Synthesizing Music with Logic Gate Networks

Ian Clester
ijc@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

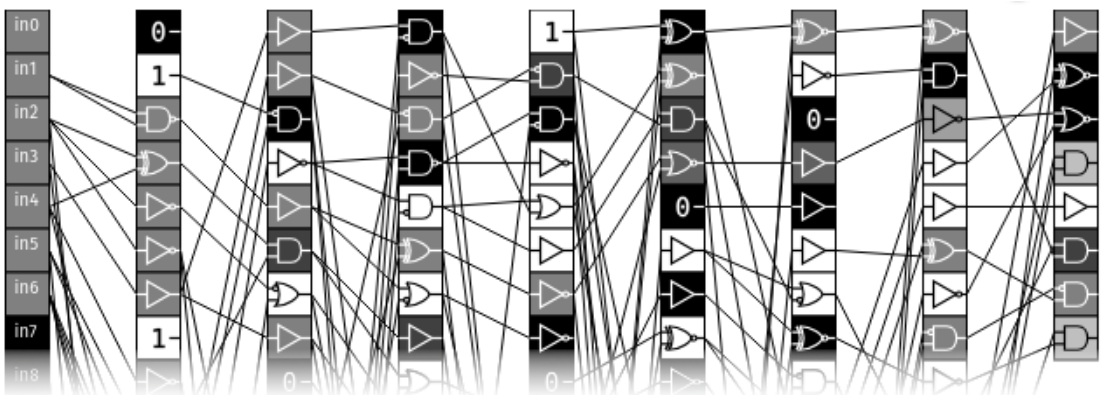

Figure 1: Visualization of a logic gate network generating audio samples

Abstract

Small digital circuits consisting of basic logic gates (AND, XOR, etc.) are capable of generating surprisingly complex musical output. In this paper, I present physical and web-based interfaces for exploring the space of audio-generating logic gate networks and 'bending' such networks via touch (or mouse) gestures to interfere with their operation and change their output while they are running. This work follows in the vein of bytebeat practices, in which music is generated by short code snippets at the level of individual audio samples, but takes things further by relying on an even lower-level form of computation. In addition to presenting the system, I offer some preliminary analysis of why these logic gate networks tend to produce musical output.

Keywords

computer music, logic gate synthesis, bytebeat, live coding, circuit bending

1 Introduction

Complex output can emerge from simple mechanisms. Here, I explore the musical potential of small logic gate networks that generate audio samples as a function of time. Despite the small size and low level of abstraction of these networks, which generate audio samples directly from basic digital building blocks, they are capable of generating surprisingly rich musical output, simultaneously determining all levels of musical organization from timbre up to form.

Of course, a sufficiently large logic gate network could encode any possible mapping between input time and output sample. (Any function over a finite set of inputs can be expressed as a truth table, and any truth table can be implemented by combinational logic.) However, I find that rather small networks (e.g. 100-200 gates) can produce surprisingly rich musical output. Examples of some small logic gate networks and their audio output are available in the supplementary materials (see `a_few_networks.mp4`).

In this paper, I present `glitchgate`,[1] an application for exploring the space of such networks and manipulating them through live interaction. I describe the design and implementation of `glitchgate`, which includes interfaces for running in the browser (via WebAssembly and Audio Worklets), including on mobile devices with touchscreens, and on the Bela platform in combination with a Trill Square. Additionally, I consider why these networks tend to produce musical output.

2 Related Work

A primary inspiration for this work is *bytebeat*. bytebeat is a musical practice which involves writing short code expressions (typically in C) that describe audio as a function of time, generating sound directly, sample-by-sample [8]. For example, the expression `((t>>10)&42)*t` generates audio which is distinctly melodic.

The bytebeat approach is rather unusual compared to the conventional computer music approach, which tends to feature a rigid separation of different layers (control vs. synthesis, score vs. orchestra). Bytebeat expressions, in contrast, simultaneously determine everything — timbre, rhythm, melody, structure — from the level of individual audio samples on up. This unity also characterizes the logic gate networks described in this work, with the difference that the networks operate at an even lower level of abstraction, as they are built on individual logic gates operating on individual bits rather than C's bitwise & arithmetic operators operating on 32-bit integers.

The aesthetic of these networks and this application relates to that of *glitch* [5]. Aside from the raw, low-fidelity, aliased digital sounds produced by these networks, the primary mode of interaction (described in §3) is based on interfering with an

This work is licensed under a Creative Commons Attribution 4.0 International License.
NIME '25, June 24–27, 2025, Canberra, Australia
© 2025 Copyright held by the owner/author(s).

[1] The source code for `glitchgate` is available at https://github.com/ijc8/glitchgate, and the web version is live at https://ijc8.me/glitchgate.

existing network, injecting points of failure to make it 'glitch' in different ways.

This work also relates to compression-oriented artistic practices, such as the aforementioned bytebeat, the demoscene [4] (which tends to focus on generating interesting audiovisual output with as little code as possible), sctweets [10] (SuperCollider compositions that fit in 140 characters or less), and ScoreCard (generative music programs that fit in a QR code) [6]. In this vein, I note aesthetic kinship to artworks such as Protoroom's "SmallBig_SØ",[2] which is centered on the 'big' space of possibilities offered by 'small' formulas (code expressions), and Tristan Perich's 1-Bit Symphony, which fits a 40-minute, five-movement symphony written in assembly language on a single microprocessor with just 8KB of RAM [11].

I also draw inspiration from works which make musical computation visible. Dave Griffith's Betablocker exposes the operations of an 8-bit musical machine to the audience as it executes [2]. Orca is a 2D programming language for music whose execution unfolds visibly over time [9]. Similarly, pieces featuring artificial life, such as Jack Armitage's work with Tölvera [1], tend to foreground the simulation itself. `glitchgate` likewise uses visuals to expose the operation of the network and suggest sites for interaction.

`glitchgate` allows for intervening in the network while it is executing. Thus, it relates to other practices involving the live manipulation of sonic machinery, such as circuit bending [7], live patching, and live coding. It also bears some resemblance to interactive machine learning techniques, especially those focused on exploring latent spaces within (neural) networks [3]. `glitchgate` bears some similarity to these in its interface, with the distinction that it supports exploring the space of nearby (logic gate) networks, rather than the latent space encoded by a larger (neural) network.

3 Design

`glitchgate` is an application for creating and executing audio-generating logic gate networks. Such networks consist of an input layer, several layers of logic gates,[3] and an output layer. The input layer consists of several bits encoding the time of the current audio sample, which is fed to the subsequent layer as a binary integer. Each of the middle layers consists of logic gates whose operands are outputs from the preceding layer. The output layer consists of several bits (from the last layer of gates) encoding the output sample as an unsigned integer. Note that the number of bits in the input layer determines the periodicity of the entire network, while the number of bits in the output layer determines the bit depth of the audio.

During execution, the current sample time is fed into the network to compute the corresponding audio sample. The sample time is incremented and the process repeats. Note that the sample time overflows to zero once every 2^n samples, where n is the number of bits in the input layer. Following conventions from bytebeat, the sampling rate is 8kHz, so the whole network is evaluated 8000 times a second.

The `glitchgate` interface displays the gates in each layer and the connections between them. The connections are hidden in some modes to make the interface more compact for ease of interaction. The interface also features real-time visualization of network operation, in which each gate is colored according to its average output during the previous block of audio samples.

The user can interact with the network in a few ways, spanning a range of granularity. On the fine-grained end, the network can be modified directly by editing a textual description of the network. This enables the user to modify individual gates or their operands, although in practice it is primarily useful as a way to save/load entire networks that sound interesting. On the coarse-grained end, the user can generate an entirely new network with the "Randomize" button, which preserves the shape of the network (the number and sizes of layers) but randomizes all the gates within the layers and the connections between them. "Randomize" is useful for quickly exploring the space of networks to find something interesting.

In between these extremes, the user can interact with the network while it is running by painting over it, masking gates by temporarily replacing them with `0`s. Masking a gate nullifies its output, affecting all gates in subsequent layers that depend on it. Conceptually, this is akin to shorting input pins to GND in a digital circuit, unplugging cables in a modular synthesis patch, or ablating neurons in a neural network. In all cases, signal is replaced with silence.

This network-painting interaction is the primary method of manipulation: beginning with a network that sounds interesting, one can then perform live 'network surgery', exploring the workings of the network via temporary ablations. The network can be restored to its original state at any point. As I discuss in §5.2, `glitchgate`'s visualization features can help guide the player in identifying interesting sites for intervention.

4 Implementation

`glitchgate` comprises two interfaces: one for the web and one for Bela. Both are built on the same small C core which runs the logic gate network. The C core serves as a network interpreter, iterating through the layers and dispatching the appropriate operation for each gate. I opted to interpret networks rather than compile them (by e.g. generating and compiling C specifically for the current network, with fixed logic operations) for the sake of responsiveness, particularly when painting over the network.

Both the web and Bela versions use this core to execute the network. For the web version, the core is compiled to WebAssembly and called from an audio worklet. In the Bela version, it is simply compiled along with the C++ as a part of the Bela project. Both versions support running the network, playing its output, randomizing the network, and painting over the network to mask the output of gates. In the web version, painting can be accomplished with mouse or touch screen, and on the Bela it can be done with the Trill Square.

Only the web version supports directly editing the network or visualizing its execution. However, the Bela version can be used in conjunction with the web version via WebSockets, in which case the Bela is used for control and synthesis, while the web interface is used for visualization.

5 Discussion

5.1 Musical Biases of Logic Gate Networks

I stumbled upon the musical potentials of small logic gate networks rather by accident in the course of a project focused on training difflogic [12] networks for musical applications. I soon

[2] https://protoroom.kr/works/smallbig/

[3] There are 16 supported logic gates, corresponding to the $2^{2^2} = 16$ possible truth tables of two inputs. Note that this set includes gates which take one input (NOT and the identity function), 'gates' which take no inputs (constant 0 and 1), and gates which are merely operand-flipped versions of others.

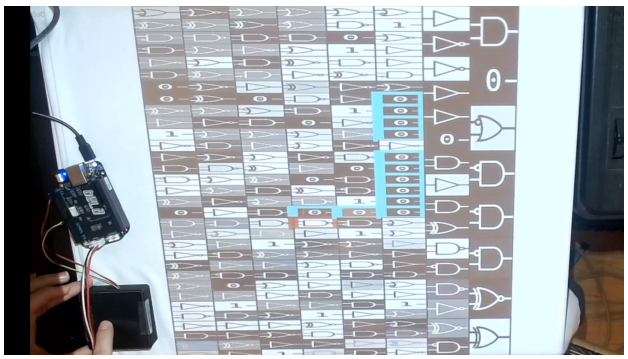

Figure 2: Intervening in a playing network on a Bela (with visualization projected from the web interface)

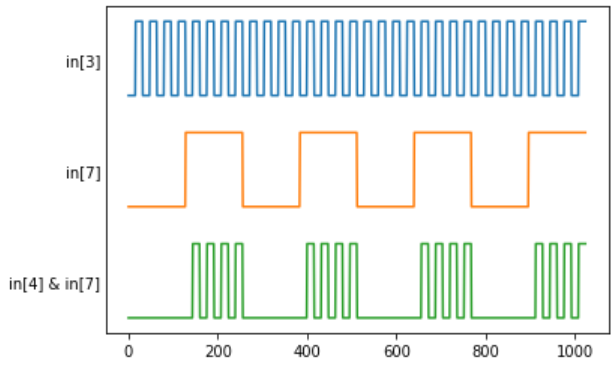

Figure 3: Example of an AND gate combining inputs of different periodicities, producing an output that spans multiple levels of musical perception.

noticed that, even without any training, random logic gate networks often sounded interesting, at least relative to their own apparent complexity. This observation prompted me to build `glitchgate` for exploring and manipulating logic gate networks, but it leaves open the question of why logic gate networks often sound musical. Thus, this section offers some preliminary analysis of the musical affordances of logic gates at the audio level and beyond.

To begin, consider the properties of the input to the network. By feeding in the sample time as the bits of an increasing integer, we have a binary clock. Instead of viewing this clock as representing an increasing integer, we can view it as consisting of several independent square waves of repeatedly doubling periodicity. For example, the least significant bit of the sample time flips every sample; that is, it is a square wave with a frequency of 4 kHz. The next most significant bit in the sample counter only flips every two samples; its period is twice as long as the previous bit and so its frequency is halved (2000 kHz). This pattern continues for each subsequent bit of the sample counter, so we are feeding in a family of square waves to the network with periods related by powers of two.

The gates inside the network then mix and combine these square waves using logical operations, resulting in interference patterns between different periodicities. Looking just at the input bits, we can already see important musical biases. For the less significant bits, which flip rapidly, the period-doubling relation corresponds to an audible octave relationship. For the more significant bits, where the periods are longer and the frequencies lower, the doubling instead corresponds to a subdivision relation, giving rise to rhythm, meter, and form. For example, the fourteenth bit will flip roughly every two seconds ($\frac{2^{14}}{8000Hz} = 2.048s$) whereas the thirteenth bit will flip every second ($\frac{2^{13}}{8000Hz} = 1.024s$). In other words, logic gate networks have a bias towards duple subdivisions and meters.

The rest of the network consists of logic gates applied to these inputs, with the result that every gate's output is some combination of these power-of-two periodicities—an interference pattern of square waves. The entire network is biased towards subdivisions of powers of two. Because the network generates audio at the sample level, simultaneously determining every level of musical hierarchy (from timbre to form), this effect is pervasive. The tendency towards octave relationships at the timbral level is identical to the tendency towards duple subdivisions at the rhythmic level.

A	B	$A \odot B$
0	0	1
0	1	0
1	0	0
1	1	1

A	B	$A \cdot B$
−1	−1	+1
−1	+1	−1
+1	−1	−1
+1	+1	+1

Table 1: Equivalence of XNOR (left) and multiplication (right) for binary signals: at audio rate, XNOR performs ring modulation.

After the input, which introduces power-of-two periodicities into the network, the logic gates determine how those periodicities combine and interfere. As with the inputs, we can consider the effect of the gates at different levels of perception. For example, we can see the effect of AND at the rhythmic level as a gate (in the conventional audio-processing sense): if the first operand is high, the other signal is allowed through; if the first operand is low, the second signal is blocked. Thus if the first operand is slow-changing and the second is fast-changing (e.g. a perceptible tone), the AND gate, acting as a signal gate, serves to impose of a rhythm on the tone, or (equivalently) fill the substanceless form with the tone, either way combining the rhythm of the first signal with the timbre of the second as shown in Fig. 3. If both signals are fast-changing, AND effectively performs amplitude modulation.

As another example, consider the XNOR gate (the inverse of XOR). When both inputs are the same, the output is high; when they differ, the output is low. As depicted in Table 1, this has exactly the same effect as simple multiplication of audio signals, if those signals are restricted to being high (+1) or low (-1). Thus we have a clear analogy for XNOR at the audio level: it acts as a frequency mixer. In signal processing terms, XNOR performs ring modulation.

Furthermore, we can view XOR as performing phase modulation. Logically, XOR can be viewed as a controlled NOT gate: when the first signal is low, the second signal passes through unchanged. When the first signal is high, the second signal is inverted. At audio rate, inversion is a phase change (180°), so one signal modulates the phase of the other. This observation, along with the connection to XNOR, suggests that when we restrict ourselves to binary signals, ring modulation and phase modulation converge.

5.2 Playing the Network

As mentioned in §3, I implemented visual feedback to watch the network operate. This visualization immediately suggested some intuition for why small logic gate networks, featuring simple positional binary encoding, might tend to sound musical, as described in the previous section. For the same reasons, this visualization has proven useful for manipulating the network.

`glitchgate`'s network visualization allows the player to see the output of each gate as the network runs. Because the audio rate is much faster than the display refresh rate, this visual output is averaged over the last block of audio (128 samples). Thus, rapidly-changing outputs appear in shades of gray, while slower outputs visibly blink as their outputs change across blocks. In this way, the visualization gives the player a sense of which perceptual levels gates have an effect on: a solid gray gate has an impact at the level of timbre, while a gate that slowly blinks black and white has an impact at the level of form. This gate-level visual feedback gives the player some idea of what timescale(s) (and thus musical elements) will be affected by intervention at a given node in the network.

With this information, the player can then intervene in the network via pointing and touching interactions. In the web browser, the player can use the mouse or a touchscreen to directly choose gates to mask. On the Bela, the gates are stretched and packed to fit into a square, which is mapped onto the surface of the Trill Square. The web version also supports this square layout, which is especially useful when used in conjunction with the Bela as in Fig. 2 (see also `bela_and_web.mp4` in the supplementary materials).

Overall, the spatial layout of the network is somewhat arbitrary; the same network could be laid out many different ways. The x-axis corresponds to depth within the network, as consecutive layers are laid out horizontally. However, the y-axis has no intrinsic meaning. In an effort to make the spatial relationship between gates correspond a bit more closely to the functional relationship, the gates are vertically sorted in each layer by their input operands, with the rough goal of putting gates near the gates they are fed by or feed into in preceding/succeeding layers. However, this scheme could likely be improved upon.

`glitchgate` aims to make interventions in the network feel quick and cheap. In the technical implementation, this aim is why the network is interpreted rather than recompiled every time it changes. In the interface design, this aim is why the application supports touching and dragging gestures to paint over the network in quick strokes. These interventions are meant to be ephemeral, so they disappear when the touch is released.

6 Future Work

In this paper, I have described the musical possibilities afforded by small audio-generating logic gate networks, and presented an application for playing with them.

I envision several possible improvements to `glitchgate` itself. For example, it may be useful to have tools for exporting logic gate networks to other formats, such as small generative music programs (suitable for use in ScoreCard), hardware descriptions in Verilog (suitable for implementation on FPGAs or ASICs), or patches for languages such as Max/MSP and Pure Data. I would also like to give the user more precision tools for working with the network: for example, allowing the user to inspect the output of individual nodes aurally or visually (as in a logic analyzer), and to more easily build up logic gate networks from scratch. As it is, the application is strongly oriented towards discovering musical networks, and it remains to be seen how feasible it is to compose such networks intentionally.

Another avenue to explore is enriching the space of logic gate networks. Currently, the networks supported by `glitchgate` are exclusively feedforward. Allowing for feedback would open up the possibility of memory (as in digital flip-flops or latches) and thus sequential logic, significantly expanding the space of possible networks and the complexity of their output.

Looking beyond `glitchgate`, future research might focus on generating logic gate networks to approximate specific sounds or existing pieces of music—in effect using logic gate networks as a highly-compressed (lossy) audio codec. (Such networks could then be manipulated in the ways described here.) It would also be valuable to conduct a more rigorous analysis of the spectral and perceptual possibilities of logic-gate audio synthesis, as I have only begun to scratch the surface in the discussion here.

More broadly, I hope that the work I have described here contributes to and inspires future work in frugal, unconventional, and 'low-tech' approaches to audio synthesis and music technology, as a countercurrent to trends towards ever 'smarter', costlier, and more complex models.

7 Ethical Standards

This research was supported in part by Google (as part of the 2024 Google Summer of Code), BeagleBoard.org, and Bela. One aim of this research is to find rich musical spaces and interfaces using fewer computing resources.

Acknowledgments

Thanks to Jack Armitage and Chris Kiefer for their mentorship on the Google Summer of Code project from which this work emerged.

References

[1] Jack Armitage, Victor Shepardson, and Thor Magnusson. 2024. Tölvera: Composing With Basal Agencies. 282–291. https://doi.org/10.5281/zenodo.13904854 ISSN: 2220-4806.

[2] Till Bovermann and Dave Griffiths. 2014. Computation as Material in Live Coding. *Computer Music Journal* 38, 1 (March 2014), 40–53. https://doi.org/10.1162/COMJ_a_00228 Conference Name: Computer Music Journal.

[3] Nick Bryan-Kinns, Berker Banar, Corey Ford, Courtney N. Reed, Yixiao Zhang, Simon Colton, and Jack Armitage. 2021. Exploring XAI for the Arts: Explaining Latent Space in Generative Music. https://openreview.net/forum?id=GLhY_0xMLZr

[4] Anders Carlsson. 2009. The Forgotten Pioneers of Creative Hacking and Social Networking – Introducing the Demoscene. In *Proceedings of the Third International Conference on the Histories of Media Art, Science and Technology (Re:live Media Art Histories)*.

[5] Kim Cascone. 2000. The Aesthetics of Failure: "Post-Digital" Tendencies in Contemporary Computer Music. *Computer Music Journal* 24, 4 (2000), 12–18. https://www.jstor.org/stable/3681551 Publisher: The MIT Press.

[6] Ian Clester and Jason Freeman. 2024. ScoreCard: Generative music programs as QR codes. In *Proceedings of the International Web Audio Conference*.

[7] Qubais Reed Ghazala. 2004. The Folk Music of Chance Electronics: Circuit-Bending the Modern Coconut. *Leonardo Music Journal* 14 (Dec. 2004), 97–104. https://doi.org/10.1162/0961121043067271

[8] Ville-Matias Heikkilä. 2011. Discovering novel computer music techniques by exploring the space of short computer programs. https://doi.org/10.48550/arXiv.1112.1368 arXiv:1112.1368 [cs].

[9] Hundredrabbits. [n. d.]. 100R — orca. https://100r.co/site/orca.html

[10] Felipe M. Martins and José Henrique Padovani. 2023. Be Brief: Convergences and Possibilities of Live-Coding and sctweeting. In *Proceedings of the 7th International Conference on Live Coding (ICLC2023)*. Utrecht, Netherlands. https://zenodo.org/records/7843864

[11] Tristan Perich. 2007. 1-Bit music. In *Proceedings of the 7th international conference on New Interfaces for Musical Expression (NIME '07)*. Association for Computing Machinery, New York, NY, USA, 476. https://doi.org/10.1145/1279740.1279897

[12] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. 2022. Deep Differentiable Logic Gate Networks. In *Proceedings of the 36th Conference*

on Neural Information Processing Systems. arXiv. https://doi.org/10.48550/arXiv.2210.08277 arXiv:2210.08277 [cs].