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 with control via an external MIDI keyboard. The synthesizer supports pitch-bend, modulation, attack-decay-sustain-release (ADSR) envelopes and four waveform types. The vocoder mixes this synthesized signal with the human voice through 16 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

FPGAs are uniquely suited to digital audio signal processing applications. They afford the flexibility to implement and quickly iterate on novel instruments and effects. The vocoder, which mixes a the human voice with a carrier signal, is one such instrument.

Originally intended for voice compression during wartime, the vocoder has seen extensive use in popular music. 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. 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 over this range and detecting its envelope, we have an estimate of 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-controller 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 envelope.

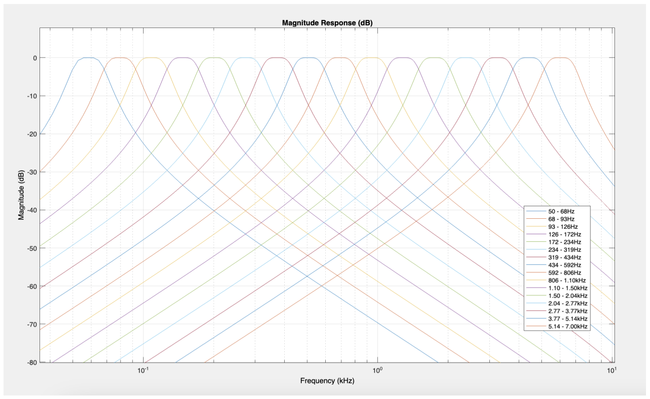

Fig. 1: Frequency response of 16 fourth-order Butterworth filters covering 50–7000Hz

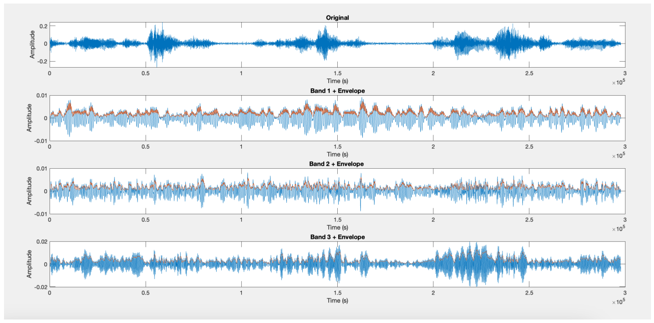

Fig. 2: Envelope detection on the first 3 filtered bands of a speaking sample

A. IIR Filter Design

Our bandpass filterbank consists of $N = 16$ fourth-order Butterworth IIR filters covering the range 50–7000Hz shown in Figure 1. This range is logarithmically split into 17 bins, with each filter's 3dB cutoff on either side set to each frequency bin endpoint. 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 flat frequency response over the band of interest, 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 determine by trying many values from 20–250Hz and picking the one with best detection results. See Figure 2 for visual results.

III. DSP Implementation

A. *Fixed-Point Arithmetic*

To keep implementation simple, VOXOS opts to do all filtering in fixed-point arithmetic. All biquad coefficients are stored in BRAM as 32-bit integers scaled by 2^{20}. This number 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.

B. *Pipelining*

Bit growth and timing become larger considerations when working with large multiplication, addition, and shifting operations. To satisfy timing constraints, each cascaded biquad performs just one of these arithmetic operations per clock cycle. This was a source of major problems, but running a base clock at a large multiple, 98.304MHz of our sampling rate 48kHz gives us leeway.

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 cycles for one filter band to produce a result based on our implementation and N is the number of desired filters. To meet timing constraints, we must run our filterbank at most at 98.304MHz/2, so the current upper bound on the number of filters is

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

This is a potential source of improvement by removing pipeline stages until we exhaust our available slack. See Figure 3 for appropriate valid in and out signals that allow for thus pipelining.

C. *Mixing*

When multiplying the 24-bit filtered carrier signal by the 24-bit envelope over N different filter bands, there is a large potential for clipping. We alleviate this by shifting each multiplied output by some arbitrary amount 2^{20} experimentally determined to give the best results. We also threshold the envelope experimentally so that only deliberate and close voices trigger vocoder output.

IV. Synthesizer

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

A. *Sine Direct Digital Synthesis*

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 48kHz for numbers that are easier to manipulate in the digital domain. Therefore, 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, so we accept some quantization noise by using the 16 MSBs of the phase to index into a table of 2^{16} values. Exploiting sinusoidal symmetry and using the 4 LSBs to linearly interpolate between consecutive table values, we get 2^{16+4} effective phase resolution for

$$2^{16}\,\text{samples} \cdot 24\,\text{bits per sample} \,/\, 4 \approx 393\text{Kbits}.$$

Other simpler waveforms are implemented as simple counters exploiting symmetry as well, then scaled to match the root-mean-square amplitude of the sine wave.

V. Audio I/O

VOXOS makes use of a 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. See Figure 3 for the implemented I/O signals over pmod.

A. *Microphone*

We use an Adafruit I2S MEMS microphone with builtin anti-aliasing filtering for voice capture. Microphone samples are 18-bit twos-complement clocked in MSB first every 64 bit times so we run the clock at $48\text{kHz} \cdot 64 = 98.304\text{MHz}/32$. These 18-bit values are sign-extended but otherwise left unmodified before filtering to maintain resolution.

B. *Line In and Out*

We also plan to support arbitrary audio input as a modulator instead of a voice signal. This is intended to add to the possibilities of the instrument. We use a Pmod I2S2 for line in and out, with samples clocked in and out every 48 bit times to account for 2 channels of 24-bit samples. Therefore we run this at $48\text{kHz} \cdot 48 = 36.864\text{MHz}/16$.

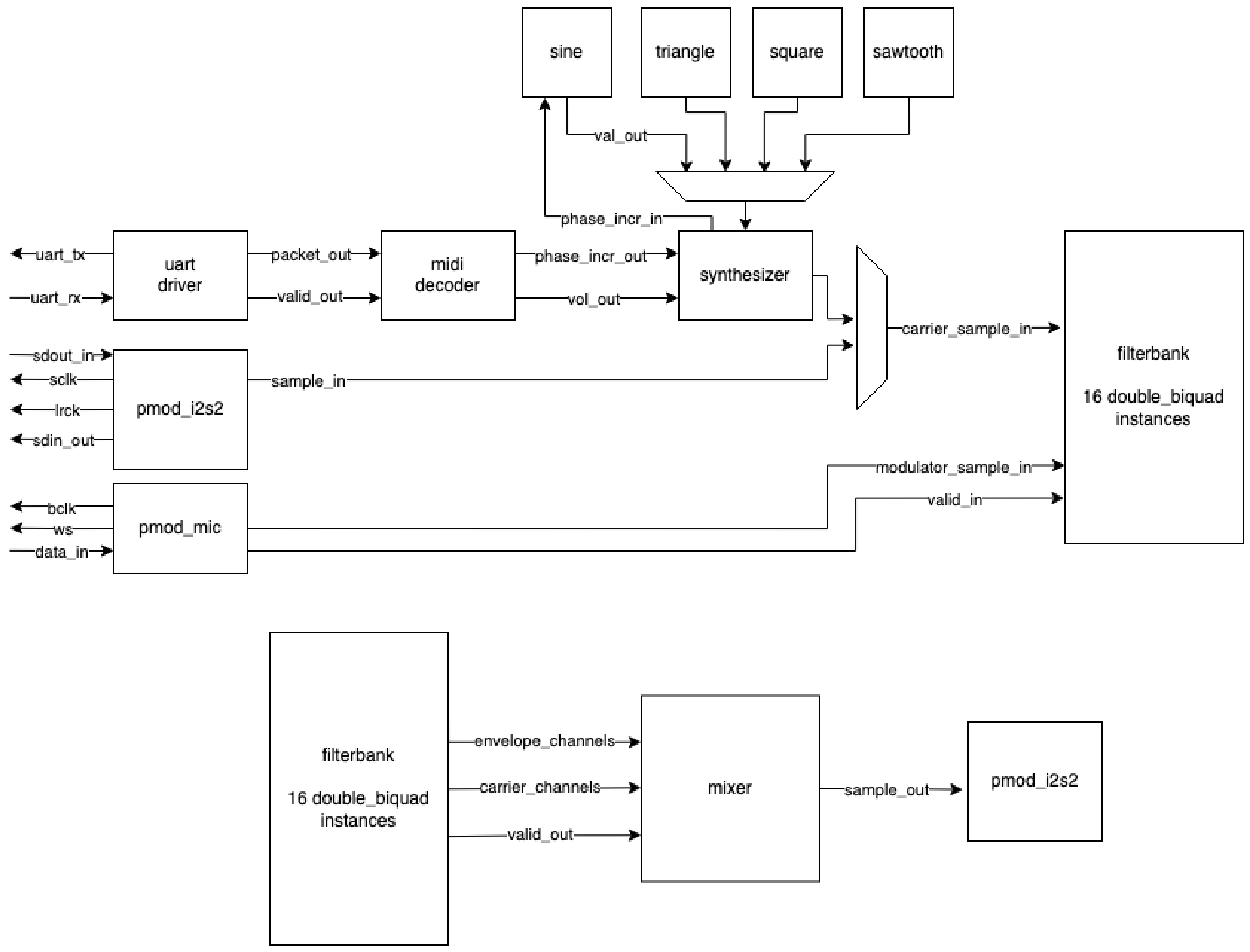

Fig. 3: Condensed block diagram. Pipelined double_biquad modules are not shown for brevity.

VI. MIDI Control

VOXOS will implement a portion of the MIDI protocol to support common musical usecases for the vocoder. Specifically:

- Note on/off events for 21-108 (standard piano keys from A0 to C8)
- Pitch-band update events
- Modulation update events
- Attack, decay, sustain, and release update events

Due to setbacks with trying to implement a driver for the MAX3421E USB host controller, progress on MIDI implementation is not done.

Instead of finishing the USB driver on the FPGA, we plan to use a computer to transparently pass MIDI packets from the controller to the FPGA over UART. Therefore, we still perform the packet handling on the FPGA but sidestep the unreliable host controller on the FPGA.

A. MIDI Protocol

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 will support [2].

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`

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.

B. Planned Modules

We will write a UART receiver that outputs a 3-byte MIDI packet when it receives a MIDI event. A MIDI module will then determine the frequency to drive the synthesizer at by calculating a phase increment from the current note that is on and the current pitch-bend value, if any. Additionally, we will use the modulation wheel value to control vibrato. Its value will control the frequency of $<$10Hz sinusoidal wave from our DDS. This will be scaled then added to the phase increment to produce a vibrato effect. The phase resolution is not of concern here.

Finally, we will have a state machine to control an amplitude scalar for the attack-decay-sustain-release envelope. When receving an attack, decay, or sustain time update, we update our internal counters. When a note on event is received, we increment up to the maximum amplitude scalar for attack, then decrement for decay until we reach a sustain level. When a note off event is received, we decrement until we reach 0 for release.

References

[1] https://en.wikipedia.org/wiki/Vocoder

[2] https://www.cs.cmu.edu/ music/cmsip/readings/MIDI%20tutorial%20 for%20programmers.html