

# PITCH SHIFT

Real-time Speech FPGA Implementation

Jake Garrison & Jisoo Jung



# OUR PROJECT GOAL

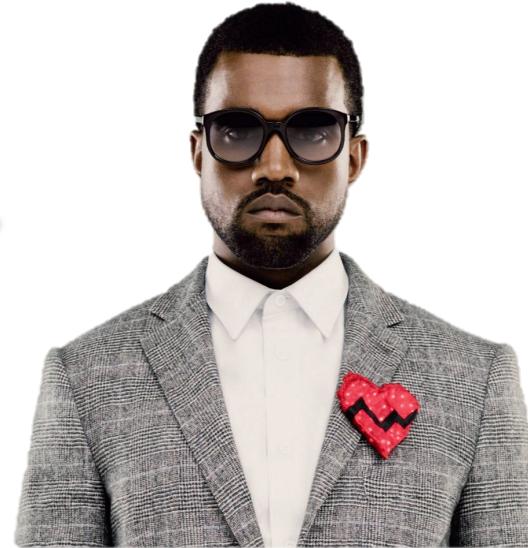
Realtime pitch shifting optimized for speech

Optional Features

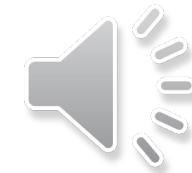
Input melody based pitch shift (autotune)

Harmony capabilities to output multiple pitch shifts to simulate a chorus

Autotune



Chorus



# HISTORY

**Definition:** *Pitch shifting is a sound recording technique in which the original pitch of a sound is raised or lowered*



- Before digital sound, this it was extremely difficult to shift the pitch by non interval values
- With tape or vinyl you could change pitch by altering the speed of playback
  - Example: Alvin and the chipmunks
- Analog octave pitch shift can be done with rectifier circuit
  - Example: Jimi Hendrix or Led Zeppelin guitar hardware



# APPLICATIONS

HAL 9000

Voice Privacy

Voice Gender or Age Change

Other Voice Effect

- Darth Vader, Alvin and the Chipmonks, SciFi Robot voice

Text to speech / Speech synthesis

Auto Tune / Pitch Correction

Harmony / Chorus

Karaoke

Sampling / Resampling Audio



# MOTIVATION

Seemed challenging... and is 😞

Fun to play with

Popular in modern music

- Most pop music relies on pitch correction

Voice privacy is becoming important

Commercial software implementations are expensive

Hardware implementations is even more expensive



## Software

Antares Auto-Tune  
Live - Pitch-...

**\$199.00**



## Hardware

Eventide  
PitchFactor ...

**\$499.00**

# EXISTING METHODS

## Analog

- Rectifier Circuit (octave only)

## Time Domain

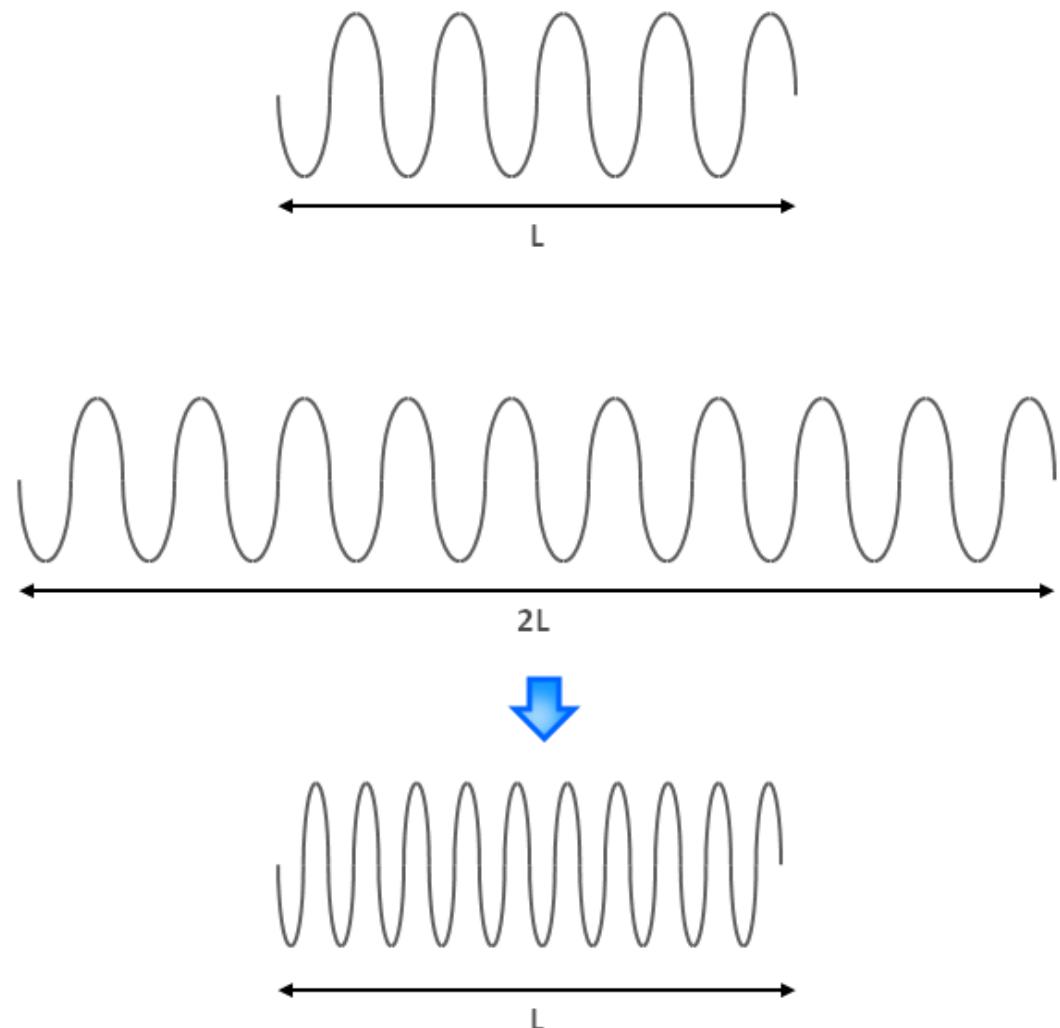
- PSOLA
- Delay Based

## Frequency Domain

- Phase Vocoder

## Physical Modeling

- Speech processing, inverse filtering (complex)



# TIME DOMAIN PITCH SHIFT

## PSOLA (Pitch Synchronous Overlap/Add)

'P' will resample the sound, changing the pitch

- Uses decimation, then LPF, then interpolation to achieve non-integer resampling
- This changes the duration

'S' uses correlation so synchronize the 'OLA'

- Uses correlation to find the best way to overlap and add the stretched or compressed sections

'OLA' is overlap and add

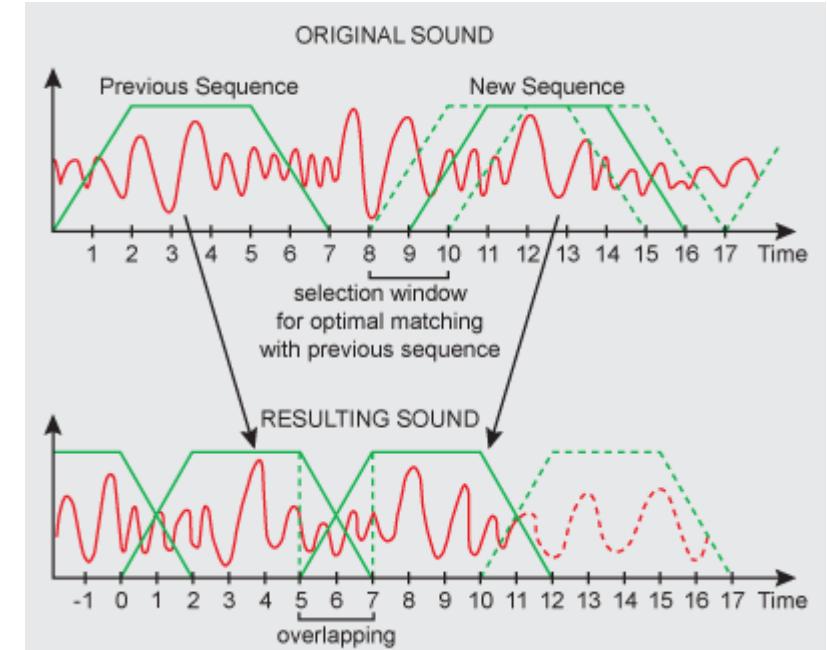
- This is what will stretch or compress the sample to the original length

### Pros:

- Fast to process
- Simple
- Ideal for tones

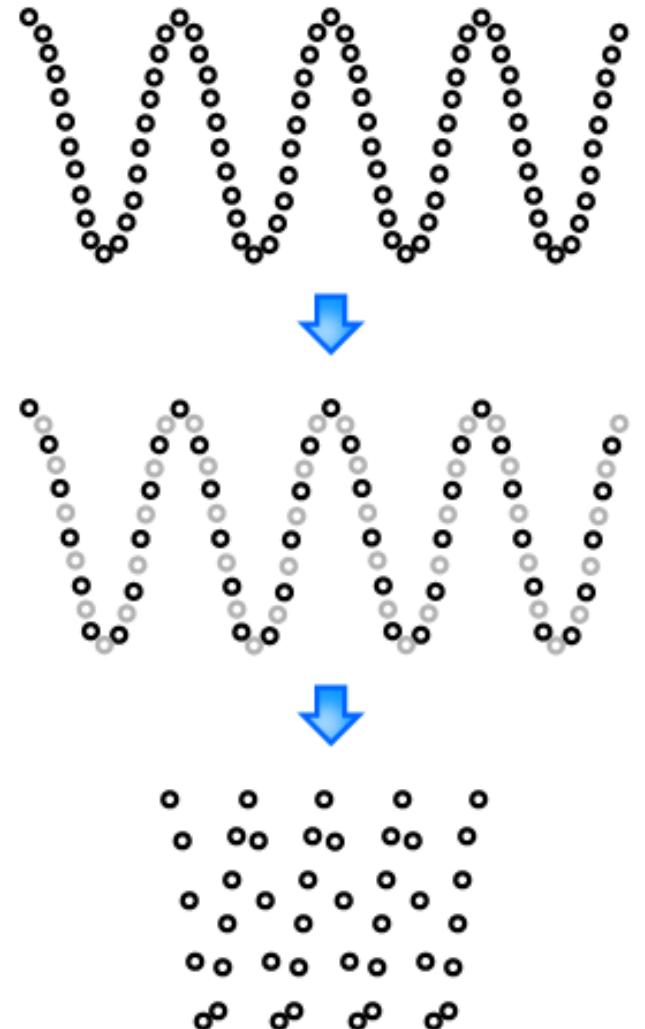
### Cons:

- Reverberation
- Lower Quality



# RESAMPLING

- Resampling changes pitch
- Stretching maintains original sample length
- Both are needed!
- Integer resampling is easy
  - 2x upsample, insert 0s every other sample
  - 2x downsample, remove every other sample
- Pitch shift requires non-integer resampling  
(stretch by 1.5, resample by 1/1.5)
  - Decimation: downsamples and filter
  - Interpolation: filter then upsample
  - FIR filter used to obey Nyquist (no aliasing!)



# TIME DOMAIN PITCH SHIFT (CONTINUED)

Delay Based (Doppler Effect)

Remember the Doppler effect (sirens)

Conceptual: Imagine capturing a moment when the siren pitch is shifted, and repeating it...perpetually

This nifty algorithm does this, and doesn't require resampling or FFT or filters!

Can actually be done with tape (Wendy Carlos)

Relies on two independent delays crossfading

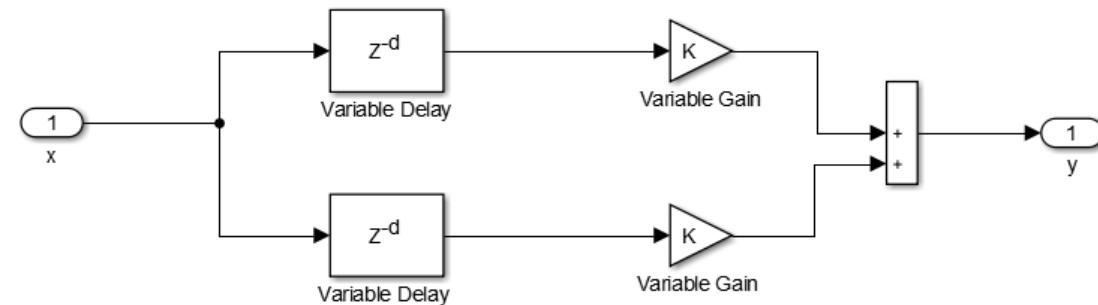
Best for our FPGA friend

## Pros:

- Fast to process
- No filtering
- No resampling
- Better than SOLA

## Cons:

- Unwanted random frequencies
- Confusing



# FREQUENCY DOMAIN PITCH SHIFT

## Phase Vocoder

Relies on STFT manipulation

### Steps

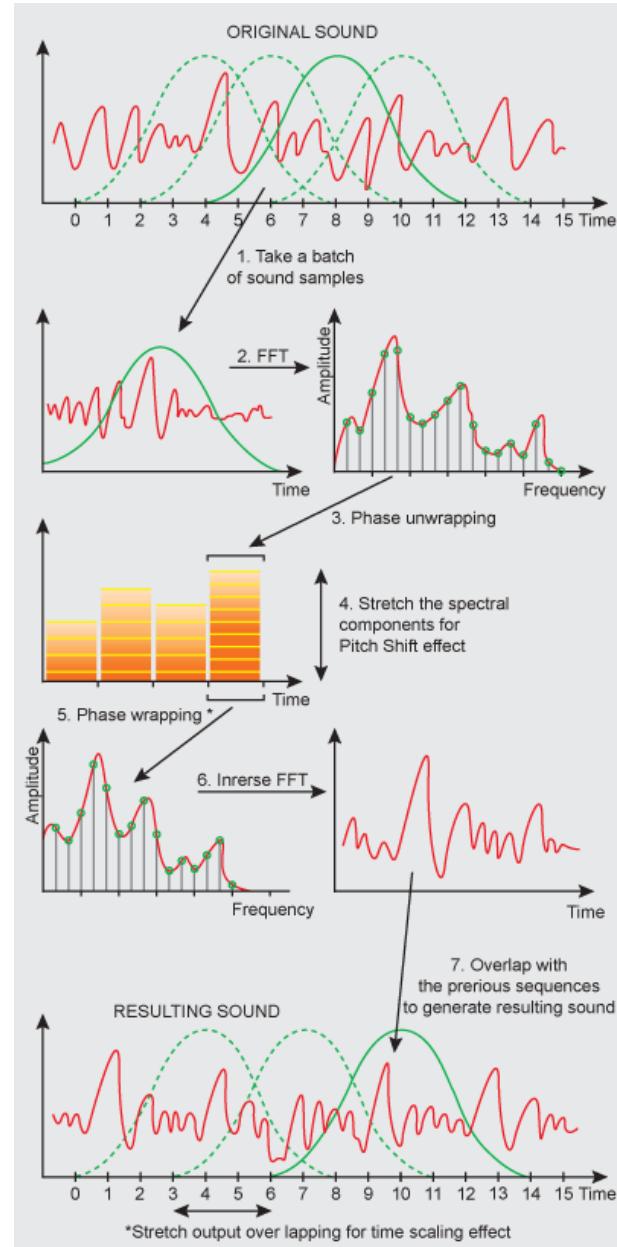
- Analysis (Both Domains)
  - Windowing (hanning window), then FFT
  - Overlap windows by X %
- Processing (Frequency Domain)
  - Stretch spectral content (lots of math here!)
  - Adjust phase to change pitch and transition between bins
- Synthesis (Time Domain)
  - IDFT, then windowing to smooth
  - Sample is time stretched
  - Resample to pitch shift

### Pros:

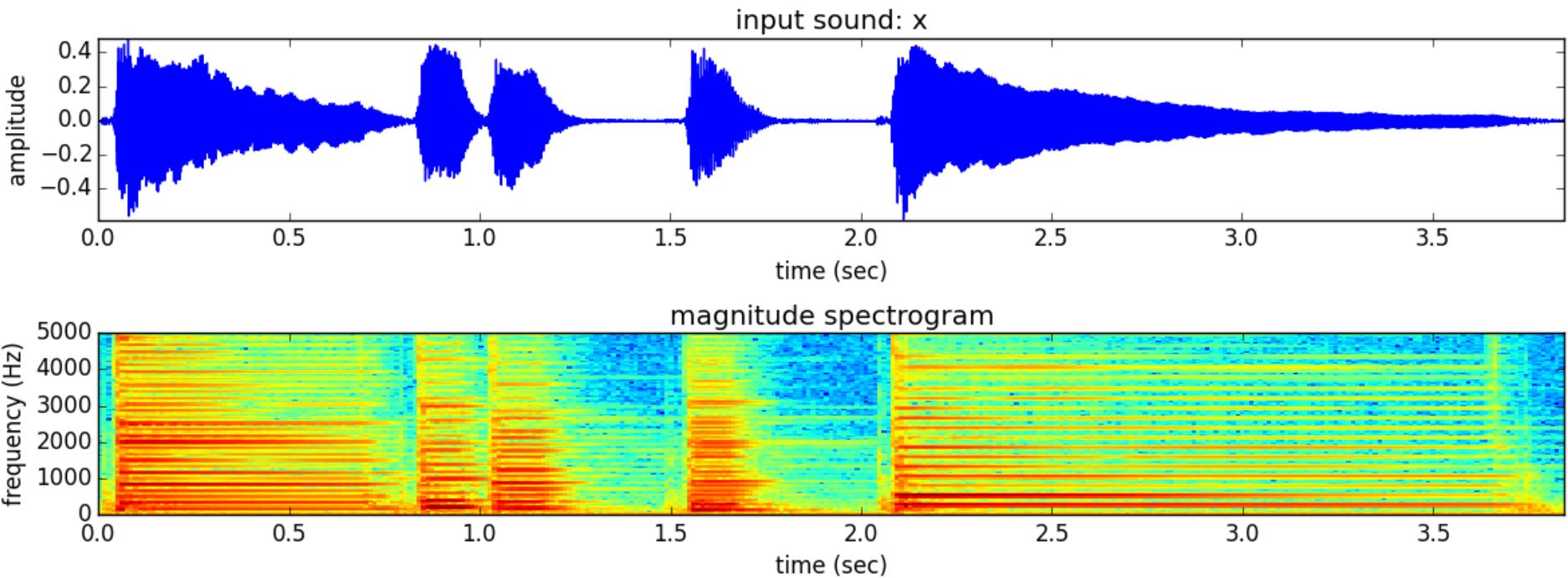
- Best quality
- Customizable
- Ideal for speech

### Cons:

- Several FFT + arctan
- Slow
- Complex



# STFT



- Piano above. Can see harmonics and decay. Shows, time, frequency and magnitude
- Combine several FFTs to create STFT

# CURRENT STATE

## Matlab Demo

- ✓ ▪ Phase Vocoder (wav file)
- ✓ ▪ Delay Based (wav file)
- ✓ ▪ Post processing analysis



SUCCESS



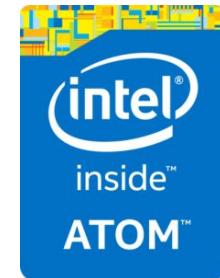
FAIL



IN PROGRESS

## C Demo (compiles on Atom Processor)

- ✓ ▪ Phase Vocoder (wav file)
- ✓ ▪ Phase Vocoder (real-time)
- ✗ ▪ PSOLA (wav file)
- ✓ ▪ Delay Based (wav file)

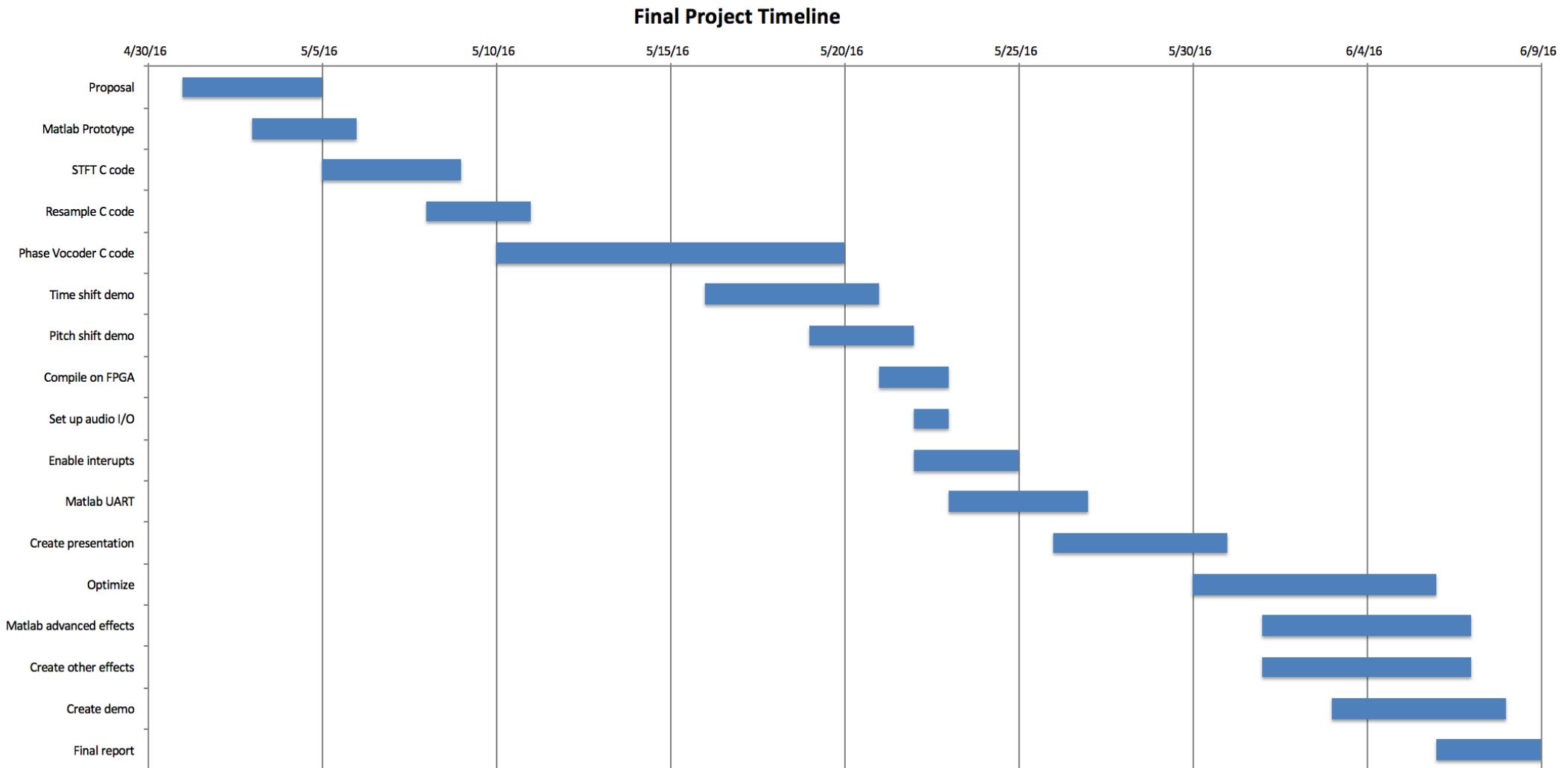


## Real-time FPGA

- ✗ ▪ Phase Vocoder
- ✗ ▪ PSOLA
- ⚠ ▪ Delay Based



# ORIGINAL TIMELINE....LOL IF ONLY



# TIMELINE

1. Research, decide to go with phase vocoder
2. Matlab phase vocoder demo
3. Write proposal
4. Phase vocoder C implementation (first wav file, then real-time on Atom)
5. Phase Vocoder FPGA attempt....lots of fail...algorithm takes too long to process
6. SOLA research...seems like it will be faster (*last weekend*)
7. SOLA C implementation (wav file)...resampling non integer is too complicated
8. Delay method research...faster and less complex
9. Delay Matlab Demo
10. Delay C implementation (*yesterday*)
11. Ping Pong buffer Research
12. Delay FPGA attempt (*work in progress*)
13. Make slides and present

# STRUGGLES

Algorithms are complicated

- Lots of math
- Complex and hard to debug
- Not really covered in class or labs

Not many C implementation examples on the internet

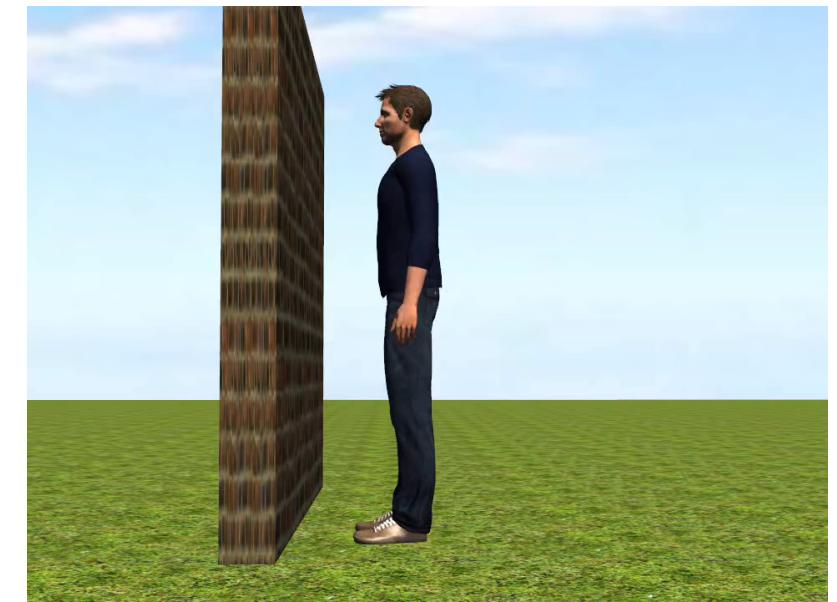
No way to know if it works until you write the whole algorithm

- We don't have much info on the hardware limitations

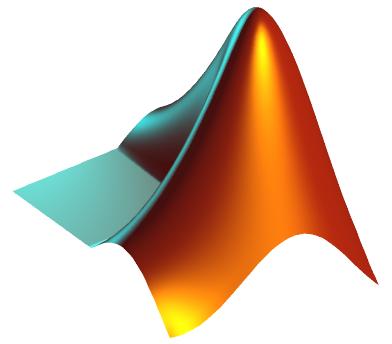
Try...Hope...Fail...Try...Hope...Fail...Try...Hope...Fail... →

Real-time processing

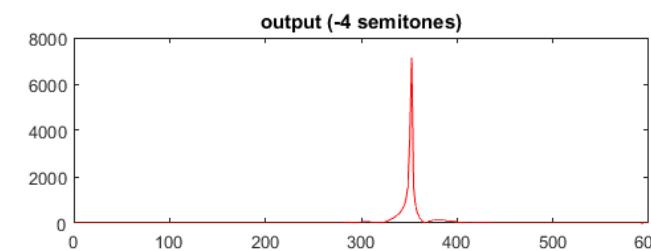
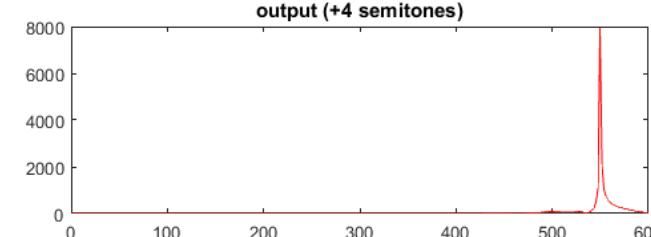
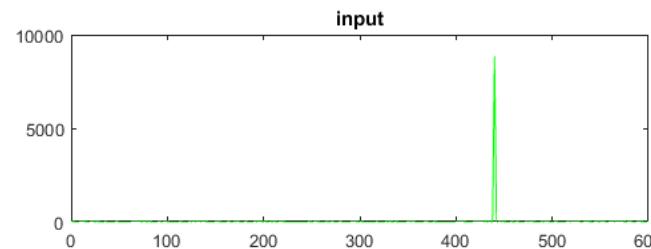
- Samples fill buffer before processing completes
- Solutions
  - use faster algorithm... ugh
  - find better hardware... no thanks
  - write lower level code... no thanks
- Scheduling (Ping Pong buffer)



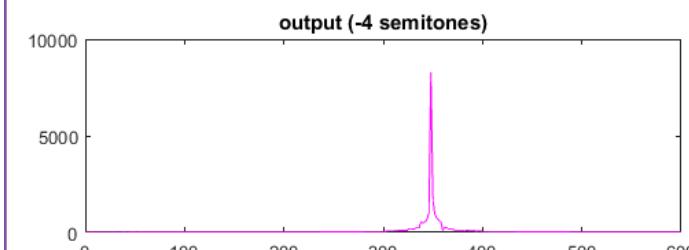
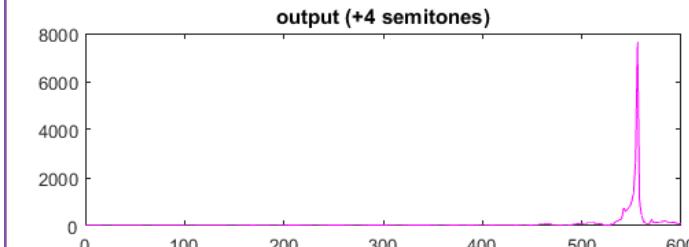
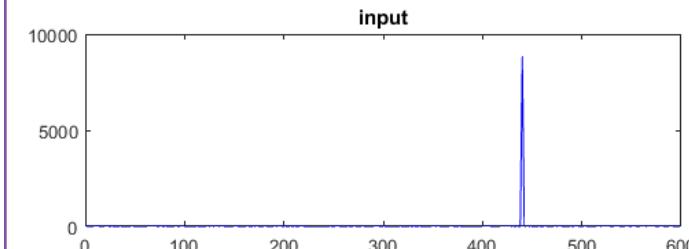
# MATLAB DEMO (440 Hz SINE INPUT)



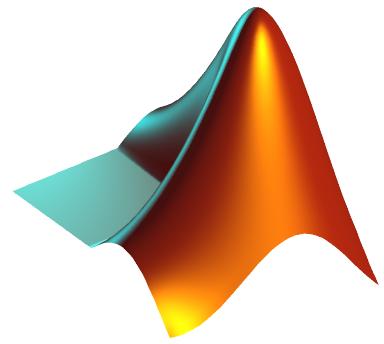
## Phase Vocoder



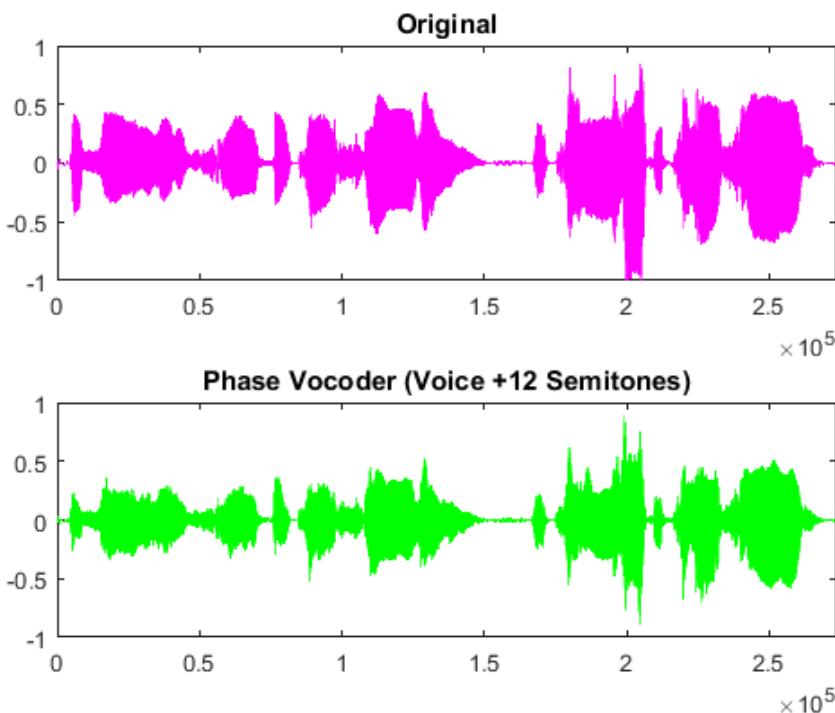
## Delay Method



# MATLAB DEMO (SINGING)



## Phase Vocoder



Original



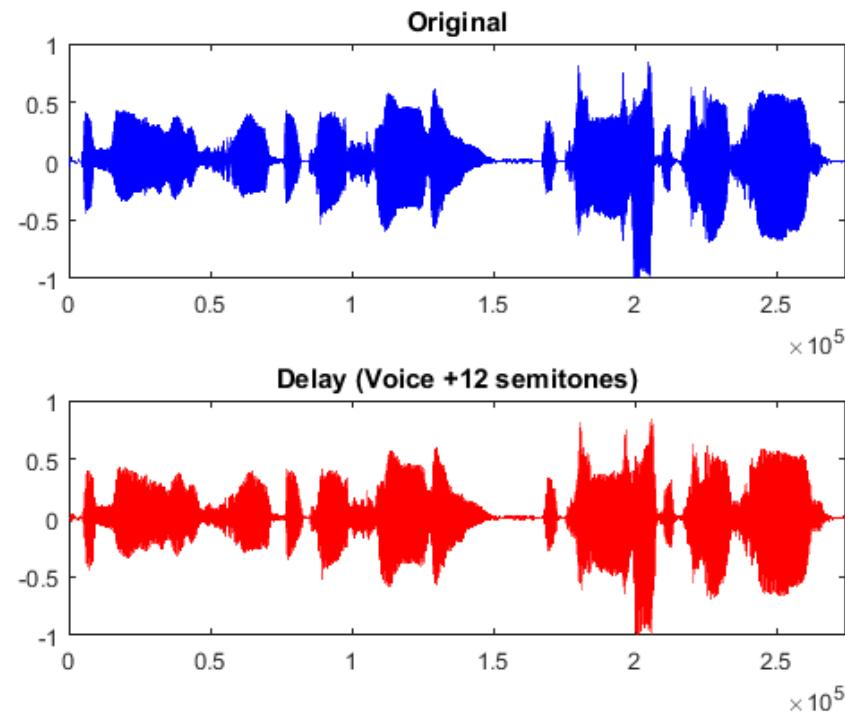
+12



-12



## Delay Method



Original



+12



-12



\* Note the shifted output is the same length

# C DEMOS



	Original	Phase Vocoder	Delay Based *		
Sample	0 semitones Factor: 0	+4 semitones Factor: 1.26	-4 semitones Factor: 0.79	+4 semitones Factor: 1.26	-4 semitones Factor: 0.79
Sine Wave 440 Hz					
Singing Voice					

\*Delay based semitone calculation is not correct

# CODE COMPARISON

## Phase Vocoder

- Matlab : ~180 lines
- C : ~450 lines

## Delay Method

- Matlab : ~150 lines
- C : ~ 300

## Delay C Code Snippet



```
//          buff a and b are circular
// for (m = 0; m < BUFFSIZE; m++) {
for (m = 0; m < buffer_size; m++) {

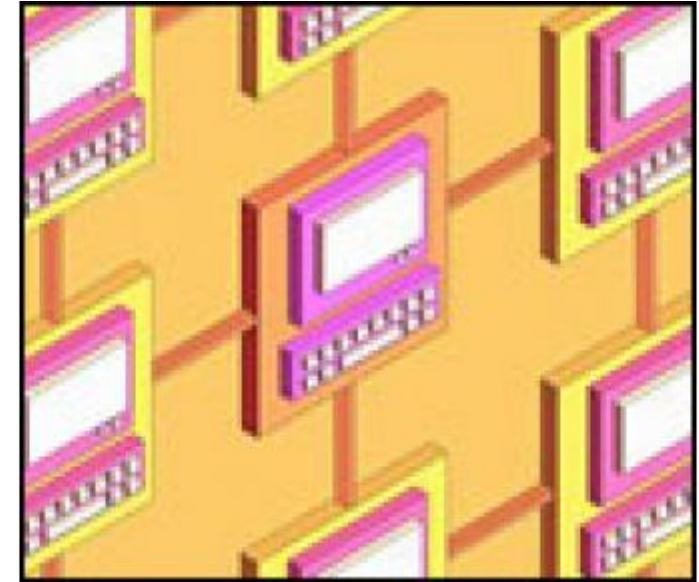
    if (((bufferAptr_float + 1) >= 0) & ((bufferAptr_float + 1) <= BUFFER_DEPTH - 1))
        bufferAptr2_float = bufferA[(int)bufferAptr_float + 1];
    else {
        if ((bufferAptr_float + 1) < 0)
            bufferAptr2_float = (BUFFER_DEPTH - 1) + (bufferAptr_float + 1);
        if ((bufferAptr_float + 1) > BUFFER_DEPTH - 1)
            bufferAptr2_float = (bufferAptr_float + 1) - (BUFFER_DEPTH - 1);
    }
    if (((bufferBptr_float + 1) >= 0) & ((bufferBptr_float + 1) <= BUFFER_DEPTH - 1))
        bufferBptr2_float = bufferB[(int)bufferBptr_float + 1];
    else {
        if ((bufferBptr_float + 2) < 0)
            bufferBptr2_float = (BUFFER_DEPTH - 1) + (bufferBptr_float + 1);
        if ((bufferBptr_float + 1) > BUFFER_DEPTH - 1)
            bufferBptr2_float = (bufferBptr_float + 1) - (BUFFER_DEPTH - 1);
    }

    bufferA[bufferABptr] = inbuf[m] + channel_feedback * bufferAptr2_float;
    bufferB[bufferABptr] = inbuf[m] + channel_feedback * bufferBptr2_float;
    //Compute Channel Outputs & system output Simplified model for now w/o interpolations.
    Aout = Ga * WET_MIX * bufferA[(int)bufferAptr_float] + DRY_MIX * inbuf[m];
    Bout = Gb * WET_MIX * bufferB[(int)bufferBptr_float] + DRY_MIX * inbuf[m];
    outbuf[m] = Aout + Bout;

    //calc delays
    if (bufferABptr > (int)bufferAptr_float)
        delaya = BUFFER_DEPTH - (bufferABptr - (int)bufferAptr_float);
    else
        delaya = (int)bufferAptr_float - bufferABptr;
    if (bufferABptr > (int)bufferBptr_float)
        delayb = BUFFER_DEPTH - (bufferABptr - (int)bufferBptr_float);
    else
        delayb = (int)bufferBptr_float - bufferABptr;
```

# REAL-TIME FPGA PROCESSING

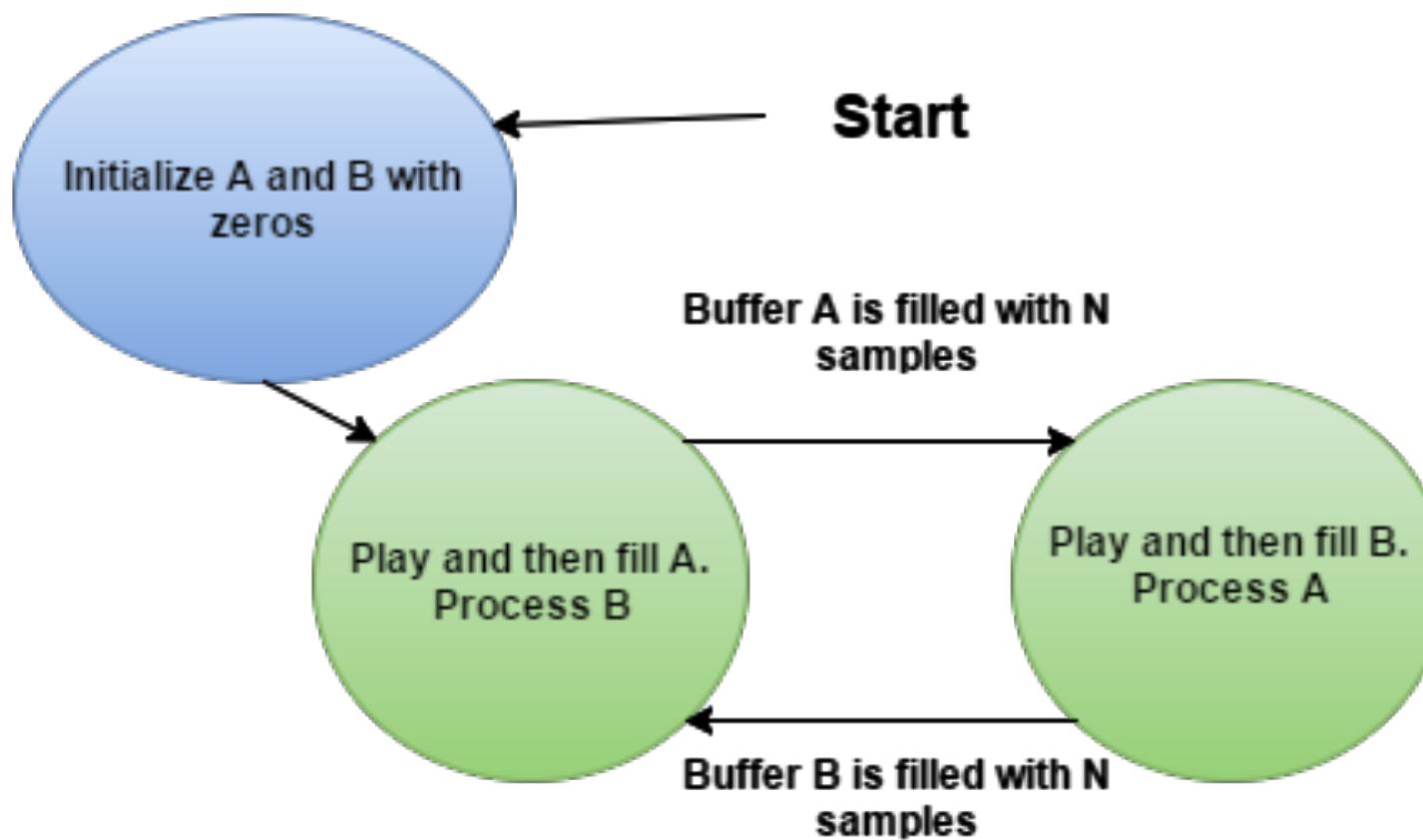
1. Fill input buffer
2. Process pitch shift for buffer
3. Play output while reading in new input



Heavy processing occurs in main, I/O sampling occurs in hardware interrupt

Note: this hinges on the fact that processing is faster than filling the buffer. This is not guaranteed... Ping Pong buffering helps

# PING PONG BUFFERING

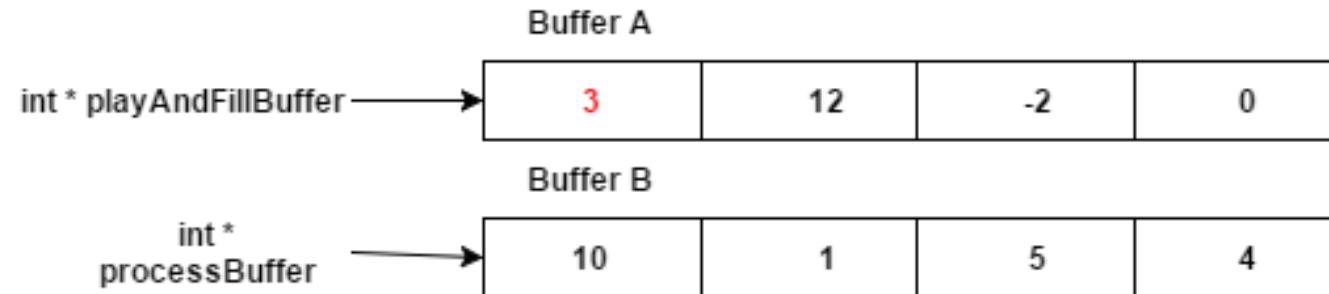


```
A: buffer PING
B: buffer PONG
int *processBuffer      = A
int *playAndFillBuffer = B
----- SCHEDULE -----
t=0 | Start:
| Initialize A and B with zeros
|
| A                                B
| Start process          Start play and fill
| .
| process done            .
| .
| play and fill done
|
| Swap ptr
|
| Start play and fill   Start process
| .
| .
| process done          .
| play and fill done
|
| Swap ptr
|
| Repeat
| .
| .
```

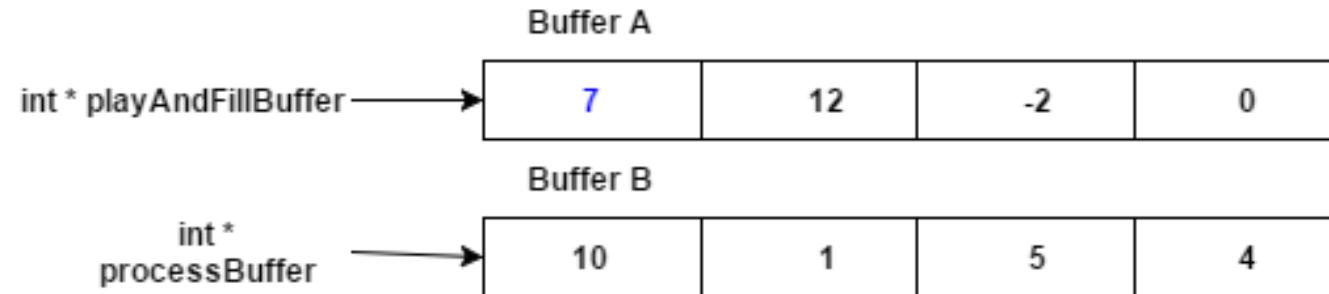
t=N V

# PING PONG BUFFERING – PLAY\_AND\_FILL

1. Read and play a sample  
at `playAndFillBuffer[i]`



2. Fill `playAndFillBuffer[i]`  
by overwriting old sample  
with the new sample



3. Increment `i` by 1

# QUESTIONS ?

