Neptune

An Intermediate Language for Music Composition on the Commodore 64

Emil S. Andersen, Felix B. Lindberg
Alberte Lohse, Nikolaj K. van Gool
Cecilie S. Vebner

AALBORG UNIVERSITY
STUDENT REPORT

Institut for Datalogi
AC Meyers Vænge 15
DK-2450 København

Title
Neptune

Project period
February - May 2025

Group
Group 8

Group members
Emil S. Andersen
Felix B. Lindberg
Alberte Lohse
Nikolaj K. van Gool
Cecilie S. Vebner

Semester
SW4

Supervisor
Léon Gondelman

Github Repository
`https://github.com/Group-CAFNE/Neptune`

Number of pages
72

Abstract
This project details the development of Neptune, an intermediate language and compiler that interacts with the Commodore 64's SID 6581 chip. Neptune's purpose is to bridge the gap between high-level programming and the functionality of retro hardware. This report outlines the source language design by expanding on the general considerations leading to its syntax. It also delves into the workings of the SID chip, exploring how to create music in the target language, 6502 Assembly.

The language and compiler are developed using OCaml, and the report expands on the entire compilation process and its implementation, from creating a lexer to a code generator, and finally, how to execute the output file. Reflecting on the final product's main limitations, the absence of ADSR and filter adjustment functionality emerges as the most critical missing feature. New features for future development are also presented. The report concludes by presenting how Neptune represents a step towards making authentic chiptune music creation more widely accessible.

Contents

Chapter 1

Introduction

We can easily take for granted the conveniences modern life provides us, the tediousness we are now exempt from. Before the internet, you would have to scour through libraries. Before computers, essays were written in pen. We can all be thankful for these innovations, but could there be some merit to exploring past approaches?

One interesting aspect to consider is the imperfections of vintage media. There is a certain art to it; the scan lines of CRT TV screens or the crackle of a vinyl. Most people will be familiar with the distinctive sound of chiptune music with its *beeps* and *boops*, and its harsh noises. This unique musical style is not created with traditional instruments, but is instead artificially synthesised by sound chips of gaming consoles and computers from the late 1970s to early 1980s (see Appendix A.1). This style of music evokes a sense of nostalgia, reminiscent of a simpler time, the golden age of video arcade games. Even more remarkably, chiptune artists had to work within the limitations of the sound chips, both in terms of the sounds they could produce, but also the technical knowledge required to utilise them.

As technology has advanced through the years, the physical sound chips of retro hardware have become obsolete. Modern software has made it possible to replicate the nostalgic sounds of chiptune without the previous constraints of the hardware. When the restraints are removed, it diminishes the creative and innovative efforts that were once essential to the genre. As technology keeps developing, we are at risk of neglecting this important aspect of our digital history. The books and websites providing the resources to learn these skills are often convoluted, antiquated, and difficult to find, further obscuring this knowledge.

Fortunately, a recent resurgence in communities surrounding the production and consumption of chiptune music shows that the appreciation for the genre has only increased. With this project, we pay homage to the early years of chiptune music and the communities surrounding it. Replicating the sound of chiptune would be an interesting project in itself, but our objective is to produce true, authentic chiptune music, and as such, the design of our language is influenced by a thorough understanding of the chosen hardware.

We have decided to focus on the sound chip SID 6581, which has been widely recognised as one of the most iconic sound chips of the chiptune era (see Appendix A.2). This chip is found in the Commodore 64, often referred to as C64, which was a computer praised for its advanced sound output when it debuted in 1982. The SID chip was popular even after newer, more sophisticated chips had been released, due to the charm of its distinctively gritty and textured sound. It can produce no more than three voice outputs in only four different waveforms, which may seem quite restrictive by current-day standards. However, these limitations, in particular, are a distinct feature of chiptune music and are one of the reasons why the SID chip remains popular to this day. It tows the line between strict limitations and creative flexibility, and therefore, composing music for it becomes a challenge of ingenuity and creativity.

Given the considerations presented above, the goal of the project is to answer the following question:

How can we design an intermediate language and a corresponding compiler that facilitates interaction with the SID chip, enhancing accessibility for chiptune music production?

The product of this project is *Neptune*, a Domain Specific Language (DSL), as well as a compiler designed to generate Commodore 64 executable files from Neptune code. Together, these are designed specifically to facilitate the aforementioned interaction. The most important mission in this regard is to abstract from the complexities of the hardware architecture and lean more towards music composition. There are many possibilities of how a language like this could look, both in terms of structure and priorities. Rather than pursuing a single direction, which would limit the use cases for our language, we have chosen to design an *intermediate* language that promotes extensibility. The idea is for it to be as basic as possible, while still attaining the objective of being removed from the intricacies of the C64's native language. We have taken a step towards removing the barrier for future developers, allowing them to explore what the SID chip is capable of. To provide a general understanding of the capabilities of our Neptune and the compiler, see figure 1.1.

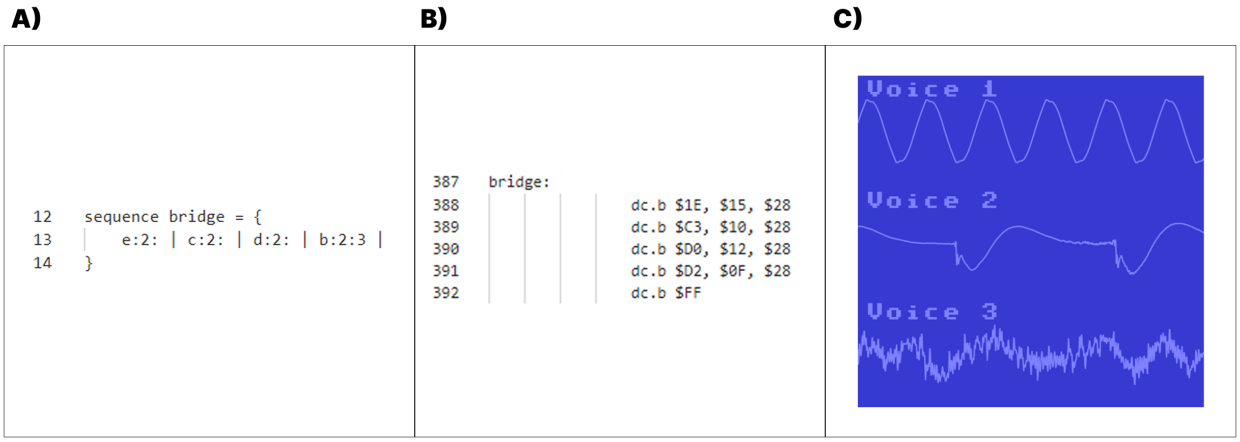


Figure 1.1: This provides an overview of the compilation process of the Tetris theme. First, we are presented with a snippet of the source code in the language of Neptune (A), the corresponding target code in 6502 Assembly (B), and finally, a visualisation of the sound output, using the oscilloscope SidWizPlus (C)

A demonstration of our product can be seen in the video linked below, in which Neptune is used to compose the well-known Tetris theme.

`https://www.youtube.com/watch?v=EqYkdAdsoK8`

Outline

In the following chapters, we will use concrete Neptune source code as a running example to expand on the process of developing Neptune and our compiler (the source code example can be seen in Appendix H.1). Chapter 2 will expand on the design of Neptune and formalise the syntax and semantics of the language. In Chapter 3, we determine the target language. Here, we present relevant information regarding the SID chip and the Commodore 64 to enable the creation of a music routine for producing sound. In Chapter 4, the implementation of the compiler is presented by exploring every phase and component in the compilation process. Furthermore, we present how we have attempted to validate the compiler's functionality by conducting unit, integration, and acceptance tests. In Chapter 5, we discuss the quality of the final product by reflecting on the limitations of both the compiler and Neptune, and how they can be improved and developed in the future. Finally, in Chapter 6, we conclude the whole project.

Chapter 2

Language Design

In this chapter, we will expand on the general considerations leading to the syntax of our language, Neptune. Then, the source language is formalised by specifying the syntax and semantics, and finally, the language is evaluated by exploring its readability, writability, and reliability.

2.1 Designing the Source Language

The first decision to be made when designing a language is which programming paradigm is most fitting for the context of the language. The two primary paradigms, imperative and declarative, each constitute a distinct model for problem-solving with rules and principles. Where an imperative programming language will describe sequences of instructions to accomplish a task, a declarative programming language describes the tasks that must be accomplished. Therefore, declarative programming focuses more on the end result than the computations that produce the result [1].

In the case of music, it is rarely defined by variables or computations with control structures, but instead with notation of predefined notes, like sheet music. For this reason, our source language is purely declarative, as users can only specify what music should be played, rather than controlling the procedures for producing the music [2, p. 714]. In the following, we will explain the process of designing syntax for notes, sequences, musical parameters, and voices for Neptune.

Notes

As the aim is to create a language for writing chiptune music for the Commodore 64, we take inspiration from the music theory upon which chiptune music is based, as it could make the syntax more intuitive for people with musical knowledge to comprehend and build upon. The manual for the Commodore 64 shows that the SID chip follows Western music theory, and therefore it is most natural that our syntax does as well [3]. Consequently, we have explored existing solutions for compiling music with a programming language. Among other tools, we found the music engraving tool Lilypond, which compiles programs into sheet music following Western music theory and notation [4]. Figure 2.1 shows the LilyPond syntax that influenced our design. The syntax consists of a list of notes represented by a tone name and a fractional duration. The tones are represented by one of the letter

names: C, D, E, F, G, A, and B [5]. In Western music theory, there are 12 distinct tones, so accidentals are used for the remaining five notes. Accidentals are symbols, such as sharps (♯) and flats (♭), that indicate that a tone should be raised or lowered [6, p. 7]. In the LilyPond syntax in figure 2.1, the sharp accidental is represented with the letters *is*. The fractional duration of a note is indicated by the number following the tone name. In our language, the fractional duration is limited to the range of 1 to 16 for the sake of simplicity (see Appendix B.2 for an explanation of the numbers) [6, p. XIV].

```
g8 fis a4
g:8:3 f#:8:3 a:4:3
```

Figure 2.1: At the top, the LilyPond syntax for a sequence of notes is shown, with the corresponding syntax in Neptune shown below

In LilyPond's syntax, if a note lacks a duration, it inherits the previous note's duration. To simplify implementation and enhance comprehensibility, we have chosen a standardised structure for notes in our language. As we are designing an intermediate language, this standardised note structure should make it straightforward for external developers to build upon. The structure will be that each note will be written with a tone letter name, an optional accidental, a duration, and an octave.

As mentioned, Western music has twelve distinct notes, though a piano has more than twelve keys. This is due to the repeating pattern of the twelve tones, and the octave represents how low or high on the piano a tone should be played (see Appendix B.3). Consequently, the note A4, which is the tone A in the fourth octave, has a higher pitch than A3 (see Appendix B.5). Therefore, providing control over the octave is crucial for creating varied and interesting melodies. LilyPond's symbol for rest notes, which are pauses in the music, is also used in our syntax. These notes are represented by the letter *r*, followed only by a duration. For accidentals, the sharp symbol ♯ is used for raising a note, following traditional music notation, and an underscore _ for lowering a note.

Sequences

The next step is to define the grouping of these notes, which enables the writing of melodies. To visually group notes together, they are written in sequences, using curly brackets '{' and '}' as delimiters. Similar to LilyPond, whitespace is used to separate notes within sequences, as this allows users to format the sequences however they prefer, which can improve comprehensibility. Users also have the possibility to write the vertical bar '|' in the sequences, as it is reminiscent of a separator used in Western music notation. Furthermore, sequences are assigned identifiers to easily manage and reuse them without repeatedly defining the entire sequence. To provide flexibility, sequences can be any length, as this supports defining an entire song within a single sequence or multiple shorter sequences that can be combined. The assignment operator '=' is used to visually assign sequences to identifiers for better comprehensibility, which is a common practice in most programming languages. An example of concrete syntax of the source language, which defines sequences of notes, can be seen in figure 2.2.

```
sequence seq1 = { c:1:4 }
sequence seq2 = { d:4:4 e_:4:4 }
sequence seq3 = { g:8:3 f#:8:3 a:4:3 }
```

Figure 2.2: An example of the Neptune syntax for defining sequences of notes

Musical Parameters

As mentioned previously, the fractional duration of each note is denoted by a number. However, this number alone cannot give a duration, as the duration is relative to the tempo, being the number of beats per minute and the time signature (see Appendix B.2 and B.4) [7]. Therefore, it is important to add syntax that allows control over the musical parameters: tempo, time signature, and standard pitch. The pitch of every note is calculated based on the frequency of the standard pitch, which is the note A4. By adjusting the standard pitch, you can raise or lower every note [8]. For an example of the concrete syntax, see figure 2.3.

Voices

Finally, there should be structures that allow sequences to be combined into whole sections of music. Among many possible designs, the most straightforward solution is to take inspiration from the voice system of the SID chip (see section 3.1). The sound played by the SID chip consists of three voices, which the syntax will reflect, with each voice being represented as an array of sequences. Moreover, the use of different waveforms is central to producing chiptune music, as it allows for recreating the sounds of different instruments. The four waveforms supported by the SID chip are *triangle*, *sawtooth*, *variable pulse*, and *noise*.

Consequently, for each sequence, a waveform can be specified, making each element of the voice array a pair of a sequence and its associated waveform. Together with the flexible length of sequences, this allows users to change the waveform dynamically, which is consistent with the functionality of the SID chip. Figure 2.3 shows a complete Neptune program, including the syntax for voices. For a more detailed explanation of waveforms, see Appendix D.

```
tempo = 120
timeSignature = (4,4)
standardPitch = 440

sequence seq1 = { c:1:4 }
sequence seq2 = { d:4:4 e_:4:4 }
sequence seq3 = { g:8:3 f#:8:3 a:4:3 }

voice1 = [(seq1, square)]
voice2 = [(seq2, triangle), (seq2, triangle)]
voice3 = [(seq3, sawtooth), (seq2, noise)]
```

Figure 2.3: An example of a complete program written in Neptune, with syntax for defining the musical parameters, sequences, and voices

2.2 The Formal Syntax and Semantics of Neptune

When formalising a programming language, it is important to cover the *syntax* and *semantics*. The syntax of a programming language is the rules that define the structure and arrangement of its statements and expressions. Semantics is the meaning of those statements and expressions, and how they are interpreted [2, p. 134].

2.2.1 Syntax

The grammar of a language specifies permissible arrangements of words with a set of production rules. We will in this section use grammar and *context-free grammar* (CFG) interchangeably to refer to the grammar of our language.

We use EBNF (Extended Backus-Naur Form), a formal notation for specifying programming language syntax, to define the grammar of our language. Aligned with EBNF notation, abstractions denoted with <> are nonterminals. These nonterminals are defined on the left-hand side. They can have multiple distinct definitions, representing various possible syntactic structures. On the right-hand side are the terminals. Terminals consist of lexemes[1] and tokens [2, p. 138]. It should be noted that in our CFG, the production rules are denoted with ::= to distinguish between the left-hand side and right-hand side. The assignment operator '=' is part of the language syntax and is simply a terminal in the grammar.

[1] Lexemes are the lowest levels of syntactic units. A lexeme is a sequence of characters in the source language that matches a pattern defined by a token

$$
\begin{aligned}
\langle file \rangle ::= & \\
& \text{tempo} = \langle int \rangle \\
& \text{timeSignature} = (\langle int \rangle, \langle int \rangle) \\
& \text{standardPitch} = \langle int \rangle \\
& \langle seqdef \rangle^{*} \\
& \text{voice1} = [(\langle ident \rangle, \langle waveform \rangle)^{*}] \\
& \text{voice2} = [(\langle ident \rangle, \langle waveform \rangle)^{*}] \\
& \text{voice3} = [(\langle ident \rangle, \langle waveform \rangle)^{*}] \\
\langle seqdef \rangle ::= & \text{ sequence } \langle ident \rangle = \langle seq \rangle \\
\langle seq \rangle ::= & \{\langle note \rangle^{+}\} \\
\langle note \rangle ::= & \langle tone \rangle \ \langle acc \rangle ? \ : \ \langle frac \rangle \ :? \ \langle oct \rangle ? \\
\langle tone \rangle ::= & \ a \mid b \mid c \mid d \mid e \mid f \mid g \mid r \\
\langle acc \rangle ::= & \ \# \mid \_ \\
\langle frac \rangle ::= & \ 1 \mid 2 \mid 4 \mid 8 \mid 16 \\
\langle oct \rangle ::= & \ 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \\
\langle waveform \rangle ::= & \text{ noise} \mid \text{vPulse} \mid \text{sawtooth} \mid \text{triangle} \\
\langle ident \rangle ::= & \langle string \rangle \\
\langle string \rangle ::= & \text{ [a-z A-Z]}^{+} \\
\langle int \rangle ::= & \text{ [0-9]}^{+}
\end{aligned}
$$

The CFG illustrates that our language is *syntactically strict* as it specifies the valid structure and order of tokens. Any deviations from these rules will not be tolerated, as this strictness will be enforced during parsing (this will be expanded on further in section 4.4). As one can observe, the vertical bar '|' mentioned in section 2.1 is not included in the grammar. This is due to it being treated as whitespace as it should not be parsed as it serves no purpose besides readability (see Appendix F.1).

2.2.2 Semantics

Semantics is the study of meaning. In regard to programming languages, we look at the behaviour that occurs behind the syntax [9, p. 3].

Mainstream programming languages are usually defined by operational semantics, which is a formal way of defining the meaning of a programming language by its computational steps. However, in our case, operational semantics make little sense since our language does not perform any actual computation; it only processes predefined sequences of declared notes. The meaning of our language is music, or to be more precise, sound waves. This invites us to consider a different approach to formally describing the semantics of Neptune, namely denotational semantics, where the mathematical functions mapped to our programming language constructs are defined [2, p. 161]. The semantics of our language can be denoted by the following function:

$[\![\cdot]\!] : neptune \to (\mathbb{R} \to \mathbb{R})$

This is a function which maps our language to a function $\mathbb{R} \to \mathbb{R}$ which synthesises a sound wave. In the following, we will provide insight on how sound waves are synthesised based on the syntax of Neptune (we will not delve further into the precise denotations of the semantics, as this is outside the scope of our project). The shape of the synthesised sound wave - its waveform - depends on what is declared in the source language: *triangle*, *sawtooth*,*vPulse*, or *noise*. A full overview of how waveforms are synthesised, as well as graphical representations of each waveform, can be found in Appendix D.4.

The duration of the sound wave for each note is calculated from its fractional duration in the context of both the tempo and time signature. A more thorough explanation is shown in section 4.5. This duration determines the horizontal length of the sound wave. Sequences are merely concatenations of the sound waves synthesised by individual notes, and each voice is a concatenation of the sequence of sound waves it contains. This leaves us with three overlapping sound waves. These can be thought of as separate or combined into one complex sound wave, such as with additive synthesis (see Appendix D.4).

2.3 Language Evaluation Criteria

As previously mentioned, Neptune is a Domain Specific Language intended to facilitate interaction with the SID chip at an intermediate level of abstraction. With this in mind, we will explore the three main language evaluation criteria: *readability*, *writability*, and *reliability* [2, p. 31]. These three metrics provide us with a more specific terminology to evaluate our design decisions.

2.3.1 Readability

Readability refers to the ease with which users can comprehend and build upon a program written in a specific language. This is especially important when it comes to maintenance and extension of both programs written in the language as well as the language itself.

As Neptune is intended to be an intermediate language, and future developers demand an intuitive understanding, readability is crucial. We aim for a high degree of simplicity, which relates to these three points:

- **Overall simplicity:** The syntax will have a limited scope.
- **Sparse feature multiplicity:** Little to no variation in the way different features can be implemented.
- **No operator overloading:** The purpose of each operator is clear and distinct.

This is all possible due to the declarative nature of our language, as we will not have any meaningful control flow. We aim to support the functionality of the SID chip, not to add any clever functionality or shortcuts. The cost of this high degree of simplicity is a verbose language, which is tedious to write programs for. That is not an issue in our specific case, as our language is not meant to be used

as is. We are leaving room for any user to extend the language to fit their specific use case, which means that a more generalised and simplistic approach is fully appropriate.

On the other hand, the orthogonality[2] of our language is sparse. Anything resembling a data type has its own specific rules and limitations. Parameters are isolated in the header of the file and used only in calculations within the compilation process. It is also not possible for users to add custom parameters or define any of their usages. Sequences are defined with their own distinct syntax, and only intersect with voices in that their names are used to execute them in the desired order. The effect of this is that the language is very inflexible. This is not an issue in the context of our language. We leave space for extensions, which could branch into many directions and introduce different degrees of flexibility.

In alignment with the previous points, we have a list of keywords.

Keyword	**Category**	**Syntax Example**
`tempo`	Parameter	`tempo = 300`
`timeSignature`	Parameter	`timeSignature = (4,4)`
`standardPitch`	Parameter	`standardPitch = 440`
`sequence`	Sequence	`sequence seq1 = {c:4:5 d:4:5 e:2:5}`
`voice1`	Voice	`voice1 = [(seq1, vPulse), (seq1, sawtooth)]`
`voice2`	Voice	`voice2 = [(seq2, vPulse), (seq2, sawtooth)]`
`voice3`	Voice	`voice3 = [(seq3, vPulse), (seq3, sawtooth)]`
`vPulse`	Waveform	`voice1 = [(seq1, vPulse)]`
`sawtooth`	Waveform	`voice1 =[(seq1, sawtooth)]`
`triangle`	Waveform	`voice2 = [(seq1, triangle)]`
`noise`	Waveform	`voice3 = [(seq1, noise)]`
`a,b,c,d,e,f,g`	Note	`sequence seq1 = {c:4:5 d:4:5 e:2:5}`
`r`	Rest	`sequence seq1 = {c:4:5 r:4 e:2:5}`

Table 2.1: Specified keywords in our source language

Each keyword has a distinct purpose in that the meaning behind the syntax remains the same regardless of the context it is placed in. The context in which different syntactic structures are allowed is, in fact, highly restricted. For instance, a note cannot be directly declared in a voice, and sequence names cannot take the form of any keyword.

2.3.2 Writability

Writability, in contrast to readability, concerns the ease with which users can write programs in a specific language. Though these two concepts are strongly related, there is a difference in the matter of perspective. As the previous section hints at, writability is not prioritised as highly in our endeavour.

The expressivity of our language is very limited. Writing more complex pieces of music would require many lines of dense code, since there are no shortcuts for specific purposes. This is by design, as the idea is not that music will be written directly at this intermediate level.

[2] A language is orthogonal if a relatively small set of its primitives can be combined to build control and data structures. Crucially, every combination of primitives must be *legal* and *meaningful*, implying a consistent set of rules for how data types can be used [2, p. 33]

Extending our language does require an understanding of its syntax, but only for the purpose of generating it from a more efficient and expressive language. However, no matter how the language is extended, it is eventually forced to comply with the strictness of our language. This increases the likelihood that input accepted by our compiler will be compatible with the C64 and produce some sound output.

2.3.3 Reliability

Reliability refers to a language being consistently functional in all relevant use cases. Reliability is also closely tied to readability and writability, as a language which is easier to read and write is usually also more reliable [2, p. 39].

There are some additional areas in which reliability can be measured and evaluated:

- **Type checking:** Since our language has no types in the traditional sense, static errors will be related to lexing and parsing rather than typing.
- **Error handling:** Since program execution is non-alterable during runtime, we will not focus on handling dynamic errors.
- **Aliasing:** We will have no aliasing, as sequences cannot point to other sequences, they must be defined as separate entities. Only one name is mapped to each sequence.

Chapter 3

Target Language

Before we determine how to transition from the source to the target language, we must first have a clear understanding of what we aim to achieve. Our objective is to create a program that generates sound on the Commodore 64, which necessitates a thorough understanding of how to communicate with the Commodore 64's sound chip, the SID 6581. We must therefore gain proper insight into the functionality of the hardware of the sound chip and its sound-generating capabilities. In this chapter, we will expand on the most necessary functionalities of the SID 6581 (greater detail of the hardware can be found in Appendix C). Furthermore, we should comprehend how to communicate with the SID chip, as our chosen target language will shape our interaction with it.

3.1 SID 6581

The SID 6581 is a single-chip music generator that consists of three independent voices. These voices can be executed individually or simultaneously to produce compound sounds. The sound chip provides a wide frequency range, high-resolution control of pitch, tone colour, and dynamics [10].

The block diagram in figure 3.1 shows the internal structure of the SID chip. Each voice has a tone oscillator, an envelope generator, and an amplitude modulator. The tone oscillator is responsible for generating a sound wave with the pitch and waveform specified within its registers. The amplitude modulator controls the volume dynamics of each note, based on four values within the envelope generator: Attack, Decay, Sustain, and Release (ADSR). The filter and volume are controlled globally across all three voices [10].

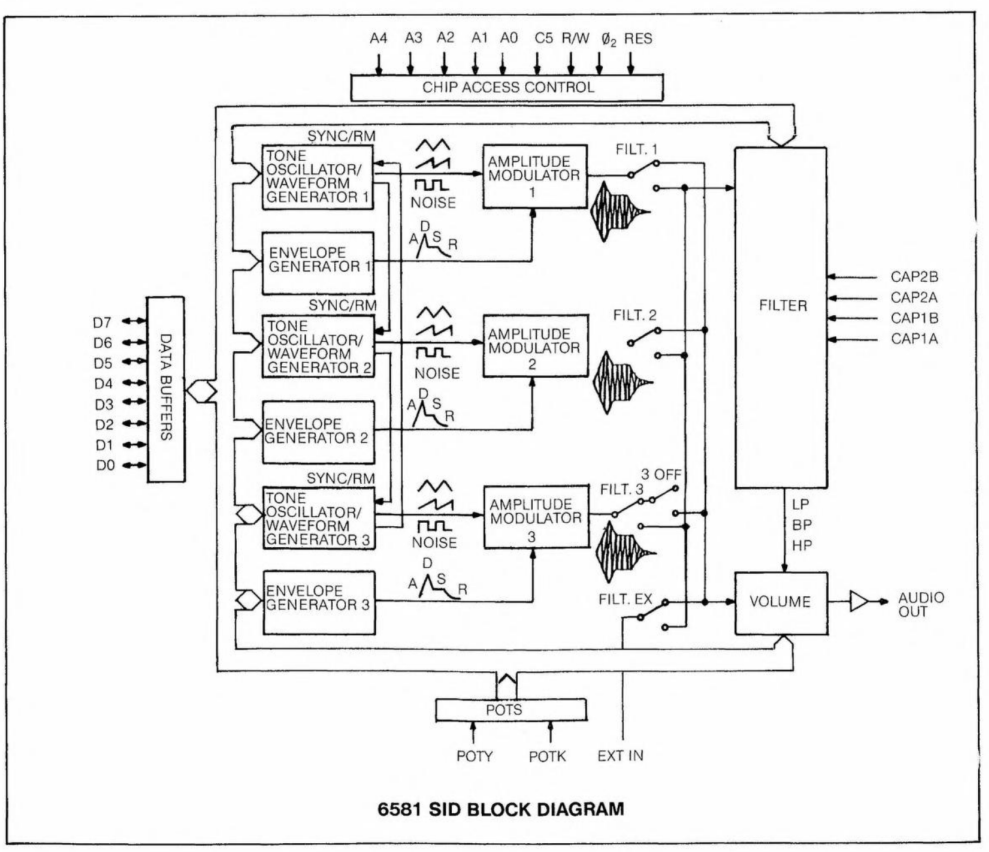


Figure 3.1: Block diagram of the SID 6581 sound chip [10]

3.1.1 Voices

Based on our understanding of the workings of the SID 6581, we define a voice as follows: *A voice transmits a sound signal which generates and manipulates a single sound wave.*

In the SID 6581, the individual voices operate independently of one another, yet can create a polyphonic[1] sound if executed in conjunction with one another. The sound output of a voice is produced with different waveforms: triangle, sawtooth, variable pulse or noise. A feature of the SID 6581 is that it is possible to change the waveform of a voice dynamically [10].

3.1.2 Oscillators

An oscillator is a circuit that generates continuous alternating waveforms. A waveform is a simple representation of a sound wave, typically visualised with a graph depicting its displacement over time. Figure 3.2 shows a visual representation of each of the SID chip's four waveforms. An explanation of waveforms, frequency, amplitude, and harmonics can be found in Appendix D.

[1]The simultaneous combination of two or more tones or melodic lines [11]

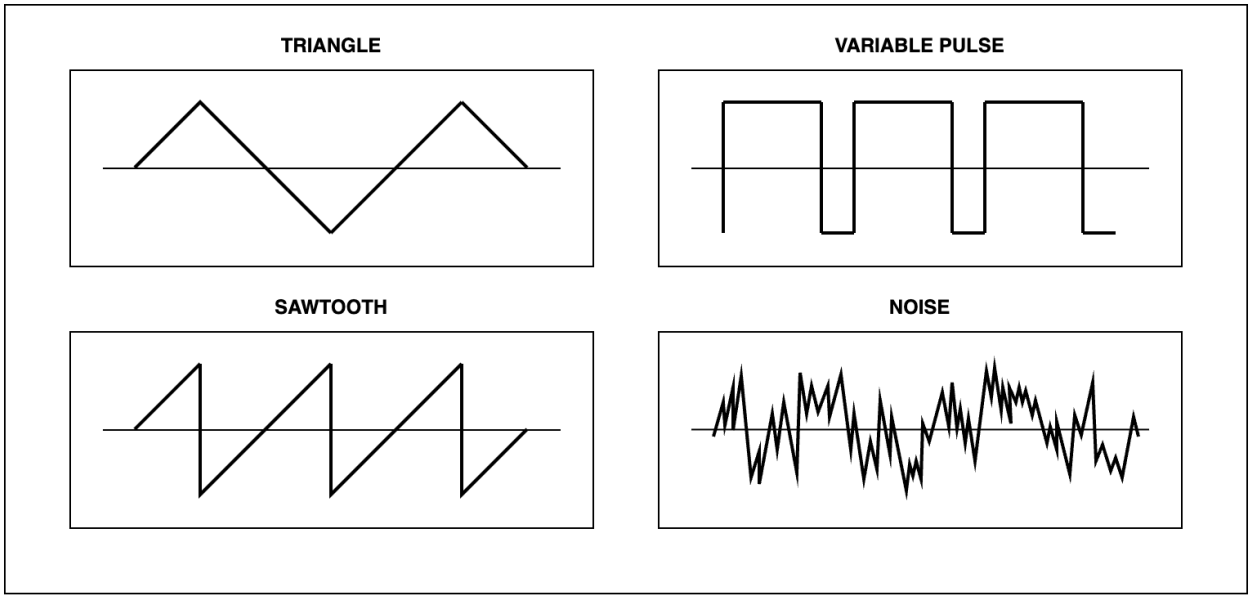


Figure 3.2: The four waveforms of the SID chip: triangle, sawtooth, variable pulse, and noise [3]

- The **triangle** waveform has only odd harmonics and has a mellow and flute-like sound.
- The **sawtooth** waveform contains both even and odd harmonics, giving it a bright and brassy sound.
- The **variable pulse** waveform has a varied sound, fluctuating from a bright and hollow square wave to a nasal and reedy [2] pulse wave. Adjusting the pulse width registers changes the harmonic contents of the pulse waveform, affecting how its tone qualities are produced.
- The **noise** waveform is the last waveform of the SID 6581. Its sound quality ranges from a low rumbling to hissing white noise [10].

When it comes to controlling a sound wave's frequency, each oscillator has two 8-bit registers for this purpose, a low byte and a high byte. These 16 bits combined will result in the frequency of the generated sound wave [3]. As an example, to set the frequency of Voice 1, you could set the low frequency to the hexadecimal value $FB and the high frequency to $04. The combined frequency value would be $04FB (1275 in decimal). This value would then be used by the SID to calculate the frequency at which the oscillator's waveform should be generated, which corresponds to the tone D#.

3.1.3 Envelopes

An instrument playing a certain note creates a sound whereof volume has a certain shape which changes over time. When pressing and holding a piano key, it instantly produces a high volume note and drops to silence over time. These ramp up, note sustain, note decay, and release times can be defined as envelopes. Multiple types of envelopes exist, such as AHD[3], HADSR[4], and many more.

[2] A thin and high sound
[3] Attack, Hold, Decay
[4] Hold, Attack, Decay, Sustain, Release

Between these, the SID chip is only able to use the ADSR[5] type [12].

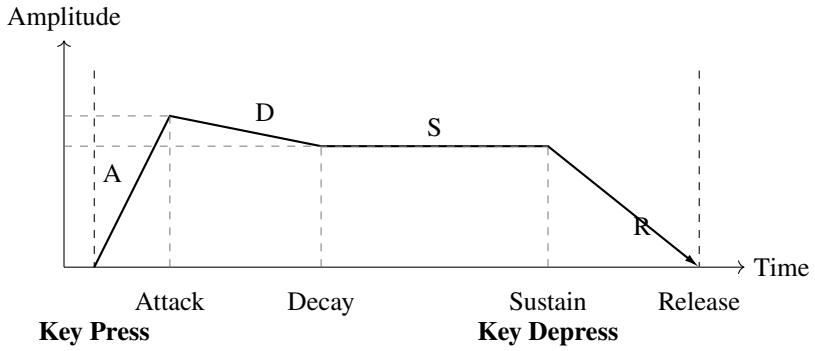


Figure 3.3: ADSR Example

The ADSR Envelope within the SID 6581 is digital and is controlled using 4-bit parameters, which in turn offer 16 steps for each phase.

- **Attack**
  The attack phase regulates the time it takes for the sound to rise in amplitude from complete silence. A higher value equals slower rise times [10].
- **Decay**
  The decay phase defines the time it takes for a sound to fall in amplitude, from the current waveform's volume to the sustain level. A higher value equals slower decays [10].
- **Sustain**
  The sustain level is the amplitude at which the sound is held before being released. The amplitude at which it is being held is controlled with the value, which specifies a percentage of the peak volume, so the sustain parameter sets the target amplitude that the decay parameter seeks [10].
- **Release**
  The release parameter controls the time it takes for the sound to fade out to silence after the note is released. A higher value results in a longer release time [10].

The envelope generators can be modified dynamically and have separate values for each different voice.

3.1.4 Filter

The filter serves the purpose of generating elaborate and dynamic tone colours. This is done by changing the harmonic structure of a waveform (see Appendix D.3), thereby producing a customised sound output by removing specific frequencies from the final sounds [3].

It is a multimode resonant filter, as it consists of three different modes: a low-pass filter (LP), a high-pass filter (HP), and a bandpass filter (BP) [13]. The LP filter allows lower frequencies to pass, rejecting those above the specified cut-off. Rejection is done by attenuation, meaning the strength

[5]Attack, Decay, Sustain, Release

of the sound will be gradually reduced as the voice is routed through the filter. Adversely, the HP filter will pass the frequencies above the cut-off and attenuate the ones below. The BP filter will pass frequencies within a specified range of the cut-off and attenuate the frequencies outside of the range [13]. As visualised in figure 3.1, a voice goes through its amplitude modulator. From there, it will either be routed through the filter or bypassed. For the filter to have an effect on the audio output, one of the three filter modes must be selected.

3.1.5 SID Control Registers

The different functionalities in the SID chip can be controlled through a range of addresses within the C64's memory, spanning from $D400 to $D418. The full mapping of these registers can be seen in Appendix C.2. To sum up, each voice has its own set of the seven following registers:

VOICE REGISTERS

Register Index	Memory Addresses	Name	Description
0	$D400 / $D407 / $D40E	Frequency (Low)	Defines the low byte of a note's frequency
1	$D401 / $D408 / $D40F	Frequency (High)	Defines the high byte of a note's frequency
2	$D402 / $D409 / $D410	Pulse Width (Low)	Defines the low byte of the width of a variable pulse waveform
3	$D403 / $D40A / $D411	Pulse Width (High)	Defines the high byte of the width of a variable pulse waveform
4	$D404 / $D40B / $D412	Control Register	Each bit controls a different feature, such as the waveform or gate bit
5	$D405 / $D40C / $D413	Attack/Decay	Defines the Attack and Decay, with four bits assigned to each
6	$D406 / $D40D / $D414	Sustain/Release	Defines the Sustain and Release, with four bits assigned to each

Figure 3.4: An overview of the registers dedicated to each voice in the SID chip, with descriptions of each register. The full mapping of registers can be seen in Appendix C.2

In addition, there are four filter registers which control features such as the filter cutoff, selected filters, and global volume.

3.2 Choice of Target Language

Interacting with the SID chip and programming its behaviour can be done at three levels of abstraction.

At the lowest level, we have machine code. This is simply a series of bytes encoding instructions, values, and memory addresses. It has to be in the specific format that the Commodore 64's microprocessor expects and is able to interpret. Writing programs in machine code is the most efficient option, at the cost of both readability and writability.

The second level of abstraction is 6502 Assembly, which is simply machine code styled to be human readable [14]. The 6502 Assembly language introduces mnemonic representations of instruction codes in the form of three-letter abbreviations. Traditionally, this was a method to outline the program structure, which could then be converted into machine code by hand. This is known as *hand assembly*. The process has since been automated by introducing a piece of software called an *assembler*. There are many varieties of assemblers with slight syntax variations and shortcuts, such as labels to represent addresses in memory.

The third and final level of abstraction is BASIC, which is the high-level language built into the initial prompt of the C64. This language was designed to be widely accessible [15]. The Commodore 64 shipped with version 2.0 of BASIC, which was outdated even at the time. It is limited in many aspects, and especially rushed when it comes to sound and graphics [13]. Instructions for creating music in BASIC strongly resemble their lower-level counterparts, as users are still required to read and write to addresses in memory.

A comparison of these three abstraction levels can be seen in figure 3.5. It shows how the master volume of the SID chip is set to its maximum value.

BASIC	**6502 Assembly (mnemonic format)**	**6502 Machine code (raw bytes)**
POKE 54296, 15	LDA #$0F STA $D418	A9 0F 8D 18 D4

Figure 3.5: Setting the SID chip to maximum volume in three levels of abstraction, from BASIC down to machine code

In BASIC, the keyword POKE is used to write to the memory address 54296, setting its value to 15. In 6502 Assembly, this instruction is split into two steps, as we cannot move data directly between memory addresses at this level and below [14, p. 11]. First, `LDA` (LoaD Accumulator) loads the hexadecimal equivalent of 15 into the accumulator[6]. Only then can we use `STA` (STore Accumulator) to write the value to the memory address D418, which is the hexadecimal equivalent of 54296. In machine code, instructions are in the form of hexadecimal values. Here, `LDA` is `A9`, and `STA` is `8D`. The order of bytes in the memory address is also in reverse order, since the C64's microprocessor is *little endian* [7] [17].

[6] A register which holds a hexadecimal value. This value can be copied and modified without affecting any memory. It is used as an effective temporary storage of values, and can be used to perform arithmetic operations [3]

[7] Little endian means that bytes are processed in reverse order, least significant byte first [16, p. 78]

We have chosen 6502 Assembly as the target language of our compiler. It is low-level and thus very efficient, but remains at a decent level of readability. This helps us to more easily evaluate the output of the compiler, with fewer conversions and mnemonic lookups. The marginal increase in readability of BASIC in this context is far outweighed by its slow processing speed.

As mentioned earlier, assembly language introduces the need for an assembler. Countless assemblers can be found online, and there are not many resources comparing the benefits and drawbacks of them all. The general consensus in online forums is that the choice of assembler should be based on one's preference for certain syntax styles [18].

For this project, we have chosen the assembler *DASM*. It is open source, cross-platform, and supports the 6502 architecture. This means that the assembler works on Windows, Mac, and Linux, and the entirety of the source code is freely available on GitHub [19]. In addition, we find the documentation to be comprehensible. The syntax has a lot of flexibility, such as not being case-sensitive. All of this means we can spend less time on the assembler, allowing us to focus on compilation between our source language and target language.

3.3 Outlining the Assembly File

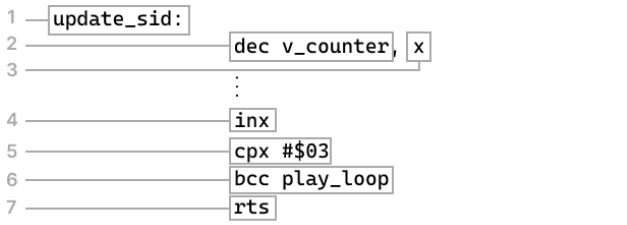
Before our compiler can be implemented, it is vital to understand the structure of the assembly file it should compile to.

3.3.1 Assembly Terminology

An assembly file is a set of instructions and data declarations. Instructions direct the CPU to perform specific operations, and data declarations define data that is inserted at specified memory addresses. Every line of assembly code spans one or more addresses in memory, which are used in jump and branch instructions[8]. Instead of memorising these addresses, we can define labels and use them to refer to specific addresses in the code.

Instructions can be grouped as *routines*, which are comparable to functions of higher-level languages. A routine has a label which denotes its starting address, and additional labels can be used to start the routine from a later point. Routines can call other routines, subroutines, by using jump or branch instructions with their respective labels. Figure 3.6 shows an example of how the different components of a routine look in the code.

[8] A jump instruction jumps to an address unconditionally, while a branch instruction jumps to an address only if a certain condition is met



1 Label of the routine

2 Instruction with an argument indexed by the X register value

3 Accessing the value in the X register

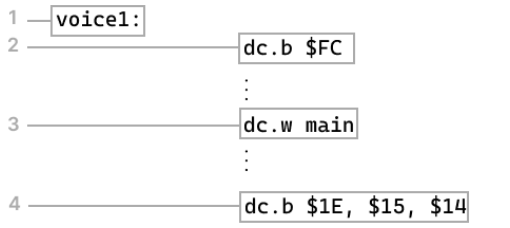
4 Instruction with no argument which increments value in the X register

5 Instruction which compares X to an absolute value

7 Instruction to return to where the subroutine was called

Figure 3.6: The components of a routine

Data declarations in our target code use directives from DASM to write bytes into memory. We mainly use `dc.b` (Define Constant Byte), which takes a list of comma-separated bytes and writes them to memory sequentially. We also use `dc.w` (Define Constant Word), which instead takes a list of comma-separated words. A word in this specific context is always a 16-bit value, which is written into memory as two separate bytes. Data declarations, like routines, can be grouped and labelled for easy access. An example is shown in figure 3.7



1 Label of the data declarations

2 Definition of single byte written to memory

3 Definition of address written to memory as two bytes

4 Definition of three bytes written to memory

Figure 3.7: The components of data declarations

While data declarations are useful for storing bigger chunks of data, there are also three available registers X, Y, and A. They are quick to access, as they are close to the CPU, and are used for short-term, dynamically changing variables. The A register, also known as the accumulator, has the additional role of performing arithmetic operations and moving data.

3.3.2 Preliminaries

The assembly file should, as required by DASM, start with a header defining the processor as well as the origin of our program, which is the exact memory address a BASIC program is defined in. This is due to the fact that running assembly code in the C64 prompt requires it to be called as a subroutine using the keyword `SYS`. This is done in our code by constructing a BASIC program as a series of data declarations. The construction of the BASIC program can be seen in Appendix C.4.

3.3.3 Initialisation

The next step is to define initial values for the SID chip. First, the volume is set to the maximum value to ensure sufficient sound output, and the filter register is cleared as it is not used in our current scope. Then, addresses at the beginning of each voice's byte data are assigned to the equivalent pointer variables (see sections 3.3.5 and 3.3.6).

3.3.4 Setting the Raster Routine

Since we are dealing with music, and as such wish to have control of the timing of notes changing over time, we are defining our own custom raster routine. The raster routine refers to the refreshing of pixels on the screen, something which happens once every 20 milliseconds, or frames. These refreshes, or raster IRQs (Interrupt ReQuests), can be used as a framework for the timing of notes. By first disabling interrupts completely, the IRQ vector can be updated to hold a routine of our choosing, followed by re-enabling interrupts. This routine is simply named `raster` (see section 3.3.7), and will now be called on every raster IRQ until the Commodore 64 is reset, even after the main program finishes its execution.

3.3.5 Music Data

Before we delve into the music routine, which will handle playing music, we will first describe how the music itself is written, the *music data*. Music data defines the structure of voices and sequences as blocks of byte data. The hexadecimal value of each byte determines its meaning (see figure 3.8).

Byte	Interpreted as
$00-$F8	High frequency, low frequency or duration of note
$F9	Noise waveform
$FA	Variable pulse waveform
$FB	Sawtooth waveform
$FC	Triangle waveform
$FD	Jump to address
$FE	Enter sequence
$FF	Exit sequence

Figure 3.8: Interpretation of each byte within music data

Due to the syntax of our source language, there is a strict pattern that music data will follow depending on whether the data defines a voice or a sequence.

Sequences contain one or more rows of three bytes, each of which represents a note. The notes will be processed in the order of low frequency, high frequency, and duration, and are expected to strictly follow this order. All sequences end with the instruction to exit the sequence. As an example, if we have the following syntax in our source code:

```
sequence seq1 = { c:1:4 }
```

It should be compiled into the following assembly code, assuming a tempo of 120 BPM:

```
1  seq1:          dc.b $C3, $10, $60
2                 dc.b $FF
```

Voices are divided into three blocks of byte data, labelled as *voice1*, *voice2*, and *voice3*. For each sequence defined in a voice, there will first be a byte representation of the sound wave, then an instruction to enter a sequence, followed by the label of that sequence. All voices end with a jump to the beginning of the voice. As an example, if we define `voice1` with the sequence from above in a variable pulse waveform, we have the following source code:

```
voice1 = [(seq1, vPulse)]
```

This should compile into the following assembly code:

```
1  voice1:        dc.b $FA
2                 dc.b $FE
3                 dc.w seq1
4                 dc.b $FD
5                 dc.w voice1
```

3.3.6 Variables

To track which notes are currently being played, we need a set of long-term variables that can easily be navigated and updated. These contain information about the current state of each voice, as well as information related to each waveform. Just as with music data, there are some labelled rows of byte data that can be accessed during program execution. A full overview can be found in Appendix C.4.

Voice variables contain three columns of byte data, one column for each voice. Each row is labelled as the variable it contains. For instance, the variable `v_counter` holds the remaining duration of a note being played in each voice. Accessing the value for a specific voice is possible by using a voice index, which we have dedicated the X register to. The indices 0, 1, and 2 are added to the address of a variable to access the desired value. As an example, if the index in register X is currently 1, it means that `voice2` is currently being processed. To get the remaining duration of the note played in this

voice, we use the instruction `lda v_counter,x`. We use absolute indexed addressing mode[9] to load the address of `v_counter` plus 1, thus loading the value of the second column into the accumulator. The same method is used when writing new values into variables.

Instrument variables work similarly to voice variables, where each row is labelled with a different variable name, but here, each of the four columns is tied to different waveforms. To navigate the four instruments, we use an instrument index stored in the Y register. Each instrument stores the hexadecimal representation of the waveform, ADSR, as well as information about the variable pulse.

3.3.7 Music Routine

We are now left with only our music routine itself, which is a set of interconnected subroutines that update the values in the SID chip every frame. As the flowchart in figure 3.9 shows, there are four main components to the music routine. Firstly, we have our raster routine, which is what we have now stored in the IRQ vector. Then, we have the main loop, which is what iterates through each voice and updates the values according to our music data and variables. Within this main loop, there are two possible branch conditions: One for fetching notes and one for initialising notes. Notes must be fetched two frames in advance, which means that when they are meant to be played, the information will be readily available. Fetching in advance also allows us to turn off the gate bit of the previous tone, starting the release cycle and allowing for a smooth transition between notes. We will now provide more details for each of these four components of the music routine.

Raster routine uses a series of `NOP` (No OPeration) instructions to alter the timing of notes very slightly. This gives a more precise sound output. We then enter the main loop of our music routine, play one frame of music, and finally acknowledge the raster interrupt. This is an important step, as it ensures the music routine is not called infinitely, but only once every frame.

Main loop processes all three voices in succession. We reserve the X variable as an index. By using this index, when voice data or voice variables are accessed, only information relevant to the voice currently being processed is available. For each voice, we first check if any of the two branch conditions are met. When two frames of a note's duration remain, it jumps to the fetch routine. When the duration reaches 0, it jumps to the initialisation routine. Regardless of whether any branches were taken or not, we return to the last part of the main loop, which updates the SID chip values: frequency, waveform, and ADSR. We also decrement the counter, which tracks the duration of the current note. After all three voices are processed, we return to the raster routine.

Initialise note happens at the frame where a new note should start playing. The high frequency of the fetched note is used to check whether or not it is a rest note, since there are no tones with a high or low frequency of 0. Therefore, when the high frequency is 0, we simply turn off the sound output by setting the gate bit to 0. In the case of a note with sound output, we instead store the low and high frequencies of the note, as well as the waveform and ADSR for the fetched instrument. In both cases, we end the initialisation routine by storing the fetched note duration in the corresponding voice variable.

[9] This mode of addressing uses a base address and an index value to get the effective address [14, p. 77]

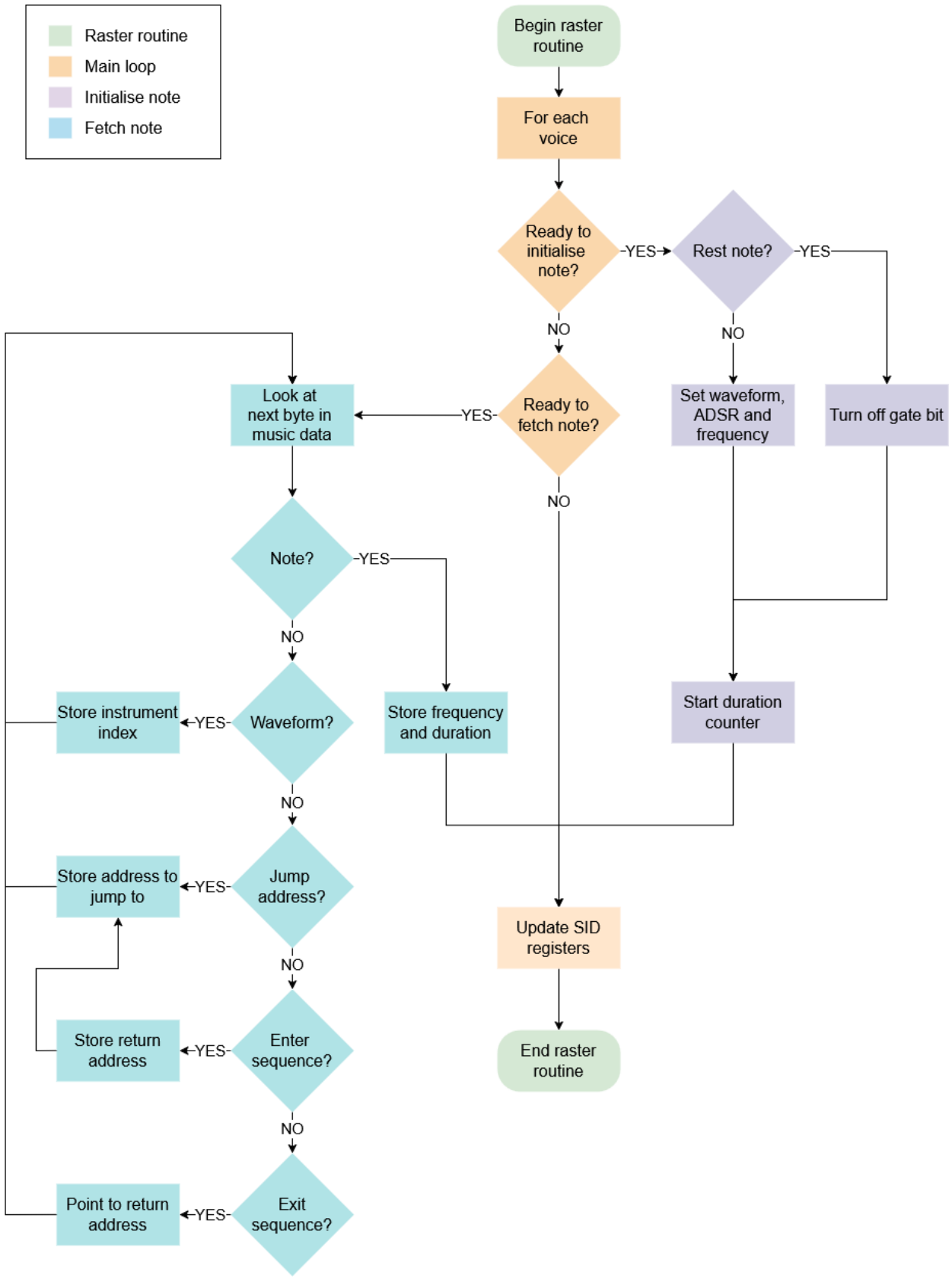


Figure 3.9: This flowchart depicts the overall structure of our music routine. The raster routine is called on each frame, iterates through all three voices in the main loop, and initialises or fetches notes when needed

Fetch note happens when a new note is two frames away from being initialised. This step goes through music data, byte by byte, only returning to the main loop once a new note has been found. As explained in section 3.3.5, the value of each byte determines its meaning. If the value of the current byte is $F9 or below, we know that we have found a note. This byte and the following two are expected to be the frequency low byte, frequency high byte, and duration in frames. This information is stored in separate variables, so for instance, the low frequency is not stored in `v_freqlo`, but `v_freqlo_new`. This way, they are readily available at the time of initialisation, not written to the SID chip immediately.

In case the current byte is not a note, we must determine what other purpose this byte serves. Values between $F9 and $FC represent a change of waveform.

The remaining bytes are reserved for the `jump_addr`, `enter_seq`, and `exit_seq` routines, respectively. They serve as indicators to jump to the correct routine. In `jump_addr`, we expect the following two bytes to represent an address in the music data we should jump to, which we will then store in the pointer variables. In `enter_seq`, we save the return address we go to when the sequence is exited, and then jump to the address of the sequence using the `jump_addr` routine. In `exit_seq`, we simply set the pointer to the previously obtained return address.

In any of the cases where we do not find a note, we process the byte as mentioned above, and then repeat the entire fetch routine with the next byte. The fetch routine will not terminate until a note is fetched.

Chapter 4

Compiler Design

This chapter will expand on the implementation of our compiler. The entire compilation process is explored, from the lexical analysis of the source language to the generation of assembly code, and finally, the execution of the resulting output file. Afterwards, we will present how we have conducted the testing of our compiler utilising three different methods of testing: *unit, integration,* and *acceptance testing*. The chapter will cover the design decisions and the rationale behind our implementation, supported by examples and code snippets.

4.1 Specifications

We have used the *MoSCoW* model to determine the requirements of our compiler, which has been documented in Appendix E. The most significant feature of our compiler is that it must be able to translate our source language Neptune into assembly code (see section 3.2). Additionally, the assembly code output by the compiler must be able to interact with the SID chip's control registers, which are in charge of determining the waveform in each voice. Furthermore, the compiler should integrate an external assembler for converting the assembly code into machine code.

4.2 Compiler Architecture

An overview of the process of compiling our source language can be seen in figure 4.1. Our compiler consists of various phases, which can be divided into two main parts, the frontend and backend.

The compiler's frontend incorporates the lexer, parser, and translator. The first phase in the compilation process is where the lexer performs a lexical analysis, a process of deconstructing our source language into tokens. In the following phase, the parser generates an *intermediate representation* (IR), a representation of the tokens of the source language, in the form of an *abstract syntax tree* (AST). The generated AST, which we refer to as the source AST, is a high-level IR as it is close to our source language, and it is the first of two ASTs in our compiler. The next phase, the translation, is when the initial source AST is translated into another AST, referred to as the target AST, as it is a low-level IR that is a bit closer to our low-level target language, 6502 Assembly. Through these last two phases, a symbol table is used to store the sequences.

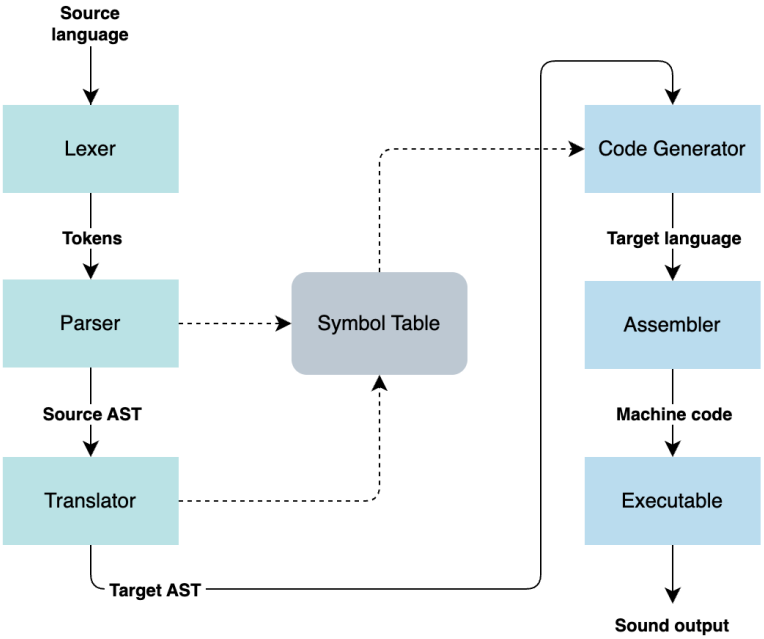


Figure 4.1: The diagram is a visualisation of our compiler architecture. It shows the different phases of the compiler, split into frontend and backend, as well as a symbol table used for storing sequence information

The backend's role is to convert the target AST into low-level machine code to be executed on the Commodore 64. The first phase of the backend is the generation of an output file, written in 6502 Assembly, based on the target AST and symbol table, containing music data for the sequences and voices. In the following phase, an external assembler is used to assemble our target language into an executable file, written in machine code, such that it can be executed on the Commodore 64. The assembler is integrated into the compiler, allowing end-users to compile our source language directly into an executable file with a single command line. The machine code can then be executed on an external C64 emulator, producing a sound output authentic to the hardware of the SID 6581.

We have created our compiler using OCaml, a language that was covered in our educational course material, which offers great tools for implementing lexers and parsers. Moreover, OCaml's pattern matching facilitates clear and concise case analyses, making vertical extensions[1] of the compiler straightforward, in comparison to object-oriented languages. In OCaml, vertical extensions can be developed in a single function, whereas in object-oriented languages, changes will often require updating multiple classes [21] [20].

A typical compiler design includes a type-checking and an optimisation phase in the backend. However, we have opted out of including these phases in our compiler, due to the declarative nature of our source language, meaning that there are no functions or computations to optimise, or explicit data types that require checking.

[1] Vertical extensions refer to the addition or updating of features within a system [20]

4.3 Lexer

The lexer is the first phase of the compiler; this is where the lexical analysis happens, which is the first step of the syntactic analysis. In this phase, the lexer takes the source language as input and produces a stream of tokens as output, which will then be handled by the parser in the next step of the syntactic analysis.

To create the lexer, we have implemented OCaml's standard tool for lexical analysis, *OCamllex*. This lexer follows a declarative lexing approach by utilising regular expressions to define legal tokens [22] [2]. During compilation, the source language appears to the compiler as a single string of characters. Therefore, the lexical analyser is responsible for categorising the characters into different types of tokens based on some predefined rules [23]. It performs the lexical analysis by matching the input character strings to the regular expression (regex) patterns and performs the semantic action paired with the regex in the lexing rules (see Appendix F.1).

- *Regular expressions* are formal notations that describe a variety of tokens required by programming languages. Each regex is essentially a sequence of characters that defines a search pattern [23].
- *Semantic actions* inform the lexer of what to do when it matches a regex pattern. Semantic actions typically instruct the lexer to return a token and sometimes perform some other action as well.

We have designed our lexer to recognise the following token types: *keywords, identifiers, tones, literals, operators, delimiters, comments,* and *white spaces*. To differentiate between keywords and identifiers, we implemented a hashtable that stores all reserved keywords, allowing for efficient lookup. When the lexer matches a potential identifier, it uses a helper function to search through the hashtable to decide whether it is an identifier or keyword. This function is called within the semantic action associated with the `ident` regex (see Appendix F.2).

We chose to implement a separate lexing state for handling comments to avoid comments mistakenly being interpreted as code. When the lexer encounters a start-of-comment delimiter '/*', it jumps to a separate lexer state that is solely concerned with handling comments. This state ensures that the lexer ignores any input between the comment delimiters '/*' and '*/'. Once the lexer encounters the end-of-comment delimiter, it will return to the general lexing rules.

At the end of the lexing rules, we have a catch-all rule for handling inputs that do not match any of the specified regex patterns. With this follows a semantic action wherein an exception is raised. The raised exception calls a helper function that uses the OCamllex module *Lexing* to retrieve the location in the lexbuf where the error occurred. This function helps provide precise error reporting by identifying the position in the input that caused the lexical failure.

4.4 Parser

The parser is the second phase of the compiler and the main phase of the syntactic analysis. It tries to match tokens constructed in the lexical analysis with the production rules from the context-free grammar (see section 2.2.1), and if it matches, the AST (see figure 4.2) is generated. Additionally, the parser detects syntactical errors, signalling the position of the error's occurrence in an error message [24]. This is handled in a similar way to the lexer (see section 4.3). It is important to note that the parser of our language terminates execution of the compilation when an error occurs, and does not show any following errors.

In this project, we have chosen to use *Menhir* as our parser generator instead of writing the parser ourselves, similar to how OCamllex generates our lexer. Menhir takes the defined production rules and generates the compiler's actual parser based on these rules. Menhir parses bottom-up, using the LR(1) parsing method, and is considered as the most powerful parser of all deterministic parsers in practice [25]. It scans the input from left to right and searches for the right-hand side of the production rules to build the derivation tree.

The parser receives the tokens generated by the lexer and builds an AST according to Neptune's grammar. An example of this AST building process is the `params` rule (see listing 4.1), which defines how the musical parameters are declared and set. The example shows that when the input file is parsed, a record corresponding to the `params` type is constructed with the values of tempo, time signature, and standard pitch, which is then added to the AST. Moreover, it shows that the parser expects tokens in a specific order: `TEMPO`, `TIMESIG`, and `STDPITCH`.

```
1    params:
2  | TEMPO ASSIGN t = INT
3    TIMESIG ASSIGN SP npm = INT COMMA bnv = INT EP
4    STDPITCH ASSIGN sp = INT
5    { ...
6      {tempo = Some t; timesig = Some (npm,bnv); stdpitch = Some sp}
7    }
```

Listing 4.1: The rule of params

The source code in listing 4.2 is parsed according to the `params` rule. During parsing, it will check whether the grammar rules are followed. If so, the semantic action will be executed, constructing the AST nodes for the tempo, time signature, and standard pitch.

```
1 tempo = 120
2 timeSignature = (4,4)
3 standardPitch = 440
```

Listing 4.2: Source code example for Neptune taken from H.1

In addition to constructing the AST, the parser stores the sequences in a symbol table, and an identifier is later used to track the specific sequence in the backend. The rule of `seqdef` performs two semantic actions as portrayed in listing 4.3. First, a helper function is called, taking the sequence identifier and the sequence body, which is the actual note list, as key-value pairs, where the identifier will be used to access the sequence stored in the symbol table. Then, an AST node of the `seqdef` type is constructed.

```
1     seqdef:
2   | SEQUENCE id = ident ASSIGN LCB sb = seq RCB
3
4     {
5       add_sequence id sb;
6       {name = id; seq = sb}
7     }
```

Listing 4.3: The rule of params

During parsing, we also ensure the prevention of duplicate sequence identifiers. Based on the source code in Appendix H.1, the following AST is generated (see figure 4.2). This AST is a representation of our source language (see Appendix H.1), and holds all information written in the input file.

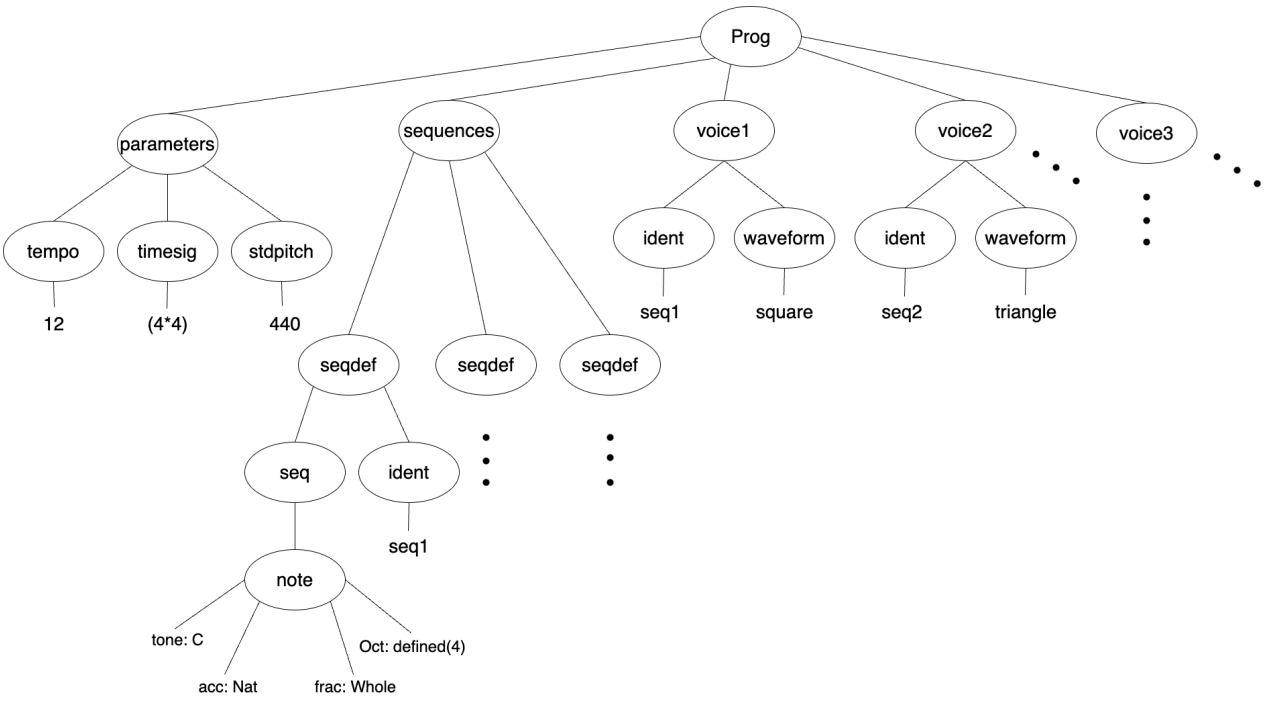


Figure 4.2: A visualisation of the source AST constructed with our parser

4.5 Translation of AST

The translator uses a series of functions to unpack the source AST generated by the parser and reconstruct its fragments into the equivalent target AST. Some of the translation is rudimentary, such as with voices, where the structure is preserved between ASTs and only the type is changed. The more interesting aspect of the translator is the translation of notes. In the source AST, notes consist of a tone, an accidental, a fraction, and an octave. However, this format is not compatible with our music routine (see section 3.3.7), which requires a high and low frequency, and a duration in frames. Calculating these values now simplifies code generation in the next phase of compilation.

To calculate the high and low frequency of a note, we must first find the frequency output in hertz (Hz). In the case of rest notes, both values are simply set to 0. In all other cases, the standard pitch is used as the baseline and then incrementally modified by a number of semitones[2] offsets. These offsets are not linearly spaced out, but instead follow the ratio $r = \sqrt[12]{2} \approx 1.05946$, where r is one semitone offset [26]. In the translator, we use three different helper functions to determine offsets for the tone, accidental, and octave (see listing 4.4).

```
1  let base_offset = function
2    | Ast_src.C -> -9 | Ast_src.D -> -7 | Ast_src.E -> -5 | Ast_src.F ->
       -4
3    | Ast_src.G -> -2 | Ast_src.A -> 0 | Ast_src.B -> 2
4
5  let acc_offset = function
6    | Ast_src.Nat -> 0
7    | Ast_src.Sharp -> 1
8    | Ast_src.Flat -> -1
9
10 let oct_offset = function
11   | Ast_src.Defined o -> (o - 4) * 12
12   | _ -> 0
```

Listing 4.4: Matching the tone, accidental and octave to their respective offsets from the standard pitch

[2] A semitone refers to the distance between each of the twelve tones in Western music theory (see Appendix B.3)

The sum of these offsets gives us the total offset S, which is used to adjust the standard pitch F_{std} to the specific tone. With this information, the output frequency F_{out} can be calculated using the following formula:

$$F_{out} = F_{std} \cdot 2^{\frac{S}{12}}$$

Then, the following formula is used to convert the frequency output F_{out} to a frequency F_n which is compatible with the SID chip [3, p. 187]:

$$F_n = F_{out}/0.06097$$

Finally, to divide this frequency into its high frequency F_{hi} and low frequency F_{lo}, the following calculations are made:

$$F_{hi} = \lfloor F_n/256 \rfloor$$

$$F_{lo} = \lfloor F_n - (256 \cdot F_{hi}) \rfloor$$

The only aspect of the note left to translate is its duration. In the source AST, a note only contains its fractional duration. Converting this to an absolute duration in frames[3] requires us to use both the tempo and time signature. To this end, a duration reference for the entire source file is calculated. This is simply the absolute duration of a sixteenth note. The function can be seen in figure 4.5.

```
1    let get_duration_ref () =
2        ...
3        let bnv_duration = 3000 / tempo in
4        match bnv with
5            | 1 -> bnv_duration / 16
6            | 2 -> bnv_duration / 8
7            | 4 -> bnv_duration / 4
8            | 8 -> bnv_duration / 2
9            | 16 -> bnv_duration
```

Listing 4.5: Calculating the reference point for duration of notes

First, the amount of frames per minute (3000) is divided with the amount of beats per minute (tempo), resulting in the duration of a beat, or basic note[4]. Then, depending on the basic note value, this duration is divided to give the duration of a sixteenth note specifically.

[3] 20 millisecond intervals, as mentioned in section 3.3.7

[4] The basic note value is the second value of the time signature, and denotes the beats in which the tempo is counted (see Appendix B.4)

Now, it is possible to multiply this duration to match it up with any note type, as seen in listing 4.6.

```
1    let get_note_duration f =
2        let duration_ref = get_duration_ref () in
3        match f with
4            | Ast_src.Whole -> duration_ref * 16
5            | Ast_src.Half -> duration_ref * 8
6            | Ast_src.Quarter -> duration_ref * 4
7            | Ast_src.Eighth -> duration_ref * 2
8            | Ast_src.Sixteenth -> duration_ref
```

Listing 4.6: Calculating the note duration in frames

Returning to the example from Appendix H.1, we can use the first note of the first sequence, `c:1:4`, to demonstrate how it would be translated. Since its tone is 'c', the offset here would be -9 semitones. The lack of accidentals and the fact that the octave is 4 means that there would be no additional offsets. Therefore, the resulting output frequency would be $F_{out} = 440 \cdot 2^{\frac{-9}{12}} \approx 261.63$. Converting to the SID-compatible frequency value gives us $F_n = 261.63/0.06097 \approx 4,291.05$. This results in the following high and low frequencies, which match the table in the Programmer's Reference Guide for the Commodore 64 [3, p. 385].

$$F_{hi} = \lfloor 4,291.05/256 \rfloor = 16$$

$$F_{lo} = \lfloor 4,291.05 - (256 \cdot 16) \rfloor = 195$$

As for the duration, we first use the tempo $t = 120$ and basic note value $b = 4$ to find the duration reference d:

$$d = \lfloor \frac{\frac{3000}{t}}{b} \rfloor = \lfloor \frac{\frac{3000}{120}}{4} \rfloor = 6$$

This means that every sixteenth note in this example will last six frames. Then, for the note `c:1:4` with fractional duration $f = 1$, the duration reference is simply multiplied by 4, giving us a duration of 24 frames.

This process continues for all notes in the source AST, as well as the previously mentioned more rudimentary translations, gradually constructing the new target AST. Its structure can be seen in figure 4.3.

The target AST is more compact when compared to the previous iteration. This is in part due to parameters not being included. Since all notes have been translated to contain absolute frequencies and durations, the parameters have served their purpose and are no longer needed. Even more noticeably, sequences are not depicted in this AST. Sequence identifiers are now merely tied to entries in the symbol table, wherein sequences have been updated to contain a list of notes in the updated format.

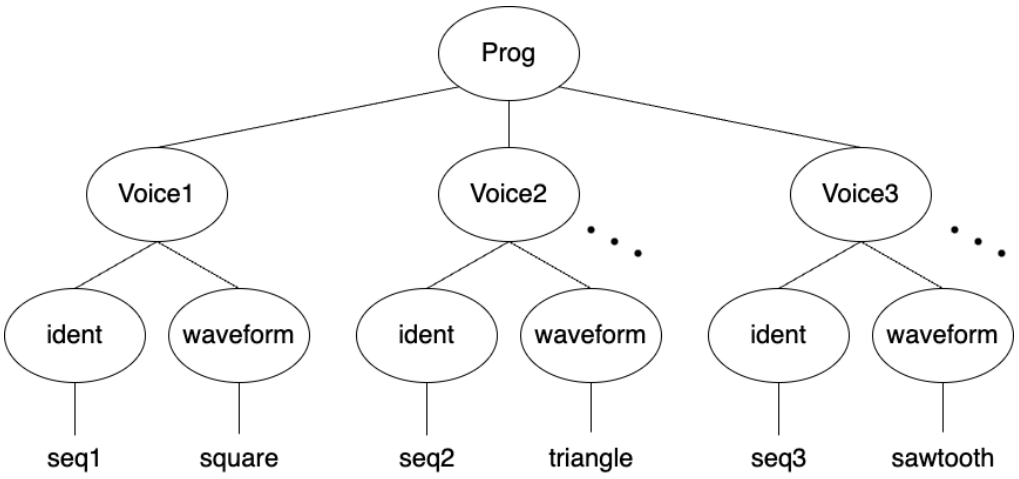


Figure 4.3: A visualisation of the target AST constructed with our translator

4.6 Code Generator

In the code generation phase, the target AST and the sequences stored in the symbol table are used to create the assembly file we compile to. The process starts by deleting the existing output file, if one exists, and then creating a new one. Afterwards, it inserts the music routine (see section 3.3.7) as a preamble[5] into the new output file. Following the preamble, it uses the target AST constructed by the translator and retrieves the symbol table containing sequences to write the assembly code, which defines the music data as defined in section 3.3.5. To ensure that the output file complies with 6502 Assembly, an instruction set is defined, containing the entire 6502 instruction set, as well as assembler directives[6]. When constructing assembly instructions for the output file, this instruction set is used to ensure that an instruction's arguments comply with the specific instruction's maximum and minimum number of arguments. For general file operations, OCaml's standard library is used; this library allows for simple operations such as writing, appending, or deleting lines in a given file [28].

To generate the assembly code for the music data for the voices, the voices of the target AST are iterated through. For each iteration, using the sequence identifier and waveform, assembly instructions are created using the instruction set and written to the output file. Moreover, the function will write the following in the output file: the voice's label, the instruction to repeat the voice, and other instructions mandated by the music routine (see section 3.3.5). As an example, the Neptune code in figure 4.4 is compiled into the assembly code seen in listing 4.7:

```
voice1 = [(seq1, vPulse)]
```

Figure 4.4: Source code example for Neptune taken from H.1

Listing 4.7 shows how the correct voice label and sequence identifier are written to the output file. Moreover, in accordance with the music routines mandated structure (see section 3.3), instructions to

[5] Introductory code, that is included at the beginning of a program

[6] Assembler directives are commands for the assembler that control its operation and settings, and do not get translated into executable machine instructions [27]

define constant bytes or words (respectively `dc.b` and `dc.w`) are written. In the example below, these instructions define the waveform's value $FA, then instruct the routine to enter the sequence $FE, which is entered on the next line. Lastly, the instruction to repeat the voice is created by writing the value $FD and the voice label:

```
1    voice1:          dc.b $FA
2                     dc.b $FE
3                     dc.w seq1
4                     dc.b $FD
5                     dc.w voice1
```

Listing 4.7: An example of some of the assembly code generated when compiling the Neptune code seen in Appendix H.1

Similarly, another function iterates through the sequences stored in the symbol table to write the assembly code for the music data of the sequences. Each sequence contains a note list, which is iterated through, creating an assembly instruction for each note, with its high and low frequency values, as well as the duration. The values are converted to hexadecimal for the music routine to comprehend. Every constructed instruction is added to a list, which is finally iterated through and written to the output file. Finally, the instruction to end the sequence is created. Looking at a sequence definition from the example code in Appendix H.1, we find the following syntax:

```
sequence seq1 = { c:1:4 }
```

Figure 4.5: Source code example for Neptune taken from H.1

The code in figure 4.5 is compiled into the following assembly code (see listing 4.8), with the correct sequence identifier and the hexadecimal values for high and low frequency, respectively $C3 and $10, and duration $60, calculated in the translation phase. The second instruction with value $FF is also written, instructing the music routine to exit the sequence:

```
1    seq1:            dc.b $C3, $10, $60
2                     dc.b $FF
```

Listing 4.8: An example of some of the assembly code generated when compiling the Neptune code seen in Appendix H.1

4.7 Assembler & Executable

After the code generation phase, an assembly output file is produced. This file cannot be directly executed on a Commodore 64 or an emulator, as it must first be converted into machine code. This conversion process is done through an assembler, and as mentioned in section 3.2, we have chosen to use DASM. The DASM assembler features, besides the 6502 Assembly instruction set, its own set of assembler directives, and our compiler utilises the following directives: `dc.b`, `dc.w`, `ORG`, and `PROCESSOR`, where each is not directly an instruction of the Commodore 64's CPU, but an additional function made in the assembler. After DASM has finished processing the assembly file, it generates an executable file, which contains the machine code that plays the music.

The integration of the DASM assembler into the compiler is done through a system command call. This call is only triggered when the compiler is run with the `-dasm` flag. This is done to allow the end-user to choose if they want an executable output file. We have also implemented other optional arguments to the command, like displaying the generated ASTs or a debugging option, allowing users insight into the functionality of the compiler, and to facilitate further development.

The final phase of the compiler is the executable phase, where the compiled machine code can be executed directly on the hardware, or in our case, an emulator [2, p. 49]. We chose to execute our program on a Commodore 64 emulator rather than an actual C64 computer for efficiency. Emulators are user-friendly as they can be kept locally, such that one can simply drag and drop the machine code of the compiled program into the emulator. This provides an optimised development phase as we have been able to test our compiler continuously and efficiently.

We selected *Vice*, a popular free cross-platform emulator, as our emulator of choice. Vice not only emulates the Commodore 64, but also the intricate features of its SID chip. Additionally, we opted for the *reSID* engine, a specific version of Vice that provides a much more accurate emulation of the SID chip. This version also allows us to select the exact SID model we want to emulate, which is the SID 6581. Another advanced feature of the reSID engine is the selection of different sampling methods for the SID signal output, such as fast, interpolation, and resampling. Of these, we have chosen to use the resampling setting for the execution of our program as it provides the best sound quality. Although this setting is rather CPU-intensive, it is perfectly compatible with our program as it is non-interactive [29]. As opposed to DASM, Vice is not integrated into our compiler and must be run separately. Therefore, one has to transfer the machine code file generated from the compiler into the Vice emulator, which then generates the sound output.

4.8 Testing

By conducting tests, we aim to verify the functionality of our product by investigating whether functions or whole components work as expected. It is important to note that our tests will not prove the correctness of the program as it is difficult to test every single case, but by having broad test coverage we can instead provide an indication that the individual functions or components are functioning correctly, which gives us confidence that the overall program works as expected. This section will expand on how we have created unit, integration, and acceptance tests to enhance the quality of our product.

4.8.1 Unit Testing

Unit tests are tests made on smaller components of a larger program to establish that the components work correctly and as expected. Unit testing can uncover small errors in the logic of programs, whereby catching and correcting them can help to make a program more reliable, minimising unexpected errors [30]. For unit testing, we have used *Ounit2*, which is an OCaml testing framework that provides a test environment for grouping tests into suites and other functionalities like assert functions [31]. Our testing methodology has been to test functions after their creation, even though the best practice

for testing is to do the opposite.

The general structure of our unit tests is to mock the data needed for the test, run the specific function, and finally use an assert function to verify that the result is as expected. An example of such a test can be seen in listing 4.9. This test aims to verify that the `note_translate` function in the AST translator correctly translates the source AST note to the target AST note. It creates a mock source AST note and a corresponding mock target AST note, which is the expected result of the function. Finally, the test asserts that the expected note is equal to the output of the translate function when applied to the source AST note.

```
let test_note_translate_sound _ctx =
    let note = AST_SRC.Sound (C, Nat, Whole, Defined 4) in
    let expected_note = {
        AST_TGT.highfreq = 16; AST_TGT.lowfreq = 195; AST_TGT.duration
   = 96
    } in
    assert_equal expected_note (AST_TRSL.note_translate note)
```

Listing 4.9: Unit test for the `note_translate` function

Instead of having to write the same code for mocking data in multiple tests, it is possible to use functions that set the testing environment by mocking any data to be used. This is how most of the symbol table tests have been conducted. For these tests, there is a setup and teardown function that retrieves the symbol table, clears it, and mocks a sequence. A test is run with the symbol table as a parameter, and when the test is done, the symbol table is cleared again to ensure that each test is independent and not influenced by the previous test.

Finally, there are unit tests that aim to assert that the right exceptions are raised. An example of such a test can be seen in listing 4.10. The test uses the `assert_raises` function to assert that an exception with the correct error message is raised when a function is run. In the case of this test, the exception is expected to be raised as the same sequence is added to the symbol table twice.

```
let test_add_sequence2 id seq =
  SYM.add_sequence id seq;
  assert_raises (EXC.SyntaxErrorException "Sequences id's cannot be
   duplicated. Each sequence must have a unique id.")
      (fun () -> SYM.add_sequence id seq)
```

Listing 4.10: Unit test for the `add_sequence` function

4.8.2 Integration & Acceptance Testing

Having verified that components of the compiler behave as expected, their interaction will now be the focus. Integration tests aim to evaluate how components or modules of a larger system communicate and interact.

Our integration testing approach is to test the whole compiler as a single unit, and it involves compiling a series of Neptune programs, some expected to compile and some expected to raise specific exceptions. Inspecting the ASTs, the error messages, and the output file for these programs enables verification that each component in the compilation process produces the expected result. The series of tests has been organised into folders to enable automation of the execution of the tests using GitHub Actions[7], which compiles all files within these folders.

One test defines a single sequence, with `seq1` as its identifier, containing one note. This test aims to verify that the compiler correctly parses the input file and produces a valid output file. Inspecting the source AST verifies that the sequence is properly parsed. Furthermore, inspecting the output file confirms that the sequence has been added, with the identifier `seq1`, indicating that the translator and the code generator functioned as expected. Finally, executing the generated executable on the Commodore 64 emulator confirms the playback of a single note. For an overview of all integration and acceptance tests, see Appendix G.1.

Another test involves a negative octave value for a note. This test shows that our error handling could be improved. Compiling the file, the error message *"Lexical Error: Invalid input, expected a token at line 8 character 23"* is displayed. The lexer correctly identifies the error, but the error message could be more specific.

This testing approach is also used for our acceptance tests. Acceptance testing is a methodology that aims to verify that the entire product meets its specified requirements from a user's perspective. Typically, acceptance testing includes actual users, to enable the user to assert that the product works as intended during typical usage [33]. However, since Neptune is an intermediate language, user acceptance testing is not as relevant. As a result, acceptance tests are conducted using the programs created by us, the developers, to simulate typical input. Therefore, the acceptance test approach has aimed to verify that the compiler produces the correct output for a specific program, adding confidence that the product will perform in production.

Ideally, a user acceptance test should have been performed on software developers with musical knowledge, to ensure that the source language's syntax is comprehensible and that its design provides a foundation for them to build their own musical tools. Moreover, the ideal integration testing process would be integrating components and performing tests step by step, as this would help pinpoint which specific components behave unexpectedly. However, as the results of the different components in the compilation process can be inspected, insight into the behaviour of smaller units is achievable. Additionally, the compiler's structured error handling enhances insight into component behaviour.

[7] GitHub Actions is a GitHub tool for automating and executing workflows directly in a repository [32]

Chapter 5

Reflections

In this chapter, we will reflect on the final product, expanding on its limitations and proposing potential improvements. Additionally, we will outline features for future development.

Sound Design

The ADSR and filter settings significantly influence the characteristic sound of chiptune music. Consequently, the limited scope of our syntax, which does not provide access to the full capabilities of the SID chip, presents significant limitations. For instance, compiled Neptune programs use predefined ADSR settings, the filters are disabled, and the volume cannot be adjusted. Although interesting and complex melodies can be produced with our compiler, the product inherently limits the potential for varied and complex sounds. Therefore, adding syntax and functionality to adjust the filter and ADSR settings would be an important improvement.

Compiler Architecture

Limitations can also be found in the compiler's architecture. In its current state, two separate ASTs and a symbol table are used in the compilation process, with only minor changes from the first to the second AST. While it is standard practice to have multiple ASTs and a symbol table, our compiler could have used a single AST. Symbol tables are used to keep track of and store information through the compiler's different phases, but for our compiler, which does not rely on variables and functions, and which uses an assembler where memory addresses are irrelevant, symbol table management becomes redundant. Therefore, for the *current* version of our compiler, a simpler solution would have been to store the sequences in the source AST, remove the target AST and the symbol table, and then perform the translations as a part of the code generation phase directly from the source AST. This solution would have improved compilation time. However, *future improvements* to the language would add complexity to the compiler, meaning that the need for a second or multiple ASTs is likely. For instance, with the possibility of adding control structures to Neptune, a symbol table becomes essential to keep track of variable names and scopes [34]. Therefore, while the compiler architecture might not be optimal for the current iteration, it provides a solid foundation for future development.

High-Level Languages Extending Neptune

Following the development of our language Neptune, we can consider high-level languages that could extend Neptune. Having taken inspiration from LilyPond's syntax (see section 2.1), a fitting option would be to compile LilyPond programs to Neptune. LilyPond's syntax also supports writing chords, which would enable users to produce chiptune music through vertical composition instead of horizontal. In Neptune, it is most straightforward to compose melodies (horizontal), whereas vertical composition has a greater focus on chords and harmonies [6, p. 73]. Other ideas could include incorporating control structures in the language, which will be expanded on in the following.

Control Structures

Contrary to most programming languages, Neptune does not possess control structures, however, adding these could be an interesting extension. Imagine writing a for-loop that repeats a certain note, or a sequence of notes. Such control structures would be even more vital if Neptune had dynamic sound generation rather than static sound. With control structures and an interactive sound system, we could make sounds based on actions made on the C64. Suppose a sound defined in a sequence is played if a user clicks the space key. Now, imagine that the Commodore 64, while playing music, made graphics based on the sound it was playing. All this could be introduced with the implementation of control structures in Neptune, but as we have described Neptune as intermediate, it could also be introduced by others.

Testing with Hardware

Having verified that the compiler's output, an executable file, can run on a Commodore 64 emulator, we then considered its execution on an actual C64. With this project's limited resources, getting hold of a Commodore 64 has not been possible, meaning that we cannot verify that the output can be executed on the actual hardware. If there ever was a possibility to test the compiler's output on the hardware, the method of testing it is not simple. It would require writing the executable file to a floppy disk, which then could be inserted into a floppy disk drive for the C64. Performing this test would certainly be interesting, and only after can we assert that our compiler is truly authentic to the Commodore 64 and its SID chip.

Percussion Algorithm

The traditional method of introducing percussion to chiptune music is to distribute the percussive elements to the three voices. Achieving this is possible with our source language, but the method is exhausting, as users must find or allocate the spaces for percussion in the melodies themselves. Therefore, developing a feature where users could create a separate percussion sequence, which the compiler could distribute onto the three voices, would be an attractive improvement.

Chapter 6

Conclusion

With a rise in interest in old hardware and chiptune music, the objective in this project has been to design an intermediate, domain-specific language to interact with the Commodore 64's SID chip to generate chiptune music. It could be argued that we have achieved this goal, as our language, Neptune, allows users to write music, compile it into an executable file, and run it on a Commodore 64 emulator. Neptune is a relatively verbose and structured language that is less writable, but very readable, arguably making it a good intermediary language due to its clarity and declarative structure.

Having conducted unit, integration, and acceptance tests to validate our compiler, we have a reasonable level of confidence that it works as expected. However, in terms of testing the compiled music, we cannot definitively confirm that the compiler works as intended, due to us not having access to a Commodore 64. Therefore, full validation of compiled code on the target hardware is a crucial next step. Additionally, performing user acceptance tests with potential developers is essential in confirming Neptune's practical relevance.

As Orson Welles once said, "The enemy of art is the absence of limitations", and in the world of chiptune, the inherent limitations of retro hardware such as the Commodore 64's sound chip are exactly what fuels unique artistic pieces of music. Neptune gives developers and musicians an easy way to conform to such restraints, both in the sense of creating music directly with the compiler and further development by using it in other projects.

Appendices

Appendix A

Chiptune

Chiptune is an electronic music genre that employs the sound chips found in vintage arcade machines, gaming consoles, and computers from the late 1970s to early 1980s [13]. This musical style evokes a sense of nostalgia, reminiscent of the simple and synthesised sounds of classic arcade and video console games from the golden age of video arcade games. Its artists work within the limitations of retro hardware, leading to simple yet innovative and expressive musical output.

A.1 The Origin of Chiptune

Chiptune as a music genre originated in the late 1980s as a homage to retro video game music [35]. To get a better understanding of the origin of the genre, it is necessary to first understand the development of early video game sound. In early home computers of the late 1970s, such as the IBM PC and Apple II, audio outputs were directly connected to and controlled by the computer's CPU (Central Processing Unit). Producing anything more advanced than simple clicks and beep sounds would therefore require most of the CPU's runtime [36]. Consequently, the CPU could not generate sound while simultaneously displaying other elements of the game, such as motion graphics and colours, which is why the video games of that era did not feature continuous music. A notable example is the classic video game 'Pong', created in 1972, which was one of the first video games to include sound [13]. The game only produces a simple beep sound as more complex sounds would interfere with the rendering of the game.

During the late 1970s, developers started implementing dedicated sound chips to produce video game music. With these sound chips, music would be generated independently from the CPU. 'Space Invaders', 1978, was the first game to feature continuous music throughout gameplay. Programmable Sound Generators (PSG) were introduced with the 8-bit chips in the early 1980s. These sound chips were more advanced, allowing for more sophisticated and nuanced music production. These powerful 8-bit PSG chips allowed for multi-channel sound, meaning they could play more than one note simultaneously. By the early 1980s, most computers and game consoles would have dedicated sound chips to take the load away from the CPU, relieving it of the dual load of displaying game graphics and producing a music output simultaneously. Many video game consoles and computers of this

time, such as the Commodore 64 and Nintendo Entertainment System (NES), would feature these improved sound chips. An example of this is the iconic soundtrack of 'Lazy Jones', a retro video game developed for the Commodore 64 in 1984 by British video game composer Roger Whittaker [13].

While this music type initially emerged as part of retro video games, it began to take form as its own music genre, with artists taking inspiration from the distinctive 8-bit sound. They began producing music which emulated these nostalgic sounds using more advanced technology [35].

A.2 Sound Chips

The sound chip is the foundation of chiptune music. How sound chips generate sound is a defining factor of the chiptune music style. As the defining characteristics of chiptune music are based on the capabilities of sound chips, understanding the basic foundations of a sound chip provides a greater understanding of the chiptune music genre.

A sound chip is an integrated circuit designed to process audio signals and produce sound through digital, analogue, or mixed-mode electronics [37]. In the context of retro gaming consoles, sound chips were responsible for generating the iconic synthetic sounds of early video games. As one of the most common sound chips implemented in vintage video game consoles and computers is the PSG, a digital sound chip with an analogue output.

The majority of the early produced PSGs had three independent pulse wave channels, and most had an additional channel dedicated to producing noise. Some of the early PSGs had limited sawtooth capabilities, but they were primarily restricted to pulse and noise. The SID (Sound Interface Device) is a specific type of PSG, developed in 1982, that was more advanced than its counterparts. The SID chip introduced an additional triangle wave and allowed for each of its three channels to independently produce any of the four waveforms. This versatile sound chip played a huge role in popularising the genre of chiptune music, as it was used in the Commodore 64, 1982, a popular home computer at the time. The advancement and versatility of the SID chip, hand in hand with the Commodore 64, allowed many musicians to experiment with digital sounds [38].

Appendix B

Music Theory

This appendix expands on some of the theory of Western music used for developing Neptune.

B.1 Note Types

To compose the structure of a melody, its *rhythm*, a hierarchy of different notes is used (see figure B.1). A whole note can be divided into two half notes, each half note into two quarter notes, and so on. The fractional property of each note indicates its duration in accordance with the tempo (see section B.4). For each type of note, there also exists an equivalent rest symbol representing a pause [6, p. 9].

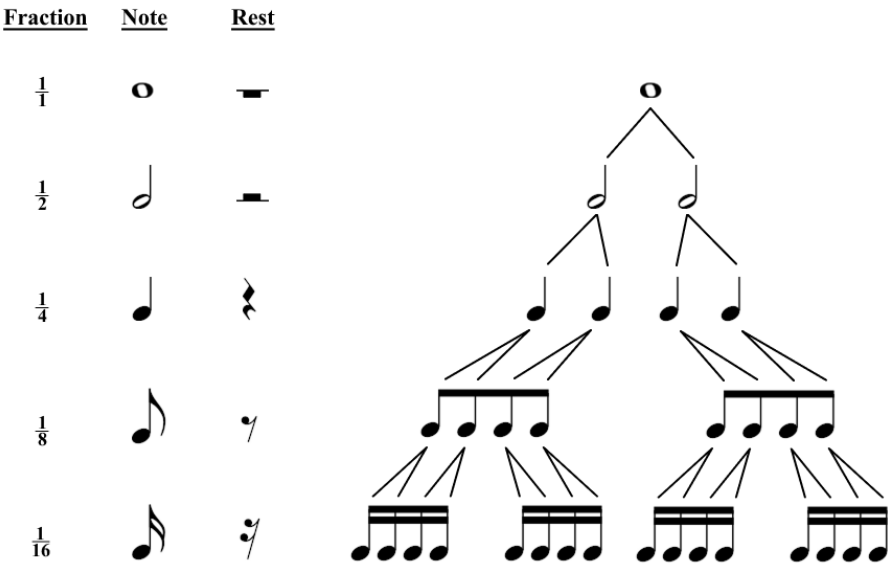


Figure B.1: An overview of the notes subdividing the whole note, their respective fractions, and equivalent rest symbols

B.2 Time Signatures

To order a sequence of notes and rests over time, they are placed within a sequence of *measures*, which are separated by vertical bars. The measure delimitates the space available for notes with a *time signature*, expressed as you would a fraction. The upper value specifies the number of basic note values (beats) per measure, and the lower defines the basic note value itself [6, p. 11].

The most common time signature is 4/4, and thus it is often referred to as *common time* [39]. It has a basic note value of 4 (quarter note), with four of these quarter notes per measure (see figure B.2).

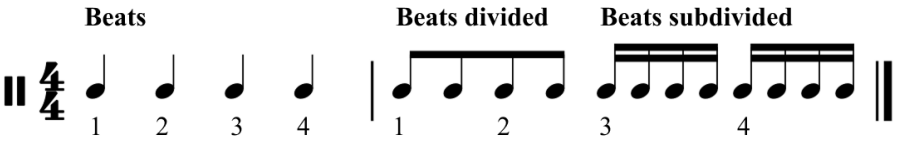


Figure B.2: Notation of two measures in common time

B.3 Tones

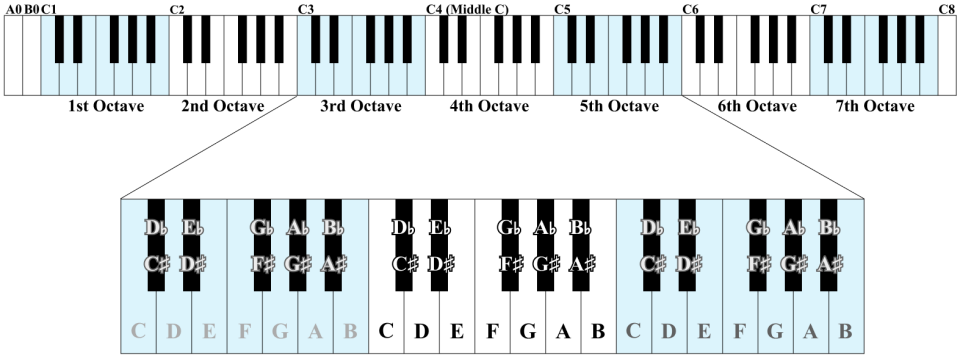


Figure B.3: This figure shows the structure of the twelve tones on a piano and how the structure repeats. The interval between two adjacent piano keys is called a semitone, also referred to as a half step [6, p. 7]

B.4 Tempo

The tempo of a piece of music assigns a duration of time to each of its beats. This is measured in the unit of beats per minute (BPM) [7]. For example, common time has a basic note value of 4, so the beats in this context are quarter notes. Given a tempo of 60 beats per minute, this means each quarter note will have a duration of one second. If the tempo were 120 BPM, each quarter note would last only half a second.

B.5 Pitch

Each note represents a certain pitch, which is the highness or lowness of a tone. For example, playing the note A in the middle of a piano will produce a frequency of 440 Hz[1]. Playing a note an octave higher, its frequency is double that of the original note, so playing the A an octave above the middle A will produce a frequency of 880 hertz. The other notes in an octave are distributed with an equal ratio of frequency between them. This system of tuning is called the *equal temperament* [6, p. xiv] [26].

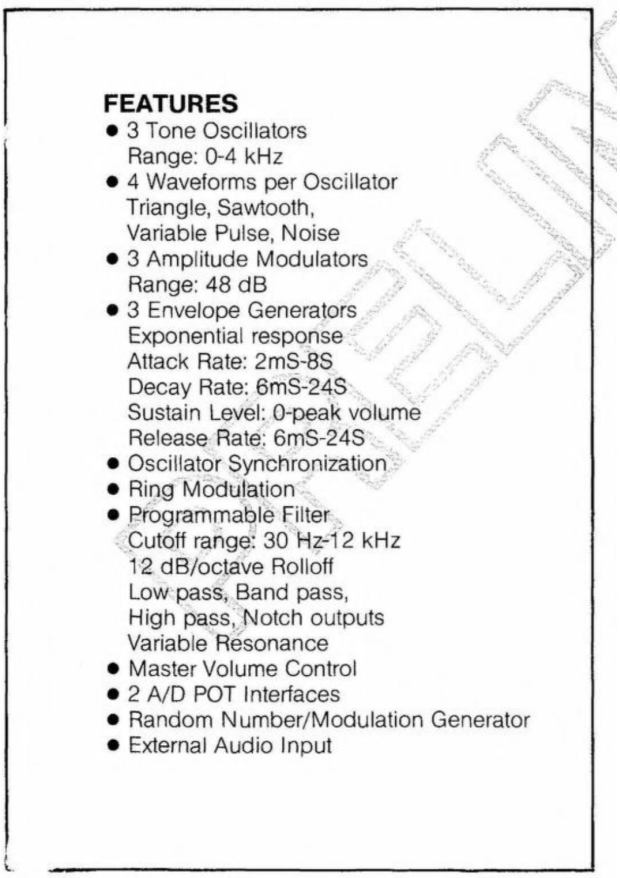
[1]This frequency is often considered the standard pitch as it is the reference point for tuning musical instruments [8]

Appendix C

Hardware

This appendix expands on the SID 6581 chip, as well as includes figures that document how to program the Commodore 64 in both 6502 Assembly language and BASIC.

C.1 SID 6581 Features



FEATURES

- 3 Tone Oscillators
  Range: 0-4 kHz
- 4 Waveforms per Oscillator
  Triangle, Sawtooth,
  Variable Pulse, Noise
- 3 Amplitude Modulators
  Range: 48 dB
- 3 Envelope Generators
  Exponential response
  Attack Rate: 2mS-8S
  Decay Rate: 6mS-24S
  Sustain Level: 0-peak volume
  Release Rate: 6mS-24S
- Oscillator Synchronization
- Ring Modulation
- Programmable Filter
  Cutoff range: 30 Hz-12 kHz
  12 dB/octave Rolloff
  Low pass, Band pass,
  High pass, Notch outputs
  Variable Resonance
- Master Volume Control
- 2 A/D POT Interfaces
- Random Number/Modulation Generator
- External Audio Input

Figure C.1: Complete list of the features of the SID 6581 [10]

C.2 SID 6581 Control Registers

These tables within figure C.2 and C.3 provide an overview of the different registers of the SID chip [10][40]. The register index is the relative index of the register within the chip, while the memory address denotes the absolute address in the memory of the C64. The name gives a short description of the data contained in the registers, while the eight bits of data contained within it are below, labelled D7-D0. Some registers, such as Frequecy (Low) and Frequency (High), use two addresses to represent a 16 bit value. Others, such as Attack/Decay, represent two 4-bit values. In some cases, as with the Control Register, each bit functions as a toggle for a specific feature.

VOICE 1

Register Index	Memory Address	Name	D7	D6	D5	D4	D3	D2	D1	D0
0	$D400	Frequency (Low)	F7	F6	F5	F4	F3	F2	F1	F0
1	$D401	Frequency (High)	F15	F14	F13	F12	F11	F10	F9	F8
2	$D402	Pulse Width (Low)	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0
3	$D403	Pulse Width (High)	-	-	-	-	PW11	PW10	PW9	PW8
4	$D404	Control Register	Noise	VPulse	Sawtooth	Triangle	Test	Ring Mod	Sync	Gate
5	$D405	Attack/Decay	A3	A2	A1	A0	D3	D2	D1	D0
6	$D406	Sustain/Release	S3	S2	S1	S0	R3	R2	R1	R0

VOICE 2

Register Index	Memory Address	Name	D7	D6	D5	D4	D3	D2	D1	D0
7	$D407	Frequency (Low)	F7	F6	F5	F4	F3	F2	F1	F0
8	$D408	Frequency (High)	F15	F14	F13	F12	F11	F10	F9	F8
9	$D409	Pulse Width (Low)	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0
10	$D40A	Pulse Width (High)	-	-	-	-	PW11	PW10	PW9	PW8
11	$D40B	Control Register	Noise	VPulse	Sawtooth	Triangle	Test	Ring Mod	Sync	Gate
12	$D40C	Attack/Decay	A3	A2	A1	A0	D3	D2	D1	D0
13	$D40D	Sustain/Release	S3	S2	S1	S0	R3	R2	R1	R0

Figure C.2: Overview of SID chip registers (1/2)

VOICE 3

Register Index	Memory Address	Name	D7	D6	D5	D4	D3	D2	D1	D0
14	$D40E	Frequency (Low)	F7	F6	F5	F4	F3	F2	F1	F0
15	$D40F	Frequency (High)	F15	F14	F13	F12	F11	F10	F9	F8
16	$D410	Pulse Width (Low)	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0
17	$D411	Pulse Width (High)	-	-	-	-	PW11	PW10	PW9	PW8
18	$D412	Control Register	Noise	VPulse	Sawtooth	Triangle	Test	Ring Mod	Sync	Gate
19	$D413	Attack/Decay	A3	A2	A1	A0	D3	D2	D1	D0
20	$D414	Sustain/Release	S3	S2	S1	S0	R3	R2	R1	R0

FILTER

Register Index	Memory Address	Name	D7	D6	D5	D4	D3	D2	D1	D0
21	$D415	Filter Cutoff (Low)	-	-	-	-	-	FC2	FC1	FC0
22	$D416	Filter Cutoff (High)	FC10	FC9	FC8	FC7	FC6	FC5	FC4	FC3
23	$D417	Filter/Resonance	RES3	RES2	RES1	RES0	Filter EX	Filter 3	Filter 2	Filter 1
24	$D418	Volume/Filter Select	Voice 3 off	High Pass	Band Pass	Low Pass	VOL3	VOL2	VOL1	VOL0

Figure C.3: Overview of SID chip registers (2/2)

C.3 6502 Assembly & BASIC

IN ASSEMBLY

```
sys:
            dc.b $0b,$08
            dc.b $0a,$00
            dc.b $9e,$32,$30,$36,$31
            dc.b $00
            dc.b $00,$00
```

IN BASIC

```
10  SYS2061
```

BYTE STRUCTURE

No. of bytes	Memory Address	Definition	B4	B3	B2	B1	B0	Interpretation
2	$0801	Address of next line	-	-	-	$0b	$08	Address $080B (little endian)
2	$0803	Line number	-	-	-	$0a	$00	Line number 10 (little endian)
1+	$0805	Tokenized BASIC code	$9e	$32	$30	$36	$31	Tokens 'SYS', '2', '0', '6', and '1'
1	$080A	End of line	-	-	-	-	$00	Terminates the line
2	$080B	End of program	-	-	-	$00	$00	Terminates the program

Figure C.4: This figure shows how we use 6502 Assembly byte data to structure a program in BASIC. This BASIC program is what executes in the C64 prompt and calls the remaining machine code stored from memory at address $0801 (2061 in decimal, which is what is used in BASIC). First, a comparison is shown between the written assembly code and the BASIC code it is interpreted as. Below, a description of the byte structure is elaborated on

C.4 Assembly Variables

VOICE VARIABLES

Name	Description	Initial values Voice1 (index 0)	Voice2 (index 1)	Voice3 (index 2)
v_regindex	Relative index of registers within SID chip	$00	$07	$0E
v_freqlo	Low frequency of current note	$00	$00	$00
v_freqlo_new	Low frequency of most recently fetched note	$00	$00	$00
v_freqhi	High frequency of current note	$00	$00	$00
v_freqhi_new	High frequency of most recently fetched note	$00	$00	$00
v_instr	Instrument index for current note	$00	$00	$00
v_pulselo	Low byte of current note's pulse width	$00	$00	$00
v_pulsehi	High byte of current note's pulse width	$00	$00	$00
v_waveform	Waveform of current note	$00	$00	$00
v_ad	Attack and decay rate of current note	$00	$00	$00
v_sr	Sustain and release rate of current note	$00	$00	$00
v_counter	Remaining duration of current note	$02	$02	$02
v_counternew	Duration of most recently fetched note	$00	$00	$00
v_ptrlo	Low byte of pointer to next byte in music data	$00	$00	$00
v_ptrhi	High byte of pointer to next byte in music data	$00	$00	$00
v_rtnlo	Low byte of most recently fetched return address	$00	$00	$00
v_rtnhi	High byte of most recently fetched return address	$00	$00	$00

Figure C.5: This table depicts all voice variables within our assembly code, as well as the initial values they contain

INSTRUMENT VARIABLES

Name	Description	Initial values			
		noise (index 0)	vPulse (index 1)	sawtooth (index 2)	triangle (index 3)
i_pulselo	Low byte of pulse width	$00	$00	$07	$0E
i_pulsehi	High byte of pulse width	$00	$02	$00	$00
i_pulsespeed	Pulse speed	$00	$20	$00	$00
i_ad	Attack and decay rate	$0a	$09	$58	$0a
i_sr	Sustain and release rate	$00	$00	$aa	$f0
i_waveform	Waveform	$81	$41	$21	$11

Figure C.6: This table depicts all instrument variables within our assembly code, as well as the initial values they contain

Appendix D

Sound Waves

Sound is caused by vibrations, and these vibrations are transmitted through the air in the form of sound waves. The sound waves are then simply a displacement of molecules in the medium from their resting position and back [41]. In figure D.1, such a graphical representation of the displacement and change over time is shown, with the x-axis typically representing time and the y-axis representing the amplitude of the displacement.

D.1 Amplitude

Amplitude can be seen as the maximum displacement of a given sound wave and is measured from zero to the maximum amount displaced (see figure D.1). To simplify, the amplitude can be thought of as loudness, with an amplitude of 0, a sound wave following the X-axis would represent complete silence. In contrast a higher amplitude would produce an actual sound and would increase in loudness if amplitude was increased [42].

D.2 Frequency

Frequency measures how many times a waveform iterates in a certain period of time, and is often measured in hertz (Hz), the number of repetitions per second. The waveform seen in figure D.1 is periodic, meaning that it repeats at regular intervals over time indefinitely. These are typically generated with a computer, as instruments and voices tend to be more complex, producing waveforms that contain multiple frequencies [42].

D.3 Harmonics

Harmonics are additional waveforms with different frequencies that are created by certain waveforms. This is because complex waveforms can be described as a bunch of different sine waves layered on top of each other, making the sine wave the fundamental waveform[1] (see section D.5) [42]. It is possible to hear the constant frequency of a sine wave, while other waveforms have varying harmonic

[1] Also known as the root frequency

frequencies that contribute to their overall sound. Harmonics are always multiples of the fundamental frequency. In even harmonics, if the fundamental frequency is 1 Hz, the second harmonic is 2 Hz, the fourth harmonic 4 Hz, and this pattern continues to infinity. Odd harmonics include the root frequency, the 3rd, 5th harmonics, and so on.

D.4 Waveforms

In general, waveforms are representations of sound over time, and there exist many types of waveforms. As stated earlier, the sine wave (see figure D.1), is the fundamental waveform, a smooth periodic wave that moves between positive and negative values. It is defined by the sine function $y(t) = A \sin 2\pi ft + \phi$ in which A is the amplitude, f is the frequency, and ϕ is the phase[2]. Sine waves are important for synthesising audio, because they, as stated in Fourier's theorem (see section D.5), can recreate any other periodic waveform [43]. They produce a clear tone without any harmonics.

D.5 Sine Waves

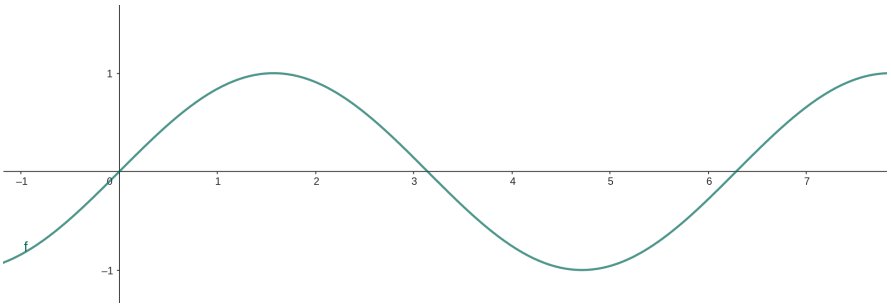


Figure D.1: Simple sine wave, described by $f(x) = sin(x)$

Note

Fourier's theorem states "Any periodic signal is composed of a superposition of pure sine waves, with suitably chosen amplitudes and phases, whose frequencies are harmonics of the fundamental frequency of the signal." [44] Following this theorem, a periodic square wave or any other periodic waves can be described as a series of sinusoidal functions.

In the following sections, the waveforms are described using additive synthesis [45]. This method of synthesising waveforms is **not** the method used inside of the SID 6581 chip [10], which is subtractive synthesis. Although due to subtractive synthesis being more abstract when needing to visualise it, due to filtering frequencies makes it harder to visualise, we have chosen to use additive synthesis to illustrate the waveforms in our project.

D.5.1 The Triangle Waveform

The triangle waveform (see figure D.2) is periodic and has a triangle shape. An ideal triangle wave mostly consists of odd harmonics, and the amplitude of these harmonics decreases to the square of

[2]Phase is the timing of when the wave cycle begins.

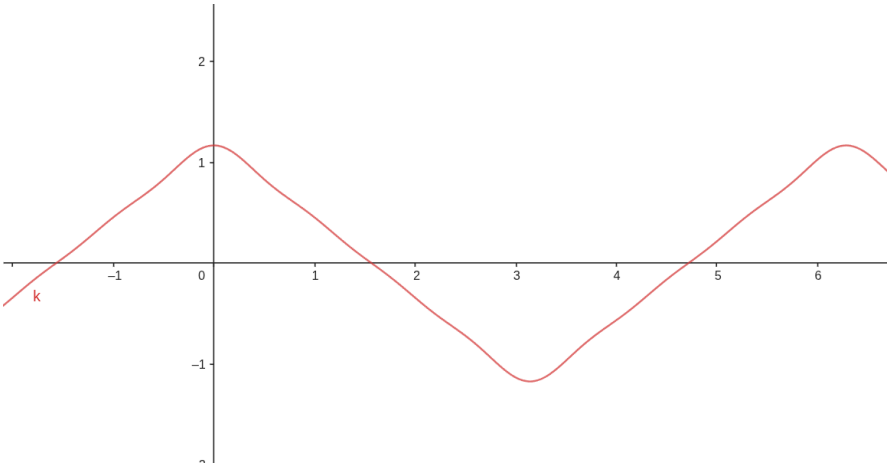


Figure D.2: Simple Triangle wave created with additive synthesis [46]

the harmonic number, thus making the 3rd harmonic 1/9th of the amplitude of the fundamental. This waveform has a flute-like and pure sounding tone [46].

D.5.2 The Sawtooth Waveform

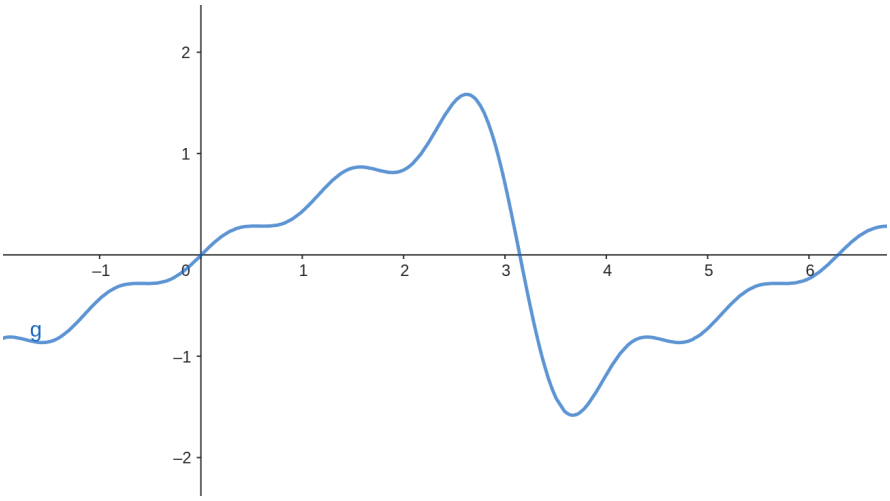


Figure D.3: Simple sawtooth wave created with additive synthesis with four harmonics

The sawtooth waveform (see figure D.3) is periodic and has a ramp shape that rises. It contains both odd and even harmonics and their amplitudes decrease inversely in relation to the harmonic numbers. The sawtooth wave has many harmonic frequencies, making it bright and buzzy [47].

D.5.3 The Variable Pulse Waveform

The variable pulse waveform (see figure D.4) is a periodic waveform that alternates between two levels of amplitude, high and low. The shape of the variable pulse waveform is determined by the *duty cycle*, which is the ratio of the time before switching between the two levels. In relation to harmonics, the variable pulse harmonics are dependent on the duty cycle and is not fixed like with triangle or sawtooth. The sound produced by the waveform is also duty cycle dependent, A higher duty cycle

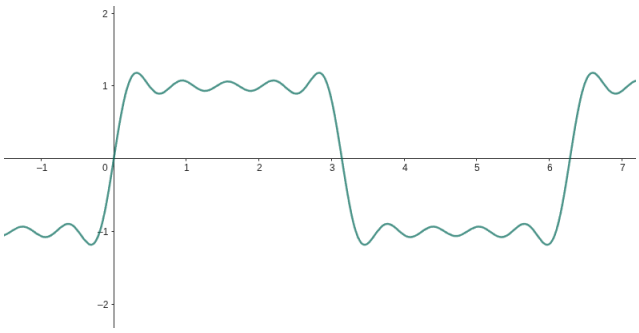


Figure D.4: Variable Pulse wave created with additive synthesis with five harmonics [46]

produces a hollow sound, while a smaller one turns it into a thinner sound [46].

D.5.4 The Noise Waveform

The noise waveform is a non-periodic waveform that has a more random pattern or has a completely pseudo-random pattern over time. In terms of usage in sound synthesis, when played quickly, it has a more percussive sound, almost like a snare or hi-hat [46].

Appendix E

MoSCoW Specifications

We use the MoSCoW model to prioritise and identify the specifications of our compiler. The model is divided into four sub-categories, each representing the level of prioritisation, as follows: *Must Haves, Should Haves, Could Haves,* and *Will Not Haves* [48].

Must Haves

ID	Description	Traceability
M01	The compiler must take a Neptune source file as input, and output an assembly file	Section 4.2
M02	The compiler must have a lexer that performs a lexical analysis on the input file/source language and outputs a stream of tokens	Section 4.2
M03	The compiler must translate the high-level AST generated from the source language into a low-level AST resembling the target language	Section 4.2
M04	The assembly code output by the compiler must be assembled into machine code by an external assembler	Section 4.2
M05	The compiled program must be executable on an external Commodore 64 emulator and generate proper sound output	Section 4.2
M06	Precise and informative compilation errors must be raised if compilation fails	N/A

Figure E.1: The must-have specifications are the most essential requirements for creating a minimum viable product. These requirements define the functionalities that the product should guarantee to deliver

Should Haves

ID	Description	Traceability
S01	The compiled program should utilise all three of the SID's voices	Section 3.1
S02	The compiled program should support multiple voices playing simultaneously, providing a complex sound output coherent with the SID 6581	Section 3.1.1
S03	The compiled program should support various distinct sequences to be played simultaneously by different voices	Section 3.1
S04	The compiled program should be able to generate sound output of varying waveforms within each voice	Section 3.1.1
S05	An assembler should be integrated in the compiler	Section 4.2

Figure E.2: The should-have specifications are non-essential features, meaning they are not necessary for the basic functionalities of the product, but add significant features and optimisation

Could Haves

ID	Description	Traceability
C01	The compiled program could accommodate user-defined envelopes	Section 3.1.3
C02	The compiled program could change ADSR values dynamically	Section 3.1.3
C03	The compiled program could handle user-defined frequency cut-off filters	Section 3.1.4

Figure E.3: The could-have specifications are features that would enhance the product if implemented, but would not have a significant impact if left out

Will Not Haves

ID	Description	Traceability
W01	The compiled program will not accommodate user interaction when run on the Commodore 64 emulator, as the focus of this project is utilisation of the hardware's sound generation capabilities	N/A
W02	The compiled program will not utilise the 'fourth' voice, allowing for more than three voices, as we follow the limitations of the SID 6581. As the fourth voice is not an actual voice but rather a glitch, we have decided not to include a fourth voice in our program	N/A

Figure E.4: The will-not-have specifications are features that we have decided not to prioritise for this project

Appendix F

Lexer

This appendix presents our regular expressions and lexing rules.

F.1 Regular Expressions

```
let digit = ['0'-'9']
let int = digit+

let whitespace = [' ' '\t' '|']+
let newline = "\r\n" | '\n' | '\r'
let tone = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "r"
let letter = ['a'-'z' 'A'-'Z']
let ident = letter (letter | '-' | digit)+
```

Listing F.1: Regular expressions

F.2 Lexing Rules

```
1  rule read = parse
2      | whitespace {read lexbuf}
3      | newline {Lexing.new_line lexbuf; read lexbuf}
4      | tone {TONE (Lexing.lexeme lexbuf)}
5      | ident as s {ident_or_keyword s}
6      | int {INT (int_of_string (Lexing.lexeme lexbuf))}
7      | "/*" {comment lexbuf}
8      | "#"  {SHARP}
9      | "_" {FLAT}
10     | "{" {LCB}
11     | "}" {RCB}
12     | "[" {LSB}
13     | "]" {RSB}
14     | "(" {SP}
15     | ")" {EP}
16     | ":" {COLON}
17     | "," {COMMA}
18     | "=" {ASSIGN}
19     | eof {EOF}
20     | _ {raise (LexicalErrorException
21         ("Invalid input, expected a token " ^ lexeme_error lexbuf))}
22
23 (...)
24
25 and comment = parse
26     | "*/" {read lexbuf}
27     | newline {unterminated_comment lexbuf}
28     | _ {comment lexbuf}
29     | eof {unterminated_comment lexbuf}
```

Listing F.2: Lexing rules

Appendix G

Testing

G.1 Integration and Acceptance Tests

This Appendix presents the results of the integration and acceptance tests. They are organised into successful tests that are expected to compile in figure G.1 and failing tests that are expected to display an error message in figure G.2. The figures show all tests' names, descriptions, and results.

SUCCESSFUL TESTS

Name	Description	Result
test000	One-note sequence inserted in voice1	Successful compilation. Output file produces a single note
test001	One-note sequence inserted in voice2	Successful compilation. Output file produces a single note
test002	One-note sequence inserted in voice3	Successful compilation. Output file produces a single note
test003	One-note sequences inserted in voice1 and voice2	Successful compilation. Output file produces two simultaneous notes
test004	One-note sequences inserted in voice2 and voice3	Successful compilation. Output file produces two simultaneous notes
test005	One-note sequences inserted in voice1 and voice3	Successful compilation. Output file produces two simultaneous notes
test006	One-note sequences inserted in all three voices	Successful compilation. Output file produces a full chord
test007	Multi-note sequences inserted in all three voices	Successful compilation. Output file produces a sequence of chords
test008	One sequence contains notes without defined octave	Successful compilation. Output file produces C major scale in three waveforms
test009	Comments inserted in some sequences and voices	Successful compilation. Output file disregards what has been commented out
test010	Lowered standard pitch of a melody	Successful compilation. Output file produces the melody in a lower pitch

Figure G.1: An overview of the tests that are expected to succeed

FAILING TESTS

Name	Description	Result
test000	Parameters are defined in the wrong order	*Parsing Error: Syntax error at line 5 character 1-13*
test001	Parameter names are misspelled	*Parsing Error: Syntax error at line 4 character 1-5*
test002	Standard pitch parameter is absent	*Parsing Error: Syntax error at line 7 character 1-8*
test003	Invalid value assigned to time signature	*Invalid Argument Error: Invalid basic note value in time signature, expected '1', 2', 4', 8', '16'*
test004	Delimiters of sequences are absent	*Parsing Error: Syntax error at line 9 character 1-8*
test005	Empty sequence is defined	*Parsing Error: Syntax error at line 11 character 11-19*
test006	Duplicate name of defined sequences	*Syntax Error: Sequences id's cannot be duplicated. Each sequence must have a unique id.*
test007	Invalid tones x, y and z are used	*Lexical Error: Invalid input, expected a token at line 8 character 19*
test008	Invalid duration 47 is used	*Invalid Argument Error: Invalid duration, expected '1', 2', 4', 8', '16'*
test009	Fraction of a note is absent	*Parsing Error: Syntax error at line 8 character 21*
test010	Invalid octave -5 is used (negative value)	*Lexical Error: Invalid input, expected a token at line 8 character 23*
test011	Invalid octave 8 is used (exceeds 7)	*Invalid Argument Error: Invalid octave, expected an integer between 0 and 7*
test012	Undefined sequence identifier used in voice	*Syntax Error: Sequences must be defined before adding to a voice*
test013	Invalid waveforms 'pulse' and 'sawtootg' are used	*Invalid Argument Error: Invalid waveform, expected 'noise', 'vPulse', 'sawtooth', 'triangle'*
test014	Delimiter of comment is missing	*Syntax Error: Unterminated comment at line 16*
test015	Multi-line comment is used	*Syntax Error: Unterminated comment at line 8*

Figure G.2: An overview of the tests that are expected to fail

Appendix H

Neptune Sample Code

This appendix presents two example programs, one is a small program to present the syntactic structure of Neptune, while the other is a bigger program based on the Tetris theme (these sample codes can also be found in our Github repository, linked in the Synopsis page). Following is the assembly code of the compiled Tetris program, which highlights the vast contrast between the source and target code. The Tetris program consists of just 59 lines of code, while its assembly counterpart is an impressively 504 lines long. This illustrates the considerable effort that has gone into developing our target languages.

H.1 Mini Sample Code

```
1  tempo = 120
2  timeSignature = (4,4)
3  standardPitch = 440
4
5  sequence seq1 = { c:1:4 }
6  sequence seq2 = { d:4:4 e_:4:4 }
7  sequence seq3 = { g:8:3 f#:8:3 a:4:3 }
8
9
10 voice1 = [(seq1, square)]
11 voice2 = [(seq2, triangle), (seq2, triangle)]
12 voice3 = [(seq3, sawtooth), (seq2, noise)]
```

Listing H.1: Small source code example for Neptune

H.2 Tetris Sample Code

```
tempo = 150
timeSignature = (2,4)
standardPitch = 440

sequence main = {
    e:4:  b:8:3 c:8: | d:8: e:16: d:16: c:8: b:8:3 |
    a:4:3 a:8:3 c:8: | e:4:  d:8  c:8  |
    b:4:3 r:8   c:8  | d:4              e:4        |
    c:4   a:4:3      | a:4:3 r:4       |
    r:8 d:4:    f:8: | a:4:             g:8: f:8:  |
    e:4:  r:8   c:8: | e:4:  d:8: c:8: |
    b:4:3 r:8   c:8: | d:4:             e:4:       |
    c:4:  a:4:3      | a:4:3 r:4       |
}

sequence bridge = {
    e:2: | c:2: | d:2: | b:2:3 |
}

sequence bridgeA = {
    c:2: | a:2:3 | g#:4:3 b:4:3 | e:4: r:4|
}

sequence bridgeB = {
    a:4:3 e:4: | a:2: | g#:2 | r:2 |
}

sequence bass1 = {
    e:8:2  e:8:3  e:8:2  e:8:3  | e:8:2  e:8:3  e:8:2  e:8:3  |
    a:8:1 a:8:2 a:8:1 a:8:2     | a:8:1 a:8:2 a:8:1 a:8:2     |
    g#:8:1 g#:8:2 g#:8:1 g#:8:2 | g#:8:1 g#:8:2 g#:8:1 g#:8:2 |
    a:8:1 a:8:2 a:8:1 a:8:2     | a:8:1 a:8:1 b:8:1 c:8:2     |
}

sequence bass2 = {
    d:8:2 d:8:3 d:8:2 d:8:3 | d:8:2 d:8:3 d:8:2 d:8:3 |
    c:8:2 c:8:3 c:8:2 c:8:3 | c:8:2 c:8:3 c:8:2 c:8:3 |
    b:8:1 b:8:2 b:8:1 b:8:2 | b:8:1 b:8:2 b:8:1 b:8:2 |
    a:8:1 a:8:2 a:8:1 a:8:2 | a:8:1 a:8:2 a:8:1 a:8:2 |
}

sequence bass3 = {
```

```
    a:8:1 a:8:2 a:8:1 a:8:2     | a:8:1 a:8:2 a:8:1 a:8:2     |
    g#:8:1 g#:8:2 g#:8:1 g#:8:2 | g#:8:1 g#:8:2 g#:8:1 g#:8:2 |
    a:8:1 a:8:2 a:8:1 a:8:2     | a:8:1 a:8:2 a:8:1 a:8:2     |
    g#:8:1 g#:8:2 g#:8:1 g#:8:2 | r:8    g#:8:2 g#:4:2        |
}

sequence kickHit = { d:16:0 }
sequence kickResonate = { d:8:0 r:16 }
sequence snare = { a:8:5 r:8 }

voice1 = [(main, triangle), (bridge, triangle), (bridgeA, triangle),
          (bridge, triangle), (bridgeB, triangle)]

voice2 = [(bass1,sawtooth), (bass2, sawtooth),
          (bass3, sawtooth), (bass3, sawtooth)]

voice3 = [(kickHit,noise), (kickResonate,vPulse), (snare, noise)]
```

Listing H.2: Tetris source code example for Neptune

H.3 Tetris Assembly Code

```
1                       processor 6502
2                       org 2049
3
4   temp1               = $fb
5   temp2               = $fc
6
7
8   sys:
9                       dc.b $0b,$08
10                      dc.b $0a,$00
11                      dc.b $9e
12                      dc.b $32,$30,$36,$31
13                      dc.b $00
14                      dc.b $00,$00
15
16  start:
17                      jsr $1000
18
19                      sei
20
21                      lda #<raster
22                      sta $0314
23                      lda #>raster
24                      sta $0315
25
26                      lda #50
27                      sta $d012
28                      lda $d011
29                      and #$7f
30                      sta $d011
31
32                      lda #$7f
33                      sta $dc0d
34
35                      lda #$01
36                      sta $d01a
37
38                      lda $dc0d
39
40                      cli
41
42                      rts
43
44
45  raster:
46                      nop
47                      nop
48                      nop
49                      nop
50                      nop
51                      nop
52                      nop
53                      nop
54                      nop
55
56                      jsr play
57
58                      dec $d019
59                      jmp $ea31
60
61
62                      org $1000
63  play_init:
64                      ldx #$00
65
66                      lda #$00
67                      sta $d417
68
69                      lda #$0f
70                      sta $d418
71
72                      lda #<voice1
73                      sta v_ptrlo,x
74                      lda #>voice1
75                      sta v_ptrhi,x
76
77                      inx
78                      lda #<voice2
79                      sta v_ptrlo,x
80                      lda #>voice2
81                      sta v_ptrhi,x
82
83                      inx
84                      lda #<voice3
85                      sta v_ptrlo,x
86                      lda #>voice3
87                      sta v_ptrhi,x
88
89                      rts
90
91
92  gatebit_off:
93                      lda #$00
94                      sta v_waveform,x
95                      jmp counter_init
96
97  note_init:
98                      lda v_freqlo_new,x
99                      beq gatebit_off
100                     sta v_freqlo,x
101                     lda v_freqhi_new,x
102                     sta v_freqhi,x
103
104                     ldy v_instr,x
105
106                     lda i_pulselo,y
107                     sta v_pulselo,x
108                     lda i_pulsehi,y
109                     sta v_pulsehi,x
110
111                     lda i_waveform,y
112                     sta v_waveform,x
113
114                     lda i_ad,y
115                     sta v_ad,x
116                     lda i_sr,y
117                     sta v_sr,x
118
119 counter_init:
120                     lda v_counternew,x
121                     sta v_counter,x
122
123                     jmp update_sid
124
125
126
127 play:
128                     ldx #$00
129
130 play_loop:
131                     lda v_counter,x
132                     beq note_init
133
134                     cmp #$02
135                     beq fetch
136
137 update_sid:
138                     dec v_counter,x
139                     ldy v_regindex,x
140
141                     lda v_freqlo,x
142                     sta $d400,y
143                     lda v_freqhi,x
144                     sta $d401,y
145
146                     lda v_pulselo,x
147                     sta $d402,y
148                     lda v_pulsehi,x
149                     sta $d403,y
150
151                     lda v_waveform,x
152                     sta $d404,y
153
154                     lda v_ad,x
155                     sta $d405,y
156                     lda v_sr,x
157                     sta $d406,y
158
159                     inx
160                     cpx #$03
161                     bcc play_loop
162                     rts
163
164
165 fetch:
166                     lda v_ptrlo,x
167                     sta temp1
168                     lda v_ptrhi,x
169                     sta temp2
170
171                     ldy #$00
172
173 fetch_loop:
174                     lda (temp1),y
175
176                     cmp #$f9
177                     bcc fetch_note
178
179                     iny
180
181                     cmp #$fd
182                     bcc load_instr
183
184                     beq jump_addr
185
186                     cmp #$fe
187                     beq enter_seq
188
189                     jmp exit_seq
190
191 enter_seq:
192                     lda temp1
193
194                     clc
195                     adc #$04
196                     sta v_rtnlo,x
197
198                     lda temp2
199                     adc #$00
200                     sta v_rtnhi,x
201
202 jump_addr
203                     lda (temp1),y
204                     sta v_ptrlo,x
205
206                     iny
207                     lda (temp1),y
208                     sta v_ptrhi,x
209
210                     jmp fetch
211
212 exit_seq:
213                     lda v_rtnlo,x
214                     sta v_ptrlo,x
215                     lda v_rtnhi,x
216                     sta v_ptrhi,x
217
218                     jmp fetch
219
220 load_instr:
221                     sec
222                     sbc #$F9
223                     sta v_instr,x
224                     jmp fetch_loop
225
226 fetch_note:
227                     sta v_freqlo_new,x
228
229                     iny
230                     lda (temp1),y
231                     sta v_freqhi_new,x
232
233                     iny
234                     lda (temp1),y
235                     sta v_counternew,x
236
237                     lda v_waveform,x
238                     and #$fe
239                     sta v_waveform,x
240
241                     iny
242                     tya
243                     clc
244                     adc temp1
245                     sta v_ptrlo,x
246
247                     lda temp2
248                     adc #$00
249                     sta v_ptrhi,x
250
251                     jmp update_sid
252
```

Figure H.1: Tetris target code example for Neptune (1/2)

```
253
254 v_regindex:     dc.b $00,$07,$0E
255 v_freqlo:       dc.b $00,$00,$00
256 v_freqlo_new:   dc.b $00,$00,$00
257 v_freqhi:       dc.b $00,$00,$00
258 v_freqhi_new:   dc.b $00,$00,$00
259 v_instr         dc.b $00,$00,$00
260 v_pulselo       dc.b $00,$00,$00
261 v_pulsehi       dc.b $00,$00,$00
262 v_waveform:     dc.b $00,$00,$00
263 v_ad:           dc.b $00,$00,$00
264 v_sr:           dc.b $00,$00,$00
265 v_counter:      dc.b $02,$02,$02
266 v_counternew:   dc.b $00,$00,$00
267 v_ptrlo:        dc.b $00,$00,$00
268 v_ptrhi:        dc.b $00,$00,$00
269 v_rtnlo:        dc.b $00,$00,$00
270 v_rtnhi:        dc.b $00,$00,$00
271
272
273 i_pulselo:      dc.b $00,$00,$00,$00
274 i_pulsehi:      dc.b $00,$02,$00,$00
275 i_pulsespeed:   dc.b $00,$20,$00,$00
276 i_ad:           dc.b $0a,$09,$58,$0a
277 i_sr:           dc.b $00,$00,$aa,$f0
278 i_waveform:     dc.b $81,$41,$21,$11
279
280 voice1:
281                 dc.b $FC
282                 dc.b $FE
283                 dc.w main
284                 dc.b $FC
285                 dc.b $FE
286                 dc.w bridge
287                 dc.b $FC
288                 dc.b $FE
289                 dc.w bridgeA
290                 dc.b $FC
291                 dc.b $FE
292                 dc.w bridge
293                 dc.b $FC
294                 dc.b $FE
295                 dc.w bridgeB
296                 dc.b $FD
297                 dc.w voice1
298 voice2:
299                 dc.b $FB
300                 dc.b $FE
301                 dc.w bass1
302                 dc.b $FB
303                 dc.b $FE
304                 dc.w bass2
305                 dc.b $FB
306                 dc.b $FE
307                 dc.w bass3
308                 dc.b $FB
309                 dc.b $FE
310                 dc.w bass3
311                 dc.b $FD
312                 dc.w voice2
313 voice3:
314                 dc.b $F9
315                 dc.b $FE
316                 dc.w kickHit
317                 dc.b $FA
318                 dc.b $FE
319                 dc.w kickResonate
320                 dc.b $F9
321                 dc.b $FE
322                 dc.w snare
323                 dc.b $FD
324                 dc.w voice3
325 main:
326                 dc.b $1E, $15, $14
327                 dc.b $D2, $0F, $0A
328                 dc.b $C3, $10, $0A
329                 dc.b $D0, $12, $0A
330                 dc.b $1E, $15, $05
331                 dc.b $D0, $12, $05
332                 dc.b $C3, $10, $0A
333                 dc.b $D2, $0F, $0A
334                 dc.b $18, $0E, $14
335                 dc.b $18, $0E, $0A
336                 dc.b $C3, $10, $0A
337                 dc.b $1E, $15, $14
338                 dc.b $D0, $12, $0A
339                 dc.b $C3, $10, $0A
340                 dc.b $D2, $0F, $14
341                 dc.b $00, $00, $0A
342                 dc.b $C3, $10, $0A
343                 dc.b $D0, $12, $14
344                 dc.b $1E, $15, $14
345                 dc.b $C3, $10, $14
346                 dc.b $18, $0E, $14
347                 dc.b $18, $0E, $14
348                 dc.b $00, $00, $14
349                 dc.b $00, $00, $0A
350                 dc.b $D0, $12, $14
351                 dc.b $5F, $16, $0A
352                 dc.b $30, $1C, $14
353                 dc.b $1D, $19, $0A
354                 dc.b $5F, $16, $0A
355                 dc.b $1E, $15, $14
356                 dc.b $00, $00, $0A
357                 dc.b $C3, $10, $0A
358                 dc.b $1E, $15, $14
359                 dc.b $D0, $12, $0A
360                 dc.b $C3, $10, $0A
361                 dc.b $D2, $0F, $14
362                 dc.b $00, $00, $0A
363                 dc.b $C3, $10, $0A
364                 dc.b $D0, $12, $14
365                 dc.b $1E, $15, $14
366                 dc.b $C3, $10, $14
367                 dc.b $18, $0E, $14
368                 dc.b $18, $0E, $14
369                 dc.b $00, $00, $14
370                 dc.b $FF
371 snare:
372                 dc.b $61, $38, $0A
373                 dc.b $00, $00, $0A
374                 dc.b $FF
375 kickResonate:
376                 dc.b $2D, $01, $0A
377                 dc.b $00, $00, $05
378                 dc.b $FF
379 bridgeA:
380                 dc.b $C3, $10, $28
381                 dc.b $18, $0E, $28
382                 dc.b $4D, $0D, $14
383                 dc.b $D2, $0F, $14
384                 dc.b $1E, $15, $14
385                 dc.b $00, $00, $14
386                 dc.b $FF
387 bridge:
388                 dc.b $1E, $15, $28
389                 dc.b $C3, $10, $28
390                 dc.b $D0, $12, $28
391                 dc.b $D2, $0F, $28
392                 dc.b $FF
393 bass1:
394                 dc.b $47, $05, $0A
395                 dc.b $8F, $0A, $0A
396                 dc.b $47, $05, $0A
397                 dc.b $8F, $0A, $0A
398                 dc.b $47, $05, $0A
399                 dc.b $8F, $0A, $0A
400                 dc.b $47, $05, $0A
401                 dc.b $8F, $0A, $0A
402                 dc.b $86, $03, $0A
403                 dc.b $0C, $07, $0A
404                 dc.b $86, $03, $0A
405                 dc.b $0C, $07, $0A
406                 dc.b $86, $03, $0A
407                 dc.b $0C, $07, $0A
408                 dc.b $86, $03, $0A
409                 dc.b $0C, $07, $0A
410                 dc.b $53, $03, $0A
411                 dc.b $A6, $06, $0A
412                 dc.b $53, $03, $0A
413                 dc.b $A6, $06, $0A
414                 dc.b $53, $03, $0A
415                 dc.b $A6, $06, $0A
416                 dc.b $53, $03, $0A
417                 dc.b $A6, $06, $0A
418                 dc.b $86, $03, $0A
419                 dc.b $0C, $07, $0A
420                 dc.b $86, $03, $0A
421                 dc.b $0C, $07, $0A
422                 dc.b $86, $03, $0A
423                 dc.b $86, $03, $0A
424                 dc.b $F4, $03, $0A
425                 dc.b $30, $04, $0A
426                 dc.b $FF
427 kickHit:
428                 dc.b $2D, $01, $05
429                 dc.b $FF
430 bass2:
431                 dc.b $B4, $04, $0A
432                 dc.b $68, $09, $0A
433                 dc.b $B4, $04, $0A
434                 dc.b $68, $09, $0A
435                 dc.b $B4, $04, $0A
436                 dc.b $68, $09, $0A
437                 dc.b $B4, $04, $0A
438                 dc.b $68, $09, $0A
439                 dc.b $30, $04, $0A
440                 dc.b $61, $08, $0A
441                 dc.b $30, $04, $0A
442                 dc.b $61, $08, $0A
443                 dc.b $30, $04, $0A
444                 dc.b $61, $08, $0A
445                 dc.b $30, $04, $0A
446                 dc.b $61, $08, $0A
447                 dc.b $F4, $03, $0A
448                 dc.b $E9, $07, $0A
449                 dc.b $F4, $03, $0A
450                 dc.b $E9, $07, $0A
451                 dc.b $F4, $03, $0A
452                 dc.b $E9, $07, $0A
453                 dc.b $F4, $03, $0A
454                 dc.b $E9, $07, $0A
455                 dc.b $86, $03, $0A
456                 dc.b $0C, $07, $0A
457                 dc.b $86, $03, $0A
458                 dc.b $0C, $07, $0A
459                 dc.b $86, $03, $0A
460                 dc.b $0C, $07, $0A
461                 dc.b $86, $03, $0A
462                 dc.b $0C, $07, $0A
463                 dc.b $FF
464 bass3:
465                 dc.b $86, $03, $0A
466                 dc.b $0C, $07, $0A
467                 dc.b $86, $03, $0A
468                 dc.b $0C, $07, $0A
469                 dc.b $86, $03, $0A
470                 dc.b $0C, $07, $0A
471                 dc.b $86, $03, $0A
472                 dc.b $0C, $07, $0A
473                 dc.b $53, $03, $0A
474                 dc.b $A6, $06, $0A
475                 dc.b $53, $03, $0A
476                 dc.b $A6, $06, $0A
477                 dc.b $53, $03, $0A
478                 dc.b $A6, $06, $0A
479                 dc.b $53, $03, $0A
480                 dc.b $A6, $06, $0A
481                 dc.b $86, $03, $0A
482                 dc.b $0C, $07, $0A
483                 dc.b $86, $03, $0A
484                 dc.b $0C, $07, $0A
485                 dc.b $86, $03, $0A
486                 dc.b $0C, $07, $0A
487                 dc.b $86, $03, $0A
488                 dc.b $0C, $07, $0A
489                 dc.b $53, $03, $0A
490                 dc.b $A6, $06, $0A
491                 dc.b $53, $03, $0A
492                 dc.b $A6, $06, $0A
493                 dc.b $00, $00, $0A
494                 dc.b $A6, $06, $0A
495                 dc.b $A6, $06, $14
496                 dc.b $FF
497 bridgeB:
498                 dc.b $18, $0E, $14
499                 dc.b $1E, $15, $14
500                 dc.b $30, $1C, $28
501                 dc.b $9B, $1A, $28
502                 dc.b $00, $00, $28
503                 dc.b $FF
504
```

Figure H.2: Tetris target code example for Neptune (2/2)

Bibliography

Bibliography

[1] V. Fulber-Garcia, “Imperative and Declarative Programming Paradigms,” https://www.baeldung.com/cs/imperative-vs-declarative-programming, 2024, [Accessed 2025 May 7].

[2] R. W. Sebesta, *Concepts of Programming languages*, 11th ed. Pearson Education Limited, 2016, [Accessed 2025 Apr 11].

[3] *Commodore 64 Programmer’s Reference Guide*, https://archive.org/details/c64-programmer-ref/, Commodore Computer, 1982, [Accessed 2025 Mar 12].

[4] “LilyPond - Music notation for everyone,” https://lilypond.org/index.html, [Accessed 2025 May 8].

[5] C. Wright, “Lecture 5 - Melody: Notes, Scales, Nuts and Bolts,” https://oyc.yale.edu/music/musi-112/lecture-5#, 2012, [Accessed 2025 Feb 28].

[6] B. Benward and M. Saker, *Music in Theory and Practice*, M. Ryan, Ed. McGraw-Hill, 2008.

[7] S. Chase, “What Is Tempo In Music? A Complete Guide,” https://hellomusictheory.com/learn/tempo/, 2024, [Accessed 2025 Feb 23].

[8] A. Steen, “432 vs. 440 Hz Frequencies: A Comprehensive Examination,” https://primesound.org/432-vs-440, 2024, [Accessed 2025 Feb 21].

[9] H. Hüttel, *Transition and Trees: An Introduction to Structural Operational Semantics*. Cambridge, 2010, [Accessed 2025 May 7].

[10] *6581 Sound Interface Device (SID)*, https://archive.org/details/mos_6581_sid_preliminary_october_1982/mode/2up, Commodore Semiconductor Group, 1982, [Accessed 2025 Mar 10].

[11] M. DeVoto, “polyphony,” https://www.britannica.com/art/polyphony-music, 2023, [Accessed 2025 Feb 27].

[12] A. Price, “How to use basic ADSR filter envelope parameters,” https://www.musicradar.com/tuition/tech/how-to-use-basic-adsr-filter-envelope-parameters-578874, 2023, [Accessed 2025 Mar 12].

[13] K. B. McAlpine, *Bits and Pieces: A History of Chiptunes*. New York, NY : Oxford University Press, 2018, [Accessed 2025 Feb 23].

[14] J. Butterfield, *Machine Language for the Commodore 64, 128, and Other Commodore Computers*. Prentice Hall Press, 1986.

[15] "Remembering Computing Legend Thomas Kurtz," https://fas.dartmouth.edu/news/2024/11/remembering-computing-legend-thomas-kurtz, 2025, [Accessed 2025 May 12.

[16] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective 3rd Edition*. Pearson Education Limited, 2016, [Accessed 2025 Mar 6].

[17] "6502 Architecture," http://www.6502.org/users/obelisk/6502/architecture.html, 2002, [Accessed 2025 May 20].

[18] "First 6502 assembly program," http://forum.6502.org/viewtopic.php?t=4948, 2017, [Accessed 2025 May 20].

[19] "dasm - 8bit macro assembler," https://dasm-assembler.github.io, 2021, [Accessed 2025 May 22].

[20] L. Gondelman, "Languages and Compilation - Lecture 1 - Introduction," https://homes.cs.aau.dk/~lego/compil25/lectures/1/lecture1-pp.pdf, 2025, [Accessed 2025 May 16].

[21] L. Gondelman, "Languages and Compilation - Lecture 2 - Abstract Syntax, Semantics," https://homes.cs.aau.dk/~lego/compil25/lectures/2/lecture2-pp.pdf, 2025, [Accessed 2025 May 16].

[22] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, "The OCaml Manual," https://ocaml.org/manual/5.3/index.html#, 2025, [Accessed 2025 May 12].

[23] C. N. Fischer, R. K. Cytron, and J. Richar J. Leblanc, *Crafting a Compiler*. Pearson Education Limited, 2010, [Accessed 2025 May 5].

[24] L. Gondelmann, "Languages and Compilation - Lecture 4 - Lecture 4 - Parsing, Part 1," https://homes.cs.aau.dk/~lego/compil25/index.html, 2025, [Accessed 2025 May 12].

[25] F. Pottier, "Menhir: A LR(1) parser generator," https://gallium.inria.fr/~fpottier/menhir/, 2024, [Accessed 2025 May 12].

[26] M. Délèze, "Calculation of the frequency of the notes of the equal tempered scale," https://www.deleze.name/marcel/en/physique/musique/Frequences-en.pdf, [Accessed 2025 Feb 21].

[27] "2.3: Assembler Directives," https://eng.libretexts.org/Bookshelves/Electrical_Engineering/Electronics/Implementing_a_One_Address_CPU_in_Logisim_(Kann)/02%3A_Assembly_Language/2.03%3A_Assembler_Directives, 2023, [Accessed 2025 May 15].

[28] "File Manipulation," https://ocaml.org/docs/file-manipulation#, [Accessed 2025 May 13].

[29] "The Versatile Commodore Emulator," https://vice-emu.sourceforge.io/vice_toc.html, 2024, [Accessed 2025 May 19].

[30] K. Rana, “Unit Testing,” https://artoftesting.com/unit-testing#Best_Practices_for_Unit_Testing, 2025, [Accessed 2025 May 9.

[31] M.-M. Zeeman and S. L. Gall, “OUnit: xUnit testing framework for OCaml,” https://ocaml.org/p/ounit2/2.2.3/doc/index.html#ounit:-xunit-testing-framework-for-ocaml, [Accessed 2025 May 9.

[32] “GitHub Actions documentation,” https://docs.github.com/en/actions, [Accessed 2025 May 14].

[33] “Acceptance testing,” https://en.wikipedia.org/wiki/Acceptance_testing, 2025, [Accessed 2025 May 11.

[34] “Compiler Design - Symbol Table,” https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm, [Accessed 2025 May 21].

[35] D. Schwebel, “A Guide to Chiptune Music,” https://www.synthcentric.com/articles/a-guide-to-chiptune-music, [Accessed 2025 Feb 22].

[36] “The era of 8bit music,” https://www.retrogames-online.com/musik/, 2025, [Accessed 2025 Feb 22].

[37] A. Watley, “How Sound Chips Work: A Simple Explanation,” https://www.ac3filter.net/how-sound-chips-work/, 2023, [Accessed 2025 Feb 24].

[38] “Programmable Sound Generator,” https://www.thealmightyguru.com/Wiki/index.php?title=Programmable_sound_generator, 2025, [Accessed 2025 Feb 22].

[39] M. Aichele, “Understanding Time Signatures and Meters: A Musical Guide,” https://www.libertyparkmusic.com/musical-time-signatures/, 2018, [Accessed 2025 Feb 23].

[40] S. Leemon, *Mapping the Commodore 64 and 64C*, 1st ed. Compute!, 1987.

[41] “Understanding Sound Waves and How They Work,” https://science.howstuffworks.com/sound-info.htm#pt2, 2023, [Accessed 2025 Mar 7].

[42] J. Comeau, “Lets Learn About Waveforms,” https://pudding.cool/2018/02/waveforms/, 2024, [Accessed 2025 Feb 21].

[43] “Sine Wave,” https://mathematicalmysteries.org/sine-wave/, [Accessed 2025 Feb 23].

[44] W. Roberts, “Fourier Series,” https://w1.mtsu.edu/faculty/wroberts/teaching/fourier_4.php, [Accessed 2025 Feb 23].

[45] G. Reid, “Introduction to Additive Synthesis,” https://www.soundonsound.com/techniques/introduction-additive-synthesis, [Accessed 2025 Feb 20].

[46] E. W. Weisstein, “Fourier series,” https://mathworld.wolfram.com/FourierSeries.html, 2024, [Accessed 2025 May 16].

[47] E. W. Weisstein, “Sawtooth Wave,” https://mathworld.wolfram.com/SawtoothWave.html, [Accessed 2025 Feb 23].

[48] “MoSCoW Prioritization,” https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html, [Accessed 2025 Mar 19].