Music Synthesizer

Keelin Becker-Wheeler

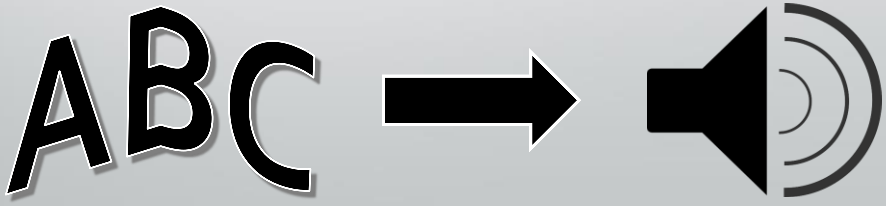
Motivation

Voice Synthesis / Text-to-Speech

i.e. **Producing sound in response to a stream of input.**

Applications in:

1. Communication / translation 
2. Artificial intelligence / human-machine interactions
3. Synthesized music



Problem

Hardware acceleration of audio synthesis, in response to streamed input.

Important in areas where pre-recording is not practical or possible.

- Live + fast translation.

→ I will show methods for responding to a live-stream of input.

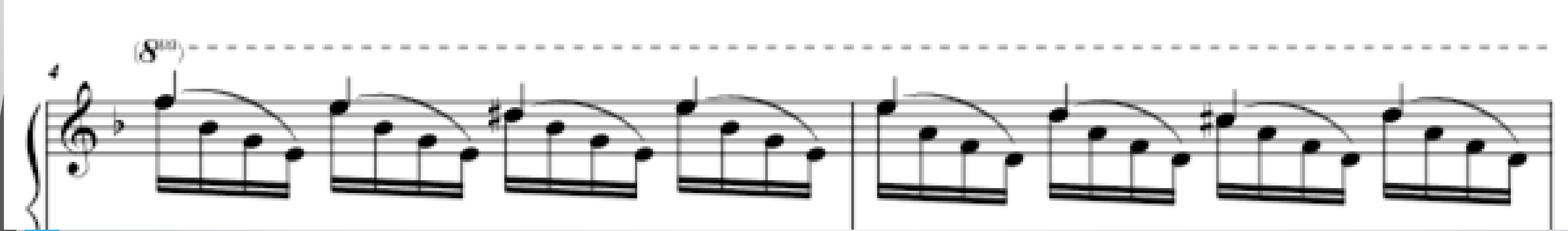
Many approaches I found don't seem to synthesize voice on the hardware.
Disregarding proprietary techniques, i.e. approaches I could not find sources/information for.

- Concatenative techniques that splice together pre-existing sound.
- Modification techniques that add effects onto a voice input.

→ I will show methods for synthesizing new sounds on the FPGA hardware.

Goal

- Given a sequence of musical notes.. Play them!



F6Bb5G5E5 E6Bb5G5E5 D#6Bb5G5E5 E6Bb5G5E5 E6A5F5D5 D6A5F5D5 C#6A5F5D5 D6A5F5D5

- Analogous to text-to-speech (synthesizing audio in response to streamed input), but more obtainable in time frame and without extensive knowledge on topics.

How are notes represented?

MIDI Note Values

Rules:

- Octaves change at C notes. Start at -1 octave.
- 12 base notes in octave: C, C#, D, D#, E, F, F#, G, G#, A, A#, B
- Each next base note adds 1 to the value.
- Additional sharps(#) and flats(b) add and subtract 1 from value.

7-bit Range:

- C-1 = value 0
- G9 = value 127

Note	-1	0	1	2	3	4	5	6	7	8	9
C	0	12	24	36	48	60	72	84	96	108	120
C#	1	13	25	37	49	61	73	85	97	109	121
D	2	14	26	38	50	62	74	86	98	110	122
D#	3	15	27	39	51	63	75	87	99	111	123
E	4	16	28	40	52	64	76	88	100	112	124
F	5	17	29	41	53	65	77	89	101	113	125
F#	6	18	30	42	54	66	78	90	102	114	126
G	7	19	31	43	55	67	79	91	103	115	127
G#	8	20	32	44	56	68	80	92	104	116	
A	9	21	33	45	57	69	81	93	105	117	
A#	10	22	34	46	58	70	82	94	106	118	
B	11	23	35	47	59	71	83	95	107	119	

MIDI Note Frequencies

- Given frequency of one note, can find frequency of any other note.

Based on standard point:

- A4 = concert pitch = 440Hz

Rules:

- Each octave scales frequency up or down by 2. (A3 = 220Hz, A5=880Hz)
- Each next note in octave scales evenly spaced within exponential double.

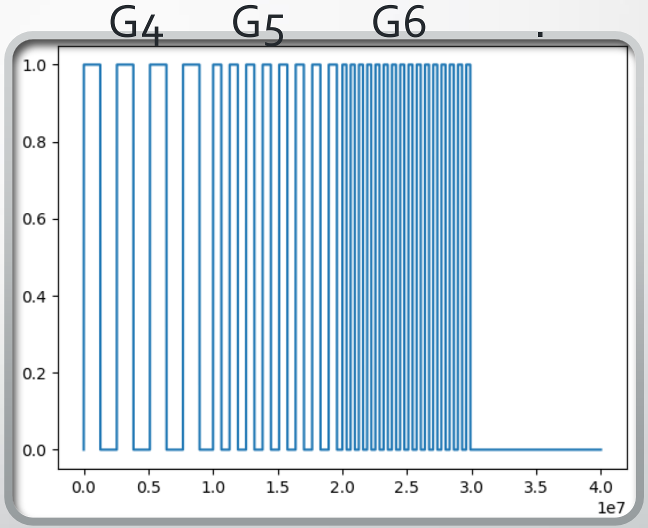
$$Freq(value) = 440Hz * 2^{(value\ -\ 69)\ /\ 12.0}$$

PWM Waveform Generation

- Audio port interface on selected FPGA written as LOW or HIGH.

Repeat for a given wave frequency:

1. Write HIGH.
2. Wait half wave period.
3. Write LOW.
4. Wait half wave period.

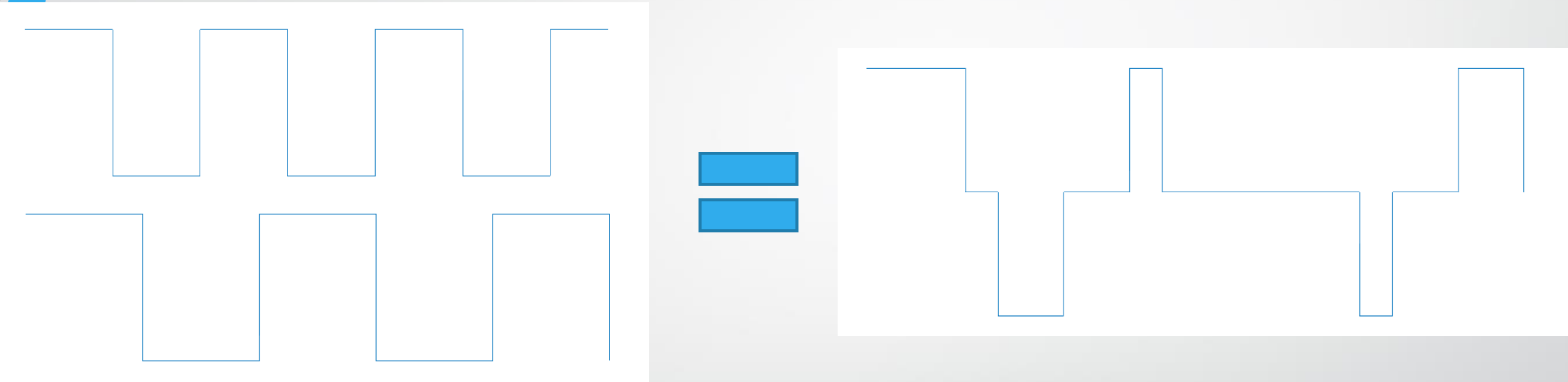


- FPGA clock at 100MHz
- Half period clock ticks for a given frequency are: 100MHZ / (2*freq)

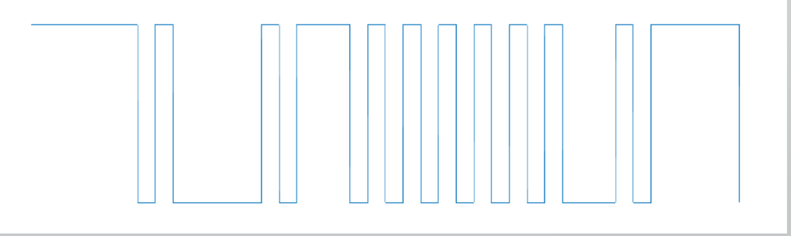
How to play multiple notes?

i.e. analogous to constructing complicated speech sounds by combining simpler waveforms

PWM Chord Generation



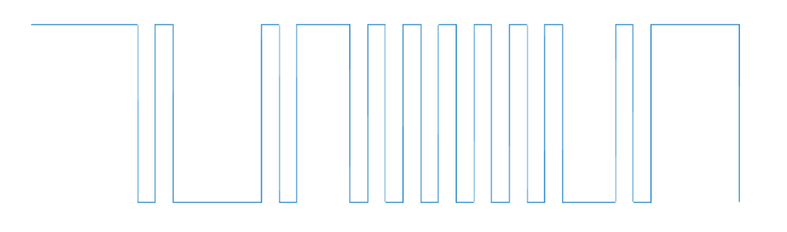
1. Add simultaneous waveforms.
2. Apply duty cycle to get values between 0 and 1.

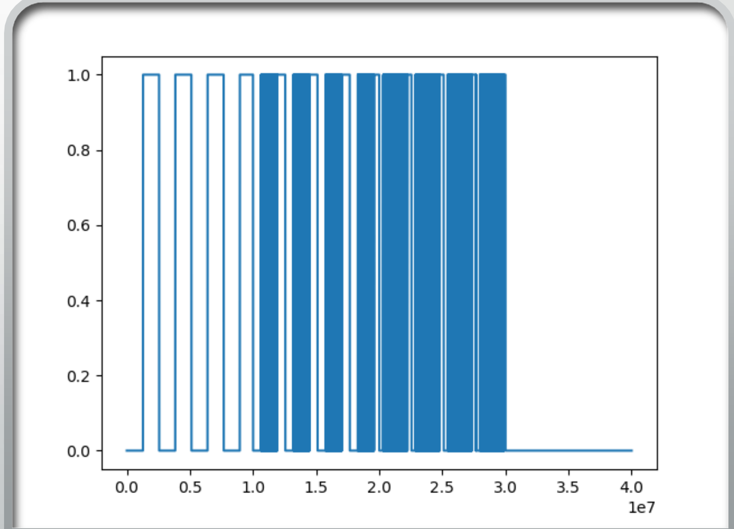


PWM Chord Generation

Concern:

- Duty cycle is ambiguous with a higher frequency note!





Solution:

- Use a high sampling rate, taking advantage of high frequency of FPGA.

- The high rate of change should not conflict with possible note frequencies.

G4(G4G5)(G4G5G6).

Can we increase quality of sound?

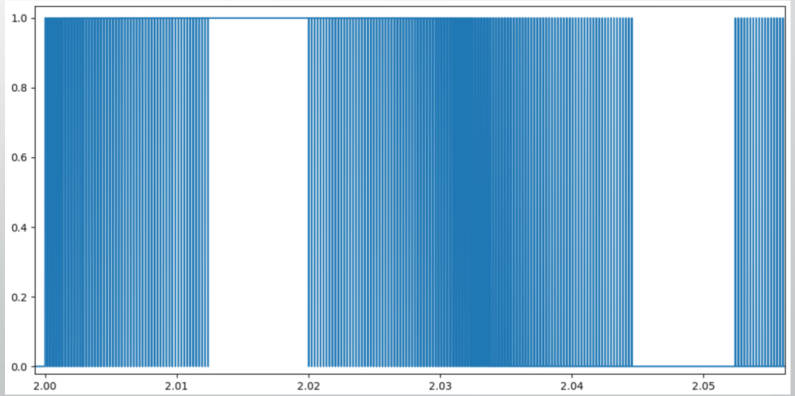
i.e. using custom waveforms in synthesis for more complicated sound components

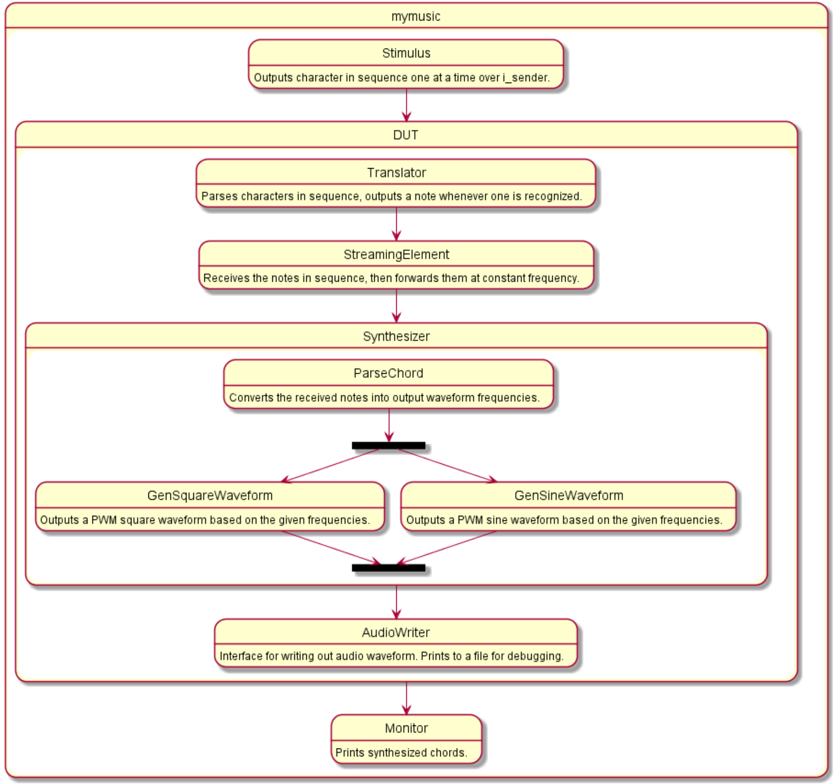
Sinusoidal Generation

1. Store one period of a sine wave.
2. Scan wave at different speeds to get target frequencies.

- 4096 samples in sine table, each sample 4-bits.
- Ticks to wait before incrementing position in table: 100MHz / (4096*freq)

Value from sine lookup table used to apply duty cycle in output PWM.





Model+ Simulation

- **Basic model simulated in SpecC.**
- **After verified, ported to other languages.**
  - **CPU -> Python**
  - **FPGA -> VHDL**

Implementation

CPU: Intel Core i7-8550U

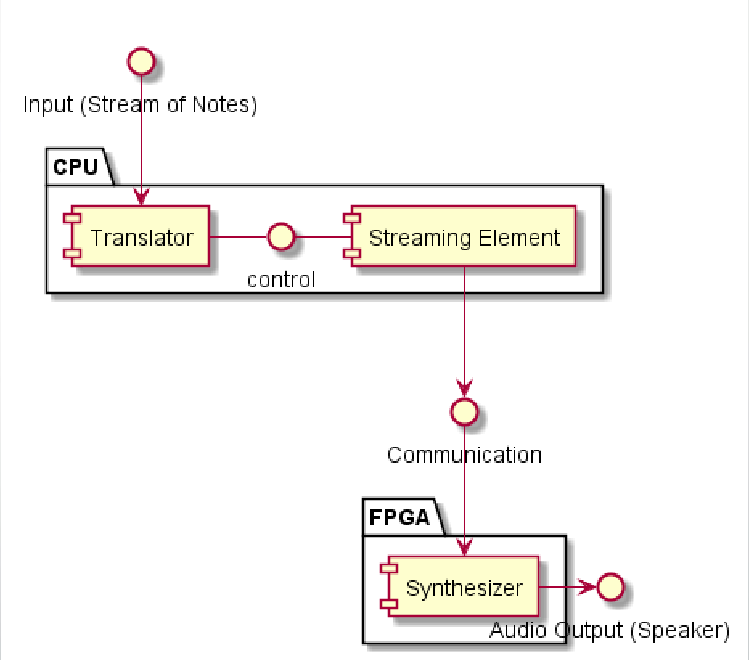
- Laptop
- Python

Communication:

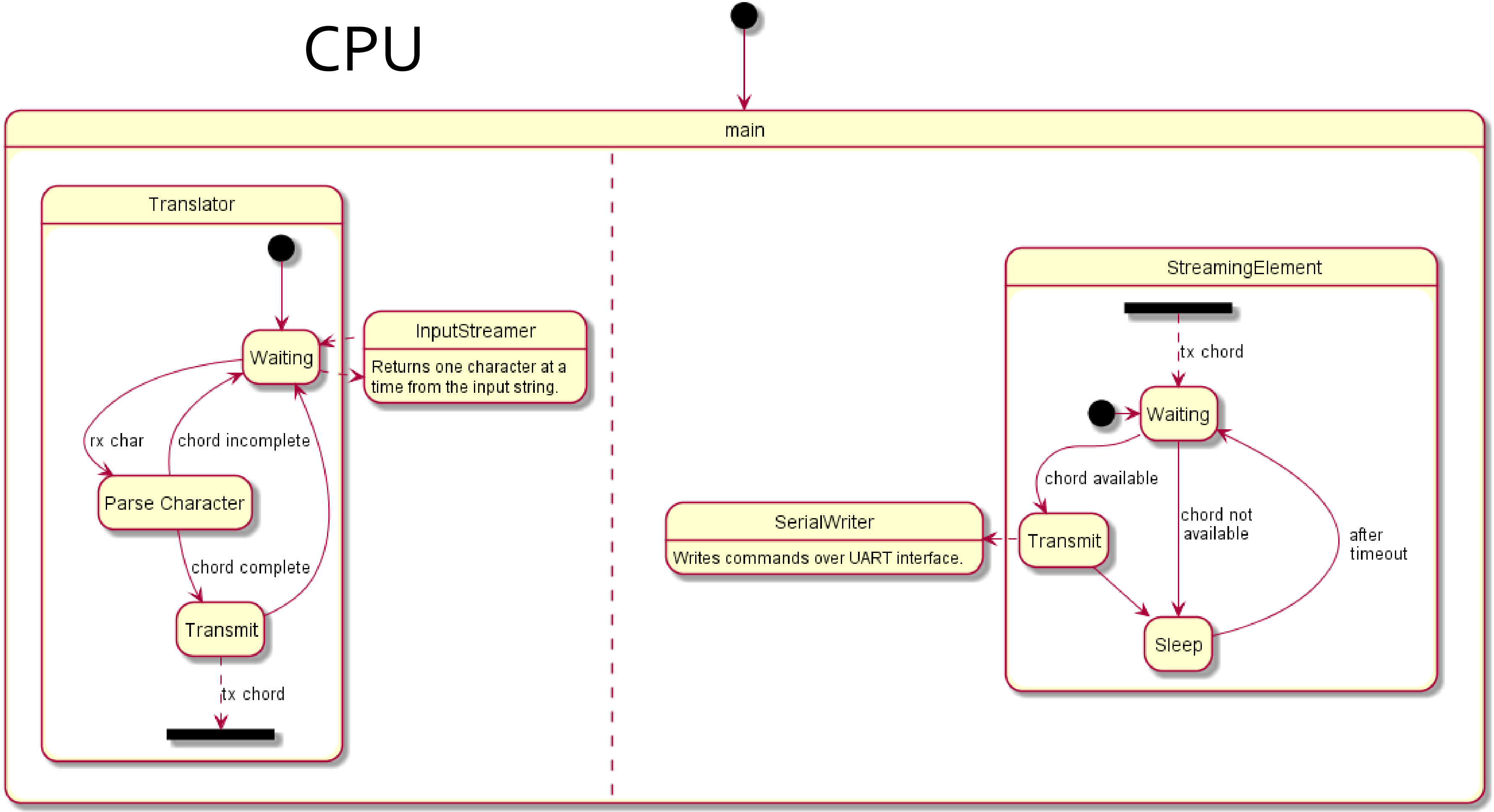
- UART

FPGA: Xilinx Artix-7

- Nexys 4 DDR Dev Board
- VHDL

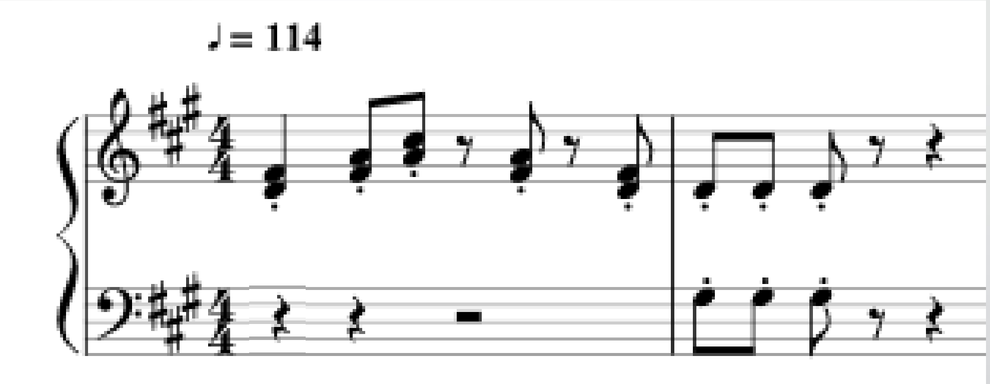


CPU


main
Translator
Waiting
rx char
chord incomplete
Parse Character
chord complete
Transmit
tx chord
InputStreamer
Returns one character at a time from the input string.
StreamingElement
tx chord
Waiting
chord available
Transmit
chord not available
after timeout
Sleep
SerialWriter
Writes commands over UART interface.

Translator

- .   = pause
- \-   = repeat
- ( )   = begin/end chord
- b   = flatten note
- #   = sharpen note
- 0-9   = octave
- A-G = note

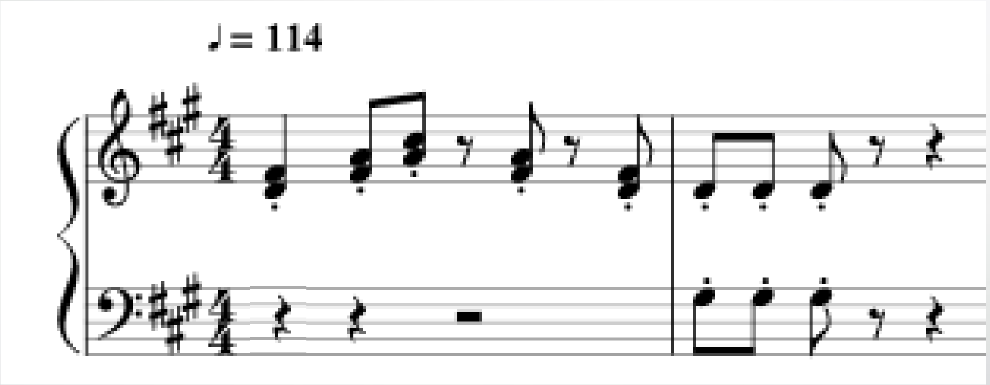


(D4F#4)--.(F#4A4).(A4C#5)...(F#4A4)...(D4F#4).(D4G#3).(D4G#3).(D4G#3).......

- Parses characters in a sequence. Outputs a chord whenever one is finished.
  e.g. "("={-1,-1}   "D"={14,-1}   "4"={62,-1}   "F"={62, 17}   "#"={62, 18}   "4"={62, 66}   ")"=done

Streaming Element

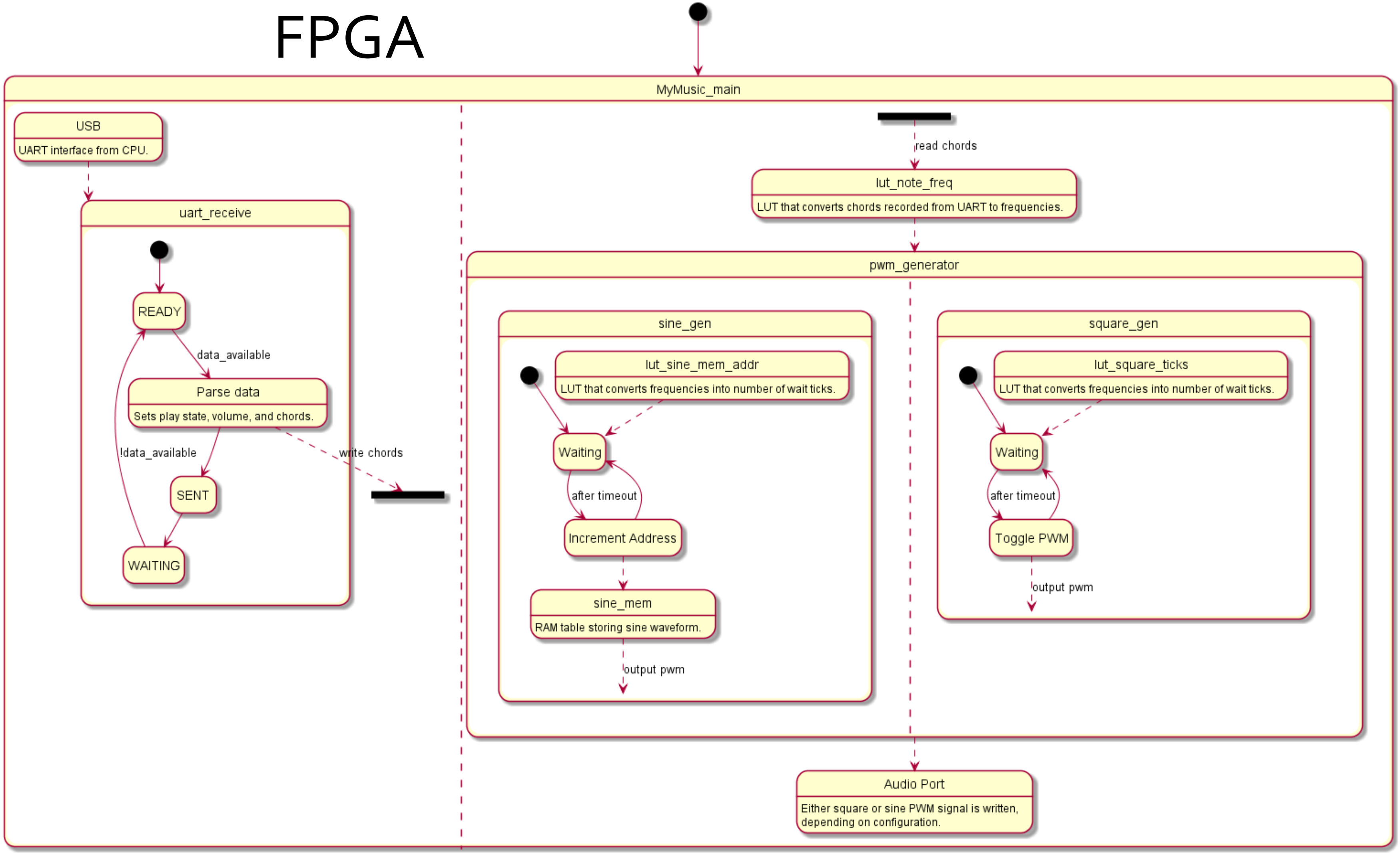
- Receives notes in sequence, forwards them at constant frequency.



(D4F#4)--.(F#4A4).(A4C#5)...(F#4A4)...(D4F#4).(D4G#3).(D4G#3).(D4G#3).......

- 114 beats per minute: Wait 60/114 sec. between each transmitted chord.

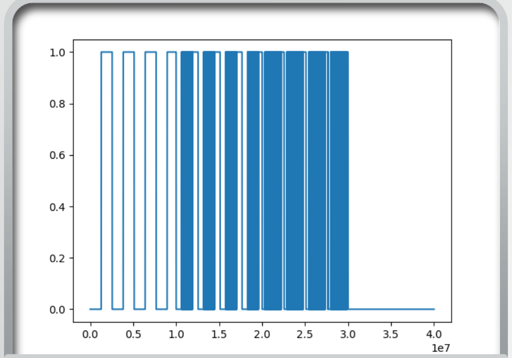
FPGA

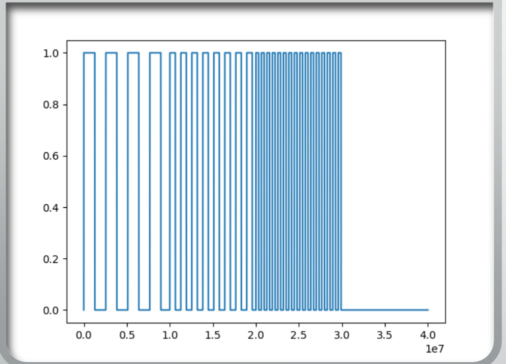

MyMusic_main
USB
UART interface from CPU.
uart_receive
READY
data_available
Parse data
Sets play state, volume, and chords.
!data_available
write chords
SENT
WAITING
read chords
lut_note_freq
LUT that converts chords recorded from UART to frequencies.
pwm_generator
sine_gen
lut_sine_mem_addr
LUT that converts frequencies into number of wait ticks.
Waiting
after timeout
Increment Address
sine_mem
RAM table storing sine waveform.
output pwm
square_gen
lut_square_ticks
LUT that converts frequencies into number of wait ticks.
Waiting
after timeout
Toggle PWM
output pwm
Audio Port
Either square or sine PWM signal is written, depending on configuration.

Results

- Receives correct notes at correct times.
- Writes correct waveform output.

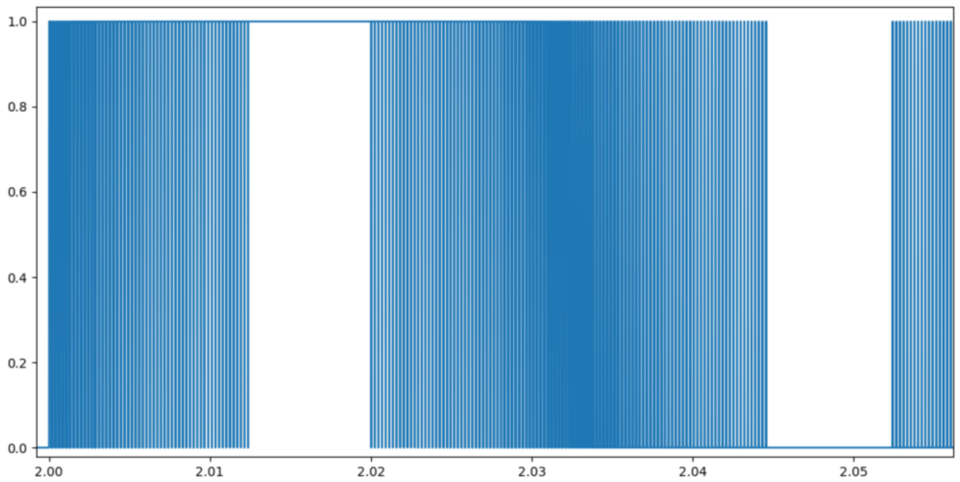
G4(G4G5)(G4G5G6).





G4G5G6.

Square Waves



G5

Sine Wave

Methodology: Pros and Cons

1. Simulated on SpecC.
2. Implemented components in other languages (Python+VHDL).

Pros:

- Quick to iterate and verify functionality (vs 11min. compile time for FPGA).
- Easier simulation in SpecC than Vivado.
- Easier to complete un-abstracted implementation in other languages.

Cons:

- Simulation abstracted from implementation.
- Translation of code between languages.

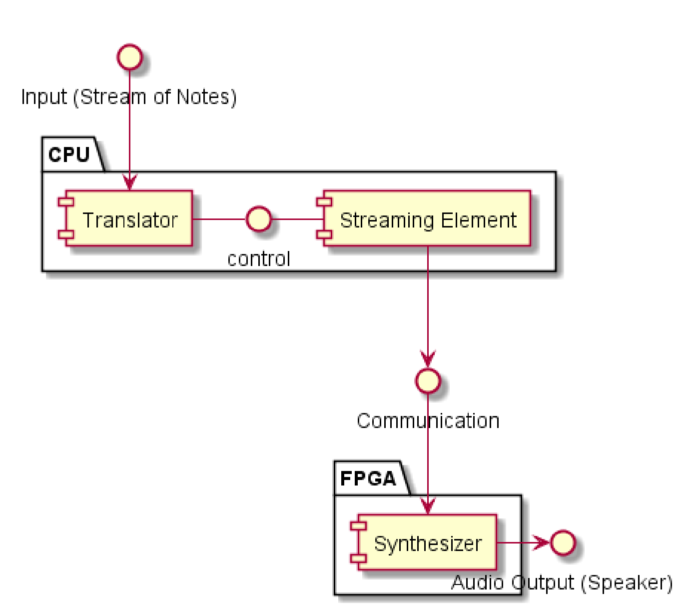
Solution: Pros and Cons

Pros:

- "No limit" on length of input sequence.
- Inherent parallelism in synthesis.
- Acceleration of FPGA.

Cons:

- Variable communication timing due to OS-scheduling / CPU contention.



Findings

- Dedicated computing resources important in timing critical applications.

- Sound synthesis can be complex.

In all, I have successfully showed:

- Example system that synthesizes sounds in response to streamed input.
- Methods for constructing complicated sounds on FPGA by:
  - Concatenating simpler sounds.
  - Referencing a waveform table.

Questions?