VHDL Examples

Introduction

This section of the lecture notes contains some VHDL examples and a description of the operation of that code and comments on some important features. Nearly all of these examples will work on the M4A5 devices used in this course.

Example 1 Reaction Timer

Design a reaction timer with the following specifications:

1 Pressing SW1 clears the display and starts the process.

2 At some time more than approximately 4 seconds after the start button has been pushed and before approximately 8 seconds after the start button has been pushed, the seven segment display should start counting.

3 The counter is frozen to display the reaction time as soon as the Stop button (SW3) is pressed.

Notes:

The display should count in decimal mode (i.e. the numbers 0 to 9 only) and include as many digits as can be fitted into the CPLD. It is possible to use part of a digit, like the numbers 0 and 1 only for the most significant digit, so that a 3 and a half digit display is better than a three digit display.

It is desirable to use the 2 kHz clock to drive the display.

It is desirable for the display to display the reaction time in a meaningful manner by say counting in approximately hundred's of a second or milliseconds.

It is desirable for the counter to stop if no stop button is pushed, by either halting at the maximum possible count or clearing the counter to zero.

Solution

A random time is produced by having a counter running whenever power is applied to the circuit. The random time is the time between the circuit being powered up and the time the start button is pressed. This counter "Count" is a 14 bit counter to frequency divide the 2kHz clock and produce a 8 second period for the rollover of that clock. (i.e. the rollover from Count = 11111111111111 to 00000000000000).

Pressing the start button, sets a WaitRan variable to 1, denoting wait for the next rollover of the counter and clears the RunDisp variable to 0, denoting hold the display value. These two variables are flags controlling the counting and display operations. Pressing the start button also clears the most significant bit of the counter to 0, but does not effect the other bits of the counter, so that between 4 and 8 seconds are required before the counter rolls over.

© 1990-2008, C. J. Kikkert, James Cook University

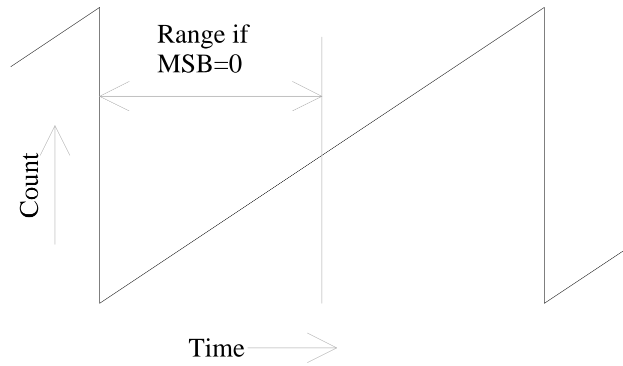

Figure 1. Counter Value with time.

Display

Once the counter rolls over, the RunDisp flag is set to 1 causing the display to start counting. The display counter consists of 4 decade counters, with a value 9 in a display digit, causing that digit to be set to 0 on the next input to that display bit counter. The output from the 4 decade counters are connected to the seven segment displays to display the value of these decade counters. The stop button clears the RunDisp variable to 0, thus holding the display value. If the stop button is not pressed, then the counter counts to 9999 and halts there.

The block diagram for the whole system is shown in Figure 2. The 2 KHz clock is firstly divided by 2, to give a 1kHz clock, which then drives the Least significant digit of the display. The reaction time is thus displayed in milliseconds.

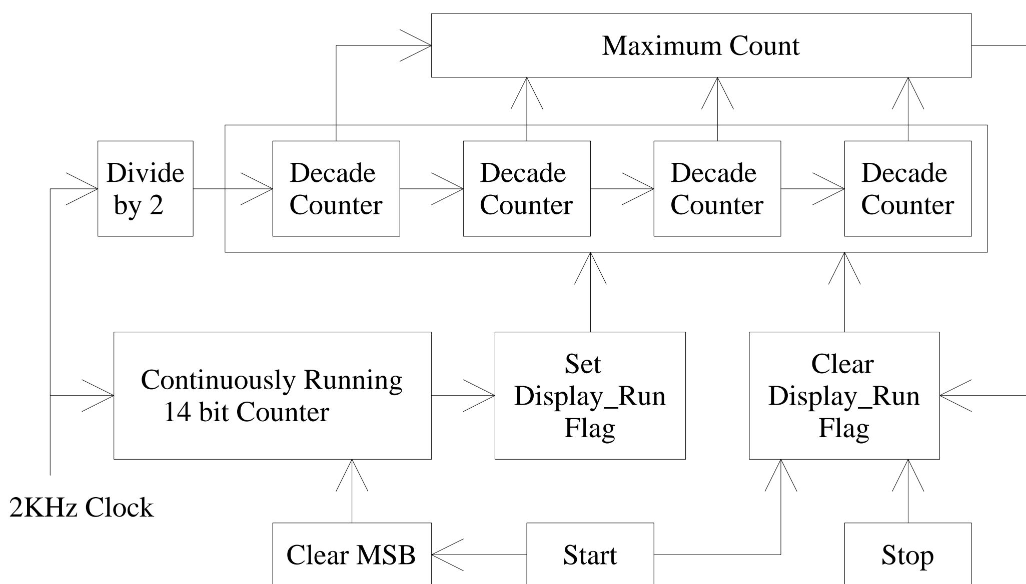

Figure 2. Block Diagram of Reaction Timer Operation

VHDL Code

The VHDL code corresponding to the above block diagram is as follows:

```
-- Reaction Timer for EE4306 VHDL Assignment
-- Solution Author C. J. Kikkert Aug 2004
-- Operation: Press Start (SW1) clears the display and stops it incrementing
--         it also clears the MSB of the 14 bit random time counter 'Count',
--         resulting in a 4 to 8 second time before the counter rolls over.
--         This counter runs as long as the circuit is powered up, giving a random time
--         between the powering up of the circuit and the start button being pressed.
-- The counter rolling over starts the display incrementing
-- Pressing the Stop button stops the display and holds the displayed value
-- If no stop button is pressed, the display stops at the largest value 9999,
-- corresponding to 9.999 seconds. The accuracy of the timer is determined by the
--RC values of the oscillator and is within about 5%.
-- A module for driving the seven segment display on the Lattice Boards
library ieee;
use ieee.std_logic_1164.all;

entity SevenSeg is
        Port (Hexin: in std_logic_vector (3 downto 0);
                SevSegOut: out std_logic_vector (6 downto 0));
end;

architecture SevenSeg_arch of SevenSeg is
begin
        process(Hexin)
        begin
                Lab0: case Hexin is
                --need to have inverse of output coded here as is normal for these Development Boards
                        when X"0" => SevSegOut <= "0000001";      --0
                        when X"1" => SevSegOut <= "1001111";      --1
                        when X"2" => SevSegOut <= "0010010";      --2
                        when X"3" => SevSegOut <= "0000110";      --3
                        when X"4" => SevSegOut <= "1001100";      --4
                        when X"5" => SevSegOut <= "0100100";      --5
                        when X"6" => SevSegOut <= "0100000";      --6
                        when X"7" => SevSegOut <= "0001111";      --7
                        when X"8" => SevSegOut <= "0000000";      --8
                        when X"9" => SevSegOut <= "0000100";      --9
                        when X"A" => SevSegOut <= "0001000";      --A
                        when X"B" => SevSegOut <= "1100000";      --b
                        when X"C" => SevSegOut <= "0110001";      --C
                        when X"D" => SevSegOut <= "1000010";      --d
                        when X"E" => SevSegOut <= "0110000";      --E
                        when others => SevSegOut <= "0111000";    --F
                end case Lab0;
        end process;
end;
-- end of seven segment display driver

-- The main program for the Reaction Timer
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity ReacTime is
        Port (  Clk : in std_logic ;                    --2KHz to permit 0.001sec resolution
                SevSeg0: out std_logic_vector (6 downto 0);  -- Seven segment display LSB
                SevSeg1: out std_logic_vector (6 downto 0);  -- Seven segment display
                SevSeg2: out std_logic_vector (6 downto 0);  -- Seven segment display
                SevSeg3: out std_logic_vector (6 downto 0);  -- Seven segment display MSB
                Stop:    in std_logic;                  --stop button SW3
                Start: in std_logic);                   --start button SW1

        attribute loc : string;
        attribute loc of Clk: signal is "P11" ;
        attribute loc of Start: signal is "P9";  --Start button SW1
        attribute loc of Stop: signal is "P31"; --Stop button SW3
        attribute loc of SevSeg3: signal is "P2 P3 P4 P5 P6 P7 P8" ; -- Pin allocation of Displays
        attribute loc of SevSeg2: signal is "P14 P15 P16 P17 P18 P19 P20" ;
        attribute loc of SevSeg1: signal is "P24 P25 P26 P27 P28 P29 P30" ;
        attribute loc of SevSeg0: signal is "P36 P37 P38 P39 P40 P41 P42" ;
end;
```

© 1990-2008, C. J. Kikkert, James Cook University

```
architecture ReacTime_arch of ReacTime is
        component SevenSeg
                port (Hexin: in std_logic_vector;                   -- input for Display Module
                SevSegOut : out std_logic_vector);                  -- output from Display Module
        end component;

signal  CountB0: std_logic_vector (3 downto 0);         -- LS Digit Decade Display Counter.
signal  CountB1: std_logic_vector (3 downto 0);         -- Digit 2 Decade Display Counter.
signal  CountB2: std_logic_vector (3 downto 0);         -- Digit 3 Decade Display Counter.
signal  CountB3: std_logic_vector (3 downto 0);         -- Digit 4 Decade Display Counter.
signal  Count: std_logic_vector (13 downto 0); -- 14 bit counter, 8 second interval for 2.048kHz clock
--signal Count: std_logic_vector (6 downto 0);          -- 8 bit short counter for testing
signal  RunDisp: std_logic;     --Flag to increment Display counter
signal  WaitRan: std_logic;     --wait for random start time
begin
        D0: SevenSeg port map (CountB0, SevSeg0);  -- Seven Segment Display LSDigit
        D1: SevenSeg port map (CountB1, SevSeg1);  -- Seven Segment Display Digit 2
        D2: SevenSeg port map (CountB2, SevSeg2);  -- Seven Segment Display Digit 3
        D3: SevenSeg port map (CountB3, SevSeg3);  -- Seven Segment Display MSDigit

        P_Clock2: process
        begin  -- counter running always with 8 second period
        wait until rising_edge(Clk);
                Count <= Count + "0000000000001";  -- always increment Counter
--                      Count <= Count + "0000001";  -- use this for testing only
                if ((Count = "000000000000") and (WaitRan = '1')) then
                        RunDisp <= '1';         --Random time arrived, Start the display
                        -- Note this branch should also specify WaitRan, Count and CountB1 to CountB3
                elsif (Start = '0') then        -- start button pressed go when button released.
                        WaitRan <= '1';         -- Wait for random starting time.
                        RunDisp <= '0';         -- do not increment the display
                        Count(13) <= '0';       -- clear MSB of counter to ensure >4 sec to rollover
                        -- should specify other bits of Count as: Count(12 downto 0) <= Count(12 downto 0);
--                      Count(6) <= '0';        -- clear MSB of short test counter
                        CountB0 <= "0000";      --set the display to 0000
                        CountB1 <= "0000";
                        CountB2 <= "0000";
                        CountB3 <= "0000";
                elsif (Stop = '0') then         -- stop button pressed
                        RunDisp <= '0';         -- Stop the display incrementing
                        WaitRan <= '0';         -- Disable waiting for random time
                        -- Note this branch should also specify Count and CountB1 to CountB3
                elsif ((RunDisp = '1') and (Count(0) = '1')) then -- increment display until stop
                                                -- Count(0)=1 results in 1024Hz clock
                        if (CountB0 = X"9") and (CountB1 = X"9") and (CountB3 = X"9")
                                and (CountB3 = X"9") then
                                                -- maximum count reached halt the display on 9999
                                RunDisp <= '0';         -- Stop Counter (same as Stop)
                                WaitRan <= '0';         -- Disable waiting for random time
                        elsif (CountB0 = X"9") then     -- increment display counter
                        -- Note this branch should also specify WaitRan, and Count
                                CountB0 <= "0000";
                                if (CountB1 = X"9") then -- Maximum, increment next digit
                                        CountB1 <= "0000";
                                        if (CountB2 = X"9") then -- Maximum, increment next digit
                                                CountB2 <= "0000";
                                                CountB3 <= CountB3 + "0001"; -- increment digit
                                                -- Note CountB3 is stopped at max value, so no rollover
                                        else
                                                CountB2 <= CountB2 + "0001"; -- increment digit
                                        end if;                         -- end B2 Count branch
                                else
                                        CountB1 <= CountB1 + "0001";    -- increment digit
                                end if;                                 --end B1 Count branch
                        else
                                CountB0 <= CountB0 + "0001";    -- increment digit
                        end if;                                 --B0 Count
                end if;                                         --end mode selection
end process P_Clock2;
end ReacTime_arch;
```

© 1990-2008, C. J. Kikkert, James Cook University

This design uses the MA4A5. For this design a smaller fit is obtained using the "Speed" optimisation constraint than with the Area optimisation constraint. That is not normal. In addition the above code will fit with the precision compiler, but not with the Synario compiler. Using the default Optimization Constraints and the Precision compiler for ispLever 5.0 gives:

Design_Summary
~~~~~~~~~~~~~~

Total Input Pins :	3
Total Output Pins :	28
Total Bidir I/O Pins :	0
Total Flip-Flops :	32
Total Product Terms :	249
Total Reserved Pins :	0
Total Reserved Blocks :	0

Device_Resource_Summary
~~~~~~~~~~~~~~~~~~~~~~

	Total Available	Used	Available		Utilization
Dedicated Pins					
Input-Only Pins	..	..	..	-->	..
Clock/Input Pins	2	1	1	-->	50%
I/O Pins	32	30	2	-->	93%
Logic Macrocells	64	64	0	-->	100%
Input Registers	32	0	32	-->	0%
Unusable Macrocells	..	0	..		
Total Block Inputs	132	120	12	-->	90%
Logical Product Terms	320	260	60	-->	81%
Product Term Clusters	64	64	0	-->	100%

Different compiler versions will give slightly different fitting results.

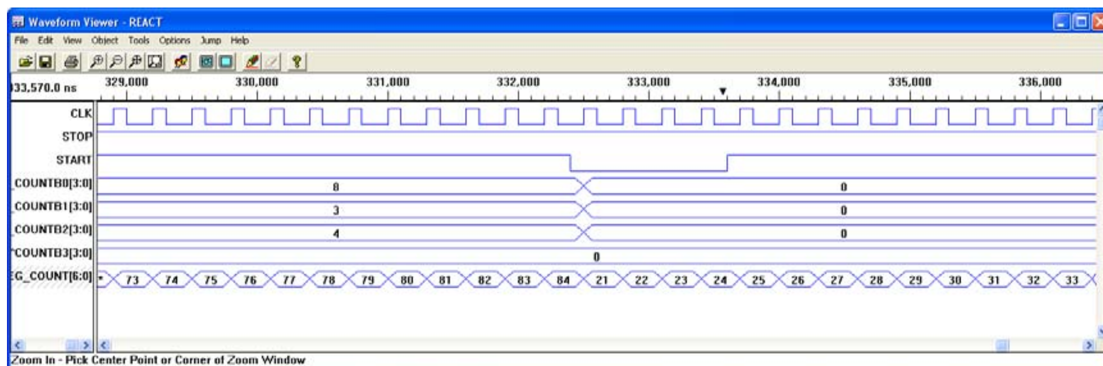

Figure 3. Waveform showing how the most significant bit of the count(6:0) is cleared.

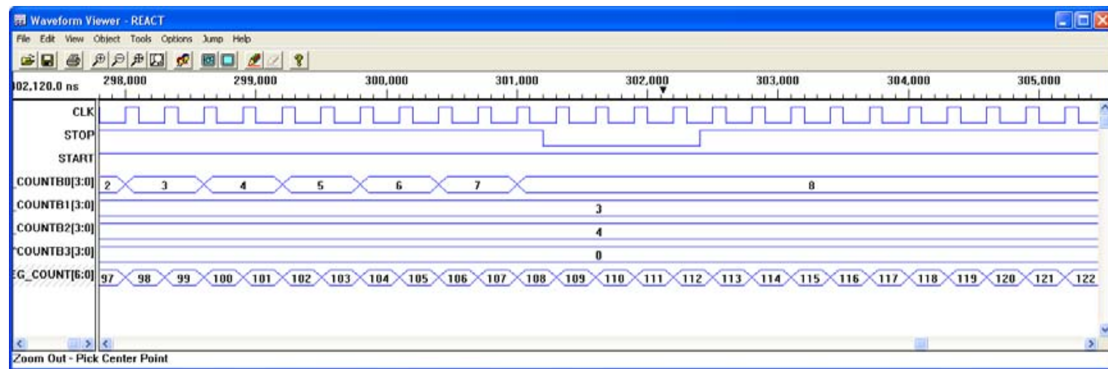

Figure 4. Waveform showing how the count being halted and the reaction time being displayed

For testing the design short, 8 bit, counter is used and the long 14 bit counter is commented out. For normal operation the short counter is commented out and the long counter is enabled. Commenting out the code as needed, allows a rapid change between the full realisation and the testing only realisation. The resulting waveforms are shown in figures 3 and 4

The VHDL code just fits in the CPLD and the reaction timer operates as required.

Second Solution Principle of operation

The above solution requires a 14 bit counter for producing the random time and a 16 bit counter for storing the displayed value. By only using one counter to drive both the display and produce the random time, a smaller solution is obtained. To use the one counter, the display needs to be turned OFF when the random time is counted and the display needs to be ON when the reaction time is being evaluated. This can be achieved by adding an extra input to the seven segment display, to have the Hex input displayed or have the display blanked or showing say - - - -, when the random delay is being evaluated. See the Sevenseg component in the VHDL code below, to see how that is achieved.

To produce the random delay, the counters are incremented directly from the 2 kHz clock when the start button is being pushed. The randomness will result from the different length of time that the start button is pushed. To obtain the correct delay, the counters are cycled between 2000 and 5999, resulting in a 4 to 8 second delay before 0000 is reached.

The states are as follows:

1. The Start button is pushed: Increment each digit individually to produce the random number. The MSB digit cycles between 2 and 5, MSB2 digit cycles between 0 and 9, LSB2 digit counts up in hex, LSB digit counts down in hex, to produce a random number.
2. Once the Start button is released, the counter counts as a normal decade counter.
3. Once 999 is reached the ShowDisp flag is set, turning on the display to show the count value. This is the cue to the operator to press the stop button.
4. When the stop button is pressed or the display reaches 9999, the Halt flag is set which causes the display to stop incrementing

The resulting code is as follows:

© 1990-2008, C. J. Kikkert, James Cook University

```vhdl
-- Reaction Timer for EE4306 Assignment
-- Solution2  Author C. J. Kikkert Aug 2004

-- Operation: Press Start (SW1) blanks the display (shows ---- )
--          it also increments the display counters at a 2 kHz rate,
--          resulting in a 4 to 8 second time before the counter rolls over.
-- The counter rolling over starts the display incrementing
-- Pressing the Stop button stops the display incrementing and holds the displayed value
-- If no stop button is pressed, the display stops at the largest value 9999,
--          corresponding to 9.999 seconds. The accuracy of the timer is determined by the
--          RC values of the oscillator and is within about 5%.
-- A module for driving the seven segment display on the Lattice Boards
library ieee;
use ieee.std_logic_1164.all;

entity SevenSeg is      -- Note display now has 2 inputs, one to blank the display, the other the Hex number.
        Port (  ShowDisp: in std_logic; --this is the only line changed
                Hexin: in std_logic_vector (3 downto 0);
                SevSegOut: out std_logic_vector (6 downto 0));
end;

architecture SevenSeg_arch of SevenSeg is
begin
        process(ShowDisp, Hexin)        --combinational logic ShowDispl and Hexin in sensitivity list
        begin
                if (ShowDisp = '1') then
                        Lab0: case Hexin is
                        --need to have inverse of output coded here as normal for development board
                        -- A to F are not used here but best to use full display
                        -- a 5 bit input is used to give: one bit to turn the counter ON and OFF
                        -- and 4 bits for the value to be displayed.
                                when X"0" => SevSegOut <= "0000001";      --0
                                when X"1" => SevSegOut <= "1001111";      --1
                                when X"2" => SevSegOut <= "0010010";      --2
                                when X"3" => SevSegOut <= "0000110";      --3
                                when X"4" => SevSegOut <= "1001100";      --4
                                when X"5" => SevSegOut <= "0100100";      --5
                                when X"6" => SevSegOut <= "0100000";      --6
                                when X"7" => SevSegOut <= "0001111";      --7
                                when X"8" => SevSegOut <= "0000000";      --8
                                when X"9" => SevSegOut <= "0000100";      --9
                                when X"A" => SevSegOut <= "0001000";      --A
                                when X"B" => SevSegOut <= "1100000";      --b
                                when X"C" => SevSegOut <= "0110001";      --C
                                when X"D" => SevSegOut <= "1000010";      --d
                                when X"E" => SevSegOut <= "0110000";      --E
                                when others => SevSegOut <= "0111000";    --F
                        end case Lab0;
                else    -- Display OFF Hide value of Hexin.
                        SevSegOut <= "1111110";                           -- show ----
                end if;
        end process;
end;
-- end of seven segment display driver
```

Note how the seven segment display component now has two inputs and one seven bit wide output. The two inputs result in the minimum modification required to the standard seven segment display module, while still being able to flip the display from a counting mode to a mode where the count value is not shown.

```vhdl
-- The main program for the Reaction Timer Second Solution
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity ReacTime is
        Port (  Clk : in std_logic ;                --2KHz to permit 0.001sec resolution
                SevSeg0: out std_logic_vector (6 downto 0);   -- Seven segment display LSB
                SevSeg1: out std_logic_vector (6 downto 0);   -- Seven segment display
```

© 1990-2008, C. J. Kikkert, James Cook University

```
                SevSeg2: out std_logic_vector (6 downto 0);  -- Seven segment display
                SevSeg3: out std_logic_vector (6 downto 0);  -- Seven segment display MSB
                Stop:    in std_logic;                --stop button SW3
                Start: in std_logic);                 --start button SW1

        attribute loc : string;
        attribute loc of Clk: signal is "P11" ;
        attribute loc of Start: signal is "P9";  --Start button SW1
        attribute loc of Stop: signal is "P31"; --Stop button SW3
        attribute loc of SevSeg3: signal is "P2 P3 P4 P5 P6 P7 P8" ; -- Pin allocation of Displays
        attribute loc of SevSeg2: signal is "P14 P15 P16 P17 P18 P19 P20" ;
        attribute loc of SevSeg1: signal is "P24 P25 P26 P27 P28 P29 P30" ;
        attribute loc of SevSeg0: signal is "P36 P37 P38 P39 P40 P41 P42" ;
end;

architecture ReacTime_arch of ReacTime is
        component SevenSeg
                port (  ShowDisp: in std_logic;                  -- Display ON/OFF
                        Hexin: in std_logic_vector;              -- input for Display Module
                        SevSegOut : out std_logic_vector);       -- output from Display Module
        end component;

signal  CountB0: std_logic_vector (3 downto 0);         -- LS Digit Decade Display Counter.
signal  CountB1: std_logic_vector (3 downto 0);                 -- Digit 2 Decade Display Counter.
signal  CountB2: std_logic_vector (3 downto 0);         -- Digit 3 Decade Display Counter.
signal  CountB3: std_logic_vector (3 downto 0);         -- Digit 4 Decade Display Counter.
signal  CClk1: std_logic;          -- 1 kHz clock
--signal Count: std_logic_vector (6 downto 0);   -- 8 bit short counter for testing
signal  ShowDisp: std_logic;    --Flag to turn Display counter ON and OFF ShowDisp=1 display ON
signal  Halt: std_logic;        --Run or Hold Display Halt = 1 stop counter
signal  Max : std_logic;        --Max = 1 when display = 9999
begin
        D0: SevenSeg port map (ShowDisp,CountB0, SevSeg0);      -- Seven Segment Display LSDigit
        D1: SevenSeg port map (ShowDisp,CountB1, SevSeg1);      -- Seven Segment Display Digit 2
        D2: SevenSeg port map (ShowDisp,CountB2, SevSeg2);      -- Seven Segment Display Digit 3
        D3: SevenSeg port map (ShowDisp,CountB3, SevSeg3);      -- Seven Segment Display MSDigit

        P_Clock2: process
        begin  -- counter running always with 8 second period
        wait until rising_edge(Clk);    -- Clk is a 2048Hz clock
                Clk1 <= not Clk1;       --Toggle 2 kHz clock to produce the 1 kHz clock for display time
                if ((Start = '0') or ((Stop = '0') and (ShowDisp = '0'))) then
                -- start button pressed or Stop button pushed before the display in OFF randomise the display
                        Halt <= '0';                    -- Run the counter (very fast while display blanked).
                        ShowDisp <= '0';                -- blank the display
                        --loop CountB3 from 2 to 5 at 2kHz clock rate to provide random time 4 to 8 seconds
                        if (CountB3 < X"5" ) then       --5999 gives 4 seconds to 9999
--                      if (CountB3 < X"9" ) then       --test only
                                CountB3 <= CountB3 + X"1";  --increment display at 2kHz rate
                        else
                                CountB3 <= X"2";            --2000 gives 8 seconds to 9999
--                              CountB3 <= X"9";            --test only want  very short time
                        end if;
                        --loop CountB2 from 0 to 9 at 2kHz clock rate to provide random 0.1 second values
                        if (CountB2 < X"9" ) then           -- MSB2 displays numbers 0 to 9
                                CountB2 <= CountB2 + X"1";  --increment the display MSB at 2kHz rate
                        else
                                CountB2 <= X"0";            -- rollover MSB2
--                              CountB2 <= X"8";            -- test only
                        end if;
                        --loop CountB1 up and Count B0 down for random 0.01 and 0.001 second values
                                CountB1 <= CountB1 + X"1";  -- increment the display 2LSB at 2kHz rate
                                CountB0 <= CountB0 - X"1";  -- decrement the display LSB
                        -- CountB3, CountB2 and CountB1 have different rollover rates
                        -- number produced depends on how long the start button is pressed, ie random.

                elsif ((Halt = '1') or (Clk1 = '0')) then  --Clk 2kHz
                -- Don't increment the display of Halt flag = 1 or Clk1 = 0, to get 0.001 sec increments
                        CountB3 <= CountB3;             -- keep the same Counter values
                        CountB2 <= CountB2;
                        CountB1 <= CountB1;
                        CountB0 <= CountB0;
```

```
        -- all below only is done when Clk1 = 1 to get the 0.001 sec increments
        elsif (Stop = '0') and (ShowDisp = '1') then
        -- stop button pressed when display ON, Stop incrementing and show reaction time
                Halt <= '1';                    -- Halt the diaplay
                CountB3 <= CountB3;             -- keep the same Counter values
                CountB2 <= CountB2;
                CountB1 <= CountB1;
                CountB0 <= CountB0;
        elsif ((max = '1') and (ShowDisp = '1')) then
                Halt <= '1';            --counter reached 9999, Halt the display Stop not pushed
                CountB3 <= CountB3;             -- keep the same Counter values
                CountB2 <= CountB2;
                CountB1 <= CountB1;
                CountB0 <= CountB0;

        -- For all code below, the counter is incrementing
        else    -- run display counter until stop button pressed or max is reached
                if ((max = '1') and (Showdisp = '0')) then
                --random time has elapsed, display counter reached 9999 with display blanked
                --keep counter running and turn on the display
                        ShowDisp <= '1';        -- turn ON display
                end if;
                --      Increment the display counter
                if (CountB0 = X"9") then                -- Maximum for this digit increment next digit
                        CountB0 <= X"0";
                        if (CountB1 = X"9") then        -- Maximum for this digit increment next digit
                                CountB1 <= X"0";
                                if (CountB2 = X"9") then -- Maximum for this digit increment next digit
                                        CountB2 <= X"0";
                                        if (CountB3 = X"9") then -- Max for digit increment next digit
                        -- this rollover only occurs when at end of random time, start show display ON
                        -- CountB3 is stopped at max value, so no rollover then
                                                CountB3 <= X"0";
                                        else
                                                CountB3 <= CountB3 + X"1";
                                        end if;                 -- end CountB3
                                else
                                        CountB2 <= CountB2 + X"1";   -- increment digit
                                end if;                         -- end CountB2
                        else
                                CountB1 <= CountB1 + X"1";      -- increment digit
                        end if;                                 --end CountB1
                else
                        CountB0 <= CountB0 + X"1";              -- increment digit
                end if;                                         --end CountB0
        end if;                                                 --end mode selection

    end process P_Clock2;

    --A process to get the maximum count value of 9999.
    --To minimise hardware this is done as a 12 bit wide and gate in a separate process
    -- 9 = 1001, max=1 if display 9999
    process (CountB0, CountB1, CountB2, CountB3)
    begin
        max <= (CountB0(3) and (not CountB0(2)) and (not CountB0(1)) and CountB0(0)
        and CountB1(3) and (not CountB1(2)) and (not CountB1(1)) and CountB1(0)
        and CountB2(3) and (not CountB2(2)) and (not CountB2(1)) and CountB2(0)
        and CountB3(3) and (not CountB3(2)) and (not CountB3(1)) and CountB3(0));
    end process;
end ReacTime_arch;
```

The resulting waveforms are shown in figures 5 to 7. To produce these waveforms, the count times are shortened, as shown in the commented out code in the previous pages, so that the required transitions can be shown.

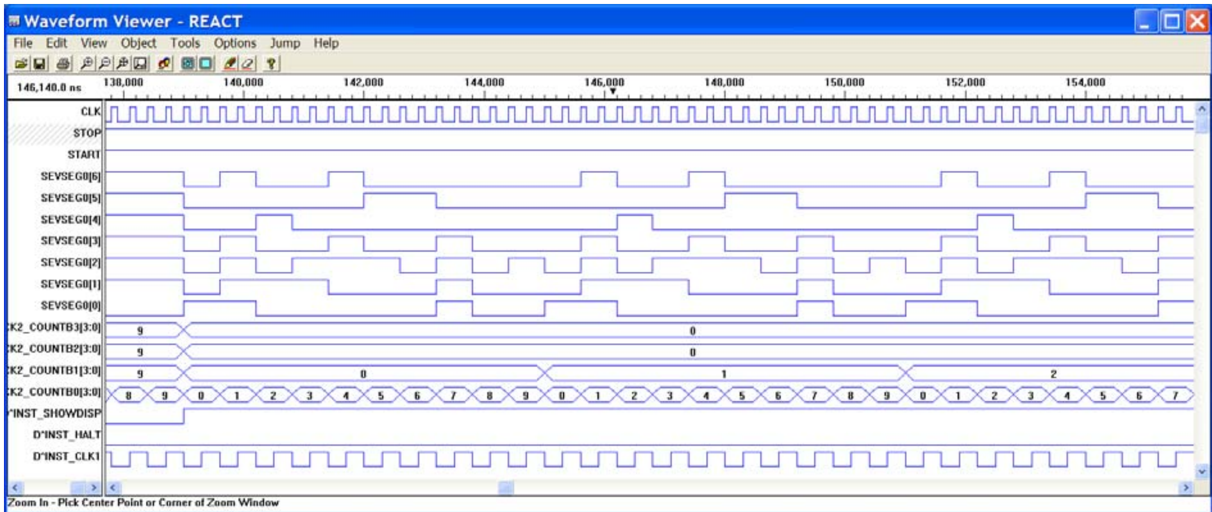

Figure 5. The time when the counter reaches 9999 and turns ON the display.

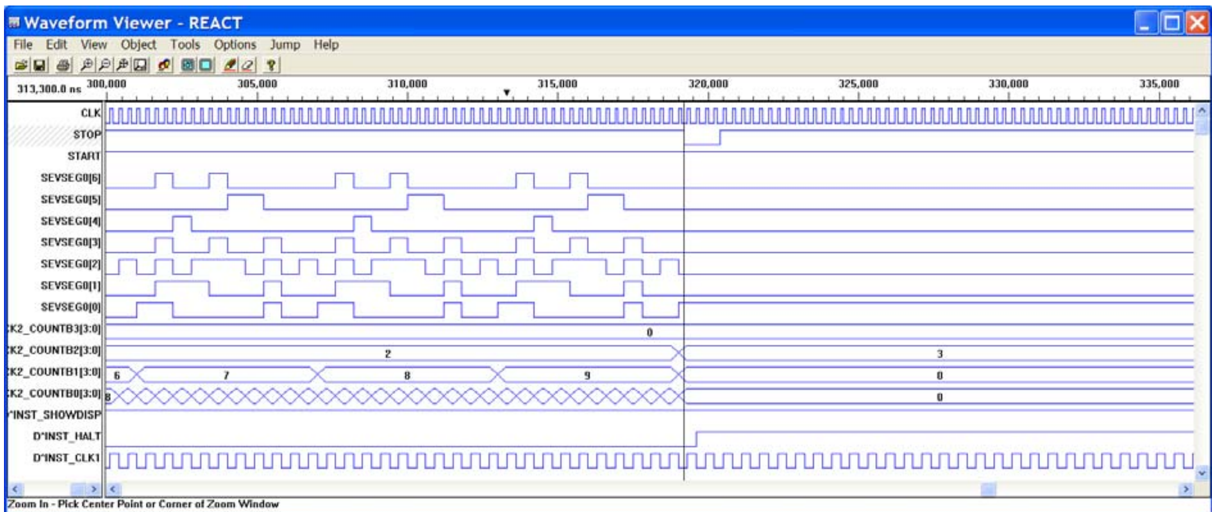

Figure 6. The Stop button being pushed to stop the counter incrementing.

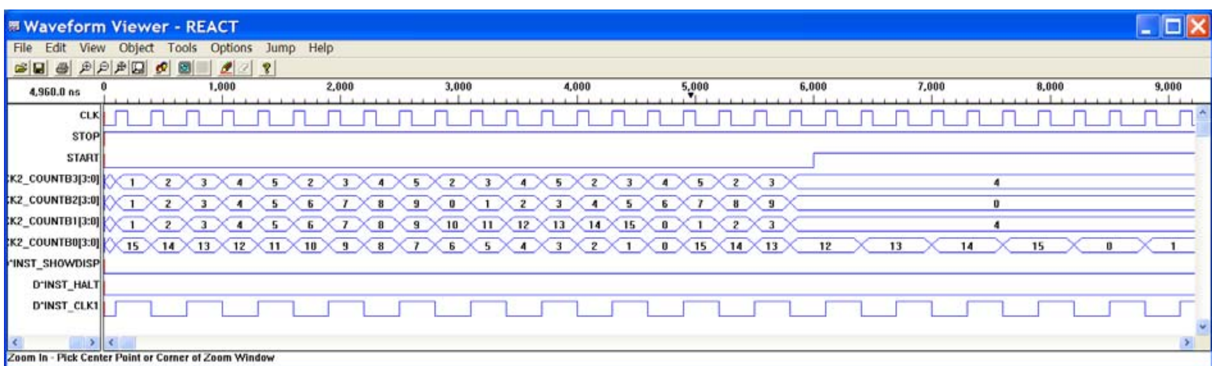

Figure 7. The rapid counting when the Start Button is pushed.

The fitter report when using ispLever4, and the Precision synthesiser with default settings are as follows:

Device_Resource_Summary
~ ~

	Total Available	Used	Available	Utilization	
Dedicated Pins					
Input-Only Pins	..	..	..	-->	..
Clock/Input Pins	2	1	1	-->	50%
I/O Pins	32	30	2	-->	93%
Logic Macrocells	64	51	13	-->	79%
Input Registers	32	0	32	-->	0%
Unusable Macrocells	..	0	..		
CSM Outputs/Total Block Inputs	132	80	52	-->	60%
Logical Product Terms	320	225	95	-->	70%
Product Term Clusters	64	58	6	-->	90%

It can be seen that this realisation uses 13 less macrocells than the first realisation, when the fitting is optimised for speed. One more macrocell is used when the fitting is optimised for area.

When ispLever Classic1.2 is used with the Synplify synthesiser and “area” settings, the following fitting results are obtained.

Device_Resource_Summary
~ ~

	Total Available	Used	Available	Utilization	
Dedicated Pins					
Input-Only Pins	..	..	..	-->	..
Clock/Input Pins	2	1	1	-->	50%
I/O Pins	32	30	2	-->	93%
Logic Macrocells	64	54	19	-->	84%
Input Registers	32	0	32	-->	0%
Unusable Macrocells	..	0	..		
CSM Outputs/Total Block Inputs	132	88	44	-->	66%
Logical Product Terms	320	194	126	-->	60%
Product Term Clusters	64	53	11	-->	82%

This realisation results in more macrocells but less gates being used. The “speed” setting requires the same number of macrocells but needs 232 logical product terms.

Example 2 An FFSK Modulator

This was the VHDL assignment for EE4306 in 2007 and the problem statement is as follows:

Some dedicated radio transmissions use a simple FFS (Fast Frequency Shifting) modulation, where 4800 the Modulation: FFSK (fast frequency shift keying) modulation, where the data rate and the modulated signal are synchronized and a Logic 1 represented by one half a cycle of 2400Hz and a Logic 0 represented by one full cycle of 4800Hz. The Tone frequencies are phase continuous; transitions of the data occur at the zero crossing point (A modem chip producing this data is CML Microcircuits, CMX469A IC.). The required input and output waveforms are shown in figure 1.

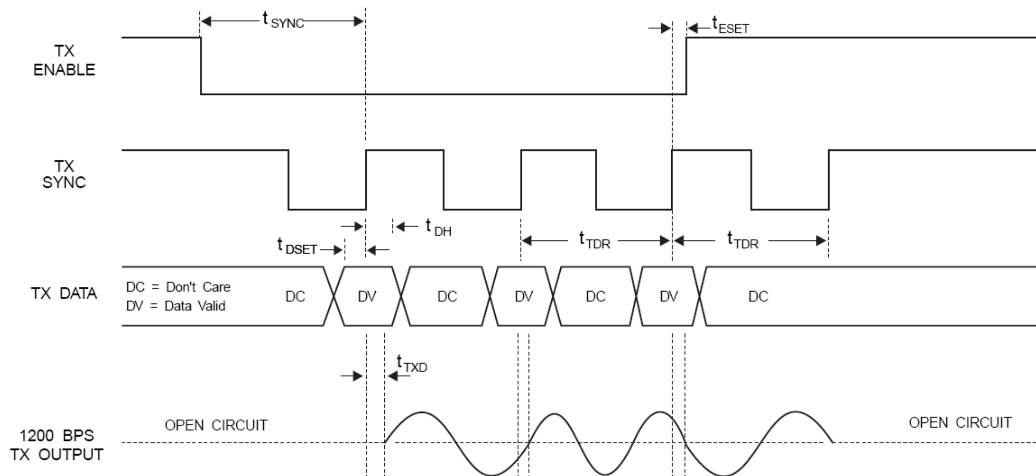

Figure 1. Waveforms for a FFSK transmitter. (from Microcircuits)

The inputs to the FFSK transmitter are a clock of 2 MHz. A TX enable control signal, a serial data input, which is obtained as an RS232 like data stream, using logic level voltages. The TX sync clock and any other clock signals required by your design shown in figure 1 are derived from the 3.6864 MHz clock input. The t_{SYNC} time shown in figure 1, can be as small as needed to ensure that the circuit functions properly. In practice the TX enable signal is generated as a Data_Valid signal by a UART (or equivalent CPLD/FPGA code). The TXdata can be assumed to be synchronised to the TX synch signal, but to allow for timing errors, the TX data must be buffered.

Using the ispMach M4A5 Lattice development boards that are used our labs, in conjunction with an 8 bit DAC like the DAC0800. Write and fully test VHDL code to operate as a FFSK generator. The FFSK generator is to have the following specifications:

1 The clock is to be pin 11 on the IC.

2 Pins 14, 15, 16, 17, 18, 19, 20 and 21 on the M4A5 are connected to the 8 Data inputs of the DAC, with pin 14 being the most significant bit and pin 21 being the least significant bit. The M4A5 is to generate the 8 bit values corresponding to the Sinewave output.

3 The TX_ENABLE is to be connected to Pins 24, and the TX data is to be connected to Pin 26. (See Note below)

4 The pins will be connected by ribbon cable to boards containing the DAC and data inputs.

5 An AD557 8 bit DAC is used as the DAC. This device includes a data latch and thus requires a clock signal to latch the digital data. The clock for latching the data is to be on pin 8 of the iM4a5 and the ribbon cable and is connected to pin 9 (/CE) of the AD557. A Chip select is also required. The chip select is connected to pin 6 of the im4A5 and the ribbon cable and is connected to pin 10 (/CS) of the AD557. The data is latched when the Boolean expression (/CS or /CE) goes Low and the conversion starts then.

© 1990-2008, C. J. Kikkert, James Cook University

Notes:

You will need to be efficient in the way the Sinewave is generated, and the frequency is controlled, otherwise the code will not fit. We have a DAC board and control board that will be used for testing the submissions.

I will use a ispM4A5 development board for generating the TX_Enable and the TX data signals. During the development of the code, I did find a very convenient way of testing the FFSK modulation was by generating suitable TX_enable and TX data signals inside the same CPLD that produces the FFSK. (These is enough spare capacity to allow for this.) It is permissible to use this option for your final submission. Please specify in your submission document and the VHDL code, which option you use, to permit me to accurately test your submission.

Solution

Principles of Operation

The FFSK modulator consists of two parts. The first part being a sinewave generator, which produces the required waveform, and the second part being the control circuitry, which varies the frequency of the sinewave in the required manner and which starts and stops the waveform under the influence of the TX_Enable signal, the Input Data (TX_Data) and the Data Clock (TX_Sync), as shown in figure 1.

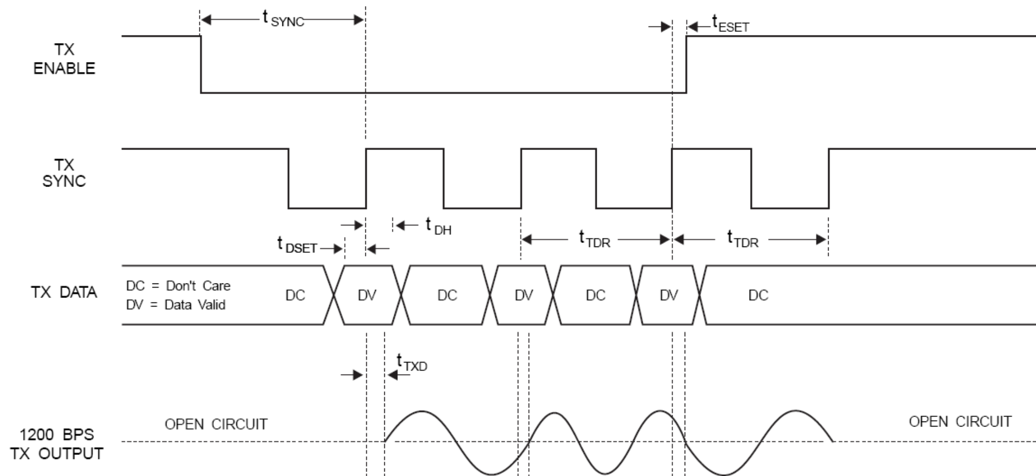

Figure 1. Waveforms for a FFSK transmitter. (from Microcircuits)

Sinewave Generator

The sinewave generator must produce the required output waveform in such a manner that no unwanted signal components are obvious. The waveforms produced are 2400Hz and 4800Hz sinewaves. These are produced by an 8 bit DAC and are transmitted by a Radio. The DAC output in addition to the required signal, produces signal components aliased around the sampling frequency.

A sampling frequency of 8 times the 4800 Hz, i.e. 38.4 kHz will produce an alias at 33.6 kHz (38.4 kHz – 4.8 kHz), which is well outside the typical 10 kHz input bandwidth of the radio and these alias signals will thus be rejected. For this system to

© 1990-2008, C. J. Kikkert, James Cook University

work, one will thus only need to generate 8 samples for the whole sinewave. A sinewave with 8 samples per waveform is easy to generate and in for this example, sinewaves with up to 256 samples per waveform can fit in the CPLD and produce much cleaner waveforms. However 8 samples per sinewave is more than adequate.

It is interesting that no students investigated the minimum sampling requirements as done above.

There are two basic ways by which the sinewave can be generated. Firstly, the whole waveform can be generated in one lookup table. If 8 samples are used for the whole sinewave, then this method is the easiest to implement. Secondly one quadrant of a sinewave can be implemented in a look-up table and the whole sinewave can then be produced by the appropriate logic gating, as shown below:

```
SineWave_P: process(SampleCount, SinQ, TX_Latch)
begin   -- Form Sinewave from quadrants
        if (SampleCount(6 downto 5) = "00") then    --produce sinewave first quadrant
                Phase <= SampleCount(4 downto 0);
                Sine <= SinQ ;
        elsif (SampleCount(6 downto 5) = "01") then--second quadrant
                Phase <= not SampleCount(4 downto 0);
                Sine <= SinQ ;
        elsif (SampleCount(6 downto 5) = "10") then --third quadrant
                Phase <= SampleCount(4 downto 0);
                Sine <= not SinQ ;
        else                                        --fourth quadrant
                Phase <= not SampleCount(4 downto 0);
                Sine <= not SinQ ;
        end if;
end process SineWave_P;
```

In this example, a counter “SampleCount” is used to produce the time samples. A lookup table, generated using an Excel spreadsheet then converts the count to the sinewave quadrant. The two most significant bits (5 and 6) of the SampleCount determine which quadrant the sample is in and does the required flipping of the time axis and Sine amplitude to produce the whole sinewave. The lookup table for a sinewave with an 8 bit output and 128 samples per cycle is shown below.

```
Ttable: process(Phase)
begin   -- This whole table is be generated using Excel one Quadrant only
        case Phase is   --Sine Lookup Table
        when "00000" => SinQ <= X"83"; -- 128Sin(  1.4    deg)
        when "00001" => SinQ <= X"89"; -- 128Sin(  4.2    deg)
        when "00010" => SinQ <= X"8F"; -- 128Sin(  7.0    deg)
        when "00011" => SinQ <= X"95"; -- 128Sin(  9.8    deg)
        when "00100" => SinQ <= X"9C"; -- 128Sin(  12.7   deg)
        when "00101" => SinQ <= X"A2"; -- 128Sin(  15.5   deg)
        when "00110" => SinQ <= X"A8"; -- 128Sin(  18.3   deg)
        when "00111" => SinQ <= X"AE"; -- 128Sin(  21.1   deg)
        when "01000" => SinQ <= X"B3"; -- 128Sin(  23.9   deg)
        when "01001" => SinQ <= X"B9"; -- 1024Sin( 26.7   deg)
        when "01010" => SinQ <= X"BF"; -- 128Sin(  29.5   deg)
        when "01011" => SinQ <= X"C4"; -- 128Sin(  32.3   deg)
        when "01100" => SinQ <= X"C9"; -- 128Sin(  35.2   deg)
        when "01101" => SinQ <= X"CE"; -- 128Sin(  38.0   deg)
        when "01110" => SinQ <= X"D3"; -- 128Sin(  40.8   deg)
        when "01111" => SinQ <= X"D8"; -- 128Sin(  43.6   deg)
        when "10000" => SinQ <= X"DC"; -- 128Sin(  46.4   deg)
        when "10001" => SinQ <= X"E0"; -- 128Sin(  49.2   deg)
        when "10010" => SinQ <= X"E4"; -- 128Sin(  52.0   deg)
        when "10011" => SinQ <= X"E8"; -- 128Sin(  54.8   deg)
        when "10100" => SinQ <= X"EC"; -- 128Sin(  57.7   deg)
        when "10101" => SinQ <= X"EF"; -- 128Sin(  60.5   deg)
        when "10110" => SinQ <= X"F2"; -- 128Sin(  63.3   deg)
```

© 1990-2008, C. J. Kikkert, James Cook University

```
        when "10111" => SinQ <= X"F5"; -- 128Sin(  66.1     deg)
        when "11000" => SinQ <= X"F7"; -- 128Sin(  68.9     deg)
        when "11001" => SinQ <= X"F9"; -- 128Sin(  71.7     deg)
        when "11010" => SinQ <= X"FB"; -- 128Sin(  74.5     deg)
        when "11011" => SinQ <= X"FC"; -- 128Sin(  77.3     deg)
        when "11100" => SinQ <= X"FE"; -- 128Sin(  80.2     deg)
        when "11101" => SinQ <= X"FF"; -- 128Sin(  83.0     deg)
        when "11110" => SinQ <= X"FF"; -- 128Sin(  85.8     deg)
        when others  => SinQ <= X"FF"; -- 128Sin(  88.6     deg)
        end case;
    end process Ttable;
```

In order for the waveform to be continuous as the counter goes from one quadrant to the next, a half a sample offset is required for the sinewave. The last sample in the table is thus at 88.6 degrees and the next sample is at 92.4 degrees and has the amplitude of X"FF" or 127. For the third and fourth quadrants, all the bits of the output are inverted, so that the output is X"00". A value of '1000000' in the first quadrant will thus map as '0111111' in the fourth quadrant. The zero value will thus be half a bit below '1000000', i.e. 127.5. That must be taken into account when generating the sinewave, otherwise distortion will occur at the zero crossover. It is easy to accommodate this in the formulas used for calculating the sinewave. The amplitude of the sinewave can be 128, so that the largest value is 127.5+128 = 255.5, which will be rounded to 255 and the smallest value is 127.5-128 = -0.5, which will be rounded to 0. Alternately a sinewave amplitude of 127.5 can also be used, however that will result in a slightly larger quantisation noise. (See EE3700). The above table has a zero value of 127.5 and a sinewave amplitude of 128.

If the sinewave is generated with the first sample at zero phase, then the a single sample will be required at 90 degrees. When a whole sinewave is stored, this can easily be implemented, however when a quarter sinewave is stored, this is difficult to implement. When 128 or '10000000' is considered to be zero, then if the sinewave is to be symmetrical with that, then the largest value will be 255 and the smallest value is 1. This required the addition of one in quadrant 3 and 4 as shown below:

```
Sine <= not SinQ + "00000001";
```

This requires additional hardware, so that it may require less hardware to use a lookup table for the whole waveform.

Control Logic

The control logic is required to produce the correct frequencies and start and stop the waveforms. The Xtal has a frequency of 3.686400 MHz. To produce a Data clock of 4800 Hz, this Xtal is divided by 768 which is 3x256. It is thus most logical to divide the Xtal frequency by 3 and then further divide that by a binary number to produce the sampling frequency and then divide the sampling frequency in order to provide the Data Clock. If desired the data clock can then be divided further to produce test data and test Tx_Enable signals. To minimise the amount of hardware required, the total number of bits in the frequency divider counters should be kept as small as possible.

If 64 samples are used for the whole sinewave, then for a 2400 Hz sinewave, a sampling clock of 153.6 kHz is required, which can be obtained by dividing the Xtal frequency by 24 and a 4800 Hz sinewave requires a sampling clock of 307.2 kHz, which can be obtained by dividing the Xtal by 12.

© 1990-2008, C. J. Kikkert, James Cook University

There are three different ways that the 2400 Hz and 4800 Hz frequencies can be obtained. The numbers shown below are for a sinewave with 64 samples in the lookup table for the whole sinewave (or 16 samples per quarter waveform).

1. Use a 153.6 kHz sampling frequency and increment the phase counter by one to produce 2400 Hz and increment the phase counter by two to produce 4800 Hz. This has the disadvantage that the effective sampling frequency for the 4800 Hz is reduced to 32 samples per cycle. However that will not cause any unwanted aliases.

   The relevant VHDL code is

```
If (Data = '1') then
        SampleCount <= SampleCount +"0000001";
        -- Increase the phase by one for 2400 Hz
else    -- Increase the phase by two for 4800 Hz
        SampleCount <= SampleCount +"0000010";
end if;
```

2. Use a 307.2 kHz sampling frequency to produce the 2400 Hz and a 614.4 kHz sampling frequency to produce the 4800 Hz. The number of samples per cycle is thus kept the same but the sampling frequency is changed by simply selecting the appropriate tapping point on the frequency divider chain. The relevant code is:

```
clockdiv: process       -- Generates the Sampling Clock frequency of 307.2 kHz and 153.6 KHz.
        Begin           -- This is a circuit for a frequency divide by 12 = divide by 3 follower by a divide by 4
                        -- Using two cascaded counters is less hardware than a single divide by 12
        wait until rising_edge (Clk);
                if (XCount1 = "10") then -- if value = 2 then reset counter to 0. to give divide by 3
                        XCount1 <= "00";
                        XCount2 <= XCount2 + "01";    -- divide by 4 to give 307 KHz
                else
                        XCount1 <= XCount1 + "01";
                        XCount2 <= XCount2;           -- hold the value
                end if;
                if (Data = '1') then
                        SClk <= XCount2(2);    -- use a divide by 6=3x2 to give 614.4 kHz
                else
                        SClk <= not XCount2(1);        --use a divide by 12=3x4 to give 307.2 kHz
                --inversion required to ensure no cp missed when data is changed (adds transition)
                end if;
        end process clockdiv;
```

3. Use a 307.2 kHz sampling frequency and for a 64 sample per cycle waveform, consisting of 16 samples per quadrant. The phase value for the sinewave is contained in a counter called: SampleCount. Use SampleCount(4 downto 1) for 2400Hz and SampleCount(3 downto 0) for 4800 Hz, as shown below:

```
SineWave_P: process(SampleCount, SinQ, TX_Latch)
begin   -- Form Sinewave from quadrants
        if (Data = 1) then;      --want 2400 Hz
                if(SampleCount(6 downto 5) = "00") then      --produce sinewave first quadrant
                        Phase <= SampleCount(4 downto 1);
                        Sine <= SinQ ;
                elsif(SampleCount(6 downto 5) = "01") then  --second quadrant
                        Phase <= not SampleCount(4 downto 1);
                        Sine <= SinQ ;
                elsif(SampleCount(6 downto 5) = "10") then  --third quadrant
                        Phase <= SampleCount(4 downto 1);
                        Sine <= not SinQ ;
                else                                         --fourth quadrant
                        Phase <= not SampleCount(4 downto 1);
                Sine <= not SinQ ;
                end if;
        else    -- want 4800Hz copy code above with change in SampleCount vector array values
                if(SampleCount(5 downto 4) = "00") then      --produce sinewave first quadrant
                        Phase <= SampleCount(3 downto 0);
```

© 1990-2008, C. J. Kikkert, James Cook University

```
                Sine <= SinQ ;
          elsif(SampleCount(5 downto 4) = "01") then  --second quadrant
                Phase <= not SampleCount(3 downto 0);
                Sine <= SinQ ;
          elsif(SampleCount(5 downto 4) = "10") then  --third quadrant
                Phase <= SampleCount(3 downto 0);
                Sine <= not SinQ ;
          else                                        --fourth quadrant
                Phase <= not SampleCount(3 downto 0);
          Sine <= not SinQ ;
          end if;
    end process SineWave_P;
```

External Synchronisation

The external input data must be latched. This TX_Sync Data clock can be generated internally, as part of the sampling counting process, or it can be applied externally. This TX_Sync clock must latch both the TX_Data and the TX_Enable, so that when changes to the TX_Enable occur the output waveforms are not changed abruptly.

As shown in figure1, the sinewave must start at zero, so that the first active value must correspond to the sampling counter having a zero value and correspond to the first entry in the lookup table. The new Data is latched by the TX_Sync Clock so that the sinewave starts at the TX_Sync Clock pulse.

When the TX_Sync clock is generated externally, then the sample counter may be at a value which does not correspond to approximately 0 or 180 degrees. The TX_Sync clock must thus reset the sample counter to the first sample of the first or third quadrant. If the sample counter is closest to 0 degrees, then it is set to start in the first quadrant, if it is closest to 180 degrees then it is set to start in the third quadrant.

The relevant VHDL code for this is:

```
SCounter_P: process   --Counter for the Sampled Data Phase
begin
wait until rising_edge (SClk);  -- SClk is the sample clock
      DClkDel <= DClk;        --Data Clock delayed by one SClk pulse
      if (TX_Latch = '1') then
            SampleCount <= "0000001";
      -- Set Counter to one, to make Sine start at close to 0 degrees.
      --Under normal conditions one CP delay
      elsif ((DClkDel = '0') and (DClk = '1')) then
      --DClk has just had a 0->1 transition New Data available
            if ((SampleCount(6 downto 5) = "11") or (SampleCount(6 downto 5) = "00")) then
                  -- SampleCount closest to 0000001 i.e. close to 0 degrees, allowing for one CP delay
                  SampleCount <= "0000001";
                  -- Set Counter to zero. to make Sine start at close to 0 degrees.
                  Data_Latch <= DatIn;  --Latch the input data
                  TX_Latch <= TX_Enable; --Latch TX_Enable to start Sinewave correctly
            else        --SampleCount closest to 1000000
                  SampleCount <= "1000000";
                  -- Set Counter to 180 degrees to make Sine start at close to 180 degrees.
            end if;
      else        -- generate the sinewave
            SampleCount <= SampleCount +"0000001";    -- Increase the phase
      end if;
end process SCounter_P;
```

CODE Examples

Both these code examples work well. The first example of VHDL code uses a quarter wave sinewave with 128 samples per cycle. Wrap around for line overflow has been used to ensure no text is lost. The code includes. The code is as follows:

© 1990-2008, C. J. Kikkert, James Cook University

```
--This is a Sinewave Generator based on a Lookup table containing the first quadrant of a Sine wave.
--A half a sampling period offset is applied to ensure that each quadrant is identical.
--The lookup table is generated using an Excel spreadsheet, that is directly copied into the code.
--Author C. J. Kikkert September 2007
	library ieee;
	use ieee.std_logic_1164.all;
	use ieee.std_logic_arith.all;
	use ieee.std_logic_unsigned.all;

	entity SineTable_E is
	port (Phase: in std_logic_vector (4 downto 0);	--Quarter wave Phase for lookup table
			SinQ: out std_logic_vector (7 downto 0));		--Quarter wave sinewave from lookup
	table
	end SineTable_E;

	architecture SineTable_A of SineTable_E is
	begin
		Ttable: process(Phase)
		begin	-- This whole table is be generated using Excel one Quadrant only
			case Phase is	--Sine Lookup Table
			when "00000" => SinQ <= X"83"; -- 1024Sin(	1.4	deg)
			when "00001" => SinQ <= X"89"; -- 1024Sin(	4.2	deg)
			when "00010" => SinQ <= X"8F"; -- 1024Sin(	7.0	deg)
			when "00011" => SinQ <= X"95"; -- 1024Sin(	9.8	deg)
			when "00100" => SinQ <= X"9C"; -- 1024Sin(	12.7	deg)
			when "00101" => SinQ <= X"A2"; -- 1024Sin(	15.5	deg)
			when "00110" => SinQ <= X"A8"; -- 1024Sin(	18.3	deg)
			when "00111" => SinQ <= X"AE"; -- 1024Sin(	21.1	deg)
			when "01000" => SinQ <= X"B3"; -- 1024Sin(	23.9	deg)
			when "01001" => SinQ <= X"B9"; -- 1024Sin(	26.7	deg)
			when "01010" => SinQ <= X"BF"; -- 1024Sin(	29.5	deg)
			when "01011" => SinQ <= X"C4"; -- 1024Sin(	32.3	deg)
			when "01100" => SinQ <= X"C9"; -- 1024Sin(	35.2	deg)
			when "01101" => SinQ <= X"CE"; -- 1024Sin(	38.0	deg)
			when "01110" => SinQ <= X"D3"; -- 1024Sin(	40.8	deg)
			when "01111" => SinQ <= X"D8"; -- 1024Sin(	43.6	deg)
			when "10000" => SinQ <= X"DC"; -- 1024Sin(	46.4	deg)
			when "10001" => SinQ <= X"E0"; -- 1024Sin(	49.2	deg)
			when "10010" => SinQ <= X"E4"; -- 1024Sin(	52.0	deg)
			when "10011" => SinQ <= X"E8"; -- 1024Sin(	54.8	deg)
			when "10100" => SinQ <= X"EC"; -- 1024Sin(	57.7	deg)
			when "10101" => SinQ <= X"EF"; -- 1024Sin(	60.5	deg)
			when "10110" => SinQ <= X"F2"; -- 1024Sin(	63.3	deg)
			when "10111" => SinQ <= X"F5"; -- 1024Sin(	66.1	deg)
			when "11000" => SinQ <= X"F7"; -- 1024Sin(	68.9	deg)
			when "11001" => SinQ <= X"F9"; -- 1024Sin(	71.7	deg)
			when "11010" => SinQ <= X"FB"; -- 1024Sin(	74.5	deg)
			when "11011" => SinQ <= X"FC"; -- 1024Sin(	77.3	deg)
			when "11100" => SinQ <= X"FE"; -- 1024Sin(	80.2	deg)
			when "11101" => SinQ <= X"FF"; -- 1024Sin(	83.0	deg)
			when "11110" => SinQ <= X"FF"; -- 1024Sin(	85.8	deg)
			when others  => SinQ <= X"FF"; -- 1024Sin(	88.6	deg)

		end case;
		end process Ttable;
	end SineTable_A;


	-- main program
	-- This uses the quarter wave sinewave component with 256 samples per period to generate an FFSK signal.
	-- The Xtal clock is 3.6864 MHz. The data rate is 4800 Hz. For 256 samples per second,
	-- The sampling rate is 256*4800 = 1.2288 MHz. This is one third of the Xtal frequency.
	-- For the 4800 Hz waveform the sampling clock to the Sinewave generator is 1.2288 MHz.
	-- For the 2400 Hz waveform it is half that frequency. i.e. 614.4 kHz
	--	Author C. J. Kikkert September 2007.

	-- Note for demonstration of the waveforms, DClk is made an output pin as well, normally it would be
	-- an internal signal only.

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

```
use ieee.std_logic_arith.all;

entity FFSK_E is
	port(Clk: in std_logic;
		CS: out std_logic;
		SClk: inout std_logic ;
		DClk: inout std_logic ;
	SampleCount: inout std_logic_vector (6 downto 0);	--128 steps per sinewave, Sampling Clock for
generating the sinewave.
		TX_Enable: inout std_logic;	--Control for start of transmission (Normally inputtemporary
provided as output for testing)
		DatIn: inout std_logic;		--Serial Input Data (normally input temporary provided as
output for testing)
		Sine: out std_logic_vector (7 downto 0));	--8 bit DAC for FFSK output
	attribute loc : string;
	attribute loc of Clk: signal is "P11" ;
	attribute loc of TX_Enable: signal is "P24" ;	-- Pin allocation for TX_Enable falling edge initiates
sinewave.
	attribute loc of DatIn: signal is "P26";	-- TX_Data
	attribute loc of SClk: signal is "P8" ;	--Sampling Clock = !RW control line low for DAC reading the
data from M4A5
	attribute loc of DClk: signal is "P28" ;	--Data Clock for latching the input data and generating the
waveform
	attribute loc of CS: signal is "P6" ;	--|CS clock for latching data to DAC must be low always. (High
deselects DAC)
	attribute loc of Sine:signal is "P14, P15, P16, P17, P18, P19, P20, P21" ;-- Sine Output DB7 to DB0
end FFSK_E;

architecture FFSK_A of FFSK_E is
	component SineTable_E
	port (Phase: in std_logic_vector;	--Quarter wave Phase for lookup table
			SinQ: out std_logic_vector);		--Quarter wave sinewave from lookup table
	end component;

	signal XCount1: std_logic_vector (1 downto 0);	--Xtal Divide by 3 to produce 2x614.4 kHz from 3.6864
MHz.
	signal XCount2: std_logic_vector (1 downto 0);	-- divide by 4 to produce 614.4 kHz and 307.2 kHz
sampling waveforms
	signal DatCount: std_logic_vector (10 downto 0);	--Divide by 16*256 to produce DataClock Data
and Enable test signals.
--	signal SampleCount: std_logic_vector (6 downto 0);	--128 steps per sinewave, Sampling Clock for
generating the sinewave.
	signal SinQ: std_logic_vector (7 downto 0);	--Quarter wave Sine
	signal Phase: std_logic_vector(4 downto 0);	--Phase for Sinewave
	signal Data_Latch, TX_Latch, DClkDel: std_logic; -- Latched data and Data Clock delayed by one SClk
pulse
--	signal DClk; std_logic;	-- Data Clock for latching the input data and generating the waveform

begin
	Lookup: SineTable_E port map (Phase, SinQ);	-- Sine lookup table mapping

	CS <= '0';	--Constant Set Chip select on DAC to be ON
	clockdiv: process	-- Generates the Sampling Clock frequency of 614.4 kHz and 1.2288 MHz.
	begin
	wait until rising_edge (Clk);
		if (XCount1 = "10") then	-- if value = 2 then reset counter to 0. to give divide by 3
			XCount1 <= "00";
			XCount2 <= XCount2 + "01";	-- divide by 4 to give 307 KHz
		else
			XCount1 <= XCount1 + "01";
			XCount2 <= XCount2;		-- hold the value
		end if;
		if (Data_Latch = '1') then
			SClk <= XCount2(0);	--614.4 kHz
		else
			SClk <= not XCount2(1);	--307.2 kHz inversion required to ensure no cp missed
		end if;
	end process clockdiv;

	DataCounter_P: process	-- This generates the required test signals
	-- Part of this is NOT REQUIRED for external data.
	begin
```

```
	wait until rising_edge(XCount1(1));	-- 307 KHz clock
		if (DatCount >= "10100000000") then
			DatCount <= "00000000000";
		else
			DatCount <= DatCount + "00000000001";
		end if;
	end process DataCounter_P;

	Gating_P: process(DatCount)
	begin
		DClk <= DatCount(7);				-- 4800 Hz data clock
		DatIn <= DatCount(8);		-- RS232 Data input, now generated internal test only
		TX_Enable <= DatCount(9) and DatCount(8);	-- Data Valid Input, now generated internal test
only
	end process Gating_P;

	SCounter_P: process	--Counter for the Sampled Data Phase
	begin
	wait until rising_edge (SClk);	-- SClk is the sample clock
		DClkDel <= DClk;		--Data Clock delayed by one SClk pulse
		if (TX_Latch = '1') then
				SampleCount <= "0000001";	-- Set Counter to one. to make Sine start at
close to 0 degrees. Under normal conditions one CP delay
		elsif ((DClkDel = '0') and (DClk = '1')) then	--DClk has just had a 0->1 transition New Data
available
			if ((SampleCount(6 downto 5) = "11") or (SampleCount(6 downto 5) = "00")) then
				-- SampleCount closest to 0000000 i.e. 0 degrees
				SampleCount <= "0000001";	-- Set Counter to zero. to make Sine start at
close to 0 degrees.
--				Data_Latch <= DatIn;	--Latch the input data
--				TX_Latch <= TX_Enable; --Latch TX_Enable to start Sinewave correctly

			else --SampleCount closest to 1000000
				SampleCount <= "1000000";	-- Set Counter to zero. to make Sine start at
close to 180 degrees.
			end if;
		else -- generate the sinewave
			SampleCount <= SampleCount +"0000001";	-- Increase the phase
		end if;
	end process SCounter_P;

	SineWave_P: process(SampleCount, SinQ, TX_Latch)
	begin	-- Form Sinewave from quadrants
		if (TX_Latch = '1') then --Latched TxEnable = 1 or new data start with Phase =0
			Sine <= "10000000";	-- Sine = zero.
		elsif(SampleCount(6 downto 5) = "00") then	--produce sinewave
			Phase <= SampleCount(4 downto 0);
			Sine <= SinQ ;
		elsif(SampleCount(6 downto 5) = "01") then
			Phase <= not SampleCount(4 downto 0);
			Sine <= SinQ ;
		elsif(SampleCount(6 downto 5) = "10") then
			Phase <= SampleCount(4 downto 0);
			Sine <= not SinQ ;
		else
			Phase <= not SampleCount(4 downto 0);
			Sine <= not SinQ ;
		end if;
	end process SineWave_P;

	DataLatch_P: process --Code will not fit if this is used to latch data
	begin
	wait until rising_edge(DClk);
			Data_Latch <= DatIn;	--Latch the input data
			TX_Latch <= TX_Enable; --Latch TX_Enable to start Sinewave correctly
	end process DataLatch_P;

end FFSK_A;
```

Second Realisation

The second code example uses a whole sinewave of 64 samples per cycle. This code does not include the sample counter synchronisation as the Data and Tx_Enable waveforms are generated internally.

```
-- An FFSK Module for EE4306 Assignment 2007.
-- Author C. J. Kikkert, August 2007.
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity SineTable_E is
port (Phase: in std_logic_vector (5 downto 0);  --Quarter wave Phase for lookup table
      Sin: out std_logic_vector (7 downto 0));        --Quarter wave sinewave from lookup table
end SineTable_E;

architecture SineTable_A of SineTable_E is
begin
   Ttable:process(Phase)
   begin   -- This whole table is be generated using Excel
      case Phase is   --Sine Lookup Table
      when "000000" => Sin <= X"80"; -- 128Sin(   0.0 deg)
      when "000001" => Sin <= X"8C"; -- 128Sin(   5.6 deg)
      when "000010" => Sin <= X"99"; -- 128Sin(   11.3    deg)
      when "000011" => Sin <= X"A5"; -- 128Sin(   16.9    deg)
      when "000100" => Sin <= X"B1"; -- 128Sin(   22.5    deg)
      when "000101" => Sin <= X"BC"; -- 128Sin(   28.1    deg)
      when "000110" => Sin <= X"C7"; -- 128Sin(   33.8    deg)
      when "000111" => Sin <= X"D1"; -- 128Sin(   39.4    deg)
      when "001000" => Sin <= X"DA"; -- 128Sin(   45.0    deg)
      when "001001" => Sin <= X"E2"; -- 128Sin(   50.6    deg)
      when "001010" => Sin <= X"EA"; -- 128Sin(   56.3    deg)
      when "001011" => Sin <= X"F0"; -- 128Sin(   61.9    deg)
      when "001100" => Sin <= X"F5"; -- 128Sin(   67.5    deg)
      when "001101" => Sin <= X"FA"; -- 128Sin(   73.1    deg)
      when "001110" => Sin <= X"FD"; -- 128Sin(   78.8    deg)
      when "001111" => Sin <= X"FE"; -- 128Sin(   84.4    deg)
      when "010000" => Sin <= X"FF"; -- 128Sin(   90.0    deg)
      when "010001" => Sin <= X"FE"; -- 128Sin(   95.6    deg)
      when "010010" => Sin <= X"FD"; -- 128Sin(   101.3   deg)
      when "010011" => Sin <= X"FA"; -- 128Sin(   106.9   deg)
      when "010100" => Sin <= X"F5"; -- 128Sin(   112.5   deg)
      when "010101" => Sin <= X"F0"; -- 128Sin(   118.1   deg)
      when "010110" => Sin <= X"EA"; -- 128Sin(   123.8   deg)
      when "010111" => Sin <= X"E2"; -- 128Sin(   129.4   deg)
      when "011000" => Sin <= X"DA"; -- 128Sin(   135.0   deg)
      when "011001" => Sin <= X"D1"; -- 128Sin(   140.6   deg)
      when "011010" => Sin <= X"C7"; -- 128Sin(   146.3   deg)
      when "011011" => Sin <= X"BC"; -- 128Sin(   151.9   deg)
      when "011100" => Sin <= X"B1"; -- 128Sin(   157.5   deg)
      when "011101" => Sin <= X"A5"; -- 128Sin(   163.1   deg)
      when "011110" => Sin <= X"99"; -- 128Sin(   168.8   deg)
      when "011111" => Sin <= X"8C"; -- 128Sin(   174.4   deg)
      when "100000" => Sin <= X"80"; -- 128Sin(   180.0   deg)
      when "100001" => Sin <= X"74"; -- 128Sin(   185.6   deg)
      when "100010" => Sin <= X"67"; -- 128Sin(   191.3   deg)
      when "100011" => Sin <= X"5B"; -- 128Sin(   196.9   deg)
      when "100100" => Sin <= X"4F"; -- 128Sin(   202.5   deg)
      when "100101" => Sin <= X"44"; -- 128Sin(   208.1   deg)
      when "100110" => Sin <= X"39"; -- 128Sin(   213.8   deg)
      when "100111" => Sin <= X"2F"; -- 128Sin(   219.4   deg)
      when "101000" => Sin <= X"26"; -- 128Sin(   225.0   deg)
      when "101001" => Sin <= X"1E"; -- 128Sin(   230.6   deg)
      when "101010" => Sin <= X"16"; -- 128Sin(   236.3   deg)
      when "101011" => Sin <= X"10"; -- 128Sin(   241.9   deg)
      when "101100" => Sin <= X"0B"; -- 128Sin(   247.5   deg)
      when "101101" => Sin <= X"06"; -- 128Sin(   253.1   deg)
      when "101110" => Sin <= X"03"; -- 128Sin(   258.8   deg)
      when "101111" => Sin <= X"02"; -- 128Sin(   264.4   deg)
```

© 1990-2008, C. J. Kikkert, James Cook University

```
        when "110000" => Sin <= X"01"; -- 128Sin(   270.0   deg)
        when "110001" => Sin <= X"02"; -- 128Sin(   275.6   deg)
        when "110010" => Sin <= X"03"; -- 128Sin(   281.3   deg)
        when "110011" => Sin <= X"06"; -- 128Sin(   286.9   deg)
        when "110100" => Sin <= X"0B"; -- 128Sin(   292.5   deg)
        when "110101" => Sin <= X"10"; -- 128Sin(   298.1   deg)
        when "110110" => Sin <= X"16"; -- 128Sin(   303.8   deg)
        when "110111" => Sin <= X"1E"; -- 128Sin(   309.4   deg)
        when "111000" => Sin <= X"26"; -- 128Sin(   315.0   deg)
        when "111001" => Sin <= X"2F"; -- 128Sin(   320.6   deg)
        when "111010" => Sin <= X"39"; -- 128Sin(   326.3   deg)
        when "111011" => Sin <= X"44"; -- 128Sin(   331.9   deg)
        when "111100" => Sin <= X"4F"; -- 128Sin(   337.5   deg)
        when "111101" => Sin <= X"5B"; -- 128Sin(   343.1   deg)
        when "111110" => Sin <= X"67"; -- 128Sin(   348.8   deg)
        when others      => Sin <= X"74"; -- 128Sin( 354.4   deg)
        end case;
    end process Ttable;
end SineTable_A;



-- main program
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity Sinewave is
    port(Clk: in std_logic;
        CS: out std_logic;
        DClk: inout std_logic ;
        TX_Enable: inout std_logic;      --Control for start of transmission (Normally inputtemporary provided as
output for testing)
        DatIn: inout std_logic;          --Serial Input Data (normally input temporary provided as output for testing)
        Sine: out std_logic_vector (7 downto 0));    --8 bit DAC for FFSK output
    attribute loc : string;
    attribute loc of Clk: signal is "P11" ;
    attribute loc of TX_Enable: signal is "P24" ;    -- Pin allocation for TX_Enable falling edge initiates sinewave.
    attribute loc of DatIn: signal is "P26";     -- TX_Data
    attribute loc of DClk: signal is "P8" ; --!RW control line low for DAC reading the data from M4A5
    attribute loc of CS: signal is "P6" ;    --|CS clock for latching data to DAC must be low always. (High deselects
DAC)
    attribute loc of Sine:signal is "P14, P15, P16, P17, P18, P19, P20, P21" ;  -- Sine Output DB7 to DB0
end Sinewave;

architecture Sinewave_arc of Sinewave is
    component SineTable_E
    port (Phase: in std_logic_vector;   --Quarter wave Phase for lookup table
            Sin: out std_logic_vector);      --Quarter wave sinewave from lookup table
    end component;

    signal Count: std_logic_vector (5 downto 0);     --64 phase steps in whole sinewave
    signal XCount: std_logic_vector (3 downto 0);    --Xtal Divide by 12 to produce 307 kHz from 3.6864 MHz.
    signal DatCount: std_logic_vector (7 downto 0); --Divide by 256 to produce Data and Enable test signals.

begin
    Lookup: SineTable_E port map (Count, Sine); -- Sine lookup table mapping

    CS <= '0';
    clockdiv: process
    begin
    wait until rising_edge(Clk);
        if (XCount = X"B") then      -- if value = 11 then reset counter to 0.
            XCount <= X"0";
        else
            XCount <= XCount + X"1";
        end if;
    end process clockdiv;


    Gating_P: process (XCount(3), DatCount)
```

```
begin
   DClk <= XCount(3);
   DatIn <= DatCount(5);       -- RS232 Data input, now generated internal
   TX_Enable <= DatCount(7);   -- Data Valid Input, now generated internal
end process Gating_P;

DataCounter_P: process  -- This generates the required test signals
begin
wait until rising_edge(DClk);
   if (DatCount >= "101000000") then
      DatCount <= "00000000";
   else
      DatCount <= DatCount + "00000001";
   end if;
end process DataCounter_P;

counter_P: process
begin
wait until rising_edge(DClk);   -- DCount(3) is the sample clock
   if (TX_Enable = '1') then
      Count <= "000000";  -- Set Counter to zero. to make output = zero.
   elsif (DatIn = '1') then    -- TX_Enable = 0 ie generate sinewave
      Count <= Count +"000001";   -- 2400 Hz
      else
      Count <= Count + "000010";  -- 4800 Hz
   end if;
end process counter_P;
end Sinewave_arc;
```

Example 3 An Electronic Lock

A cheap key-pad obtainable from Dick Smith Electronics or Jaycar is to be used together with a ispLSI2031 or M4A5 CPLD for an electronic lock.

The requirements for this lock are:

1. It must have at least 6 digit key code, ie 080903 is a suitable code
2. It must provide an output high when the door is to be opened. This logic level high then drives a door latch amplifier, whose design is not part of this assignment.
3. The lock is to have a Keypad reset or keypad clear, operated by the # key
4. It must provide a logic level high alarm output when three successive # key pressed have been made without a correct key code.
5. The correct key code clears the alarm counter and also turns off any alarm.
6. For the demo version only, the * key is to also clear the alarm counter and turn off the alarm.
7. Complementary outputs to drive a buzzer are to be provided. These use any suitable AC (clock derived) signal in a push-pull arrangement.
8. The master keycode must be able to be changed easily. A suitable test master codes is 892003.

To make the required CPLD pin connections compatible with these test boards, you should use the following pin assignments:

The clock is connected to pin 11.

The Row scanning lines for the keypad are to be connected to pin 30 for the top row, pin 25 for the second row from the top, pin 26 for the third row from the top and pin 28 for the bottom row.

© 1990-2008, C. J. Kikkert, James Cook University

The Column scanning lines for the keypad are to be connected to pin 29 for the left column, pin 31 for the middle column and pin 27 for the right column.

The Door Open signal is to be connected to pin 3.

The Alarm signal is to be connected to pin 4.

The PBuzzer signal is to be connected to pin 5 and its complementary NBuzzer signal is to be connected to pin 6.

For verification of the design the length of correct keystrokes counter is to be connected to pins 44 to 41, with pin 41 being the least significant bit.

Keypad information:

Connector information:

Keypad connections	1	2	3	4	5	6	7	8
Function	NC	Col1	Row0	Col0	Row3	Col2	Row2	Row1
CPLD pin number		31	30	29	28	27	26	25

Row0 is at the top (keys 1,2,3) and Row3 is at the bottom (keys *, 0, #)

Col0 is on the left (keys 1, 4, 7, *) and Col2 is on the right (keys 3, 6, 9, #)

Pressing a key shorts the corresponding row and column together.

```
-- A design for an Electronic Lock using a Dick Smith Keypad
-- and an ispMach4A5 CPLD device on the development board
--
-- Author C. J. Kikkert Aug 2003
-- for assignment for EE4306
--
-- Keypad connections Pin 1 to 8:       NC,Col1,Row0,Col0,Row3,Col2,Row2,Row1
-- Device pins                          1   2    3    4    5    6    7    8
-- Corresponding CPLD required      25   26   27   28   29   30   31 for Mach4
-- Corresponding CPLD required      31   30   29   28   27   26   25 for ispLSI2032
-- Row0 is at the top (keys 1,2,3) and Row3 is at the bottom (keys *, 0, #)
-- Col0 is on the left (keys 1, 4, 7, *) and Col2 is on the right (keys 3, 6, 9, #)
-- Pressing a key shorts the corresponding row and column together.
-- The inputs are pulled high on all pins for the development board.
```

© 1990-2008, C. J. Kikkert, James Cook University

```
-- Apply a pulsed LOW scan to the columns and observe the corresponding row inputs.
-- So that No key press results in all inputs high.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity KeyLock is
port(   Clk: in std_logic;
        Row: in std_logic_vector (3 downto 0);          -- 4 rows
        Col: inout std_logic_vector (2 downto 0);       -- 3 columns
--      SevSegOut: out std_logic_vector (6 downto 0);  -- Seven segment display LS Digit test only
        State: inout integer range 0 to 31;       --State numbers for the lock 2x number digits plus error, reset.
        NumTries: inout std_logic_vector (2 downto 0);

--      ScanCount: inout integer range 0 to 7;
        Alarm, PBuzzer: inout std_logic;
        NBuzzer, Dopen: out std_logic);
-- pin assignements here
        attribute loc : string;
        attribute loc of Clk : signal is "p11";
        attribute loc of Row : signal is "p28 p30 p31 p26";    -- bottom to top. 2nd right Display
        attribute loc of Col : signal is "p29 p25 p27";        -- left to right 2nd right Display
        attribute loc of Dopen : signal is "p37";
        attribute loc of Alarm : signal is "p36";
        attribute loc of PBuzzer : signal is "p38";
        attribute loc of NBuzzer : signal is "p39";
--      attribute loc of SevSegOut:signal is "P2, P3, P4, P5, P6, P7, P8";--pin allocation of LSI2032 display
        attribute loc of State : signal is "p15 p14 p19 p20 p21";      --for test only
        attribute loc of NumTries : signal is "p16 p17 p18";           --for test only
end;

architecture KeyLock_Arch of KeyLock is
--signal Count: std_logic_vector (1 downto 0);   -- 2 bit counter to scan Columns
signal Key: integer range 0 to 15; -- Key value
signal KeyP: integer range 0 to 15; -- Key pressed on keyboard
signal ScanCount: integer range 0 to 3;
--signal NumTries: std_logic_vector (2 downto 0);

--Enter the required key code here, 10 digits
constant Master1 :integer := 0;                 --required key combination
constant Master2 :integer := 7;
constant Master3 :integer := 4;
constant Master4 :integer := 7;
constant Master5 :integer := 8;
constant Master6 :integer := 1;
constant Master7 :integer := 4;
constant Master8 :integer := 2;
constant Master9 :integer := 5;
constant Master10 :integer := 9;

begin
        ColGen_P: process
        begin
        wait until rising_edge(Clk);
                case Col is
                        when "011" => Col <= "101";   --pull down left col
                        when "101" => Col <= "110";   --pull down middle col
                        when others => Col <= "011";  --pull down right col and outside start
                end case;
        end process ColGen_P;

        KeyScan_p:      process
        begin
        wait until falling_edge(Clk);
        --This uses the Count value to generate the Low Row signal and check if a button is pressed
        --by checking if a Low appears on the relevant column.
        --At each change in the Count value, the corresponding Key is determined
        --Note we will have 3 Key=15 values per one Non15 Key values when a key is pressed.
        --This is a combinational logic process
        --Scancount is the number of column scans done without a key-press. Scancount <3 for keypress
```

```
	--So if Scancount = 3 then no valid key has been pressed
	if (Row = "0111") then-- Top Row
		case Col is
			when "110" => KeyP <= 10; ScanCount <= 0;		-- key * pressed Clear alarm
			--	only for open uni display unit. Normally keep alarm on till correct key
			when "101" => KeyP <= 0; ScanCount <= 0; -- key 0 pressed
			when "011" => KeyP <= 11; ScanCount <= 0;		-- key # pressed Keypad reset
			when others =>		if(ScanCount < 3) then ScanCount <= ScanCount +1;
						else ScanCount <= 3;
						end if;
					-- no key or more than one keys pressed
			end case;
	elsif (Row = "1011") then		-- Second Row from top
			case Col is
				when "110" => KeyP <= 7; ScanCount <= 0;	-- key 7 pressed
				when "101" => KeyP <= 8; ScanCount <= 0;	-- key 8 pressed
				when "011" => KeyP <= 9; ScanCount <= 0;	-- key 9 pressed
				when others => if(ScanCount < 3) then ScanCount <= ScanCount +1; else
ScanCount <= 3; end if;
					-- no key or more than one keys pressed
			end case;
		elsif (Row = "1101") then		-- Third row from top
			case Col is
				when "110" => KeyP <= 4; ScanCount <= 0;	-- key 4 pressed
				when "101" => KeyP <= 5; ScanCount <= 0;	-- key 5 pressed
				when "011" => KeyP <= 6; ScanCount <= 0;	-- key 6 pressed
				when others => if(ScanCount < 3) then ScanCount <= ScanCount +1; else
ScanCount <= 3; end if;
					-- no key or more than one keys pressed
			end case;
		elsif (Row = "1110") then		-- Bottom row
			case Col is
				when "110" => KeyP <= 1; ScanCount <= 0;	-- key 1 pressed
				when "101" => KeyP <= 2; ScanCount <= 0;	-- key 2 pressed
				when "011" => KeyP <= 3; ScanCount <= 0;	-- key 3 pressed
				when others => if(ScanCount < 3) then ScanCount <= ScanCount +1; else
ScanCount <= 3; end if;
					-- no key or more than one keys pressed
			end case;
		else --Wrong row set to no key pressed
				if(ScanCount < 3) then ScanCount <= ScanCount +1; else ScanCount <= 3;
end if;
					-- no key or more than one keys pressed
		end if;
	end process Keyscan_p;

	KeyStore_p: process
	-- This stores the last key pressed and if no key press has occurred for 8 scan count, then a nokey is
stored.
	begin
		wait until rising_edge(Clk);
		if (ScanCount < 3) then
			Key <= KeyP;	--Key value for code check
		else
			Key <= 15;	--no key pressed
		end if;
	end process	KeyStore_p;

--	KeyPressDisplay_p: process
--	begin
--		wait until rising_edge(Clk);
--		Lab0: case State is
--		-- need to have inverse of output coded here since the
--		-- board displays include inverters.
--	when 0 => SevSegOut <= "0000001"; --0
--	when 1 => SevSegOut <= "1001111"; --1
--	when 2 => SevSegOut <= "0010010"; --2
--	when 3 => SevSegOut <= "0000110"; --3
--	when 4 => SevSegOut <= "1001100"; --4
--	when 5 => SevSegOut <= "0100100"; --5
--	when 6 => SevSegOut <= "0100000"; --6
--	when 7 => SevSegOut <= "0001111"; --7
```

© 1990-2008, C. J. Kikkert, James Cook University

```
--      when 8 => SevSegOut <= "0000000";  --8
--      when 9 => SevSegOut <= "0000100";  --9
--      when 10 => SevSegOut <= "0001000";         --A
--      when 11 => SevSegOut <= "1100000";         --b
--      when 12 => SevSegOut <= "0110001";         --C
--      when 13 => SevSegOut <= "1000010";         --d
--                      when 14 => SevSegOut <= "0110000";         --E
--                      when 15 => SevSegOut <= "0111000";         --F
--                      when 16 => SevSegOut <= "1110111";         -- element d only
--                      when 17 => SevSegOut <= "1111110";         -- element g only
--                      when 18 => SevSegOut <= "0111111";         -- element a only
--                      when 19 => SevSegOut <= "0110111";         -- element a and d
--                      when 31 => SevSegOut <= "1100011";         --u
--      when others => SevSegOut <= "1111111";    --display off
--              end case Lab0;
--      end process KeyPressDisplay_p;

        KeyPressCheck_p: process
        begin
        wait until rising_edge(Clk);
                -- Key check
                -- This is best done using state diagrams however VHDL does not handle those.
                if (Key = 10) then                              --*  key turn off alarm in this version only
                        State <= 0;                     -- clear State go to Start (state=0)
                        NumTries <= "000";                      -- clear error counter, demo version only.
                        Dopen <= '0';                           -- release Door open Latch
                else    -- valid numeric key press do key validity check
                        if (State = 0) then     --First Key in code
                                if (Key = Master1) then --correct key
                                        State <= State + 1;
                                elsif ((Key = 11) or (Key = 10) or (Key = 15)) then
                                        --Star or Hash Key pressed or No Key stay in state 0
                                        State <= State;-- no change same key still pressed
                                else
                                        State <= 31;    -- wrong code lock not open
                                end if;
                                Dopen <= '0';                           -- door closed, lock not open

                                NumTries <= NumTries;
                        elsif (State = 1) then
                                if (Key = Master1) then
                                        State <= State;-- no change same key still pressed
                                elsif (Key = 15) then --key released
                                        State<=State + 1;
                                else
                                        State <= 31;    -- wrong code Error State
                                end if;
                                Dopen <= '0';                           -- door closed, lock not open

                                NumTries <= NumTries;
                                -- progresses to next State by key release

                        elsif (State = 2) then  --Second Key in code
                                if (Key = Master2) then --correct key
                                        State<=State + 1;
                                elsif (Key = 15) then --key released
                                        State<=State;
                                else
                                        State <= 31;    -- wrong code Error State
                                end if;
                                Dopen <= '0';                           -- door closed, lock not open

                                NumTries <= NumTries;
                        elsif (State = 3) then
                                if (Key = Master2) then
                                        State <= State;-- no change same key still pressed
                                elsif (Key = 15) then --key released
                                        State<=State + 1;
                                else
                                        State <= 31;    -- wrong code Error State
                                end if;
```

© 1990-2008, C. J. Kikkert, James Cook University

```
			Dopen <= '0';				-- door closed, lock not open

			NumTries <= NumTries;
			-- progresses to next State by key release

		elsif (State = 4) then	--Third Key in code
			if (Key = Master3) then --correct key
				State<=State + 1;
			elsif (Key = 15) then --key released
				State<=State;
			else
				State <= 31;	-- wrong code Error State
			end if;
			Dopen <= '0';				-- door closed, Error State

			NumTries <= NumTries;
		elsif (State = 5) then
			if (Key = Master3) then
				State <= State;-- no change same key still pressed
			elsif (Key = 15) then --key released
				State<=State + 1;
			else
				State <= 31;	-- wrong code Error State
			end if;
			Dopen <= '0';				-- door closed, lock not open

			NumTries <= NumTries;
			-- progresses to next State by key release

		elsif (State = 6) then	--Fourth Key in code
			if (Key = Master4) then --correct key
				State<=State + 1;
			elsif (Key = 15) then --key released
				State<=State;
			else
				State <= 31;	-- wrong code lock not open
			end if;
			Dopen <= '0';				-- door closed, lock not open

			NumTries <= NumTries;
		elsif (State = 7) then
			if (Key = Master4) then
				State <= State;-- no change same key still pressed
			elsif (Key = 15) then --key released
				State<=State + 1;
			else
				State <= 31;	-- wrong code Error State
			end if;
			Dopen <= '0';				-- door closed, lock not open

			NumTries <= NumTries;
			-- progresses to next State by key release

		elsif (State = 8) then	--Fifth Key in code
			if (Key = Master5) then --correct key
				State<=State + 1;
			elsif (Key = 15) then --key released
				State<=State;
			else
				State <= 31;	-- wrong code Error State
			end if;
			Dopen <= '0';				-- door closed, lock not open

			NumTries <= NumTries;
		elsif (State = 9) then
			if (Key = Master5) then
				State <= State;-- no change same key still pressed
			elsif (Key = 15) then --key released
				State<=State + 1;
			else
				State <= 31;	-- wrong code Error State
			end if;
```

© 1990-2008, C. J. Kikkert, James Cook University

```
            Dopen <= '0';                           -- door closed, lock not open

            NumTries <= NumTries;
            -- progresses to next State by key release

      elsif (State = 10) then   --Sixth Key in code
            if (Key = Master6) then --correct key
                  State<=State + 1;
            elsif (Key = 15) then --key released
                  State<=State;
            else
                  State <= 31;      -- wrong code Error State
            end if;
            Dopen <= '0';                           -- door closed, lock not open

            NumTries <= NumTries;
      elsif (State = 11) then
            if (Key = Master6) then
                  State <= State;-- no change same key still pressed
            elsif (Key = 15) then --key released
                  State<=State + 1;
            else
                  State <= 31;      -- wrong code Error State
            end if;
            Dopen <= '0';                           -- door closed, lock not open

            NumTries <= NumTries;
            -- progresses to next State by key release

      elsif (State = 12) then   --Seventh Key in code
            if (Key = Master7) then --correct key
                  State<=State + 1;
            elsif (Key = 15) then --key released
                  State<=State;
            else
                  State <= 31;      -- wrong code Error State
            end if;
            Dopen <= '0';                           -- door closed, lock not open

            NumTries <= NumTries;
      elsif (State = 13) then
            if (Key = Master7) then
                  State <= State;-- no change same key still pressed
            elsif (Key = 15) then --key released
                  State<=State + 1;
            else
                  State <= 31;      -- wrong code Error State
            end if;
            NumTries <= NumTries;
            Dopen <= '0';                           -- door closed, lock not open

            -- progresses to next State by key release

      elsif (State = 14) then   --Seventh Key in code
            if (Key = Master8) then --correct key
                  State <= State + 1;
            elsif (Key = 15) then --key released
                  State<=State;
            else
                  State <= 31;      -- wrong code Error State
            end if;
            Dopen <= '0';                           -- door closed, lock not open

            NumTries <= NumTries;
      elsif (State = 15) then
            if (Key = Master8) then
                  State <= State;-- no change same key still pressed
            elsif (Key = 15) then --key released
                  State<=State + 1;
            else
                  State <= 31;      -- wrong code Error State
            end if;
```

© 1990-2008, C. J. Kikkert, James Cook University

```
				NumTries <= NumTries;
				Dopen <= '0';				-- door closed, lock not open

				-- progresses to next State by key release

			elsif (State = 16) then	--Seventh Key in code
				if (Key = Master9) then --correct key
					State <= State + 1;
				elsif (Key = 15) then --key released
					State<=State;
				else
					State <= 31;	-- wrong code Error State
				end if;
				Dopen <= '0';				-- door closed, lock not open

				NumTries <= NumTries;
			elsif (State = 17) then
				if (Key = Master9) then
					State <= State;-- no change same key still pressed
				elsif (Key = 15) then --key released
					State<=State + 1;
				else
					State <= 31;	-- wrong code Error State
				end if;
				NumTries <= NumTries;
				Dopen <= '0';				-- door closed, lock not open

				-- progresses to next State by key release

			elsif (State = 18) then	--Eight Key in code
				if (Key = Master10) then		--correct key
					State <= State + 1;
				elsif (Key = 15) then --key released
					State<=State;
				else
					State <= 31;	-- wrong code Error State
				end if;
				Dopen <= '0';				-- door closed, lock not open

				NumTries <= NumTries;
			elsif (State = 19) then
				if ((Key = Master10) or (Key = 15)) then
					State <= State;-- reached the correct code
					Dopen <= '1';				-- Open the door

					NumTries <= "000";
				else
					State <= 30; -- Any key press resets the lock when key released
					Dopen <= '0';
					NumTries <= "000";
				end if;
			elsif (State = 30) then --Reset to state 0 when
				if (Key = 15) then	--key released
					State <= 0;
					Dopen <= '0';
					NumTries <= NumTries;
				else
					State <= State;--wait for key release to reset system
					Dopen <= '0';
					NumTries <= NumTries;
				end if;
			else	--Have error state 31 or startup error. wait for # key
				if (Key = 11) then --# Key to get out of error state
					State <= 30;			-- go to Start (state=0) after key
release
					Dopen <= '0';				-- door closed, lock not open
					if (NumTries(2) = '0') then	-- increment if less than 3, ie 3 resets
max
						NumTries <= NumTries + "001";
					end if;
				else --do nothing
					State <= 31;	-- State = 31, Error
```

```
                    Dopen <= '0';
                    NumTries <= NumTries;
                end if;
            end if;  --end keyState check
        end if;  --end key check
    end process     KeyPressCheck_p;


    Buzzer_p: process(Alarm, Clk, NumTries(2))
    begin
            Alarm <= NumTries(2);
            PBuzzer <= Clk and Alarm;           --LSB of counter ie half the clock rate
            NBuzzer <= (not Clk) and Alarm;
    end process Buzzer_p;

end KeyLock_Arch;
```

© 1990-2008, C. J. Kikkert, James Cook University