EE4306 VHDL assignment

James McGeachin

Table of Contents

Introduction........ 1
Design........ 2
System components........ 2
Design variables........ 3
Control logic state machine design........ 3
Implementation........ 5
Sine wave lookup table........ 5
Resource usage........ 6
Testing........ 7
Simulation........ 7
Demonstration code implemented on CPLD........ 8
Use of demonstration code........ 9
Conclusion........ 9
References........ 9
Appendix A – Main Module........ 10
Appendix B – Prescaler........ 13
Appendix C – Main Counter........ 14
Appendix D – Sine Wave Lookup Table........ 16
Appendix E – Test Code........ 20

Introduction

This report documents the design, implementation and testing of a VHDL module that implements an FFSK modulator for data transmission into a Lattice ispMACH M4A5 CPLD. The general operation of this modulator is to be similar to the transmit chain of the CMX469A MSK modem.

This modulator is to operate at 4800 baud and use a full cycle of a 4.8kHz sine wave cycle to represent a mark and half of a 2.4kHz sine wave cycle to represent a space. To implement FFSK/MSK transmission the output sine wave must be phase continuous and all data transitions will occur at sine wave zero crossing. A 3.6864MHz crystal will be used to clock the CPLD.

The analog signal output is achieved by interfacing the CPLD to an Analog Devices AD557 digital to analog converter. This interface is a eight bit parallel bus that is latched at the DAC end using an active low latch control signal. The DAC is enabled using an active low enable signal.

Design

The design of this project built off the structure of a sine wave generator (a free running counter connected to a look up table which converted the linear output of the counter into a sine wave).

System components

The main counter and sine wave look up table modules were designed to allow the width of the main counter to be variable. The output of the look up table was fixed at eight bits to satisfy the requirement for an eight bit output. The use of a smaller main counter will increase the quantisation of the digital to analog converter output. This became a design trade off that was considered after the system was implemented and tested.

Additional logic modules were added to satisfy the requirements of the assignment. A dividing counter was used to divide the input clock down to an acceptable rate for running the main counter and a control logic module was designed to control the main counter and interface the PLD to the input data and the digital to analog converter. A block diagram showing the planned modules and the interfacing between them is presented in figure 1.

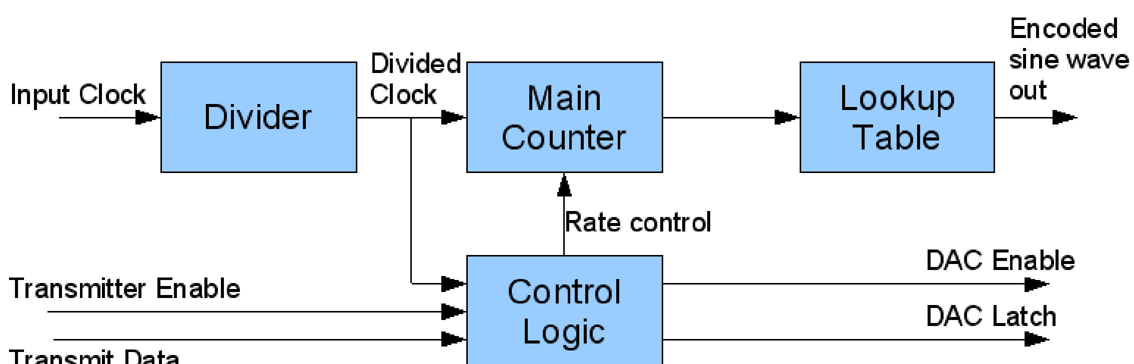

Figure 1: Block diagram showing logic modules and interconnections

Design variables

Once this architecture was decided on, the main design variables were the width of the main counter and the frequency of the external clock input. The table below shows the effect of these two design variables on the size of the clock divider counter and the Nyquist frequency of the DAC output. The Nyquist frequency of the sampled output was used as a measure of the systems output performance. At higher Nyquist frequencies, the output is easier to filter to remove the effects of quantisation.

Table 1 – Design variable combinations and resulting system properties

External Clock Frequency	Main counter width	Internal clock divisor	Clock divider width	Output Nyquist frequency
3.6864MHz	5 bits	48	6 bits	76.8kHz
3.6864MHz	6 bits	24	5 bits	153.6kHz
3.6864MHz	8 bits	6	4 bits	614.4kHz
2MHz	5 bits	13	4 bits	76.8kHz

After implementing the design in VHDL, the system was tested to find the best main counter width to use with the 3.6864MHz clock input while being able to fit on the ispMach CPLD. A main counter width of 6 bits was chosen as a result of this testing.

Control logic state machine design

When designing the control logic, a finite state machine was used due to the procedural nature of the chosen data interface. All internal and external control outputs are operated from the state machine. This approach was chosen to minimise hardware usage and to decrease the complexity of the code.

To satisfy the requirements for the data interface for this design, an asynchronous approach was taken to sampling the data. The main logic, upon being enabled using the transmit enable input, would wait until an edge is found in the data and then wait for half a bit width before latching the data and beginning transmission. Due to this half bit period offset, every bit of the data input will be latched exactly in the middle of the bit transmission.

While transmitting data, the clock for latching the input data is gated by the zero crossing of the output sine wave. This means that the next data byte will not be latched in until the main counter has transmitted the

previous bit. The logic that implements the zero crossing signal is contained in the main counter module. Having this logic external to the state diagram avoids duplicating counters and ensures that the main logic follows the FFSK criteria by only clocking the input data buffer at sine wave zero crossings.

From these design ideas and the CMX469A specifications, the state machine was designed. The final state diagram for the state machine used in the control logic is presented in figure 2. In this state machine, two logic signals from inside the PLD are used. The main counter zero crossing is a signal from the main counter to indicate that it has finished transmitting a bit of data and is ready to receive the next bit of data. In the 'WAIT HALF A BIT' state, the main counter, preloaded with three quarters of its full count, counts upwards until it rolls over and outputs a zero crossing to time the half bit period. The edge detect logic uses the data buffer and latches in the TX_DATA signal every state machine iteration, if the latched data and TX_DATA are different, then an edge has been found.

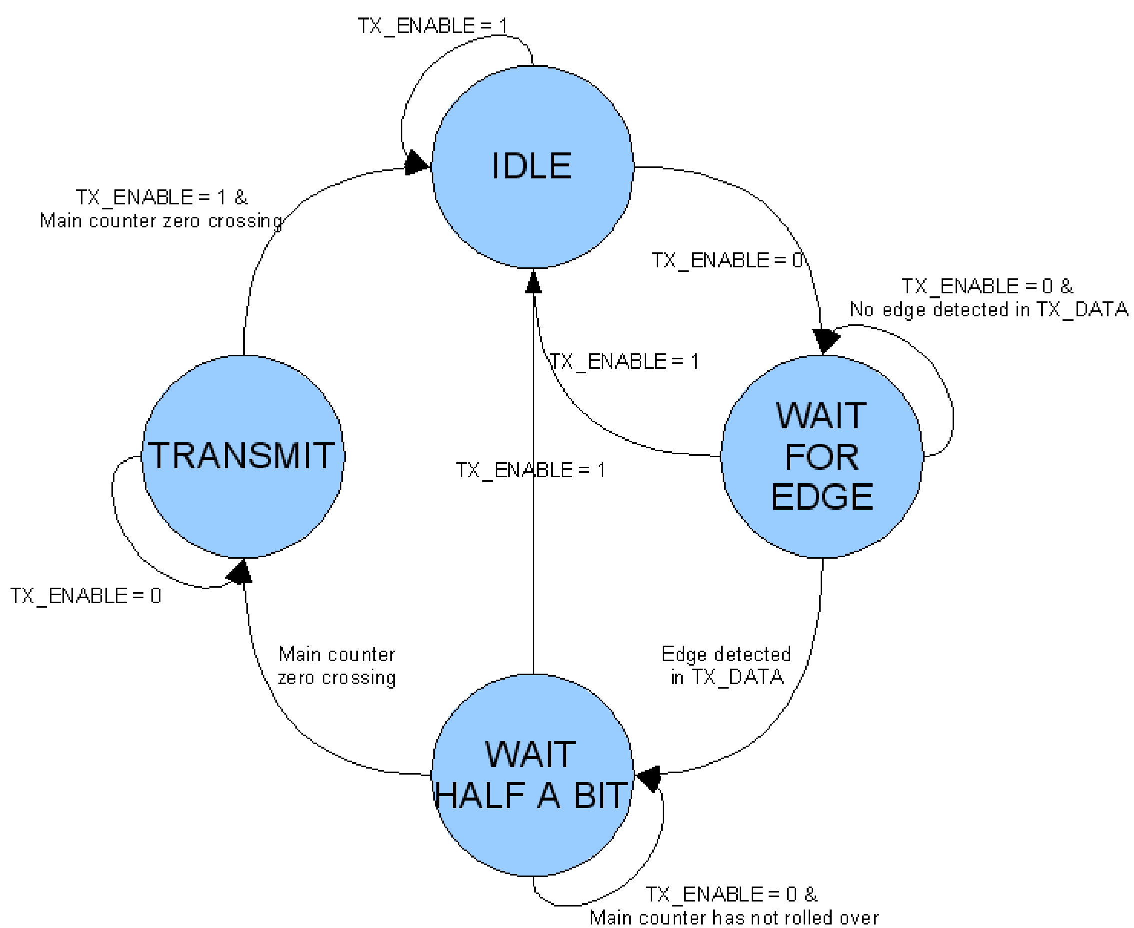

Figure 2: State diagram showing operation of main control logic

Implementation

The implementation of this design was accomplished by writing VHDL modules to represent each of the separate logic modules in figure 1. This approach allowed the more complex modules to be tested independently. The implemented modules formed a tree hierarchy with a main module at the root of the tree. The main module implemented the control logic, pin assignments and interconnected the other logic modules in the system. Through the use of generics, the width of the main counter and the divisor used in the prescaler can be set in the main module and will propagate through the code. This allowed several design combinations to be tested before the final design was settled on. A diagram showing the final tree is presented in figure 3.

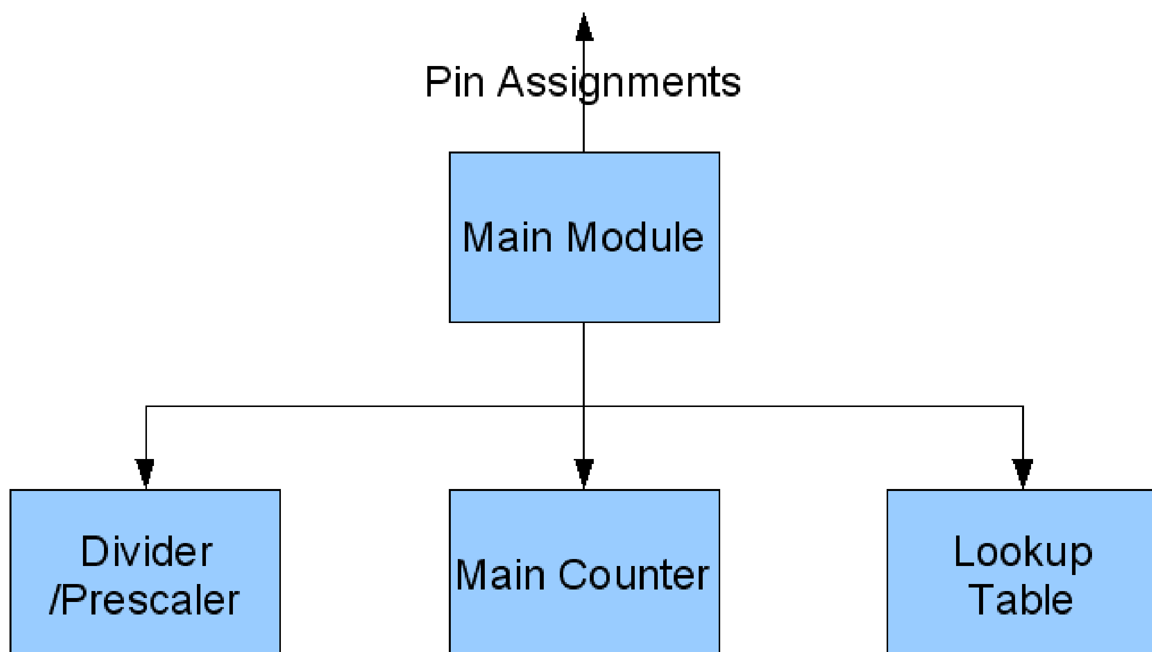

Figure 3: Diagram showing hierarchy of implemented VHDL modules

Sine wave lookup table

The sine wave lookup table was generated for this assignment using a Python script. Initially the lookup table consisted of a direct lookup table between the input and the sine wave output. This approach is known as a full sine wave lookup table. After the code was tested and modified, the lookup table module was altered to only output the positive half of the output sine wave and this output was inverted based upon the most significant bit of the input. This is known as a half sine wave lookup table. The half sine wave lookup table was used because the macrocell layout in the ispMACH M4A5 incorporate an XOR gate after the

combinational logic array[1]. This XOR gate would allow the output of the combinational logic to be inverted based upon a signal. The use of the half wave lookup table saved one macrocell of resources compared to a full sine wave module.

Resource usage

When estimating the number of resources used by this design, the number of flip flops and number of output pins that would be required to implement the solution gives a reasonably good lower bound approximation as this design does not require a large amount of combinational logic. The following flip flops/registers and output pins are used in this design:

- State register – 2 F/F
- Buffered data – 1 F/F
- DAC control outputs – 2 Output Pins
- Clock divider – 5 F/F
- Main counter - 6 F/F
- Main counter rollover flag – 1 F/F
- Sine wave LUT – 8 Output Pins

Summing these resources gives an approximate resource usage of 25 macrocells. After the code was implemented and tested, it was compiled using Synplify. The number of macrocells used by the compiled code was 28. This result is similar to the approximation made above. An excerpt from the fitter report is shown below.

```
Device_Resource_Summary
~~~~~~~~~~~~~~~~~~~~~~~
                                 Total
                               Available  Used  Available
Utilization
Dedicated Pins
  Input-Only Pins                  ..       ..      ..     -->    ..
  Clock/Input Pins                  2        1       1     -->   50%
I/O Pins                           32       12      20     -->   37%
Logic Macrocells                   64       28      36     -->   43%
  Input Registers                  32        0      32     -->    0%
  Unusable Macrocells              ..        0      ..

CSM Outputs/Total Block Inputs    132       38      94     -->   28%
Logical Product Terms             320      103     217     -->   32%
Product Term Clusters              64       34      30     -->   53%
```

Listing 1: Fitter report for compiled design

Testing

After the implementation of the design was finalised, it was tested using both simulation and by implementing it on the PLD using test code to generate an output waveform that was passed the a digital to analog converter and read using an oscilloscope.

Simulation

As each logic module in the assignment was implemented in its own VHDL module, it was possible to test each VHDL module as it was written. This decreased the complexity of the task of debugging the code. The timing diagrams for the clock divider and main counter are shown in figures 5 and 6 respectively.

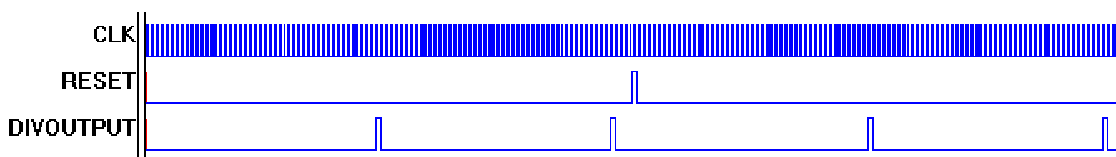

Figure 4: Timing diagram generated during clock divider simulation

The simulation of the prescaler/clock divider, shown in figure 5, demonstrates the asynchronous reset functionality and the divided clock output of the prescaler. The asynchronous reset function of this module is not used in the finished system.

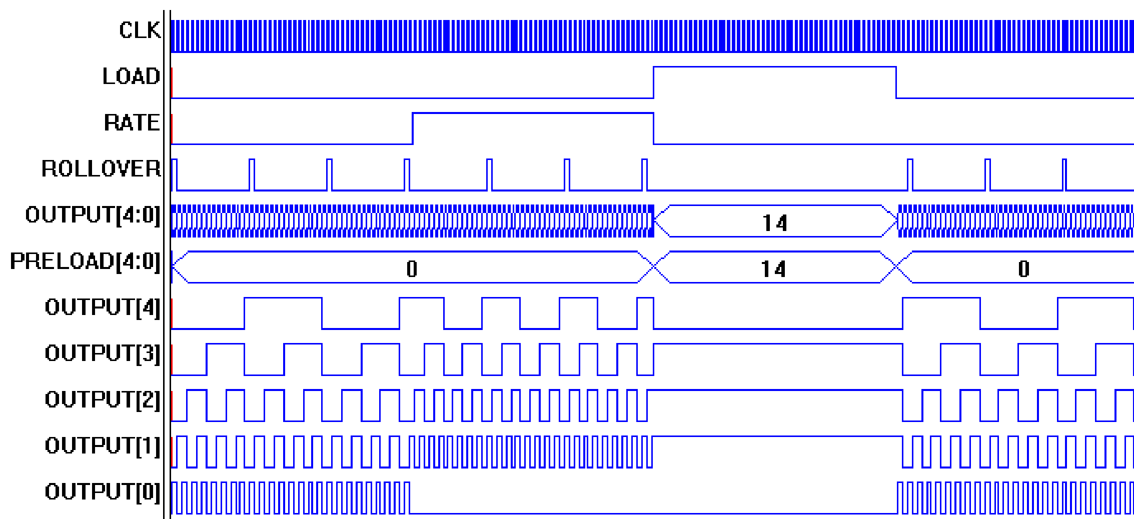

Figure 5: Timing diagram generated during main counter simulation.

The simulation of the main counter, as shown in figure 6, covered the change of count rate and asynchronous loading of the counter. The rollover signal shown is to generate TX_SYNC in the main code and is used to latch the input data buffer. Once the separate modules had been written, they were combined and the final code was simulated and debugged. A timing diagram showing the results of this simulation are shown in figure 6.

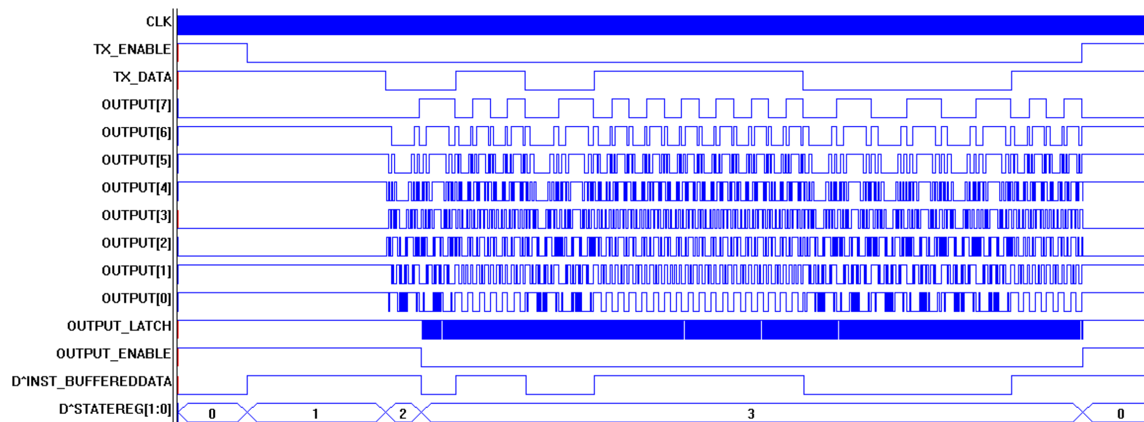

Figure 6: Timing diagram showing the transmission of a data frame

Demonstration code implemented on CPLD

In order to test the code using a CPLD evaluation board/DAC/Oscilloscope testing set up. A VHDL module was written to go 'in between' the final implementation and the pins on the PLD. This module implemented

a separate clock divider and a counter to stimulate the TX_ENABLE and TX_DATA pins in order to test the code. This code is presented in Appendix E. This test implementation was tested successfully using the hardware and an oscilloscope. However, due to a number of factors, no data was recorded during these tests.

Use of demonstration code

The demonstration code used to test the assignment starts operating once the PLD is powered on. Pins 24 and 42 are used to output TX_DATA and TX_ENABLE respectively. All other pin assignments are as they were specified in the assignment specification. The demonstration code was compiled using Synplify.

The procedure to set up and use the CPLD board and DAC set up is;

1. Remove the clock selection jumper from the CPLD board
2. Connect the DAC board using the ribbon cable to the header to the left of the CPLD on the CPLD board, the ribbon cable should run over the clock select header.
3. Power on the CPLD board, program the CPLD with the JEDEC file if it has not already been done
4. Once powered on and programmed, the demonstration code will be running. Using TX_ENABLE (pin 42) as a trigger TX_DATA and the output sine wave can be traced using the oscilloscope.

Conclusion

The design and implementation fulfill the requirements of the assignment and test code was written to allow the functionality of the code to be demonstrated. A possible enhancement to the current implementation include investigating if using a quarter sine wave lookup table would reduce logic use. As the current implementation uses less than half of the available CPLD logic resources, the implementation could possibly be extended to include an FFSK receive chain to create an FFSK modem in programmable logic.

References

1. ispMACH family data sheet, Lattice Semiconductor, p11, fig. 6

Appendix A – Main Module

```
--EE4306 VHDL Assignment main module
--James McGeachin 2007

--This file contains the control logic, the instantiation of the logic modules and the pin assignments for
--the EE4306 VHDL Assignment 2007

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

entity Main_E is
generic (
        maincounterwidth : integer := 6; --Controls the width of the main counter used to generate the sine wave
        freqdivisor : integer := 24 ); --Controls the divisor of the prescaler

port(
        clk: in std_logic ; --The main clock that this PLD runs off (specified to be 3.6864MHz)
        output: out std_logic_vector ( 7 downto 0 ); --DAC data output
        output_latch : out std_logic ; --DAC latch control (active low)
        output_enable : out std_logic ; --DAC enable control (active low)
        tx_enable: in std_logic ; --Transmitter enable control signal (active low)
        tx_data: in std_logic ); --Serial data to be transmitted

        --Pin assignments as specified in the assignment sheet
        attribute loc : string;
        attribute loc of clk : signal is "11" ;
        attribute loc of output : signal is "14, 15, 16, 17, 18, 19, 20, 21";
        attribute loc of output_latch : signal is "8";
        attribute loc of output_enable : signal is "6";
        attribute loc of tx_enable : signal is "24";
        attribute loc of tx_data : signal is "26";

        --State machine state definition
        type state is (IDLE, WAITEDGE, WAITHALFBIT, TRANSMIT);
end;

architecture Main_A of Main_E is

component Prescaler_E is
generic( divisor : integer);
port(
        clk: in std_logic ;
        reset: in std_logic ;
```

```
        divoutput : out std_logic );
end component;

component MainCounter_E is
generic(
        width: integer);
port(
        clk: in std_logic ;
        load : in std_logic ;
        preload : in std_logic_vector ( width - 1 downto 0);
        rate : in std_logic ;
        output: inout std_logic_vector ( width - 1 downto 0 ) ;
        rollover : out std_logic );
end component;

component SineLookup_E is
generic (width: integer);
port(
        CountInput: in std_logic_vector ( width - 1 downto 0 );
        Output: out std_logic_vector ( width - 1 downto 0 ) );
end component;

signal internalclk : std_logic; --Carries the divided clock that is used to clock the state machine and main counter

--Main counter control signals
signal counterload : std_logic; --Asynchronous load signal
signal counterrate : std_logic; --Rate selection signal

--Counter outputs
signal counteroutput : std_logic_vector (maincounterwidth - 1 downto 0); --main counter output ('sawtooth wave')
signal tx_sync : std_logic; --counter rollover/sync clock - used to synchronise the data buffer with the counter

--State machine register
signal statereg : state register:= IDLE;

begin

--Define the components used in this design, using a six bit main counter to drive the sine wave output, therefore the
--divider ratio and main counter preload are defined accordingly
Prescaler: Prescaler_E generic map (freqdivisor) port map (clk, '0', internalclk);
MainCounter: MainCounter_E  generic map (maincounterwidth) port map (internalclk, counterload, "110001", counterrate,
counteroutput, tx_sync);
SineLookup: SineLookup_E generic map (maincounterwidth) port map (counteroutput, output);

process(internalclk)
variable buffereddata : std_logic; --This variable is used to buffer the input on tx_data for use in the state machine
begin
```

```
if falling_edge(internalclk) then
        case statereg is
                --The PLD is powered on but tx_enable is not asserted)
                when IDLE =>
                        if (tx_enable = '0') then
                                buffereddata := tx_data;
                                statereg <= WAITEDGE;
                        else
                                statereg <= IDLE;
                        end if;
                --tx_enable has been asserted, waiting for an edge in tx_data to synchronise with
                when WAITEDGE =>
                        if (tx_enable = '1') then
                                statereg <= IDLE;
                        elsif (tx_data /= buffereddata) then
                                statereg <= WAITHALFBIT;
                        else
                                statereg <= WAITEDGE;
                        end if;
                --Once an edge in the data has been found, the main counter counts half a bit width
                --(output enable is off) before starting to sample the data.
                when WAITHALFBIT =>
                        if (tx_enable = '1') then
                                statereg <= IDLE;
                        elsif (tx_sync = '1') then
                                buffereddata := tx_data;
                                statereg <= TRANSMIT;
                        else
                                statereg <= WAITHALFBIT;
                        end if;
                --Transmit the data as it is sampled, if we get to the end of a bit transmission and
                --tx_enable has been deasserted, go back to the IDLE state
                when TRANSMIT =>
                        if (tx_sync = '1') then
                                if (tx_enable = '1') then
                                        statereg <= IDLE;
                                else
                                        statereg <= TRANSMIT;
                                        buffereddata := tx_data;
                                end if;
                        else
                                statereg <= TRANSMIT;
                        end if;
                --This will generate an error because the case statement is fully defined (2 bits/ 4 states)
                --I left it in for good coding practices in case the state machine is expanded
                when others =>
```

```
                                statereg <= IDLE;
                end case;
        end if;

        --If the PLD is not transmitting data, we want to control the rate of the main counter
        if (statereg /= TRANSMIT) then
                counterrate <= '0';
        else
                counterrate <= buffereddata;
        end if;
end process;

process (statereg, internalclk)
begin
        --If we don't want the main counter counting, hold it in a preloaded state ready for use
        if (statereg = WAITEDGE or statereg = IDLE) then
                counterload <= '1';
        else
                counterload <= '0';
        end if;

        --Operate the DAC control signals, we only want the DAC operating when we want to transmit something
        if (statereg = TRANSMIT) then
                output_enable <= '0';
                output_latch <= not internalclk;
        else
                output_enable <= '1';
                output_latch <= '1';
        end if;

end process;


end Main_A;
```

Appendix B – Prescaler

```
--EE4306 VHDL clock prescaler
--James McGeachin 2007

--The prescaler divides the input clk by the specified divisor and outputs the
--divided output on output

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

```
entity Prescaler_E is

generic( divisor : integer := 48); --Divisor definition, clk is divided by this amount

port(
        clk: in std_logic ; --Clock to be divided
        reset: in std_logic ; --Asynchronous reset
        divoutput : out std_logic ); --Divided clock output
end;

architecture Prescaler_A of Prescaler_E is
begin
        process(clk, reset)
        variable counter : integer range divisor downto 0;
        begin
                if (reset = '1') then
                        counter := 0;
                        divoutput <= '0';
                elsif (rising_edge(clk)) then
                        if (counter >= (divisor - 1)) then
                                counter := 0;
                                divoutput <= '1';
                        else
                                counter := counter + 1;
                                divoutput <= '0';
                        end if;
                end if;
        end process;
end Prescaler_A;
```

Appendix C – Main Counter

```
--EE4306 VHDL Assignment main counter
--James McGeachin 2007

--The main counter is used to drive the lookup table sinewave output at different frequencies based upon
--the rate input.  If rate is high, then the counter is driven twice as fast as the clock input whereas if
--rate is low, then the counter driven at normal rate.  Special logic is used so that the counter signals
--a 'rollover' at a constant rate independent of the actual rate of the counter.

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

entity MainCounter_E is
```

```
generic(
        width : integer := 5 ); --Specifies the width of the counter output

port(
        clk: in std_logic ; --Clock input
        load : in std_logic ; --Asynchronous load control signal
        preload : in std_logic_vector (width - 1 downto 0); --Asynchronous load data
        rate : in std_logic ; --Rate control input
        output: inout std_logic_vector ( width - 1 downto 0 ) ; --Counter output data
        rollover : out std_logic );  --Rollover output
end;

architecture MainCounter_A of MainCounter_E is
signal rollovercounter : std_logic;
begin
        process(clk, load, preload)
        begin
                if (load = '1') then
                        --Asynchronous load
                        output <= preload;
                        rollovercounter <= '0';
                        rollover <= '0';
                elsif (rising_edge(clk)) then
                        --Increment the counter based on the rate input
                        if (rate = '1') then
                                --Increment counter by two while fixing the LSB to '0'
                                --This is done to make the logic for the rollover signal easier
                                output(width - 1 downto 1) <= output(width - 1 downto 1) + '1';
                                output(0) <= '0';
                        else
                                output <= output + '1';
                        end if;

                        --The counter can 'roll over' when the output is either zero or half of the counter width
                        --A counter bit is implemented so that the rollover output is asserted at the same rate
                        --regardless of counter rate.  This is done by counting the number of possible rollovers
                        --and outputting a rollover when it is allowable.  In this context, when we are running the
counter
                        --at the fast rate (rate = '1') then we want to halve the number of rollover outputs
                        if ((output = CONV_STD_LOGIC_VECTOR(0, width)) or (output = '1'&CONV_STD_LOGIC_VECTOR(0, width-
1))) then
                                if (rate = '0') then
                                        rollover <= '1';
                                        rollovercounter <= '0';
                                elsif (rate = '1' and rollovercounter = '1') then
                                        rollover <= '1';
                                        rollovercounter <= '0';
```

```
                        else
                                rollover <= '0';
                                rollovercounter <= not rollovercounter;
                        end if;
                else
                        rollover <= '0';
                        rollovercounter <= rollovercounter;
                end if;
        end if;
    end process;
end MainCounter_A;
```

Appendix D – Sine Wave Lookup Table

```
--EE4306 VHDL Assignment sinewave lookup table
--This component implements a variable width input, eight bit output sine wave lookup table.

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

entity SineLookup_E is
generic (width : integer := 5); --Width of the input data

port(
        CountInput: in std_logic_vector ( width - 1 downto 0 ); --Input 'sawtooth' data
        Output: out std_logic_vector ( 7 downto 0 ) ); --Output 'sinewave' data

end;

architecture SineLookup_A of SineLookup_E is
signal halfwave : std_logic_vector(7 downto 0);
begin
        process (CountInput)
        begin
                --This case statement implements a half sine wave look up table.  The MSB
                --of the input data is replaced with a '0' and a padding of '0''s is added
                --as the least significant bits
                case "0"&CountInput(width - 2 downto 0)&CONV_STD_LOGIC_VECTOR(0, 8-width) is
                        when  X"00"  => halfwave <=  X"80" ;
                        when  X"01"  => halfwave <=  X"83" ;
                        when  X"02"  => halfwave <=  X"86" ;
                        when  X"03"  => halfwave <=  X"89" ;
                        when  X"04"  => halfwave <=  X"8c" ;
                        when  X"05"  => halfwave <=  X"8f" ;
                        when  X"06"  => halfwave <=  X"92" ;
```

```
when  X"07"  => halfwave <=  X"95" ;
when  X"08"  => halfwave <=  X"98" ;
when  X"09"  => halfwave <=  X"9b" ;
when  X"0a"  => halfwave <=  X"9e" ;
when  X"0b"  => halfwave <=  X"a2" ;
when  X"0c"  => halfwave <=  X"a5" ;
when  X"0d"  => halfwave <=  X"a7" ;
when  X"0e"  => halfwave <=  X"aa" ;
when  X"0f"  => halfwave <=  X"ad" ;
when  X"10"  => halfwave <=  X"b0" ;
when  X"11"  => halfwave <=  X"b3" ;
when  X"12"  => halfwave <=  X"b6" ;
when  X"13"  => halfwave <=  X"b9" ;
when  X"14"  => halfwave <=  X"bc" ;
when  X"15"  => halfwave <=  X"be" ;
when  X"16"  => halfwave <=  X"c1" ;
when  X"17"  => halfwave <=  X"c4" ;
when  X"18"  => halfwave <=  X"c6" ;
when  X"19"  => halfwave <=  X"c9" ;
when  X"1a"  => halfwave <=  X"cb" ;
when  X"1b"  => halfwave <=  X"ce" ;
when  X"1c"  => halfwave <=  X"d0" ;
when  X"1d"  => halfwave <=  X"d3" ;
when  X"1e"  => halfwave <=  X"d5" ;
when  X"1f"  => halfwave <=  X"d7" ;
when  X"20"  => halfwave <=  X"da" ;
when  X"21"  => halfwave <=  X"dc" ;
when  X"22"  => halfwave <=  X"de" ;
when  X"23"  => halfwave <=  X"e0" ;
when  X"24"  => halfwave <=  X"e2" ;
when  X"25"  => halfwave <=  X"e4" ;
when  X"26"  => halfwave <=  X"e6" ;
when  X"27"  => halfwave <=  X"e8" ;
when  X"28"  => halfwave <=  X"ea" ;
when  X"29"  => halfwave <=  X"eb" ;
when  X"2a"  => halfwave <=  X"ed" ;
when  X"2b"  => halfwave <=  X"ee" ;
when  X"2c"  => halfwave <=  X"f0" ;
when  X"2d"  => halfwave <=  X"f1" ;
when  X"2e"  => halfwave <=  X"f3" ;
when  X"2f"  => halfwave <=  X"f4" ;
when  X"30"  => halfwave <=  X"f5" ;
when  X"31"  => halfwave <=  X"f6" ;
when  X"32"  => halfwave <=  X"f8" ;
when  X"33"  => halfwave <=  X"f9" ;
when  X"34"  => halfwave <=  X"fa" ;
```

```
when  X"35"  => halfwave <=  X"fa" ;
when  X"36"  => halfwave <=  X"fb" ;
when  X"37"  => halfwave <=  X"fc" ;
when  X"38"  => halfwave <=  X"fd" ;
when  X"39"  => halfwave <=  X"fd" ;
when  X"3a"  => halfwave <=  X"fe" ;
when  X"3b"  => halfwave <=  X"fe" ;
when  X"3c"  => halfwave <=  X"fe" ;
when  X"3d"  => halfwave <=  X"ff" ;
when  X"3e"  => halfwave <=  X"ff" ;
when  X"3f"  => halfwave <=  X"ff" ;
when  X"40"  => halfwave <=  X"ff" ;
when  X"41"  => halfwave <=  X"ff" ;
when  X"42"  => halfwave <=  X"ff" ;
when  X"43"  => halfwave <=  X"ff" ;
when  X"44"  => halfwave <=  X"fe" ;
when  X"45"  => halfwave <=  X"fe" ;
when  X"46"  => halfwave <=  X"fe" ;
when  X"47"  => halfwave <=  X"fd" ;
when  X"48"  => halfwave <=  X"fd" ;
when  X"49"  => halfwave <=  X"fc" ;
when  X"4a"  => halfwave <=  X"fb" ;
when  X"4b"  => halfwave <=  X"fa" ;
when  X"4c"  => halfwave <=  X"fa" ;
when  X"4d"  => halfwave <=  X"f9" ;
when  X"4e"  => halfwave <=  X"f8" ;
when  X"4f"  => halfwave <=  X"f6" ;
when  X"50"  => halfwave <=  X"f5" ;
when  X"51"  => halfwave <=  X"f4" ;
when  X"52"  => halfwave <=  X"f3" ;
when  X"53"  => halfwave <=  X"f1" ;
when  X"54"  => halfwave <=  X"f0" ;
when  X"55"  => halfwave <=  X"ee" ;
when  X"56"  => halfwave <=  X"ed" ;
when  X"57"  => halfwave <=  X"eb" ;
when  X"58"  => halfwave <=  X"ea" ;
when  X"59"  => halfwave <=  X"e8" ;
when  X"5a"  => halfwave <=  X"e6" ;
when  X"5b"  => halfwave <=  X"e4" ;
when  X"5c"  => halfwave <=  X"e2" ;
when  X"5d"  => halfwave <=  X"e0" ;
when  X"5e"  => halfwave <=  X"de" ;
when  X"5f"  => halfwave <=  X"dc" ;
when  X"60"  => halfwave <=  X"da" ;
when  X"61"  => halfwave <=  X"d7" ;
when  X"62"  => halfwave <=  X"d5" ;
```

```
                    when  X"63"  => halfwave <=  X"d3" ;
                    when  X"64"  => halfwave <=  X"d0" ;
                    when  X"65"  => halfwave <=  X"ce" ;
                    when  X"66"  => halfwave <=  X"cb" ;
                    when  X"67"  => halfwave <=  X"c9" ;
                    when  X"68"  => halfwave <=  X"c6" ;
                    when  X"69"  => halfwave <=  X"c4" ;
                    when  X"6a"  => halfwave <=  X"c1" ;
                    when  X"6b"  => halfwave <=  X"be" ;
                    when  X"6c"  => halfwave <=  X"bc" ;
                    when  X"6d"  => halfwave <=  X"b9" ;
                    when  X"6e"  => halfwave <=  X"b6" ;
                    when  X"6f"  => halfwave <=  X"b3" ;
                    when  X"70"  => halfwave <=  X"b0" ;
                    when  X"71"  => halfwave <=  X"ad" ;
                    when  X"72"  => halfwave <=  X"aa" ;
                    when  X"73"  => halfwave <=  X"a7" ;
                    when  X"74"  => halfwave <=  X"a5" ;
                    when  X"75"  => halfwave <=  X"a2" ;
                    when  X"76"  => halfwave <=  X"9e" ;
                    when  X"77"  => halfwave <=  X"9b" ;
                    when  X"78"  => halfwave <=  X"98" ;
                    when  X"79"  => halfwave <=  X"95" ;
                    when  X"7a"  => halfwave <=  X"92" ;
                    when  X"7b"  => halfwave <=  X"8f" ;
                    when  X"7c"  => halfwave <=  X"8c" ;
                    when  X"7d"  => halfwave <=  X"89" ;
                    when  X"7e"  => halfwave <=  X"86" ;
                    when  others  => halfwave <=  X"83" ;
            end case;
    end process;

    process(halfwave, CountInput(width - 1))
    begin
            --If the MSB of the input data is high, then the halfwave signal
            --must be inverted to form the other half of the sine wave
            if (CountInput(width - 1) = '1') then
                    output <= not halfwave;
            else
                    output <= halfwave;
            end if;
    end process;
end SineLookup_A;
```

Appendix E – Test Code

```
--EE4306 VHDL Assignment Test Code
```

```
--James McGeachin 2007

--This module implements a test harness for the EE4306 assignment
--All pins are brought straight through the module and assigned to
--pins except for tx_enable and tx_data

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

entity DemoCode_E is

port(
        clk: in std_logic ;
        output: out std_logic_vector ( 7 downto 0 );
        output_latch : out std_logic ;
        output_enable : out std_logic ;
        trigger : out std_logic ;
        chan2 : out std_logic );

        attribute loc : string;
        attribute loc of clk : signal is "11" ;
        attribute loc of output : signal is "14, 15, 16, 17, 18, 19, 20, 21";
        attribute loc of output_latch : signal is "8";
        attribute loc of output_enable : signal is "6";
        attribute loc of trigger : signal is "42";
        attribute loc of chan2 : signal is "24";
end;

architecture DemoCode_A of DemoCode_E is

component Main_E is
port(
        clk: in std_logic ;
        output: out std_logic_vector ( 7 downto 0 );
        output_latch : out std_logic ;
        output_enable : out std_logic ;
        tx_enable: in std_logic ;
        tx_data: in std_logic );
end component;

signal baudclock : std_logic; --Clock for the outputstate counter that runs at the baud rate
signal outputstate : integer range 11 downto 0;

--These signals are interfaced directly to the pins in the assignment
```

```
signal tx_data : std_logic;
signal tx_enable : std_logic;
begin

MainProg: Main_E port map (clk, output, output_latch, output_enable, tx_enable, tx_data);

        --Test outputs
        chan2 <= tx_data;
        trigger <= tx_enable;

        --Divide the counter down to the baud rate
        process(clk)
        variable counter : integer range 768 downto 0;
        begin
                if (rising_edge(clk)) then
                        if (counter < 767) then
                                counter := counter + 1;
                                baudclock <= '0';
                        else
                                counter := 0;
                                baudclock <= '1';
                        end if;
                end if;
        end process;

        --Increment the counter to change the input into the Main_E module to test the code
        process(baudclock)
        begin
                if (rising_edge(baudclock)) then
                        if (outputstate < 11) then
                                outputstate <= outputstate + 1;
                        else
                                outputstate <= 0;
                        end if;
                end if;

        end process;

        --Using the outputstate counter, define the input into the Main_E module
        process(outputstate)
        begin
                case outputstate is
                when 0 =>
                        tx_enable <= '1';
                        tx_data <= '1';
                when 1 =>
```

```
                tx_enable <= '0';
                tx_data <= '1';
            when 2 =>
                tx_enable <= '0';
                tx_data <= '0';
            when 3 =>
                tx_enable <= '0';
                tx_data <= '1';
            when 4 =>
                tx_enable <= '0';
                tx_data <= '0';
            when 5 =>
                tx_enable <= '0';
                tx_data <= '1';
            when 6 =>
                tx_enable <= '0';
                tx_data <= '0';
            when 7 =>
                tx_enable <= '0';
                tx_data <= '0';
            when 8 =>
                tx_enable <= '0';
                tx_data <= '1';
            when 9 =>
                tx_enable <= '0';
                tx_data <= '1';
            when 10 =>
                tx_enable <= '0';
                tx_data <= '1';
            when others =>
                tx_enable <= '1';
                tx_data <= '1';
            end case;
        end process;

end DemoCode_A;
```