Ups and Downs: Building a Digital Pitch Shifter

Michael Salib

November 17, 2001

Contents

1 Overview

I designed and implemented a digital pitch shifter (DPS). This device produces an output audio signal that is a higher or lower pitched version of the input audio signal. The amount of shifting is adjustable by the user. The device can output an exact copy of the input signal, a shifted version, or a combination of the shifted version and the original signal. The combination mode can be used to produce interesting audio effects.

The DPS works by digitizing the input audio signal and manipulating it in the digitial domain. The DPS converts the digital pitch shifted beck into the analog domain before outputting it. Shifting is accomplished by stepping through a list of samples of the digitzied input. By adjusting the fractional step size, we can change the rate at which samples of the input signal are played back. For step values less than one, this process corresponds to stretching the signal in time, or decreasing the pitch. For step values greater than one, this process corresponds to compressing the signal in time, which is equivalent to increasing the pitch.

Although this pitch shifting algorithm is simple to implement, that simplicity comes at the cost of reduced output quality. Because the DPS must work in real time, it does not have access to the entirety of the input signal. Consequently, it uses buffers to store small pieces of the input signal. When stepping past the end of a chunk (which always happens for stepping values greater than one), the DPS wraps around to the beggining of the chunk. The effect of this algorithm is to fill the empty space created by compressing a piece of the signal in time with copies of itself. As a result of this copying, the quality of the shifted signal is greatly reduced. However, these effects are negligible compared to the quality degradation associated with sampling the input signal at low rates using low precision sampling techniques.

The DPS design I describe here makes use of an analog to digital converter (ADC) and digital to analog converter (DAC) that operate with 8 bits of data per sample. In addition, it uses a 64 kilobyte RAM for storing input and output data chunks. The remainder of the system consists of componants I designed and then implemented using complex programmable logic devices programmed in VHDL. These componants include modules that synchronize external inputs, generate appropriate audio timing signals, and accumulate the input and shifted signals. Finally, I designed and implemented a microprogrammed control unit (MCU) to manage control and status signals from the other componants along with MCU programs for testing and running the DPS. The resulting system works properly and meets all of its specifications.

In the course of implementing the DPS, I created a set of software tools to convert MCU assembler programs into a form more amenable to CPLD synthesis. The first program converts ROM images into VHDL source files, allowing easy embedding of program code into an MCU implemented in a CPLD. This approach was what I actually used during the construction of the DPS. The second program converts MCU assembler programs into finite state machines implemented entirely in VHDL. The use of these tools greatly simplified the consturction process.

2 Description

The DPS consists of several componants connected with a shared eight bit data bus. These include the DAC, the ADC, the storage unit, and the signal accumulator. In addition, the DPS also makes use of an MCU to control the other componants and arbitrate who should be reading from or writing to the shared bus at any time.

2.1 Synchronizer

The synchronizer simply synchronizes user inputs to ensure the system only sees user input that does not transition during system clock changes. This prevents metastable situations. The synchronizer also converts the pitch up and down inputs from levels into pulses. That ensures that when the user presses on the pitch up or pitch down button, the system sees only one pulse on that input line.

2.2 Timing Unit

The Timing Unit simply divides down the system clock of 1.8432 MHz to either 9600 Hz or 19.2 kHz depending on the synchronized frequency selection input set by the user. It consists of a simple counter whose initial value depends on the frequency selected.

2.3 Storage Unit

The storage unit is the most complex componants of the system. It provides the addressing needed by the RAM. It maintains two seperate counters for RAM addresses; one is used for sampling input audio signals while the other is used for shifting through chunks of previously sampled data in preperation for output.

2.4 Signal Accumulator

The signal accumulator simply combines pairs of samples as directed by the MCU. Because these samples are both 8 bits wide and the output must also be 8 bits wide, the input data is shifted right by one bit when adding two signals together. This corresponds to taking the average of the two signals rather than their direct sum.

2.5 Microprogrammed Control Unit

The MCU is the heart of the system. It consists of a 16 bit ROM whose address is sourced by either an address counter or a portion of the previous ROM output depending on the results of a condition selector. This selector is used to support conditional and unconditional branches.

2.5.1 Instruction Format

I used the suggested instruction format. The only changes I made were in choosing my own conditional jump inputs and assertion signals. It is located in Appendix B.1.

2.5.2 MCU Programs

I wrote two main MCU programs. The first was a test program used to determine if the system could sample audio input data and play the sample back immediately without further processing. The source code for this program is located in Appendix B.3.

The second program was used for actually running the system. It is available in Appendix B.2.

2.6 MCU Implementation Techniques

To avoid excessive wiring, I decided to implement my MCU in a CPLD. However, the CPLDs in the lab kit don't have enough free IO pins to support both the ROM data and address inputs in addition to the other IO needed to implement the DPS. The solution I came up with was to write a small compiler that would compile the output of the assembler into VHDL source. That way, instead of burning my assembler code into a ROM, I could burn it into a CPLD. This MCU implementation is exactly like the one described in the lab handout; the only difference is that the ROM is implemented in VHDL.

I developed two compilers. The first one, compileToCPLD, is what I used for this lab. The second one, asm2fsm, is a more ambitious experimental project. Instead of converting assembler output into a VHDL ROM, it converts assembler output into a Finite State Machine written in VHDL. In theory, this should give the synthesis tools much more leeway to optimize the resulting system since the VHDL compiler can make state assignements in an optimal fashion. In practice, this compiler produced VHDL code that compiled to much more resource hungry designs.

3 Testing and Debugging

B.3 I grealy simplified debugging by reducing the wiring effort needed with my alternative MCU implementation. The debugging I did perform was largely composed of watching program adderss and contol signals on the logic analyzer. Besides the expected minor bugs, I did have serious problems with bus contention and RAM addressing. Specifically, I had difficulty keeping RAM address lines stable during multiple consecutive writes.

A VHDL Source Code Listings

A.1 Toplevel Source

```
-- note: the sampling_frequency_select listed here does nothing
```

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

use work.storageUnit;
use work.synchronizer;
use work.signalAccumulator;
use work.mcu;
use work.timingUnit2;

entity cpld is
  port (
    -- synchronizer inputs
    clkx, a_pitch_up, a_pitch_down           : in  std_logic;
    a_pass_shifted, a_pass_original          : in  std_logic;
    a_not_reset                              : in  std_logic;
    a_buffer_size                            : in  unsigned(3 downto 0);
    -- to and from the MCU that now lives here
    adc_not_ready, freq_selectx : in std_logic;
    ram_oe, ram_we, adc_enable, adc_sample, dac_enable : out std_logic;

    -- storage unit control inputs
--    st_count, st_clear_samp, st_clear_shift   : in  std_logic;
--    st_swap_buf, st_shift_buf, st_shift_count : in  std_logic;
    -- accumulator control inputs
--    sa_loadadd, sa_clear, sa_oe               : in std_logic;
    -- outputs
--    fullx, pass_originalx, pass_shiftedx      : out std_logic;
    data_bus                                 : inout unsigned(7 downto 0);
    ram_addressx                             : out unsigned(11 downto 0));

  attribute pin_avoid of cpld                : entity is
    "11 21 22 32 42 43 44 53 63 64 74 83"&  -- Vdd, Gnd, VPP
    " 13 "&  -- This is IO-9. Can screw up the clock of C1. Be
                          -- careful when using this.
    " 23 62 65 "&
    --" 71 "&   --must be grounded for K1 interface

    -- this line lists all the logic analyzer connections...
    --"3 4 5 6 7 8 9 10 15 16 17 18 67 68 69 70 71 75 76 77 78 79 80 81 82"&

    " 14 35 41 51 72 "; -- Used by Programmer. No external connection.

  attribute pin_numbers of cpld :entity is
```

```
    -- first buf_size and ram_addresses
    "a_buffer_size(0):24 a_buffer_size(1):25 "&
    "a_buffer_size(2):26 a_buffer_size(3):27 "&
    "ram_addressx(0):28 ram_addressx(1):29 ram_addressx(2):30 ram_addressx(3):31 "&
    "ram_addressx(4):33 ram_addressx(5):34 ram_addressx(6):36 ram_addressx(7):37 "&
    "ram_addressx(8):38 ram_addressx(9):39 ram_addressx(10):40 ram_addressx(11):45 "&
    -- now storage unit control inputs and status output
    --"st_count:3 st_clear_samp:4 st_clear_shift:5 st_swap_buf:6 "&
    --"st_shift_buf:7 st_shift_count:8 fullx:9 "&
    -- finally, asynchronous inputs (a_not_reset is in a funny place)
    "a_pitch_up:46 a_pitch_down:47 a_pass_shifted:48 a_pass_original:49 "&
    "a_not_reset:52 "&
    --"sa_loadadd:57 sa_clear:55 sa_oe:56 "&
    --"pass_shiftedx:60 pass_originalx:61  " &
    "data_bus(0):75 data_bus(1):76 data_bus(2):77 data_bus(3):78 "&
    "data_bus(4):79 data_bus(5):80 data_bus(6):81 data_bus(7):82 "&
    "adc_not_ready:55 adc_enable:56 adc_sample:57 dac_enable:60  "&
    "ram_oe:66 ram_we:54 freq_selectx:61";
end cpld;

architecture x of cpld is
  signal not_resetx, pitch_upx, pitch_downx, resetx : std_logic;
  signal pass_shiftedx, pass_originalx, fullx : std_logic;
  signal buffer_sizex : unsigned(3 downto 0);
  signal foo : unsigned(11 downto 0);

  signal st_count, st_clear_samp, st_clear_shift : std_logic;
  signal st_swap_buf, st_shift_buf, st_shift_count : std_logic;
  signal sa_loadadd, sa_clear, sa_oe               : std_logic;

  signal timing_sample : std_logic;
begin  -- x

  resetx <= not(not_resetx);

  mcu_in_a_box : mcu port map (
    clk => clkx,
    reset => resetx,
    st_full => fullx,
    adc_not_ready => adc_not_ready,     --from adc
    timing_sample => timing_sample,     --from timing unit outside
    pass_shifted => pass_shiftedx,
    pass_original => pass_originalx,
```

```
  ram_oe => ram_oe,                   -- to outside
  ram_we => ram_we,                   -- to outside
  adc_enable => adc_enable,           -- to outside
  adc_sample => adc_sample,           -- to outside
  dac_enable => dac_enable,           -- to outside

  st_count => st_count,
  st_clear_samp => st_clear_samp,
  st_clear_shift => st_clear_shift,
  st_swap_buf => st_swap_buf,
  st_shift_buf => st_shift_buf,
  st_shift_count => st_shift_count,
  sa_loadadd => sa_loadadd,
  sa_clear => sa_clear,
  sa_oe => sa_oe);

tu2 : timingUnit2 port map (
  clk         => clkx,
  freq_select => freq_selectx,
  sampling    => timing_sample);

su   : storageUnit port map (
  clk                  => clkx,
  not_reset            => not_resetx,
  pitch_up             => pitch_upx,
  pitch_down           => pitch_downx,
  count                => st_count,
  clear_samp           => st_clear_samp,
  clear_shift          => st_clear_shift,
  swap_buf             => st_swap_buf,
  shift_buf            => st_shift_buf,
  shift_count          => st_shift_count,
  buffer_size          => buffer_sizex,
  full                 => fullx,
  ram_address          => ram_addressx);

sync : synchronizer port map (
  clk                  => clkx,
  a_pitch_up           => a_pitch_up,
  a_pitch_down         => a_pitch_down,
  a_pass_shifted       => a_pass_shifted,
  a_pass_original      => a_pass_original,
  a_not_reset          => a_not_reset,
  a_buffer_size        => a_buffer_size,
```

```
    pitch_up             => pitch_upx,
    pitch_down           => pitch_downx,
    pass_shifted         => pass_shiftedx,
    pass_original        => pass_originalx,
    not_reset            => not_resetx,
    buffer_size          => buffer_sizex);

  sigacc : signalAccumulator port map (
    clk        => clkx,
    sa_loadadd => sa_loadadd,
    sa_clear   => sa_clear,
    sa_oe      => sa_oe,
    data_bus   => data_bus);
end x;
```

A.2 MCU

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

library work;
use work.cpldROM;

entity mcu is
  port (
    clk, reset                                            : in  std_logic;
    st_full, adc_not_ready, timing_sample                 : in  std_logic;
    pass_shifted, pass_original                           : in  std_logic;
    -- inverted outputs
    ram_oe, ram_we, adc_enable, adc_sample, dac_enable        : out std_logic;
    -- non inverted outputs
    st_count, st_clear_samp, st_clear_shift, st_swap_buf      : out std_logic;
    st_shift_buf, st_shift_count, sa_loadadd, sa_clear, sa_oe : out std_logic);
end mcu;

architecture x of mcu is
  signal currentAddress, nextAddress, jmp_address : unsigned(7 downto 0);
  signal ROMdata                                  : unsigned(15 downto 0);
  signal mcu_assert, jmp_or_not                   : std_logic;
```

```
  signal condition_selector                      : unsigned(2 downto 0);
  -- assertions from the PROM
  signal clrleds, l0, l1, l2, l3, l4, l5, l6, l7  : std_logic;
begin  -- x

  clk_proc: process(clk)
  begin  -- process
    if rising_edge(clk) then
      if reset = '1' then
        currentAddress <= "00000000";
      else
        currentAddress <= nextAddress;
      end if;
    end if;
  end process;

  rom : cpldROM port map (
    address => currentAddress,
    data    => ROMdata);

  mcu_assert <= ROMdata(15);
  condition_selector <= ROMdata(14 downto 12);
  jmp_address <= ROMdata(7 downto 0);

  with condition_selector select
    jmp_or_not <=
    st_full  when "000",
    adc_not_ready  when "001",
    timing_sample  when "010",
    pass_shifted  when "011",
    pass_original  when "100",
    '1' when others;

  comb: process(mcu_assert, jmp_or_not, currentAddress, jmp_address)
  begin
    if mcu_assert = '0' then
      if jmp_or_not = '1' then
        nextAddress <= jmp_address;
      else
        nextAddress <= currentAddress + 1;
      end if;
    else
      nextAddress <= currentAddress + 1;
    end if;
```

```
  end process;


  ff: process(clk)
    begin
      if rising_edge(clk) then
        if mcu_assert = '1' then
          ram_oe <= not(ROMdata(0));
          ram_we <= not(ROMdata(1));
          adc_enable <= not(ROMdata(2));
          adc_sample <= not(ROMdata(3));
          dac_enable <= not(ROMdata(4));

          st_count <= ROMdata(5);
          st_clear_samp <= ROMdata(6);
          st_clear_shift <= ROMdata(7);
          st_swap_buf <= ROMdata(8);
          st_shift_buf <= ROMdata(9);
          st_shift_count <= ROMdata(10);
          sa_loadadd <= ROMdata(11);
          sa_clear <= ROMdata(12);
          sa_oe <= ROMdata(13);
        end if;
      end if;
    end process ff;


end x;
```

A.3 Timing Unit

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

entity timingUnit2 is
  port (
    clk, freq_select : in  std_logic;
    sampling         : out std_logic);
end timingUnit2;


-- 96 = 1100000
```

```
-- 192 = 11000000

architecture x of timingUnit2 is
  signal count, starting_count_value : unsigned(7 downto 0);
begin  -- x

  starting_count_value <= "01100000" when freq_select = '0'
                          else "11000000";

  countdown: process(clk, count, starting_count_value)
    begin
      if rising_edge(clk) then
        if count = "00000000" then
          count <= starting_count_value;
          sampling <= '1';
        else
          count <= count - 1;
          sampling <= '0';
        end if;
      end if;
    end process countdown;

end x;
```

A.4 Synchronizer

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

-- this should not go in the MCU CPLD. If it does,
--  it needs a clk enable input.

entity synchronizer is
  port (
    -- a_foo means the asynchronous version of foo
    pitch_up, pitch_down                 : out std_logic;
    pass_shifted, pass_original          : out std_logic;
    not_reset      : out std_logic;
    buffer_size                          : out unsigned(3 downto 0);
    clk, a_pitch_up, a_pitch_down        : in  std_logic;
    a_pass_shifted, a_pass_original      : in  std_logic;
    a_not_reset : in  std_logic;
    a_buffer_size                        : in  unsigned(3 downto 0));
```

```
end synchronizer;

architecture x of synchronizer is
  signal u, v, x, y : std_logic;
begin  -- x
  sync: process(clk)
    begin
      if rising_edge(clk) then
        -- synchronize these signals
        pass_original <= a_pass_original;
        pass_shifted <= a_pass_shifted;
        not_reset <= a_not_reset;
        --sampling_frequency_select <= a_sampling_frequency_select;
        buffer_size <= a_buffer_size;

        -- pulsify a_pitch_up/down
        u <= a_pitch_up;
        v <= u;

        x <= a_pitch_down;
        y <= x;
      end if;
    end process sync;

  pitch_up <= not(v) and u;
  pitch_down <= not(y) and x;

end x;
```

A.5 Storage Unit

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

entity storageUnit is
  port (
    clk, not_reset                  : in  std_logic;
    pitch_up, pitch_down            : in  std_logic;
    clear_shift, clear_samp, count  : in  std_logic;
    shift_buf, shift_count, swap_buf : in  std_logic;
    buffer_size                     : in  unsigned(3 downto 0);
    full                            : out std_logic;
    pitch_multiplier_out            : out unsigned(4 downto 0);
```

```
    ram_address                      : out unsigned(11 downto 0));
end storageUnit;

architecture x of storageUnit is
  signal buf_sel, internal_full                    : std_logic;
  signal pitch_multiplier                          : unsigned(4 downto 0);
  signal internal_ram_address                      : unsigned(10 downto 0);
  -- counters
  signal sampling_counter                          : unsigned(10 downto 0);
  signal shift_counter, extended_pitch_multiplier : unsigned(16 downto 0);
begin  -- x
  pitch_multiplier_out <= pitch_multiplier(4 downto 0);

  pitch_mult: process(clk, pitch_up, pitch_down, not_reset)
    begin
      if rising_edge(clk) then
        if not_reset = '0' then
          pitch_multiplier <= "01000"; -- default value of one
        elsif pitch_up = '1' then
          pitch_multiplier <= pitch_multiplier + 1;
        elsif pitch_down = '1' then
          pitch_multiplier <= pitch_multiplier - 1;
        else
          pitch_multiplier <= pitch_multiplier;
        end if;
      end if;
    end process pitch_mult;


  samplingCounter: process(clk)
    begin
      if rising_edge(clk) then
        if clear_samp = '1' then
          sampling_counter <= "00000000000";
        elsif count = '1' then
          sampling_counter <= sampling_counter + 1;
        else
          sampling_counter <= sampling_counter;
        end if;
      end if;
    end process samplingCounter;

  extended_pitch_multiplier <= "000000000" & pitch_multiplier & "000";
  shiftingCounter: process(clk)
```

```
    begin
      if rising_edge(clk) then
        if clear_shift = '1' then
          shift_counter <= "00000000000000000";
        elsif count = '1' then
          shift_counter <= shift_counter + extended_pitch_multiplier;
        else
          shift_counter <= shift_counter;
        end if;
      end if;
    end process shiftingCounter;

  internal_ram_address <= sampling_counter when shift_count = '0'
                          else shift_counter(16 downto 6);
  ram_address(10 downto 0) <= internal_ram_address;
  ram_address(11) <= shift_buf xor buf_sel;

  tflipflop: process(clk)
    begin
      if rising_edge(clk) then
        if swap_buf = '1' then
          buf_sel <= not(buf_sel);
        end if;
      end if;
    end process tflipflop;

  -- full detector
  fulldet : process(clk)
    begin
      if rising_edge(clk) then
        --full <= internal_full;
        if ((internal_ram_address(10 downto 7) = buffer_size)
            and (internal_ram_address(6 downto 0) = "1111111")) then
          full <= '1';
        else
          full <= '0';
        end if;
      end if;
    end process fulldet;

--     internal_full <='1' when ((internal_ram_address(10 downto 7) = buffer_size)
--                        and internal_ram_address(6 downto 0) = "0000000") --"1111111")
--                         else '0';
```

```
end x;
```

A.6 Signal Accumulator

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

entity signalAccumulator is
  port (
    clk                         : in    std_logic;
    sa_loadadd, sa_clear, sa_oe : in    std_logic;
    data_bus                    : inout unsigned(7 downto 0));
end signalAccumulator;

architecture x of signalAccumulator is
  signal data, data_out, half_data_bus, half_data : unsigned(7 downto 0);
begin  -- x
  half_data_bus <= '0' & data_bus(7 downto 1);
  half_data <= '0' & data(7 downto 1);
  accum : process(clk)
    begin
      if rising_edge(clk) then
        if sa_clear = '1' then
          data <= "00000000";
        elsif sa_loadadd = '1' then
          data <= data + half_data_bus;
        else
          data <= data;
        end if;
      end if;
    end process accum;

  output : process(clk, sa_oe, data)
    begin
      if sa_oe = '1' then
        data_bus <= data;
      else
        data_bus <= (others => 'Z');
      end if;
    end process output;
end x;
```

A.7 MCU ROM

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

entity cpldROM is
  port (
    address : in  unsigned(7 downto 0);
    data    : out unsigned(15 downto 0));
end cpldROM;


architecture x of cpldROM is

begin  -- x

  with address select
    data <=
X"8000" when X"00",
X"90c0" when X"01",
X"90c0" when X"02",
X"90c0" when X"03",
X"8000" when X"04",
X"8000" when X"05",
X"8000" when X"06",
X"2009" when X"07",
X"7007" when X"08",
X"8000" when X"09",
X"900c" when X"0a",
X"8004" when X"0b",
X"100c" when X"0c",
X"8004" when X"0d",
X"8007" when X"0e",
X"8807" when X"0f",
X"8004" when X"10",
X"8000" when X"11",
X"8000" when X"12",
X"3015" when X"13",
X"701c" when X"14",
X"4022" when X"15",
X"8600" when X"16",
X"8600" when X"17",
```

```
X"8601" when X"18",
X"8601" when X"19",
X"8611" when X"1a",
X"702c" when X"1b",
X"8000" when X"1c",
X"8000" when X"1d",
X"8001" when X"1e",
X"8001" when X"1f",
X"8011" when X"20",
X"702c" when X"21",
X"8600" when X"22",
X"8600" when X"23",
X"8601" when X"24",
X"8601" when X"25",
X"8e01" when X"26",
X"8000" when X"27",
X"8000" when X"28",
X"a010" when X"29",
X"a000" when X"2a",
X"8000" when X"2b",
X"8020" when X"2c",
X"8000" when X"2d",
X"0030" when X"2e",
X"7037" when X"2f",
X"81c0" when X"30",
X"81c0" when X"31",
X"81c0" when X"32",
X"80c0" when X"33",
X"80c0" when X"34",
X"8000" when X"35",
X"7037" when X"36",
X"8000" when X"37",
X"8400" when X"38",
X"8400" when X"39",
X"8400" when X"3a",
X"8400" when X"3b",
X"003e" when X"3c",
X"703f" when X"3d",
X"8080" when X"3e",
X"8000" when X"3f",
X"8000" when X"40",
X"7007" when X"41",
    "----------------" when others;
```

```
end x;
```

B Assembler Source Code Listings

B.1 MCU Specification

```
/* mcutest.sp                                               */
/* assembler spec for debugging and testing of 163-based MCU */
/* created 2-26-98                                          */
/* (adapted from mcu.sp for AM29C10A-based MCU)             */

/*************************************************************/
/* Instruction Word Organization:                            */
/*   conditional branches              0cccxxxx aaaaaaaa      */
/*   unconditional branches            0111xxxx aaaaaaaa      */
/*   assertion statements              1sssssss ssssssss      */
/*    where c = status selection                             */
/*          a = alternative address, i.e. jump address       */
/*          s = assertion signals                            */
/*************************************************************/

op <15:0>;                      /* Indicates the available bits             */
address op <7:0>;               /* Indicates bit locations for addresses    */
value op <7:0>;

/*
 * There is nothing magic about upper case.
 * You may change things to lower case as you wish.
 * Remember, the assembler maps all characters to lower case anyway!
 */

/*
 * Instruction set for your MCU
 */

CJMP op<15>=%b0; /* Conditional JuMP */

JMP op<15:12>=%b0111; /* unconditional JuMP */

ASSERT op<15>=%b1; /* unconditional ASSERT */

/* These are defined so that you may use them to make your code more
 * readable.  Their use is not required. */
```

```
IF      nop;
THEN    nop;
TRUE    op<14:12>=%b111;        /* This causes the 151 to output true */
RESET op<15:0>=%b0111000000000000;

/* Assertions */
ram_oe op<0>=1;
ram_we op<1>=1;
adc_enable op<2>=1;
adc_sample op<3>=1;
dac_enable op<4>=1;
st_count op<5>=1;
st_clear_samp op<6>=1;
st_clear_shift op<7>=1;
st_swap_buf op<8>=1;
st_shift_buf op<9>=1;
st_shift_count op<10>=1;
sa_loadadd op<11>=1;
sa_clear op<12>=1;
sa_oe op<13>=1;

/*
 * Status signals:  Switches and frequency divider output OSC
 * Make sure that all status signals that change during mcu operation
 * are synchronized to the system /CLK
 */

st_full op<14:12>=0;
adc_not_ready op<14:12>=1;
timing_sample op<14:12>=2;
pass_shifted    op<14:12>=3;
pass_original   op<14:12>=4;
```

B.2 MCU Assembly Code

```
/* mcutest.as                                                */
/* assembler code for debugging and testing of 163-based MCU */
/* created 2-26-98                                           */
/* (inspired by mcu.as for AM29C10A-based MCU)               */
```

```
# SPEC_FILE = mcu.sp; /* This statement is required at the
                           beginning of the ASSEM_FILE. It tells
                           where the SPEC_FILE can be found. */

# LIST_FILE = new_mcu.lst; /* This statement specifies the name for
                           the assembler listing file.  If not
                           included, no listing will be created */

# MASK_COUNT = 8;          /* This statement is required to mask out 8
                           bits of the 16 bit op-code to produce 2 PROM
                           files. Use with the 'assem16to8' command. */

# SET_ADDRESS = 0;         /* This statement tells the program at what
                           address to start assembling. The address
                           given is a hexadecimal number. */

# LOAD_ADDRESS = 100;      /* This statement, if used AFTER the
                           SET_ADDRESS statement, determines the
                           beginning PROM address for the program
                           image. The address is in HEX.  */
REAL_START:
assert ;
assert st_clear_shift st_clear_samp sa_clear;
assert st_clear_shift st_clear_samp sa_clear;
assert st_clear_shift st_clear_samp sa_clear;
assert ;
assert ;
assert ;

START: CJMP timing_sample SAMPLE_READY;
JMP START ;
SAMPLE_READY:
assert ;
assert adc_enable adc_sample sa_clear;
assert adc_enable ;

/* read from ADC and write it to sampling buffer */
ADC_WAIT:
CJMP adc_not_ready ADC_WAIT;
/* if we get to here, that means the ADC is ready to read from */
assert adc_enable;

/* adc_enable has been high since we've
been in a cjmp loop. we need to keep it
```

```
high in order to keep the adc writing to
the bus */

assert adc_enable ram_oe ram_we;
assert adc_enable ram_oe ram_we sa_loadadd;
assert adc_enable ;  /* why? */
assert ;
/* clear adc_enable so the adc stops writing to the bus */
assert ;

/* decide wheather we're passing original, shifted, or both */
CJMP PASS_SHIFTED SHIFTED1;
JMP ORIGINAL ;
SHIFTED1:
CJMP PASS_ORIGINAL BOTH ;
/* we're passing the shifted version only */
/* read sample from SHIFTING buffer and write to DAC */
assert st_shift_count st_shift_buf;
assert st_shift_count st_shift_buf;
assert ram_oe st_shift_count st_shift_buf ;
assert ram_oe st_shift_count st_shift_buf ;
assert ram_oe st_shift_count st_shift_buf dac_enable;
JMP COUNTER_INCREMENT ;

ORIGINAL:
/* we're passing the original version only */
/* read sample from SAMPLING buffer and write to DAC */
assert ;
assert ;
assert ram_oe ;
assert ram_oe ;
assert ram_oe dac_enable;
JMP COUNTER_INCREMENT ;

BOTH:
/* we're passing both original and shifted */
/* read sample from SHIFTING buffer and write to ACCUMULATOR */
assert st_shift_count st_shift_buf;
assert st_shift_count st_shift_buf;
assert ram_oe st_shift_count st_shift_buf ;
assert ram_oe st_shift_count st_shift_buf ;
assert ram_oe st_shift_count st_shift_buf sa_loadadd;
assert ;
assert ;
```

```
/* now write combined signal to DAC */
assert sa_oe dac_enable ;
assert sa_oe ;
assert ;
/* JMP COUNTER_INCREMENT ; */


COUNTER_INCREMENT:
/* increment both counters */
assert st_count ;
assert ;
/* check if sample counter is full */
CJMP st_full SAMP_BUF_FULL;
JMP SAMP_BUF_NOT_FULL ;
SAMP_BUF_FULL:
assert st_swap_buf st_clear_samp st_clear_shift;
assert st_swap_buf st_clear_samp st_clear_shift;
assert st_swap_buf st_clear_samp st_clear_shift;
assert st_clear_samp st_clear_shift;
assert st_clear_samp st_clear_shift;
assert ;
JMP SAMP_BUF_NOT_FULL ;

SAMP_BUF_NOT_FULL:
assert ;
/* check if shifting counter is full */
assert st_shift_count ;
assert st_shift_count ;
assert st_shift_count ;
assert st_shift_count ;
CJMP st_full SHIFT_COUNT_FULL
/* relies on having st_shift_count set in previous instruction */;

JMP SHIFT_COUNT_NOT_FULL;

SHIFT_COUNT_FULL:
assert st_clear_shift ;

SHIFT_COUNT_NOT_FULL:
assert ;
assert ;

JMP START ;
```

B.3 MCU Audio Test Assembly Code

```
/* mcutest.as                                                */
/* assembler code for debugging and testing of 163-based MCU */
/* created 2-26-98                                           */
/* (inspired by mcu.as for AM29C10A-based MCU)               */

# SPEC_FILE = mcu.sp; /* This statement is required at the
                         beginning of the ASSEM_FILE. It tells
                         where the SPEC_FILE can be found. */

# LIST_FILE = mcu_audio_test.lst; /* This statement specifies the name for
                         the assembler listing file.  If not
                         included, no listing will be created */

# MASK_COUNT = 8;        /* This statement is required to mask out 8
                         bits of the 16 bit op-code to produce 2 PROM
                         files. Use with the 'assem16to8' command. */

# SET_ADDRESS = 0;       /* This statement tells the program at what
                         address to start assembling. The address
                         given is a hexadecimal number. */

# LOAD_ADDRESS = 100;    /* This statement, if used AFTER the
                         SET_ADDRESS statement, determines the
                         beginning PROM address for the program
                         image. The address is in HEX.  */

TST_START: assert ;
assert adc_enable adc_sample;
assert adc_enable ;
TST_ADC_WAIT:
CJMP adc_not_ready TST_ADC_WAIT;
/* if we get to here, that means the ADC is ready to read from */
assert adc_enable dac_enable; /* adc_enable has been high since we've
                                 been in a cjmp loop. we need to keep it
 high in order to keep the adc writing to
 the bus */
assert ;
JMP TST_START ;
```

C VHDL ROM Tools Code Listings

C.1 Assmebler to VHDL ROM Converter

```
#!/usr/bin/env python
import sys, re, string


vhdlTemplate = """
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity cpldROM is
  port (
    address : in  unsigned(7 downto 0);
    data    : out unsigned(15 downto 0));
end cpldROM;


architecture x of cpldROM is

begin  -- x

  with address select
    data <=
%s
    "----------------" when others;

end x;
"""

class assemblerFile:
    headerRE = re.compile(r'\#\s*([A-z_]+)\s*=\s*([A-z0-9]+)\s*\;')

    def __init__(self, fname):
        f = open(fname, 'r')
        self.loadFile(f)
        f.close()

    def loadFile(self, f):
        codeLines = []
        headerLines = []
        for line in f.readlines():
            if line[0] == '#':
                headerLines.append(line)
            else:
                codeLines.append(int(line, 16))

        for headerLine in headerLines:
            key, val = self.headerRE.findall(headerLine)[0]
            setattr(self, key, int(val))
```

```
        self.codeLines = codeLines

    def compile(self):
        resultList = []
        startAddress = 0 #getattr(self, 'load_address', 0)
        # load_address=100 means add 0x100 to all addresses, but we
        # interpret it to mean add decimal 100 to all addresses...
        for index in range(len(self.codeLines)):
            data = string.zfill(hex(self.codeLines[index])[2:], 4)
            addr = string.zfill(hex(index + startAddress)[2:], 2)
            resultList.append('\tX"%s" when X"%s",' % (data, addr))
        return vhdlTemplate % string.join(resultList, '\n')



if __name__ == '__main__':
    fname = sys.argv[1]
    a = assemblerFile(fname)
    print a.compile()
```

C.2 Assembler to Finite State Machine Converter

```
#!/usr/bin/env python
import sys, re, string


vhdlTemplate = """
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity cpldROM is
  port (
    address : in  unsigned(7 downto 0);
    data    : out unsigned(15 downto 0));
end cpldROM;


architecture x of cpldROM is

begin  -- x

  with address select
    data <=
%s
    X"0000" when others;

end x;
"""
```

```
fsm_template = open('vhdlTemplate.vhd', 'r').read()

assert_template = """
   when state_%(state_num)s =>
    next_state <= state_%(next_state_num)s;
    outputs <= "%(assert_bits)s";
"""
jmp_template = """
   when state_%(state_num)s =>
    next_state <= state_%(next_state_num)s;
    outputs <= outputs;
"""
cjmp_template = """
   when state_%(state_num)s =>
    outputs <= outputs;
    if inputs(%(input_selector)s) = '1' then
     next_state <= state_%(cjmp_state_num)s;
    else
     next_state <= state_%(next_state_num)s;
    end if;
"""

class assemblerFile:
    headerRE = re.compile(r'\#\s*([A-z_]+)\s*=\s*([A-z0-9]+)\s*\;')

    def __init__(self, fname):
        f = open(fname, 'r')
        self.loadFile(f)
        f.close()

    def loadFile(self, f):
        codeLines = []
        headerLines = []
        for line in f.readlines():
            if line[0] == '#':
                headerLines.append(line)
            else:
                codeLines.append(int(line, 16))

        for headerLine in headerLines:
            key, val =  self.headerRE.findall(headerLine)[0]
            setattr(self, key, int(val))

        self.codeLines = codeLines

    def compile(self):
        resultList = []
        startAddress = 0 #getattr(self, 'load_address', 0)
        # load_address=100 means add 0x100 to all addresses, but we
        # interpret it to mean add decimal 100 to all addresses...
        for index in range(len(self.codeLines)):
            data = string.zfill(hex(self.codeLines[index])[2:], 4)
```

```
            addr = string.zfill(hex(index + startAddress)[2:], 2)
            resultList.append('\tX"%s" when X"%s",' % (data, addr))
        return vhdlTemplate % string.join(resultList, '\n')

    def compileFSM(self):
        bits = map(bitString, self.codeLines)
        fsm_lines = []
        state_dict = {}
        state_num_to_line_map = {}
        next_state_dict = {}
        for address in range(len(bits)):
            instruction = bits[address]
            state_dict[address] = 1
            fsm_vhdl_line = self.generateFSMvhdlLine(instruction, address, state_dict, next_state_dict)
            state_num_to_line_map[address] = fsm_vhdl_line
            #fsm_lines.append(fsm_vhdl_line)

        for state_num, fsm_line in state_num_to_line_map.items():
            if next_state_dict.has_key(state_num):
                fsm_lines.append(fsm_line)


        state_list = []
        for num in state_dict.keys():
            if next_state_dict.has_key(num):
                state_list.append('state_%i' % num)
        state_list.sort()
        state_str = string.join(state_list, ', ')

        return fsm_template % {'cases': string.join(fsm_lines, ''),
                               'state_list': state_str}

    def generateFSMvhdlLine(self, instruction, address, state_dict, next_state_dict):
        mcu_assert = instruction[-1]
        state_num = address
        if mcu_assert == '1':
            # we're asserting
            next_state_num = address + 1 # since we're not jumping,
            # just go to the next instruction
            assert_bits = instruction[:-1]
            # we're going to output the asserts
            #  in this instruction
            next_state_dict[next_state_num] = 1
            return assert_template % {'state_num': state_num,
                                      'next_state_num': next_state_num,
                                      'assert_bits': assert_bits}
        else:
            # we're jumping
            selector = instruction[-4:-1]

            if selector == '111':
                # its a jmp, an unconditional branch
                next_state_num = int(instruction[:8], 2)
```

```
            state_dict[next_state_num] = 1
            next_state_dict[next_state_num] = 1
            return jmp_template % {'state_num': state_num,
                                   'next_state_num': next_state_num}
        else:
            # this is a cjmp, a conditional branch
            next_state_num = address + 1
            cjmp_state_num = int(instruction[:8], 2)
            next_state_dict[next_state_num] = 1
            next_state_dict[cjmp_state_num] = 1
            state_dict[cjmp_state_num] = 1
            input_selector = int(selector, 2)
            return cjmp_template % {'state_num': state_num,
                                    'next_state_num': next_state_num,
                                    'cjmp_state_num': cjmp_state_num,
                                    'input_selector': input_selector}


def bitString(integer):
    bitList = []
    for i in range(16):
        bitList.append( (integer >> i) & 1)
    strList = map(str, bitList)
    #strList.reverse()
    return string.join(strList, '')

if __name__ == '__main__':
    fname = sys.argv[1]
    a = assemblerFile(fname)
    print a.compileFSM()
    #print a.compile()
    #for x in map(bitString, range(70)): print x
```

C.3 VHDL Template used for Assembler to FSM Conversion

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

entity mcu_fsm is
  port (
    clk, reset     : in  std_logic;
    inputs         : in  std_logic_vector(7 downto 0);
    assert_outputs : out std_logic_vector(14 downto 0));
end mcu_fsm;


architecture python of mcu_fsm is
```

```
  type stateType is (%(state_list)s);
  signal present_state, next_state : stateType;
  signal outputs : std_logic_vector(14 downto 0);
begin  -- python
  assert_outputs <= outputs;

  state_clk: process(clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        present_state <= state_0;
      else
        present_state <= next_state;
      end if;
    end if;
  end process state_clk;

  state_comb: process(present_state, inputs, outputs)
  begin
    case present_state is
%(cases)s


      when others =>
        null; --next_state <= start;

    end case;
  end process state_comb;

end python;
```