# PD82

## 8-Channel, 2-Operator Phase Distortion Synthesizer

Project Summary

Name: Osman Celimli
Date: 5/7/2010
Course: 0306-561-01

## Project Description:

The PD82 is an 8-Channel, 2-Operator Phase distortion audio synthesizer supporting full stereo panning on a per-channel basis. Synthesizer control is performed by interfacing the PD82 with a host CPU, and 15-Bit parallel audio output is provided for easy connection to a binary DAC. The design and implementation of this sound generator not only investigates the various ways to generate sound in real time, but also the use of an FPGA to contain hardware which may be interfaced with an existing microcomputer system.

Phase Distortion synthesis in terms of audio is the process of generating a complex waveform by dynamically adjusting the phase of one waveform by the amplitude of another. Thus resulting in a more vibrant, and sometimes completely different output. Each waveform generator may be considered to be an operator with a structure shown to the right in Figure 1. The operator has its own frequency and volume and is given a “feedin” value used to adjust the phase of the waveform it is generating.

In the case of the PD82, eight channels of two operators each are used to generate audio, with the two operators of each channel having independent frequencies, volumes, waveforms, and phases.

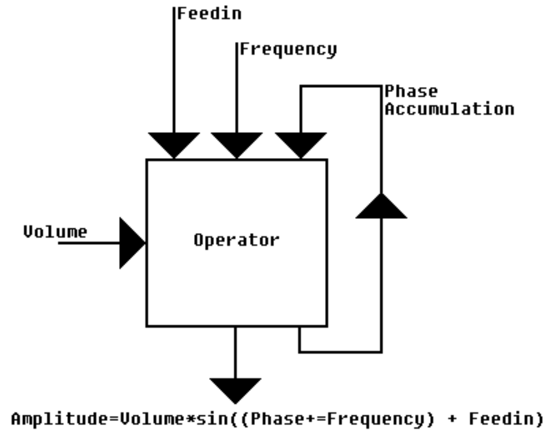


**Figure 1:** Phase Distortion Operator

The channel structure of the PD82 is shown to the left in Figure 2. Two independent operators generating quadrant adjustable sine waves are used to create the resulting waveform for a single channel. The output of the first operator (0) is used to adjust the phase of the second operator (1). As both operators may have different base frequencies and waveforms, several unique harmonics may be generated in a single channel.

The output of both operators is configured for stereo panning on a per channel basis with 8-Bit precision, and the output of all eight channels is summed and asserted every 208 clocks.

No ADSR is implemented in hardware, and thus must be controlled by the host system.

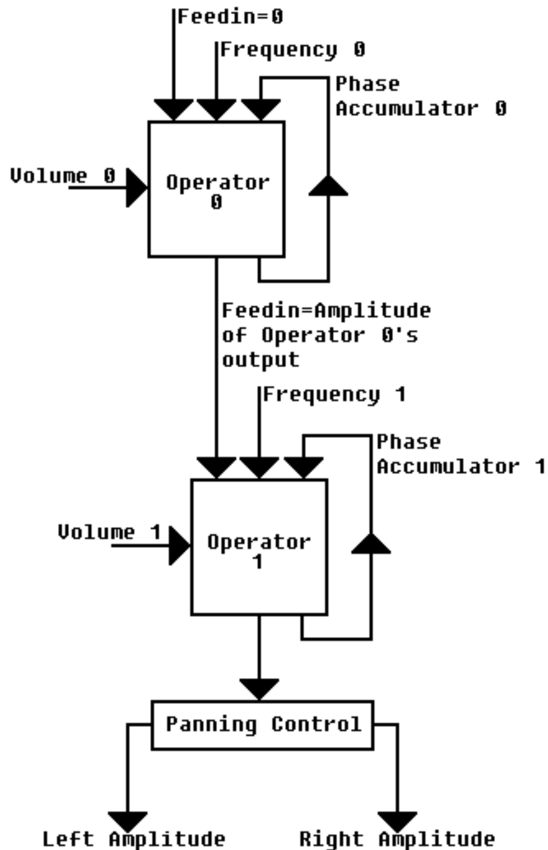


**Figure 2:** PD82 Channel Layout

## Device Pinout and Memory Map:

The PD82 was implemented in a 48 pin package with assignments as follows:

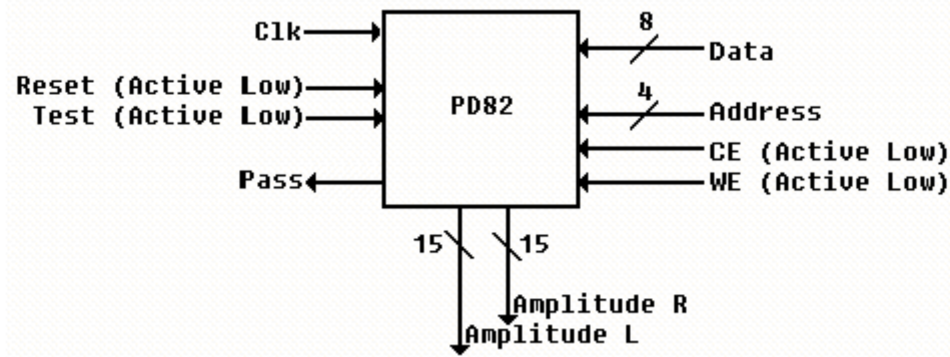


**Figure 3:** PD82 Pinout

Given the 8-bit data bus and 4-bit address bus, the chip's memory map was laid out as shown in the table below. Addresses $0-$2 are global, while $6-$F are channel-specific and effect the current channel selected for writing with register $0. All registers are write only.

| Address | Description | Format |
|---|---|---|
| $0 | Channel Select | xxxxxCCC |
| $1 | Channel On | 76543210 |
| $2 | Channel Off | 76543210 |
| $6 | Panning (L) | VVVVVVVV |
| $7 | Panning (R) | VVVVVVVV |
| $8 | Operator 0 Frequency (Low) | LLLLLLLL |
| $9 | Operator 0 Frequency (High) | HHHHHHHH |
| $A | Operator 0 Volume | VVVVVVVV |
| $B | Operator 0 WaveMode | xxxxMMMM |
| $C | Operator 1 Frequency (Low) | LLLLLLLL |
| $D | Operator 1 Frequency (High) | HHHHHHHH |
| $E | Operator 1 Volume | VVVVVVVV |
| $F | Operator 1 WaveMode | xxxxMMMM |

**Table I:** PD82 Memory Map

- **Detailed Register Descriptions:**
  - **$0: Channel Select-** Used to select the channel whose state will be effected by writes to registers $6 through $F. Given there are only eight channels in the synthesizer, the acceptable range of values is 0-7.
  - **$1: Channel On-** May be considered as a "Key On" command. Each bit in the byte wide register corresponds to one channel in the synthesizer, thus multiple channels may be enabled at once.
  - **$2: Channel Off-** May be considered as a "Key Off" command. Each bit in the byte wide register corresponds to one channel in the synthesizer, thus multiple channels may be disabled at once.
  - **$6-7: Panning (L/R)-** The stereo panning of the current channel. $00 results in silence, while $FF is maximum volume through the respective speaker.
  - **$8-9, $C-D: Frequency (Low,High)-** The current frequency of the respective operator.
  - **$A, $E: Volume-** The current volume of the respective operator. $00 results in silence, while $FF is maximum volume. Operator 1's volume directly effects the output amplitude, while Operator 0's is the intensity of its modulation of Operator 1's phase.

- **$B, $F: WaveMode-** Enables and disables given quarters of the sinusoidal lookup table used by the operators. A “1” in one of the respective four register bits enables the table region, while a “0” forces it to always read zero. Thus, alternative modulation and output waveforms may be generated.

## Testing Model:

The PD82 was synthesized on an Altera DE2 board’s Cyclone II FPGA, with an operational clock of 50Mhz. This was then interfaced with a Super Nintendo Entertainment System running an application used to poke values into the PD82’s registers. As the two devices operated on different voltages (5V and 3.3V for the Super Nintendo and Cyclone II respectively), a unidirectional voltage converter was used to ensure the safety of the FPGA. Audio output from the PD82 was provided by a binary DAC. Both the binary DAC and interface to the Super Nintendo were connected through the DE2’s general purpose IO ports.

**Figure 4:** PD82 Hardware Testing Configuration

A simple built in self test was also included, asserting an active low “Pass” pin which indicates proper operation of the synthesizer. Test mode is activated by pulling the “Test” pin low, and then restoring it to a high. All channels are activated and checked to ensure they do not pass a certain threshold.

## Device Performance:

Processing and updating a single channel was found to require 26 cycles, thus a total of 208 cycles are required for the chip to completely update its state and assert the next set of output amplitudes. A breakdown of the 26 cycles to update a single channel is shown below.

| Clock | Chip | Channel | Operator |
|---|---|---|---|
| 0 | Channel Configuration Feed | Fetch Configuration | Fetch Configuration |
| 1 | | | |
| 2 | Amplitude Accumulation | Operator Configuration Feed | |
| 3 | Output Assertion | | |
| 4 | Output Accumulation Reset | Delay | Phase Calculation |
| 5 | Accept Register Writes | | Sine ROM Strobe |
| 6 | | | Sine ROM Read |
| 7 | | | Amplitude Calculation |
| 8 | | Operator Configuration Fetch | Store Configuration |
| 9 | | | |
| 10 | | Delay | |
| 11 | | | |
| 12 | | | |
| 13 | | | Fetch Configuration |
| 14 | | | |
| 15 | | Operator Configuration Feed | |
| 16 | | | |
| 17 | | Delay | Phase Calculation |
| 18 | | | Sine ROM Strobe |
| 19 | | | Sine ROM Read |
| 20 | | | Amplitude Calculation |
| 21 | | Operator Configuration Fetch | Store Configuration |
| 22 | | | |
| 23 | Assert Register Write | Amplitude Calculation | |
| 24 | Assert Channel On/Offs | Store Configuration | |
| 25 | Channel Configuration Fetch | | |

**Table II:** Cycle Breakdown of Channel Processing

## Project Conclusions:

Overall, the project was a success, with the synthesizer functioning as designed and generating acceptable waveforms for its application. The resulting programming model for the chip was simple to work with, and a sound driver would not be difficult to implement. Overall chip complexity was low, and it could easily be included in an FPGA or CPLD with a CPU and other various components to make a low cost microcontroller with audio output.

However, it was also noted that improvements could have been made. Design complexity and power consumption would have been reduced through the use of a log transformed sine table, as this would have allowed for amplitude calculations to be performed through additions rather than multiplications. Additionally, writes to the chip were frequently missed due to both synchronization issues and a poor implementation of capturing bus writes. Serial output of the audio amplitude instead of parallel may have also been desirable.

```
------------------------------------------------
-- File: PD82.vhd
-- Entity: pd82
-- Architecture: behavior
-- Author: Osman Celimli
-- Created: 4/20/2010
-- Modified: 5/07/2010
-- VHDL'93
-- Description: The overall component structure
-- of the PD82.
------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
--Custom Type Pack
library work;
use work.PD82Types.all;
----------------------------
-- Entity for the PD82 Chip.
----------------------------
entity pd82 is
               --Ports
               Port( clk,reset,test : in std_logic;
                     --Signal out L/R
                     amplitude_l,amplitude_r : out std_logic_vector(14 downto 0);
                     --Address/Data and Chip/Write Enable (Active low)
                     data : in std_logic_vector(7 downto 0);
                     address : in std_logic_vector(3 downto 0);
                     ce,we : in std_logic;
                      pass : out std_logic);
end entity pd82;
----------------------------------
-- Architecture for the PD82 Chip.
----------------------------------
architecture behavior of pd82 is
  --------------------------
  --PD Channel Component
  --------------------------
  component pdchannel is
               Port( clk,reset,test : in std_logic;
                     amplitude_l,amplitude_r : out std_logic_vector(10 downto 0);
                     settings_in : in chconfig;
                     settings_out : out chconfig);
  end component pdchannel;
        ----------
        -- Signals
        ----------
        --For the channel
        signal ch_clk,ch_reset,ch_test : std_logic;
        signal ch_amplitude_l,ch_amplitude_r : std_logic_vector(10 downto 0);
        signal ch_settings_in,ch_settings_out : chconfig;
        --------------------------------
        -- Architecture code starts here
        --------------------------------
        begin
               --Map in Channel
               channelmap: component pdchannel port map(ch_clk, ch_reset, ch_test,
                                                      ch_amplitude_l,ch_amplitude_r,
                                                      ch_settings_in,ch_settings_out);
               --Also clock it, plus reroute reset.
               ch_clk <= clk;
               ch_reset <= reset;
               ch_test <= test;

        update : process (clk,test,reset)
               --Internal Vars
               variable mode : unsigned(4 downto 0);
               variable currentchannel : unsigned(2 downto 0);
               variable channels : chipconfig;
               variable accumulator_l,accumulator_r : signed(14 downto 0);
               variable fetch_l,fetch_r : signed(10 downto 0);
               variable output_l,output_r : std_logic_vector(14 downto 0);
               --For Data write
```

```
variable write_req,testm : std_logic;
variable write_data : std_logic_vector(7 downto 0);
variable write_address : std_logic_vector(3 downto 0);
variable write_channel : std_logic_vector(2 downto 0);
begin
  --Assert Output
  amplitude_l <= output_l;
  amplitude_r <= output_r;

  --Simple test logic.
  pass <= not (output_l(14) and testm);

  --Test
  if(test = '0') then
    --All channels on, all wavemodes on op 1 to 1100
    --All volumes & pannings max
       testm := '1';
    for i in 0 to 7 loop
      channels(i).ch_start := '1';
      channels(i).opsettings(1).wavemode := "1111";
      channels(i).panning_l := "11111111";
      channels(i).panning_r := "11111111";
    end loop;
    --All channel frequencies set to
    --additively generate a square pulse
    channels(0).opsettings(1).frequency := "0000000000001010";
    channels(1).opsettings(1).frequency := "0000000000101000";
    channels(2).opsettings(1).frequency := "0000000001010000";
    channels(3).opsettings(1).frequency := "0000000001111000";
    channels(4).opsettings(1).frequency := "0000000010100000";
    channels(5).opsettings(1).frequency := "0000000011001000";
    channels(6).opsettings(1).frequency := "0000000011110000";
    channels(7).opsettings(1).frequency := "0000000100011000";
    channels(0).opsettings(1).volume := x"FF";
    channels(1).opsettings(1).volume := x"55";
    channels(2).opsettings(1).volume := x"24";
    channels(3).opsettings(1).volume := x"1C";
    channels(4).opsettings(1).volume := x"17";
    channels(5).opsettings(1).volume := x"11";
    channels(6).opsettings(1).volume := x"0F";
    channels(7).opsettings(1).volume := x"0D";

  --Reset
  elsif(reset='0') then
    mode := "00000";
    currentchannel := "000";
    write_req := '0';
       testm := '0';
    --Clear all channels
    for i in 0 to 7 loop
      channels(i) := zerochannel;
    end loop;

  --State Update
  elsif(clk'EVENT and clk='1') then
    ----------------------------------
    -- 0-1, Feed Channel Configuration
    ----------------------------------
    if (mode <= 1) then
      --Feed channel
      ch_settings_in <= channels(to_integer(currentchannel));
      --Increment mode
       mode := mode + 1;
           ------------------------------------------------
    -- 2, Accumulate output of last channel processed
    ------------------------------------------------
    elsif (mode = 2) then
      --Accumulate
      accumulator_l := accumulator_l + fetch_l;
      accumulator_r := accumulator_r + fetch_r;
      --Increment mode
       mode := mode + 1;
           -------------------------------
```

```
-- 3, Output Calculation Part 1
-------------------------------
elsif (mode = 3) then
  --If we're processing channel 0...
  --Convert Accumulators to unsigned,
  --Then store in output vars.
  if(currentchannel="000") then
    if( accumulator_l < 0) then
      output_l := std_logic_vector((not accumulator_l) + 1);
    else
      output_l := std_logic_vector(accumulator_l);
    end if;
    if( accumulator_r < 0) then
      output_r := std_logic_vector((not accumulator_r) + 1);
    else
      output_r := std_logic_vector(accumulator_r);
    end if;
  end if;
  --Increment mode
   mode := mode + 1;
       -------------------------------
-- 4, Output Calculation Part 2
-------------------------------
elsif (mode = 4) then
  --If we're processing channel 0...
  --then clear accumulators
  if(currentchannel="000") then
    accumulator_l := "000000000000000";
    accumulator_r := "000000000000000";
  end if;
  --Increment mode
   mode := mode + 1;
       --------------------------------
-- 5-22, Check for Writes on bus
--------------------------------
elsif (mode <= 22) then
  --If CE and WE are both low, the chip is
  --being written to, capture the address and data.
  --We'll give this a big cycle range as a poor
  --way to ensure we capture the write.
  if(ce='0' and we='0') then
    --Assert that we had a write request,
    --fetch the data
    write_req := '1';
    write_data := data;
    write_address := address;
  end if;
  --Increment mode
   mode := mode + 1;
------------------------------------
-- 23, Analyze and assert data write
------------------------------------
elsif (mode = 23) then
  --Was anything actually written?
  if(write_req='1') then
    --Written to a global register
    if(write_address(3 downto 2)="00") then
      --Channel Select
      if(write_address(1 downto 0)="00") then
        write_channel := write_data(2 downto 0);
      --Channel On
      elsif(write_address(1 downto 0)="01") then
        for i in 0 to 7 loop
          channels(i).ch_start := write_data(i);
        end loop;
      --Channel Off
      elsif(write_address(1 downto 0)="10") then
        for i in 0 to 7 loop
          channels(i).ch_stop := write_data(i);
        end loop;
      end if;
    --Written to a channel specific register
    else
```

```
                         --Channel Panning (L)
                         if(write_address="0110") then
                           channels(to_integer(unsigned(write_channel))).panning_l :=
write_data;
                         --Channel Panning (R)
                         elsif(write_address="0111") then
                           channels(to_integer(unsigned(write_channel))).panning_r :=
write_data;
                         --Operator 0 Frequency Low
                         elsif(write_address="1000") then

channels(to_integer(unsigned(write_channel))).opsettings(0).frequency :=

channels(to_integer(unsigned(write_channel))).opsettings(0).frequency(15 downto 8) &
write_data;
                         --Operator 0 Frequency High
                         elsif(write_address="1001") then

channels(to_integer(unsigned(write_channel))).opsettings(0).frequency :=
                             write_data &
channels(to_integer(unsigned(write_channel))).opsettings(0).frequency(7 downto 0);
                         --Operator 0 Volume
                         elsif(write_address="1010") then

channels(to_integer(unsigned(write_channel))).opsettings(0).volume := write_data;
                         --Operator 0 WaveMode
                         elsif(write_address="1011") then

channels(to_integer(unsigned(write_channel))).opsettings(0).wavemode := write_data(3
downto 0);
                         --Operator 1 Frequency Low
                         elsif(write_address="1100") then

channels(to_integer(unsigned(write_channel))).opsettings(1).frequency :=

channels(to_integer(unsigned(write_channel))).opsettings(1).frequency(15 downto 8) &
write_data;
                         --Operator 1 Frequency High
                         elsif(write_address="1101") then

channels(to_integer(unsigned(write_channel))).opsettings(1).frequency :=
                             write_data &
channels(to_integer(unsigned(write_channel))).opsettings(1).frequency(7 downto 0);
                         --Operator 1 Volume
                         elsif(write_address="1110") then

channels(to_integer(unsigned(write_channel))).opsettings(1).volume := write_data;
                         --Operator 1 WaveMode
                         elsif(write_address="1111") then

channels(to_integer(unsigned(write_channel))).opsettings(1).wavemode := write_data(3
downto 0);
                         --Invalid Address
                         else
                           --Do nothing
                         end if;
                       end if;
                     end if;
                     --Increment mode
                      mode := mode + 1;
                   ---------------------------------------------
                   -- 24, Analyze and assert channel start/stops
                   ---------------------------------------------
                   elsif (mode = 24) then
                     --Check for Channel on/off
                     for i in 0 to 7 loop
                       if(channels(i).ch_start='1') then
                         --Set Channel state to on, clear phase accumulators
                         channels(i).ch_start := '0';
                         channels(i).ch_on := '1';
                         channels(i).opsettings(0).phase_accumulator :=
"000000000000000000";
```

```
                         channels(i).opsettings(1).phase_accumulator :=
"000000000000000000";
                       elsif(channels(i).ch_stop='1') then
                         --Shut off channel
                         channels(i).ch_stop := '0';
                         channels(i).ch_on := '0';
                       end if;
                     end loop;
                     --Increment mode
                      mode := mode + 1;
                   -------------------------------------------
                   -- 25, Fetch Channel Configuration + Output
                   -------------------------------------------
                   elsif (mode = 25) then
                     --If a write request has been performed,
                     --discard the channel's updated data.
                     --This will make the channel skip a bit,
                     --but I think that's livable for the reduced complexity.
                     if(write_req='1') then
                       --Clear Write Request, ignore updated data.
                       write_req := '0';
                     else
                       channels(to_integer(currentchannel)) := ch_settings_out;
                     end if;
                     --Fetch output amplitudes if channel is enabled
                     if(channels(to_integer(currentchannel)).ch_on = '1') then
                       fetch_l := signed(ch_amplitude_l);
                       fetch_r := signed(ch_amplitude_r);
                     else
                       fetch_l := "00000000000";
                       fetch_r := "00000000000";
                     end if;
                     --Increment Channel, Wrapping if needed.
                     if (currentchannel = "111") then currentchannel := "000";
                     else currentchannel := currentchannel + 1;
                     end if;
                     --Back to state 0
          mode := "00000";
                   end if;
                 end if;
    end process update;
end architecture behavior;
```

```
------------------------------------------------
-- File: PD82Types.vhd
-- Author: Osman Celimli
-- Created: 4/12/2010
-- Modified: 4/20/2010
-- VHDL'93
-- Description: Typedefs used in the PD82
------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package PD82Types is
	--Typedef for Operator I/O
	type opconfig is record
		frequency : std_logic_vector(15 downto 0);
		wavemode : std_logic_vector(3 downto 0);
		phase_accumulator : std_logic_vector(17 downto 0);
		volume : std_logic_vector(7 downto 0);
	end record opconfig;

	--Array of two Operator Configurations
	type op2config is array(1 downto 0) of opconfig;

	--Entire channel configuration settings
	type chconfig is record
	  ch_start,ch_stop,ch_on : std_logic;
	  opsettings : op2config;
	  panning_l,panning_r : std_logic_vector(7 downto 0);
	end record chconfig;

	--Zeroed Channel (for reset)
	constant zerochannel : chconfig := (
	   --Channel State
	   ch_start => '0',
	   ch_stop => '0',
	   ch_on => '0',
	   --Panning
	   panning_l => "00000000",
	   panning_r => "00000000",
	   --Operators
	   opsettings => (
	     others => (frequency => "0000000000000000",
	      wavemode => "0000",
	      phase_accumulator => "000000000000000000",
	      volume => "00000000"))
	);

	--Array of eight channel configurations
	type chipconfig is array(7 downto 0) of chconfig;
end package PD82Types;
```

```
-----------------------------------------------
-- File: Channel.vhd
-- Entity: pdchannel
-- Architecture: behavior
-- Author: Osman Celimli
-- Created: 4/07/2010
-- Modified: 4/20/2010
-- VHDL'93
-- Description: A single channel structure
-- for the PD82 Synth.
-----------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
--Custom Type Pack
library work;
use work.PD82Types.all;
------------------------------------------------------
-- Entity for one Phase Distortion Channel (2 Operators)
------------------------------------------------------
entity pdchannel is
               --Ports
               Port( clk,reset,test : in std_logic;
                     --Signal out L/R
                     amplitude_l,amplitude_r : out std_logic_vector(10 downto 0);
                     --We'll leave the entire signal block
                     --as separate I/O, so ADSR is simple to add
                     --later on.
                     settings_in : in chconfig;
                     settings_out : out chconfig);
end entity pdchannel;
------------------------------------------------
-- Architecture for one Phase Distortion Channel
------------------------------------------------
architecture behavior of pdchannel is
  -------------------------
  --PD Operator Component
  -------------------------
  component pdoperator is
    Port( clk,reset,test : in std_logic;
               frequency : in std_logic_vector(15 downto 0);
               wavemode : in std_logic_vector(3 downto 0);
               phase_accumulator_in : in std_logic_vector(17 downto 0);
               phase_accumulator_out : out std_logic_vector(17 downto 0);
               volume : in std_logic_vector(7 downto 0);
               amplitude_out : out std_logic_vector(17 downto 0);
               feedin : in std_logic_vector(10 downto 0));
  end component pdoperator;
        ----------
        -- Signals
        ----------
        --For the operator
        signal op_clk,op_reset,op_test : std_logic;
        signal op_frequency : std_logic_vector(15 downto 0);
        signal op_wavemode : std_logic_vector(3 downto 0);
        signal op_phase_accumulator_in,op_phase_accumulator_out : std_logic_vector(17
downto 0);
        signal op_volume : std_logic_vector(7 downto 0);
        signal op_amplitude_out : std_logic_vector(17 downto 0);
        signal op_feedin : std_logic_vector(10 downto 0);
        --------------------------------
        -- Architecture code starts here
        --------------------------------
        begin
               --Map in Operator
               operatormap: component pdoperator port map(op_clk, op_reset, op_test,

op_frequency,op_wavemode,

op_phase_accumulator_in,op_phase_accumulator_out,
                                                                     op_volume,
                                                                     op_amplitude_out,
                                                                     op_feedin);
```

```
            --Also clock it, plus reroute reset.
            op_clk <= clk;
            op_reset <= reset;
            op_test <= test;

     chupdate : process (clk,reset,test)
            --Internal Vars
            variable mode : unsigned(4 downto 0);
            variable int_config : chconfig;
            variable int_feedback : std_logic_vector(10 downto 0);
            variable int_amplitude_l,int_amplitude_r : std_logic_vector(19 downto 0);
            begin
              --Test
              if(test='0') then
                --Do nothing
              --Reset
              elsif(reset='0') then
                mode := "00000";

              --State Update
              elsif(clk'EVENT and clk='1') then
                ----------------------------
                -- 0-1, Fetch configurations
                ----------------------------
                if (mode < 2) then
                  int_config := settings_in;
                  --Increment mode
                  mode := mode + 1;
                -----------------------
                -- 2-3, Feed Operator 0
                -----------------------
                elsif (mode < 4) then
                  --Copy in operator 0 settings
                  --No feedin.
                  op_frequency <= int_config.opsettings(0).frequency;
                  op_wavemode <= int_config.opsettings(0).wavemode;
                  op_phase_accumulator_in <=
int_config.opsettings(0).phase_accumulator;
                  op_volume <= int_config.opsettings(0).volume;
                  op_feedin <= "00000000000";
                  --Increment mode
                  mode := mode + 1;
                ---------------------------
                -- 4-7, Wait for Operator 0
                ---------------------------
                elsif (mode < 8) then
                  --Increment mode
          mode := mode + 1;
        ----------------------------------
        -- 8-9, Fetch data from Operator 0
        ----------------------------------
        elsif (mode < 10) then
          --Copy back changed settings,
          --along with feedback for operator 1.
          int_config.opsettings(0).phase_accumulator := op_phase_accumulator_out;
          int_feedback := op_amplitude_out(17) & op_amplitude_out(17)
                           & op_amplitude_out(17 downto 9);
          --Increment mode
          mode := mode + 1;
          -----------------------
         -- 10-14, Wait a bit...
         -----------------------
                elsif (mode < 15) then
                  --Increment mode
                   mode := mode + 1;
                -------------------------
                -- 15-16, Feed Operator 1
                -------------------------
                elsif (mode < 17) then
                  --Copy in operator 1 settings
                  --Feedin from operator 0's output
                  op_frequency <= int_config.opsettings(1).frequency;
                  op_wavemode <= int_config.opsettings(1).wavemode;
```

```
                     op_phase_accumulator_in <=
int_config.opsettings(1).phase_accumulator;
                     op_volume <= int_config.opsettings(1).volume;
                     op_feedin <= int_feedback;
                     --Increment mode
                     mode := mode + 1;
                   -----------------------------
                   -- 17-20, Wait for Operator 1
                   -----------------------------
                   elsif (mode < 21) then
                     --Increment mode
          mode := mode + 1;
        ------------------------------------
        -- 21-22, Fetch data from Operator 1
        ------------------------------------
        elsif (mode < 23) then
          --Copy back changed settings,
          --along with feedback for operator 1.
          int_config.opsettings(1).phase_accumulator := op_phase_accumulator_out;
          int_feedback := op_amplitude_out(17) & op_amplitude_out(17)
                             & op_amplitude_out(17 downto 9);
          --Increment mode
          mode := mode + 1;
        -----------------------------------
        -- 23, Calculate L and R amplitudes
        -----------------------------------
        elsif (mode = 23) then
          int_amplitude_l := std_logic_vector(signed(int_feedback)
                               * signed("0" & int_config.panning_l));
          int_amplitude_r := std_logic_vector(signed(int_feedback)
                               * signed("0" & int_config.panning_r));
          --Increment mode
          mode := mode + 1;
        --------------------------------
        -- 24-25, Return operator output
        --------------------------------
        elsif (mode < 26) then
          settings_out <= int_config;
          amplitude_l <= int_amplitude_l(18 downto 8);
          amplitude_r <= int_amplitude_r(18 downto 8);
          --Back to state 0
          if (mode = 24) then mode := mode + 1;
          else mode := "00000";
          end if;
                   end if;
                 end if;
               end process chupdate;
end architecture;
```

```
-------------------------------------------------
-- File: Operator.vhd
-- Entity: pdoperator
-- Architecture: behavior
-- Author: Osman Celimli
-- Created: 4/01/2010
-- Modified: 4/07/2010
-- VHDL'93
-- Description: A single Phase Distortion operator
-- for the PD82 Synth.
-------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-------------------------------------------
-- Entity for the Phase Distortion Operator
-------------------------------------------
entity pdoperator is
               Port( clk,reset,test : in std_logic;
                       ----------------------------------------
                       --Frequency is 16-bits and formatted as:
                       -- 15-8: Phase Index Increment
                       -- 7-0: Phase Increment Decimal
                       ----------------------------------------
                       frequency : in std_logic_vector(15 downto 0);
                       -----------------------------------------
                       --WaveMode is a 4-Bit value formatted as:
                       -- 3: Phase 0-255 Enable
                       -- 2: Phase 256-511 Enable
                       -- 1: Phase 512-767 Enable
                       -- 0: Phase 768-1024 Enable
                       -----------------------------------------
                       wavemode : in std_logic_vector(3 downto 0);
                       ------------------------------------------------
                       --The previous and next phase accumulator value:
                       -- 17-8: Phase Index Increment
                       -- 7-0: Phase Increment Decimal
                       ------------------------------------------------
                       phase_accumulator_in : in std_logic_vector(17 downto 0);
                       phase_accumulator_out : out std_logic_vector(17 downto 0);
                       -------------------------------------------------------------------
                       --Operator volume is a flat 8-bit value, 0 is "off," $FF is loudest
                       --Amplitude is a 18-bit signed value
                       -------------------------------------------------------------------
                       volume : in std_logic_vector(7 downto 0);
                       amplitude_out : out std_logic_vector(17 downto 0);
                       -------------------------------------------------------------------
                       -- Feedin is from last operator, zeroed if there is no previous op.
                       -------------------------------------------------------------------
                       feedin : in std_logic_vector(10 downto 0));
end entity pdoperator;

-------------------------------------------------
-- Architecture for the Phase Distortion Operator
-------------------------------------------------
architecture behavior of pdoperator is
  --------------------------
  --Sine Table ROM component
  --------------------------
  component sinetable is
    Port(clk : in std_logic;
         mode : in std_logic_vector(3 downto 0);
              address : in std_logic_vector(9 downto 0);
              data_out : out std_logic_vector(8 downto 0));
  end component sinetable;
       ----------
       -- Signals
       ----------
       signal sine_address : std_logic_vector(9 downto 0);
       signal sine_data : std_logic_vector(8 downto 0);
       signal sine_mode : std_logic_vector(3 downto 0);
       signal intclk : std_logic;
       --------------------------------
```

```
        -- Architecture code starts here
        --------------------------------
        begin
                --Map in Sine ROM
                sinerommap: component sinetable port map(intclk, sine_mode, sine_address,
sine_data);
                --Also clock it and reroute waveform mode.
                sine_mode <= wavemode;
                intclk <= clk;

                opupdate : process (clk,reset,test)
                --Internal Vars
                variable mode: unsigned(3 downto 0);
                variable int_phase_accumulator: std_logic_vector(17 downto 0);
                variable int_phase_address : std_logic_vector(18 downto 0);
                variable int_frequency: std_logic_vector(15 downto 0);
                variable int_volume: std_logic_vector(7 downto 0);
                variable int_sineread : std_logic_vector(8 downto 0);
                variable int_feedin: std_logic_vector(10 downto 0);
                begin
                  --Test (Hold state)
                  if(test='0') then
                        --Do nothing
                  --Reset (Mode=0)
                  elsif(reset='0') then
                    mode := "0000";
                  --Process Channels
                        elsif(clk'EVENT and clk='1') then
                                --------------------
                                --Mode 0-3, Data fetch
                                --------------------
                                if (mode < 4) then
                                        --Copy input feeds to internal regs
                                        int_phase_accumulator := phase_accumulator_in;
                                        int_frequency := frequency;
                                        int_volume := volume;
                                        int_feedin := feedin;
                                        --Increment State
                                        mode := mode + 1;
                                ------------------------------------------------------------
                                --Mode 4, Phase Update & ROM Address Calculation
                                ------------------------------------------------------------
                                elsif (mode = 4) then
                                        --Phase Accumulator updates
                                        int_phase_accumulator :=
std_logic_vector(unsigned(int_phase_accumulator) + unsigned(int_frequency));
                                        --ROM Address Calculation
                                        --Address = (Phase Accumulator + Feedin=Last
Operator Amp<<7)(17 downto 7)
                                        int_phase_address := (std_logic_vector(signed("0" &
int_phase_accumulator) + signed(int_feedin & "0000000")));
                                        --Increment State
                                        mode := mode + 1;
                                ---------------------
                                --Mode 5, Sine Strobe
                                ---------------------
                                elsif (mode = 5) then
                                  --Strobe ROM
                                        sine_address <= int_phase_address(17 downto 8);
                                        --Increment State
                                        mode := mode + 1;
                                ---------------------
                                --Mode 6, Sine Fetch
                                ---------------------
                                elsif (mode = 6) then
                                  --Fetch Data
                                        int_sineread := sine_data;
                                        --Increment State
                                        mode := mode + 1;
                                -------------------------------------------------
                                --Mode 8-12 Resgister store, Amplitude Calculation
                                -------------------------------------------------
                                elsif (mode < 13) then
```

```
                                --Output Scaled Amplitude (sinedata*volume)(Bits 17-9
valid)
                                amplitude_out <= std_logic_vector(signed(int_sineread) *
signed("0" & int_volume));
                                --Write Back updated phase values
                                phase_accumulator_out <= int_phase_accumulator;
                                --Wait, then return to state 0.
                                if (mode < 12) then mode := mode + 1;
                                else mode := "0000";
                                end if;
                              end if;
                      end if;
              end process opupdate;
end architecture;
```

```
------------------------------------------------
-- File: SineTable.vhd
-- Entity: sinetable
-- Architecture: behavior
-- Author: Osman Celimli
-- Created: 4/01/2010
-- Modified: 4/01/2010
-- VHDL'93
-- Description: The sine lookup table used in
-- the PD82. The table is 10-bit indexed into its
-- 256 entries, which are inverted as needed based
-- upon the position in the sinusoid. Each entry
-- is an 8-bit unscaled sine sample which is adjusted
-- and signed before output.
------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

----------------------------------
-- Entity for the Sine Lookup Table
----------------------------------
entity sinetable is
	Port( clk : in std_logic;
		mode : in std_logic_vector(3 downto 0);
		address : in std_logic_vector(9 downto 0);
		data_out : out std_logic_vector(8 downto 0));
end entity sinetable;

----------------------------------------
-- Architecture for the Sine Lookup Table
----------------------------------------
architecture behavior of sinetable is
	--Typedef for Sine ROM.
	type sinerom is array (255 downto 0) of std_logic_vector (7 downto 0);
	--------------------
	--The ROM data itself
	--------------------
	constant sinedat : sinerom := (
		0 => x"00", 1 => x"01", 2 => x"03", 3 => x"04",
		4 => x"06", 5 => x"07", 6 => x"09", 7 => x"0a",
		8 => x"0c", 9 => x"0e", 10 => x"0f", 11 => x"11",
		12 => x"12", 13 => x"14", 14 => x"15", 15 => x"17",
		16 => x"19", 17 => x"1a", 18 => x"1c", 19 => x"1d",
		20 => x"1f", 21 => x"20", 22 => x"22", 23 => x"24",
		24 => x"25", 25 => x"27", 26 => x"28", 27 => x"2a",
		28 => x"2b", 29 => x"2d", 30 => x"2e", 31 => x"30",
		32 => x"31", 33 => x"33", 34 => x"35", 35 => x"36",
		36 => x"38", 37 => x"39", 38 => x"3b", 39 => x"3c",
		40 => x"3e", 41 => x"3f", 42 => x"41", 43 => x"42",
		44 => x"44", 45 => x"45", 46 => x"47", 47 => x"48",
		48 => x"4a", 49 => x"4b", 50 => x"4d", 51 => x"4e",
		52 => x"50", 53 => x"51", 54 => x"53", 55 => x"54",
		56 => x"56", 57 => x"57", 58 => x"59", 59 => x"5a",
		60 => x"5c", 61 => x"5d", 62 => x"5f", 63 => x"60",
		64 => x"61", 65 => x"63", 66 => x"64", 67 => x"66",
		68 => x"67", 69 => x"69", 70 => x"6a", 71 => x"6c",
		72 => x"6d", 73 => x"6e", 74 => x"70", 75 => x"71",
		76 => x"73", 77 => x"74", 78 => x"75", 79 => x"77",
		80 => x"78", 81 => x"7a", 82 => x"7b", 83 => x"7c",
		84 => x"7e", 85 => x"7f", 86 => x"80", 87 => x"82",
		88 => x"83", 89 => x"84", 90 => x"86", 91 => x"87",
		92 => x"88", 93 => x"8a", 94 => x"8b", 95 => x"8c",
		96 => x"8e", 97 => x"8f", 98 => x"90", 99 => x"92",
		100 => x"93", 101 => x"94", 102 => x"95", 103 => x"97",
		104 => x"98", 105 => x"99", 106 => x"9a", 107 => x"9c",
		108 => x"9d", 109 => x"9e", 110 => x"9f", 111 => x"a1",
		112 => x"a2", 113 => x"a3", 114 => x"a4", 115 => x"a5",
		116 => x"a7", 117 => x"a8", 118 => x"a9", 119 => x"aa",
		120 => x"ab", 121 => x"ac", 122 => x"ae", 123 => x"af",
		124 => x"b0", 125 => x"b1", 126 => x"b2", 127 => x"b3",
		128 => x"b4", 129 => x"b5", 130 => x"b7", 131 => x"b8",
		132 => x"b9", 133 => x"ba", 134 => x"bb", 135 => x"bc",
```

```
            136 => x"bd", 137 => x"be", 138 => x"bf", 139 => x"c0",
            140 => x"c1", 141 => x"c2", 142 => x"c3", 143 => x"c4",
            144 => x"c5", 145 => x"c6", 146 => x"c7", 147 => x"c8",
            148 => x"c9", 149 => x"ca", 150 => x"cb", 151 => x"cc",
            152 => x"cd", 153 => x"ce", 154 => x"cf", 155 => x"d0",
            156 => x"d1", 157 => x"d1", 158 => x"d2", 159 => x"d3",
            160 => x"d4", 161 => x"d5", 162 => x"d6", 163 => x"d7",
            164 => x"d7", 165 => x"d8", 166 => x"d9", 167 => x"da",
            168 => x"db", 169 => x"dc", 170 => x"dc", 171 => x"dd",
            172 => x"de", 173 => x"df", 174 => x"df", 175 => x"e0",
            176 => x"e1", 177 => x"e2", 178 => x"e2", 179 => x"e3",
            180 => x"e4", 181 => x"e4", 182 => x"e5", 183 => x"e6",
            184 => x"e6", 185 => x"e7", 186 => x"e8", 187 => x"e8",
            188 => x"e9", 189 => x"ea", 190 => x"ea", 191 => x"eb",
            192 => x"ec", 193 => x"ec", 194 => x"ed", 195 => x"ed",
            196 => x"ee", 197 => x"ee", 198 => x"ef", 199 => x"ef",
            200 => x"f0", 201 => x"f1", 202 => x"f1", 203 => x"f2",
            204 => x"f2", 205 => x"f3", 206 => x"f3", 207 => x"f3",
            208 => x"f4", 209 => x"f4", 210 => x"f5", 211 => x"f5",
            212 => x"f6", 213 => x"f6", 214 => x"f6", 215 => x"f7",
            216 => x"f7", 217 => x"f8", 218 => x"f8", 219 => x"f8",
            220 => x"f9", 221 => x"f9", 222 => x"f9", 223 => x"fa",
            224 => x"fa", 225 => x"fa", 226 => x"fa", 227 => x"fb",
            228 => x"fb", 229 => x"fb", 230 => x"fb", 231 => x"fc",
            232 => x"fc", 233 => x"fc", 234 => x"fc", 235 => x"fd",
            236 => x"fd", 237 => x"fd", 238 => x"fd", 239 => x"fd",
            240 => x"fd", 241 => x"fe", 242 => x"fe", 243 => x"fe",
            244 => x"fe", 245 => x"fe", 246 => x"fe", 247 => x"fe",
            248 => x"fe", 249 => x"fe", 250 => x"fe", 251 => x"fe",
            252 => x"fe", 253 => x"fe", 254 => x"fe", 255 => x"ff");
    --------------------------------
    -- Architecture code starts here
    --------------------------------
    begin
        romaccess : process (clk)
        variable internal_address: integer;
        begin
            if (clk'EVENT and clk='1') then
                if (address(9 downto 8)= "00") then
                    --Phase is from 0 to 255,
                    --No index or output conversion
                    if mode(3) = '0' then
                        data_out <= "000000000";
                    else
                        internal_address :=
to_integer(unsigned(address(7 downto 0)));
                        data_out <= "0" & sinedat(internal_address);
                    end if;
                elsif (address(9 downto 8)= "01") then
                    --Phase is from 256 to 511
                    --Index is inverted (xor)
                    if mode(2) = '0' then
                      data_out <= "000000000";
                    else
                      internal_address := to_integer(unsigned(address(7
downto 0) xor x"ff"));
                      data_out <= "0" & sinedat(internal_address);
                    end if;
                elsif (address(9 downto 8)= "10") then
                    --Phase is from 512 to 767
                    --Index is inverted and appended with "1" for
negation
                    if mode(1) = '0' then
                      data_out <= "000000000";
                    else
                      internal_address := to_integer(unsigned(address(7
downto 0) xor x"ff"));
                      data_out <= "1" & sinedat(internal_address);
                    end if;
                elsif (address(9 downto 8)= "11") then
                    --Phase is from 768 to 1024
                    --Index is uninverted and appended with "1" for
negation
```

```
                                      if mode(0) = '0' then
                                        data_out <= "000000000";
                                      else
                                        internal_address := to_integer(unsigned(address(7
downto 0)));
                                        data_out <= "1" & sinedat(internal_address);
                                      end if;
                              end if;
                      end if;
              end process romaccess;
end architecture behavior;
```