ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

MIDI Synthesizer

Daniel Engeler, D-ITET / 7 Samuel Nobs, D-ITET / 7

February 3, 2003

Abstract

This document is the report of a Semester Thesis at the Department of Information Technology and Electrical Engineering. It describes the steps from a project idea to its implementation in silicon using VHDL and professional design tools, such as Synopsys and Silicon Ensemble.

The chip designed is a synthesizer with MIDI interface. The tone generator performs additive sound synthesis: it consists of 8 sine oscillators whose amplitude envelopes vary over time. The parameters of these amplitude envelopes and some additional parameters can be adjusted using MIDI at runtime. The sound output is digital, sampled at 44.1 kHz with a resolution of 16 bits.

Authors:	Daniel Engeler	danengel@ee.ethz.ch
	Samuel Nobs	nobssa@ee.ethz.ch
Advisor:	Stephan Oetiker	oes@iis.ee.ethz.ch
Co-Advisor:	Simon Häne	haene@iis.ee.ethz.ch
Supervisors:	Hubert Kaeslin	kaeslin@iis.ee.ethz.ch
	Norbert Felber	felber@iis.ee.ethz.ch
Professor:	Wolfgang Fichtner	fichtner@iis.ee.ethz.ch

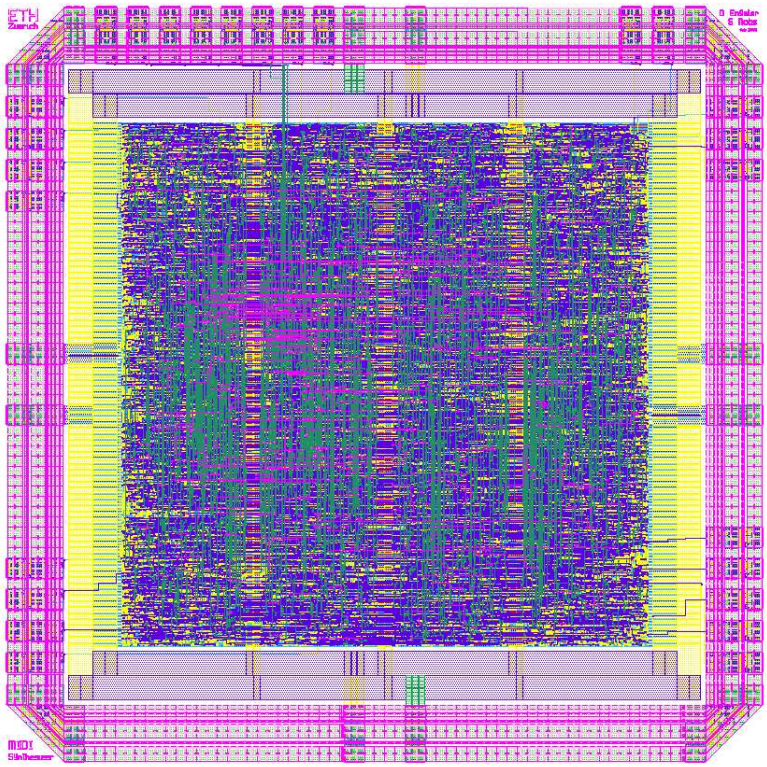

Figure 1: Final Layout of the MIDI Synthesizer

Contents

Chapter 1

Basics

1.1 Sound Synthesis

In contrast to acoustic instruments, synthesizers do not produce sounds by the vibration of mechanical parts inside them. They output electrical signals corresponding to the sound waveform they produce. The three main classes of synthesizers are the following ones, named by their way of generating waveforms:

- **Analog** synthesizers generate their waveforms at runtime using analog voltage controlled oscillators.
- **Digital** synthesizers play back waveforms that have been calculated or recorded before and are stored in memory.
- **Virtual-analog** synthesizers emulate analog sound synthesis using digital signal processing technology. The sound output is calculated at runtime. The synthesizer designed in this semester thesis belongs to this class.

The classes of analog and virtual-analog synthesizers can be split again into two subclasses[1]: some of them perform additive synthesis, and some do subtractive synthesis.

- **Additive** synthesis is based on the fact that every periodic waveform can be represented by a fundamental sine wave and its harmonics. Changing the amplitudes of these harmonics yields different waveforms. Our synthesizer performs additive synthesis.
- **Subtractive** synthesis works the other way round: waveforms containing lots of harmonics like rectangle, triangle and sawtooth waves are used, and then some harmonics are amplified or damped using a filter.

There is one thing that all analog and virtual-analog synthesizers have in common: envelope generators. They are used to vary sound parameters like amplitude, pitch, or cut-off frequencies of filters over time to make the sounds more vivid. The most widely used is the "simplified ADSR-type", which we used for the amplitude envelope. This acronym stands for Attack, Decay, Sustain, Release, denoting the four phases of such an envelope generator. Please refer to figure 3.11 on page 33 for a graphical description of these four phases.

In the simplified ADSR model, the time required for the amplitude to rise to its maximum level is called attack time. The time it takes for the sound to die away by dissipating its energy is called decay time. If the

[1] No need to say that there exists no sharp borderline between these two classes.

sound is stopped before it has had a chance to die away completely, the time it takes before it stops sounding is called release time.

For example, an organ sound has quite a fast attack, very little decay, a sustain that lasts as long as the performer holds down the key, and a fast release. Another example is the trumpet sound, having a slower attack than the organ due to the fact that the air is blown by a human. The sound then decays slightly and sustains as long as the player keeps blowing. The release is slower than an organ's.

In a trumpet sound there is sometimes a slight variation in pitch when the player starts to blow, and such variations can be modeled using a pitch envelope. This type of envelope is not implemented.

Changes in tone color, the so called timbre, over time may be implemented using a filter whose cutoff frequency is varied using an envelope generator. Although no filter is present in our synthesizer, we can vary the timbre slowly over time by setting different ADSR parameters for the fundamental wave and its harmonics.

1.2 MIDI

MIDI is the acronym for Musical Instruments Digital Interface. It is a communication standard developed in 1983 by the major electronic instruments manufacturers to extend the possibilites of their products. Several devices can be connected in a chain by simply plugging a cable from one device to the other. MIDI then allows music to be played remotely and to completely control a synthesizer from another device. Songs can be played, edited and stored electronically in a format that is understood by virtually every music making box on the market.

MIDI does not actually store music in its "natural" form, i.e. waves which can be played back directly. Instead, think of MIDI as a sheet of music where only instructions about *what* to play are given. A note on the sheet corresponds to MIDI "note-on" and "note-off" messages. However as not only it is indicated what, but also *how* to play a song, additional MIDI messages can change volume and modulation as well as tempo, timbre and pitch. More advanced applications use MIDI to indicate the starting and stopping points of a song or the metric position within a song.

The basis of MIDI communication is the byte which is serially transmitted at a bit rate of 31.25 kBit/s. MIDI messages always consist of a status byte (MSB[2] set) followed by one or more data bytes (MSB cleared). A status byte includes four bits indicating the channel to which the current message is addressed. By assigning different channels to each device used, this mechanism allows to send messages specifically to one of them.

A "note on" message for example consists of a status byte with the value 144 and two data bytes, the note number and the velocity. All notes are numbered, e.g. the standard pitch (440 Hz) is assigned 69. The higher the velocity, the harder the key was pressed. According to it, synthesizers (ours included) usually increase the volume of the note being played. More sophisticated approaches also change its nature to make it sound harder.

As MIDI is a realtime protocol, at the time a "note on" message is sent, it is not yet known when the note is released again. There is a separate "note off" message with a status byte 128. Another important category are the "control change" messages. After a status byte 176 follow two data bytes, the controller number and its new value. For example, our synthesizer uses controller 7 to set the main volume, so to the set the main volume to 55, send the three bytes 176, 7, 55.

To efficiently send many messages of the same type in a row, e.g. five note-on messages to play an accord, the so-called *running status* is used: A complete message consists of the status byte (note-on, note-off or control change), followed by the two according data bytes. If the following message has the same status byte, it may be omitted. For example, to set controller 7 to 55 and controller 9 to 120, instead of sending 176, 7, 55 for the first and 176, 9, 120 for the second controller, simply send 176, 7, 55, 9, 120.

[2] Most significant bit

MIDI has established itself as a standard which is used world-wide by all artists and music producers who deal with electronic tools. In its 20 years of existence, it has seen many refinements to keep pace with other developments in the industry. However, its simplicity and universal character are still unchallenged.

Chapter 2

Simulink Model

2.1 Top level

To gain a fundamental understanding of the processes that will be implemented on this chip and to be able to prove that the sound output produced by this device is good, we had to create a behavioural model first. Instead of starting to write VHDL code we decided to use Simulink/Matlab, taking the following advantages into account:

- graphical and easy to use interface
- all results can be analyzed directly in Matlab
- nice displaying and debugging gimmicks

The main disadvantage of this approach is that the model obtained is not very close to hardware. This means that the effects of finite word lengths are not taken into consideration, and most of the elements used are not realized in the way they are in hardware. Also, there is no need to care about clock domains, which makes it easier to create a functional model. These drawbacks might be covered in Simulink, but this would mean lots of effort which we wanted to avoid. For the moment we could live with those drawbacks because the option to have all results at hand directly and to be able to tune the system graphically is so valuable. First this functional model must be brought to a point where it sounds good, and only then we will start caring about the detailed structure, i.e. finite state machines, clock domains, and timing aspects.

The top level of this Simulink model is shown in figure 2.1. It consists of

- a MIDI interface
- a register bank containing all configuration data
- 8 oscillator sections including
  - a sine wave generator
  - a rectifier
  - an ADSR envelope generator
  - a multiplier applying the envelope to the signal
  - a multiplier adjusting the individual volumes
- an adder building the sum of the signals of the 8 oscillator sections

- a multiplier for the volume depending on the note-on velocity
- a multiplier for the main volume
- a look-up table containing the frequencies for the oscillators
- an output block writing the signal obtained to the Matlab workspace

Obviously, this model is rather straightforward and not optimized for neither speed nor efficiency.

2.2 MIDI interface

2.2.1 Overview

The first part of the MIDI interface (figure 2.2) is a finite state machine reading the MIDI stream from an external file. This file contains one MIDI byte per line, preceded by the point in time where reception of this byte should occur. Additionally, a comment starting with the % sign may be added to each line. If there is a data item ready at *midi_out* to be processed by the second part of the Interface, the signal *dataRdy* is active for one clock cycle. Please remember that so far this is true for this simulation model only, as clocking considerations will be dealt with later in the design process. As this FSM will be implemented in a completely different way on the chip, no further explanations are given.

The second part of the interface is another finite state machine acting accordingly to the MIDI bytes received from the first part. This block will be referred to as "MIDI FSM" from now on in this text. Recognized messages are forwarded to the corresponding output port of the interface, other data is ignored.

A clock generator is included to trigger these two FSMs, and there is an additional block that decrements the key number[1] by 36. Also, there is a display plotting the velocity and key number of note-on messages to facilitate the debugging process.

2.2.2 Finite State Machine

The model of the MIDI FSM was built using the Stateflow extension of Simulink which allows to create such automatons by defining statecharts. After system initialization, the state *init* is active. Also, when the device is being reset, the MIDI FSM will reenter this state. This reset is synchronous because it is not a signal but a MIDI message. State transitions become active on the rising edge of the clock only if *dataRdy* is high, which means that the next MIDI byte is available at the *midi_in* port.

The MIDI interface only recognizes note-on, note-off and a subset of the control change messages directed to the MIDI channel defined by the constant *m_chan* (see the MIDI Implementation Chart on page 53); all other messages will be simply ignored except the system realtime message *reset* used to reset the device and the system exclusive messages which require some special treatment as described later in this section.

note-on messages tell the FSM to enter the *note_on* state. The next data byte received is interpreted as the key number corresponding to the key hit on the keyboard. This number is then output via the *ntNum* port. The following data byte represents the velocity of the key hit. This value is output at the *ntVel* port only if the last preceding key was released already. If it has not been released yet, this means that the notes are played *legato* and the last value of the velocity will be preserved[2]. Now the *ntGate* signal is set to high, and it will stay high until it is told to stop doing so by a note-off message.

[1] our synthesizer handles key numbers 36-107

[2] This is because playing *legato* prevents the ADSR Envelope from being restarted, so a change in velocity would yield a step in the sound volume.

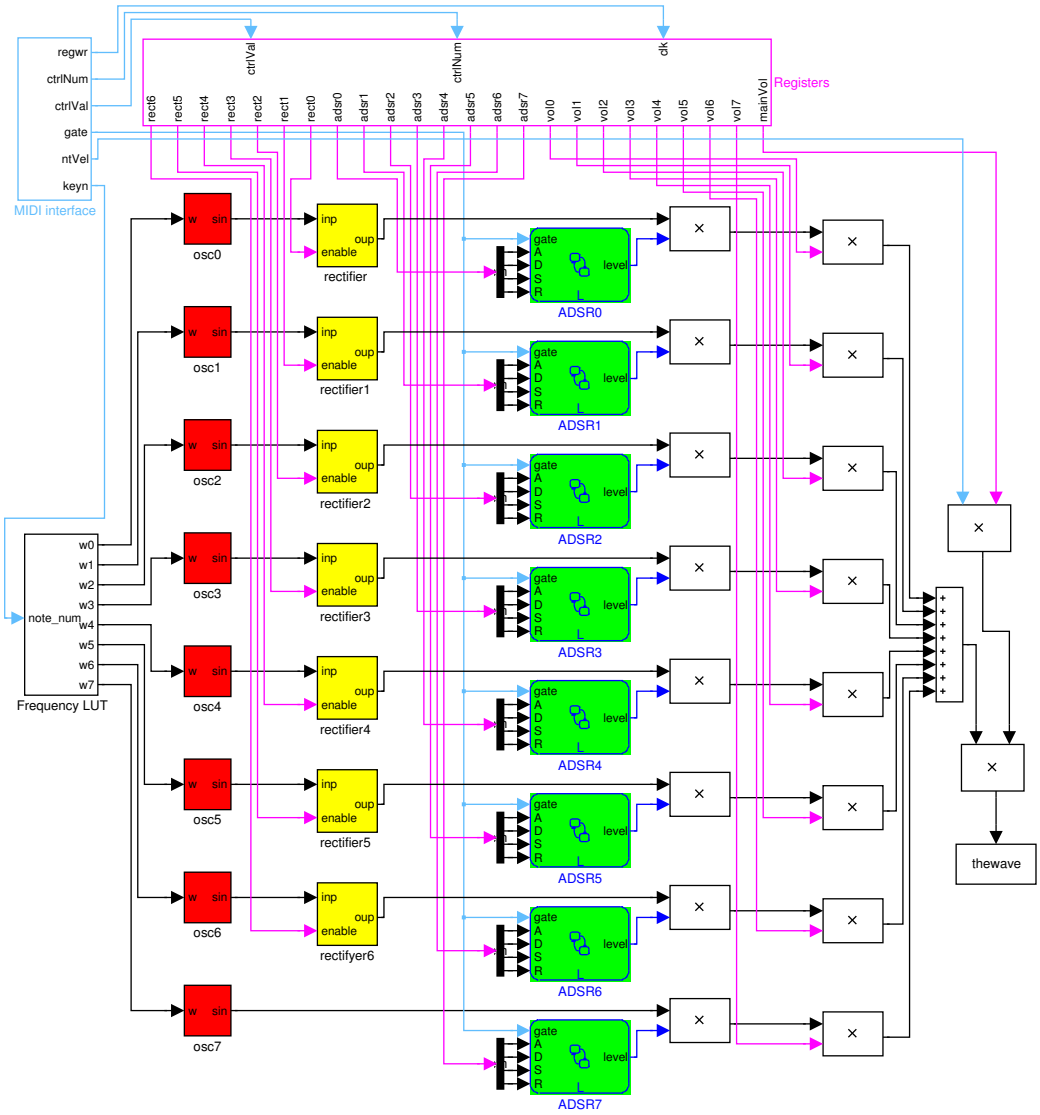

Figure 2.1: Top level of the Simulink model

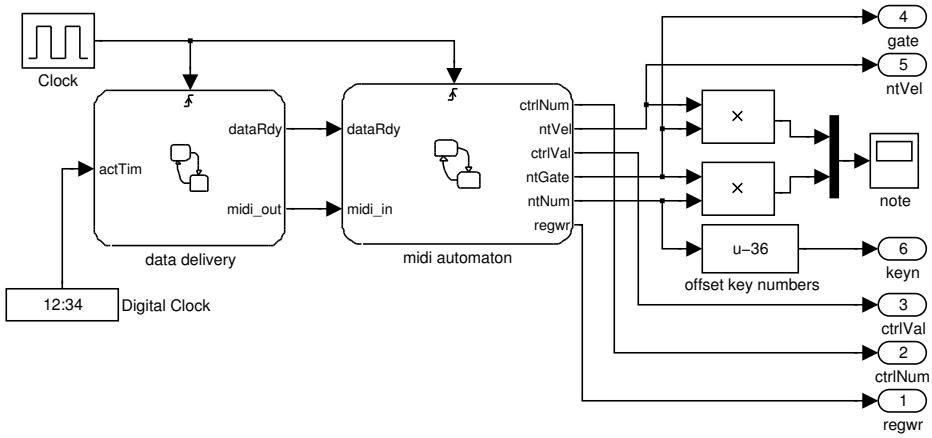

Figure 2.2: MIDI interface

note-off messages force the FSM into the *note_off* state where the first data byte represents the number of the key released and the second data byte corresponds to the release velocity. If the number of the released key is the same as the number of the key pressed before, the *ntGate* signal is set to low[3], which means the note will be stopped. The release velocity is ignored.

control change messages are used to store new values in the configuration registers. They bring the FSM into the *control* state, where the subsequent byte is output as a controller number on the port *ctrlNum*. At this moment, the signal *regwr* that triggers the registers storing the configuration data is set to low. The next byte is regarded as the new value for this controller number. *regwr* gets rised to high, telling the according register in the register bank to overwrite its current value with the new one.

The state *sysex* was first used to handle system exclusive messages, but now it handles all other unrecognized messages too, including messages for other MIDI channels. This state will be entered upon reception of one of these messages and left as soon as the next status byte is recognized.

2.3 Register Bank

All configuration data like oscillator volumes, shapes, and the characteristics of the ADSR envelope generators is stored in the register bank depicted in figure 2.4. The basic building block of most of the constructs in this bank is a single register-like block (figure 2.5) that stores the value (in the range of 0...255) at the input *dat* only if a rising edge occurs on the input signal *clk* and the signal *sel* is equal to a given constant *ctrl* that represents the controller number handled by this register. So for every controller number we need a copy of this basic block.

Registers for similar controllers are grouped into larger register banks except for the Main Volume Register and the Rectify Register, which consist of one such basic register. Additionally, the byte in the Rectify Register is split into 7 bits to control rectification of the 7 lower oscillators. The Volume Register contains 8 basic registers, one for each oscillator section, and the ADSR Register consists of 8 blocks each containing 4 registers for the 4 parameters of the ADSR envelope generators. Also, there are some displays included for debugging and tracing purposes. Figure 2.6 (left) shows an excerpt from the ADSR Register reduced to 2 blocks instead of 8.

[3]notes played *legato* would be stopped when a key other than the one corresponding to the actual note is released, which is not desirable.

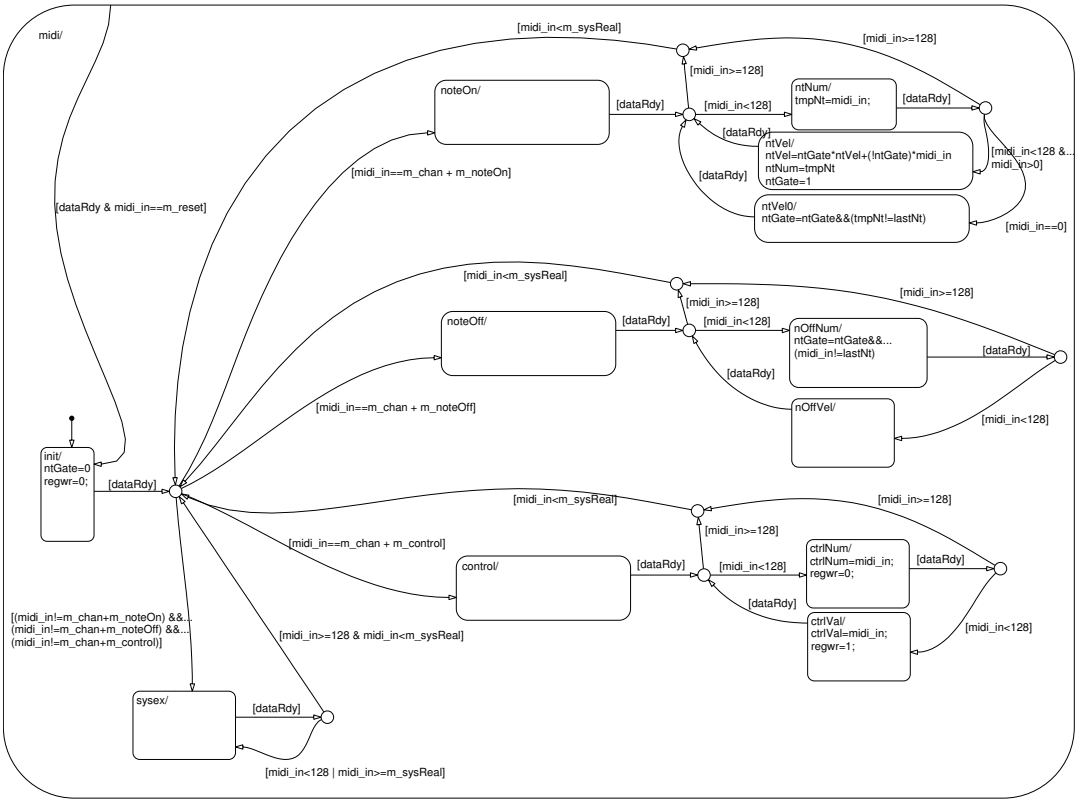

Figure 2.3: MIDI FSM

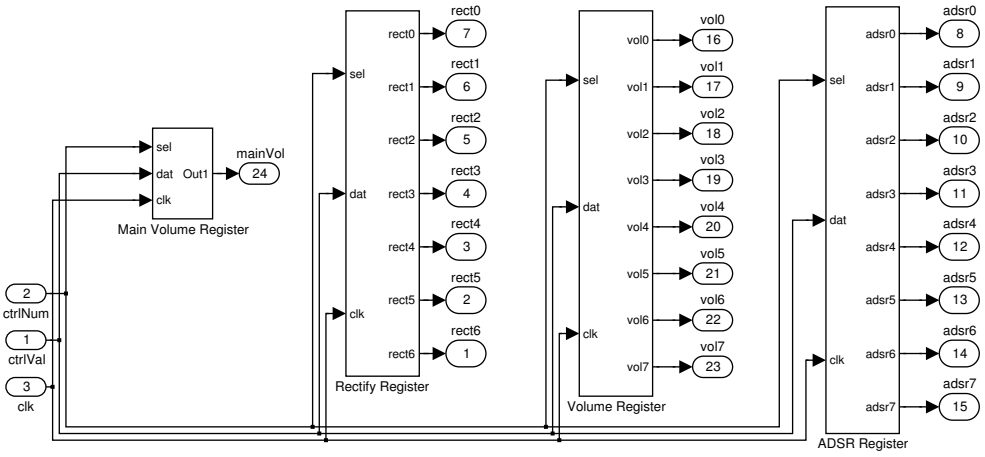

Figure 2.4: Register bank

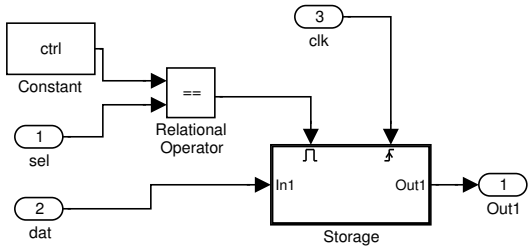

Figure 2.5: Basic register

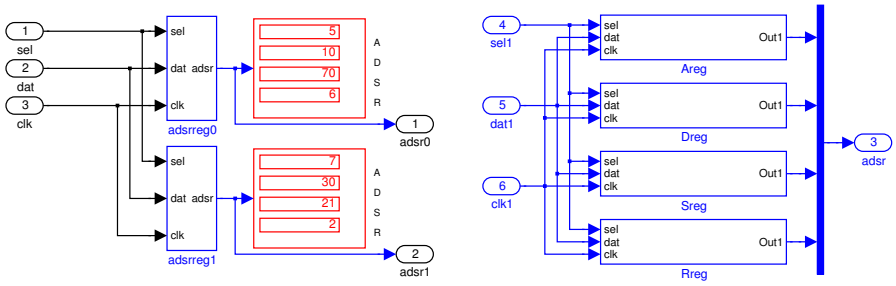

Figure 2.6: Excerpt from the ADSR Register bank

The right side of the figure corresponds to the inner structure of one of these blocks.

2.4 Oscillators

There are eight oscillator sections which are used to form the sound by a fundamental wave and seven harmonics.

2.4.1 Wave Form

Wave form generation is limited to sine and rectangular waves. There is one sine generator per section that reads the frequency at which it should oscillate from the frequency lookup table. At this point, the built-in `sin` function is used. Also, no quantisation is performed. The rectifier block forwards the data received at the *inp* port to the port *oup* without modification if *enable* is low, otherwise it passes $+1$ or -1 depending on the sign of *inp*. In the 8th oscillator section we removed the rectify block, because normally the frequency of that oscillator is so high that there is almost no audible difference between a sine and rectangle wave. This has the advantage that we may come up with one MIDI byte only to control these rectifiers[4].

2.4.2 Other approaches

Before we decided to go for additive sound synthesis using eight sine oscillators, we evaluated other possibilities. The main alternative we took into consideration was to "draw" a waveform with arbitrary parameters a, b, and c. See figure 2.7 for what we had in mind. We refer to "drawing" a line as the according algorithm has its origins in computer graphics.

[4] only 7 bits of the MIDI byte contain information because the MSB is always zero for data bytes

```
v = 1
n = 0
for h = 1 to H
    draw(h,v)
    v = v + V'
    n = n + N
    if n >= H
        v = v + 1
        n = n - H
    end if
end for
```

Table 2.1: Line-drawing algorithm

Choosing a set of parameters, many different waveforms can be generated. For example, a sawtooth-wave is obtained by setting $a = b = T/2, c = T$, a square-wave using $a = 0, b = c = T/2$.

By adding a low-frequency, small-amplitude oscillation to each parameter, a rich sound is produced. The acoustical results were not as good as with the final approach using eight sine oscillators, nevertheless it would have been satisfactory. Unfortunately, the algorithm to "draw" the necessary lines proved ill-suited for hardware implementation.

Drawing a period as in figure 2.7 mainly consists of drawing some lines each into a matrix of given dimensions H and V. In figure 2.8, an example is shown where $H = 20$ and $V = 16$. In practice, V would be ± 65536, whereas H would range from 1 for a square-wave up to a few hundred for low-frequency sawtooth-waves.

Firstly, the slope V/H has to be transformed into a mixed fraction

$$\frac{V}{H} = V' + \frac{N}{H} \quad \text{with } V', N \in \mathbb{N} \text{ and } N < H, \tag{2.1}$$

after which the algorithm in table 2.1, represented in pseudo-code, does the actual drawing. It is easy to implement as it requires only a few additions and one comparison at each step.

The real problem lies in how to transform the slope V/H into a mixed fraction. A naive approach would look like the algorithm in table 2.2, however, for $V = 65536$ and $H = 1$, many thousand cycles are required. A more efficient algorithm is depicted in table 2.3.

In some cases many cycles are still required, which would not easily allow to change the waveform at runtime. It is also not known a priori how many cycles the algorithm will take. As the procedure is very much dependant on its input data it is more suitable for a general purpose processor than a streamlined ASIC.

The above mentioned problems have led us to reject the line-drawing approach altogether and go for additive sound synthesis. It is not only much more suited for hardware implementation but also provides richer sounds.

2.4.3 Envelope

A very important part of each oscillator section is the ADSR envelope generator which is implemented as two FSMs (see figure 2.9).

The FSM *adsr_states* cares about the 4 basic phases of the envelope: the *attack* phase that starts when a key is hit, the *decay* phase that begins when the *attack* phase is done, the *sustain* phase after the *decay*

```
V′ = 0
while V >= H
    V = V − H
    V′ = V′ + 1
end while
```

Table 2.2: Simple decomposition of a fraction V/H

```
N = V
V′ = 0

n = H
v = 1
while 2 * n < N
    n = n * 2
    v = v * 2
end while

while N >= H
    while n > N
        n = n / 2
        v = v / 2
    end while
    N = N − n
    V′ = V′ + v
end while
```

Table 2.3: More efficient decomposition of a fraction V/H

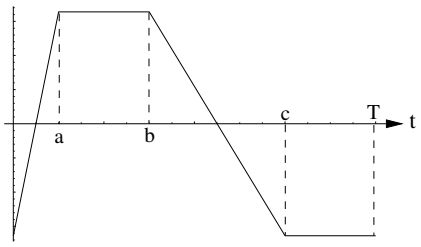

Figure 2.7: Example period generated by the line-drawing algorithm we had in mind

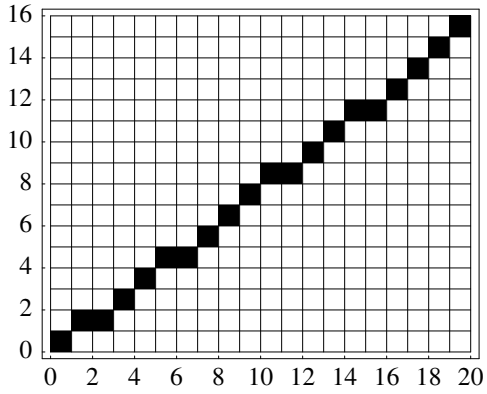

Figure 2.8: Example line with $H = 20$ and $V = 16$

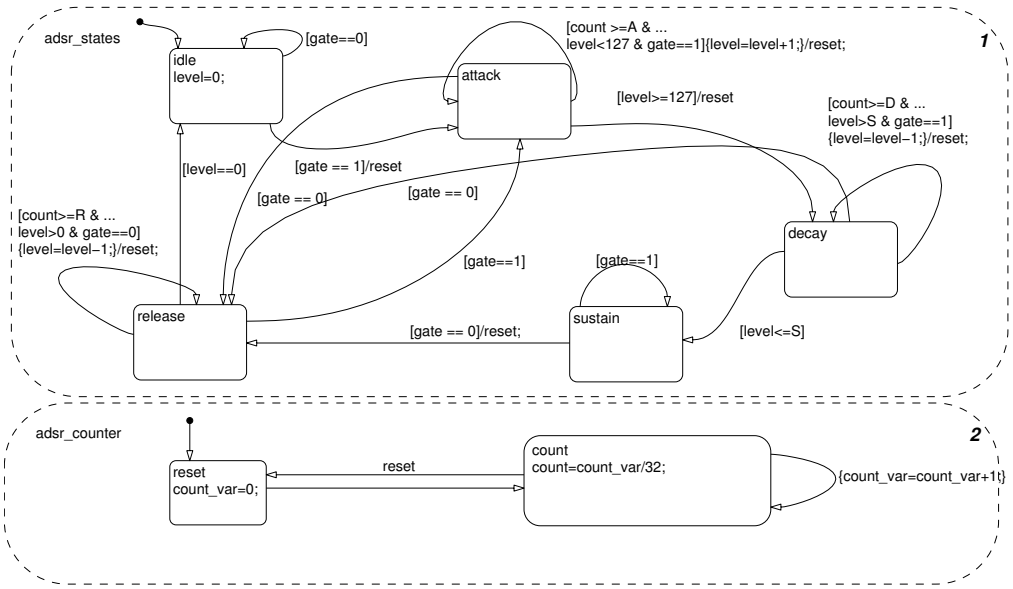

Figure 2.9: ADSR envelope generator

phase, and the *release* phase that starts when the key is released. The FSM *adsr_counter* is a counter whose counting variable is divided[5] by 32 because otherwise it would count too fast (the clock is at 44.1 kHz).

The *adsr_states* FSM is in the *idle* state at the beginning. As soon as the *gate* signal changes from 0 to 1, i.e. a key is pressed, the *adsr_counter* is reset, and the *attack* state is entered. As soon as the counter reaches *A*, *level* is incremented by 1, the counter is reset, and *attack* is reentered; *A* is the factor by which the counter is slowed down additionally and corresponds to the attack time of the ADSR. This process is continued until *level* reaches its maximum value[6] 127. Now the counter is reset again and the *decay* state is entered, where *level* is decremented with the slowing factor *D* (the decay time). When *level* reaches the value of *S* (the sustain level), the state *sustain* is entered where the FSM waits on the signal *gate* to go from 1 to 0 (the key is released). Then the state *release* is entered and *level* is decremented to zero with the slowing factor *R*, which is the release time. Then the machine waits for the next note-on message. If the key is released while *attack* or *decay* is active, the state *release* is entered directly. If a key is pressed while *release* is active, *attack* will be activated.

2.5 Frequency Lookup-Table

There is not much to say about the frequency lookup-table. It's simply a device that outputs the fundamental frequency and the first seven multiples of this frequency according to the key number found on input *note_num*. See the table in Appendix C for more information.

2.6 Sound Analysis

2.6.1 Single Oscillator

To test the sound output of the synthesizer when it should play one note, we started a simulation with the MIDI bytes in table 2.4. First, ADSR configuration data for the fundamental oscillator is sent: very short attack time, short release time, a sustain level that represents half of the maximum amplitude, and a short release time. Then the oscillator is told to output a sine wave[7] as soon as a note-on message arrives; the main volume and the individual volume of this oscillator are set to the maximum value, 127. Then a note with note number 72 is started with maximum velocity. The note number 72 corresponds to a fundamental frequency of 523.25 Hz. A plot of the signal obtained is shown in figure 2.10. Looking at this plot, it seems like the model did its job well, but the audible result is not as convincing as the graphical: there is some ugly noise in the signal.

A spectrographical analysis of the sound output does visualize this noise (figure 2.11): besides the expected horizontal line at around 520 Hz, there is some energy distributed over the whole frequency range between 0 and 0.5 seconds at the beginning and between 2 and 2.2 seconds at the end of the signal[8]. Comparing this observation with the time plot in figure 2.10 shows that there is some coincidence between the occurrence of noise and the activity of the ADSR: decreasing or increasing the level in the ADSR produces unwanted distortions in the output.

Let's have a look at the signal generated by the ADSR envelope generator. As we can see in figure 2.12, we have a rather lousy step size in the edges of the ADSR signal, which is most likely the cause for this noise. The spectrogram plot in figure 2.13 indeed shows some ugly disturbances during these edges, the quantisation noise of the ADSR. This indicates that a resolution of 7 bits of the amplitude range is not high enough. If this resolution is extended to more bits, there is significantly less noise in the signal as can be seen in figure 2.14 where a spectrogram is shown for ADSR resolutions 8 through 12: The quantisation

[5] again: this is not how it is done in VHDL, but the easiest way in Simulink...

[6] We will analyze later whether 127 steps in resolution are enough or not. So all these factors are subject to change.

[7] this command is optional because sine wave is the default

[8] intensity goes from blue colors for low values over green, yellow, orange to dark red for high values

```
0.0    10110000 % controller running
0.0    01000000 % 64,A oscillator 0
0.0    00000000 % 0
0.0    01000001 % 65,D oscillator 0
0.0    00001010 % 10
0.0    01000010 % 66,S oscillator 0
0.0    01000000 % 64
0.0    01000011 % 67,R oscillator 0
0.0    00000101 % 5
0.0    00001001 % 9, Sine/Rect Select
0.0    00000000 % no osc set to rect
0.0    00000111 % 7, main volume
0.0    01111111 % 127
0.0    01111111 % 127, volume osc0
0.0    01010000 % 100
0.0    10010000 % note on
0.0    01001000 % 523.25 Hz
0.0    01111111
2.0    10000000 % note off
2.0    01001000
2.0    01111111
```

Table 2.4: Midi stream for a tone using one oscillator, including oscillator configuration

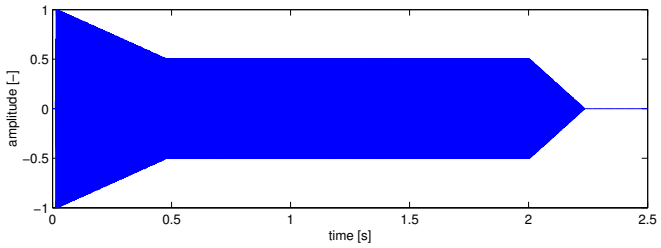

Figure 2.10: Time-Amplitude diagram of sound output for a tone using one oscillator

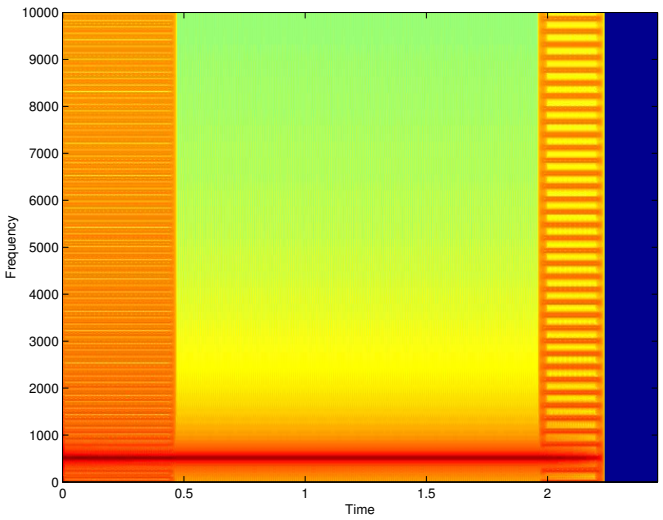

Figure 2.11: Spectrogram of sound output for a tone using one oscillator

noise is reduced for higher resolutions. As a side effect, the timing is slightly changed because increasing the resolution means a reduction of the maximum slope, which limits the minimum attack time. Acoustically, a resolution of 9 bits is enough to reduce the quantisation noise to a level where it is not audible anymore.

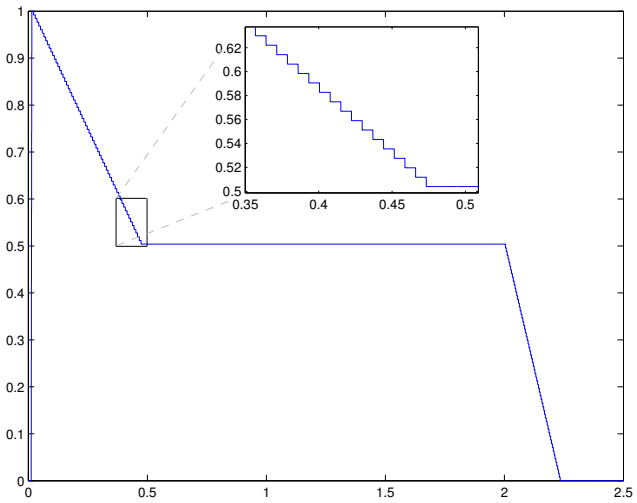

Figure 2.12: Time-Amplitude diagram of the ADSR of the fundamental oscillator section

2.6.2 Several Oscillators

The result of a simulation using 5 oscillators is shown in figure 2.15 as a spectrogram and in figure 2.16 as a 3d-spectrogram. The only difference to the listing in table 2.4 are additional commands for the remaining ADSR parameters and the 4 oscillator volumes, so we abandon a listing of these commands. In the 2d-plot, the lines at 130, 260, 390, 520, and 650 Hz represent the fundamental frequency and 4 harmonic frequencies. The quantization noise produced by the ADSRs is present too, but with significantly less energy. In the 3d-plot, the 5 ADSR envelopes can be seen clearly.

2.6.3 Subsequent Notes

Playing subsequent notes with different note numbers sometimes results in clicks in the signal at the point where the frequencies of the output change. This problem occurs when two notes are too close in time to each other to let the ADSR go back to zero after the first note before the second note is played, because in the model we simply change the argument in a sine function when the frequency shift should be performed. As a consequence we sometimes have large steps in the output signal (see figure 2.17). Ideally each oscillator should wait for the next zero crossing and then perform the frequency shift, starting a new period. Implementing this feature in Simulink is a tedious task as we simply use the sine function at the moment, so we leave this error unresolved in the Simulink model because we would have to implement a special form of the sine function. In the VHDL model where we use a completely different approach for the oscillators, we will take care of this difficulty.

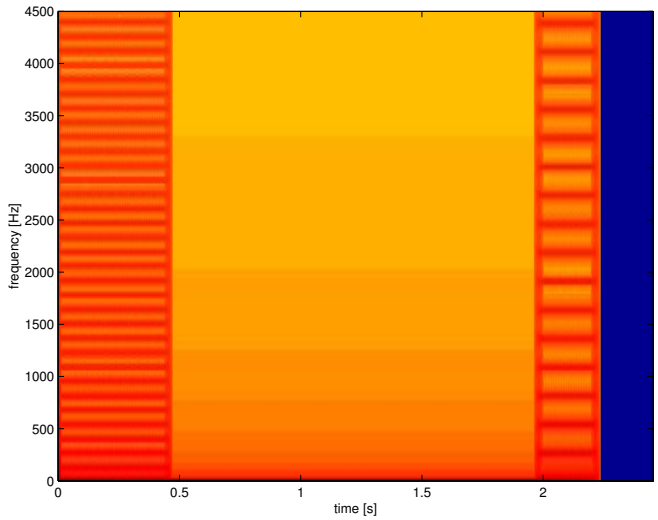

Figure 2.13: Spectrogram of the ADSR of the fundamental oscillator section

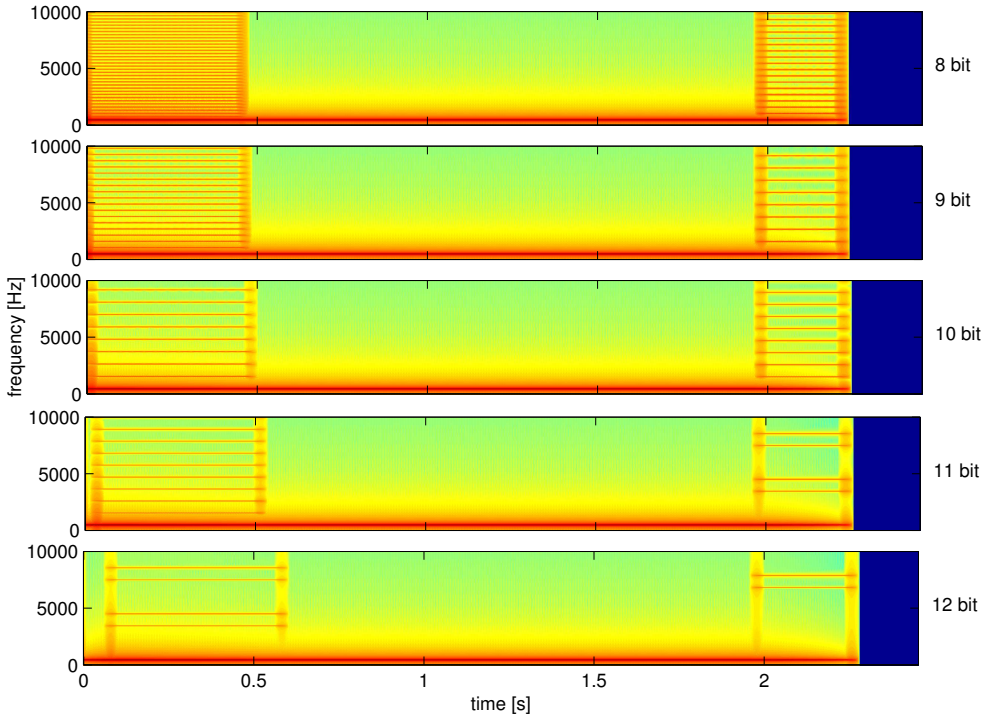

Figure 2.14: Spectrogram of the Signal with ADSR amplitude resolutions of 8-12 bits

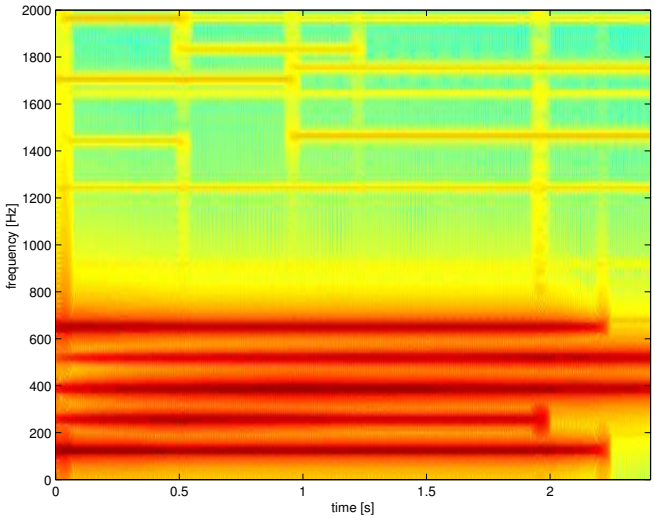

Figure 2.15: Spectrogram of a tone using 5 oscillators

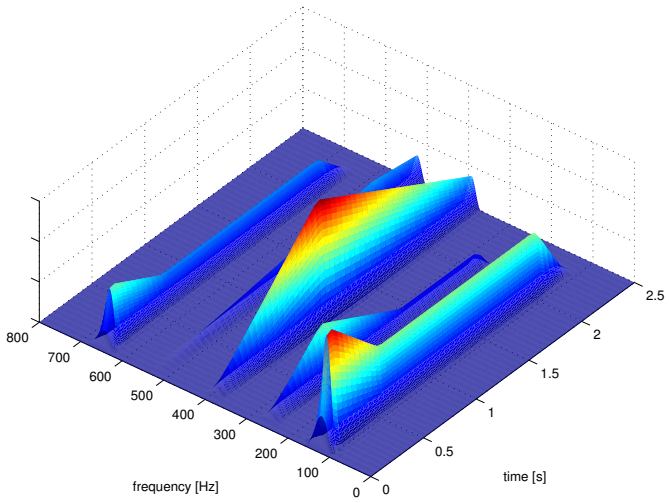

Figure 2.16: 3d plot of the spectrogram data of a tone using 5 oscillators

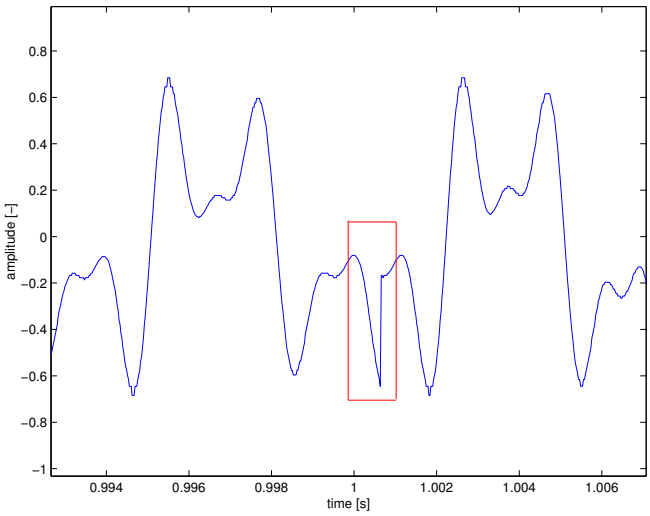

Figure 2.17: Unwanted step in the output signal

Chapter 3

Implementation

3.1 System Overview

Performing a bottom-up approach, we first designed the functional units and then combined them. Working in a modular way like that is very useful when operating in a team. We only had to define the functionality and interface of each block. Then each of us picked a block and designed and tested it until all units were created. Please refer to figure 3.1 to get the *big picture* before proceeding to the following sections.

MIDI data is fed serially to the asynchronous receiver where it is synchronized and handed over to the MIDI controller as parallel bytes. The controller then decides whether the data received is a control, note-on or note-off message. Control data is written to the configuration register, note data is fed to the frequency lookup-table. The frequency lookup table initializes and starts the 8 sine oscillators corresponding to the frequency information contained in the note-on message. The oscillators are the first blocks of 8 almost identical sound generation sections that work in a parallel manner.

The oscillator in such a section forwards the waveform at its output to the rectifier where the sine may be converted to a rectangular waveform. The signal now passes to the complex multiplier where it is multiplied with the main volume, the volume corresponding to the velocity[1], with the volume associated with the section, and with the amplitude envelope generated by the ADSR blocks. Then the waveform is mapped to the stereo panorama, and the section outputs the left and the right channel to the mixers.

The mixers sum up the outputs of all 8 sections and hand the result over to the I^2S-controller which finally outputs the digital sound data in two different formats.

3.2 Asynchronous Receiver

The asynchronous receiver is a Finite State Machine (FSM) that reads the bit-serially incoming MIDI data and writes it to a shift register to generate bit-parallel output. The MIDI byte is received in reverse order, i.e. the start bit is followed by the LSB, and the MSB comes last, prior to stop bit reception. As the global clock is 5.6448 MHz and the clock used in the MIDI protocol is 31.25 kHz $\pm 1\%$, there are

$$\frac{5,644,800}{31,250} = 180.63 \approx 180 \text{ cycles}$$

available for each bit. To synchronize, the asynchronous receiver which is shown in figure 3.3 waits for the signal to change from 1 to 0; this is the start bit. An idle MIDI line is a line with no current flowing, so the signal found at the input is constant 1 due to the inverting character of the optoisolator in the circuit depicted

[1] the velocity the key generating this sound was hit

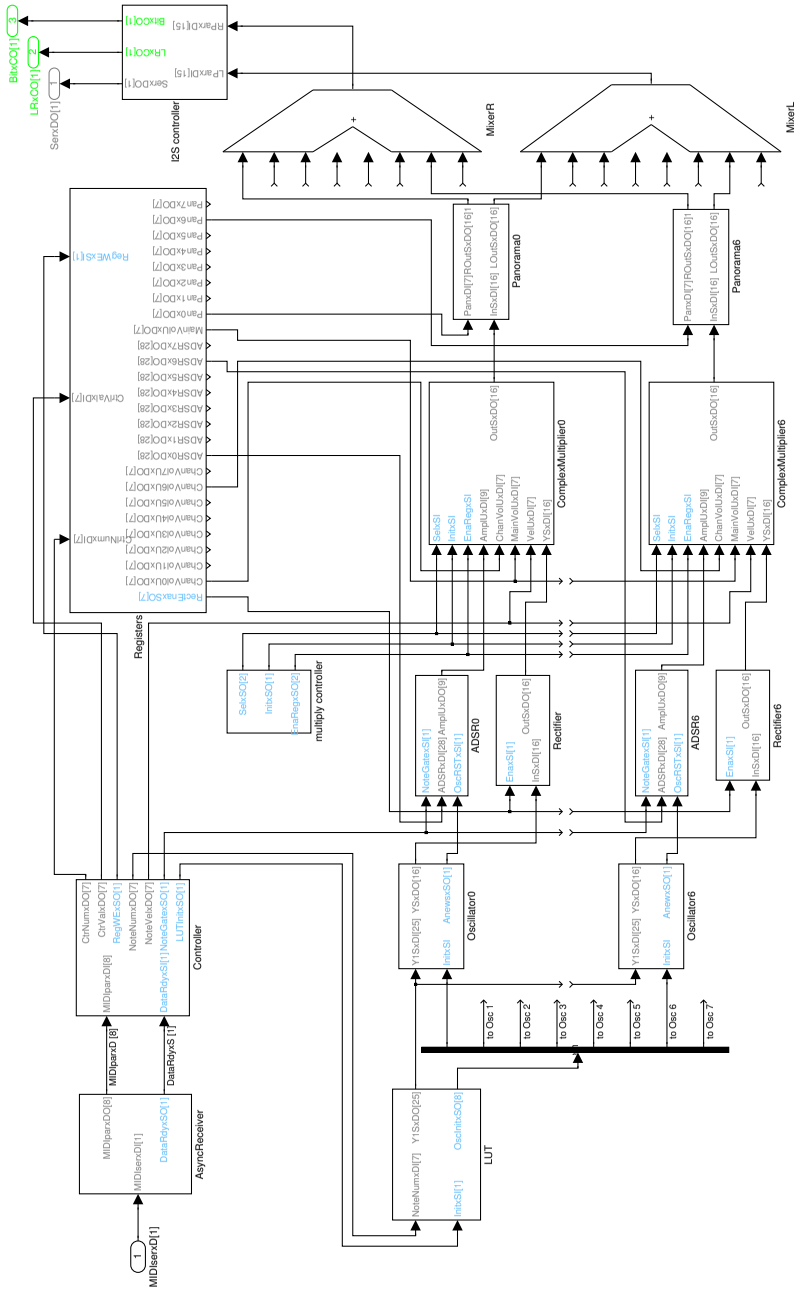

Figure 3.1: Block diagram of the Synthesizer

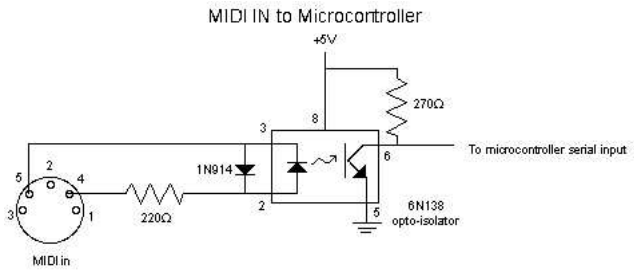

Figure 3.2: Circuit used to connect the MIDI line to the microchip

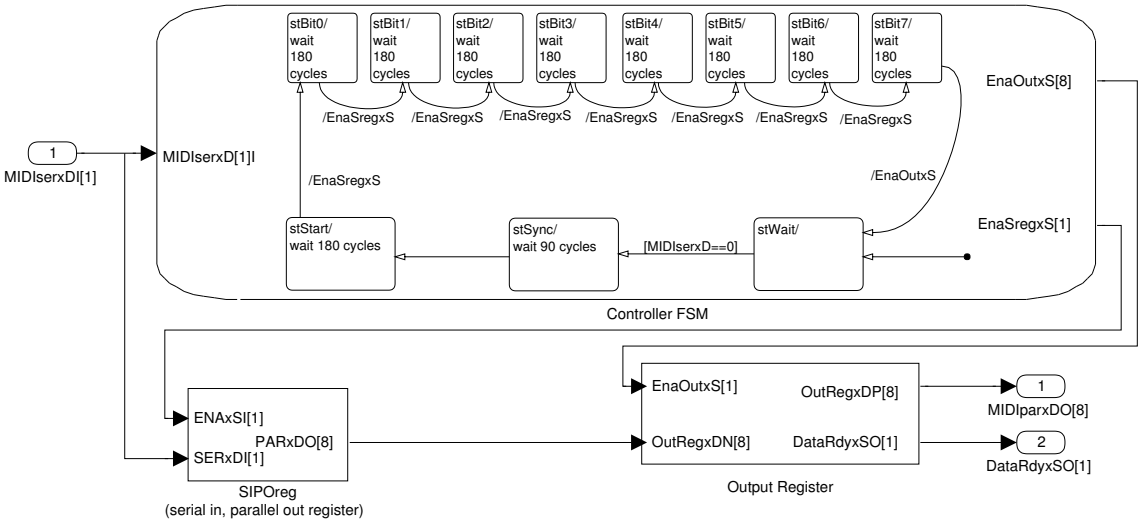

Figure 3.3: Asynchronous receiver

in figure 3.2. It is general practice to use a similar circuit to connect the MIDI cable to the microchip. After receiving the start bit, the receiver waits 90 cycles to synchronize to the center of the pulse, henceforth it reads the signal every 180 cycles and pushes the bits found in this manner into a shift register until the stop bit is found[2], which is the 10^{th} bit and has the value 1. At this moment the byte in the shift register is copied to an output register, and the MIDI controller is signalled to pickup the byte prepared. We can state the following for the bit rate of the incoming MIDI signal:

$$\frac{31,250 \cdot 10 \cdot 180}{90 + 9 \cdot 180} = 29,605 \text{ Hz} < f_{t_{midi}} < 32,894 \text{ Hz} = \frac{31,250 \cdot 9 \cdot 180}{90 + 10 \cdot 180}$$

So this receiver will allow for a tolerance of $\pm 5\%$ in bit rate, which is enough. The VHDL code for this block can be found in `AsyncRecv.vhd` on page 60.

[2]We do not explicitly check for the stop bit. The asynchronous receiver stops at the 10^{th} bit and does not care about its value.

3.3 MIDI Controller

The MIDI controller (see figure 3.4) is an FSM reading a MIDI byte from its input port *MIDIparxDI* when it is told to do so by *DataRdyxSI* being set to 1 and jumps to the appropriate state according to this MIDI byte. Its outputs are hold at a constant value during one clock period using an output register.

Please have a look at the file `MIDIcontroller.vhd` on page 71 for the code discussed in this section.

3.3.1 Note-On Message

If a MIDI byte is a note-on message for the current[3] MIDI channel, its status byte fulfills the condition

$$MIDIxD = MChannelxDI + \text{mNoteOn}$$

where *MIDIxD* is the value of *MIDIparxDI* converted to an unsigned number, *MChannelxDI* is the MIDI channel[4] the FSM listens to, and mNoteOn is a constant set to 144 (0x90) which is the value of a note-on status byte for channel 1. No matter which state is active at the moment, the FSM jumps to the state **stNoteOn** upon reception of such a note-on status byte.

If the next byte is a data byte[5], its value is stored in *NoteNumTempxD*. This value is interpreted as a note number, i.e. the number corresponding to the key pressed on the keyboard. The note number is not propagated to the output yet, as the note velocity, which is the data byte received next, might be zero and so this note-on message would need some special treatment as explained later.

As mentioned, the following data byte represents the key velocity. If the velocity is non-zero, *NoteNumxD* is set to *NoteNumTempxD* and *LUTinitxD* is set to 1 for the next clock period, signalling the frequency lookup-table to reset the oscillators. If *GatexD* is 0, i.e. no other key was pressed before the key the actual note-on message corresponds to, the velocity value is assigned to *NoteVelxD*, otherwise *NoteVelxD* is retained unchanged. Then the FSM enters the state **stNtVel**. If the velocity is zero, this message is interpreted as a note-off message: if the key released is the same as the key pressed last, *GatexD* is set to 0 causing the note to die away, otherwise *GatexD* remains unchanged. Now the state **stNtVel0** gets activated.

The next data byte will be interpreted as the note number for the next note, and the same things happen as in the transition from **stNoteOn** to **stNtNum** before[6], and so on.

3.3.2 Note-Off Message

If a MIDI byte is a note-off status byte and its destination is the channel the FSM listens to, it has the following value:

$$MIDIxD = MChannelxDI + \text{mNoteOff}$$

with the constant mNoteOff equal to 128 (0x80). This byte activates the state **stNoteOff** in the FSM, no matter which state is active at the moment. If a data byte is received next, *GatexD* is set to zero only if the key released has the same note number as the key pressed last; otherwise the former value of *GatexD* is used again. Then the state **stNOffNum** is entered. As soon as the next data byte is received, the FSM jumps to the state **stNOffVel** without doing any assignments because we don't care about the note release velocity. If yet another data byte arrives, the running status starts again at **stNOffNum**, doing the same as in the transition from **stNoteOff** to **stNOffNum**.

[3] The current channel is set externally by the four pins *MIDIChanxDI[4...0]*

[4] the values 0-15 correspond to the channels 1-16

[5] data bytes do have a value less than mDataByte=127 (0x80), which means that their MSB is zero

[6] a so-called running status is entered.

signal name	init value
GatexD/N	0
NoteNumxD/N	36
NoteVelxD/N	0
CtrNumxD/N	0
CtrValxD/N	0
RegWExD/N	0
LUTinitxD/N	0

Table 3.1: Initial values for the outputs of the MIDI controller

3.3.3 Control Change Message

Whatever state is active at the moment, the state **stControl** is entered if a MIDI byte is a status byte for a control change message on the channel the FSM listens to:

$$MIDIxD = MChannelxDI + \text{mContChange}$$

with the constant mContChange equal to 176 (0xB0). The next incoming data byte is the controller number; its value is assigned to *CtrNumxD* and the state **stCtrNum** is entered. The data byte arriving next activates the state **stCtrVal**, setting *CtrValxD* to the value of this byte and telling the register bank to store this value by rising *RegWExD*. Because the FSM is in a running status at the moment, the next data byte restarts the loop and tells the FSM to jump into **stCtrNum**, assigning its value to *CtrNumxD*.

3.3.4 Other Messages

When a reset message arrives, i.e. *MIDIxD* equals mReset which has the value 255 (0xFF), the FSM jumps to the state **stInit**, resetting the signals listed in table 3.1.

This kind of reset we call a *soft reset* because it is synchronous in contrast to the *hard reset* using *RSTxRBI* which is asynchronous. The state **stInit** remains active as long as no other status byte is received.

If a system realtime message (a byte with a value $\geq$mSysReal=248 (0xF8)) is recognized, the presently active state is not left, so messages of that kind are ignored.

If an unimplemented status byte or a status byte for another MIDI channel than the one listened to is received, the state **stSysEx** is entered. The FSM stays there until the next valid and implemented status byte shows up. The states **stInit** and **stSysEx** are not left upon reception of a data byte.

3.4 Configuration Register

Possibly the simplest block on this chip is the configuration register where the controller values are stored. It is organized as a bank of 42 7-bit registers, i.e. one register for each controller number. An RTL sketch of one of these registers is shown in figure 3.5. A number NUM is assigned to each register, which is the controller number the register listens to. If the value of *SELxSI* equals NUM and *ENAxSI* is high at the same moment, the value *REGxDI* gets stored and propagated to *REGxDO*. If the register is reset, its default value is restored. For the default values of the different registers refer to table 3.2. These defaults simply represent a nice sound and do not have a particular meaning.

To reduce the number of output ports of the configuration registers, the outputs of the configuration registers containing the ADSR parameters are combined into one signal in the manner shown in figure 3.6.

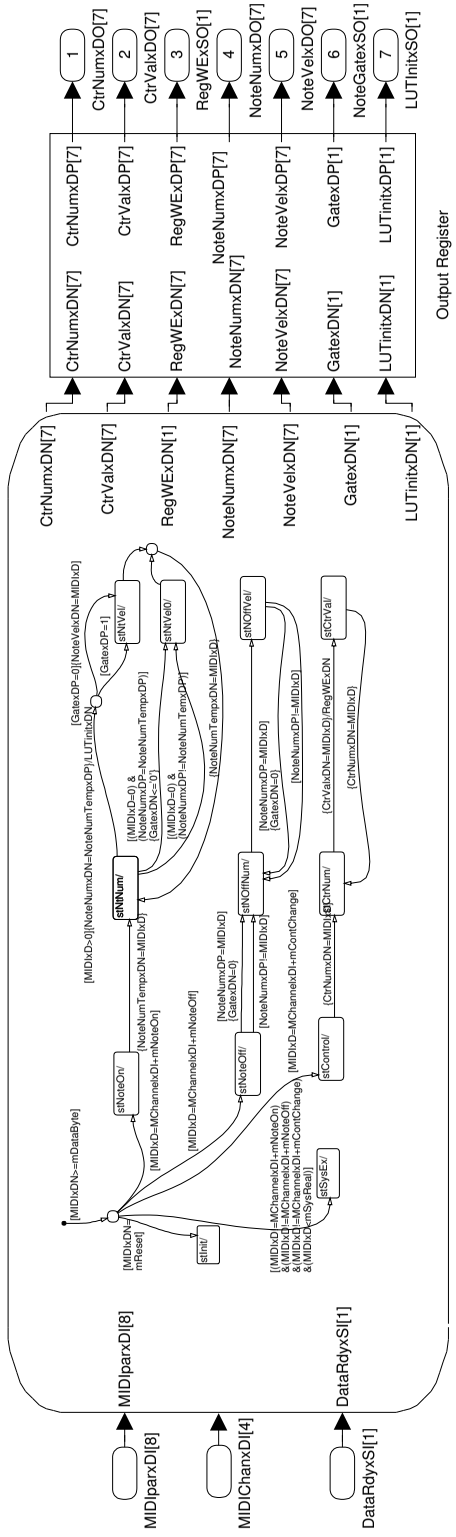

Figure 3.4: MIDI controller

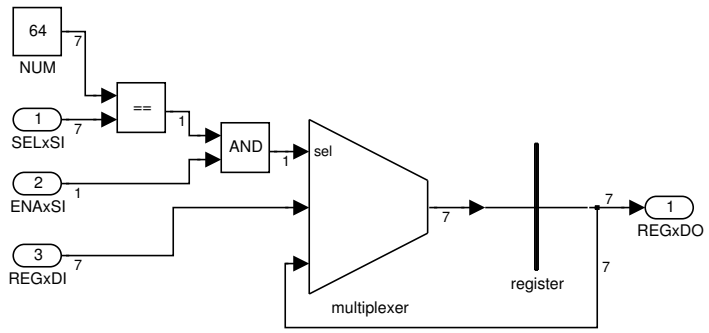

Figure 3.5: 7-bit register used in the register bank

Main volume	100
Sine/Rectangle select	0
Individual Volume 0	127
Individual Volume 1	92
Individual Volume 2	29
Individual Volume 3	52
Individual Volume 4	10
Individual Volume 5	10
Individual Volume 6	0
Individual Volume 7	10

Individual Panorama 0	64
Individual Panorama 1	0
Individual Panorama 2	127
Individual Panorama 3	123
Individual Panorama 4	40
Individual Panorama 5	94
Individual Panorama 6	64
Individual Panorama 7	92

	A	D	S	R
Oscillator 0	27	32	44	63
Oscillator 1	0	9	0	5
Oscillator 2	14	22	0	11
Oscillator 3	127	0	127	3
Oscillator 4	22	35	35	35
Oscillator 5	3	127	0	41
Oscillator 6	0	0	127	0
Oscillator 7	127	48	11	18

Table 3.2: Default values for the registers

3.5 Lookup-Table

The synthesizer is capable of playing 72 notes covering 6 octaves, ranging from notes number 36 (C2) to 107 (B7) [7]. To play a note, each of the eight oscillators needs to be initialized with two 25 bit wide parameters y_1 and c. As storage capacity is limited on the chip, we were looking for the most area-efficient implementation.

A naive approach would simply store all values in a lookup table, requiring

$$72 \text{ notes} \cdot 8 \text{ oscillators} \cdot 2 \text{ parameters} \cdot 25 \text{ bit} = 28,800 \text{ bit} \tag{3.1}$$

To generate a sine wave with the frequency f, an oscillator needs to know

$$y_1 = \sin(\omega \Delta t) \quad \text{and} \quad c = (\omega \Delta t)^2 - 2, \tag{3.2}$$

where $\omega = 2\pi f$.

For small values of $\omega \Delta t$,

$$y_1 = \sin(\omega \Delta t) \approx \omega \Delta t. \tag{3.3}$$

As this approximation turned out to be feasible, the oscillator calculates c itself after initialization.

[7] See appendix C for a complete listing

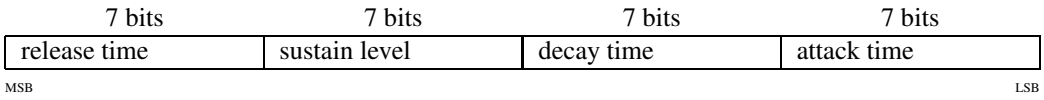

Figure 3.6: Combined ADSR signals

As the initial value y_1 influences only the amplitude of the sine wave and not the frequency, a small error can be allowed. It has been found experimentally that the amplitude error is smaller than 2% for frequencies lower than 10 kHz and at most 10% at 20 kHz, which is well acceptable.

With this knowledge we can reduce the table size to

$$72 \text{ notes} \cdot 8 \text{ oscillators} \cdot 25 \text{ bit} = 14,400 \text{ bit}. \tag{3.4}$$

3.5.1 Indexing

In the simple table mentioned above many values are redundant. One possibility to reduce memory requirements is thus to store every parameter only once and use a separate index table. When limiting the highest frequency to 20 kHz (which is barely audible), 320 different frequencies must be stored (compared to 72·8=576 with a full table). 320 entries require a 9 bit index, so we need

$72 \text{ notes} \cdot 8 \text{ oscillators} \cdot 9 \text{ bit}$	=	5,184 bit for the index
$320 \text{ frequencies} \cdot 25 \text{ bit}$	=	8,000 bit for the table
		13,184 bit.

The reduction from 14,400 to 13,184 bit is not overwhelming enough to justify the extra complexity, so we better stick to the full table.

3.5.2 Generation at Runtime

The approximation of $y_1 \approx \omega \Delta t$ allows for a dramatic reduction in storage requirements. For a given note with base frequency f_0, the ith oscillator is running at $f_i = i f_0$. As the initial value $y_{1,i} = 2\pi f_i \Delta t$ is proportional to f_0, the initial values of all oscillators can be calculated one after another by simple accumulation and don't have to be stored anymore.

See figure 3.7 for an RTL-schematic of the lookup table and the file `LUT.vhd` on page 69 for the VHDL code. Register *Y10xD* holds the initial value of the first oscillator. Further values are accumulated in *AccuxD*. Each bit of *OscInitxS* (not shown in figure 3.7) is connected to the initialization input of one oscillator. After every accumulation, it is shifted by one position, so one after another each oscillator is initialized to play the according frequency.

To obtain accurate results when accumulating $y_{1,1}$ up to the eightfold, 3 additional bits are stored. Storage requirements are thus

$$72 \text{ notes} \cdot 28 \text{ bit} = 2,016 \text{ bit}, \tag{3.5}$$

but as this table is implemented using combinational logic, area requirements depend on the actual values used[8].

[8] and the smartness of the VHDL compiler

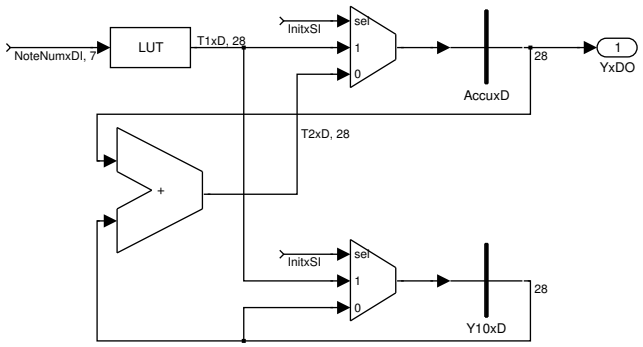

Figure 3.7: RTL model of the accumulator of the Lookup Table

3.6 Oscillators

The synthesizer uses eight sine oscillators running at frequencies ranging from 60 Hz to 20 kHz. To generate a sine wave, the two simple equations

$$\frac{\partial^2}{\partial t^2}\sin(\omega t) = -\omega^2 \sin(\omega t) \tag{3.6}$$

$$\frac{\partial}{\partial t} f(t) \approx \frac{f(t+\frac{\Delta t}{2}) - f(t-\frac{\Delta t}{2})}{\Delta t} \tag{3.7}$$

lead to the recursion

$$\sin(\omega(t+\Delta t)) = -\sin(\omega(t-\Delta t)) + (2-(\omega\Delta t)^2)\sin(\omega t). \tag{3.8}$$

Using $c = 2-(\omega\Delta t)^2$ and $y_k = sin(\omega k \Delta t)$, this can be written as

$$y_{k+1} = cy_k - y_{k-1} \tag{3.9}$$

with the initial values $y_0 = 0$ and $y_1 = \sin(\omega\Delta t)$.

To simplify the calculation of c, we use $\tilde{c} = -c = -2 + (\omega\Delta t)^2$ instead. The reason for this lies in the oscillator's internal number representation, where adding -2 is equivalent to simply setting bit number 24. The modified algorithm produces different outputs $\tilde{y}_k = (-1)^k y_k$, but since we only use every fourth value $y_k^{out} = \tilde{y}_{4k} = (-1)^{4k} y_{4k} = y_{4k}$ the change has no consequences. See below for more information.

3.6.1 Constraints

To implement this algorithm in hardware, a suitable word width b and step size Δt must be chosen. Several conditions must be satisfied:

- Δt must be small enough to enable stable recursion even at high frequencies where only a small number of steps are available for each period. To keep the oscillator circuit as simple as possible, Δt should be a multiple of $1/f_s$.
- b must be big enough to keep errors in amplitude and frequency minimal, but not too big as storage requirements and circuit complexity grow at least linearly with b. Especially at low frequencies, c gets very close to 2 and must be represented precise enough. It has been found experimentally that

a relative frequency error of 0.5 % is not perceptible, so this was chosen as an upper bound for the analysis. Amplitude errors were of no concern as the human hearing is very insensitive to them.

A Matlab/C-model[9] of the algorithm was written and many alternatives tested. To determine the relative frequency error, sample waves at different frequencies were generated and their spectra analysed and compared with the desired result. It has been found that $\Delta t = 1/(4f_s)$ and $b = 25$ with a resolution of 2^{-23} are the optimal parameters.

Of course it must be guaranteed that the recursion formula (3.9) on which our chip relies so heavily is stable. If we had infinite number precision, the poles of the according transfer function would lie precisely on the unit circle. Regrettably, infinite die size was not within budget. In practice, all simulations and the FPGA-prototype worked well even for hours, however this is by far no proof that it works for *all* frequencies and for *all* times.

Unfortunately, we were unable to find a mathematical proof[10] that takes into consideration the step size Δt and the finite word width b. The problem seems to originate in the fact that the rounding operations to consider the finite word widths are non-linear.

A less elegant way to guarantee stability is to look for periodicity. If for a given c and y_1, the recursion (3.9) enters the same state[11] twice, it will repeat this loop forever and remain stable. As the oscillator only plays a number of predefined frequencies, this approach offered a potential brute-force proof which could be performed easily.

A C-program was written that provides a bit-true simulation of the oscillator and the frequency lookup-table. While generating wave-samples, it looks for repetitions whenever two consecutive samples change sign. This simple and straightforward search showed the pleasant result that the oscillator enters a periodic loop for all frequencies and should remain stable. It was interesting to see that some loops only take about 13,000 iterations, whereas the longest loop requires as much as 280 million cycles.

3.6.2 Fixed Point Arithmetic

We denote $C[24:0] = (C[24]\dots C[0])$ as the bit vector representing a given value c in the oscillator and $C[i]$ the i-th bit of it, $C[0]$ being the least significant bit:

$$c = -2C[24] + \sum_{k=0}^{23} 2^{k-23} C[k] \tag{3.10}$$

The valid range of c is thus

$$c \in \{-2, -2+2^{-23}, \dots, -2^{-23}, 0, 2^{-23}, \dots, 2-2^{-23}\} \tag{3.11}$$

All values in the recursion (3.9) are in the same range of (3.11), so we can restrict arithmetical operations to results which are in the valid range. Addition and subtraction are standard operations in this two's complement format. To multiply $z = c \cdot y$, we use

$$z = \underbrace{(C[23:0] \cdot Y[23:0])[47:24]}_{z_1} \underbrace{-2 \cdot (C[24] \cdot Y[23:0] + Y[24] \cdot C[23:0])}_{z_2}. \tag{3.12}$$

The first part, z_1, is calculated by standard unsigned serial multiplication which works as follows: First, set *YsxD* = *YxDI*. At each step,

- shift *YsxD*[23:0] down one bit,

[9] While unbeatable for matrix manipulations, Matlab is orders of magnitude slower than a dedicated C-program when calculating recursions.

[10] If you know a solution, the authors would be most happy if you could drop them a line.

[11] By state we refer to the oscillators internal storage elements, y_k and y_{k-1}.

- shift *AccuxD* down one bit,
- if *YsxD*[0]=1, add *CxD*[23:0] to *AccuxD*.

The second part z_2 is handled by separately adding the negated and shifted value $\mathrm{Y}[23:0]$ or $\mathrm{C}[23:0]$, respectively, to the first part.

Again note that the multiplier yields correct results only if $c \cdot y$ is inside the range of (3.11). An RTL-schematic of the multiplier is shown in figure 3.9, for a description of some internal control signals see table 3.3. The VHDL code of the multiplier is in the file `multiplier25s25s.vhd` on page 74.

3.6.3 Implementation

See figure 3.10 for an RTL-schematic of the circuit which actually generates a sine wave. To initialize, apply *Y1xDI* and set *InitxSI* for one cycle. To run it at a given frequency f, set *Y1xDI* $= \mathrm{round}(2^{23}\, 2\pi f\, \Delta t)$ where $\Delta t = 1/(4 \cdot 44.1\,\mathrm{kHz})$.

When the oscillator receives a *InitxSI* signal to switch to another frequency, it first waits for the current period to finish. Otherwise, a distorting sound would be audible. The registers *Y1BufxD* and *InitBufxS* thus store *Y1xDI* and *InitxSI*, respectively, which are applied externally for one cycle only. The end of a period is detected by waiting for the sign bit of the output register to change from 1 to 0.

As soon as the end of a period is reached, the oscillator starts calculating $c = (\omega\Delta t)^2 - 2 \approx y_1^2 - 2$. This is done by simply feeding the multiplier with *Y1BufxD* and setting bit 24 of the result, which is equivalent to subtracting -2. When *CxD* is thus ready, oscillation can begin. *Y1xD* is set to $(1 - \frac{1}{8}) \times$ *Y1BufxD* because the approximation $y_1 = \sin(\omega\Delta t) \approx \omega\Delta t$ yields a slightly too big y_1 which could cause overflows as values bigger than $2 - 2^{-23}$ cannot be represented.

The VHDL code of the oscillator can be found in the file `oscillator.vhd` on page 78.

3.6.4 Goertzel's Algorithm

The above recursion formula for generating a sine wave is similar to Goertzel's algorithm: Using

$$f[k] = u[k]\sin(\Omega k), \tag{3.13}$$

where $u[k]$ is the unit step function, the z-Transform $F(z)$ of $f[k]$ is

$$F(z) = \frac{\sin(\Omega)z^{-1}}{1 - 2\cos(\Omega)z^{-1} + z^{-2}}. \tag{3.14}$$

As can be seen from the observer canonical form in figure 3.8, the equivalent recursion formula is

$$f[k+1] = 2\cos(\Omega)f[k] - f[k-1] \tag{3.15}$$

with initial values $f[0] = 0$ and $f[1] = \sin(\Omega)$. This is almost the same as (3.9), where $c = 2 - (\omega\Delta t)^2 \approx 2\cos(\omega\Delta t)$.

Goertzel's algorithm uses two initialization values per oscillator, $\sin(\Omega)$ and $\cos(\Omega)$, which would have to be stored separately for every possible frequency. Our approximation (3.3) allows two savings:

- c doesn't have to be stored but is computed at runtime by the oscillators themselves. This wouldn't be possible using eq. (3.15) as $\cos(\Omega)$ cannot easily be calculated given $\sin(\Omega)$.
- As discussed in section 3.5.2, the initial values y_1 for each oscillator are computed by the lookup table upon request by simple accumulation. Goertzel's $\sin(\Omega)$ cannot be calculated as simply. However a small amplitude error could be allowed as the frequency would remain the same, so y_1 could be accumulated too. c would still have to be stored, though.

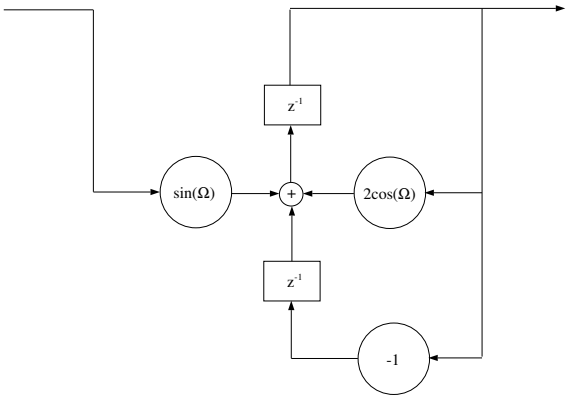

Figure 3.8: Observer canonical form of the Goertzel sine generator

CntxD	*ModexS*	*CYselxS*	*FeedbackxS*	*DonexS*
0 . . . 23	0	x	0	0
24 . . . 28	0	x	1	0
29	1	0	1	0
30	1	1	1	0
31	x	x	x	1

Table 3.3: Internal control signals of the multiplier

Goertzel's recursion 3.15 is more accurate than our approximation and might thus work with smaller word widths. Less area would be required for the initial values as well as the oscillator circuit.

We did not implement Goertzel's algorithm as we discovered it only late in the design process where so many fundamental changes throughout the whole design were out of question. However, we strongly assume that the much bigger storage requirements would not allow a practical realisation.

3.7 Rectifier

The rectifier block described in the file `rectifier.vhd` on page 82 simply passes *InSxDI* to *OutSxDO* if *ENAxSI* is high. Otherwise, -32,767 or 32,767 is passed to *OutsxDO* depending on the sign bit of *InSxDI* which transforms the sine wave into a rectangle wave.

3.8 ADSR Envelope Generator

The implementation of the ADSR envelope generator in VHDL is based on the ADSR used in the Simulink model of the synthesizer.

First, the wide signal *ADSRxDI* as depicted in figure 3.6 is decomposed into its components, namely *AxD*, *DxD*, *SxD* and *RxD*, which stand for attack time, decay time, sustain level and release time, respectively. The meaning of these terms can be seen in figure 3.11: the attack time is the time it takes the amplitude to reach the maximum level, the decay time is the time the amplitude would need to go back to zero if it isn't stopped and hold at the sustain level, and the release time determines the speed at which the amplitude tends towards zero as soon as the key is released if the sustain level was set to the maximum amplitude. So the

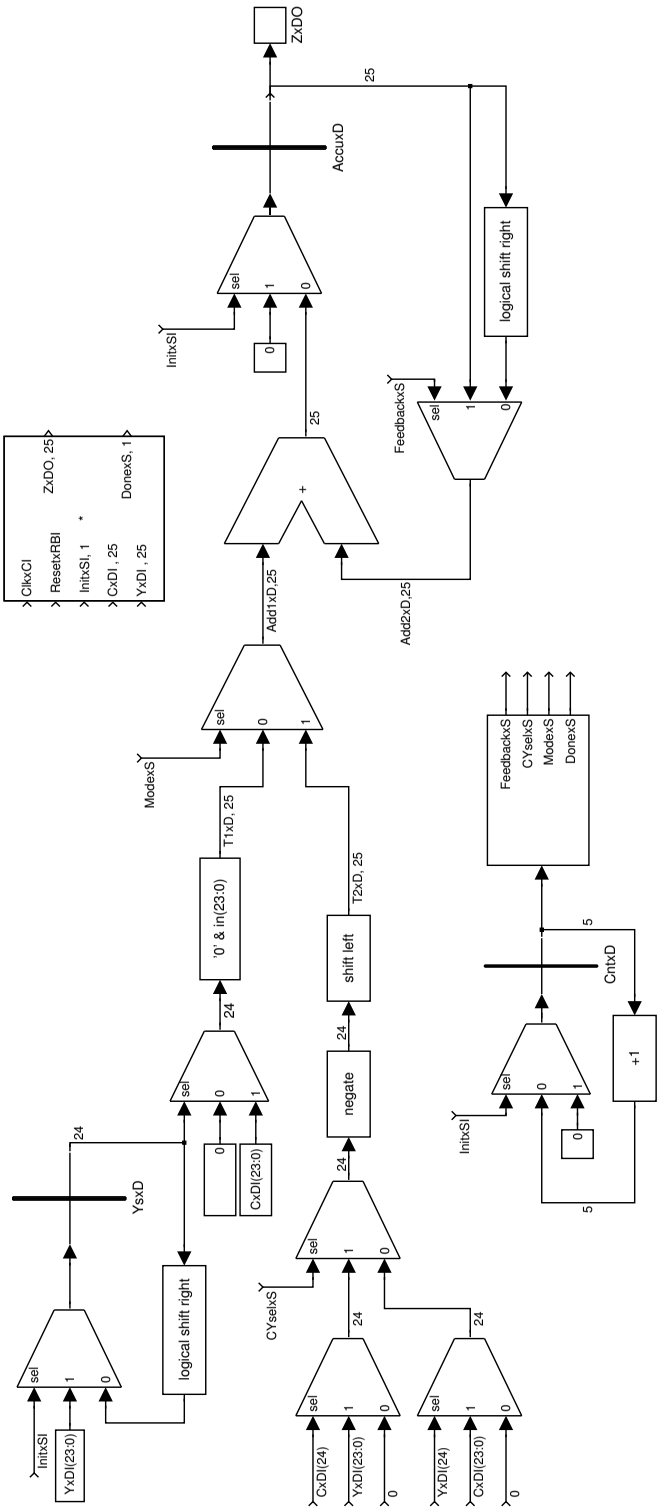

Figure 3.9: RTL model and schematic of the Multiplier

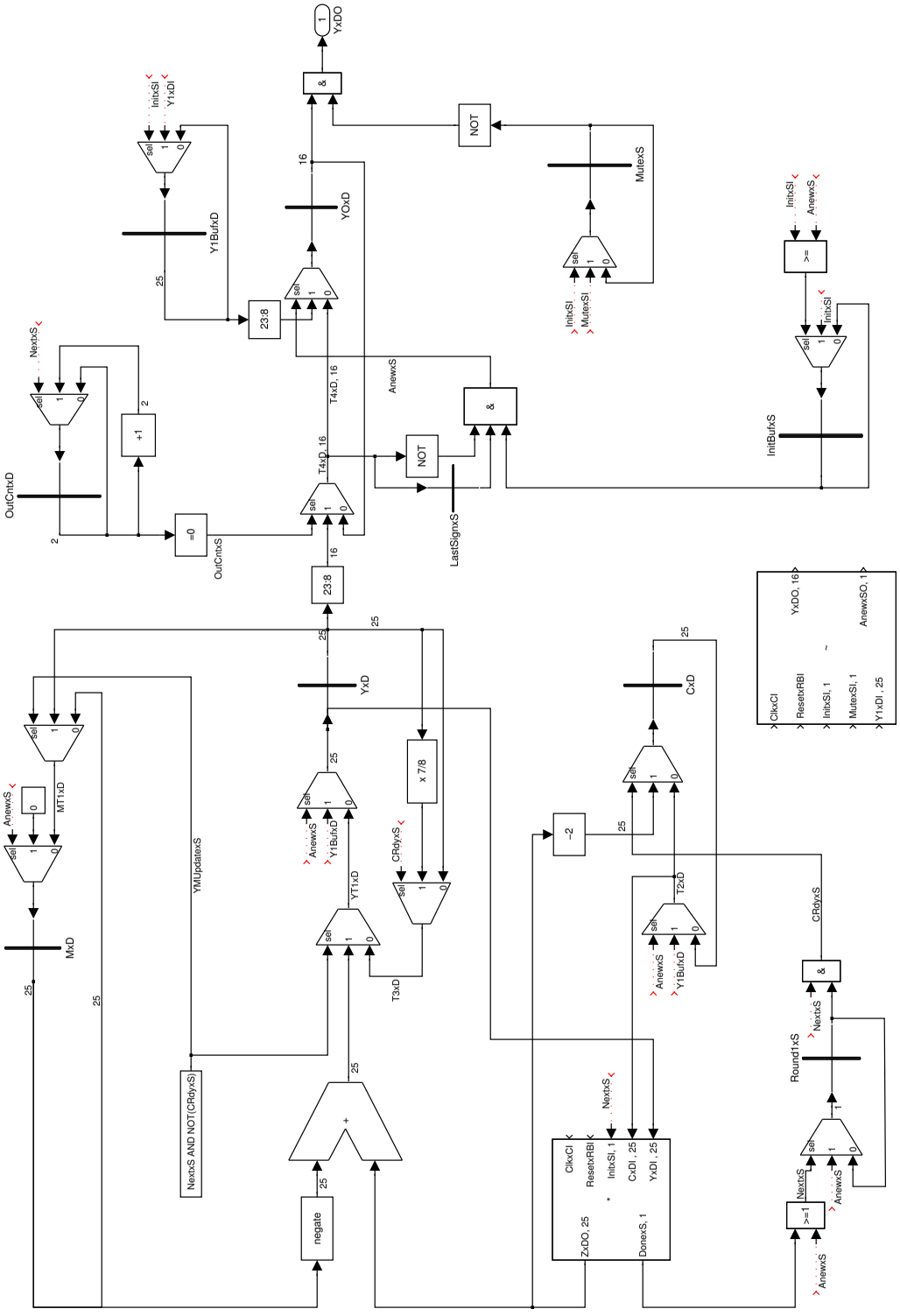

Figure 3.10: RTL model and schematic of the Oscillator

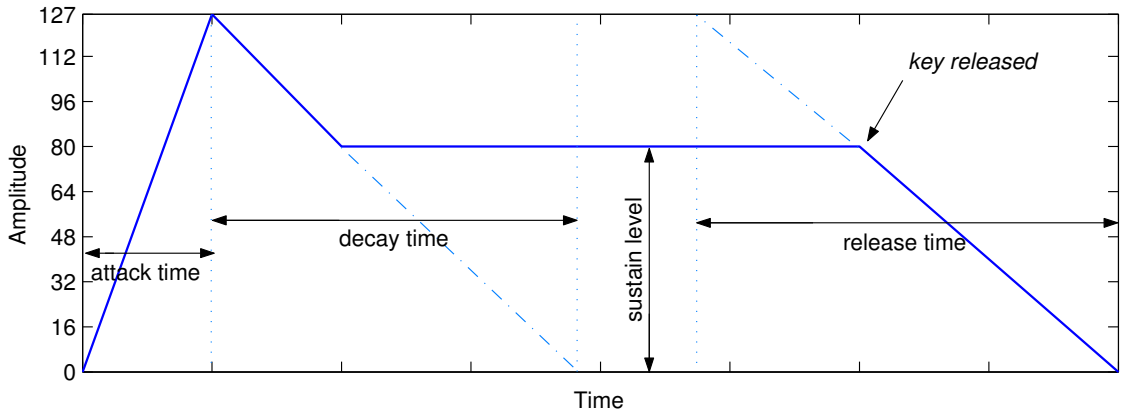

Figure 3.11: Parameters of the ADSR envelope generator

decay and the release time are rather a measure of steepness for the slope of the decay than a time constant. The attack, decay and release times can be adjusted from 0.9 ms to 5.88 s in steps of 46.3 ms which should be sufficient for all practical purposes.

A short description of the FSM representing the ADSR envelope generator as depicted in figure 3.12 follows:

stIdle is the state being active first after a reset or power on. As soon as a key is hit, i.e. *NoteGatexSI* goes to 1, and a note-on message is received, the state **stAttack** is activated and a counter used as a stop watch is started using the signal *CtStartxS*.

stAttack is where the amplitude gets increased according to the attack time *AxD*: as soon as the stop watch counter reaches *AxD*, *AmpxD* is incremented by 1 and the counter is restarted. This step is repeated until *AmpxD* is maximum, then the FSM jumps to **stDecay**, restarting the stop watch. If the key is released before the maximum amplitude is achieved, the state **stRelease** is activated.

stDecay is the state that decreases the amplitude. Each time the stop watch equals *DxD*, *AmpxD* is decremented by 1 and the watch is restarted. Unless the key is released and the FSM jumps to *stRelease*, this process is repeated until *AmpxD* gets at *SxD*, which forces the FSM into the state **stSustain**.

stSustain is a state where the FSM simply waits for the key to be released, and then it jumps to **stRelease**.

stRelease is the point where *AmpxD* fades away completely, i.e. its value is decremented by 1 each time the stop watch equals *RxD*. As soon as 0 is reached, the state ***stIdle*** is activated. If a key is hit during the release phase, **stAttack** is activated again.

The source code for the ADSR can be found in the file `ADSR.vhd` on page 59.

3.9 Complex Multiplier

The name of this block may lead to misinterpretation of its functionality: this is not a multiplier to handle complex numbers. We name it *complex* because it does more than just multiplying two numbers to get a product: it calculates the product of a 16 bit signed number with a 9 bit unsigned and three 7 bit unsigned numbers. Basically it consists of a multiplier taking a 9 bit unsigned number and a 15 bit unsigned number

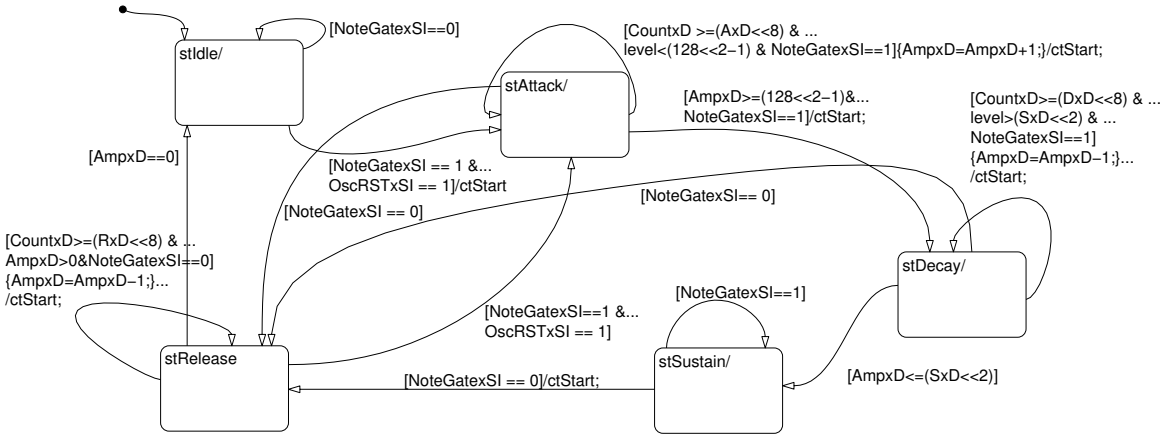

Figure 3.12: FSM representing the ADSR envelope generator

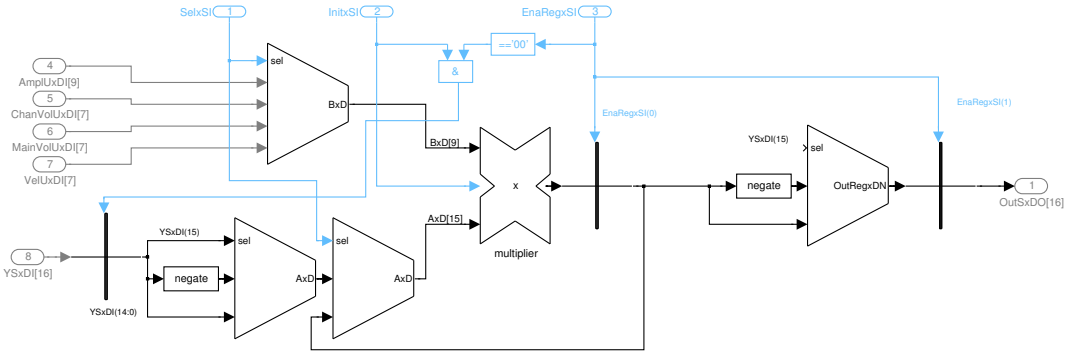

Figure 3.13: RTL diagram of the complex multiplier

(the sign bit is handled separately) as arguments and returning a 15 bit unsigned result. Also, an external counter generating the control signals listed in table 3.4, and some multiplexers are used. Please have a look at figures 3.13 and 3.14, the RTL drafts of the complex and the simple multiplier.

The four multiplications are performed sequentially, so there are four very similar rounds to be finished until the final result is available at the output.

In the first round *SelxSI* selects *AmplUxDI* as the first operand and *YSxDI* as the second one. *YSxDI* is negated if its value is negative, so we can proceed as if it was a positive number, using only 15 bits without the sign bit. For the next clock cycle *InitxSI* is high, which tells the simple multiplier to start the calculation. During the next 9 cycles, the first intermediate result is worked out, which is stored in the register *MulRegxD* as soon as bit 0 of *EnaRegxSI* is 1.

This is the point where the second round is started and *SelxSI* selects *ChanVolUxDI* for the first operand and the result of the preceding operation, *MulRegxD*, for the second operand. As *MulRegxD* is unsigned, there is no need for the negation performed in the first round, but the first operand, a 7 bit unsigned number, needs to be extended to a 9 bit unsigned number.

Simply shifting this number to the left by two positions leads to an error for large values: a value of 127 for *ChanVolUxDI* means *maximum volume*; converted to a 9 bit number, this should be 511, but shifting two

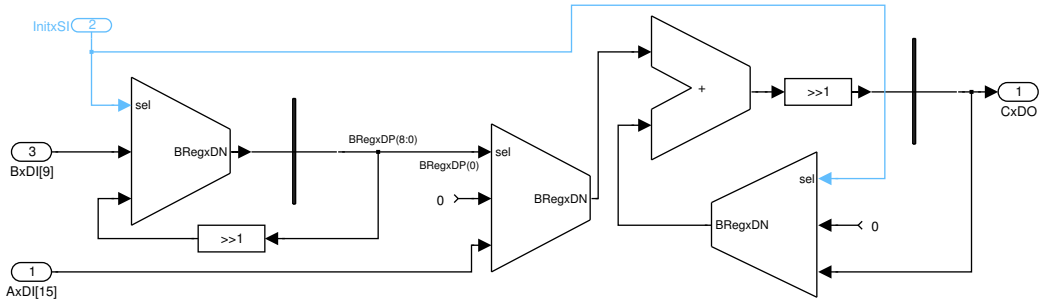

Figure 3.14: RTL diagram of the multiplier used in the complex multiplier

signal	width	description
SelxSI	2	selects *AmplUxDI*, *ChanVolUxDI*, *MainVolUxDI* or *VelUxDI* for the first operand and *YSxDI* or the result of the preceding multiplication for the second operand
InitxSI	1	starts the simple multiplier
EnaRegxSI	2	bit 0 enables the register after the simple multiplier, *MulRegxD*, bit 1 enables the output register, *OutRegxD*

Table 3.4: Control signals used in the complex multiplier

positions yields 508. To avoid this error, we do not simply append two 0s, but the two most significant bits. As a result, the error is spread over the whole range of values for the 9 bit number, and the maximum volume is 511 as desired.

The same considerations can be made for the third and the fourth round. When the fourth round terminates, bit 1 of *EnaRegxSI* is high which tells the output register to store the final result, which is negated first if *YSxDI* was negative. As the sign of *YSxDI* must not change during the whole calculation, *YSxDI* is buffered in an input register.

The result of the complex multiplication can be read at the output *OutSxDO* after 46 clock cycles. A time plot of the control signals during this multiplication process is provided in figure 3.15. The VHDL code for the abovementioned blocks is in the files `ComplexMultiplier.vhd` on page 65, `mul15u9u.vhd` on page 77 and `MultiplyController.vhd` on page 73.

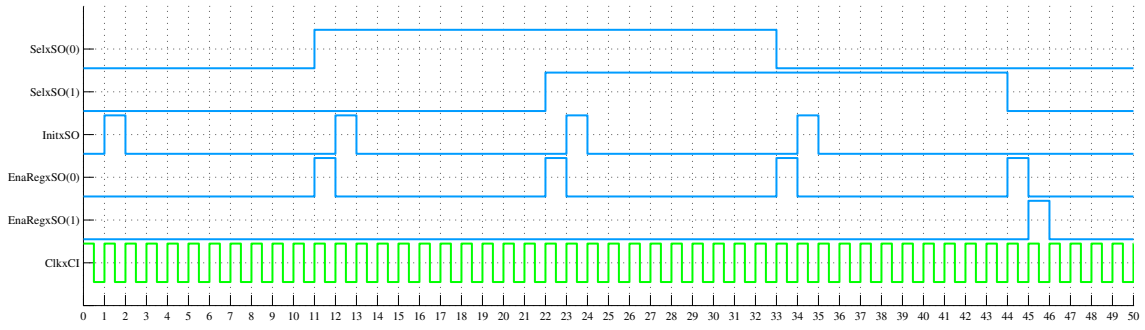

Figure 3.15: Control signals generated by the multplier controller

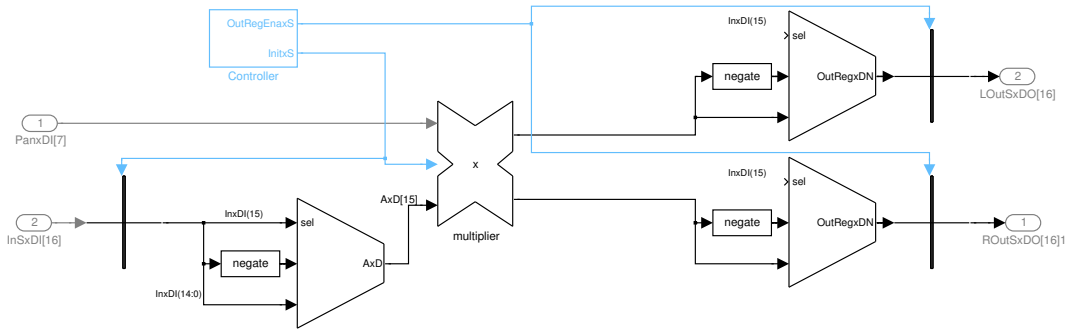

Figure 3.16: RTL sketch of the Panorama Controller

3.10 Panorama Controller

The Panorama Controller's job is to place each of the 8 so-far monophonic oscillator sections in the stereo panorama. To describe the position of a channel, we use a "panning factor" ranging from 0 (fully left) to 127 (fully right).

As the range of 0 to 127 is of even cardinality, one of the following options must be chosen:

- Represent *fully left* and *fully right* with a panning factor of 0 and 127, respectively, being unable to represent *center* exactly due to the even cardinality of the factor range.
- Exactly represent *center* using either 63 or 64, being unable to represent *fully right* or *fully left* exactly.
- Discard either the highest or the lowest value of the range to have a cardinality of 127, which is odd, enabling us to represent *fully left*, *fully right* and *center* exactly.

We chose the first option because this yields the simplest multiplication algorithm and the misalignment of *center* is not audible. Basically, the following calculation is performed:

$$\begin{aligned} \mathit{ROutSxDO} &= \mathit{InSxDI} \cdot \mathit{PanUxDI} \\ \mathit{LOutSxDO} &= \mathit{InSxDI} \cdot (127 - \mathit{PanUxDI}) \end{aligned}$$

The functionality of the multiplier used in the Panorama Controller is based on the multiplier in the Complex Multiplier, but it takes advantage of the fact that the products of the 15 bit number with a 7 bit number and the bitwise inverted 7 bit number can be obtained simultaneously at the extra cost of only one accumulator, as can be seen in the RTL sketch in figure 3.17.

The calculations are controlled by a counter spawning *InitxS* at clock cycle 1 to initialize the multiplier and to store *InSxDI* in the input register *InRegxD*, and *EnaOutRegxS* at clock cycle 9 to store the results *C1xD* and *C2xD* in the output registers *ROutRegxD* and *LOutRegxD*. As done in the Complex Multiplier, the sign is removed from the audio input and applied again to the audio output.

The VHDL code can be found in `Panorama.vhd` on page 80 and `mul15u7uc.vhd` on page 76.

3.11 Mixing Stages

The final stages in the synthesizer are the mixing stages for the left and right audio channel which add all eight – now stereophonic – oscillator section to form the output signal. Eight 16 bit signals of the Complex

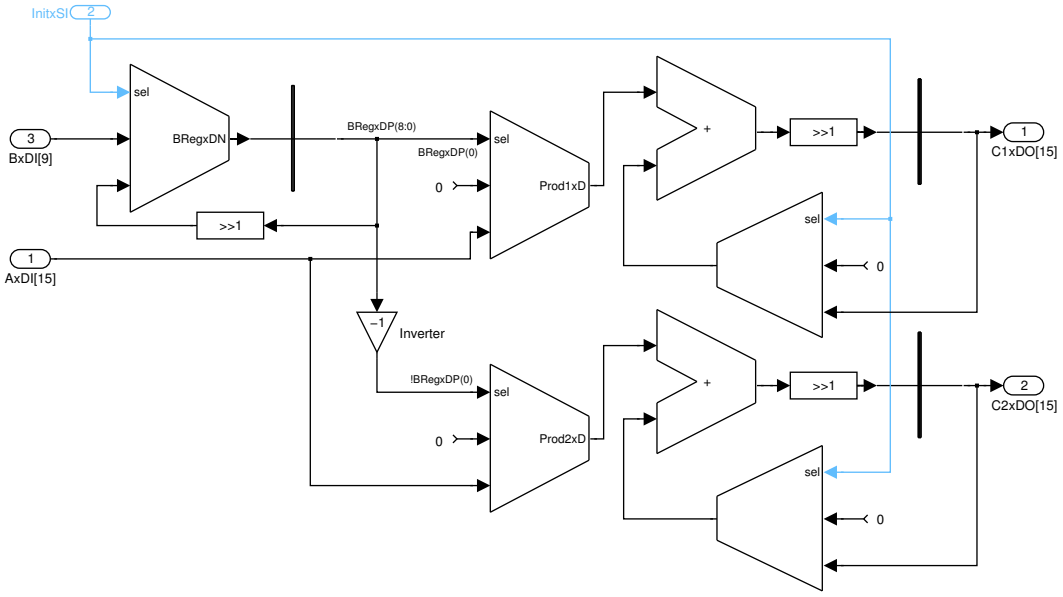

Figure 3.17: RTL sketch of the multiplier used in the Panorama Controller

signal	width	description
InitxS	1	starts the multiplier and enables the input register
EnaOutRegxS	1	enables the output registers

Table 3.5: Control signals used in the Panorama Controller

Multipliers are taken as inputs, added and output. As the output is sampled at 44.1 kHz which is the 128th fold of the internal chip clock frequency, the mixing stage adds its eight signals and waits for the current sample period to finish.

See figure 3.18 for an RTL-schematic of the mixing stage. Not shown is the 7 bit counter *CntxD* which controls the signals shown in table 3.6.

3.12 I²S-controller and DAC

To communicate with the externally connected digital-to-analog converter (DAC) of choice, the PCM1725 by Burr-Brown, we need a block that implements the I²S protocol. In the data sheet of this device [3] the following can be read:

CntxD	*MuxSelxS*	*AccuInitxS*	*OutInitxS*
0 . . . 7	0 . . . 7	0	0
8	x	1	1
9 . . . 127	x	1	0

Table 3.6: Internal control signals of the mixing stage

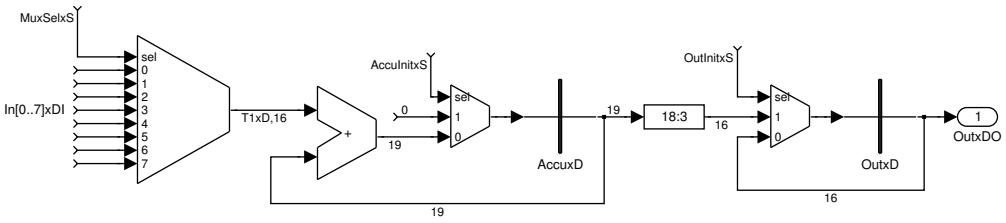

Figure 3.18: RTL model of the mixing stage

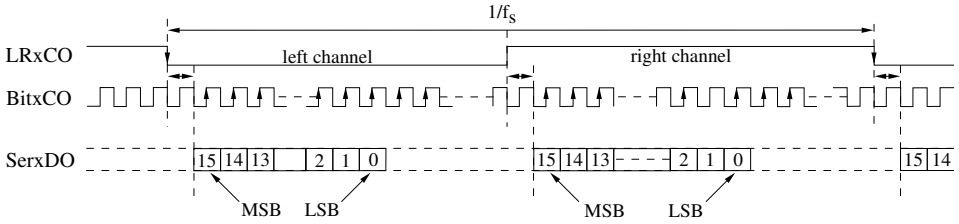

Figure 3.19: I^2S data input timing

> The PCM1725 is a complete low cost stereo DAC, operating off of a $256f_s$ or $384f_s$ system clock. The DAC contains a 3^{rd}-order $\Sigma\Delta$ modulator, a digital interpolation filter and an analog output amplifier. The PCM1725 accepts 16 bit input data either in normal or I^2S-formats. The digital filter performs an 8x interpolation function and includes de-emphasis at 44.1 kHz. The PCM1725 can accept digital audio sampling frequencies from 16 kHz to 96 kHz, always at 8x oversampling.

We decided to use the I^2S-format because it is the most widely used. The I^2S-controller provides two clocks: the left-right-clock *LRxCO* and the bit clock *BitxCO*. Also, it outputs the audio data serially at *SerxDO*.

As a consequence of the fact that the PCM needs a system clock of

$$256 \cdot f_s = 256 \cdot 44100\,\text{Hz} = 11.2896\,\text{MHz}$$

we need a clock divider for our chip to use the same clock oscillator for the Synthesizer and the DAC, because the Synthesizer works at $128f_s$.

First, *LRxCO* is low for 64 clock cycles, telling the PCM1725 that the data for the left channel is being transmitted next. *BitxCO* is a clock that is half as fast as the chip's internal clock. During the first cycle of *BitxCO*, no audio data is sent. On the next falling edge, the MSB of the audio data is present on *SerxDO*, and on each following falling edge the next bit is sent until all 16 bits are transmitted. Then *SerxDO* is idle for the remaining 15 cycles of *BitxCO*. Next, *LRxCO* is set to high for 64 clock cycles, and the data for the right channel is being transmitted.

Due to the fact that no accurate DA-converter was at hand when we downloaded a prototype to an FPGA (see section 4.5), we had to substitute the PCM1725 by the AD1856 and to implement the timing behaviour (figure 3.20) of this device too. This addendum only needs two more ports, namely *LLExSO* and *RLExSO*, so we decided to use this extended compatibility not only for the FPGA and to implement it in the final design.

The data is clocked into the DA-converter on positive edges of *BitxCO* and is latched into its input register on the negative going latch enable *LLExSO* for the left channel and *RLExSO* for the right channel.

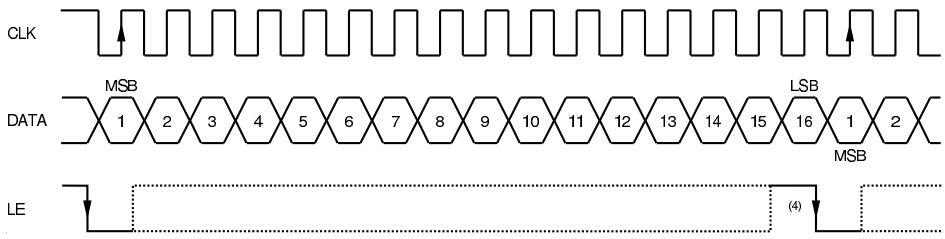

Figure 3.20: Data input timing for the AD1856

The dotted lines in the *LE* signal in figure 3.20 mean that the latch enable signals can be either low or high during that phase; we decided to keep them high.

This controller is implemented using a simple counter and a PISO shift register; the VHDL code can be found in `i2scontroller.vhd` on page 68.

3.13 Chip

The top entity containing all functional blocks has to be embedded in a chip entity defining the pads and synchronization facilities for asynchronous inputs. Here, we only have to bother with the asynchronous reset, even though the MIDI input and the MIDI channel select signals and also the clock select signal are asynchronous. There is no need to synchronize the MIDI input as this task is performed in the asynchronous receiver. Furthermore, the MIDI channel select signals and the clock select signal are not synchronized because normally they are set at the beginning and left untouched during operation.

Another significant feature we place in the chip entity is the clock divider which is needed to adapt the system clock to the clock required by the DAC (see section 3.12 for further details). This clock divider can be disabled by setting *CLKSelxSI*. As it makes no sense to divide the clock during a scan test, we combine the select signal for the clock divider with *ScanEnxT* with a logical OR to disable the clock divider for that special operation mode.

As the reset signal for the clock divider would be connected to the reset tree, resulting in a huge delay with possible hold violations as a consequence when the reset is released, leaving the dividing flipflop in a metastable state. As the clock is too important a signal, this violation must be avoided by all means, so a second reset synchronizer for the clock divider only is needed.

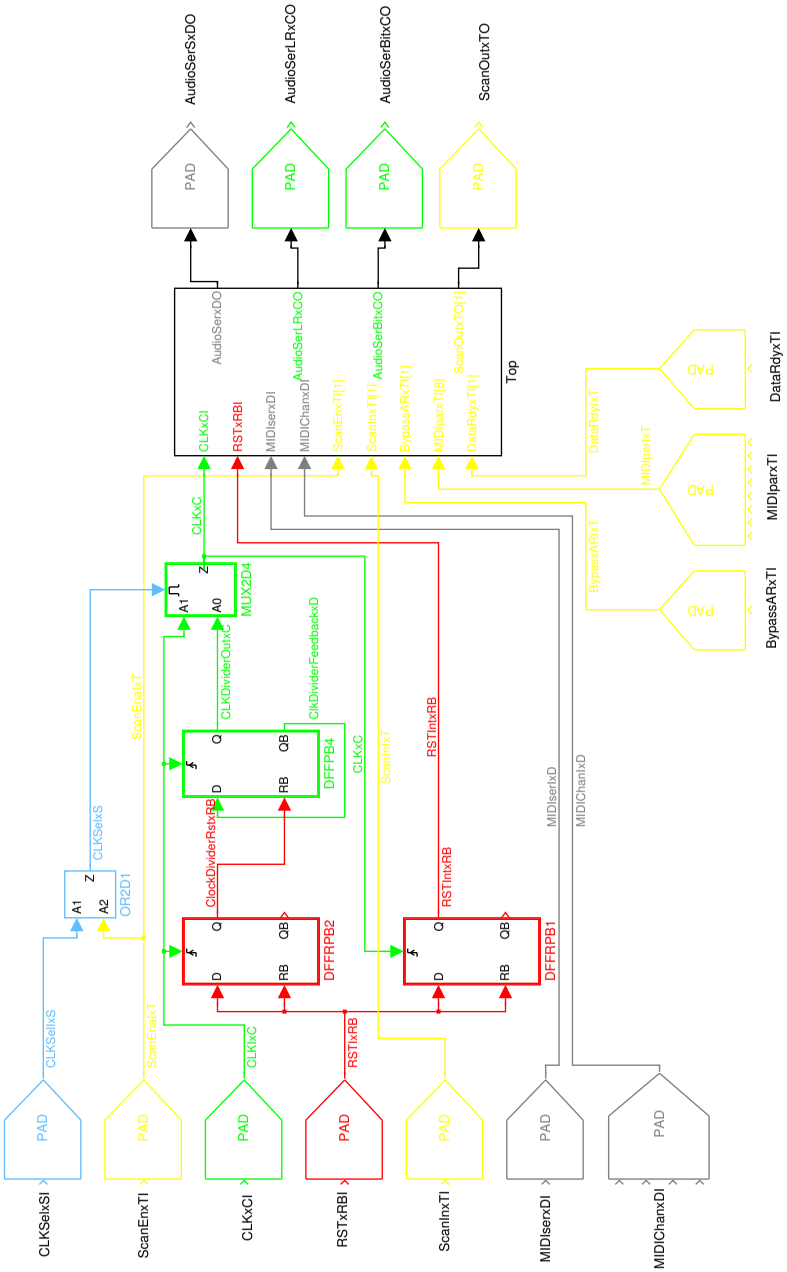

Figure 3.21: Chip with pads

Chapter 4

Testing

4.1 Test Vector Generation

4.1.1 Test Vectors for the Asynchronous Receiver

Because the analog signal at the MIDI input is sampled at the internal clock frequency (11.2896 MHz), we need a huge amount of test vectors to verify the functionality of the asynchronous receiver. Writing a file containing 11,289,600 entries for a single second of simulation time is not practical, so we designed the graphical user interface shown in figure 4.1 to compose the data stream needed. This GUI offers four possibilities to add MIDI bits and bytes to the data stream:

- note-on and note-off messages
- control change messages
- arbitrary bytes, e.g. to compose system exclusive messages
- repeated bits to introduce pauses or errors

A preview of the data stream generated is shown in the GUI and updated on each addition of bytes or bits. The data can be stored in a file containing stimuli for the testbench. Also, a file containing the expected responses consisting of the bytes that should be recognized correctly[1] is generated. The file with the stimuli has two columns; the first column is one bit for the reset signal, the second column one bit for the actual level of the MIDI input.

4.2 Testbench

To simulate and verify the synthesizer, a testbench is required which provides an environment similar to the real world: The fabricated chip will be soldered on a board and connected to a keyboard or a computer from which it receives a MIDI bit stream. The output will be fed to an amplifier and then played to the stunned audience.

As the synthesizer is fed by a real-time bit stream, a testbench could either

- store the complete input stream sampled at the MIDI bit rate, or
- add timing information to each byte and store necessary events only.

[1] one byte per line

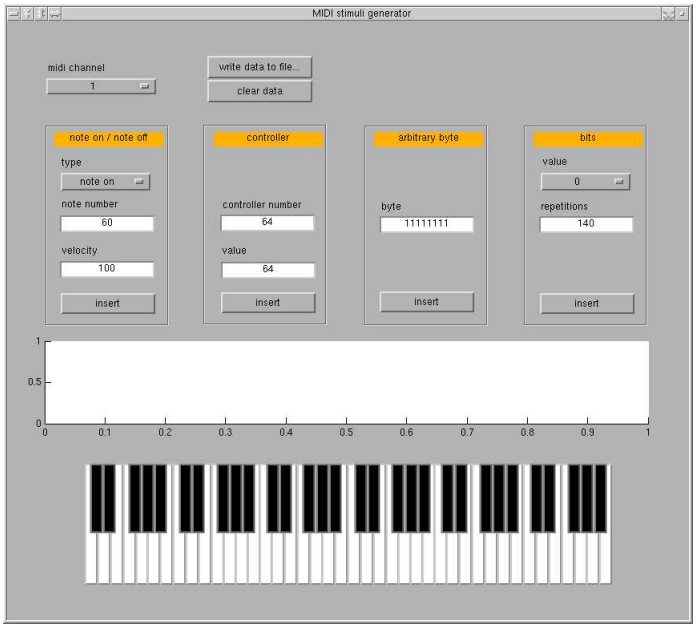

Figure 4.1: Graphical interface to generate test vectors

Usually a MIDI bit stream is not utilized all the time but only when keys are pressed and released. Additional control change messages may be transmitted too, but still the first approach would be a waste of memory as mostly padding zeros would be stored.

For the second approach, there already exists a standardized solution. The Standard MIDI File (SMF) format puts MIDI events together with their timing information in a file.

4.2.1 Standard MIDI File Format

An SMF consists of a header identifying the file, tracks and tempo with which the data bytes are to be transmitted. A track is made up of a sequence of events, which each includes a time quantity to wait for after the previous event. The main part of an event is a message which is either transmitted directly to the synthesizer or is interpreted as a non-MIDI event. Normal MIDI events include note on/off, SysEx, aftertouch, control change etc. Non-MIDI events in contrast set the command channel, tempo, or add some proprietary data to the file.

The testbench ignores all non-MIDI events except tempo change. Normal MIDI messages are simply sent to the synthesizer which is capable of handling all possible events.

4.2.2 VHDL Testbench

We use a traffic light-like simulation control: A traffic light signal is set to orange at the beginning while the testbench initializes. The traffic light then changes to green. When this happens, all other processes begin to work, including:

- Clock generation
- Timing calculations: Timing information in an SMF is given by preceeding each MIDI message by an additional byte. For this value, the unit “parts” is used. Of course the duration of one such “part” must

be known. For example, if one part is 10 μs and we read 5 as the timing byte before a message, we simply wait for 50 μs before sending the command.

MIDI doesn't use parts, however. Two other measurements are used, "parts per quarter note" (PPQN) and "Tempo" (μs per quarter note). Whenever timing information changes, a separate process calculates μs/part = Tempo / PPQN.

- Audio output logger: The only output we have is the serial bit stream in I^2S-format with its bit and channel clocks. Instead of simply writing all these outputs to a file we emulate an I^2S-receiver inside the testbench which is an easy task. The stereo audio data is written as two columns to a text file which can be read by e.g. Matlab.

- MIDI interface: To send a MIDI byte to the synthesizer, a separate process is used. Two semaphores *ToggleToSendxS* and *ToggleWhenSentxS* and a data byte *SendMexD* serve to communicate with the process. The semaphores do what their names say: If you want to send a byte, set *SendMexD* to the according value, toggle *ToggleToSendxS* and wait until *ToggleWhenSentxS* changes. In the meantime, the sender takes care of the MIDI protocol: Send the start bit, the eight data bits, then the stop bit.

- File parser: A SMF is read and analyzed which is a fairly straightforward procedure. Generally spoken the whole job consists of parsing the file and track headers followed by a huge `case`-statement which takes care of all possible command bytes that may occur in an SMF.

An additional feature of the testbench is the validation of the scan-chain. At a specific clock cycle, the entire testbench is stopped, the scan-enabling input *ScanEnxTI* set and the scan-test started. For as many clock cycles as there are registers[2], the output of the scan-chain *ScanOutxTO* is fed back into *ScanInxTI* and at the same time stored in a vector. This procedure is repeated once, resulting in another vector of all on-chip registers. If the two vectors are equal, the scan-chain is implemented correctly. The testbench then continues normal operation which should of course generate the same output as if the scan-chain validation had not been performed.

See the file `testbench.vhd` on page 83 for further details. A quite useful website with some documentation on SMFs can be found at [6].

4.3 Automated Test Equipment

4.3.1 General Considerations

For post-production testing, an HP83000 ASIC Verification System is available. The fabricated chip will be connected to the ATE which is fed with simulation data. For each clock cycle to perform, all input pins are set to the given values and all output pins are checked against the predefined results.

In mass-production the time required for testing one unit should be as short as possible: in a minimum number of clock cycles the highest possible fault coverage should be obtained.

There are two types of tests, functional ones and scan-tests. During functional tests, the chip is fed with stimuli data as it would receive input during normal operation. scan-tests in contrast directly access register contents to locate possible production faults.

To suit a design to scan-tests, almost no design-specific work is necessary. For the sake of efficient routing, the sequence of the scannable registers may be set in a way to minimize interconnect delays. Multiple scan chains may be considered to facilitate access to different units. Due to the simplicity and small size of our design, we decided not to further change anything about Synopsys' automatically generated scan-chain.

[2] there are 3575 scannable one-bit registers, excluding reset synchronization and clock divider registers

Functional testing is a different story however. To be efficient, special precautions have to be taken for most applications. Our synthesizer has one major flaw: In normal operation it will receive some MIDI messages and generate the according sound. Then it will pause for eventually many seconds until it is used again. At its clock frequency of 5.6448 MHz, many million cycles will thus pass which is far too much for efficient testing. There are three main reasons for this problem:

- Slow human interaction. Normally no more than a few MIDI messages are sent per second as even the most gifted musician is orders of magnitude slower than the chip clock.
- Slow MIDI protocol. The MIDI protocol is very slow in respect to the system clock. Every bit sent takes 180 clock cycles, which means that an entire MIDI byte including start and stop bits is 1800 clock cycles in duration.
- Low audio frequency. For every audio sample that is output, 128 clock cycles pass.

We regarded the first two items as the most important ones. Firstly, there is no need to imitate human behaviour when performing a functional test, because we only want to know whether the chip works or not; we do not need a beautiful sound at the output. So we may send MIDI commands much faster than a human ever could to test more functionality in the same time. Secondly, for testing purposes we may neglect the MIDI protocol and feed the MIDI bytes much faster to the controller. The third item cannot be improved as these 128 clock cycles are needed for calculating purposes.

4.3.2 Bypassing the Asynchronous Receiver

As stated above, MIDI data is fed to the controller overly slow. The speed of the incoming data is limited by the asynchronous receiver which expects the data rate at its input to be 31.25 kHz. The asynchronous receiver converts the serial input to parallel data bytes and hands them over to the MIDI controller. This controller is not bound to the MIDI data rate anymore and can be fed at system clock speed.

To bypass the receiver, *BypassARxT* is set to 1 which results in *MIDIparxT* and *DataRdyxT* being directly connected to the MIDI controller. To send a MIDI byte to the controller, the byte must be present at *MIDIparxT* and *DataRdyxT* high for one period. Theoretically, this approach makes it possible to send one MIDI byte per clock cycle, but normally we will keep it slower.

Whenever the asynchronous receiver is bypassed, its functionality is not tested. A separate test should thus be performed with the bypass disabled to verify that the asynchronous receiver really works, but this test may include only a few MIDI bytes.

4.4 Emagic Logic Frontend

To generate the Standard MIDI Files for the testbench we use the sequencer software *Logic* by Emagic. In this program we can graphically compose the MIDI data. To make the handling of the controllers easier, a graphical frontend (see figure 4.3) has been developed.

4.5 Rapid prototyping using an FPGA

4.5.1 Creating the Prototype

As the size of the final design is small enough to fit on an FPGA and simulation times rised above acceptable limits, we created a rapid prototype of our chip. The tools used for that purpose were *Synplify* by Synplicity to compile the code and the *Xilinx Design Manager* to implement it on the FPGA *Virtex XCV400* by Xilinx. The FPGA was soldered on a test board of the laboratory.

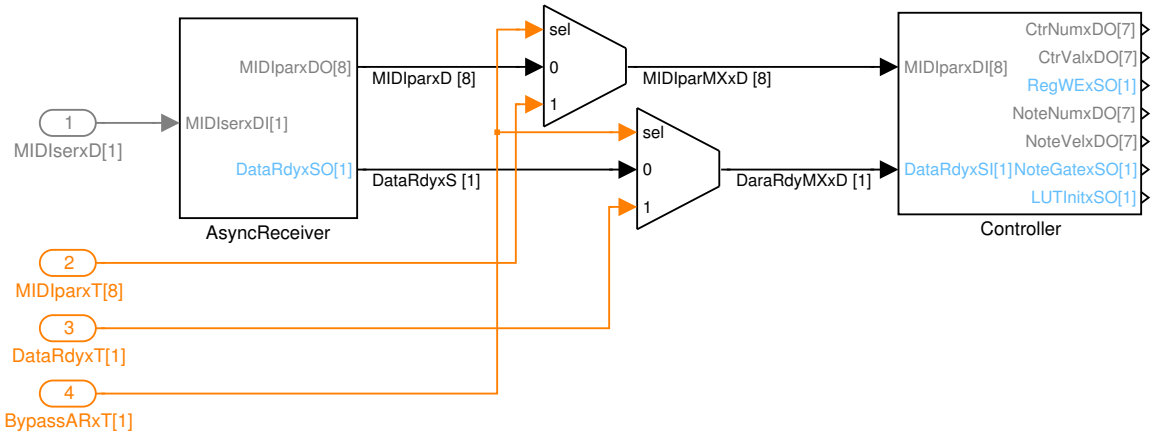

Figure 4.2: Bypassing the asynchronous receiver

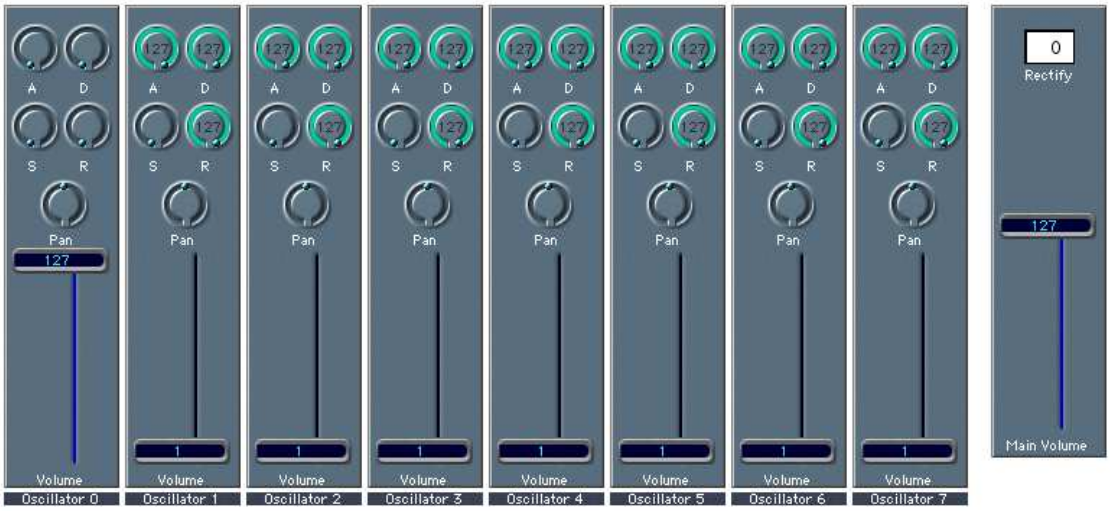

Figure 4.3: Emagic Logic frontend

For the implementation we used the file `top.vhd` on page 92 which has been extended to make use of the LEDs on the board:

- The yellow LED is lit as long as a key is pressed, i.e. when *NoteGatexS* is 1.
- The green LED is lit upon MIDI activity. This LED has no direct counterpart in the original set of signals in the design, but it is deviated from *DataRdyxS*. Because this signal is high for one clock cycle only when a MIDI byte is read and these 177 ns are too short a time for the LED to be illuminated[3], we extended this time period using a 19 bit counter to a minimum high time of around 0.1 s.

The pin assignments for the design are listed in table 4.1.

```
#PINLOCK_BEGIN

##########################################################################
#          Pinning for MIDI Synthesizer, top_fpga.vhd                    #
##########################################################################

NET "MIDIserxDI"     LOC="P202"; # the incoming midi stream
NET "MIDIChanxDI(3)" LOC="P42" ; # the channel we listen to
NET "MIDIChanxDI(2)" LOC="P46" ; # the channel we listen to
NET "MIDIChanxDI(1)" LOC="P41" ; # the channel we listen to
NET "MIDIChanxDI(0)" LOC="P34" ; # the channel we listen to
NET "AudioSerSxDO"   LOC="P224"; # serial sound output
NET "AudioSerLLExSO" LOC="P230"; # latch enable left side
NET "AudioSerRLExSO" LOC="P231"; # latch enable right side
NET "AudioSerLRxCO"  LOC="P209"; # left/right channel clock for D/A
NET "AudioSerBitxCO" LOC="P201"; # serial audio output

NET "CLKxCI"  LOC="P89";         # the clock
NET "RSTxRBI" LOC="P65";         # asynchronous reset, active low

NET "YellowLEDxSO"    LOC = "P78"; # note on status
NET "GreenLEDxSO"     LOC = "P79"; # midi activity

#PINLOCK_END
```

Table 4.1: Pin mapping for the FPGA, `top.ucf`

To connect the FPGA to other equipment[4] we built a small circuit board. The circuit was first built on an evaluation board where we could change cable routing with low effort. After all bugs had been fixed, it was soldered. See figure 4.4 for the circuitry surrounding the FPGA.

At the input of the FPGA, an optocoupler is needed to convert the current of about 5 mA delivered by the MIDI cable into a voltage of about 5 V. Also, it serves to galvanically isolate the MIDI cable from the FPGA to prevent possible electrical problems. At the output a digital-to-analog converter does its job to have an analog signal which can be listened to using headphones.

Because the optocoupler needed, Sharp's PC900V, was not in stock, we removed it from an old MIDI interface at hand and reused it on our board. Alternatively, the PC900V, the PC910, or the 6N137 could have been used, but these weren't available either. Some other types were evaluated by trial-and-error, but none of them worked.

The PCM1725 by Burr-Brown, the digital-to-analog converter we chose, was not in stock either and the delivery would have taken 6 weeks when ordered, so we decided to switch to a different DAC for the FPGA prototype and to adapt our design accordingly. As the only suitable DAC we found, the AD1856 by Analog Devices [4], is a mono device, we needed two of them to get stereo conversion. Unfortunately, the AD1856 does not implement the I^2S-protocol but uses a "latch enable" input to read in the 16 bit data, so we had to change our I^2S-controller slightly.

[3] Even if it were technically possible to light the LED that fast, it would be too short to perceive the effect by eye.

[4] We used Emagic Logic on an Apple PowerBook to send MIDI messages. Various clock generators, power supplies and analysers were needed too.

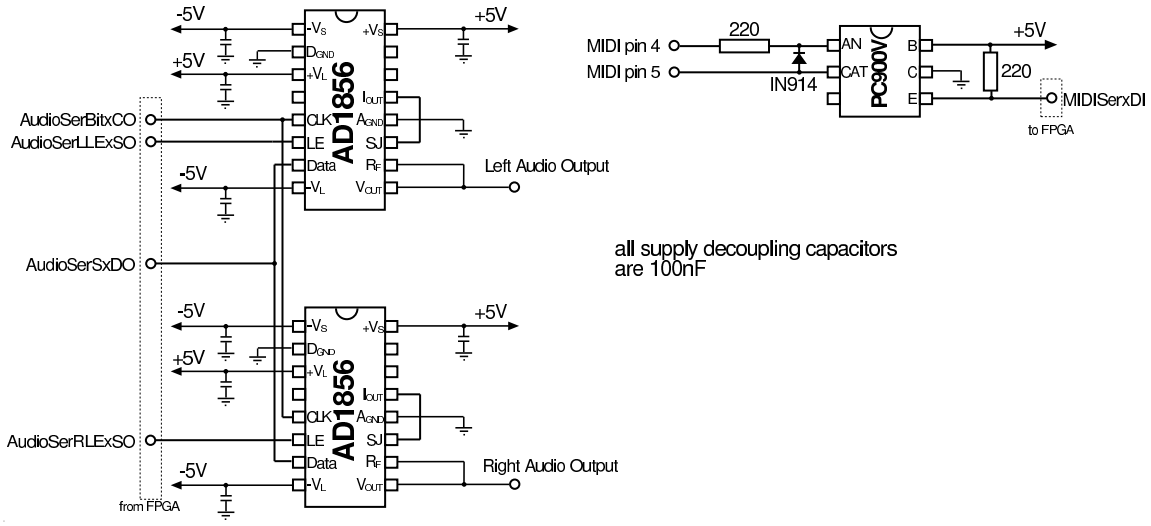

Figure 4.4: Schematic of the DAC and MIDI board

There were some problems first due to the fact that the AD1851 is a drop-in replacement for the AD1856 and the technical specifications of the AD1856 are not available anymore: the recommended circuit schematic did not work because pin 8 of the AD1851 is unconnected, but pin 8 of the AD1856 needs to be connected to -5 V as we read in the specs for the DAC56 by Burr-Brown [5] which is identical to the AD1856.

4.5.2 Tests performed with the Prototype

Possibly the best reason for having a prototype is its realtime behaviour. Until now, to obtain 1 second of audio data by means of post-layout, gate-level simulation, it took 2 days on the fastest machine we could get. The FPGA offered the advantage that we could change the VHDL code (i.e. alter the default settings for the sound), synthesize and download it to the FPGA, and see and hear the results immediately. A synthesize-download cycle takes about half an hour which is a huge improvement over software-based simulation.

The tests with the FPGA prototype showed that all controllers are recognized correctly and all unimplemented messages are ignored. Also, the behaviour and the sound output when note-on messages are sent are as expected. Stated simply: It works!

Experimental changes in the clock frequency yielded the following results:

- The MIDI interface works correctly between 5.36 MHz and 5.94 MHz. This is an error of $\pm 5\%$ as predicted in section 3.2.
- The sound generators did their job correctly for clock frequencies ranging from 0 to 15 MHz. This test was done by pressing a key and then changing the clock frequency. As soon as the clock frequency is outside the range allowed for the MIDI interface, the key can be released without the MIDI interface recognizing it, so the sound continues. Altering the clock frequency changes the pitch of the sound proportionally.

The second result is the definitive proof that we won't have any timing problems on the final chip, because the design runs on an FPGA even when threefold overclocked!

Appendix A

MIDI Synthesizer Data Sheet

Functional Description

This device is a monophonic MIDI synthesizer with 16 bit digital stereo sound output, sampled at 44.1 kHz. The waveform is generated using 8 sine/rectangle oscillators representing the fundamental frequency and the first seven harmonics.

Additional components such as an optocoupler at the input and a digital-to-analog converter (DAC) at the output are needed for operation.

To allow for various types of DACs, two formats of digital output are implemented: the widely used I^2S-format and a format for DACs requiring a latch enable signal. For details about the signals present at the output refer to figure A.2.

Device Operation and Configuration

To configure the device, the following parameters can be adjusted:

- Main volume of the sound output
- Volume of each individual oscillator
- Location in the stereo panorama of each oscillator
- Amplitude envelope of each oscillator defined by attack time, decay time, sustain level, and release time. Refer to figure A.3 for more information
- Rectangle or sine wave output of each oscillator

These parameters can all be adjusted by sending appropriate MIDI messages. Please refer to the MIDI Implementation Chart for more information. To set the MIDI channel the device listens to, adjust the four pins *MIDIChanxDI*[3...0] accordingly.

Sound output may be generated without prior configuration as reasonable default settings are active after reset. Any device that wants to communicate with this synthesizer via MIDI should do so according to the MIDI specification that can be found on the website of the MIDI Manufacturers Association, http://www.midi.org.

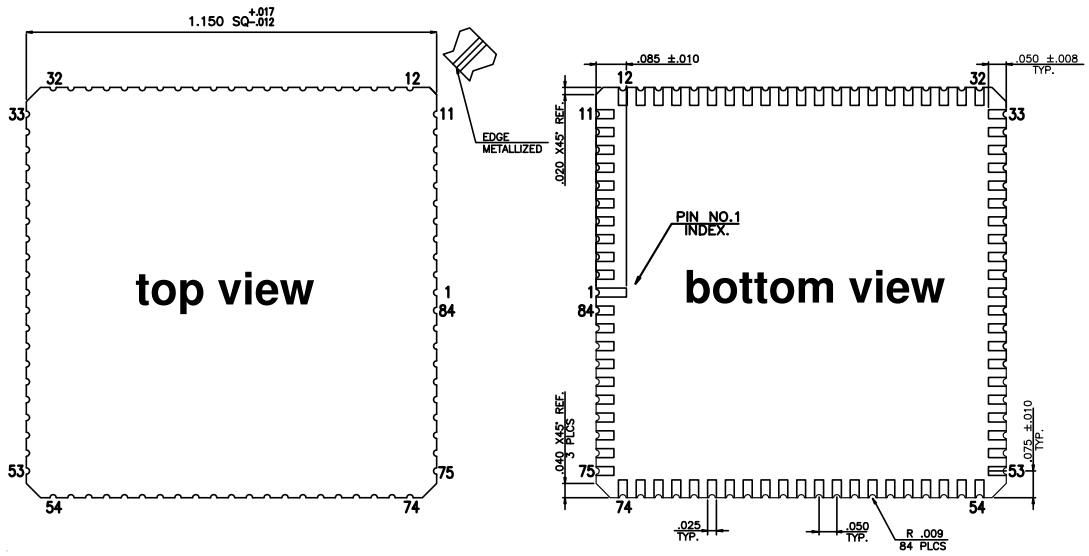

Figure A.1: CLCC84 Package and Pin Locations

Physical Characterization

Process	UMC Taiwan, 0.25 μm, five metal layers, one poly-silicon layer
Supply voltage	2.5 V
I/O voltage	max 3.3 V
Core size	2.435 mm x 2.435 mm (including sealring)
Package	CLCC84
Clock frequency	11.2896 MHz in normal operation 5.6448 MHz in test mode. See description of pin *CLKSelxSI* for more information.
Power Consumption	$\approx$ 390 mW (based on simulation node activities)

2, 23, 44, 65	V_{DD} core
21, 42, 63	V_{SS} core
12, 33, 54, 75	V_{DD} pads
11, 32, 53, 74	V_{SS} pads
13	*AudioSerSxDO*
14	*AudioSerLLExSO*
15	*AudioSerRLExSO*
16	*AudioSerBitxCO*
17	*AudioSerLRxCO*
29	*RSTxRBI*
30	*CLKxCI*
31	*CLKSelxSI*
34	*DataRdyxTI*
35	*BypassARxTI*
45...52	*MIDIparxTI*[7...0]
55...58	*MIDIChanxDI*[3...0]
70	*ScanInxTI*
71	*ScanEnxTI*
72	*ScanOutxTO*
73	*MIDIserxDI*

Table A.1: Pin Assignments for the CLCC84 Package

Pin Description

Pin	Purpose
	MIDI
MIDIserxDI	MIDI serial data input. The MIDI bit stream is read at 31.25 kHz by the Asynchronous Receiver.
MIDIChanxDI[3:0]	MIDI Channel select input. Select the MIDI channel the MIDI controller listens to. Bit 3 is the MSB.
	Audio
AudioSerSxDO	Serial audio output. The 16 bit digital stereo audio data is serially output, MSB first. Each bit is valid on the rising edge of *AudioSerBitxCO*.
AudioSerLLExSO	Serial audio left channel latch enable output. Used in conjunction with some types of DAC to tell them when to read the data for the left channel. Refer to figure A.2 for timing information.
AudioSerRLExSO	Serial audio right channel latch enable output. Used in conjunction with some types of DAC to tell them when to read the data for the right channel. Refer to figure A.2 for timing information.
AudioSerLRxCO	I^2S-protocol left/right channel clock output. When set to high/low, the data on *AudioSerSxDO* corresponds to the right/left channel. Refer to figure A.2 for timing information.
AudioSerBitxCO	*AudioSerSxDO* is valid on the rising edge of the *AudioSerBitxCO* bit clock output.

Control	
CLKxCI	Clock input. The frequency is either 128 or 256 times the sampling frequency of 44.1 kHz, i.e. 5.6448 MHz or 11.2896 MHz, respectively. See the description of pin *CLKSelxS* on how to set the desired value.
CLKSelxSI	Clock divider select input. Setting it low activates the clock divider, the externally applied clock at *CLKxCI* should then run at 11.2896 MHz. Setting *CLKSelxSI* high bypasses the clock divider, *CLKxCI* is then expected to be at 5.6448 MHz. *CLKSelxSI* is overridden when *ScanEnxTI* is set.
RSTxRBI	Global asynchronous reset input. Setting it low yields a complete reset of the chip. Keep *RSTxRBI* high during normal operation.
Test	
ScanEnxTI	Scan enable input. When set to high, the Scan chain is activated and the clock divider disabled.
ScanInxTI	Scan chain input. When *ScanEnxT* is set, at each clock cycle the value of *ScanInxTI* is fed into the Scan chain.
ScanOutxTO	Scan chain output. When *ScanEnxT* is set, *ScanOutxTO* is the output of the Scan chain which gets shifted by one bit at each clock cycle. As the last register of the scan chain is the same as the output register of the digital audio output, *ScanOutxTO* is galvanically connected with *AudioSerxDO*.
MIDIparxTI[7:0]	MIDI data test input. When *BypassARxTI* is set, *MIDIparxTI* is used to feed MIDI data directly to the MIDI controller, bypassing the Asynchronous Receiver and thus resulting in a huge speedup which is necessary during ATE-operation. These signals are not used during normal operation. For more information, please refer to the documentation.
DataRdyxTI	MIDI data ready input. When *BypassARxTI* is active, *DataRdyxTI* tells the MIDI controller to read the value at *MIDIparxTI*.
BypassARxTI	Asynchronous Receiver bypass enable input.
Power	
VDD_c	2.5 V power supply (core)
VSS_c	power ground (core)
VDD_r	2.5 V power supply (ring)
VSS_r	power ground (ring)

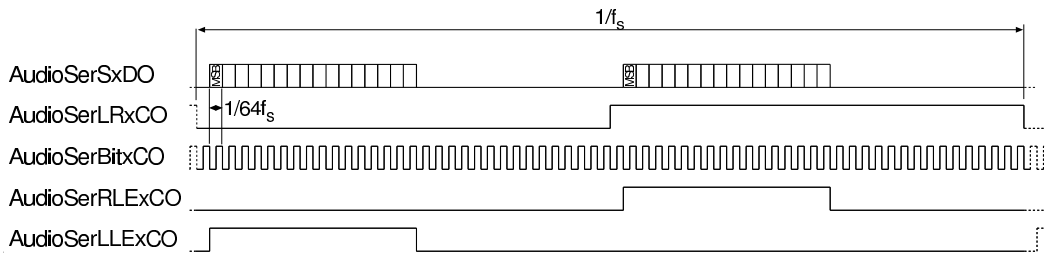

Figure A.2: Digital Audio Output

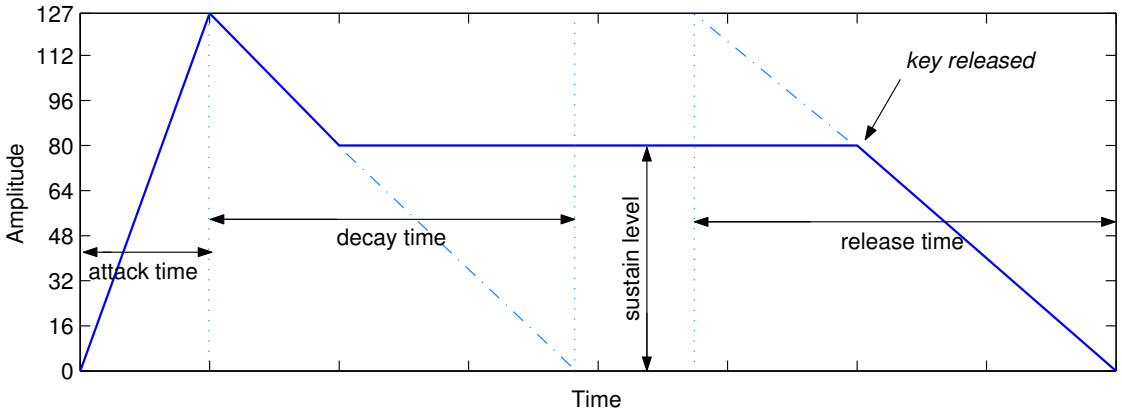

Figure A.3: Parameters of the ADSR envelope generator

Appendix B

MIDI Implementation Chart

Function		Tx	Rx	Remarks
Basic Channel	default	x	1-16	set by Channel Pins
	changed	x	x	
Mode	default	x	x	
	Messages	x	x	
	changed	x	x	
Note Number		x	36-107	C2-B7 (65.405-3951 Hz)
	True Voice	*******	x	
Velocity	Note On	x	o	
	Note Off	x	x	
After Touch	Keys	x	x	
	Channels	x	x	
Pitch Bender		x	x	
Control Change	7	x	o	Main Volume
	9	x	o	Sine / Rectangle Select
	64-67	x	o	Osc0: A, D, S, R
	68-71	x	o	Osc1: A, D, S, R
	72-75	x	o	Osc2: A, D, S, R
	76-79	x	o	Osc3: A, D, S, R
	80-83	x	o	Osc4: A, D, S, R
	84-87	x	o	Osc5: A, D, S, R
	88-91	x	o	Osc6: A, D, S, R
	92-95	x	o	Osc7: A, D, S, R
	102-109	x	o	individual Osc Volumes
	110-117	x	o	individual Osc Pans
Program Change		x	x	
	True #	x	x	
System Exclusive		x	x	
System Common	Song Pos	x	x	
	Song Sel	x	x	
	Tune	x	x	
System Realtime	Clock	x	x	
	Commands	x	x	
Aux Messages	Reset	x	o	

o : implemented
x : not implemented

Appendix C

Note and Frequency Listing

The following table lists all notes which the synthesizer can play in the following format:

MIDI note number | note name | base frequency (Hz)

2nd octave		
36	C	65.4063
37	C#	69.2956
38	D	73.4161
39	D#	77.7817
40	E	82.4068
41	F	87.3070
42	F#	92.4986
43	G	97.9988
44	G#	103.8261
45	A	110.0000
46	A#	116.5409
47	B	123.4708

3rd octave		
48	C	130.8127
49	C#	138.5913
50	D	146.8323
51	D#	155.5634
52	E	164.8137
53	F	174.6141
54	F#	184.9972
55	G	195.9977
56	G#	207.6523
57	A	220.0000
58	A#	233.0818
59	B	246.9416

4th octave		
60	C	261.6255
61	C#	277.1826
62	D	293.6647
63	D#	311.1269
64	E	329.6275
65	F	349.2282
66	F#	369.9944
67	G	391.9954
68	G#	415.3046
69	A	440.0000
70	A#	466.1637
71	B	493.8833

5th octave		
72	C	523.2511
73	C#	554.3652
74	D	587.3295
75	D#	622.2539
76	E	659.2551
77	F	698.4564
78	F#	739.9888
79	G	783.9908
80	G#	830.6093
81	A	880.0000
82	A#	932.3275
83	B	987.7666

6th octave		
84	C	1046.5022
85	C#	1108.7305
86	D	1174.6590
87	D#	1244.5079
88	E	1318.5102
89	F	1396.9129
90	F#	1479.9776
91	G	1567.9817
92	G#	1661.2187
93	A	1760.0000
94	A#	1864.6550
95	B	1975.5332

7th octave		
96	C	2093.0045
97	C#	2217.4610
98	D	2349.3181
99	D#	2489.0158
100	E	2637.0204
101	F	2793.8258
102	F#	2959.9553
103	G	3135.9634
104	G#	3322.4375
105	A	3520.0000
106	A#	3729.3100
107	B	3951.0664

Appendix D

Lessons Learned

From an educational point of view, stumbling blocks are most important when learning. These are some of the major problems we came across:

- Don't forget to model inaccuracy

  One problem that bothered us from the beginning was the clicking noise that's audible when the oscillators change frequency. The problem apparently consisted in the oscillators switching at different times which caused a burst of eight discontinuities in the output signal. One of the different approaches to reduce the noise was thus to synchronize all oscillators before they start playing the new tone by letting them wait silently for eachother. A Matlab model looked promising, the change was quickly implemented. Unfortunately, it made the noise only worse.

  The model let eight imaginary oscillators play at precise multiples of the base frequency. As the zeros of the lowest oscillator always aligned with zeros of the upper ones, the synchronization didn't change anything. In the RTL model of the synthesizer however, there exists a frequency mismatch of about 0.5while waiting for eachother, the sound gets horribly distorted which can be seen in figure D.1. Introducing a random frequency mismatch in the Matlab model of course revealed the error.

- Gate-level simulation

  After the first synthesis runs, we wanted to run a gate level simulation of the whole chip. The chip pads, reset synchronizer and clock divider were already included in the netlist. The first simulation runs showed that the internal (divided) clock was always zero. The weird thing was that the inputs of the multiplexer of which the clock emerged were correct.

  It turned out that as the clock tree was not inserted yet, the multiplexer had to carry the full load of more than 3,000 clock inputs. It took it 0.5 ms to charge this huge capacitance which is of course too much. We then removed the pads and the clock divider from the simulation.

- Don't trust the Testbench

  The first RTL-simulations didn't work very well, the audio output was distorted and unusable. Even after lengthy code examinations no error could be found.

  The "bug" was in the testbench itself. The clock generating process was counting nanoseconds, namely $1\ \mathrm{s}/(44100 \cdot 128) = 177.1542\ \mathrm{ns}$. The audio output logging process itself was waiting for $1\ \mathrm{s}/44100 = 22.6757\ \mu\mathrm{s}$. This should work well, but as unfortunately the simulator, ModelSim, based its time calculations on nanoseconds, truncation errors caused the testbench to drop every 150th value. We then changed the audio logger to count clock cycles which made it independent from the actual clock period. Finally the usage of the $\mathrm{I}^2\mathrm{S}$-protocol made it even simpler as a dedicated bit clock is used for synchronisation purposes.

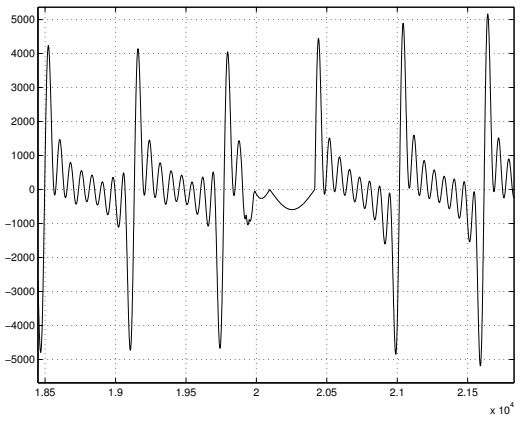

Figure D.1: Waveform of a “synchronized” change in frequency

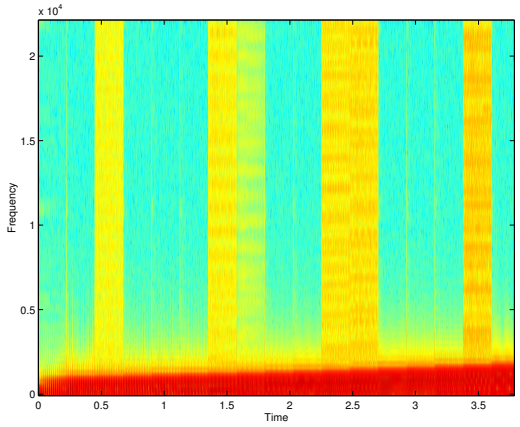

Figure D.2: Spectrogram of a distorted scale

- I/O Buffering

  At some frequencies, the first audio sample after a zero point had its sign bit wrong. The problem was that the complex multiplier assumed its input to be constant during the whole sample time. To simplify multiplication, the sign bit is extracted at the beginning of the cycle and restored at the end. As the oscillator and the multiplier are not synchronized, the sign bit sometimes changed in between a calculation. Figure D.2 shows the spectogram of the resulting sound.

  From then on, we made sure that all functional blocks buffer either their input or output. Any block that needs input data to be valid during more than one clock cycle must store this value in a buffer. Don’t trust the preceding block, it might get unsynchronized in some way. The extra effort and an additional input register may save many hours of looking for bugs.

Appendix E

Acknowledgements

Many thanks to

- Stephan Oetiker and Simon Haene for their ever ready and generous support,
- the whole staff of the “IC and System Design and Test” research group at IIS who spent many hours in our lab,
- Hubert Kaeslin and Norbert Felber for their famous lecture,
- Hans-Peter Mathys and Hans-Jörg Gisler for helping us to build the FPGA board,
- the Integrated Systems Laboratory for all the money they spent on our education.

Appendix F

VHDL Source

ADSR.vhd

```
-------------------------------------------------------------------------
-- Title : ADSR Envelope Generator
-- Project :
-------------------------------------------------------------------------
-- File : ADSR.vhd
-- Author : Samuel Nobs <nobssa@ee.ethz.ch>
-- Company : Integrated Systems Laboratory, ETH Zurich
-- Created : 2002/11/08
-- Last update: 2002/12/02
-- Platform : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------
-- Description: generates the signal envelope
-- with attack, decay, sustain and release phases
-------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------
-- Revisions :
-- Date Version Author Description
-- 2002/11/08 1.0 Samuel Nobs Created
-------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity ADSR is

  port (
    NoteGatexSI : in  std_logic;                -- high when key is pressed
    OscRSTxSI   : in  std_logic;                -- high for 1 period
                                                -- as soon as the oscillator
                                                -- starts a new frequency
    ADSRxDI     : in  unsigned(27 downto 0);    -- values for A, D, S, R
    AmpIUxDO    : out unsigned(8 downto 0);     -- the envelope
    CLKxCI      : in  std_logic;                -- clock
    RSTxRBI     : in  std_logic);               -- async reset, active low

end ADSR;

architecture rtl of ADSR is

  -- adsr signals
  signal AxD : unsigned(6 downto 0);
  signal DxD : unsigned(6 downto 0);
  signal SxD : unsigned(6 downto 0);
  signal RxD : unsigned(6 downto 0);

  -- adsr states
  type AdsrState is (stIdle, stAttack, stDecay, stSustain, stRelease);
  signal ADSRxDN : AdsrState;               -- next state
  signal ADSRxDP : AdsrState;               -- previous state

  -- output register
  signal AmpxDN : unsigned(8 downto 0);  -- next value for the amplitude
  signal AmpxDP : unsigned(8 downto 0);  -- previous value for the amplitude

  -- counter signals
  constant COUNTLEN : natural := 16;    -- number of bits for counter
                                        -- use this number to scale
                                        -- the time constants (>7)
                                        -- 8->23 ms, 9->46 ms, 10-> 92 ms and
                                        -- so forth
  signal    CountxDN  : unsigned(COUNTLEN-1 downto 0);  -- next value
  signal    CountxDP  : unsigned(COUNTLEN-1 downto 0);  -- previous value
  signal    CtStartxS : std_logic;        -- resets counter to zero

  -- shift helper signal
  signal ShiftxD : unsigned(COUNTLEN-8 downto 0);

begin  -- rtl

  -- split ADSRxDI
  AxD <= ADSRxDI(6 downto 0);
  DxD <= ADSRxDI(13 downto 7);
  SxD <= ADSRxDI(20 downto 14);
  RxD <= ADSRxDI(27 downto 21);

  -- assign vector of zeros for shifting
  ShiftxD <= (others => '0');

  -- purpose: calculates the state transitions and outputs of
  --          the ADSR envelope generator
  -- type   : combinational
  -- inputs : ADSRxDP, AmpxDP
  -- outputs: ADSRxDN, AmpxDN
  adsrcalculate : process (ADSRxDN, ADSRxDP, AmpxDP, AxD, CountxDP, DxD,
                           NoteGatexSI, OscRSTxSI, RxD, ShiftxD, SxD)
  begin  -- process adsrcalculate
    -- defaults
    ADSRxDN               <= ADSRxDP;
    AmpxDN                <= AmpxDP;
    CtStartxS             <= '0';
    -- nondefaults
    case ADSRxDP is
      -- ************************************* IDLE
      when stIdle     =>
        ADSRxDN           <= stIdle;
        if NoteGatexSI = '1' and OscRSTxSI = '1' then  -- key is hit
          ADSRxDN         <= stAttack;
          CtStartxS       <= '1';
        end if;
        -- ************************************* ATTACK
      when stAttack   =>
        if NoteGatexSI = '1' then
          -- key is still pressed
          if CountxDP >= (AxD&ShiftxD) and AmpxDP < 128*4-1 then
            -- increase amplitude, if maximum is not reached yet
            AmpxDN        <= AmpxDP+1;
            CtStartxS     <= '1';
          elsif AmpxDP >= 128*4-1 then
            -- if maximum is reached, jump to decay state
            ADSRxDN       <= stDecay;
            CtStartxS     <= '1';
          end if;
        else
          -- key is released, change to release state
          ADSRxDN         <= stRelease;
          CtStartxS       <= '1';
        end if;
        -- ************************************* DECAY
      when stDecay    =>
        if NoteGatexSI = '1' then
          -- key is still pressed
          if CountxDP >= (DxD&ShiftxD) and AmpxDP > SxD&"00" then
            -- decrease amplitude, if sustain level is not reached yet
            AmpxDN        <= AmpxDP-1;
            CtStartxS     <= '1';
          elsif AmpxDP <= SxD&"00" then
            -- if sustain level is reached, change to sustain state
            ADSRxDN       <= stSustain;
          end if;
        else
          -- key is released, change to release state
          ADSRxDN         <= stRelease;
          CtStartxS       <= '1';
        end if;
        -- ************************************* SUSTAIN
      when stSustain =>
```

```
      -- stay here as long as key remains pressed
      if NoteGatexSI = '0' then
        -- change to release state as soon as key is released
        ADSRxDN       <= stRelease;
        CtStartxS     <='1';
      end if;
      -- ************************************* RELEASE
    when stRelease =>
      if not (NoteGatexSI = '1' and OscRSTxSI = '1') then
        -- if key is not hit
        if COuntxDP >= (RxD&ShiftxD) and AmpxDP > 0 then
          -- decrement amplitude until sound completely died out
          AmpxDN      <= AmpxDP-1;
          CtStartxS   <='1';
        elsif AmpxDP = 0 then
          -- if amplitude is zero, go to idle state
          ADSRxDN     <= stIdle;
        end if;
      else
        -- if key is hit again, restart from attack state
        ADSRxDN       <= stAttack;
        CtStartxS     <='1';
      end if;
    when others     => null;
  end case;
end process adsrcalculate;

-- purpose: update state of adsr envelope generator
-- type   : sequential
-- inputs : CLKxCI, RSTxRBI, ADSRxDN
-- outputs: ADSRxDP
adsrassign : process (CLKxCI, RSTxRBI)
begin  -- process adsrassign
  if RSTxRBI = '0' then                 -- asynchronous reset (active low)
    ADSRxDP <= stIdle;
  elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
    ADSRxDP <= ADSRxDN;
  end if;
end process adsrassign;

-- purpose: update output register
-- type   : sequential
-- inputs : CLKxCI, RSTxRBI, AmpxDN
-- outputs: AmpxDP
outputregassign : process (CLKxCI, RSTxRBI)
begin  -- process outputregassign
  if RSTxRBI = '0' then                 -- asynchronous reset (active low)
    AmpxDP <= to_unsigned(0, 9);
  elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
    AmpxDP <= AmpxDN;
  end if;
end process outputregassign;

-- connect to output pin
AmplUxDO <= AmpxDP;

-- purpose: increment counter value
-- type   : combinational
-- inputs : CountxDP, CtStartxS
-- outputs: CountxDN
countercalculate : process (CountxDP, CtStartxS)
begin  -- process countercalculate
  if CtStartxS = '1' then               -- reset to zero
    CountxDN <= to_unsigned(0, COUNTLEN);
  else
    CountxDN <= CountxDP+1;             -- increment
  end if;
end process countercalculate;

-- purpose: assigne incremented value
-- type   : sequential
-- inputs : CLKxCI, RSTxRBI, CountxDN
-- outputs: CountxDP
counterassign : process (CLKxCI, RSTxRBI)
begin  -- process counterassign
  if RSTxRBI = '0' then                 -- asynchronous reset (active low)
    CountxDP <= to_unsigned(0, COUNTLEN);
  elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
    CountxDP <= CountxDN;
  end if;
end process counterassign;


end rtl;
```

AsyncRecv.vhd

```
-------------------------------------------------------------------------------
-- Title      : asynchronous receiver
-- Project    : MIDI Synthesizer
-------------------------------------------------------------------------------
-- File       : AsyncRecv.vhd
-- Author     : Samuel Nobs <nobssa@ee.ethz.ch>
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/10/31
-- Last update: 2003/01/30
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: reads bit-serial midi data and output them as parallel bytes,
--              notifying the midi interface that the data is ready to be read
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author         Description
-- 2002/10/31  1.0      Samuel Nobs    Created
-- 2002/11/04           Samuel Nobs    Verified using selected vectors
-- 2002/11/04           Samuel Nobs    Verified using a large amount of
--                                     random vectors
-- 2002/11/04           Samuel Nobs    GateLevel Netlist (AsyncRecvNL.vhd) verified
--                                     using selected vectors and random vectors
-- 2002/11/27           Samuel Nobs    Added synchronizing registers to input
--                                     avoiding metastability
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity AsyncRecv is
  port (
    MIDIserxDI : in  std_logic;         -- serial midi data at input
    MIDIparxDO : out std_logic_vector(7 downto 0);
                                        -- parallel midi data at output
    DataRdyxSO : out std_logic;         -- tell midi interface that MIDI byte
                                        -- is available
    CLKxCI     : in  std_logic;         -- hmm...what could this be?
    RSTxRBI    : in  std_logic);        -- async reset, active low
end AsyncRecv;


architecture rtl of AsyncRecv is

  -- states of the main fsm
  subtype asyncFsmSt is std_logic_vector(3 downto 0);  --state type
  signal FSMxDN : asyncFsmSt;                          -- next state
  signal FSMxDP : asyncFsmSt;                          -- previous state
```

```
  constant stWait   : asyncFsmSt := "0000";  -- waiting for startbit
  constant stSync   : asyncFsmSt := "0001";  -- synchronize to center of pulse
  constant stStart  : asyncFsmSt := "0010";  -- start bit
  constant stBit0   : asyncFsmSt := "0011";  -- first data bit
  constant stBit1   : asyncFsmSt := "0100";
  constant stBit2   : asyncFsmSt := "0101";
  constant stBit3   : asyncFsmSt := "0110";
  constant stBit4   : asyncFsmSt := "0111";
  constant stBit5   : asyncFsmSt := "1000";
  constant stBit6   : asyncFsmSt := "1001";
  constant stBit7   : asyncFsmSt := "1010";  -- last last bit


  signal EnaSregxS     : std_logic;      -- enable shift register
  signal EnaSregTmpxS  : std_logic;      -- unregistered enable for the shift register

  -- output register
  signal OutRegxDN                      : std_logic_vector(7 downto 0);
  signal OutRegxDP                      : std_logic_vector(7 downto 0);
  -- enable output register
  signal EnaOutxS                       : std_logic;
  -- buffers signal coming from the SIPO register
  signal MIDIbufparxD                   : std_logic_vector(7 downto 0);
  -- sync register at input
  signal In1xDN, In2xDN, In1xDP, In2xDP : std_logic;
  -- serial midi data after input sync registers
  signal MIDIserxD                      : std_logic;



  -- counter signals
  signal COUNTxDN     : unsigned(7 downto 0);  -- next state for counter
  signal COUNTxDP     : unsigned(7 downto 0);
                                               -- previous state for counter
  signal StartCountxS : std_logic;             -- used to start the counter
  signal EnaFSMxS     : std_logic;             -- allow the FSM to continue
  -- mode 0: count to 70, mode 1: count to 140
  signal CountModexS  : std_logic;


  -- the shift register we need
  component SIPOreg
    port (
      SERxDI  : in  std_logic;
      PARxDO  : out std_logic_vector(7 downto 0);
      ENAxSI  : in  std_logic;
      CLKxCI  : in  std_logic;
      RSTxRBI : in  std_logic);
  end component;

begin  -- rtl

-- synchronize asynchronous input data using two registers
-- to reduce the danger of metastability
  In1xDN <= MIDIserxDI;
  In2xDN <= In1xDP;

  sync : process (CLKxCI, RSTxRBI)
  begin  -- process sync
    if RSTxRBI = '0' then               -- asynchronous reset (active low)
      In1xDP <= '1';                    -- idle MIDI line is 1
      In2xDP <= '1';
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      In1xDP <= In1xDN;
      In2xDP <= In2xDN;
    end if;
  end process sync;

  MIDIserxD <= In2xDP;


-- purpose: calculate the next state for the fsm
-- type : combinational
-- inputs :
-- outputs:
  fsmcalculate : process (EnaFSMxS, FSMxDP, MIDIserxD)
  begin  -- process fsmcalculate
    FSMxDN                  <= FSMxDP;
    StartCountxS            <= '0';
    EnaSregTmpxS            <= '0';
    EnaOutxS                <= '0';
    if EnaFSMxS = '1' or FSMxDP = stWait then  -- counter is done
      case FSMxDP is
        when stWait  =>
          FSMxDN            <= stWait;
          if MIDIserxD = '0' then           -- yeehaw, the start bit
            FSMxDN          <= stSync;
            StartCountxS    <= '1';         -- start the counter
          end if;
        when stSync  =>
          FSMxDN            <= stStart;
          StartCountxS      <= '1';
        when stStart =>
          FSMxDN            <= stBit0;
          StartCountxS      <= '1';
          EnaSregTmpxS      <= '1';
        when stBit0  =>
          FSMxDN            <= stBit1;
          StartCountxS      <= '1';
          EnaSregTmpxS      <= '1';
        when stBit1  =>
          FSMxDN            <= stBit2;
          StartCountxS      <= '1';
          EnaSregTmpxS      <= '1';
        when stBit2  =>
          FSMxDN            <= stBit3;
          StartCountxS      <= '1';
          EnaSregTmpxS      <= '1';
        when stBit3  =>
          FSMxDN            <= stBit4;
          StartCountxS      <= '1';
          EnaSregTmpxS      <= '1';
        when stBit4  =>
          FSMxDN            <= stBit5;
          StartCountxS      <= '1';
          EnaSregTmpxS      <= '1';
        when stBit5  =>
          FSMxDN            <= stBit6;
          StartCountxS      <= '1';
          EnaSregTmpxS      <= '1';
        when stBit6  =>
          FSMxDN            <= stBit7;
          StartCountxS      <= '1';
          EnaSregTmpxS      <= '1';
        when stBit7  =>
          FSMxDN            <= stWait;
          StartCountxS      <= '1';
          EnaOutxS          <= '1';
        when others  => FSMxDN <= stWait;   -- SynopsysDC insists on this
      end case;
    end if;
    if FSMxDP = stSync then
      CountModexS           <= '0';         -- count to 90
    else
      CountModexS           <= '1';         -- count to 180
    end if;
  end process fsmcalculate;

  -- purpose: update the next state for the fsm
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, FSMxDN
  -- outputs: FSMxDP
  fsmassign : process (CLKxCI, RSTxRBI)
```

```
  begin  -- process fsmassign
    if RSTxRBI = '0' then                -- asynchronous reset (active low)
      FSMxDP     <= stWait;
      EnaSregxS <= '0';
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      FSMxDP     <= FSMxDN;
      EnaSregxS <= EnasRegTmpxS;
    end if;
  end process fsmassign;

  -- purpose: increment the counter
  -- type : combinational
  -- inputs : StartCountxS, CountModexS
  -- outputs: COUNTxDN, EnaFSMxS
  countcalculate : process (COUNTxDP, CountModexS, StartCountxS)
  begin  -- process countcalculate
    COUNTxDN    <= COUNTxDP+1;
    if COUNTxDP = 90 and CountModexS = '0' then       -- allow FSM to continue
      EnaFSMxS <= '1';
    elsif COUNTxDP = 180 and CountModexS = '1' then  -- allow FSM to continue
      EnaFSMxS <= '1';
    else
      -- prevent FSM from continuing
      EnaFSMxS <= '0';
    end if;
    if StartCountxS = '1' then                       -- restart counter
      COUNTxDN <= to_unsigned(0, 8);
    end if;
  end process countcalculate;

  -- purpose: update the counter
  -- type : sequential
  -- inputs : CLKxCI, RSTxRBI, COUNTxDN
  -- outputs: COUNTxDP
  countupdate : process (CLKxCI, RSTxRBI)
  begin  -- process countupdate
    if RSTxRBI = '0' then                -- asynchronous reset (active low)
      COUNTxDP <= to_unsigned(0, 8);
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      COUNTxDP <= COUNTxDN;
    end if;
  end process countupdate;

  -- now lets connect the shift register
  u_ShiftRegister : SIPOreg port map (
    SERxDI  => MIDIserxD,
    PARxDO  => MIDIbufparxD,
    ENAxSI  => EnaSregxS,
    CLKxCI  => CLKxCI,
    RSTxRBI => RSTxRBI);

  -- purpose: calculate output register
  -- type : combinational
  -- inputs : EnaOutxS, OutRegxDN
  -- outputs: OutRegxDP
  outputregcalculate : process (EnaOutxS, MIDIbufparxD, OutRegxDP)
  begin  -- process outputregcalculate
    OutRegxDN    <= OutRegxDP;
    if EnaOutxS = '1' then
      OutRegxDN <= MIDIbufparxD;
    end if;
  end process outputregcalculate;


  -- purpose: assign output register, keeps output constant and
  -- signals the interface to read in the MIDI byte
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, EnaOutxS, MIDIbufparxD
  -- outputs: MIDIparxDO
  outputregassign : process (CLKxCI, RSTxRBI)
  begin  -- process outputreg
    if RSTxRBI = '0' then                -- asynchronous reset (active low)
      OutRegxDP    <= "00000000";
      DataRdyxSO    <= '0';
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      OutRegxDP    <= OutRegxDN;
      DataRdyxSO    <= '0';
      if EnaOutxS = '1' then
        DataRdyxSO <= '1';
      end if;
    end if;
  end process outputregassign;

  MIDIparxDO <= OutRegxDP;               -- assign output;
end rtl;
```

bigadder.vhd

```
-------------------------------------------------------------------------------
-- Title : Big Adder
-- Project : MIDI Synthesizer
-------------------------------------------------------------------------------
-- File : bigadder.vhd
-- Author : sem02w5 stud account <sem02w5@badile3.ee.ethz.ch>
-- Company : Integrated Systems Laboratory, ETH Zurich
-- Created : 2002/11/14
-- Last update: 2002/12/09
-- Platform : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: Big adder which adds 8 16 bit values from the complex
-- multipliers
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions :
-- Date Version Author Description
-- 2002/11/14 1.0 danengel Created
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-------------------------------------------------------------------------------
-- Description of I/O signals
-------------------------------------------------------------------------------

-- ClkxCI: Clock

-- ResetxRBI: Asynchronous reset, active low

-- In0xDI .. In7xDI:  Data inputs, each 16 bits wide, 2's complement

-- OutxDO:  The result of the addition of all eight inputs, updated every eight
-- cycles.

entity bigadder is
  port (
    ClkxCI     : in  std_logic;
    ResetxRBI  : in  std_logic;
    In0xDI     : in  unsigned(15 downto 0);
    In1xDI     : in  unsigned(15 downto 0);
    In2xDI     : in  unsigned(15 downto 0);
    In3xDI     : in  unsigned(15 downto 0);
    In4xDI     : in  unsigned(15 downto 0);
    In5xDI     : in  unsigned(15 downto 0);
    In6xDI     : in  unsigned(15 downto 0);
    In7xDI     : in  unsigned(15 downto 0);
    OutxDO     : out unsigned(15 downto 0));
```

```
end bigadder;

architecture RTL of bigadder is
  -- registers
  signal AccuxDN, AccuxDP : unsigned(18 downto 0);
  signal OutxDN, OutxDP   : unsigned(15 downto 0);
  signal CntxDN, CntxDP   : unsigned(6 downto 0);
  -- helpers
  signal MuxSelxS         : unsigned(2 downto 0);
  signal AccuInitxS       : std_logic;
  signal OutInitxS        : std_logic;
  signal TIxD             : unsigned(15 downto 0);

begin
  -- counter
  process(CntxDP)
  begin
    CntxDN <= CntxDP + 1;
  end process;

  -- control signals
  process(CntxDP)
  begin
    -- MuxSelxS
    MuxSelxS <= CntxDP(2 downto 0);

    -- AccuInitxS
    if CntxDP < 8 then
      AccuInitxS <= '0';
    else
      AccuInitxS <= '1';
    end if;

    -- OutInitxS
    if CntxDP = 8 then
      OutInitxS <= '1';
    else
      OutInitxS <= '0';
    end if;
  end process;

  -- accumulate
  process(AccuInitxS, AccuxDP, TIxD)
  begin
    if AccuInitxS = '1' then
      AccuxDN <= (others => '0');
    else
      -- dirty sign extension from 16 to 19 bit
      AccuxDN <= (TIxD(15) & TIxD(15) & TIxD(15) & TIxD) + AccuxDP;
    end if;
  end process;

  -- select input
  process(In0xDI, In1xDI, In2xDI, In3xDI, In4xDI, In5xDI, In6xDI, In7xDI,
          MuxSelxS)
  begin
    case to_integer(MuxSelxS) is
      when 0      => TIxD <= In0xDI;
      when 1      => TIxD <= In1xDI;
      when 2      => TIxD <= In2xDI;
      when 3      => TIxD <= In3xDI;
      when 4      => TIxD <= In4xDI;
      when 5      => TIxD <= In5xDI;
      when 6      => TIxD <= In6xDI;
      when 7      => TIxD <= In7xDI;
      when others => TIxD <= (others => '0');
    end case;
  end process;

  -- buffer output
  process(AccuxDP, OutInitxS, OutxDP)
  begin
    if OutInitxS = '1' then
      OutxDN <= AccuxDP(18 downto 3);
    else
      OutxDN <= OutxDP;
    end if;
  end process;

  -- output
  OutxDO <= OutxDP;

  -- update registers
  process(ClkxCI, ResetxRBI)
  begin
    if ResetxRBI = '0' then
      AccuxDP <= (others => '0');
      OutxDP  <= (others => '0');
      CntxDP  <= (others => '0');
    elsif ClkxCI'event and ClkxCI = '1' then
      AccuxDP <= AccuxDN;
      OutxDP  <= OutxDN;
      CntxDP  <= CntxDN;
    end if;
  end process;

end RTL;
```

Chip.vhd

```
-------------------------------------------------------------------------------
-- Title      : Chip
-- Project    : MIDI Synthesizer
-------------------------------------------------------------------------------
-- File       : Chip.vhd
-- Author     : Daniel Engeler <danengel@ee.ethz.ch>,
--              Samuel Nobs <nobssa@ee.ethz.ch>
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/11/28
-- Last update: 2003/01/13
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: Chip level description with reset synchronisation and
--              i/o pads
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author      Description
-- 2002/11/28  1.0      sem02w5     Created
-- 2003/01/09           Samuel Nobs updated for AsyncRecv-bypassing
-------------------------------------------------------------------------------

library ieee, synopsys, umc_vhdl_pads, umc_vhdl_core;
use ieee.std_logic_1164.all;              -- IEEE std_logic
use ieee.std_logic_arith.all;             -- std_logic_arith (synopsys)
use synopsys.attributes.all;              -- synthesis attributes (synopsys)
use umc_vhdl_pads.PAD_COMPONENTS.all;     -- UMCL 025 pad declarations the VHDL way
use umc_vhdl_core.CORE_COMPONENTS.all;

entity chip is

  port (
    CLKxCI     : in  std_logic;
    CLKSelxSI  : in  std_logic;
    RSTxRBI    : in  std_logic;

    ScanEnxTI  : in  std_logic;
    ScanInxTI  : in  std_logic;
```

```
    ScanOutxTO : out std_logic;

    MIDIparxTI : in std_logic_vector(7 downto 0);
    DataRdyxTI : in std_logic;
    BypassARxTI: in std_logic;

    MIDIserxDI      : in  std_logic;
    MIDIChanxDI     : in  unsigned(3 downto 0);
    AudioSerSxDO    : out std_logic;
    AudioSerLLExSO  : out std_logic;
    AudioSerRLExSO  : out std_logic;
    AudioSerLRxCO   : out std_logic;
    AudioSerBitxCO  : out std_logic);
end chip;

architecture structural of chip is

  -- generated internal signals
  signal RSTIntxRB              : std_logic;  -- synchronized reset
  signal CLKDividerFeedbackxD   : std_logic;
  signal CLKDividerOutxC        : std_logic;
  signal CLKDividerRstxRB       : std_logic;  -- reset for clockdivider
  signal CLKxC                  : std_logic;  -- divided or undivided clock
  signal CLKSelxS               : std_logic;

  -- signals for connection of pads to core circuitry
  -- ( SignalxX     is the outer connection (pad),
  --   SignalIxX    is the inner connection of the pad cell)
  signal CLKIxC           : std_logic;
  signal CLKSelIxS        : std_logic;
  signal RSTIxRB          : std_logic;
  signal ScanEnIxT        : std_logic;
  signal ScanInIxT        : std_logic;
  signal ScanOutIxT       : std_logic;
  signal MIDIparIxT       : std_logic_vector(7 downto 0);
  signal DataRdyIxT       : std_logic;
  signal BypassARIxT      : std_logic;
  signal MIDIserIxD       : std_logic;
  signal MIDIChanIxD      : unsigned(3 downto 0);
  signal AudioSerSIxD     : std_logic;
  signal AudioSerLLEIxS   : std_logic;
  signal AudioSerRLEIxS   : std_logic;
  signal AudioSerLRIxC    : std_logic;
  signal AudioSerBitIxC   : std_logic;

  component top
    port (
      MIDIserxDI      : in  std_logic;
      MIDIChanxDI     : in  unsigned(3 downto 0);
      AudioSerSxDO    : out std_logic;
      AudioSerLLExSO  : out std_logic;
      AudioSerRLExSO  : out std_logic;
      AudioSerLRxCO   : out std_logic;
      AudioSerBitxCO  : out std_logic;
      ScanEnxTI       : in  std_logic;
      ScanInxTI       : in  std_logic;
      ScanOutxTO      : out std_logic;
      MIDIparxTI      : in  std_logic_vector(7 downto 0);
      DataRdyxTI      : in  std_logic;
      BypassARxTI     : in  std_logic;
      CLKxCI          : in  std_logic;
      RSTxRBI         : in  std_logic);
  end component;

begin

  ----------------------------------------------------------------------
  -- input and output pads:
  --  WC3I40 : CMOS input pad
  --  WC3O10 : CMOS output pad 2mA

  pad_CLKxCI : WC3I40
    port map (PAD => CLKxCI, DI => CLKIxC);

  pad_CLKSelxSI : WC3I40
    port map (PAD => CLKSelxSI, DI => CLKSelIxS);

  pad_RSTxRBI : WC3I40
    port map (PAD => RSTxRBI, DI => RSTIxRB);

  pad_ScanEnxTI : WC3I40
    port map (PAD => ScanEnxTI, DI => ScanEnIxT);

  pad_ScanInxTI : WC3I40
    port map (PAD => ScanInxTI, DI => ScanInIxT);

  pad_ScanOutxTO : WC3O10
    port map (DO => ScanOutIxT, PAD => ScanOutxTO);

  pad_MIDIserxDI : WC3I40
    port map (PAD => MIDIserxDI, DI => MIDIserIxD);

  generate_MIDIChanxDI_label : for i in 0 to 3 generate
    pad_MIDIChanxDI             : WC3I40
      port map (PAD => MIDIChanxDI(i), DI => MIDIChanIxD(i));
  end generate;

  generate_MIDIparxTI_label : for i in 0 to 7 generate
    pad_MIDIparxTI              : WC3I40
      port map (PAD => MIDIparxTI(i), DI => MIDIparIxT(i));
  end generate;

  pad_DataRdyxTI : WC3I40
    port map (PAD => DataRdyxTI, DI => DataRdyIxT);

  pad_BypassARxTI : WC3I40
    port map (PAD => BypassARxTI, DI => BypassARIxT);

  pad_AudioSerSxDO : WC3O10
    port map (DO => AudioSerSIxD, PAD => AudioSerSxDO);

  pad_AudioSerLLExSO : WC3O10
    port map (DO => AudioSerLLEIxS, PAD => AudioSerLLExSO);

  pad_AudioSerRLExSO : WC3O10
    port map (DO => AudioSerRLEIxS, PAD => AudioSerRLExSO);

  pad_AudioSerLRxCO : WC3O10
    port map (DO => AudioSerLRIxC, PAD => AudioSerLRxCO);

  pad_AudioSerBitxCO : WC3O10
    port map (DO => AudioSerBitIxC, PAD => AudioSerBitxCO);

  ----------------------------------------------------------------------
  -- Clock divider
  -- DRIVE STRENGTH??
  ClkDividerRegister : DFFRPB4
    port map (D => CLKDividerFeedbackxD, RB => CLKDividerRstxRB, CK => CLKIxC,
              Q => CLKDividerOutxC, QB => CLKDividerFeedbackxD);
  -- Reset Synchronizer for Clock Divider

  ClkDividerResetSynch : DFFRPB1
    port map (D => RSTIxRB, RB => RSTIxRB, CK => CLKIxC, Q => CLKDividerRstxRB);

  ClkDividerMux : MUX2D4
    port map (A0 => CLKDividerOutxC, A1 => CLKIxC, SL => CLKSelxS, Z => CLKxC);

  ClkDividerOR : OR2D2
    port map (A1 => CLKSelIxS, A2 => ScanEnIxT, Z => CLKSelxS);

  ----------------------------------------------------------------------
  -- reset synchronization (with feedforward)
```

```
  ResetSynchronizer : DFFRPB4
    port map ( D => RSTIxRB, RB => RSTIxRB, CK => CLKxC,
               Q => RSTIntxRB );

-------------------------------------------------------------
  -- instantiate the synthesizer
  core_mod : top
    port map (
      MIDIserxDI      => MIDIserIxD,
      MIDIChanxDI     => MIDIChanIxD,
      AudioSerSxDO    => AudioSerSIxD,
      AudioSerLLExSO  => AudioSerLLEIxS,
      AudioSerRLExSO  => AudioSerRLEIxS,
      AudioSerLRxCO   => AudioSerLRIxC,
      AudioSerBitxCO  => AudioSerBitIxC,
      ScanEnxTI       => ScanEnIxT,
      ScanInxTI       => ScanInIxT,
      ScanOutxTO      => ScanOutIxT,
      MIDIparxTI      => MIDIparIxT,
      DataRdyxTI      => DataRdyIxT,
      BypassARxTI     => BypassARIxT,
      CLKxCI          => CLKxC,
      RSTxRBI         => RSTIntxRB );

end structural;
```

ComplexMultiplier.vhd

```
-------------------------------------------------------------------------------
-- Title : Complex Multiplier
-- Project : MIDI Synthesizer
-------------------------------------------------------------------------------
-- File : ComplexMultiplier.vhd
-- Author : Samuel Nobs <nobssa@ee.ethz.ch>
-- Company : Integrated Systems Laboratory, ETH Zurich
-- Created : 2002/11/13
-- Last update: 2002/12/02
-- Platform : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: Multiplies one 16bit signed number with 1 9bit unsigned
-- and 3 7bit unsigned's
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions :
-- Date Version Author Description
-- 2002/11/13 1.0 Samuel Nobs Created
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ComplexMultiplier is

  port (
    -- data ports
    AmplUxDI     : in  unsigned(8 downto 0);   -- output of adsr
    ChanVolUxDI  : in  unsigned(6 downto 0);   -- channel volume
    MainVolUxDI  : in  unsigned(6 downto 0);   -- main volume
    VelUxDI      : in  unsigned(6 downto 0);   -- key velocity
    YSxDI        : in  unsigned(15 downto 0);  -- output of rectifier
    OutSxDO      : out unsigned(15 downto 0);  -- multiplication result
    -- control ports
    SelxSI       : in  std_logic_vector(1 downto 0);  -- selection of multiplicand
    InitxSI      : in  std_logic;             -- multiplicator init
    EnaRegxSI    : in  std_logic_vector(1 downto 0);  -- enable registers
    CLKxCI       : in  std_logic;             -- clock
    RSTxRBI      : in  std_logic );           -- async reset, active low

end ComplexMultiplier;

architecture rtl of ComplexMultiplier is

  -- helper signals
  signal AxD    : unsigned(14 downto 0);  -- multiplicand
  signal BxD    : unsigned(8 downto 0);   -- multiplicand
  signal CxD    : unsigned(14 downto 0);  -- output of the multiplier
  signal NegxD  : unsigned(15 downto 0);  -- negation of InRegxDN

  -- registers
  signal MulRegxDN : unsigned(14 downto 0);
  signal MulRegxDP : unsigned(14 downto 0);
  signal OutRegxDN : unsigned(15 downto 0);
  signal OutRegxDP : unsigned(15 downto 0);
  signal InRegxDN  : unsigned(15 downto 0);
  signal InRegxDP  : unsigned(15 downto 0);

  -- the multiplier we need
  component mul15u9u
    port (
      AxDI   : in  unsigned(14 downto 0);
      BxDI   : in  unsigned(8 downto 0);
      CxDO   : out unsigned(14 downto 0);
      InitxSI : in  std_logic;
      CLKxCI : in  std_logic;
      RSTxRBI : in  std_logic);
  end component;

begin  -- rtl

  -- purpose: calculate next state of input register
  -- type : combinational
  -- inputs : InitxSI, YSxDI, InRegxDP
  -- outputs: InRegxDN
  inreg_calc : process (InRegxDP, InitxSI, SelxSI, YSxDI)
  begin  -- process inreg_calc
    if InitxSI = '1' and SelxSI = "00" then  -- only update at the first InitxSO
                                             -- of four
      InRegxDN <= YSxDI;
    else
      InRegxDN <= InRegxDP;
    end if;
  end process inreg_calc;

  -- purpose: update input register
  -- type : sequential
  -- inputs : CLKxCI, RSTxRBI, InRegxDN
  -- outputs: InRegxDP
  inreg_update : process (CLKxCI, RSTxRBI)
  begin  -- process inreg_update
    if RSTxRBI = '0' then                   -- asynchronous reset (active low)
      InRegxDP <= (others => '0');
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      InRegxDP <= InRegxDN;
    end if;
  end process inreg_update;

  -- purpose: select one of the for multiplicands
  -- type   : combinational
  -- inputs : AmplUxDI,ChanColUxDI,MainVolUxDI,VelUxDI,SelxSI
  -- outputs: BxD
  mux1 : process (AmplUxDI, ChanVolUxDI, MainVolUxDI, SelxSI, VelUxDI)
  begin  -- process mux1
    case SelxSI is
      when "00"   =>
        BxD <= AmplUxDI;
      when "01"   =>
        BxD <= ChanVolUxDI&ChanVolUxDI(6 downto 5);  -- extend to 9 bits
```

```
        -- instead of filling up with zeros
        -- we use the two most significant bits
        -- for that purpose to use the full range
        -- of valid values
      when "11"   =>
        BxD <= MainVolUxDI&MainVolUxDI(6 downto 5);  -- extend to 9 bits, see above
      when "10"   =>
        BxD <= VelUxDI&VelUxDI(6 downto 5);  -- extend to 9 bits, see above
      when others => null;
    end case;
  end process mux1;

  -- negate input InRegxDP
  NegxD <= (0-InRegxDP);

  -- purpose: select second multiplicator or result of previous multiplication
  -- type   : combinational
  -- inputs : InRegxDP, MulRegxDP
  -- outputs: AxD
  mux2 : process (InRegxDP, MulRegxDP, NegxD, SelxSI)
  begin  -- process mux2
    if SelxSI = "00" then
      if InRegxDP(15) = '1' then         -- remove sign
        AxD <= NegxD(14 downto 0);
      else
        AxD <= InRegxDP(14 downto 0);
      end if;
    else
      AxD   <= MulRegxDP;
    end if;
  end process mux2;

  u_mult : mul15u9u
    port map (
      AxDI   => AxD,
      BxDI   => BxD,
      CxDO   => CxD,
      InitxSI => InitxSI,
      CLKxCI  => CLKxCI,
      RSTxRBI => RSTxRBI);

  -- purpose: if EnaRegxSI(0) is 1, store output of multiplier
  -- type   : combinational
  -- inputs : CxD, MulRegxDP, EnaRegxSI
  -- outputs: MulRegxDN
  mulreg_calc : process (CxD, EnaRegxSI, MulRegxDP)
  begin  -- process mulreg_calc
    if EnaRegxSI(0) = '1' then
      MulRegxDN <= CxD;
    else
      MulRegxDN <= MulRegxDP;
    end if;
  end process mulreg_calc;

  -- purpose: if EnaRegxSI(1) is 1, store result in output register
  -- type   : combinational
  -- inputs : OutRegxDP, MulRegxDP, EnaRegxSI
  -- outputs: OutRegxDN
  outreg_calc : process (EnaRegxSI, InRegxDP, MulRegxDP, OutRegxDP)
  begin  -- process outreg_calc
    if EnaRegxSI(1) = '1' then
      if InRegxDP(15) = '1' then
        OutRegxDN <= (0-('0'&MulRegxDP));
      else
        OutRegxDN <= '0'&MulRegxDP;
      end if;
    else
      OutRegxDN   <= OutRegxDP;
    end if;
  end process outreg_calc;

  -- purpose: update multiplication register and output register
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, OutRegxDN, MulRegxDN
  -- outputs: OutRegxDP, MulRegxDP
  regupdate : process (CLKxCI, RSTxRBI)
  begin  -- process regupdate
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      OutRegxDP <= (others => '0');
      MulRegxDP <= (others => '0');
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      OutRegxDP <= OutRegxDN;
      MulRegxDP <= MulRegxDN;
    end if;
  end process regupdate;

  -- connect output
  OutSxDO <= OutRegxDP;

end rtl;
```

ConfReg.vhd

```
-------------------------------------------------------------------------------
-- Title : Configuration Register Bank
-- Project : MIDI Synthesizer
-------------------------------------------------------------------------------
-- File : ConfReg.vhd
-- Author : Samuel Nobs <nobssa@ee.ethz.ch>
-- Company : Integrated Systems Laboratory, ETH Zurich
-- Created : 2002/11/07
-- Last update: 2002/12/13
-- Platform : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: stores all data received as midi controllers in registers
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions :
-- Date        Version Author        Description
-- 2002/11/07 1.0     Samuel Nobs   Created
-- 2002/11/07         Samuel Nobs   tested with selected vectors
-- 2002/11/22 1.01    Samuel Nobs   changed default values
-- 2002/11/28 1.10    Samuel Nobs   pan controller added
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ConfReg is

  port (
    -- inputs
    CtrlNumxDI    : in  unsigned(6 downto 0);   -- Controller Number
    CtrlValxDI    : in  unsigned(6 downto 0);   -- controller value
    RegWExSI      : in  std_logic;              -- register write enable
    -- ouputs
    RectEnaxSO    : out unsigned(6 downto 0);   -- rectifier enable
    ChanVol0UxDO  : out unsigned(6 downto 0);   -- individual volumes
    ChanVol1UxDO  : out unsigned(6 downto 0);
    ChanVol2UxDO  : out unsigned(6 downto 0);
    ChanVol3UxDO  : out unsigned(6 downto 0);
    ChanVol4UxDO  : out unsigned(6 downto 0);
    ChanVol5UxDO  : out unsigned(6 downto 0);
    ChanVol6UxDO  : out unsigned(6 downto 0);
    ChanVol7UxDO  : out unsigned(6 downto 0);
    ChanPan0UxDO  : out unsigned(6 downto 0);   -- individual panning factor
    ChanPan1UxDO  : out unsigned(6 downto 0);
    ChanPan2UxDO  : out unsigned(6 downto 0);
```

```
    ChanPan3UxDO : out unsigned(6 downto 0);
    ChanPan4UxDO : out unsigned(6 downto 0);
    ChanPan5UxDO : out unsigned(6 downto 0);
    ChanPan6UxDO : out unsigned(6 downto 0);
    ChanPan7UxDO : out unsigned(6 downto 0);
    ADSR0xDO     : out unsigned(27 downto 0);  -- adsr config data
    ADSR1xDO     : out unsigned(27 downto 0);
    ADSR2xDO     : out unsigned(27 downto 0);
    ADSR3xDO     : out unsigned(27 downto 0);
    ADSR4xDO     : out unsigned(27 downto 0);
    ADSR5xDO     : out unsigned(27 downto 0);
    ADSR6xDO     : out unsigned(27 downto 0);
    ADSR7xDO     : out unsigned(27 downto 0);
    MainVolUxDO  : out unsigned(6 downto 0);
    -- control
    CLKxCI       : in  std_logic;
    RSTxRBI      : in  std_logic);            -- main volume

end ConfReg;

architecture rtl of ConfReg is

  component reg7b
    generic (
      NUM     :     integer;
      DEFAULT :     integer);
    port (
      ENAxSI  : in  std_logic;
      SELxDI  : in  unsigned(6 downto 0);
      REGxDI  : in  unsigned(6 downto 0);
      REGxDO  : out unsigned(6 downto 0);
      CLKxCI  : in  std_logic;
      RSTxRBI : in  std_logic);
  end component;


  type pardefs is array (0 to 7) of integer;
  -- default settings for parameters
  --
  -- oscillator:                 0   1   2   3   4   5   6   7
  constant  A_DEF : pardefs := ( 27,  0, 14,127, 22,  3,  0,127);  -- A
  constant  D_DEF : pardefs := ( 32,  9, 22,  0, 35,127,  0, 48);  -- D
  constant  S_DEF : pardefs := ( 44,  0,  0,127, 35,  0,127, 11);  -- S
  constant  R_DEF : pardefs := ( 63,  5, 11,  3, 35, 41,  0, 18);  -- R
  constant  P_DEF : pardefs := ( 64,  0,127,123, 40, 94, 64, 92);  -- Pan
  constant VL_DEF : pardefs := (127, 92, 20, 52, 10, 10,  0, 10);  -- Vol

  constant MV_DEF : integer := 127;     -- default main volume
  constant RC_DEF : integer := 0;       -- default for rectify

  signal ChanVolxD : unsigned(55 downto 0);   -- all volumes in one signal
  signal ChanPanxD : unsigned(55 downto 0);   -- all pans in one signal
  signal ADSRxD    : unsigned(223 downto 0);  -- all adsr data in one signal
begin  -- rtl

  -- instantiate rectify enable register
  u_reg_rect : reg7b
    generic map (
      NUM     => 9,
      DEFAULT => RC_DEF)
    port map (
      ENAxSI  => RegWExSI,
      SELxDI  => CtrlNumxDI,
      REGxDI  => CtrlValxDI,
      REGxDO  => RectEnaxSO,
      CLKxCI  => CLKxCI,
      RSTxRBI => RSTxRBI);

  -- instantiate main volume register
  u_reg_mvol : reg7b
    generic map (
      NUM     => 7,
      DEFAULT => MV_DEF)
    port map (
      ENAxSI  => RegWExSI,
      SELxDI  => CtrlNumxDI,
      REGxDI  => CtrlValxDI,
      REGxDO  => MainVolUxDO,
      CLKxCI  => CLKxCI,
      RSTxRBI => RSTxRBI);

  -- instantiate volume registers
  volumes      : for vol in 0 to 7 generate
  begin
    u_reg_vol : reg7b
      generic map (
        NUM     => 102+vol,
        DEFAULT => VL_DEF(vol))
      port map (
        ENAxSI  => RegWExSI,
        SELxDI  => CtrlNumxDI,
        REGxDI  => CtrlValxDI,
        REGxDO  => ChanVolxD(7*(vol+1)-1 downto 7*vol),
        CLKxCI  => CLKxCI,
        RSTxRBI => RSTxRBI);

  end generate volumes;
  -- assign volumes
  ChanVol0UxDO <= ChanVolxD(6 downto 0);
  ChanVol1UxDO <= ChanVolxD(13 downto 7);
  ChanVol2UxDO <= ChanVolxD(20 downto 14);
  ChanVol3UxDO <= ChanVolxD(27 downto 21);
  ChanVol4UxDO <= ChanVolxD(34 downto 28);
  ChanVol5UxDO <= ChanVolxD(41 downto 35);
  ChanVol6UxDO <= ChanVolxD(48 downto 42);
  ChanVol7UxDO <= ChanVolxD(55 downto 49);

  -- instantiate pan registers
  panoramas : for pan in 0 to 7 generate
    u_pan     : reg7b
      generic map (
        NUM     => 110 + pan,
        DEFAULT => P_DEF(pan))
      port map (
        ENAxSI  => REGWExSI,
        SELxDI  => CtrlNumxDI,
        REGxDI  => CtrlValxDI,
        REGxDO  => ChanPanxD(7*(pan+1)-1 downto 7*pan),
        CLKxCI  => CLKxCI,
        RSTxRBI => RSTxRBI);

  end generate panoramas;
  -- assign panoramas
  ChanPan0UxDO <= ChanPanxD(6 downto 0);
  ChanPan1UxDO <= ChanPanxD(13 downto 7);
  ChanPan2UxDO <= ChanPanxD(20 downto 14);
  ChanPan3UxDO <= ChanPanxD(27 downto 21);
  ChanPan4UxDO <= ChanPanxD(34 downto 28);
  ChanPan5UxDO <= ChanPanxD(41 downto 35);
  ChanPan6UxDO <= ChanPanxD(48 downto 42);
  ChanPan7UxDO <= ChanPanxD(55 downto 49);


  -- instantiate adsr registers
  adsrs        : for adsr in 0 to 7 generate
    u_adsr_A : reg7b
      generic map (
        NUM     => 4*adsr+64,
        DEFAULT => A_DEF(adsr))
      port map (
        ENAxSI  => REGWExSI,
        SELxDI  => CtrlNumxDI,
```

```
            REGxDI   => CtrlValxDI,
            REGxDO   => ADSRxD(28*adsr+6 downto 28*adsr),
            CLKxCI   => CLKxCI,
            RSTxRBI => RSTxRBI);

        u_adsr_D : reg7b
          generic map (
            NUM      => 4*adsr+65,
            DEFAULT => D_DEF(adsr))
          port map (
            ENAxSI   => REGWExSI,
            SELxDI   => CtrlNumxDI,
            REGxDI   => CtrlValxDI,
            REGxDO   => ADSRxD(28*adsr+13 downto 7+28*adsr),
            CLKxCI   => CLKxCI,
            RSTxRBI => RSTxRBI);

        u_adsr_S : reg7b
          generic map (
            NUM      => 4*adsr+66,
            DEFAULT => S_DEF(adsr))
          port map (
            ENAxSI   => REGWExSI,
            SELxDI   => CtrlNumxDI,
            REGxDI   => CtrlValxDI,
            REGxDO   => ADSRxD(28*adsr+20 downto 14+28*adsr),
            CLKxCI   => CLKxCI,
            RSTxRBI => RSTxRBI);

        u_adsr_R : reg7b
          generic map (
            NUM      => 4*adsr+67,
            DEFAULT => R_DEF(adsr))
          port map (
            ENAxSI   => REGWExSI,
            SELxDI   => CtrlNumxDI,
            REGxDI   => CtrlValxDI,
            REGxDO   => ADSRxD(28*adsr+27 downto 21+28*adsr),
            CLKxCI   => CLKxCI,
            RSTxRBI => RSTxRBI);

    end generate adsrs;
    -- assign adsr data
    ADSR0xDO <= ADSRxD(27 downto 0);
    ADSR1xDO <= ADSRxD(55 downto 28);
    ADSR2xDO <= ADSRxD(83 downto 56);
    ADSR3xDO <= ADSRxD(111 downto 84);
    ADSR4xDO <= ADSRxD(139 downto 112);
    ADSR5xDO <= ADSRxD(167 downto 140);
    ADSR6xDO <= ADSRxD(195 downto 168);
    ADSR7xDO <= ADSRxD(223 downto 196);
end rtl;
```

i2scontroller.vhd

```
-------------------------------------------------------------------------------
-- Title      : Philips I2S DA-Format Controller
-- Project    :
-------------------------------------------------------------------------------
-- File       : i2scontroller.vhd
-- Author     : Samuel Nobs <nobssa@ee.ethz.ch>
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/11/14
-- Last update: 2003/01/31
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: Converts parallel audio data to the widely used
--              I2S format, which is a serial format
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author          Description
-- 2002/11/14  1.0      Samuel Nobs     Created
-- 2002/11/28  1.01     Samuel Nobs     added support for stereo output
-- 2002/12/06  1.02     Daniel Engeler  added support for latched DACs
-------------------------------------------------------------------------------

library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;

entity i2scontroller is

  port (
    RParxDI : in  unsigned(15 downto 0);  -- parallel audio data, left side
    LParxDI : in  unsigned(15 downto 0);  -- parallel audio data, right side
    SerxDO  : out std_logic;              -- serial audio data
    LRxCO   : out std_logic;              -- channel clock
    BitxCO  : out std_logic;              -- bit clock
    RLExSO  : out std_logic;              -- latch enable right side
    LLExSO  : out std_logic;              -- latch enable left side
    CLKxCI  : in  std_logic;              -- internal clock
    RSTxRBI : in  std_logic);             -- async reset, active low
end i2scontroller;

architecture rtl of i2scontroller is

  -- counter
  signal CountxDN : unsigned(6 downto 0);
  signal CountxDP : unsigned(6 downto 0);

  -- PISO register
  signal PISOxDN : unsigned(15 downto 0);
  signal PISOxDP : unsigned(15 downto 0);

  -- input register
  signal RParxDN, RParxDP : unsigned(15 downto 0);
  signal LParxDN, LParxDP : unsigned(15 downto 0);

begin  -- rtl

  -- increment counter
  CountxDN <= CountxDP+1;

  -- purpose: update counter
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, CountxDN
  -- outputs: CountxDP
  count_assign : process (CLKxCI, RSTxRBI)
  begin  -- process count_assign
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      CountxDP <= (others => '0');
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      CountxDP <= CountxDN;
    end if;
  end process count_assign;

  -- purpose: set Latch Enable (LE) outputs
  -- type   : combinational
  -- inputs : CountxDP
  -- outputs: LLExSO, RLExSO
  LE_calc : process (CountxDP)
  begin  -- process LE_calc
    -- left channel
    if CountxDP >= 2 and CountxDP <= 33 then
--    if CountxDP = 32 or CountxDP = 33 then
      LLExSO <= '1';
    else
```

```
      LLExSO <= '0';
    end if;

    -- right channel
    if CountxDP >= 66 and CountxDP <= 97 then
--    if CountxDP = 96 or CountxDP = 97 then
      RLExSO <= '1';
    else
      RLExSO <= '0';
    end if;
  end process LE_calc;

  -- purpose: calculate next value for PISO register
  -- type   : combinational
  -- inputs : CountxDP,PISOxDP,ParxD
  -- outputs: PISOxDN
  piso_calc : process (CountxDP, LParxDP, PISOxDP, RParxDP)
  begin  -- process piso_calc
    PISOxDN  <= PISOxDP;
    if CountxDP = 1 then
      PISOxDN <= LParxDP;                   -- read parallel data, left side
    elsif CountxDP = 65 then
      PISOxDN <= RParxDP;                   -- read parallel data, right side
    elsif CountxDP(0) = '1' then
      PISOxDN <= PISOxDP(14 downto 0)&'0';  -- shift register content up
    end if;
  end process piso_calc;

  -- purpose: calculate input registers
  -- type   : combinational
  -- inputs : RParxDI, LParxDI, RParxDP, LParxDP, CountxDP
  -- outputs: RParxDN, LParxDN, MonoParxDN
  inreg_calc : process (CountxDP, LParxDI, LParxDP, RParxDI, RParxDP)
  begin  -- process inreg_calc
    if CountxDP = 0 then
      RParxDN <= RParxDI;
      LParxDN <= LParxDI;
    else
      RParxDN <= RParxDP;
      LParxDN <= LParxDP;
    end if;
  end process inreg_calc;

  -- purpose: update input registers
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, RParxDN, LParxDN
  -- outputs: RParxDP, LParxDP
  inreg_update : process (CLKxCI, RSTxRBI)
  begin  -- process inreg_update
    if RSTxRBI = '0' then                   -- asynchronous reset (active low)
      RParxDP <= (others => '0');
      LParxDP <= (others => '0');
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      RParxDP <= RParxDN;
      LParxDP <= LParxDN;
    end if;
  end process inreg_update;

  -- purpose: update PISO register
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, PISOxDN
  -- outputs: PISOxDP
  piso_assign : process (CLKxCI, RSTxRBI)
  begin  -- process piso_assign
    if RSTxRBI = '0' then                   -- asynchronous reset (active low)
      PISOxDP <= (others => '0');
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      PISOxDP <= PISOxDN;
    end if;
  end process piso_assign;

  -- assign outputs
  LRxCO  <= CountxDP(6);
  BitxCO <= CountxDP(0);
  SerxDO <= PISOxDP(15);
end rtl;
```

LUT.vhd

```
-------------------------------------------------------------------------------
-- Title      : Lookup Table
-- Project    :
-------------------------------------------------------------------------------
-- File       : LUT.vhd
-- Author     : sem02w5/danengel <sem02w5@badile3.ee.ethz.ch>
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/11/13
-- Last update: 2002/12/09
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: Generates the start values YlxDI for the oscillators and
-- initalizes them.
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author  Description
-- 2002/11/13  1.0      danengel    Created
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-------------------------------------------------------------------------------
-- description of I/O signals
-------------------------------------------------------------------------------

-- ClkxCI: Clock

-- ResetxRBI: Asynchronous reset, active low

-- InitxSI: Set InitxSI and NoteNumxDI for one cycle to start the
-- calculation of eight consecutive initial values which are output to
-- the connected oscillators.

-- NoteNumxDI: The MIDI note number for which appropriate initial
-- values for the oscillators should be output.  Set together with
-- InitxSI.

-- OscInitxSO: Eight bits which are each connected to a synthesizer to
-- tell them to initialize.

-- MutexSO: The mute output which is connected to all oscillators.  If
-- the frequency of an oscillator would get too high (>20 kHz), mute
-- it so it won't produce some meaningless aliased output.

-- YlxDO: The initial value output which is connected to all
-- oscillators.


-- Note that MutexSI and YlxDI are handled by a connected oscillator
-- only if its InitxSI input is set.  Because of that we can connect
-- one single MutexSO and YlxDO to all of them.

-- The table is generated by a MATLAB-script which simply calculates
-- Yl = round(2*pi*f / (44100*4) * 2^26) for all frequencies
-- corresponding to notes 36 up to 107.  Note 69 is exactly 440 Hz,
-- the factor between two consecutive notes is 2^(1/12).
```

```
entity LUT is
  port (
    ClkxCI      : in  std_logic;
    ResetxRBI   : in  std_logic;
    InitxSI     : in  std_logic;
    NoteNumxDI  : in  unsigned(6 downto 0);
    OscInitxSO  : out unsigned(7 downto 0);
    MutexSO     : out std_logic;
    Y1xDO       : out unsigned(24 downto 0));
end LUT;

architecture RTL of LUT is
  -- registers
  signal AccuxDN, AccuxDP        : unsigned(27 downto 0);
  signal Y10xDN, Y10xDP          : unsigned(27 downto 0);
  signal OscInitxSN, OscInitxSP  : unsigned(7 downto 0);
  -- helpers
  signal T1xD                    : unsigned(27 downto 0);
  signal T2xD                    : unsigned(27 downto 0);

begin

  -- outputs
  OscInitxSO <= OscInitxSP;
  Y1xDO      <= AccuxDP(27 downto 3);

  -- OscInit
  process(InitxSI, OscInitxSP)
  begin
    if InitxSI = '1' then
      OscInitxSN <= "00000001";
    else
      -- shift left logical
      OscInitxSN <= OscInitxSP(6 downto 0) & '0';
    end if;
  end process;

  -- MutexDO
  process (AccuxDP)
  begin
    if AccuxDP > 47806965 then          -- round(2*pi*20000*2^26/(44100*4))
      MutexSO <= '1';
    else
      MutexSO <= '0';
    end if;
  end process;

  -- AccuxD = here we accumulate Y1xD for the oscillator
  process(InitxSI, T1xD, T2xD)
  begin
    if InitxSI = '1' then
      AccuxDN <= T1xD;
    else
      AccuxDN <= T2xD;
    end if;
  end process;

  -- Y10xD = initial value for the first oscillator
  process(InitxSI, T1xD, Y10xDP)
  begin
    if InitxSI = '1' then
      Y10xDN <= T1xD;
    else
      Y10xDN <= Y10xDP;
    end if;
  end process;

  -- Add Y10xD to AccuxD
  T2xD <= AccuxDP + Y10xDP;

  -- get Y10xD
  process(NoteNumxDI)
  begin
    case to_integer(NoteNumxDI) is
      -- this part generated by lut.m
-------------------------------------------------------------------------------
      when 0      => T1xD <= "0000000000100110001010111000";
      when 1      => T1xD <= "0000000000100110001010111000";
      when 2      => T1xD <= "0000000000100110001010111000";
      when 3      => T1xD <= "0000000000100110001010111000";
      when 4      => T1xD <= "0000000000100110001010111000";
      when 5      => T1xD <= "0000000000100110001010111000";
      when 6      => T1xD <= "0000000000100110001010111000";
      when 7      => T1xD <= "0000000000100110001010111000";
      when 8      => T1xD <= "0000000000100110001010111000";
      when 9      => T1xD <= "0000000000100110001010111000";
      when 10     => T1xD <= "0000000000100110001010111000";
      when 11     => T1xD <= "0000000000100110001010111000";
      when 12     => T1xD <= "0000000000100110001010111000";
      when 13     => T1xD <= "0000000000100110001010111000";
      when 14     => T1xD <= "0000000000100110001010111000";
      when 15     => T1xD <= "0000000000100110001010111000";
      when 16     => T1xD <= "0000000000100110001010111000";
      when 17     => T1xD <= "0000000000100110001010111000";
      when 18     => T1xD <= "0000000000100110001010111000";
      when 19     => T1xD <= "0000000000100110001010111000";
      when 20     => T1xD <= "0000000000100110001010111000";
      when 21     => T1xD <= "0000000000100110001010111000";
      when 22     => T1xD <= "0000000000100110001010111000";
      when 23     => T1xD <= "0000000000100110001010111000";
      when 24     => T1xD <= "0000000000100110001010111000";
      when 25     => T1xD <= "0000000000100110001010111000";
      when 26     => T1xD <= "0000000000100110001010111000";
      when 27     => T1xD <= "0000000000100110001010111000";
      when 28     => T1xD <= "0000000000100110001010111000";
      when 29     => T1xD <= "0000000000100110001010111000";
      when 30     => T1xD <= "0000000000100110001010111000";
      when 31     => T1xD <= "0000000000100110001010111000";
      when 32     => T1xD <= "0000000000100110001010111000";
      when 33     => T1xD <= "0000000000100110001010111000";
      when 34     => T1xD <= "0000000000100110001010111000";
      when 35     => T1xD <= "0000000000100110001010111000";
      when 36     => T1xD <= "0000000000100110001010111000";
      when 37     => T1xD <= "0000000000101000011100001001";
      when 38     => T1xD <= "0000000000101010110110000010";
      when 39     => T1xD <= "0000000000101101011001000101";
      when 40     => T1xD <= "0000000000110000000101110101";
      when 41     => T1xD <= "0000000000111001011110011110";
      when 42     => T1xD <= "0000000000111010111111010000";
      when 43     => T1xD <= "0000000000111001001100001011";
      when 44     => T1xD <= "0000000000111100100101110101";
      when 45     => T1xD <= "0000000001000000001100011010";
      when 46     => T1xD <= "0000000001000100000000101101";
      when 47     => T1xD <= "0000000001001000000011100010";
      when 48     => T1xD <= "0000000001001100010101110000";
      when 49     => T1xD <= "0000000001010000111000010010";
      when 50     => T1xD <= "0000000001010101101100000101";
      when 51     => T1xD <= "0000000001011010110010001011";
      when 52     => T1xD <= "0000000001100000001011101010";
      when 53     => T1xD <= "0000000001100101111001101101";
      when 54     => T1xD <= "0000000001101011111101100000";
      when 55     => T1xD <= "0000000001110010011000010111";
      when 56     => T1xD <= "0000000001111001001011101001";
      when 57     => T1xD <= "0000000010000000011000110101";
      when 58     => T1xD <= "0000000010001000000000101101";
      when 59     => T1xD <= "0000000010010000000111000101";
      when 60     => T1xD <= "0000000010011000101011100000";
      when 61     => T1xD <= "0000000010100001110000100011";
      when 62     => T1xD <= "0000000010101011011000001001";
      when 63     => T1xD <= "0000000010110101100100010110";
      when 64     => T1xD <= "0000000011000000010111010101";
```

```
        when 65     => TlxD <= "00000000110010111100110110011";
        when 66     => TlxD <= "00000000111010111111011000000";
        when 67     => TlxD <= "00000000111001001100001011110";
        when 68     => TlxD <= "00000000111100100101110100011";
        when 69     => TlxD <= "00000001000000001100011010011";
        when 70     => TlxD <= "00000001000100000000010110110";
        when 71     => TlxD <= "00000001001000000011100011001";
        when 72     => TlxD <= "00000001001100010101111000000";
        when 73     => TlxD <= "00000001010000011100001000110";
        when 74     => TlxD <= "00000001010101101110000010010";
        when 75     => TlxD <= "00000001011010110010001011100";
        when 76     => TlxD <= "00000001100000001011101010011";
        when 77     => TlxD <= "00000001100101111001101110010";
        when 78     => TlxD <= "00000001101011111101011111111";
        when 79     => TlxD <= "00000001110010011000010111011";
        when 80     => TlxD <= "00000001111001001011010011O";
        when 81     => TlxD <= "00000010000000011000110100l0";
        when 82     => TlxD <= "00000010001000000001011010ll";
        when 83     => TlxD <= "00000010010000000111000100l0";
        when 84     => TlxD <= "00000010011000101011110000001";
        when 85     => TlxD <= "00000010100001110000100011OO";
        when 86     => TlxD <= "00000010101011011000001001OO";
        when 87     => TlxD <= "00000010110101100100010101ll";
        when 88     => TlxD <= "00000011000000010111010100ll";
        when 89     => TlxD <= "00000011001011110011011001OO";
        when 90     => TlxD <= "00000011010111111010111111l0";
        when 91     => TlxD <= "00000011100100110000101101l0";
        when 92     => TlxD <= "00000011110010010111010010ll";
        when 93     => TlxD <= "00000100000000011000110100l0l";
        when 94     => TlxD <= "00000100010000000010110101ll";
        when 95     => TlxD <= "00000100100000001110001001OO";
        when 96     => TlxD <= "00000100110001010111000000l0";
        when 97     => TlxD <= "00000101000011100001000110OO";
        when 98     => TlxD <= "00000101010110110000001001000";
        when 99     => TlxD <= "00000101101011001000101011ll";
        when 100    => TlxD <= "00000110000000101110101001O1";
        when 101    => TlxD <= "00000110010111100110110010O1";
        when 102    => TlxD <= "00000110101111110101111111OO";
        when 103    => TlxD <= "00000111001001100001011011O1";
        when 104    => TlxD <= "00000111100100101110100l0lll";
        when 105    => TlxD <= "00001000000001100011010010l0";
        when 106    => TlxD <= "00001000100000000101101011l0";
        when 107    => TlxD <= "00001001000000011100010010O1";
        when 108    => TlxD <= "00001001000000011100010010O1";
        when 109    => TlxD <= "00001001000000011100010010O1";
        when 110    => TlxD <= "00001001000000011100010010O1";
        when 111    => TlxD <= "00001001000000011100010010O1";
        when 112    => TlxD <= "00001001000000011100010010O1";
        when 113    => TlxD <= "00001001000000011100010010O1";
        when 114    => TlxD <= "00001001000000011100010010O1";
        when 115    => TlxD <= "00001001000000011100010010O1";
        when 116    => TlxD <= "00001001000000011100010010O1";
        when 117    => TlxD <= "00001001000000011100010010O1";
        when 118    => TlxD <= "00001001000000011100010010O1";
        when 119    => TlxD <= "00001001000000011100010010O1";
        when 120    => TlxD <= "00001001000000011100010010O1";
        when 121    => TlxD <= "00001001000000011100010010O1";
        when 122    => TlxD <= "00001001000000011100010010O1";
        when 123    => TlxD <= "00001001000000011100010010O1";
        when 124    => TlxD <= "00001001000000011100010010O1";
        when 125    => TlxD <= "00001001000000011100010010O1";
        when 126    => TlxD <= "00001001000000011100010010O1";
        when 127    => TlxD <= "00001001000000011100010010O1";
-------------------------------------------------------------------------------
        when others => TlxD <= "00001001000000011100010010O1";
      end case;
    end process;

    -- update registers
    process(ClkxCI, ResetxRBI)
    begin
      if ResetxRBI = '0' then
        AccuxDP      <= (others => '0');
        Y10xDP       <= (others => '0');
        OscInitxSP   <= (others => '0');
      elsif ClkxCI'event and ClkxCI = '1' then
        AccuxDP      <= AccuxDN;
        Y10xDP       <= Y10xDN;
        OscInitxSP   <= OscInitxSN;
      end if;
    end process;

  end RTL;
```

MIDIcontroller.vhd

```
  -------------------------------------------------------------------------------
  -- Title      : MIDI controller
  -- Project    : MIDI Synthesizer
  -------------------------------------------------------------------------------
  -- File       : MIDIcontroller.vhd
  -- Author     : Samuel Nobs <nobssa@ee.ethz.ch>
  -- Company    : Integrated Systems Laboratory, ETH Zurich
  -- Created    : 2002/11/04
  -- Last update: 2002/12/10
  -- Platform   : ModelSim (simulation), Synopsys (synthesis)
  -------------------------------------------------------------------------------
  -- Description: processes the midi bytes read from the async receiver
  --              and outputs the corresponding control signals
  -------------------------------------------------------------------------------
  -- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
  -------------------------------------------------------------------------------
  -- Revisions  :
  -- Date        Version  Author       Description
  -- 2002/11/04  1.0      Samuel Nobs  Created, basic FSM implemented
  -- 2002/11/06  1.1      Samuel Nobs  Completed
  -- 2002/11/06           Samuel Nobs  RTL and Netlist verified using
  --                                   selected vectors
  -------------------------------------------------------------------------------

  library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

  entity MIDIcontroller is

    port (
      MIDIparxDI   : in  std_logic_vector(7 downto 0);  -- the midi byte
      DataRdyxSI   : in  std_logic;         -- tells when byte shall be read
      MChannelxDI  : in  unsigned(3 downto 0);  -- midi channel
      CtrNumxDO    : out unsigned(6 downto 0);  -- controller number
      CtrValxDO    : out unsigned(6 downto 0);  -- controller value
      RegWExSO     : out std_logic;         -- register write enable
      NoteNumxDO   : out unsigned(6 downto 0);  -- note number
      NoteVelxDO   : out unsigned(6 downto 0);  -- note velocity
      NoteGatexSO  : out std_logic;         -- note gate: key pressed or not ?
      LUTInitxSO   : out std_logic;         -- init signal for the LUT
      CLKxCI       : in  std_logic;         -- clock
      RSTxRBI      : in  std_logic);        -- async reset, active low

  end MIDIcontroller;

  architecture rtl of MIDIcontroller is

    -- states of the controller fsm
    type controllerFsmSt is (stInit,
                             stNoteOn,
                             stNtNum,
                             stNtVel,
```

```
                                stNtVel0,
                                stNoteOff,
                                stNOffNum,
                                stNOffVel,
                                stControl,
                                stCtrlNum,
                                stCtrlVal,
                                stSysEx);     -- the states
  signal CtrlFSMxDP : controllerFsmSt;  -- previous state
  signal CtrlFSMxDN : controllerFsmSt;  -- next state

  -- types
  subtype MIDIByte is unsigned(7 downto 0);  -- represents a midi byte

  -- helper signals
  signal MIDIxD : MIDIByte;              -- the midi byte, unsigned

  -- constants
  constant mReset      : MIDIByte := to_unsigned(255, 8);  -- reset message
  constant mNoteOn     : MIDIByte := to_unsigned(144, 8);  -- note on message
  constant mNoteOff    : MIDIByte := to_unsigned(128, 8);  -- note off message
  constant mContChange : MIDIByte := to_unsigned(176, 8);  -- control change message
  constant mSysReal    : MIDIByte := to_unsigned(248, 8);  -- lowest val. for sys real
  constant mDataByte   : MIDIByte := to_unsigned(128, 8);  -- data bytes are <128

  -- output registers
  signal GatexDP : std_logic;            -- register for note gate
  signal GatexDN : std_logic;

  signal NoteNumxDP : unsigned(6 downto 0);  -- register for note numbers
  signal NoteNumxDN : unsigned(6 downto 0);

  signal NoteVelxDP : unsigned(6 downto 0);  -- register for note velocity
  signal NoteVelxDN : unsigned(6 downto 0);

  signal CtrNumxDP : unsigned(6 downto 0);  -- register for the
  signal CtrNumxDN : unsigned(6 downto 0);  --       controller number

  signal CtrValxDP : unsigned(6 downto 0);  -- register for the
  signal CtrValxDN : unsigned(6 downto 0);  --       controller value

  signal RegWExDP : std_logic;           -- register for the WE signal
  signal RegWExDN : std_logic;

  signal LUTinitxDP : std_logic;         -- register for the lut init
  signal LUTinitxDN : std_logic;         -- signal


  -- helper register
  signal NoteNumTempxDP : unsigned(6 downto 0);  -- buffers the note number
  signal NoteNumTempxDN : unsigned(6 downto 0);


begin  -- rtl

  MIDIxD <= unsigned(MIDIparxDI);        -- its more fun to calculate with unsigned's

  -- purpose: processes the midi bytes
  -- type   : combinational
  -- inputs : DataRdyxSI
  -- outputs:
  fsmcalculate : process (CtrNumxDP, CtrValxDP, CtrlFSMxDP, DataRdyxSI,
                          GatexDP, MChannelxDI, MIDIxD, NoteNumTempxDP,
                          NoteNumxDP, NoteVelxDP)
  begin  -- process fsmcalculate
    -- defaults
    CtrlFSMxDN                <= CtrlFSMxDP;
    GatexDN                   <= GatexDP;
    NoteNumxDN                <= NoteNumxDP;
    NoteVelxDN                <= NoteVelxDP;
    NoteNumTempxDN            <= NoteNumTempxDP;
    CtrNumxDN                 <= CtrNumxDP;
    CtrValxDN                 <= CtrValxDP;
    RegWExDN                  <= '0';
    LUTinitxDN                <= '0';
    -- nondefaults
    if DataRdyxSI = '1' then              -- byte waits for being read
      if MIDIxD < mDataByte then          -- midi byte is a data byte
        case CtrlFSMxDP is
          when stInit     =>
            CtrlFSMxDN      <= stInit;
          when stNoteOn   =>
            CtrlFSMxDN      <= stNtNum;
            NoteNumTempxDN  <= MIDIxD(6 downto 0);
          when stNtNum    =>
            if MIDIxD = 0 then
              CtrlFSMxDN    <= stNtVel0;
              if NoteNumxDP = NoteNumTempxDP then  -- don't stop note if the key
                                                   -- released is not the same as
                                                   -- the last key pressed
                GatexDN     <= '0';
              end if;
            else
              CtrlFSMxDN    <= stNtVel;
              GatexDN       <= '1';
              LUTinitxDN    <= '1';
              if GatexDP = '0' then        -- don't assign velocity if another key
                                           -- is pressed already
                NoteVelxDN  <= MIDIxD(6 downto 0);
              end if;
              NoteNumxDN    <= NoteNumTempxDP;
            end if;
          when stNtVel    =>
            CtrlFSMxDN      <= stNtNum;  -- running status
            NoteNumTempxDN  <= MIDIxD(6 downto 0);
          when stNtVel0   =>
            CtrlFSMxDN      <= stNtNum;  -- running status
            NoteNumTempxDN  <= MIDIxD(6 downto 0);
          when stNoteOff =>
            CtrlFSMxDN      <= stNOffNum;
            if NoteNumxDP = MIDIxD then  -- don't stop note if the key
                                         -- released
                                         -- is not the same as the key pressed last
              GatexDN       <= '0';
            end if;
          when stNOffNum =>
            CtrlFSMxDN      <= stNOffVel;
          when stNOffVel =>
            CtrlFSMxDN      <= stNOffNum;  -- running status
            if NoteNumxDP = MIDIxD then  -- don't stop note if the key
                                         -- released
                                         -- is not the same as the key pressed last
              GatexDN       <= '0';
            end if;
          when stControl =>
            CtrlFSMxDN      <= stCtrlNum;
            CtrNumxDN       <= MIDIxD(6 downto 0);
          when stCtrlNum =>
            CtrlFSMxDN      <= stCtrlVal;
            CtrValxDN       <= MIDIxD(6 downto 0);
            RegWExDN        <= '1';
          when stCtrlVal =>
            CtrlFSMxDN      <= stCtrlNum;  -- running status
            CtrNumxDN       <= MIDIxD(6 downto 0);
          when stSysEx    =>
            CtrlFSMxDN      <= stSysEx;  -- loop while no other status byte is received
          when others     =>
            CtrlFSMxDN      <= stInit;   -- so we're on the safe side
        end case;
      else
```

```
        -- status bytes
        if MIDIxD = mReset then           -- the only system realtime message implemented
          CtrlFSMxDN          <= stInit;   -- reset
          GatexDN             <= '0';
          NoteNumxDN          <= to_unsigned(36, 7);  -- the lowest note
          NoteVelxDN          <= to_unsigned(0, 7);   -- no velocity
          CtrNumxDN           <= to_unsigned(0, 7);   -- controller 0
          CtrValxDN           <= to_unsigned(0, 7);   -- value 0
          RegWExDN            <= '0';       -- disable register write
          LUTinitxDN          <= '0';
        elsif MIDIxD = MChannelxDI+mNoteOn then
          CtrlFSMxDN          <= stNoteOn;
        elsif MIDIxD = MChannelxDI+mNoteOff then
          CtrlFSMxDN          <= stNoteOff;
        elsif MIDIxD = MChannelxDI+mContChange then
          CtrlFSMxDN          <= stControl;
        else
          CtrlFSMxDN          <= CtrlFSMxDP;  -- ignore system realtime messages
          if MIDIxD < mSysReal then       -- sysex or unimplemented message
            CtrlFSMxDN        <= stSysEx;
          end if;
        end if;
      end if;
    end if;
  end process fsmcalculate;

  -- purpose: update output registers
  -- type : sequential
  -- inputs : CLKxCI, RSTxRBI
  -- outputs:
  outregassign : process (CLKxCI, RSTxRBI)
  begin  -- process outregassign
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      GatexDP      <= '0';
      NoteNumxDP   <= to_unsigned(36, 7);  -- the lowest note
      NoteVelxDP   <= to_unsigned(0, 7);   -- no velocity
      CtrNumxDP    <= to_unsigned(0, 7);   -- controller 0
      CtrValxDP    <= to_unsigned(0, 7);   -- value 0
      RegWExDP     <= '0';                 -- disable register write
      LUTinitxDP   <= '0';
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      GatexDP      <= GatexDN;
      NoteNumxDP   <= NoteNumxDN;
      NoteVelxDP   <= NoteVelxDN;
      CtrNumxDP    <= CtrNumxDN;
      CtrValxDP    <= CtrValxDN;
      RegWExDP     <= RegWExDN;
      LUTinitxDP   <= LUTinitxDN;
    end if;
  end process outregassign;

  -- purpose: update buffering registers
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, NoteNumTmpxDN
  -- outputs: NoteNumTempxDP
  tempregassign : process (CLKxCI, RSTxRBI)
  begin  -- process tempregassign
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      NoteNumTempxDP <= to_unsigned(0, 7);
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      NoteNumTempxDP <= NoteNumTempxDN;
    end if;
  end process tempregassign;

  -- connect output registers to outer world
  NoteGatexSO <= GatexDP;
  NoteNumxDO  <= NoteNumxDP;
  NoteVelxDO  <= NoteVelxDP;
  CtrNumxDO   <= CtrNumxDP;
  CtrValxDO   <= CtrValxDP;
  RegWExSO    <= RegWExDP;
  LUTinitxSO  <= LUTinitxDP;

  -- purpose: update the controller fsm
  -- type : sequential
  -- inputs : CLKxCI, RSTxRBI, CtrlFSMxDN
  -- outputs: CtrlFSMxDP
  fsmupdate : process (CLKxCI, RSTxRBI)
  begin  -- process fsmupdate
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      CtrlFSMxDP <= stInit;
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      CtrlFSMxDP <= CtrlFSMxDN;
    end if;
  end process fsmupdate;

end rtl;
```

MultiplyController.vhd

```
-------------------------------------------------------------------------------
-- Title      : Multiplier controller
-- Project    :
-------------------------------------------------------------------------------
-- File       : MultiplyController.vhd
-- Author     : Samuel Nobs <nobssa@ee.ethz.ch>
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/11/13
-- Last update: 2002/11/28
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: Controls the multipliers using a 7bit counter
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author           Description
-- 2002/11/13  1.0      Samuel Nobs      Created
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity MultiplyController is

  port (
    SelxSO     : out std_logic_vector(1 downto 0);  -- select multiplication input
    InitxSO    : out std_logic;           -- tell multiplier to start
    EnaRegxSO  : out std_logic_vector(1 downto 0);  -- tell registers to store value
    CLKxCI     : in  std_logic;           -- clock
    RSTxRBI    : in  std_logic);          -- async reset, active low

end MultiplyController;

architecture rtl of MultiplyController is

  -- registers
  signal Sel0xSN     : std_logic;
  signal Sel0xSP     : std_logic;
  signal Sel1xSN     : std_logic;
  signal Sel1xSP     : std_logic;
  signal InitxSN     : std_logic;
  signal InitxSP     : std_logic;
  signal EnaReg0xSN  : std_logic;
  signal EnaReg0xSP  : std_logic;
  signal EnaReg1xSN  : std_logic;
  signal EnaReg1xSP  : std_logic;

  -- counter
```

```
  signal CountxDN : unsigned(6 downto 0);
  signal CountxDP : unsigned(6 downto 0);

begin  -- rtl

  -- purpose: calculate controller signals
  -- type   : combinational
  -- inputs : SellxSP,Sel0xSP,CountxDP
  -- outputs: SellxSN,Sel0xSN,InitxSN,EnaReglxSN,EnaReg0xSN
  contsig_calc : process (CountxDP, Sel0xSP, Sel1xSP)
  begin  -- process contsig_calc
    Sel0xSN       <= Sel0xSP;
    Sel1xSN       <= Sel1xSP;
    InitxSN       <='0';
    EnaReg0xSN    <='0';
    EnaReg1xSN    <='0';
    if CountxDP = 0 or CountxDP = 11 or CountxDP = 22 or CountxDP = 33 then
      InitxSN     <='1';
    end if;
    if CountxDP = 10 or CountxDP = 21 or CountxDP = 32 or CountxDP = 43 then
      EnaReg0xSN <='1';
    end if;
    if CountxDP = 44 then
      EnaReg1xSN <='1';
      Sel1xSN     <='0';
    end if;
    if CountxDP = 10 then
      Sel0xSN     <='1';
    end if;
    if CountxDP = 21 then
      Sel1xSN     <='1';
    end if;
    if CountxDP = 32 then
      Sel0xSN     <='0';
    end if;
  end process contsig_calc;

  -- purpose: assign controller signals
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, SellxSN,Sel0xSN,InitxSN,EnaReglxSN,EnaReg0xSN
  -- outputs: SellxSP,Sel0xSP,InitxSP,EnaReglxSP,EnaReg0xSP
  contsig_assign : process (CLKxCI, RSTxRBI)
  begin  -- process contsig_assign
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      Sel0xSP     <='0';
      Sel1xSP     <='0';
      InitxSP     <='0';
      EnaReg0xSP <='0';
      EnaReg1xSP <='0';
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      Sel0xSP     <= Sel0xSN;
      Sel1xSP     <= Sel1xSN;
      InitxSP     <= InitxSN;
      EnaReg0xSP <= EnaReg0xSN;
      EnaReg1xSP <= EnaReg1xSN;
    end if;
  end process contsig_assign;

  -- connect outputs
  EnaRegxSO <= EnaReg1xSP&EnaReg0xSP;
  InitxSO    <= InitxSP;
  SelxSO     <= Sel1xSP&Sel0xSP;

  -- purpose: increment counter
  -- type   : combinational
  -- inputs : CountxDP
  -- outputs: CountxDN
  ct_increment : process (CountxDP)
  begin  -- process ct_increment
    CountxDN <= CountxDP+1;
  end process ct_increment;

  -- purpose: assign updated value to counter
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, CountxDN
  -- outputs: CountxDP
  ct_assign : process (CLKxCI, RSTxRBI)
  begin  -- process ct_assign
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      CountxDP <= (others => '0');
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      CountxDP <= CountxDN;
    end if;
  end process ct_assign;

end rtl;
```

multiplier25s25s.vhd

```
-------------------------------------------------------------------------------
-- Title      : Multiplier
-- Project    : MIDI Synthesizer
-------------------------------------------------------------------------------
-- File       : multiplier25s25s.vhd
-- Author     : Daniel Engeler
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/10/25
-- Last update: 2002/12/09
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description:
-- Serial multiplication of two signed 25 bit numbers
-- ZxDO = CxDI * YxDI
-- All numbers = 25 bit, (-2, 1, 0.5, ..), range from -2 to 2-(2^-23).
-- Make sure the result isn't outside this range.
-------------------------------------------------------------------------------
-- Usage:
-- 1. Apply CxDI and YxDI.  They must remain constant for the next 28 cycles.
-- 2. Set InitxSI for one cycle
-- 3. After 27 cycles, DonexS is set and ZxDO may be read.
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions :
-- 2002/10/25 1.0 danengel Created
-- 2002/11/05 1.1 danengel Removed input buffers, optimized area a bit
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-------------------------------------------------------------------------------
-- description of I/O signals
-------------------------------------------------------------------------------

-- ClkxCI: Clock

-- ResetxRBI: Asynchronous reset, active low

-- InitxSI: Tell the multiplier to initialize and start a new multiplication.

-- DonexSO: Indicates when multiplication is over and the result is ready.

-- CxDI: The first factor

-- YxDI: The second factor

-- ZxDO: The result
```

```
entity multiplier25s25s is
  port (
    ClkxCI     : in  std_logic;
    ResetxRBI  : in  std_logic;
    InitxSI    : in  std_logic;
    DonexSO    : out std_logic;
    CxDI       : in  unsigned(24 downto 0);
    YxDI       : in  unsigned(24 downto 0);
    ZxDO       : out unsigned(24 downto 0));
end multiplier25s25s;

architecture rtl of multiplier25s25s is
  -- registers
  signal AccuxDN, AccuxDP : unsigned(24 downto 0);
  signal YsxDN, YsxDP     : unsigned(23 downto 0);
  signal CntxDN, CntxDP   : unsigned(4 downto 0);

  -- we control the multiplexers
  signal FeedbackxS : std_logic;
  signal CYselxS    : std_logic;
  signal ModexS     : std_logic;

  -- output wrapper
  signal DonexS : std_logic;

  -- helpers
  signal Add1xD : unsigned(24 downto 0);
  signal Add2xD : unsigned(24 downto 0);
  signal T1xD   : unsigned(24 downto 0);
  signal T2xD   : unsigned(24 downto 0);
  signal T3xD   : unsigned(23 downto 0);

begin
  -- connect outputs
  DonexSO <= DonexS;
  ZxDO    <= AccuxDP;

  -- shift Ys down and multiply its LSB with CxDI
  process(CxDI, InitxSI, YsxDP, YxDI)
  begin
    -- shift Ys
    if InitxSI = '1' then
      YsxDN <= YxDI(23 downto 0);
    else
      YsxDN <= '0' & YsxDP(23 downto 1);
    end if;

    -- multiply
    if YsxDP(0) = '1' then
      T1xD <= '0' & CxDI(23 downto 0);
    else
      T1xD <= (others => '0');
    end if;
  end process;


  -- The middle two parts
  process(CYselxS, CxDI, T3xD, YxDI)
  begin
    if CYselxS = '1' then

      if YxDI(24) = '1' then
        T3xD <= CxDI(23 downto 0);
      else
        T3xD <= (others => '0');
      end if;

    else

      if CxDI(24) = '1' then
        T3xD <= YxDI(23 downto 0);
      else
        T3xD <= (others => '0');
      end if;

    end if;

    T2xD <= (0-T3xD) & '0';
  end process;


  -- Accu & Adder
  process(AccuxDP, Add1xD, Add2xD, FeedbackxS, InitxSI, ModexS, T1xD, T2xD)
  begin
    -- accumulate
    if InitxSI = '1' then
      AccuxDN <= (others => '0');
    else
      AccuxDN <= Add1xD+Add2xD;
    end if;

    -- first operand
    if ModexS = '0' then
      Add1xD <= T1xD;
    else
      Add1xD <= T2xD;
    end if;

    -- second operand
    if FeedbackxS = '1' then
      Add2xD <= AccuxDP;
    else
      Add2xD <= '0' & AccuxDP(24 downto 1);
    end if;
  end process;


  -- the counter
  process(CntxDP, InitxSI)
  begin
    if InitxSI = '1' then
      CntxDN <= (others => '0');
    else
      CntxDN <= CntxDP + 1;
    end if;
  end process;


  -- a few signals controlled by the counter
  process(CntxDP, FeedbackxS)
  begin
    -- ModexS
    if CntxDP = 29 or CntxDP = 30 then
      ModexS <= '1';
    else
      ModexS <= '0';
    end if;

    -- CYSelxS
    if CntxDP = 30 then
      CYSelxS <= '1';
    else
      CYSelxS <= '0';
    end if;

    -- FeedbackxS
    if CntxDP >= 24 and CntxDP <= 30 then
      FeedbackxS                <= '1';
    else
      FeedbackxS                <= '0';
    end if;
```

```
    -- DonexS
    if CntxDP = 31 then
      DonexS <= '1';
    else
      DonexS <= '0';
    end if;
  end process;

  -- update registers
  process(ClkxCI, ResetxRBI)
  begin
    if ResetxRBI = '0' then
      YsxDP    <= (others => '0');
      AccuxDP  <= (others => '0');
      CntxDP   <= (others => '0');
    elsif ClkxCI'event and ClkxCI = '1' then
      YsxDP    <= YsxDN;
      AccuxDP  <= AccuxDN;
      CntxDP   <= CntxDN;
    end if;
  end process;

end rtl;
```

mul15u7uc.vhd

```
-------------------------------------------------------------------------------
-- Title      : Multiplier 15u7uc
-- Project    : MIDI synthesizer
-------------------------------------------------------------------------------
-- File       : mul15u7uc.vhd
-- Author     : Samuel Nobs <nobssa@ee.ethz.ch>
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/11/13
-- Last update: 2002/11/28
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: calculates the product of a 15 bit unsigned number with
--              a 7 bit unsigned number and the bitwise inverted number,
--              must be externally controlled,
--              needs 9 clock cycles for calculation
--
--              C1xDO = AxDI*BxDI
--              C2xDO = AxDI*(127-BxDI)
-------------------------------------------------------------------------------
-- Usage:       Input needs to be constant during the calculation
--              Output is valid only during 11th cycle
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author         Description
-- 2002/11/13  1.0      Samuel Nobs    created (based on mul15u9u)
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity mul15u7uc is

  port (
    AxDI     : in  unsigned(14 downto 0);  -- the 15bit unsigned
    BxDI     : in  unsigned(6 downto 0);   -- the 9bit unsigned
    C1xDO    : out unsigned(14 downto 0);  -- the 15bit unsigned outputs
    C2xDO    : out unsigned(14 downto 0);
    InitxSI  : in  std_logic;              -- init signal
    CLKxCI   : in  std_logic;              -- clock
    RSTxRBI  : in  std_logic);             -- async reset, active low

end mul15u7uc;

architecture rtl of mul15u7uc is

  -- helper signals
  signal PR1xD   : unsigned(14 downto 0);  -- 1bit x 15bit products
  signal PR2xD   : unsigned(14 downto 0);
  signal Sum1xD  : unsigned(15 downto 0);  -- outputs of adders
  signal Sum2xD  : unsigned(15 downto 0);

  -- registers
  signal BRegxDN  : unsigned(6 downto 0);
  signal BRegxDP  : unsigned(6 downto 0);
  signal Accu1xDN : unsigned(14 downto 0);
  signal Accu1xDP : unsigned(14 downto 0);
  signal Accu2xDN : unsigned(14 downto 0);
  signal Accu2xDP : unsigned(14 downto 0);

begin  -- rtl

  -- purpose: return 0 if init is high, else perform shift-right-logical
  --          and return value
  -- type   : combinational
  -- inputs : BRegxDP, InitxSI
  -- outputs: BRegxDN
  bshiftreg_calc : process (BRegxDP, BxDI, InitxSI)
  begin  -- process bshiftreg
    if InitxSI = '1' then
      BRegxDN <= BxDI;
    else
      BRegxDN <= '0'&BRegxDP(6 downto 1);
    end if;
  end process bshiftreg_calc;

  -- purpose: store BRegxDN in BRegxDP
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, BRegxDN
  -- outputs: BRegxDP
  bshiftreg_assign : process (CLKxCI, RSTxRBI)
  begin  -- process bshiftreg_assign
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      BRegxDP <= (others => '0');
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      BRegxDP <= BRegxDN;
    end if;
  end process bshiftreg_assign;

  -- purpose: calculate AxDI times LSB(BRegxDN)
  --          and AxDI times !LSB(BRegxDN)
  -- type   : combinational
  -- inputs : BRegxDP
  -- outputs: PR1xD, PR2xD
  bit_prod : process (AxDI, BRegxDP)
  begin  -- process 1bit_prod
    if BRegxDP(0) = '1' then
      PR1xD <= AxDI;
      PR2xD <= (others => '0');
    else
      PR2xD <= AxDI;
      PR1xD <= (others => '0');
    end if;
  end process bit_prod;

  -- purpose: add partial products
  -- type   : combinational
  -- inputs : PRxD, InitxSI
  -- outputs: SumxD
  add_calc : process (Accu1xDP, Accu2xDP, InitxSI, PR1xD, PR2xD)
```

```
  begin  -- process accu_calc
    if InitxSI = '1' then
      Sum1xD <= (others => '0');
      Sum2xD <= (others => '0');
    else
      Sum1xD <= ('0'&Accu1xDP)+('0'&PR1xD);
      Sum2xD <= ('0'&Accu2xDP)+('0'&PR2xD);
    end if;
  end process add_calc;

  -- shift result of addition
  Accu1xDN <= Sum1xD(15 downto 1);
  Accu2xDN <= Sum2xD(15 downto 1);

  -- purpose: store shifted result AccuXxDN in AccuXxDP
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, Accu1xDN, Accu2xDN
  -- outputs: Accu1xDP, Accu2xDP
  accu_assign : process (CLKxCI, RSTxRBI)
  begin  -- process accu_assign
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      Accu1xDP <= (others => '0');
      Accu2xDP <= (others => '0');
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      Accu1xDP <= Accu1xDN;
      Accu2xDP <= Accu2xDN;
    end if;
  end process accu_assign;

  -- connect output
  C1xDO <= Accu1xDP;
  C2xDO <= Accu2xDP;

end rtl;
```

mul15u9u.vhd

```
-------------------------------------------------------------------------------
-- Title      : Multiplier 15u9u
-- Project    : MIDI synthesizer
-------------------------------------------------------------------------------
-- File       : mul15u9u.vhd
-- Author     : Samuel Nobs <nobssa@ee.ethz.ch>
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/11/13
-- Last update: 2002/11/28
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: calculates the product of a 15 bit unsigned number with
--              a 9 bit unsigned number, must be externally controlled,
--              needs 11 clock cycles for calculation
-------------------------------------------------------------------------------
-- Usage:       Input needs to be constant during the calculation
--              Output is valid only during 11th cycle
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author         Description
-- 2002/11/13  1.0      Samuel Nobs    created
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity mul15u9u is
  port (
    AxDI     : in  unsigned(14 downto 0);  -- the 15bit unsigned
    BxDI     : in  unsigned(8 downto 0);   -- the 9bit unsigned
    CxDO     : out unsigned(14 downto 0);  -- the 15bit unsigned output
    InitxSI  : in  std_logic;              -- init signal
    CLKxCI   : in  std_logic;              -- clock
    RSTxRBI  : in  std_logic);             -- async reset, active low

end mul15u9u;

architecture rtl of mul15u9u is

  -- helper signals
  signal PRxD  : unsigned(14 downto 0);  -- 1bit x 15bit product
  signal SumxD : unsigned(15 downto 0);  -- output of adder

  -- registers
  signal BRegxDN : unsigned(8 downto 0);
  signal BRegxDP : unsigned(8 downto 0);
  signal AccuxDN : unsigned(14 downto 0);
  signal AccuxDP : unsigned(14 downto 0);

begin  -- rtl

  -- purpose: return 0 if init is high, else perform shift-right-logical
  --          and return value
  -- type   : combinational
  -- inputs : BRegxDP, InitxSI
  -- outputs: BRegxDN
  bshiftreg_calc : process (BRegxDP, BxDI, InitxSI)
  begin  -- process bshiftreg
    if InitxSI = '1' then
      BRegxDN <= BxDI;
    else
      BRegxDN <= '0'&BRegxDP(8 downto 1);
    end if;
  end process bshiftreg_calc;

  -- purpose: store BRegxDN in BRegxDP
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, BRegxDN
  -- outputs: BRegxDP
  bshiftreg_assign : process (CLKxCI, RSTxRBI)
  begin  -- process bshiftreg_assign
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      BRegxDP <= (others => '0');
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      BRegxDP <= BRegxDN;
    end if;
  end process bshiftreg_assign;

  -- purpose: calculate AxDI times LSB(BRegxDN)
  -- type   : combinational
  -- inputs : BRegxDP
  -- outputs: PRxD
  bit_prod : process (AxDI, BRegxDP)
  begin  -- process 1bit_prod
    if BRegxDP(0) = '1' then
      PRxD <= AxDI;
    else
      PRxD <= (others => '0');
    end if;
  end process bit_prod;

  -- purpose: add partial products
  -- type   : combinational
  -- inputs : PRxD, InitxSI
  -- outputs: SumxD
  add_calc : process (AccuxDP, InitxSI, PRxD)
  begin  -- process accu_calc
    if InitxSI = '1' then
```

```
        SumxD <= (others => '0');
      else
        SumxD <= ('0'&AccuxDP)+('0'&PRxD);
      end if;
    end process add_calc;

    -- shift result of addition
    AccuxDN <= SumxD(15 downto 1);

    -- purpose: store shifted result AccuxDN in AccuxDP
    -- type   : sequential
    -- inputs : CLKxCI, RSTxRBI, AccuxDN
    -- outputs: AccuxDP
    accu_assign : process (CLKxCI, RSTxRBI)
    begin  -- process accu_assign
      if RSTxRBI = '0' then                 -- asynchronous reset (active low)
        AccuxDP <= (others => '0');
      elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
        AccuxDP <= AccuxDN;
      end if;
    end process accu_assign;

    -- connect output
    CxDO <= AccuxDP;

  end rtl;
```

oscillator.vhd

```
  -------------------------------------------------------------------------------
  -- Title : Sinus Generator
  -- Project : MIDI Synthesizer
  -------------------------------------------------------------------------------
  -- File : oscillator.vhd
  -- Author : Daniel Engeler (danengel@ee.ethz.ch)
  -- Company : Integrated Systems Laboratory, ETH Zurich
  -- Created : 2002/10/26
  -- Last update: 2002/12/11
  -- Platform : ModelSim (simulation), Synopsys (synthesis)
  -------------------------------------------------------------------------------
  -- Description
  -- Generate a nice sine wave
  -------------------------------------------------------------------------------
  -- Usage
  -- Apply Y1xD, set InitxSI for one cycle
  -- The oscillator waits for the current period to finish
  -- New output every 128 cycles
  -------------------------------------------------------------------------------
  -- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
  -------------------------------------------------------------------------------
  -- Revisions :
  -- 2002/10/26 1.0 danengel Created
  -- 2002/10/31 1.1 danengel We now calculate C ourselves from Y1
  -- 2002/11/06 1.2 danengel Use multiplier v1.1
  -------------------------------------------------------------------------------

  library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

  -------------------------------------------------------------------------------
  -- description of I/O signals
  -------------------------------------------------------------------------------

  -- ClkxCI: Clock

  -- ResetxRBI: Asynchronous reset, active low

  -- InitxSI: Set InitxSI for one cycle and at the same time apply Y1xDI
  -- and MutexDI to kindly request to change frequency.  The oscillator
  -- will wait until one period is finished and then smoothly change.

  -- MutexSI: If MutexSI is set when InitxSI is set, the oscillator's
  -- output will always be zero.  Internally it runs normally, so it
  -- will wait until an invisible period is over before changing
  -- frequency.

  -- AnewxSO: Is up for one cycle when the current period is finished
  -- and we can re-initialize the oscillator to switch to a new
  -- frequency.

  -- Y1xDI: Initial value, equals to round(2*pi*f / (4*44100) * 2^23)

  -- YxDO: Sound output, 16 bit mono, 2's complement, 44100 Hz

  entity oscillator is
    port (
      ClkxCI     : in  std_logic;
      ResetxRBI  : in  std_logic;
      InitxSI    : in  std_logic;
      MutexSI    : in  std_logic;
      AnewxSO    : out std_logic;
      Y1xDI      : in  unsigned(24 downto 0);
      YxDO       : out unsigned(15 downto 0));
  end oscillator;

  -------------------------------------------------------------------------------
  -- description of internal signals
  -------------------------------------------------------------------------------

  -- YxD holds the current Y

  -- MxD holds the old Y

  -- CxD is the constant for the algorithm, c=2-(2*pi*f*dt)^2

  -- Round1xS says its the first multiplication while we're calculating CxD

  -- CRdyxS says the first multiplication is done

  -- NextxS tells the multiplier to initialize

  -- YMupdatexS is is up when a new output is ready, that is every four
  -- multiplications, except the first one when we calculate CxD and
  -- CRdyxS is high.

  -- OutCntxD counts four multiplications so we can downsample the output.

  -- YOxD holds the last downsampled output

  -- DonexS says the multiplier is finished

  -- InitxS buffers InitxSI so we can wait until a period is finished

  -- AnewxS is up for one cycle when the current period is finished and
  -- we can re-initialize the oscillator to switch to a new frequency.

  -- LastSignxS is the sign bit of the last output YxDO

  architecture rtl of oscillator is

    -- Registers
    signal YxDN, YxDP                : unsigned(24 downto 0);
    signal MxDN, MxDP                : unsigned(24 downto 0);
    signal CxDN, CxDP                : unsigned(24 downto 0);
    signal Round1xSN, Round1xSP      : std_logic;
    signal OutCntxDN, OutCntxDP      : unsigned(1 downto 0);
    signal YOxDN, YOxDP              : unsigned(15 downto 0);
```

```
    signal InitBufxSN, InitBufxSP    : std_logic;
    signal LastSignxSN, LastSignxSP  : std_logic;
    signal Y1BufxDN, Y1BufxDP        : unsigned(24 downto 0);
    signal MutexSN, MutexSP          : std_logic;

    -- Status and Helpers
    signal AnewxS                    : std_logic;
    signal OutCntxS                  : std_logic;
    signal DonexS                    : std_logic;
    signal CRdyxS                    : std_logic;
    signal NextxS                    : std_logic;
    signal YMupdatexS                : std_logic;
    signal MT1xD                     : unsigned(24 downto 0);
    signal YT1xD                     : unsigned(24 downto 0);
    signal CT1xD                     : unsigned(24 downto 0);
    signal MulCntxS                  : std_logic;
    signal T1xD                      : unsigned(24 downto 0);
    signal T2xD                      : unsigned(24 downto 0);
    signal T3xD                      : unsigned(24 downto 0);
    signal T4xD                      : unsigned(15 downto 0);

    component multiplier25s25s
      port (
        ClkxCI     : in  std_logic;
        ResetxRBI  : in  std_logic;
        InitxSI    : in  std_logic;
        DonexSO    : out std_logic;
        CxDI       : in  unsigned(24 downto 0);
        YxDI       : in  unsigned(24 downto 0);
        ZxDO       : out unsigned(24 downto 0));
    end component;

begin
    -- MutexSI buffer
    -- MutexSI is buffered so we can mute until we change frequency again
    process (InitxSI, MutexSI, MutexSP)
    begin
      if InitxSI = '1' then
        MutexSN <= MutexSI;
      else
        MutexSN <= MutexSP;
      end if;
    end process;

    -- output
    -- which is eventually muted
    process (MutexSP, YOxDP)
    begin
      if MutexSP = '0' then
        YxDO <= YOxDP;
      else
        YxDO <= (others => '0');
      end if;
    end process;

    -- this is necessary that the following ADSR knows when we're switching
    -- frequencies, otherwise the ADSR and the oscillator are out of synch
    AnewxSO <= AnewxS;

    -- YMupdatexS
    YMupdatexS <= NextxS and not(CRdyxS);

    -- Multiplier init
    NextxS <= DonexS or AnewxS;

    -- Y1 Buffer
    process(InitxSI, Y1BufxDP, Y1xDI)
    begin
      if InitxSI = '1' then
        Y1BufxDN <= Y1xDI;
      else
        Y1BufxDN <= Y1BufxDP;
      end if;
    end process;

    -- InitxSI buffer
    process(AnewxS, InitBufxSP, InitxSI)
    begin
      if (InitxSI or AnewxS) = '1' then
        InitBufxSN <= InitxSI;
      else
        InitBufxSN <= InitBufxSP;
      end if;
    end process;

    -- end of period detector
    process(InitBufxSP, LastSignxSP, T4xD)
    begin
      LastSignxSN <= T4xD(15);
      AnewxS      <= (not T4xD(15)) and LastSignxSP and InitBufxSP;
    end process;

    -- output counter
    process(NextxS, OutCntxDP)
    begin
      if NextxS = '1' then
        OutCntxDN <= OutCntxDP + 1;
      else
        OutCntxDN <= OutCntxDP;
      end if;

      if OutCntxDP = 0 then
        OutCntxS <= '1';
      else
        OutCntxS <= '0';
      end if;
    end process;

    -- output buffer
    process(AnewxS, OutCntxS, T4xD, Y1BufxDP, YOxDP, YxDP)
    begin
      if OutCntxS = '1' then
        T4xD <= YxDP(23 downto 8);
      else
        T4xD <= YOxDP;
      end if;

      if AnewxS = '1' then
        YOxDN <= Y1BufxDP(23 downto 8);
      else
        YOxDN <= T4xD;
      end if;
    end process;

    -- Round1
    process(AnewxS, NextxS, Round1xSP)
    begin
      if NextxS = '0' then
        Round1xSN <= Round1xSP;
      else
        Round1xSN <= AnewxS;
      end if;
    end process;

    -- C
    process(AnewxS, CRdyxS, CxDP, NextxS, Round1xSP, T1xD, T2xD, Y1BufxDP)
    begin
      CRdyxS <= NextxS and Round1xSP;

      if CRdyxS = '1' then
        -- Actually, c=2-(w*dt)^2.  Here we calculate -c because we save
        -- an adder.  The oversampled wave changes its sign with each
```

```
      -- iteration, but as the output is fourfold downsampled, nobody
      -- can see that.
      CxDN <= '1' & T1xD(23 downto 0);
    else
      CxDN <= T2xD;
    end if;

    if AnewxS = '1' then
      T2xD <= Y1BufxDP;
    else
      T2xD <= CxDP;
    end if;
  end process;

  -- M
  process(AnewxS, MT1xD, MxDP, YMupdatexS, YxDP)
  begin
    if YMupdatexS = '1' then
      MT1xD <= YxDP;
    else
      MT1xD <= MxDP;
    end if;

    if AnewxS = '1' then
      MxDN <= (others => '0');
    else
      MxDN <= MT1xD;
    end if;
  end process;

  -- Y
  process(AnewxS, CRdyxS, MxDP, T1xD, T3xD, Y1BufxDP, YMupdatexS, YT1xD, YxDP)
  begin
    if CRdyxS = '0' then
      T3xD <= YxDP;
    else
      T3xD <= YxDP - ("000" & YxDP(24 downto 3));
    end if;

    if YMupdatexS = '1' then
      YT1xD <= T1xD(24 downto 0) - MxDP;
    else
      YT1xD <= T3xD;
    end if;

    if AnewxS = '1' then
      YxDN <= Y1BufxDP;
    else
      YxDN <= YT1xD;
    end if;
  end process;

  -- instantiate multiplier
  u_multiplier : multiplier25s25s
    port map (
      ClkxCI     => ClkxCI,
      ResetxRBI  => ResetxRBI,
      InitxSI    => NextxS,
      DonexSO    => DonexS,
      CxDI       => T2xD,
      YxDI       => YxDN,
      ZxDO       => T1xD);

  -- update registers
  process(ClkxCI, ResetxRBI)
  begin
    if ResetxRBI = '0' then
      -- initial values to run at 7000 Hz
      YOxDP        <= to_unsigned(8170, 16);
      YxDP         <= to_unsigned(2091555, 25);
      MxDP         <= to_unsigned(0, 25);
      CxDP         <= to_unsigned(17458350, 25);
      OutCntxDP    <= (others => '0');
      InitBufxSP   <= '0';
      LastSignxSP  <= '0';
      Round1xSP    <= '0';
      Y1BufxDP     <= (others => '0');
      MutexSP      <= '0';
    elsif ClkxCI'event and ClkxCI = '1' then
      YOxDP        <= YOxDN;
      OutCntxDP    <= OutCntxDN;
      YxDP         <= YxDN;
      MxDP         <= MxDN;
      CxDP         <= CxDN;
      InitBufxSP   <= InitBufxSN;
      LastSignxSP  <= LastSignxSN;
      Round1xSP    <= Round1xSN;
      Y1BufxDP     <= Y1BufxDN;
      MutexSP      <= MutexSN;
    end if;
  end process;

end rtl;
```

Panorama.vhd

```
-------------------------------------------------------------------------------
-- Title      : Panorama
-- Project    :
-------------------------------------------------------------------------------
-- File       : Panorama.vhd
-- Author     : Samuel Nobs <nobssa@ee.ethz.ch>
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/11/27
-- Last update: 2002/12/02
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: converts the mono input into a stereo output
--              using a panning factor
--
--              ROutSxDO=PanxDI*InSxDI
--              LOutSxDO=InSxDI-ROutSxDO
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author        Description
-- 2002/11/27  1.0      Samuel Nobs   created
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Panorama is

  port (
    PanxDI    : in  unsigned(6 downto 0);    -- panning factor
    InSxDI    : in  unsigned(15 downto 0);   -- mono input
    LOutSxDO  : out unsigned(15 downto 0);   -- left output
    ROutSxDO  : out unsigned(15 downto 0);   -- right output
    CLKxCI    : in  std_logic;               -- clock
    RSTxRBI   : in  std_logic);              -- async reset, active low

end Panorama;

architecture rtl of Panorama is

  -- registers
```

```
  signal LOutRegxDN : unsigned(15 downto 0);
  signal LOutRegxDP : unsigned(15 downto 0);
  signal ROutRegxDN : unsigned(15 downto 0);
  signal ROutRegxDP : unsigned(15 downto 0);
  signal InRegxDN   : unsigned(15 downto 0);
  signal InRegxDP   : unsigned(15 downto 0);

  -- counter
  signal CountxDN : unsigned(6 downto 0);
  signal CountxDP : unsigned(6 downto 0);

  -- controller signals
  signal InitxS       : std_logic;        -- initialize multiplier
                                          -- and enable input register
  signal OutRegEnaxS : std_logic;         -- enable output registers

  -- helper signals
  signal AxD   : unsigned(14 downto 0);  -- input 1 of multiplier
  signal C1xD  : unsigned(14 downto 0);  -- multiplier outputs
  signal C2xD  : unsigned(14 downto 0);
  signal NegxD : unsigned(15 downto 0);  -- negation of InRegxDN

  -- the multiplier we need
  component mul15u7uc
    port (
      AxDI   : in  unsigned(14 downto 0);
      BxDI   : in  unsigned(6 downto 0);
      C1xDO  : out unsigned(14 downto 0);
      C2xDO  : out unsigned(14 downto 0);
      InitxSI : in  std_logic;
      CLKxCI  : in  std_logic;
      RSTxRBI : in  std_logic);
  end component;

begin  -- rtl

-- controller counter
  CountxDN     <= CountxDP+1;
  InitxS       <='1' when CountxDP = 1 else '0';
  OutRegEnaxS  <='1' when CountxDP = 9 else '0';
  -- purpose: increment the counter
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, CountxDN
  -- outputs: CountxDP
  cont_update : process (CLKxCI, RSTxRBI)
  begin  -- process cont_update
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      CountxDP <= to_unsigned(0, 7);
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      CountxDP <= CountxDN;
    end if;
  end process cont_update;

-- purpose: calculate next state of input register
-- type : combinational
-- inputs : InitxS, YSxDI, InRegxDP
-- outputs: InRegxDN
  inreg_calc : process (InRegxDP, InSxDI, InitxS)
  begin  -- process inreg_calc
    if InitxS = '1' then
      InRegxDN <= InSxDI;
    else
      InRegxDN <= InRegxDP;
    end if;
  end process inreg_calc;

-- purpose: update input register
-- type : sequential
-- inputs : CLKxCI, RSTxRBI, InRegxDN
-- outputs: InRegxDP
  inreg_update : process (CLKxCI, RSTxRBI)
  begin  -- process inreg_update
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      InRegxDP <= (others => '0');
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      InRegxDP <= InRegxDN;
    end if;
  end process inreg_update;

  -- assign first input of multiplier
  AxD <= InRegxDP(14 downto 0) when InRegxDP(15) = '0' else NegxD(14 downto 0);


  -- instantiate multiplier
  multiplier : mul15u7uc
    port map (
      AxDI   => AxD,
      BxDI   => PanxDI,
      C1xDO  => C1xD,
      C2xDO  => C2xD,
      InitxSI => InitxS,
      CLKxCI  => CLKxCI,
      RSTxRBI => RSTxRBI);

  -- negate input InRegxDP
  NegxD <= (0 - InRegxDP);

  -- purpose: calculate output register, negate values if necessary
  -- type   : combinational
  -- inputs : CxD, InRegxDP
  -- outputs: LOutRegxDP, LOutRegxDN
  outreg_calc : process (C1xD, C2xD, InRegxDP, LOutRegxDP, OutRegEnaxS,
                         ROutRegxDP)
  begin  -- process outreg_calc
    if OutRegEnaxS = '1' then
      if InRegxDP(15) = '1' then
        ROutRegxDN <= (0 - ('0'&C1xD));
        LOutRegxDN <= (0 - ('0'&C2xD));
      else
        ROutRegxDN <= '0'&C1xD;
        LOutRegxDN <= '0'&C2xD;
      end if;
    else
      ROutRegxDN  <= ROutRegxDP;
      LOutRegxDN  <= LOutRegxDP;
    end if;
  end process outreg_calc;

  -- purpose: update output registers
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, ROutRegxDN, LOutRegxDN
  -- outputs: ROutRegxDP, LOutRegxDP
  outreg_update : process (CLKxCI, RSTxRBI)
  begin  -- process outreg_update
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      ROutRegxDP <= (others => '0');
      LOutRegxDP <= (others => '0');
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      ROutRegxDP <= ROutRegxDN;
      LOutRegxDP <= LOutRegxDN;
    end if;
  end process outreg_update;

  -- connect outputs
  ROutSxDO <= ROutRegxDP;
  LOutSxDO <= LOutRegxDP;

end rtl;
```

rectifier.vhd

```
-------------------------------------------------------------------------------
-- Title      : Rectifier
-- Project    : MIDI Synthesizer
-------------------------------------------------------------------------------
-- File       : rectifier.vhd
-- Author     : Samuel Nobs <nobssa@ee.ethz.ch>
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/11/11
-- Last update: 2002/12/05
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: if enabled, this block converts the sine wave at its
--              input to a rectangle wave
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author          Description
-- 2002/11/11  1.0      Samuel Nobs     Created
-- 2002/11/11           Samuel Nobs     verified using selected testvectors
-- 2002/12/05           Samuel Nobs     corrected to [-32767,32767]
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity rectifier is

  port (
    ENAxSI   : in  std_logic;                -- enables the rectifier
    InSxDI   : in  unsigned(15 downto 0);    -- the input sine wave
    OutSxDO : out unsigned(15 downto 0));   -- the output wave

end rectifier;

architecture rtl of rectifier is

begin  -- rtl

  OutSxDO(15) <= InSxDI(15);
  OutSxDO(14) <= ((not ENAxSI) and InSxDI(14)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(13) <= ((not ENAxSI) and InSxDI(13)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(12) <= ((not ENAxSI) and InSxDI(12)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(11) <= ((not ENAxSI) and InSxDI(11)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(10) <= ((not ENAxSI) and InSxDI(10)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(9)  <= ((not ENAxSI) and InSxDI(9)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(8)  <= ((not ENAxSI) and InSxDI(8)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(7)  <= ((not ENAxSI) and InSxDI(7)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(6)  <= ((not ENAxSI) and InSxDI(6)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(5)  <= ((not ENAxSI) and InSxDI(5)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(4)  <= ((not ENAxSI) and InSxDI(4)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(3)  <= ((not ENAxSI) and InSxDI(3)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(2)  <= ((not ENAxSI) and InSxDI(2)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(1)  <= ((not ENAxSI) and InSxDI(1)) or (ENAxSI and (not InSxDI(15)));
  OutSxDO(0)  <= ((not ENAxSI) and InSxDI(0)) or (ENAxSI);  -- we want 32767
                                                            -- and    -32767

end rtl;
```

reg7b.vhd

```
-------------------------------------------------------------------------------
-- Title      : 7 bit register
-- Project    :
-------------------------------------------------------------------------------
-- File       : reg7b.vhd
-- Author     : Samuel Nobs <nobssa@ee.ethz.ch>
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/11/07
-- Last update: 2002/11/28
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: this 7 bit register is enabled only if the ENAxSI
--              is 1 and SELxDI matches the generic value NUM
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author          Description
-- 2002/11/07  1.0      Samuel Nobs     Created
-- 2002/11/07           Samuel Nobs     verified using selected testvectors
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity reg7b is

  generic (
    NUM     : integer := 0;
    DEFAULT : integer := 0);

  port (
    ENAxSI  : in  std_logic;
    SELxDI  : in  unsigned(6 downto 0);
    REGxDI  : in  unsigned(6 downto 0);
    REGxDO  : out unsigned(6 downto 0);
    CLKxCI  : in  std_logic;
    RSTxRBI : in  std_logic);
end reg7b;

architecture rtl of reg7b is
  signal REGxDN : unsigned(6 downto 0);  -- next value
  signal REGxDP : unsigned(6 downto 0);  -- previous value
begin  -- rtl

  -- purpose: accept value if this register is enabled and selected
  -- type   : combinational
  -- inputs : REGxDP, REGxDI, ENAxSI, SELxDI
  -- outputs: REGxDN
  calculate : process (ENAxSI, REGxDI, REGxDP, SELxDI)
  begin  -- process calculate
    if ENAxSI = '1' and SELxDI = to_unsigned(NUM, 7) then
      REGxDN <= REGxDI;
    else
      REGxDN <= REGxDP;
    end if;
  end process calculate;

  -- purpose:
  assign : process (CLKxCI, RSTxRBI)
  begin  -- process assign
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      REGxDP <= to_unsigned(DEFAULT, 7);
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      REGxDP <= REGxDN;
    end if;
  end process assign;

  REGxDO <= REGxDP;

end rtl;
```

SIPOreg.vhd

```
-------------------------------------------------------------------------------
-- Title      : serial in, parallel out register
-- Project    : MIDI Synthesizer
-------------------------------------------------------------------------------
-- File       : SIPOreg.vhd
-- Author     : sem02w5 stud account  <sem02w5@badile7.ee.ethz.ch>
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/10/31
-- Last update: 2002/11/28
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: reads serial data from input to make it available as
--              parallel output data
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author  Description
-- 2002/10/31  1.0      sem02w5 Created
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity SIPOreg is
  port (
    SERxDI   : in  std_logic;                      -- serial input
    PARxDO   : out std_logic_vector(7 downto 0);   -- parallel output
    ENAxSI   : in  std_logic;                      -- enable signal
    CLKxCI   : in  std_logic;                      -- clock
    RSTxRBI  : in  std_logic                       -- async reset, active low
    );
end SIPOreg;

architecture rtl of SIPOreg is
  signal PARNxD : std_logic_vector(7 downto 0) := "00000000";  -- next state
  signal PARPxD : std_logic_vector(7 downto 0) := "00000000";  -- present state
begin  -- rtl

-- purpose: push next state with SERxDI
-- type : combinational
-- inputs : ENAxSI, SERxDI, PARNxD
-- outputs: PARNxD
  shiftreg : process (ENAxSI, SERxDI, PARPxD)
  begin  -- process shiftreg
    PARNxD       <=PARPxD;
    if ENAxSI = '1' then
      PARNxD(7) <= SERxDI;
      PARNxD(6) <= PARPxD(7);
      PARNxD(5) <= PARPxD(6);
      PARNxD(4) <= PARPxD(5);
      PARNxD(3) <= PARPxD(4);
      PARNxD(2) <= PARPxD(3);
      PARNxD(1) <= PARPxD(2);
      PARNxD(0) <= PARPxD(1);
    end if;
  end process shiftreg;

  -- purpose: turns next state into actual state
  -- type   : sequential
  -- inputs : CLKxCI, RSTxRBI, PARNxD
  -- outputs: PARPxDO
  assignreg : process (CLKxCI, RSTxRBI)
  begin  -- process assignreg
    if RSTxRBI = '0' then                 -- asynchronous reset (active low)
      PARPxD <= "00000000";
    elsif CLKxCI'event and CLKxCI = '1' then  -- rising clock edge
      PARPxD <= PARNxD;
    end if;
  end process assignreg;

  PARxDO <= PARPxD;
end rtl;
```

testbench.vhd

```
-------------------------------------------------------------------------------
-- Title      : Testbench
-- Project    :
-------------------------------------------------------------------------------
-- File       : testbench.vhd
-- Author     : sem02w5 stud account  <sem02w5@badile3.ee.ethz.ch>
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/11/18
-- Last update: 2003/01/31
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: Read a Standard MIDI File (SMF) and feed it to the
-- synthesizer
-------------------------------------------------------------------------------
-- MIDI-specific information taken from
-- http://jedi.ks.uiuc.edu/~johns/links/music/midifile.htm
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author      Description
-- 2002/11/18  1.0      danengel    Created
-- 2003/01/10           Samuel Nobs top with bypass inserted
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- DON'T BEAUTIFY!!  ATE FILE HEADER STRINGS GET MESSED UP
-------------------------------------------------------------------------------

library STD;
use STD.textio.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_textio.all;
use WORK.simulstuff.all;
use WORK.UTILITY.all;                   -- hex/binary conversion of integers

entity testbench is
  generic (
    VectorGen  : boolean := false;      -- generate vectors for ATE
    Purpose    : string  := "useful";   -- the purpose of the vector file
    VectorFile : string  := "out.vec";  -- name of the vector file
    ScanTest   : boolean := false;      -- check Scan-chain
    Bypass     : boolean := false;      -- bypass AsyncReceiver to speed up
    DeltaTimeFactor : real := 1.0);     -- scale MIDI Delta Times
end testbench;

-------------------------------------------------------------------------------

architecture behavioural of testbench is
  component chip
    port (
      CLKxCI          : in  std_logic;
      CLKSelxSI       : in  std_logic;
      RSTxRBI         : in  std_logic;
      ScanEnxTI       : in  std_logic;
      ScanInxTI       : in  std_logic;
      ScanOutxTO      : out std_logic;
```

```
      MIDIparxTI      : in  std_logic_vector(7 downto 0);
      DataRdyxTI      : in  std_logic;
      BypassARxTI     : in  std_logic;
      MIDIserxDI      : in  std_logic;
      MIDIChanxDI     : in  unsigned(3 downto 0);
      AudioSerSxDO    : out std_logic;
      AudioSerLLExSO  : out std_logic;
      AudioSerRLExSO  : out std_logic;
      AudioSerLRxCO   : out std_logic;
      AudioSerBitxCO  : out std_logic);
  end component;

  -- constants
  -- 1/(44100*128) s, should be 177.154195011338 ns
  constant PERIOD             : time                  := 177 ns;
  constant RESETTIME          : time                  := 200 ns;
  constant WATCHTHECLOCKTIME  : time                  := 10 ns;
  constant MIDIbitPERIOD      : time                  := 32 us;  -- 1/31250 s
  -- do some stuff periodically after so many cycles passed, like
  -- closing/reopening log files to make them accessible
  constant LOGCYCLES          : natural                   := 10000;
  -- ASCII-values for the strings "MThd" and "MTrk"
  constant MThd               : unsigned(31 downto 0)
    := "01001101010101000011010000110010 0";
  constant MTrk               : unsigned(31 downto 0)
    := "01001101010101000011100100110101 1";
  constant SMF_NAME           : string                := "input.mid";
  constant AUDIO_FILE_NAME    : string                := "audio.dat";
  constant TIME_FILE_NAME     : string                := "time.log";
  constant REGISTERS          : natural := 3575;

  signal MIDIserxD      : std_logic;
  signal MIDIChanxD     : unsigned(3 downto 0);
  signal AudioSerSxD    : std_logic;
  signal AudioSerLLExS  : std_logic;
  signal AudioSerRLExS  : std_logic;
  signal AudioSerLRxC   : std_logic;
  signal AudioSerBitxC  : std_logic;
  signal ScanEnxT       : std_logic;
  signal ScanInxT       : std_logic;
  signal ScanOutxT      : std_logic;
  signal MIDIparxT      : std_logic_vector(7 downto 0);
  signal DataRdyxT      : std_logic;
  signal BypassARxT     : std_logic;
  signal CLKxC          : std_logic;
  signal RSTxRB         : std_logic;

  signal DummyLo : std_logic := '0';
  signal DummyHi : std_logic := '1';

  -- traffic light controlling the simulation: orange -> green -> red
  signal SIMPROGRESSxS        : ResolveTrafficLight trafficlight := orange;

  -- miscellaneous
  signal ClockCountxS         : integer                          := 0;
  signal AudioSamplesCountxS  : integer                          := 0;
  signal WatchTheClockxS      : std_logic;
  signal GlobalTimexD         : time                             := RESETTIME;
  signal ScanCheck    : std_logic_vector(2*REGISTERS-1 downto 0);
  signal ScanClockCountxS : integer := 0;
  signal final_cycle : natural;

  -- signals controlling tempo (default 120 BPM)
  signal MIDI_ppqn    : natural := 128;     -- parts per quarter note
  signal MIDI_tempo   : natural := 499968;  -- us per quarter note
  signal timer_tempo  : natural := 3906;    -- us per part

  -- the toggling semaphores to synchronize the file parser and the sender
  signal ToggleToSendxS    : std_logic := '0';  -- toggled by the file parser only
  signal ToggleWhenSentxS  : std_logic := '0';  -- toggled by the sender only
  signal SendMexD          : unsigned(7 downto 0);  -- the byte to be sent

  -- simulation files
  type character_file is file of character;
  file SMF       : character_file;       -- Standard MIDI File
  file AudioFile : text;                 -- parallel audio out at 44100 Hz
  file TimeFile  : text;                 -- tell the time at regular intervals

  -- test vector file
  file TestFile : text;                  -- the output file for the testvectors

  ------------------------------------------------------------------------
  -- some functions
  ------------------------------------------------------------------------

  -- convert a byte to hex string
  function Byte2HexString(b : unsigned(7 downto 0)) return string is
  begin
    return int2str(to_integer(b(7 downto 4)), hex) &
      int2str(to_integer(b(3 downto 0)), hex);
  end Byte2HexString;

  -- convert a byte to bit string
  function Byte2BinaryString(b : unsigned(7 downto 0)) return string is
    variable s1                 : string(1 to 9);
    variable s2                 : string(1 to 8);
  begin
    -- what a dirty trick to get a nice 8 character wide string
    s1 := int2str(256+to_integer(b), bin);
    s2 := s1(2 to 9);
    return s2;
  end Byte2BinaryString;

  -- read a byte (8 bit) from a character_file
  function ReadByte(file f : character_file) return unsigned is
    variable c              : character;
    variable b              : unsigned(7 downto 0);
  begin
    read(f, c);
    b := to_unsigned(integer'val(character'pos(c)), 8);
    return b;
  end ReadByte;

  -- read a halfword (16 bit) from a character_file
  function ReadHalf(file f : character_file) return unsigned is
  begin
    return ReadByte(f) & ReadByte(f);
  end ReadHalf;

  -- read a word (32 bit) from a character_file
  function ReadWord(file f : character_file) return unsigned is
  begin
    return ReadHalf(f) & ReadHalf(f);
  end ReadWord;

  -- read a variable length quantity from a character_file at most
  -- four bytes each of which the lower 7 bits are concatenated until
  -- the byte with MSB=1
  function ReadVLQ(file f : character_file) return unsigned is
    variable accu           : unsigned(31 downto 0) := (others => '0');
    variable thebyte        : unsigned(7 downto 0);
    variable done           : boolean               := false;
    variable count          : natural               := 0;
    variable hexstring      : string(1 to 9)        := (others => 'X');

  begin
    while not done loop
      thebyte               := ReadByte(f);
      hexstring(count*2+1) := hex2char(to_integer(thebyte(7 downto 4)));
      hexstring(count*2+2) := hex2char(to_integer(thebyte(3 downto 0)));
      count                 := count + 1;
```

```
      if count > 4 then
        report "Variable Length Quantity exceeded four bytes"
          severity failure;
      end if;
      if thebyte(7) = '0' then
        done := true;
      end if;
      accu    := accu(24 downto 0) & thebyte(6 downto 0);
    end loop;
    report "Reading VLQ " & integer'image(to_integer(accu))
      & " ($" & hexstring(1 to 2*count) & ")"
      severity note;
    return accu;
  end ReadVLQ;

  -- ignore the following n bytes from a character_file
  procedure Ignore(file f : character_file; n : natural) is
    variable thebyte      : unsigned(7 downto 0);
  begin
    for counter in 1 to n loop
      thebyte := ReadByte(f);
      if endfile(f) then
        report "End of file reached while ignoring " & natural'image(n) & " bytes"
          severity failure;
      end if;
    end loop;
  end Ignore;

begin
  ------------------------------------------------------------------------
  -- global control
  ------------------------------------------------------------------------
  process
    variable status                 : file_open_status;
  begin
    -- report generics
    if Bypass = true then
      report "Generic Bypass is set .....: Bypassing Asynchronous Receiver"
        severity note;
    end if;
    if VectorGen = true then
      report "Generic VectorGen is set .: Generating test vectors"
        severity note;
    end if;

    -- handle generics
    if Bypass = true then
      BypassARxT <= '1';
    else
      BypassARxT <= '0';
    end if;

    -- open MIDI file
    file_open(status, SMF, SMF_NAME, read_mode);
    assert status = open_ok
      report FileOpenMessage(SMF_NAME, status)
      severity failure;

    -- open audio file
    file_open(status, AudioFile, AUDIO_FILE_NAME, write_mode);
    assert status = open_ok
      report FileOpenMessage(AUDIO_FILE_NAME, status)
      severity warning;

    -- reset
    RSTxRB <= '0';
    wait for RESETTIME;
    RSTxRB <= '1';
    wait for RESETTIME;

    SIMPROGRESSxS <= green;
    wait until SIMPROGRESSxS = red;

    report "Simulation run completed after " & integer'image(ClockCountxS) &
      " clock cycles (non-scanned), " & integer'image(ScanClockCountxS) &
      " clock cyles (scanned) and " & integer'image(AudioSamplesCountxS) &
      " audio samples."
      severity note;

    -- done
    file_close(SMF);
    file_close(AudioFile);
    if VectorGen = true then
      file_close(TestFile);
    end if;
    report "Global control done." severity note;
    wait;
  end process;

  ------------------------------------------------------------------------
  -- clock generation
  ------------------------------------------------------------------------
  process
    variable counter : natural;
  begin
    CLKxC <= '0';
    wait until SIMPROGRESSxS = green;

    while SIMPROGRESSxS = green loop
------------------------------------------------------------------------
      -- Scan test;  for normal clock generation see below

      -- Scan Test: Read out whole scan chain into the upper half of a
      -- huge vector, while at the same time re-inserting it into
      -- ScanInxT.  Do this again into the lower half.  If the two
      -- halfes are equal, the scan chain should work.
      if ScanTest=true and ClockCountxS = 10000 and SIMPROGRESSxS = green then
        report "Starting Scan-Test at " & integer'image(ClockCountxS) &
          " clock cycles" severity note;
        ScanEnxT <= '1';

        for counter in 2*REGISTERS-1 downto 0 loop
          -- tell coarsely in which half we are
          if ScanClockCountxS = 0 then
            report "Beginning pass one of two" severity note;
          end if;
          if ScanClockCountxS = REGISTERS then
            report "Beginning pass two of two" severity note;
          end if;

          -- feed back ScanOut to ScanIn
          ScanInxT <= ScanOutxT;

          -- store ScanOut in a huge vector containing snapshots of
          -- all registers
          ScanCheck(counter) <= ScanOutxT;
          if (ScanClockCountxS/1000)*1000 = ScanClockCountxS then
            report "Completed scanning " & integer'image(ScanClockCountxS)
              & " of 2x " & integer'image(REGISTERS) & " registers";
          end if;

          -- don't forget to clock, but count ScanClockCountxS instead
          -- of ClockCountxS to stop the other processes while
          -- scanning
          wait for WATCHTHECLOCKTIME;
          CLKxC <= '1';
          ScanClockCountxS <= ScanClockCountxS + 1;
          wait for PERIOD/2;
          CLKxC <= '0';
          wait for PERIOD/2 - WATCHTHECLOCKTIME;
        end loop;  -- counter
```

```
        -- compare
        -- we never know if an even or odd number of inverting scan
        -- flip-flops is used.
        if ( ScanCheck(2*REGISTERS-1 downto REGISTERS)
             /= ScanCheck(REGISTERS-1 downto 0)) and
           (ScanCheck(2*REGISTERS-1 downto REGISTERS) /=
            not ScanCheck(REGISTERS-1 downto 0)) then
          report "Scan_test_failed" severity failure;
        end if;

        ScanEnxT <= '0';
        report "Succesfully_completed_Scan-Test_which_took_" &
          integer'image(ScanClockCountxS) &
          "_clock_cycles,_continuing_normal_operation"
          severity note;
      end if;

      -- don't scan during normal operation
      ScanEnxT <= '0';
      ScanInxT <= '1';                    -- don't care

------------------------------------------------------------------------
      -- normal clock generation

      -- WatchTheClockxS is used as a guard signal for the MIDI file
      -- parser to prevent timing violations at the asynchronous input
      -- MIDIserxDI
      WatchTheClockxS <= '1';
      wait for WATCHTHECLOCKTIME;
      CLKxC           <='1';
      ClockCountxS    <= ClockCountxS + 1;
      wait for WATCHTHECLOCKTIME;
      WatchTheClockxS <= '0';
      wait for PERIOD/2 - WATCHTHECLOCKTIME;
      CLKxC           <='0';
      wait for PERIOD/2 - WATCHTHECLOCKTIME;

      GlobalTimexD <= GlobalTimexD + PERIOD;
    end loop;
    report "Clock_Generation_done" severity note;
    wait;
  end process;

  ------------------------------------------------------------------------
  -- automatic tempo calculation
  ------------------------------------------------------------------------
  -- The tempo in a SMF is set by two values: PPQN (parts per quarter
  -- note) and Tempo (microseconds per quarter note).  Timing
  -- information in a SMF is given in parts, so we need to know how
  -- long that is.
  process
  begin
    wait until SIMPROGRESSxS = green;

    while SIMPROGRESSxS = green loop
      wait on MIDI_ppqn, MIDI_tempo, timer_tempo, SIMPROGRESSxS;

      if MIDI_ppqn /= 0 then
        timer_tempo <= MIDI_tempo / MIDI_ppqn;
      end if;

      if timer_tempo'event then
        report "Automatic_tempo_calculation_resulted_to_" &
          integer'image(timer_tempo) & "_microseconds_per_part"
          severity note;
      end if;

    end loop;
    report "Automatic_tempo_calculation_done." severity note;
    wait;
  end process;

------------------------------------------------------------------------
-- tell the time
------------------------------------------------------------------------
process
  variable theline : line;
  variable status  : file_open_status;
  variable cycles  : natural;
begin
  wait until SIMPROGRESSxS = green;

  while SIMPROGRESSxS = green loop

    cycles := ClockCountxS + LOGCYCLES;
    while ClockCountxS < cycles and SIMPROGRESSxS=green loop
      wait for PERIOD;
    end loop;

      file_open(status, TimeFile, TIME_FILE_NAME, write_mode);
    assert status = open_ok
      report FileOpenMessage(TIME_FILE_NAME, status) severity warning;

    write(theline, time'image(GlobalTimexD));
    writeline(TimeFile, theline);

    file_close(TimeFile);

    report "Time_is_" & time'image(GlobalTimexD) &
      "_after_" & integer'image(ClockCountxS + ScanClockCountxS) &
      "_clock_cycles" severity note;
  end loop;
  report "Time_teller_done." severity note;
  wait;
end process;

------------------------------------------------------------------------
-- log audio output
------------------------------------------------------------------------
process
  variable theline  : line;
  variable status   : file_open_status;
  variable lastcycles : natural := 0;
  variable bitCount : natural;
  variable audioPar : unsigned(15 downto 0);

begin
  wait until SIMPROGRESSxS = green;

  while SIMPROGRESSxS = green loop

    -- behave like an I2S-DAC:  read the serial output and assemble it to
    -- chunks of 16 bits, left/right channel after another.
    -- try to stop while scanning

    -- wait for new channel
    if SIMPROGRESSxS=green then
      wait until AudioSerLRxC'event;
    end if;
    if ScanEnxT = '1' then
      wait until ScanEnxT = '0';
    end if;

    -- skip first bit as defined by the I2S-standard
    if SIMPROGRESSxS=green then
      wait until AudioSerBitxC'event and AudioSerBitxC = '1';
    end if;
    if ScanEnxT = '1' then
      wait until ScanEnxT = '0';
    end if;

    -- read 16 bits
```

```
    for bitCount in 15 downto 0 loop
      if SIMPROGRESSxS=green then
        wait until AudioSerBitxC'event and AudioSerBitxC = '1';
      end if;
      if ScanEnxT = '1' then
        wait until ScanEnxT = '0';
      end if;
      audioPar(bitCount) := AudioSerSxD;
    end loop;  -- bitCount

    -- write out
    -- each channel as a separate column
    write(theline, to_integer(signed(audioPar)));
    if AudioSerLRxC = '1' then
      write(theline, ' ');
    else
      writeline(AudioFile, theline);
      AudioSamplesCountxS <= AudioSamplesCountxS + 1;
    end if;

    -- close and reopen the audio file periodically
    -- so we can access it during the lengthy simulation
    if ClockCountxS >= lastcycles + LOGCYCLES then
      lastcycles := ClockCountxS;

      file_close(AudioFile);
      file_open(status, AudioFile, AUDIO_FILE_NAME, append_mode);
      assert status = open_ok
        report FileOpenMessage(AUDIO_FILE_NAME, status)
        severity warning;
    end if;

  end loop;
  report "Audio logger done." severity note;
  wait;
end process;

-------------------------------------------------------------------------------
-- write test vector file for ATE
-------------------------------------------------------------------------------
process
  variable status  : file_open_status;
  variable theline : line;
begin
  if VectorGen = true then
    -- open vector file
    file_open(status, TestFile, VectorFile, write_mode);
    assert status = open_ok
      report FileOpenMessage(VectorFile, status)
      severity warning;

    -- print header for the test file
    write(theline, string'("# authors   : S. Nobs, D. Engeler"));
    writeline(TestFile, theline);
    write(theline, string'("# file      : "));
    write(theline, VectorFile);
    writeline(TestFile, theline);
    write(theline, string'("# design    : MIDI Synthesizer"));
    writeline(TestFile, theline);
    write(theline, string'("# purpose   : "));
    write(theline, Purpose);
    writeline(TestFile, theline);
    write(theline, string'("# ================================="));
    writeline(TestFile, theline);
    write(theline, string'("#"));
    writeline(TestFile, theline);
    write(theline, string'("# INPUT pins"));
    writeline(TestFile, theline);
    write(theline, string'("INP"));
    writeline(TestFile, theline);
    write(theline, string'("MIDIserxDI,"));
    writeline(TestFile, theline);
    write(theline, string'("MIDIChanxDI0,"));
    writeline(TestFile, theline);
    write(theline, string'("MIDIChanxDI1,"));
    writeline(TestFile, theline);
    write(theline, string'("MIDIChanxDI2,"));
    writeline(TestFile, theline);
    write(theline, string'("MIDIChanxDI3,"));
    writeline(TestFile, theline);
    write(theline, string'("RSTxRBI,"));
    writeline(TestFile, theline);
    write(theline, string'("CLKxCI,"));
    writeline(TestFile, theline);
    write(theline, string'("CLKSelxSI,"));
    writeline(TestFile, theline);
    write(theline, string'("ScanEnxTI,"));
    writeline(TestFile, theline);
    write(theline, string'("ScanInxTI,"));
    writeline(TestFile, theline);
    write(theline, string'("BypassARxTI,"));
    writeline(TestFile, theline);
    write(theline, string'("DataRdyxTI,"));
    writeline(TestFile, theline);
    write(theline, string'("MIDIparxTI0,"));
    writeline(TestFile, theline);
    write(theline, string'("MIDIparxTI1,"));
    writeline(TestFile, theline);
    write(theline, string'("MIDIparxTI2,"));
    writeline(TestFile, theline);
    write(theline, string'("MIDIparxTI3,"));
    writeline(TestFile, theline);
    write(theline, string'("MIDIparxTI4,"));
    writeline(TestFile, theline);
    write(theline, string'("MIDIparxTI5,"));
    writeline(TestFile, theline);
    write(theline, string'("MIDIparxTI6,"));
    writeline(TestFile, theline);
    write(theline, string'("MIDIparxTI7;"));
    writeline(TestFile, theline);
    write(theline, string'("# OUTPUT pins"));
    writeline(TestFile, theline);
    write(theline, string'("OUP"));
    writeline(TestFile, theline);
    write(theline, string'("AudioSerSxDO,"));
    writeline(TestFile, theline);
    write(theline, string'("AudioSerLLExSO,"));
    writeline(TestFile, theline);
    write(theline, string'("AudioSerRLExSO,"));
    writeline(TestFile, theline);
    write(theline, string'("AudioSerLRxCO,"));
    writeline(TestFile, theline);
    write(theline, string'("AudioSerBitxCO,"));
    writeline(TestFile, theline);
    write(theline, string'("ScanOutxTO;"));
    writeline(TestFile, theline);
    write(theline, string'("# ================================="));
    writeline(TestFile, theline);
    write(theline, string'("TABLE_FORMAT"));
    writeline(TestFile, theline);
    write(theline, string'(" C CCCC CCC CC CC CCCCCCCC C CCCC C;"));
    writeline(TestFile, theline);
    write(theline, string'("#"));
    writeline(TestFile, theline);
    write(theline, string'("# M MMMM RCC SS BD MMMMMMMM A AAAA S"));
    writeline(TestFile, theline);
    write(theline, string'("# I IIIIIIII I I uuuu C"));
    writeline(TestFile, theline);
    write(theline, string'("# D DDDDDDDD I I dddd A"));
    writeline(TestFile, theline);
    write(theline, string'("# I IIIIIIII I I iiii N"));
    writeline(TestFile, theline);
```

```
      write(theline, string'("#␣s␣CCCC␣RCe␣EI␣sR␣ppppppppp␣o␣oooo␣O"));
      writeline(TestFile, theline);
      write(theline, string'("#␣e␣hhhh␣BIl␣nn␣sd␣aaaaaaaa␣S␣SSSS␣u"));
      writeline(TestFile, theline);
      write(theline, string'("#␣r␣aaaa␣I:x␣xx␣Ay␣rrrrrrrr␣e␣eeee␣t"));
      writeline(TestFile, theline);
      write(theline, string'("#␣x␣nnnn␣::S␣TT␣Rx␣xxxxxxxx␣r␣rrrr␣x"));
      writeline(TestFile, theline);
      write(theline, string'("#␣D␣xxxx␣::I␣II␣xT␣TTTTTTTT␣S␣LRLB␣T"));
      writeline(TestFile, theline);
      write(theline, string'("#␣L␣DDDD␣:::␣::␣TI␣IIIIIIII␣x␣LLRi␣O"));
      writeline(TestFile, theline);
      write(theline, string'("#␣:␣IIIL␣:::␣::␣I:␣01234567␣D␣EExt␣:␣"));
      writeline(TestFile, theline);
      write(theline, string'("#␣:␣0123␣:::␣::␣::␣::::::::␣I␣xxCx␣:␣"));
      writeline(TestFile, theline);
      write(theline, string'("#␣:␣::::␣:::␣::␣::␣::::::::␣:␣SSOC␣:␣"));
      writeline(TestFile, theline);
      write(theline, string'("#␣:␣::::␣:::␣::␣::␣::::::::␣:␣OO:O␣:␣"));
      writeline(TestFile, theline);
      write(theline, string'("DATA_SECTION"));
      writeline(TestFile, theline);

      -- data
      wait until SIMPROGRESSxS = green;
      while SIMPROGRESSxS = green loop
        wait until CLKxC'event and CLKxC = '1';
        write(theline, string'("␣"));
        write(theline, MIDIserxD);
        write(theline, '␣');
        write(theline, std_logic_vector(MIDIChanxD));
        write(theline, '␣');
        write(theline, RSTxRB);
        write(theline, string'("11␣"));    -- CLK{xC,SelxS}
        write(theline, string'("0X␣"));    -- Scan{En,In}xTI
        write(theline, BypassARxT);
        write(theline, DataRdyxT);
        write(theline, string'("␣"));
        write(theline, std_logic_vector(MIDIparxT));
        write(theline, string'("␣"));
        wait for 0.8 * PERIOD;
        write(theline, AudioSerSxD);
        write(theline, '␣');
        write(theline, AudioSerLLExS);
        write(theline, AudioSerRLExS);
        write(theline, AudioSerLRxC);
        write(theline, AudioSerBitxC);
        write(theline, string'("␣X;␣"));   -- ScanOutxTO
        -- ModelSim chokes because it may be red already and the file
        -- thus closed
        if SIMPROGRESSxS = green then
          writeline(TestFile, theline);
        end if;
      end loop;

    end if;
    report "ATE␣logger␣done." severity note;
    wait;
  end process;

  ------------------------------------------------------------------------
  -- send a MIDI byte to the synthesizer
  ------------------------------------------------------------------------
  process
  begin
    wait until SIMPROGRESSxS = green;

    if Bypass = true then
      ----------------------------------------------------------------------
      -- bypass AsyncReceiver
      -- in this process we try as hard as possible not do anything
      -- while we're scanning
      while SIMPROGRESSxS = green loop
        -- idle midi line for the async receiver
        MIDIserxD <= '1';
        MIDIparxT <= (others => '0');
        DataRdyxT <= '0';

        if SIMPROGRESSxS=green then
          wait until ToggleToSendxS'event;
        end if;
        -- unfortunately, simply "wait until ..." doesn't work so we
        -- check ScanEnxT first
        if ScanEnxT = '1' then
          wait until ScanEnxT = '0';
        end if;

        wait until CLKxC'event and CLKxC = '1';
        wait for WATCHTHECLOCKTIME;
        MIDIparxT <= std_logic_vector(SendMexD);
        if ScanEnxT = '1' then
          wait until ScanEnxT = '0';
        end if;
        DataRdyxT <= '1';
        wait until CLKxC'event and CLKxC = '1';
        wait for WATCHTHECLOCKTIME;
        if ScanEnxT = '1' then
          wait until ScanEnxT = '0';
        end if;
        DataRdyxT <= '0';

        ToggleWhenSentxS <= not ToggleWhenSentxS;
        report "Sent␣byte␣" & integer'image(to_integer(SendMexD))
          & "␣=␣$" & Byte2HexString(SendMexD)
          & "␣=␣%" & Byte2BinaryString(SendMexD)
          & "␣to␣the␣synthesizer"
          severity note;
      end loop;

    else
      ----------------------------------------------------------------------
      -- normal operation, not bypassing AsyncReceiver
      while SIMPROGRESSxS = green loop
        -- idle bypass input
        MIDIparxT <= (others => '0');
        DataRdyxT <= '0';

        wait until ToggleToSendxS'event;

        -- start bit
        if WatchTheClockxS = '1' then
          wait until WatchTheClockxS = '0';
        end if;
        MIDIserxD <= '0';
        wait for MIDIbitPERIOD;
        if ScanEnxT = '1' then
          wait until ScanEnxT = '0';
        end if;

        -- the byte
        for counter in 0 to 7 loop
          if WatchTheClockxS = '1' then
            wait until WatchTheClockxS = '0';
          end if;
          MIDIserxD <= SendMexD(counter);
          wait for MIDIbitPERIOD;
        end loop;

        -- stop bit
        if WatchTheClockxS = '1' then
          wait until WatchTheClockxS = '0';
        end if;
```

```
        MIDIserxD <= '1';
        wait for MIDIbitPERIOD;

        ToggleWhenSentxS <= not ToggleWhenSentxS;
        report "Sent byte " & integer'image(to_integer(SendMexD))
          & " = $" & Byte2HexString(SendMexD)
          & " = #" & Byte2BinaryString(SendMexD)
          & " to the synthesizer"
          severity note;
      end loop;

    end if;

    report "MIDI byte sender done." severity note;
    wait;
  end process;

  ------------------------------------------------------------------------
  -- THE FILE PARSER
  ------------------------------------------------------------------------
  process

    -- temporary
    variable thebyte : unsigned(7 downto 0);
    variable thehalf : unsigned(15 downto 0);
    variable theword : unsigned(31 downto 0);
    variable theint  : natural;
    variable thetime : time;
    variable thestr  : string(1 to 9);
    variable thecycles : natural;

    -- for running status we need to know the last command and its
    -- numbers of arguments
    variable lastcmd      : unsigned(7 downto 0) := ( others => '0');
    variable lastcmdargs  : natural;
    variable sysexlength  : natural;
    variable channel      : unsigned(3 downto 0);
    variable notenumber   : unsigned(7 downto 0);
    variable velocity     : unsigned(7 downto 0);
    variable ctrlnumber   : unsigned(7 downto 0);
    variable ctrlvalue    : unsigned(7 downto 0);

    -- more serious
    variable done   : boolean := false;
    variable status : file_open_status;

  begin
    wait until SIMPROGRESSxS = green;
    -- from here on we serially parse this file
    ------------------------------------------------------------------------
    -- check header
    ------------------------------------------------------------------------

    -- MThd
    if ReadWord(SMF) /= MThd then
      report "Illegal file header : not a valid Standard MIDI File."
        severity failure;
    else
      report "MThd header recognized" severity note;
    end if;

    -- length must be 6
    if ReadWord(SMF) /= to_unsigned(6, 32) then
      report "MThd header : Length must be 6."
        severity failure;
    else
      report "MThd header length = 6, fine" severity note;
    end if;

    -- only type 0 is recognized
    if ReadHalf(SMF) /= to_unsigned(0, 16) then
      report "MThd header : Only type 0 (single track) is recognized."
        severity failure;
    else
      report "SMF Format 0, good" severity note;
    end if;

    -- number of tracks
    if ReadHalf(SMF) /= to_unsigned(1, 16) then
      report "MThd header : Type 0 (single track) must contain one track only."
        severity failure;
    else
      report "One track, good" severity note;
    end if;

    -- parts per quarter note (PPQN)
    thehalf := ReadHalf(SMF);
    if thehalf(15) = '1' then
      report "MThd header : only PPQN are recognized"
        severity failure;
    else
      MIDI_ppqn <= to_integer(thehalf);
      report natural'image(to_integer(thehalf)) & " parts per quarter note";
    end if;


    ------------------------------------------------------------------------
    -- read the track
    ------------------------------------------------------------------------

    -- MTrk
    if ReadWord(SMF) /= MTrk then
      report "Currently only MTrk chunks are understood after the MThd header."
        severity failure;
    else
      report "MTrk chunk starts here" severity note;
    end if;

    -- length
    -- ignored because of the FF 2F 00 end-of-track message
    report natural'image(to_integer(ReadWord(SMF))) & " bytes in the track";

    -- loop
    while not done loop
      -- read delta-time for which we wait

      -- instead of simply "wait for thetime" we then we couldn't stop
      -- the file parser while performing the Scan-test
      theint  := to_integer(ReadVLQ(SMF));
      thetime := DeltaTimeFactor * real(timer_tempo) * real(theint) * 1.0 us;
      thecycles := thetime / PERIOD;
      final_cycle <= ClockCountxS + thecycles;
      wait for 1 ns;

      report "Waiting for Delta Time = " & integer'image(theint) & " parts = " &
        time'image(thetime) & " = " & integer'image(thecycles) & " clock cycles"
        severity note;

      -- actually we would prefer
      -- wait until ClockCountxS > final_cycle
      -- but that screws up AudioSerSxDO for whatever reason
      while ClockCountxS < final_cycle loop
        wait for PERIOD;
      end loop;

      -- this one should be a command or running status
      thebyte          := ReadByte(SMF);
      case to_integer(thebyte) is
                                          -- SysEx
        when 240 =>
          report "Sending SysEx message" severity note;
          sysexlength := to_integer(ReadVLQ(SMF));
```

```
    for counter in 1 to sysexlength loop
      SendMexD        <= readbyte(SMF);
      ToggleToSendxS <= not ToggleToSendxS;
      wait until ToggleWhenSentxS'event;
    end loop;  -- counter

                                    -- SysEx continuation
  when 247 =>
    report "Ignoring SysEx continuation message. Please send only proper SysEx"
      severity warning;
    Ignore(SMF, to_integer(ReadVLQ(SMF)));

                                    -- note off
  when 128 to 143 =>
    lastcmd      := thebyte;
    lastcmdargs := 2;
    channel      := thebyte(3 downto 0);
    notenumber   := ReadByte(SMF);
    velocity     := ReadByte(SMF);
    report "Sending note off " & integer'image(to_integer(notenumber)) &
      " with velocity " & integer'image(to_integer(velocity)) &
      " on channel " & integer'image(to_integer(channel))
      severity note;

    SendMexD        <= thebyte;
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

    SendMexD        <= notenumber;
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

    SendMexD        <= velocity;
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

                                    -- note on
  when 144 to 159 =>
    lastcmd      := thebyte;
    lastcmdargs := 2;
    channel      := thebyte(3 downto 0);
    notenumber   := ReadByte(SMF);
    velocity     := ReadByte(SMF);
    report "Sending note on " & integer'image(to_integer(notenumber)) &
      " with velocity " & integer'image(to_integer(velocity)) &
      " on channel " & integer'image(to_integer(channel))
      severity note;

    SendMexD        <= thebyte;
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

    SendMexD        <= notenumber;
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

    SendMexD        <= velocity;
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

                                    -- polyphonic key pressure/aftertouch
  when 160 to 175 =>
    lastcmd      := thebyte;
    lastcmdargs := 2;
    report "Sending some Polyphonic key pressure/aftertouch" severity note;
    SendMexD        <= ReadByte(SMF);
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;
    SendMexD        <= ReadByte(SMF);
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

                                    -- control change
  when 176 to 191 =>
    lastcmd      := thebyte;
    lastcmdargs := 2;
    channel      := thebyte(3 downto 0);
    ctrlnumber   := ReadByte(SMF);
    ctrlvalue    := ReadByte(SMF);
    report "Sending control change " & integer'image(to_integer(ctrlnumber)) &
      " to new value " & integer'image(to_integer(ctrlvalue)) &
      " on channel " & integer'image(to_integer(channel))
      severity note;

    SendMexD        <= thebyte;
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

    SendMexD        <= ctrlnumber;
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

    SendMexD        <= ctrlvalue;
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

                                    -- program change
  when 192 to 207 =>
    lastcmd      := thebyte;
    lastcmdargs := 1;
    report "Sending some program change" severity note;

    SendMexD        <= thebyte;
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

    SendMexD        <= ReadByte(SMF);
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

                                    -- channel pressure/aftertouch
  when 208 to 223 =>
    lastcmd      := thebyte;
    lastcmdargs := 1;
    report "Sending some channel pressure/afterouch" severity note;

    SendMexD        <= thebyte;
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

    SendMexD        <= ReadByte(SMF);
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

                                    -- pitch bend change
  when 224 to 239 =>
    lastcmd      := thebyte;
    lastcmdargs := 2;
    report "Sending some pitch bend change" severity note;

    SendMexD        <= thebyte;
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

    SendMexD        <= ReadByte(SMF);
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;

    SendMexD        <= ReadByte(SMF);
    ToggleToSendxS <= not ToggleToSendxS;
    wait until ToggleWhenSentxS'event;
```

```
          -- the following are SMF-only data which never occur in a realtime
          -- MIDI stream
-------------------------------------------------------------------------------
          -- $FF: non-MIDI events
        when 255 =>
          report "Non-MIDI event: " severity note;
          thebyte := ReadByte(SMF);
          case to_integer(thebyte) is

            -- Ignore Sequence Number
            when 0 =>
              report "Ignoring sequence number" severity note;
              Ignore(SMF, 3);

              -- Ignore Text, Copyright, Sequence/Track name, Instrument, Lyric,
              -- Marker, Cue Point
            when 1 to 7 =>
              report "Ignoring some text, marker or cue point" severity note;
              Ignore(SMF, to_integer(ReadVLQ(SMF)));


              -- MIDI Channel
            when 32 =>
              if ReadByte(SMF) /= 1 then
                report "Expecting $01 after $FF20 (Channel number message)"
                  severity failure;
              end if;
              MIDIChanxD <= ReadByte(SMF)(3 downto 0);
              report "Using Channel " & natural'image(to_integer(MIDIChanxD));

              -- Ignore MIDI port
            when 33 =>
              report "Ignoring MIDI port" severity note;
              Ignore(SMF, 2);

              -- End Of Track
            when 47 =>
              if ReadByte(SMF) /= 0 then
                report "Expecting $00 after $FF2F (End Of Track message)"
                  severity failure;
              end if;
              report "End of Track"
                severity note;
              done := true;

              -- Tempo
            when 81 =>
              if ReadByte(SMF)/= 3 then
                report "Expecting $03 after $FF51 (Tempo message)"
                  severity failure;
              end if;
              theint := to_integer(ReadByte(SMF) & ReadByte(SMF) & ReadByte(SMF));
              report natural'image(theint) & " microseconds per quarter note"
                severity note;
              MIDI_tempo <= theint;

              -- Ignore SMPTE offset
            when 84 =>
              if ReadByte(SMF) /= 5 then
                report "Expecting $05 after $FF54 (SMPTE offset message)"
                  severity failure;
              end if;
              report "Ignoring SMPTE offset" severity note;
              Ignore(SMF, 5);

              -- Ignore Time Signature
            when 88 =>
              if ReadByte(SMF) /= 4 then
                report "Expecting $04 after $FF58 (Time signature message)"
                  severity failure;
              end if;
              report "Ignoring Time Signature" severity note;
              Ignore(SMF, 4);

              -- Ignore Key Signature
            when 89 =>
              if ReadByte(SMF) /= 2 then
                report "Expecting $02 after $FF58 (Key signature message)"
                  severity failure;
              end if;
              report "Ignoring Key Signature" severity note;
              Ignore(SMF, 2);

              -- Ignore proprietary data
            when 127 =>
              report "Ignoring proprietary data" severity note;
              Ignore(SMF, to_integer(ReadVLQ(SMF)));

            when others =>
              report "Ignoring unknown byte after $FF (non-MIDI message)"
                severity failure;
          end case;
          -- end of non-MIDI events
-------------------------------------------------------------------------------

        when others =>
          -- that should be a running status
          report "Running status with command $"
            & Byte2HexString(lastcmd) & " using "
            & integer'image(lastcmdargs) & " argument(s)"
            severity note;

          SendMexD         <= thebyte;
          ToggleToSendxS <= not ToggleToSendxS;
          wait until ToggleWhenSentxS'event;

          if lastcmdargs > 1 then
            for counter in 1 to lastcmdargs-1 loop
              SendMexD         <= ReadByte(SMF);
              ToggleToSendxS <= not ToggleToSendxS;
              wait until ToggleWhenSentxS'event;
            end loop;
          end if;

      end case;

      -- are we done?
      if not(done) and endfile(SMF) then
        report "End of file reached without End-of-Track message" severity warning;
        done := true;
      end if;

      -- end of "while not done loop"
    end loop;
    SIMPROGRESSxS <= red;
    report "Finished reading the MIDI file." severity note;

    report "SMF parser done." severity note;
    wait;
  end process;

-- instantiate the one and only synthesizer
  u_chip : chip
    port map (
      CLKxCI          => CLKxC,
      CLKSelxSI       => DummyHi,
      RSTxRBI         => RSTxRB,
      ScanEnxTI       => ScanEnxT,
      ScanInxTI       => ScanInxT,
      ScanOutxTO      => ScanOutxT,
      MIDIparxTI      => MIDIparxT,
```

```
      DataRdyxTI      => DataRdyxT,
      BypassARxTI     => BypassARxT,
      MIDIserxDI      => MIDIserxD,
      MIDIChanxDI     => MIDIChanxD,
      AudioSerSxDO    => AudioSerSxD,
      AudioSerLLExSO  => AudioSerLLExS,
      AudioSerRLExSO  => AudioSerRLExS,
      AudioSerLRxCO   => AudioSerLRxC,
      AudioSerBitxCO  => AudioSerBitxC );

end behavioural;
```

top.vhd

```
-------------------------------------------------------------------------------
-- Title      : Synthesizer Top File
-- Project    :
-------------------------------------------------------------------------------
-- File       : top.vhd
-- Author     : Samuel Nobs <nobssa@ee.ethz.ch>
-- Company    : Integrated Systems Laboratory, ETH Zurich
-- Created    : 2002/11/11
-- Last update: 2003/01/15
-- Platform   : ModelSim (simulation), Synopsys (synthesis)
-------------------------------------------------------------------------------
-- Description: Combines all blocks into the final synthesizer
-------------------------------------------------------------------------------
-- Copyright (c) 2002 Integrated Systems Laboratory, ETH Zurich
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author       Description
-- 2002/11/11  1.0      Samuel Nobs  Created
-- 2003/01/09           Samuel Nobs  AsyncRecv may be bypassed now for improved
--                                   testability
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top is

  port (
    MIDIserxDI      : in  std_logic;      -- the incoming midi stream
    MIDIChanxDI     : in  unsigned(3 downto 0);
                                          -- the channel we listen to
    AudioSerSxDO    : out std_logic;      -- serial sound output
    AudioSerLLExSO  : out std_logic;      -- latch enable left side
    AudioSerRLExSO  : out std_logic;      -- latch enable right side
    AudioSerLRxCO   : out std_logic;      -- left/right channel clock for DAC
    AudioSerBitxCO  : out std_logic;      -- bit clock for DAC

    ScanEnxTI   : in  std_logic;
    ScanInxTI   : in  std_logic;
    ScanOutxTO  : out std_logic;

    MIDIParxTI      : in  std_logic_vector(7 downto 0);
                                          -- parallel midi stream
    DataRdyxTI      : in std_logic;       -- data ready signal
    BypassARxTI     : in std_logic;       -- bypass async receiver

    CLKxCI  : in std_logic;               -- the clock
    RSTxRBI : in std_logic );             -- asynchronous reset, active low

end top;


architecture rtl of top is

  -----------------------------------------------------------------------------
  -- declare everything we need
  -----------------------------------------------------------------------------
  component i2scontroller
    port (
      RParxDI : in  unsigned(15 downto 0);
      LParxDI : in  unsigned(15 downto 0);
      SerxDO  : out std_logic;
      LRxCO   : out std_logic;
      BitxCO  : out std_logic;
      RLExSO  : out std_logic;
      LLExSO  : out std_logic;
      CLKxCI  : in  std_logic;
      RSTxRBI : in  std_logic);
  end component;

  component AsyncRecv
    port (
      MIDIserxDI : in  std_logic;
      MIDIparxDO : out std_logic_vector(7 downto 0);
      DataRdyxSO : out std_logic;
      CLKxCI     : in  std_logic;
      RSTxRBI    : in  std_logic);
  end component;

  component MIDIcontroller
    port (
      MIDIparxDI   : in  std_logic_vector(7 downto 0);
      DataRdyxSI   : in  std_logic;
      MChannelxDI  : in  unsigned(3 downto 0);
      CtrNumxDO    : out unsigned(6 downto 0);
      CtrValxDO    : out unsigned(6 downto 0);
      RegWExSO     : out std_logic;
      NoteNumxDO   : out unsigned(6 downto 0);
      NoteVelxDO   : out unsigned(6 downto 0);
      NoteGatexSO  : out std_logic;
      LUTInitxSO   : out std_logic;
      CLKxCI       : in  std_logic;
      RSTxRBI      : in  std_logic);
  end component;

  component ConfReg
    port (
      CtrlNumxDI    : in  unsigned(6 downto 0);
      CtrlValxDI    : in  unsigned(6 downto 0);
      RegWExSI      : in  std_logic;
      RectEnaxSO    : out unsigned(6 downto 0);
      ChanVol0UxDO  : out unsigned(6 downto 0);
      ChanVol1UxDO  : out unsigned(6 downto 0);
      ChanVol2UxDO  : out unsigned(6 downto 0);
      ChanVol3UxDO  : out unsigned(6 downto 0);
      ChanVol4UxDO  : out unsigned(6 downto 0);
      ChanVol5UxDO  : out unsigned(6 downto 0);
      ChanVol6UxDO  : out unsigned(6 downto 0);
      ChanVol7UxDO  : out unsigned(6 downto 0);
      ChanPan0UxDO  : out unsigned(6 downto 0);
      ChanPan1UxDO  : out unsigned(6 downto 0);
      ChanPan2UxDO  : out unsigned(6 downto 0);
      ChanPan3UxDO  : out unsigned(6 downto 0);
      ChanPan4UxDO  : out unsigned(6 downto 0);
      ChanPan5UxDO  : out unsigned(6 downto 0);
      ChanPan6UxDO  : out unsigned(6 downto 0);
      ChanPan7UxDO  : out unsigned(6 downto 0);
      ADSR0xDO      : out unsigned(27 downto 0);
      ADSR1xDO      : out unsigned(27 downto 0);
      ADSR2xDO      : out unsigned(27 downto 0);
      ADSR3xDO      : out unsigned(27 downto 0);
      ADSR4xDO      : out unsigned(27 downto 0);
      ADSR5xDO      : out unsigned(27 downto 0);
      ADSR6xDO      : out unsigned(27 downto 0);
      ADSR7xDO      : out unsigned(27 downto 0);
```

```
      MainVolUxDO   : out unsigned(6 downto 0);
      CLKxCI        : in  std_logic;
      RSTxRBI       : in  std_logic);
  end component;

  component ADSR
    port (
      NoteGatexSI : in  std_logic;
      OscRSTxSI   : in  std_logic;
      ADSRxDI     : in  unsigned(27 downto 0);
      AmplUxDO    : out unsigned(8 downto 0);
      CLKxCI      : in  std_logic;
      RSTxRBI     : in  std_logic);
  end component;

  component rectifier
    port (
      ENAxSI  : in  std_logic;
      InSxDI  : in  unsigned(15 downto 0);
      OutSxDO : out unsigned(15 downto 0));
  end component;

  component LUT
    port (
      ClkxCI      : in  std_logic;
      ResetxRBI   : in  std_logic;
      InitxSI     : in  std_logic;
      NoteNumxDI  : in  unsigned(6 downto 0);
      OscInitxSO  : out unsigned(7 downto 0);
      MutexSO     : out std_logic;
      Y1xDO       : out unsigned(24 downto 0));
  end component;

  component oscillator
    port (
      ClkxCI      : in  std_logic;
      ResetxRBI   : in  std_logic;
      InitxSI     : in  std_logic;
      MutexSI     : in  std_logic;
      AnewxSO     : out std_logic;
      Y1xDI       : in  unsigned(24 downto 0);
      YxDO        : out unsigned(15 downto 0));
  end component;

  component MultiplyController
    port (
      SelxSO     : out std_logic_vector(1 downto 0);
      InitxSO    : out std_logic;
      EnaRegxSO  : out std_logic_vector(1 downto 0);
      CLKxCI     : in  std_logic;
      RSTxRBI    : in  std_logic);
  end component;

  component ComplexMultiplier
    port (
      AmplUxDI     : in  unsigned(8 downto 0);
      ChanVolUxDI  : in  unsigned(6 downto 0);
      MainVolUxDI  : in  unsigned(6 downto 0);
      VelUxDI      : in  unsigned(6 downto 0);
      YSxDI        : in  unsigned(15 downto 0);
      OutSxDO      : out unsigned(15 downto 0);
      SelxSI       : in  std_logic_vector(1 downto 0);
      InitxSI      : in  std_logic;
      EnaRegxSI    : in  std_logic_vector(1 downto 0);
      CLKxCI       : in  std_logic;
      RSTxRBI      : in  std_logic);
  end component;

  component Panorama
    port (
      PanxDI    : in  unsigned(6 downto 0);
      InSxDI    : in  unsigned(15 downto 0);
      LOutSxDO  : out unsigned(15 downto 0);
      ROutSxDO  : out unsigned(15 downto 0);
      CLKxCI    : in  std_logic;
      RSTxRBI   : in  std_logic);
  end component;

  component bigadder
    port (
      ClkxCI     : in  std_logic;
      ResetxRBI  : in  std_logic;
      In0xDI     : in  unsigned(15 downto 0);
      In1xDI     : in  unsigned(15 downto 0);
      In2xDI     : in  unsigned(15 downto 0);
      In3xDI     : in  unsigned(15 downto 0);
      In4xDI     : in  unsigned(15 downto 0);
      In5xDI     : in  unsigned(15 downto 0);
      In6xDI     : in  unsigned(15 downto 0);
      In7xDI     : in  unsigned(15 downto 0);
      OutxDO     : out unsigned(15 downto 0));
  end component;

  ---------------------------------------------------------------------------
  -- define nets
  ---------------------------------------------------------------------------

  -- nets connecting Async Receiver with Bypass Mux
  signal MIDIparxD : std_logic_vector(7 downto 0);
  signal DataRdyxS : std_logic;

  -- nets connecting Bypass Mux with MIDI controller
  signal MIDIparMXxD: std_logic_vector(7 downto 0);
  signal DataRdyMXxS: std_logic;

  -- nets connecting MIDI controller and configuration register
  signal CtrNumxD : unsigned(6 downto 0);
  signal CtrValxD : unsigned(6 downto 0);
  signal RegWExS  : std_logic;

  -- net connecting MIDI controller and complex mutliplier
  signal NoteVelxD : unsigned(6 downto 0);

  -- net connecting MIDI controller and ADSR
  signal NoteGatexS : std_logic;

  -- nets connecting MIDI controller and LUT
  signal NoteNumxD : unsigned(6 downto 0);
  signal LUTInitxS : std_logic;

  -- nets connecting lut with oscillators
  signal OscInitxS : unsigned(7 downto 0);  -- one bit for each oscillator
  signal Y1SxD     : unsigned(24 downto 0);  -- one Y1 for all oscillators
  signal MutexS    : std_logic;          -- one Mute signal for all oscillators

  -- net connecting oscillator with rectifier
  signal YxD : unsigned(127 downto 0);

  -- nets connecting configuration register with complex multiplier
  signal ChanVolUxD : unsigned(55 downto 0);
  signal MainVolUxD : unsigned(6 downto 0);

  -- nets connecting configuration register with panorama
  signal ChanPanUxD : unsigned(55 downto 0);

  -- net connecting configuration register with rectifier
  signal RectEnaxS : unsigned(6 downto 0);

  -- net connecting configuration register with adsr
  signal ADSRxD : unsigned(223 downto 0);

  -- net connecting rectifier with complex multiplier
```

```
  signal RectOutSxD : unsigned(111 downto 0);

  -- net connecting adsr with complex multiplier
  signal AmplUxD : unsigned(71 downto 0);

  -- net connecting multiply controller with complex multiplier
  signal MulSelxS     : std_logic_vector(1 downto 0);
  signal MulInitxS    : std_logic;
  signal MulEnaRegxS : std_logic_vector(1 downto 0);

  -- net connecting complex multiplier with panorama
  signal OutSxD : unsigned(127 downto 0);

  -- nets connecting panorama with bigadders
  signal ROutSxD : unsigned(127 downto 0);
  signal LOutSxD : unsigned(127 downto 0);

  -- nets connecting bigadders with i2scontroller
  signal LParSxD : unsigned(15 downto 0);
  signal RParsxD : unsigned(15 downto 0);

  -- oscillator synch nets
  signal OscRdyxS     : std_logic_vector(7 downto 0);

begin  -- rtl

  u_async_receiver : AsyncRecv
    port map (
      MIDIserxDI => MIDIserxDI,
      MIDIparxDO => MIDIparxD,
      DataRdyxSO => DataRdyxS,
      CLKxCI     => CLKxCI,
      RSTxRBI    => RSTxRBI);


  -- Bypass Mux bypassing AsyncRecv
  bypassmux : process (BypassARxTI, DataRdyxS, DataRdyxTI, MIDIParxTI,
                       MIDIparxD)
  begin
    if BypassARxTI = '1' then                -- bypass
      DataRdyMXxS <= DataRdyxTI;
      MIDIparMXxD <= MIDIParxTI;
    else                                     -- don't bypass
      DataRdyMXxS <= DataRdyxS;
      MIDIparMXxD <= MIDIparxD;
    end if;
  end process bypassmux;


  u_midi_controller : MIDIcontroller
    port map (
      MIDIparxDI   => MIDIparMXxD,
      DataRdyxSI   => DataRdyMXxS,
      MChannelxDI => MIDIChanxDI,
      CtrNumxDO    => CtrNumxD,
      CtrValxDO    => CtrValxD,
      RegWExSO     => RegWExS,
      NoteNumxDO   => NoteNumxD,
      NoteVelxDO   => NoteVelxD,
      NoteGatexSO => NoteGatexS,
      LUTInitxSO   => LUTInitxS,
      CLKxCI       => CLKxCI,
      RSTxRBI      => RSTxRBI);

  u_conf_reg : ConfReg
    port map (
      CtrlNumxDI    => CtrNumxD,
      CtrlValxDI    => CtrValxD,
      RegWExSI      => RegWExS,
      RectEnaxSO    => RectEnaxS,
      ChanVol0UxDO => ChanVolUxD(6 downto 0),
      ChanVol1UxDO => ChanVolUxD(13 downto 7),
      ChanVol2UxDO => ChanVolUxD(20 downto 14),
      ChanVol3UxDO => ChanVolUxD(27 downto 21),
      ChanVol4UxDO => ChanVolUxD(34 downto 28),
      ChanVol5UxDO => ChanVolUxD(41 downto 35),
      ChanVol6UxDO => ChanVolUxD(48 downto 42),
      ChanVol7UxDO => ChanVolUxD(55 downto 49),
      ChanPan0UxDO => ChanPanUxD(6 downto 0),
      ChanPan1UxDO => ChanPanUxD(13 downto 7),
      ChanPan2UxDO => ChanPanUxD(20 downto 14),
      ChanPan3UxDO => ChanPanUxD(27 downto 21),
      ChanPan4UxDO => ChanPanUxD(34 downto 28),
      ChanPan5UxDO => ChanPanUxD(41 downto 35),
      ChanPan6UxDO => ChanPanUxD(48 downto 42),
      ChanPan7UxDO => ChanPanUxD(55 downto 49),
      ADSR0xDO      => ADSRxD(27 downto 0),
      ADSR1xDO      => ADSRxD(55 downto 28),
      ADSR2xDO      => ADSRxD(83 downto 56),
      ADSR3xDO      => ADSRxD(111 downto 84),
      ADSR4xDO      => ADSRxD(139 downto 112),
      ADSR5xDO      => ADSRxD(167 downto 140),
      ADSR6xDO      => ADSRxD(195 downto 168),
      ADSR7xDO      => ADSRxD(223 downto 196),
      MainVolUxDO   => MainVolUxD,
      CLKxCI        => CLKxCI,
      RSTxRBI       => RSTxRBI);

  u_lut : LUT
    port map (
      ClkxCI       => CLKxCI,
      ResetxRBI    => RSTxRBI,
      InitxSI      => LUTInitxS,
      NoteNumxDI => NoteNumxD,
      OscInitxSO => OscInitxS,
      MutexSO      => MutexS,
      Y1xDO        => Y1SxD);

  osc_sections : for index in 0 to 7 generate
  begin

    u_oscillator : oscillator
      port map (
        ClkxCI      => CLKxCI,
        ResetxRBI => RSTxRBI,
        InitxSI     => OscInitxS(index),
        MutexSI     => MutexS,
        AnewxSO     => OscRdyxS(index),
        Y1xDI       => Y1SxD,
        YxDO        => YxD(16*index+15 downto 16*index));

    u_adsr : ADSR
      port map (
        NoteGatexSI => NoteGatexS,
        OscRSTxSI    => OscRdyxS(index),
        ADSRxDI      => ADSRxD(28*index+27 downto 28*index),
        AmplUxDO     => AmplUxD(9*index+8 downto 9*index),
        CLKxCI       => CLKxCI,
        RSTxRBI      => RSTxRBI);

    first_seven    : if index < 7 generate
      u_rectifier : rectifier
        port map (
          ENAxSI  => RectEnaxS(index),
          InSxDI  => YxD(16*index+15 downto 16*index),
          OutSxDO => RectOutSxD(16*index+15 downto 16*index));

      u_complexmultiplier : ComplexMultiplier
        port map (
          AmplUxDI     => AmplUxD(9*index+8 downto 9*index),
          ChanVolUxDI => ChanVolUxD(7*index+6 downto 7*index),
          MainVolUxDI => MainVolUxD,
```

```
            VelUxDI       => NoteVelxD,
            YSxDI         => RectOutSxD(16*index+15 downto 16*index),
            OutSxDO       => OutSxD(16*index+15 downto 16*index),
            SelxSI        => MulSelxS,
            InitxSI       => MulInitxS,
            EnaRegxSI     => MulEnaRegxS,
            CLKxCI        => CLKxCI,
            RSTxRBI       => RSTxRBI);

    end generate first_seven;

    last                          : if index = 7 generate
      u_last_complexmultiplier : ComplexMultiplier
        port map (
          SelxSI        => MulSelxS,
          InitxSI       => MulInitxS,
          EnaRegxSI     => MulEnaRegxS,
          AmplUxDI      => AmplUxD(9*index+8 downto 9*index),
          ChanVolUxDI   => ChanVolUxD(7*index+6 downto 7*index),
          MainVolUxDI   => MainVolUxD,
          VelUxDI       => NoteVelxD,
          YSxDI         => YxD(16*index+15 downto 16*index),
          OutSxDO       => OutSxD(16*index+15 downto 16*index),
          CLKxCI        => CLKxCI,
          RSTxRBI       => RSTxRBI);
    end generate last;

    u_panorama : Panorama
      port map (
        PanxDI     => ChanPanUxD(7*index+6 downto 7*index),
        InSxDI     => OutSxD(16*index+15 downto 16*index),
        LOutSxDO   => LOutSxD(16*index+15 downto 16*index),
        ROutSxDO   => ROutSxD(16*index+15 downto 16*index),
        CLKxCI     => CLKxCI,
        RSTxRBI    => RSTxRBI);
  end generate osc_sections;

  u_multiplycontroller : MultiplyController
    port map (
      SelxSO      => MulSelxS,
      InitxSO     => MulInitxS,
      EnaRegxSO   => MulEnaRegxS,
      CLKxCI      => CLKxCI,
      RSTxRBI     => RSTxRBI);

  u_bigadder_left : bigadder
    port map (
      ClkxCI      => CLKxCI,
      ResetxRBI   => RSTxRBI,
      In0xDI      => LOutSxD(15 downto 0),
      In1xDI      => LOutSxD(31 downto 16),
      In2xDI      => LOutSxD(47 downto 32),
      In3xDI      => LOutSxD(63 downto 48),
      In4xDI      => LOutSxD(79 downto 64),
      In5xDI      => LOutSxD(95 downto 80),
      In6xDI      => LOutSxD(111 downto 96),
      In7xDI      => LOutSxD(127 downto 112),
      OutxDO      => LParSxD);

  u_bigadder_right : bigadder
    port map (
      ClkxCI      => CLKxCI,
      ResetxRBI   => RSTxRBI,
      In0xDI      => ROutSxD(15 downto 0),
      In1xDI      => ROutSxD(31 downto 16),
      In2xDI      => ROutSxD(47 downto 32),
      In3xDI      => ROutSxD(63 downto 48),
      In4xDI      => ROutSxD(79 downto 64),
      In5xDI      => ROutSxD(95 downto 80),
      In6xDI      => ROutSxD(111 downto 96),
      In7xDI      => ROutSxD(127 downto 112),
      OutxDO      => RParSxD);

  u_i2scontroller : i2scontroller
    port map (
      RParxDI  => RParSxD,
      LParxDI  => LParSxD,
      SerxDO   => AudioSerSxDO,
      LRxCO    => AudioSerLRxCO,
      BitxCO   => AudioSerBitxCO,
      RLExSO   => AudioSerRLExSO,
      LLExSO   => AudioSerLLExSO,
      CLKxCI   => CLKxCI,
      RSTxRBI  => RSTxRBI);

end rtl;
```

Bibliography

[1] Rothstein, Joseph: *MIDI: a comprehensive introduction*, A-R Editions, Inc. (1992)

[2] Lehrman, Paul D., and Tully, Tim: *MIDI for the Professional*, Amsco Publications (1993)

[3] Data Sheet for PCM1725, Burr-Brown Corp. / Texas Instruments Inc. (1997)

[4] Data Sheet for AD1851/AD1861, rev. A, Analog Devices (no date given)

[5] Data Sheet for DAC56, Burr-Brown Corp. / Texas Instruments Inc. (2000)

[6] `http://jedi.ks.uiuc.edu/~johns/links/music/midifile.htm` (valid in November 2002)