Melody Generator

Raphaël Bouilhol, Victor Verbeke, Alexis Rossi

May 2018 - v1.0

Introduction

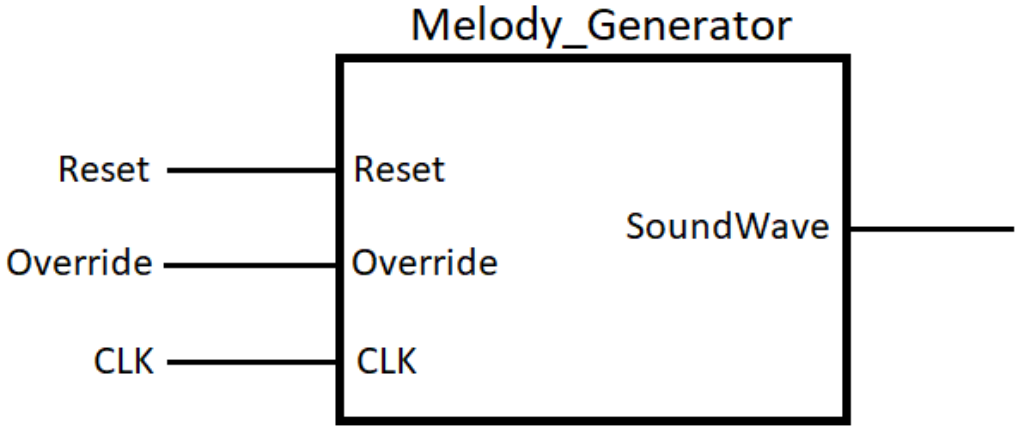
This interface was created as part of a digital electronics course. It is made from scratch, as we made this project between the months of February and May 2018. It has been coded in the frame of a VHDL course and is for learning purpose.
We provided, along with the RTL code, a self-testing VHDL Test Bench and a script for Altera® Quartus.
The Melody Generator is a simple project made to play music sheets hardcoded into the circuits. It plays notes from A0 up to G#3, and from an eighth of a second to a full second.

Features

- Fully synchronous solution : one clock, one global reset.
- Speed is based on the clock frequency.
- The output signal is a mono square wave.
- Simple controls : a start button, a reset button.

Note : Only works with the reset button pressed. Releasing the reset button and pressing it again makes a clean reset.

Module Description



Simple representation of the inputs and outputs of the design

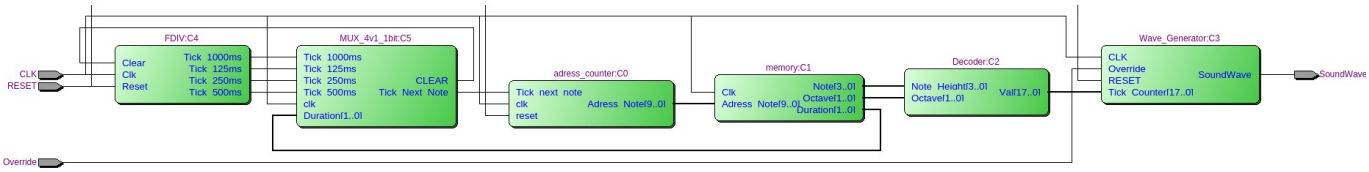
Inputs

CLK - Clocking source for all the synchronous design.
Reset - Asynchronous and global reset for the design.
Override - Used in debug to set SoundWave to 0, hence producing no sound.

Outputs

SoundWave - Square wave, music produced by the design.

In-Depth Analysis



Frequency Divider (FDIV)

This component generates multiple ticks, used later to know when we go to the next note.

Inputs

CLK - Clocking source for all the synchronous design.
Reset - Asynchronous and global reset for the design.
Clear - Reset of the tick counters inside the component

Outputs

Tick_1000ms - Generates a tick every second
Tick_500ms - Generates a tick every half of a second
Tick_250ms - Generates a tick every fourth of a second
Tick_125ms - Generates a tick every eighth of a second

Multiplexer (MUX_4v1_1bit)

This component choose when to go to the next note using the ticks generated by FDIV and the duration of the note sent by the memory.

Inputs

Tick_1000ms - Tick coming in every second
Tick_500ms - Tick coming in every half of a second
Tick_250ms - Tick coming in every fourth of a second
Tick_125ms - Tick coming in every eighth of a second
CLK - Clocking source for all the synchronous design.
Duration [1..0] - Duration of the note played, coded on two bits.

Outputs

Clear - Used in FDIV to clear out every intern tick counter.
Tick_Next_Note - Used by the Address Counter to go the the next note in the memory.

Address Counter

Gives the information to the memory of what note should be played next, through the note

address in the rom.

Inputs

Tick_Next_Note - Used in the component to increment Adress_Note.
CLK - Clocking source for all the synchronous design.
Reset - Asynchronous and global reset for the design.

Outputs

Adress_Note [9..0] - Address of the wanted note on 10 bits.

Memory

The hard-coded notes used in the device are stocked as signals in the memory. In order to modify the music, you must edit the component “rom”, used in the memory Using the address counter, it delivers the correct note, octave and duration of the note to the decoder.

Inputs

Adress_Note [9..0] - Address of the wanted note on 10 bits.
CLK - Clocking source for all the synchronous design.

Outputs

Note [3..0] - Height of a note (from A to G#).
Octave [1..0] - Value of the octave (from A0 to A3, for instance).
Duration [1..0] - Length of the note (described in the multiplexer).

Decoder

The decoder converts a note in a number of ticks needed to form a correct soundwave. That number is sent to the Wave Generator afterwards. Asynchronous component.

Inputs

Note_Height [3..0] - The note to create.
Octave [1..0] - Octave of the note to create.

Outputs

Val [17..0] - Number of ticks needed to toggle SoundWave in the Wave_Generator.

Wave Generator

This component toggles SoundWave (hence creating an effective sound signal) using the value sent by the Decoder.

Inputs

Tick_Counter [17..0] - Number of ticks needed to toggle SoundWave
CLK - Clocking source for all the synchronous design.
Reset - Asynchronous and global reset for the design.
Override - Used in debug to set SoundWave to 0, hence producing no sound.

Outputs

SoundWave - Sound wave, square signal at the frequency required.

Coding The Memory

If you want, you can reprogram the memory. In order to do that, you must understand the adequate signal, mem (in the component vhd, located in memory). It is an array of 8-bit vector, and each vector can be decomposed like this :

```
Note <= Vector (7 downto 4);
Octave <= Vector (3 downto 2);
Duration <= Vector (1 downto 0);
```

Now, you can write any note based on this decomposition :

```
Duration = "00" => The note is 1s long.
Duration = "01" => The note is 0.5s long.
Duration = "10" => The note is 0.25s long.
Duration = "11" => The note is 0.125s long.

Octave = "00" => The note is on the first octave.
Octave = "01" => The note is on the second octave (frequency x 2).
Octave = "10" => The note is on the third octave (frequency x 4).
Octave = "11" => The note is on the fourth octave (frequency x 8).
```

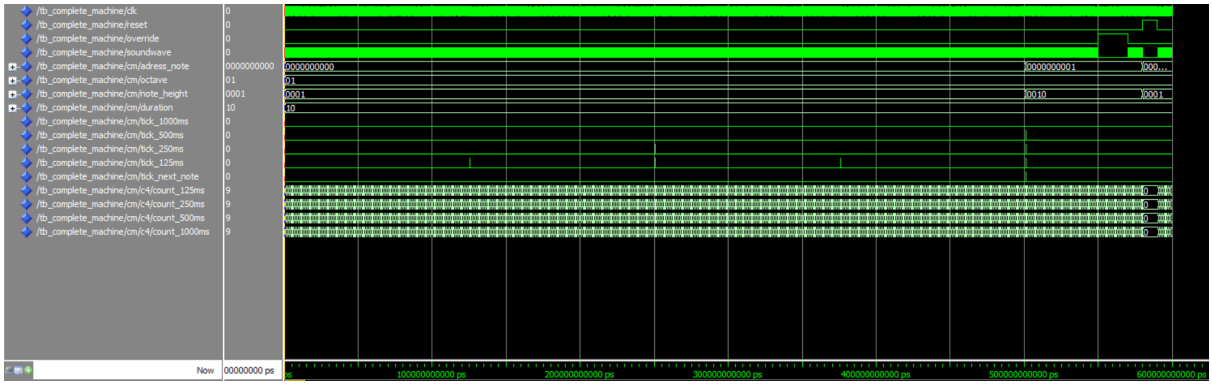
Value of the 4-bit vector Note	Associated note	Frequency (first octave)
0000	A	440 Hz
0001	A#	466 Hz
0010	B	494 Hz
0011	C	262 Hz
0100	C#	277 Hz
0101	D	294 Hz
0110	D#	311 Hz
0111	E	330 Hz
1000	F	349 Hz
1001	F#	370 Hz
1010	G	392 Hz
1011	G#	415 Hz
Others	Not a note = Silence	0 Hz

Implementation

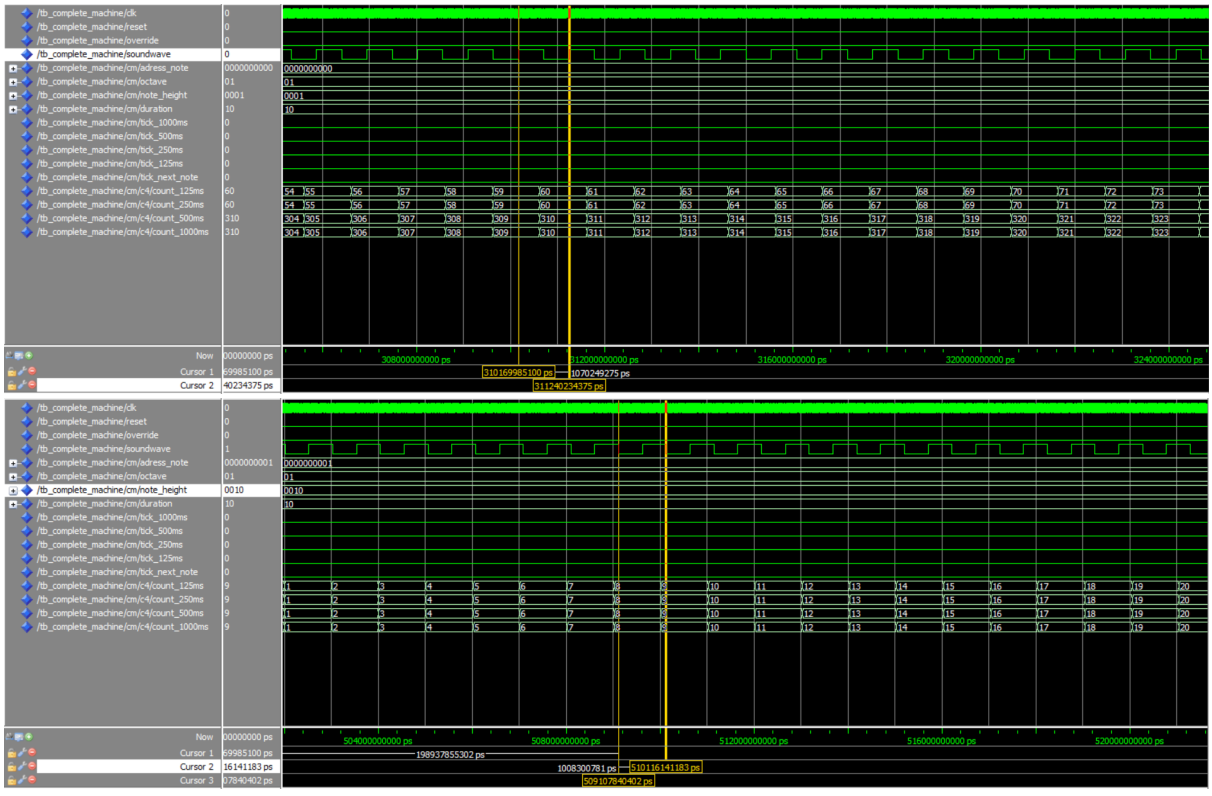
Using the device is simple. Two inputs are used : the clock from the FPGA board and a button (reset). An additional button is used in debug mode (override).

Simulation

Test benches for every component are provided with the source code.
Since our project is based on music (hence on notes), the simulation of the result can be long to run. Here are some results.



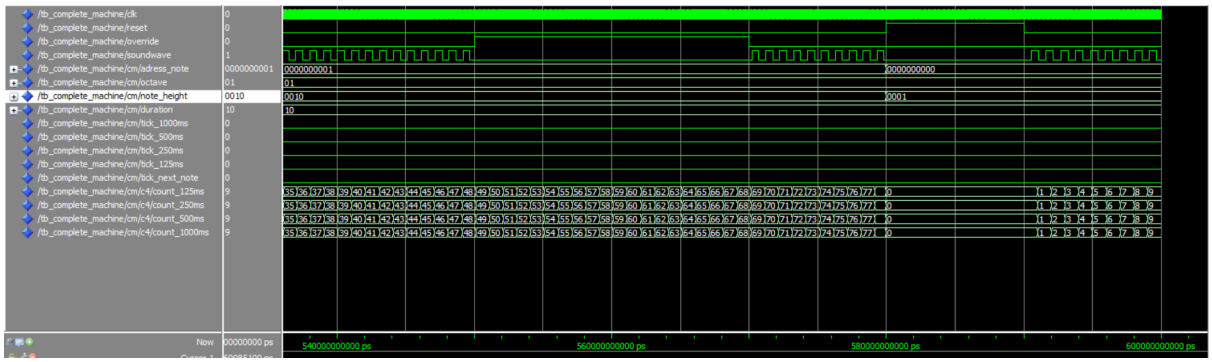
Simulation of the Complete Machine.



The two first notes of the music, produced in simulation.

As you can see, the sound wave produced has two different wavelengths when the two notes are played. If you (want to, as it can be tedious) count manually the frequency, the two notes are a A#1 and a B1 (respectively 932 Hz, period of approx 1.07ms and 988 Hz, approx 1.01ms, the periods are indicated with cursors on the 2nd and 3rd screenshot), as wanted.

Another way to see if the produced note is correct : run the design and analyse the soundwave on an oscilloscope.



Reset and Override tests.

The reset and the override and perfectly working, the reset stops the sound and all counters while coming back at the start of the partition, whereas the override just stops the soundwave signal from producing sound.

Simplest Application

To simplest application is an implementation with a speaker. Through a 3.5mm audio Jack cable, you can connect the music player to speakers.
In most speakers, there is power amplifier that uses a power supply, but in the case of audio headset, there is none.
In order to hear the music in both ears (most headphones are in stereo), the soundwave must be applied to the two jack outputs.
This application is trivial and can help verify that the hardware is correctly connected and in working order.

Reference Design

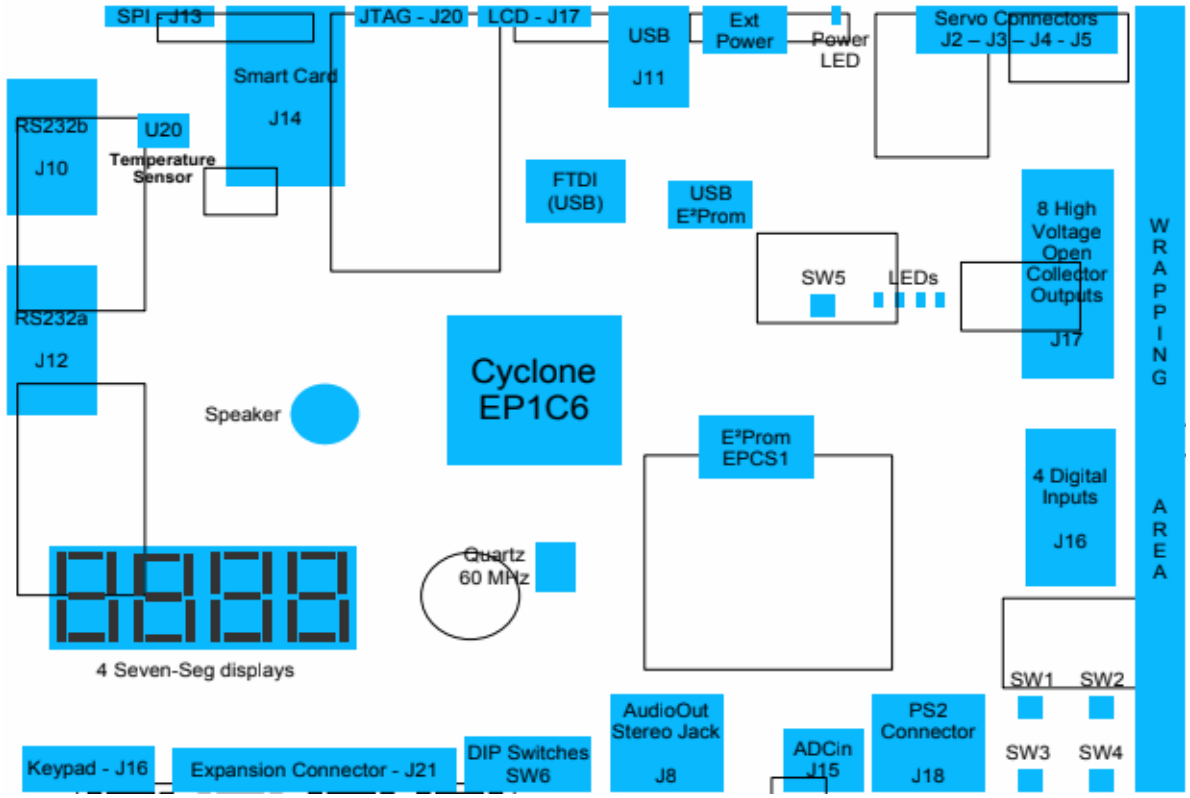
We used for our tests the following equipment:

- 1x Tornado Card, from ALSE, using a Cyclone 1C6 from Altera, provided with
  - 1x USB Blaster,
  - 2x USB Cable (one for the Blaster, the other one for the power supply).
- 1x Headphone (Syberia v1, Steelseries).

In order to make the whole system work, you simply have to connect the headphone to the card through the AudioOut Stereo Jack (as you can see in the schematic view below).

We were using **ModelSim v6.5b** for the coding and the simulation, and **Quartus** in order to

put the design into the FPGA board.



Schematic view of the Tornado Card

Every code used in order to mount the card on Quartus is provided with this document.

Conclusion

This simple device is the result of a three students work. It is easy to understand, as it structure is simple and it is basic VHDL, but it can be tedious to rewrite a new song on it.
The actual song on the source code is Plug In Baby, by Muse.