VOXOS: Vocoder on FPGA

Andrew Li
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
andyli@mit.edu

***Abstract*—We present VOXOS, a high-fidelity synthesizer and vocoder implemented entirely on FPGA without any use of IP. The synthesizer supports pitch-bend, vibrato, attack-decay-sustain-release (ADSR) envelopes and four waveform types with control via an external MIDI keyboard. The vocoder mixes this synthesized signal with the human voice or arbitrary line-in through 24 fourth-order infinite impulse response (IIR) filters to produce a high-fidelity 24-bit vocoded output at 48kHz.**

***Index Terms*—vocoder, synthesizer, audio, digital signal processing**

I. Motivation

A vocoder is an instrument that performs speech synthesis by "projecting" a vocal audio signal's features onto a carrier signal. Originally intended for voice compression during wartime, the vocoder has seen extensive use in popular music for its robotic effect. Yet, there is no prior open source art we could find on implementing the instrument on an FPGA. The effect is not challenging to understand and implement mathematically, and offers an informative first-look into digital signal processing while balancing audio quality and resource utilization.

II. Overview

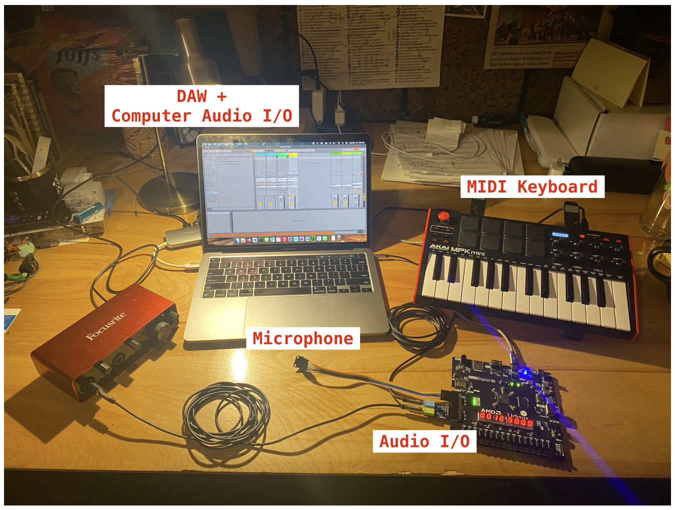


Fig. 1: Full vocoder system with typical audio production usecase

The VOXOS system consists of:

- A RealDigital Urbana FPGA board compatible with Diligent's Arty S7-50 board
- A SPH0645 MEMS microphone from Adafruit
- A Pmod I2S2 module from Diligent
- A MIDI keyboard

Figure 1 demonstrates a typical usecase where VOXOS' output is recorded in a digital audio workstation (DAW). Since the output is analog line-level, it can be used directly with headphones or downstream amplifiers and effects chains. A computer is necessary to VOXOS to receive MIDI messages.

III. Audio Processing

The vocoder was invented in 1928 at Bell Labs by Homer Dudley as means of speech compression and resynthesis. It was later integrated into SIGSALY, an encrypted voice communication system during World War II [1].

The theory is relatively simple: human speech usually resides in the range of 50–5000Hz. By routing this signal through many band-pass filters spanning this range and detecting its envelope, we have the approximate vocal spectral content over time. This binned spectral content is more efficient to transmit than the original vocal signal. Critically, we can resynthesize this vocal signal by mapping this spectral content onto a carrier signal. This is done by using the same filterbank on the carrier signal and then amplifying each band by the height of the corresponding vocal band envelope.

Dudley originally implement his vocoder through a series of high-Q analog bandpass filters and voltage-controlled amplifiers. In the digital domain, we can achieve a similar result through infinite impulse response (IIR) filters and a dot-product between the carrier and vocal envelope.

A. IIR Filter Design

Our bandpass filterbank consists of $N = 24$ fourth-order Butterworth IIR filters covering the range 50–7000Hz shown in Figure 2. This range is logarithmically split into 25 bins, with each filter's 3dB cutoff on either side set to each frequency bin endpoint. The higher 7kHz limit was chosen to accommodate all the synthesizer's fundamental frequencies without an additional highpass filter. These filters were realized as cascaded biquads via MATLAB.

We chose to implement IIR filters for their well-behaved frequency response to accurately estimate vocal spectral content. While finite impulse response (FIR) filters have greater stability, they require higher order, and thus FPGA resources, to achieve comparable frequency response. Specifically, fourth-order Butterworth IIR filters were chosen for their maximally

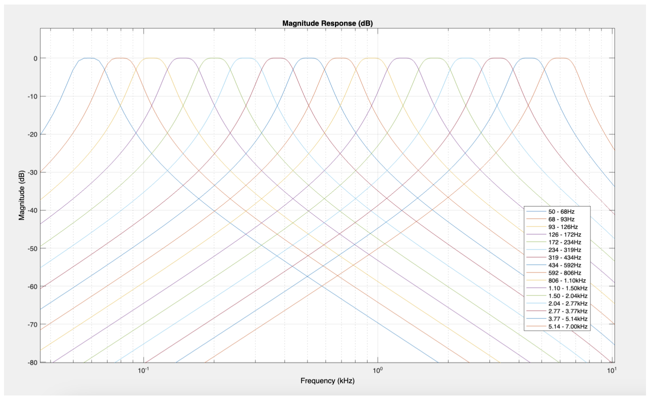


Fig. 2: Frequency response of 16 fourth-order Butterworth filters covering 50–7000Hz

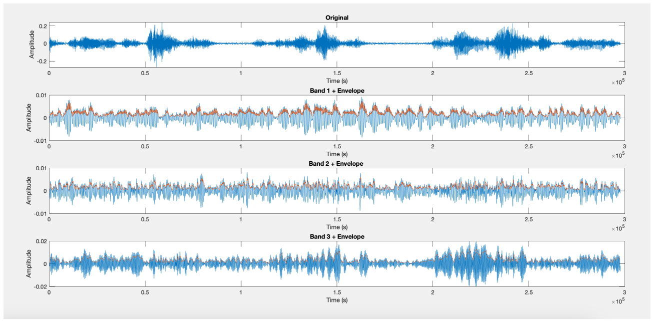


Fig. 3: Envelope detection on the first 3 filtered bands of a speaking sample

flat frequency response over the band of interest and increased rolloff within FPGA resource constraints.

B. Envelope Detection

We implement envelope detection on the vocal signal bands by using a fourth-order lowpass Butterworth filter with cutoff frequency of 100Hz on the rectified signal. The cutoff was determined by trying many values from 20–250Hz and picking the one with best detection results. See Figure 3 for visual results.

C. Noise Generation

Some sounds are not well-suited for this type of resynthesis. In particular, purely consonant sounds, unvoiced sounds (plosives, sibilants, fricatives) that involve stopping the vocal cords are hard to capture via frequency band analysis. This is due to their high frequency, like /s/, or low energy content, like /n/ at the end of "sun." In Dudley's original vocoder, he added a noise generator which became the fundamental carrier when it detected high frequency input.

We experimented with a similar approach by combining a highpass filter and 31-bit LSFR noise generation. But through testing an preliminary implementation on MATLAB, we found this to not produce any better sounding output. Depending on the final mix, this would also greatly obscure some other sounds. Given that during testing, square and sawtooth carriers produced good clarity for these sounds, we theorize that these waveforms already have enough harmonics, and that the 7kHz endpoint is sufficient to track and reproduce these speech artifacts. We did not finish implementing this in the VOXOS.

IV. DSP Implementation

A. Coefficient Generation

VOXOS performs all digital signal processing in fixed-point arithmetic. Biquad filter coefficients were generated in MATLAB first in variable-precision arithmetic, scaled by 2^{20}, rounded to the near whole number, then stored in a memory file as 32-bit signed integers. The scaling factor was chosen to give comfortable precision to the smallest coefficients of the envelope lowpass filter. Intermediate calculations are sign-extended to 64-bits to prevent overflow and right-shifting to remove the scaling factor is performed as late possible to maximize resolution.

B. Pipelining and Timing

Bit growth and timing become larger considerations when working with wide multiplication, addition, and shifting operations. In particular, VOXOS computes with 24-bit audio samples and 32-bit coefficients with 64-bit intermediates. To satisfy timing constraints, each cascaded biquad performs just one multiply, add, or shift operation per clock cycle.

Furthermore, the RealDigital Urbana FPGA has only 120 DSP48 slices. In experimentation, this equates to at most 28 cascaded biquads synthesized at once. To achieve voice, envelope, and carrier filtering, we serially timeshare these biquads. In particular, to allow for future filter extensibility, we require at least $35 \cdot N$ clock cycles between vocoded outputs where 35 is the number of pipeline stages for one filter based on our implementation and N is the number of desired filters.

We have a system clock of 98.304MHz, or 2048 times our 48kHz sampling rate. Through testing, we found that we must run our filterbank at most at 98.304MHz / 2 to meet timing constraints. Therefore the current upper bound on the number of filters is

$$\frac{98.304/2\,\text{MHz}}{48\text{kHz}} = 1024 \geq 35 \cdot N \implies N \leq 29. \tag{1}$$

This is a potential source of improvement in a future implementation by removing pipeline stages until exhausting available slack.

C. Mixing

When performing the dot-product on the 24-bit filtered carrier signal and the 24-bit envelope over N different filter bands, we have high probability of clipping. Much like how we pipelined the biquad filters, we pipeline the mixing process to first perform just one multiplication, addition, or shift per cycle.

Products, which are 48-bit, are saved to a 64-bit temp. On the next cycle, they are added to a 64-bit accumulator. After each band product is summed, the signal is potential $48 + \log_2 N \approx 53$ bits wide. We finally do a arithmetic right shift by 24, an experimentally determined amount that prevents most clipping and has satisfactory volume. Note that we do not shift by 29 back to 24-bit because filterbands are not perfectly correlated. Different filterbands would never produce a mixed result that would saturate the given width and the resulting audio would be too quiet.

V. Synthesizer

The second part of a vocoder is carrier signal generation via a synthesizer. We implement a synthesizer capable of generating sine, triangular, square, and sawtooth waves through direct digital synthesis (DDS).

A. Sine DDS

Instead of the complexity of calculating sinusoidal values, we store a lookup table of precomputed values and index into these values based on a phase that increments depending on the desired synthesized frequency.

In order to achieve high-fidelity synthesizer output, we require cent-resolution, i.e. 1/100 of a semitone, at 20Hz, the general lower-bound on human hearing. Therefore we need at least

$$\frac{(2^{1/12}-1)\,\text{Hz/semitone}}{100\,\text{cents/Hz}} \cdot 20\text{Hz} = 0.012\text{Hz resolution.}$$

If we want to produce up to 20kHz, the general upper-bound on human hearing, we need $\log_2(20,000/0.012) \leq 21$ bits to achieve this resolution. We round up to 24 bits at 192kHz which gives 192kHz / $2^{24} \approx 0.011$Hz resolution. Therefore, to produce a desired frequency f, we use a phase increment

$$\text{increment} = f \bigg/ \left(\frac{192\text{kHz}}{2^{24}}\right) \tag{2}$$

that is added to the phase accumulator at 192kHz.

Our ideal phase is 24-bit and ideal output at a given phase is 24-bit. But a BRAM lookup table of width 24 and depth 2^{24} is prohibitively large for the FPGA. Instead, we store 2^{14} samples of a quarter-period with amplitude 2^{23} to span the entire 24-bit width. Then, using the 2 MSBs from the phase, we index backwards and/or negate the sample to achieve a full period. Therefore we achieve 2^{16} effective phase resolution for

$$2^{14}\,\text{samples} \cdot 24\,\text{bits per sample} \approx 393\text{Kbits}.$$

These samples were generated via a Python script as 23-bit values, stored in a memory file, and synthesized as DRAM via SystemVerilog's `$readmemb`. The sign MSB was added in the module output through symmetry.

Square, triangle, and sawtooth waveforms are implemented as counters exploiting symmetry as well, then scaled to match the root-mean-square amplitude of the sine wave.

VI. Audio I/O

VOXOS makes use of two external peripherals to enable voice or arbitrary input and mixed output. We maintain two base clocks: 98.304MHz and 36.864MHz, and implement I2S according to provided datasheets. Due to resource constraints, we support only mono audio I/O.

Using a switch, the user can choose between microphone input or line-in input as the vocoder's modulator. This, along with the synthesizer carrier output, is routed to the filterbank and mixer before the mixed result is routed to line-out.

A. Microphone

The SPH0645 I2S MEMS microphone has a builtin anti-aliasing filtering for voice capture, so we only handle the I2S driver implementation. For mono input, we set word-select to 0. Microphone samples are then 18-bit twos-complement clocked in MSB first every 64 bit times so. Therefore, we run its serial clock (SCLK) at $48\text{kHz} \cdot 64 = 98.304\text{MHz}$ / 32. These values are then left-shifted by 6 to produce 24-bit signed values.

B. Line In and Out

We also implement an I2S driver for the Pmod I2S2 module. The line-out driver has a master clock (MCLK) at 36.864MHz to allow for an SCLK of 36.864MHz / 16 = 48kHz $\cdot$ 48 to be generated for 2-channel 24-bit audio to be clocked in at the sampling rate. The same mono output is sent to both channels via a 48kHz left-right clock (LRCK), but this clocking scheme is necessary to communicate with the device correctly. The line-in driver shares a similar structure. The MCLK, SCLK, and LRCK are shared and we only sample the right-channel output.

VII. MIDI Control

VOXOS implements a portion of the MIDI protocol to support common musical usecases. These include:

- Note on/off events for notes 12-108 (C0 to C8), a slight extension of the standard piano keys
- Pitch-band update events
- Modulation update events for vibrato
- Attack, decay, sustain, and release update events

Due to setbacks with trying to implement a driver for the MAX3421E USB host controller, we were not able to implement a EQ control for the vocoder frequency bands.

A. UART Communication

We originally intended to write a hardware driver for the onboard MAX3421E USB host controller to directly communicate with MIDI devices. Without a working USB packet analyzer, this was extremely hard to debug, especially when we followed the documentation and programming guides very closely. More details on our efforts and possible future work is listed in the Appendix.

Instead, we connect a MIDI keyboard to a computer, use its software drivers to receive raw MIDI packet bytes in Python, then transparently send them to the FPGA over UART. See Figure 5 for an example of the Python script sending note events. We used Manta's UART receiver module [4] to receive messages at a 3Mbps baudrate.

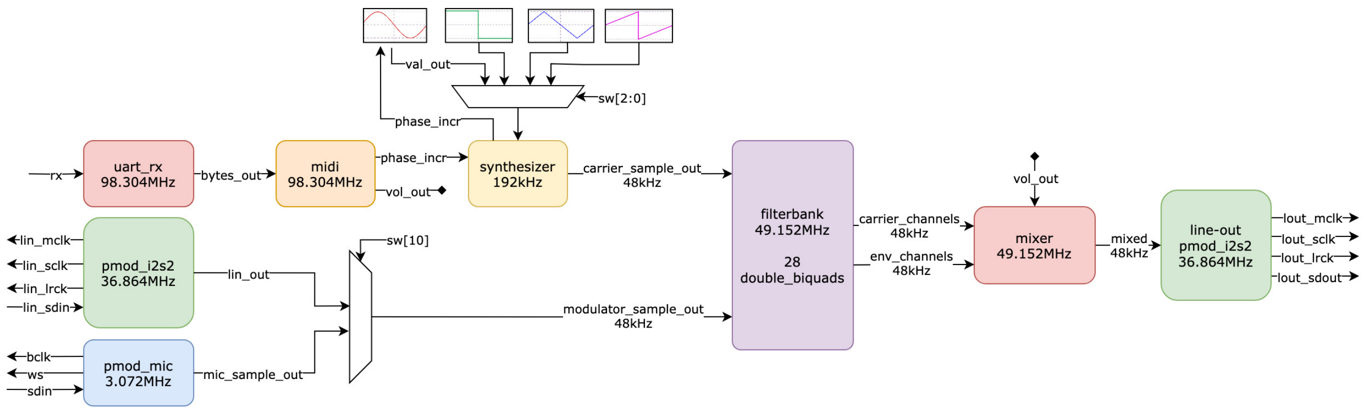


Fig. 4: Condensed block diagram. Pipelined double_biquad modules are not shown for brevity.

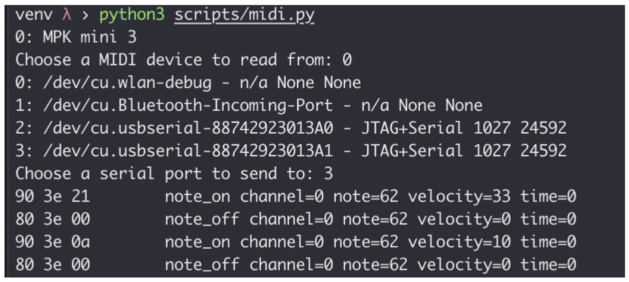

```
venv λ › python3 scripts/midi.py
0: MPK mini 3
Choose a MIDI device to read from: 0
0: /dev/cu.wlan-debug - n/a None None
1: /dev/cu.Bluetooth-Incoming-Port - n/a None None
2: /dev/cu.usbserial-88742923013A0 - JTAG+Serial 1027 24592
3: /dev/cu.usbserial-88742923013A1 - JTAG+Serial 1027 24592
Choose a serial port to send to: 3
90 3e 21        note_on channel=0 note=62 velocity=33 time=0
80 3e 00        note_off channel=0 note=62 velocity=0 time=0
90 3e 0a        note_on channel=0 note=62 velocity=10 time=0
80 3e 00        note_off channel=0 note=62 velocity=0 time=0
```

Fig. 5: Python script that sends MIDI bytes to the FPGA

TABLE I
SUPPORTED MIDI PACKETS

Type	Status Byte	Data Bytes
Note on	`1001 CCCC`	`0PPP PPPP 0VVV VVVV`
Note off	`1000 CCCC`	`0PPP PPPP 0VVV VVVV`
Pitch bend	`1110 CCCC`	`0LLL LLLL 0MMM MMMM`
Modulation wheel	`1011 CCCC`	`0000 0001 0MMM MMMM`
Attack time	`1011 CCCC`	`0100 1001 0MMM MMMM`
Decay time	`1011 CCCC`	`0100 1011 0MMM MMMM`
Sustain time	`1011 CCCC`	`0100 1000 0MMM MMMM`

B. MIDI Module

MIDI packets consist of a **status** byte and **data** bytes. The following is a table of status and data byte values for the packets VOXOS supports [2].

Here, `C` represents part of the channel number, `P` is pitch from $0-127$, `V` is velocity, `L, M` are least and most significant bits respectively. This is necessary for pitchbend which is a 14-bit value where 0x2000 is the center.

The MIDI module detects when the packet bytes change and processes the new event. When it receives a Note On event, it registers the pitch value between 12-108. This value is used to look up the corresponding phase increment to send to the synthesizer. This lookup table was generated in Python with a reference frequency table [3] and rounding the results from equation 2 and synthesized as DRAM.

When a Note Off event is received, we check if the event is for the most-recently-pressed note. This allows for smoother note transitions especially when playing faster. Currently, we only support one key pressed, but this is easily extensible.

C. Pitchbend

Pitchbend involves changing the pitch of a synthesizer given an intensity value. The AKAI MPK mini keyboard we use does not use the least significant bits of the intensity, so we have 127 possible pitchbend values. We chose to map these onto a ± 1 semitone range, where an intensity of 64 = 0x20 is the center.

Pitch is proportional to the phase increment, so we can multiply the phase increment by $2^{\pm 1/12}$ to achieve the desired effect at the extremes. The rest of the values were mapped logarithmically to produce a smooth variation. Using fixed-point arithmetic with scaling factor 2^{20} again, we produced pitchbend factors for intensities 0 to 127. When a pitchbend event arrives, we update the factor and right-shift to remove the scalar. These were also synthesized as DRAM.

D. Modulation and Vibrato

The modulation wheel was mapped to vibrato. Vibrato is essentially an oscillating pitchbend, with greater intensity mapped to faster oscillation. We used our sine module as a low-frequency oscillator (LFO) with varying frequency between 0 and 10Hz. Another DRAM lookup table was generated for 127 phase increment values corresponding to 0 and 10Hz. On receiving a modulation wheel event, its intensity indexes directly into this table.

Because the LFO output is 24-bit, we right-shift, convert to unsigned, then add an offset to map the sine wave onto values $[32, 96]$ which index into the pitchbend table. This gives the vibrato effect half-semitone range. The pitchbend and vibrato multiplication and shifts are all combinational to reduce complexity, but can be pipelined for timing.

E. ADSR Envelope

An ADSR envelope allows for amplitude control over time so that the synthesizer output can grow and decay over time. See Figure 6 for a visualization. The MIDI module also implements a state machine for attack-decay-sustain-release

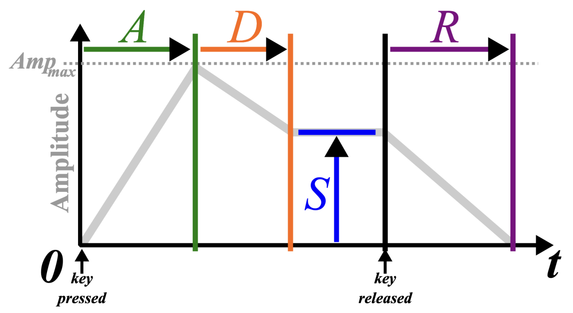


Fig. 6: ADSR envelope that controls amplitude over time. Image from Wikipedia.

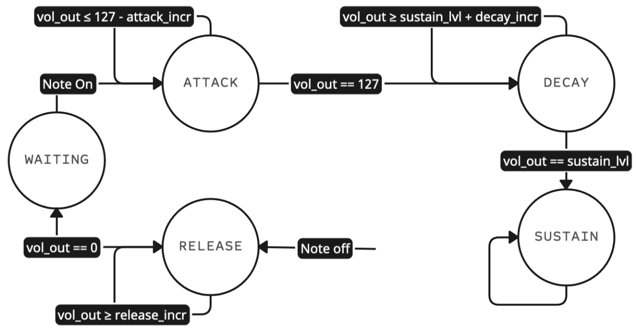


Fig. 7: Attack-Decay-Sustain-Release state machine, with conditions to prevent overshoot

envelope support illustrated in Figure 7. This generates a 0-127 volume scalar that is applied to output of the mixer before a right-shift by 7 to maintain unity gain.

Attack, decay, and release are units of time. A higher attack time setting corresponds a longer time until the synthesizer reaches its maximum volume after key press. Similarly for decay and release. Therefore when we receive these events we set an attack, decay, or relase increment that is $128 - V$ where V is the event value in the packet. In contrast, sustain is a level setting, so we directly set V.

When a note on event is received, we increment the volume scalar by the current attack increment up to 127, then decrement by the decay increment until we reach the sustain level. When a note off event is received, we decrement until we reach 0 for release. See Figure 7 for the state machine, which includes conditions to prevent volume overshoot and clipping. To perceive this envelope on human timescales, we perform an increment every 2^{20} master clock cycles with a counter. This gives us a maximum attack time of up to

$$\frac{1\,\text{increment}}{98.3\text{MHz}/2^{20}} \cdot 128\,\text{increments} \approx 1.4\,\text{seconds}$$

VIII. Evaluation

A. Project Success

One of the main goals for this project was audio quality. A lot of effort was put into investigating bit-growth, rounding techniques [5], filter stability, and comparing against commericial options like the Roland VP330, Behringer VC340. With respect to usability, the instrument can be used effectively as a performance instrument with enough functionality to be interesting in our opinion.

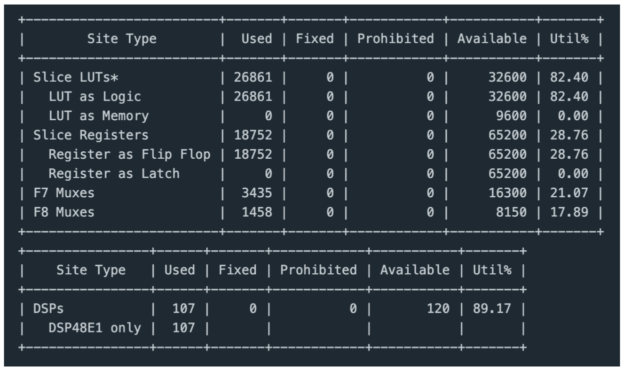
We met all goals except for the vocoder band EQ stretch goal. We did not fully implement noise-generation due to its lack of benefit for output quality. Through testing, VOXOS performs best when the carrier has many harmonics, such as with the square and sawtooth waves, to allow for better unvoiced feature capture.

Another goal that often made this harder was using as little IP as possible. This was the author's first time working extensively with audio DSP, HDLs, and MATLAB. All drivers, with exception of the UART RX module, were testbenched and written manually. Additionally, filters were designed and implemented from first principles. These designs were evaluated against filters generated via MATLAB's HDL Coder and empirically had better sound results.

With these in mind, VOXOS still has room for future work. In particular:

- Better handling of clipping. VOXOS produces satisfactory results with occasional almost-resonant clipping with certain waveforms and certain frequencies.
- Standalone USB MIDI support with a hardware driver
- Multi-touch support to play chords
- Real subtractive synthesis support with adjustable synthesizer filters

B. Resource Utilization



```
+-------------------------+-------+-------+------------+-----------+-------+
|        Site Type        |  Used | Fixed | Prohibited | Available | Util% |
+-------------------------+-------+-------+------------+-----------+-------+
| Slice LUTs*             | 26861 |     0 |          0 |     32600 | 82.40 |
|   LUT as Logic          | 26861 |     0 |          0 |     32600 | 82.40 |
|   LUT as Memory         |     0 |     0 |          0 |      9600 |  0.00 |
| Slice Registers         | 18752 |     0 |          0 |     65200 | 28.76 |
|   Register as Flip Flop | 18752 |     0 |          0 |     65200 | 28.76 |
|   Register as Latch     |     0 |     0 |          0 |     65200 |  0.00 |
| F7 Muxes                |  3435 |     0 |          0 |     16300 | 21.07 |
| F8 Muxes                |  1458 |     0 |          0 |      8150 | 17.89 |
+-------------------------+-------+-------+------------+-----------+-------+
+----------------+------+-------+------------+-----------+-------+
|    Site Type   | Used | Fixed | Prohibited | Available | Util% |
+----------------+------+-------+------------+-----------+-------+
| DSPs           |  107 |     0 |          0 |       120 | 89.17 |
|   DSP48E1 only |  107 |       |            |           |       |
+----------------+------+-------+------------+-----------+-------+
```

Fig. 8: Vivado resource utilization report

Our final design had 4.206ns of available slack and used nearly 90% of the DSP48s onboard for 24 cascaded biquad filters. This is very close to the practical maximum without optimizing the 64-bit temps and accumulators very carefully, as we observed 28 filters would take up 123 blocks through an attempted synthesis. Any effort to further timeshare the synthesized filters would involve more pipeline stages further restricting the bound given in inequality 1. Additionally, we cannot run the filters directly at the master 98.304MHz without massively violating timing.

IX. Retrospective

Overall the project was incredibly valuable for the learning experience besides creating a fun instrument. A few key takeaways:

- *Testbench always.* Especially if you're writing finnicky drivers. YOLO'ing and hoping for the best will lead to lots and lots of pain in the future when you spend a week debugging a weird "filter stability issue" that turns out to be a cycle-off error that you should've caught three weeks back.
- *Hardware isn't software.* You can't just write 5 64-bit multiplications in a row for 24 modules and expect to meet timing. You also can't expect a fast iteration cycle when builds take 10 minutes. Ensure you fully understand a timing diagram or datasheet before sinking a ton of time.
- *Implement AXI.* VOXOS two different base clocks and many different clock domains that become hard to keep track of even with extensive comments. Weird pipelining bugs between clock domains will inevitably arise. We should have implemented AXI early on for all our modules to make this problem a lot more manageable, or at least some kind of shared registering and valid/ready scheme.

X. Appendices

A. USB Host Controller Driver

The first three weeks of the project were spent on developing a hardware driver for the onboard MAX3421E USB host controller. This effort was almost successful, but without a packet analyzer we hit a wall. For the sake of posterity, we will describe our progress.

- The MAX3421E communicates over SPI. It has 32 internal register that allow you to handle interrupts for USB connection, speed probing, data receive and transmit status, among others. We wrote an SPI controller to do multi-byte register reads and writes.
- One should follow a software driver as a reference as the provided programming guide [8] is lackluster (and contains errors). We relied on a few implementations [6] [7].

The general setup flow involves:

1) Set the MAX3421E to operate in full-duplex mode. Therefore, you can read USB status and registers while also writing data and you don't have to deal with tri-stating things.
2) Perform a hardware reset unless you can't set the HOST bit to operate as a USB host
3) Check if the oscillator is ready after the reset with the OSCOKIRQ bit
4) Set GPOUT0 high which enables 5V for peripherals
5) Set HOST bit, DPPULLDN, DNPULLDN for connection detection
6) Set the SAMPLEBUS bit then read J, K states in the USB status bits to see if a USB was connected and what speed it operates at
7) Wait 200ms for the USB bus to settle
8) Wait for SOF interrupts to come in, so the bus is ready to use
9) If you know your peripheral's USB descriptors, you can skip enumerating the device. Otherwise read up on GET_DEVICE and GET_CONFIGURATION requests
10) Set an address for your peripheral by clocking in the 8-byte SET_ADDRESS SETUP packet into SUDFIFO to endpoint and address 0, and write 0 to HXFR to trigger a SETUP packet send
11) Read the USB status bits and wait until you get a clean HXFRSTATUS bit. If you get a busy or NACK, retry.
12) Then write 0x80 to HXFR to trigger a STATUS packet send which ends the SET_ADDRESS command.
13) Wait 20ms for the peripheral to actually change its address
14) Do 10 but now with a SET_CONFIGURATION command, and with the address you just set and still endpoint 0. This is where we got stuck. No matter what we tried, the peripheral would always respond with a 0xE status, or J-state error. There wasn't any documentation on this so without a packet analyzer we could not find out where in the transaction we went wrong.
15) Theoretically, after you set the configuration, you can now send in periodic BULK_IN packets, which would return back the 3-byte MIDI packets we needed. Alas, we were so close!

This entire flow is documented in more detail in `max3421_spi.sv` and `usb_controller.sv`.

B. Code Repository

The code is on GitHub at https://github.com/Li357/voxos

C. Acknowledgements

Thanks to Joseph Feld and Joe Steinmeyer for their help in trying to make USB work, fixed-point and DSP advice, and for all the TAs/LAs in lab all semester!

References

[1] https://en.wikipedia.org/wiki/Vocoder
[2] https://www.cs.cmu.edu/ music/cmsip/readings/MIDI%20tutorial%20for%20programmers.html
[3] https://en.wikipedia.org/wiki/Piano_key_frequencies
[4] https://github.com/fischermoseley/manta
[5] https://zipcpu.com/dsp/2017/07/22/rounding.html
[6] https://github.com/calvinlclee3/fpga_soc/blob/master/software/text_mode_vga/usb_kb/MAX3421E.c
[7] https://github.com/jakakordez/max3421e-stm32/blob/master/Src/MAX3421E.c
[8] https://www.analog.com/media/en/technical-documentation/user-guides/max3421e-programming-guide.pdf