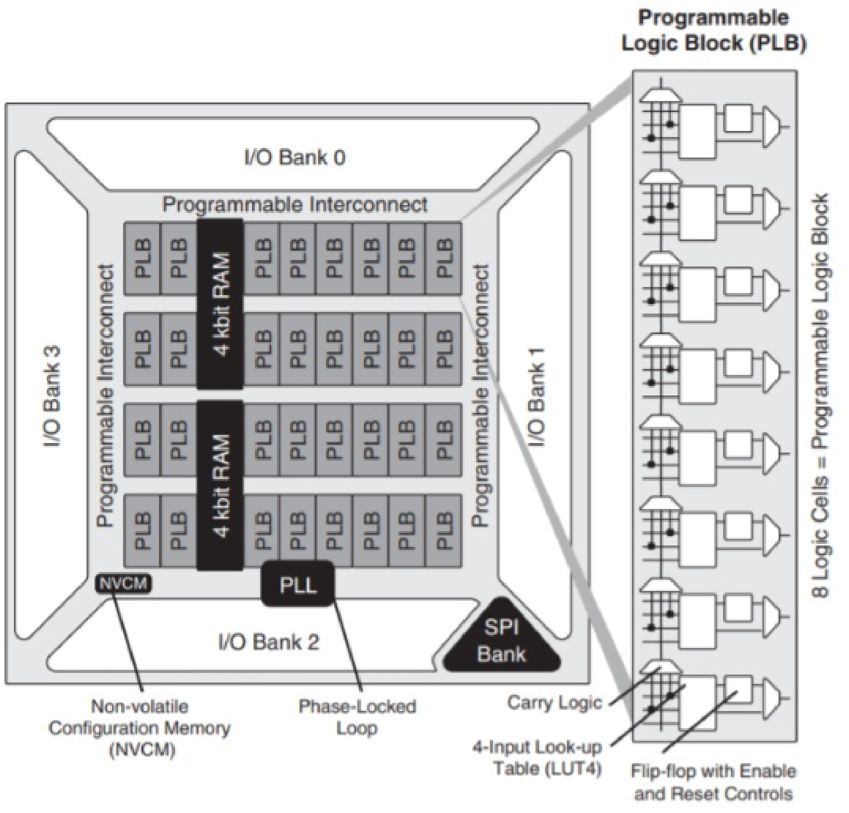

Programmable Logic Block (PLB)
I/O Bank 0
Programmable Interconnect
I/O Bank 3
I/O Bank 1
I/O Bank 2
PLB
4 kbit RAM
NVCM
PLL
SPI Bank
8 Logic Cells = Programmable Logic Block
Non-volatile Configuration Memory (NVCM)
Phase-Locked Loop
Carry Logic
4-Input Look-up Table (LUT4)
Flip-flop with Enable and Reset Controls

Figure 2.17: ICE40LP/HX diagram

Each cell contains a D flip-flop and a Lookup Table (LUT). Most FPGAs also include RAM for data storage, a phase-locked loop (PLL) for clock multiplication, and I/O banks for digital input/output. Some FPGAs feature additional peripherals, such as communication blocks (e.g., SPI), ADCs, or even embedded CPUs for sequential code execution. Programming an FPGA involves configuring the connections between these blocks and peripherals to implement a custom digital circuit.

The LUTs and D flip-flops in an FPGA are connected via the switch matrix [25], a network of switches interconnecting PLBs and other resources. While the specific construction of the switch matrix is proprietary, it is typically envisioned as a 2D grid of wires and multiplexers that control interconnections.

Some examples of usual applications for FPGAs are:

- **Parallel I/O operations:** For instance, displays made up of thousands or millions of LEDs require a very fast and independent control of many hardware components. Existing LED cubes offer smooth frame rates because they are controlled by FPGAs. A CPU would struggle to feed the required data to all of the LEDs (or driver chips) that fast.

- **Data acquisition (DAQ):** You can create custom digital logic that samples and buffers sensor data very quickly, which can then be read by a CPU later for analysis.

- **Specialized computations:** Depending on the application, digital circuits can often be configured to perform specific math operations much faster than a CPU, which is configured to execute generic operations (e.g. math, load, store).

- **Custom processor:** Because an FPGA allows you to build an almost limitless number of digital circuits, you can actually implement one or more CPUs in an FPGA. This offers the ability to create customized CPUs.

HDLs (Hardware Description Languages) are used to specify FPGA designs. These languages are not procedural; the code is not executed sequentially in a processor. Instead, HDLs are synthesized, a process that converts the design into gate-level representations that can be implemented by the FPGA, effectively transforming code into a digital circuit. Common HDLs include VHDL and Verilog. Figure 2.18 shows the diagram and Verilog implementation of a half-adder [7].

```
module Add_half
 (sum, c_out, a, b);
 input   a, b;
 output c_out, sum;
 xor (sum, a, b);
 and (c_out, a, b);
endmodule
```

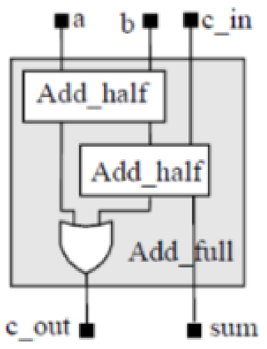


Figure 2.18: Example of half adder implementation in Verilog

2.4 State of the art and current solutions

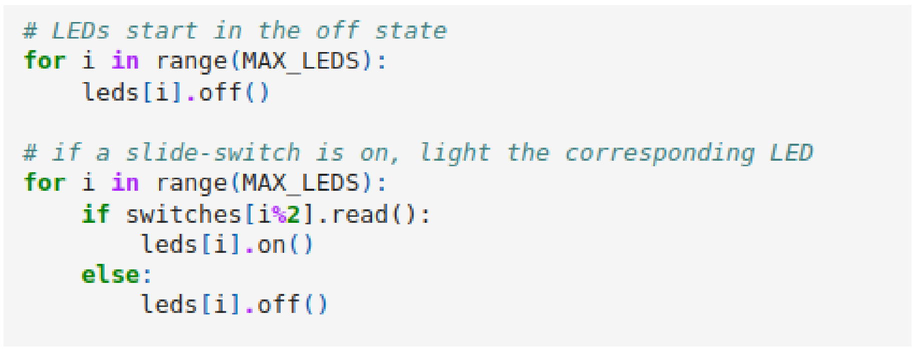
As mentioned in the introduction and Section 2.1, current efforts to address the challenge of efficiently abstracting high-performance code typically focus on two approaches: developing new programming languages with specialized abstractions to enhance the programmer's experience or utilizing specialized hardware (e.g., GPUs) for specific applications, which is abstracted from the programmer through multiple layers of abstraction.

These solutions face notable adoption barriers. Languages like Carbon, Rust, and Zig, though powerful, have a steeper learning curve compared to high-level languages like Python or JavaScript. This is reflected by their lower adoption rates (see Section 2.1). Furthermore, the high cost of custom hardware fabrication limits scalability, making it viable only for widely used applications such as video rendering, machine learning, and cryptocurrency mining.

This section reviews prior efforts to implement high-level abstractions in hardware to optimize general-purpose high-level languages, with a focus on Python due to its popularity among researchers.

2.4.1. High-level based abstractions

Several tools enable software engineers to use high-level languages, such as Python, to program FPGAs by abstracting HDL languages. An example of such a framework is PYNQ [33], released by Xilinx in 2017. Figure 2.19 shows an example of PYNQ code.



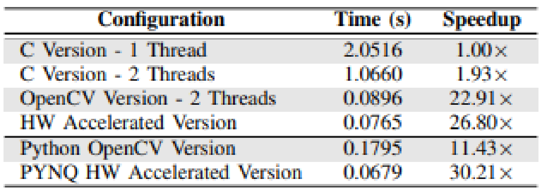
```
# LEDs start in the off state
for i in range(MAX_LEDS):
    leds[i].off()

# if a slide-switch is on, light the corresponding LED
for i in range(MAX_LEDS):
    if switches[i%2].read():
        leds[i].on()
    else:
        leds[i].off()
```

Figure 2.19: PYNQ Example

Research has also explored the feasibility of leveraging these frameworks to accelerate Python code execution without negatively impacting development time.

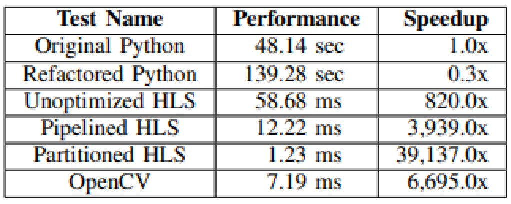
The authors of [10] argue that as FPGAs evolve to incorporate more heterogeneous processing elements, such as ARM cores, a shift toward software-oriented development environments is necessary. This paper uses PYNQ to compare development time and performance speedups when using FPGAs with Python versus C and OpenCV. Figure 2.20 presents the results.



Configuration	Time (s)	Speedup
C Version - 1 Thread	2.0516	1.00×
C Version - 2 Threads	1.0660	1.93×
OpenCV Version - 2 Threads	0.0896	22.91×
HW Accelerated Version	0.0765	26.80×
Python OpenCV Version	0.1795	11.43×
PYNQ HW Accelerated Version	0.0679	30.21×

Figure 2.20: PYQN, C and OpenCV comparison

In contrast, [12] introduces an open-source infrastructure and tool suite designed to integrate FPGA accelerators into Python applications, with the goal of streamlining the process and enhancing productivity for both novice and expert users. Figure 2.21 presents the results.



Test Name	Performance	Speedup
Original Python	48.14 sec	1.0x
Refactored Python	139.28 sec	0.3x
Unoptimized HLS	58.68 ms	820.0x
Pipelined HLS	12.22 ms	3,939.0x
Partitioned HLS	1.23 ms	39,137.0x
OpenCV	7.19 ms	6,695.0x

Figure 2.21: Hot&Spicy results

2.4.2. Synthesizing high-level languages into HDL

Another approach is to translate high-level programming languages directly to HDL. Some research has already been done on this aspect (mostly around Python).

[5] Proposes a Python-based high-level programming framework to simplify the programming and optimization of CPU-FPGA heterogeneous systems. The framework compiles Python into HDL, effectively abstracting the hardware details from programmers. Figure 2.22 illustrates the proposed FPGA design flow.

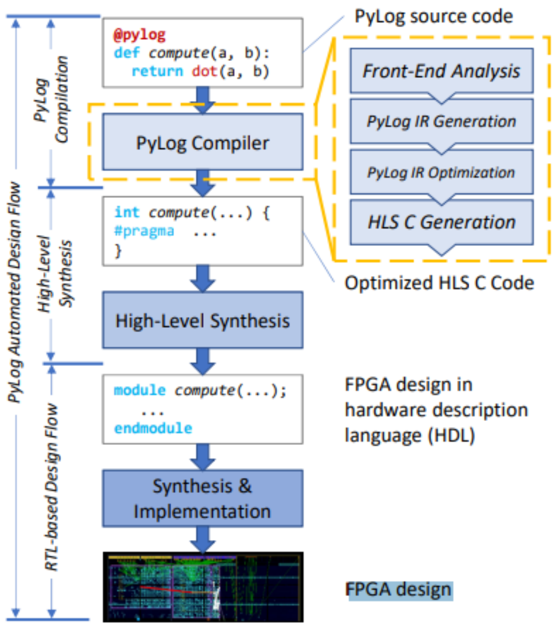


Figure 2.22: FPGA Design Flow with PyLog

[16] Also serves as a proof of concept for a Python-to-FPGA compiler, built on the Numba Just-In-Time (JIT) compiler for Python and the Intel FPGA SDK for OpenCL. This enables Python users to leverage FPGA cards as accelerators for Python code. Figure 2.23 illustrates the proposed FPGA design flow.

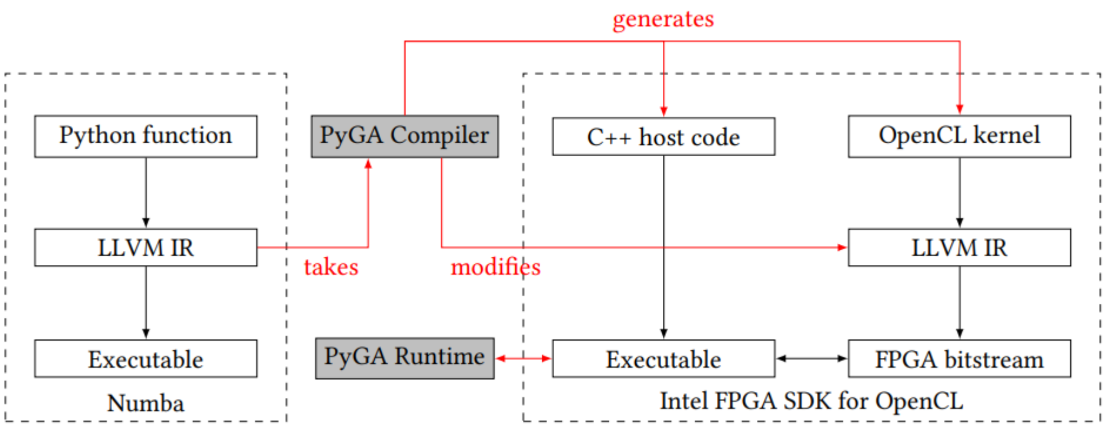
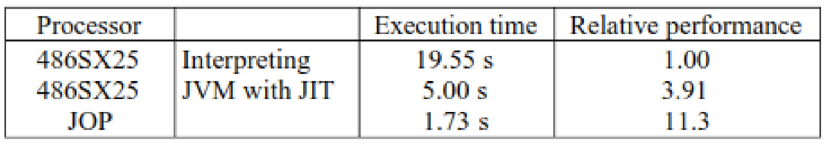


Figure 2.23: FPGA Design Flow with PyGa

2.4.3. Design custom processors and interpreters

Another approach is to design custom processors that natively execute programming languages typically interpreted or compiled for a classical CPU. This requires developing a new instruction set and CPU architecture for each language.

[11] Proposes a hardware implementation of the Java Virtual Machine (JVM), with a focus on real-time applications. Figure 2.24 shows speedups of up to 11x, running at 24 MHz, compared to a standard JVM running on an Intel processor.



Processor		Execution time	Relative performance
486SX25	Interpreting	19.55 s	1.00
486SX25	JVM with JIT	5.00 s	3.91
JOP		1.73 s	11.3

Figure 2.24: JOP and 486SX15 performance comparison

In [6], the implementation of a Java microprocessor core in silicon to accelerate Java execution is described. The design achieves a running frequency of up to 21 MHz, with approximately 50% utilization of the FPGA's area.

In summary, FPGAs have been explored in various approaches to accelerate the execution of high-level languages while maintaining abstraction. All the cited works demonstrate significant speedups compared to traditional machines or even optimized hardware running C/C++ code.

However, most of these approaches are either heavily focused on embedded systems or specific scientific computing tasks (optimizing only certain computations) or are outdated in terms of comparison with current language performance. For example, the last two cited papers are the only examples of interpreters implemented on reconfigurable hardware, yet their comparisons are against 25 MHz Intel CPUs from the early 2000s.

2.5 Tools and programming languages used

This section provides an overview of the design, programming, simulation, synthesis, and testing tools used in the development of this thesis:

- **VSCode:** code editor [26].
- **Apio:** FPGA development ecosystem [28].
- **Yosis:** framework for RTL synthesis tools [29].
- **Python:** scripting programming language [31].
- **KiCad:** open source tool for PCB design [27].
- **GTKWave:** fully featured GTK+ wave viewer [30].
- **Saleae logic Anayzer:** logic analyser for debugging physical digital signals [32].

All software tools and resources used to develop this thesis are open-source and free to use.

3 Design

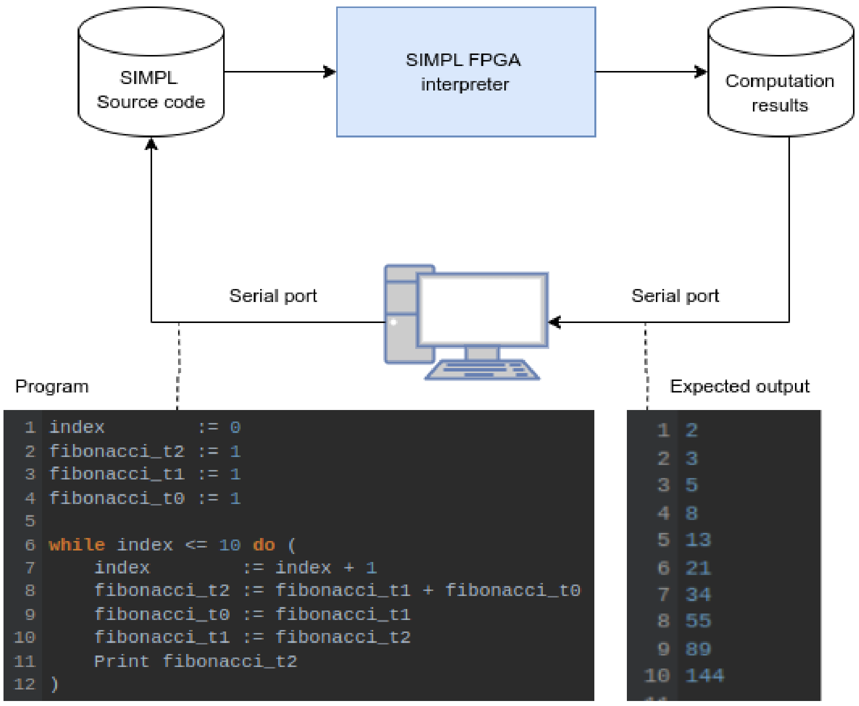
The following section covers the requirements and high-level design of the entire system, along with its individual components.

3.1 System requirements

From the goals specified in Section 1.2 and as a proof of concept, we can derive the following requirements for the system:

- The system must be implementable on affordable FPGAs with clock frequencies between 24 MHz and 100 MHz, logic cell counts between 10,000 and 100,000, and priced between below and 50€.
- The system must implement a JIT (just in time) compiler that translates and executes the SIMPLER language (modification of the SIMPL language).
- The system must accept input programs in ASCII format via a serial communication protocol compatible with USB and accessible through the Python Serial library.
- The system must output the computation results by a Serial communication protocol that can be made compatible with USB and available through the Python Serial Library.
- The system must be specified through the use of high-level abstractions in a custom Python framework. Allowing for the extensibility of this thesis to other languages.
- The JIT compiler scanner running in computing resources as the ones specified in 3.1 must have a throughput greater than 10,000,000 tokens per second (Javascript V8 JIT performance) running in an Intel Core I5 at 3 GHz.
- The JIT compiler parser running in computing resources as the ones specified in 3.1 must have a throughput greater than 10,000,000 tokens per second (Javascript V8 JIT performance) running in an Intel Core I5 at 3 Ghz.
- The JIT compiler running in computing resources as the ones specified in 3.1 must consume less than 20 Watts (Javascript V8 JIT performance) running in a single Intel I5 core at 3 GHz.

Verilog has been selected as the HDL for implementing the interpreter, and Python has been chosen for developing the compiler generator framework. Figure 3.1 illustrates the system during the code execution process.



```
index        := 0
fibonacci_t2 := 1
fibonacci_t1 := 1
fibonacci_t0 := 1

while index <= 10 do (
    index        := index + 1
    fibonacci_t2 := fibonacci_t1 + fibonacci_t0
    fibonacci_t0 := fibonacci_t1
    fibonacci_t1 := fibonacci_t2
    Print fibonacci_t2
)
```

```
2
3
5
8
13
21
34
55
89
144
```

Figure 3.1: System diagram during code execution

As shown, one of the modifications to the SIMPL language is the addition of a Print instruction. The system's expected output is to return the results of any Print statements in the source code via the output serial port.

On the other hand, Figure 3.2 displays a system diagram during the interpreter synthesis process.

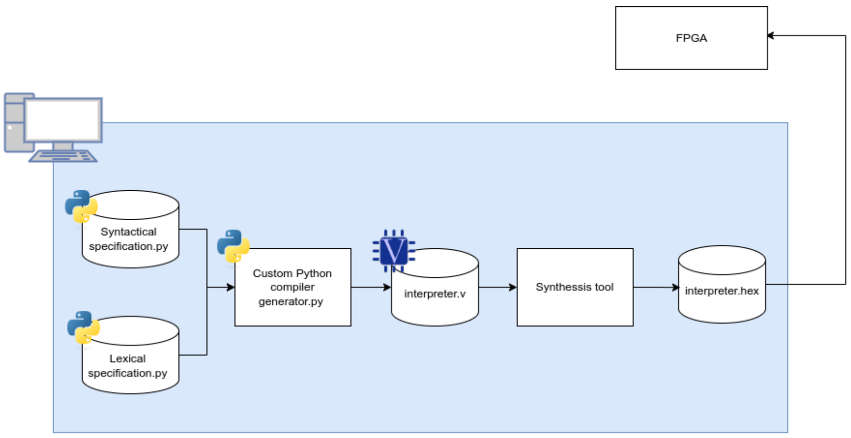
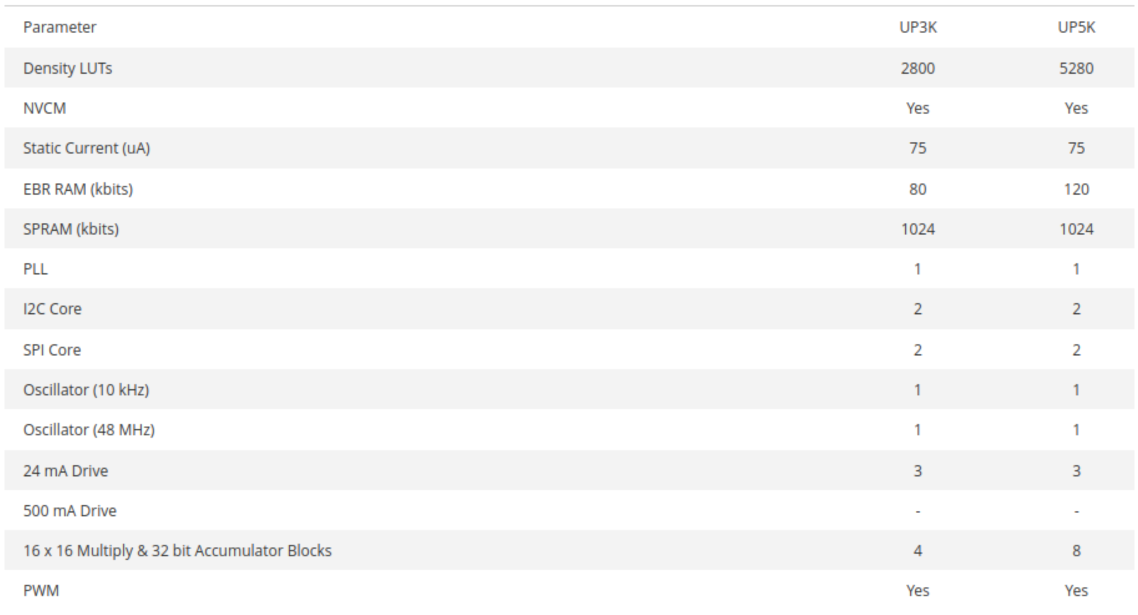


Figure 3.2: Interpreter synthesis process

These requirements have been simplified due to budget constraints. In an industry setting, a heterogeneous computing system would typically be used, with a lightweight server and a hard CPU providing programming access to the FPGA. The FPGA interpreter accelerator could be connected to the main system bus via PCI Express (a high-speed serial expansion bus standard with speeds up to 242 GB/s), allowing direct access to the network card through DMA for network applications. In this configuration, the FPGA interpreter would act as a PCI Express hardware accelerator, similar to GPUs. However, the cost of PCI Express-compatible FPGAs and large-scale dedicated RAM exceeds the budget for this project.

3.2 System design

Based on the requirements specified in Section 3.1, the FPGA selected for this project is the ICE40-UP5K [34], a low-power, low-cost FPGA designed for general-purpose edge and mobile applications. Figure 3.3 shows its characteristics. Its integrated I2C core, 1 Mb of SPRAM, and 120 Kb of EBR RAM meet our application needs while keeping the cost of a single FPGA under 10€.



Parameter	UP3K	UP5K
Density LUTs	2800	5280
NVCM	Yes	Yes
Static Current (uA)	75	75
EBR RAM (kbits)	80	120
SPRAM (kbits)	1024	1024
PLL	1	1
I2C Core	2	2
SPI Core	2	2
Oscillator (10 kHz)	1	1
Oscillator (48 MHz)	1	1
24 mA Drive	3	3
500 mA Drive	-	-
16 x 16 Multiply & 32 bit Accumulator Blocks	4	8
PWM	Yes	Yes

Figure 3.3: ICE40-UP5K Specifications

However, the limited number of logic cells (only 5,000) on a single device necessitates dividing the system into two parts and using two FPGAs. The first FPGA will host the SIMPL JIT interpreter, while the second will execute the result of the JIT compilation, simulating a microprocessor. For this, two ICE40-UP5K Breakout Boards [35] will be used. Figure 3.4 shows an image of one of these boards.

The number and layout of General I/O connectors on this board limit the bus connecting the two boards to 9 bits. Due to the testing setup, high-frequency signals cannot be transmitted through this bus, so communication between the two FPGAs will operate

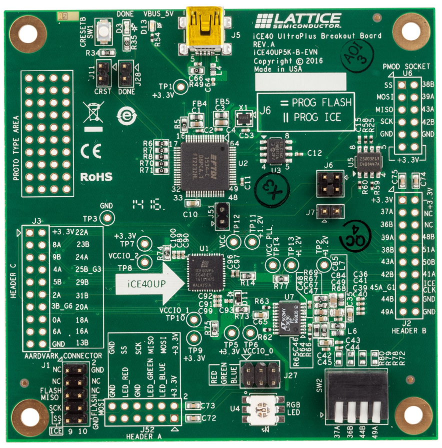


Figure 3.4: ICE40-UP5K Breakout Boards

at 10 kHz. The 9-bit bus will be divided as follows: 8 bits for transmitting machine code instructions from the interpreter to the microprocessor's instruction memory and 1 bit for the bus clock signal.

I2C [53] will be used to transmit both SIMPL source code and computation results. I2C is a synchronous, multi-controller/multi-target, single-ended serial communication bus developed by Philips Semiconductors in 1982, with a bitrate of up to 5 Mbit/s. It meets the requirements in Section 3.1, is accessible via the Python Serial library using I2C-to-USB devices, and offers high error resistance. Given the transmission distance of less than 50 cm (I2C supports 1 MHz over this range), it is ideal for this setup. Figure 3.5 shows the I2C-to-USB device used.

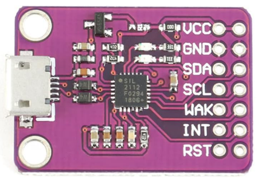


Figure 3.5: I2C to USB device used

Figure 3.6 presents the final hardware diagram of the system, detailing all communication lines and their operating frequencies.

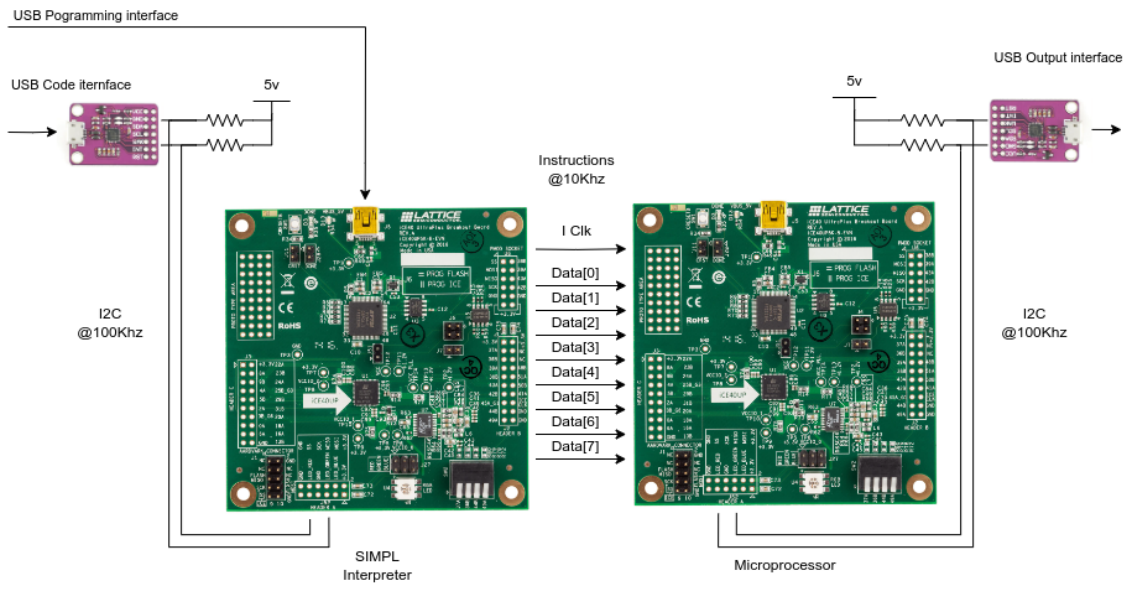
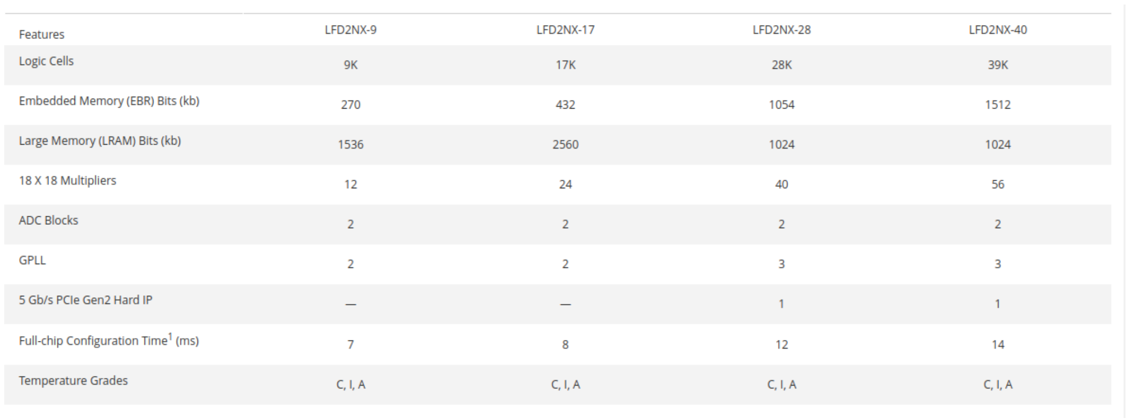


Figure 3.6: System hardware diagram

As shown, the speed of both the I2C bus and the instruction bus will be the system's bottleneck. However, these limitations can be easily mitigated with a higher budget. More expensive FPGAs can offer up to 1 million logic cells, allowing for significantly larger systems and integrated PCIe support. For example, Figure 3.7 shows the specifications of the Lattice Certus-NX family, including the LFD2NX-40 model with 39,000 logic cells and PCIe 2 compatibility, priced at 60€.

3.3 Language specification (SIMPLER)

To meet the system requirements, we will modify the original SIMPL specification to create a new language, referred to as SIMPLER. This version includes additional gram-



Features	LFD2NX-9	LFD2NX-17	LFD2NX-28	LFD2NX-40
Logic Cells	9K	17K	28K	39K
Embedded Memory (EBR) Bits (kb)	270	432	1054	1512
Large Memory (LRAM) Bits (kb)	1536	2560	1024	1024
18 X 18 Multipliers	12	24	40	56
ADC Blocks	2	2	2	2
GPLL	2	2	3	3
5 Gb/s PCIe Gen2 Hard IP	—	—	1	1
Full-chip Configuration Time[1] (ms)	7	8	12	14
Temperature Grades	C, I, A	C, I, A	C, I, A	C, I, A

Figure 3.7: Specifications of the Lattice Certus-NX family

matical rules to simplify code generation, a Print instruction, and boolean variables. The SIMPLER language has the following features:

- **Variables:** all variables are 32 bit integers, in case of boolean variables they are stored as a 0 or 1 integers respectively.
- **Constants:** To avoid using internal 32-bit registers for storing token attributes (where constants are stored during compilation), constants are limited to 8-bit unsigned numbers.
- **Conditional and loop structures:** same as in the SIMPL language: if .. then .. else .. and while .. do .. loops.
- **I/O output:** SIMPLER includes a Print instruction that outputs the contents of the specified variable via the I2C bus.

3.3.1. Lexical specification of SIMPLER

Listing 3.1 presents the lexical specification of SIMPLER, using regular expressions [36] as the formalism. Note that "\" is used as an escape character to distinguish the ASCII "+" from the regular expression quantifier.

3.3.2. Syntax specification of SIMPLER

Listing 3.2 includes the Syntactic specification of SIMPLER. The formalism used is context-free grammars; for more information on non-contextual grammars, check [37].

The main difference from the original SIMPL specification is the introduction of new intermediate expressions, which aim to simplify the interpreter's hardware implementation.

For example, replacing "ID" with "Var $\rightarrow$ ID" simplifies the semantic and code generation process, allowing us to specify both at the end of the reduction. Additionally, complex expressions like "If Boolean then APAR Instruction CPAR else APAR Instruction CPAR" have been shortened using intermediate expressions. This reduces the number of symbols on the right-hand side of expressions to a maximum of 3, enabling the use of a 32-bit register to store operands and results for semantic operations.

```
ID         →  [a-z][A-Za-z0-9]*
CONST      →  [0-9]*
BLANK      →  [\n \t\r]+
NOT        →  !
LESSEQ     →  <=
EQUALS     →  =
MULT       →  \*
PLUS       →  \+
MINUS      →  -
PRINT      →  Print
SKIP       →  SKIP
ASSIGN     →  :=
SEMICOLON  →  ;
IF         →  if
THEN       →  then
ELSE       →  else
APAR       →  \(
CPAR       →  \)
WHILE      →  while
DO         →  do
TRUE       →  True
FALSE      →  False
OR         →  ||
```

Listing 3.1: Lexical specification of SIMPLER

```
Program → Instruction
Instruction → SKIP
Instruction → ID ASSIGN Arithmetic
Instruction → ID ASSIGN Boolean
Instruction → Instruction SEMICOLON Instruction
Instruction → IfThen IfBody Else
Instruction → WhileDo WhileBody
Instruction → Print Var
WhileDo → While Boolean DO
WhileBody → APAR Instruction CPAR
While → WHILE
IfThen → IF Boolean THEN
Else → ELSE APAR Instruction CPAR
IfBody → APAR Instruction CPAR
Arithmetic → CONST
Arithmetic → Var
Arithmetic → Arithmetic PLUS Arithmetic
Arithmetic → Arithmetic MINUS Arithmetic
Arithmetic → Arithmetic MULT Arithmetic
Boolean → Arithmetic EQUALS Arithmetic
Boolean → Arithmetic LESSEQ Arithmetic
Boolean → Boolean OR Boolean
Boolean → NOT Boolean
Var → ID
```

Listing 3.2: Syntax specification of SIMPLER

```
Program → Instruction
Instruction → SKIP
Instruction → ID ASSIGN Arithmetic [creaVar(ID)]
Instruction → ID ASSIGN Boolean [creaVar(ID)]
Instruction → Instruction SEMICOLON Instruction
Instruction → IfThen IfBody Else
Instruction → WhileDo WhileBody
Instruction → Print Var
WhileDo → While Boolean DO
WhileBody → APAR Instruction CPAR
While → WHILE
IfThen → IF Boolean THEN
Else → ELSE APAR Instruction CPAR
IfBody → APAR Instruction CPAR
Arithmetic → CONST
Arithmetic → Var
Arithmetic → Arithmetic PLUS Arithmetic [Arithmetic = creaVarTemp()]
Arithmetic → Arithmetic MINUS Arithmetic [Arithmetic = creaVarTemp()]
Arithmetic → Arithmetic MULT Arithmetic [Arithmetic = creaVarTemp()]
Boolean → Arithmetic EQUALS Arithmetic [Boolean = creaVarTemp()]
Boolean → Arithmetic LESSEQ Arithmetic [Boolean = creaVarTemp()]
Boolean → Boolean || Boolean [Boolean = creaVarTemp()]
Boolean → NOT Boolean [Boolean = creaVarTemp()]
Var → ID [Var = creaVar(ID)]
```

Listing 3.3: Semantic specification of SIMPLER

3.3.3. Semantic specification of SIMPLER

Listing 3.3 presents the semantic specification of SIMPLER, using attribute grammars as the formalism. For more information on attribute grammars, refer to [38]. Specifically, we have used an attribute grammar where each symbol has a single 8-bit attribute.

Note the use of the functions "creaVarTemp()" and "creaVar()," which are part of the symbol table. These will be explained in detail later. Additionally, the "Arithmetic," "Boolean," and "Var" attributes store the memory address where the result of the expression will be stored.

3.3.4. Examples

Some examples of short SIMPLER programs are listed in this section.

Program in Listing 3.4 prints the numbers from 1 to 10.

```
index := 0;
while index <= 10 do (
    index := index + 1;
    Print index
)
```

Listing 3.4: SIMPLER For loop

Program in Listing 3.5 calculates and prints the first 13 numbers of the Fibonacci series.

```
index        := 0;
fibonacci_t0 := 1;
fibonacci_t1 := 1;
temp := 0;
Print fibonacci_t0;
Print fibonacci_t1;

while index <= 10 do (
    index := index + 1;
    temp := fibonacci_t0 + fibonacci_t1;
    fibonacci_t0 := fibonacci_t1;
    fibonacci_t1 := temp;
    Print fibonacci_t1
)
```

Listing 3.5: SIMPLER Fibonacci calculator loop

Program in Listing 3.6 calculates and prints the first 12 powers of 2.

```
index    := 0;
power    := 1;
Print index;

while index <= 10 do (
    index := index + 1;
    power := power * 2;
    Print power
)
```

Listing 3.6: SIMPLER Power of 2 calculator

Program in Listing 3.7 prints the even numbers for 0 to 10.

```
index    := 0;
power    := 1;
Print index;

while index <= 10 do (
    rest := index;
    while 0 <= index do (
        rest := rest - 2
    );

    if rest = 0 then (
        Print index
    ) else (
        skip
    )
)
```

Listing 3.7: SIMPLER even numbers calculator

3.4 Light-Weight Verilog Compiler Generator (LWVCG)

This section describes the high-level design of the custom Python framework developed to generate Verilog compilers, referred to as LWVCG (Light Weight Verilog Compiler Generator). Based on the simplifications in Section 3.3, the generator is divided into two distinct parts:

- **Scanner generator:** This module will take the lexical specification of the language and generate a Verilog module that raises a flag when a token is detected and indicates the detected token as an integer. Additionally, it will generate a Verilog test bench to verify the correct behavior of the generated module.

- **Parser generator:** This module will take the syntactic, semantic, and code generation specifications of the language and generate a Verilog file to handle parsing. The generated Verilog module will receive detected tokens as input and output the 8-bit bus carrying the generated machine code instructions, along with flags such as "parsing done" and "begin code execution" to ensure proper integration with the microcontroller.

Figure 3.8 displays these blocks, along with their inputs, outputs, and interconnections.

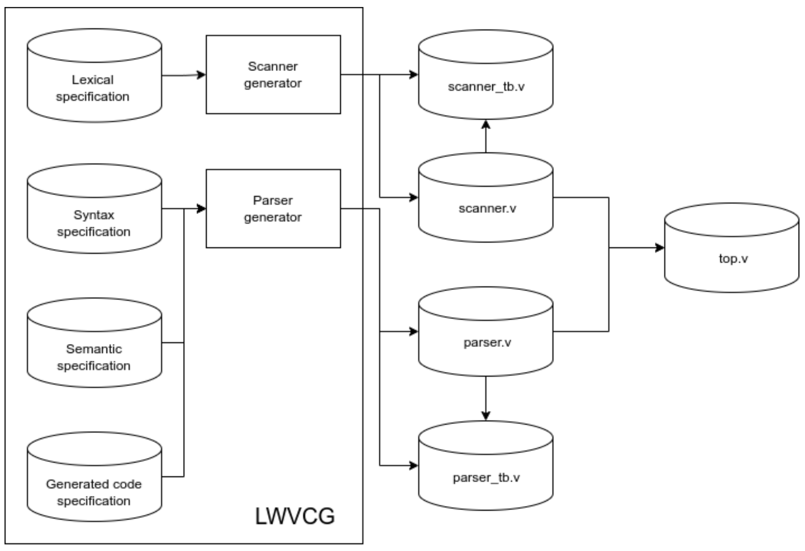


Figure 3.8: LWVCG Module diagram

Lastly, these two generated files will be used by a third top module, which will be coded by hand to map the input signals to the scanner, integrate the scanner and parser, and map the parser signals to the output.

For the syntactic and lexical language specifications, we will use plain Python dictionaries to map tokens to lists: lists of tokens for syntactic specifications and lists of strings for lexical specifications. For semantic and code generation specifications, dictionaries will map a list of tokens (representing a reduction) to a list of either AttributeOperations or machine code instructions. Tokens will be represented by numerical values using an Enum, and AttributeOperations will be defined using a plain Python data class. Listings 3.8, 3.9 3.10, and 3.11 provide examples of these Python specification dictionaries for the SIMPLER language.

```
SIMPLER_LEXIC_SPECIFICATION = {
    Symbol.PLUS       : ["+"],    Symbol.MINUS      : ["-"],    Symbol.
        PRINT      : ["Print"],
    Symbol.EQUALS     : ["="],    Symbol.LESSEQ     : ["<="],    Symbol.
        NOT        : ["!"],
    Symbol.OR        : ["||"],    Symbol.TRUE       : ["true"],    Symbol.
        FALSE      : ["false"],
    Symbol.SKIP       : ["skip"],  Symbol.ASSIGN     : [":="],    Symbol.
        SEMICOLON  : [";"],
```

Listing 3.8: Example of LWVCG SIMPLER Lexic specification

```
SIMPL_SINTAX_SPECIFICATION = {
    Symbol.Program      : [
        [ Symbol.Instruction , Symbol.END ]
    ],
    Symbol.Instruction  : [
        [ Symbol.SKIP ],
        [ Symbol.ID, Symbol.ASSIGN, Symbol.Arithmetic ],
        [ Symbol.ID, Symbol.ASSIGN, Symbol.Boolean ],
        [ Symbol.Instruction , Symbol.SEMICOLON, Symbol.Instruction ],
        [ Symbol.IfThen , Symbol.IfBody , Symbol.Else ],
        [ Symbol.WhileDo, Symbol.Body ],
        [ Symbol.PRINT, Symbol.Var ]
    ],
```

Listing 3.9: Example of LWVCG SIMPLER Syntax specification

```
SIMPLER_SEMANTIC_SPECIFICATION = {
    (Symbol.Instruction , Symbol.SKIP):
        AttributeOperation(None, Operation.NOP, None),
    (Symbol.Instruction , Symbol.ID, Symbol.ASSIGN, Symbol.Arithmetic):
        AttributeOperation(Attribute.RIGHT_HAND_1,
        Operation.ASSIGN_CREATE_VAR_FROM_ID,
        Attribute.RIGHT_HAND_1),
    (Symbol.Instruction , Symbol.ID, Symbol.ASSIGN, Symbol.Boolean):
        AttributeOperation(Attribute.RIGHT_HAND_1,
        Operation.ASSIGN_CREATE_VAR_FROM_ID,
        Attribute.RIGHT_HAND_1),
```

Listing 3.10: Example of LWVCG SIMPLER Syntax specification

```
SIMPLER_CODE_GENERATION_SPECIFICATION = {

    (Symbol.Instruction , Symbol.ID, Symbol.ASSIGN, Symbol.Arithmetic)
          : [
        "8d6d0000",       # lw $t5, 0($t3)
        "ad2d0000"        # sw $t5, 0($t1)
    ],

    (Symbol.Arithmetic , Symbol.CONST)
                                                         : [
        "ad090000"        # sw $t1, 0($t0)
    ],
    (Symbol.Arithmetic , Symbol.Var )
                                                            : [
        "8d2d0000",       # lw $t5, 0($t1)
        "ad0d0000"        # sw $t5, 0($t0)
    ],
}
```

Listing 3.11: Example of LWVCG SIMPLER Code generation specification

For code generation, we will use Python string interpolation combined with a functional approach to map the initial specifications to a single string representing the Verilog code for the final "scanner", "scanner_tb", "parser", and "parser_tb" files. Listing 3.12 shows an example of a Python code generation function that interpolates Verilog code with Python variables.

3.5 Interpreter design

As stated in Section 3.4, the interpreter will be divided into two modules. Both of them will be running at the default FPGA clock frequency (24 Mhz) One for the scanner and one for the parsing and code generation. However, the frequency of the interpreter's input and output buses will be a bottleneck and cause these modules to operate in a blocking behavior instead of in a pipelined way. That is to say, the parsing module will not start parsing the input until the lexical analyzer is done analyzing all the input tokens, and the parser cannot parse the following token until the instruction output bus is done transmitting all the instructions for the past reduction. Figure 3.9 shows a diagram of this process.

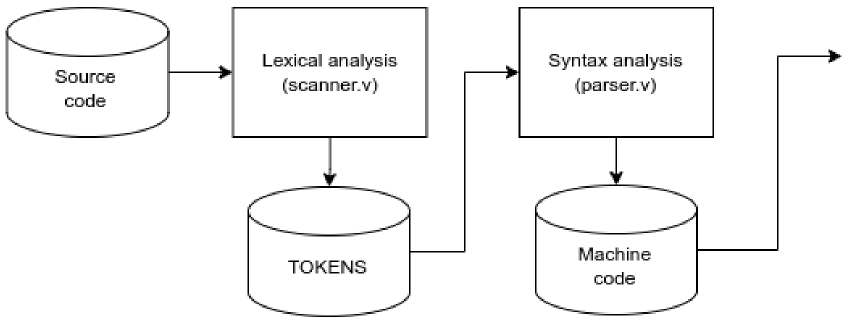


Figure 3.9: Interpreter process

```python
def get_token_encoder(self):
    tokens = ["CONST", "ID", "COMMENT"] + list(self.tokens_dict.keys()
        )
    assings = "".join([f"assign signals[{str(i)}] = {token};\n\t" for i
        , token in enumerate(tokens)])

    for i,t in enumerate(tokens):
        print(i + 1, t)

    return f"""
wire [{len(tokens)}:0] signals;
    {assings}

token_encoder
#(
    .SIGNAL_COUNT({len(tokens)}),
    .ENCODER_OUT_COUNT({ceil(log2(len(tokens)+1))})
)
encoder
(
    .clk(clk),
    .en(en),
    .rst(rst),
    .signals(signals),
    .encoded(token)
); """
```

Listing 3.12: Code generation example

The algorithms used to implement both modules, along with their hardware details, will be explained later.

3.6 Processor design

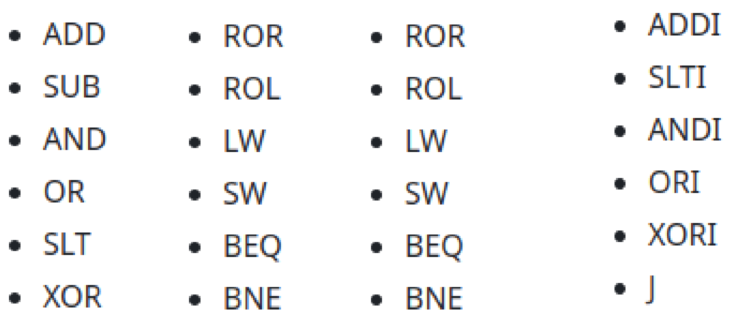
For the processor, we will use an open-source 32-bit MIPS softcore Verilog implementation from [39], running at 6 MHz (due to limitations in the design). Initially, this processor occupied 300% of the ICE40UP5K FPGA, so several optimizations and changes were required:

- **Address space optimizations:** The address space was initially implemented as distributed memory on the FPGA, causing significant overhead in LCU utilization. This was optimized by migrating to the RAM blocks available in the ICE40UP5K FPGA, reducing LCU utilization to 150%.

- **Registers optimizations:** The register space was implemented as distributed memory on the FPGA, which is challenging to optimize since registers typically cannot be migrated to RAM due to delays. To address this, a secondary high-speed clock (4x the system clock) was introduced to drive the register RAM memory. This allowed the registers to be migrated to RAM while maintaining the ability to perform updates within one clock cycle. This optimization reduced LCU utilization to 80%.

- **ROM Memory changes:** The instruction ROM was modified to allow write operations from the instruction bus.

Figure 3.10 shows the subset of instructions from the original MIPS architecture that are supported in this current implementation.



- ADD
- SUB
- AND
- OR
- SLT
- XOR

- ROR
- ROL
- LW
- SW
- BEQ
- BNE

- ROR
- ROL
- LW
- SW
- BEQ
- BNE

- ADDI
- SLTI
- ANDI
- ORI
- XORI
- J

Figure 3.10: @vsilchuk MIPS implementation supported instructions

4 Implementation

This section will review the implementation details in algorithms, software, and hardware of each of the components mentioned in Section 3.

This section covers the implementation details of the algorithms, software, and hardware for each component mentioned in during the design.

We will use Big-O notation to describe the worst-case spatial and time complexities of the algorithms. Since these algorithms are implemented in hardware, time complexity often translates directly to clock cycles and space complexity to LUT usage. For example, a time complexity of $\mathcal{O}(N)$ implies the algorithm will take N clock cycles, and a space complexity of $\mathcal{O}(N)$ indicates that N LUTs will be required if implemented in distributed memory.

4.1 Lexical analysis

4.1.1. Algorithms and data structures used

Regular expressions are the formalism used for lexical specification. Regular expressions can be translated into Non-Deterministic Finite Automata (NFA) and later into Deterministic Finite Automata (DFA). This can be efficiently implemented in software using state machines and graph representations.

The most commonly used algorithm for converting a regular expression into an NFA is "Thompson's Construction Algorithm" [40]. This algorithm recursively creates small NFAs for each part of the regular expression, combining them using epsilon () transitions. These transitions can be later simplified to optimize the automaton size.

NFAs allow multiple transitions for a single state-symbol pair, including -transitions. Simulating an NFA has a time complexity of $\mathcal{O}(2^m \cdot n)$ and a space complexity of $\mathcal{O}(m)$, where n is the number of input symbols and m is the number of states.

The following rules are depicted according to Aho et al. (2007) [1]. The empty expression is converted to the DFA of Figure 4.1.

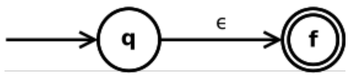


Figure 4.1: Rule applied for emtpy expression

A symbol "a" of the input alphabet is converted to the DFA of Figure 4.2.

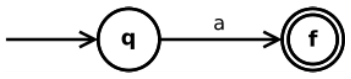


Figure 4.2: Rule applied for a symbol a

In what follows, N(s) and N(t) are the NFAs of the subexpressions s and t, respectively. The union expression "s | t" is converted to the NFA of Figure 4.3.

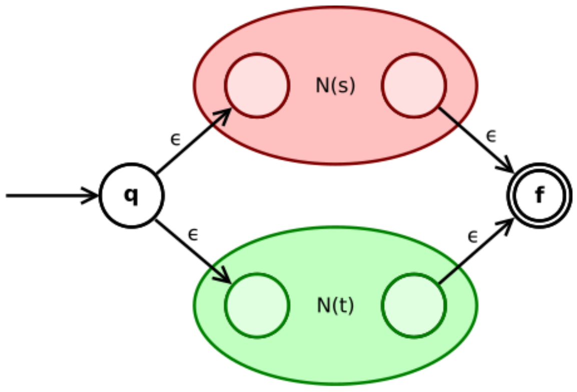


Figure 4.3: Rule applied for s | t

The concatenation expression "st" is converted to the DFA of Figure 4.4.

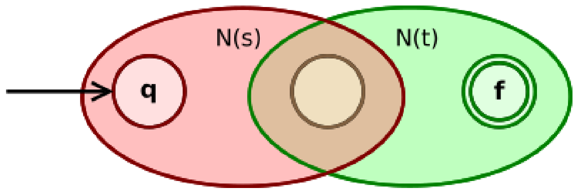


Figure 4.4: Rule applied to concatenation expression

The Kleene star expression "s*" is converted to the NFA of Figure 4.5.

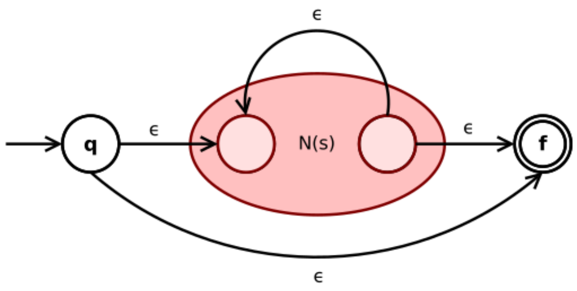


Figure 4.5: Rule applied to "s*" expression

As an example, the regular expression "(ϵ | a*b)" would be converted to the NFA shown in Figure 4.6.

We use the union operator to create a single expression that recognizes all tokens in the language, such as "(if | else | then | [0-9]+... | do | while)", then use Thompson's algorithm to convert this expression into an NFA and finally convert the resulting NFA to a DFA using the Powerset construction algorithm [41].

However, this approach increases the number of possible states to 2^m in the worst case (where m is the number of NFA states). Given the FPGA's 5000 LUT limit, we could only accommodate up to approximately 13 states in the original NFA.

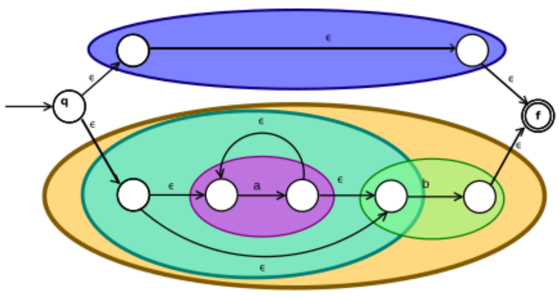


Figure 4.6: NFA resulting of converting the expression "(ϵ | a*b)"

To maintain a time and space complexity of $\mathcal{O}(n)$, a different approach has been followed by dividing expressions into plain keywords (simple concatenation) and more complex expressions (involving Kleene star, quantifiers, or unions).

We can handle complex expressions using four separate expressions: one for IDs "[a-z][A-Z0-9]*", one for constants "[0-9]+", one for blanks "[\n\t]", and then using just one expression for keywords "(if | else | then | while...)". These 4 expressions can be evaluated in parallel.

Figure 4.7 shows the resulting DFAs after applying Thompson's construction algorithm and simplifying epsilon transitions to the ID, CONST and BLACK expressions.

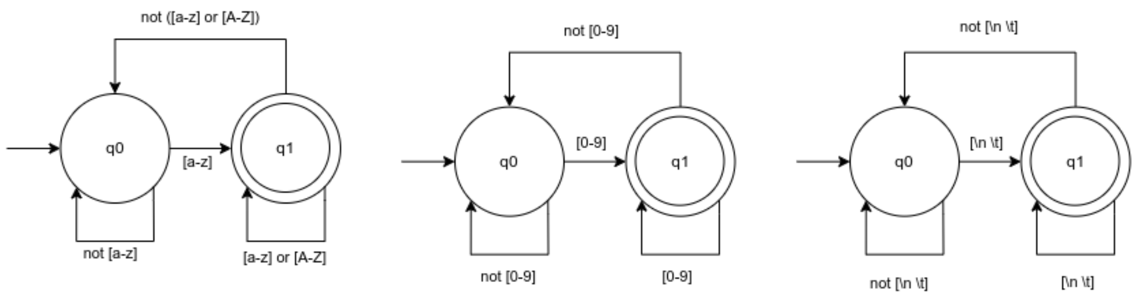


Figure 4.7: DFNs resulting to apply Thomoson's algorithm to IDs, CONSTs and BLANK expressions

For keywords, since they consist of simple symbol sequences, we can use a Trie (Prefix Tree) [42] to identify them. A Trie is a k-ary search tree used to locate specific keys, often strings, by linking nodes based on individual characters rather than the entire key. To access a key (e.g., for retrieval, modification, or removal), the Trie is traversed depth-first, following the links representing each character.

Building a Trie is straightforward: we iterate through all keywords while traversing the tree. If we find a missing transition new node with the corresponding symbol is added. Figure 4.8 shows an example of a Trie for the keywords "if", "else", ";", and "skip".

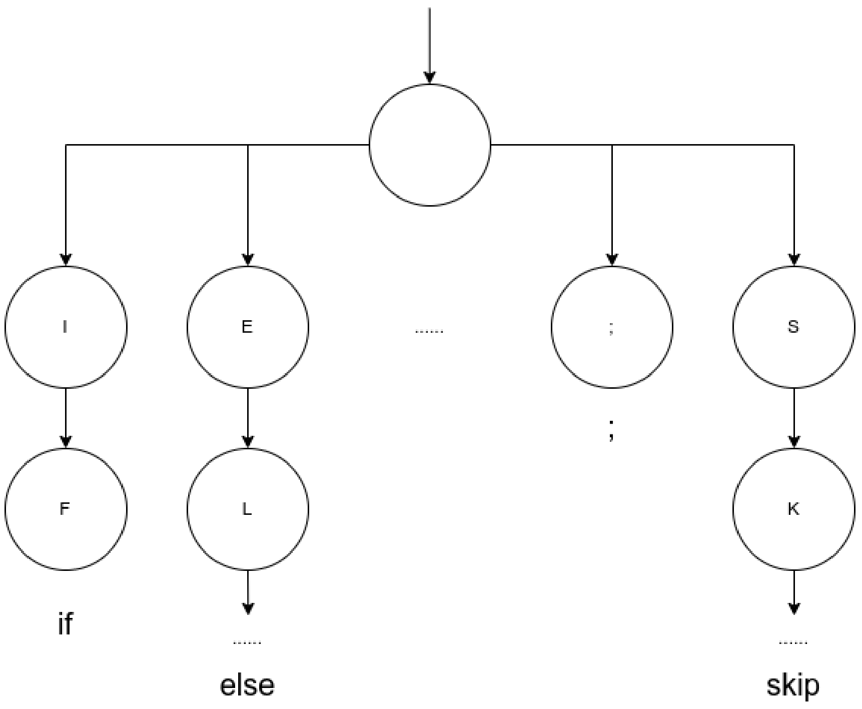


Figure 4.8: Example of Trie for the keyworkds "if", "else", ";" and "skip"

In this way, through the use of a Trie and 3 DFAs (assuming parallel execution of all of them), we can perform the lexical analysis for the language with time complexity of $\mathcal{O}(n)$ and space complexity of $\mathcal{O}(m1 + m2 + m3 + m4)$ with n being the number of characters in the source code and m1, m2, m3 and m4 being the number of states of the DFAs and nodes in the Trie.

Note that all the algorithms mentioned in this section to build the Trie and DFAs will not run on the FPGA; they will run on the LWVCG Python framework built by us. After this, each node will be implemented as a Verilog class. Therefore, only the Trie search and DFA simulation Algorithms will run on the FPGAs.

4.1.2. Hardware implementation

As stated in Section 3, the 100 kHz I2C bus for the source code requires polling the communication channel until a new character is received. Consequently, the workflows for source code upload and lexical analysis are depicted in Figure 4.9.

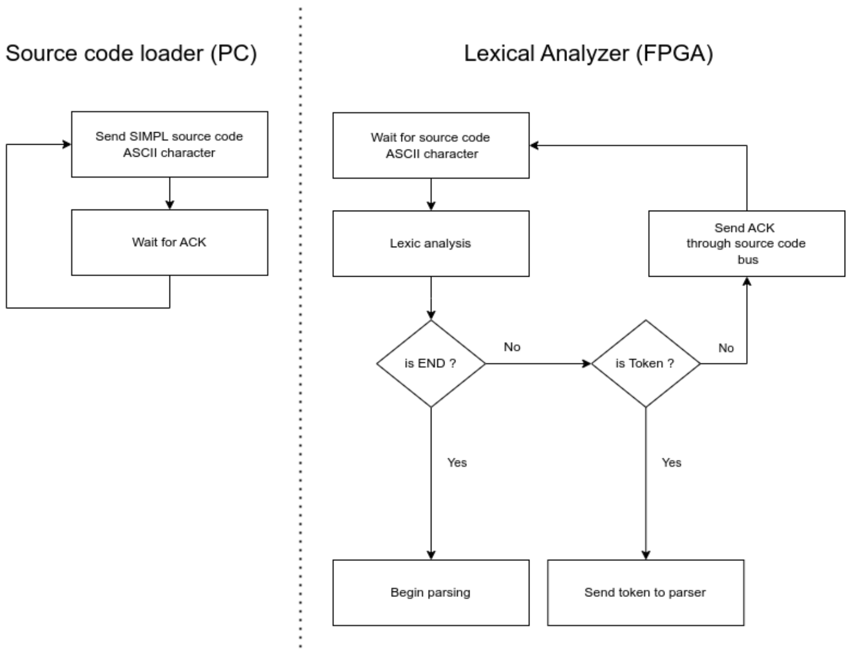


Figure 4.9: Communication workflow with the lexical analyzer

If the received character is the end character "\x00", the finished flag will be raised. Otherwise, an ACK message will be sent to the code upload script, indicating that the FPGA is ready for the next character. During lexical analysis, all detected tokens will be passed to the syntactic analyzer. Note that some steps in this workflow will be executed in parallel on the FPGA.

Figure 4.10 shows the hardware block diagram of the lexical analyzer.

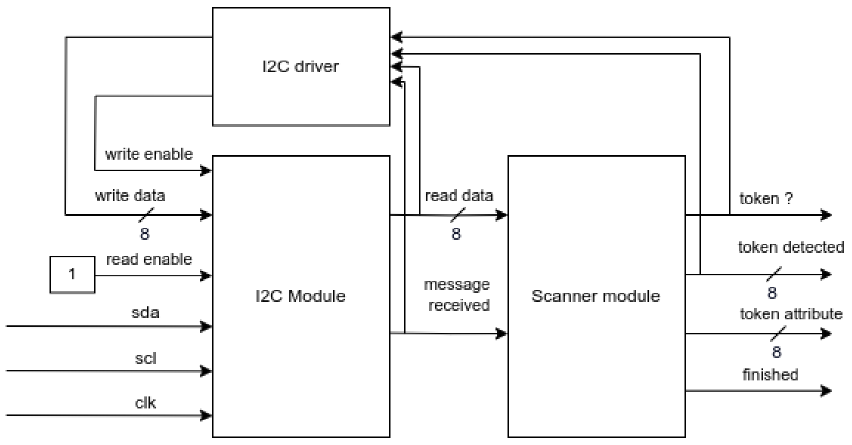


Figure 4.10: Lexical analyzer's hardware block diagram

The I2C driver will implement the state machine shown in Figure 4.9 to manage the reception and transmission of messages to the code upload script. Due to reported issues with the ICE40UP5K's built-in I2C module, a custom soft-core I2C master module was built in Verilog following the NXP specification [43].

To enhance time efficiency and optimize hardware usage, the scanner module has been divided into four separate modules: the IDs DFA, CONSTs DFA, BLANKs DFA, and the keywords Trie. Figure 4.11 shows the hardware block diagram of the scanner module.

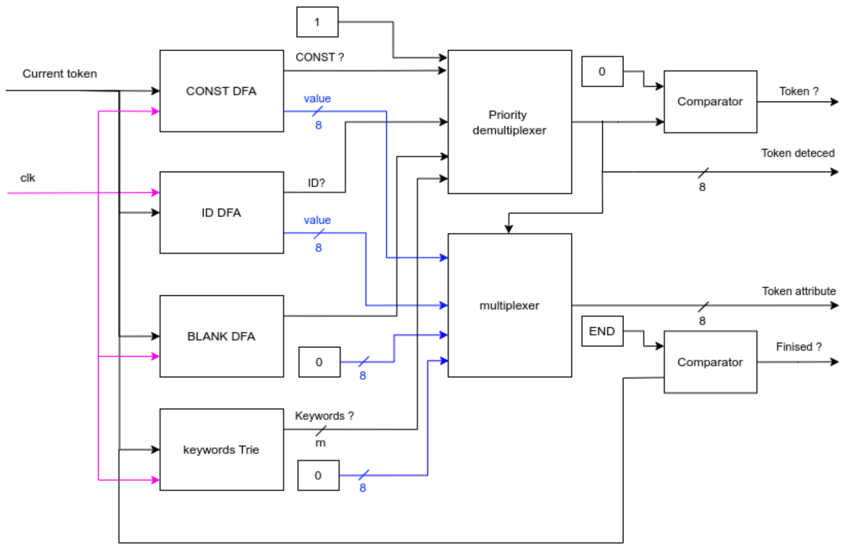


Figure 4.11: Scanner module's hardware block diagram

As can be seen, these four modules execute in parallel, and the correct output is chosen using a priority demultiplexer. This demultiplexer allows us to, from a set of x24 1-bit flags, get which of the tokens is active in the form of an 8-bit unsigned identifier, whenever two of the flags are on (in the case of for example the analysis of the string "if" which will turn on the IF token and the ID DFA) the order in which the inputs are specified allows us to choose priority between them.

The multiplexer uses this identifier to choose which attribute will be passed to the syntactic analyzer as the current token attribute.

Note that the I2C message received flag is used as the clock signal for the lexical analyzer, preventing a single input from being analyzed twice.

The Verilog code for these 4 modules will be automatically generated using the LWVCG Python framework, which is based on the specification of the SIMPLER language.

4.1.3. LWCVG Verilog code generation

Figure 4.12 shows the workflow followed by LWVCG for the generation of the "ID_DFA.v", "CONST_DFA.v", "BLANK_DFA.v" and "keywords.v" files stated previously.

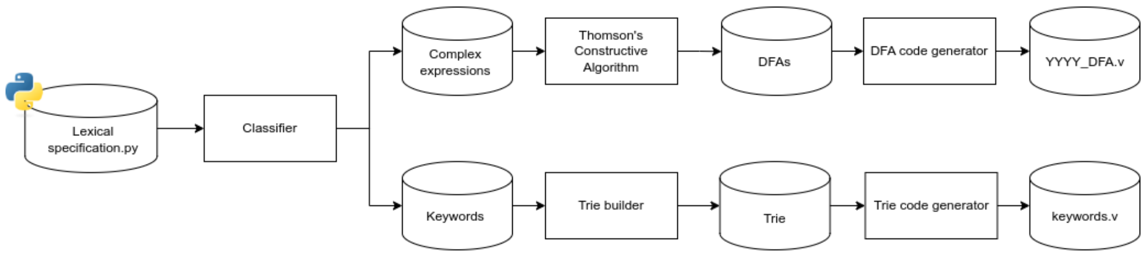


Figure 4.12: LWVCG workflow for generating the scanner

As shown, a classifier function separates regular expressions containing Kleene stars, one-or-more, and union operators from simpler expressions (keywords). Builder functions then construct the respective data structures for each module, and a Verilog code generation module creates the final files based on the specified templates.

Figure 4.13 presents the hardware block diagram for the implementation of a Trie.

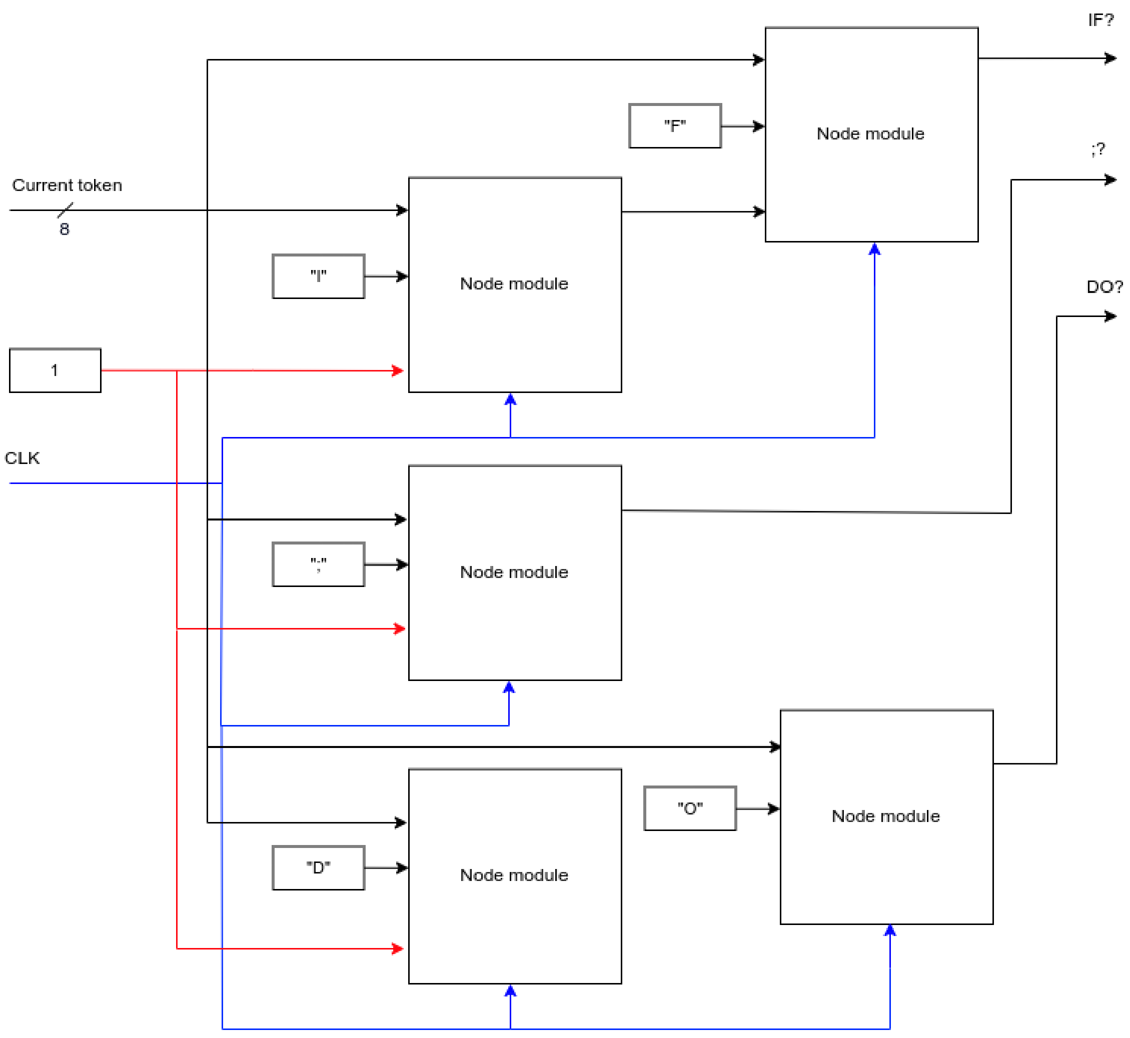


Figure 4.13: Trie's implementation hardware block diagram

Each node in the Trie is represented by a Node module. These modules have an 8-bit parameter for the character that triggers the transition between parent and child and a 1-bit input flag indicating whether the parent is active. Each node uses a 1-bit register to store its state, and the current character is passed as an 8-bit argument to each node. The root nodes (depth 0) receive a 1 for the "isParentOn" parameter. Figure 4.14 shows the hardware block diagram of a Trie's node.

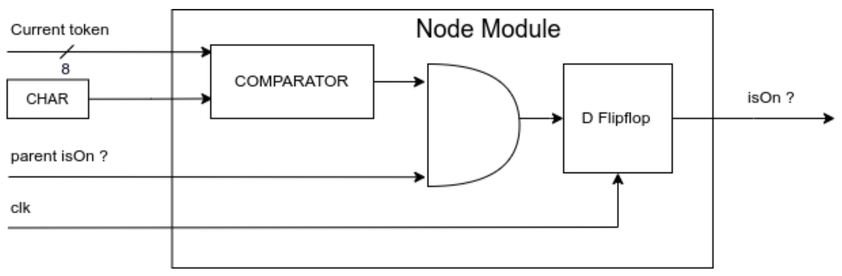


Figure 4.14: Trie's node hardware block diagram

As for the "YYY_DAF.v" files, the DAF is implemented using a basic Verilog implementation of a state machine. With the small manual modification (for the ID and CONSTS modules) extra logic was added in order to calculate the attribute for both tokens. The attribute for the IDs consists of a basic hashing function: an 8-bit XOR of all the

ASCII characters contained in the string. In this way, we limit the size of the attributes passed to the syntactic analyzer and avoid big strings being passed along to the interpreter. For the CONSTS, the value consists of the unsigned 8-bit value of the constant specified in the program.

Listing 4.1 shows the Verilog template used to implement the mentioned state machines.

```
module {NAME} (
    input        clk,
    input        rst,
    input  [7:0] current_token,
    output       is_on,
    output       attribute_value
);

    localparam FINAL_STATE = 0;
    reg [7:0] current_state = 0;
    always @(posedge clk or posedge rst) begin
        if(rst == 1) begin
            current_state = 0;
        end else begin
            {ALL_STATE_TRANSITIONS}
            if(state == FINAL_STATE ) begin
                is_on = 1:
            end

        end
    end

end module
```

Listing 4.1: Verilog basic state machine implementation

4.2 Syntax analysis

4.2.1. Used algorithms and data structures

The output of the syntax analysis is the abstract syntax tree from the given terminals (leaves) and the start symbol of the grammar. Figure 4.15 shows the syntactic specification of a small language and the parse tree for the expression "a * (a+b)".

As it can be seen, given the grammar $\mathcal{G} = (\mathsf{N}, \mathsf{T}, \mathsf{P}, S)$ with a set N of *non-terminal symbols*, a set T of *terminal symbols* (with $\mathsf{N} \cap \mathsf{T} = \emptyset$), a set P of *grammar productions* $\alpha \rightarrow \beta$ (with $\alpha \in \mathsf{N}$ and $\beta \in (\mathsf{N} \cup \mathsf{T})^*$), and a start symbol $S \in \mathsf{N}$, a parse tree, also known as a derivation tree, is defined as follows:

- The root is labeled with the start symbol S.
- Each leaf is labeled with a symbol from $\mathsf{T} \cup \{\varepsilon\}$.
- Each internal node is labeled with a symbol from N.
- Let $A \in \mathsf{N}$ be the label of a node, and let $x_1, ..., x_n$ be the labels of its child nodes (from left to right). Then: $A \rightarrow x_1 \cdots x_n \in \mathsf{P}$.

$$
\begin{array}{ll}
S \to S + T & (1) \\
S \to T & (2) \\
T \to T * F & (3) \\
T \to F & (4) \\
F \to (S) & (5) \\
F \to a & (6) \\
F \to b & (7)
\end{array}
$$

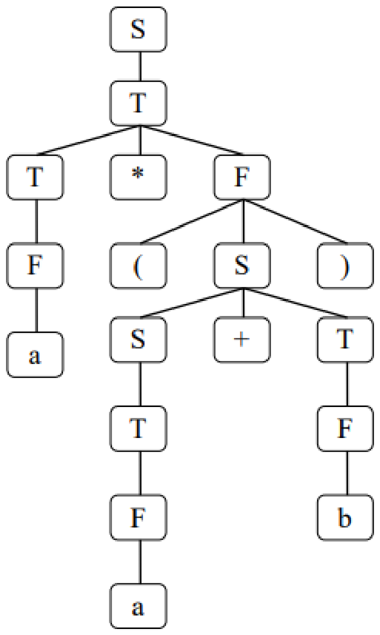


Figure 4.15: Example of language and parse tree for the expression "a * (a + b)"

In the derivation process, if the leftmost non-terminal symbol is always replaced, a leftmost derivation has occurred. Conversely, replacing the rightmost non-terminal results in a rightmost derivation. A parse tree has a unique left parse and a unique right parse.

As outlined in Section 2, the formalism used in syntactic analysis is a pushdown automaton, which is used to recognize context-free languages. A pushdown automaton differs from classical automata in that it has a stack where symbols are pushed and popped. The top symbol on the stack, along with the input, determines the next transition.

A pushdown automaton is defined as $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, z_0, \mathcal{F})$, where:

- Q is a finite set of states.
- Σ is the input alphabet (a finite set of symbols).
- Γ is the stack alphabet (a finite set of stack symbols).
- δ is the transition function:
- q_0 is the initial state.
- z_0 is the initial stack symbol.
- $\mathcal{F}$ is the set of accepting states.

Figure 4.16 shows the structure of a pushdown automaton.

```
Succesors(Program)       → {  Succesors(Instruction)\rightarrow ,  ; ,  )  }
Succesors(WhileDo)       → {  (  }
Succesors(WhileBody)     → {  ,;,)Succesors(While)\rightarrow
    !,ID,CONSTSuccesors(IfThen)\rightarrow(Successors(Else)\rightarrow ,  ; ,  )  }
Successors(IfBody)       → {  else  }
Successors(Arithmetic)   → {  +,  −,  ∗,  =,  <=,  ,;,),do,then,||Successors(Boolean)\
    rightarrow ,  ; ,  ) ,  do,  then ,  ||  }
Successors  (Var)        → {  +,  −,  ∗,  =,  <=,
```

Listing 4.2: Successors function applied to SIMPLER grammar

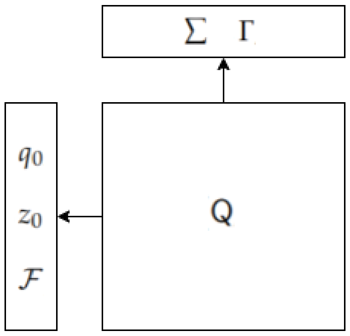


Figure 4.16: Structure of a Pushdown Automaton

The goal is to find an algorithm to construct a pushdown automaton from our syntax specification to recognize the language. From this recognition process, we can derive the list of reductions applied, allowing us to build the corresponding syntax tree. The recognition process can be either top-down or bottom-up, depending on the reduction order.

During syntax analysis, non-determinism may arise, such as when multiple rules exist for the same non-terminal symbol (e.g., A → a1 | a2 | ... | an). This can be addressed using backtracking, tabular, or deterministic algorithms. Backtracking and tabular methods have time complexities of $\mathcal{O}(2^n)$ and $\mathcal{O}(n^2)$, respectively, and are not considered due to inefficiency. Deterministic algorithms, which achieve linear time complexity, are more suitable but apply only to a subset of CFGs.

Although these algorithms are more complex, the fact that they don't run at runtime makes this acceptable. Deterministic algorithms rely on two key functions: firsts(x) and successor(x). Given a non-terminal symbol X, firsts(x) returns the set of terminal symbols that can appear in any reduction of X, while successor(x) returns the set of non-terminal symbols that can follow X in any language string.

For brevity, we will not detail the algorithms used to calculate these functions. However, Listings 4.2 and 4.3 display the results of applying these functions to the non-terminal symbols of the SIMPLER grammar.

To perform the intended bottom-up analysis, we will use an LR parser. This parser operates as a pushdown automaton, as illustrated in Figure 4.17.

```
Firsts(Program)      → { IF, ID, while, Print, Skip }
Firsts(Instruction)  → { IF, ID, while, Print, Skip }
Firsts(WhileDo)      → { While }
Firsts(WhileBody)    → { ( }
Firsts(While)        → { while }
Firsts(IfThen)       → { if }
Firsts(Else)         → { else }
Firsts(IfBody)       → { ( }
Firsts(Arithmetic)   → { COSNT, ID }
Firsts(Boolean)      → { !, CONST, ID }
Firsts(Var)          → { ID }
```

Listing 4.3: Firsts function applied to SIMPLER grammar

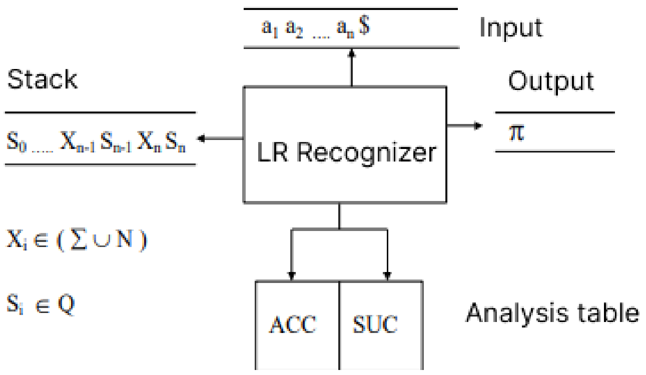


Figure 4.17: Enter Caption

The transition function of an LR recognizer is given by a parsing table. This parsing table is divided into two subtables:

A subtable called "action" determines the action (shift or reduce) that the automaton should perform. In the case of an LR(1) parser, this subtable is defined as follows:

$$Qx(\sum \cup \{\$\}) \to \{des, red_i, err, accept\}$$

where "des" indicates that the action to be performed is a shift of a symbol from the input string to the stack, red_i means that the corresponding reduction for production number r should be applied, err indicates that an error has occurred (the analyzed string does not belong to the language recognized by the automaton), and "accept" indicates that the analysis has concluded successfully, recognizing the string.

Also, a subtable called "goto" indicates the state to which the automaton should transition after performing each shift or reduction action. This subtable is defined as follows:

$$Q \times (\sum \cup N) \to Q \cup \{err\}$$

The four types of actions in the action subtable applied to an LR(1) recognizer automaton produce the following movements:

Displacement: This action is executed if $acc[S_n, a_i] = des$ and $suc[S_n, a_i] = S_{n+1}$. Through this action, the symbol from the input string (a_i) is pushed onto the stack, and following it, the state to which the automaton transitions (s_{n+1}) is also pushed onto the stack.

Reduction: This action is executed if $acc[S_n, a_i] = red - j$ y $suc[S_k, A] = S_{n+1}$ with j being the production : $A \rightarrow X_{k+1}X_{k+2} + ... + X_n$. Through this action, the right-hand side of a production and the states following each symbol $(X_{k+1}X_{k+2} + ... + X_n)$ are popped from the stack, and the non-terminal from the left-hand side of the production (A) is then pushed onto the stack.

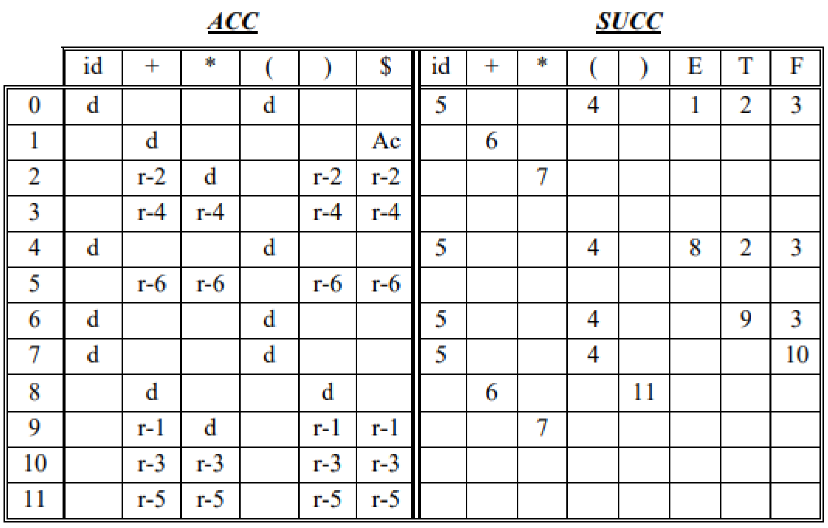
Accept: The final configuration of the automaton will be $(S_0, X_n, S_n, \$, \pi)$ if $acc[S_n, \$] = accept$. Through this action, parsing of the source code of the language will be finished.

Error: The input string does not belong to the language recognized by the automaton. Through this action, a syntax error will be raised in the interpreter and sent back to the source code bus.

Figure 4.18 shows an example of a bottom-up SLR(1) (type of LR analysis) analysis table for the Grammar previously shown in 4.15.

Each of the prefixes of the rightmost sentential forms that can appear on the stack during shift-reduce parsing is called a viable prefix.

The simplest form of deterministic bottom-up parsing is LR(0) parsing. To obtain the parsing tables (action and goto) that determine the transition function of the LR(0) parser, we will construct the recognizer automaton for viable prefixes mentioned previously. To build this automaton, we will start with the definition of an "LR(0) item." Using two functions (closure and goto) applied to sets of LR(0) items; we will gradually construct the recognizer automaton for viable prefixes.



	ACC						SUCC							
	id	+	*	(	)	$	id	+	*	(	)	E	T	F
0	d			d			5			4		1	2	3
1		d				Ac		6						
2		r-2	d		r-2	r-2			7					
3		r-4	r-4		r-4	r-4								
4	d			d			5			4		8	2	3
5		r-6	r-6		r-6	r-6								
6	d			d			5			4			9	3
7	d			d			5			4				10
8		d			d			6			11			
9		r-1	d		r-1	r-1			7					
10		r-3	r-3		r-3	r-3								
11		r-5	r-5		r-5	r-5								

Figure 4.18: Example of SLR(1) analysis table

An LR(0) item is a production of the grammar with a marker (dot) placed between the symbols on its right-hand side.

The dot in an LR(0) item separates the part already parsed (on the stack) from the part yet to be parsed (remaining input). For example, the LR(0) item [E → E + . T] from the grammar in Figure 4.15 indicates that "E +" is on the stack, and the input is expected to match a sequence of terminals w such that $T \rightarrow^* w$.

The algorithm in Listing 4.4 defines the closure operation for a set I of LR(0) items. The closure of I includes all items in I, and if any item has a non-terminal (B) immediately to the right of the dot, new LR(0) items are added. These new items are the productions of B with the marker at the beginning of their right-hand side. This process repeats until no new items can be added to the set.

```
Function Clausure(I) // For LR(0) items
Clausure(I) := I;
repeat
    foreach [A → a . B b] in Clausure(I) do
        foreach (B → y ) in P: [B → . y ] in Clausure(I) do
    Clausure(I) := Clausure(I) U {[B → .y ]};
until no more elements are added to Clausure(I);
```

Listing 4.4: Clausure operation algorithm

```
Function Successor(I , X) // For LR(0) items
J := ∅;

foreach [A → a.xb] in I do
    J = J U {[A → ax.b]};

sucesor (I, X) := Clausure (J)
```

Listing 4.5: Successor operation algorithm

The successor operation for a set I of LR(0) items, for a symbol $x \in (N \cup \Sigma)$, is defined by the algorithm in Listing 4.5. In this algorithm, it can be seen that the successor of a set of LR(0) items I for a symbol x is obtained by finding the LR(0) items in I that have the symbol x immediately to the right of the dot. These items are selected, the dot is moved to the right of the symbol x, and the closure of these new items is calculated.

Starting from the initial item $[S' \rightarrow .S]$ and using the successor operation, we can compute the canonical collection of LR(0) item sets using Algorithm 4.6. First, the closure of $[S' \rightarrow .S]$ is calculated. New sets of LR(0) items are then obtained by applying the successor operation to this initial set for all grammar symbols. Successors of these new sets are computed for all symbols, and the process continues until no sets remain whose successors have not been calculated. The process eventually terminates because some successor sets will be empty while others will repeat.

Figure 4.19 shows the canonical collection of LR(0) item sets of the grammar shown in Figure 4.15.

```
Function Cannonical(N, ∑, P, S)

C = { Clausure({ [ S   → . S] }) };

repeat
    foreach I not marked in C do
        foreach X in (N U S) do
            if Successor(I, X) ≠ ∅ then
                C := C U Successor(I, X);
            mark I
until no set in C is marked;

Cannonical(N, ∑, P, S) := C;
```

Listing 4.6: Canonical collection of LR(0) item sets calculation

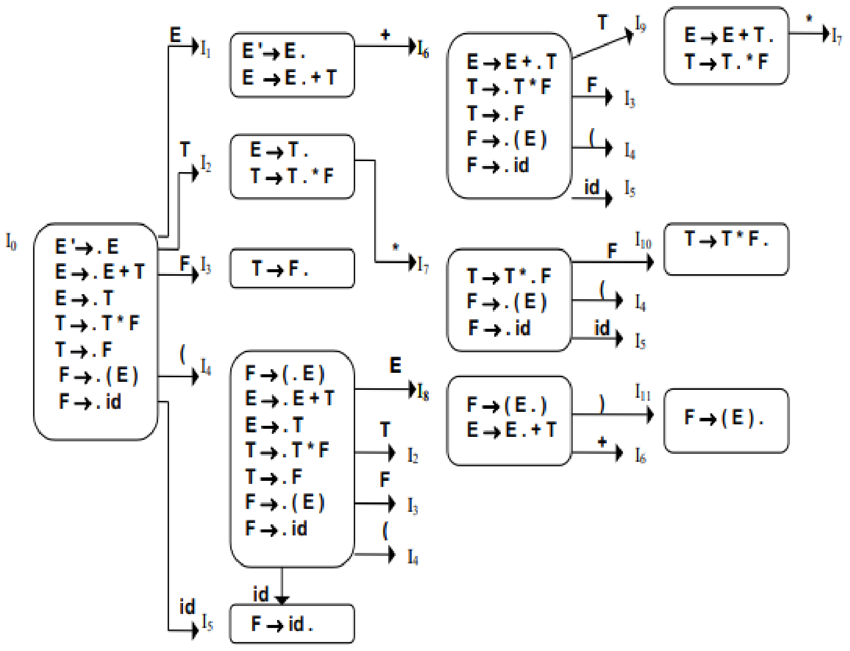


Figure 4.19: Canonical collection of LR(0) item sets example

After implementing these algorithms in Python, we can generate the canonical collection of LR(0) item sets for the SIMPLER grammar. Figure 4.20 displays the results as an automatically generated directed graph.

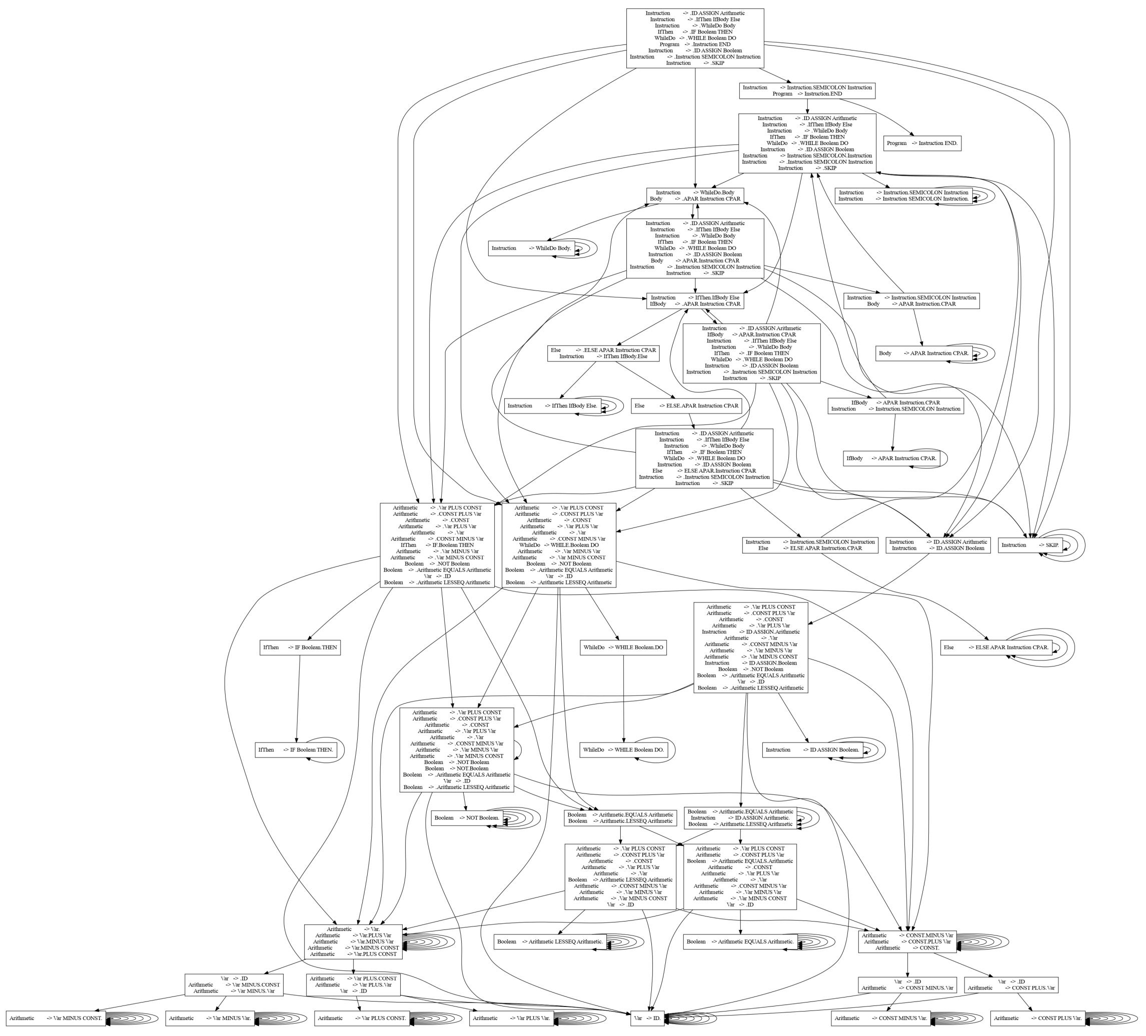
By interpreting each LR(0) item as a state, we can use the canonical collection of LR(0) item sets to build the action and goto tables directly. If an item has a dot at the end ([E $\rightarrow$ T .]), this indicates a reduction. A conflict arises if a set indicates both a reduction and a shift.

There are two approaches to resolving these conflicts: using a more powerful analysis technique, such as SLR(1), or resolving them manually by specifying which action takes precedence. In our case, the conflicts stem from the associative properties of Boolean and Instruction reductions, so we resolve them manually by enforcing left associativity always.

4.2.2. Hardware implementation

As noticed in the previous section, when the scanner detects a token, it raises a flag to the parser for one clock cycle. There are also 8-bit lines carrying the detected token ID and its related attribute. Once scanning is complete, a final flag notifies the parser to start.

Given the low frequency of the source code bus (100 kHz), the simplest approach is to wait until scanning is finished before starting the parsing process. Although pipelining could be implemented, the overhead of waiting until scanning is finished is minimal compared to the bus speed. This approach simplifies debugging while still meeting the proof-of-concept requirements. The parser will follow the workflow shown in Figure 4.21.


Instruction -> .ID ASSIGN Arithmetic
Instruction -> .IfThen IfBody Else
Instruction -> .WhileDo Body
IfThen -> .IF Boolean THEN
WhileDo -> .WHILE Boolean DO
Program -> .Instruction END
Instruction -> .ID ASSIGN Boolean
Instruction -> .Instruction SEMICOLON Instruction
Instruction -> .SKIP
Instruction -> Instruction.SEMICOLON Instruction
Program -> Instruction.END
Instruction -> .ID ASSIGN Arithmetic
Instruction -> .IfThen IfBody Else
Instruction -> .WhileDo Body
IfThen -> .IF Boolean THEN
WhileDo -> .WHILE Boolean DO
Instruction -> .ID ASSIGN Boolean
Instruction -> Instruction SEMICOLON.Instruction
Instruction -> .Instruction SEMICOLON Instruction
Instruction -> .SKIP
Program -> Instruction END.
Instruction -> WhileDo.Body
Body -> .APAR Instruction CPAR
Instruction -> Instruction.SEMICOLON Instruction
Instruction -> Instruction SEMICOLON Instruction.
Instruction -> WhileDo Body.
Instruction -> .ID ASSIGN Arithmetic
Instruction -> .IfThen IfBody Else
Instruction -> .WhileDo Body
IfThen -> .IF Boolean THEN
WhileDo -> .WHILE Boolean DO
Instruction -> .ID ASSIGN Boolean
Body -> APAR.Instruction CPAR
Instruction -> .Instruction SEMICOLON Instruction
Instruction -> .SKIP
Instruction -> IfThen.IfBody Else
IfBody -> .APAR Instruction CPAR
Instruction -> Instruction.SEMICOLON Instruction
Body -> APAR Instruction.CPAR
Instruction -> .ID ASSIGN Arithmetic
IfBody -> APAR.Instruction CPAR
Instruction -> .IfThen IfBody Else
Instruction -> .WhileDo Body
IfThen -> .IF Boolean THEN
WhileDo -> .WHILE Boolean DO
Instruction -> .ID ASSIGN Boolean
Instruction -> .Instruction SEMICOLON Instruction
Instruction -> .SKIP
Else -> .ELSE APAR Instruction CPAR
Instruction -> IfThen IfBody.Else
Body -> APAR Instruction CPAR.
Instruction -> IfThen IfBody Else.
Else -> ELSE.APAR Instruction CPAR
IfBody -> APAR Instruction.CPAR
Instruction -> Instruction.SEMICOLON Instruction
Instruction -> .ID ASSIGN Arithmetic
Instruction -> .IfThen IfBody Else
Instruction -> .WhileDo Body
IfThen -> .IF Boolean THEN
WhileDo -> .WHILE Boolean DO
Instruction -> .ID ASSIGN Boolean
Else -> ELSE APAR.Instruction CPAR
Instruction -> .Instruction SEMICOLON Instruction
Instruction -> .SKIP
IfBody -> APAR Instruction CPAR.
Arithmetic -> .Var PLUS CONST
Arithmetic -> .CONST PLUS Var
Arithmetic -> .CONST
Arithmetic -> .Var PLUS Var
Arithmetic -> .Var
Arithmetic -> .CONST MINUS Var
IfThen -> IF.Boolean THEN
Arithmetic -> .Var MINUS Var
Arithmetic -> .Var MINUS CONST
Boolean -> .NOT Boolean
Boolean -> .Arithmetic EQUALS Arithmetic
Var -> .ID
Boolean -> .Arithmetic LESSEQ Arithmetic
Arithmetic -> .Var PLUS CONST
Arithmetic -> .CONST PLUS Var
Arithmetic -> .CONST
Arithmetic -> .Var PLUS Var
Arithmetic -> .Var
Arithmetic -> .CONST MINUS Var
WhileDo -> WHILE.Boolean DO
Arithmetic -> .Var MINUS Var
Arithmetic -> .Var MINUS CONST
Boolean -> .NOT Boolean
Boolean -> .Arithmetic EQUALS Arithmetic
Var -> .ID
Boolean -> .Arithmetic LESSEQ Arithmetic
Instruction -> Instruction.SEMICOLON Instruction
Else -> ELSE APAR Instruction.CPAR
Instruction -> ID.ASSIGN Arithmetic
Instruction -> ID.ASSIGN Boolean
Instruction -> SKIP.
IfThen -> IF Boolean.THEN
WhileDo -> WHILE Boolean.DO
Arithmetic -> .Var PLUS CONST
Arithmetic -> .CONST PLUS Var
Arithmetic -> .CONST
Arithmetic -> .Var PLUS Var
Instruction -> ID ASSIGN.Arithmetic
Arithmetic -> .Var
Arithmetic -> .CONST MINUS Var
Arithmetic -> .Var MINUS Var
Arithmetic -> .Var MINUS CONST
Instruction -> ID ASSIGN.Boolean
Boolean -> .NOT Boolean
Boolean -> .Arithmetic EQUALS Arithmetic
Var -> .ID
Boolean -> .Arithmetic LESSEQ Arithmetic
Else -> ELSE APAR Instruction CPAR.
Arithmetic -> .Var PLUS CONST
Arithmetic -> .CONST PLUS Var
Arithmetic -> .CONST
Arithmetic -> .Var PLUS Var
Arithmetic -> .Var
Arithmetic -> .CONST MINUS Var
Arithmetic -> .Var MINUS Var
Arithmetic -> .Var MINUS CONST
Boolean -> .NOT Boolean
Boolean -> NOT.Boolean
Boolean -> .Arithmetic EQUALS Arithmetic
Var -> .ID
Boolean -> .Arithmetic LESSEQ Arithmetic
IfThen -> IF Boolean THEN.
WhileDo -> WHILE Boolean DO.
Instruction -> ID ASSIGN Boolean.
Boolean -> NOT Boolean.
Boolean -> Arithmetic.EQUALS Arithmetic
Boolean -> Arithmetic.LESSEQ Arithmetic
Boolean -> Arithmetic.EQUALS Arithmetic
Instruction -> ID ASSIGN Arithmetic.
Boolean -> Arithmetic.LESSEQ Arithmetic
Arithmetic -> .Var PLUS CONST
Arithmetic -> .CONST PLUS Var
Arithmetic -> .CONST
Arithmetic -> .Var PLUS Var
Arithmetic -> .Var
Boolean -> Arithmetic LESSEQ.Arithmetic
Arithmetic -> .CONST MINUS Var
Arithmetic -> .Var MINUS Var
Arithmetic -> .Var MINUS CONST
Var -> .ID
Arithmetic -> .Var PLUS CONST
Arithmetic -> .CONST PLUS Var
Boolean -> Arithmetic EQUALS.Arithmetic
Arithmetic -> .CONST
Arithmetic -> .Var PLUS Var
Arithmetic -> .Var
Arithmetic -> .CONST MINUS Var
Arithmetic -> .Var MINUS Var
Arithmetic -> .Var MINUS CONST
Var -> .ID
Arithmetic -> Var.
Arithmetic -> Var.PLUS Var
Arithmetic -> Var.MINUS Var
Arithmetic -> Var.MINUS CONST
Arithmetic -> Var.PLUS CONST
Boolean -> Arithmetic LESSEQ Arithmetic.
Boolean -> Arithmetic EQUALS Arithmetic.
Arithmetic -> CONST.MINUS Var
Arithmetic -> CONST.PLUS Var
Arithmetic -> CONST.
Var -> .ID
Arithmetic -> Var MINUS.CONST
Arithmetic -> Var MINUS.Var
Arithmetic -> Var PLUS.CONST
Arithmetic -> Var PLUS.Var
Var -> .ID
Var -> .ID
Arithmetic -> CONST MINUS.Var
Var -> .ID
Arithmetic -> CONST PLUS.Var
Arithmetic -> Var MINUS CONST.
Arithmetic -> Var MINUS Var.
Arithmetic -> Var PLUS CONST.
Arithmetic -> Var PLUS Var.
Var -> ID.
Arithmetic -> CONST MINUS Var.
Arithmetic -> CONST PLUS Var.
Figure 4.20: SIMPLER canonical collection of LR(0) item sets

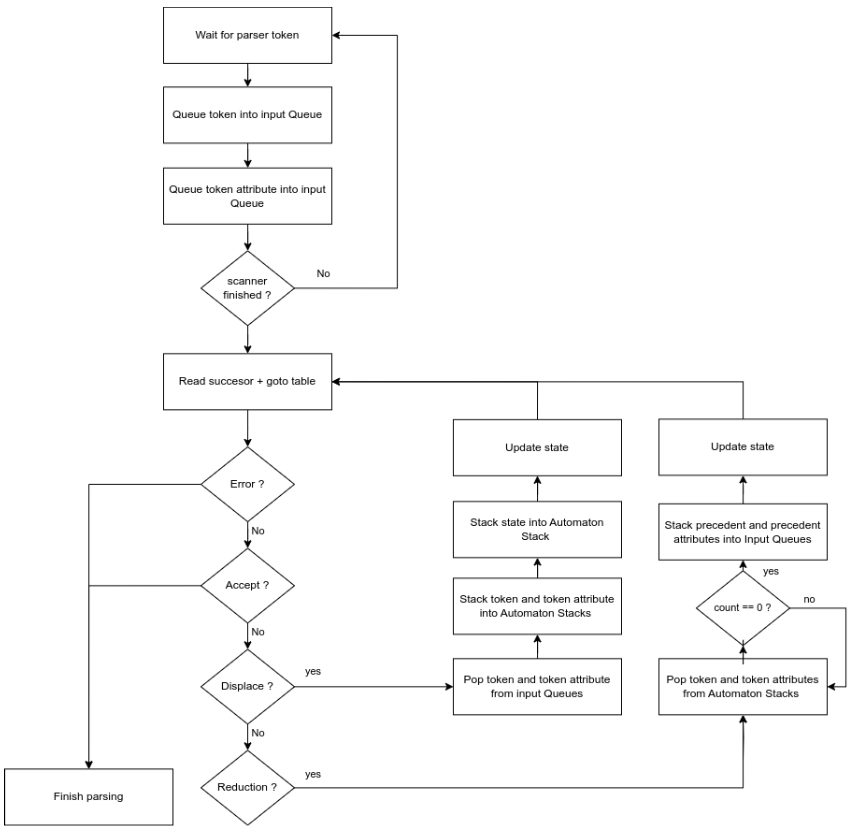

Wait for parser token
Queue token into input Queue
Queue token attribute into input Queue
scanner finished ?
No
Read succesor + goto table
Error ?
No
Accept ?
No
Displace ?
yes
No
Reduction ?
yes
Finish parsing
Update state
Stack state into Automaton Stack
Stack token and token attribute into Automaton Stacks
Pop token and token attribute from input Queues
Update state
Stack precedent and precedent attributes into Input Queues
yes
count == 0 ?
no
Pop token and token attributes from Automaton Stacks

Figure 4.21: Parser's workflow

As shown, once the scanner is finished, the parsing workflow will begin following the steps outlined in Section 4.2.

During lexical analysis, the parser will need to store tokens from the scanner in memory, using a FIFO structure to use them once the scanner is finished. Also, since the pushdown automaton relies on a stack, a custom Verilog double-ended queue (dqueue) has been implemented to function as both a stack and a queue for both of these purposes.

A dqueue [44] is a data structure where elements can be added or removed from both the front (head) and back (tail). For this, the ICE40UP5K's 4 kb embedded RAM blocks have been utilized, allowing for up to 30 dqueues, each with a maximum size of 8 bits x 512 depth.

The dqueue is implemented as a circular buffer, which uses a fixed memory space and two pointers: one for the head and one for the tail. Adding values to the head increments the pointer while removing them decrements it. The tail behaves oppositely, with additions decrementing the pointer and removals incrementing it. Figure 4.22 visualizes this.

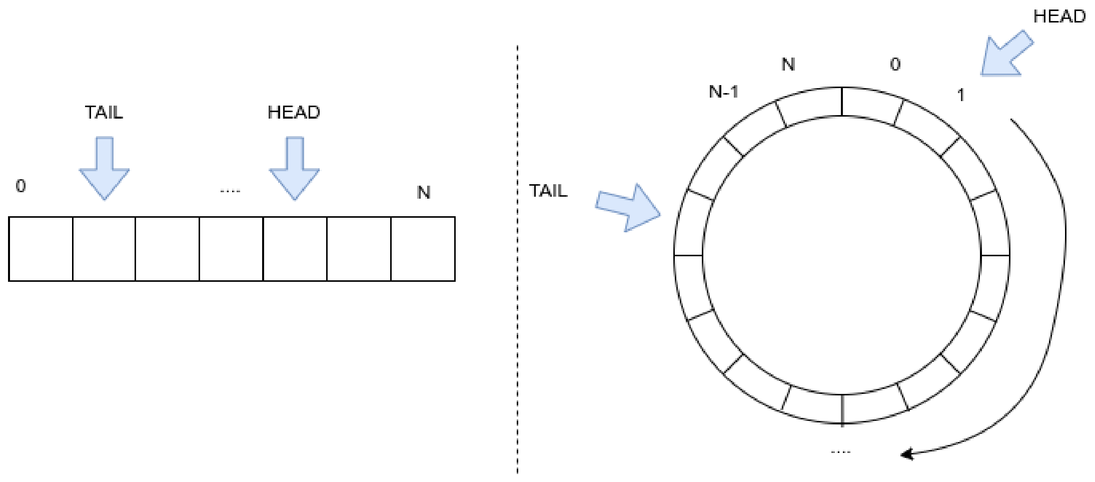


Figure 4.22: Circular buffer visualization

Dqueues support both stack and queue operations, making them versatile for this implementation. Therefore, the Verilog EBR-based dqueue will be crucial moving forward.

To maintain an optimal size for the EBR blocks (512), two 8-bit queues will be used: one for tokens and one for token attributes, rather than a single 16-bit queue. Figure 4.23 shows the hardware block diagram of the parser.

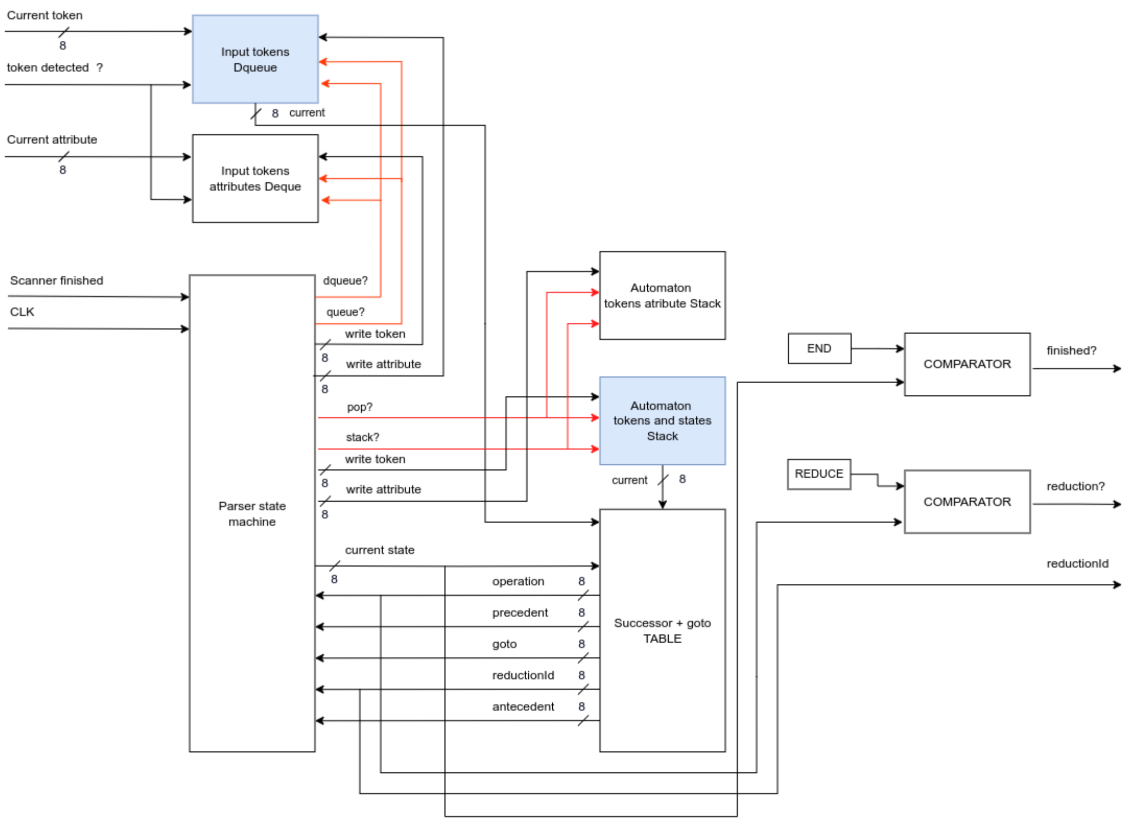


Figure 4.23: Parser hardware block diagram

As shown, four dqueues are used to store the automaton stack and token inputs, limiting the program size to 512 tokens (which can be expanded by allocating more RAM blocks). These RAM blocks are connected to the parser's main state machine, which implements the workflow shown in Figure 4.21. A separate module holds the successor and goto tables as a lookup table, used by the main state machine to determine and apply the next action. A comparator drives the reduction detection signal. The parser's output is a 9-bit bus, with 8 bits representing the detected reduction ID and a flag raised upon reduction detection.

LWVCG will need to automatically generate only the successor/goto table based on the language's syntax specification.

4.2.3. LWVCG Verilog code generation

Figure 4.24 displays the workflow to generate the successor and goto table run by LWVCG.

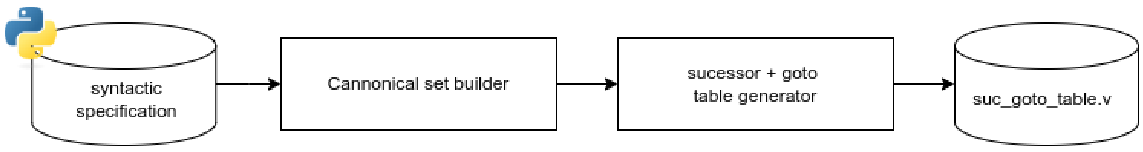
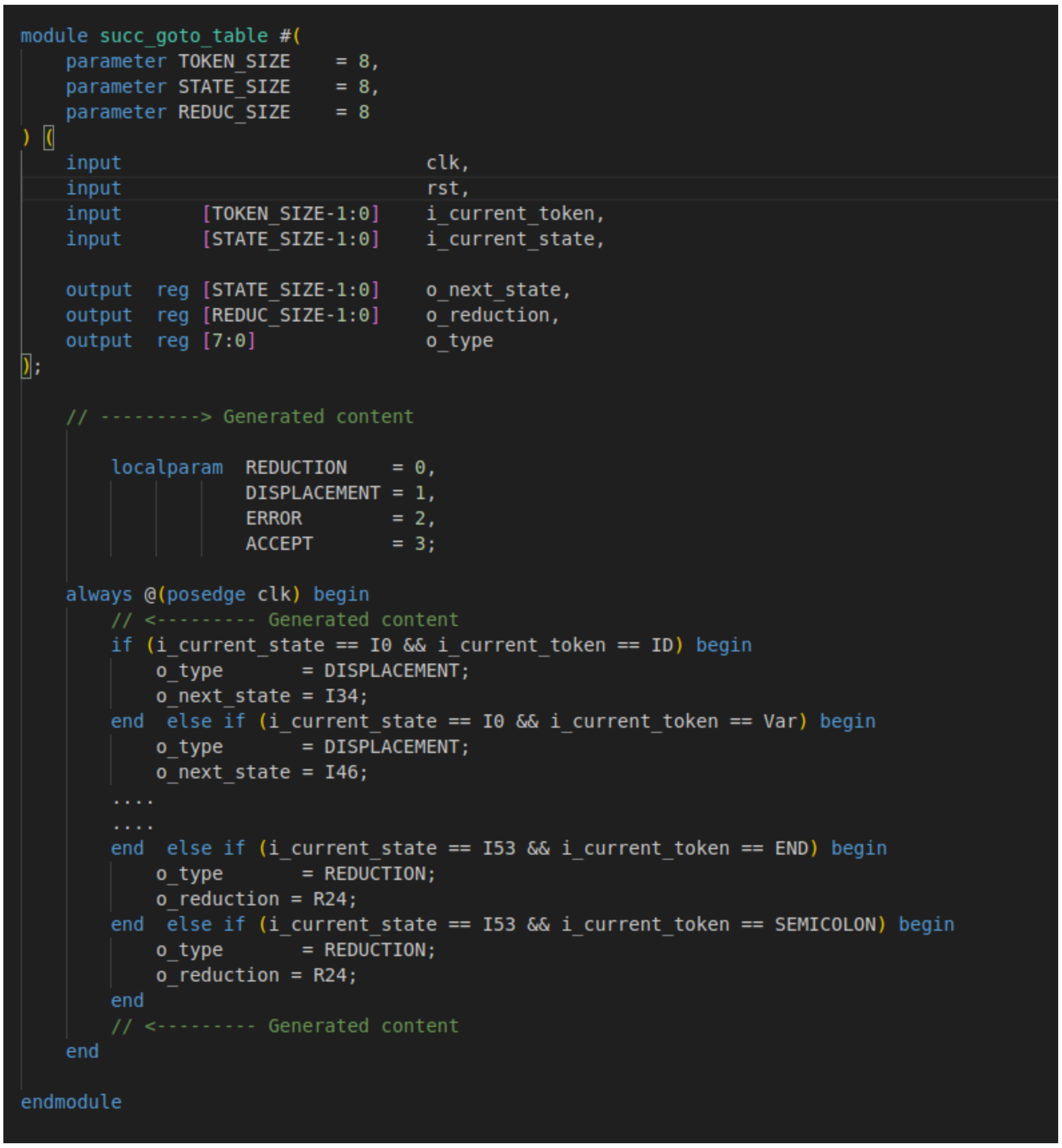


Figure 4.24: LWVCG workflow to generate parser

As shown, a single file is generated containing a module that takes the top of the stack and input queue and returns the corresponding action, reduction ID (if applicable), precedent, and antecedent. Although this lookup table could be implemented using an EBR block, the canonical collection of LR(0) sets for SIMPLER contains 50 elements. To implement this table in memory, we would need approximately 14 kb (50 x 35 tokens x 8 bits). Since most cells in the table will be empty, we use the approach shown in Figure 4.25.



```
module succ_goto_table #(
    parameter TOKEN_SIZE    = 8,
    parameter STATE_SIZE    = 8,
    parameter REDUC_SIZE    = 8
) (
    input                           clk,
    input                           rst,
    input       [TOKEN_SIZE-1:0]    i_current_token,
    input       [STATE_SIZE-1:0]    i_current_state,

    output  reg [STATE_SIZE-1:0]    o_next_state,
    output  reg [REDUC_SIZE-1:0]    o_reduction,
    output  reg [7:0]               o_type
);

    // ---------> Generated content

        localparam  REDUCTION    = 0,
                    DISPLACEMENT = 1,
                    ERROR        = 2,
                    ACCEPT       = 3;

    always @(posedge clk) begin
        // <--------- Generated content
        if (i_current_state == I0 && i_current_token == ID) begin
            o_type       = DISPLACEMENT;
            o_next_state = I34;
        end  else if (i_current_state == I0 && i_current_token == Var) begin
            o_type       = DISPLACEMENT;
            o_next_state = I46;
        ....
        ....
        end  else if (i_current_state == I53 && i_current_token == END) begin
            o_type       = REDUCTION;
            o_reduction = R24;
        end  else if (i_current_state == I53 && i_current_token == SEMICOLON) begin
            o_type       = REDUCTION;
            o_reduction = R24;
        end
        // <--------- Generated content
    end

endmodule
```

Figure 4.25: Implementation of successor and goto table in Verilog

With this approach, we use if-else statements in Verilog to implement the lookup table. Instead of consuming RAM, we use distributed memory, reducing the table size from 14 kb to 2 kb (a 700% reduction), with only 225 if-else statements compared to the original 50 x 25 entries. This implementation will be synthesized by Verilog into a multiplexer that selects the table outputs based on the specified logic.

4.3 Semantic analysis

As outlined in the design chapter, attribute grammars are the formalism used for semantic analysis. Semantic analysis and operations are embedded into the parser. Before applying a reduction, the attributes of both the right- and left-hand sides of the reduction (stored in the attribute queues) are used to compute new attributes. Once calculated, the new attribute is pushed into the attribute input queue.

4.3.1. Hardware implementation

Figure 4.26 shows in green the modifications to the parser workflow needed in order to implement the semantic analysis and semantic operations.

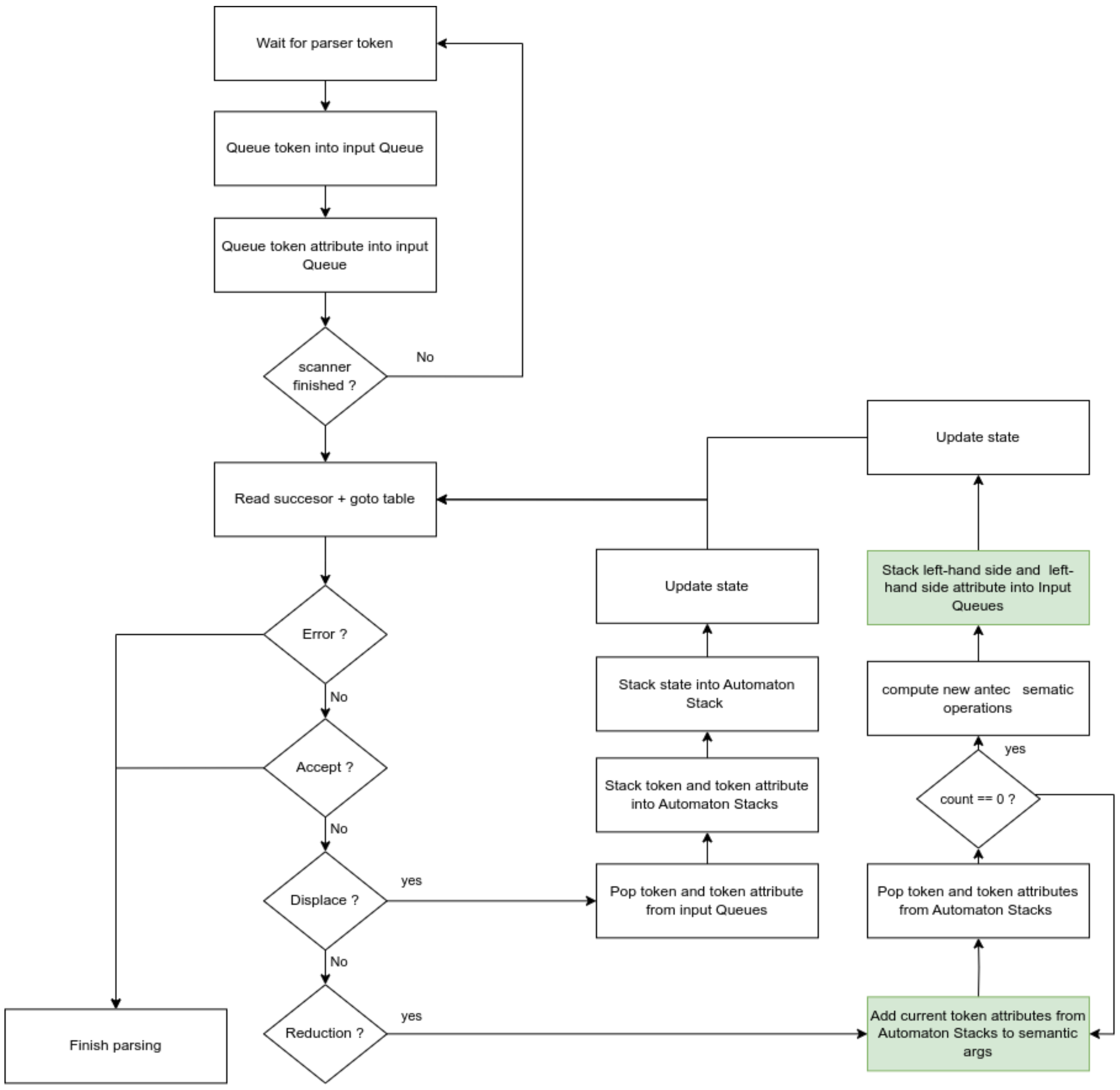

Wait for parser token
Queue token into input Queue
Queue token attribute into input Queue
scanner finished ?
No
Read succesor + goto table
Error ?
No
Accept ?
No
Displace ?
yes
No
Reduction ?
yes
Finish parsing
Update state
Stack state into Automaton Stack
Stack token and token attribute into Automaton Stacks
Pop token and token attribute from input Queues
Update state
Stack left-hand side and left-hand side attribute into Input Queues
compute new antec sematic operations
yes
count == 0 ?
Pop token and token attributes from Automaton Stacks
Add current token attributes from Automaton Stacks to semantic args

Figure 4.26: Modifications to the parser workflow

Figure 4.27 shows the modifications needed to the parser hardware block diagram in order to implement these changes.

A new hardware module has been introduced to handle semantic operations and analysis. This module takes a 32-bit bus as input, consisting of four 8-bit buses representing the attributes of the current reduction (as reductions were limited to 3 symbols in the design section to allow these representations), along with the ID of the current reduction. The module performs semantic operations on these variables and returns the result via a 32-bit output bus.

The semantic operator module is automatically generated by the custom LWVCG Python framework based on the specified semantic rules.

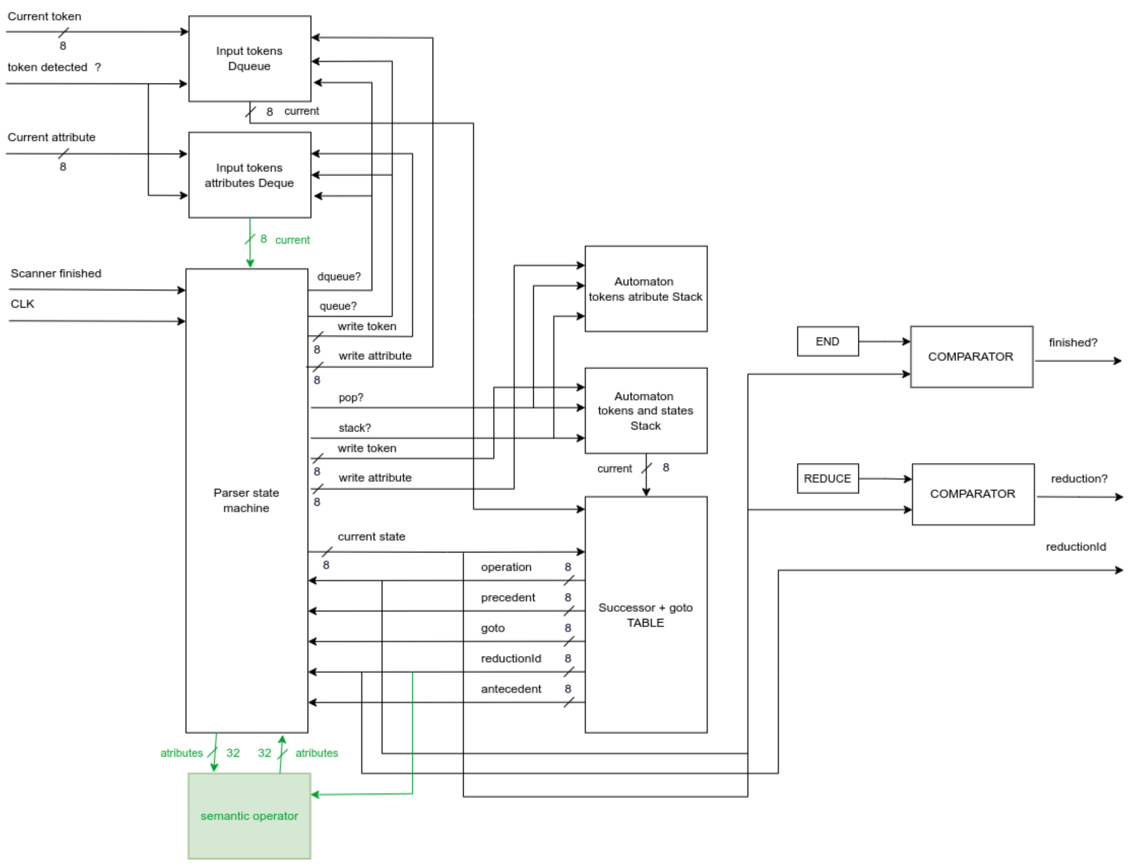


Figure 4.27: Modifications to the parser hardware block diagram

4.3.2. LWVCG Verilog code generation

Figure 4.28 shows the workflow followed by LWVCG to generate the Verilog semantic operator file.

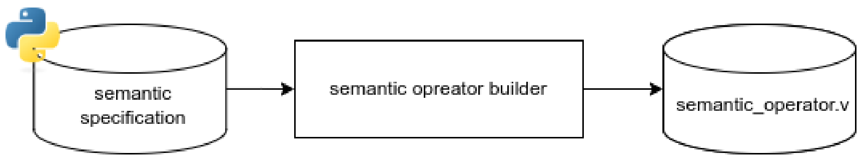
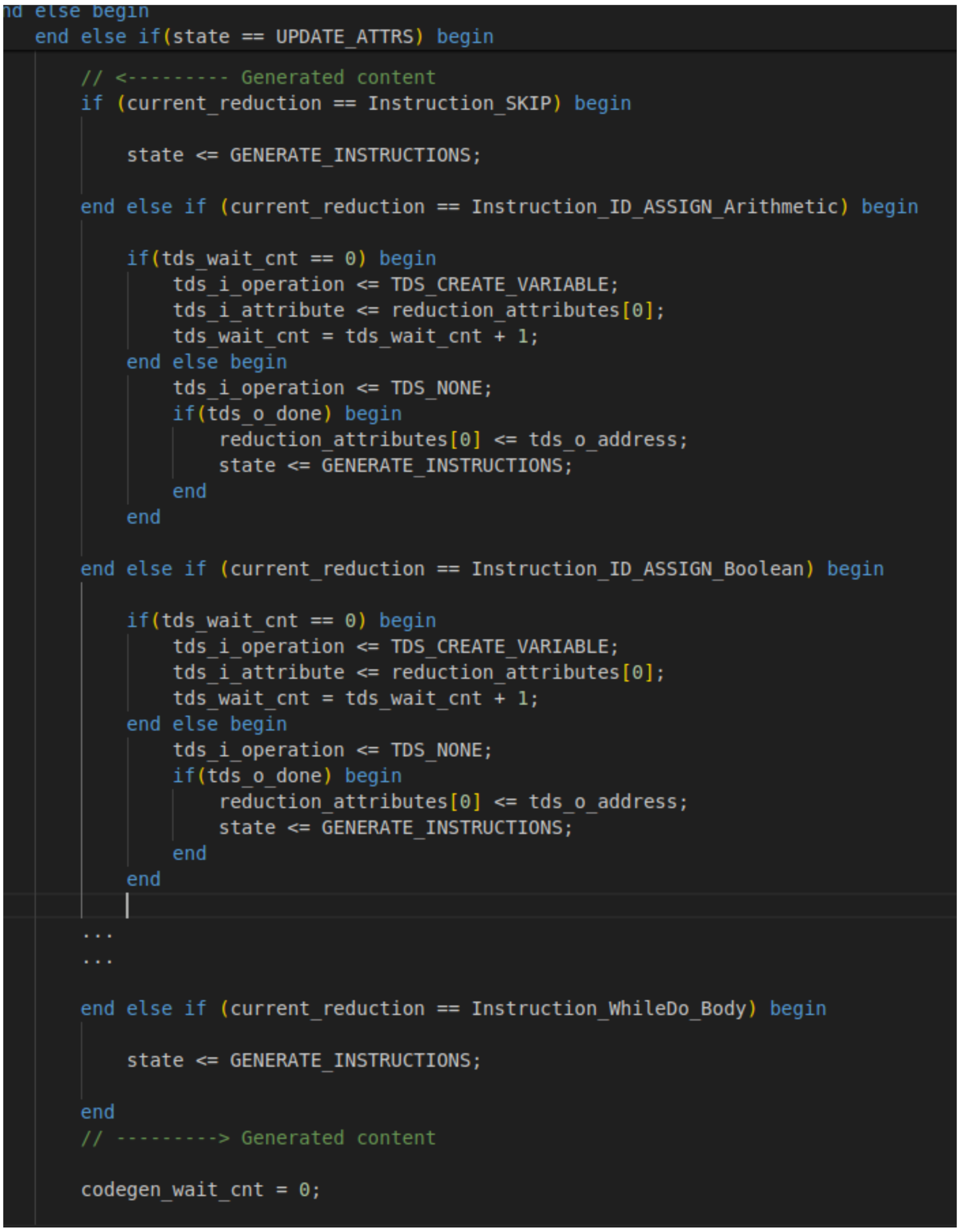


Figure 4.28: Semantic operator code generation workflow

For the semantic specification, a discrete amount of semantic operations has been modeled using a Python data class with 3 attributes (left-hand side of operation, operation type, and right-hand of operation). Then, a dictionary is used to translate from reduction to attribute operation. Note that due to the design of the language (mentioned in Section 3), there are no extra semantic checks needed. Figure 4.29 shows the semantic operator file generated.

As can be observed, our approach closely resembles the generation of successor and goto tables.

Note that most of SIMPLER's operations are related to the Symbol Table. The next section will explain the integration between the semantics operator and TOS and its workings.



```
nd else begin
   end else if(state == UPDATE_ATTRS) begin

        // <--------- Generated content
        if (current_reduction == Instruction_SKIP) begin

            state <= GENERATE_INSTRUCTIONS;

        end else if (current_reduction == Instruction_ID_ASSIGN_Arithmetic) begin

            if(tds_wait_cnt == 0) begin
                tds_i_operation <= TDS_CREATE_VARIABLE;
                tds_i_attribute <= reduction_attributes[0];
                tds_wait_cnt = tds_wait_cnt + 1;
            end else begin
                tds_i_operation <= TDS_NONE;
                if(tds_o_done) begin
                    reduction_attributes[0] <= tds_o_address;
                    state <= GENERATE_INSTRUCTIONS;
                end
            end

        end else if (current_reduction == Instruction_ID_ASSIGN_Boolean) begin

            if(tds_wait_cnt == 0) begin
                tds_i_operation <= TDS_CREATE_VARIABLE;
                tds_i_attribute <= reduction_attributes[0];
                tds_wait_cnt = tds_wait_cnt + 1;
            end else begin
                tds_i_operation <= TDS_NONE;
                if(tds_o_done) begin
                    reduction_attributes[0] <= tds_o_address;
                    state <= GENERATE_INSTRUCTIONS;
                end
            end

        ...
        ...

        end else if (current_reduction == Instruction_WhileDo_Body) begin

            state <= GENERATE_INSTRUCTIONS;

        end
        // ---------> Generated content

        codegen_wait_cnt = 0;
```

Figure 4.29: LWVCG generated contents of semantic operator module

4.4 Machine code Generation

4.4.1. Algorithms and data structures used

In this phase, we introduce the use of the symbol table, which plays a key role in generating compiled machine code instructions by organizing memory (variable positions, types, etc.).

A symbol table [45] is a data structure used by language translators (e.g., compilers or interpreters) to associate identifiers, constants, procedures, and functions with information about their declaration or usage in the source code.

Common data structures for implementing symbol tables include trees, linear lists, and hash tables. The symbol table is accessed during various compilation phases, typically starting with lexical analysis. In our case, it is accessed during the semantic phase, as earlier access was unnecessary for the SIMPLER language.

A widely used data structure for symbol tables is the hash table, which provides efficient lookup times regardless of the number of elements. A hash table [47] maps keys to values using a hash function to compute an index (or hash code) into an array of buckets, where the corresponding value is stored. Figure 4.30 visualizes a hash table.

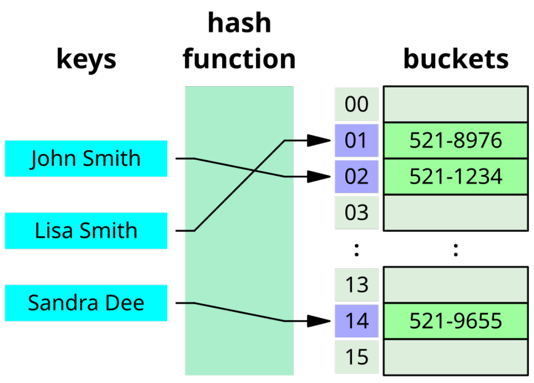
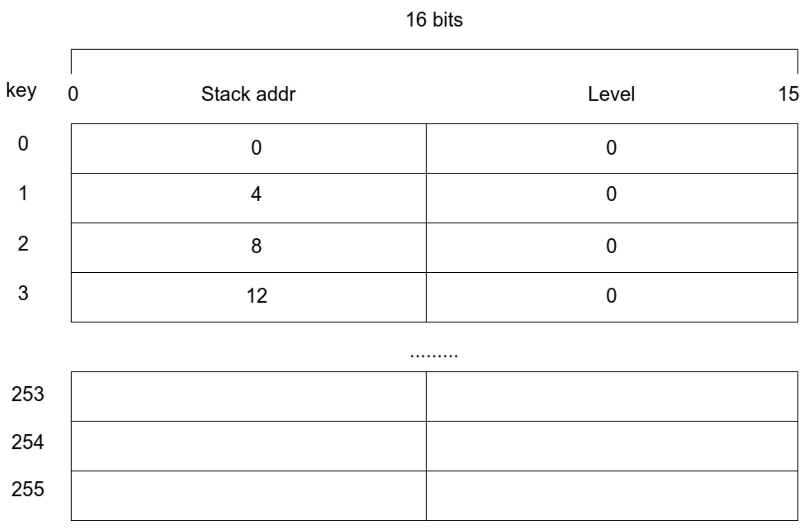


Figure 4.30: Hash table visualization

Therefore, as it can be easily implemented by using a contiguous chunk of memory we will implement a custom hash table for use in the Symbol table. However, to simplify the implementation, the buckets will have a size of 1. That is to say, writing two times an element with the same key will result in overwriting the value. This limitation is easily solvable for a system with a larger FGPGA and LC count.

Also, in our case, the data stored inside the symbol table will take the structure detailed in Figure 4.31.


16 bits

key	0 Stack addr	Level 15
0	0	0
1	4	0
2	8	0
3	12	0
	....	
253		
254		
255		

Figure 4.31: Symbol Table structure

In this structure, 16 bits of memory will be used per symbol to store the address this Symbol will have in the stack and the level of the block to which it pertains. The latter will not be used as, in order to simplify Symbol Table operations, the stack pointer is not reset after a code block.

4.4.2. Hardware implementation

In order to update the contents of the Symbol table during parsing and generate instructions, two steps are added to the parsing workflow during the application of a Reduction. One for performing read/write operations in the Symbol Table and another one to generate all the instructions that will result from the translation of that particular reduction. Figure 4.32 shows the modified workflow with the changes in color green.

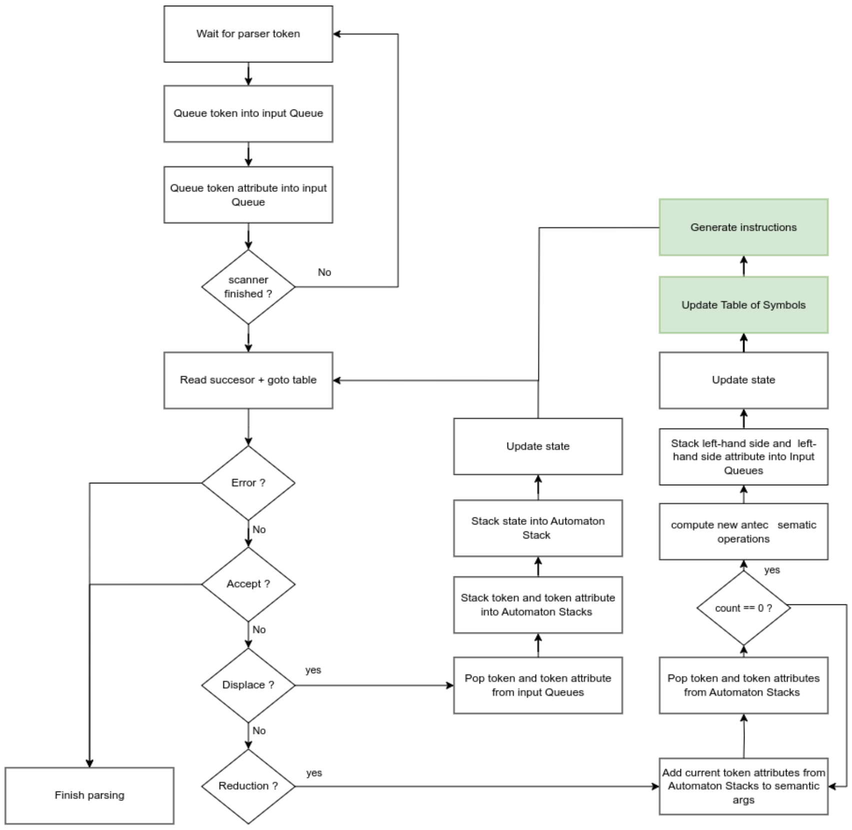

Wait for parser token
Queue token into input Queue
Queue token attribute into input Queue
scanner finished ?
No
Read succesor + goto table
Error ?
No
Accept ?
No
Displace ?
yes
No
Reduction ?
yes
Finish parsing
Pop token and token attribute from input Queues
Stack token and token attribute into Automaton Stacks
Stack state into Automaton Stack
Update state
Generate instructions
Update Table of Symbols
Update state
Stack left-hand side and left-hand side attribute into Input Queues
compute new antec sematic operations
yes
count == 0 ?
Pop token and token attributes from Automaton Stacks
Add current token attributes from Automaton Stacks to semantic args

Figure 4.32: Parsing workflow modifications for generating machine code

The algorithm in Listing 4.7 shows a pseudo-code implementation of the update Symbol Table step present in the mentioned workflow.

For SIMPLER, the key in the symbol table is the attribute of IDS, which, as explained in the design chapter, is an XOR hash of all the characters in the string.

The algorithm in Listing 4.8 provides a pseudo-code implementation of the instruction generation step in the workflow.

The algorithm starts by generating instructions to load the content of the attributes of each token in the reduction into registers t0, t1, t2, and t3. After that, it accesses a set of X consecutive positions in an instruction template memory with a Y offset. Both X and Y are constants retrieved from an array that contains the offset and size of the translations of each reduction ordered by their ID.

Note that this strategy to pass arguments and generate the translations is really inefficient in terms of register usage. However, it is enough for this proof of concept. Figure

```
// Inputs being (key, operation)
// Outputs being (resultStackAddress, resultKey)

if operation is none then
    nothing;

else if operation is createVariable() then
    hashTable[key] := currentStackAddress & (currentLevel) << 8;
    resultStackAddress := currentStackAddress;
    currentStackAddress := currentStackAddress + 4;
    result_level := currentLevel;

else if operation is createTemporalVariable() then
    key := temporalVariablesCount;
    resultKey := key;
    hashTable[key] := currentStackAddress & (currentLevel) << 8;
    resultStackAddress := currentStackAddress;
    currentStackAddress := currentStackAddress + 4;
    temporalVariablesCount := temporalVariablesCount + 1;
    result_level := currentLevel;

else if operation is getVariable() then
    resultStackAddress :=  hashTable[key];
```

Listing 4.7: Pseudo-code to update the Symbol table

```
// Inputs being (reductionId, arg0, arg1, arg2, arg3)

translationSize := translationSizes[reductionId];
memoryOffset := translationOffsets[reductionId];
index := 0;

sendInstruction("2008" + hex(arg0)); // Load to register t0 the literal in arg0
sendInstruction("2009" + hex(arg1)); // Load to register t1 the literal in arg1
sendInstruction("200a" + hex(arg2)); // Load to register t2 the literal in arg2
sendInstruction("200b" + hex(arg3)); // Load to register t3 the literal in arg3

while index < translationSize do
    sendInstruction(instructionTemplatesMemory[memoryOffset + index]);
    index : index + 1;
```

Listing 4.8: Pseudo-code to generate instructions

4.33 shows how the token attributes (args) are packed into a 32-bit value by the semantic operator.

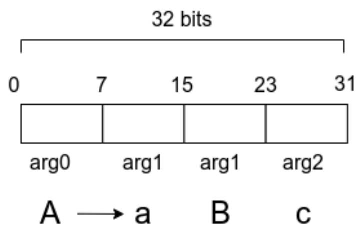


Figure 4.33: Semantic operator pack of reduction's token attributes

The symbol table and instruction generation logic are divided into two different modules. While the symbol table will be only accessed by the semantic operator module, the instructor generator module will be operating in parallel based on the current reduction ID, semantic operator attributes, and reduction detected flag. Figure 4.34 shows the modifications needed in the parsing hardware block diagram to include both modules.

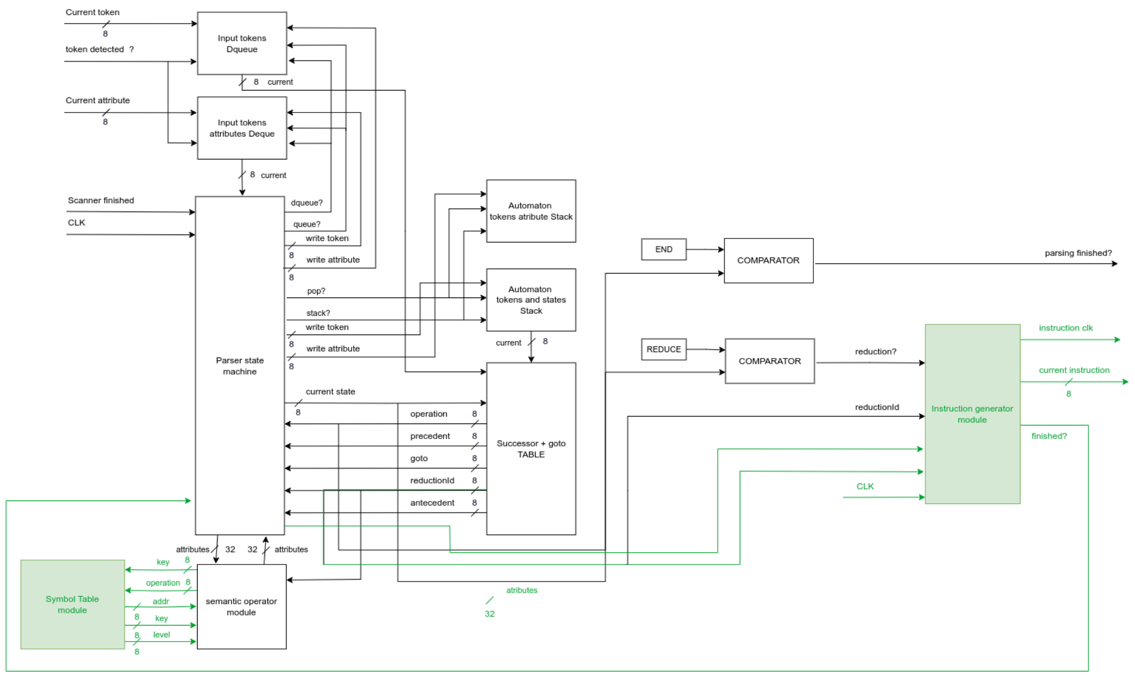


Figure 4.34: Machine code generation modifications in parsing hardware block diagram

Figure 4.35 shows the symbol table hardware block diagram.

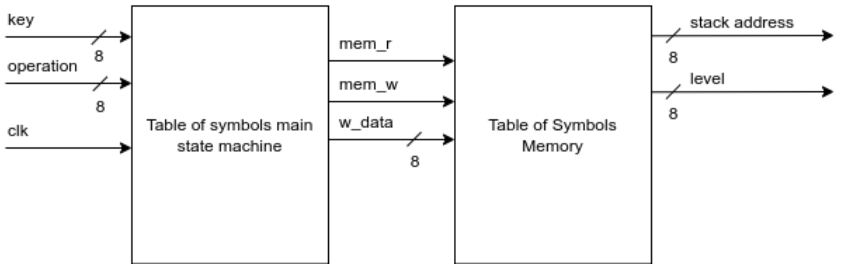


Figure 4.35: Symbol table hardware block diagram

The logic presented in Algorithm 4.7 is implemented as in the Symbol Table main state machine. As mentioned before, the table of Symbols memory is an implementation of a hash table that uses a block of embedded RAM of 16bit x 256 (4kit) as storage.

Figure 4.36 shows the instruction generation module hardware block diagram.

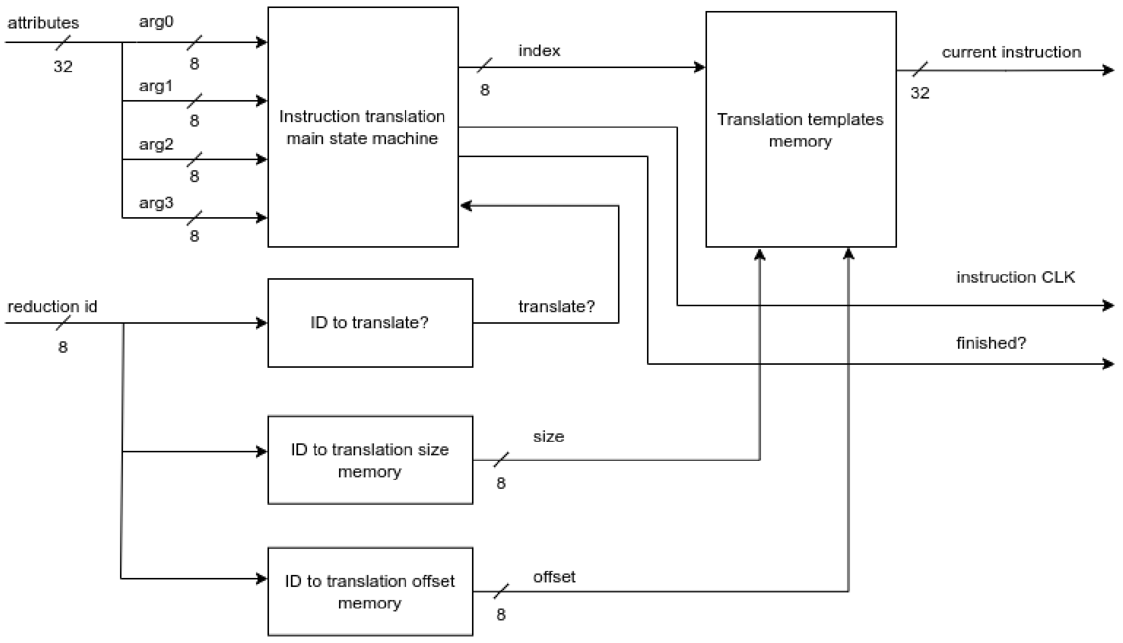


Figure 4.36: Instruction generation module hardware block diagram

The logic presented in Algorithm 4.8 is implemented in the instruction translation main state machine. The size and offset arrays the algorithm uses will be implemented as distributed memory blocks. The memory for the instruction templates will be implemented as an embedded memory block of 32 bits x 255 (8 kb). This memory contains the translations of all the reductions into machine code instructions, and both arrays mentioned provide the offset (memory position) and size (quantity of instructions) given a reduction ID.

Only the instruction generation module is automatically generated by the LWVCG Python framework from the code translation specification file.

4.4.3. LWVCG Verilog code generation

Figure 4.37 shows the LWVCG workflow for generating the instructions translation module. In this case, the code generation is only used to statically initiate the synthesis of the contents of the memories mentioned before.

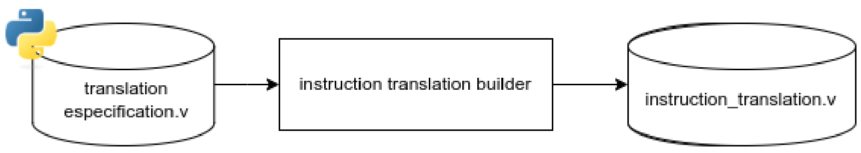

translation especification.v
instruction translation builder
instruction_translation.v

Figure 4.37: Code generation LWVCG workflow

4.5 Code execution and Source code uploader

For the code execution, as mentioned in Section 3 an open source implementation of a 32 bit Mips processor is used with the use of some optimizations. The main modification is changing the instruction ROM memory to an embedded block of RAM that allows writing.

The source code uploader is a simple Python script that uses the Python Serial library to upload via the source code I2C bus the SIMPLER code to the parser FPGA. This script follows the communication protocol described in Section 4.1.

5 Testing and Validation

During these tests, the SIMPLER programs in Listings 5.1 and 5.2 will be used as inputs.

```
index        := 1;
f0           := 1;
f1           := 1;
temp         := 0;
Print f0;
Print f1;

while index <= 11 do (
    index := index + 1;
    temp := f0 + f1;
    f0 := f1;
    f1 := temp;
    Print f1
)
```

Listing 5.1: Fibonacci calculator

```
index        := 1;
while index <= 255 do (
    index := index + 1;
    Print index
)
```

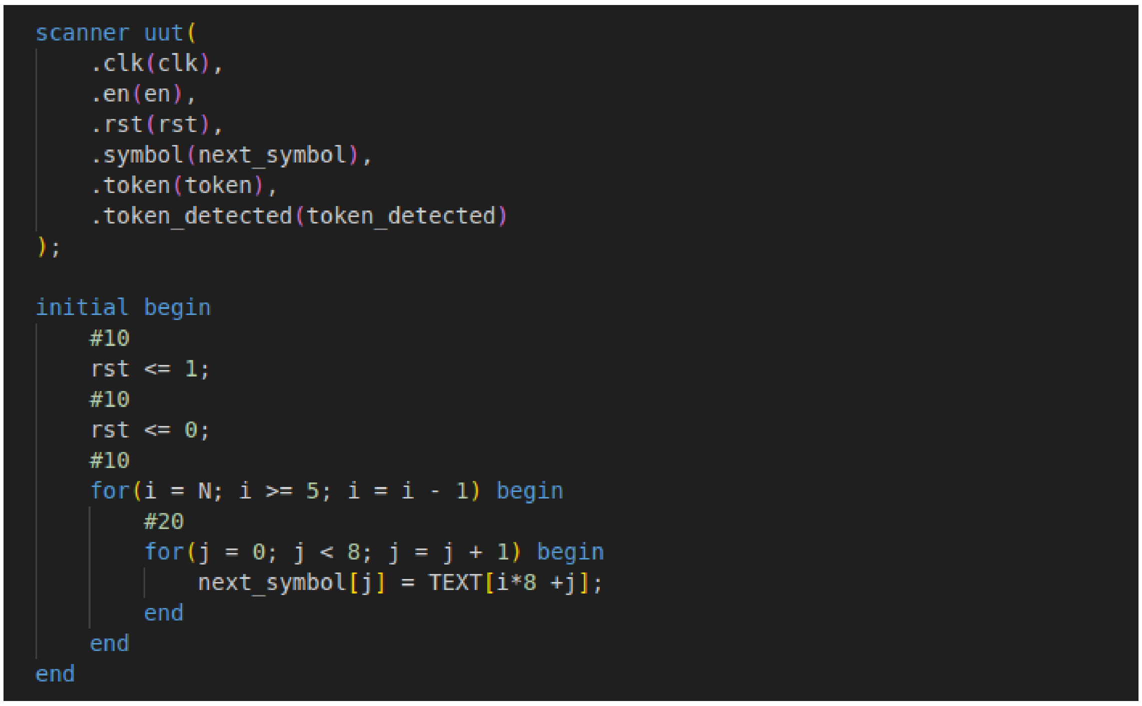
Listing 5.2: For loop

5.1 Simulations in GTKWave

This section reviews the simulations and results for the main test benches developed during this thesis: the scanner test bench (lexical analysis), the parser test bench (syntactical analysis, semantic analysis, code generation), and the code execution of the compiled code.

All simulation screenshots are from GTKWave [30], a fully-featured wave viewer for Unix, Win32, and Mac OSX that supports LXT, LXT2, VZT, FST, GHW, and Verilog VCD/EVCD files. GTKWave is integrated with the Apio toolset [28], an open-source ecosystem for FPGA boards. The hardware consumption calculations provided are from Apio's synthesis tool.

5.1.1. Scanner results



```
scanner uut(
    .clk(clk),
    .en(en),
    .rst(rst),
    .symbol(next_symbol),
    .token(token),
    .token_detected(token_detected)
);

initial begin
    #10
    rst <= 1;
    #10
    rst <= 0;
    #10
    for(i = N; i >= 5; i = i - 1) begin
        #20
        for(j = 0; j < 8; j = j + 1) begin
            next_symbol[j] = TEXT[i*8 +j];
        end
    end
end
```

Figure 5.1: Scanner Verilog test bench

The procedure for the test is: Re-execute the LWVCG framework to ensure the lexical analyzer code is up-to-date, save a screenshot of the IDs assigned to each token by the LWVCG framework, run the Verilog test bench, and compare the detected tokens and attributes using the LWVCG screenshot to ensure they match the expected results.

Figure 5.2 shows the results of the test execution for the program in Listing 5.2. As well as the hardware utilization statistics for the scanner module.

This test bench only tests the scanner module, not the I2C communication. The scanner processes one ASCII character per clock cycle, highlighting the I2C bus limitations. In this simulation, the scanner runs at a 50 MHz clock frequency (20 ns period).

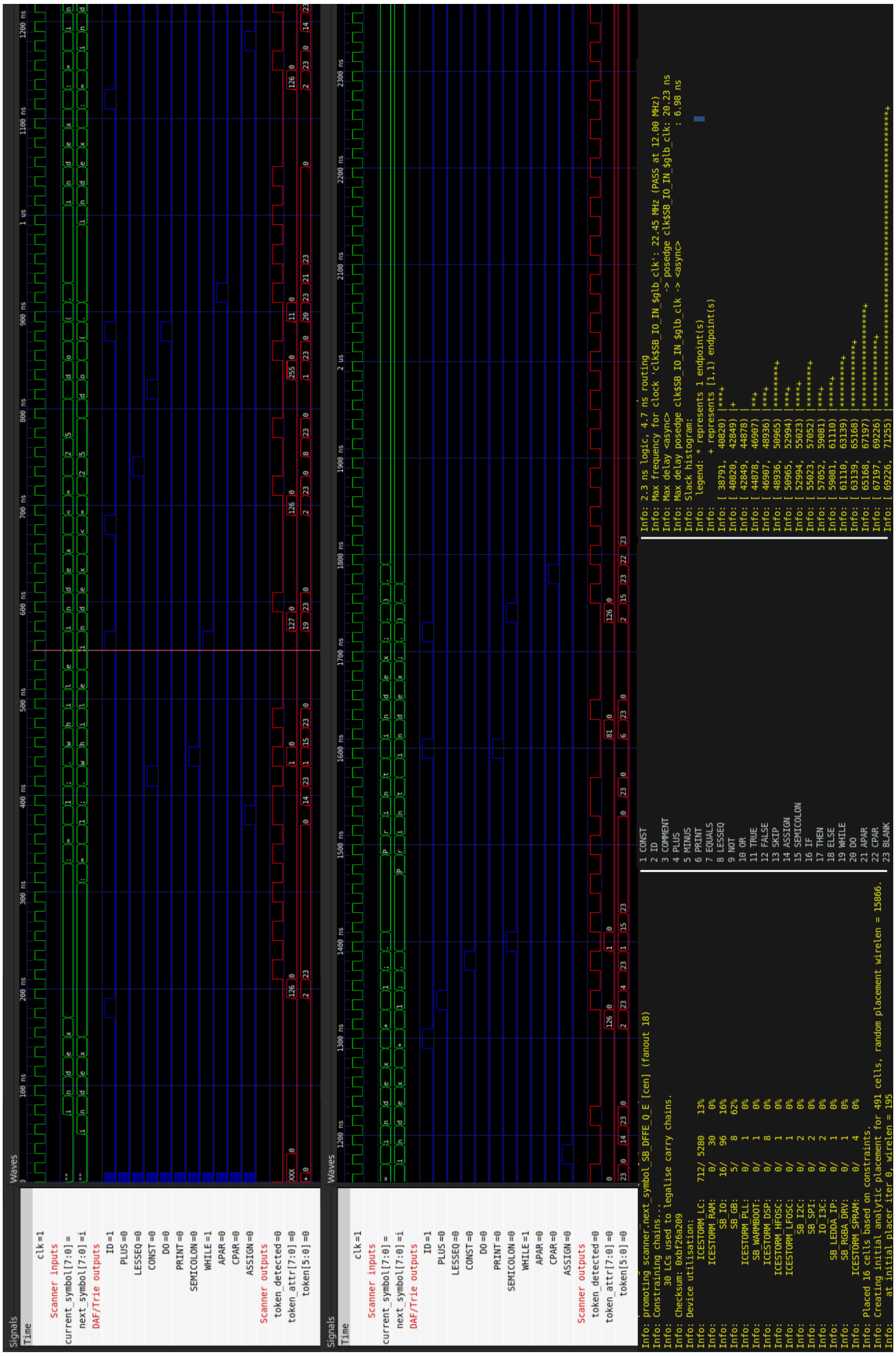
Green lines represent the inputs to the scanner module (current source code char, buffered source code char, and system clock). Blue lines represent the DFA and Trie results, indicating when a final state is high for a given token. Red lines show the scanner module's output signals.

Looking at the DFA and Trie outputs, the lexical analysis results are as follows (in chronological order): ID, ASSIGN, CONST, SEMICOLON, WHILE, ID, LESSEQ, CONST, APAR, ID, ASSIGN, ID, PLUS, CONST, SEMICOLON, PRINT, ID, CPAR. These results are correct.

The detected token values (ignoring codes 0 and 23, which represent null and BLANK tokens) are 2, 14, 1, 15, 19, 2, 8, 1, 20, 21, 2, 14, 2, 4, 1, 15, 6, 2, 22. These results are also correct using the translation table from the LWVCG framework.

For the token attribute signals, the observed attributes for relevant tokens (IDs and CONSTs) are 126, 1, and 255 for the "index" (ID), 1 (CONST), and 255 (CONST), respectively. These results are correct, as 126 corresponds to the XOR hash of the String "index."

In the simulation, the entire lexical analysis was completed between timestamps 50 ns and 1790 ns, resulting in an estimated duration of 1740 ns. Given that the source code contained 87 characters, this corresponds to approximately 20 ns per character (50 MHz).



```
Info: promoting scanner.next_symbol_SB_DFFE_Q_E [cen] (fanout 18)
Info: Constraining chains...
Info:       30 LCs used to legalise carry chains.
Info: Checksum: 0xbf26a209
Info: Device utilisation:
Info:            ICESTORM_LC:   712/ 5280    13%
Info:           ICESTORM_RAM:     0/   30     0%
Info:                  SB_IO:    16/   96    16%
Info:                  SB_GB:     5/    8    62%
Info:           ICESTORM_PLL:     0/    1     0%
Info:            SB_WARMBOOT:     0/    1     0%
Info:           ICESTORM_DSP:     0/    8     0%
Info:         ICESTORM_HFOSC:     0/    1     0%
Info:         ICESTORM_LFOSC:     0/    1     0%
Info:                 SB_I2C:     0/    2     0%
Info:                 SB_SPI:     0/    2     0%
Info:                 IO_I3C:     0/    2     0%
Info:            SB_LEDDA_IP:     0/    1     0%
Info:            SB_RGBA_DRV:     0/    1     0%
Info:         ICESTORM_SPRAM:     0/    4     0%
Info: Placed 16 cells based on constraints.
Info: Creating initial analytic placement for 491 cells, random placement wirelen = 15866.
Info:     at initial placer iter 0, wirelen = 195
```

```
1 CONST
2 ID
3 COMMENT
4 PLUS
5 MINUS
6 PRINT
7 EQUALS
8 LESSEQ
9 NOT
10 OR
11 TRUE
12 FALSE
13 SKIP
14 ASSIGN
15 SEMICOLON
16 IF
17 THEN
18 ELSE
19 WHILE
20 DO
21 APAR
22 CPAR
23 BLANK
```

```
Info: 2.3 ns logic, 4.7 ns routing
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 22.45 MHz (PASS at 12.00 MHz)
Info: Max delay <async>                       -> posedge clk$SB_IO_IN_$glb_clk: 20.23 ns
Info: Max delay posedge clk$SB_IO_IN_$glb_clk -> <async>                      : 6.98 ns
Info: Slack histogram:
Info:  legend: * represents 1 endpoint(s)
Info:          + represents [1,1) endpoint(s)
Info: [ 38791,  40820) |***+
Info: [ 40820,  42849) |+
Info: [ 42849,  44878) |
Info: [ 44878,  46907) |**+
Info: [ 46907,  48936) |***+
Info: [ 48936,  50965) |********+
Info: [ 50965,  52994) |***+
Info: [ 52994,  55023) |****+
Info: [ 55023,  57052) |********+
Info: [ 57052,  59081) |***+
Info: [ 59081,  61110) |*****+
Info: [ 61110,  63139) |*********+
Info: [ 63139,  65168) |************+
Info: [ 65168,  67197) |*******************+
Info: [ 67197,  69226) |*************+
Info: [ 69226,  71255) |*********************************************************+
```

Figure 5.2: GTKWave and Apio results for Algorithm 5.2

However, the simulation does not account for the minimum delay required for the design. According to the timing report, the maximum clock frequency for the design is

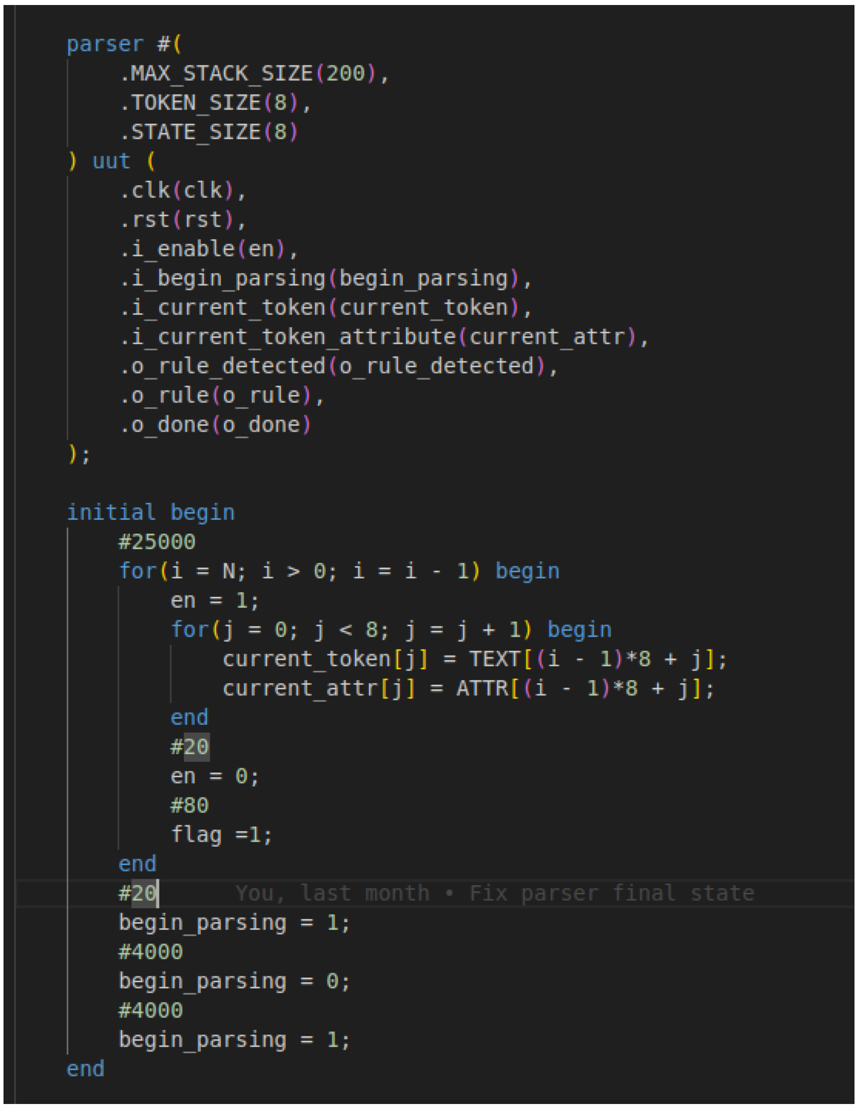
22.45 MHz, which is still well above the 12 MHz at which the scanner will operate on the ICE40UP5K.

As for hardware consumption, as shown in Figure 5.2 the scanner uses only 13% of the FPGA resources (712 LUTs) and 0 embedded RAM blocks, as the entire scanner is implemented in distributed memory.

Thus, these tests confirm that the scanner module operates successfully, with a maximum clock frequency of 22.45 MHz, a throughput of one character per clock cycle, and minimal FPGA resource usage.

5.1.2. Parser results

The parser test bench validates the correct operation of the syntactical analysis, semantic analysis, and code generation. Therefore, the test bench consists of passing to the parser the code of the token and attribute signals from the previous test bench into the parser. Then, the flag for beginning the parsing process is raised, and the machine code instructions start to be generated. Figure 5.3 shows the Verilog code for the parser test bench.



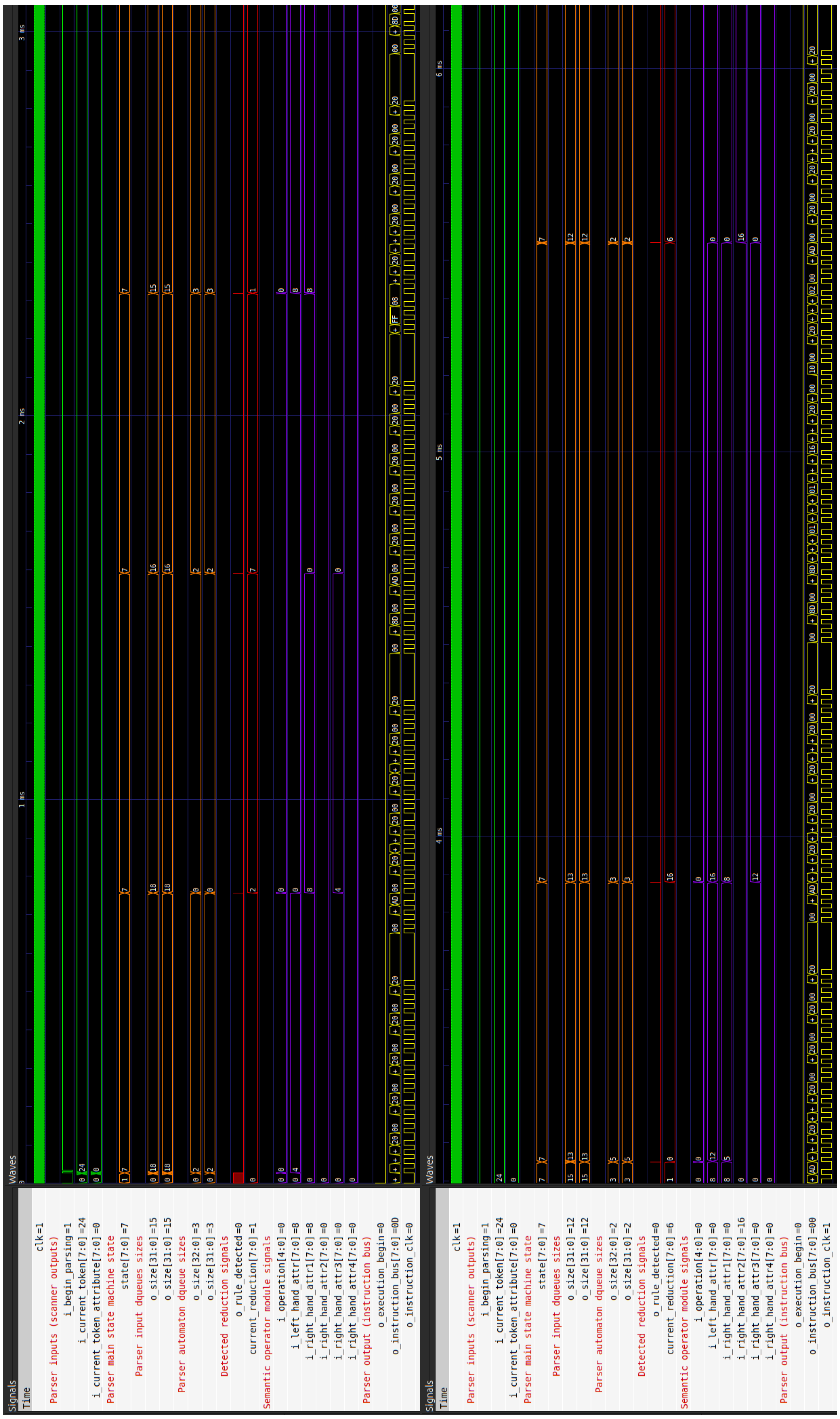
```verilog
parser #(
    .MAX_STACK_SIZE(200),
    .TOKEN_SIZE(8),
    .STATE_SIZE(8)
) uut (
    .clk(clk),
    .rst(rst),
    .i_enable(en),
    .i_begin_parsing(begin_parsing),
    .i_current_token(current_token),
    .i_current_token_attribute(current_attr),
    .o_rule_detected(o_rule_detected),
    .o_rule(o_rule),
    .o_done(o_done)
);

initial begin
    #25000
    for(i = N; i > 0; i = i - 1) begin
        en = 1;
        for(j = 0; j < 8; j = j + 1) begin
            current_token[j] = TEXT[(i - 1)*8 + j];
            current_attr[j] = ATTR[(i - 1)*8 + j];
        end
        #20
        en = 0;
        #80
        flag =1;
    end
    #20       You, last month • Fix parser final state
    begin_parsing = 1;
    #4000
    begin_parsing = 0;
    #4000
    begin_parsing = 1;
end
```

Figure 5.3: Parser Verilog test bench

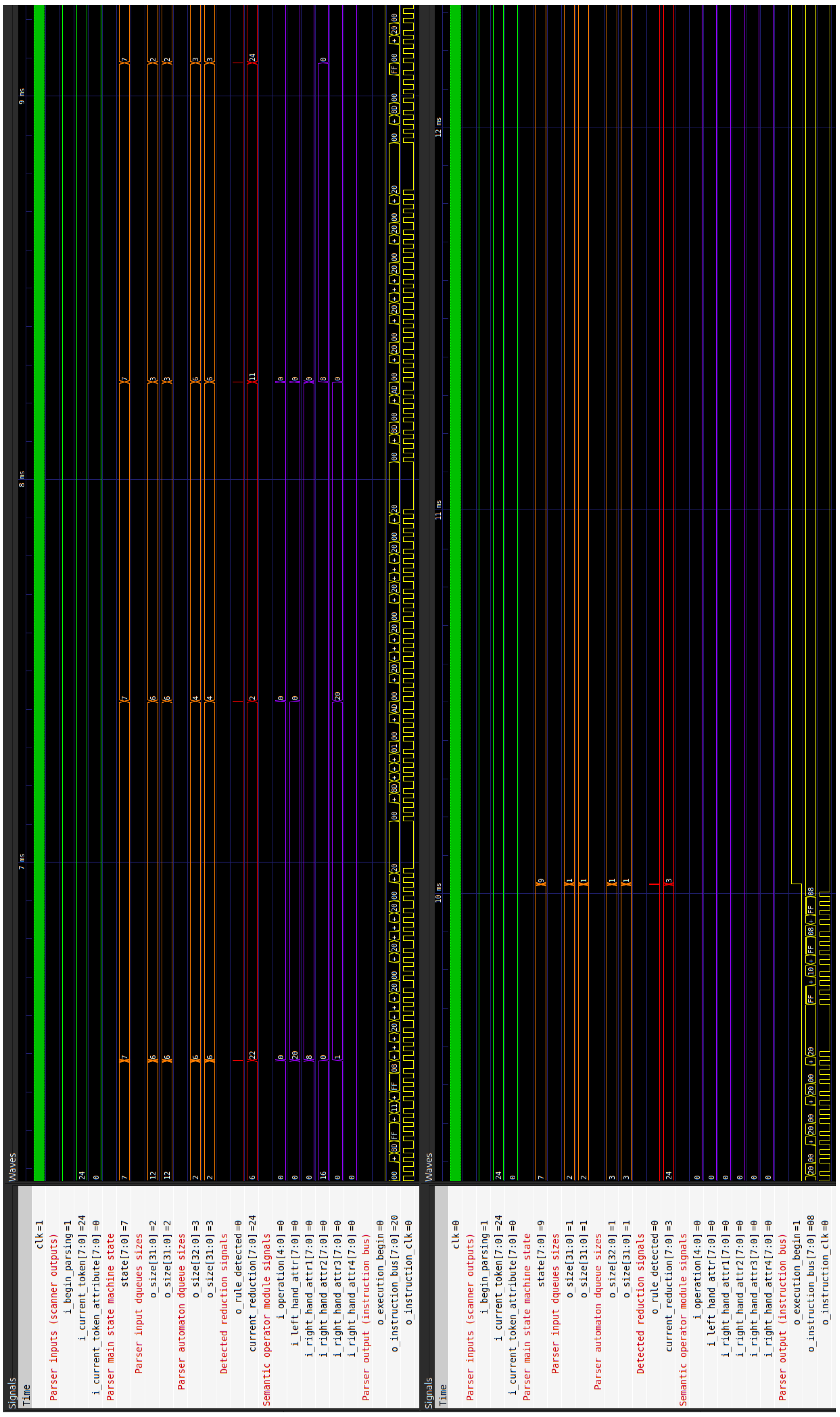
The procedure for the test is: Re-execute the LWVCG framework to ensure the syntax analyzer code is up-to-date, save a screenshot of the IDs assigned to each reduction by the LWVCG framework, run the Verilog test bench, and finally, using the LWVCG screenshot, compare the detected reductions, semantic operations, and generated instructions to check for correctness.

Figures 5.4 and 5.5 show the GTKWave results of running the parser test bench.


Signals
Time
clk=1
Parser inputs (scanner outputs)
i_begin_parsing=1
i_current_token[7:0]=24
i_current_token_attribute[7:0]=0
Parser main state machine state
state[7:0]=7
Parser input dqueues sizes
o_size[31:0]=15
o_size[31:0]=15
Parser automaton dqueue sizes
o_size[32:0]=3
o_size[31:0]=3
Detected reduction signals
o_rule_detected=0
current_reduction[7:0]=1
Semantic operator module signals
i_operation[4:0]=0
i_left_hand_attr[7:0]=8
i_right_hand_attr1[7:0]=8
i_right_hand_attr2[7:0]=0
i_right_hand_attr3[7:0]=0
i_right_hand_attr4[7:0]=0
Parser output (instruction bus)
o_execution_begin=0
o_instruction_bus[7:0]=0D
o_instruction_clk=0

Signals
Time
clk=1
Parser inputs (scanner outputs)
i_begin_parsing=1
i_current_token[7:0]=24
i_current_token_attribute[7:0]=0
Parser main state machine state
state[7:0]=7
Parser input dqueues sizes
o_size[31:0]=12
o_size[31:0]=12
Parser automaton dqueue sizes
o_size[32:0]=2
o_size[31:0]=2
Detected reduction signals
o_rule_detected=0
current_reduction[7:0]=6
Semantic operator module signals
i_operation[4:0]=0
i_left_hand_attr[7:0]=0
i_right_hand_attr1[7:0]=0
i_right_hand_attr2[7:0]=16
i_right_hand_attr3[7:0]=0
i_right_hand_attr4[7:0]=0
Parser output (instruction bus)
o_execution_begin=0
o_instruction_bus[7:0]=00
o_instruction_clk=1

Figure 5.4: First half of the parser test bench GTKWave results


Signals
Time
clk=1
Parser inputs (scanner outputs)
i_begin_parsing=1
i_current_token[7:0]=24
i_current_token_attribute[7:0]=0
Parser main state machine state
state[7:0]=7
Parser input dqueues sizes
o_size[31:0]=2
o_size[31:0]=2
Parser automaton dqueue sizes
o_size[32:0]=3
o_size[31:0]=3
Detected reduction signals
o_rule_detected=0
current_reduction[7:0]=24
Semantic operator module signals
i_operation[4:0]=0
i_left_hand_attr[7:0]=0
i_right_hand_attr1[7:0]=0
i_right_hand_attr2[7:0]=0
i_right_hand_attr3[7:0]=0
i_right_hand_attr4[7:0]=0
Parser output (instruction bus)
o_execution_begin=0
o_instruction_bus[7:0]=20
o_instruction_clk=0

Signals
Time
clk=0
Parser inputs (scanner outputs)
i_begin_parsing=1
i_current_token[7:0]=24
i_current_token_attribute[7:0]=0
Parser main state machine state
state[7:0]=9
Parser input dqueues sizes
o_size[31:0]=1
o_size[31:0]=1
Parser automaton dqueue sizes
o_size[32:0]=1
o_size[31:0]=1
Detected reduction signals
o_rule_detected=0
current_reduction[7:0]=3
Semantic operator module signals
i_operation[4:0]=0
i_left_hand_attr[7:0]=0
i_right_hand_attr1[7:0]=0
i_right_hand_attr2[7:0]=0
i_right_hand_attr3[7:0]=0
i_right_hand_attr4[7:0]=0
Parser output (instruction bus)
o_execution_begin=1
o_instruction_bus[7:0]=08
o_instruction_clk=0

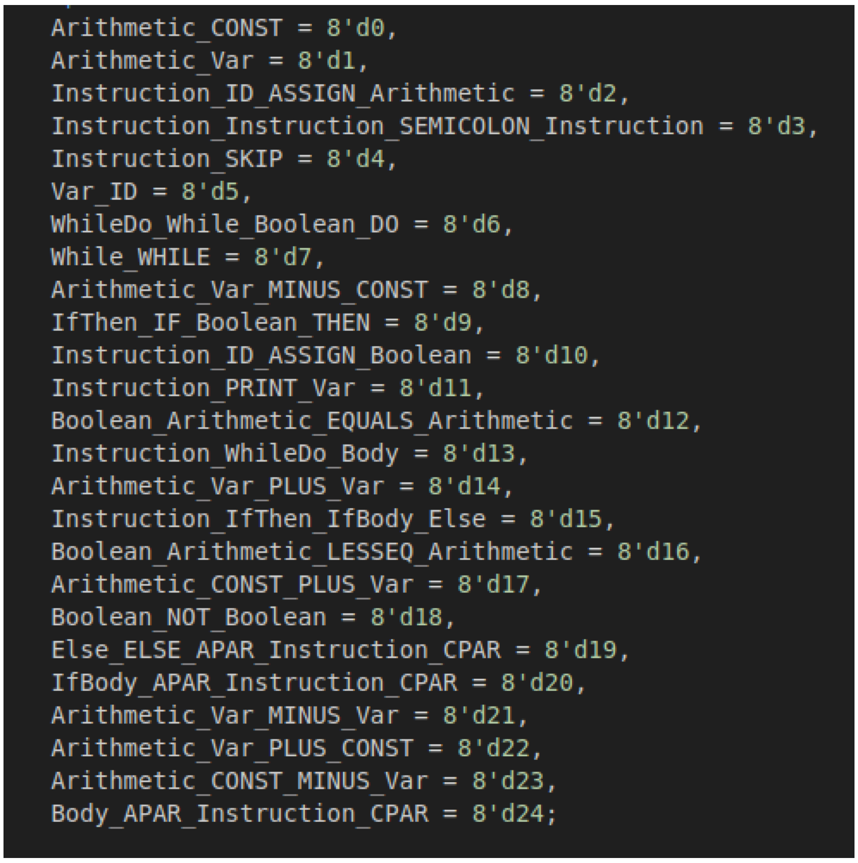
Figure 5.5: Second half of the parser test bench GTKWave results

Due to their large size, the simulation results have been divided into four screenshots shown in the mentioned figures. Note that each screenshot has a time axis at the top and is chronologically ordered from top to bottom.

The green signals represent the input signals of the parser module (input tokens and token attributes), orange signals show intermediate results (sizes of the automaton and input queues), red signals represent outputs from the syntax analysis, violet signals show outputs from the semantic analysis (operations in the Symbol table), and yellow signals represent the machine code generator module outputs (8-bit instruction bus connecting the FPGAs and the instruction clock).

It is clear that the output instruction bus speed limits the parser's execution speed. Observing the signal for the parser's main state machine, we can see that it is mostly in the IDLE state until the instruction bus completes a reduction, at which point the parsing resumes. This is due to the bus's low frequency (10 kHz) compared to the parser's clock frequency (12 MHz).

Upon checking the correctness of the detected reductions, we observe that the generated syntax abstract tree (as seen from the "current_reduction" signal) is: 0, 2, 7, 5, 1, 0, 16, 6, 5, 22, 2, 5, 11, 3, 24, 13, and 3. Due to the simulation scale, some values appear small. Using the LWVCG output shown in Figure 5.6, we can translate this to the applied reductions, with the result shown in Listing 5.3. This confirms the correct operation of the syntax analyzer.



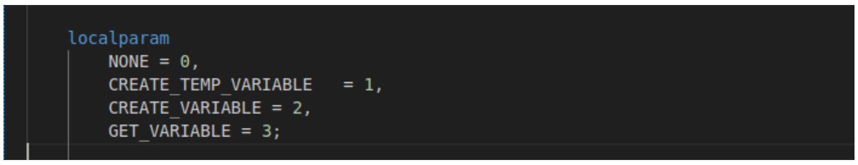
```
Arithmetic_CONST = 8'd0,
Arithmetic_Var = 8'd1,
Instruction_ID_ASSIGN_Arithmetic = 8'd2,
Instruction_Instruction_SEMICOLON_Instruction = 8'd3,
Instruction_SKIP = 8'd4,
Var_ID = 8'd5,
WhileDo_While_Boolean_DO = 8'd6,
While_WHILE = 8'd7,
Arithmetic_Var_MINUS_CONST = 8'd8,
IfThen_IF_Boolean_THEN = 8'd9,
Instruction_ID_ASSIGN_Boolean = 8'd10,
Instruction_PRINT_Var = 8'd11,
Boolean_Arithmetic_EQUALS_Arithmetic = 8'd12,
Instruction_WhileDo_Body = 8'd13,
Arithmetic_Var_PLUS_Var = 8'd14,
Instruction_IfThen_IfBody_Else = 8'd15,
Boolean_Arithmetic_LESSEQ_Arithmetic = 8'd16,
Arithmetic_CONST_PLUS_Var = 8'd17,
Boolean_NOT_Boolean = 8'd18,
Else_ELSE_APAR_Instruction_CPAR = 8'd19,
IfBody_APAR_Instruction_CPAR = 8'd20,
Arithmetic_Var_MINUS_Var = 8'd21,
Arithmetic_Var_PLUS_CONST = 8'd22,
Arithmetic_CONST_MINUS_Var = 8'd23,
Body_APAR_Instruction_CPAR = 8'd24;
```

Figure 5.6: LWVCG results for generating parser

As for the semantic operator module, observing the signal "i_operation," the following operations are applied: 1, 2, 3, 1, 1, 3, 1, 2, 3. Referring to the implementation of the semantic operator module in Figure 5.7 these operations translate to:

```
(0) Arithmetic → CONST
(2) Instruction → ID ASSIGN Arithmetic
(7) While → WHILE
(5) Var → ID
(1) Arithmetic → Var
(0) Arithmetic → CONST
(16) Boolean → Arithmetic LESSEQ Arithmetic
(6) WhileDo → While Boolean DO
(5) Var → ID
(22) Arithmatic → Var PLUS CONST
(2) Instruction → ID ASSIGN Arithmetic
(5) Var → ID
(11) Instruction → PRINT Var
(3) Instruction → Instruction SEMICOLON Instruction
(24) Body → APAR Instruction CPAR
(13) Instruction → WhileDo Body
(3) Instruction → Instruction SEMICOLON Instruction
```

Listing 5.3: Syntax analyzer simulation result translated



```
localparam
    NONE = 0,
    CREATE_TEMP_VARIABLE   = 1,
    CREATE_VARIABLE = 2,
    GET_VARIABLE = 3;
```

Figure 5.7: Semantic operator operation codes

CREATE_TEMP_VARIABLE, CREATE_VARIABLE, GET_VARIABLE, CREATE_TEMP_VARIABLE, CREATE_TEMP_VARIABLE, GET_VARIABLE, CREATE_TEMP_VARIABLE, CREATE_VARIABLE, and GET_VARIABLE.

When combined with the argument and current reduction signals, the resulting operations are shown. Listing 5.4 presents these translations, used to deduce the semantic operations being applied.

```
(1) Arithmetic = creaTempVariable(0)    // For const 0
(2) ID = createVariable(126)            // For ID "index"
(3) Var = getVariable(126)              // Get "index"
(1) Arithmetic = creaTempVariable(255) // For const 255
(1) Boolean = creaTempVariable(0)       // For Boolean result
(3) Var = getVariable(126)              // Get "index"
(1) Arithmetic = creaTempVariable(1)    // For const 1
(2) ID = createVariable(126)            // For ID "index" (overwrite)
(3) Var = getVariable(126)              // Get "index"
```

Listing 5.4: Syntax analyzer simultaion result translated

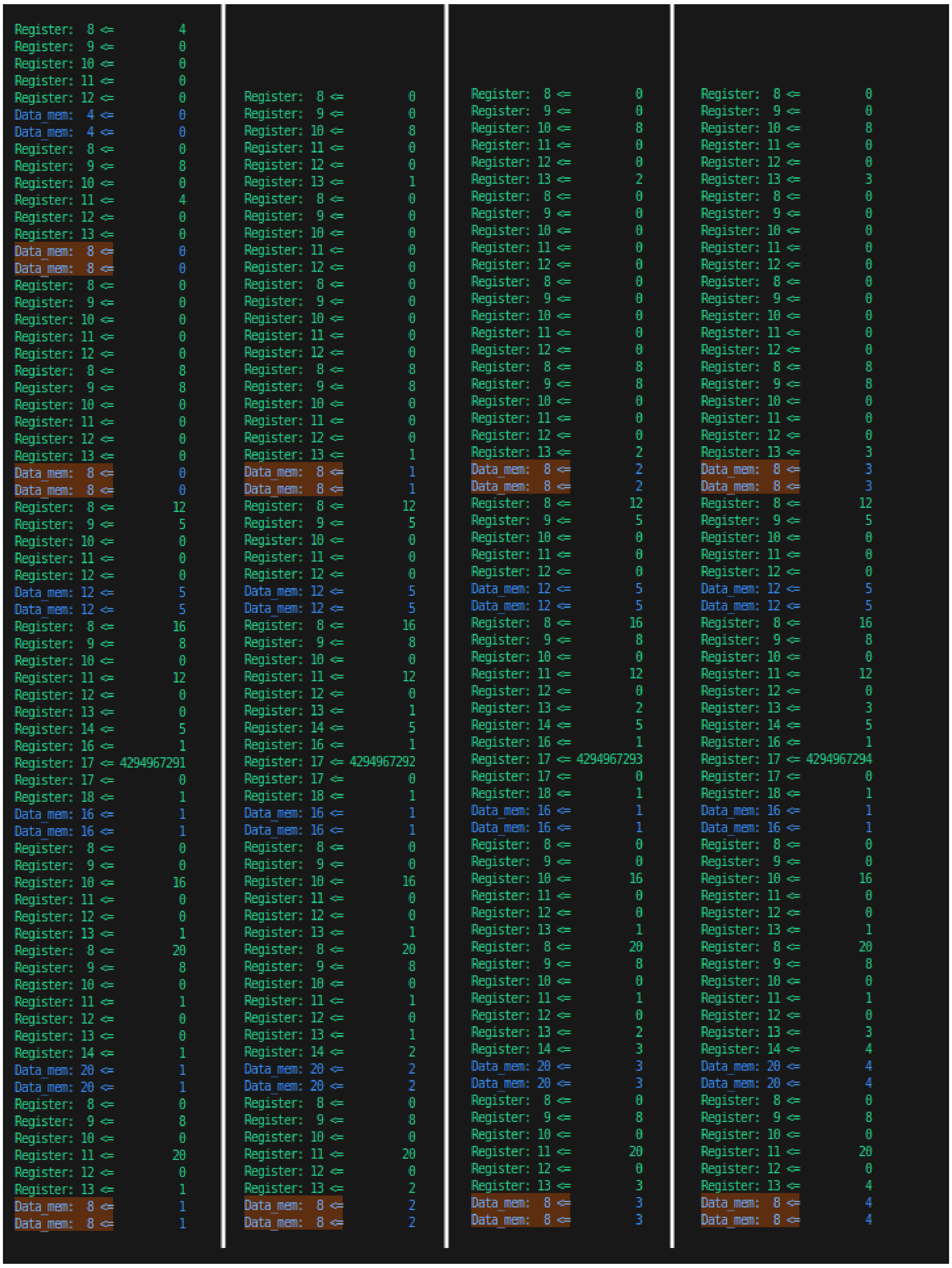
Which confirms the correct operation of the semantic analyzer. Regarding the generated instructions, by checking the contents of the instruction bus, we observe the machine code shown in Listing 5.5. While there are too many instructions generated for a simple for loop, it is important to note that, during design, many optimization algorithms were omitted to simplify the compiler design for fitting into a single FPGA with a limited budget. As a result, an average of 6 machine code instructions (based on empirical testing) are generated per reduction, leading to bloated machine code.

```
20080004
20090001
200a0000
200b0000
200c0000
ad090000
20080000
20090008
200a0000
200b0004
200c0000
8d6d0000
ad2d0000
20080000
20090000
200a0000
200b0000
200c0000
20080008
20090008
200a0000
200b0000
200c0000
8d2d0000
ad0d0000
2008000c
200900ff
200a0000
200b0000
200c0000
ad090000
20080010
20090008
200a0000
200b000c
200c0000
8d2d0000
8d6e0000
01ae802a
01ae8822
16200002
20110001
10000001
20110000
02119025
ad120000
20080000
20090000
200a0010
200b0000
200c0000
8d4d0000
11a0001C
20080014
20090008
200a0000
200b0001
200c0000
8d2d0000
01ab7020
ad0e0000
20080000
20090008
200a0000
200b0014
200c0000
8d6d0000
ad2d0000
20080000
20090000
200a0008
200b0000
200c0000
8d4d0000
ff000000
20080000
20090000
200a0000
200b0000
200c0000
1000FFBB
```

Listing 5.5: Machine code generated for for loop

This limitation can be addressed by implementing well-known register optimization algorithms, which are beyond the scope of this proof of concept.

To keep this section concise, we will not analyze the generated machine code in detail. Instead, we ran it in a MIPS 32 emulator, yielding the results shown in Figure 5.8. As seen, following the memory operations assigned to memory position 8 (which, based on the semantic operations breakdown, corresponds to the variable "index"), we can clearly observe the for loop executing, confirming the correct operation of the machine code generation module.



```
Register:  8 <=          4
Register:  9 <=          0
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Data_mem:  4 <=          0
Data_mem:  4 <=          0
Register:  8 <=          0
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=          4
Register: 12 <=          0
Register: 13 <=          0
Data_mem:  8 <=          0
Data_mem:  8 <=          0
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Register:  8 <=          8
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Register: 13 <=          0
Data_mem:  8 <=          0
Data_mem:  8 <=          0
Register:  8 <=         12
Register:  9 <=          5
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Data_mem: 12 <=          5
Data_mem: 12 <=          5
Register:  8 <=         16
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=         12
Register: 12 <=          0
Register: 13 <=          0
Register: 14 <=          5
Register: 16 <=          1
Register: 17 <= 4294967291
Register: 17 <=          0
Register: 18 <=          1
Data_mem: 16 <=          1
Data_mem: 16 <=          1
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=         16
Register: 11 <=          0
Register: 12 <=          0
Register: 13 <=          1
Register:  8 <=         20
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=          1
Register: 12 <=          0
Register: 13 <=          0
Register: 14 <=          1
Data_mem: 20 <=          1
Data_mem: 20 <=          1
Register:  8 <=          0
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=         20
Register: 12 <=          0
Register: 13 <=          1
Data_mem:  8 <=          1
Data_mem:  8 <=          1
```

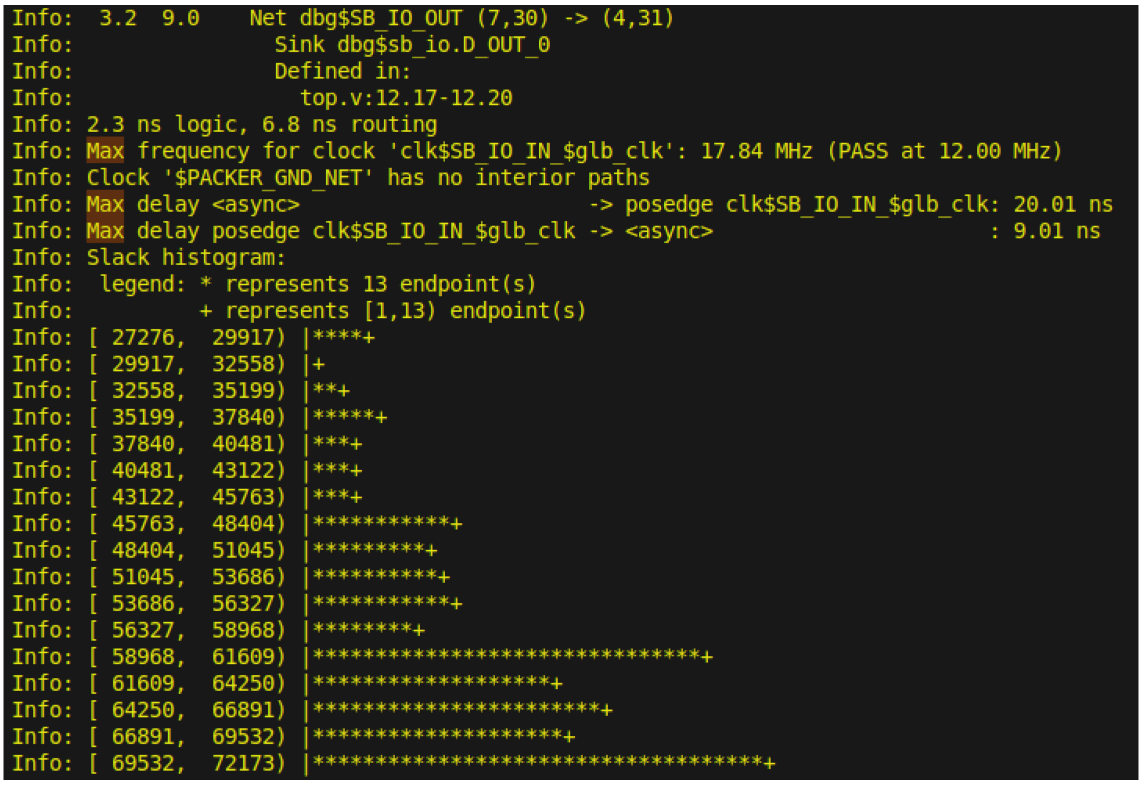
```
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=          8
Register: 11 <=          0
Register: 12 <=          0
Register: 13 <=          1
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Register:  8 <=          8
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Register: 13 <=          1
Data_mem:  8 <=          1
Data_mem:  8 <=          1
Register:  8 <=         12
Register:  9 <=          5
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Data_mem: 12 <=          5
Data_mem: 12 <=          5
Register:  8 <=         16
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=         12
Register: 12 <=          0
Register: 13 <=          1
Register: 14 <=          5
Register: 16 <=          1
Register: 17 <= 4294967292
Register: 17 <=          0
Register: 18 <=          1
Data_mem: 16 <=          1
Data_mem: 16 <=          1
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=         16
Register: 11 <=          0
Register: 12 <=          0
Register: 13 <=          1
Register:  8 <=         20
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=          1
Register: 12 <=          0
Register: 13 <=          1
Register: 14 <=          2
Data_mem: 20 <=          2
Data_mem: 20 <=          2
Register:  8 <=          0
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=         20
Register: 12 <=          0
Register: 13 <=          2
Data_mem:  8 <=          2
Data_mem:  8 <=          2
```

```
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=          8
Register: 11 <=          0
Register: 12 <=          0
Register: 13 <=          2
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Register:  8 <=          8
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Register: 13 <=          2
Data_mem:  8 <=          2
Data_mem:  8 <=          2
Register:  8 <=         12
Register:  9 <=          5
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Data_mem: 12 <=          5
Data_mem: 12 <=          5
Register:  8 <=         16
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=         12
Register: 12 <=          0
Register: 13 <=          2
Register: 14 <=          5
Register: 16 <=          1
Register: 17 <= 4294967293
Register: 17 <=          0
Register: 18 <=          1
Data_mem: 16 <=          1
Data_mem: 16 <=          1
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=         16
Register: 11 <=          0
Register: 12 <=          0
Register: 13 <=          1
Register:  8 <=         20
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=          1
Register: 12 <=          0
Register: 13 <=          2
Register: 14 <=          3
Data_mem: 20 <=          3
Data_mem: 20 <=          3
Register:  8 <=          0
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=         20
Register: 12 <=          0
Register: 13 <=          3
Data_mem:  8 <=          3
Data_mem:  8 <=          3
```

```
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=          8
Register: 11 <=          0
Register: 12 <=          0
Register: 13 <=          3
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Register:  8 <=          8
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Register: 13 <=          3
Data_mem:  8 <=          3
Data_mem:  8 <=          3
Register:  8 <=         12
Register:  9 <=          5
Register: 10 <=          0
Register: 11 <=          0
Register: 12 <=          0
Data_mem: 12 <=          5
Data_mem: 12 <=          5
Register:  8 <=         16
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=         12
Register: 12 <=          0
Register: 13 <=          3
Register: 14 <=          5
Register: 16 <=          1
Register: 17 <= 4294967294
Register: 17 <=          0
Register: 18 <=          1
Data_mem: 16 <=          1
Data_mem: 16 <=          1
Register:  8 <=          0
Register:  9 <=          0
Register: 10 <=         16
Register: 11 <=          0
Register: 12 <=          0
Register: 13 <=          1
Register:  8 <=         20
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=          1
Register: 12 <=          0
Register: 13 <=          3
Register: 14 <=          4
Data_mem: 20 <=          4
Data_mem: 20 <=          4
Register:  8 <=          0
Register:  9 <=          8
Register: 10 <=          0
Register: 11 <=         20
Register: 12 <=          0
Register: 13 <=          4
Data_mem:  8 <=          4
Data_mem:  8 <=          4
```

Figure 5.8: MIPS emulator interpretation of the generated code up to iteration 4

Figure 5.9 shows the Apio time analysis report of the parser. For the ICE40UP5K, the maximum design frequency is 17.84 MHz, which is well above the 12 MHz at which it will be running.



```
Info:  3.2  9.0    Net dbg$SB_IO_OUT (7,30) -> (4,31)
Info:                Sink dbg$sb_io.D_OUT_0
Info:                Defined in:
Info:                  top.v:12.17-12.20
Info: 2.3 ns logic, 6.8 ns routing
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 17.84 MHz (PASS at 12.00 MHz)
Info: Clock '$PACKER_GND_NET' has no interior paths
Info: Max delay <async>                      -> posedge clk$SB_IO_IN_$glb_clk: 20.01 ns
Info: Max delay posedge clk$SB_IO_IN_$glb_clk -> <async>                      : 9.01 ns
Info: Slack histogram:
Info:  legend: * represents 13 endpoint(s)
Info:          + represents [1,13) endpoint(s)
Info: [ 27276,  29917) |****+
Info: [ 29917,  32558) |+
Info: [ 32558,  35199) |**+
Info: [ 35199,  37840) |*****+
Info: [ 37840,  40481) |***+
Info: [ 40481,  43122) |***+
Info: [ 43122,  45763) |***+
Info: [ 45763,  48404) |***********+
Info: [ 48404,  51045) |*********+
Info: [ 51045,  53686) |**********+
Info: [ 53686,  56327) |***********+
Info: [ 56327,  58968) |********+
Info: [ 58968,  61609) |*******************************+
Info: [ 61609,  64250) |*******************+
Info: [ 64250,  66891) |***********************+
Info: [ 66891,  69532) |********************+
Info: [ 69532,  72173) |************************************+
```

Figure 5.9: Apio synthesis report for timing analysis on parser and scanner module.

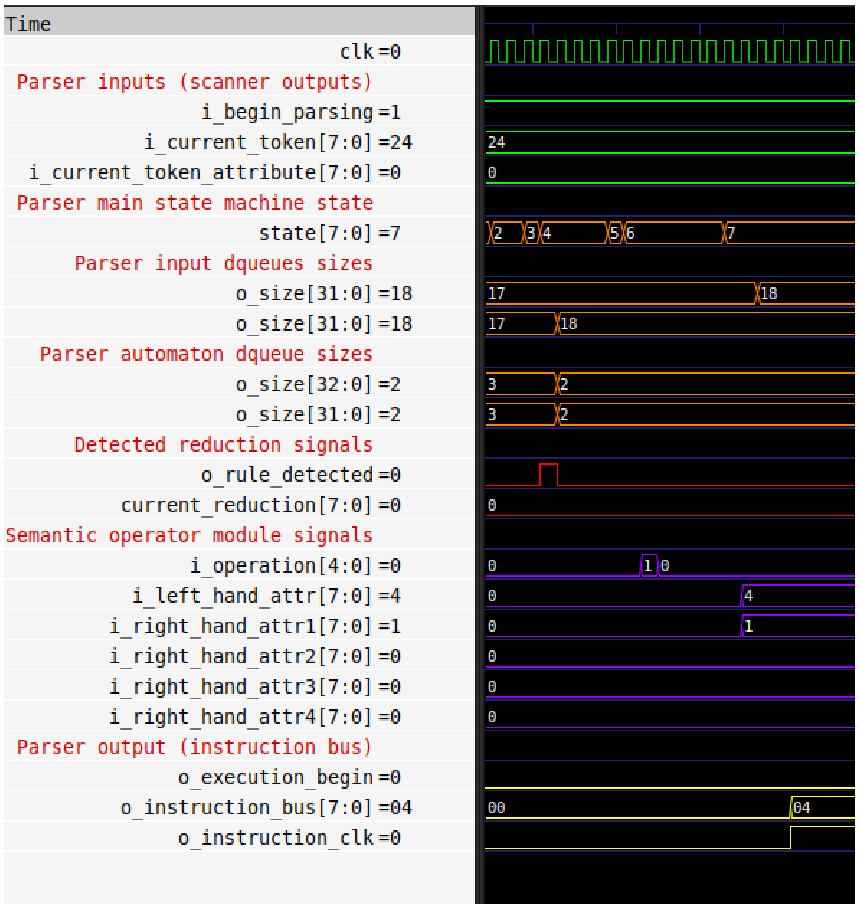
As noted earlier, the limitations of the testing setup in this proof of concept restrict the instruction bus's speed to 10 kHz. To assess the potential time efficiency of this solution, we will estimate the parsing time assuming a bus speed of 1 clock cycle per bit (achievable with larger FPGAs).

In this scenario, the parser state machine transitions through the states shown in Figure 5.10. States 2, 3, 4, 5, 6, and 7 represent:

PARSE, READ_TRANSITION_TABLE, OBTAIN_REDUCTION_ATTRS, UPDATE_TDS, UPDATE_ATTRS, and GENERATE_INSTRUCTIONS

respectively. State 2 starts at 450 ns, and state 7 starts at 730 ns, giving the time taken for a single parser step as $T_{parser} = 480 + T_{transmission}$ ns. In clock cycles, this is $T_{parser} = 20 + T_{transmission}$.

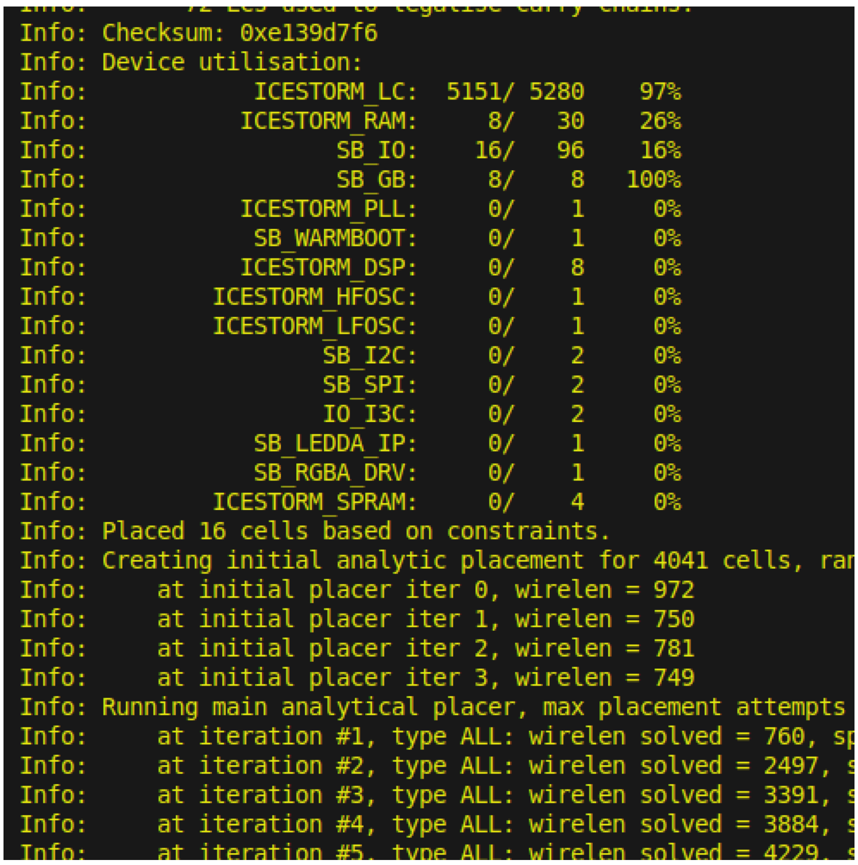
Assuming the average case for instruction generation (6 instructions per reduction) and a bus speed of 1 bit per clock cycle, $T_{transmission} = 32 * 6 * 1 = 192$ ns. Therefore, $T_{parser} = 192$ clock cycles. At 12 MHz, this results in a 62 kHz frequency (62 parser steps per second), which is significantly slower than the scanner, largely due to the communication with the microcontroller, which accounts for 90% of the clock cycles spent in parsing.



```
Time
                                clk=0
 Parser inputs (scanner outputs)
                     i_begin_parsing=1
                i_current_token[7:0]=24
      i_current_token_attribute[7:0]=0
 Parser main state machine state
                          state[7:0]=7
      Parser input dqueues sizes
                       o_size[31:0]=18
                       o_size[31:0]=18
   Parser automaton dqueue sizes
                       o_size[32:0]=2
                       o_size[31:0]=2
      Detected reduction signals
                     o_rule_detected=0
              current_reduction[7:0]=0
Semantic operator module signals
                    i_operation[4:0]=0
                i_left_hand_attr[7:0]=4
              i_right_hand_attr1[7:0]=1
              i_right_hand_attr2[7:0]=0
              i_right_hand_attr3[7:0]=0
              i_right_hand_attr4[7:0]=0
 Parser output (instruction bus)
                   o_execution_begin=0
             o_instruction_bus[7:0]=04
                   o_instruction_clk=0
```

Figure 5.10: Parser main state machine during a single step (worst case scenario)

Figure 5.11 shows the Apio hardware consumption report for synthesizing both the parser and scanner modules. The added consumption of the parser is 84% of the ICEUP5K's logic cells (4435 logic cells) and 8 out of the 30 embedded RAM blocks. This confirms the need, as stated in Section 3, to use a separate FPGA to host the microcontroller and handle code execution.

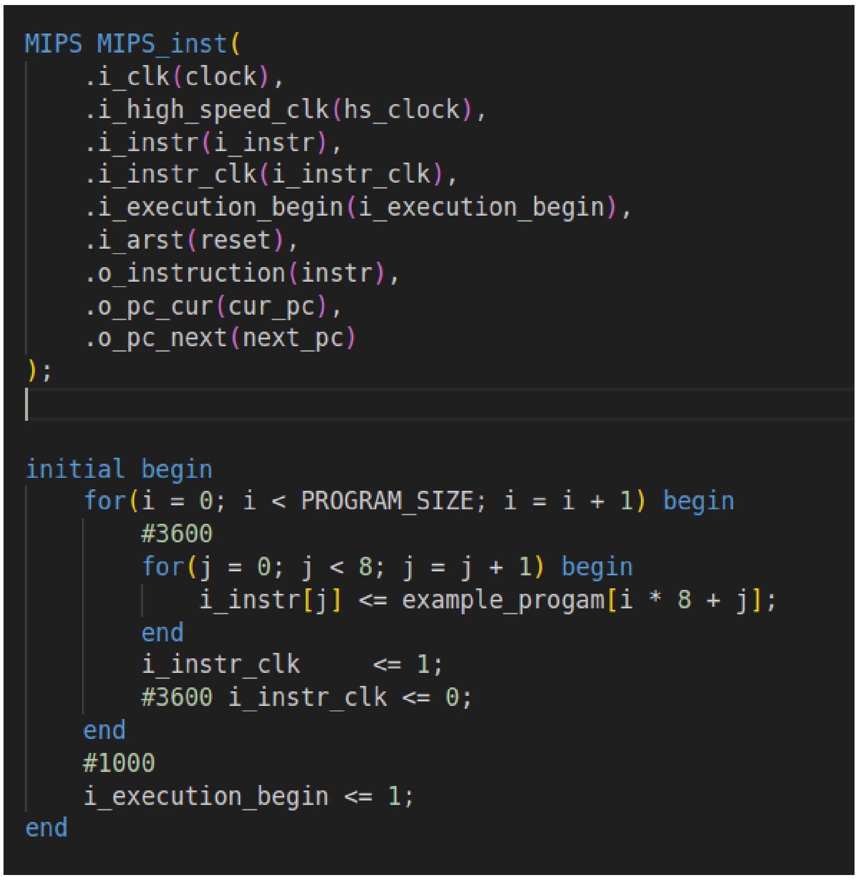


```
Info: Checksum: 0xe139d7f6
Info: Device utilisation:
Info:          ICESTORM_LC:  5151/ 5280    97%
Info:         ICESTORM_RAM:     8/   30    26%
Info:                SB_IO:    16/   96    16%
Info:                SB_GB:     8/    8   100%
Info:         ICESTORM_PLL:     0/    1     0%
Info:          SB_WARMBOOT:     0/    1     0%
Info:         ICESTORM_DSP:     0/    8     0%
Info:       ICESTORM_HFOSC:     0/    1     0%
Info:       ICESTORM_LFOSC:     0/    1     0%
Info:               SB_I2C:     0/    2     0%
Info:               SB_SPI:     0/    2     0%
Info:               IO_I3C:     0/    2     0%
Info:          SB_LEDDA_IP:     0/    1     0%
Info:          SB_RGBA_DRV:     0/    1     0%
Info:       ICESTORM_SPRAM:     0/    4     0%
Info: Placed 16 cells based on constraints.
Info: Creating initial analytic placement for 4041 cells, ran
Info:     at initial placer iter 0, wirelen = 972
Info:     at initial placer iter 1, wirelen = 750
Info:     at initial placer iter 2, wirelen = 781
Info:     at initial placer iter 3, wirelen = 749
Info: Running main analytical placer, max placement attempts
Info:     at iteration #1, type ALL: wirelen solved = 760, sp
Info:     at iteration #2, type ALL: wirelen solved = 2497, s
Info:     at iteration #3, type ALL: wirelen solved = 3391, s
Info:     at iteration #4, type ALL: wirelen solved = 3884, s
Info:     at iteration #5, type ALL: wirelen solved = 4229, s
```

Figure 5.11: Apio synthesis hardware consumption report for parser and scanner module

5.1.3. Code execution results

The test bench developed for the code execution phase tests the changes introduced to the open-source implementation of the MIPS 32-bit microcontroller. These changes include the implementation of a custom print instruction via I2C, modifications to the ROM memory to enable programming the MIPS 32 microcontroller from the instruction bus, custom instructions to handle addresses for conditional branches, and optimizations to general memory, register memory, and ROM memory. Figure 5.12 shows the Verilog code for the test bench.



```
MIPS MIPS_inst(
    .i_clk(clock),
    .i_high_speed_clk(hs_clock),
    .i_instr(i_instr),
    .i_instr_clk(i_instr_clk),
    .i_execution_begin(i_execution_begin),
    .i_arst(reset),
    .o_instruction(instr),
    .o_pc_cur(cur_pc),
    .o_pc_next(next_pc)
);

initial begin
    for(i = 0; i < PROGRAM_SIZE; i = i + 1) begin
        #3600
        for(j = 0; j < 8; j = j + 1) begin
            i_instr[j] <= example_progam[i * 8 + j];
        end
        i_instr_clk     <= 1;
        #3600 i_instr_clk <= 0;
    end
    #1000
    i_execution_begin <= 1;
end
```

Figure 5.12: MIPS 32 module test bench

As shown, this code sends machine instructions through the instruction bus to program the MIPS 32 microcontroller and then initiates program execution. Note that the test program (Listing 5.2) has been modified to run only 6 iterations of the loop to reduce simulation time.

Figure 5.13 displays the simulation results. The top screenshot shows the complete simulation, while the bottom screenshot zooms in on the code execution phase. Green signals represent inputs to the modified MIPS 32 (instruction bus), red signals represent internal signals of the MIPS 32 microcontroller (register and address space accesses), and violet signals represent outputs of the modified MIPS 32 microcontroller (I2C computation output bus).

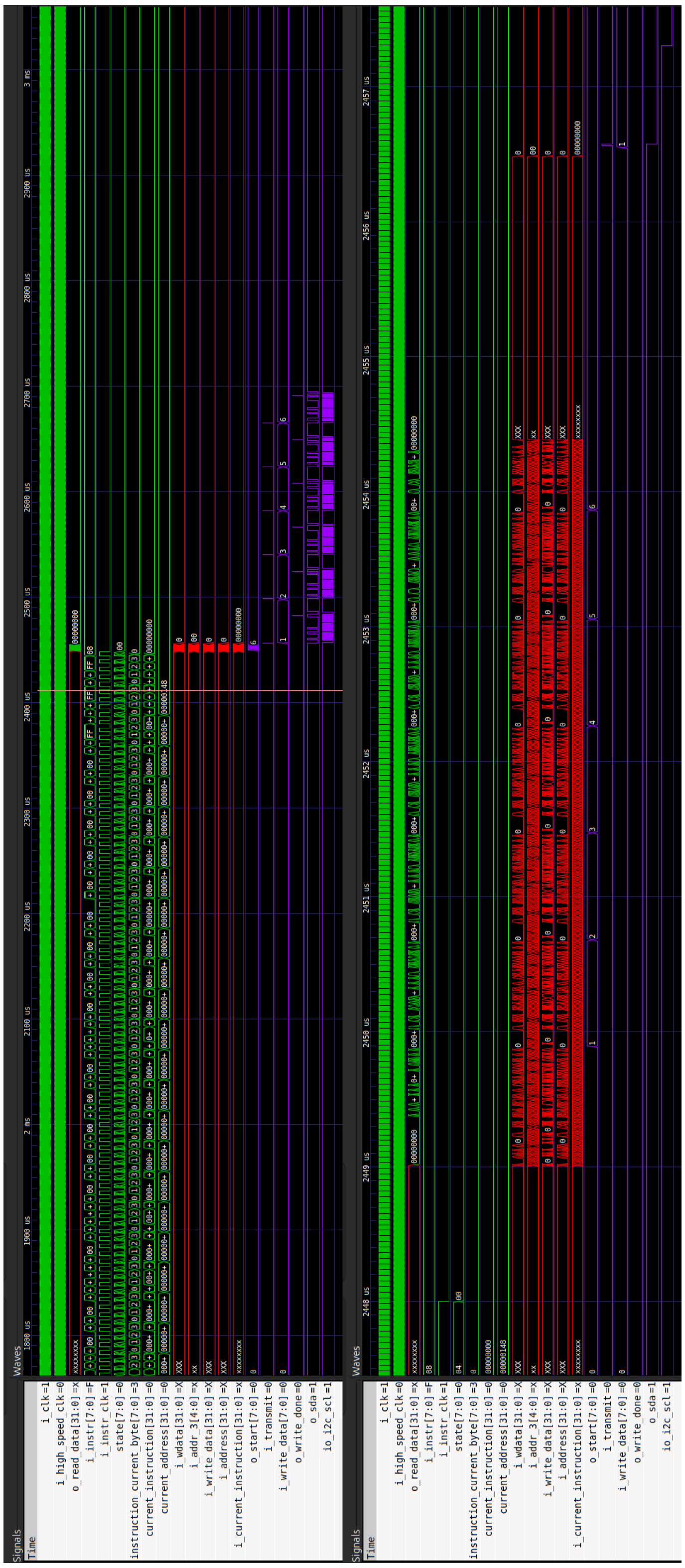


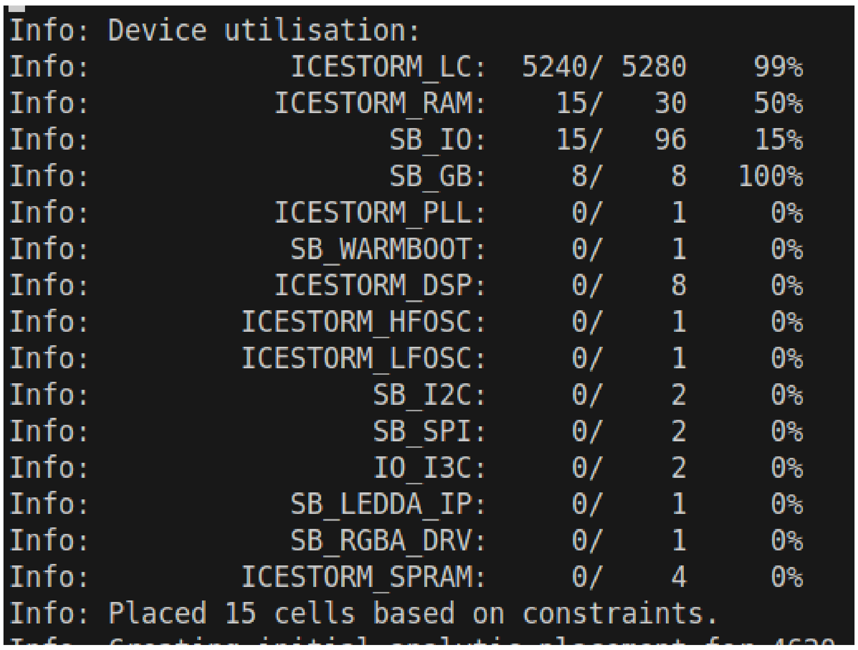
Figure 5.13: GTKWave MIPS 32 test bench results

The instruction bus speed also appears to be a bottleneck for code execution. The first section of the simulation, which only shows input changes, corresponds to the phase where the MIPS32 is being programmed. The section with red signal changes represents the MIPS32 microcontroller executing the programmed code, and the violet signals indicate the code's output. Examining the I2C write data signal confirms correct operation, with the outputs being 1, 2, 3, 4, 5, and 6.

Note that the output is not sent until the code finishes executing, as the implemented Print function is asynchronous. The contents are buffered and transmitted only after code execution is complete.

We will not analyze the machine code's execution time deeply, as the original MIPS32 design was not modified. However, we observe that code execution takes roughly 60 µs (10 µs per loop iteration). Keep in mind that this simulation runs at 100 MHz, whereas the target frequency for the ICE40UP5K is 6 MHz.

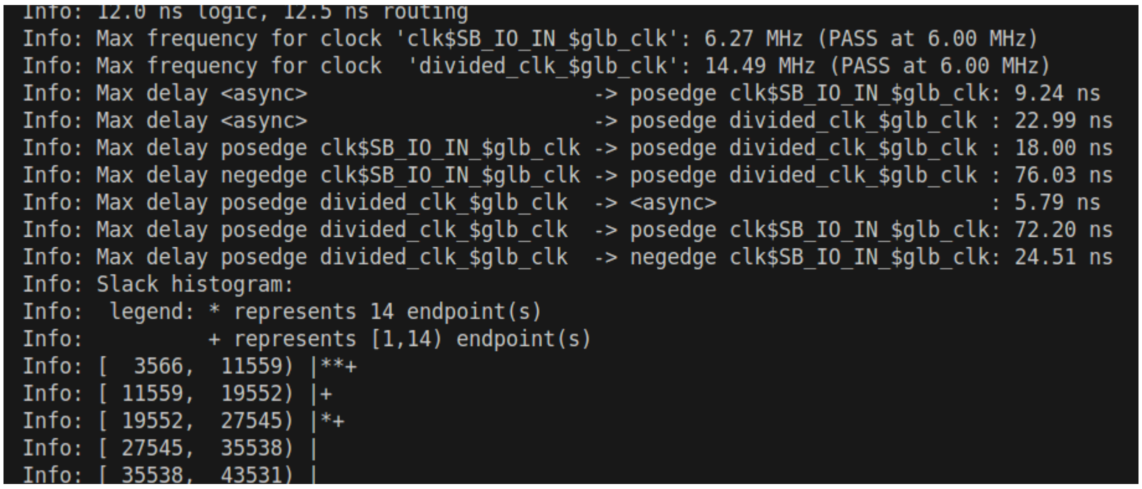
Figure 5.14 shows the hardware consumption of the MIPS32 module. The 50% RAM usage is due to optimizations, while the LC usage is at 99%, again highlighting the necessity of splitting the project across two ICE40UP5K devices.



```
Info: Device utilisation:
Info:            ICESTORM_LC:  5240/ 5280    99%
Info:           ICESTORM_RAM:    15/   30    50%
Info:                  SB_IO:    15/   96    15%
Info:                  SB_GB:     8/    8   100%
Info:           ICESTORM_PLL:     0/    1     0%
Info:            SB_WARMBOOT:     0/    1     0%
Info:           ICESTORM_DSP:     0/    8     0%
Info:         ICESTORM_HFOSC:     0/    1     0%
Info:         ICESTORM_LFOSC:     0/    1     0%
Info:                 SB_I2C:     0/    2     0%
Info:                 SB_SPI:     0/    2     0%
Info:                 IO_I3C:     0/    2     0%
Info:            SB_LEDDA_IP:     0/    1     0%
Info:            SB_RGBA_DRV:     0/    1     0%
Info:         ICESTORM_SPRAM:     0/    4     0%
Info: Placed 15 cells based on constraints.
```

Figure 5.14: MIPS32 module Apio hardware consumption report

Figure 5.15 shows the Apio timing analysis report for the MIPS32 module. The frequency is 6.27 MHz, which is just above our 6 MHz target.



```
Info: 12.0 ns logic, 12.5 ns routing
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 6.27 MHz (PASS at 6.00 MHz)
Info: Max frequency for clock  'divided_clk_$glb_clk': 14.49 MHz (PASS at 6.00 MHz)
Info: Max delay <async>                       -> posedge clk$SB_IO_IN_$glb_clk: 9.24 ns
Info: Max delay <async>                       -> posedge divided_clk_$glb_clk : 22.99 ns
Info: Max delay posedge clk$SB_IO_IN_$glb_clk -> posedge divided_clk_$glb_clk : 18.00 ns
Info: Max delay negedge clk$SB_IO_IN_$glb_clk -> posedge divided_clk_$glb_clk : 76.03 ns
Info: Max delay posedge divided_clk_$glb_clk  -> <async>                      : 5.79 ns
Info: Max delay posedge divided_clk_$glb_clk  -> posedge clk$SB_IO_IN_$glb_clk: 72.20 ns
Info: Max delay posedge divided_clk_$glb_clk  -> negedge clk$SB_IO_IN_$glb_clk: 24.51 ns
Info: Slack histogram:
Info:  legend: * represents 14 endpoint(s)
Info:          + represents [1,14) endpoint(s)
Info: [  3566,  11559) |**+
Info: [ 11559,  19552) |+
Info: [ 19552,  27545) |*+
Info: [ 27545,  35538) |
Info: [ 35538,  43531) |
```

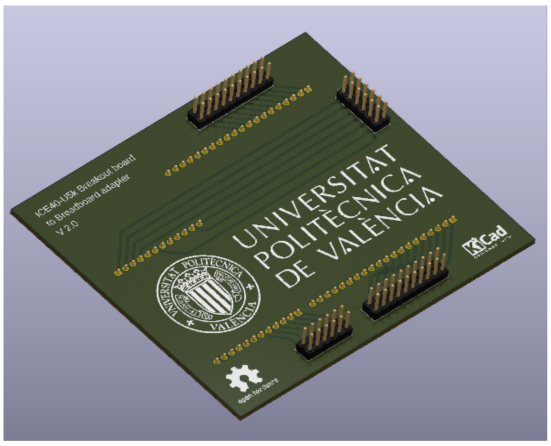
Figure 5.15: Apio timing analysis report for the MIPS32 module

5.2 On-premise tests

All tests in this section will use Algorithm 5.1. Additionally, all images presented in this section are from the SaleaeLogic software, not to be confused with GTKWave from the previous section. While GTKWave is used for simulations, SaleaeLogic is used to capture real-life digital 3.3 V or 5 V signals from the FPGAs.

5.2.1. Testing setup

To ensure clean and durable connections between boards during testing, the PCB shown in Figure 5.16 was used. This simple 2-layer PCB serves as an adapter from the ICE40UP5K Breakout board to a breadboard using pin headers. It also helps to keep the instruction bus connections as short as possible, minimizing unwanted noise. The PCB was fabricated and shipped by Aisler [48], a German PCB manufacturer.



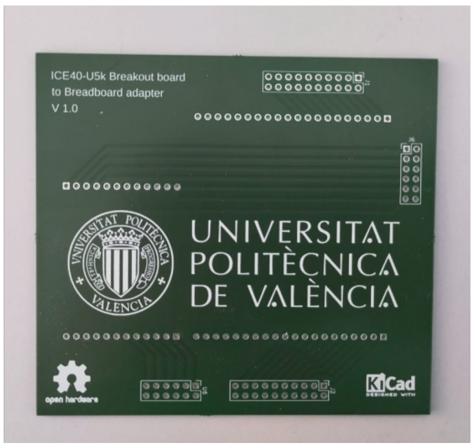


Figure 5.16: Kicad render of ICE40UP5K Breakout board to breadboard adapter and PCB Fabricated.

Figure 5.17 shows the resulting testing setup. All connections are made across four breadboards. The I2C to USB modules are located at the top right and left, connected to a laptop. The device at the top center is the SaleaeLogic analyzer, which will be used to debug and display the logic signals in this section. The two black PCBs on top of the adapter PCBs mentioned earlier are the two ICE40UP5K breakout boards. The breadboard shows the connections for the 8-bit instruction bus and two I2C buses.

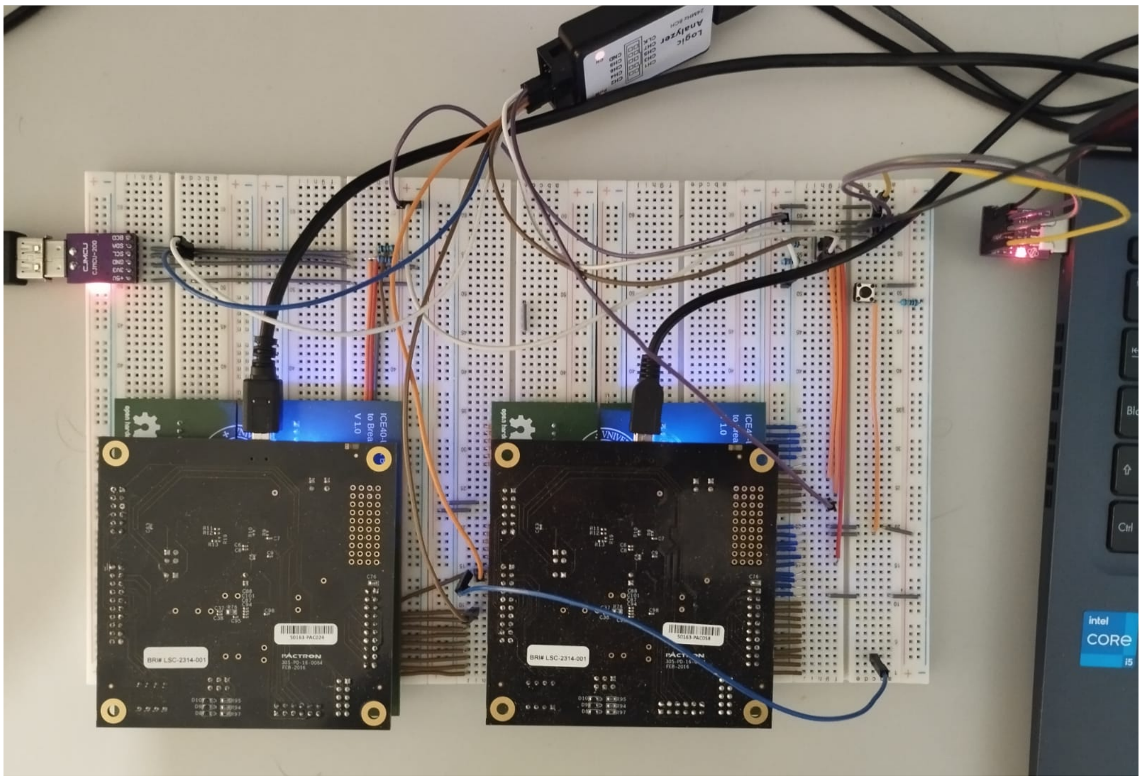


Figure 5.17: Testing setup of both interpreter and code execution board.

For measuring the power consumed by the devices, the digital USB tester shown in Figure 5.18 was used. This device enables the measurement and storage of current and power data for the FPGA boards during testing.

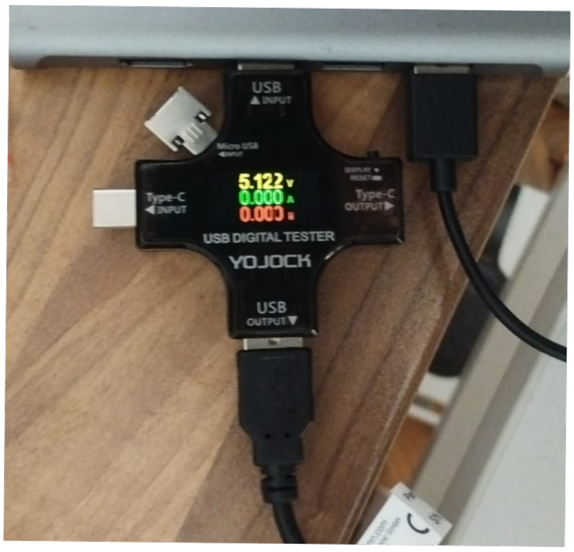
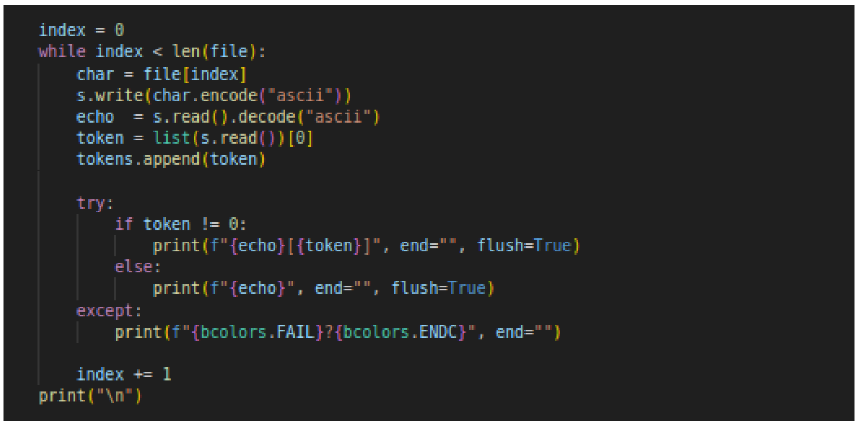


Figure 5.18: Digital USB tester

As for the testing script, Figure 5.19 shows the Python script used to upload the source code to the interpreter via Python's Serial module.



```
index = 0
while index < len(file):
    char = file[index]
    s.write(char.encode("ascii"))
    echo  = s.read().decode("ascii")
    token = list(s.read())[0]
    tokens.append(token)

    try:
        if token != 0:
            print(f"{echo}[{token}]", end="", flush=True)
        else:
            print(f"{echo}", end="", flush=True)
    except:
        print(f"{bcolors.FAIL}?{bcolors.ENDC}", end="")

    index += 1
print("\n")
```

Figure 5.19: Source code updloader script

5.2.2. Results

The following signals have been measured: sda and scl lines of the source code I2C bus, sda and scl lines of the computing results bus, and lastly, clk and begin execution signals of the instruction bus. Figure 5.20 shows the results for the tests measured by the SaleaeLogic software.

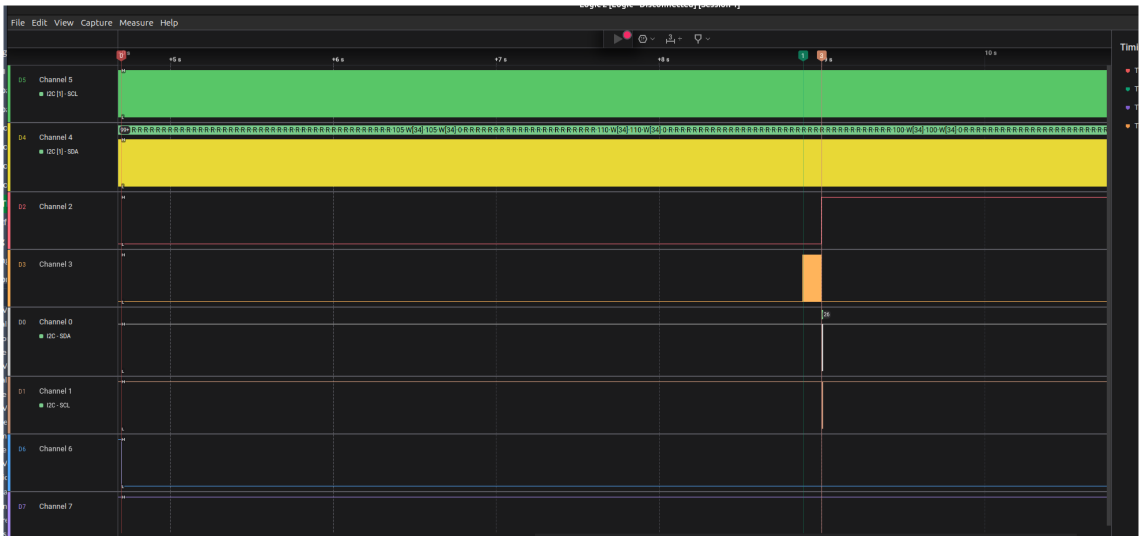


Figure 5.20: SaleaeLogic Fibonacci source code test

The top two channels correspond to the source code bus, the next two channels represent the begin execution flag and instruction bus clock signal, and the final two channels correspond to the computing results bus. As indicated by the markers in the image, the results are divided into three sections: code upload and scanning, machine code generation and parsing, and finally, results being transmitted on the results bus.

Figure 5.21 shows the machine code upload and results. Each positive edge of the clock signal in this bus corresponds to 1/4 of a machine code instruction being uploaded to the execution FPGA. The top two channels correspond to the source code bus, the next two channels represent the begin execution flag and instruction bus clock signal, and the final two channels correspond to the computing results bus. As indicated by the markers in the image, the results are divided into three sections: code upload and scanning, machine code generation and parsing, and finally, results being transmitted on the results bus.

Figure 5.21 shows the machine code upload and results. Each positive edge of the clock signal in this bus corresponds to 1/4 of a machine code instruction being uploaded to the execution FPGA.

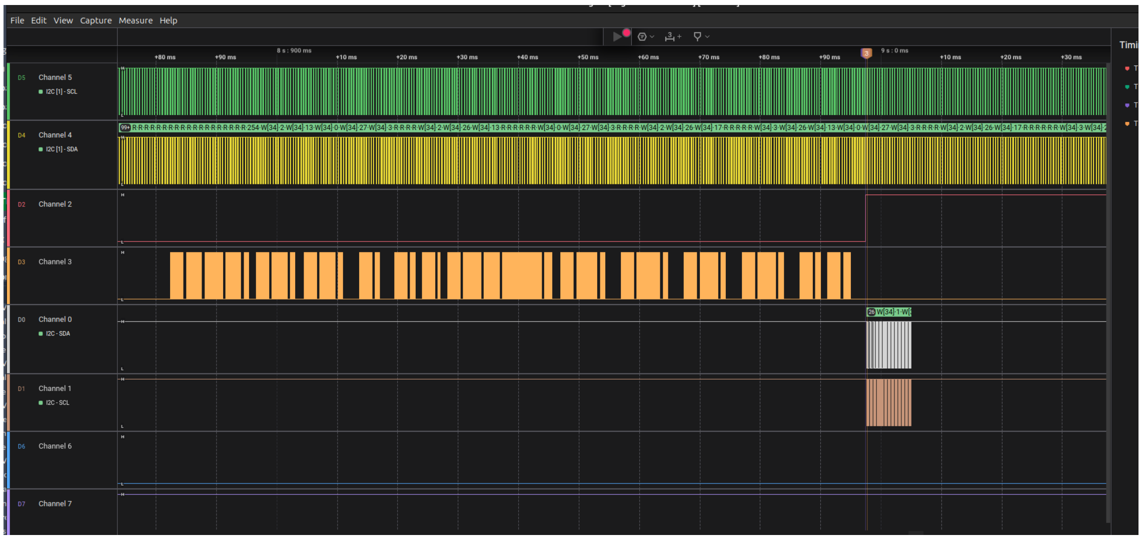


Figure 5.21: SaleaeLogic Machine code generation and scanner

Finally, Figure 5.22 shows a zoomed-in view of the results being returned to the computer. As decoded by the SaleaeLogic software and displayed in the format "w[34] RESULT", the results of the print statements are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 133, and 233. This is correct and corresponds to the first 13 numbers of the Fibonacci series.

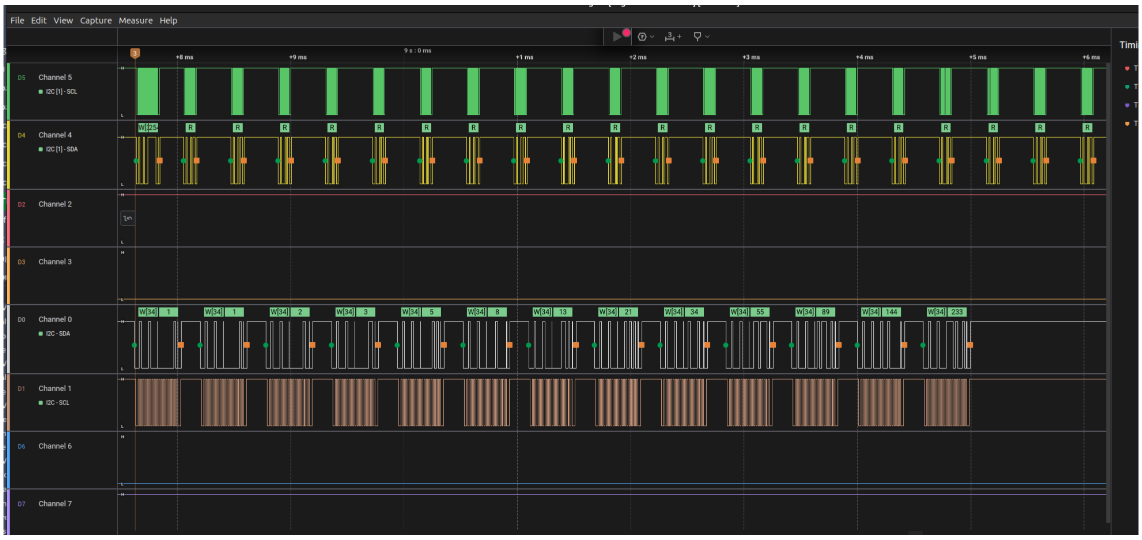


Figure 5.22: SaleaeLogic test results

As shown in the image, uploading the source code takes approximately 3 seconds. This is expected, as each character is echoed and the system polls until Python responds. Parsing and machine code generation take 115 ms, code execution takes 200 µs, and sending the results back (once the computation is finished) takes 7.3 ms. These times are significantly higher than those from simulations, as the speed of the I2C and instruction buses limits execution speed. Finally, the USB tester shows a steady 0.12 A at 5.01 volts being consumed by each board throughout the tests, totaling 0.60 W per board and 1.2 W overall.

6 Conclusions

In recent decades, the software industry has experienced a phenomenon named "code bloat," where the exponential growth of hardware resources has caused a lack of attention to code efficiency. However, this exponential increase in hardware growth is slowing, and optimizations are emerging at higher software levels (hardware architecture, software, and compilers) instead of lower levels (transistor technology). These optimizations will be challenging for an industry that has been trained to prioritize development speed over efficiency.

Recent benchmarks for the JavaScript V8 engine demonstrate that 10% to 40% of the execution time for the most widely used programming language (according to the Stack-Overflow survey) is dedicated to runtime compilation [49]. This runtime compilation allows the use of higher abstractions while hindering performance. Therefore, promoting ease of coding in high-level programming languages often requires implementing complex compilers and algorithms that run at execution time on top of several software layers. This thesis shows that even with limited resources, significant speedups, and energy savings can be achieved by bringing these high-level abstractions and algorithms closer to the hardware. Effectively shorting the size of the stack on which high-level abstractions to the minimum necessary, directly on the silicon / FPGA fabric.

6.1 Impact on time/space efficiency

For this proof of concept, the bandwidth of communication buses in low-budget FPGAs limited the interpretation speed. However, the same interpreter (using only 5k logic cells and 15 5b RAM blocks) with a faster communication protocol such as PCI Express can achieve up to 22.45 MB per second scanner throughput and 17 million reductions per second. This is twice the performance of the JavaScript V8 engine (10 MB per second) running on an Intel i5 3 GHz system (with a clock frequency three orders of magnitude higher).

While these speedups apply only to code compilation and not execution, they suggest that high-level compiler algorithms using dynamic data structures and sequential logic can be significantly accelerated when implemented in hardware. This is still true even when implementing these layers in FPGAs, which typically have lower clock frequencies and more limitations than ASICs; this demonstrates the potential for programmable hardware to enhance not only the performance of high-level algorithms but also to retain in hardware the reprogramming flexibility usually associated with the software. This possibility opens the door for consumer electronics and servers with FPGAs that can be reprogrammed at runtime, optimizing performance for the currently running programming language.

6.2 Impact on energy efficiency

Additionally, the potential energy savings are substantial. This proof of concept ran on an FPGA board consuming only 0.60 watts, while a modern i5 processor can consume up to 20 watts (just the processor) during intensive workloads [50]. For example, the i5-13400 CPU averages 218.9 watts during the Cinebench R23 multithreaded test. Mid-range desktop PCs typically consume around 350-600 watts under high workloads, with the i5-13400 CPU accounting for 36% to 62% of total power consumption during high-load scenarios. Given that many of today's applications run in interpreted high-level languages, these potential energy savings are highly significant.

6.3 Future work

More research is needed to translate these energy savings and speedups into a more realistic environment with a larger and more complete language, such as JavaScript. These results are especially relevant when applied to code execution optimizations, as in JavaScript, compilation accounts for only 10% to 40% of total execution time.

This work has demonstrated how reprogrammable hardware can enhance high-level language abstractions, delivering high speed and energy savings to the software industry while maintaining development efficiency and ease of programming. In the near future, we aim to explore further the potential of using reprogrammable hardware to build more efficient software systems.

Starting by enhancing the speed of the instruction, source code, and computation buses, this solution would be better suited as a high-level language hardware accelerator. Following an architecture similar to GPUs, these boards could use PCI Express and large amounts of RAM to communicate with the system's main CPU and perform calculations. For example, in the case of JavaScript, the language could be compiled into bytecode at runtime, with a bytecode CPU (mapping each bytecode instruction 1:1 to machine code) implemented on the accelerator. This approach would allow for intermediate code optimizations common in JavaScript while providing a modular, extensible solution. Additionally, the new CPU could implement custom memory controllers and hardware-level support for high-level concepts like garbage collection and object creation.

7 Relation with completed studies

The following subjects have a close relationship with the knowledge used to implement this thesis:

- **11542 Computer Fundamentals**
- **11540 Computer Science Physics**
- **11544 Computing Technologies**
- **11552 Computers structures**
- **11557 Programming languages, technologies and paradigms**
- **11564 Automata theory and formal languages**
- **11590 Programming languages and language processors**
- **11593 Algorithms**

Bibliography

Articles, Books, Conferences, Reports

[1] Aho, Alfred V.; Lam, Monica S.; Sethi, Ravi; Ullman, Jeffrey D. Compilers : Principles, Techniques, & Tools (2nd ed.). Pearson Addison-Wesley, 2007.

[2] Cooper, Keith and Torczon, Linda . Engineering a Compiler. *Morgan Kaufmann; 2nd edition*, 2012.

[3] Feynman, Richard P. There's plenty of room at the bottom. In Horace D. Gilbert (editor), *Miniaturization*. Reinhold Publishing, pages 282–296, 1961.

[4] Huang, Sitao ; Wu, Kun ; Jeong, Hyunmin ; Wang, Chengyue . PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow. *IEEE Transactions on Computers, 70(12):2015-2028*, 2021.

[5] Huang, Sitao ; Wu, Kun; Rahul Chalamalasetti, Sai. A Python-based High-Level Programming Flow for CPU-FPGA Heterogeneous Systems. In *Proc. of the 2021 IEEE/ACM Programming Environments for Heterogeneous Computing*, pages 20-26, November, 2021.

[6] Kim, Austin and Chang, Morris . Designing a Java microprocessor core using FPGA technology. *Computing & Control Engineering 11(3):135 – 141*, 2000.

[7] LaMeres, Brock J. Introduction to Logic Circuits & Logic Design with Verilog. *Springer*, 2017.

[8] Leiserson, Charles E. and Thompson, Neil C. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science 368, eaam9744*, 2020.

[9] Moore, Gordon E. Cramming more components onto integrated circuits. *Electronics, 38(8):114-117*, 1965.

[10] Schmidt, Andrew G and Weisz, Gabriel . Evaluating Rapid Application Development with Python for Heterogeneous Processor-based FPGAs. In *Proc. of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 121-124, IEEE Press 2017.

[11] Schoeberl, Martin . JOP: A Java Optimized Processor. In *Proc. of On The Move to Meaningful Internet Systems, OTM 2003 Workshops*, LNCS 2889:346-359, 2003.

[12] Skalicky, Sam snf Monson, Joshua . Hot & Spicy: Improving Productivity with Python and HLS for FPGAs. *In Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 85-92, IEEE Press, 2018.

[13] Takamaeda-Yamazaki, Shinya. PyVerilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL In *Proc. of Applied Reconfigurable Computing, ARC 2015*, LNCS 9040:451–460, Springer 2015.

[14] Turing, Alan M. Computing Machinery and Intelligence. *Mind 59:433-460*, 1950.

[15] Turing, Alan M.. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230-265, 1936.

[16] Uguen, Yohann and Petit, Eric . PyGA: A Python to FPGA Compiler Prototype *Proc. of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems, AI-SEPS 2018, pages 11 - 15*, 2018.

[17] von Neumann, John. First Draft of a Report on EDVAC. More School of Electrical Engineering, University of Pennsylvania, June 30, 1945.

[18] Winskel, Glynn . The formal semantics of programming languages *Foundations of Computing Series*, The MIT Press, 1993.

Web pages

[19] First commercially available microprocessor Intel 404 https://es.wikipedia.org/wiki/Intel_4004.

[20] Coding From 1849 to 2022: a Guide to The Timeline of Programming Languages IEEE computer society https://www.computer.org/publications/tech-news/insider-membership-news/timeline-of-programming-languages.

[21] Survey of most popular and used technologies Stackoverflow https://survey.stackoverflow.co/2024/.

[22] ITRS, International Technology Roadmap for Semiconductors 2.0 executive report (2015) https://www.semiconductors.org/wp-content/uploads/2018/06/0_2015-ITRS-2.0-Executive-Report-1.pdf

[23] Introduction to FPGAs Digikey https://www.digikey.com/en/maker/projects/introduction-to-fpga-part-1-what-is-an-fpga/3ee5f6c8fa594161a655a9f960060893

[24] iCE40LP/HX ShawnHymel GithubRE IS NO SUCH FIGUREFigure 4.1.3 shows the Verilog template used to imple- https://github.com/ShawnHymel/introduction-to-fpga/raw/main/datasheets/iCE40LPHXFamilyDataSheet.pdf

[25] General Fabric and Routing Resources FPGAKey https://www.fpgakey.com/tutorial/section674

[26] Visual Studio Code VSCode documentation https://code.visualstudio.com/docs

[27] Kicad EDA website https://www.kicad.org/

[28] Apio Documentation website https://Apiodoc.readthedocs.io/en/stable/

[29] Yosis Synthesis Github https://github.com/YosysHQ/yosys

[30] GTKWave viewer Sourceforge website https://gtkwave.sourceforge.net/

[31] Python Spain website https://es.Python.org/

[32] Saleae logic Analyzers website https://www.saleae.com/?

[33] PYNQ Introduction website https://pynq.readthedocs.io/en/latest/

[34] ICE40-UP5K FPGA, Lattice products website https://www.latticesemi.com/en/Products/FPGAandCPLD/iCE40UltraPlus

[35] ICE40-UP5K Breakout Board, Mouser electronics https://eu.mouser.com/new/lattice-semiconductor/lattice-ice40-ultraplus-breakout-board/

[36] Regular expressions Wikipedia https://en.wikipedia.org/wiki/Regular_expression

[37] Context-free Grammars Wikipedia https://en.wikipedia.org/wiki/Context-free_grammar

[38] Attribute Grammars Wikipedia https://en.wikipedia.org/wiki/Attribute_grammar

[39] @vsilchuk Softcore Verilog 32 bit MIPS processor implementation , Github https://github.com/vsilchuk/Verilog_HDL_single_cycle_MIPS_processor

[40] Thompson's construction , Wikipedia, https://en.wikipedia.org/wiki/Thompson%27s_construction

[41] Powerset construction algorithm, Wikipedia, https://en.wikipedia.org/wiki/Powerset_construction

[42] Trie Wikipedia, https://en.wikipedia.org/wiki/Trie

[43] I2C specification NPX manual, https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[44] Double-ended queue, Wikipedia, https://en.wikipedia.org/wiki/Double-ended_queue

[45] Symbol table, Wikipedia, https://en.wikipedia.org/wiki/Symbol_table

[46] Hash table, Wikipedia, https://en.wikipedia.org/wiki/Hash_table

[47] Hash table, Wikipedia, https://en.wikipedia.org/wiki/Hash_table

[48] Aisler PCB manufacturer, webpage, https://aisler.net/

[49] Javascript V8 engine benchmarks, webpage, https://v8.dev/blog/real-world-performance

[50] Processor power consumption study, webpage, https://v8.dev/blog/real-world-performance

[51] Field-Programmable Gate Arrays, Wikipedia, https://en.wikipedia.org/wiki/Field-programmable_gate_array

[52] Soft Microprocessor, Wikipedia, https://en.wikipedia.org/wiki/Soft_microprocessor

[53] I2C, Wikipedia, https://en.wikipedia.org/wiki/I%C2%B2C

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	**Alto**	**Medio**	**Bajo**	**No Procede**
ODS 1. **Fin de la pobreza.**				**x**
ODS 2. **Hambre cero.**				**x**
ODS 3. **Salud y bienestar.**				**x**
ODS 4. **Educación de calidad.**				**x**
ODS 5. Igualdad de género.				**x**
ODS 6. **Agua limpia y saneamiento.**				**x**
ODS 7. Energía asequible y no contaminante.	**x**			
ODS 8. **Trabajo decente y crecimiento económico.**	**x**			
ODS 9. Industria, innovación e infraestructuras.	**x**			
ODS 10. **Reducción de las desigualdades.**				**x**
ODS 11. **Ciudades y comunidades sostenibles.**				**x**
ODS 12. Producción y consumo responsables.				**x**
ODS 13. Acción por el clima.				**x**
ODS 14. Vida submarina.				**x**
ODS 15. **Vida de ecosistemas terrestres.**				**x**
ODS 16. **Paz, justicia e instituciones sólidas.**				**x**
ODS 17. **Alianzas para lograr objetivos.**				**x**

etsinf
Escola Tècnica Superior d'Enginyeria **Informàtica**

ETS Enginyeria Informàtica
Camí de Vera, s/n. 46022. València
T +34 963 877 210
F +34 963 877 219
etsinf@upvnet.upv.es - www.inf.upv.es

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

El trabajo presentado propone una solución al hecho de que escribir código eficiente es tedioso, difícil y requiere una gran cantidad de tiempo. Esto ha causado en los últimos años la adopción de prácticas y metodologías en el campo del desarrollo de software en las que se abusa de la abundancia de recursos de hardware para implementar código ineficiente pero rápido de programar mediante abstracciones y lenguajes de alto nivel. Dado que el trabajo propuesto demuestra la viabilidad de ahorros en tiempo de ejecución y energía considerables, esto tendría un impacto en:

- **Energía asequible:** ya que la reducción del consumo energético de la infraestructura que soporta la industria del software sería es considerable.
- **Crecimiento económico:** ya que limitaciones físicas nos impiden seguir consiguiendo hardware a la velocidad que lo hemos estado haciendo, mientras que las tecnológicas y gigantes de software siguen creciendo a la misma velocidad de siempre. Si las tendencias no cambian, llegará un punto donde La industria del software necesitara de optimizaciones como la propuesta para seguir creciendo.
- **Industria, innovación e infraestructuras:** ya que es una alternativa que no se ha propuesto antes (a pesar de que ha habido intentos parecidos) y afecta directamente a la industria del software.