Universidade de Aveiro
2021

André da Silva Pinho

rtfss: A Hardware Description Language for Digital Audio Processing

rtfss: Uma Linguagem de Descrição de Hardware para Processamento Digital de Áudio

Universidade de Aveiro
2021

André da Silva
Pinho

rtfss: A Hardware Description Language for Digital Audio Processing

rtfss: Uma Linguagem de Descrição de Hardware para Processamento Digital de Áudio

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Tomás António Mendes Oliveira e Silva, Professor do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

Dedico este trabalho aos meus pais, Armando e Alice.

agradecimentos / acknowledgements

Esta tese não é apenas produto do meu esforço, mas também de todas as pessoas que me influenciaram de forma marcante ao longo da minha vida.

Em primeiro lugar, quero agradecer profundamente aos meus pais, Armando Pinho e Alice Pinho, por todo o apoio que me deram durante o meu percurso académico. As condições por eles proporcionadas foram o fundamento do meu sucesso nos últimos cinco anos.

Quero agradecer ao Professor Tomás Oliveira e Silva, que aceitou ajudar-me a encontrar tópico e orientar esta tese. As suas ideias e sugestões foram bastante valiosas e desafiaram-me a elevar o trabalho ao nível que se apresenta agora. Tê-lo como orientador fez com que fosse o meu primeiro e último Professor do DETI neste curso. Gostava também de estender este agradecimento a todos os outros Professores da Universidade que contribuíram para a minha formação, e aos restantes membros do DETI.

Quero agradecer à Margarida, que foi uma das primeiras pessoas deste curso que se tornou minha amiga, e que me acabou por aturar na maior parte dos trabalhos das cadeiras. Devo a ela, entre outras coisas, a minha introdução à vida académica. Posteriormente, o Diego juntou-se a nós e a Dream Team foi formada. É de notar que foi por influência musical do Diego que acabei por ter a ideia do tema desta tese. Estimo todos os momentos que passámos juntos e espero que, no futuro, possamos voltar a juntar-nos para despejar WD-40 e fita-cola numa *hackathon* qualquer.

A estes dois amigos, junto o restante grupo de amigos que fiz ao longo de ECT, com quem me diverti imenso e partilhei inúmeras, longas, e por vezes desesperadas noites no DETI a fazer trabalhos. Ao Rodrigo, Dinis, Miguel, Francisco, Fred, Guilherme e demais, reservo um grande obrigado. Vimos o nosso último ano do curso ser parado subitamente graças a um vírus, mas fica o desejo de compensar quando houver hipótese.

Quero agradecer também ao André e ao Neves, dois amigos de longo termo, que, por muito que não tenhamos tido hipótese de fazer muita coisa nos últimos tempos, valorizo a nossa amizade e o apoio que recebo de ambos.

Gostava também de agradecer aos meus Professores da Escola Mário Sacramento, especialmente à Professora Isabel Órfão, que durante o Ensino Secundário me consolidou o conhecimento e o interesse em Matemática.

Por fim, devo um agradecimento aos meus Professores do Conservatório de Música de Aveiro, com destaque ao Professor Nuno Caçote e à Professora Teresa Macedo. Estes, ajudaram-me a desenvolver a minha vertente artística e acabaram por, indiretamente, influenciar a minha escolha do tema de bastantes projetos desenvolvidos neste curso, incluindo, inevitavelmente, esta tese.

Resumo

O interesse em sintetização de áudio em circuitos digitais e sistemas computacionais remonta à época em que as linguagens de programação Lisp, COBOL e Fortran ainda estavam a ser conceptualizadas. Esta era (por volta de 1950) marca o surgimento da linguagem de programação MUSIC I. Desde então, dado o aumento do poder de computação, houve um influxo de novas linguagens capazes de sintetizar áudio. Atualmente, linguagens como SuperCollider, Pd (pure data), Max/MSP, ChucK e Faust estão no espetro das mais populares. Estas linguagens, apesar de terem diferentes abordagens e designs, têm um aspeto comum: são todas linguagens de computador. Embora este tipo de solução conceda bastante flexibilidade, ela apresenta um problema: restrições temporais, particularmente latência de áudio.

Neste documento, uma nova Linguagem de Processamento de Áudio, denominada *rtfss*, é introduzida. Esta linguagem foi concebida para ser compilada principalmente para dispositivos de *hardware*, de forma a tirar partido das suas capacidades de paralelismo. Para complementar a descrição formal da linguagem, será também apresentado um compilador capaz de lidar com um subconjunto das capacidades da linguagem. Este compilador é apto para analisar uma descrição feita em *rtfss* e de compilar para uma linguagem de nível inferior chamada VHDL. Um dos principais destaques de *rtfss* (e do seu compilador) é que a arquitetura de *hardware* sintetizada é segmentada (*pipelined*). Este formato de arquitetura digital é usado em circuitos lógicos para maximizar a frequência de funcionamento e a eficiência do design.

Ao longo deste documento, a descrição formal da linguagem e a implementação do compilador serão meticulosamente analisados e explicados. Serão também ilustrados alguns exemplos de casos de uso nesta linguagem, acompanhados pelo seu processo de compilação.

Neste documento é também descrita uma arquitetura mais simples de outra linguagem de processamento de áudio para *hardware*. Esta linguagem serviu como fundação ao desenvolvimento de *rtfss*. Do desenvolvimento desta arquitetura derivou uma coletânea de blocos de *hardware* em VHDL para tratamento de áudio e de MIDI.

Abstract

The pursuit of audio synthesis in digital circuitry and computing systems dates back to the time where programming languages such as Lisp, COBOL and Fortran were still being conceptualized. This era (the 1950s) marks the appearance of the MUSIC I programming language. Since then, and as the processing capabilities of computers kept rising, there was an influx of new languages capable of synthesizing audio. Languages such as SuperCollider, Pd (pure data), Max/MSP, ChucK and Faust are among the most popular ones, nowadays. These languages, although having different approaches and designs, have one aspect in common: they are all computer languages. Albeit these solutions grant great flexibility, they present an issue: timing constraints, particularly audio latency.

In this document, a new Audio Processing Language, called *rtfss*, is introduced. This language was designed to be compiled mainly to hardware targets, exploiting their inherent parallelism. To complement the formal description of the language, a compiler that implements a subset of the language's capabilities will also be presented. This compiler is capable of analysing a design description written in *rtfss* and compile it to the lower level hardware description language VHDL. One of the main highlights of *rtfss* (and its compiler) is that the hardware architecture it synthesizes is pipelined. This type of digital architecture is used in logic designs to maximize the working frequency and the efficiency of the design.

Throughout this document, both the language's formal specification and the compiler's implementation will be thoroughly analysed and explained. Some use-case examples of this language, accompanied by their compilation procedure, will also be illustrated and examined.

In this document, it is also described a simpler architecture of another Hardware Audio Processing Language, which served as a stepping stone towards the development of *rtfss*. As a product of its development, some useful VHDL Audio (and MIDI 1.0) related hardware blocks were implemented.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

After the inception and widespread of reasonably powerful personal computers, digital sound processing has become the industry standard for audio creation, manipulation and production. The most common platforms for digital audio are the Digital Audio Workstations (DAW) [4]. In DAWs, artists and audio engineers can record, process and produce audio. DAWs are the center piece of music production in most music studios, and are, arguably, the most popular kind of digital audio tool.

Another set of tools for sound processing that had an increase in popularity are the audio programming languages. These tools first appeared way before the inception of DAWs (the first commercial DAW was released in 1978 [4]). The earliest audio computer language was created in 1957 and was called MUSIC I [5]. From then on, more audio programming languages with diverse approaches were created. Some of these languages, such as Pd or Max/MSP, are visual languages where the programmer draws the block diagram of the audio processor architecture. Others like SuperCollider, ChucK or Faust, are text-based and are built upon the functional programming paradigm [5]. Some of these languages will be described more thoroughly in Chapter 2.

Audio processing systems have one particularity that separates them from most general-purpose systems: they must work in real-time. These systems have to be particularly precise with their timing. They work with fixed timing deadlines to perform the calculations needed to generate the new audio values (audio samples) [5]. This is a problem when working with software-based audio processing systems. Most of the environments in which these softwares run are multitasking scheduled operating systems. In these systems, the CPU time has to be split with all the processes it is running [6]. Having this environment means that the audio processing software has no guarantee of when it will be executed, and for what amount of time. To try to smooth the effect of this behaviour, audio software processes audio in chunks (buffers) [2] and attempts to increase its scheduling priority. When the software is scheduled in, it processes all the samples present on a pre-determined buffer and places the values in another buffer. The existence of buffers means that the audio processed by the software will have a fixed amount of latency (given by the size, in samples, of the buffer), which inherently can hurt the usability of this solution. However, if the software is fast enough or the scheduler gave enough time to the process, then the audio processor does not runs into problems and the output is generated within the set deadlines. However, if the software is not able to process the full buffer, issues arise. The deadlines still need to be fulfilled, so the system outputs phony samples. One way to solve this is to increase the buffer sizes, further smoothing out the effects of the scheduler. However, increasing the buffer size will yet again deteriorate the responsiveness of the system. The latency will increase and the system might not be suitable for real-time applications anymore. Studies found that audio latencies over 10 milliseconds with jitter on digital instruments, in some applications, is enough to start being noticeable [7].

1.1 Motivation

Aware of these issues, in this document we propose a different approach to digital audio processing. Instead of trying to find different ways to optimize audio processing algorithms, in order to reduce buffer sizes on real-time audio software processing, our solution theoretically does not even need to consider this kind of problems. We propose a digital hardware-based approach for audio processing. Hardware audio solutions do not suffer from the problems that multitasking brings to software audio solutions, since they are designed to be application-specific. Hardware generated for this purpose is only focused on generating audio, so it is a much more optimized system. Furthermore, hardware solutions grant complete control over the timing of the system, making it desirable for time-critical environments [1].

While audio processing in digital hardware is no innovation, the creation of these systems is not trivial. Designing algorithms in digital hardware is much more demanding (and poses very different challenges) than implementing the same algorithm in software. Moreover, the skill set for developing hardware is relatively distant from the one for developing software. All these characteristics make hardware solutions less favourable to implement [8].

So, to simplify the creation of hardware solutions for audio processing, in this thesis a new audio processing language for digital hardware is proposed. This language is called *rtfss* (Real-Time FPGA Sound Synthesis). The language was created to generate optimized hardware based on a high-level description of audio algorithms. This language is text-based and has functional programming characteristics. The objective of *rtfss* is to behave as much as possible as any other audio programming language, while being able to be synthesized for hardware.

The *rtfss* language allows manipulation of the sample values of sound streams. These sound streams are controlled by a pulse signal that sets the sampling frequency of that audio path. Streams have direct access over their previous values by using a special type of indexing. This characteristic makes the implementation of time-based sound processors (like digital filters) very easy and straight-forward. Another very useful feature of *rtfss* is the built-in support for fixed-point arithmetic (all the conversions are ensured by the language). The language also allows the interaction of multiple audio sources that work at different pulse rates (sample frequencies). Finally, it is also worth noting that *rtfss* is able to handle the MIDI protocol (explained in Section 2.4) and interact with it.

Along with the *rtfss* language, a compiler that implements a subset of *rtfss* was also created. It was implemented in C++, using ANTLR4 for language recognition. It is capable of generating hardware designs using VHDL. Both the *rtfss* language and compiler will be explained further in this document. To serve as an example of the potential of this language, the following segment illustrates the implementation of an Infinite Impulse Response (IIR) filter (explained in Appendix A):

```
cblock@smpp main(: I16 smp_in : I16 lp_out){
    //Low-pass aprox 0.2 Normalized Frequency IIR Filter
    I16.2@smpp lowpass = I16.2((lowpass'-1)>>1+(smp_in+(smp_in'-1))>>2);
    lp_out=I16(lowpass);
}
```

This example creates an audio hardware device that has one input integer (audio) stream (`smp_in`) and one integer (output) audio stream (`lp_out`). Both audio streams are 16 bit integer values (`I16`) and are controlled by the pulse `smpp`. Internally, the hardware block (`cblock` called `main`) calculates the audio stream values with the fixed-point precision of 16 integer bits and 2 fractional bits (`I16.2`) on a audio stream called `lowpass`. This stream is also controlled by the pulse `smpp`. The result of this internal operation is exposed to the output by casting the sizes of the operation down to the size of the output. The mathematical formula that this system implements is

$$y(n) = 0.5\ y(n-1) + 0.25\ (x(n) + x(n-1)).$$

The multiplications made in that formula were replaced (for convenience) by bit shifts (`>>1` and `>>2`). The recursive (time) relation present on the expression is easily solved in *rtfss* with the use of the "value at instance" operator (`'`). This operator allows the direct access of previous sample values (`lowpass'-1` and `smp_in'-1`). In this example, every stream is controlled by the same synchronization pulse (`smpp`). This pulse, in reality, can be seen as the sampling frequency of the system.

1.2 Project Evolution

During the planning and development of this thesis, the project's objectives shifted quite significantly. The original thesis' vision was to develop a musician oriented visual language with similar characteristics to Pd or Max/MSP (explained in Section 2.5.2). The innovation of the proposed version would be the possibility to compile that visual language into a hardware target. To do so, we envisioned the following architecture (presented from the bottom to the top):

- The existence of a palette of hardware blocks (in VHDL) where each would serve a simple and defined purpose. These blocks would be parameterizable, to allow their input and output sizes to be adaptable to specific situations;
- The creation of a high-level language, whose objective would be to specify audio signal processing algorithms. Every operation of this language would have a one-to-one relation to an already existent hardware block. The aim of the compiler of this language would be only to create a top-level VHDL entity, that would glue the existent VHDL blocks together to implement the algorithms specified in the language;
- The creation of a graphical language and interface similar to Pd, that would allow the visual representation of audio signal processing algorithms. This graphical program would be able to generate code in the language specified in the previous item.

This architecture was used through a significant part of the development of the thesis, but we then reached the conclusion that it had several drawbacks. Firstly, the compiler and the language offered no valuable flexibility. They would be oblivious to what intrinsic logic they would be generating. Another issue was hardware timing constraints. The way that the compiler would synthesize hardware would imply that the timing characteristics of the hardware would be hidden away inside each of the pre-built blocks. This leads to another problem: optimization. The hardware generated would be highly unoptimized, both in terms of the amount of logic produced and the maximum frequency, it would allow being run at. Furthermore, having a fixed set of blocks (like it happens in some other computer languages [5]) would seriously limit the algorithmic possibilities of the language. Having all of this in mind, it was decided to change the target of the project.

The updated (and final) architecture of the project shifts the focus into only the second part of the previous architecture: the language and compiler. This means that the step of the creation of a high-level language/interface and the creation of the VHDL hardware blocks are not needed anymore. But, before the architecture was pivoted, several hardware blocks were already created. Although these blocks are not (directly) used by the new architecture, they are still useful VHDL blocks for audio processing and MIDI handling. For this reason, they will be presented in Chapter 5.

1.3 Contributions

With this project, we intended to enrich the collection of languages for audio synthesis and processing. The new language proposed has the advantage of offering an unusual, although important, compilation target: digital hardware. As will be explained throughout this document, both the language and compiler proposed, due to their complexity, are still evolving. We intend to proceed with the development of this project, in order to achieve a mature enough status that would grant its use outside the academic context. This project can also be used as groundwork to other more advanced languages that pursuit similar objectives.

1.4 Document Outline

The rest of the thesis is organized into the following parts:

- Chapter 2 gives an overview of some topics and work related to this thesis. In it, the technologies of FPGA, ANTLR and MIDI are explained. Then, some computer music languages (SuperCollider, Pd, Max/MSP, ChucK and Faust) are described;
- Next, in Chapter 3, the *rtfss* language is explained and its formal definition is described. To assist the definition of the language, some snippets of code will be exemplified;
- Then, in Chapter 4, the *rtfss* language's compiler is presented. In this chapter, the whole compilation process is thoroughly delineated. Also, to assist the explanation, some diagrams representing each of the steps are presented;
- In Chapter 5, some VHDL hardware blocks that resulted from the exploration of the alternative project architecture are enumerated and explained. These blocks can be used to process and create audio in hardware, and to handle the MIDI protocol;
- Finally, Chapter 6 gives some concluding remarks over the project and its possibilities for future work.

Chapter 2

Background and Related Work

In this Chapter, we introduce some concepts and technologies that had an impact on the implementation of the project. We also present some related work that has similarities with the project described in this document. Some of the related work served as inspiration for the idealization of this project.

2.1 FPGA Device

An FPGA (Field-Programmable Gate Array) is a digital hardware device that contains reprogrammable logic [1]. It is made of an array of programmable logic blocks that can be configured to implement any digital logic. FPGAs come in the form of integrated circuits. They are very useful to prototype hardware designs since they can be reprogrammed with orders of magnitude less effort than a normal fixed logic chip can be produced. A big advantage of FPGAs is that they can be tailored to fit specific applications, where they replace software. Since in hardware the designer has full control on the circuit, (in a significant amount of cases) hardware solutions become much more efficient, comparing to the software solutions [9]. The simplified typical architecture of an FPGA is illustrated in Figure 2.1. In this Figure, the `LB` blocks are programmable logic blocks (PLB). These blocks usually contain a configurable look-up table (used to implement combinational logic) and an optional flip-flop (allows the creation of sequential logic). The `SW` blocks are the programmable interconnections (switches). These allow the routing of the inputs and outputs of the PLB to each other or to the input/output pins. The black squares represent the input/output pins of the FPGA.

FPGAs require specific languages and compiler tool-chains to be programmed. To be precise, the best way to describe the process is to design, not to program. The most common hardware description languages are VHDL and Verilog [1]. These languages have one common paradigm: concurrency. Most of the instructions on these languages are "executed" at the same time. In reality, they are parallelized in the hardware. This is a very different mindset compared to normal software programming languages. Hardware description languages offer little complexity abstraction from the circuitry they generate. Some of these languages allow some hardware abstraction, to the point that they behave similarly to sequential programming languages (VHDL, for example). But, the use of these features is mostly frowned upon, since they handicap the usefulness of hardware design.

FPGAs were used in this project in both architectures. They are used as the target hardware of both the hardware blocks designed for the initial architecture and as the design

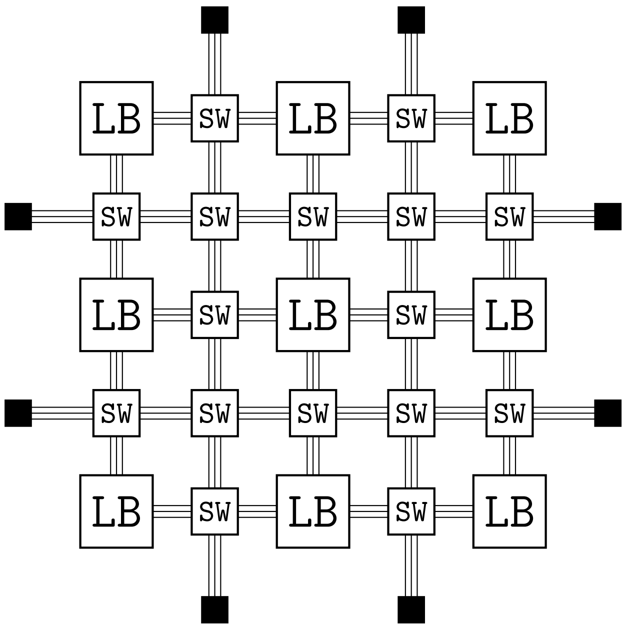


Figure 2.1: Simplified diagram of the architecture of a FPGA [1].

target device of the *rtfss* compiler. The FPGA development board used in this project is the DE2-115 board by Terasic. This FPGA development board contains the FPGA chip Cyclone IV EP4CE115F29, made by Altera (now owned by and called Intel). Along with the FPGAs, Altera also provides an Integrated Development Environment (IDE), called Altera Quartus Prime. This IDE contains a compiler tool-chain that accepts languages such as VHDL and Verilog.

2.2 VHDL Language

VHDL (VHSIC [Very High Speed Integrated Circuit] Hardware Description Language) is a hardware description language. Among other uses (such as simulation of logic designs), it can be used to develop hardware designs for FPGAs. As mentioned before, it is based in the concurrent computing paradigm [10]. This means that the instructions on this language are meant to be fulfilled at the same time.

In VHDL, hardware blocks are represented with entities (`entity`). An entity has a name, a set of input and output ports and constant parameters (that can have default values). The constant parameters enable the possibility of creating parameterizable entities. An entity also has a set of at least one architecture (`architecture`). While the entity acts as the interface of the hardware block, an architecture corresponds to the implementation of the logic that controls the data flows through the ports. Normally, an entity only has one architecture, but, in some cases, different realizations over the same interface can be useful. To instantiate an entity, its name, the input ports, output ports (optionally), generic parameters (optional to the parameters which have a default value) and the name of the desired architecture have to be specified.

The two main datatypes of VHDL are `std_logic` and `std_logic_vector`. The `std_logic` type represents a logic value (can be 1,0 or other values such as the high impedance state), while the `std_logic_vector` is a vector of `std_logic`. This vector has to have a defined size at compile-time. There are also two numeric types in VHDL: `signed` and `unsigned`. These allow the direct use of arithmetic operations over them [10].

The VHDL language was used in this project in both architectures. Firstly, it was used to create the hardware blocks that would serve as the building blocks for the compiler to construct with. This language is also used by the *rtfss* compiler, in order to describe the logic of the compiled design. The language used in this project obeys the *VHDL-2008* standard.

2.3 ANTLR Parser Generator

ANTLRv4 (or ANTLR4) is the fourth iteration over a lexer (lexical analyser) and parser (syntactic analyser) generator tool. For the sake of simplicity, ANTLRv4 will be called just ANTLR. ANTLR is a tool that eases the segmentation and organization of structured data. ANTLR consumes a formal language specification and, with it, generates code capable of analysing data that follows the described formal language specification. The formal language specification shall be referred to as *grammar* from now on. ANTLR can accept all possible grammars, with the exception of the ones that contain indirect left recursion [11]. The parsing mechanism used by ANTLR was custom created and is an improved version of the *LL(*)* (Left to right with Leftmost derivation) mechanism: *Adaptive LL(*)* [12].

The code generated by ANTLR analyses its input, generates a token stream and then generates a tree representation. This representation is called a *parse tree*. Although ANTLR's main target is Java, it allows the generation of, for example, Python and C++.

ANTLR is used in this project on the new architecture. It is used by the *rtfss* compiler to interpret the *rtfss* language. To do so, the *rtfss* was fully described as a grammar. This grammar was fed into the ANTLR tool, and since the compiler is implemented in C++, the target language of ANTLR was also set to C++. ANTLR was also expected to be used in the initial architecture to aid the implementation of its compiler. However, that architecture was rethought before the development of the compiler started.

The use of Bison and Flex was considered, and tested. Bison is a parser generator that converts grammars into parser tables [13]. Flex is a lexical analyser that is usually coupled with Bison to perform complete language recognition. However, the advantages of their use didn't make up for the increased implementation cost.

2.4 MIDI Protocol

MIDI (Musical Instrument Digital Interface) is a data communication protocol targeted at audio and sound devices. It is capable of transmitting, among other kinds of data, musical performance data in a very compact and efficient manner [3]. Musical performance data is, for example, the note and velocity (how hard the note was pressed) of the keystrokes on a piano keyboard or the change of position of a slider (controller). MIDI is an industry-wide standard protocol made by several companies and can be found in the majority of musical gear and software. Aside from the data communication protocol, MIDI also standardizes the physical connectors and cables that allow multiple MIDI-enabled devices to communicate physically [2]. This allows the communication, for example, of a MIDI keyboard to a synthesizer rack unit that does not contain a keyboard.

To exchange information, MIDI has a set of messages that express state updates of a MIDI device. These messages are specified in a binary fashion (organized in bytes) and can have variable size. There are two kinds of messages: Channel Messages and System Messages. Channel messages are used to transmit the musical performance data. In a certain MIDI communication, these data are transferred over channels. A channel is usually seen as a source device for those events. There are sixteen possible channel identifiers on a channel message. A System Message is a message that affects the complete MIDI ecosystem (like song selection). It does not contain channel information or number.

Every byte is also marked on the most significant bit if it is a status byte or a data byte. When the byte is a status byte, then the most significant bit is set to one. When the bit is set to zero, then the byte is a data byte. This marking allows the detection of errors and the resynchronization of the MIDI receiver.

System messages are not relevant on the context of this document, so every message discussed here will be a Channel Message (and thus have a channel number). The MIDI messages that are relevant to this document are the MIDI Voice Channel Messages. These are composed by a status byte followed by one or two data bytes [3]. Table 2.1 contains the binary representation of MIDI Voice Channel Messages. The byte sending order, according to the order used in Table 2.1, is from left to right. The meaning of each of the messages indicated in Table 2.1 is:

- Note On Message: This message is sent when a key is pressed on a MIDI device. This message contains note number and velocity value. The `nnnn` represents the channel number, the `kkkkkkk` represents the note number and the `vvvvvvv` represents the velocity value;

- Note Off Message: This message is sent when a key is released on a MIDI device. This message contains note number and velocity value. The `nnnn` represents the channel number, the `kkkkkkk` represents the note number and the `vvvvvvv` represents the velocity value;

- Polyphonic Key Pressure Message: This message is sent when the pressure (after-touch) of a certain pressed key changes. This message contains note number and the new after-touch value. The `nnnn` represents the channel number, the `kkkkkkk` represents the note number and the `vvvvvvv` represents the pressure value;

- Control Change Message: This message is sent when a controller (for example a slider, pedal or switch) changes value on a MIDI device. This message contains the controller number and the new value. The `nnnn` represents the channel number, the `ccccccc` represents the controller number and the `vvvvvvv` represents the controller value;

- Program Change Message: This message is sent to a MIDI device to request the change of its current program (patch, for example) to a new one. This message contains only the number of the new program. The `nnnn` represents the channel number and the `ppppppp` represents the program number;

- Channel Pressure Message: This message is sent when the pressure (after-touch) of a channel changes. This message contains only the new value of the channel pressure. The `nnnn` represents the channel number and the `vvvvvvv` represents the channel pressure value;

- Pitch Bend Change Message: This message is sent when the pitch bend value of a certain channel changes. This message contains only the new pitch bend value. The `nnnn` represents the channel number, the `lllllll` represents the least significant bits of the pitch bend value and the `mmmmmmm` represents the most significant bits of the pitch bend value. Together, `mmmmmmmlllllll` represent the pitch bend value.

Type	Status Byte	Data Byte 0	Data Byte 1
Note On	`1000nnnn`	`0kkkkkkk`	`0vvvvvvv`
Note Off	`1001nnnn`	`0kkkkkkk`	`0vvvvvvv`
Polyphonic Pressure	`1010nnnn`	`0kkkkkkk`	`0vvvvvvv`
Control Change	`1011nnnn`	`0ccccccc`	`0vvvvvvv`
Program Change	`1100nnnn`	`0ppppppp`	
Channel Pressure	`1101nnnn`	`0vvvvvvv`	
Pitch Bend Change	`1110nnnn`	`0lllllll`	`0mmmmmmm`

Table 2.1: Binary representation of MIDI Voice Channel Messages [3].

Both the note number and velocity properties take up seven bits (ranges from 0 to 127). The note number property is a direct correspondence to a musical note. The difference between two consecutive note numbers is of one semitone. The central C note (C4) has the note number 60. So, since the A4 is 9 semitones away from the C4, the note number of A4 is 69. This representation covers all the notes present on a piano, with notes to spare [3]. The velocity property represents the intensity of the note it is associated with. The lower the number is, the softer the note was played. So, zero velocity represents silence, velocity 1 represents *pianississimo* and velocity 127 represents the the loudest value (*fortississimo*). Pitch bend values take up fourteen bits (ranges from 0 to 16383). Since the pitch bend can go up and down, the center value (no pitch bend) is (as close as possible to) the half value of the representation: 8192.

Recently, in February of 2020, the MIDI Manufacturers Association (along with the Association of Musical Electronics Industry) announced the new standard *MIDI 2.0*. This new standard brings great improvements to the first edition of the standard. Two of the biggest differences are the increase of the resolution of the control values and the possibility of having two-way MIDI communication (*MIDI 1.0* only allows uni-directional communication per cable/conversation) [14]. Unfortunately, this version of MIDI is not used in this project, for two main reasons. First, this project was already being developed at the time of publication of the new standard. So, porting the project to the new standard would require deep modifications of already built modules, and a restudy of the protocol to understand all the ways the new version would benefit the project. The second reason is the lack of software and hardware devices that already implement the new standard. This use of *MIDI 2.0* would complicate the testing and verification process of the project, since we could not rely on current audio gear. So, the MIDI standard version used in this document is *MIDI 1.0*. From this point on, every reference in this document to MIDI is referring to version *MIDI 1.0*.

The MIDI Protocol is used in both architectures, for similar purposes. The initial architecture has a set of hardware blocks that interact and interface with the MIDI protocol. These blocks were made to then control the rest of the audio hardware blocks they were connected to. On the new architecture, MIDI is integrated with the *rtfss* language, and also allows direct manipulation and the freedom to control any parameter with it.

2.5 Computer Music Languages

Computer Music Languages are a niche subset of programming languages that are oriented at audio synthesis. These languages are specifically designed to allow the direct description of sound signal processing algorithms. The first appearance of an audio programming language dates back to 1957, with the creation of Music I [5]. Since then, there were created a significant number of languages that offer different features and have different foreseeable uses. Audio synthesis has some interesting properties that lead to different language paradigms, compared to general-purpose programming languages. Arguably, the most important aspect of audio synthesis is the requirement of real-time processing. Live audio signal processors need to have the capability of having round-trip processing times with latencies in the order of a few tenths of the millisecond. One study found that, as the latency is increased on digital instruments, musicians struggled to keep a regular pace and even attempted to play ahead of time to compensate for the latency of the system [15]. Even worse than that, another study predicts that the ideal round-trip delay (between action on the instrument and response of the instrument) should be 10 milliseconds with no jitter [7]. Even when the systems are not real-time, they need to control when the data shall be delivered, in order to oblige with the right timing. This kind of restrictions requires special care in the design choices of computer music languages.

In this Section, some of the most popular computer music languages will be summarized. These languages will be presented in chronological order.

2.5.1 SuperCollider

The SuperCollider is a bundle that combines an audio synthesis language and an execution environment [5]. It first appeared in 1996 and is still in active development. SuperCollider was designed to allow real-time synthesis. In terms of architecture, SuperCollider separates the synthesis engine (*scsynth*) from the control procedure (compositional language *sclang*). These two reside in separate processes that use a messaging system to communicate [16]. Having this separation allows SuperCollider to give much higher execution priority to the synthesis engine than to the control procedure. This kind of architecture gives the overall system better latencies compared to single process monolithic architectures. With this architecture, it is also possible to change what the synthesis engine is doing live. Every time the programmer wants to add a new piece of code to the synthesis engine, the *sclang* process has to compile it first and then send it to the *scsynth*.

The *sclang* is a functional object-oriented programming language. This means that the sound algorithms are mapped into classes and objects. Every instruction in SuperCollider is terminated by a semicolon (`;`). The following code is an example of a program written in SuperCollider (adapted from [17]):

```
{SinOsc.ar(SinOsc.kr(8,mul:50)+LFNoise0.kr(1).range(500, 1500)+1000)}.play;
```

This code contains one code block (surrounded by curly brackets) that is played on instantiation (`.play`). It spawns two sine wave oscillators (`SinOsc`) and a (pseudo-random) noise source (`LFNoise0`). Each of these blocks is accessed using `.ar` or `.kr`. These two messages (modifiers) set the blocks to audio rate (`ar`) or control rate (`kr`). Audio rate units are evaluated at the sampling frequency of the system. Control rate units are evaluated less frequently to, again, give priority to more important blocks. The first sine oscillator generates a tone that is frequency controlled by the two other blocks. The frequency of the second sine oscillator is 8Hz and its output (that normally is between -1 and 1) is multiplied by 50 (`mul:50`), so it goes from -50 and 50. This oscillator creates a vibrato effect on the first oscillator. Then, the noise source generates values at 1Hz (set on the argument of `kr`). The output of this block is then scaled to have the range of 500 to 1500 (`range(500, 1500))`). This block changes the perceived pitch every second. Finally, the final frequency value is shifted up 1000Hz (done by adding 1000). The result of this patch is an ominous sound. Another way to write the same program is:

```
{SinOsc.ar(SinOsc.kr(8,mul:50,add:1000)+LFNoise0.kr(1,500,1000))}.play;
```

The argument order of the `LFNoise0.kr` generator is frequency (`freq`), multiplication value (`mul`) and addition value (`add`).

SuperCollider is open-source and still maintained to the date of redaction of this document. Thanks to its properties, SuperCollider can be used by musicians to live code synthesizers and sound processors.

2.5.2 Pd and Max/MSP

The Pd (pure data) and Max/MSP are visual computer music languages [5]. These languages were developed in the late 1990s. The creator of Pd worked on Max (the predecessor of MAX/MSP), so both languages have a similar mindset. While Pd is an open-source project, MAX/MSP is a commercial solution. Both languages offer a graphical user interface (GUI) where the programs should be designed in. Due to the commercial nature of Max/MSP, our focus will be on Pd.

In these languages, the creation of audio processing programs is made by connecting various pre-built modules together to generate more complex logic [18]. In these visual languages, the designs elaborated are called patches. These patches are composed by a mixture of objects, message boxes and number boxes [19]. Patches can be changed while they are being executed. Objects behave like functions or blocks that have certain inputs or outputs. Similar to SuperCollider (Section 2.5.1), Pd also has the separation between audio rate processing and control rate processing. Objects working at audio rate generate values at the sample rate frequency. Objects working at the control rate only generate values when there is a trigger. Objects that run at audio rate must have a tilde (~) after the name of the object. These objects can also have an argument list that proceeds the name of the object. Messages (and message boxes) are used to pass information between objects and also serve as triggers (bangs). Number boxes serve as the simplest way to store values. These can then be used to do calculations or to serve as inputs to objects.

To provide an example in Pd, the example made from the SuperCollider (Section 2.5.1) was replicated in this language. Since Pd is a visual language, the patch is on Figure 2.2. The object blocks are the completely rectangular blocks. Number boxes are the rectangular boxes with the cut on the top right corner. The other boxes (rectangle with a dent on the right) are message boxes. The audio-rate paths have bold lines, while control rate paths are marked with regular lines. The inputs of the blocks are marked on the top of the blocks and the outputs are marked at the bottom.

The objects present in the example (Figure 2.2) are [19]:

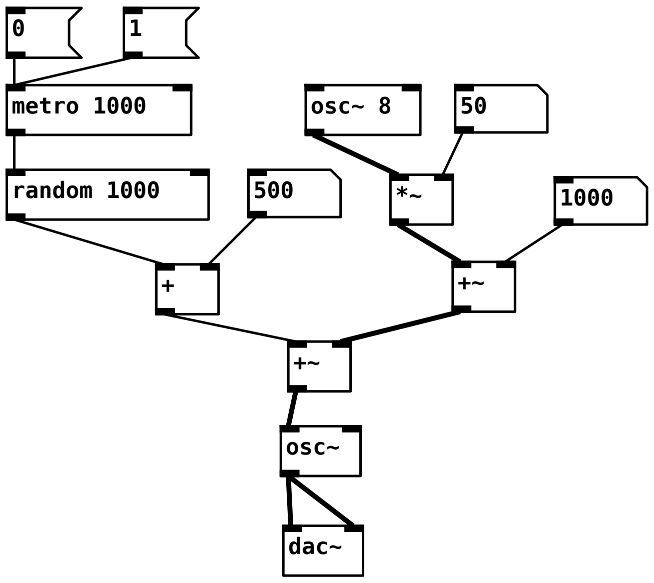


Figure 2.2: Example of an Audio Signal Patch in Pd for a frequency modulated sine wave.

- `dac`: Represents the DAC (Digital to Analog Converter) of the computing system. Sample values that are routed to the inputs of this object will be redirected to the audio card of the computer. The left input corresponds to the left channel on the audio card and the right input to the right channel;

- `osc`: Represents a cosine oscillator. This block can have an argument that sets the default frequency of oscillation. Another way to set the oscillation frequency is by connecting the left input to a number source. This number source, when the block is given no default frequency, must be in audio rate. Finally, the only output of this object is the sample value of the cosine;

- `+` and `*`: Represent, respectively, a sum and a multiplication. These objects allow the use of arithmetic operations to manipulate numbers. The left input of the object corresponds to the left operand of the binary operation, and the right input corresponds to the right operand of the binary operation. Naturally, the only output of these objects is the result of their operation;

- `random`: Represents a pseudo-random integer generator. This object generates values between zero and the value of the first argument minus one. Even if this argument is not supplied, the object also relies on the numeric value present on the right input to set the range. In order to generate a new random value, the object waits for a bang (trigger) on the left input. Also through this input, it is possible to set a seed value for the random number generator. This is done by sending the message `seed <seed_value>`. As expected, the only output of this block is the generated random value;

- `metro`: Represents a metronome. This object is capable of generating bangs at regular intervals. These bangs can then be used to control other objects. The object accepts as an argument the delay between bangs (period of the trigger). This value can also be set by sending a numeric value to the right input of the object. On the left input, the object expects a bang or a message with a nonzero message (or bang message) in order to start the metronome. If the object receives a zero message (or a stop message) on this input, then the metronome stops. The only output of this block is the bangs it generates.

Like in the SuperCollider example (Section 2.5.1), the main architecture consists of two (co)sine oscillators and a random source of values. The first oscillator is frequency controlled by the sum of values from three sources. First, the second (co)sine oscillator oscillates at a fixed frequency of 8Hz, that is amplified fifty times. This oscillator gives a vibrato (8Hz oscillation that increases and decreases the center frequency by 50Hz). The second source is a random value source. This source is created by getting values from a `random` object that is controlled by a `metro` object. This value is scaled to fit the desired range. The `metro` object is controlled by the message blocks `0` and `1`. Finally, the last source is a constant value of 1000, that shifts all base frequency to 1000Hz. The value of the first oscillator is piped to both inputs of the `dac` object. This way, the output sound is stereo.

These languages can be used by musicians to produce sounds and generative music. Both languages are still popular among artists and are even used in video games to generate procedural audio [20]. The Max/MSP language is still in active development and has also received integration (called *Max for Live*) with a Digital Audio Workstation (DAW) called Ableton Live. Pd is open-source and still is maintained to the date of redaction of this document.

2.5.3 ChucK

The ChucK language is a strongly timed audio computer language [5]. It first appeared in 2002 and is a text-based language that natively supports concurrency. ChucK was initially designed to be a live coding language. One of the most important operators in this language is the ChucK operator (`=>`). This operator is mainly used to patch unit generator (modules) together [21]. The main statements (constructs) of this language are called *shred*. A *shred* is a thread module that runs within the ChucK environment. Each *shred* should be used to generate sound or sample data. In order to control time, ChucK has two built-in variable types: `dur` and `time`. The `dur` variables store durations (relative time). The `time` variables store absolute time (in relation to the start of the ChucK environment). Assigning a `dur` into a `time` variable (using the ChucK operator), advances the `time` variable by the quantity present on the `dur` variable. The master synchronization element of ChucK is `now`. The keyword `now` is a variable of type `time`. To control time, one can assign (ChucK operator) the `now` variable with a duration.

To clarify the syntax of the ChucK language, the previously used example of the frequency modulated sine wave was translated to ChucK:

```
SinOsc u => Gain ug => SinOsc s => dac;
2=>s.sync;
8=>u.freq;
50=>ug.gain;
while(true){
    Math.random2(500,1500) + 1000 => s.freq;
    1::second => now;
}
```

The first line of the example creates the main data-flow of the program. A sine oscillator (`SinOsc` unit generator) is patched to a gain stage, which is then patched to another sine oscillator. This last sine oscillator is patched to the output audio device (`dac`). The second sine oscillator (that is going to be modulated) has the property `sync` set to 2. This property makes this oscillator interpret its input as a frequency modulation. The other valid values for the `sync` property are 0 which sets the oscillator to sync the frequency to the input or 1 which sets the oscillator to sync the phase to the input. Then, the frequency of the first oscillator is set to 8Hz (`8=>u.freq`) and the gain of the gain stage is set to 50 (`50=>ug.gain`). Finally, the program enters an infinite loop where a random value between 500 and 1500 is fetched (`Math.random2(500,1500)`) and added to 1000. The resulting value is then used to set the base frequency of the second oscillator. At the end of every iteration, the program waits one second by attributing the duration (`dur`) `1::second` to the `time` variable `now`. This infinite loop, in reality, refreshes the random value of the frequency of the modulated sine oscillator every second.

ChucK is an open-source project and still is maintained to the date of redaction of this document. The current version of ChucK (1.4.0.1) was released in April 2020 [22].

2.5.4 Faust

The Faust (Functional AUdio STream) programming language is a text-based language for audio processing [5]. It first appeared in 2002 and, like SuperCollider (Section 2.5.1), uses functional programming (not object-oriented) to design digital audio processors. The Faust project also contains a vast collection of compilation targets, and is an open-source effort.

While the previous four languages (especially the first three) gave some focus to music creation, this one is more focused in sound processing. As such, this language is statically compiled and the programs created cannot be changed while they are being executed. Also, opposed to what happens in the other languages, Faust does not distinguish between audio rate and control rates. Moreover, while previous languages rely on pre-built blocks or functions, Faust gives all the elementary operations that can be used to create and sculpt sound. Faust presents an unusual syntax, comparing to normal functional programming languages [23]. This syntax is used to represent a block diagram of the system. It uses an algebra called *block-diagram algebra*, that has the following operators:

- Sequential Composition (`A : B`): This operator denotes the connection of the outputs of block `A` to the inputs of block `B`. This new block, generated by the operation, has as inputs the inputs of `A` and as outputs the outputs of `B`;

- Parallel Composition (`A , B`): This operator denotes that the blocks `A` and `B` should be placed one in top of the other (not connected to each other). The inputs of the block that is generated by the operation are the inputs of both `A` and `B`, and the outputs of the block are all the outputs of `A` and `B`;

- Split Composition (`A <: B`): This operator denotes that the outputs of the block `A` should be distributed (and duplicated) to the inputs of block `B`. This operator shall only be used when the number of inputs of block `B` is a multiple of the number of outputs of block `A`. This new block, generated by the operation, has as inputs the inputs of `A` and as outputs the outputs of `B`;

- Merge Composition (`A :> B`): This operator denotes that the outputs of block `A` should be merged down to the inputs of block `B`. This operator shall only be used when the number of outputs of block `A` is a multiple of the number of inputs of block `B`. This new block, generated by the operation, has as inputs the inputs of `A` and as outputs the outputs of `B`;

- Recursive Composition (`A ~ B`): This operator denotes that block `A` should have a feedback loop that passes through block `B`. The inputs of block `B` should be connected to corresponding outputs of block `A`. Similarly, the outputs of block `B` should be connected to the respective inputs of block `A`. The inputs of the resulting block are the inputs of block `A` that were not connected by block `B`. The outputs of the resulting block are all the outputs of block `A`;

This algebra can be seen as the text representation of the block structure used, for example, in Pd (Section 2.5.2). This algebra contains two more elements: the identity block (_) and the cut block (`!`). The identity is a block that contains only one input and one output. In this block, the input is directly wired to the output. The cut block is a block that has only one input and no outputs. It is used to end connections. To exemplify a swap of connections between block `A` and `B`, the following syntax can be used [23]:

```
A : ((_,_) <: (!,_,_,!)) : B
```

This expression grabs two input channels from block `A`, splits them into four channels, and ignores the top and bottom one. This makes a two output block that has the outputs swapped. This block is then directly connected to block `B`.

Every instruction in Faust is terminated by a semicolon (`;`). Like most functional programming languages, Faust has a top-level *main*-like entity. In Faust, this entity is called `process` [24]. So, to create a program that simply reads the sound card's input channels, swaps them, and pipes them to the output channel, we have

```
process = (_,_) <: (!,_,_,!);
```

Faust provides a standard library with a vast collection of blocks (`stdfaust.lib`). The use of these blocks raises the abstraction level of the language to a level similar to that of SuperColider (Section 2.5.1) or ChucK (Section 2.5.3). Faust is capable of drawing diagrams of programs. The diagram from that program is in Figure 2.3. The same example provided in the three previous cases was converted to Faust. The Faust code to that example, implemented in the most *block-diagram algebra* pure way, is

```
import("stdfaust.lib");

process = 1000
        , ((8 : os.osc) , 50 : *)
        , ((1 , no.noise : ba.downSample , 500 : *) , 1000 : +)
        :> os.osc
        <: (_,_);
```

The diagram generated by Faust for this code is shown in Figure 2.4. Analysing from top to bottom, we have three blocks that are merged down to a `os.osc` and then merged up to two signals (`(_,_)`). The first three blocks are used to set the modulated frequency. The first block (`1000`) is always equal to 1000 (base frequency). The second block (`((8:os.osc),50:*)`) is responsible for creating the vibrato. To achieve that, the value `8` is piped to the input of a sine oscillator (`os.osc`). The block `os.osc` is declared inside the standard library and receives a frequency value as input and creates a sample value as output. The result of the composition of the last two blocks (8:os.osc) is then stacked over the value 50 to generate a composed block that is fed to the input of a multiplication block (`*`). The last block of these three blocks is responsible for randomly changing the frequency offset of the modulation. To do so, it down-samples the values that come from a noise generator to 1Hz. The down-sampling is done by the block `ba.downSample` (from the standard library) that receives as inputs the new frequency and the audio source, and that outputs the new sampled values. The output of the down-sampler is then amplified by 500 (by stacking it with the number 500 and piping the result block to the input of a multiplying block) and shifted by 1000 (to make the value range between 500 and 1000). Now, these three resulting blocks are merged down to form one only source. This source is piped into the input of another sine oscillator. This sine oscillator is responsible for generating the resulting modulated sine wave. Finally, to output the sound in stereo, the output of the sine is split into two.

This is, certainly, the most cryptic way of writing this example. For clarity, some modifications can (and should) be made:

```
import("stdfaust.lib");

base_freq = 1000;
vibrato = (8 : os.osc) , 50 : *;
random_freq = (1 , no.noise : ba.downSample , 500 : *) , 1000 : +;
process = base_freq , vibrato , random_freq :> os.osc <: (_,_);
```

Even beyond that, for the programmers that wish to avoid this kind of algebra as much as possible, Faust also allows the classical functional language syntax:

```
import("stdfaust.lib");

base_freq = 1000;
vibrato = os.osc(8)*50;
random_freq = ba.downSample(1,no.noise)*500+1000;
process = os.osc(base_freq+vibrato+random_freq) <: (_,_);
```

All three of these examples generate exactly the same audio processors. However, to generate Figure 2.4 (where all the created blocks are visible), only the first example is useful. The foundation behind Faust, the Grame Research Lab, provides an Online IDE where Faust programs can be tested out without any setup. The Online IDE can be accessed at `https://faustide.grame.fr/` (available at the time of writing of this document).

Faust is open-source and still is maintained to the date of redaction of this document.

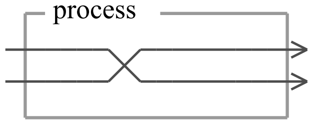


Figure 2.3: Faust Channel Swapper Program Diagram.

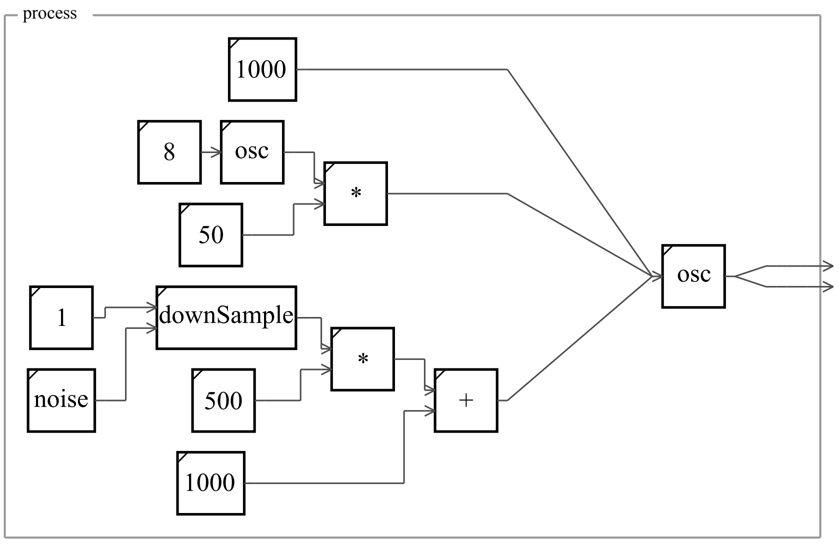


Figure 2.4: Faust Frequency Modulated Sine Wave Program Diagram.

2.6 Summary

The solutions presented here for digital audio synthesis are valid solutions, but neither of them use or create custom hardware to aid the creation of audio. Furthermore, these languages all fall into the already mentioned problem of multi-processing limitations (Section 1.1). In the next two Chapters (Chapter 3 and 4), the *rtfss* language and partial compiler will be introduced and explained. This language instead of relying on software solutions for audio synthesis, takes the approach of hardware design. This solution uses all the concepts presented in this present chapter before the introduction of computer music languages.

Chapter 3

The *rtfss* Language

The *rtfss* (RealTime FPGA Sound Synthesis) language is an audio processing high-level hardware description language. It was designed to ease the creation of digital sound processors and sound synthesizers. The main advantage of *rtfss* is that the hardware description generated by its compiler is purely sequential and pipelined. This ensures, for example, optimized use of the resources of an FPGA. Furthermore, *rtfss*'s syntax resembles the mathematical notation of signal processing algorithms. The main characteristic of this last feature is that previous sample values calculated from data streams can be accessed easily using negative integer indexes. Another feature worth mentioning here is that the language natively handles *MIDI 1.0*.

The *rtfss* language shares some technical similarities to other HDLs (hardware description languages). The main similarity to those HDLs is that *rtfss* follows the concurrency paradigm. In a concurrent language, the instructions are meant to be executed at the same time. This comes naturally in hardware, since the objective of HDLs is to generate parallel hardware architectures. Despite *rtfss* being a concurrent language, it allows the programmer to specify sequential statements. These sequential statements can be useful, for example, to keep the code clean. The compiler then morphs these statements to generate concurrent statements. Another feature similar to other HDLs is that all data streams can have an associated amount of bits (sample resolution). All the arithmetic operations are scaled accordingly to the dimensions of the operands. This parameter can even be controlled (but solved at compile-time) by constants.

In this Chapter, we discuss the design decisions of the language, its formal specification and grammar.

3.0 Formal Specification

The code of *rtfss* is arranged in blocks (similar to functions or black boxes) and each block (called CBlock) has a collection of statements. These statements can be split into multiple lines without invalidating their meaning. A source file can contain many of these blocks. Before the declaration of the first CBlock, there is a section where another source files can be imported, so that the CBlocks defined in the imported source file can also be used within the scope of the current file.

3.1 Comments Syntax

The language allows the creation of line and block comments. The line comments can start anywhere in a line and take up the rest of that line. Line comments start with a double forward-slash (`//`). Block comments can also start and end anywhere in the same or different lines. Every character that is inside a block comment is ignored by the language. Block comments start with a forward slash and an asterisk (`/*`) and end with an asterisk and a forward slash (`*/`). These have the following syntax:

```
//This is a line comment, everything is ignored
//after the double forward slash

/*
This is a block comment.
Everything between the delimiters will be ignored.*/
```

3.2 Pulse Statement

The synchronization pulse is the *rtfss*'s way of specifying the timing/throughput to which a certain data stream should oblige. Pulses are control types and are declared inside CBlocks. There are some reserved pulses that serve different purposes:

- `max`: Specifies to the compiler that any stream associated with this pulse should have the fastest clock possible (allowed by the physical constraints of the hardware);

- `const`: Specifies to the compiler that any stream associated with this pulse is, in fact, a constant stream, or const (for short). Consts are solved at compile-time and can be used in CBlocks to pre-calculate values will be used on the hardware. Const streams can be assigned to non-const streams, but the opposite is not valid.

To declare a pulse, two properties are required: the name of the pulse (that should be unique within the scope of this declaration and the inner scopes derived from this one), and the frequency or period of the pulse. The accepted time scales are Hertz (denoted with `Hz`), kiloHertz (denoted with `kHz`), seconds (denoted with `s`) and milliseconds (denoted with `ms`). The syntax follows the rule:

```
pulse <pulse_name> <time_val> <scale>;
```

While it is possible that different pulses having the same frequency/period might be merged on compilation, the *rtfss* language does not guarantee such an outcome. When two streams with the same pulse are used in an operation, no issues arise. But, when two streams with different pulses interact, the value from the slowest stream might be read several times. This means, on the present specification of this language, that the compiler will not perform any interpolation between stream values. This should be studied upon in future work. Thus, assignment between streams with different pulses infer a delay of one cycle of the master clock. The following example shows the creation of four pulses that use different time scales:

```
pulse very_slow 1s;
pulse fast 1ms;
pulse slow 100Hz;
pulse audio_rate 48kHz;
```

Since this language is sound-oriented, most pulses related with audio streams should range from 44.1kHz to 48kHz. Slower pulses are useful when creating control streams or other miscellaneous purposes. Each CBlock has an associated pulse that should be (but not forcefully) used by its streams. A CBlock that is instantiated with a `const` pulse will be also resolved at compile-time.

3.3 Stream Statement

The main statements present in CBlocks involve data streams and are called streams. A stream specifies how data is to be manipulated. When declaring a stream, four parameters need to be specified: the stream's name, the synchronization pulse, word type and the word size.

The stream's name (identifier) allows this stream to be referenced within the same scope (or lower scopes that derive from that one). This feature eases the creation of larger algorithms. The identifier must start with a letter (upper or lower case) or an underscore. The remaining symbols of the identifier can be letters (also upper or lower case), numbers or underscores. The minimum length of an identifier is one character.

The pulse synchronizes the throughput of this specific stream. When two streams are used together to form another stream, the pulse of the stream that is being attributed to is the pulse that will prevail.

The word type specifies if the stream revolves around fixed point arithmetic, floating point, or if it is a MIDI stream. The following types are valid:

- `I`: Signed fixed point stream (for example: `I8@spulse (...)`);
- `U`: Unsigned fixed point stream (for example: `U16.2@spulse (...)`);
- `F`: Floating point stream (for example: `F32@const (...)`);
- `midi`: MIDI stream (for example: `midi@spulse (...)`).

The floating-point type can only be used for compile-time operations and it is not synthesizable to hardware. The `midi` type represents a stream of MIDI commands (control sequences) specified by the MIDI standard [3]. The use of the `midi` datatype is aided by specific operators present on the stream arithmetic (explained on Section 3.3.6). Finally, the word size states the dimension of the integer and fractional parts of the word (when it is a fixed point stream), or the full size (if it is a floating-point). The word size is denoted using a fractional number in which the integer part represents the integer size and the fractional part represents the fractional size. This size can also be specified by reference (using a const stream). Streams can only be declared and assigned inside CBlocks. Each stream has to have one (and only) declaration, but can have multiple assignments. Stream declarations have the following syntax:

```
<word_type><word_size>@<pulse_name> <stream_name> [ = <assignment> ] ;
```

The word type and word size fields compose a single word and cannot be separated. Between that word and the pulse name, there is a "@" serving as a separator. Whitespaces are allowed between the "@" and the other two words. The stream name comes after the pulse name. The stream name should be unique within the current scope, derived inner scopes and the parent scope (stopping at the CBlock level). The declaration of streams does not require an assignment. If there is an assignment, recursion is only allowed if it refers to previous stream values (explained on Section 3.3.2). Some examples of declarations:

```
I16@pulse1 my_stream;
Iabb@const s = -3;
U4.2@max ctrl0 = 2.25;
```

While explicit streams have to be defined with a declaration, there are two types of streams that are unnamed: implicit streams (intermediate streams generated by arithmetic operations) and numeric literals (const streams). When handling a numeric literal, the compiler should attribute to that literal the smallest stream size possible to accurately represent the integer and fractional part of the literal. The maximum stream size that the conversion is allowed to use is set by the dominant datatype currently present in the stream it is being attributed to. By default, the dominant datatype is the datatype of the stream where the statement is being attributed to. But, if there is a cast that contains (even indirectly) the literal, then the datatype of the cast becomes dominant over that cast's boundaries. Casts will be explained further on Section 3.3.6. The next example illustrates the use of a cast to set the dominant datatype:

```
U16@const a = 412;      //Dominant datastream U16 (unsigned 16-bit integer)
U4.1@const b = 4.25;    //Dominant datastream U4.1 (fixed-point unsigned
                        //with 4-bit integer part and 1-bit fractional part)
I4.2@const c = (I4) 2.25; //Dominant datastream I4 (signed 16-bit integer)
```

The second statement of the example should generate a warning on compilation, since the fractional part of the numeral cannot be accurately represented in the stream. The last statement, to be precise, will generate an error, because there is a resolution mismatch between the datatype of the stream and its attribution.

Each stream can have multiple assignments. An assignment can be paired with a declaration, or be a standalone statement. When a stream has multiple assignments, and they do not have any relation with each other, the statement that is the further down is used. When multiple assignments refer to each other, the reference is recursively satisfied from the bottom statement to the top statement. This allows, for example, the fragmentation of a stream declaration. Stream assignments have the following syntax:

```
<stream_name> <assignment_operand> <assignment> ;
```

When the assignment is paired with the stream declaration, the assignment operand has to be an equal sign (=). In this case, the equal sign indicates the meaning of attribution. However, if the stream attribution is stand-alone, in addition to the attribution operand, the following are also valid:

- +=: Add and assign;

- `-=`: Subtract and assign;
- `*=`: Multiply and assign;
- `/=`: Divide and assign;
- `%=`: Modulus and assign;

These operands are a contraction of their traditional counterparts. For example, `A+=B` is the same as `A=A+B`. When using these contracted forms, the programmer must be aware of one peculiarity on the sizing rules. This characteristic will be explained further ahead in Section 3.3.8. The arithmetic operations will also be described in detail further ahead in Section 3.3.6. Some examples of assignments (without stream declarations):

```
my_stream=(x+3)*y;
my_stream=y;
s += 10;
ctrl0-=ctrl0/2;
```

Although inside the same stream the order of assignment is relevant, when dealing with different streams this is not the case. Since this language is concurrent, the order of declaration or assignment between different streams is irrelevant. This means that even if the statements are intertwined, their relative position of statement is ignored. Therefore, the following example is valid:

```
b=a;
a=3;
a+=2;
b+=3+c;
c=10;
U8.0@const a;
U8.0@const b;
U8.0@const c;
```

When fully resolved, the previous example will be equivalent to:

```
U8.0@const a=3+2;
U8.0@const b=a+3+c;
U8.0@const c=10;
```

3.3.1 Numeric Literal Representation

The representation of numeric values in the streams of the *rtfss* language has several forms. By default, the representation is done in decimal (base 10). Nevertheless, the language also allows the representation of numeric values in two more bases: octal (base 8) and hexadecimal (base 16). To represent a numeric value in octal, the number must be preceded by `0o`. However, if the intent is to represent in hexadecimal, the prefix must be `0x` or `0h`. The decimal base also has the optional prefix `0d`. To exemplify, the numeric value `12` (twelve), in base 10, can be represented in octal with `0o14` and with `0xC` in hexadecimal:

```
U16@const my_num10=12;  //Twelve in decimal
U16@const my_num8=0o14;  //Twelve in octal
U16@const my_num16=0xC; //Twelve in hexadecimal
```

3.3.2 Previous Stream Values

One of the main features of this language is the ability to access past stream values effortlessly. This is achieved by the use of the operator "value at instant" that is represented with an "'". This operator can be used in an individual expression or on an expression that manipulates multiple streams. On the left of the operator, the stream/streams should be placed, and the desired instant on the right. This mechanism works with relative instants. So, if the desired value is the value present in the stream in the last pulse trigger, then the value on the right should be -1. Positive values or zero do not affect the expression and, technically return the current stream value. Figure 3.1 contains a diagram of this behaviour. This mechanism allows the programmer to define pseudo recursive statements. A stream can use itself to define the next value. The syntax used is the following:

```
<stream_expression> ' <constant_expression>
```

or

```
<stream_expression> ' <stream_expression>
```

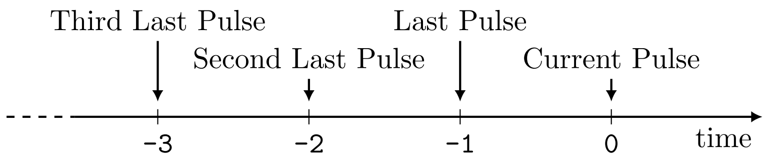


Figure 3.1: Stream Previous Instants timeline.

The simple use case of this feature is to define algorithms that inherently use previously defined instants as values to generate the current value. An example of this kind of algorithm is a Finite Impulse Response digital filter or a discrete convolution. They use a chain of previous stream values (with sums and multiplications) to create the current value. This kind of algorithms use the first form of syntax stated above. The second form is used to generate variable past instant streams. These are less common but are useful for the creation of variable time delay lines. Since the expression on the right of the operand is also a stream, the instant wanted can vary for each pulse trigger. Some examples of the use of this feature are:

```
I8.0@p a=a'(-1)-a'(-2);
I16.0@p b=b'a;
```

By default, on start-up, all previous stream values are zero. This behaviour can be overridden by defining the start-up values. To do so, the programmer needs to make an attribution to a stream indexed by a previous stream value. The syntax is the following:

```
<stream_name>'<instant>=<startup_value>;
```

When the stream is indexed by another stream, it is mandatory to specify the furthermost previous instant that the algorithm requires. This can be defined by indexing the desired stream (with index 1) the maximum depth required. Syntax used:

```
<stream_name>'1=<max_depth>;
```

If, after the hardware is synthesized, the stream is indexed with a value over the maximum depth value, the value used will be the one available at the instant of the maximum depth (saturation is used).

Summarizing, these two last features can be used and combined to generate arguably powerful time-dependent algorithms. In the next example, both properties are used and explained with comments:

```
I8.0@p a=a'(-1)-a'(-2); //Creating recursive, but constant
                        //depth stream "a"

a'(-1)=10;              //Setting the start-up value
a'(-2)=1;               //of instant -1 and -2

I16.0@p b=b'a;          //Creating recursive and
                        //variable depth stream "b"

b'(1)=5;                //Setting b's max depth to 5. This means
                        //that everything before b'(-5) will
                        //be equal to b'(-5)
```

3.3.3 Stream Synchronization

One of the main features of *rtfss* is that all the hardware generated is sequential. This means that all the expressions described in it are analysed and split into stages that are then organized in a pipeline-like architecture. Although this is beneficial for hardware synthesis, it means that algorithms with different complexity levels suffer different amounts of delay (stages in the pipeline). If those algorithms are to be combined, while no race condition will occur, the streams can end up using previous stream values. This is due to the fact that a compiler can be programmed to only synchronize operations inside a stream, and not between different streams (not the case on the implementation of the compiler presented on Chapter 4). Even though this problem is rare, since one of the streams needs to have a delay longer than the period of a pulse, the lack of mechanisms to handle this would result in undefined behaviour in those situations.

In order to solve this issue, *rtfss* has a stream synchronization mechanism called "Relative Stream Delay". This mechanism is exposed as an arithmetic operation that given two streams, returns the relative delay of the stream on the right operand relative to the left operand. The syntax is the following:

```
<reference_stream> gap <tested_stream> = <relative_delay>;
```

If the stream in the left operand is slower than the one on the right operand, the value returned is positive, otherwise, it is negative. The value returned is a signed const with 32 bits. A handy way to use it to synchronize the streams is to combine this feature with the one presented on Section 3.3.2:

```
I8.0@p a;
I16.0@p b;

I8.0@p sync_a;
I16.0@p sync_b;

// (...)
// Abridged code block that sets 'a' and 'b' algorithms
// Stream 'a' has a delay of 2 pulses. Stream 'b' has no delay.

sync_a=a'(b gap a); //(b gap a) returns -2,
                    //sync_a has a two units delay is introduced;
sync_b=b'(a gap b); //(a gap b) returns +2,
                    //sync_b has no delay introduced;
```

This feature only works when both streams share the same pulse. The stream sizes or types can vary. Stream synchronization does not apply to const streams, since all values are calculated "simultaneously". This cannot be used in `midi` streams.

3.3.4 Stream Buses

Because *rtfss* is a hardware description language, it is not feasible to introduce the creation and manipulation of arrays and their classical use cases in sequential languages. However, the language introduces one alternative to classical arrays: Stream Buses. A stream bus is a numbered and ordered collection of streams that share the same pulse, datatype and size. The behaviour of a member of a stream bus is equal to the behaviour of a stream. The size of the bus has to be determined at compile-time, but can be specified by a const stream. This means that the stream bound to an index of the bus cannot change after compilation. If there is more than one attribution to the same index of a bus, the one furthest down has priority, and dependencies are solved the same way as normal streams. The following syntax creates a stream bus:

```
<datatype>@<pulse> <bus_name>[<bus_size>];
```

To access a position of a bus, we simply index the bus with the wanted position. In the following example, a bus with two slots is created and the first slot is bound to a stream:

```
I16@mypulse ctrl;
I16@mypulse mybus[2];
mybus[0]=ctrl;
```

The *rtfss* language allows for a more compact initialization of a stream bus. This representation resembles a classic array initialization:

```
<datatype>@<pulse> <bus_name>[<bus_size>] = { <value0> , <value1> , ... };
```

The values are contained between the curly braces and are separated by commas. An example of use of this form is the following:

```
I1.5@const my_arr[5] = {0.256, 0.571, -0.141, 0, 0.9};

I16@mypulse ctrl0;
I16@mypulse ctrl1;
I16@mypulse mybus[2] = {ctrl0,ctrl1};
```

When the pulse is a `const`, as expected, all values are calculated and replaced at compile-time, so the bus ends up as a look up table. In this case, the stream bus is called const array / constant array. This mechanism eases programming and is the most straightforward way of generating a read-only memory block on hardware.

3.3.5 Cyclic Dependencies

The syntax used by *rtfss* is very flexible since it allows the full control of time dependencies. However, this flexibility comes with a cost: cyclic dependencies are syntactically allowed. A cyclic dependency occurs when a stream refers to itself in its assignment, or when there is a chain of assignments that creates a circular reference. A cyclic dependency is fundamentally unsynthesizable and does not have a valid mathematical meaning. To avoid problems, *rtfss*'s compiler should detect all the occurrences of this type and stop the compilation process. Below, two examples of cyclic dependencies are shown:

```
I8.0@p a=a+1;       //ERROR: Self dependence. Uncompilable!

I8.0@p b=c+1;       //First declaration, no error
I8.0@p d=b;         //So far, so good
I8.0@p c=d/3;       //ERROR: Dependence chain b->c->d->b. Uncompilable!
```

Cyclic dependencies are only allowed when the chain is broken by the use of previous stream values. Accessing a previous value of a stream can be seen as accessing a completely different stream. This stream does not have any kind of dependency, so the chain is broken:

```
I8.0@p a=a'(-1)+1;  //Valid: No self dependence, a'(-1) can be seen
                    //as an independent stream.

I8.0@p b=c+1;       //First declaration, no error
I8.0@p d=b;         //So far, so good
I8.0@p c=d'(-1)/3;  //Valid: Dependence chain b->c->d->b was broken by
                    //accessing d'-1 on c. New chains:
                    //b->c->d'(-1) and d->b
```

3.3.6 Numeric Stream Arithmetic

The *rtfss* language offers a complete set of operators allowed on streams. These operators range from basic operations (sum, sub, comparisons, etc...) to more complex operations (property of, value at instant, etc...). These operations are not allowed on MIDI streams and most are valid on const floats. As mentioned before, streams can only be declared and assigned inside CBlocks. So, naturally, they can also only be manipulated inside CBlocks. Each operator has a specific set of rules, but the main rationale is that, by default, the operation is lossless and suffers no possible overflow. This means that the datatypes are resized (to an extent) accordingly to the current operation. This behaviour is not used when manipulating with const float streams. Most operations yield some kind of delay or pipeline stage. The introduction of such is handled internally and is mostly hidden to the programmer. In this Section, each of these operations will be listed and explained.

+ (Sum)

The sum is a binary operation between two streams/constants. The result is the mathematical sum of the value of the current instant of both operands. The resulting datatype of the operation is a stream if at least one of the operands is a stream, or a constant if both operands are constant. If the operation is done between unsigned/signed streams, the result is signed but if it is performed only between unsigned, then the result is unsigned. If one of the operands is a const float, the result is also a const float. When using signed / unsigned, the resulting stream size abides by the following rule:

```
Xaa.bb+Xcc.dd = X(max(aa,cc)+1).max(bb,dd)
```

The size of the integer part of the result is the biggest size of the integer part of the operands, plus one. The size of the fractional part is the biggest one from the operands.

- (Subtraction)

The subtraction is a binary operation between two streams/constants. The result is the mathematical subtraction of the value of the current instant of both operands. The resulting datatype of the operation is a stream if at least one of the operands is a stream, or a constant if both operands are constant. If the operation is done between unsigned/signed streams, the result is signed but if it is performed only between unsigned, then the result is unsigned. If one of the operands is a const float, the result is also a const float. When using signed / unsigned, the resulting stream size abides by the same rule as the sum:

```
Xaa.bb-Xcc.dd = X(max(aa,cc)+1).max(bb,dd)
```

- (Symmetric)

The symmetric is a unary on a stream/constant. The result is the mathematical symmetric of the value of the current instant of the operand. The resulting datatype of the operation is preserved. If the operation is done on a signed stream, the result is also signed. But, if the operation is done on an unsigned, the result is also signed stream. In both cases, an extra bit is added to the final size to avoid overflow. If it is done on a const float, the type is preserved. Thus, when using signed / unsigned, the resulting stream size abides by the following rule:

```
-Xaa.bb = X(aa+1).bb
```

* (Multiplication)

The multiplication is a binary operation between two streams/constants. The result is the mathematical multiplication of the value of the current instant of both operands. The resulting datatype of the operation is a stream if at least one of the operands is a stream, or a constant if both operands are constant. If the operation is done between unsigned/signed streams, the result is signed but if it is performed only between unsigned, then the result is unsigned. If one of the operands is a const float, the result is also a const float. When using signed / unsigned, the resulting stream size abides by the following rule:

```
Xaa.bb*Xcc.dd = X(aa+cc).(bb+dd)
```

/ (Division)

The division is a binary operation between two streams/constants. The result is the mathematical division of the value of the current instant the left operand over the right operand. The resulting datatype of the operation is a stream if at least one of the operands is a stream, or a constant if both operands are constant. If the operation is done between unsigned/signed streams, the result is signed but if it is performed only between unsigned, then the result is unsigned. If one of the operands is a const float, the result is also a const float. When using signed / unsigned, the resulting stream size abides by the following rule:

```
Xaa.bb/Xcc.dd = X(aa+dd).(bb+cc)
```

% (Modulus)

The modulus is a binary operation between two streams/constants. The result is the mathematical modulus of the value of the current instant of the left operand by the right operand. The resulting datatype of the operation is a stream if at least one of the operands is a stream, or a constant if both operands are constant. If the operation is done between unsigned/signed streams, the result is signed but if it is performed only between unsigned, then the result is unsigned. Both operands must be fully integer. When using signed / unsigned, the resulting stream size abides by the following rule:

```
Xaa%Xcc = Xmin(aa,cc)
```

<< (Shift Left)

The shift left operation is a binary operation between a stream/const and a const. The result is the bitwise left shift of bits of the left operand. The amount of bits to shift is specified on the const present on the right operand. The resulting datatype of the operation is a stream if the left operand is a stream, or a constant if both operands are constant. Negative or fractional shifts are not allowed, so the right operand must be unsigned and integer. The result type is preserved from the left operand. The left operand is also not allowed to be a const float. The resulting stream size abides by the following rule:

```
Xaa.bb<<Ucc = X(aa+Ucc).max(0,bb-Ucc)
```

`>>` (Shift Right)

The shift right operation is a binary operation between a stream/const and a const. The result is the bitwise left shift of bits of the left operand. The amount of bits to shift is specified on the const present on the right operand. The resulting datatype of the operation is a stream if the left operand is a stream, or a constant if both operands are constant. Negative or fractional shifts are not allowed, so the right operand must be an unsigned and integer. The result type is preserved from the left operand. If the left operand is signed, the signal is propagated on the shift. The left operand is also not allowed to be a const float. The resulting stream size abides by the following rule:

```
Xaa.bb>>Ucc = Xmax(0,aa-Ucc).(bb+Ucc)
```

`<<<` (Rotate Left)

The rotate left operation is a binary operation between an unsigned stream / unsigned const and a const. The result is the bitwise left rotate of bits of the left operand. The bits that are shifted out of the word take the space on the beginning of the word. If the rotate amount is equal to the size of the word, nothing happens. The amount of bits to rotate is specified on the const present on the right operand. The resulting datatype of the operation is a stream if the left operand is a stream, or a constant if both operands are constant. Negative or fractional rotates are not allowed, so the right operand must be unsigned and integer. The result type is preserved from the left operand. The left operand is also not allowed to be a const float. Finally, a rotate operation has the same datatype size as the datatype of the left operand. The resulting stream size abides by the following rule:

```
Uaa.bb<<<Ucc = Uaa.bb
```

`>>>` (Rotate Right)

The rotate right operation is a binary operation between an unsigned stream / unsigned const and a const. The result is the bitwise right rotate of bits of the left operand. The bits that are shifted out of the word take the space on the end of the word. If the rotate amount is equal to the size of the word, nothing happens. The amount of bits to rotate is specified on the const present on the right operand. The resulting datatype of the operation is a stream if the left operand is a stream, or a constant if both operands are constant. Negative or fractional rotates are not allowed, so the right operand must be unsigned and integer. The result type is preserved from the left operand. The left operand is also not allowed to be a const float. Finally, a rotate operation has the same datatype size as the datatype of the left operand. The resulting stream size abides by the following rule:

```
Uaa.bb>>>Ucc = Uaa.bb
```

`== != < > <= >=` (Logic Comparisons)

The logic comparisons are a set of binary operations between two streams/constants. The result is the numeric comparison of the value of the current instant of both operands. The comparisons must be done with operands of the same type. The result of this operation is always an unsigned integer stream with only one bit. This one bit is set to one when the relation of the comparison is satisfied, and zero otherwise. The operations have the following meanings:

- Equal to (`==`): This comparison is satisfied when both operands have the same number;
- Not equal to (`!=`): This comparison is satisfied when both operands have different number;
- Less than (`<`): This comparison is satisfied when the number present on the left operand is numerically smaller than number present on the right operand;
- More than (`>`): This comparison is satisfied when the number present on the left operand is numerically bigger than number present on the right operand;
- Less or equal to (`<=`): This comparison is satisfied when the number present on the left operand is numerically smaller or equal to number present on the right operand;
- More or equal to (`>=`): This comparison is satisfied when the number present on the left operand is numerically bigger or equal to number present on the right operand.

The resulting datatype of the operation is a stream if at least one of the operands is a stream, or a constant if both operands are constant. One or both the operands can be a const float, but only if the other is also a const. The resulting stream size abides by the following rule:

```
Xaa.bb (== != < > <= >=) Xcc.dd = U1.0
```

`and or xor not` **(Logic Operations)**

Since *rtfss* is a hardware description language, it has a set of logic operations that allow for direct boolean logic bitwise manipulation of data. This set of operands are binary operands and are used on a pair of stream / consts. Due to the nature of floats (floating point representation), this kind of operations cannot be used with float consts. The resulting datatype of the operation is a stream if at least one of the operands is a stream, or a constant if both operands are constant. If the operation is done between unsigned/signed streams, the result is signed but if it is performed only between unsigned, then the result is unsigned. By nature, this kind of operation does not overflow, so the resulting stream/const will not be bigger than the biggest operand:

```
Xaa.bb (and or xor not) Xcc.dd = X(max(aa,cc)).(max(bb,dd))
```

`&` **(Property Of)**

The operation “property of” is a miscellaneous unary operator that returns a certain property of the operand. When the operand is a pulse, the operator returns the frequency of that pulse. But, when used on a bus or in a const array, the operator returns the size of the bus/array. The result of this operation is a const unsigned fixed-point value with 24 bits on the integer part, and 8 bits on the fractional part. This operation is also allowed on const floats. Since this operation is solved at compile-time, it does not introduce any delay or pipeline stage. The resulting stream size abides by the following rule:

```
&Xaa.bb = U24.8
```

' (Value At Instance)

The operator “value at instance” is a miscellaneous binary operator between a stream and a stream/const. This operation allows the access of previous sample values of the left operand's stream by the amount defined by the right operand's stream/const. This operation was already described in greater detail on Section 3.3.2. Since the delay cannot be fractional, the stream/const on the right must be an integer stream, and cannot be a const float. Since the stream is only a delay of the left operand's stream, then datatype and size from the left operand's stream are preserved. So, the resulting stream size follows this rule:

```
Xaa.bb'Xcc = Xaa.bb
```

`gap` (Relative Stream Delay)

The gap is a binary operation between two streams. The result is a const signed integer that represents the relative stream delay between both streams. The operation can be done between unsigned and signed. The value is calculated by analysing the depth of each pipeline generated from each stream and comparing the differential of length. This operation is only valid between streams of the same pulse. Neither operands can be a const float. The resulting stream size abides by the following rule:

```
Xaa.bb gap Xcc.dd = I32.0
```

(Casting)

rtfss does not allow implicit typecasting, so explicit casting mechanisms must be used. Explicit type casting is a miscellaneous binary operator between a stream type and a stream/-const. This operation returns a new stream that has the value of the right operand but has the type of the left operand. The operand has two forms:

- `(type) stream`: C like cast;
- `type (stream)`: C++ like cast (functional notation).

Both forms can be used and yield exactly the same result. Casting does not introduce any delay or pipeline stages. When using signed / unsigned, the resulting stream size abides to the following rule:

```
(typeXaa.bb) Xcc.dd = typeXcc.dd
typeXaa.bb (Xcc.dd) = typeXcc.dd
```

3.3.7 `midi` Stream Arithmetic

All the previous stream operators do not apply to `midi` streams. These kinds of streams have their own set of operators. These operators' job is to extract data from the `midi` streams that can be used for numeric operations. All the `midi` stream operators convert a stream from `midi` to numeric streams, so in the current state of the specification, it is not possible to create a `midi` stream, only consume its values. For the MIDI notes, since MIDI allows polyphony, *rtfss* uses a concept called voices. A voice can be seen as a single sequence of notes. Multiple voices allow multiple notes to be processed at the same time. For example, the maximum amount of parallel notes on two voices is two notes. The maximum number of voices corresponds to the total number of possible notes, so it is 128 (7-bit value). Voices are used in some of `midi` stream operators. The *rtfss* specification provides the following `midi` operators:

noteof (Note Number of Channel)

This operator allows the extraction of the note number from a `midi` stream, on a given channel (and voice). It is a quaternary operator that resembles a short CBlock instantiation. The first operator takes the role of a const stream input on a CBlock, and it specifies the maximum number of voices allowed (with a seven-bit integer unsigned const). The second operand (also resembling a const input) specifies the desired voice to extract and is represented similarly with a seven-bit integer unsigned const. The third operand (resembling stream input) is a `midi` stream and the fourth operand is an unsigned four-bit integer stream that specifies the desired channel. The result of the operand is an unsigned stream that has seven bits of size (follows the MIDI protocol). It has the following syntax:

```
noteof(U7,U7:midi,U4) = U7
```

freqof (Frequency of Channel)

This operator allows the extraction of the frequency of the note from a `midi` stream, on a given channel (and voice). The value of the frequency follows the twelve-tone equal temperament tuning (12-TET) [3]. It is a quaternary operator that resembles a short CBlock instantiation. The first operator takes the role of a const stream input on a CBlock, and it specifies the number of the highest voice (with a seven-bit integer unsigned const). The second operand (also resembling a const input) specifies the desired voice to extract and is represented similarly with a seven-bit integer unsigned const. The third operand (resembling stream input) is a `midi` stream and the fourth operand is an unsigned four-bit integer stream that specifies the desired channel. The result of the operand is an unsigned stream that has the size of the dominant datatype of the stream it is present on. It has the following syntax:

```
freqof(U7,U7:midi,U4) = Ua.b , where a.b is the size of the dominant datatype
```

velof (Velocity of Channel)

This operator allows the extraction of the velocity value from a `midi` stream, on a given channel (and voice). It is a quaternary operator that resembles a short CBlock instantiation. The first operator takes the role of a const stream input on a CBlock, and it specifies the number of the highest voice (with a seven-bit integer unsigned const). The second operand (also resembling a const input) specifies the desired voice to extract and is represented similarly with a seven-bit integer unsigned const. The third operand (resembling stream input) is a `midi` stream and the fourth operand is an unsigned four-bit integer stream that specifies the desired channel. The result of the operand is an unsigned stream that has seven bits of size (follows the MIDI protocol). It has the following syntax:

```
velof(U7,U7:midi,U4) = U7
```

patof (Polyphonic Aftertouch of Channel)

This operator allows the extraction of the note after-touch value from a `midi` stream, on a given channel (and voice). It is a quaternary operator that resembles a short CBlock instantiation. The first operator takes the role of a const stream input on a CBlock, and it specifies the number of the highest voice (with a seven-bit integer unsigned const). The second operand (also resembling a const input) specifies the desired voice to extract and is represented similarly with a seven-bit integer unsigned const. The third operand (resembling stream input) is a `midi` stream and the fourth operand is an unsigned four-bit integer stream that specifies the desired channel. The result of the operand is an unsigned stream that has seven bits of size (follows the MIDI protocol). It has the following syntax:

```
patof(U7,U7:midi,U4) = U7
```

nntof (New Note trigger of Channel)

This operator allows the extraction of the trigger of a new note from a `midi` stream, on a given channel (and voice). It is a quaternary operator that resembles a short CBlock instantiation. The first operator takes the role of a const stream input on a CBlock, and it specifies the number of the highest voice (with a seven-bit integer unsigned const). The second operand (also resembling a const input) specifies the desired voice to extract and is represented similarly with a seven-bit integer unsigned const. The third operand (resembling stream input) is a `midi` stream and the fourth operand is an unsigned four-bit integer stream that specifies the desired channel. The result of the operand is an unsigned stream with only one bit. This bit goes to high for one pulse cycle when the note present on the `midi` stream of the desired channel changed, otherwise it is always low. It has the following syntax:

```
nntof(U7,U7:midi,U4) = U1
```

ccof (Controller of Channel)

This operator allows the extraction of a certain controller value from a `midi` stream, on a given channel. It is a ternary operator where the first operand is the `midi` stream, the second operand is an unsigned four-bit integer stream that specifies the desired channel and the last operand is an unsigned seven-bit integer stream that specifies the desired controller. The result of the operand is an unsigned stream that has seven bits of size (follows the MIDI protocol). It has the following syntax:

```
ccof(midi,U4,U7) = U7
```

pof (Program of Channel)

This operator allows the extraction of the program (patch) number from a `midi` stream, on a given channel. It is an unary operator where the only operand is the `midi` stream. The result of the operand is an unsigned stream that has seven bits of size (follows the MIDI protocol). It has the following syntax:

```
pof(midi) = U7
```

cpof (Channel Pressure of Channel)

This operator allows the extraction of the channel pressure (or channel after-touch) value from a `midi` stream, on a given channel. It is a unary operator where the only operand is the `midi` stream. The result of the operand is an unsigned stream that has seven bits of size (follows the MIDI protocol). It has the following syntax:

```
cpof(midi) = U7
```

pbend (Pitch Bend of Channel)

This operator allows the extraction of the channel pitch bend value from a `midi` stream, on a given channel. It is a unary operator where the only operand is the `midi` stream. The result of the operand is an unsigned stream that has fourteen bits of size (on the MIDI protocol, pitch bends have double the precision compared to normal controllers). It has the following syntax:

```
pbend(midi) = U14
```

The voice allocation on the `noteof`, `freqof`, `velof`, `nntof` and `patof` operators is made by allocating the first unused voice. This means that the voice numbered the highest will only be used when all the others are already allocated with a note. A voice is allocated when the `midi` stream receives a Note On MIDI message and is deallocated when the `midi` stream receives a Note Off MIDI message. To serve as an example, the following code will extract from a `midi` stream the note value from two voices, and the velocity value from the first voice (on channel 5):

```
//Assuming a midi stream called "midi_in"
//controlled by a pulse called "ppulse"

U7@ppulse note0 = noteof(1,0:midi_in,5);
U7@ppulse note1 = noteof(1,1:midi_in,5);
U7@ppulse vel0 = velof(1,0:midi_in,5);
```

3.3.8 Stream Sizing on Contracted Operators

As mentioned before on Section 3.3, using a contracted operator (`+=`,`-=`,`*=`...) on an attribution to a stream is *mostly* equivalent of doing the extended form. But, one issue particular to *rtfss* arises. Since the language assures that the operations are lossless, every operation that is contracted generates a bigger stream word size than the target stream. The way that *rtfss* deals with this issue is to internally handle the contracted form as a cast to the target stream size on the expanded operation:

```
U16@const a = 4;
a+=6;
```

This example is equal to:

```
U16@const a = 4;
a=U16(a+6);
```

Contracted operators were added to this language, like in many others, as syntactic sugar. Initially, the inclusion of contracted operators in this language was scrapped because of the stream sizing restrictions. Further ahead on the development of the language, the solution presented was found and it was decided it was a good compromise. This way, the use of contracted operators should be handled with care to avoid overflows and unexpected behaviours.

3.4 CBlock

Like most programming languages, *rtfss* has mechanisms to organize code. But since this is not a sequential programming language, functions are not available. Instead, a black-box block approach is used. This also goes in line with hardware programming languages. In *rtfss*, blocks are called CBlocks. A CBlock has a set of input streams, output streams and const (input) streams. In addition to those, a CBlock also has an associated pulse that should be the preferred pulse inside the block. This pulse is used to synchronize the inputs and outputs of the block. A CBlock declaration follows the syntax below:

```
cblock@<pulse_name> <cblock_name>(<consts_in>:<streams_in>:<streams_out>){
    //Code
}
```

A CBlock has to have at least one output stream, but it can have zero input stream or zero const inputs. Const inputs can be used to, for example, parametrize stream dimensions and other const operations inside the block. While the output streams are both readable and writable, the input streams are read-only. To specify the name of the block inputs and outputs, the declaration is similar to regular streams, but instead of the semicolon, the separator are commas and the pulse is not specified. The syntax is as follows:

```
(<datatype><stream_size> <stream_name> , ...)
```

Furthermore, the pulse specified on the CBlock declaration is then bound to a real pulse when the block is instantiated. But, if the pulse name is const, then the CBlock becomes a const CBlock. A const CBlock is exclusivity used to generate const values / buses. They are not synthesized to hardware and are solved at compile-time. Their status of const block cannot be overridden on instantiation. When using const blocks, the declaration changes a little bit: the `<streams_in>` slot disappears, and the `<streams_out>` slot becomes `<const_out>`:

```
cblock@const <const_cblock_name>(<consts_in> : <consts_out>){
    //Code
}
```

On the other cases, the use of reserved pulse names serve as an indication of the intended use of the block, but not an obligation. The next code block serves as some examples of CBlock declarations:

```
//Consts used to size streams and to manipulate streams
cblock@mp gain(U8 in_size,U8 out_size,U8 amnt : Uin_size is : Uout_size os){
    os=(Uout_size) is*amnt;
}
```

```
//No input consts, only one in stream and one out stream
cblock@mp double(: U8 is : U9 os){
    os=is<<1; //more efficient than out=is*2;
}

//Consts CBlock
cblock@const inv_square(F32 inv : F32 invv,F32 invsqrv){
    invv=1.0/inv;
    invsqrv=invv*invv;
}
```

CBlocks can be instantiated inside other CBlocks. In order to instantiate a CBlock, all the wanted inputs and outputs should be already created as local streams. The same applies to the selected pulse. There are two types of CBlock instantiation: the normal form and the shortened form. The syntax for the normal form is:

```
[<streams_out>] = <cblock_name>@<pulse_name> (<consts_in> : <streams_in>);
```

In the case of const cblocks:

```
[<consts_out>] = <const_cblock_name>@const (<consts_in>);
```

This type of declaration is an independent statement that cannot be combined with stream arithmetic. CBlocks can be instantiated as many times as needed, but the hardware generated for a block will most probably be replicated on compilation. This means that, by definition, the compiler is not obliged to reuse the hardware from one block to multiple instances. Moreover, CBlock recursivity is not allowed.

Another way of instantiating CBlocks is the shortened form. The shortened form allows the programmer to combine the instantiation with its use on stream arithmetic as if it was only one stream. To be able to use the shortened form, the CBlock to be instantiated needs to only have one output. If this is true, the instantiation takes the shape of classical function return, with these syntaxes:

```
//Normal CBlock
<cblock_name>@<pulse_name> (<consts_in> : <streams_in>)

//Const CBlock
<const_cblock_name>@const (<consts_in>)
```

The next code block serves as some examples to both CBlock instantiation forms of the CBlocks declared in the last example:

```
U8@ps sound;
U8@ps ampsound;
[ampsound]=gain@ps(8,8:sound); //Normal form

U8@ps norm;
```

```
U9@ps doub_norm_inc=double@ps(:norm)+1; //Shortened form

F32 inv;
F32 invv;
F32 invsqrv;
[invv,invsqrv]=inv_square(inv); //Normal form
```

Since all stream declaration and manipulation are done inside a CBlock, there is a necessity to use an uppermost CBlock that is always instantiated. This block is called `main` and behaves as the top-level entity for the hardware design. The interface that the `main` CBlock has is reflected to interface that the target generated hardware will have. The default pulse of this block has the frequency of the sample rate (that the I/O operates) of the system. For example, a `main` CBlock declaration could be similar to this:

```
cblock@smppulse main(:
    midi midi_in,I16 ctrl_in[8],U16 uctrl_in[8],I16 audio_in[2] :
    midi midi_out,I16 ctrl_out[8],U16 uctrl_out[8],I16 audio_out[2]){
    //Code
}
```

3.5 `if` Statement

The *rtfss* language allows the use of conditional logic flows. These can be achieved using logic gate manipulation (low-level), or using `if` statements. `if` statements allow the selection of stream statements / code blocks by the logic value of an expression. A basic `if` statement can have just a code block for if the logic statement is true. But, it can also present an alternative code block in the case the logic expression is false. This is achieved with an `else` statement. The syntax to both cases is:

```
if ( <logic_expression> ) {
    <stream_attributions> //Logic expression true
} //Only if statement

if ( <logic_expression> ) {
    <stream_attributions> //Logic expression true
}
else {
    <stream_attributions> //Logic expression false
}
```

It is worth noting that although a choice is being made, all branches need to be synthesized on `if` statements. This is due to the fact that a compiler cannot guess what will, and will not be executed. There is an exception to this rule, of course, when the logic expression is solvable at compile-time. In this case, the compiler can optimize the code by removing the `if` statement, and only placing the logic from the correct branch. The *rtfss* does not enforce the compiler to do this optimization, so it should not be completely relied upon.

The *rtfss* `if` statement can also allow more alternative statements. This is implemented by the introduction of `elseif` statements. There can be as many elseif statements as needed. The syntax used is the following:

```
if ( <logic_expression_0> ) {
    <stream_attributions> //expr_0 true
}
elseif (<logic_expression_1> ) {
    <stream_attributions> //expr_0 false and expr_1 true
}
(...)
else {
    <stream_attributions> //expr_0 false, expr_1 false, ...
}
```

The `if` statements can only be present inside CBlocks. Stream default values can also be present inside `if`, `elseif` or `else` statement code blocks. Finally, the last thing worth noting about these statements is that they should be avoided because they instigate sequential logic thought processes. Hardware description language should not be handled with sequential logic thought processes since their parallelism capabilities can only be exploited when the thought process is concurrent.

3.6 `for` Statement

The *rtfss* language also allows the instantiation of processing cycles / loops. They can be achieved by instantiating `for` statements. The `for` statements are an alternative to code replication (naive approach). Every `for` cycle has a variable associated. This variable symbolizes the number of the present iteration. On creation of a `for` cycle, the bounds of the loop must be specified. These are the lower bound and the upper bound of the value of the variable. Optionally, the increment of the cycle can be altered (by default it is 1). The syntax used for `for` statements is:

```
for ( <for_variable> in <lower_bound> to <upper_bound> ){
    <for_code_block>
} //Without specified increment

for ( <for_variable> in <lower_bound> to <upper_bound> inc <inc_value> ){
    <for_code_block>
} //With specified increment
```

Both the lower and upper bound, and the increment must be constant values / constant streams. The `for` statements can only be present inside CBlocks. Internally, the *rtfss* compilers can implement `for` cycles by, for example, creating a Finite State Machine where every state represents one iteration. Another way would be to replicate the naive implementation: unfold the cycles. It is up to the compiler to solve and find the best implementation. Similarly to the `if` statements, the used of these statements should be avoided, because they instigate sequential logic thought process. The use of `for` cycles can create an unwanted increase on the implementation complexity. Here are some examples of the use of `for` loops:

```
I16@const[10] lucas;
lucas[0]=2;
lucas[1]=1;

for(i in 2 to 10){
    lucas[i]=lucas[i-1]+lucas[i-2];
}
```

3.7 Import Syntax

The *rtfss* language is designed to be flexible and to allow the breakdown of complex algorithms into simpler blocks. The language also encourages the creation of basic building blocks and their arrangement into libraries. To do so, *rtfss* allows source code imports. A source code import can be used by writing, before the declaration of the first CBlock, the keyword `use`, followed by the name of the file to import. A *rtfss* compiler, parsing this line, will consume all the CBlocks present on that file (and all the CBlocks present on their imports). These can then be used on the current source file as they were declared on that file. The visibility of imports is unlimited. This means that if a file A imports a file B, and the file B imports a file C, then the CBlocks present on file C are visible on file A. This needs to be considered upon when naming CBlocks. Cyclic includes should be detected by the compiler, and their use is not allowed. The import syntax is the following:

```
use <file_name> ;
```

3.8 Compilation Targets

As mentioned before, *rtfss* is a sound-focused hardware description language. As such, the main target of the language is hardware devices. Certainly, the language can still be compiled to regular computer programming languages, but it would most likely diminish the advantages of using *rtfss*. Nowadays, the most popular hardware target devices (other than fixed hardware) are FPGAs [1]. Most FPGA hardware synthesis tools allow the compilation of VHDL or Verilog. Designing a language capable of being compiled directly to FPGA placement netlists would be a substantial undertaking. So, instead, *rtfss* is designed to be compiled to another (lower-level) hardware description language (for example, VHDL). The output of the compiled source code should then be fed into a hardware synthesis tool (compiler chaining).

3.9 Example Designs in *rtfss*

Now that the formal specification of the *rtfss* language is described, we can analyse in more detail some designs made in *rtfss*.

3.9.1 IIR Filter Design Revisited

First, we are going to look again at the example presented in Section 1.1:

```
cblock@smpp main(: I16 smp_in : I16 lp_out){
    //Low-pass aprox 0.2 Normalized Frequency IIR Filter
    I16.2@smpp lowpass = I16.2((lowpass'-1)>>1+(smp_in+(smp_in'-1))>>2);
    lp_out=I16(lowpass);
}
```

We can see that this design only contains one CBlock (`main`), and that the CBlock is controlled by the system's sample frequency (pulse named `smpp`). This design contains one integer stream input (`smp_in`) and one integer stream output (`lp_out`), but does not contain any const stream inputs. While the inputs and outputs have 16 bits of resolution, the internal stream `lowpass` has an extra two bits on the fractional part of the datatype. Since the operations inside the `lowpass` stream expand the size of the intermediates datatype, a cast is put before the attribution is made. This cast cuts the datatype of the intermediate calculations back to the size of the internal stream. The `value at instant` operator is used in both `lowpass` and `smp_in` streams to access the previous stream values needed for the filter. The sizes of the intermediate datatypes are outlined in Figure 3.2. The diagram present on this Figure is a modified simpler version of a graph generated by the *rtfss* compiler (explained on 4.8).

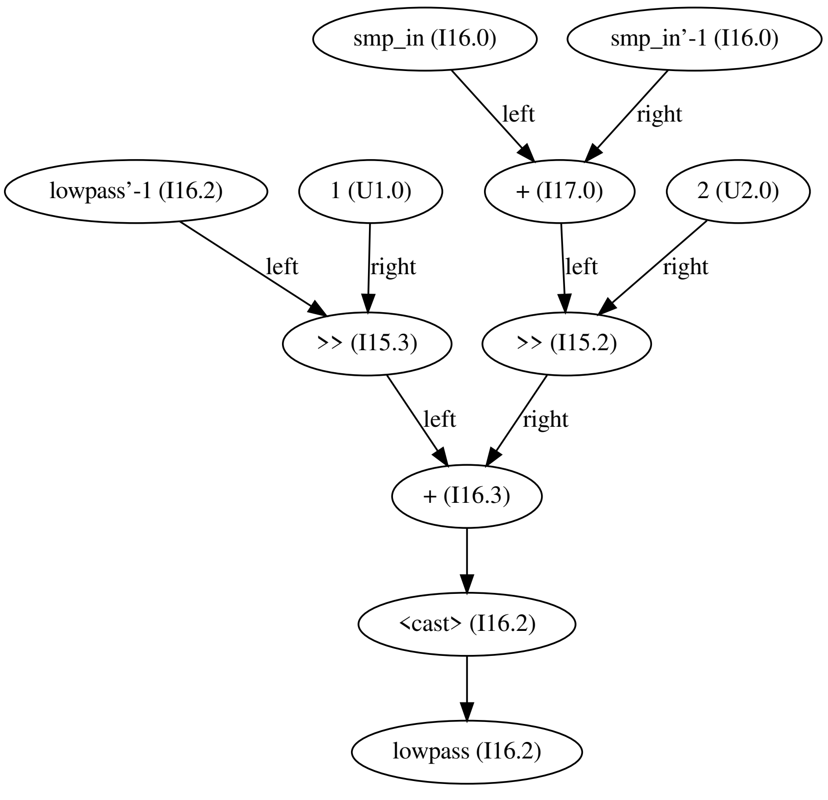


Figure 3.2: Example diagram of *rtfss* Stream Arithmetics.

3.9.2 Filtered Square Wave Design

This filter CBlock can be combined with other CBlocks to generate more intricate designs. The following example generates a filtered 2kHz square wave:

```
cblock@smpp lp_filter(: I16 smp_in : I16 lp_out){
    //Low pass 0.2 Normalized Frequency IIR Filter
    I16.2@smpp lowpass = I16.2((lowpass'-1)>>1+(smp_in+(smp_in'-1))>>2);
    lp_out=I16(lowpass);
}

cblock@frqp square(:: I16 sqr_wave){
    //Square wave oscillator
    //Output frequency = frqp pulse freq / 2
    sqr_wave = not (sqr_wave'-1);
    sqr_wave'-1 = I16(0x7FFF);
}

cblock@smpp main(:: I16 smp_out){
    pulse square_p 4000Hz; //Used to generate the 2kHz square wave
    smp_out=lp_filter@smpp(: square@square_p(:));
}
```

This design contains three CBlocks. The `lp_filter` CBlock is the same CBlock of the previous example (0.2 normalized frequency low-pass IIR filter) but with a different name. The `square` CBlock generates a fixed frequency square wave. The frequency of the square wave is half of the frequency of the pulse. Internally this module has a stream that inverts its value every pulse trigger. Since the default value at instant `-1` is the highest positive value possible on that resolution (hexadecimal `0x7FFF` equivalent to 32767 in signed decimal), when the stream inverts the new value is the lowest possible number on that resolution (hexadecimal `0x8000` equivalent to -32768 in signed decimal). One full wave cycle is achieved in two pulse triggers, so to generate a 2kHz square wave, the pulse of this block must work at 4kHz. The final CBlock is the `main` CBlock. This CBlock creates a new pulse to be used on the square wave module and instantiates both `lp_filter` and `square` CBlocks. Since both CBlocks only contain one output, the short instantiation is used. The output of the `square` CBlock is fed to the only input of the `lp_filter` CBlock. The output of this last CBlock is placed on the only output of the `main` CBlock. This design will be revisited and compiled on Appendix C.

3.10 Summary

The main motivation behind this thesis is to facilitate the creation of hardware solutions for audio processing chains. In this Chapter, a new language for hardware description of sound synthesis was introduced and explained. This language, called *rtfss*, offers some of the features present in normal computer programming languages while still also providing the traditional elements of a hardware description language. This language is designed to create an optimized logic circuitry by the used of a pipelined architecture.

In the next Chapter, a compiler that implements a representative part of the *rtfss* language specification is proposed and analysed.

Chapter 4

The *rtfss* Compiler

4.1 Preamble

As part of the thesis work, it was decided to implement a *rtfss* compiler that would handle a subset of the specification. The technologies used for this task are the following:

- C++: General-purpose programming language;
- Boost: A popular set of C++ libraries;
- ANTLR: Used to implement a *rtfss* compliant lexer and parser;
- VHDL: The hardware description language that is the target language of this compiler;
- CMake (and Make): CMake is a tool used for managing the build process of this compiler. Make is a building automation tool and is used by CMake as the underlying base;
- DOT (Optional): Graph description language. It is used to represent (and later visualize) all the graphs (and trees) the compiler generates.

The language chosen for this compiler was C++. It is a low-level language that allows complete control of how the memory is managed. It is also a compiled language, so it benefits from the perks of being compiled (for example, static optimizations and lower execution overhead). However, even though it is low level, there is a standard library (called C++ Standard Library) with a vast collection of already made tools and algorithms that speed up the implementation of more complex programs. There is also an external library, called Boost, that complements the Standard Library.

In addition to the C++ Standard Library, the Boost libraries were selected to allow the ease of two operations: string templating and arbitrary size number representation. For string templating, Boost offers the *Boost Format* library. For arbitrary size number representation, Boost has a library for dynamic bitsets located on `<boost/dynamic_bitset.hpp>`.

ANTLR is a lexer and parser generator. It uses a *LL(*)* approach to analyse the input and generate a tree representation of the input. This representation is called the *parse tree.* In order for ANTLR to work, a grammar that fully represents the target language specification must be created. Given this grammar, ANTLR generates code that serves as a lexer and parser. Although ANTLR's main target is Java, it allows the generation of, for example, Python and C++. For a more detailed explanation, refer to Section 2.3.

VHDL is a hardware description language that allows the description of digital systems. VHDL serves as the target compilation language of the *rtfss* compiler. VHDL code can be compiled to work on Field Programmable Gate Arrays (FPGAs, explained in Section 2.1), generate Application-Specific Integrated Circuits (ASICs) or others. For a more detailed explanation, refer to Section 2.2.

A compiler is usually a program that is composed of several different source files and objects. Each of these objects should be compiled individually and then linked together to generate the final program. To do so, normally a programmer would choose to have a `Makefile` and store in it all the instructions to compile the project. Then, to compile, the programmer invokes the command `make` on the same directory the `Makefile` is present, and the compilation process is triggered. The Make toolchain offers desirable tools that allow this compilation process, but it lacks one big thing: library management. This is where CMake comes in. CMake allows the specification of the libraries needed to compile the project, and they will be included "automatically". Furthermore, in CMake, the programmer can specify the internal objects and generate their own libraries. The dependencies of the libraries should also be mentioned. With all this information, the CMake generates a `Makefile` that handles all the libraries, linkage and compilation without needing to specify the individual compilation commands. Since the *rtfss* compiler relies on the ANTLR and Boost libraries, the use of CMake was obvious. The CMake created is responsible for getting both these libraries and for compiling the whole project.

The *rtfss* compiler is heavily based on trees and graph data-structures. Dealing with these data structures implies the manipulation of memory (mostly dynamic memory), and most importantly it implies the heavy manipulation of pointers. In order to aid the development of the compiler, it was decided to create auxiliary classes/objects that would allow the visualization of the trees and the graphs that the compiler generates. Our choice was to use the DOT representation and the GraphViz program. GraphViz (Graph Visualization Software) is a visualization software that allows the creation of a visual representation of graphs described in the DOT language [25]. While compiling, the compiler gives the option of showing the internal trees/graphs with all the relevant connections and information. These graphs are now used on this document to give concrete examples and to better explain the internal structure of the compiler. All the Figures that use the output of a synthesized graph from the compiler are labelled with the word "GraphViz".

The *rtfss* compiler's architecture is broken into multiple logic blocks. These blocks act as stages on the compilation process and act upon the data generated by the preceding blocks. Thus, this compiler can be seen as a pipeline of tasks that at the beginning consumes *rtfss* code, and at the end generates VHDL. These block stages will be explained in detail in this Chapter. A simplified view of the architecture of the compiler is shown in Figure 4.1, but to summarize, the compiler:

1. Generates a parse tree (using ANTLR);

2. Translates the parse tree into an abstract syntax tree;

3. Does multiple operations and manipulations over the tree;

4. Converts the tree into an architectural graph;

5. Does operations over that graph; converts the graph into the "final representation" graph;

6. Translates this final graph into the target language VHDL.

4.1.1 *rtfss*'s Compiler Limitations

The *rtfss* language is an arguably complete language that suits the needs of sound synthesis (and maybe beyond). But, the compiler implemented (at least at this stage) only implements a subset of the formal specification of the language. This is due mainly to the time window used to develop the project and its innate complexity. Some key features of the language are not functional, since they might be only partially implemented by some of the compiler's functional blocks. Even so, the compiler in its current state is completely functional and can be used to create hardware. Throughout this Chapter, the limitations of the compiler will be explained, but for clarity, the following list contains a summary:

- Only one CBlock (the `main`) is allowed, so all the algorithms must be specified in there;
- The import mechanism is not implemented. This is due to the fact that only the `main` CBlock is allowed, so even if imports were implemented, they would serve no use because the `main` CBlock cannot instantiate others;
- Constant stream inputs in CBlocks are allowed but are ignored. This, again, is due to the fact that only the `main` CBlock is allowed, so this feature would not provide useful functionality;
- The `midi` streams and correspondent operators are not operational on this stage of the compiler. They are accepted as valid syntax, but they do not generate any hardware;
- The `if` statements are not allowed, and subsequently `elseif` and `else` statements are not allowed either;
- The `for` statements are not allowed;
- Stream Buses are not implemented;
- The "property of" (`&`) and "relative stream delay" (`gap`) operators are not implemented, so their use will raise an error;
- Despite the last item stream instants are implemented but their use is restricted to compile-time constants on the time index. This means that a stream cannot time index another stream. Otherwise, it is synthesizable.

4.1.2 Support Example

To help understand better the compiler, a *rtfss* example design will be processed by the compiler. As the various stages of compilation are explained in this Chapter, the example will be used to reveal the state of the internal structures of the compiler.

The example we will be using is the IIR filter example that was already presented in Section 1.1 and Appendix A. This example does not cross the current restrictions of the *rtfss*

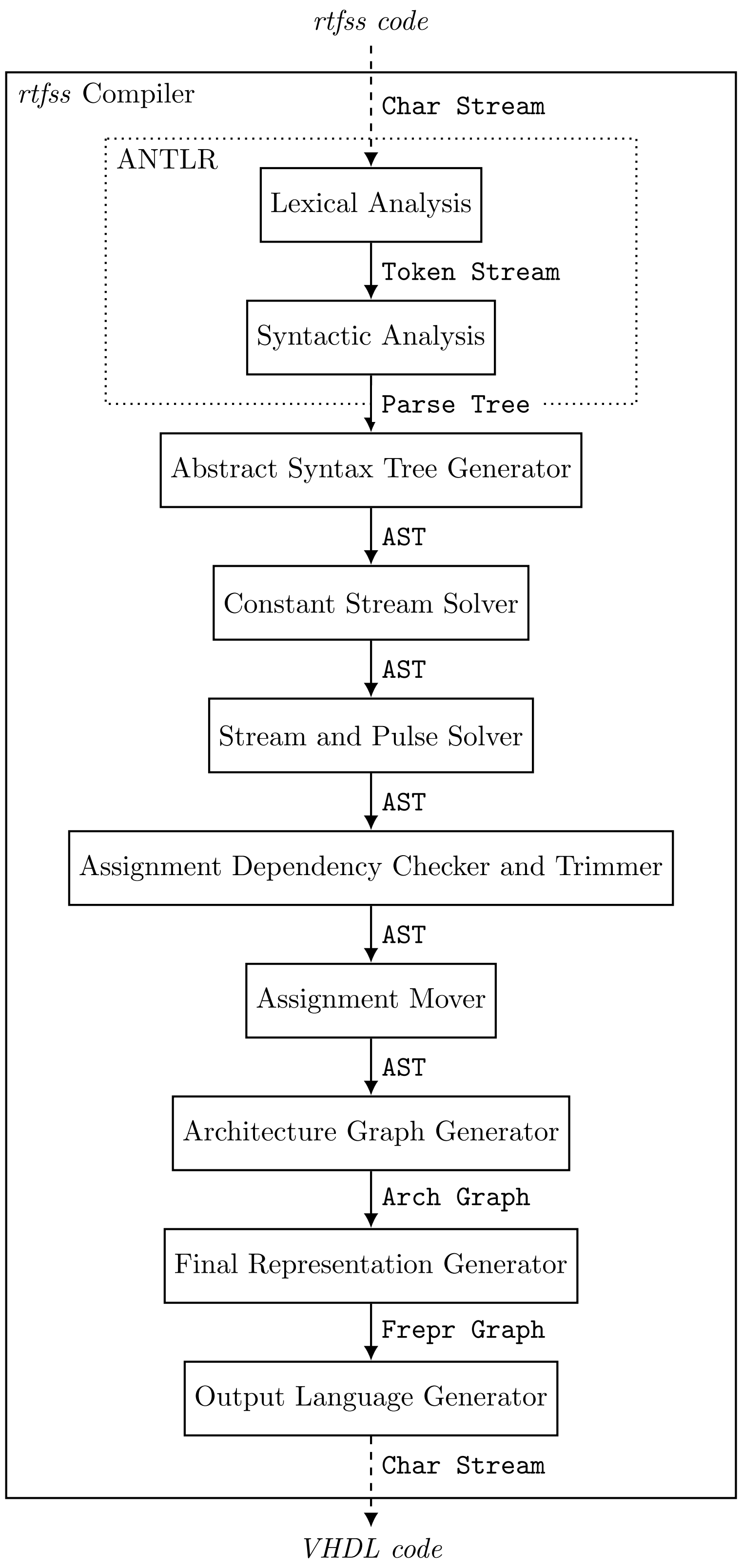


Figure 4.1: Simplified diagram of *rtfss* compiler processing stages.

compiler. However, the example was rewritten to better demonstrate various processing steps of the compiler:

```
cblock@smpp main(: I16 smp_in : I16 lp_out){
    lp_out=I16(lowpass);
    I16.2@smpp lowpass;
    lowpass = lowpass'-(1+1);
    lowpass = I16.2(smp_in+(smp_in'-1)>>2);
    lowpass += (lowpass'-1)>>1;
}
```

This expanded version is slightly different, but functionally equivalent to the original version. The internal audio stream is split into three attributions (and one separated declaration). The first of those three attributions is redundant since the next one overwrites this one. The last attribution uses a contracted operator, so it uses the expression from the last attribution. This expanded version contains one more cast than the normal one. The first cast (second attribution) is needed to align the sizes of the intermediate calculations with the size of the stream. Yet, since the size of the intermediate calculations is smaller than the internal audio stream (`I15.2` compared to `I16.2`), it does not affect the values. The second cast is implicit on the contracted operator. When this operator gets expanded, it will create a cast to the size of the target stream.

4.2 Input Tokenization and Syntactic Analysis

Input tokenization and syntactic analysis is usually the first step performed by a compiler. In this step, the input code is received and parsed. The character stream is then fed into a tokenizer. The tokenizer's job is to figure out, according to predetermined rules, the tokens present on the character stream. The tokens generated from this analysis are the output of the lexer. After the tokenization, the syntactic analysis consumes the tokens generated by the tokenizer and builds a syntactic parse tree following a set of rules.

The full ANTLR *rtfss* compliant grammar used on the *rtfss* compiler is in Appendix D.

4.2.1 Lexical Analysis

To better understand the tokenization process, consider for example, a tokenizer present on a pseudo-calculator. When given the input `a=1+2;`, the tokenizer could break that character stream into the following token sequence: `<identifier val='a'> <equals> <numeral val=1> <plus> <numeral val=2> <semicolon>`. If, while processing the character stream, no token rule fits the sequence of characters (and the input is at the end of stream), then the tokenizer raises an error and the compilation process stops.

Tokenization, in the *rtfss* compiler, is performed by ANTLR and is the first compilation step. The full *rtfss* specification is allowed by the lexer. The rules that the *rtfss* compiler's tokenizer follows can be split into two groups: fixed matching rules and dynamic matching rules.

The fixed matching rules are rules that match a specific (fixed) sequence of characters. These rules are composed of reserved words and syntactic symbols (such as operators). The fixed matching rules that the *rtfss* compiler has are:

- USE: Matches the character sequence use. Used on import statements;
- CBLOCK: Matches the character sequence cblock. Used on CBlock declaration;
- IF, ELSEIF and ELSE: Matches the character sequence if, elseif, else. Used on 'if', 'elseif' and 'else' statements;
- FOR : Matches the character sequence for. Used on 'for' statements;
- IN, TO, INC: Matches the character sequence in, to, inc. Used on 'for' statements;
- CURLY_OPEN and CURLY_CLOSE: Matches the character { and }. Used as the beginning and ending delimiter of code blocks;
- BRAC_OPEN and BRAC_CLOSE: Matches the character [and]. Used as the beginning and ending delimiter of a stream bus declaration or indexing;
- PAR_OPEN and PAR_CLOSE: Matches the character (and). Used as a beginning and ending delimiter in various situations;
- ARG_GROUP_SEP: Matches the character :. Used as a separator between the inputs and outputs of a CBlock;
- COMMA: Matches the character ,. Used as a separator in multiple occasions;
- PULSE_SEP: Matches the character @. Used as a separator on Streams to separate its name from its pulse;
- TERMINATOR: Matches the character ;. Used as the terminator of most of the statements of *rtfss*;
- MAX and CONST: Matches the character sequence max, const. Used as the identifier of a max and a const pulse;
- MIDI: Matches the character sequence midi. Used as the identifier of a midi stream;
- EQUAL: Matches the character =. Used on stream attributions / declarations;
- ADD, SUB, MULT, DIV, MOD: Matches the characters +, -, *, /, %. Used on arithmetic expressions. Their meaning is bounded to their symbol (already explained in Section 3.3.6);
- SL, SR, RL, RR: Matches the character streams <<, >>, <<<, >>>. Used on arithmetic expressions. Their meaning is bounded to their symbol (Section 3.3.6);
- AND, OR, XOR, NOT: Matches the character streams and, or, xor, not. Used on arithmetic expressions. Their meaning is bounded to their symbol (Section 3.3.6);
- GAP, PROP, INS: Matches the character streams gap, &, ^. Used on arithmetic expressions. Their meaning is bounded to their symbol (Section 3.3.6);
- NOTEOF, FREQOF, VELOF, PATOF, NNTOF: Matches the character streams noteof, freqof, velof, patof, nntof. Used on midi stream arithmetic expressions. Their meaning is bounded to their symbol (Section 3.3.7);

- CCOF: Matches the character stream ccof. Used on midi stream arithmetic expressions. Their meaning is bounded to their symbol (Section 3.3.7);
- POF, CPOF, PBEND: Matches the character streams pof, cpof, pbend. Used on midi stream arithmetic expressions. Their meaning is bounded to their symbol (Section 3.3.7).

To match tokens that do not have a fixed value, but have a fixed structure (for example, identifiers and numerics), the tokenizer has dynamic matching rules. To help organize the lexical rules of the compiler, a ANTLR feature called fragment is used. A fragment is a lexical rule that cannot be matched alone, but can be combined to generate a bigger rule. Some of the tokenizer rules that are of the fragment are dynamic matching rules. The main fragment rules the tokenizer has are the following:

- SIGNED_TYPE_PREFIX, UNSIGNED_TYPE_PREFIX and FLOATING_TYPE_PREFIX: Matches the character I (for signed), U (for unsigned) and F (for float);
- PULSE_LITERAL_SUFFIX: Matches the suffix of a pulse literal. The valid values for this rule are: s (seconds), ms (milliseconds), Hz (Hertz), kHz (kilo Hertz);
- FRAC_SEPARATOR: Matches the character ., which acts as a fractional numeric separator;
- OCT_PREFIX and HEX_PREFIX: Matches the prefix of numerics. In other words, character 0 (for octal) and the character stream 0x (for hexadecimal);
- OCT_DIGIT, DEC_DIGIT and HEX_DIGIT: Matches a numeral digit. It matches a character that is a number from zero to seven (octal), number from zero to nine (decimal) and number from zero to nine or letter from 'a' to 'f' (undercase or uppercase);
- OCT_INT, DEC_INT and HEX_INT: Matches a integer numeral. It is composed by a sequence of one or more OCT_DIGIT that is preceded by the OCT_PREFIX (for octal), one or more DEC_DIGIT (for decimal), or one or more HEX_DIGIT that is preceded by the HEX_PREFIX;
- DEC_FRAC: Matches a decimal fraction. It matches a FRAC_SEPARATOR that is preceded by one or more DEC_DIGIT and succeeded by zero or more DEC_DIGIT, or that is preceded by zero or more DEC_DIGIT and succeeded by one or more DEC_DIGIT;
- OCT_FRAC: Matches a decimal fraction. It matches a FRAC_SEPARATOR that is preceded by one or more OCT_DIGIT and succeeded by zero or more OCT_DIGIT, or that is preceded by zero or more OCT_DIGIT and succeeded by one or more OCT_DIGIT. All of this must be preceded by a OCT_PREFIX;
- HEX_FRAC: Matches a decimal fraction. It matches a FRAC_SEPARATOR that is preceded by one or more HEX_DIGIT and succeeded by zero or more HEX_DIGIT, or that is preceded by zero or more HEX_DIGIT and succeeded by one or more HEX_DIGIT All of this must be preceded by a HEX_PREFIX.

Most of the compiler tokenizer's dynamic matching rules rely on the fragment lexical rules. The *rtfss* compiler has the following dynamic matching rules:

- COMBOP: Matches the contracted operators +=, -=, *=, /= and %=;

- LOGIC_OP: Matches all the possible logic comparison operators >, <, >=, <=, == and !=;
- OCT_LITERAL, DEC_LITERAL and HEX_LITERAL: Represents a numeric literal in each of the three possible representations. For each, the rule accepts the integer alternative or the fixed alternative. For example, for the hexadecimal base, the rule HEX_LITERAL matches HEX_INT or HEX_FRAC;
- NUM_LITERAL: Represents a numeric literal represented in any base. It matches with a OCT_LITERAL, DEC_LITERAL or HEX_LITERAL;
- PULSE_LITERAL: Matches a value that can be set on a pulse. It matches a NUM_LITERAL followed by the PULSE_LITERAL_SUFFIX;
- SIGNED_TYPE: Matches a specification of a signed type. It accepts a SIGNED_TYPE_PREFIX followed by a DEC_DECIMAL or IDENTIFIER;
- UNSIGNED_TYPE: Matches a specification of a unsigned type. To match this rule, it expects a UNSIGNED_TYPE_PREFIX followed by a DEC_DECIMAL or IDENTIFIER;
- FLOATING_TYPE: Matches a specification of a floating type. To match this rule, it expects a FLOATING_TYPE_PREFIX followed by a DEC_DECIMAL or IDENTIFIER;
- IDENTIFIER: Matches a character stream that starts with a letter (undercase or uppercase) or an underscore, and that can have afterwards a sequence of numbers and / or letters and / or underscores;
- STRING_LITERAL: Matches a sequence of any character, that is preceded and proceeded by quotes;
- BLOCK_COMMENT: Matches a sequence of any character, that is preceded by a front slash and an asterisk and is proceeded by an asterisk and one front slash. All the tokens of this kind, are then discarded by the tokenizer since they do not hold valuable information to the compilation process;
- LINE_COMMENT: Matches a sequence of any character (except a new line), that is preceded by two front slashes. All the tokens of this kind, are then discarded by the tokenizer since they do not hold valuable information to the compilation process;
- WS: Matches a sequence of one or more white space characters. All the tokens of this kind, are then discarded by the tokenizer since they do not hold valuable information to the compilation process.

Support Example Analysis

The Token Stream generated by ANTLR4 while processing the example present in Section 4.1.2 is the following:

```
['cblock',<CBLOCK>], ['@',<PULSE_SEP>], ['smpp',<IDENTIFIER>], ['main',<
    IDENTIFIER>], ['(',<PAR_OPEN>], [':',<ARG_GROUP_SEP>], ['I16',<
    SIGNED_TYPE>], ['smp_in',<IDENTIFIER>], [':',<ARG_GROUP_SEP>], ['I16',<
    SIGNED_TYPE>], ['lp_out',<IDENTIFIER>], [')',<PAR_CLOSE>], ['{',<
    CURLY_OPEN>], ['lp_out',<IDENTIFIER>], ['=',<EQUAL>], ['I16',<SIGNED_TYPE
    >], ['(',<PAR_OPEN>], ['lowpass',<IDENTIFIER>], [')',<PAR_CLOSE>], [';',<
    TERMINATOR>], ['I16.2',<SIGNED_TYPE>], ['@',<PULSE_SEP>], ['smpp',<
    IDENTIFIER>], ['lowpass',<IDENTIFIER>], [';',<TERMINATOR>], ['lowpass',<
    IDENTIFIER>], ['=',<EQUAL>], ['lowpass',<IDENTIFIER>], [''',<INS>],
    ['-',<SUB>], ['(',<PAR_OPEN>], ['1',<NUM_LITERAL>], ['+',<ADD>], ['1',<
    NUM_LITERAL>], [')',<PAR_CLOSE>], [';',<TERMINATOR>], ['lowpass',<
    IDENTIFIER>], ['=',<EQUAL>], ['I16.2',<SIGNED_TYPE>], ['(',<PAR_OPEN>],
    ['smp_in',<IDENTIFIER>], ['+',<ADD>], ['(',<PAR_OPEN>], ['smp_in',<
    IDENTIFIER>], [''',<INS>], ['-',<SUB>], ['1',<NUM_LITERAL>], [')',<
    PAR_CLOSE>], ['>>',<SR>], ['2',<NUM_LITERAL>], [')',<PAR_CLOSE>], [';',<
    TERMINATOR>], ['lowpass',<IDENTIFIER>], ['+=',<COMBOP>], ['(',<PAR_OPEN
    >], ['lowpass',<IDENTIFIER>], [''',<INS>], ['-',<SUB>], ['1',<NUM_LITERAL
    >], [')',<PAR_CLOSE>], ['>>',<SR>], ['1',<NUM_LITERAL>], [';',<TERMINATOR
    >], ['}',<CURLY_CLOSE>], ['<EOF>',<EOF>]
```

This list is a simplified version of the ANTLR4 lexical analysis output where each pair surrounded by brackets is a token. The first element of the pair is the actual string that represents the token, and the second element is the token type.

4.2.2 Syntactic Analysis

After the tokenization, comes the syntactic analysis. This analysis uses the token stream created by the tokenizer and tries to match the sequence to the grammar rules. As the parser matches rules to the consumed tokens, it starts incrementally building a tree where each node represents a rule and the terminal nodes represent the tokens. This tree (the parse tree) is the output of the parser. Like the tokenizer, if the parser is unable to resolve rules in order to fit all the consumed tokens, the parser raises an error and the compilation process stops. The syntactic analysis, in the *rtfss* compiler, is performed by ANTLR and is the second compilation step. This step also represents the final step performed by the source code that ANTLR generated. Similarly to the tokenizer, the parser is fully compliant of the *rtfss* specification. The *rtfss* compiler's lexer has the following rules (the most straightforward rules will also be explained):

- `pulse_freq`: This rule represents pulse name, and matches the token `MAX`, `MAX` or `IDENTIFIER`;
- `varSizeType`: This rule represents a type of stream that has a variable dimension (signed, unsigned or float). It matches the tokens `SIGNED_TYPE`, `UNSIGNED_TYPE` or `FLOATING_TYPE`;
- `fixedSizeType`: This rule represents a type of stream that has a fixed dimension. So far *rtfss* only has `midi` as a fixed size type, so, it matches the token `MIDI_TYPE`;
- `data_type`: This rule represents any type of stream type. So it matches the matches the rule `fixedSizeType` or `varSizeType`;
- `stream_id`: This rule represents the use of a stream identifier, complete with the possibility of being indexed or having a time instant. So, it matches a `IDENTIFIER`, followed

by an optional `expr` (surrounded by a `BRAC_OPEN` and a `BRAC_CLOSE`), followed by an optional `expr` (that is preceded by a `INS`);

- `var_name`: This rule represents the use of a stream identifier, with the possibility of being indexed. So, it matches a `IDENTIFIER`, followed by an optional `expr` (surrounded by a `BRAC_OPEN` and a `BRAC_CLOSE`);
- `expr`: This is the main rule that matches stream expressions. It is the only rule that is recursive and, in the case of binary operators, it even is left recursive. ANTLR only allows left recursive rules when they are direct. This means that the recursion is made inside the rule and does not refer to another rule to do the circular reference. This rule handles the following operations: symmetric, both cast operations, CBlock short instantiation, stream bus initialization, property of, relative stream delay, shift left and right, rotate left and right, all bitwise logic operators, multiplication, division, sum, subtraction and all logical comparisons. Additionally, it also handles the `midi` stream operators note of, frequency of, velocity of, polyphonic aftertouch of, new note trigger of, controller of, program of, channel pressure of and pitch bend of. It also can match a `stream_id` or a `NUM_LITERAL`. Finally, it also matches an `expr` that is surrounded by `PAR_OPEN` and `PAR_CLOSE`. This last one is useful to set the precedence of operations. To better understand this rule, its declaration (in ANTLR) is similar to the following:

```
expr: SUB expr
    | PAR_OPEN data_type PAR_CLOSE expr
    | data_type PAR_OPEN expr PAR_CLOSE
    | cblock_inst_short
    | PAR_OPEN expr PAR_CLOSE
    | CURLY_OPEN (expr COMMA)* expr CURLY_CLOSE
    | PROP expr | expr GAP expr
    | expr (SL|SR) expr | expr (RL|RR) expr
    | expr AND expr | expr OR expr
    | expr XOR expr | NOT expr
    | expr MULT expr | expr DIV expr
    | expr MOD expr | expr ADD expr
    | expr SUB expr | expr LOGIC_OP expr
    | NOTEOF PAR_OPEN expr COMMA expr ARG_GROUP_SEP expr COMMA expr
      PAR_CLOSE
    | FREQOF PAR_OPEN expr COMMA expr ARG_GROUP_SEP expr COMMA expr
      PAR_CLOSE
    | VELOF PAR_OPEN expr COMMA expr ARG_GROUP_SEP expr COMMA expr
      PAR_CLOSE
    | PATOF PAR_OPEN expr COMMA expr ARG_GROUP_SEP expr COMMA expr
      PAR_CLOSE
    | NNTOF PAR_OPEN expr COMMA expr ARG_GROUP_SEP expr COMMA expr
      PAR_CLOSE
    | CCOF PAR_OPEN expr COMMA expr COMMA expr PAR_CLOSE
    | POF PAR_OPEN expr PAR_CLOSE | CPOF PAR_OPEN expr PAR_CLOSE
    | PBEND PAR_OPEN expr PAR_CLOSE
    | stream_id | NUM_LITERAL;
```

- `assign_op`: This rule symbolizes all possible assignment operators. So, it matches a `EQUAL` or `COMBOP`;
- `var_assign`: This rule specifies a stream assignment. It matches a `stream_id` followed by a `assign_op` and ends with an `expr`;
- `data_decl`: This rule specifies a stream declaration (and possible built-in assignment). It maches a `data_type` followed by the token `PULSE_SEP` and the rule `pulse_freq`. It then expects the rule `stream_id` to match and, optionally if there is an attribution, a `EQUAL` and a `expr`;
- `pulse_decl`: This rule represents a pulse declaration. So, it matches a `PULSE`, an `IDENTIFIER` and a `PULSE_LITERAL`;
- `cblock_inst_short`: This rule represents the short instantiation of a CBlock;
- `cblock_inst_short`: This rule represents the normal instantiation of a CBlock;
- `for_statm`: This rule specifies the structure of a 'for' statement. It matches the following tokens: `FOR`, `PAR_OPEN`, `IDENTIFIER`, `IN`, `expr`, `TO`, `expr`, optionally `INC` and `expr`, `PAR_CLOSE` and `code_block`;
- `if_statm`: This rule specifies the structure of an 'if' (with optional 'elseif' and 'else') statement. To match the part of the 'if', it expects a `IF`, `PAR_OPEN`, `expr`, `PAR_CLOSE` and a `code_block`. Then, if there is one or more 'elseif', it expects a `ELSEIF`, `PAR_OPEN`, `expr`, `PAR_CLOSE`, and a `code_block` for each of the 'elseif's. Finally, if there is a 'else' statement, then it expects a `ELSE` and a `code_block`;
- `nonterminated_statm`: This rule gathers all the statements that are not terminated (by a `;`). These are `if_statm` and `for_statm`;
- `terminated_statm`: This rule gathers all the statements that are terminated (by a `;`). These are `var_assign`, `data_decl`, `pulse_decl` and `cblock_inst`;
- `statm`: This rule gathers all kinds of statements (regardless of how they are terminated). So, it accepts a `nonterminated_statm` or a `terminated_statm`;
- `code_block`: This rule models the specification of a code block. A code block has a `CURLY_OPEN` followed by an arbitrary amount of `statm` and ends with a `CURLY_CLOSE`;
- `cblock_arg_list`: This rule serves as an auxiliary rule to match a arguments enumeration of a CBlock. It matches a (possibly empty) sequence of pairs of `data_type` and `var_name`. Between each pair, the rule expects a `COMMA`;
- `cblock_args`: This rule serves as an auxiliary rule to match two or three lists of arguments (`cblock_arg_list`). It matches three lists if the const list is present, or two if it is not. Between each `cblock_arg_list` there should be a `ARG_GROUP_SEP`;
- `cblock_decl`: This rule specifies the declaration of a CBlock. It expects a `CBLOCK` followed by `PULSE_SEP` and a `pulse_freq`. After that, the rule matches a `IDENTIFIER`, the rule `cblock_args` and ends with a `code_block`;

- `use_stat`: This rule models the import syntax. It expects the token `USE` followed by a `IDENTIFIER` or a `STRING_LITERAL`;
- `entry_point`: This is the main rule of the syntactic analyser. It serves as the entry point where it starts to match the input tokens. This rule expects an arbitrary amount of `use_stat`, followed by another arbitrary amount of `cblock_decl`. In the end, it expects a `EOF` token. This token is generated by the lexer, when no more characters are to be consumed.

In order to work with the ANTLR parse tree, ANTLR provides parse tree walkers and listener/visitor interfaces. In smaller projects, these facilities can be used as the foundation of a complete project, but since *rtfss*'s compiler requires complex manipulation of data structures, these tools alone will not suffice. So, only one visitor is implemented. This visitor is responsible for converting the output parse tree into the *rtfss* Abstract Syntax Tree (explained in the next Section). Furthermore, having only one visitor whose job is only to translate trees, reduces the dependency of the compiler on ANTLR. This means, that changing lexer and/or parser only requires changes to the tree translation code.

The ANTLR visitor used on the *rtfss* compiler responsible for translating the ANTLR parse tree into the *rtfss*'s Abstract Syntax Tree is a module called `parse_tree_translator`. The visitor requires that, for every syntactic rule of the grammar, there is a function to handle it. When the tree traverse starts, the top rule's function is called (in this case, the function is called `visitEntry_point`). Each function is then responsible for fetching the data present in the rule and for processing it. It then should decide which of the inner rules it should visit and in what order. As it visits the rules, it progressively builds the Abstract Syntax Tree and stores all the relevant info inside their nodes.

In this translation, some of the *rtfss* features start to be lost. Particularly, in this phase, all the `midi` stream operators have no correspondence to the *rtfss*'s Abstract Syntax Tree, so they are simply ignored by the translator. Also, while the import syntax has correspondence on the *rtfss*'s Abstract Syntax Tree, it serves no intrinsic purpose. The best place for imports to be handled would be even before the current tokenization and syntactic analysis, by a preprocessor. This preprocessor would have a grammar of its own to get only the `use` statements and handle them. Another place for them to be handled would be here, during translation. But, since the feature is not currently implemented, it is simply translated to an Abstract Syntax Tree node.

Support Example Analysis

The Syntactic Parse Tree generated by ANTLR4 while processing the example present in Section 4.1.2 is shown in Figure 4.2. In this tree, every non-terminal node represents an ANTLR grammar rule, and every single token lexical token is present. In this stage of compilation, every symbol in this tree is represented with strings.

4.3 *rtfss*'s Abstract Syntax Tree

After the creation of the parse tree, it is translated into an Abstract Syntax Tree. An Abstract Syntax Tree (AST) is a tree structure used to represent code. The main objective of an AST is to represent only meaningful connections and code structure. While a parse

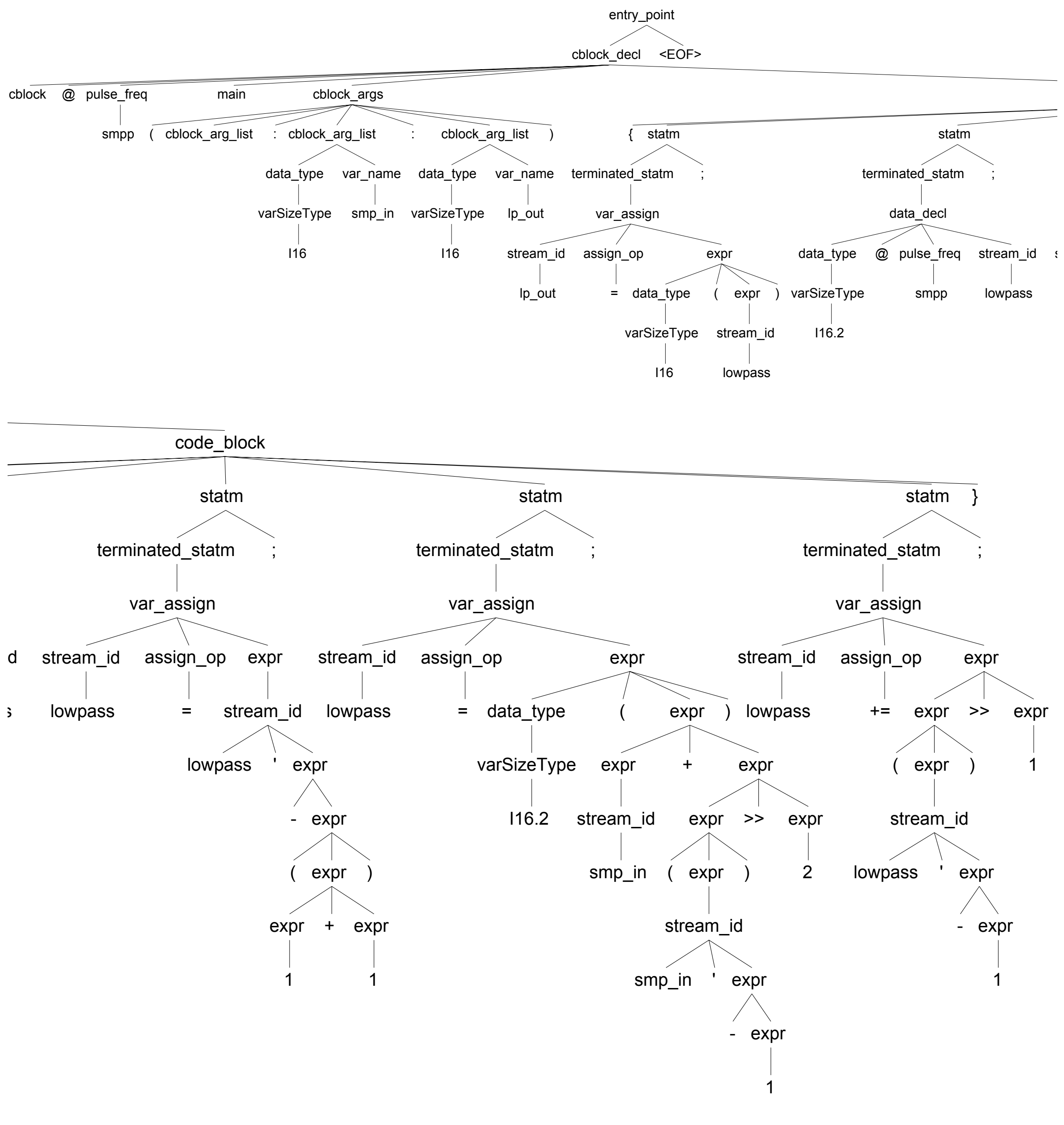

entry_point
cblock_decl <EOF>
cblock @ pulse_freq main cblock_args
smpp (cblock_arg_list : cblock_arg_list : cblock_arg_list) { statm statm
data_type var_name data_type var_name terminated_statm ; terminated_statm ;
varSizeType smp_in varSizeType lp_out var_assign data_decl
I16 I16 stream_id assign_op expr data_type @ pulse_freq stream_id
lp_out = data_type (expr) varSizeType smpp lowpass
varSizeType stream_id I16.2
I16 lowpass
code_block
statm statm statm }
terminated_statm ; terminated_statm ; terminated_statm ;
var_assign var_assign var_assign
stream_id assign_op expr stream_id assign_op expr stream_id assign_op expr
lowpass = stream_id lowpass = data_type (expr) lowpass += expr >> expr
lowpass ' expr varSizeType expr + expr (expr) 1
- expr I16.2 stream_id expr >> expr stream_id
(expr) smp_in (expr) 2 lowpass ' expr
expr + expr stream_id - expr
1 1 smp_in ' expr 1
- expr
1

Figure 4.2: Full Syntactic Parse Tree (split without overlap) generated by ANTLR4 (using the *rtfss* grammar) of the example of Section 4.1.2.

tree has every token that is found in the original tokenized text sorted by its grammar rules, in the AST, the tree is stripped only to the logical meaning of operations. This means that structures used in the parse tree's founding grammar are not present in this tree.

rtfss's Abstract Syntax Tree is one of the two main foundations of the *rtfss*'s compiler. It serves as the main data structure to the majority of the algorithms that precede the transformation of the tree to a graph representation. A node of the AST is represented by an object of a specific class. Each node type has a specific class that inherits from a base class. This base class, called `ast`, has the following main attributes:

- `std::weak_ptr<ast> parent`: A weak pointer to a `ast` that represents the parent tree node of the current node. If this node is the top node, `parent` points to `nullptr`;
- `ast_op op`: `op` is an object of the enum `ast_op`. This enum contains all possible types of `ast` child classes. This is useful for down-casts.

So, the way the tree is represented in memory is that each node has the list of pointers to the child nodes and a pointer to the parent. Since the programming language used for this compiler is C++, traditional pointers and memory allocations could be used. However, the use of these mechanisms is prone to memory leaks and other related issues. C++ offers an alternative: smart pointers. Smart pointers are data structures that act like classic pointers, but they have a tight control of the memory they point to. Specifically, for this compiler, shared pointers and weak pointers were used. A shared pointer (`std::shared_ptr`) keeps track of how many pointers point to a certain memory address (reference counter). When the counter reaches zero, it means no pointer is pointing to that memory location, so it deallocates it. Typically, this mechanism works out of the box for most uses. But, in tree structures that require a parent connection, an issue arises: cyclic dependencies. If a node has a pointer that points to another node, and this node has a pointer that points to the first node, neither nodes will be deallocated. This happens because the reference counters of each node are (at least) always one. The correct way to solve this is to have one of those pointers be a weak pointer. A weak pointer (`std::weak_ptr`) is a pointer which can be derived from a shared pointer. However, a weak pointer does not alter the state of the reference counter. So, from the point of view of a shared pointer, weak pointers do not exist. That is the reason why the parent pointer of the `ast` is a weak pointer, and not a shared pointer. All of the other pointers used in the nodes are shared pointers.

In this class, there are also auxiliary variables used to generate GraphViz drawings. This class has a protected constructor that restricts instantiation only to classes that extend this one. This is enforced because this class alone does not have any meaning, it is abstract. Moreover, this class also has pure `virtual` methods. These methods do not have a base implementation, so they force the succeeding extending classes to implement them. There are three pure `virtual` methods in the class that: get all the children from the current node, remove a certain child from the node, and duplicate the current node.

Beyond the node classes, some auxiliary classes were created to complement the main structure:

- `pulse_id`: This class parses and stores a pulse identifier from a string. It also distinguishes the type of the pulse;
- `datatype`: This class parses and stores the datatype of a stream;

- `time_scale`: This class parses and stores the timescale of a pulse;
- `assign_op`: This class parses and stores the assignment operator from an assignment statement;
- `logic_op`: This class parses and stores the logic operators used on stream expressions.

Not every class extends `ast` directly. There are three subtypes of AST that give specific functionality to the subsequent nodes. These three subtypes are `ast_arith`, `ast_unary_arith`, and `ast_bin_arith`. Similarly to `ast`, these classes are also interface classes, and not instantiable alone. The class `ast_arith` extends `ast`, and is extended by `ast_unary_arith` and `ast_bin_arith`. The special property these nodes have over the normal ones is that these can be used in arithmetic expressions. The class `ast_unary_arith` serves to be extended by the class who represents an unary operator. The class `ast_bin_arith` serves to be extended by the class who represents an binary operator.

The following classes extend the base class AST:

- `ast_start`: This is the top node of the tree, containing an attribute that represents the name of the input file and a `std::vector` of `std::shared_ptr<ast>`. This last attribute contains, essentially, all the CBlock declarations and include statements;
- `ast_include`: This node represents the inclusion of another source file. It contains the path to that file;
- `ast_cblk_decl`: This node contains a declaration of a CBlock. This object stores the name of the block, a `pulse_id` that represents the pulse associated with this CBlock and a `std::vector` of `std::shared_ptr<ast>` that represent the code associated with the CBlock. The interface signal and constants are also stored in this block and are composed of three `std::vector` of `std::shared_ptr<ast_id>` (const in, signal in and signal out);
- `ast_cblk_inst`: This node represents the instantiation a CBlock inside another CBlock. The structure of this node is similar to the one of `ast_cblk_decl`. It stores the name of the CBlock to be instantiated, the intended pulse (on a `pulse_id` object), two `std::vector` of `std::shared_ptr<ast_arith>` to represent the const and signal in (they do not need to be identifiers, expressions here are allowed), and finally a `std::vector` of `std::shared_ptr<ast_id>` to represent the signal outs (here only pure signal assigns are allowed);
- `ast_pulse`: This node represents the creation of a new pulse. This node can only be found inside a `ast_cblk_decl`. It holds a `pulse_id`, a `time_scale` (to represent the scale used in the pulse) and a `double` that represents the pulse value;
- `ast_if`: The `ast_if` node represents an if statement. This node can represent also elseifs statements and an else statement. So, this node stores the logical expression of the if in a `std::shared_ptr<ast_arith>`, a code block for the if statement in a `std::vector` of `std::shared_ptr<ast>`, and an optional "else" code block stored in the same way as the "if" code block. Finally, to represent elseif statements, a vector of pairs of logic statements and code blocks (vectors) are used;

- `ast_for`: The `ast_for` node represents a for statement. This node contains the "for" var name, a `std::shared_ptr<ast_arith>` to represent the lower bound, another to represent the upper bound, and another to represent the variable increment. Finally, it also has a `std::vector` of `std::shared_ptr<ast>` to represent the "for" code blocks;
- `ast_decl`: This node represents the declaration of a stream / const. A declaration can also have an assignment associated. This node stores a `std::shared_ptr<ast_id>` for the name of the stream and an optional `std::shared_ptr<ast_id>` for the attribution;
- `ast_assign`: Finally, this node represents an assignment to a stream / const. The representation is similar to `ast_decl`, but it also has a `assign_op` to represent the assignment operator.

As mentioned before, not every node inherits directly `ast`. The following classes extend directly `ast_arith`:

- `ast_id`: This node represents a stream identifier. It contains the stream name, a `datatype` to indicate the datatype associated with this node and a `pulse_id` to represent the pulse associated to this pulse. Furthermore, in order to represent an index, this node has a `std::shared_ptr<ast_arith>`. Since the index is a `ast_arith`, an expression is allowed. Similarly to the index, the node also stores if there is an instant associated with it (value at instant operator) with an `ast_arith`. All of the members except the name are optional;
- `ast_s_cblk_inst`: This node represents the short instantiation of a CBlock inside another CBlock. The structure of this node is very similar to the one of `ast_cblk_inst`. The only difference is that the final `std::vector` of `std::shared_ptr<ast_id>` does not exist;
- `ast_unary_arith`: Already explained above. Serves as a base class to unary operators;
- `ast_bin_arith`: Already explained above. Serves as a base class to binary operators;
- `ast_cast`: This class contains a cast. A cast object has two members: the target expression (`std::shared_ptr<ast_arith>`) and the target `datatype`;
- `ast_arrinit`: This class translates the initialization of a const array with a list of expressions. To achieve that, it has a `std::vector` of `std::shared_ptr<ast_arith>`;
- `ast_num`: This class allows the representation of a numeral. It is composed of a `union` of an `int` and a `double`, and a boolean value to indicate which of the representations is being used. The way the compiler represents numbers (and then manipulates them on constant calculation) is one of its main shortcomings. This will be detailed further ahead.

The nodes already listed are mostly functional. The arithmetic operators mainly fall into the `ast_unary_arith` and `ast_bin_arith` classes.

The class `ast_unary_arith` extends `ast_arith` and implements the base behaviour of an unary operator. It has one extra member: the target. This target is represented with a `std::shared_ptr<ast_arith>`. *rtfss* has three unary operators:

- `ast_sim`: Symmetric operator;
- `ast_prop`: Property operator;
- `ast_not`: Bitwise not operator.

The class `ast_bin_arith` extends `ast_arith` and implements the base behaviour of a binary operator. Instead of having only one member as `ast_unary_arith` has, this one has too: right and left. These pointers are of the type `std::shared_ptr<ast_arith>`. *rtfss* has multiple binary operators:

- `ast_add`: Sum operator;
- `ast_sub`: Subtraction operator;
- `ast_mult`: Multiplication operator;
- `ast_div`: Division operator;
- `ast_mod`: Modulus operator;
- `ast_shl`: Shift left operator;
- `ast_shr`: Shift right operator;
- `ast_rtl`: Rotate left operator;
- `ast_rtr`: Rotate right operator;
- `ast_and`: Bitwise and operator;
- `ast_or`: Bitwise or operator;
- `ast_xor`: Bitwise exclusive or operator;
- `ast_gap`: Instant gap operator;
- `ast_comp`: Logic comparisons operators;

Alongside the AST and all their classes, there were created some tools in order to ease the manipulation of the data structure: `ast_visitor` and `ast_property`.

The `ast_visitor` class implements the visitor design pattern on the AST. This class is of the abstract kind and has pure virtual methods. These methods should be implemented by an extender class. Each of these methods represent a node on the AST. As such, each of these methods also has as argument a pointer to a node. This class allows to easily traverse the AST freely and to process each node type with custom code. This class serves as the base of the algorithms that manipulate the AST (described in the following Sections).

The `ast_property` class is a templated class that implements the property design pattern in the AST. The main purpose of this class is to aggregate a certain data structure (the template argument) to the nodes of the tree. This allows a dynamic way of binding more data to the nodes, without needing to change the static structure of nodes. Its use arises when algorithms need to associate processed data to a certain node. This can be used as temporary storage or to serve as the output of an algorithm. As per the `ast_visitor`, this class also is used in most of the algorithms that manipulate the AST.

4.3.1 Support Example Analysis

The AST generated by the compiler while analysing the example present in Section 4.1.2 is visible on Figure 4.3. The black arrows represent the connections from an AST node to its children. The red dashed arrows represent the `parent` weak pointer. The uppermost node is the representation of the `ast_start` node, and contains the name of the file being compiled (`iir_filter.cb`). To save space, the names of the AST nodes were contracted. Each AST node (non-literal) is marked in bold. It is relevant to observe that the tree does not contain any structural tokens, every token has a semantic meaning. The children of the AST nodes are identified with the property type on their connection edges.

4.4 Constant Stream Solver

After the AST creation is done, processing can begin. The first processing step that the compiler does is to solve most of the constant operations. This process not only helps the compiler optimize the hardware logic generated further ahead, as it is also a crucial step for solving stream instants (explained further ahead). The module responsible for this task is called `const_solver`.

The `const_solver` class implements the AST visitor class. This module performs tree branch substitution and it uses four AST properties to keep track of the state of a certain branch/node:

- `is_const`: This (boolean) property specifies if a node is solved to a constant value;
- `is_int`: This (boolean) property is only valid if the value of the `is_const` to the target node is true. It specifies if the node is an integer or a double number;
- `int_res`: This (integer) property is only valid if the value of the `is_int` to the target node is true. It contains the integer result of the target node;
- `double_res`: This (double) property is only valid if the value of the `is_int` to the target node is false. It contains the double result of the target node.

As it traverses the AST, the module behaves differently for each type of node.

If the node is not an arithmetic node or if the node is unsolvable by the module, then it checks if each of the children nodes are solved. For each node that is solved (obtained by checking the `is_const` of that children node), the compiler cuts that children node (and everything under it), and replaces it with an `ast_num` with the correct solved value. This solved value is obtained from checking `is_int` of the target node, and then based on the boolean returned by it, retrieve the correct value from `int_res` or `double_res`.

If the node is an arithmetic node, it checks if the operands are solved (check `is_const`). If at least one of the operands is not solved, then the compiler treats the current node as an unsolvable node (explained above). If they are solved, then the solver calculates the result of the operation. In this stage, the stream arithmetic rules stated in Section 3.3.6 are *mostly* followed. In the current implementation of the compiler, there is one type of rule that is not followed: sizing rules. Ideally, this module would work with a fixed point representation, when the stream type used is fixed-point, and double/float representation, when floating-point representation is needed. Instead, the compiler currently does calculations on 64-bit integers

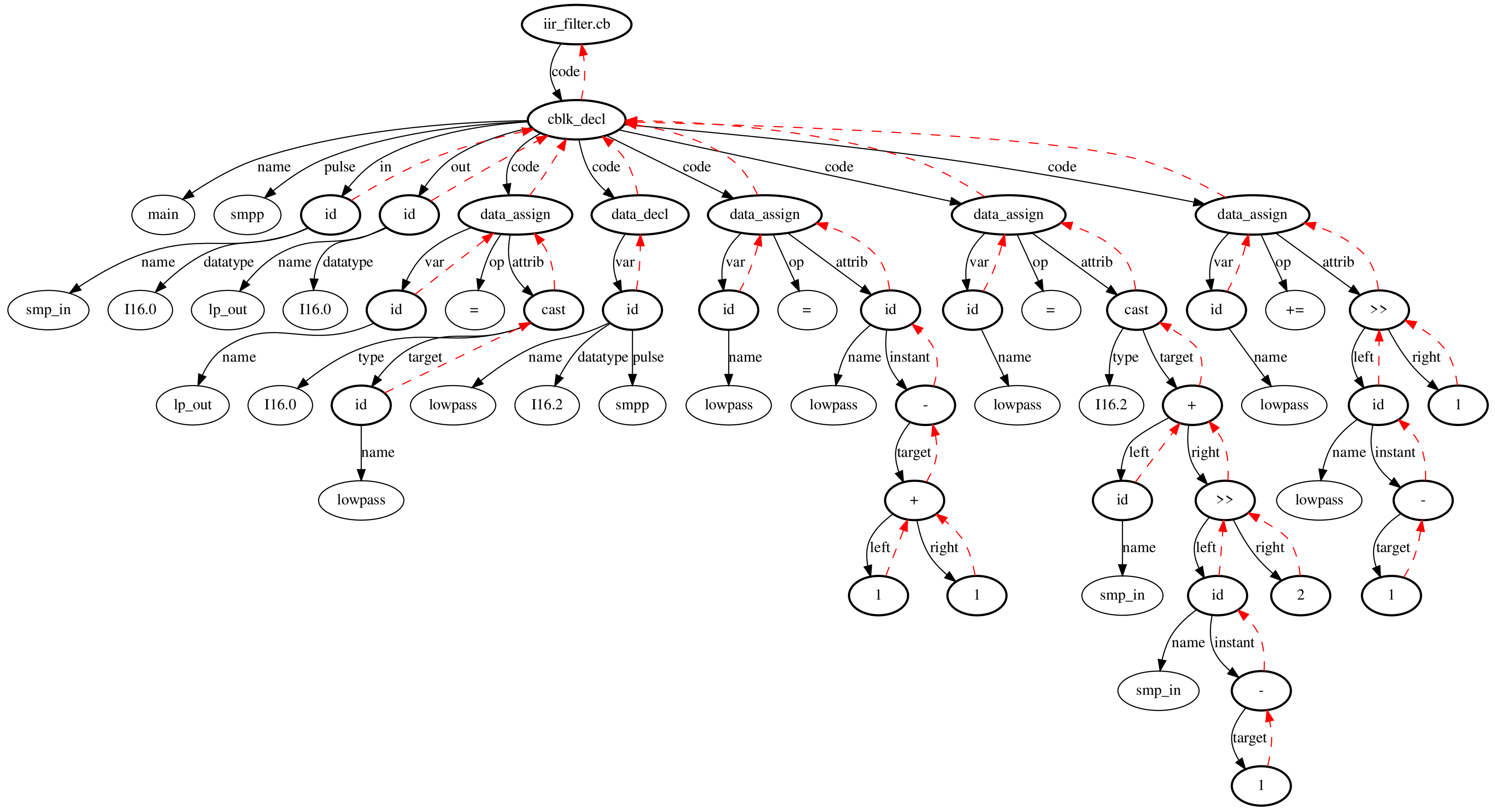


Figure 4.3: Full initial Abstract Syntax Tree of the example of Section 4.1.2 (GraphViz).

and doubles. This means that, sometimes, the theoretical result of an operation might not be the same one as the one expected. In these cases, the compiler warns the programmer with a compilation warning. After calculating the value of the current node, the node is set to const on `is_const`, the flag `is_int` of the node is set and the value is placed under `int_res` or `double_res`. The only kind of arithmetic node that is always const is the `ast_num`. When the compiler finds one, it fills the corresponding AST Properties with the node values.

As mentioned before, not all arithmetic nodes are solved. The node that are not solved on this stage are the following:

- `ast_shl`, `ast_shr`, `ast_rtl`, `ast_rtr`: there is no need to solve them, since they do not imply a hardware delay;
- `ast_gap` and `ast_prop`: at this stage of compilation, the compiler does not have enough information to solve them;
- `ast_cast`: due to the way this module is implemented, this node has no effect in this stage, so it is simply dealt as a non arithmetic;
- `ast_id`: This node cannot be resolved, because it represents another stream, and the module only does calculations within the same statement;
- `ast_and`, `ast_or`, `ast_xor`, `ast_not`: these nodes are solved by the compiler, when they are fully integer. Otherwise, they are seen as non arithmetic.

The `const_solver` module does not return any data structure, since all the operations done in it are done in place on the AST feed on construction. When this module finishes, the AST will be ready to be used by the subsequent modules.

4.4.1 Support Example Analysis

Looking again at the support example (explained in Section 4.1.2), after it is processed by this block, we can see some changes (Figure 4.4). Some AST nodes have vanished and some new have appeared. The first change is on the first assignment to the `lowpass` audio stream. Here, considering both of its operands are constant, the arithmetic operation is solved. Then, once again, since the value of the operation is const, the symmetric operator was also solved (Figure 4.5). This last step is also done on the other two *value at instant* indexes (Figure 4.6 and 4.7). No more operations are purely constant, so the compiler ends this stage.

4.5 Stream and Pulse Solver

After having the const expressions of the AST sorted, the next processing step the compiler does is discovering all streams and pulses declared and start to map them. This operation is done by a module called `var_solver`.

The `var_solver` class implements the AST visitor class. In order for the module to solve pulses and stream identifiers, it has to handle identifier visibility. For example, streams/pulses declared in one CBlock are not visible in another CBlock. This concept is called scoping. Different AST nodes have different scoping rules for streams and for pulses. A node can have three types of scoping:

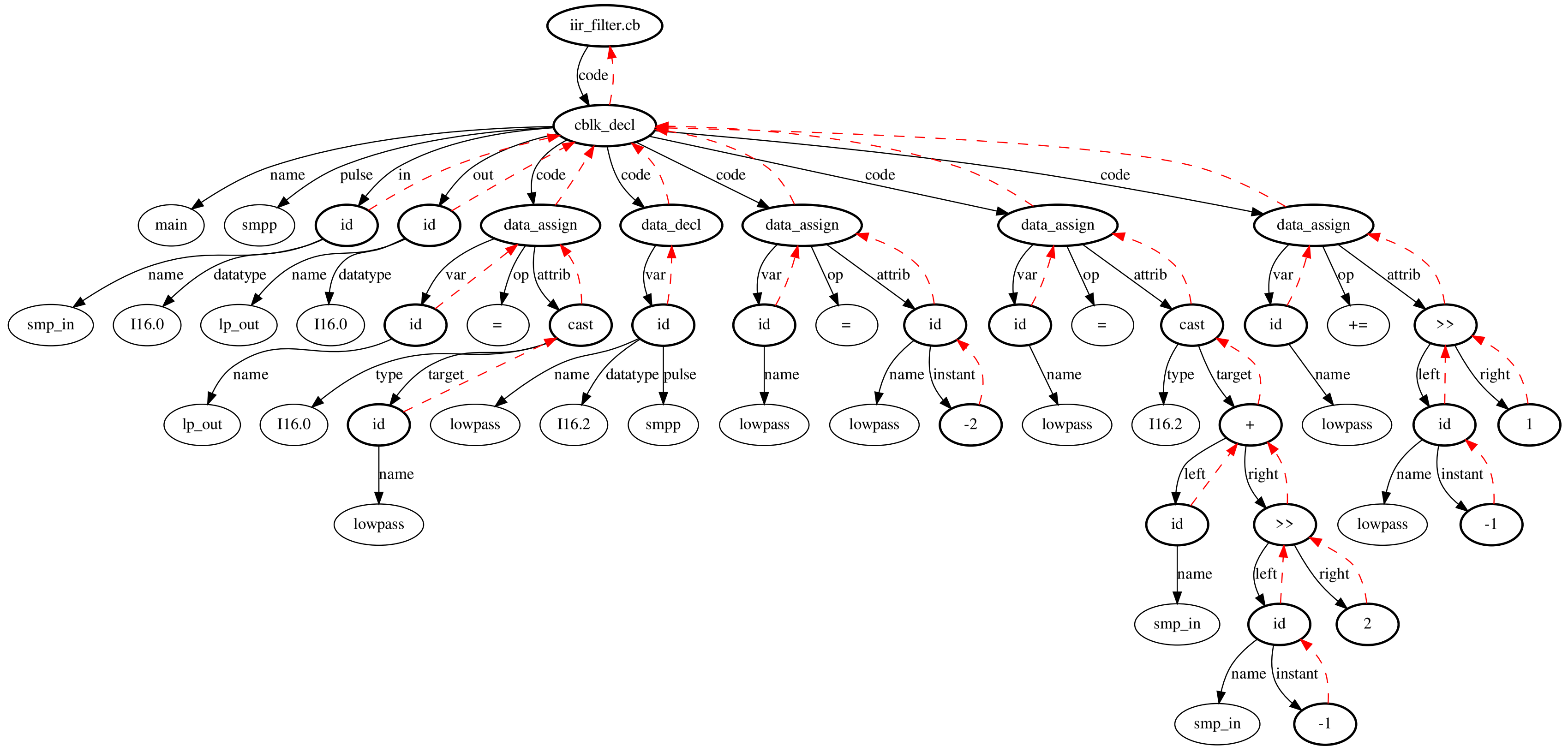


Figure 4.4: Full Abstract Syntax Tree after Constant Stream Solver of the example of Section 4.1.2 (GraphViz).

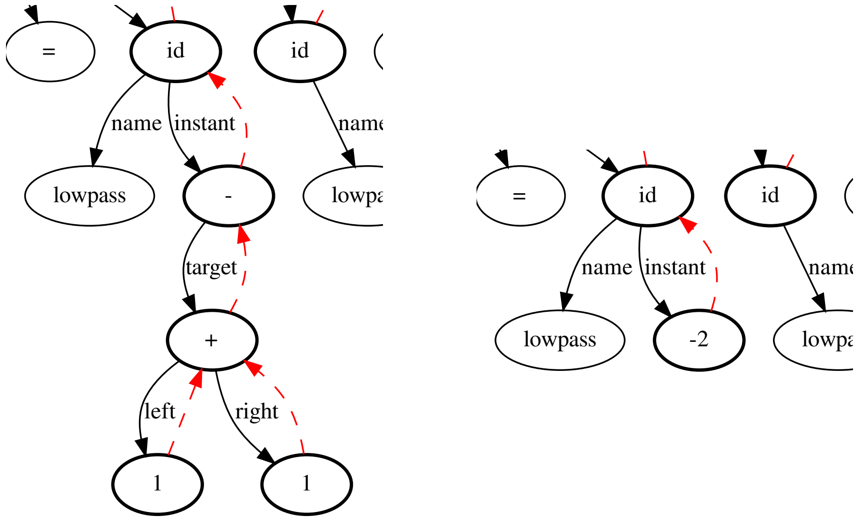


Figure 4.5: Detail 1 on the before and after modifications done on the Abstract Syntax Tree by the Constant Stream Solver on the example of Section 4.1.2 (GraphViz).

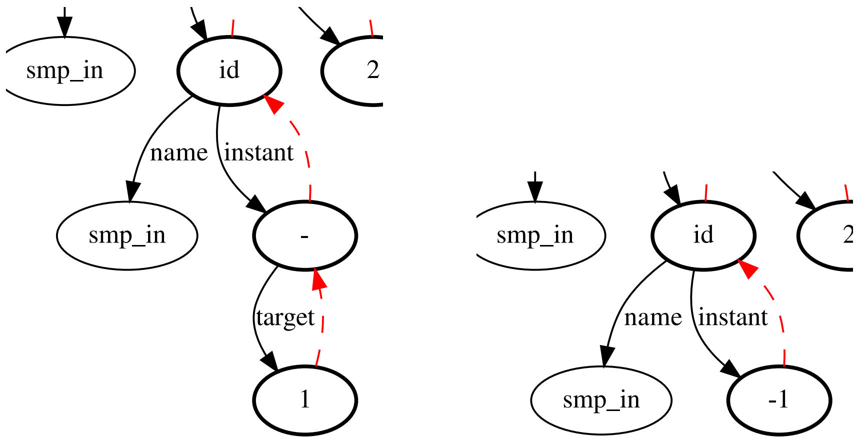


Figure 4.6: Detail 2 on the before and after modifications done on the Abstract Syntax Tree by the Constant Stream Solver on the example of Section 4.1.2 (GraphViz).

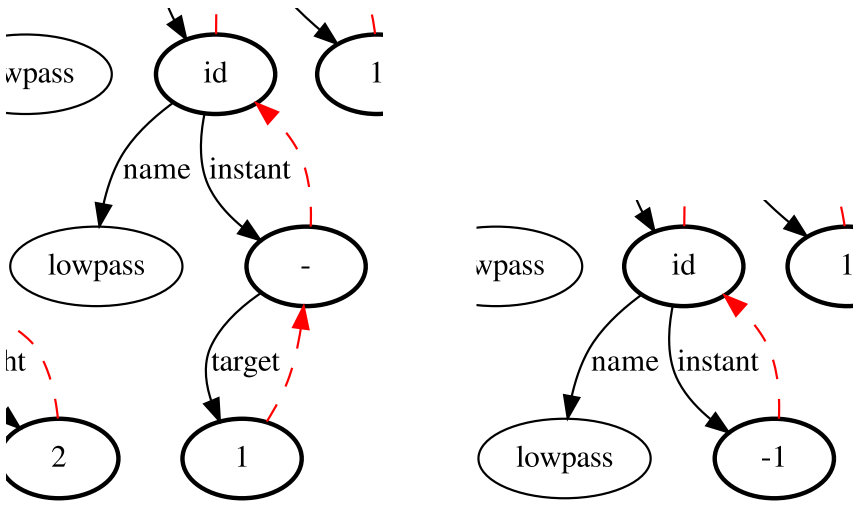


Figure 4.7: Detail 3 on the before and after modifications done on the Abstract Syntax Tree by the Constant Stream Solver on the example of Section 4.1.2 (GraphViz).

- Pass-through scoping: The current node is not a scope frontier, so the outer scope is passed to the inside;
- New scoping: The current node is a frontier to a new scope, so the outer scope is duplicated and is fed to the child nodes. This means that variables declared on the outside will be visible in the inner scope, but not the other way around;
- Hybrid scoping: The current node is a combination of pass-through scoping or different new scoping to the child nodes.

In order to represent the scopes, complementary data structures were created. Firstly, there is a class to hold the properties of a pulse. This class, called `pulse_property`, stores the type, identifier, time scale and value of a pulse. Secondly, there is a class to hold the properties of a stream. This class, called `var_property`, holds the name, unique id, datatype and pulse identifier of a stream. Both of these classes fully store all information related to pulses and streams needed for the compilation process. The main objective of the `var_solver` is to extract information and create objects of those two types. To represent a scope, a simple vector of those classes can be used.

Besides filling those structures, this module is also responsible for stamping all the expression AST nodes with extra information, using objects from the class `ast_property`. The module marks stream expressions with the pulse identifier that is being used. This is stored in an object that will be referred to as `pulse_id`. It also marks stream expressions with the stream name that is being assigned to, and in the cases of `ast_id` nodes, it points to their identifiers. This is stored in an object that will be referred to as `var_id`.

Since this class implements the AST visitor, each of the pure virtual function offered by the visitor has single scoping type to the streams and another for pulses. Most of the nodes follow the pass-through scoping type. The following nodes do not abide by that rule:

- `ast_cblk_decl`: CBlock declaration is a case where both pulse scopes and stream scopes use new scoping. This prevents the mixture of streams and pulses from other CBlocks;

- `ast_if`: This node uses hybrid scoping. In this node, each inner code block has its own pulse scope and stream scope;
- `ast_for`: This node also uses hybrid scoping. The stream scope is duplicated and the "for" variable is added to the inner scope. The pulse scope is also duplicated from the outer scope.

In this module, at the beginning of the traverse (`ast_start`), two pulses are added to the global pulse scope (the first scope has complete visibility). These two pulses are `const` with pulse id 0 and `max` with pulse id 1. While the tree is traversed and scopes are being filled, the operation of inserting into a scope also registers that pulse/stream into another list. This list is the complete collection of the streams (list called `vscopes`) and pulses (list called `pscopes`). If the input code attempts to redeclare a stream/pulse already declared in that scope, the compiler raises an error and halts.

After traversing the tree, the compiler checks if all the identifiers used for pulses and streams are declared. If a stream or pulse is not declared, the compilation process prints an error and halts.

Finally, if all operations finish successfully, the module returns the two complete collection of streams and pulses (`vscopes` and `pscopes`), and the data aggregated to the AST (`pulse_id` and `var_id`).

4.5.1 Support Example Analysis

We will now analyse the behaviour of this module when it processes the support example (explained in Section 4.1.2). When the module starts, as already explained, it places the `const` and `max` pulse types on the `pscopes` list with identifiers 0 and 1 respectively. Then, it detects the existence of the `smpp` pulse on the CBlock declaration, so it also places it on the `vscopes` list with the pulse identifier 2.

As the module traverses the CBlock it also finds, right at the beginning, both `smp_in` and `lp_out` audio streams. These streams are registered on the `vscopes` list with the stream identifiers 0 and 1 respectively. The datatype and pulse identifier of these streams is also registered on the `vscope` list. Finally, on the CBlock body, the module finds the `lowpass` stream and stores it inside the `vscope`. The identifier of this stream is 2.

During the building of these structures, the module did not find any abnormalities (such as redeclaration of streams / pulses, among others), so the compilation process keeps executing. The resulting internal structure of `vscopes` and `pscopes` is the following:

```
vscopes:
  {unique_id:0,name:smp_in,var_kind:IN,datatype:I16.0,pid:2,decl:1}
  {unique_id:1,name:lp_out,var_kind:OUT,datatype:I16.0,pid:2,decl:1}
  {unique_id:2,name:lowpass,var_kind:REGULAR,datatype:I16.2,pid:2,decl:1}

pscopes:
  {unique_id:0,pid:const,pulse_kind:UNKNOWN,decl:1}
  {unique_id:1,pid:max,pulse_kind:UNKNOWN,decl:1}
  {unique_id:2,pid:smpp,pulse_kind:CBLOCK,decl:1}
```

4.6 Assignment Dependency Checker and Trimmer

The next stage of the compilation process is to identify dependency stream assignment loops and to remove redundant/unused assignments. The module responsible for those tasks is called `assign_trimmer`. Similarly to the last module, this one is also based on a `ast_visitor`. At the current stage of the implementation of the compiler, this module, while processing, ignores `if` and `for` statements. Also, stream buses and CBlock instancing (short or normal) are disregarded.

In order to find dependency cycles and redundant/unused assignments, the first step is to traverse all assignments. However, due to *rtfss*'s assignment rules, special precautions need to be put in place.

rtfss allows multiple assignments to be made to the same stream, where the last one is the dominant assignment. If that last stream is not dependent from previous assignments to that stream, in other words, does not reference itself, all the previous assignments are ignored. But if that is not true, the previous assignment is then used and the method is repeated. This rule is used as much as needed, until reaching the first assignment. If the first assignment is reached and there is still a self-dependency, the input code is malformed. The other *rtfss* rule for assignments is that the order between assignments of different streams is irrelevant.

To represent the dependencies, `assign_trimmer` has a vector of sets, in which the index of the vector represents the stream identifier of a certain stream, and the set is the collection of stream identifiers this stream depends on. Also, to keep track of the current streams' dependency while traversing expressions, a `ast_property` is used.

So, having these *rtfss* restrictions in mind, the easiest way to solve this task is to traverse all the nodes inside the CBlocks in reverse order. This way, the dominant statement of each stream is visited first and the least dominant come in natural order. Every time an assignment is reached, the compiler first checks if the assignment is for a default stream past value. If it is, it checks if there was already a declaration on the same past value. If there is, the compiler raises a warning to inform the programmer that this assignment will be ignored, and cuts it. Either way, this assignment will not be traversed, because it will not be relevant to this module. It cannot generate cyclic dependencies, so it skips it. Otherwise, if the assignment is not to a default past value, the compiler checks the dependencies of that stream:

- If the stream dependency set is empty, meaning that this is the first time checking this stream, the compiler visits the children of the stream in order to fill the dependency set;
- If the set is not empty but the identifier of this stream is also in the set of dependencies (meaning that the last time visiting an assignment of this stream, there was a self-dependency), the compiler removes the identifier from the set and traverses the children (to fill more dependencies);
- If the stream has dependencies, but neither of them are to itself, it means that the stream's dependencies are already satisfied and that this assignment is redundant. If this is the case, the compiler simply trims (cuts) this assignment from its holder (normally, a CBlock), and moves on without travelling through the child nodes.

If, while visiting the children of an assignment, a reference to a stream (`ast_id`) is found, it is added to the dependency list of the current assignment. However, if the reference to the stream is for a past value, then it is ignored. This is due, again, to the fact that it is read-only and cannot generate dependency loops.

To better understand the concept, the following code block is the revisited and modified example shown in Section 3.3. After each expression, a visual representation of the dependency set of each stream is stated:

```
//Note: the code is read from the bottom to the top
I8.0@p a=a+1;   //a:{a}    b:{c}    c:{d}    d:{b,c}
I8.0@p b=c+1;   //b:{c}    c:{d}    d:{b,c}
I8.0@p d=a;     //c:{d}    d:{b,c}
d=b;            //c:{d}    d:{b,c}
I8.0@p c=d/3;   //c:{d}    d:{c,d}
d+=5+c;         //d:{c,d}
```

The final dependency sets of this segment of code are `a:{a} b:{c} c:{d} d:{b,c}`. It is worth to note that, since `d=b` does not contain self dependencies and that it comes after `d=a`, then `d=a` will not have any effect on the dependencies and the compiler will cut that expression from the code. It is also relevant to see that the statement `d+=5+c` creates a self dependency on `d` that is solved in `d=b`. This code will not compile for various reasons. Firstly, the stream `a` is not completely solved, because it has a self dependency. Secondly, stream `b` depends on stream `c`, which depends on `d`. Stream `d` then depends both on `b` and `c`, so there are more than two cyclic dependencies there.

Since this module does in-place modification of the AST, once this module finishes operations, nothing is returned.

After finishing traversing all the tree, all dependency sets are now complete and the redundant assignments have been erased. However, not everything is done yet. The compiler also needs to check for dependency cycles. It is worth noting that the way the dependencies were stored resembles an adjacency list. So, if we interpret them as an adjacency list, we get a directed graph where the nodes are streams and the connection from a stream A to a stream B means that stream A depends on stream B. Knowing this, the compiler can run a depth-first traversal. However, since the graph is not connected, meaning that there can be sub-graphs that are not connected to each other, the compiler needs to assume a worst-case scenario and run the algorithm where every node is a possible start node. If, while traversing the dependency graph, the compiler finds a node that it has already visited, then the compiler knows that there is a dependency cycle there. In this case, the compiler halts compilation and throws an error.

4.6.1 Support Example Analysis

Now, we shall analyse the effects of this module on the support example (explained in Section 4.1.2). The AST of the example when it exits this module is represented at (Figure 4.8. As we have already concluded, this example has a redundant attribution. The first assignment to the `lowpass` audio stream is ignored because the next statement to the `lowpass` audio stream does not depend on this one. So, this module detects that redundancy and erases the attribution (Figure 4.9).

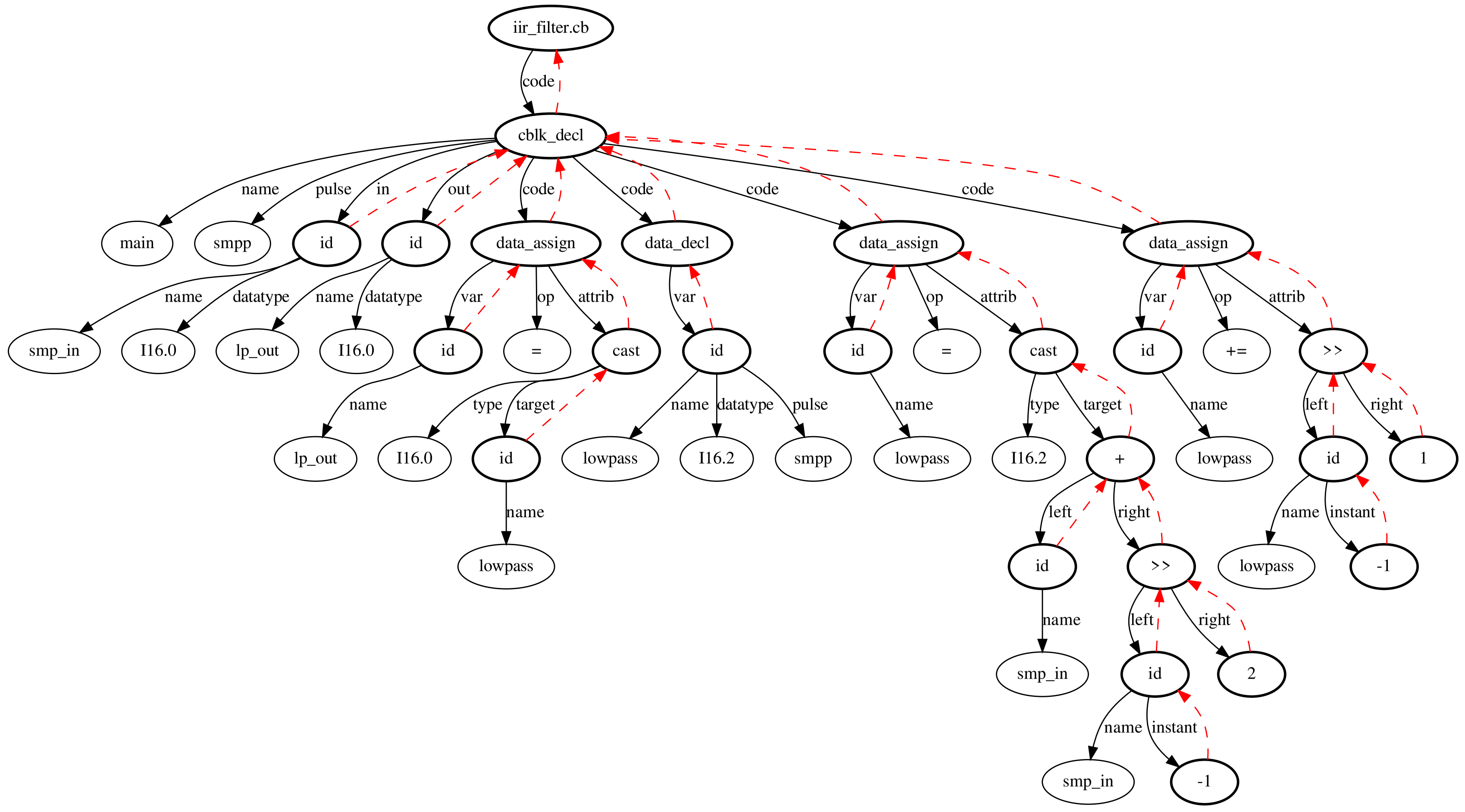


Figure 4.8: Full Abstract Syntax Tree after Assignment Trimmer module of the example of Section 4.1.2 (GraphViz).

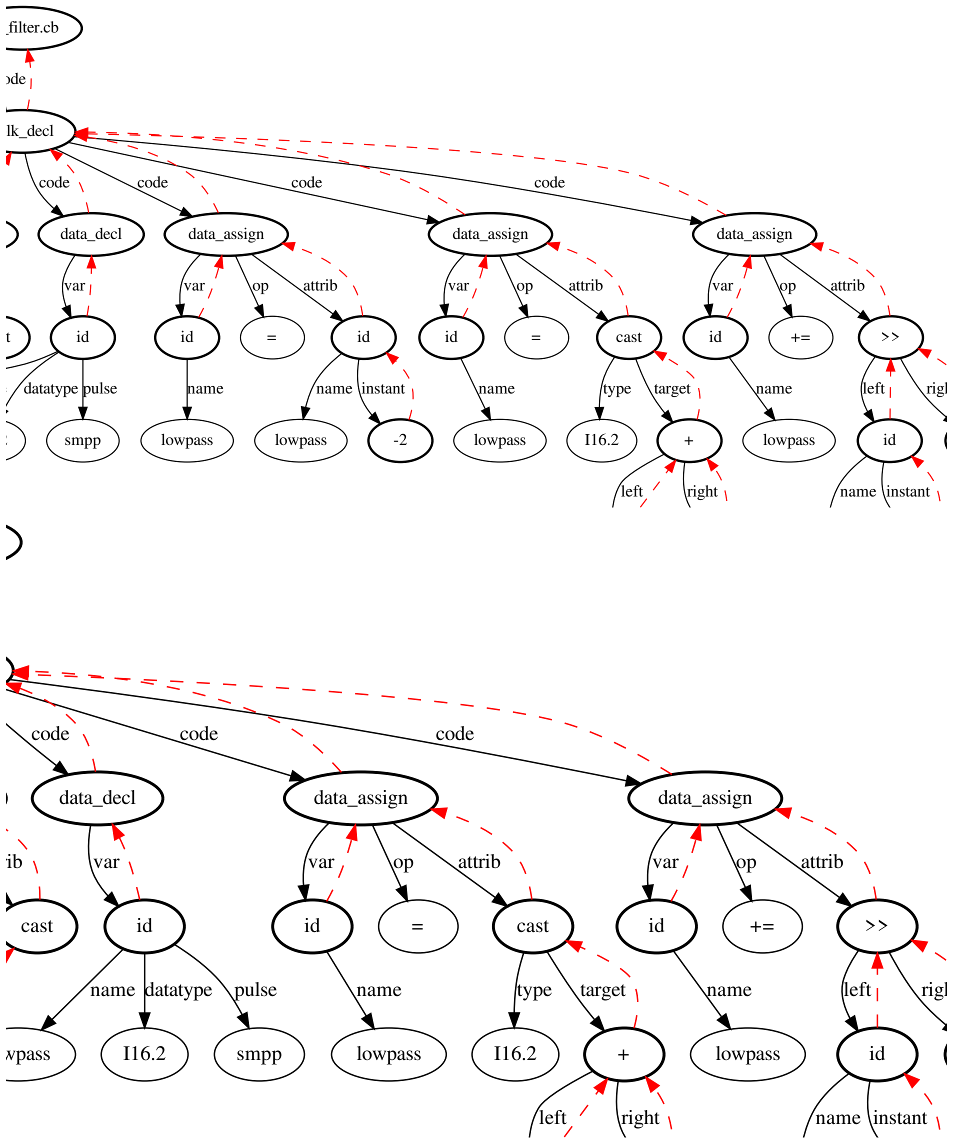


Figure 4.9: Detail 1 on the before and after modifications done on the Abstract Syntax Tree by the Assignment Trimmer module on the example of Section 4.1.2 (GraphViz).

4.7 Assignment Mover

The `assign_trimmer` is the first module that modifies the AST, in order to remove unwanted data. Now that all the unused assignments have been removed, the compiler can still slim down the AST. Moreover, since now the compiler checked that there are no stream cyclic dependencies, the multiple assignments of each stream can be combined into one full assignment that contains the complete expression intended for each stream. The module, `assign_mover`, is responsible for that action. Similarly to the last module, this module, while processing, currently ignores `if` and `for` statements. Also, stream buses and CBlock instancing (short or normal) are currently disregarded.

The main objective of this module is to move the fragments of the streams from the least dominant assignments to the more dominant assignments. To do so, the module traverses the tree and keeps track of the most recent assignments to each stream. Similarly to the `assign_trimmer`, this module is also based on the `ast_visitor` to traverse the AST. But, this time, the tree should be traversed without reversing the visit order of the nodes inside CBlocks. This is due to the fact that every assignment present in the tree is valid and if, while traversing, the compiler finds an assignment to a stream that has already been visited, it needs to know where the previous one is, to move expressions. This information is stored in a vector of shared pointers to `ast_arith`, called `assign`. The indexes of this vector are the stream ids. Similarly, this module also has another vector, but this time of booleans, to represent if the given expression of each stream id is attached to an assignment, or instead is detached. This vector is called `detached`. Finally, this module uses the `var_id` generated by the `var_solver` module to know the identifiers of each stream.

When the compiler traverses the tree, the first thing it does when it enters an assignment is to check if the assignment is made to a stream instant. If it is, and the stream instant is valid (less than zero), then the assignment is ignored. Otherwise, the assignment is processed. Inside the assignment, the compiler checks if the assignment operator is the equal (`=`). If it is not, the compiler has to unfold the operator (`+= -= *= /= %=`) into their intended binary operation, adjust the operation size, and make the assignment operator an equal. To do this, the compiler creates a new corresponding operation node and pushes the current assignment node to the right operand of the new node. For the left operand, it fetches the previous assignment from `assign`, detaches it (sets this node to false on `detached`), duplicates it and sets it to the left operand. It then creates a cast node, in order to fulfil the stream's sizing requirement and sets its target to the new operation node. Finally, it sets this cast node as the assignment target node. In the process, the compiler created a dependency (that will be handled later). Now, the compiler will traverse through the assignment's children.

For each arithmetic operation the compiler travels through, it checks if any of the operands is an `ast_id`. If it is, it then has to check if that node represents a stream or a stream instant. If it is a (valid) stream instant, then this node is ignored. Otherwise, it then checks (through `var_id`) if that node corresponds to the same stream that the compiler is processing. In case of any of the previous conditions is not met, the compiler simply continues the traverse. In the case of both stream identifiers being the same, it means that the operand has to be replaced by the previous assignment of that stream. To achieve this, the compiler gets the replacement expression from `assign`, and checks if it was already detached (variable `detached`). If it was not detached, it sets it as detached and duplicates it. If already detached, it just needs to duplicate it. Now, the compiler swaps the `ast_id` node with the expression it just fetched. Duplication is always needed, in order to keep the AST a tree, and not a graph.

Due to the way `ast_property` is built, every time a node or expression is duplicated, all the classes that implement a `ast_property` become outdated and contain wrong information. So, every time the compiler duplicates an expression, it also duplicates in all the `ast_property` the properties that refer to the duplicated nodes. It also removes the old (unreachable) properties of the original detached expressions, when they are not needed anymore.

After finishing traversing a declaration/assignment, the compiler checks if the previous declaration was detached while traversing the children. If it was, then the compiler erases that expression. It then sets itself as the most recent expression for that stream on `assign` and sets the current stream's `detached` value to true.

After the traverse is finished, the AST has only complete assignments and no more self dependencies chains. Since this module also does in place manipulations on the AST, it does not return any data.

4.7.1 Support Example Analysis

Let us look back again at the support example (explained in Section 4.1.2). After it is passed through the Stream Mover module, a relevant modification is made (Figure 4.10). After this module, both remaining assignments of the `lowpass` audio stream are combined into a single assignment (Figure 4.11). To do so, the compiler expands the `+=` operator and creates a cast node. Then, it moves the first assignment attribution branch to the left operand of the new sum operation, and the original attribution of the second assignment to the right operand. Finally, it connects that sum node to the cast and places it on the last attribution. The now empty attribution is discarded.

4.8 *rtfss*'s Architecture Graph

In the current stage of the compilation process, the AST is as slim and as concise as it can be. It is now time to take the first step down towards the hardware level representation. This first step is done by generating an architecture graph.

The architecture graph is a middleware representation of a *rtfss* design. As the name suggests, it is a graph representation where all its nodes have relatively close relation with a hardware counterpart, but the logic is still maintained in a somewhat high-level. In other words, this representation is purely arithmetic. This graph serves as the compiler's second backbone data structure. All the operations from this stage of compilation forward will be made in the architecture graph. A key aspect of this representation is that, although the name implies that it is only one graph, the compiler implicitly segregates the graph for each pulse. This makes the operations be separated by pulse domains.

Since this data structure is a graph, it cannot be (classically) represented simply using nodes and have them point to the parents and children, like in the AST. Graphs can have multiple nodes without parents, so it requires a different representation. Classically, one way of representing graphs is using an adjacency list. An adjacency list is a list of lists, where the index of the first list represents a node and the indexed list is the connections of that node. The architecture graph uses this data structure. However, it is not the main data structure of the architecture graph. The main structure is a slightly different version of this kind of adjacency list. It is an inverted adjacency list. This means that the list that a node "holds" is the list of nodes that connect to it, not the nodes it connects to. This representation was selected to satisfy the fact that the architecture graph is a collection of subgraphs of pulse

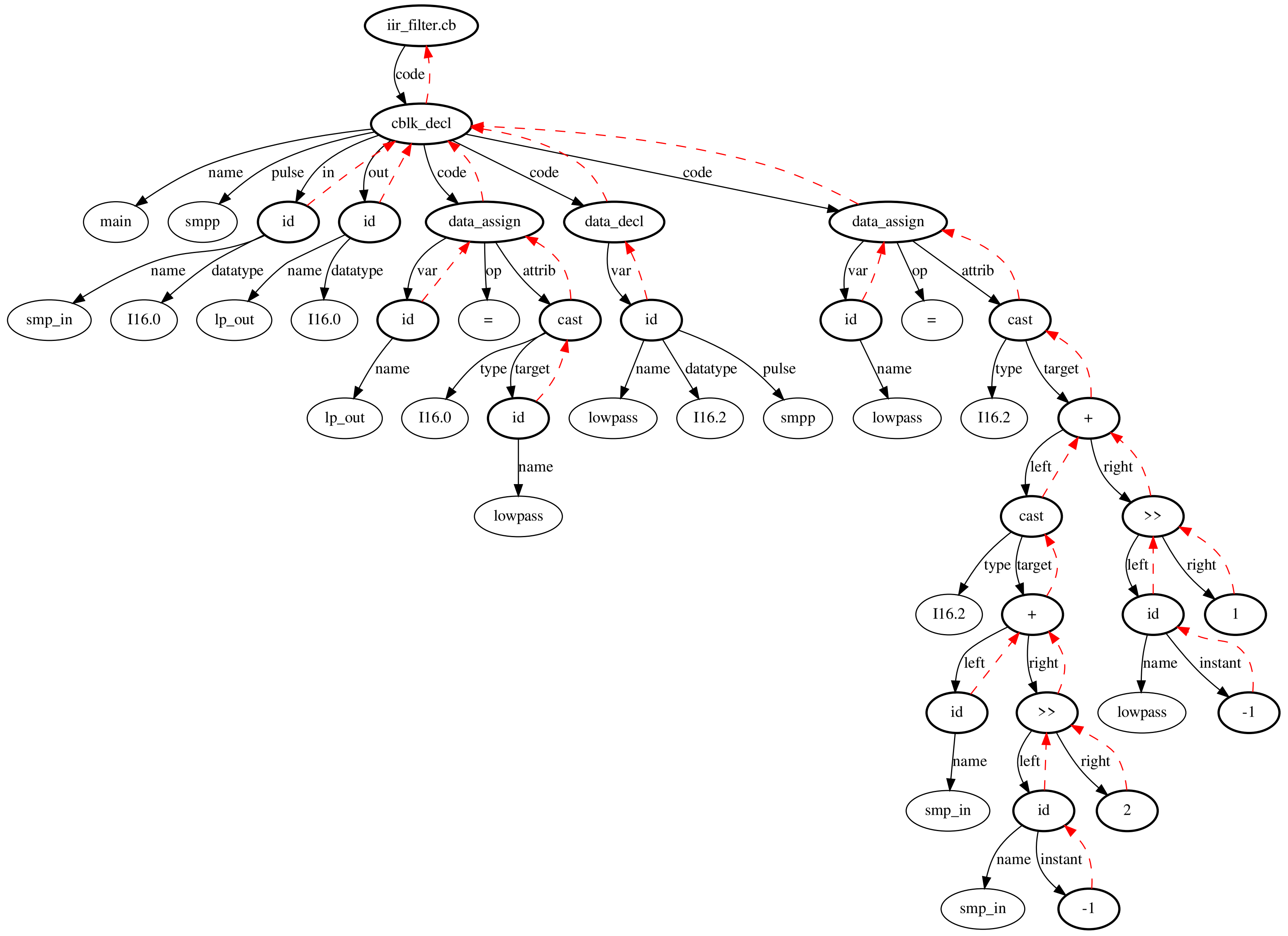


Figure 4.10: Full Abstract Syntax Tree after Assignment Mover module of the example of Section 4.1.2 (GraphViz).

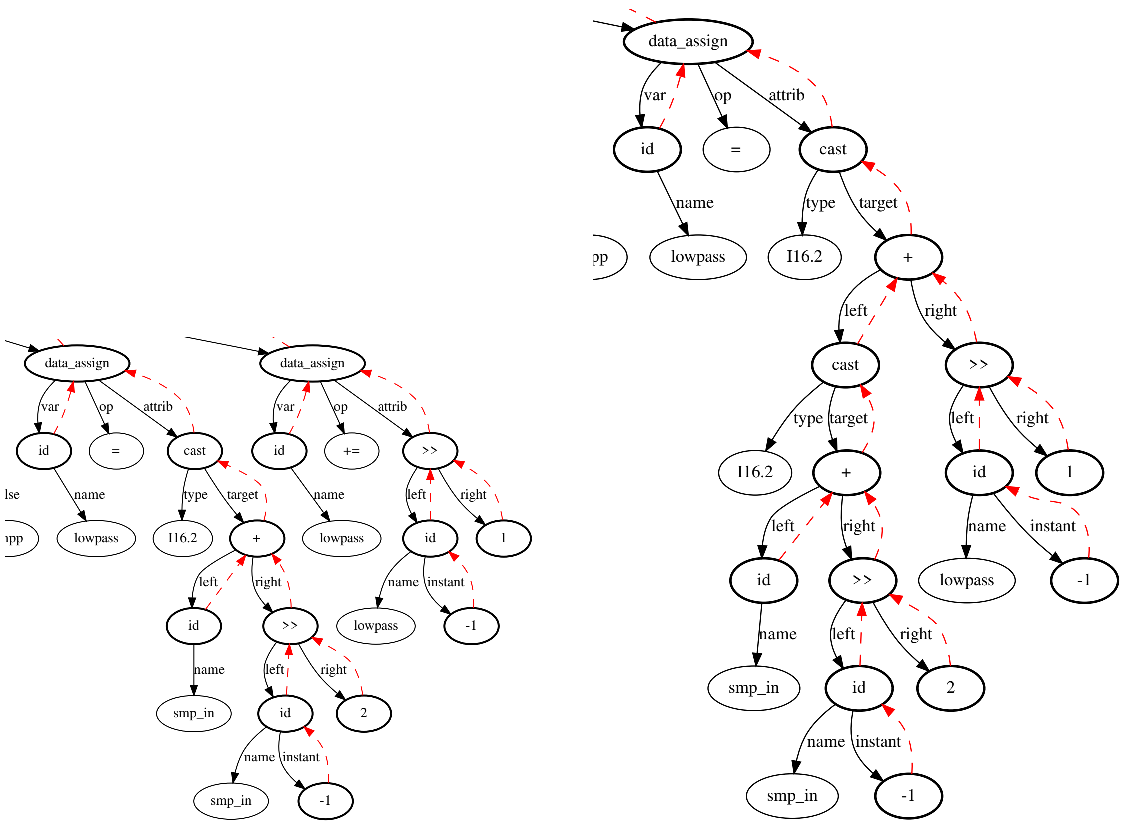


Figure 4.11: Detail 1 on the before and after modifications done on the Abstract Syntax Tree by the Assignment Mover module on the example of Section 4.1.2 (GraphViz).

domains. A node belongs to a pulse domain, so it makes more sense to know the nodes that connect to a certain node, than the other way around. Having these two redundant data structures (normal and inverted adjacency list), allows for more time optimized processing.

The architecture graph module is called `arch_graph`. It stores the inverse adjacency list (`inv_adj_list`) and the normal adjacency list (`adj_list`) as a mapping of pointers to a vector of pointers. It also stores a complete list of all the nodes present in the graph. The architecture graph nodes are in a module called `arch_node`, which has the following node types:

- `arch_add`: Sum operator, translated from `ast_add`;
- `arch_sub`: Subtraction operator, translated from `ast_sub`;
- `arch_sim`: Symmetric operator, translated from `ast_sim`;
- `arch_mult`: Multiplication operator, translated from `ast_mult`;
- `arch_div`: Division operator, translated from `ast_div`;
- `arch_mod`: Modulus operator, translated from `ast_mod`;
- `arch_shl`: Shift left operator, translated from `ast_shl`;
- `arch_shr`: Shift right operator, translated from `ast_shr`;
- `arch_rtl`: Rotate left operator, translated from `ast_rtl`;
- `arch_rtr`: Rotate right operator, translated from `ast_rtr`;
- `arch_and`: Bitwise operator and, translated from `ast_and`;
- `arch_or`: Bitwise operator or, translated from `ast_or`;
- `arch_xor`: Bitwise operator exclusive or, translated from `ast_xor`;
- `arch_not`: Bitwise operator not, translated from `ast_not`;
- `arch_comp`: Logic comparisons operators, translated from `ast_comp`;
- `arch_cast`: Cast operator, translated from `ast_cast`;
- `arch_num`: Numeral, translated from `ast_num`;
- `arch_id`: Stream identifier, translated from `ast_id` (when the instant is not applicable);
- `arch_ins`: Stream instant, translated from `ast_id` (when the instant is applicable). It is capable of storing a default value.

In the architecture graph, every node is identified with a node identifier (`node_id`). The stream identifiers are preserved from the `vscopes` when generating this new data structure. Every other kind of node gets a new identifier per instantiation. Every node also stores the identifier of the pulse which controls it (`pulse_id`). Finally, each node stores the datatype of that operation. Every node on this graph is of an arithmetic nature, so there are only two subtypes of node: `arch_unary_arith` (subset of unary operators) and `arch_bin_arith` (subset of binary operators). These have the same form as in the AST, and the nodes of this data structure are of the same type as in the AST.

Using an inverse adjacency list (or a normal adjacency list) has an inherent disadvantage: all connections between nodes lose meaning. This is due to the fact that edges between graphs that use adjacency lists are unnamed. This is a big problem to this compiler, because the meaning of right and left on an arithmetic operator would be lost. In order to solve this, the `arch_bin_arith` stores pointers to the right and left operators. These pointers shall be named shortcut pointers. In terms of memory storage, shortcut pointers are redundant, but they offer the advantage of having a defined meaning. They also provide a direct connection to adjacent nodes. Due to the cyclic nature of these pointers, similarly to what happened on the AST with the parent pointers (Section 4.3), they have to be weak pointers (`weak_ptr`).

The `arch_num` node has a different way of representing the numerals comparing to `ast_num`. While the `ast_num` stores the numeral using an integer or a floating-point, this node uses a dynamic bitset (`boost::dynamic_bitset`). The advantage of this representation is that it is a binary representation that can be directly used when generating hardware.

Similarly to the AST, the architecture graph also has two auxiliary structures: a node property associator (`arch_property`) and a graph traverser (`arch_topl_traverse`). The graph traverser implements a topological traverser over the architecture graph. In reality, when the stream instant node is instantiated, there is a possibility that a cyclic exists in the architecture graph. So, when the topological sort visits a stream instant node, it ignores if the parent has been visited or not.

If the architecture graph was only an inverse adjacency list, doing a topological sort would not be trivial (time complexity wise). This is because the algorithm needed to go through the list of unvisited nodes every time a node was visited to check if there are more nodes that can be visited. But since the architecture graph also has a normal adjacency list, for every node traversed, the algorithm can access the list of children of that node. With it, it can then check if each child has their parents already visited, using the inverted adjacency list. For each graph node, the traverser has a pure virtual function that has to be implemented by the extender class.

4.8.1 *rtfss*'s Architecture Graph Generator

The introduction of this new data structure requires more than a simple translation. Since every graph node has an embedded datatype, this module (named `arch_gen`) has to handle stream arithmetic operator sizing. Furthermore, this module has to convert every numeral to their binary representation. This module follows the stream sizing rules explained in Section 3.3.6. This is the last module to implement a `ast_visitor`.

In this module, all of the remaining unimplemented features presented in this Chapter are left behind. Here, only the `main` CBlock is used (if there is more, the current compiler implementation raises an error), both short and normal CBlock instantiations are ignored, `if` and `for` statements are also jumped over. Stream buses are ignored and both the relative stream delay (`gap`) and property of (`&`) operators are not handled. From this module forward, all the remaining features are kept and are successfully compiled.

Since this module handles the conversion of numbers to their binary representation, it has to handle dominant datatypes. A dominant datatype, as mentioned in Section 3.3, is the datatype that mandates on the size of the fractional part of a numeral in a stream. The dominant datatype is, by default, the datatype of the target of the attribution. The dominant datatype can be overridden using casts. To model this behaviour, `arch_gen` uses a `ast_property` that keeps track of the dominant datatype for each node. This module also keeps track of the `arch_node` that it generates for the AST nodes using a `ast_property` called `result`.

When a CBlock is entered, the module checks if the name of the CBlock is `main`. If it is not, the compiler raises an error and exits. This is only due to the limitations of the current implementation of the *rtfss* compiler.

When a numeral is visited, the compiler attempts to convert it to the binary representation. First, the compiler converts the integer part of the number. It tries to fit it to the least amount of bits. If this amount of bits is greater than the one set in the dominant datatype, the compiler throws a warning and truncates the integer part. A similar method is used to convert the fractional part. The conversion of the fractional part is done through the use of successive divisions. The module tries to convert the fractional part using fewer bits than that limit. If the limit is reached, the conversion stops, the compiler raises a warning (for imprecise representation) and keeps compiling. When exiting the visit function of this node, the compiler adds the generated node to `result`.

When visiting an identifier, the compiler checks if that identifier has a `arch_node` already. If the compiler finds it was already generated, it grabs a pointer to it from `result` and stores it temporarily. After that operation, the module checks if this node refers to a stream or to a stream instant. If it refers to a stream (or the instant is not valid), it places that pointer that it just stored in the `result` of this node and exits. In the case that the instant is valid, then some steps are necessary to solve this node. In order for the *rtfss* compiler to deliver past stream values (instants), it needs to create a chain of the values that the stream delivers. Every time the compiler has the need to get a stream instant, it checks if that stream instant node (`arch_ins`) has already been created. This check is done in an internal data structure called `inst_history`. This structure keeps track, for each stream identifier, the instant nodes already created. The `inst_history` is a vector of vectors of shared pointers to `arch_ins`. The outermost vector of this variable indexes the stream identifiers. The innermost vector's indexes represent the depth of the instant (being index 0 equivalent to the instant -1 and so on). If the `arch_ins` node that the module needs has already been generated, it is placed in the `result` of the current `ast_id` and exits. However, if that is not true, then the compiler has to generate the full chain until the instant value needed. For each node of that chain (`arch_ins`) that is generated, the compiler has to add an adjacency from that node to the previously generated node and needs to place this node in the `inst_history`. When the target instant is reached, the compiler places the node in the `result` of this node.

When a cast is visited, the compiler, instead of propagating the dominant datatype from the parent node, sets the current cast datatype as the new dominant datatype. In all the other type of nodes, the dominant datatype is propagated from the parent node. To aid this behaviour, this module has a `ast_property` called `dominant_dt` that keeps track of the dominant datatype of each node.

When the compiler visits an operator, the compiler creates a node for that operator, calculates the target datatype of that operation and sets it. Then, the compiler adds the left and right operand nodes and the newly created node to the graph and connects them accordingly. Internally, the compiler also sets the left and right shortcut pointers of the node, when it is a binary node. There are a couple of operators that require error checking before generating the node. These operators are the shift left, shift right, rotate left and rotate right. They require the right operand to be a constant unsigned integer stream, so the compiler checks here for those three conditions. If any of these conditions fail, the compiler raises an error and halts compilation.

When visiting an assignment or a declaration with an assignment, the compiler has to check if the attribution is for a normal stream or for a stream instant. If it is for a normal stream, it then checks whether the sizing of the attribution matches the size of the target stream or not. If they do not match, the compiler raises an error and halts compilation. Otherwise, the compiler adds both nodes to the graph and connects them. However, if the attribution is for an instant stream, a different logic is used. The current iteration of the compiler only supports compile-time solvable instants. This means that the instant requested on a stream must be a numeral or a const expression. The const expression is solved by the `const_handler` module, so when it reaches this module, it is a numeral too. Hence, the only two nodes allowed in the expression side of the attribution is a numerical or a cast. If this is not the case, the compiler raises an error and stops compiling. When dealing with a numerical, the compiler simply grabs the already converted number from `arch_num` and places it as the default value of the `arch_ins`. But, if the expression side of the attribution is a cast, then the compiler has to check if the target of the cast is a numerical or not. If it is not, it raises an error and exits. Else, the compiler fetches the value of the `ast_num` that generated the target `arch_num` and reconverts it to a binary value, using the new dominant datatype from the `arch_cast`. When the conversion is done, it sets that new value as the default value of the `arch_ins` and erases the nodes `arch_cast` and `arch_num` (and their connection) from the architecture graph.

4.8.2 Support Example Analysis

Now, we shall analyse how the support example (explained in Section 4.1.2) is converted to the Architecture Graph. In Figure 4.12 lies a graphical representation of the Architecture Graph of the support example. The graphical representation splits the architecture graph by its pulse domains. Each pulse domain has its own box. Since this example only has one pulse domain, there is only one box. This box is named *pid 2*, which is the identifier of the pulse of the domain (as shown in Section 4.5.1). The black arrows represent the adjacencies between the arch nodes. The red dashed arrows represent the shortcut pointers and their labels indicate which shortcut pointer is. The blue arrows represent adjacencies between a `arch_ins` and its source (can be a `arch_ins` or a `arch_id`). Each node has a name and two parameters in parenthesis. The first parameter is the identifier (number) of the node and the second is the datatype of that node. On this graph, it is easy to see the datatypes grow as the operations are made. The `arch_id` nodes are represented by the name of their stream. The `arch_num` nodes are represented by their numeric value (in hexadecimal). The other types of nodes are represented with their correspondent symbols. This diagram also demonstrates that every node of the Architecture Graph has an arithmetic meaning.

4.9 *rtfss*'s Final Representation Graph

Currently, the compiler does not have independent modules that manipulate the architecture graph, so the next compilation stage is to take another step down on the hardware abstraction by generating the *rtfss*' final representation graph.

The final representation graph is the third and last backbone data structure that the compiler generates. In this data structure, every node has a one-to-one relation with a

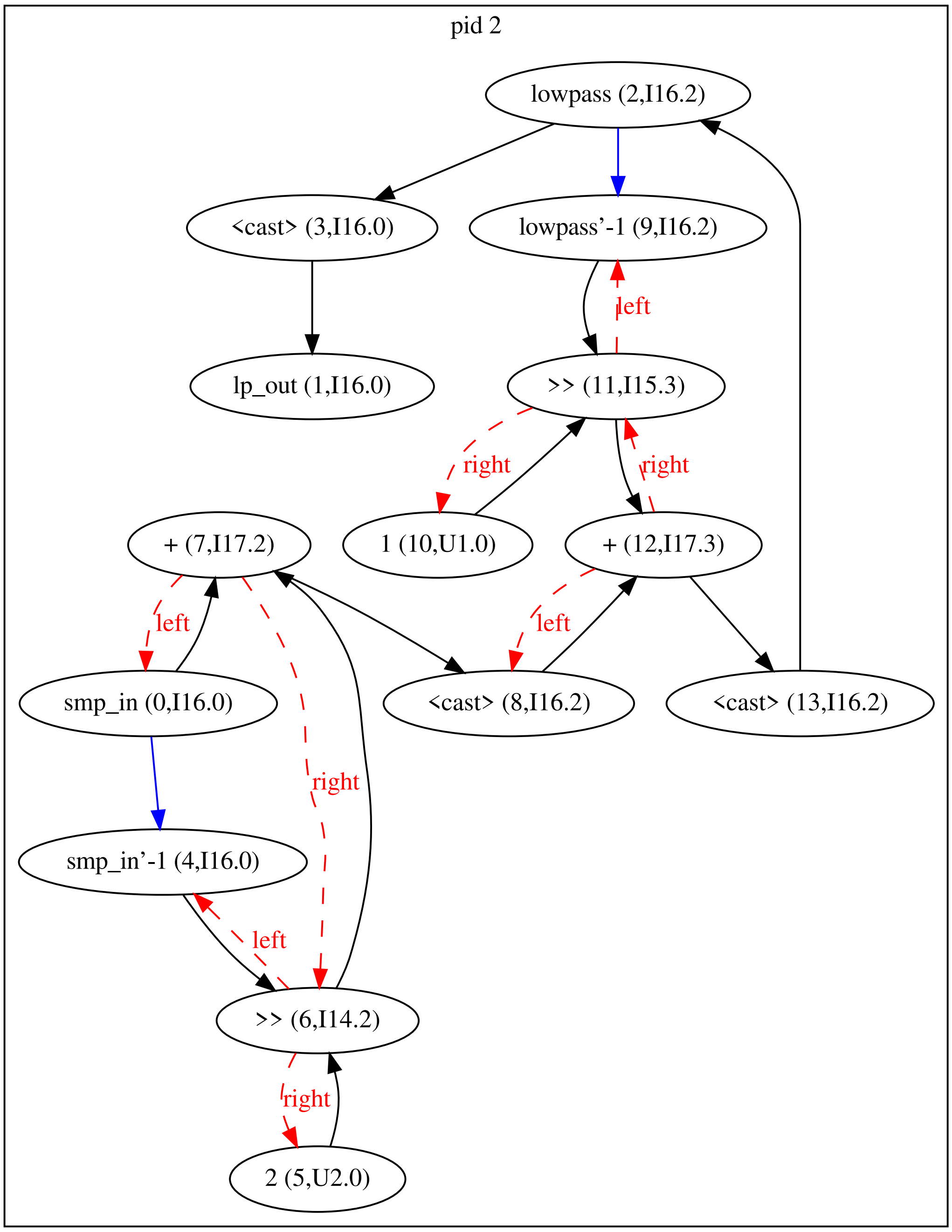


Figure 4.12: Full Architecture Graph of the example of Section 4.1.2 (GraphViz).

hardware block with almost no abstraction. All the information needed to generate target code/hardware is contained in this graph.

As the name implies, this data structure is a graph. However, the representation strategy used for this graph is different from the one used in the architecture graph. While the architecture graph uses an inverse adjacency list, with shortcut pointers and a normal adjacency list, the final representation graph bases its structure on shortcut pointers (with no inverse adjacency list). This structure is similar to the one used on AST in the way that each node stores the pointers to the adjacent nodes. Undoubtedly, this method has a critical shortcoming: if the graph is not fully connected, it has no obvious entry point. Since the final representation graph can be not fully connected, this shortcoming applies to this graph. One solution to this problem is to insert a node that connects to all the loose ends of the subgraphs. This is the solution used in the final representation graph. The compiler keeps track of all the final streams (main CBlock out streams), and at the end creates a node (that shall be named anchor node) that all those loose ends connect to. This solution is acceptable for this case, because the final representation graph is read-only and does not require nodes to be added and removed after its inception. Also, the use of this structure is preferable over the inverse / normal adjacency list, because the traversal needed in the final representation graph is rudimentary (breath kind). It also aids the generation of the target compiled code (this will be explained in Section 4.10). Not having an inverse adjacency list also means that there is no need to have an auxiliary object to store it (architecture graph has `arch_graph`).

The final representation graph's nodes are stored in a module called `frepr_node`. The nodes, similarly to the previous two backbone datatypes, have three subtypes: `frepr_arith`, `frepr_unary_arith`, `frepr_bin_arith`. Likewise, the last two extend the first one. The `frepr_arith` stores a `datatype` that is used to size the operators and thus size the hardware to be generated. The `frepr_unary_arith` stores the target as a pointer to a `frepr_node`. The `frepr_bin_arith` stores the right and left as pointers to `frepr_node`.

The final representation graph has the following node types:

- `frepr_anchor`: Serves as the graph anchor that connects to all subgraphs making this graph fully connected. Stores a vector of `frepr_node`;

- `frepr_clock`: Corresponds to the system master clock of the hardware target device. Stores the clock frequency (in an integer);

- `frepr_pulser`: Corresponds to a hardware block capable of generating cyclic pulses with fixed frequency. Stores a `time_scale` and a `double` corresponding to the pulser values;

- `frepr_ffd`: Extends the `frepr_arith` class. Corresponds to an array Flip-Flop type D. Stores name, pointer to the data input `frepr_node`, pointer to the master clock and pointer to the enable signal `frepr_node`. It can also store a default value and the name of the originating CBlock. The size of the array is defined by the inner datatype;

- `frepr_resize`: Extends the `frepr_unary_arith` class. Corresponds to the operation of adding or removing bits on the left of a hardware signal path. These bits are set to zeros if the originating stream is unsigned, and are the value of the most significant bit of the original word if the originating stream is signed (two's complement);

- `frepr_const`: Extends the `frepr_arith` class. Corresponds to the storage of a numeric constant in hardware, translated from `arch_num`. The node stores the numeral in a `boost::dynamic_bitset`;

- `frepr_add`: Extends the `frepr_bin_arith` class. Corresponds to an adder hardware block, translated from `arch_add`;

- `frepr_sub`: Extends the `frepr_bin_arith` class. Corresponds to a subtracter hardware block, translated from `arch_sub`;

- `frepr_sim`: Extends the `frepr_unary_arith` class. Corresponds to the two's complement symmetric number of a hardware signal path, translated from `arch_sim`;

- `frepr_mult`: Extends the `frepr_bin_arith` class. Corresponds to a multiplication hardware block, translated from `arch_mult`;

- `frepr_div`: Extends the `frepr_bin_arith` class. Corresponds to a division hardware block, translated from `arch_div`;

- `frepr_mod`: Extends the `frepr_bin_arith` class. Corresponds to a modulus hardware block, translated from `arch_mod`;

- `frepr_shl`: Extends the `frepr_bin_arith` class. Corresponds to making a left shift on a hardware signal path, translated from `arch_shl`. In other words, add `N` zeros on the right of the least significant bit and remove `N` bits from the most significant part of the word;

- `frepr_shr`: Extends the `frepr_bin_arith` class. Corresponds to making a right shift on a hardware signal path, translated from `arch_shr`. In other words, if the originating stream is an unsigned, add `N` zeros on the left of the most significant bit and remove `N` bits from the least significant part of the word. If the originating stream is signed, replicate the most significant bit `N` times on the left of the most significant bit and remove `N` bits from the least significant part of the word;

- `frepr_rtl`: Extends the `frepr_bin_arith` class. Corresponds to making a left rotate on a hardware signal path. In other words, moving the `N` most significant bits to the least significant part of the word, in a circular fashion;

- `frepr_rtr`: Extends the `frepr_bin_arith` class. Corresponds to making a right rotate on a hardware signal path. In other words, moving the `N` least significant bits to the most significant part of the word, in a circular fashion;

- `frepr_and`: Extends the `frepr_bin_arith` class. Corresponds to the (array of) logic gate and, translated from `arch_and`;

- `frepr_or`: Extends the `frepr_bin_arith` class. Corresponds to the (array of) logic gate or, translated from `arch_or`;

- `frepr_xor`: Extends the `frepr_bin_arith` class. Corresponds to the (array of) logic gate xor, translated from `arch_xor`;

- `frepr_not`: Extends the `frepr_unary_arith` class. Corresponds to the (array of) logic gate not, translated from `arch_not`;
- `frepr_comp`: Extends the `frepr_bin_arith` class. Corresponds to a boolean logic comparisons hardware block, translated from `arch_comp`.

The final representation graph has only one support module: an inverse breadth-first traversal. This module was designed to serve as the compiler's interface to the target code generator modules. It starts the traverse through the anchor node and starts visiting their children. Using a normal breadth-first traversal would not be beneficial, since the output code is always for a parallelized language, and thus one can argue that the inverse traverse can be more natural for these kinds of language. For every node type, the module has a custom function that has the graph node as an argument. The interface of this module is similar to the one presented on `ast_visitor` or `arch_topl_traverse`.

4.9.1 *rtfss*'s Final Representation Graph Generator

The Final Representation Graph Generator (`frepr_gen`) is the module that converts the architecture graph to the final representation graph. This module implements the class `arch_topl_traverse`. Although the architecture graph is already somewhat middle / low level, this translation represents a big step down towards the hardware level. This means that this module introduces many new nodes and has some unique manipulations and strategies.

When the module starts the translation, it runs a preamble before starting the traversal. On instantiation, the module creates the graph anchor (`frepr_anchor`) and stores it on a variable called `fanchor`. After that, the module creates the master clock node (`frepr_clock`) and stores a shared pointer to it on a variable called `mclock`. Then, right before the traversal begins, the module solves all the pulses present on the architecture. In order to do this, the module iterates over all the pulses present on the global pulse scope `pcscope`. For each non-reserved pulse (`const` and `max`), the module calculates the correct frequency of that pulse and generates a `frepr_pulser`. This node is then stored on a structure called `pulse_nodes` (vector indexed by the pulse identifier). For every reserved pulse, since the compiler at the current stage of implementation does not have the context of the target hardware capabilities, the module simply has to create a `frepr_pulser` node with a frequency of zero Hertz. These nodes are also stored in the `pulse_nodes` structure. Finally, after completing this operation, the traverse can start.

One of the main focuses of *rtfss* is to generate sequential hardware in a timing focused manner. In particular, *rtfss* is designed to generate a pipelined architecture that successfully implements the algorithms required. The module responsible for the hardware generation is `frepr_gen`. This is an arguably complex task and, although every architecture graph node has specific rules and has to be handled specifically, there is a bulk arrangement of tasks that are common among nodes. So, in order to ease the implementation complexity, some support functions were created.

The main rationale behind this module is that most arithmetic operations should be contained in its own time bubble. This ensures that the hardware solution created is purely sequential. So, the way the compiler solves an expression with multiple operations is to solve step by step each operation. The throughput of these operations is set by the master clock. However, the frequency at which these operations are started is set by the pulse domain of their expression. Hardware-wise, essentially this means that after the combinational arithmetic operation, an array of flip-flops type D is placed. Each flip-flop connects to a specific bit of the stream to serve as a buffer. The clock signal of these flip-flops is connected to the hardware master clock (stored in the variable `mclock`). Each flip-flop type D also has a enable signal, and it is with it that the compiler controls the pulse. But, the compiler cannot connect the pulser blocks directly with these enables. If they did, the operations would have the throughput of the pulser and the hardware would lose efficiency. So, what the compiler does, in fact, is to create a chain of flip-flops that connect to each pulser. These flip-flops propagate the pulse signal, introducing a clock cycle of delay on each stage. The depth of each pulse chain is defined by the depth of the stream with most operations. This chain is generated on-demand, while the module runs. One of the main function that manages the pulse chain is `get_pulse`.

The function `get_pulse`'s main goal is to provide/calculate the pulse chain step of a certain node. In order to operate, this function needs the pulse identifier of the new node, the pulse identifier of the previous node and the depth of the last node. These last two properties are contained inside a tuple (`std::pair`). This tuple is used to represent the pulse status of a certain pulse. The tuple of each final representation graph node is stored on a `arch_property` called `curr_pulse`. This function only works if the current node only has one input connection (for example, unary operators). The function firstly checks if the current pulse id is equal to the previous pulse id. If it is, then the pulse chain can be continued. So, if this is the case, the function searches if the wanted pulse chain node has already been generated. This search is done on yet another data-structure called `pulse_chain`. The `pulse_chain` is a vector of vectors of shared pointers to `frepr_ffd`, where the outermost vector indexes the pulse identifier and the innermost vector indexes the depth of the pulse chain. If the pulse chain flip-flop needed has already been generated, the function increments the depth field of the tuple and returns a shared pointer to the flip-flop. Otherwise, it generates the new pulse flip-flop node, places it on the `pulse_chain`, increments the depth filed of the tuple and returns a shared pointer to the flip-flop. In the case that the current pulse is different from the previous pulse, the compiler has to restart the pulse chain. So, the compiler changes the pulse id of the tuple to the new pulse id and the depth to zero. It then checks if the wanted pulse chain flip-flop D has already been generated. If it has, then it returns a shared pointer to it. Else, it creates a new one, inserts it in the `pulse_chain` and returns a shared pointer to it.

The last function's objective is to handle the pulse chain flip-flop of a node based on the tuple of the previous node. However, this function does not operate when there is no previous node. This happens on terminal nodes such as stream identifiers or numeric constants. For these cases, a function called `start_chain` is used. This function has a similar behaviour to `get_pulse` when the current node's pulse is different from the past pulse. It then checks if the initial pulse chain flip-flop D has already been generated. If it has, then it returns a shared pointer to it. Else, it creates it, inserts it in the `pulse_chain` and returns a shared pointer to it.

The `get_pulse` function is really useful in nodes that only have one input edge. But, most of the arithmetic operations on the *rtfss*'s grammar are binary. To ease this part of the module, an auxiliary function was also created. This function, called `resolve_bin`, is the main foundation of this module. The input of the function is the shared pointer to both `arch_node` operands (called `right` and `left`), the pulse identifier of the current node, a shared pointer to a flip-flop (used to return the pulse chain node to use, called `p_to_use`) and a pointer to a tuple of a shared pointer to a flip-flop D (pulse chain property of the node, called `np_property`). In reality, the variables used in this block are slightly different, but for the sake of simplicity and clarity, they shall be named this way on this document. While handling only one input node is easy, since we only need to check if the pulse domain is the same or not, having two input nodes doubles the possibilities. Furthermore, in some cases, having two input nodes might require delaying one of them to keep the synchronization between nodes. Having that in mind, the function returns a pair of the `frepr_ffd` that should be interpreted as the new `right` and `left`.

The behaviour of `resolve_bin` can be split into four blocks:

- `left` and `right` share the same pulse domain as the current node: When all the nodes share the same pulse domain, then the new node has to be a continuation of the pulse chain of that domain. But, first, the chains must be aligned in both inputs. So, the compiler finds which of the inputs has the shortest depth in the chain and then it creates a chain of delays (pipeline stages that only contain the buffers). These buffers are internally called synchronization bubbles. If the inputs are already synchronized, no delay is introduced. After that, the function `get_pulse` is called with either input nodes and the current pulse identifier. It then places the new pulse chain property in `np_property` and the pulse chain flip-flop D in `p_to_use`. This function will return the new pulse chain property of the current node. Finally, the function packs the new `left` and `right` into a tuple (at most one of them has changed, because of the synchronization bubble), and returns it;

- `left` shares the same pulse domain as the current node, but `right` does not: In this case, the compiler will simply follow the pulse chain of the `left`. To do so, it calls `get_pulse` with the `left` node and the target pulse identifier. Then, it places the new pulse chain property in `np_property` and the pulse chain flip-flop D in `p_to_use`. Finally, it returns `right` and `left` as is;

- `right` shares the same pulse domain as the current node, but `left` does not: In this case, the compiler will simply follow the pulse chain of the `right`. To do so, it calls `get_pulse` with the `right` node and the target pulse identifier. Then, it places the new pulse chain property in `np_property` and the pulse chain flip-flop D in `p_to_use`. Finally, it returns `right` and `left` as is;

- neither `left` nor `right` share the same pulse domain as the current node: If neither of the input nodes are in the pulse domain of the current node, then the compiler has to start a new chain. To do so, it calls `start_chain` and places the new pulse chain flip-flop in `p_to_use`. It then generates a new pulse chain property with depth zero, and places it in `np_property`. Finally, it returns `right` and `left` as is.

In the current iteration of the *rtfss* compiler, the target language is required to handle basic arithmetic integer operations. However, since *rtfss* allows fixed-point arithmetic on hardware, the compiler has to adapt its operations to work with integer arithmetic. In most cases, this adaptation only requires some sizing tricks, but, in other cases, they might require more manipulation. Although *rtfss* allows the mixture of streams with different sizes in one expression, on the hardware level the compiler ensures that all operations have a persistent amount of bits in all operands and output. This means that the compiler has to do some bitwise manipulation before feeding streams into operations. Two support functions were created to aid the implementation of such behaviour: `inclusive_datatype` and `align_stream`.

The `inclusive_datatype` is a simple function that receives two datatypes and a stream type (unsigned or signed). The function then calculates and returns a datatype that could represent both datatypes without losing data. This function is useful for creating a temporary datatype to do operations where the output size of the stream cannot represent the operands.

The final support function of this module is called `align_stream`. As the name implies, this function aligns the datatype of a stream to another datatype. The inputs of this function are the node to be aligned (`frepr_arith`) and the target datatype. The return of this function is a new node that represents the same value of the input node, but on the target datatype. In order to adjust the datastream, this function has to introduce a couple of nodes (shifts and resizes). While aligning, this function has to handle three different cases. Either the target datatype is bigger than the current datatype, or the target datatype is smaller than the current one, or they are both equal. If both datatypes are equal, the function simply returns the input node. In reality, the alignment operation's objective is to adjust the offset of the virtual comma (that separates the integer part from the fractional part). This operation can be achieved with shifts. The other part of this operation is to cut the stream to the right size. So, in the other two cases:

- Target datatype bigger than current datatype: In this case, the compiler has to resize the current stream and then shift the stream into place. The first thing that the compiler does is to resize the current stream to the target datatype. Then, the function needs to figure out if the stream needs to be shifted right or left, and the amount of bits to shift. If the target stream's fractional part is bigger than the input node's fractional part, then the compiler needs to shift left. This is due to the fact that the virtual comma is further to the left than the current one. Otherwise, the compiler needs to shift right. The amount of bits to shift is the absolute value of the difference between the sizes of the fractional parts of the datatypes;

- Target datatype smaller than current datatype: Opposed to the last case, in order to not lose the relevant part of the stream, it needs to be shifted before resized. Firstly, the compiler needs to figure out to which direction the shift has to be made, and in what quantity. Similar to the last case, the compiler shifts left when the target's datatype fractional size is bigger than the input node's datatype fractional size and shifts right otherwise. The amount of bits to shift is the absolute value of the difference between the sizes of the fractional parts of the datatypes. Finally, after shifting the stream, the compiler resizes (shrinks) the target to the desired datatype.

For each resize and shift, the compiler generates the final representation graph nodes `frepr_resize` and `frepr_shr` / `frepr_shl`. These are wired correctly and the node returned by the function will most probably be one of these. A simple use-case of this function are casts. In *rtfss*, a cast can be resolved by just calling the `align_stream` with the target node and desired datatype. Since `align_stream` only manipulates hardware signal wiring, it does not introduce delays. So, no flip-flop buffer and pulse chain stage are introduced by the function.

As mentioned before, the general process that each `arch_node` goes over to generate a `frepr_node` is relatively constant. Now that the auxiliary functions are stated, we can go more into detail on how the method looks like. For a binary arithmetic node that has the datatype expanded (in relation to the inputs), the process follows this logic:

1. Get the pointers to the `left` and `right` nodes and pulse identifier;
2. Call the `resolve_bin` function, in order to synchronize `left` and `right`, and to solve the pulse chain node of this node;
3. Align both of the input nodes that are returned by `resolve_bin` to the target output datatype using `align_stream`;
4. Create the final representation node of the current operation, feed it with the `left` and `right`;
5. Create the flip-flop array buffer node, feed it with the output of the node created on the last step, set the clock to the master clock node and the enable to the pulse chain node;
6. Place the current pulse chain property (acquired from `resolve_bin`) on the `curr_pulse`;
7. Place the flip-flop array buffer node in the `gen_frepr`.

When the node is unary, then the function `resolve_bin` will not be called. Instead, the function `get_pulse` will provide the pulse chain node property needed. When the node operation has no delay (`arch_cast`, `arch_shl`, `arch_shr`, `arch_rtl`, `arch_rtr`), the step where the compiler creates a flip-flop array is skipped, and the node created for the arithmetic operation is placed on `gen_frepr`. There are several nodes (`arch_shl`, `arch_shr`, `arch_comp`, `arch_mod`) where the output datatype cannot, or possibly cannot, fully represent the datatype of the inputs. In these nodes, instead of aligning both `left` and `right` to the datatype of the operation, they align the inputs to a bigger datatype that can represent all the nodes. Then, the operation is done normally, but, at the end, the output node has to be aligned back to the output target size. So, to do this, another call to `align_stream` is made.

With this logic, the compiler is able to traverse all the tree and generate valid final representation graph nodes. However, the module is not done yet. After finishing the traversal, the module still needs to perform some operations.

The *rtfss* language allows the programmer to set default values for stream instants. Similarly, at the current stage of implementation, the *rtfss* compiler also allows the definition of them. But, due to the way the compiler generates the final representation graph, the default values of a stream cannot be set to the nodes the architecture graph states. Instead, they should be set on the parent. This happens because the first stages of the pipelines (when the instants are used) are the stream instants, and they are connected to the back of the streams. So, in order to make default values on stream instants work, after the traversal of the `arch_graph`, the compiler has to shift back the default values. To do so, it iterates a map that was generated on traversal. This map, called `ins_map`, has shared pointers to `frepr_ffd` (that represent a stream or a stream instant) and the values are original architecture graph nodes (`arch_node`). With it, it can access the parents and sequentially pass the default values back.

After fixing the steam instant default values, the module has to anchor down all the loose sub-graphs that were generated while traversing. To do so, it iterates over all the nodes from the node list given by `arch_graph`. For each node, it checks if the node is a stream. If it is, it then checks if the stream is an output stream. Being an output stream, it fetches the final representation graph node from `gen_frepr` and anchors it to the `fanchor`.

The last step done on the final representation graph generator is to do a little fix in the pulse chains. Although, while traversing the architectural graph, the pulse chain nodes are created, in reality they are not linked together. Instead, and in order to ease the implementation of this module, this operation is made now (after the traverse is done, at the end of the module). As mentioned before, the module places each pulse chain node it creates on `pulse_chain`. Also, before the traversal, the module instantiates pulser nodes which are placed in `pulse_nodes`. So, to link the chain together, the module has to iterate over the pulser nodes, and for each one of them, iterate the vector of their flip-flop D and connect the inputs to the outputs of the previous flip-flops.

At last, after all these steps, the final representation graph is completely built. Comparing to the architecture graph, this graph has significantly more complexity and number of nodes. But now, each one of these nodes has a one-to-one relation to a hardware block/operation. The current state of compilation represents the lowest abstraction the compiler reaches. The input code is now ingrained in the connections and operations made on the pipeline, and are no longer as easy to observe as it was on previous stages of compilation. The compiler can now move on to the next (and final) module of compilation: the code generation module.

4.9.2 Support Example Analysis

We can now analyse how the support example (explained in Section 4.1.2) is converted to the Final Representation Graph. In Figure 4.13 lies a graphical representation of the Final Representation Graph of the support example. Each of the nodes represented in that Figure has a correspondent Final Representation node. Each node starts off with the node identifier in parenthesis, followed by the type of the node, has one optional extra value and ends with the datatype of that node. The *pulser* node is a `frepr_pulser`, the *ffd* node is a `frepr_ffd`, the *const* node is a `frepr_const` and the *resize* node is a `frepr_resize`. The other nodes have the meaning of the symbols that represent them. The *const* node has an extra argument that is the numeric value of the node represented in hexadecimal. The *ffd* also has an extra argument that indicates its purpose:

- *pulse_chain*: The current node is used to generate a step in a pulse chain;
- *sync_bubble*: The current node is used to delay and synchronize a signal path (audio stream);
- *ins*: The current node is a flip-flop used to propagate previous sample values (value at instant operator);
- *buffer*: The current node corresponds to the buffer of an operation. In other words, it acts as the separator between different operations (pipeline stages);
- Any other value is a name of a stream. So, this node is a flip-flop array that holds the value of a stream.

The black arrows represent the data flow of audio between the nodes. Blue arrows represent that the source node of the arrow controls the enable signal of the destination node of the arrow. In this visual representation, it is easy to see the patterns created by the compiler to align the audio streams. For example, on Figure 4.14, the node 27 gets resized from `I16.3` to `I15.3` by passing through a shifter and a resize node.

4.10 Output Language Generator

In this Section, we reach the end of the current implementation of the *rtfss* compiler. Having the final representation graph assembled, all that remains is the creation of the target output language code. There are a vast quantity of hardware description languages. The two most popular ones are Verilog and VHDL [1]. For the *rtfss* compiler, we choose VHDL as the target language (explained in Section 2.2). This choice was made due to the fact that we have a greater familiarity with it. If, in the future, there is a need to change the output language, the only module that needs to be changed is this one.

The code generation module, `target_gen`, has two classes: `vhdl_templates` and `vhdl_gen`. The `vhdl_templates` class contains an assortment of VHDL code templates that can be combined to generate hardware logic. The `vhdl_gen` is the module that traverses the final representation graph and that instantiates the templates present in `vhdl_templates`.

The `vhdl_templates` class uses string templates. String templating allows the embedment of data in a string. The behaviour of string templating is similar to the one offered by the `sprintf` method of C's standard library. The downfall of using `sprintf` is the need of using C-like strings and buffers, which are `char*`. These buffers need to have a static size, so its use in this application (although valid) is impractical. The *C++20* standard offers a similar alternative to the *Boost Format* called `std::format` (present in the header `<format>`). Unfortunately, when the *rtfss* compiler was implemented, this feature was not yet present in any C++ compiler. So, naturally, *Boost Format* was the solution used. In order to construct the output code, the templates are used to fill several string streams (`std::stringstream`). Templates are managed by functions. Each function is responsible for managing and instantiating the template. The `vhdl_templates` module has the following functions / templates:

- `vhdl_pulser`: The code present in this template declares a pulser block. This template is based on the VHDL hardware block made for the initial architecture (explained on 5.4.3). In terms of VHDL, this template contains the `entity` and `architecture` of the pulser hardware block. The only input of the function is the string stream where the template should be instantiated to;

- `vhdl_ffd`: The code present in this template declares a flip-flop D block. In terms of VHDL, this template contains the `entity` and `architecture` of the pulser hardware block. The only input of the function is the string stream where the template should be instantiated to;

- `vhdl_entity`: This template represents an empty VHDL `entity` and `architecture`. It requires as input the name of the entity, a list of generic declarations (as vector of strings), a list of port declarations (as vector of strings), the architecture name, all the `signal` declarations (already packed in a string) and all the `signal` assignments (already packed in a string). Finally, it also receives the string stream where the template should be instantiated to;

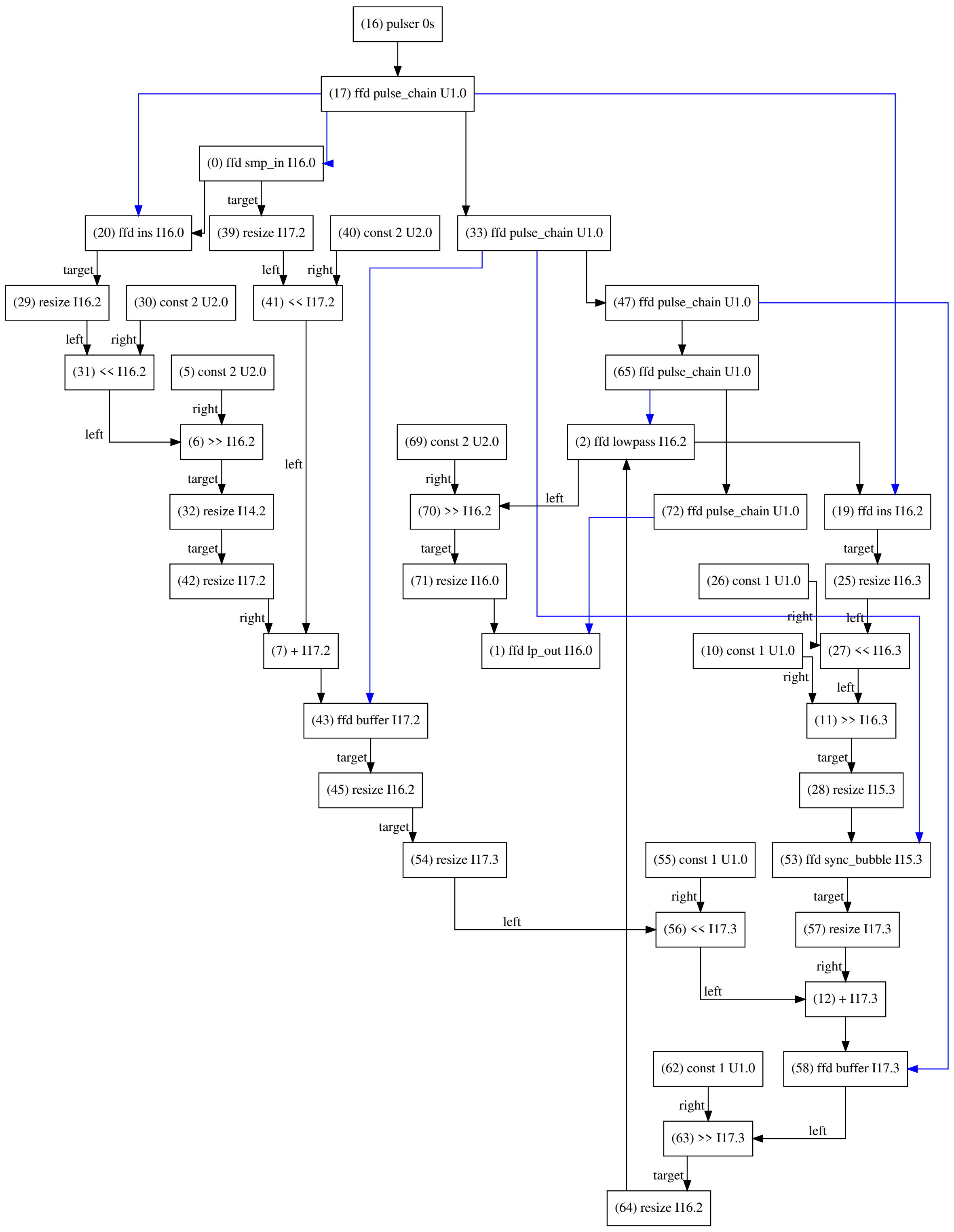


Figure 4.13: Full Final Representation Graph of the example of Section 4.1.2 (GraphViz).

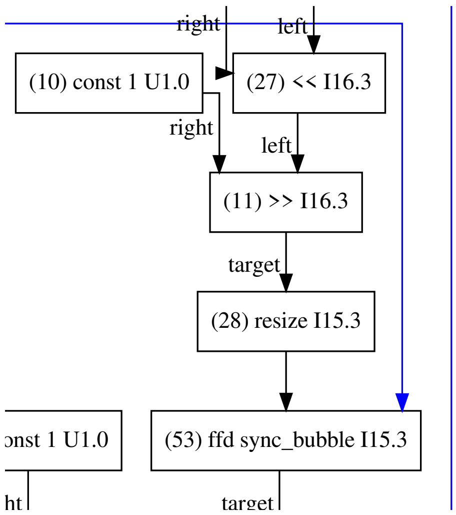


Figure 4.14: Detail of a stream alignment done on the Final Representation Graph of the example of Section 4.1.2 (GraphViz).

- `vhdl_entity_inst`: This template represents an instantiation of a VHDL block. It requires as input the name of the instantiation, the name of the entity to be instantiated, a list of generic maps (as vector of strings) and a list of port maps (as vector of strings). It also receives the string stream where the template should be instantiated to;
- `vhdl_entity_generic`: This is an auxiliary template used to aid the creation of a `vhdl_entity`. It represents the syntax used to declare that an `entity` has a `generic`. It receives the name of the generic and which type it is. The template instantiated is returned as a string;
- `vhdl_entity_port`: This is an auxiliary template that is used to aid the creation of a `vhdl_entity`. It represents the syntax used to declare that an `entity` has a `generic`. It receives the name of the port, the direction of the data flow (`in` or `out`), and the dimension (in bits) of the port. The template instantiated is returned as a string;
- `vhdl_mapping`: This is an auxiliary template used to create mappings. This is used to aid the creation of the `vhdl_entity_inst`. It requires as input an entity port name and a signal name. The template instantiated is returned as a string;
- `vhdl_signal`: This template represents the declaration of a `signal`. It requires a `signal` name and `signal` dimension. It also receives the string stream where the template should be instantiated to;
- `vhdl_assign`: This template represents an assignment from one `signal` to another `signal`. It receives the name of both `signal` and the string stream to write the template to;
- `vhdl_resize`: This template represents a call to the VHDL function `resize`. The template requires the name of the destination `signal`, name of the source `signal`, type of the source `signal`, and the new dimension the `signal` should take. It also requires a string stream to write the template to;

- `vhdl_add`, `vhdl_sub`, `vhdl_mult`, `vhdl_div`, `vhdl_mod`: These templates instantiate an addition, subtraction, multiplication, division and reminder of the integer division, respectively. They all require the name of the target `signal`, the name and type of the right and left operators. The multiplication template also requires the dimension of the target signal. This is needed, because, in VHDL, the multiplications are expanded. Like most of the other templates, these also require the string stream to write to;
- `vhdl_sim`: This template instantiates a symmetric operator. It needs the name of the target `signal`, the name of the source `signal` and the type of the source `signal`. Finally, it also requires the string stream to instantiate the template to;
- `vhdl_shl`, `vhdl_shr`, `vhdl_rtl`, `vhdl_rtr`: These templates instantiate a shift left, shift right, rotate left, rotate right, respectively. They all require the name of the target `signal`, the name and type of the left `signal`, the `constant` that specifies the shift amount. Like most of the other templates, these also require the string stream to write to;
- `vhdl_and`, `vhdl_or`, `vhdl_xor`: These templates instantiate an "and" gate array, "or" gate array and "xor" gate array, respectively. They all require the name of the target `signal`, the name of the right and left `signal`. Like most of the other templates, these also require the string stream to write to;
- `vhdl_not`: This template instantiates a "not" gate array. It needs the name of the target `signal`, the name of the source `signal` and the string stream to write to;
- `vhdl_equal`, `vhdl_diff`, `vhdl_more`, `vhdl_less`, `vhdl_moreq`, `vhdl_leseq`: These templates instantiate binary comparator blocks "equal to", "different than", "more than", "less than", "more or equal to" and "less or equal to", respectively. They require the target `signal`, and the name and type of the left and right `signal`. Like the others of this kind, it also receives the string stream to write to;
- `vhdl_const`: This template represents a declaration of a VHDL `constant`. In order to be instantiated, it requires the name of the `constant`, the dimension of the `constant` and its value. The value is represented with a `boost::dynamic_bitset`. Finally, it also requires the string stream to write to.

All `signal` are stored in `std_logic_vector`, even the ones that only have one bit. This is done to ease the synthesis of the output language. This way, there is no need to keep track of the type of the `signal`. Every time an arithmetic operation is made, the template casts all the `signal` operators to the wanted type (`unsigned` or `signed`), then does the operation and casts the result back to a `std_logic_vector`.

There are several templates that are big (for example `vhdl_pulser` and `vhdl_entity`). For these, raw strings were used. Raw strings are strings that allow a raw stream of characters, and do not process control sequences (such as newline '\n'). As such, they can be multi line. They were introduced in `C++11` and the delimiters are `R"(` (to start) and `")` (to end).

The `vhdl_gen` module is the top-level class of the "super" module `target_gen`. It implements the `frepr_rev_breath_traverse` interface, and its job is to convert the final representation graph into VHDL code. It does such task using the templates previously explained of `vhdl_templates`. These templates are instantiated while the module traverses the graph.

Although the logic of this module is quite straight-forward, there is still one relevant auxiliary function: spawn_ffd. The spawn_ffd function is responsible for instantiating a flip-flop type D array. It receives the name of the stream that is going to have the flip-flop array, its dimension, the name of the signal that has the enable, the name of the signal that is the input and the reset / default value. The function then uses the vhdl_entity_inst to instantiate the flip-flop D block.

The vhdl_gen module contains two vectors that store the ports of the top-level VHDL block (vs_ports) and store the generics of the top-level VHDL block (vs_generic). Both vectors store strings. Also, the module manages three different string streams:

- ss_out: This is the main string stream where the output code will (eventually) reside;
- ss_var_dcl: This string stream represents the section of the VHDL architecture block where signal are declared;
- ss_statm: This string stream represents the section of the VHDL architecture block where the statements are present;

Before starting the final representation graph traversal, the module needs to setup. Firstly, it starts by instantiating the vhdl_pulser template to the ss_out stream. The same is done with the vhdl_ffd template. Now that both blocks are instantiated on the output code, the compiler adds a generic called CLK_FREQ (of type real) to the generics of the top-level block. This is done using the help of the vhdl_entity_generic template, and is stored on the vs_generics. It also adds then two ports to the VHDL top-level block called clk (Master Clock) and rst (Master Reset). These two ports are created with the help of the vhdl_entity_port template and are stored on the vs_ports vector. Finally, the module can traverse the final representation graph.

The translation logic used to convert the final representation graph arithmetic nodes to VHDL code is quite constant. Essentially, when the vhdl_gen module traverses an arithmetic node, it first solves the name of the current node. In this module, the name of the signal generated by final representation graph nodes is the concatenation of n with the numeric identifier of the node. So, for example, a node with the identifier four will be called n4. After getting the name of the current node, a signal is created for it. For that, the compiler instantiates the vhdl_signal template from vhdl_templates and places it in ss_var_dcl. Then, the module fetches the node names of the operands (with the same tactic). The datatypes from the operands, when needed, are also fetched and converted to VHDL signal types signed or unsigned. Finally, the vhdl_templates of the correct operator is instantiated and placed under ss_statm.

When traversing a frepr_resize, the module gets the name and the type of the node, and simply instantiates a vhdl_resize from vhdl_templates and places it into ss_var_dcl. When traversing a frepr_const, the module gets the name and the value of the constant, and simply instantiates a vhdl_const from vhdl_templates and places it into ss_var_dcl. When the module traverses a pulser node, a VHDL pulser entity needs to be instantiated. So, it first needs to convert the timescale contained in the node to frequency (in Hertz). With it, it creates a generic mapping for the pulser frequency. It then adds the clock node to the port mappings. The output of this pulser entity gets connected to this node's name. This is also done using a port mapping. Finally, it creates a signal to the output pulse (using vhdl_signal) and instantiates the vhdl_entity_inst for the pulser entity block using all the mappings generated. When traversing a `frepr_ffd`, some extra precautions have to be taken. The compiler has to check first what the kind of flip-flop D it is. This is done by observing the `var_property` field:

- `IN`: This means that the current node represents an input of the `main` block. Then, the compiler has to add a port to `vs_ports`, through the use of `vhdl_entry_port`. The name of the port will be `p` plus the numeric identifier of the node. All inputs must be synchronized, so now the compiler creates a `signal` (with the naming convention already explained), and binds a flip-flop D to that `signal`;
- `OUT`: This means that the current node also acts as an output to the `main` block. So, the compiler creates a port mapping on the top-level entity. The name of the `signal` used on the mapping is `p` plus the numeric identifier of the node. The name of the flip-flop input and enable is fetched, and a flip-flop (with the name `n` plus the numeric identifier of the node is created), and it is placed under `ss_statm`. Finally, the name of the flip-flop is bound to the output port using a `vhdl_assign` (also placed on `ss_statm`);
- `REGULAR`: In this case, the current node is just a regular flip-flop D. These can be used for streams or for pulse chains. The distinction is made using the enable signal. If the node does not have an enable signal (pointer equal to `nullptr`), then the node is used on a pulse chain. Otherwise, it is used on a stream. In this case, the only thing the module does is to create the `signal` to store the flip-flop D and to instantiate it.

With all this, the module can traverse the whole graph and generate these fragments of VHDL output code. When the traversal is done, the only thing left to do by the module is to stitch the string streams together to form the complete output code. To do this, the only thing it has to do is to instantiate the `vhdl_entity` template present in `vhdl_templates`. The top-level entity it generates is called `rtfss` and the implementation is called `impl`. The entity generics `vs_generics` and the entity ports `vs_ports` are also fed to the template. Finally, it also feeds the string streams `ss_var_dcl` (`signal` instantiations) and `ss_statm` (statement declarations) onto the template. The output of the template is written to the `ss_out` string stream. With this, the module `vhdl_gen` ends processing and the whole compilation procedure is over. The only thing left to do, is for the compiler to output the generated code. In the current implementation of the compiler, in the `rtfss_main` top-level module, the compiler simply writes the `ss_out` string stream to the `std::cout` string stream. This makes the compiler write all the string stream to the standard output of the program.

4.10.1 Support Example Analysis

The output VHDL code generated by the compiler for the support example (explained in Section 4.1.2) is quite big, so it was placed inside the Appendix B. For sake of clarity, the code was indented manually. This was the only modification made to the code. The top-level entity of the code is the entity `rtfss`. This entity contains the input and output ports of the original *rtfss* design. The `p_lp_out` VHDL output port corresponds to the `lp_out` *rtfss* output port, and the `p_smp_in` VHDL input port corresponds to the `smp_in` *rtfss* input port.

To successfully synthesize the generated VHDL code, one change has to be made. Since the *rtfss* compiler is not aware of the sampling frequency of the system, when it generates the output code, the pulser frequency parameter is left with an inadequate value. In this case, the value written is `inf`. This value can be found on the entity instantiation `inst8` and it has to be set to the desired sample frequency. Furthermore, the frequency of the system's master clock also has to be set in the instantiation of the `rtfss` entity (generic parameter called `CLOCK_FREQ`).

The Altera Quartus Prime RTL (Register-Transfer Level) Diagram generated by compiling the output VHDL code of the support example is in Figure 4.15.

4.11 Project Code Metrics

In this Section, some metrics of the source code of the *rtfss* compiler are listed. As it stands, the compiler:

- Contains 14450 lines of code in 55 source files, broken up into:
  - 11227 lines in 28 files of C++ code (`.cpp`);
  - 2959 lines in 24 files of C++ headers (`.h`);
  - 72 lines in 2 files of C++ templates (`.tcc`);
  - 192 lines in 1 file of ANTLR4 grammar (`.g4`);
- Contains 115 C++ classes and 2 structs;
- Takes approximately 6 minutes and 30 seconds to compile from scratch using one job on an Intel i7-2620M CPU (at 2.70GHz). This time includes the compilation of the ANTLR4 runtime.

4.12 Summary

Given the proposed formal specification for the *rtfss* language present in Chapter 3, in this Chapter a partial *rtfss* compiler was implemented. This compiler was implemented in C++ with the use of ANTLR for language parsing and Boost for auxiliary data-structures. Internally the compiler is segmented into various logic blocks. These blocks operate over the internal data-structures. There are three internal data-structures: an abstract syntax tree, and two graphs. The compiler shifts from one data-structure to another to progress into a more hardware-oriented representation. At the end of this process, the compiler generates VHDL code that can be used to synthesize the specified design.

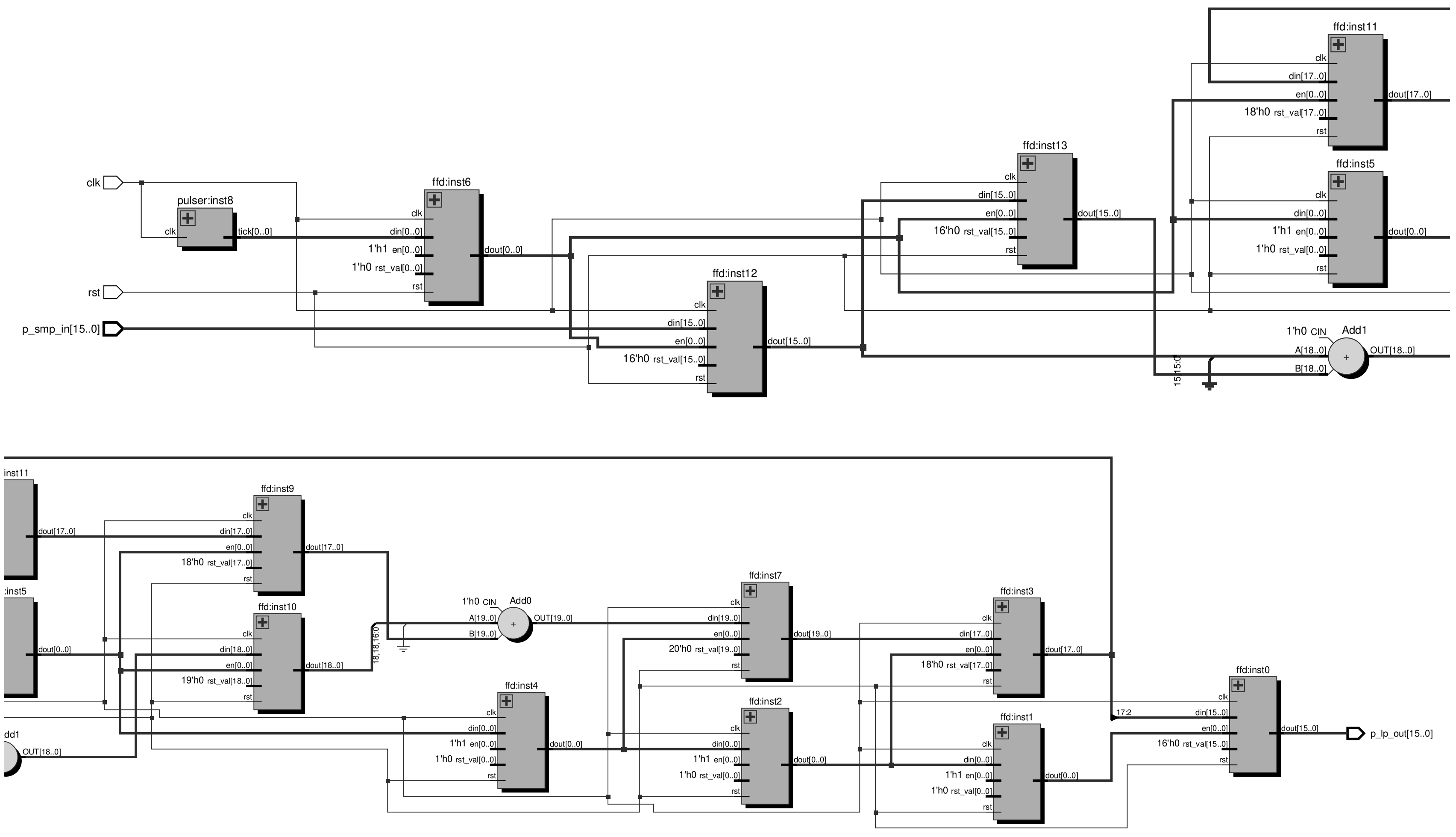


Figure 4.15: Full RTL diagram (split with overlap) done in Altera Quartus Prime of the VHDL Output Code generated for the example of Section 4.1.2.

Chapter 5

Some Audio-Oriented Hardware Blocks

In this Chapter, the collection of VHDL blocks that were made for the initial project architecture is presented. They are split into five categories according to their purpose.

While implementing these blocks, the need to do fixed point arithmetic arised. The `VHDL-2008` standard provides two fixed-point packages, called `fixed_generic_pkg` (generic module for fixed-point arithmetic) and `fixed_pkg` (an instantiation of the `fixed_generic_pkg` module with default values), that allows the instantiation and manipulation of fixed-point numerals [10]. The use of this package is the ideal solution, since the standard used to implement these VHDL blocks is also the *VHDL-2008* standard. However, one issue arises: the version of the HDL Compiler Toolchain used in this project (Intel Quartus Prime) does not currently support several *VHDL-2008* packages, including that one. So, the alternative was to use a collection of packages called `fphdl`, developed by David Bishop [26]. This library was made to support several *VHDL-2008* packages on the *VHDL-93* standard. In there lies the following files, relevant for fixed-point arithmetic: `fixed_float_types_c.vhdl` (contains the type declarations for the fixed-point package) and `fixed_pkg_c.vhdl` (contains package declaration). According to the user guide of the fixed-point package [27], these files should be placed under a library called `ieee_proposed`.

5.1 VHDL Audio Oscillator

The first VHDL modules done under the initial architecture were audio oscillators. Two kinds of oscillators and two support blocks were made for this category. Oscillators are modules responsible for creating cyclic signals according to a set frequency.

5.1.1 Sine Module (`sine`)

The first support block done for this category is called `sine`. The `sine` block is essentially a hardware block that provides an approximation of the sine value for a certain position within a cycle. The inputs of this block are a clock signal, a calculate trigger flag and the wanted phase of the sine to fetch the value from. The outputs are simply the calculated sine value and a flag that marks if the output is valid. This module also has three generic parameters that allow the change of the sine output size, the number of points the module stores inside (lookup table) and the fractional resolution of the input position. Internally, this module has a pipeline architecture that fetches points from a pre-calculated lookup table and does a linear interpolation to get a close estimate of the true sine value [28]. It is controlled by a Finite State Machine (FSM) that contains three states (`IDLE`, `WAIT_ROMO`, `GET_ROM`). On compilation of the module, it is generated a read-only memory block that contains a determined amount of points that build a quarter of a sine wave (without the last point). The module needs only to store that amount of points because those points represent a quadrant. The other three quadrants can be obtained by doing symmetry transformations on the x and y axis (Figure 5.1).

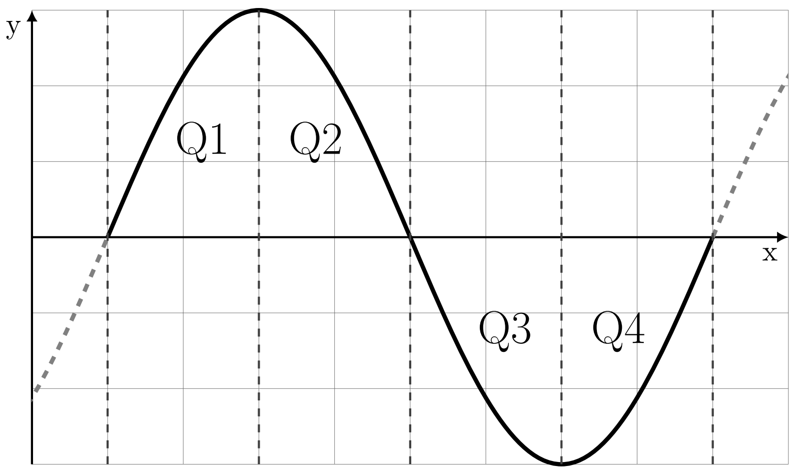


Figure 5.1: Sine wave shape and respective quadrants.

The sine values are stretched to fit the desired resolution given to the read-only memory. The number of points used in the table and the dimension of each sample is specified in the generics. Since the access to the ROM is single-port, the block needs at least two cycles to fetch all the values it needs. While the FSM is at the state `IDLE`, the address that is given to the ROM is the integer part (ignoring the fractional resolution) of the position input of the module, adjusted for the quadrant:

- In the first quadrant, no adjustment is needed;
- In the second quadrant, the double of the number of points of the table (minus two, so `TABLE_SIZE*2-2`, to be precise) is subtracted to the input position;
- In the third quadrant, the input position is subtracted with the double of the number of points of the table (`TABLE_SIZE*2-2`, again);
- In the fourth and last quadrant, the quadruple of the number of points of the table (minus four, so `TABLE_SIZE*2-2`, to be precise) is subtracted to the input position.

In order to fit a quarter of the wave into the full read-only memory, the sine wave is expanded on the x axis. To populate the values of the table it was used

$$table(n) = (2^{R-1} - 1)\sin\left(\frac{\pi}{2(S-1)}n\right).$$

The R variable is the resolution of the sample and S sets the desired table size. To avoid boundary problems, the table is generated with the sample resolution set to the maximum minus one (hence, the minus one in $2^{R-1} - 1$). This is due to the inherent property of the two's complement representation: there is one extra negative number comparing to the positive numbers [29].

When the state is not `IDLE`, the value inserted to the address port of the ROM is the input position plus one (to be able to do the interpolation). Again, the same hardware logic to adjust according to the quadrant is used on this value to make it addressable on the ROM. The output of the ROM is stored in two different flip-flop banks whether the state is the `WAIT_ROMO` or `GET_ROM`. The output of the ROM, before entering those flip-flops, passes through quadrant adjustment logic. This logic is needed to fix the offset and signal of the value, depending on which quadrant the value comes from:

- In the first and second quadrants the sine value is always positive, so no adjustment is needed;
- In the third and fourth quadrants the sine value is always negative, so a negation is applied on the output of the ROM;

After having both values from the ROM, the linear interpolation logic is the last part of this module. Now, the fractional part of the position (that was ignored for the other parts) is fetched to be used as the fractional position between both samples. The value of the interpolation is then piped to the output port of the module.

Finally, the Finite State Machine uses the following logic:

- When the state is `INIT`, and the calculation trigger flag is true, the next state is going to be `WAIT_ROMO`, the valid output flag is set to false and the requested position is fetched from the input and stored internally (and placed on the input of the ROM as already described);
- When the state is `WAIT_ROMO`, the first value from the ROM is fetched, and the next state of the FSM is `GET_ROM`;
- When the state is `GET_ROM`, the second value from the ROM is fetched, the valid flag is set to true and the next state is set to `IDLE`.

5.1.2 Generic Oscillator Module (`generic_osc`)

The `generic_osc` block is the second and last support block of this category. This block was designed to act as a bolt-on module that calculates the addresses to index a wavetable according to a variable frequency. As inputs, this block receives a clock signal, a sample enable flag (to specify when to calculate a new address), and the required frequency (in Hertz). The only output of this module is the address (phase) that should be read from the wavetable. The block also provides four generic parameters to set the sample rate of the wavetable, the size of the wavetable, the desired frequency resolution and phase resolution.

Internally, it stores the current position on a flip-flop array, and with combinational logic, calculates what will the next increment be according to the frequency present on the input of the block. The increment is calculated with the following rule:

$$inc = freq\frac{table\_size}{sample\_rate}.$$

The fraction contains constant values, so the result from that division is calculated at compile-time by the VHDL compiler. Since the frequency is a block input, it can be changed on demand, and the hardware will adjust the increment value, thus changing the speed reading of the wavetable.

5.1.3 Sine Oscillator (`sine_osc`)

The `sine_osc` is a block that acts as a sine wave oscillator. The input signals of this block are a clock signal, a sample enable flag and the desired oscillation frequency of the sine wave. The only output of this block is a sample value of the sine wave. The block expects as generic parameters the frequency value of the clock, the desired sample rate, sample resolution, size of the wavetable, fractional resolution of the input frequency and desired internal calculation fractional resolution of the phase.

This block, architecturally, uses the block `sine` to generate the sine values. It can treat the `sine` block as an abstraction of a read-only memory. Then, to generate the addresses to index the sine, this block uses the `generic_osc` block. So, technically, the `sine_osc` block serves as a top-level to instantiate a `sine` and a `generic_osc`. The generic parameters of this block are used to calculate the generic parameters of the two sub-blocks.

5.1.4 Parameterizable Oscillator (`param_osc`)

The last block of this category is the `param_osc`. This block serves as a digital oscillator that is able to reshape its wave on the fly. The inputs of this block are the hardware clock, sample enable flag and three waveshape controls. As outputs, it has the calculated sample value. The generic parameters of this block are the sample rate, the resolution of the wave shape controls, desired sample resolution, fractional frequency resolution and fractional internal phase resolution.

To control the shape of the wave, the block provides three parameters that control three different parts of a wave:

- `ctrl_a`: This input controls the rise time of the wave;
- `ctrl_b`: This input controls the up time of the wave;
- `ctrl_c`: This input controls the fall time of the wave;

These controls can be changed and the block will adapt the waveshape in real-time. These controls do not have an absolute measure of time. This is due to the fact that what really controls the full time of a cycle is the frequency the oscillator is running at. When a control is at zero, it means that the section of the wave it controls is gone. However, when a control is at the maximum value of its range, it signifies that that section of the wave should take the full cycle. To better understand the concept, Figure 5.2 provides the wave-shape generated by this block and the respective control inputs.

The biggest advantage of this block is that it can generate all the basic wave types while being capable of combining them. The last part of the wave that does not have control is

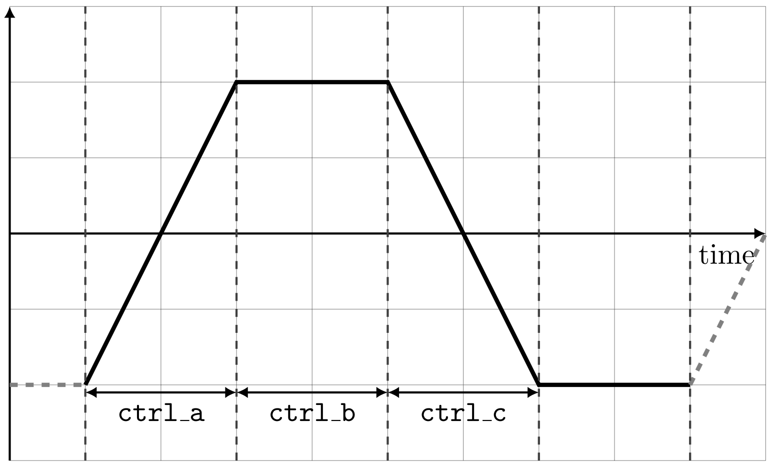
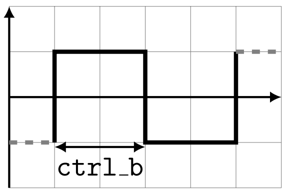


Figure 5.2: Wave-shape and control inputs of VHDL block `param_osc`.

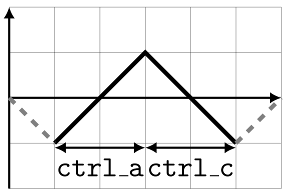
always off by default. Its size is controlled indirectly by the value of the three input controls, if the sum of all three controls is equal to one control on the maximum value, then this last part of the wave does not exist. Similarly, if the value of the controls is high enough, the last part of the wave can end up non-existent. This happens to the extent that if the value of the controls are too high, they can push other sections of the wave out of the time-frame allowed by the current frequency. In this case, the wave is simply clipped.

The following list enumerates several basic waveforms and how they can be obtained using the `param_osc` block:

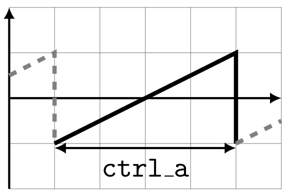
- Square Wave: Set `ctrl_a` and `ctrl_c` to zero. The `ctrl_b` control sets the duty cycle of the square wave. To achieve 50% duty cycle, `ctrl_b` should be set to the middle value of its representation. The bigger `ctrl_b` is, the larger the duty cycle will be;



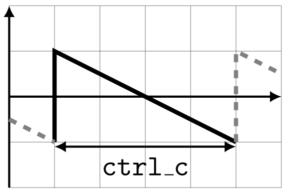
- Triangle Wave: Set `ctrl_b` to zero. The `ctrl_a` and `ctrl_c` should be set to the middle of their representation. This will provide a symmetric triangle wave. Increasing one of these two last parameters will make the wave asymmetric;



- Ascending Sawtooth Wave: Set `ctrl_b` and `ctrl_c` to zero. Set `ctrl_a` to the maximum value of the representation. To add down time before each rise of the wave, decrease `ctrl_a`;



- Descending Sawtooth Wave: Set `ctrl_a` and `ctrl_b` to zero. Set `ctrl_c` to the maximum value of the representation. To add down time after each fall of the wave, decrease `ctrl_c`.



Internally, this block is based mainly on a flip-flop array and combinational logic to calculate the increment. This block does not use a `generic_osc` since, in this particular case, having the increment calculation made inside the block reduces the hardware complexity. The flip-flop array keeps track of the current position of the wave. This block can be in four states: rise time, on time, fall time, off time. These states are coded into the two most significant bits of the flip-flop position array. For each of these states, the block uses their specific inputs (`ctrl_a` for rise time, `ctrl_b` for on time and `ctrl_c` for fall time). The output sample value of the block varies depending on which state it is:

- Rise Time: Output is equal to the current value of the position flip-flop array (excluding the state bits);
- Up Time: Output is equal to the highest value possible by the representation;
- Fall Time: Output is equal to the maximum value of the representation, minus the current value of the position flip-flop array (excluding the state bits);
- Down Time: Output is equal to the zero.

The output provided on the last list is then adjusted to be represented in two's complement, centred on zero. In addition to the position flip-flop array, there is another auxiliary flip-flop array that acts as a reset timer, to make sure that the pace of the oscillator is kept. This array is used as a counter that is incremented in a similar fashion as the `generic_osc` increments,

$$inc = freq\frac{max\_smp\_val + 1}{sample\_rate}.$$

The value *max_smp_val* is the maximum value of the sample output representation. When it overflows, it resets the position flip-flop array, starting another cycle. The increment on the position flip-flop array is calculated the same way as the reset increment, but then it is normalized to fit the control value.

5.2 VHDL Audio Filter

After the VHDL oscillators were done, it was decided to design audio filters too. In this category lies one kind of filter: state variable filter. No support blocks were made in this category, but the static variable filter uses a support block from the audio oscillators.

5.2.1 State Variable Filter (`state_var_filter`)

While Finite Impulse Response (FIR) filters are desirable for their simplicity, they do not allow neither the cut-off frequency nor the Q factor to be changed. Since these two controls are arguably important in musical expression, it was decided to implement a filter that would allow to change these parameters. The filter kind that was selected was the State Variable Filter.

A digital state variable filter is a digital counterpart of the analog state variable filter [2]. In the analog domain, it is constructed using mainly amplifier and integrator blocks. Since these have a direct translation to the digital domain, the creation of a digital state variable filter based on the analog counterpart is trivial. The biggest advantage of this filter, apart from the variable cut-off and resonance controls, is that it provides naturally the low-pass, high-pass, band-pass, and band-reject of the input all in the same block without added complexity. Figure 5.3 has the block diagram for the digital implementation of the filter.

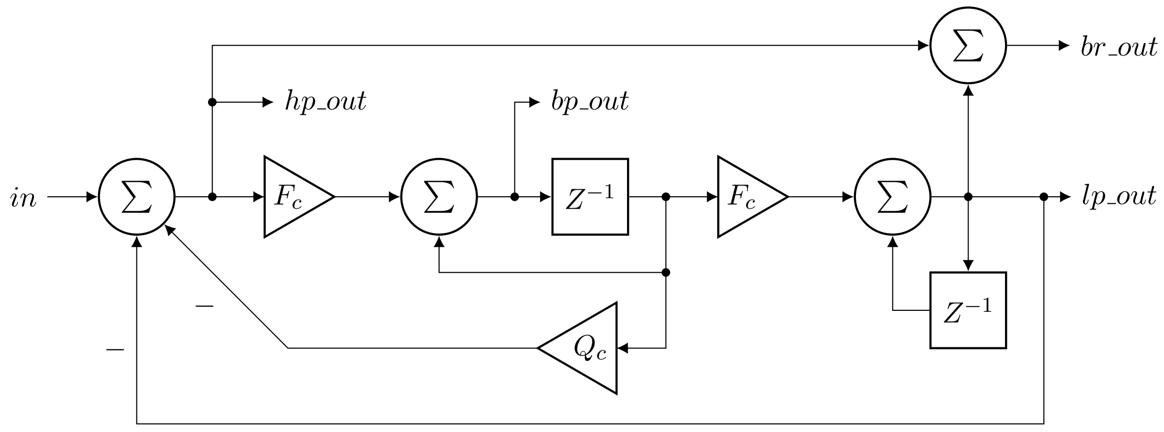


Figure 5.3: Digital State Variable Second Order Filter block diagram [2].

The filter has the following controls: F_c and Q_c. The F_c gain value controls the cut-off frequency/center frequency of the filter. Ideally, it is calculated using [2]

$$F_c = 2\sin\left(\frac{\pi F}{F_s}\right).$$

The F variable is the desired center frequency and F_s is the sampling frequency of the system. This formula can be approximated by

$$F_c \approx 2\frac{\pi F}{F_s}.$$

But it becomes increasingly inaccurate as the desired frequency gets closer to the maximum allowed frequency (half of the sampling frequency, $F \leq F_s/2$ [30]). For this reason, the use of the first one is preferred [2].

The Q_c gain value controls the Q-Factor of the system. The calculation to get that value is straight-forward,

$$Q_c = \frac{1}{q}.$$

The q variable in the equation is the desired Q-Factor. The useful range of the q parameter is from 2 ($Q_c = 0.5$) to zero ($Q_c = \infty$) [2]. When the Q-Factor is 0, the filter will self-oscillate and generates a pure (digital) sine wave.

The `state_var_filter` VHDL hardware block implements a Digital State Variable Filter. The inputs of the block are a clock, sample enable flag, sample input value, central/cut-off frequency and Q-factor value. The outputs are the four filtered results: low pass sample value, high pass sample value, band pass sample value, band reject sample value.

Internally, it instantiates the `sine` VHDL hardware block that was explained in Section 5.1.1. It is used to calculate an accurate F_c value to feed into the amplifiers. To do so, the F_c has to be converted to fit the sine argument to the table size (0 to *table_size*, opposed from 0 to 2π). So,

$$\frac{F}{F_s} \leq \frac{1}{2} \wedge F \geq 0 \Leftrightarrow 0 \leq \frac{F}{F_s} \leq \frac{1}{2} \Leftrightarrow 0 \leq \frac{\pi F}{F_s} \leq \frac{\pi}{2}.$$

So far, the conclusion to take is that the sine function will never be used above $\pi/2$. This means that only the first quadrant of the wave is really necessary and used. If we assume N as the number of points of a full cycle (so $N-1$ is the last point of the cycle), then

$$\text{(full cycle)} \sin(x) \rightarrow 0 \leq x < 2\pi \Leftrightarrow 0 \leq n \leq N-1$$

$$\text{(quarter cycle)} \rightarrow 0 \leq \frac{x}{4} \leq \frac{\pi}{2} \Leftrightarrow 0 \leq \frac{n}{4} \leq \frac{N-1}{4}$$

The scale normalization factor is $\frac{N-1}{2\pi}$, so

$$0 \leq \frac{\pi F}{F_s} \leq \frac{\pi}{2} \Leftrightarrow 0 \leq \frac{\pi F}{F_s}\frac{N-1}{2\pi} \leq \frac{\pi}{2}\frac{N-1}{2\pi} \Leftrightarrow 0 \leq \frac{F(N-1)}{2F_s} \leq \frac{N-1}{4}.$$

Therefore, the proper sine argument should be $\frac{F(N-1)}{2F_s}$. In conclusion, the full expression is

$$F_c = 2\sin\left(\frac{F(N-1)}{2F_s}\right).$$

However, we can take this process a step further. Since the required value is the double of the sine, instead of multiplying the value by two, this block instantiates the sine block with one more bit on the sample resolution. The result of multiplying this value with sample values is then shifted back to restore the correct resolution.

The rest of the block is easy to implement, since there is only the need to specify the signal path of the samples between the elementary blocks. The Z^{-1} block is generated with a flip-flop D that works at the input clock frequency and is enabled by the sample enable flag.

5.3 VHDL Audio Delay Line

To broaden the variety of types of VHDL blocks available (on the initial architecture), it was decided to also create some audio delay blocks. There were created three delay hardware blocks: `delay0`, `delay1` and `delay2`. They all have the same concept but have an increasing amount of different features.

The main rationale behind the following three hardware blocks is to emulate the behaviour of a tape delay effect. A tape delay is an analog sound effect based on a loop of tape that passes through (at least) two magnetic heads, one for writing on the tape and another for reading the tape. To decrease or increase the delay on a tape delay, the tape is sped up or slowed down. The act of changing the speed of the tape will affect the sound already recorded on it, altering the pitch.

5.3.1 Basic Variable Delay Line (`delay0`)

The `delay0` VHDL hardware block implements the behaviour of a variable delay line. As inputs, it has a clock, sample enable flag, sample value input, and speed control. The only output of this block is the sample value output. The block has the following generic parameters: clock frequency, sample rate, sample resolution, speed resolution, delay position resolution, resolution factor and maximum desired delay time.

This block emulates the behaviour of the tape delay by generating a memory block and then by circulating it while reading and writing sample values. To control the delay time, the block controls an increment that varies in proportion to the speed input parameter. Essentially, this block combines the writing and reading head into one. So, when the head reaches a position of the tape, it first reads the stored value and overwrites it with a new one. Implementation-wise, opposed to the analog counterpart, instead of moving the tape, this module moves the head. In other words, a memory pointer is stored in a flip-flop array, which is incremented in order to access another parts of the memory.

The amount of memory positions is calculated by multiplying the sample rate with the maximum delay size. This number is then divided by the resolution factor. The resolution factor allows tuning the size of the memory (by reducing the quality of the delay line).

Every time the sample enable flag turns true, the block grabs the sample value present at the current indexed position of the memory and places it in the sample output of the block. Then, it calculates the new position of the memory pointer (on a different flip-flop array) using

$$new\_pos = pos + \frac{M}{M - speed + 1}\frac{1}{resolution\_factor}, \text{ where } M = 2^{speed\_resolution}.$$

The last thing the module does in this state is to increment the current position of the memory pointer. Once the module detects that the sample enable flag has gone down, it starts fixing the memory. While the current position is different from the newly calculated position, the module places the sample value from the last valid input and increments the current pointer. This is repeated sequentially (for every clock cycle) until the current pointer position is equal to the new position.

5.3.2 Intermediate Variable Delay Line (`delay1`)

The `delay1` VHDL hardware block implements the behaviour of a variable delay line. This block is an updated version of the `delay0` block that has improved functionality. As inputs, it has a clock, sample enable flag, sample value input, speed control and reverse enable flag. The only output of this block is the sample value output. The block has the following generic parameters: clock frequency, sample rate, sample resolution, speed resolution, delay position resolution, resolution factor and maximum desired delay time.

This block is functionally similar to `delay0`, but with one relevant difference: the `delay1` block allows the memory to be travelled backwards. This is equivalent to reversing the direction of the tape on an analog tape delay.

Internally, the architectural difference to `delay0` is that, when the new position is calculated, instead of directly adding the increment, the block checks if the reverse enable flag is enabled or not. If it is, the increment is added. Otherwise, it is subtracted. The same goes for the fix process of the memory. If the increment was subtracted, the current position pointer is incremented, otherwise it is decremented.

5.3.3 Complex Variable Delay Line (`delay2`)

The last block on this category is the `delay2`. The `delay2` VHDL hardware block implements the behaviour of a variable delay line. This block is an updated version of the `delay1` block that has improved functionality. As inputs, it has a clock, sample enable flag, sample value input, read speed control, write speed control, read reverse enable flag and write reverse enable flag. The only output of this block is the sample value output. The block has the following generic parameters: clock frequency, sample rate, sample resolution, speed resolution, delay position resolution, resolution factor, maximum desired delay time and initial head offset.

This block was based on the `delay1`, but has a significant difference: the `delay2` block provides detached reading and writing heads. These heads have independent controls for speed and direction. As expected at the start, the reading head is behind the writing head. The initial offset of the reading head to the writing head is calculated based on the generic value initial head offset (in milliseconds).

The writing head has a very similar behaviour to the hybrid head that the previous two blocks had. The difference is that it does not read the value before overwriting it on the memory. The reading is all done by the reading head. The reading head only moves (and reads values) when the sample enable flag is set to true. For each head, a different increment is calculated, according to the values provided in the input of the block (speed and reverse flag of each head). The increment calculation for each head is equal to the one used on `delay1` (and `delay0`).

5.4 VHDL Audio Utility

Some more audio hardware blocks were made. But since these did not fit the last categories and are miscellaneous, this category was created. These blocks are an audio amplifier, an envelope generator and a pulse generator.

5.4.1 Audio Amplifier (`amp`)

The `amp` block is a hardware block that models the behaviour of a variable gain audio amplifier. The block only has two inputs and one output. For inputs, it has the sample value input and the gain control. As outputs, it only has the sample out value. It also has four generic parameters: sample resolution, gain control resolution, gain factor and saturation flag.

This block is the only VHDL hardware block on this collection that contains two different architectures. One architecture is made to amplify unsigned values and another to amplify signed values. The two architectures differ in the way they handle overflow and clipping. Both architectures are purely combinational, since they do not require a clock or a sample enable flag signal.

The gain factor generic value is controlled by powers of two. This means that if the gain factor is one, the real gain is unitary (this configuration can only attenuate). However, if the gain factor is, for example three, then the maximum real gain will be four. When the real gain is four, then, to achieve unitary gain, the control value has to be a fourth of the maximum control value. The formula to get the real gain is $gain = 2^{gain\_factor-1}$.

Both block architectures allow the choice between audio clipping or audio saturation, when the amplified value exceeds the maximum value of the representation. This choice is done (at compile-time) by the saturation flag. When this flag is true, the output audio will saturate. Enabling saturation increases slightly the hardware complexity of the block, since it requires an extra check and an output value override.

Internally, the block calculates an intermediate amplified result that corresponds to a lossless amplification. This is achieved by doing the multiplication between the input sample value and the input gain control value. This value is then shifted right by the resolution of the gain control, minus the resolution of the gain factor, plus one ($shift\_amount = gain\_res - gain\_factor + 1$). This shift is made to compensate the gain value of the gain factor.

By default, the output value of the block is the intermediate calculation value truncated to the sample size. When the block is generated with the saturation flag false, nothing else is done. Otherwise, and when the gain factor is more than one, the default value might be changed in case of an overflow. The saturation is handled differently in both architectures.

Saturation on the Unsigned Architecture

Overflow detection on unsigned values is straight-forward. The block only has to check the most significant bits (which are higher than the sample resolution) of the intermediate amplification calculation. If at least one of those are set to one, then overflow occurred. In terms of hardware, this operation requires doing an OR over all the most significant bits. This is done with the help of the function `or_reduce` present in the package `ieee.std_logic_misc`. Essentially, this function creates an array of OR Gates that combine all the bits provided on the input array and outputs a single bit [31]. When the maximum gain factor is bigger than one, and the result from the `or_reduce` is true, then the output gets overridden to the maximum value the sample resolution allows (every bit is equal to one).

Saturation on the Signed Architecture

Signed values (in this project) are represented in two's complement representation. This means that overflow checking is not as simple as on unsigned values. In this block, the strategy used to determine if the intermediate value does not fit the output, is to check the inverse:

check if it fits. To check if a value fits on a smaller amount of bits, the block checks if the bits of that are going to be discarded carry only the signal. In order words, it has to check if the bits to be discarded are equal to the most significant bit of the new representation. This can be implemented by applying an AND gate to all the excess bits and the most significant bit in the new representation, and to their inverse. The result of these two AND is then placed on an OR. For example, if the intermediate result has six bits, and the final representation has four bits, then (in boolean logic):

$$\text{If } intr = i_5 i_4 i_3 i_2 i_1 i_0 \text{, then}$$

$$fits = i_5 \cdot i_4 \cdot i_3 + \overline{i_5} \cdot \overline{i_4} \cdot \overline{i_3} = i_5 \cdot i_4 \cdot i_3 + (\overline{i_5 + i_4 + i_3})$$

The *intr* is the bit representation of the intermediate calculation (of size 6 bits). The *fits* expression expresses if the new value fits or not. So, if we want to check for overflow, then:

$$ovr = \overline{fits} = \overline{i_5 \cdot i_4 \cdot i_3 + (\overline{i_5 + i_4 + i_3})} = \overline{i_5 \cdot i_4 \cdot i_3} \cdot (i_5 + i_4 + i_3)$$

Generalizing to n-bit intermediate calculation to k-bit final result:

$$ovr = \overline{i_{n-1} \cdot i_{n-2} \cdot (...) \cdot i_{k-1}} \cdot (i_{n-1} + i_{n-2} + (...) + i_{k-1})$$

Similarly to `or_reduce`, the `ieee.std_logic_misc` also provides a `and_reduce` that implements a similar behaviour to `or_reduce`, but with AND gates [31]. So, implementation of this expression in pseudo-VHDL would resemble this:

```
signal s_overflow : std_logic;
signal s_interm : std_logic_vector(n-1 downto 0);
(...)
s_overflow <= (not and_reduce(s_interm(n-1 downto k-1))) or
              or_reduce(s_interm(n-1 downto k-1));
```

Finally, with this overflow flag calculated, the module can now detect clipping. The last thing it has to check before overriding the output value is if the sample clipped the top or the bottom of the representation. To do so, the block only needs to check the most significant bit of the intermediate calculation. If the bit is true, then the overflow happened on the negative part of the number so the value used to override is the biggest negative number allowed by the representation. Otherwise, the overflow happened in the positive part of the number, so the value to use to override is the biggest positive number.

5.4.2 Envelope Generator (`env_gen`)

All the previously discussed modules have a direct intervention in the creation and modification of sound. The block discussed in this Section, the envelope generator, does not have a direct intervention on the sample values. This means that the block does not receive and generate samples. Instead, this block generates a control signal.

An envelope generator is a common module in audio synthesis architectures. It is capable of generating a control signal that can be used to control parameters of other sound blocks [2]. The most common kind of envelope generator is the Attack, Decay, Sustain and Release

(ADSR) envelope generator. An ADSR envelope generator has four main controls: Attack Time, Decay Time, Sustain Level and Release Time. The last control that an envelope generator has is the gate control. The gate control is used to trigger when the envelope starts, and when should it close. A diagram of the control signal generated by an ADSR is in Figure 5.4.

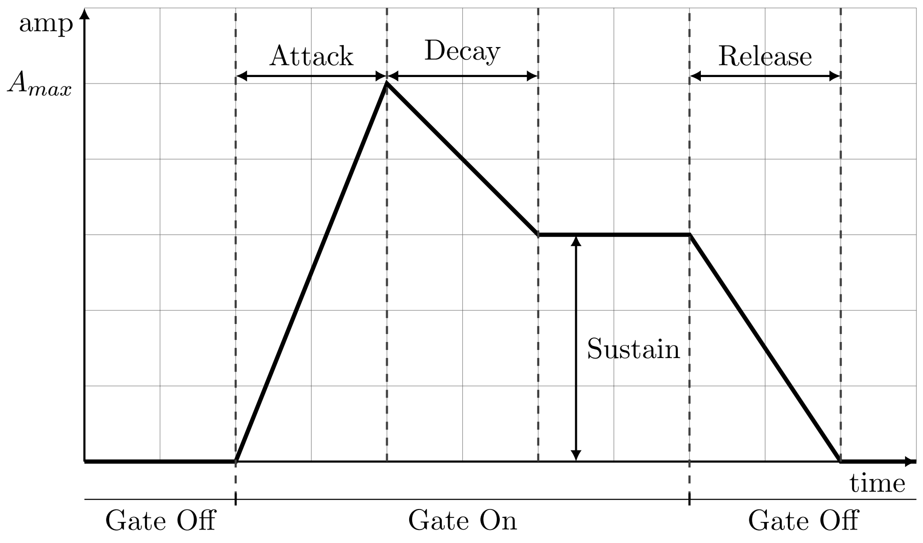


Figure 5.4: Attack Decay Sustain Release (ADSR) Envelope Generator (typical) output.

An ADSR envelope generator, typically, has five stages: one stage of each of the associated controls and an idle stage. The default state of an envelope generator is the idle state. In it, the module waits for the gate control to be opened (enabled). When it is opened, it enters the Attack state. In this stage, the output control value starts rising until it is reached the maximum value possible. The time it takes to reach that maximum is controlled by the attack time. Once it reaches the top, it starts the Decay phase (state). In this state, the output value starts falling until the sustain level is reached. Again, the time it takes to reach that level is controlled by the decay time. After the output reaches the sustain level, the state of the envelope generator changes to the Sustain state. In the Sustain state, the module holds the sustain value as the output level until the gate closes (disabled). When the gate is disabled, the envelope generator enters the Release stage. In the Release state, the output value falls until it is zero. The time the generator takes to make the output reach zero is set by the release time.

If, at any state (except the idle and release states), the gate is closed, then the module jumps instantly to the Release state. To be precise, although the attack, decay and release controls are time controls, in reality they alter the slope of rise and fall of their section of the wave. This means that, if the envelope is interrupted while it is running, the release section can be shorter in time if the current output value is smaller than the sustain value, or longer otherwise.

The `env_gen` VHDL hardware block implements the behaviour of an ADSR envelope generator. For inputs, it has a clock, sample enable flag, trigger input (gate), attack control, decay control, sustain control and release control. As output, it only has the control value out. It also has four generic parameters: the sample rate, control inputs resolution, control outputs resolution and desired time scale. This time scale parameter controls the maximum time allowed by each stage. In this implementation of an ADSR, there are only four states, since the idle state can be merged with the sustain stage, when the output is already zero.

5.4.3 Tick / Pulse Generator (`tick_creator`)

A pulse/tick is one of the most elementary blocks of a fair amount of digital hardware designs. The objective of a pulse generator is to create a periodic signal that is only active during one cycle of a certain clock. This block can be used to control and synchronize the behaviour of other hardware blocks. Its use in the initial architecture was to impose the sample rate of the system. It has only one input and one output. The input is the base clock and the output is the generated pulse. It also contains two generic parameters: the base clock frequency and the desired pulse frequency. Internally, this block has a counter that is incremented at each clock cycle. When a certain threshold is reached, the output of the block is set on high and the counter is reset. On every other value of the counter, the output is low. The threshold value is the value that is the result of the division between the base clock frequency and the desired pulse frequency.

5.5 VHDL MIDI Toolbox

The last category of VHDL blocks done for the initial architecture is the VHDL MIDI Toolbox. This toolbox contains all the blocks necessary to interpret and decode MIDI commands. Some of the blocks even build an abstraction layer that provides useful features ready to connect to other audio blocks. This category contains seven hardware blocks of varying complexity.

5.5.1 MIDI Interpreter (`midi_interpreter`)

The `midi_interpreter` VHDL hardware block is a module capable of decoding the MIDI messages received over a serial line. Once the module receives a complete MIDI message and is able to decode it, it presents the decoded message in its outputs during a clock cycle. This block has four inputs: base clock, reset flag, serial byte input and serial byte valid flag. As outputs, it has a MIDI channel number, MIDI least significant data and MIDI most significant data. In addition to these, the module has seven more output flags that signal the message type that the three other outputs have. These flags are MIDI note off event, MIDI note on event, MIDI after-touch pressure, MIDI control change, MIDI program change, MIDI channel pressure and MIDI pitch bend change flags. This module does not provide any generic parameters.

As explained in Section 2.4, MIDI has several control sequences that have different meanings. These sequences, depending on their type, have different sizes. Internally, this module contains a Finite State Machine to keep track of its status. This machine has three states: idle, first data (`DATA0`) and second data (`DATA1`). The default state (and the one that is set when the reset flag is risen) is the idle state. Additionally, by default the message type output flags are all off, unless on the clock cycles the module overrides them. Then, when the serial byte valid flag is enabled, the module reads the serial byte input field. With it, it first checks if it is a status message (most significant bit is set to one). If it is not, the block simply ignores the data. Otherwise, the module extracts the channel number and message type (status field) and sets `DATA0` as the next state. The channel number is placed on the output of the block. Now, next time the serial byte valid flag is enabled (status bit also has to be set to zero) and since the current state is `DATA0`, the module consumes another byte from the serial input. It places the received bytes on the least significant data output port and tries to decode the type:

- If the message is a program change, then the message is complete. The program change flag is enabled and the next state is the idle state;
- If the message is a channel pressure, then the message is complete. The channel pressure flag is enabled and the next state is the idle state;
- Otherwise, the message is not complete, so the next state is set to `DATA1`.

In the case that the module jumps to the `DATA1` state, then it is expecting another byte to come for the current message. So, next time the serial byte valid flag is enabled (status bit also has to be set to zero), the module consumes another byte from the serial input. It places the received bytes on the most significant data output port, sets the next state as the idle state and tries to decode the type:

- If the message is a note off event, note on event, after-touch pressure, control change or pitch bend change, then the message is complete. The correspondent flag of those events is enabled and the module finishes the process;
- Otherwise the message is not implemented by this interpreter, so the data is discarded and no output flag is enabled.

5.5.2 MIDI Voice Handler (`midi_note`)

The MIDI protocol allows the existence of polyphony inside the same channel. For a software MIDI interpreter, handling multiple notes should not pose any difficulty. But, in hardware, the same is not true. Polyphony increases the implementation complexity of a note handler module since the module has to be capable of correctly handling various notes being active at the same time. The module `midi_note` consumes note on events, note off events and after-touch pressure flags, and is capable of handling multiple active notes. These notes are placed inside voices. Each voice is capable of containing, at maximum, one note at a time. The module receives a clock, reset flag, note off event flag, note on event flag, least significant MIDI data and most significant MIDI data. The outputs of the block are a note array, note on (gate) flag array, velocity array and after-touch array. The block also contains two generic parameters: the desired amount of voices and the resolution of age counters. The size of the output arrays is controlled by the desired amount of voices. The size of each word of the note and velocity array is seven bits (as per the MIDI standard [3]). This is an experimental module that was not carefully tested since the project architecture pivoted while this block was being developed. This also meant that it is not quite optimized.

Internally, the module keeps track of the age of each voice. The age counter resolution is controlled by its generic parameter. This module also has support flip-flop arrays to keep track of the next voice to use and other parameters. The module is controlled by a Finite State Machine that contains four states: an idle state and one state for each of the messages it handles (note on, note off and aftertouch). When the reset flag is enabled, the internal states are reset and the age counters are defaulted. When the reset flag is not true then, depending on the state, the module:

- Idle State: In this state, the block is waiting for one of the input message flags to be enabled. When one of them gets enabled, the block stores the input MIDI data, and sets the next state according to the flag that was risen;

- Note On State: In this state, the block first has to check if there is an empty voice to fill in with the new note. This is checked iterating through all the voices and checking if they are on not in use. To be precise, while the block iterates the positions, it checks for the age even if the voice is being used. The oldest unused voice and oldest used voice indexes are stored on another two flip-flop arrays. This allows the block to, at the end, always use the oldest voice. This process takes, as much clock cycles as there are voices. When all the voices have been visited, the block now knows if there is free space. If there is, it places the voice on the oldest unused voice. To do so, it places there the note number, note velocity, sets the after-touch value as zero, sets the note as enabled (gate on) and updates the age of the voice. But, if all the voices are being used, then the block will overwrite the oldest voice. The process is similar to when the voice is empty, but in this case, the voice enable flag is kept on. In either case, the block resets the iteration counter and the next state is set to Idle;

- Note Off State: In this state, the block needs to find the block that contains the note that is being disabled. To do so, again, the module has to traverse all the notes and check them one by one. This is also an iterative process that should take, at maximum, as much clock cycles as there are voices. If the voice that has the desired note is found, then the block will simply turn the note enable flag off and set the velocity to the new value. If it does not find the note, it simply ignores the message. In either case, the block resets the iteration counter and the next state is set to Idle;

- After-touch State: Finally, in this state, the block needs to find the block that contains the note that requires a change of after-touch. To do so, once again, the module has to traverse all the notes and check them one by one. This process is similar to the one done on Note Off State. The only difference is that, instead of disabling the note, it updates its after-touch value. If the note is not found, it ignores the message. In both cases the block resets the iteration counter and sets the next state to Idle;

5.5.3 MIDI Note Frequency (`midi_freq`)

Now that the MIDI toolbox has a block to extract the midi notes and keep track of the voices, there is a need to make the MIDI note numbers usable. The block `midi_freq` is able to convert the MIDI note numbers into actual frequencies (assuming equal temperament) that can be directly used on other sound blocks (for example, oscillators). The only input of this block is the note number to be converted. Naturally, the only output of this block is the frequency value of the input number. This block has several generic parameters: output integer resolution, output fractional resolution, A4 note frequency (default is 440Hz), A4 note number (default is 69), and the number of equal temperament divisions (default is 12 divisions).

This block is composed basically by a read-only look up table that converts note numbers indexes to their frequency values. This table is constructed at compile-time, using a mathematical formula.

On the contemporary (western) musical tuning system, the A4 note has the frequency 440Hz. Since the human perception of sound pitch works logarithmically [32], an A5 has the double of that frequency (880Hz) and an A3 has half of that frequency (220Hz). Additionally, there are 12 tones in an octave and if equal temperament is used, then that octave interval is split in equal perception. Knowing this, it is possible to calculate the frequency of a certain note number doing a simple calculation. The note number of the A4 is 69, so

$$freq = 440 \times 2^{\frac{note-69}{12}}.$$

With the default instantiation values, the module uses this formula to calculate the table. But, this module allows for extra customization over some generation parameters. It is possible to alter the A4 center frequency, to transpose the scales (by altering the note number of A4), and to increase or decrease the number of divisions on one octave (allows the creation of microtonal scales). The previous formula with these new parameters is:

$$freq = a4\_freq\ 2^{\frac{note-a4\_num}{tet}},$$

where *tet* is the number of equal temperament divisions.

5.5.4 MIDI Channel Filter (`midi_channel_filter`)

As mentioned before on Section 2.4, the MIDI standard states that in one MIDI communication there can exist several channels. These channels are encoded on the first byte of the MIDI messages. None of the previous blocks filter the messages according to their channel. That behaviour is implemented by this block: `midi_channel_filter`.

The block `midi_channel_filter` receives a MIDI channel and all the message status flags that the block `midi_interpreter` outputs. This block outputs the same amount (and type) of flags of its input. This module also has one generic parameter: the desired filtered channel. Essentially, this block analyses the value of the channel input and compares it with the desired channel generic parameter. If they are equal, then all the input flags are transferred to the output. Otherwise, the output flags are always set to zero. This block is purely combinational.

This block is meant to serve as a middle man between the `midi_interpreter` and other MIDI blocks. Since this filter is not incorporated within the `midi_interpreter` block, a single instance of the `midi_interpreter` block can serve multiple channels on separated logic paths.

5.5.5 MIDI Control Handler (`midi_control`)

The `midi_control` block keeps track of the values that come from a MIDI controller (control change message). As inputs, it has a clock, a reset flag, the control change message flag, the least significant MIDI data and the most significant MIDI data. The only output of this block is the control value. This block has one generic parameter: the desired controller number.

Internally, this module practically behaves like a flip-flop buffer, with some control logic on the enable input. When the control message flag is true and the controller number is equal to the desired controller number, the block samples the input MIDI data bits, concatenates them to form the control value, and stores it on the flip-flop buffer. This value is retained until the next time the same conditions are met.

5.5.6 MIDI Pitch Bend Handler (`midi_pitch`)

The `midi_pitch` block keeps track of pitch bend values that come from a MIDI stream (pitch bend change message). As inputs, it has a clock, a reset flag, the pitch bend change message flag, the least significant MIDI data and the most significant MIDI data. The only output of this block is the pitch bend value. This block has no generic parameters.

Internally, this module behaves like a flip-flop buffer where the enable is connected to the pitch bend change message flag. When the pitch bend message flag is true, the block samples the input MIDI data bits, concatenate them to form the control value, and stores it on the flip-flop buffer. This value is retained until the next time the same conditions are met.

5.5.7 MIDI Channel Pressure Handler (`midi_pressure`)

The `midi_pressure` block keeps track of the channel pressure values that come from a MIDI stream (channel pressure message). As inputs, it has a clock, a reset flag, the channel pressure message flag and the least significant MIDI data (this MIDI message only takes two bytes). The only output of this block is the channel pressure value. This block has no generic parameters.

Internally, this module behaves like a flip-flop buffer where the enable is connected to the channel pressure message flag. When the channel pressure message flag is true, the block samples the input MIDI data bits, and stores it on the flip-flop buffer. This value is retained until the next time the same conditions are met.

5.6 Summary

The initial target project of this thesis was to develop a musician oriented visual language capable of being synthesized to hardware. This language would have similar characteristics to the language Pd (Section 2.5.2). Due to several reasons, this architecture ended up not being completed so that a more improved one could be developed. Some VHDL blocks were made to serve the initial architecture. Some of these blocks served as an influence to the work developed on the new architecture. These blocks can be used to generate digital audio processing/synthesis chains and to handle the MIDI protocol.

Chapter 6

Discussion and Future Work

In this thesis, we started by analysing a set of audio processing languages that are currently popular. Then, we evaluated the conception of an alternative to the typical classical audio languages. A new functional-like audio-focused language for hardware, named *rtfss*, was proposed. This language tries to provide most computer programming languages conveniences, without compromising the hardware design. Alongside other features, this language is made to be synthesized to a pipelined architecture. To further explore this language, a partial compiler was implemented. This compiler is able to analyse *rtfss* code and to synthesize it into VHDL. This compiler, albeit incomplete, is able to successfully design and manage a pipeline (with all its inherent synchronization issues). We also reflected over an alternative architecture for an audio language and analysed some hardware blocks that originated from it.

This Chapter serves as a closure to the work done in this thesis. The final remarks about how this project started, evolved and reached the current state are discussed. Finally, we make some observations about possible future work.

6.1 Final Remarks

Designing digital hardware is a rather complex task, and designing good digital hardware is an even more demanding effort. Moreover, the creation of a compiler that designs good digital hardware is an exercise usually left for large teams of experienced engineers. In this thesis, we attempted to create a language and a compiler that would focus on the synthesis of hardware designs for audio processing. It became quite clear, from the start of the development of the main architecture, that due to the time-frame of this project, the compiler would not be complete. Another factor that constrained the development of the project was the lack of experience in hardware compilers. This hindrance was one of the main reasons for having a change of architecture during this thesis. Originally, the architecture overlooked the compiler implementation, because of the underestimation of the complexity of implementation. Once this was perceived, we decided that the best step was to rethink the strategy and thus the main architecture was surfaced.

In order to have a complete *rtfss* compiler, the architecture would need a deep revamp, mainly due to the compromises taken while designing the architecture of the compiler. For example, the language allows the application of the *value at instant* operator on an arithmetic expression, while the compiler only allows the existence of them next to a stream name. This operand is caught by the compiler and is associated to a stream identifier as if it was its property. Another compromise of the compiler is the constant stream solver. This module should imitate the behaviour of hardware, by using fixed-point calculations, but instead uses floating-point arithmetics and does not handle correctly the sizing of the final value.

In our opinion, there are three significant limitations on the compiler. First, there is no support for MIDI streams. This was one of the key points of the *rtfss* language, and, unfortunately, there was no time left to implement it. The second main limitation we see in the compiler is the fact that it is not aware of the specific hardware target it will compile to. This is a very important point, being that if the compiler was aware of the hardware target, it would be able to consider the real hardware delays of the logic gates. Knowing them, the compiler would be able to pack as many operations as possible into a pipeline stage, without hurting the performance of the system. This would vastly improve the optimization of the digital design outputted by the compiler. Moreover, the compiler would be able to fill in the `max` pulse to the correct value (according to the sampling frequency of the system). The last main limitation of the compiler is that it relies too much on the arithmetic operations of VHDL. The compiler should be able to, for example, generate an adder block, using elemental logic ports. Resolving this and the previous limitation (logic timings) would vastly improve the compiler (without a substantial effort).

The *rtfss* language had a positive evolution during the development of the project. Several aspects from the syntax arose, while others were discontinued. For example, initially, the streams were not as general-purpose as they are now. There were four types of streams: `audio` stream (signed audio stream), `ctrl` stream (signed control stream), `uctrl` stream (unsigned control stream) and `midi` streams. These types had also variable sizing as the current ones have, but they were always integer type. After some deliberation, the types `audio` and `ctrl` were swapped for the more general ones `I` stream and `U` stream. One operation that turned out not being as useful as initially planned is the `gap` operator. This operator was planned to control the divergence of the delay from two streams. Although it has its use on the current specification of the language, we do think that removing it would force compilers to do inter-stream synchronization (as the one implemented does). So, the removal of the `gap` operator from the formal specification is a subject that should be assessed.

Even with all these things considered, we do believe that the project had a very positive outcome. It was a multi-disciplinary exercise, that required the employment of topics from software programming, digital hardware design, compiler design, graph theory, digital signal processing and audio synthesis. Despite of the implementation limitations, the project successfully served the purpose of simplifying the creation of hardware solutions for sound synthesis. In our opinion, the solution developed adds value to the field of languages for audio synthesis. Also, this project can be used as a starting point to other more complete architectures and similar designs.

6.2 Future Work

As already mentioned, there are several subjects in this project that can be further improved. Adding the possibility to represent numbers in binary (by the use of the prefix `0b`) should be considered. But, assuming the language will not go through a significant evolution and the compiler will not be completely redesigned, the steps that should be taken to greatly improve the project, in descending order of importance, are the following:

1. Make the compiler aware of the target hardware device, in order to be able to pack the pipeline stages with more logic;
2. Make the compiler capable of decomposing composed arithmetic operations, like addition, multiplication and division, among others;
3. Improve the constant stream solver module;
4. Implement `midi` streams in the compiler;
5. Allow the instantiation of more than one CBlock;
6. Implement *if* and *for* statements in the compiler;
7. Allow the constant stream inputs of CBlocks to control the size of other non-constant streams;
8. Implement the import mechanism and create a standard CBlock library with useful common blocks (for example, sine function).

Having these main matters resolved, in our opinion, raises the compiler to a much more matured stage. At this stage, the project could start to be used for actual real-world applications and solutions.

Other than these modifications to the compiler, we do not see any major alterations being needed in terms of software. However, in terms of hardware, one of the objectives on the early stages of the project was to create a physical MIDI interface for an FPGA. The electronic design was deliberated, but no physical implementation or testing was conducted. Implementing a hardware MIDI interface (to complement a built-in audio interface) for an FPGA would further enrich the project and create a complete ecosystem solution.

Appendix A

Simple First-Order IIR Filter Example

The filter designed and presented in this Appendix is an approximation over a first-order Butterworth low-pass filter [33]. This filter is an infinite impulse response filter (IIR) with normalized cut-off frequency of 0.2 (where 1 corresponds to half the sampling frequency). This means that if the system works at a sample-rate of 48kHz, then the filter cut-off is 4.8kHz. This filter was generated using the Matlab function call `butter(1,0.2)`. This call returns the coefficients $b_0 = 0.2452, b_1 = 0.2452, a_0 = 1.0$ and $a_0 = -0.5095$. The diagram in Figure A.1 is a valid architecture for this design. The coefficients of that Figure correspond to the coefficients returned by the Butterworth function.

Opportunely, these filter coefficients have a desirable factor: they are close to powers of two. Rounding off the coefficients to their closest power of two, we now have $b_0 = 0.25, b_1 = 0.25$ and $a_1 = -0.50$. Both the original filter and the rounded coefficients (modified) filter have very similar characteristics. The frequency response of both filters is represented in Figure A.2. The original filter is shown with a dashed line. As observed, for this application, the difference is negligible.

The mathematical representation of the filter architecture shown in Figure A.1 applied to the new filter is

$$y(n) = 0.5\ y(n-1) + 0.25\ x(n) + 0.25\ x(n-1),$$

and can be reduced to

$$y(n) = 0.5\ y(n-1) + 0.25\ (x(n) + x(n-1)).$$

This design can be further optimized (for digital hardware applications) by replacing the multiplications with bit shifts:

$$y(n) = y(n-1) \gg 1 + (x(n) + x(n-1)) \gg 2.$$

This last formula is used to implement the filter of the design first shown in Section 1.1.

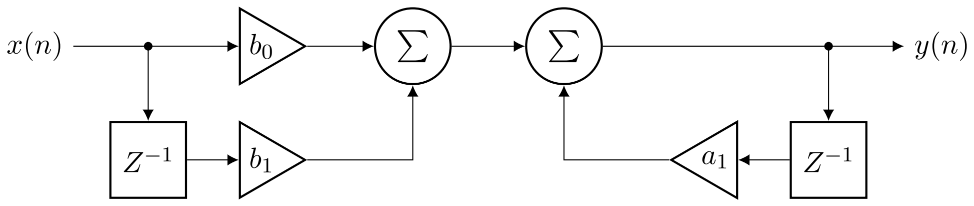


Figure A.1: First-order IIR filter diagram.

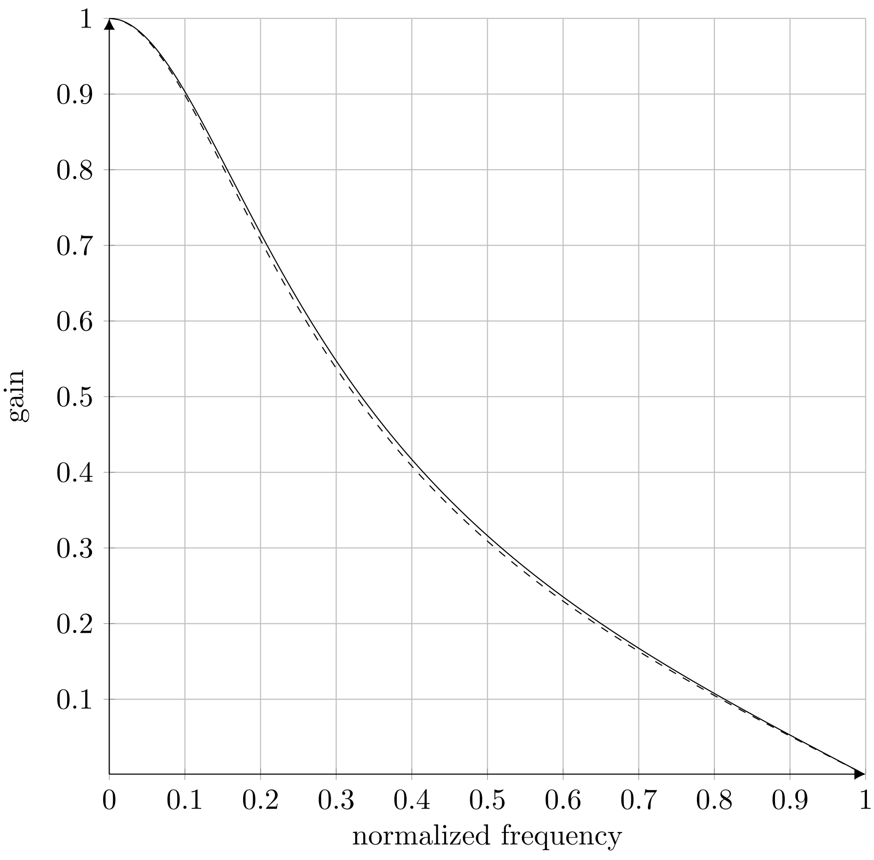


Figure A.2: Frequency response of a filter (dashed) and its approximation (solid).

Appendix B

VHDL Code Generated for the Support Example of Section 4.1.2

```
--Pre fabricated module
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

entity pulser is
  generic(CLK_FREQ : real;
          TICK_FREQ : real);
  port(clk : in std_logic;
       tick : out std_logic_vector(0 downto 0));
end pulser;

architecture behavioural of pulser is
  constant MAX_CNT : positive := positive(CLK_FREQ/TICK_FREQ);
  constant N_BITS : positive := positive(ceil(log2(real(MAX_CNT))));
  signal s_cnt : unsigned(N_BITS-1 downto 0);
begin
  process(clk)
  begin
    if rising_edge(clk) then
      s_cnt<=s_cnt+1;
      if s_cnt=MAX_CNT-1 then
        s_cnt<=(others=>'0');
      end if;
    end if;
  end process;
  tick<="1" when s_cnt=MAX_CNT-1 else "0";
end behavioural;
```

```
--Pre fabricated module
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

entity ffd is
  generic(SIZE : positive);
  port(clk : in std_logic;
       en : in std_logic_vector(0 downto 0);
       rst : in std_logic;
       rst_val : in std_logic_vector(SIZE-1 downto 0);
       din : in std_logic_vector(SIZE-1 downto 0);
       dout : out std_logic_vector(SIZE-1 downto 0));
end ffd;

architecture behavioural of ffd is
  signal s_data : std_logic_vector(SIZE-1 downto 0);
begin
  process(clk)
  begin
    if(rising_edge(clk)) then
      if(rst='1') then
        s_data<=rst_val;
      elsif(en="1") then
        s_data<=din;
      end if;
    end if;
  end process;
  dout<=s_data;
end behavioural;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity rtfss is
  generic(CLK_FREQ : real);
  port(clk : in std_logic;
       rst : in std_logic;
       p_lp_out : out std_logic_vector(15 downto 0);
       p_smp_in : in std_logic_vector(15 downto 0));
end rtfss;

architecture impl of rtfss is
  signal n1 : std_logic_vector(15 downto 0);
  signal n71 : std_logic_vector(15 downto 0);
  signal n72 : std_logic_vector(0 downto 0);
  signal n70 : std_logic_vector(17 downto 0);
  signal n65 : std_logic_vector(0 downto 0);
  signal n2 : std_logic_vector(17 downto 0);
  constant n69 : std_logic_vector(1 downto 0) := "10";
  signal n47 : std_logic_vector(0 downto 0);
  signal n64 : std_logic_vector(17 downto 0);
  signal n33 : std_logic_vector(0 downto 0);
  signal n63 : std_logic_vector(19 downto 0);
  signal n17 : std_logic_vector(0 downto 0);
  signal n58 : std_logic_vector(19 downto 0);
  constant n62 : std_logic_vector(0 downto 0) := "1";
  signal n16 : std_logic_vector(0 downto 0);
  signal n12 : std_logic_vector(19 downto 0);
  signal n56 : std_logic_vector(19 downto 0);
  signal n57 : std_logic_vector(19 downto 0);
  signal n54 : std_logic_vector(19 downto 0);
  constant n55 : std_logic_vector(0 downto 0) := "1";
  signal n53 : std_logic_vector(17 downto 0);
  signal n45 : std_logic_vector(17 downto 0);
  signal n28 : std_logic_vector(17 downto 0);
  signal n43 : std_logic_vector(18 downto 0);
  signal n11 : std_logic_vector(18 downto 0);
  signal n7 : std_logic_vector(18 downto 0);
  signal n27 : std_logic_vector(18 downto 0);
  constant n10 : std_logic_vector(0 downto 0) := "1";
  signal n41 : std_logic_vector(18 downto 0);
  signal n42 : std_logic_vector(18 downto 0);
  signal n25 : std_logic_vector(18 downto 0);
  constant n26 : std_logic_vector(0 downto 0) := "1";
  signal n39 : std_logic_vector(18 downto 0);
  constant n40 : std_logic_vector(1 downto 0) := "10";
  signal n32 : std_logic_vector(15 downto 0);
  signal n19 : std_logic_vector(17 downto 0);
  signal n0 : std_logic_vector(15 downto 0);
  signal n6 : std_logic_vector(17 downto 0);
  signal n31 : std_logic_vector(17 downto 0);
  constant n5 : std_logic_vector(1 downto 0) := "10";
  signal n29 : std_logic_vector(17 downto 0);
  constant n30 : std_logic_vector(1 downto 0) := "10";
  signal n20 : std_logic_vector(15 downto 0);
begin

  inst0 : entity work.ffd
    generic map(SIZE=>16)
    port map(clk=>clk,en=>n72,rst=>rst,rst_val=>(others=>'0'),din=>n71,dout=>
        n1);
```

```
p_lp_out <= n1;
n71 <= std_logic_vector(resize(signed(n70),16));

inst1 : entity work.ffd
  generic map(SIZE=>1)
  port map(clk=>clk,en=>"1",rst=>rst,rst_val=>(others=>'0'),din=>n65,dout=>
      n72);

n70 <= std_logic_vector(shift_right(signed(n2),to_integer(unsigned(n69))));

inst2 : entity work.ffd
  generic map(SIZE=>1)
  port map(clk=>clk,en=>"1",rst=>rst,rst_val=>(others=>'0'),din=>n47,dout=>
      n65);

inst3 : entity work.ffd
  generic map(SIZE=>18)
  port map(clk=>clk,en=>n65,rst=>rst,rst_val=>(others=>'0'),din=>n64,dout=>
      n2);

inst4 : entity work.ffd
  generic map(SIZE=>1)
  port map(clk=>clk,en=>"1",rst=>rst,rst_val=>(others=>'0'),din=>n33,dout=>
      n47);

n64 <= std_logic_vector(resize(signed(n63),18));

inst5 : entity work.ffd
  generic map(SIZE=>1)
  port map(clk=>clk,en=>"1",rst=>rst,rst_val=>(others=>'0'),din=>n17,dout=>
      n33);

n63 <= std_logic_vector(shift_right(signed(n58),to_integer(unsigned(n62))))
    ;

inst6 : entity work.ffd
  generic map(SIZE=>1)
  port map(clk=>clk,en=>"1",rst=>rst,rst_val=>(others=>'0'),din=>n16,dout=>
      n17);

inst7 : entity work.ffd
  generic map(SIZE=>20)
  port map(clk=>clk,en=>n47,rst=>rst,rst_val=>(others=>'0'),din=>n12,dout=>
      n58);
```

```
inst8 : entity work.pulser
  generic map(CLK_FREQ=>CLK_FREQ,TICK_FREQ=>inf)
  port map(clk=>clk,tick=>n16);

n12 <= std_logic_vector(signed(n56) + signed(n57));
n56 <= std_logic_vector(shift_left(signed(n54),to_integer(unsigned(n55))));
n57 <= std_logic_vector(resize(signed(n53),20));
n54 <= std_logic_vector(resize(signed(n45),20));

inst9 : entity work.ffd
  generic map(SIZE=>18)
  port map(clk=>clk,en=>n33,rst=>rst,rst_val=>(others=>'0'),din=>n28,dout=>
      n53);

n45 <= std_logic_vector(resize(signed(n43),18));
n28 <= std_logic_vector(resize(signed(n11),18));

inst10 : entity work.ffd
  generic map(SIZE=>19)
  port map(clk=>clk,en=>n33,rst=>rst,rst_val=>(others=>'0'),din=>n7,dout=>
      n43);

n11 <= std_logic_vector(shift_right(signed(n27),to_integer(unsigned(n10))))
    ;
n7 <= std_logic_vector(signed(n41) + signed(n42));
n27 <= std_logic_vector(shift_left(signed(n25),to_integer(unsigned(n26))));
n41 <= std_logic_vector(shift_left(signed(n39),to_integer(unsigned(n40))));
n42 <= std_logic_vector(resize(signed(n32),19));
n25 <= std_logic_vector(resize(signed(n19),19));
n39 <= std_logic_vector(resize(signed(n0),19));
n32 <= std_logic_vector(resize(signed(n6),16));

inst11 : entity work.ffd
  generic map(SIZE=>18)
  port map(clk=>clk,en=>n17,rst=>rst,rst_val=>(others=>'0'),din=>n2,dout=>
      n19);

inst12 : entity work.ffd
  generic map(SIZE=>16)
  port map(clk=>clk,en=>n17,rst=>rst,rst_val=>(others=>'0'),din=>p_smp_in,
      dout=>n0);

n6 <= std_logic_vector(shift_right(signed(n31),to_integer(unsigned(n5))));
n31 <= std_logic_vector(shift_left(signed(n29),to_integer(unsigned(n30))));
n29 <= std_logic_vector(resize(signed(n20),18));
```

```
  inst13 : entity work.ffd
    generic map(SIZE=>16)
    port map(clk=>clk,en=>n17,rst=>rst,rst_val=>(others=>'0'),din=>n0,dout=>
        n20);

end impl;
```

Appendix C

Another Example in *rtfss*

Revisiting the *rtfss* example first introduced in Section 4.1.2, we can observe that the design is capable of processing an input audio stream and filter it. However, that example does not synthesize new audio. In this Appendix, this example is modified to generate a 440Hz square wave that is fed to the already existent IIR filter. This design is an altered version of the example introduced in Section 3.9.2.

There are multiple possible ways to generate a square wave. For example, a counter can be used to keep track of the phase position of the wave. The output value would be low in the first half of that counter's range and high at the other half. Instead of this method, we can use a feature of *rtfss* to generate the wave: the pulse domain. In order to generate a 440Hz square wave, the sample value has to change polarity 880 times per second. To do so, we can create a pulse with the frequency of 880Hz that controls an audio stream that only needs to inverts polarity at every pulse. This was the method used in the following design:

```
cblock@smpp main(:: I16 sqr_out,I16 lp_out){
    //Generate square wave
    pulse sq_p 880Hz;
    I16@smpp sqr_smpp = sqr_wave;
    sqr_out = sqr_wave;
    I16@sq_p sqr_wave = not (sqr_wave'-1); //Square Wave of 440Hz
    sqr_wave'-1 = I16(0x7FFF);

    //Low pass 0.2 IIR Filter
    I16.2@smpp lowpass = I16.2((lowpass'-1)>>1+(sqr_smpp+(sqr_smpp'-1))>>2);
    lp_out=I16(lowpass);
}
```

This example contains a CBlock that has two output streams: `sqr_out` (unaltered square wave) and `lp_out` (filtered square wave). As explained, the square wave is generated using a pulse with double the frequency of the desired tone (pulse `sq_p`) that is fed to the stream `sqr_wave`. Every pulse cycle, this stream only needs to invert the previous value. By default, all the previous stream values are zero. So, in order for the stream to oscillate, the most recent past value has to be set. Since the stream is signed and has 16 integer bits, the correct (highest) value that should be set is $2^{16} - 1 = 32767_{10} = 7FFF_{16}$. When there is a pulse, the value flips to 8000_{16}, that is -32768_{10} in two's complement.

Now that the square wave is generated, it is redirected to the output by attributing its value to the sqr_out stream. It is important to note that this attribution crosses the border between two pulse domains (sq_p and smpp pulse domains). This is due to the fact that the CBlock is controlled by the smpp pulse. The *rtfss* specification ensures that it is capable of handling crosses between pulse domain transformations seamlessly. This effect can be observed on the Architecture Graph of this design, represented in Figure C.1.

Finally, the square wave has to be connected to the IIR filter. To do so, the square wave stream needs to be transferred again to the other pulse domain. In this case, this operation requires some attention. One might think that the only step needed is to use the sqr_wave stream inside the lowpass stream expression. But, if carefully analysed, this would not implement the desired behaviour. The filter requires the input to be indexed with previous instants. If we feed the sqr_wave stream directly, the previous values of this stream are relative to its pulse. So, it would always return the negation of the current stream value. So, by doing sqr_wave+(sqr_wave'-1), the result would always be -1, even if this operation is done inside a stream with a different pulse. To avoid this, the design has another stream named sqr_smpp on the smpp pulse domain, that only serves to translate the sqr_wave from one pulse domain to the other. Accessing the previous values of this new stream yields the wanted results.

The compilation of this design created the Architecture Graph of Figure C.1 and the Final Representation Graph of Figure C.2. The output VHDL code generated by the compiler is laid out below. To successfully compile this design, the steps explained in Section 4.10.1 should also be applied to this case:

```
--Pre fabricated module
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

entity pulser is
  generic(CLK_FREQ : real;
          TICK_FREQ : real);
  port(clk : in std_logic;
       tick : out std_logic_vector(0 downto 0));
end pulser;

architecture behavioural of pulser is
  constant MAX_CNT : positive := positive(CLK_FREQ/TICK_FREQ);
  constant N_BITS : positive := positive(ceil(log2(real(MAX_CNT))));
  signal s_cnt : unsigned(N_BITS-1 downto 0);
begin
  process(clk)
  begin
    if rising_edge(clk) then
      s_cnt<=s_cnt+1;
      if s_cnt=MAX_CNT-1 then
        s_cnt<=(others=>'0');
      end if;
    end if;
  end process;
  tick<="1" when s_cnt=MAX_CNT-1 else "0";
end behavioural;

--Pre fabricated module
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

entity ffd is
  generic(SIZE : positive);
  port(clk : in std_logic;
       en : in std_logic_vector(0 downto 0);
       rst : in std_logic;
       rst_val : in std_logic_vector(SIZE-1 downto 0);
       din : in std_logic_vector(SIZE-1 downto 0);
       dout : out std_logic_vector(SIZE-1 downto 0));
end ffd;

architecture behavioural of ffd is
  signal s_data : std_logic_vector(SIZE-1 downto 0);
begin
  process(clk)
  begin
    if(rising_edge(clk)) then
      if(rst='1') then
        s_data<=rst_val;
      elsif(en="1") then
        s_data<=din;
      end if;
    end if;
  end process;
  dout<=s_data;
end behavioural;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity rtfss is
  generic(CLK_FREQ : real);
  port(clk : in std_logic;
       rst : in std_logic;
       p_sqr_out : out std_logic_vector(15 downto 0);
       p_lp_out : out std_logic_vector(15 downto 0));
```

```
end rtfss;

architecture impl of rtfss is
  signal n0 : std_logic_vector(15 downto 0);
  signal n1 : std_logic_vector(15 downto 0);
  signal n3 : std_logic_vector(15 downto 0);
  signal n23 : std_logic_vector(0 downto 0);
  signal n54 : std_logic_vector(15 downto 0);
  signal n55 : std_logic_vector(0 downto 0);
  signal n27 : std_logic_vector(15 downto 0);
  signal n32 : std_logic_vector(0 downto 0);
  signal n19 : std_logic_vector(0 downto 0);
  signal n53 : std_logic_vector(17 downto 0);
  signal n51 : std_logic_vector(0 downto 0);
  signal n6 : std_logic_vector(15 downto 0);
  signal n26 : std_logic_vector(0 downto 0);
  signal n4 : std_logic_vector(17 downto 0);
  constant n52 : std_logic_vector(1 downto 0) := "10";
  signal n42 : std_logic_vector(0 downto 0);
  signal n22 : std_logic_vector(15 downto 0);
  signal n21 : std_logic_vector(0 downto 0);
  signal n50 : std_logic_vector(17 downto 0);
  signal n33 : std_logic_vector(0 downto 0);
  signal n20 : std_logic_vector(0 downto 0);
  signal n49 : std_logic_vector(18 downto 0);
  signal n47 : std_logic_vector(18 downto 0);
  constant n48 : std_logic_vector(0 downto 0) := "1";
  signal n16 : std_logic_vector(18 downto 0);
  signal n43 : std_logic_vector(18 downto 0);
  signal n46 : std_logic_vector(18 downto 0);
  signal n41 : std_logic_vector(17 downto 0);
  signal n44 : std_logic_vector(18 downto 0);
  constant n45 : std_logic_vector(0 downto 0) := "1";
  signal n31 : std_logic_vector(17 downto 0);
  signal n40 : std_logic_vector(16 downto 0);
  signal n11 : std_logic_vector(18 downto 0);
  signal n15 : std_logic_vector(18 downto 0);
  signal n30 : std_logic_vector(18 downto 0);
  constant n10 : std_logic_vector(0 downto 0) := "1";
  signal n39 : std_logic_vector(18 downto 0);
  constant n14 : std_logic_vector(1 downto 0) := "10";
  signal n28 : std_logic_vector(18 downto 0);
  constant n29 : std_logic_vector(0 downto 0) := "1";
  signal n37 : std_logic_vector(18 downto 0);
  constant n38 : std_logic_vector(1 downto 0) := "10";
  signal n24 : std_logic_vector(17 downto 0);
  signal n36 : std_logic_vector(16 downto 0);
  signal n13 : std_logic_vector(16 downto 0);
  signal n34 : std_logic_vector(16 downto 0);
  signal n35 : std_logic_vector(16 downto 0);
  signal n2 : std_logic_vector(15 downto 0);
  signal n25 : std_logic_vector(15 downto 0);

begin

  inst0 : entity work.ffd
    generic map(SIZE=>16)
    port map(clk=>clk,en=>n23,rst=>rst,rst_val=>(others=>'0'),din=>n3,dout=>
        n0);

  p_sqr_out <= n0;

  inst1 : entity work.ffd
    generic map(SIZE=>16)
    port map(clk=>clk,en=>n55,rst=>rst,rst_val=>(others=>'0'),din=>n54,dout=>
        n1);

  p_lp_out <= n1;

  inst2 : entity work.ffd
    generic map(SIZE=>16)
    port map(clk=>clk,en=>n32,rst=>rst,rst_val=>"0111111111111111",din=>n27,
        dout=>n3);

  inst3 : entity work.ffd
    generic map(SIZE=>1)
    port map(clk=>clk,en=>"1",rst=>rst,rst_val=>(others=>'0'),din=>n19,dout=>
        n23);

  n54 <= std_logic_vector(resize(signed(n53),16));

  inst4 : entity work.ffd
    generic map(SIZE=>1)
    port map(clk=>clk,en=>"1",rst=>rst,rst_val=>(others=>'0'),din=>n51,dout=>
        n55);

  inst5 : entity work.ffd
    generic map(SIZE=>16)
    port map(clk=>clk,en=>n26,rst=>rst,rst_val=>(others=>'0'),din=>n6,dout=>
        n27);

  inst6 : entity work.ffd
    generic map(SIZE=>1)
    port map(clk=>clk,en=>"1",rst=>rst,rst_val=>(others=>'0'),din=>n26,dout=>
      n32);

inst7 : entity work.pulser
  generic map(CLK_FREQ=>CLK_FREQ,TICK_FREQ=>inf)
  port map(clk=>clk,tick=>n19);

n53 <= std_logic_vector(shift_right(signed(n4),to_integer(unsigned(n52))));

inst8 : entity work.ffd
  generic map(SIZE=>1)
  port map(clk=>clk,en=>"1",rst=>rst,rst_val=>(others=>'0'),din=>n42,dout=>
      n51);

n6 <= not n22;

inst9 : entity work.ffd
  generic map(SIZE=>1)
  port map(clk=>clk,en=>"1",rst=>rst,rst_val=>(others=>'0'),din=>n21,dout=>
      n26);

inst10 : entity work.ffd
  generic map(SIZE=>18)
  port map(clk=>clk,en=>n51,rst=>rst,rst_val=>(others=>'0'),din=>n50,dout=>
      n4);

inst11 : entity work.ffd
  generic map(SIZE=>1)
  port map(clk=>clk,en=>"1",rst=>rst,rst_val=>(others=>'0'),din=>n33,dout=>
      n42);

inst12 : entity work.ffd
  generic map(SIZE=>16)
  port map(clk=>clk,en=>n21,rst=>rst,rst_val=>(others=>'0'),din=>n3,dout=>
      n22);

inst13 : entity work.ffd
  generic map(SIZE=>1)
  port map(clk=>clk,en=>"1",rst=>rst,rst_val=>(others=>'0'),din=>n20,dout=>
      n21);

n50 <= std_logic_vector(resize(signed(n49),18));

inst14 : entity work.ffd
  generic map(SIZE=>1)
  port map(clk=>clk,en=>"1",rst=>rst,rst_val=>(others=>'0'),din=>n23,dout=>
      n33);
```

```
inst15 : entity work.pulser
  generic map(CLK_FREQ=>CLK_FREQ,TICK_FREQ=>880.000000)
  port map(clk=>clk,tick=>n20);

n49 <= std_logic_vector(shift_right(signed(n47),to_integer(unsigned(n48))))
    ;

inst16 : entity work.ffd
  generic map(SIZE=>19)
  port map(clk=>clk,en=>n42,rst=>rst,rst_val=>(others=>'0'),din=>n16,dout=>
      n47);

n16 <= std_logic_vector(signed(n43) + signed(n46));
n43 <= std_logic_vector(resize(signed(n41),19));
n46 <= std_logic_vector(shift_left(signed(n44),to_integer(unsigned(n45))));

inst17 : entity work.ffd
  generic map(SIZE=>18)
  port map(clk=>clk,en=>n33,rst=>rst,rst_val=>(others=>'0'),din=>n31,dout=>
      n41);

n44 <= std_logic_vector(resize(signed(n40),19));
n31 <= std_logic_vector(resize(signed(n11),18));
n40 <= std_logic_vector(resize(signed(n15),17));
n11 <= std_logic_vector(shift_right(signed(n30),to_integer(unsigned(n10))))
    ;
n15 <= std_logic_vector(shift_right(signed(n39),to_integer(unsigned(n14))))
    ;
n30 <= std_logic_vector(shift_left(signed(n28),to_integer(unsigned(n29))));
n39 <= std_logic_vector(shift_left(signed(n37),to_integer(unsigned(n38))));
n28 <= std_logic_vector(resize(signed(n24),19));
n37 <= std_logic_vector(resize(signed(n36),19));

inst18 : entity work.ffd
  generic map(SIZE=>18)
  port map(clk=>clk,en=>n23,rst=>rst,rst_val=>(others=>'0'),din=>n4,dout=>
      n24);

inst19 : entity work.ffd
  generic map(SIZE=>17)
  port map(clk=>clk,en=>n33,rst=>rst,rst_val=>(others=>'0'),din=>n13,dout=>
      n36);

n13 <= std_logic_vector(signed(n34) + signed(n35));
n34 <= std_logic_vector(resize(signed(n2),17));
n35 <= std_logic_vector(resize(signed(n25),17));
```

```
  inst20 : entity work.ffd
    generic map(SIZE=>16)
    port map(clk=>clk,en=>n23,rst=>rst,rst_val=>(others=>'0'),din=>n3,dout=>
        n2);

  inst21 : entity work.ffd
    generic map(SIZE=>16)
    port map(clk=>clk,en=>n23,rst=>rst,rst_val=>(others=>'0'),din=>n2,dout=>
        n25);

end impl;
```

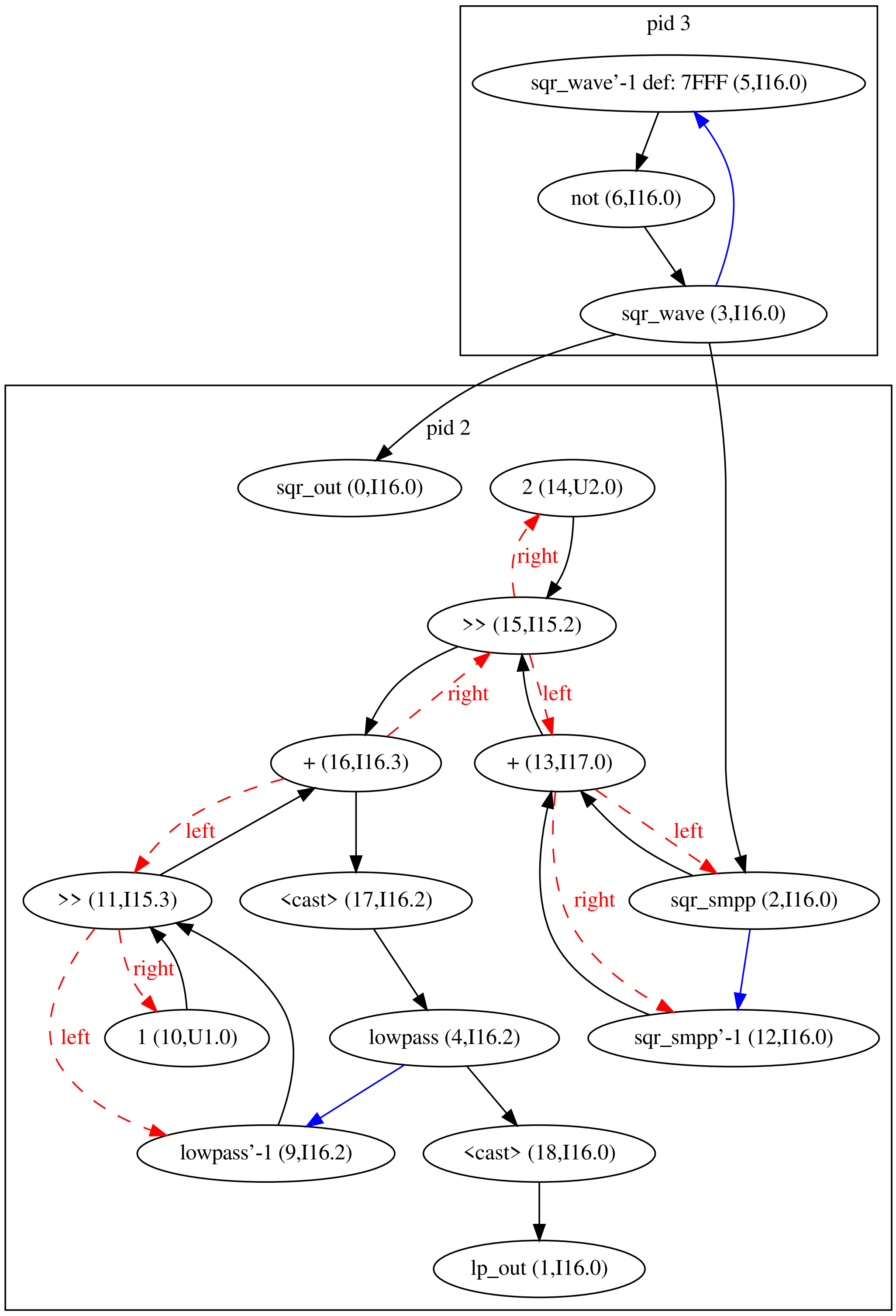


Figure C.1: Full Architecture Graph of the example in Appendix C (GraphViz).

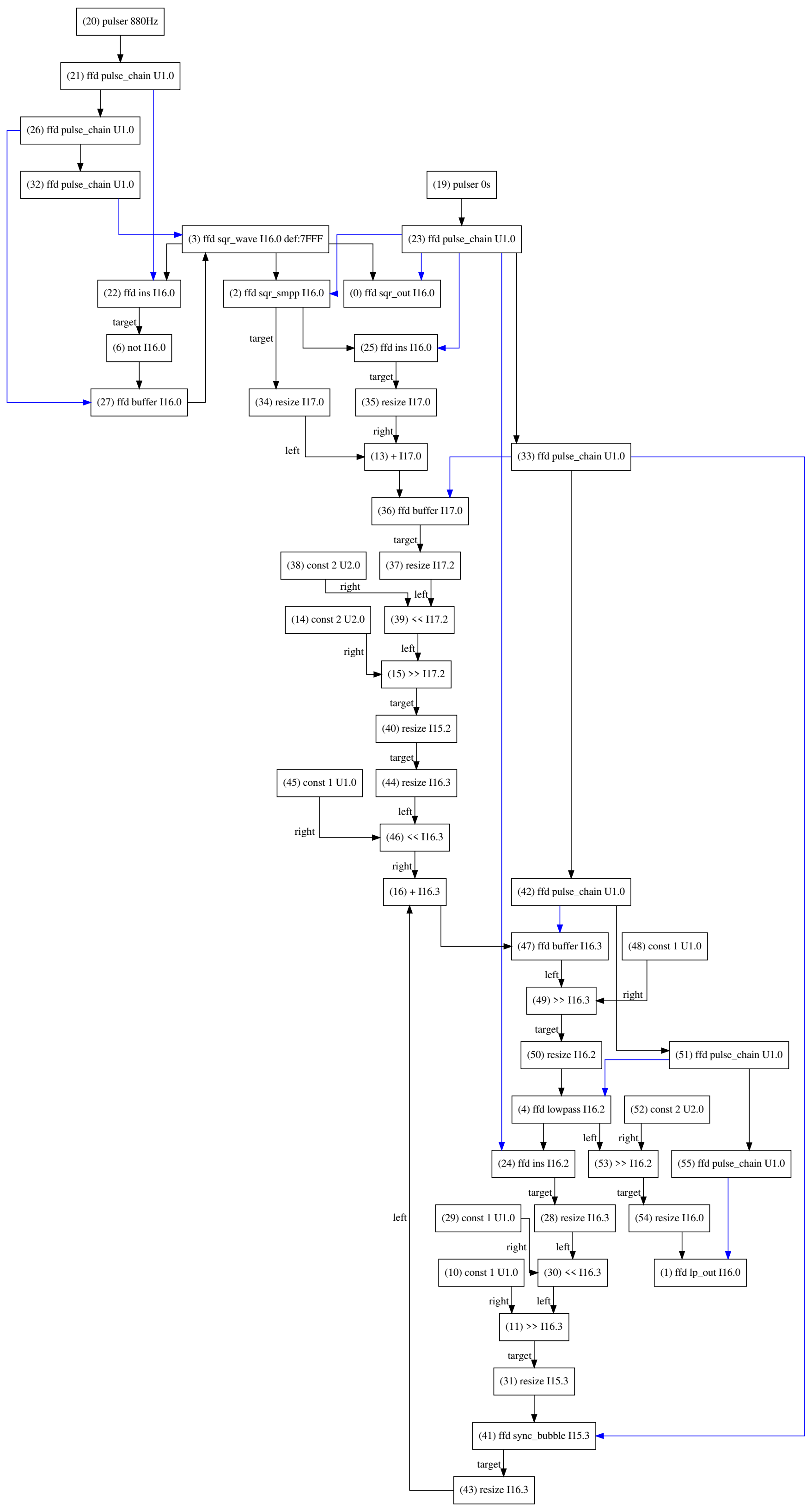


Figure C.2: Full Final Representation Graph of the example in Appendix C (GraphViz).

Appendix D

The *rtfss* ANTLR4 Grammar

```
grammar rtfss;

//statm=statement
//decl=declaration
//assign=assignment

entry_point: use_stat* cblock_decl* EOF;

use_stat: USE std_id=(IDENTIFIER | STRING_LITERAL);

cblock_decl: CBLOCK PULSE_SEP pulse_freq IDENTIFIER cblock_args code_block;

cblock_args: PAR_OPEN (const_args=cblock_arg_list ARG_GROUP_SEP)? in_args=
    cblock_arg_list ARG_GROUP_SEP out_args=cblock_arg_list PAR_CLOSE;
cblock_arg_list: ((data_type var_name COMMA)* data_type var_name)?;

code_block: CURLY_OPEN statm* CURLY_CLOSE;

statm: nonterminated_statm | terminated_statm TERMINATOR;

terminated_statm: var_assign | data_decl | pulse_decl | cblock_inst;
nonterminated_statm: if_statm | for_statm;

if_statm: IF PAR_OPEN expr PAR_CLOSE code_block
                   (ELSEIF PAR_OPEN expr PAR_CLOSE code_block)*
                   (ELSE code_block)?;

for_statm: FOR PAR_OPEN IDENTIFIER IN expr TO expr (INC expr)? PAR_CLOSE
    code_block;

cblock_inst: BRAC_OPEN (oargs+=var_name COMMA)* oargs+=var_name BRAC_CLOSE
    EQUAL cblock_inst_short;
cblock_inst_short: IDENTIFIER PULSE_SEP pulse_freq PAR_OPEN ((cargs+=expr
    COMMA)* cargs+=expr)? (ARG_GROUP_SEP ((iargs+=expr COMMA)* iargs+=expr)?)
    ? PAR_CLOSE;

pulse_decl: PULSE IDENTIFIER PULSE_LITERAL;
data_decl: data_type PULSE_SEP pulse_freq stream_id (EQUAL expr)?;
var_assign: stream_id assign_op expr;
assign_op: EQUAL | COMBOP;

expr:
             SUB expr          #negExpr
       | PAR_OPEN data_type PAR_CLOSE expr   #castAExpr
       | data_type PAR_OPEN expr PAR_CLOSE    #castBExpr
       | cblock_inst_short       #cblkExpr
       | PAR_OPEN expr PAR_CLOSE      #parExpr
       | CURLY_OPEN (expr COMMA)* expr CURLY_CLOSE #arrExpr
       | PROP expr         #propExpr
       | expr GAP expr        #gapExpr
       | expr (SL|SR) expr       #shiftExpr
       | expr (RL|RR) expr       #rotExpr
       | expr AND expr        #andExpr
       | expr OR expr        #orExpr
       | expr XOR expr        #xorExpr
       | NOT expr         #notExpr
       | expr MULT expr        #multExpr
       | expr DIV expr        #divExpr
       | expr MOD expr        #modExpr
       | expr ADD expr        #addExpr
       | expr SUB expr        #subExpr
       | expr LOGIC_OP expr       #evalExpr

       //Midi operators
       | NOTEOF PAR_OPEN expr COMMA expr ARG_GROUP_SEP expr COMMA expr
          PAR_CLOSE #noteofExpr
       | FREQOF PAR_OPEN expr COMMA expr ARG_GROUP_SEP expr COMMA expr
          PAR_CLOSE #freqofExpr
       | VELOF PAR_OPEN expr COMMA expr ARG_GROUP_SEP expr COMMA expr
          PAR_CLOSE #velofExpr
       | PATOF PAR_OPEN expr COMMA expr ARG_GROUP_SEP expr COMMA expr
          PAR_CLOSE #patofExpr
       | NNTOF PAR_OPEN expr COMMA expr ARG_GROUP_SEP expr COMMA expr
          PAR_CLOSE #nntofExpr
       | CCOF PAR_OPEN expr COMMA expr COMMA expr PAR_CLOSE      #ccofExpr
       | POF PAR_OPEN expr PAR_CLOSE            #pofExpr
       | CPOF PAR_OPEN expr PAR_CLOSE           #cpofExpr
       | PBEND PAR_OPEN expr PAR_CLOSE           #pbendExpr

       //End nodes
```

```
        | stream_id         #sidExpr
        | NUM_LITERAL         #numExpr
        ;


var_name: IDENTIFIER (BRAC_OPEN expr BRAC_CLOSE)?;
stream_id: IDENTIFIER (BRAC_OPEN idx=expr BRAC_CLOSE)? (INS ins=expr)?;
data_type: fixedSizeType | varSizeType;

fixedSizeType: MIDI_TYPE; //TODO FIX
varSizeType: SIGNED_TYPE | UNSIGNED_TYPE | FLOATING_TYPE;

pulse_freq: MAX | CONST | IDENTIFIER;

USE: 'use';
CBLOCK: 'cblock';
PULSE: 'pulse';
IF: 'if';
ELSEIF: 'elseif';
ELSE: 'else';
FOR: 'for';
IN: 'in';
TO: 'to';
INC: 'inc';

CURLY_OPEN: '{';
CURLY_CLOSE: '}';

BRAC_OPEN: '[';
BRAC_CLOSE: ']';

PAR_OPEN: '(';
PAR_CLOSE: ')';

ARG_GROUP_SEP: ':';
COMMA: ',';

PULSE_SEP: '@';

EQUAL: '=';
ADD: '+';
SUB: '-';
MULT: '*';
DIV: '/';
MOD: '%';
SL: '<<';
SR: '>>';
```

```
RL: '<<<';
RR: '>>>';
AND:'and';
OR:'or';
XOR:'xor';
NOT:'not';
GAP: 'gap';
PROP: '&';
INS: '\'';

//MIDI OPs
NOTEOF: 'noteof';
FREQOF: 'freqof';
VELOF: 'velof';
PATOF: 'patof';
NNTOF: 'nntof';
CCOF: 'ccof';
POF: 'pof';
CPOF: 'cpof';
PBEND: 'pbend';

COMBOP: [+\-*/%] EQUAL; //Combined and equal

TERMINATOR: ';';
LOGIC_OP: ([<>]'='?) | ([!=] '=');

fragment SIGNED_TYPE_PREFIX: 'I';
fragment UNSIGNED_TYPE_PREFIX: 'U';
fragment FLOATING_TYPE_PREFIX: 'F';

MAX: 'max';
CONST:'const';

MIDI_TYPE: 'midi';
SIGNED_TYPE: SIGNED_TYPE_PREFIX (DEC_LITERAL|IDENTIFIER);
UNSIGNED_TYPE: UNSIGNED_TYPE_PREFIX (DEC_LITERAL|IDENTIFIER);
FLOATING_TYPE: FLOATING_TYPE_PREFIX (DEC_INT|IDENTIFIER);

NUM_LITERAL: OCT_LITERAL|DEC_LITERAL|HEX_LITERAL;

OCT_LITERAL: OCT_INT|OCT_FRAC;
DEC_LITERAL: DEC_INT|DEC_FRAC;
HEX_LITERAL: HEX_INT|HEX_FRAC;

PULSE_LITERAL: NUM_LITERAL PULSE_LITERAL_SUFFIX;
fragment PULSE_LITERAL_SUFFIX: SECOND | MILLISECOND | HERTZ | KILOHERTZ;
fragment SECOND: 's';
fragment MILLISECOND: 'ms';
fragment HERTZ: 'Hz';
fragment KILOHERTZ: 'kHz';

fragment OCT_PREFIX: '0o';
fragment DEC_PREFIX: '0d';
fragment HEX_PREFIX: '0x' | '0h';
fragment FRAC_SEPARATOR: '.';

fragment OCT_DIGIT: [0-7];
fragment DEC_DIGIT: [8-9]|OCT_DIGIT;
fragment HEX_DIGIT: [A-Fa-f]|DEC_DIGIT;

fragment OCT_INT: OCT_PREFIX OCT_DIGIT+;
fragment DEC_INT: DEC_PREFIX? DEC_DIGIT+;
fragment HEX_INT: HEX_PREFIX HEX_DIGIT+;

fragment OCT_FRAC: OCT_PREFIX (OCT_DIGIT+ FRAC_SEPARATOR OCT_DIGIT* |
   OCT_DIGIT* FRAC_SEPARATOR OCT_DIGIT+);
fragment DEC_FRAC: DEC_PREFIX? (DEC_DIGIT+ FRAC_SEPARATOR DEC_DIGIT* |
   DEC_DIGIT* FRAC_SEPARATOR DEC_DIGIT+);
fragment HEX_FRAC: HEX_PREFIX (HEX_DIGIT+ FRAC_SEPARATOR HEX_DIGIT* |
   HEX_DIGIT* FRAC_SEPARATOR HEX_DIGIT+);

IDENTIFIER: [a-zA-Z_][a-zA-Z0-9_]*;
STRING_LITERAL: '"' .*? '"';

BLOCK_COMMENT: '/*' .*? '*/' -> skip;
LINE_COMMENT: '//' ~[\r\n]* -> skip;
WS: [ \t\n\r] -> skip;
```

Bibliography

[1] Clive Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows.* Newnes, USA, 1st edition, 2004. v, 2, 5, 6, 40, 88

[2] Hal Chamberlin. *Musical Applications of Microprocessors.* Hayden Books, USA, 2nd edition, 1985. vi, 1, 7, 103, 104, 108

[3] The MIDI Manufacturers Association (Los Angeles, California). *The Complete MIDI 1.0 Detailed Specification: Incorporating All Recommended Practices.* MIDI Manufacturers Association, 3rd edition, 2014. vii, 7, 8, 9, 21, 33, 111

[4] Wikipedia. Digital Audio Workstation. `https://en.wikipedia.org/wiki/Digital_audio_workstation`. Accessed: 23-12-2020. 1

[5] Roger B. Dannenberg. Languages for computer music. *Frontiers in Digital Humanities*, 5:26, 2018. 1, 4, 10, 11, 13, 14

[6] A. Silberschatz, G. Gagne, and P.B. Galvin. *Operating System Concepts.* Wiley, 10th edition, 2018. 1

[7] Robert Jack, Tony Stockman, and Andrew McPherson. Effect of latency on performer interaction and subjective quality assessment of a digital musical instrument. In *AM'16: Proceedings of the Audio Mostly*, pages 116–123, Norrköping Sweden, October 2016. 2, 10

[8] P.J. Ashenden. *The Designer's Guide to VHDL.* Systems on Silicon. Morgan Kaufmann, third edition, 2008. 2

[9] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. *Parallel Programming for FPGAs.* 2018. 5

[10] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages 1–640, 2009. 6, 7, 97

[11] Terence Parr. *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf, 2nd edition, 2013. 7

[12] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) parsing: The power of dynamic analysis. *SIGPLAN Not.*, 49(10):579–598, October 2014. 7

[13] John Levine and Levine John. *Flex & Bison.* O'Reilly Media, Inc., 1st edition, 2009. 7

[14] The MIDI Manufacturers Association. *MIDI 2.0 Specification Overview.* MIDI Manufacturers Association, February 2020. 9

[15] Bruno Repp and Yi-Huang Su. Sensorimotor synchronization: A review of recent research (2006-2012). *Psychonomic Bulletin & Review*, 20:403–452, February 2013. 10

[16] Scott Wilson, David Cottle, and Nick Collins. *The SuperCollider Book.* The MIT Press, 2011. 10

[17] Bruno Ruviaro. *A Gentle Introduction to SuperCollider.* Faculty Book Gallery, 2nd edition, 2014. 10

[18] Miller Puckette. Pure Data: another integrated computer music environment. In *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41, Tachikawa, Japan, January 1996. 11

[19] Johannes Kreidler. *Loadbang: Programming Electronic Music in Pd.* Wolke Publishing House, 2009. 11

[20] Kent Jolly. Usage of Pd in Spore and Darkspore. In *Pure Data Convention*, Weimar - Berlin, Germany, August 2011. 13

[21] A. Kapur, P.R. Cook, S. Salazar, and G. Wang. *Programming for Musicians and Digital Artists: Creating music with ChucK.* Manning Publications, 2015. 13

[22] ChucK Team. Chuck : Strongly-timed, concurrent, and on-the-fly music programming language website. 14

[23] Yann Orlarey, Dominique Fober, and Stephane Letz. Syntactical and semantical aspects of Faust. *Soft Computing*, 8:623–632, September 2004. 14, 15

[24] Julius Orion Smith III. Audio Signal Processing in Faust. Technical report, Center for Computer Research in Music and Acoustics, Stanford University, 2020. 15

[25] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 30(11):1203–1233, 2000. 44

[26] David W. Bishop. *VHDL-2008 Support Library.* 97

[27] David W. Bishop. *Fixed Point Package User's Guide.* 97

[28] Julius Orion Smith III. *Physical Audio Signal Processing: For Virtual Musical Instruments and Audio Effects.* W3K Publishing, 2010. 98

[29] John F. Wakerly. *Digital Design: Principles and Practices.* Prentice Hall, 4th edition, 2005. 99

[30] E. T. Whittaker. XVIII.—On the Functions which are represented by the Expansions of the Interpolation-Theory. *Proceedings of the Royal Society of Edinburgh*, 35:181–194, 1915. 103

[31] Petter Källström. *A Fairly Small VHDL Guide.* 107, 108

[32] Hugo Fastl and Eberhard Zwicker. *Psychoacoustics: Facts and Models.* Springer-Verlag, Berlin, Heidelberg, 2006. 113

[33] A.V. Oppenheim, R. Schafer, and R.W. Schafer. *Discrete-time Signal Processing.* Prentice-Hall signal processing series. Pearson, 2010. 119