Proceedings of the 2nd International Faust Conference (IFC-20), Maison des Sciences de l'Homme Paris Nord, France, December 1-2, 2020

CREATIVE USE OF BIT-STREAM DSP IN FAUST

Till Bovermann *

Universität für Angewandte Kunst Wien
Project Rotting Sounds
Vienna, Austria
till@rottingsounds.org

Dario Sanfilippo

Independent researcher
sanfilippo.dario@gmail.com

ABSTRACT

Although digital data are adorned by the myth of lossless transmission and migration, everyday experience does prove the existence of degradation and, ultimately, data loss in various forms. As it turns out, 1-bit-based information representation for audio is of particular interest in terms of digital deterioration research. We therefore introduce BITDSP as a set of FAUST library functions to explore and research artistic possibilities of bit-based algorithms with FAUST. After introducing and discussing three data formats to handle 1-bit data streams, concrete implementations of bit-based functions ranging from simple bit operations over classic delta-sigma modulations to more experimental approaches like cellular automata, recursive Boolean networks, and linear feedback shift registers are introduced. In a third part, creative applications utilising the described library are presented.

1. INTRODUCTION

Although a widespread assumption is that forms digital representation are absolute and perfect in both their storage and reproduction, everyday observation shows that degradation and decay does indeed happen on all of the many levels of digital data handling. In the artistic research project "Rotting sounds – Embracing the temporal deterioration of digital audio", we therefore explore artistic opportunities arising from obsolescence, degradation and information loss in digitally represented sound [1]. Compared to the gradual and graceful degradation and eventual disintegration of analogue sound generation and reproduction appliances, common digital tools such as PCM or MP3-based representations tend to exhibit an abrupt breach into fail and thus silence or noise. 1-bit-based information representations for audio are an exception to this and are thus of particular interest in terms of our digital deterioration research: In difference to pulse-code modulation (PCM) streams in which a bit can have one of several meanings (e.g. in an integer-based PCM representation it can denote a $1, 2, 4, 8, 16, \dots$), only determined by its position in the serial bitstream, a bit in a 1-bit stream always carries one and the same kind of information. In delta-sigma modulation (see below), this would be e.g. whether the signal increases (true) or decreases (false) at that specific point in time. In this case, deterioration in the time domain, i.e. as jitter (e.g. by an off-by-one error in interpretation of the bit, a common error in signal reconstruction) does not have any considerable effect on the resulting perceived sound, and bit rot, the sporadic misinterpretation of the value of a bit (also referred to as *bit-flip*), alters signal information in a linear fashion, independent to the actual position of the misinterpreted bit in the stream.

* This research has been funded by the Austrian Science Fund (FWF) as project AR 445-G24.

1.1. Bit-stream DSP

In BITDSP, we differentiate between basic bit-based operators with no context other than the previous state, delta-sigma modulation (DSM), and approaches that generate, respectively operate on parallel streams of bits. Examples for such operators are for example shift registers or certain kinds of cellular automata (CA). As in other DSP environments, bit-stream operators (independent of the previous differentiation) may roughly be categorised into *generators*, *filters* and *converters*. The latter are operators that are intended to be used to translate semantic information (e.g. that denoting a waveform) from one representation (e.g. PCM) to another (e.g. 1-bit audio).

Delta-sigma modulation (DSM) is an analogue-to-digital encoding technique in the 1-bit domain with a non-linear quantisation noise distribution. The elementary DSM design consists of integrating the quantisation error —the difference between the output of the modulator and the input signal— and subsequently processing the result through a comparator to output a maximum or minimum value [2, 3].

The negative feedback loop in DSM acts as a noise-shaping transfer function that pushes the quantisation noise towards the upper region of the spectrum. The order of DSM systems is determined by the number of integrators in the network, and the transfer function, with regard to the order K of a modulator, is given by $(1 - z^{-1})^K$ [4]. Hence, oversampling becomes very effective in order to move the quantisation noise away from the audible range, although high oversampling ratios are required for low-order DSM designs for adequate signal-to-noise ratios. On the other hand, first-order and second-order DSM guarantee stability for a wide range of input signals, whereas high-order DSM may result in unstable behaviours. Digital low-pass filters can remove the quantisation noise from the ultrasonic range, and downsampling can then be performed to operate at standard sampling frequencies [4].

Digital DSM can be deployed to convert multi-bit signals into 1-bit ones. For the inverse, the conversion between one-bit streams to multi-bit ones is simply carried out by averaging the pulses in the one-bit stream, which can be achieved effectively through low-pass filtering. Specifically, the cut-off frequency of the low-pass filter determines the range of the baseband, whereas the bit resolution of the coefficients of the filter determines the bit-depth of the resulting multi-bit signal [2].

The conversion to the one-bit domain opens to new possibilities for audio DSP. DSM audio signals can be combined or processed through one-bit algorithms for generating complex patterns both at timbral and formal time scales. Some of these algorithms include CA and recursive boolean networks (RBN), examples of which will be discussed later.

2. FORMS OF BIT-BASED SYNTHESIS TECHNIQUES AND THEIR IMPLEMENTATION

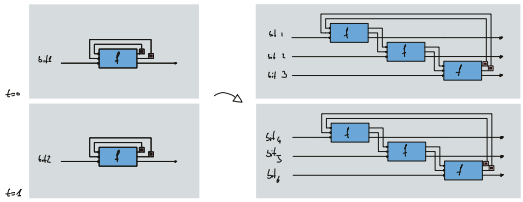

Figure 1: Unfolding a dsp function with two feedback paths for oversampling (here $N = 3$) by parallel processing.

During our explorations of bit-stream synthesis, we collected various functions and ideas related to bit-based synthesis techniques. We decided to develop and implement them in FAUST because of its stream-oriented functional approach [5, 6]. All currently implemented functions are available for public use under the MIT License in a repository[1] of FAUST library files. We subsequently present selected functions and, where applicable, explain implementation details. Before diving into the functionality itself, though, we introduce three data structures: integer-based, `bitBus<N>`, and `int32`, that we deemed interesting for 1-bit sound synthesis and that we therefore used in our implementations. While the first technique is a straight-forward representation based on standard FAUST operators and signal flow, the latter incorporate oversampling and bit-wise operation techniques which require redefinition and unfolding of feedback paths. Overcoming this challenge, however, results in a more optimised implementation that allows for most of the time necessary oversampling and compact representation of data structures.

integer-based A straight-forward implementation in which each 1-bit sample is represented as an integer value that is either $a \in \{0, 1\}$ (unipolar), or $a \in \{-1, 1\}$ (bipolar). This representation allows to implement bit-based operators utilising standard FAUST operators. For real-time applications, sample rates are, however, limited to the maximum sample-rate supported by the audio interface. Another limiting factor is that each 1-bit value is represented by (typically) 32 bits, hence computational and memory allocation is far bigger than necessary.

`bitBus<N>` (parallel processing of parallel bit-streams) The bit-stream is represented by an N-dimensional signal bus of N consecutive samples of the bit-stream in N parallel streams (see Figure 1). This results in an oversampling relative to the sample-rate by a factor of N. This approach requires unfolding of feedback paths over the N signal buses, which means that standard implementation of the FAUST libraries utilising feedback operators cannot be integrated. Instead, specific versions need to be written. The up-sampling factor is denoted by `N` in `bitBus<N>`.

`int32` (parallel processing of a compact bit representation) The bit-stream is encoded in a FAUST-native integer signal as a sequence of bits that need to be processed in parallel. As of writing this paper, an integer in FAUST typically translates into a 32bit signed integer. Implementations in `int32` are therefore equivalent to `bitBus<32>`. The resulting up-sampling factor is therefore $N = 32$).

Generally, an implementation for each of the three data structures follows a similar form. As an example for concrete implementations, we provide a block diagram for an implementation strategy for oversampling by parallel processing in Figure 1.

2.1. Conversion

2.1.1. Delta-sigma modulators

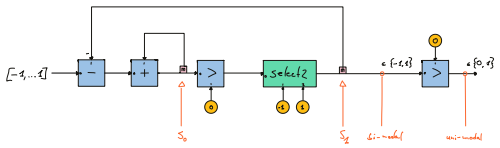

Figure 2: Block diagram of the 1st order Delta-sigma modulator as implemented for the straight-forward integer-based data format. s_0 and s_1 denote the feedback paths that need unfolding for an implementation in `bitBus<N>` resp. `int32` (cf. Figure 4).

A pre-requisite for many 1-bit-based synthesizers and filters are functions to convert pulse-code modulation (PCM), the common digital audio representation format [7] also used by FAUST, into a 1-bit form. For the described library, we chose implementations of the naive first-order delta-sigma modulator based on two feedback paths (`dsm1`, see Figure 2) and of another feedback-based modulator, this one a second-order model (`dsm2`, see Figure 3). The first-order delta-sigma modulator is so far implemented in the straight-forward integer-based data format and in `bitBus<N>` which can be converted into the more specific form of `int32` (see Figure 4).

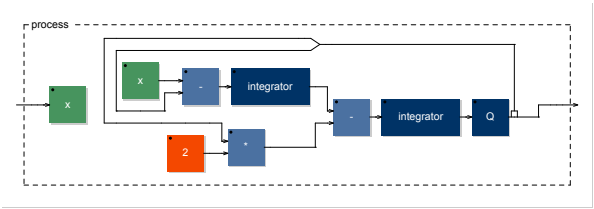

Figure 3: Block diagram of the 2nd order Delta-sigma modulator as implemented for the straight-forward integer-based data format..

2.1.2. BitDAC

For parallel bit-streams we found it interesting to be able to interpret (part of) them as integer-based PCM values. We therefore implemented `bitDAC`. It allows to select a range of parallel bits in a bit-stream and interpret them as part of an integer PCM value. Currently there is an implementation for the `int32` format:[2]

[1] https://github.com/rottingsounds/bitDSP-faust, accessed on 31.8.2020.

[2] Since the native integer data-type in FAUST is signed and right-shift operations on signed integers are dependant on used C(++)-compiler, a `ffunction` primitive had to be implemented that casts the integer into `unsigned int` before the right-shift operation. Left shift was defined similarly to assure consistent readability.

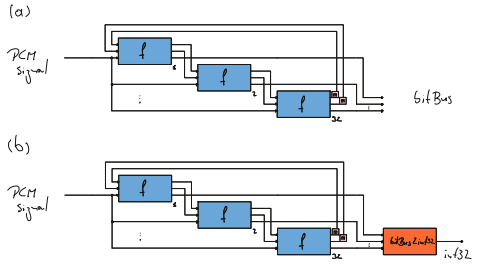

Figure 4: Block diagram of the 1st order dsm for (a) `bitBus<32>` and (b) `int32`.

```
bitDAC(offset, size, in) = normed(
 // shift selection to right-most bits
  right_shift((in & bitmask), offset)
) with {
  // maximum value that fits into size
  maxval = left_shift(1,size)-1;
  // select area of which to create PCM values
  bitmask = left_shift(maxval, offset);
  // normalise values to be between 0 and 1
  normed(out) = out / (maxval-1);
};
```

As with most FAUST definitions, `bitDAC`'s parameters can be modulated at audio-rate which can result in interesting sonic behaviour. For now, we used `bitDAC` mostly for sonifications of the below-described LFSR implementations.

2.1.3. Conversion between 1-bit data formats

For now, conversion from naive integer-based format to the other formats is done via sample-and-hold. This means for the conversion to `bitBus<N>` to set all parallel buses at a given time to the same value as the input state. The conversion to `int32` is defined as:

```
bit_to_int32(0)  = 0; // unimodal
bit_to_int32(-1) = 0; // bimodal
// -1 is encoded by all 32 bits
//    in high state
bit_to_int32(1) = -1;
```

For future investigations, dithered upsampling routines are planned to be implemented.

Similarly naive implementations were defined for conversion between `bitBus<(>N)` and `int32`:

```
bitBus_to_int32 = bitBus2int(32);
bitBus_to_int(N) =
  si.bus(N) : sum(i, N, _ << i);
```

```
int32_to_bitBus(N) = _ <: si.bus(N) :
  par(i, N, ((1 << i) & _) != 0);
```

2.2. Operators

We differentiate between basic bit-wise operators, series operators, parallel operators and a combination of the latter.

2.2.1. Bit-wise operators

Since FAUST provides primitives for the bit-wise operations `and`, `or`, and `xor`, we used those to implement bit-wise operators for all three data structures. For `int32`, bitwise `not` needed to be defined using the `ffunction` directive since FAUST does not currently provide a primitive for it.[3]

2.2.2. Bit rot and jitter

Bit-related degradation of the bit-stream is implemented by `bitRot` which, under a specified likelihood and depending on its `type` parameter, either sets a bit's value to false (`-1`), true (`1`), or reverses it (`0`).

```
bitrot(noise, chance, type) = _ <: select3(
  type+1,
  low(noise, chance, _),
  flip(noise, chance, _),
  high(noise, chance, _)
) with {
    low(noise, chance, x) =
      select2(coin(noise, chance), x, 0);
    high(noise, chance, x) =
      select2(coin(noise, chance), x, 1);
    flip(noise, chance, x) =
      select2(coin(noise, chance), x, 1-x);
    coin(noise, chance) =
      noise < chance;
};
```

An essential operation for delta-sigma-based filters is a delay. While the implementation for the simple data format and `bitBus<N>` are straight-forward, the implementation for `int32` required more attention:

```
delay(delta, x) = delay32(
  (delta % 32), x@( (delta, 32) : div )
) with {
  delay32(0, x) = x;
  delay32(delta, x) = (
    right_shift(x, delta) |
    left_shift(x', (32-delta))
  );
};
```

2.2.3. Recursive Boolean networks

Recursive Boolean Networks (RBNs) are systems inspired by Kauffman's models of gene regulatory systems, which he implemented as networks of nodes representing genes and Boolean operators applied to vertices representing rules of regulatory interactions [8]. Due to the non-linear properties of some Boolean functions, Boolean networks with recursive configurations can be used for the generation of streams with chaotic behaviours ranging from attractors to periodic oscillations and unpredictable behaviours [9]. In this project, RBNs are mainly used as complex oscillators where frequency content and dynamical behaviours are entangled and driven by the periods of the feedback loops. Below, we can see two examples of oscillators that transition through different dynamical behaviours upon variation of the feedback periods in the networks. The first example is a second-order network, containing two nodes, whereas the second example is a fourth-order network, with four

[3] This and other FAUST-related limitations unfortunately mean that some functions of the BITDSP library are currently only available in the C-based interface of FAUST.

nodes. The first network has a fully-connected topology, whereas the second one is a combination of identity and circular topologies. An important aspect that deserves consideration is the asymmetry in the behaviour of such oscillators in a time-variant configuration. Specifically, any identical configuration of the parameters, provided that these vary over time, results in different behaviours at different times as the system is determined by both its output and state variable [10, 11].

```
bool_osc1(del1, del2) = node2
letrec {
  'node1 =
    not(node1 xor node2 & node1)
    @ max(0, min(ma.SR, del1));
  'node2 =
    not(node2 xor node1 xor node2)
    @ max(0, min(ma.SR, del2));
} with {
  not(x) = rint(1 - x);
};
```

```
bool_osc0(del1, del2, del3, del4) = node1
      letrec {
           'node1 = not(node1 & node2) @ max(0,
               min(ma.SR, del1));
           'node2 = not(node2 | node3) @ max(0,
               min(ma.SR, del2));
           'node3 = not(node3 xor node4) @ max(0,
                min(ma.SR, del3));
           'node4 = not(node4 & node1) @ max(0,
               min(ma.SR, del4));
      }
      with {
           not(x) = rint(1 - x);
      };
```

2.2.4. Masked block-based operators

Block-based operators assume a certain amount of surrounding bits to be their context of operation. This makes sense especially in `bitBus<N>` and `int32` form, since here, `N` (32) bits are processed in parallel and are thus available for the operator. One sonically and functionally interesting operator to implement is applying a bit-wise operation only to specific bits within the block (e.g. via a bit-mask). As an reference for other bit-wise operators, the following is an implementation for a masked `or` in the `int32` form.

```
maskedOr(mask, a, b) = applyMask(a|b, a, mask)
with {
  applyMask(res, org, mask) =
    (mask & res) |
    (bitNot(mask) & orig);
};
```

A bit mask in `int32` form can be created by providing the indices of bits to be set to the following function:

```
bit_mask((n, ns)) =
  bit_mask(ns) | left_shift(1, n);
bit_mask(n) = left_shift(1, n);
```

Defining `b = bit_mask((1, 3));` will then create the integer 10 with bits 1 and 3 set.

2.2.5. Linear feedback shift registers

Linear feedback shift registers (LFSRs) are in widespread use for hardware-based indexing, and counting applications in which speed and efficiency are important and the order of execution or indexing can be neglected. Possibly because of the easy availability of shift-registers as integrated circuits and the well-understood theory behind LFSRs, several sound-synthesis-related hardware implementations of feedback- and shift-register-based filters, sequencers, and oscillators have been developed, most prominently in the Rungler circuit by Rob Hordijk and Grant Richter's "noise ring"[12].

LFSRs can be described in the so-called "Fibonacci form"[4], i.e. by a binary polynome that determines which bit indices (here called *taps*) of the register are combined to determine the new input bit:

$$\sum_{i}^{N} c_i * x^i$$

with N the size of the register, $c = \{c_i | i = 1..N, c_i \in \{0, 1\}\}$ the mask determining which taps have an influence on the register output, x^i the i-th tap, and $\sum$ the parity (iterative XOR) operator.

Depending on the selection of c, the output sequence of a shift register of size N can have a period of up to $2^N - 1$ [13].

The following code implements an LFSR on a 32bit-wide shift register.

```
lfsr32(mask, in) = step(mask, in) ~ _
with {
  step(mask, in, fbIn) =
    selector(in, fbIn) <:
    (parity(_ & mask), left_shift(_, 1)) : |;
  selector(a, b) =
    select2(changed(a), b, a);
  changed(x) = x != x';
};
```

Since periodicity of LFSRs depend a lot on the length of the underlying shift register [14], we also implemented a variable-length LFSR with a register length of up to 32 bits.

```
lfsr(n, mask, in) = step(n, mask, in) ~ _
with {
  masked_out(n, val) =
    right_shift(-1, 32-n) & val;
  step(n, mask, in, fbIn) =
    selector(in, fbIn) <: (
      parity(_ & mask),
      masked_out(n, left_shift(_, 1))
    ) : |;
  selector(a, b) =
    select2(changed(a), b, a);
  changed(x) = x != x';
};
```

To optimise performance, we utilised a parity implementation by Anderson implemented as a `ffunction` primitive [15].

2.2.6. Bit-based 1-dimensional cellular automata

Complex networks can be analysed based on static structural properties such as clustering, hubs, and degree distribution to find implications between these properties and how we can explore and investigate such networks. One aspect that is crucial in understanding complex networks are their dynamical behaviour and how they perform tasks. Melanie Mitchell extends the notion of network dynamics to include information processing performed by networks, and considers one main challenge to understand how such information is processed and propagated [16].

[4] There are also other forms of description but within this paper we limit ourselves to "Fibonacci".

For information propagation to be modelled and understood, it is necessary to consider how nodes and links behave locally in the processing and propagation of information, and how nodes, links, and network structures change over time in response to these behaviours. Complex networks such as the brain and the immune system are e.g. characterised by mainly having local connections and centralised control. Such networks have been described according to information processing notions, although the understanding of how such networks perform complex computations still remains limited [16].

Regular networks such as cellular arrays have been proposed as architectures for molecular computation and distributed information processing [17]. Cellular automata are one of the simplest forms of regular networks and have been investigated thoroughly by Wolfram in the wider exploration of complex systems [18].

The type of cellular automata deployed in our artistic project are 2-state (*binary*), 1-dimensional elementary cellular automata based on circular lattices. The state of each cell is determined by its state and the state of its two neighbouring cells. The output of the lattice is iteratively fed back into itself to determine the new states.

Since we have three cells and two states, there are $2^3 = 8$ cases to which a specific rule is applied to determine the outcome. Since there are eight cases and two possible outcomes for each case, there are a total of $2^8 = 256$ possible rules that determine the behaviour of an elementary CA. These rules can be encoded as positive integers between 0 and 255, called *Wolfram Number*s. The number is converted into a binary string of eight digits –appropriately zero-padded or clipped if necessary– representing the outcome for the eight possible cases.

Below we can see the FAUST code for the implementation of an elementary CA (see Appendix A.1). The parameters `L`, `R`, `I`, and `rate`, respectively, are the length, rule, initial condition, and iteration rate of the CA.

3. APPLICATION

As mentioned in Section 1, the motivation behind the BITDSP library is to artistically research properties of rotting and decay on bit-based audio representations. Applications of the presented features are therefore primarily involved with the aesthetics of rotting in terms of bit deterioration through operations otherwise deemed inadequate for such material. Since we are still at the start of the investigations, only one example is described below, however, we are planning to extent the research and are also considering the library to be the basis for upcoming workshops on digital rotting.

3.1. "Merge and dissolve"

For the "SOUND CAMPUS" of Kunstuniversität Linz at Ars Electronica Festival 2020, Till Bovermann performed with tools based on the BITDSP library together with Thomas Grill (sound), and Kathrin Hunze (visuals). The performance entitled "merge and dissolve" took place in September 2020 at a venue in Linz and was transmitted into the Metaverse, a virtual multi-client environment equipped with visual projection possibilities as well as a virtual multi-speaker setup. Metaverse was developed by the host organisation specifically for "SOUND CAMPUS".[5]

[5]https://ausstellungen.ufg.at/wildstate/project/sound-campus-merge-and-dissolve/, accessed on 2.9.2020.

As mentioned in Section 1, the working group rotting sounds, under which the performance took place, has been investigating 1-bit audio as a medium in which deterioration may take place. In this performance we focused on the aspect of this format that the electrical bit-stream can simply be played over a normal amplifier/loudspeaker combination yielding the analog sound equivalent of the sonic information embedded in the bit-stream. Both PCM-based audio processing strategies as well as strategies introduced above were integrated into a signal-network between the performers, giving rise to rather unexpected sound qualities.

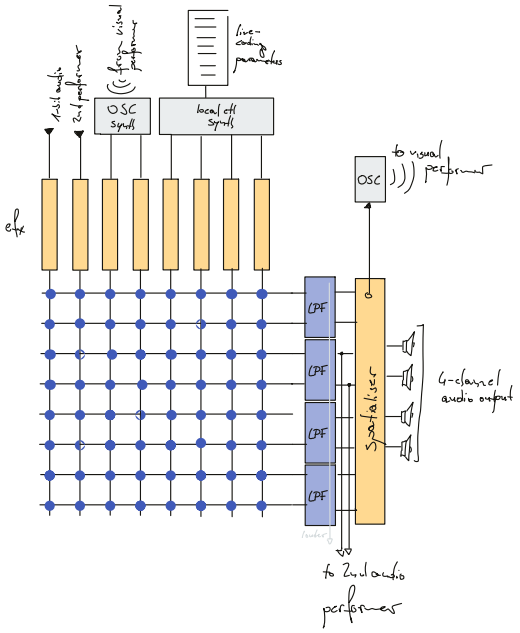

Figure 5: Overview of Till Bovermann's performance system for "merge and dissolve".

While the overall setup of the performance was highly networked and tools of the individual performers were independently developed in various forms of hardware and software[6], Till's performance system relied on a SuperCollider[7]-based performance setup that utilised custom-built UGens written in FAUST and the BITDSP library. Several incoming PCM streams and bit-streams were to be converted respectively processed. As a conceptual (and possibly also practical) decision, we agreed to play at a sample rate of 192kHz, allowing to use standard audio hard- and software and residing in the somewhat fuzzy inter-mediate between common PCM-based sample rates and "proper" DSD sample rates. Interaction with the system was handled via a monome *grid 128* controller.[8] While the lower 8 by 8 matrix of buttons of the controller was used as a switchboard to route the eight incoming signals to the respective outputs, the upper 8 by 8 matrix of buttoned served as switches to (de-)activate signal effectors. A schematic overview of the performance system is given in Figure 5. Since the performance was based around the idea to pick up each other's streams, manipulate and amplify them into an 8-channel system, the cus-

[6]more details on the performance can be found at http://rottingsounds.org.

[7]http://supercollider.github.io, accessed on 31.8.2020.

[8]https://monome.org/docs/grid/, accessed on 31.8.2020.

tom UGens based on BITDSP were kept rather simplistic in their implementation: The modularity of the system and thus the combination of modules rather than complex effecting was meant to induce sonically interesting results. Parts of the performance setup are published in a git repository.[9]

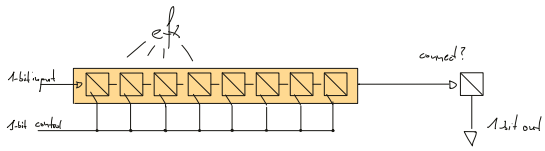

Figure 6: Effect bus of one channel in Till Bovermann's performance system for "merge and dissolve".

4. CONCLUSION

We presented our work and thoughts on the creative use of bitstream DSP in FAUST. Although we are still at the beginning of our research, we are happy to say that FAUST and its community turned out to be a welcoming environment for implementation of and discussion about a broad spectrum of bit-based operations. Unfortunately, however, since implementations of many of the described functions depend highly on a specific bit-representation of integers (in our case the unsigned, fixed width integer type `uint_32t`), the lack of such an unsigned integer type and, generally, the lack of data types with a defined bit-length[10] renders some implementations in native FAUST code impossible. In crucial situations we had to rely on the use of the `ffunction` primitive to include external C headers. We would be very grateful to have standard types according to the C++11 standard to be considered to be included in future releases of FAUST.

Apart from a growing repertoire of applications and performances, workshops around rotting sounds with BITDSP are scheduled and an installation involving CAs and dynamic oscillators is planned, we are looking forward to include more bit-stream-related functionality into the library. We especially plan to implement respectively port further applications and variations of LFSR [19], e.g. the feedback-with-carry shift register [20]. Higher-order DSM implementations for `bitBus<N>` and `int32` are also planned as well as implementations of functionality spanning the different data formats.

Eventually, we intend to offer a library that facilitates bitstream DSP for use in FAUST development with a particular focus on distortion and deterioration processes. The starting points implemented so far are performant and powerful due to core functionality provided through the FAUST language, toolchain and, most importantly its growing community. We are therefore grateful for feedback on the existing implementations and on possible further directions.

[9] https://github.com/rottingsounds/mergeDissolve/, accessed on 31.8.2020.

[10] As of writing this paper, FAUST renders its output code with compiler-specific types `int`, `float`, `double` instead of C++11 conform types.

5. REFERENCES

[1] T. Grill, T. Bovermann, and A. Schilling, "Embracing the temporal deterioration of digital audio: A manifesto," in *Proceedings of Politics of the machine*, 2018, submitted.

[2] R. Schreier and G. C. et al. Temes, *Understanding delta-sigma data converters*, vol. 74, IEEE press Piscataway, NJ, 2005.

[3] J. D. Reiss, "Understanding sigma-delta modulation: the solved and unsolved issues," *Journal of the Audio Engineering Society*, vol. 56, no. 1/2, pp. 49–64, 2008.

[4] U. Zölzer, *Digital audio signal processing*, vol. 9, Wiley Online Library, 2008.

[5] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of faust," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[6] Y. Orlarey, D. Fober, and S. Letz, "Faust: an efficient functional approach to dsp programming," *NEW COMPUTATIONAL PARADIGMS FOR COMPUTER MUSIC*, vol. Editions DELATOUR FRANCE, pp. pp.65–96, 2009.

[7] K. W. Cattermole, "Pulse code modulation: invented for microwaves, used everywhere," in *Proceedings of the 1995 International Conference on 100 Years of Radio*, 1995, pp. 184–186.

[8] Stuart A Kauffman, "Metabolic stability and epigenesis in randomly constructed genetic nets," *Journal of theoretical biology*, vol. 22, no. 3, pp. 437–467, 1969.

[9] James Gleick, *Chaos: Making a new science*, Open Road Media, 2011.

[10] Dario Sanfilippo and Andrea Valle, "Feedback systems: An analytical framework," *Computer Music Journal*, vol. 37, no. 2, pp. 12–27, 2013.

[11] Dario Sanfilippo, *Complex musical behaviours via time-variant audio feedback networks and distributed adaptation: a study of autopoietic infrastructures for real-time performance systems*, Ph.D. thesis, University of Edinburgh, 2020.

[12] "Noisering analysis by babaluma," `http://mamonu.weebly.com/wiard-noisering.html`, accessed 31.8.2020.

[13] A. Klein, *Linear Feedback Shift Registers*, pp. 17–58, Springer London, London, 2013.

[14] M. George and P. Alfke, "Linear feedback shift registers in virtex devices," *Xilinx appreciation note XAPP210*, 2007.

[15] S. E. Anderson, "bithacks," 2002, `https://graphics.stanford.edu/~seander/bithacks.html#ParityParallel`, accessed 31.08.2020.

[16] M. Mitchell, "Complex systems: Network thinking," *Artificial Intelligence*, vol. 170, no. 18, pp. 1194–1212, 2006.

[17] S. Xiao, F. Liu, A. E. Rosen, J. F. Hainfeld, N. C. Seeman, K. Musier-Forsyth, and R. A. Kiehl, "Selfassembly of metallic nanoparticle arrays by dna scaffolding," *Journal of Nanoparticle Research*, vol. 4, no. 4, pp. 313–317, 2002.

[18] S. Wolfram, *A new kind of science*, vol. 5, Wolfram media Champaign, IL, 2002.

[19] T. Clutterbuck, T. Mudd, and D. Sanfilippo, "A practical and theoretical introduction to chaotic musical systems," .

[20] M. Goresky and A. M. Klapper, "Fibonacci and galois representations of feedback-with-carry shift registers," *IEEE Transactions on Information Theory*, vol. 48, no. 11, pp. 2826–2836, 2002.

A. APPENDIX

A.1. Elementary cellular automata listing

The parameters `L`, `R`, `I`, and `rate`, respectively, are the length, rule, initial condition, and iteration rate of the CA.

```
eca(L, R, I, rate) = (si.bus(L) , init(I) :
  ro.interleave(L, 2) :
  par(i, L, +) : iterate :
  par(i, L, ba.sAndH(trigger))) ~ si.bus(L)
with {
  trigger =
    ba.period(ma.SR / max(ma.EPSILON, rate))
    == 0;
  wrap(M, N) = int(ma.frac(N / M) * M);
  w_num = zeropad_up(
    int(8 - ceil(ma.log2(R1))),
    bitConv.dec2bitBus(R1)
  ) with {
    R1 = min(255, R);
  };
  init(N) = zeropad_up(
    int(L - (floor(ma.log2(N1)) + 1)),
    bitConv.dec2bitBus(N1)
  ) : par(i, L, _ <: _ - mem)
  with {
    N1 = min(N, 2 ^ L - 1);
  };
  rule(x1, x2, x3) = ba.if(
    c1, w_num : route(8, 1, 1, 1), ba.if(
    c2, w_num : route(8, 1, 2, 1), ba.if(
    c3, w_num : route(8, 1, 3, 1), ba.if(
    c4, w_num : route(8, 1, 4, 1), ba.if(
    c5, w_num : route(8, 1, 5, 1), ba.if(
    c6, w_num : route(8, 1, 6, 1), ba.if(
    c7, w_num : route(8, 1, 7, 1),
    w_num : route(8, 1, 8, 1)
  ))))))
  ) with {
    c1 = (x1==1) & (x2==1) & (x3==1);
    c2 = (x1==1) & (x2==1) & (x3==0);
    c3 = (x1==1) & (x2==0) & (x3==1);
    c4 = (x1==1) & (x2==0) & (x3==0);
    c5 = (x1==0) & (x2==1) & (x3==1);
    c6 = (x1==0) & (x2==1) & (x3==0);
    c7 = (x1==0) & (x2==0) & (x3==1);
    c8 = (x1==0) & (x2==0) & (x3==0);
  };
  iterate = si.bus(L) <: par(
    i, L,
    route(L, 3,
      wrap(L, i - 1) + 1, 1, i + 1, 2,
      wrap(L, i + 1) + 1, 3
    ) : int(rule)
  );
};
```