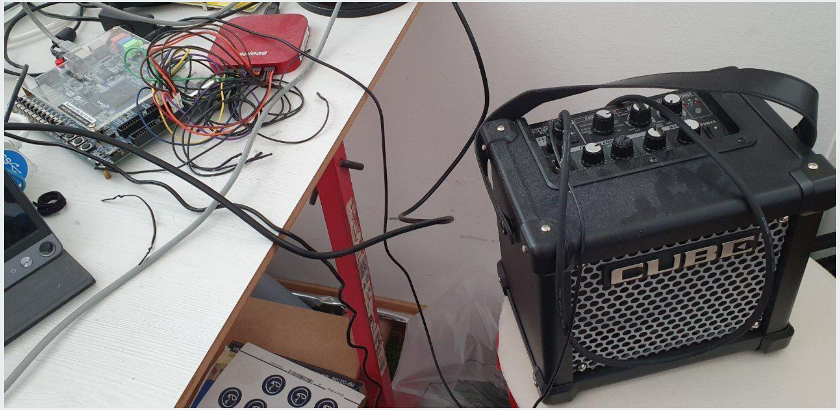
FPGA Sound Synthesizer



CS-476 Real-time Embedded Systems
Alexandre CHAU & Loïc DROZ

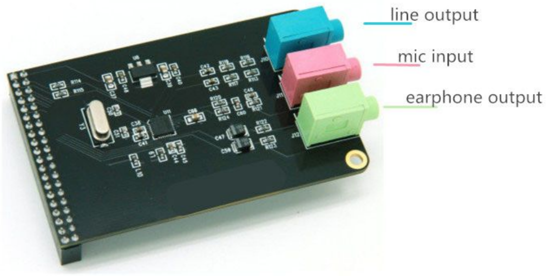
What is it?

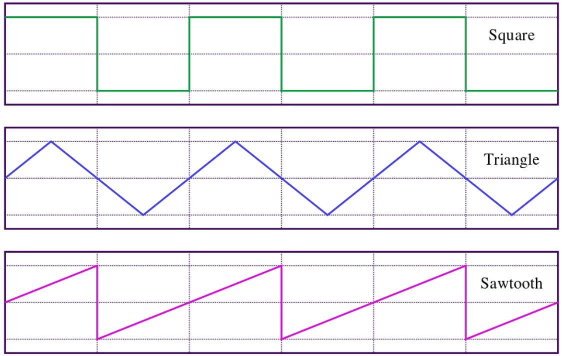
A digital sound synthesizer platform and associated full-stack toolchain built on the FPGA core of the DE1-SoC with 4 components:

Sound synthesis core	Meta scalable code generation	Low-level music abstraction library	MIDI parser tool
VHDL	Python	C	NodeJS

FPGA sound synthesis core

- Real-time sound generation with PCM samples at 96 KHz rate, 32-bit depth through WM8731
- 16 oscillators (up to 16-notes polyphony), mixer
- 3 wave types per oscillator (saw, square, and triangle)
- playback controls (mute, restart, next song)
- volume controls (-/+, default), wave type controls
- interprets a subset of the MIDI protocol (note code and event type) in real-time
- LED VU-meter (amplitude average estimation)



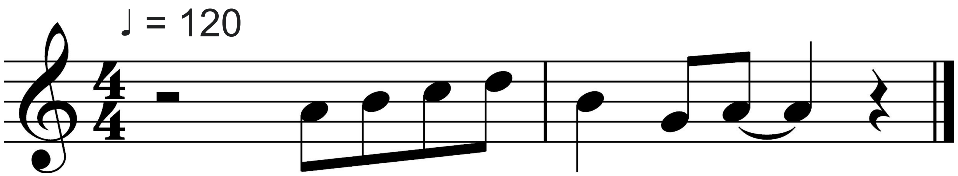


VHDL Code Generation in Python

- Python scripts to generate VHDL code dynamically
- Can specify DAC frequency and depth, number of oscillators, number of LEDs (VU-meter)
- Automatically recomputes all VHDL values that depend on these parameters such as vector widths, periodicity of waves, ...
- Saves time and prevents programmer from forgetting to update some values and thus introducing bugs, meta-maths on code
- Example: we easily switched from 48KHz, 16-bit to 96KHz, 32-bit to avoid frequency distortion for high notes

C Music Abstraction Library

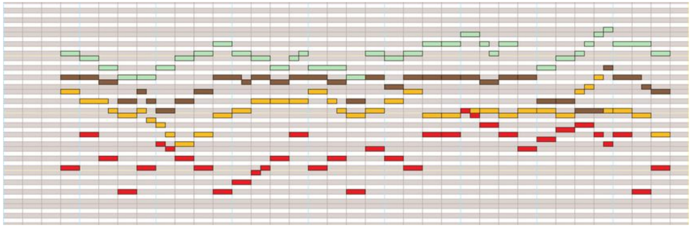
- C library of data structures and macros to easily represent and compose music programmatically
- Derived from MIDI protocol: event codes, note codes, delta times (inverted)
- CPU reads this data and pushes the appropriate message to the synthesizer device




```c
// the lick
#define THE_LICK_LENGTH 14
struct note
the_lick[THE_LICK_LENGTH] = {
    {NOTE_START | A4, 250},
    {NOTE_STOP | A4, 0},
    {NOTE_START | B4, 250},
    {NOTE_STOP | B4, 0},
    {NOTE_START | C5, 250},
    {NOTE_STOP | C5, 0},
    {NOTE_START | D5, 250},
    {NOTE_STOP | D5, 0},
    {NOTE_START | B4, 500},
    {NOTE_STOP | B4, 0},
    {NOTE_START | G4, 250},
    {NOTE_STOP | G4, 0},
    {NOTE_START | A4, 750},
    {NOTE_STOP | A4, 0},
}
```

MIDI Parser

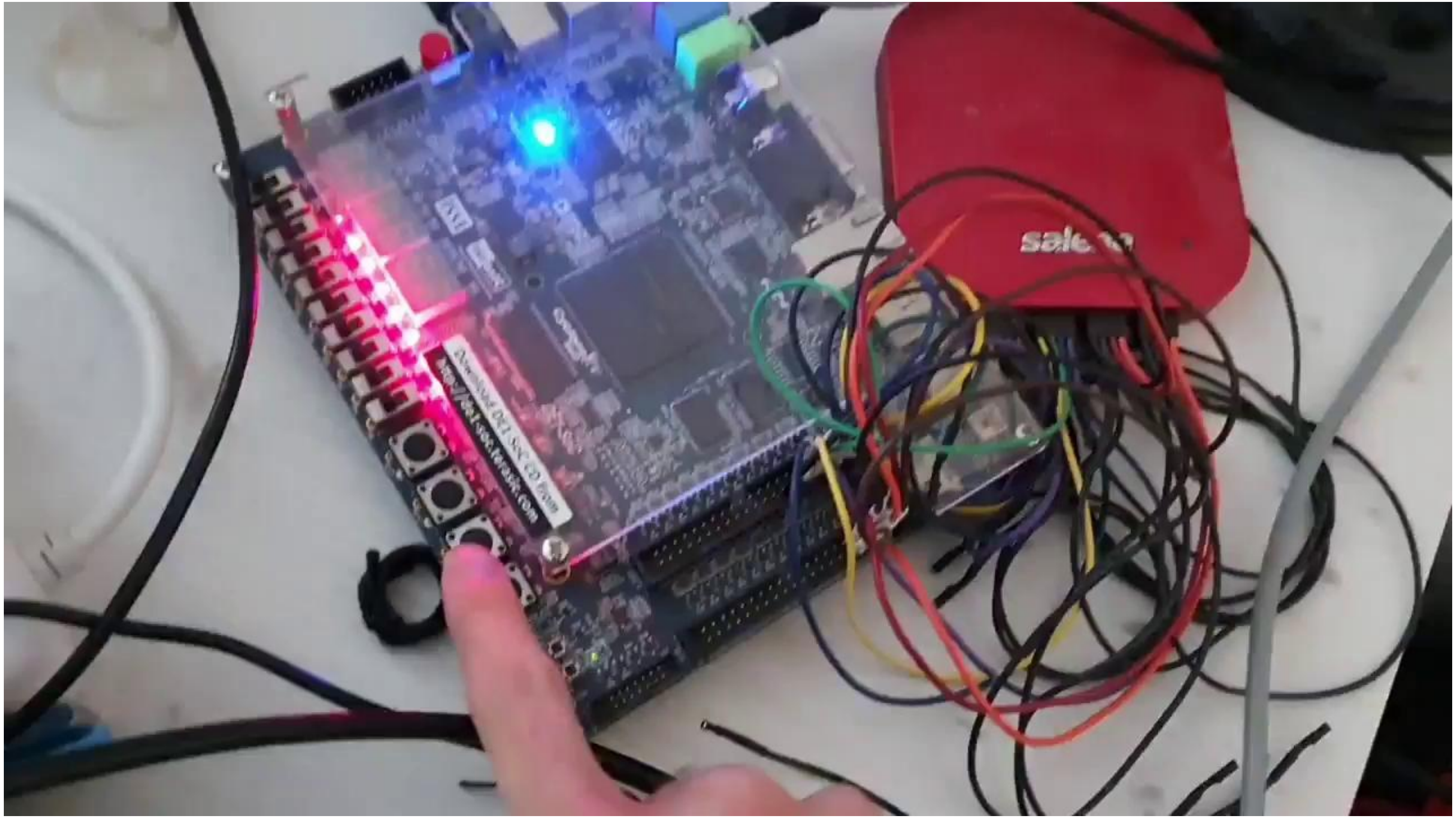
- CLI program to extract notes from a MIDI file and assemble them into our C music abstraction library format
- Generated C code can be used as is
- Implemented in functional NodeJS, ability to transpose music and scale duration of notes




```
$ node parse music.midi name -1 0.5
```

Demo

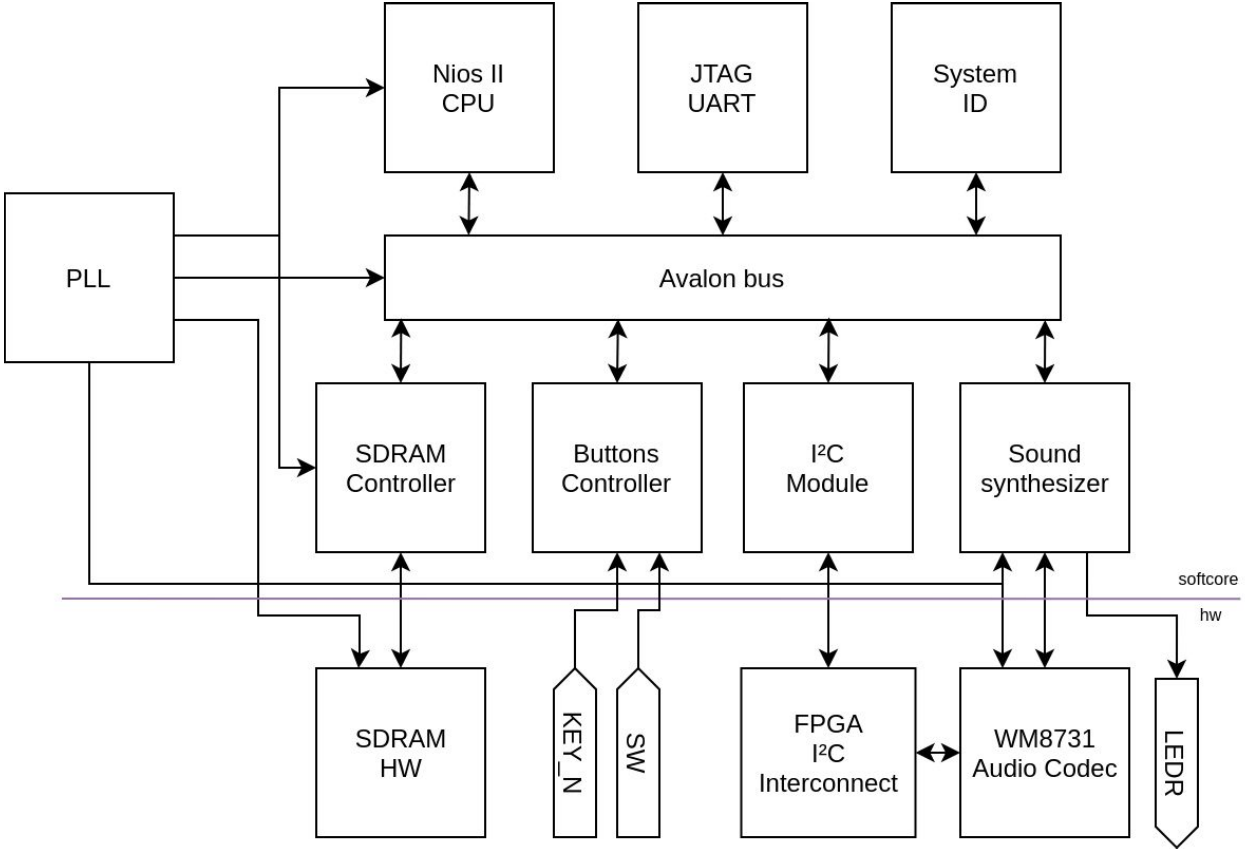
Sound synthesizer



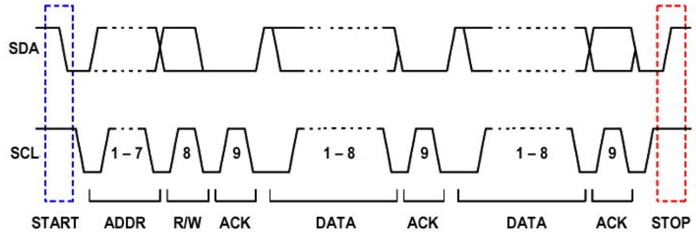
Backup video demo - sound synthesizer

System implementation

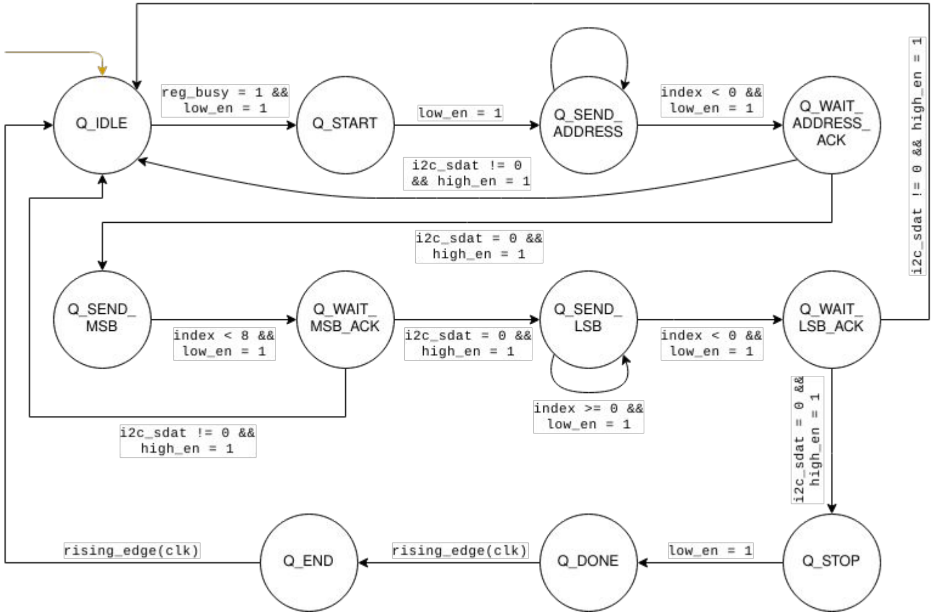
Softcore hardware architecture

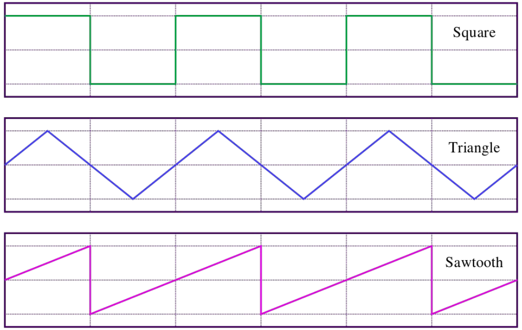


I^{2}C: Configuring the WM8731



- Avalon slave component to interface the I^{2}C interconnect of the board
- Generates SCLK at 200 KHz, holds SCLK and SDAT high when inactive
- Writes the 7-bit I^{2}C address of WM8731 (0x34, passed by CPU through register)
- Reads and writes I^{2}C messages / acks through SDAT
- Implemented as FSM that sends address and data, and waits for acks



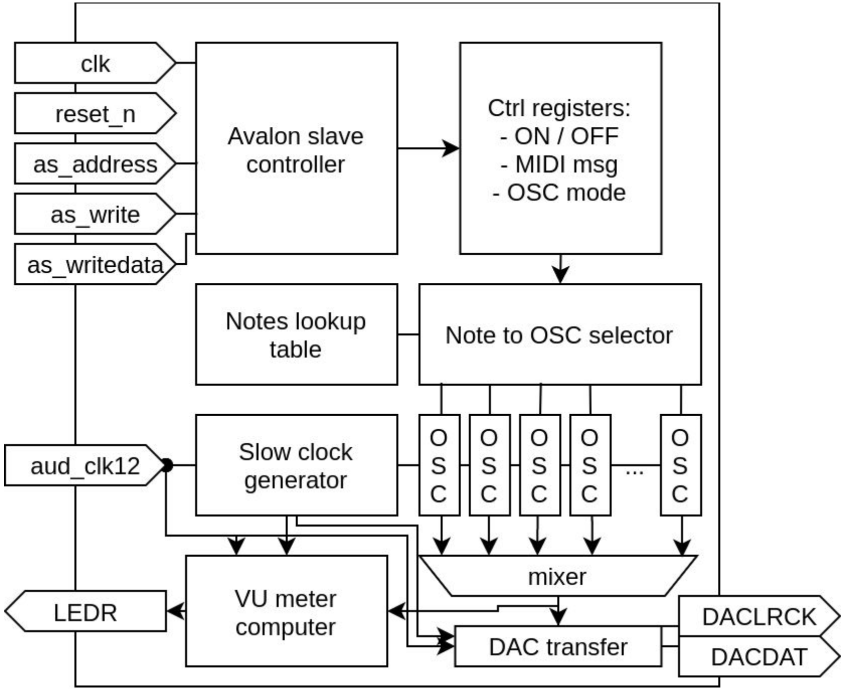


Sound synthesis: oscillators

- Generates signed PCM samples at the DAC's frequency
- Inputs: note period and linear step, min and max scaled from 1 / # osc (at generation)
- Selectable output wave type:
  - Square wave: counts until half of period and alternates between min and max
  - Saw tooth wave: increments by linear step every sample, resets to min at period
  - Triangle wave: increments by 2 linear steps every sample until half of period, then decrements by the same amount
- Sample accurate note periods and linear steps implemented as lookup table
- Ex: A4 440Hz is MIDI code 0x45, mapped to period of 218 samples at 96 KHz, linear sample difference is 1230329 with 32-bit depth

Sound synthesis slave

- Wraps the oscillators
- Responds to commands sent by Avalon master (CPU), such as notes start / stop
- Selects first free oscillator to play the new note and assigns note period and step
- Oscillators amplitude is mathematically capped at max ± DAC value / # instances
- Mixer simply adds oscillators together
- DAC transfer reads ands pushes samples to WM8731 at DAC frequency (96 KHz)



VU meter

- Real-time **volume indicator** on board LEDs
- A VU meter typically does a moving average => need to store n samples
- In our case, volume computed using **exponential smoothing** of the **absolute value** of the audio signal: $s_t = \alpha x_t + (1 - \alpha) s_{t-1}$ where **s** = volume, **x** = signal, **α** = decay factor
- Yields an **approximation** of a moving average
  - 😄 Memory: only 1 additional register required! (compared to moving avg.)
  - 😄 Very **simple** implementation
  - 😄 No multiplications / multi-cycle divisions: implemented using **shifts** and **subtractions**
  - 😒 Decay factor must be in the form **$1 / 2^n$**
  - 😒 Integer divisions => result often **biased**
  - 😒 When music stops, smoothed value does not converge exactly to 0
- Smoothed value is then converted to a **unary-like representation** for LED display

NIOS II: Note events playback

- Main program runs on a NIOS II CPU
- Music stored in array in global variables, using the music abstraction library
- Stores state of volume, current song, song cursor
- Sets up WM8731 through I^2C device
- Register ISR for buttons control IRQs (more later), waits for user input
- Reads MIDI events from SDRAM memory and loops through them, sending commands to synth device, waits in-between events

Buttons controller and usage

- Interface for **buttons 3 to 1** (0 is used for hard resetting the board) **and switches**
- Detects **falling edges** on button signals (buttons are low enable)
- Triggers an **IRQ** whenever a button is pressed, IRQ cleared by the CPU through a dedicated register
- Exports a register that outputs which button was pressed and switch value
- Event timeline: button pressed -> IRQ -> IRQ handler reads which button was pressed, main program reacts accordingly, clears IRQ -> resume playing
    - Mode 0 : Mute, reset song, next song
    - Mode 1 : Volume down, volume up, volume reset
    - Mode 2 : Saw tooth wave, square wave, triangle wave

Python: meta-generation of VHDL code

- Example: VU meter unary conversion
- Exponentially smoothed value is a binary value => need to convert it to unary
- However, conversion depends on number of LEDs available!
- We generate the code as a function of #LEDs
- More advanced usage than generics

```
vu_meter_unary_conversion = [
    f"""
                elsif to_integer(vu_meter_value) <   {2 ** i} then
                     vu_meter <=  \"{('1' * (i +
1)).zfill(VU_METER_DEPTH) }\";"""
    for i in range(1, VU_METER_DEPTH)
]
vu_meter_unary_conversion =  f"""
                if to_integer(vu_meter_value) <   {2 ** 0} then
                     vu_meter <= "  {('1' * 1).zfill(VU_METER_DEPTH) }";
{"".join(vu_meter_unary_conversion)[ 1:]}
                end if;
"""

architecture =  f"""
…
-- VU meter conversion to unary process
    vu_meter_unary_conversion : process (aud_clk12, reset_n)
    begin
         if reset_n = '0' then
              vu_meter <= (others => '0');
         elsif falling_edge(aud_clk12) then
              if sclk_en = '1' then
{vu_meter_unary_conversion[ 1:-1]}
              end if;
         end if;
    end process vu_meter_unary_conversion;
…
"""
```

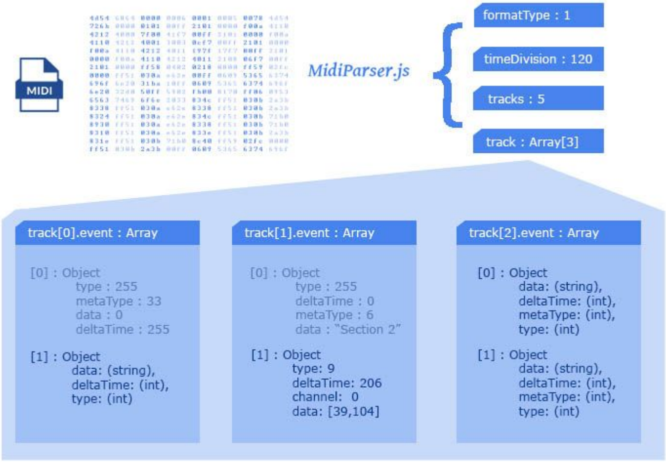
Python: meta-generation of VHDL code

- Example: VU meter unary conversion
- Exponentially smoothed value is a binary value => need to convert it to unary
- However, conversion depends on number of LEDs available!
- Which yields the following VHDL
- Also used for sample rate and depth, # of osc instances, ...

```vhdl
-- VU meter conversion to unary process
   vu_meter_unary_conversion : process (aud_clk12, reset_n)
   begin
       if reset n = '0' then
           vu meter <= (others => '0');
       elsif falling edge(aud clk12) then
           if sclk en = '1' then
               if to integer(vu meter value) < 1 then
                   vu meter <= "0000000001";
               elsif to integer(vu meter value) < 2 then
                   vu meter <= "0000000011";
               elsif to integer(vu meter value) < 4 then
                   vu meter <= "0000000111";
               elsif to integer(vu meter value) < 8 then
                   vu meter <= "0000001111";
               elsif to integer(vu meter value) < 16 then
                   vu meter <= "0000011111";
               elsif to integer(vu meter value) < 32 then
                   vu meter <= "0000111111";
               elsif to integer(vu meter value) < 64 then
                   vu meter <= "0001111111";
               elsif to integer(vu meter value) < 128 then
                   vu meter <= "0011111111";
               elsif to integer(vu meter value) < 256 then
                   vu meter <= "0111111111";
               elsif to integer(vu meter value) < 512 then
                   vu meter <= "1111111111";
               end if;
           end if;
       end if;
   end process vu_meter_unary_conversion;
```

MIDI Parser: convert MIDI files with JS CLI

- Uses midi-parse-js library which translates binary MIDI events into JSON
- Fetch all note start / stop events (0x80/90)
- Delta time expressed relative to previous event → want inverse for note duration: must aggregate all delta times until next note event
- Octave transposition, duration scaling
- Outputs C header with notes array

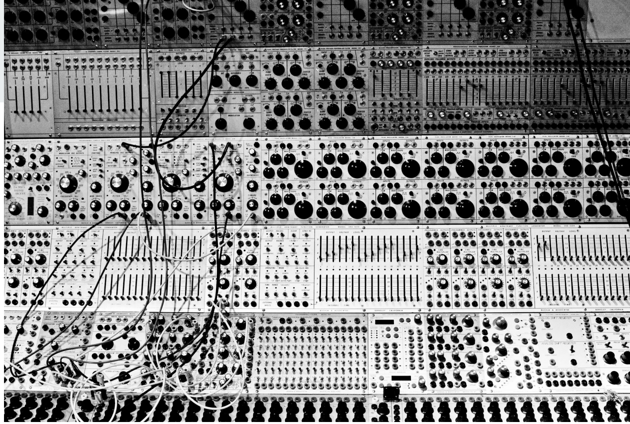



```
$ node parse <MIDI file> <C variable
  name> [transpose] [scale]
```

Demos

Python meta-programming scripts

MIDI parser



Limitations and future work

- VU meter implementation only a rough approximation of a true VU meter, also suffers from convergence issues, could implement a better one using a moving average with a FIFO
- GCC compilation issues preventing us from including more than ~8KB of static data in header files fpga_sound_synthesizer_bsp/HAL/src/alt_instruction_exception_entry.c:95: warning: Unable to reach (null) (at 0x0403075c) from the global pointer (at 0x04028160) because the offset (34300) is out of the allowed range, -32678 to 32767.
- Can add as many sound pipeline stages as wished: filters, effects, envelope and LFO modulation, white / pink / brown noise, configurable routing, …
- Scalable to bigger and smaller FPGAs
- Interpret live MIDI data through Ethernet or USB for instance

Questions

Open-source repo ?

Buying the board ?