A Digital Signal Processor
Using Finite Impulse Response Convolution

Dan Ports
October 28, 2003

6.111 Lab 3
TA: Frank Honore

Abstract

Digital signal processing can be performed using the method of finite impulse response convolution. This implementation performs signal processing by sampling the input signal at 44.287 kHz using an analog-to-digital converter, stores the sample values in a circular buffer in RAM, computes an output value by convolving the most recent 16 sample values with coefficients stored in ROM, and outputs the computed value using a digital-to-analog converter. The system is controlled by a synchronous finite state machine. Convolution is performed by a minor FSM and an 8-bit sequential multiplier. The design is realized in hardware using three chips: a FPGA, an ADC, and a DAC. Testing and debugging issues are discussed. The digital signal processor was successfully implemented and passed all tests.

Contents

List of Figures

1 Overview

This report describes a signal processor that processes an input signal based on one of sixteen selectable, predefined filters. The signal processor is implemented digitally, by converting the input signal to a digital representation, processing it using a FPGA, and outputting it through a digital-to-analog converter. Computations are performed using the method of finite impulse response convolution, with impulse responses of sixteen 8-bit samples.

2 Operation

This device accepts a differential voltage input (V_{in}) with amplitude at most ± 1.28V and processes it. The output is generated with a 1.28V offset, i.e. a range of 0V – 2.56V referenced to ground. Sampling is performed at approximately 44.287 kHz, so input signals can contain frequency components up to approximately 22.1 kHz without causing aliasing.

The RESET button should be pressed after powering on the system to ensure that all internal components are in valid states. Afterwards, the system will automatically begin sampling and processing the input signal. The filter to be applied may be chosen from the sixteen available filters using the SELECT switches.

A BYPASS switch (active high) is provided. It should be left disabled for normal operation. When enabled, the output from the ADC is passed directly to the DAC, bypassing the convolution process. This function is provided for testing purposes.

3 Design

3.1 Overview

The design of this device is represented by the block diagram shown in Figure 1. All synchronous components are connected to the standard NuBus clock. An analog-to-digital converter (ADC) and a digital-to-analog converter (DAC) are used to digitize the analog input signal and generate an output signal. These are controlled by the central Control FSM, which begins a sample and processing cycle every time it receives a signal from the Divider module. The Divider outputs a pulse once every 226 clock cycles, to maintain the appropriate sample rate. The Control FSM uses the DAC to output the most recently computed filtered sample, then simultaneously starts the analog-to-digital conversion cycle and the convolution arithmetic. When these are completed, the stores the newly-read value from the ADC in RAM, then waits for the next sample pulse from the divider.

Mathematical computations are performed by the Convolver FSM, which is controlled by the Controller FSM, and makes use of a Multiplier module. The Convolver FSM waits for a START signal from the Controller FSM, and then records the starting address given to it by the Controller FSM. It performs convolutions by summing sixteen partial products in an accumulator. Each partial product is computed by multiplying a sample stored in RAM (address given by StartAddr - i) by a coefficient stored in ROM (address given by i). The convolver FSM outputs the addresses to the RAM and ROM, then starts the multiplier. The data outputs from the RAM and ROM are connected to the multiplier. The multiplier is a sequential multiplier, requiring eight cycles to perform a multiplication. Once the multiplier finishes computation, the convolver adds the result to its accumulator and prepares for the next multiplication. When all 16 partial products have been summed, the convolver returns the highest-order 8 bits of the result. Numeric conversion modules are used throughout the design to convert sign-magnitude representations of numbers to two's complement, and vice versa.

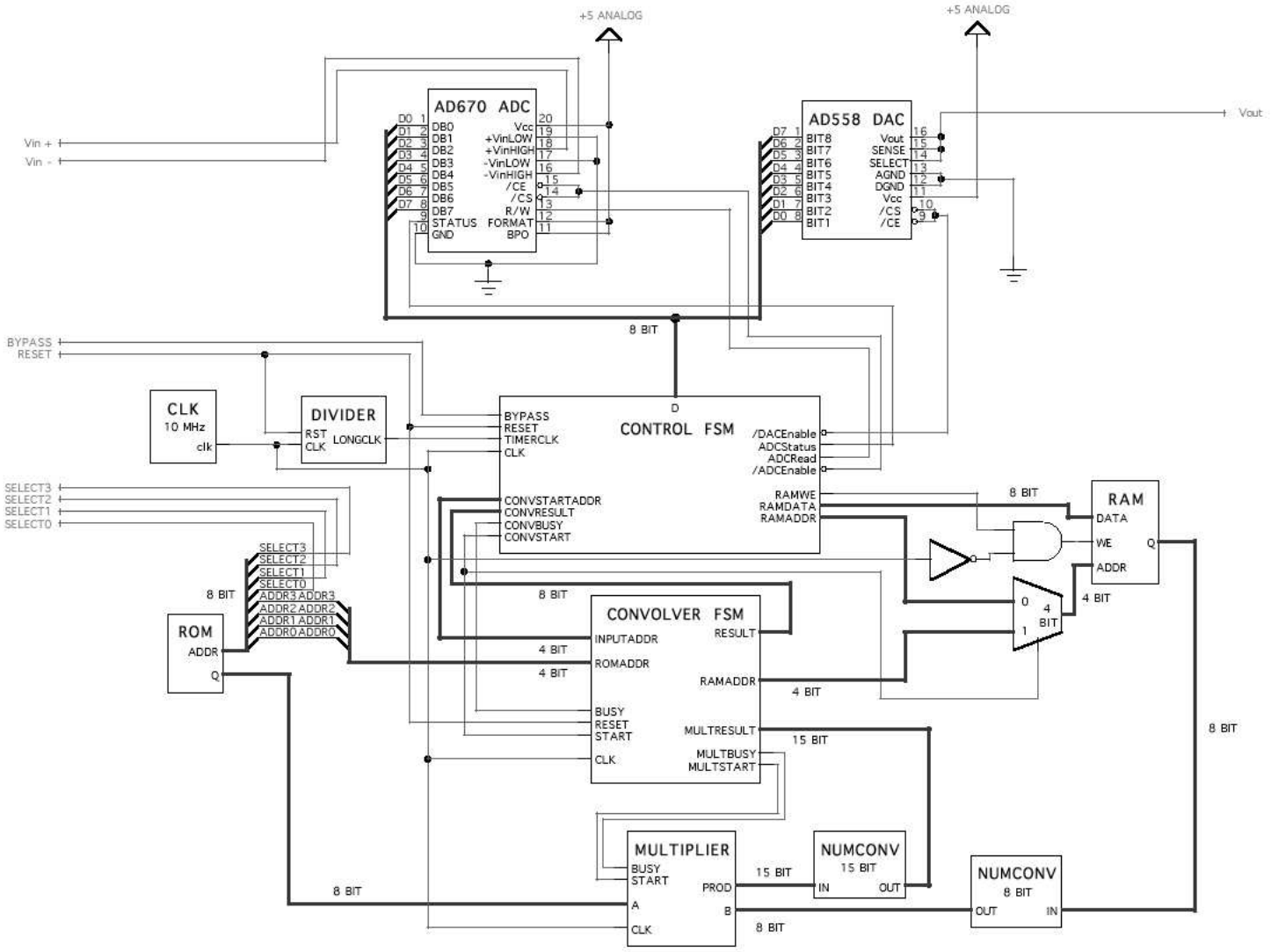

Figure 1: System block diagram

All components except for the clock, digital-to-analog converter, and analog-to-digital converter are implemented using an Altera FLEX 10K10 FPGA, as depicted in the wiring diagram in Figure 2. The VHDL files used to program the FPGA are attached in the appendix, and the design of the individual modules are described in this document. The clock is a standard 10 MHz NuBus clock, and the ADC and DAC are Analog Devices AD670 and AD558 chips respectively. A common 8-bit data bus is used for the ADC and DAC. nputs are provided using debounced switches for the SELECT and BYPASS inputs, and a debounced push-button for the RESET signal.

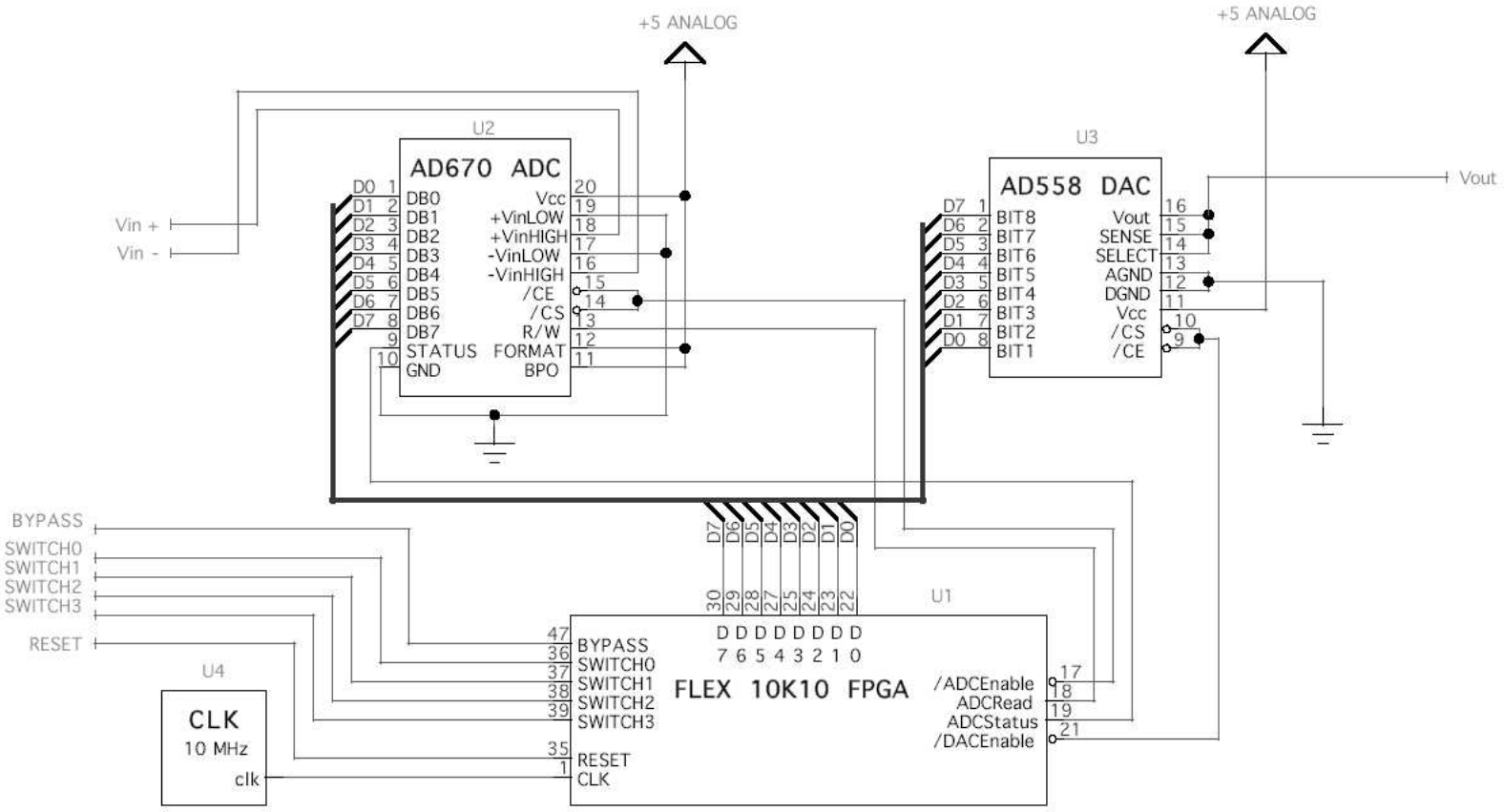

Figure 2: Hardware wiring diagram

3.2 Clock

All synchronous components in this system operate on the rising edge of a single, global clock signal. This clock signal is generated by the standard 10 MHz 25% duty cycle NuBus clock integral to the lab kit.

3.3 Control FSM

The Control FSM is the major FSM in this system. It operates the analog-to-digital and digital-to-analog converters to read and write the output signals, and starts the Convolver FSM to perform convolutions. The six possible states, their outputs, and the transitions between them are shown in the state diagram in Figure 3. All outputs not listed in a state's node in the diagram are set to their default inactive value (0 for active high signals and 1 for active low signals).

The FSM is defined by four processes that operate concurrently. One process determines the next state asynchronously as a function of the current state. The next two processes generate the outputs as a function of the current state and read the input data bus in the ADCRead state. The final process performs state transitions on the rising edge of the clock.

The system starts up in the WaitForTimer state. It remains in this state until a pulse is received from the Divider module that serves as the sample timer.

When this signal is received, the FSM transitions to the DACWrite state. In this state, the DAC's enable signal is set low (active), and the most recently computed output value is output on the data bus. The FSM stays in this state for three clock cycles. An internal variable (i) is decremented on each clock cycle to allow this interval to be measured.

After the counter reaches zero, the FSM moves to the ADCEnable state. The ADCEnable and ADCRead signals are set low in order to begin the ADC's conversion process. The ConvolverStart signal is also asserted high in order to simultaneously begin the computation of the next output value.

When the ADCStatus and ConvolverBusy input signals are asserted to indicate that the ADC and Convolver have received their start signal, the Controller FSM moves to the ADCWait state, disabling the ADCRead and ConvolverStart signals. It remains in this state until the ADC and the Convolver complete their processing, as indicated by the ADCStatus and ConvolverBusy signals both low.

The FSM then transitions to the ADCRead state, in which the ADC data bus and convolver result are read into the input and output buffers respectively. The FSM remains in this state for one clock cycle, then moves to the RAMWrite state. The most recently read input value from the ADC is stored in the RAM by setting the RAM address select signal to the value of the FSM's address counter, outputting the contents of the input buffer on the RAM data bus, and enabling the RAM's write enable signal (RAMWE).

After one cycle, the FSM returns to the WaitForTimer state and waits for the next pulse from the divider.

3.4 Convolver FSM

The Convolver FSM performs convolutions using the data values stored in the RAM and the impulse response coefficients stored in ROM. Since the impulse responses are 16 samples in length, the convolution is performed according to the equation

$$y[n] = \frac{1}{128} \sum_{i=0}^{15} x[n-i]h[i]$$

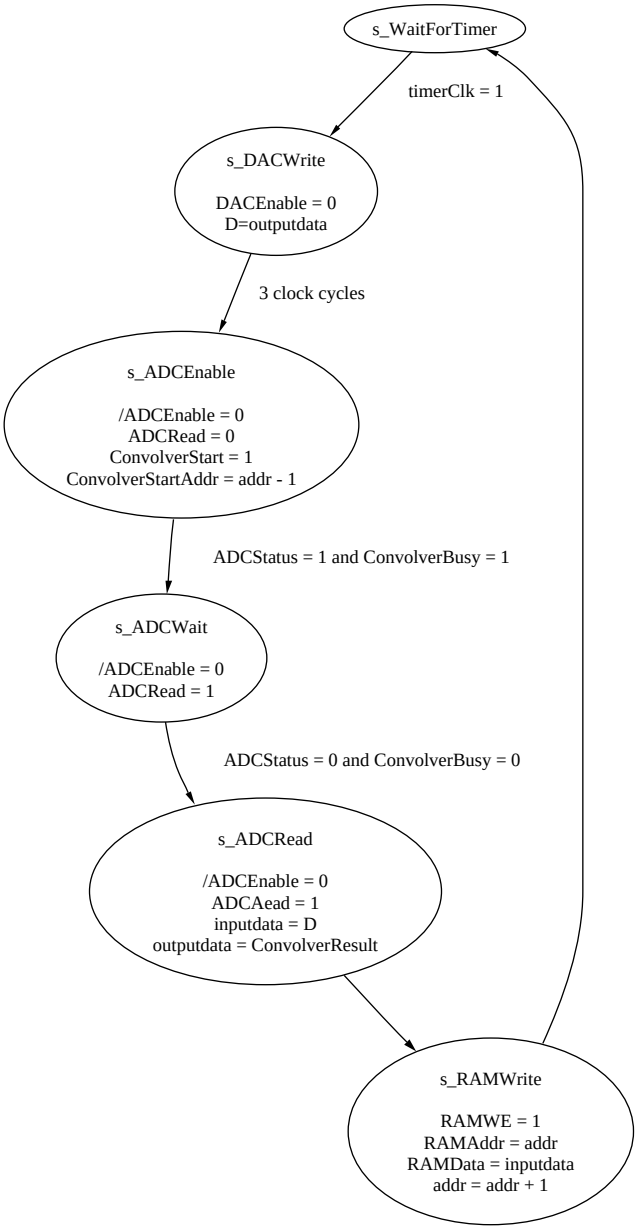

Figure 3: Control FSM state diagram

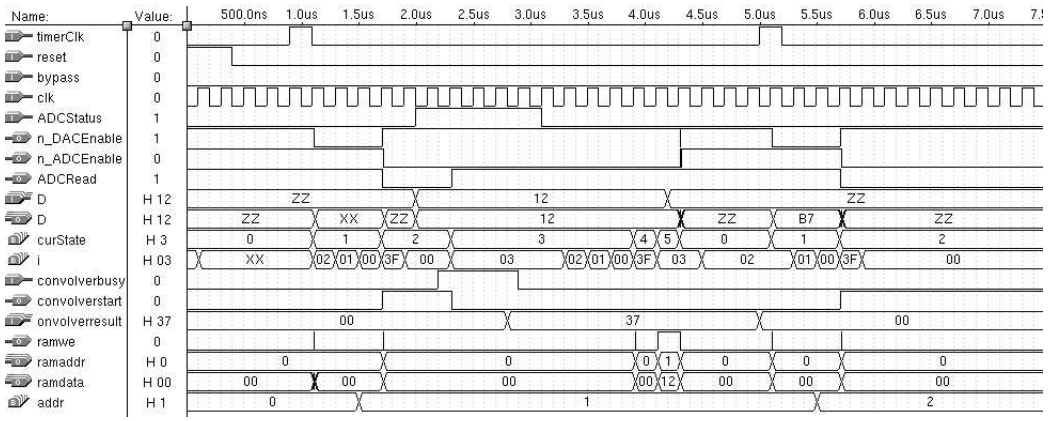

Figure 4: Timing diagram for Control FSM

which is, using the RAM and ROM implementation,

$$output = \frac{1}{128} \sum_{i=0}^{15} (RAM\,[base - i])\ (ROM\,[i])$$

The division by 128 is performed by dropping the lowest-order seven bits, and is done to scale the output by the maximum area under an impulse response (as specified in the impulse coefficients provided). This scaling ensures that the output is a proper 8-bit integer with appropriate magnitude relative to the input signal.

The convolver is implemented as a FSM with five states and a 4-bit counter variable i, in addition to a 15-bit accumulator. The five states and their transitions are depicted in the state diagram in Figure 5.

The convolver FSM is implemented using two processes that operate concurrently. One generates the next state as a function of the current state, and the other performs state transitions and updates the outputs on a rising edge of the clock.

The FSM is triggered by a start signal from the control FSM, and outputs a busy signal whenever it is currently performing a convolution. When the busy signal returns to its inactive low state, the result is available on the 8-bit output signal.

The convolver starts up in the Idle state. When the convolver receives the START signal from the control FSM, it also receives the memory offset of the current sample; this address is stored in the BaseAddr register. The convolver FSM then moves to the AddrSelect state, which outputs the appropriate values on the RAM and ROM address lines. For the RAM, this address is the value of BaseAddr - i - 1 (since BaseAddr is in fact the memory location of the sample currently being converted by the ADC while the convolver is running, we must subtract 1). The ROM address is simply i. The convolver remains in this state for one cycle to allow for the RAM and ROM access time, then moves to the StartMult state. In this state, the RAM and ROM address lines maintain their values, and the multiplier start signal is set high. When the multiplier starts and asserts its busy signal, the convolver FSM moves to the WaitMult state, and waits for the multiplier busy signal to return low, indicating that the multiplication is complete. When this signal is received, the value of the 15-bit multiplier result (in two's-complement format) is added to the accumulator. If the value of the i counter is 15, the FSM moves to the Output state; otherwise, i is incremented and the FSM returns to the AddrSelect state to compute the next partial product. In the Output state, the convolver truncates the value in the accumulator, returning the highest-order eight bits and sending them to the result output signal (again, in two's-complement format).

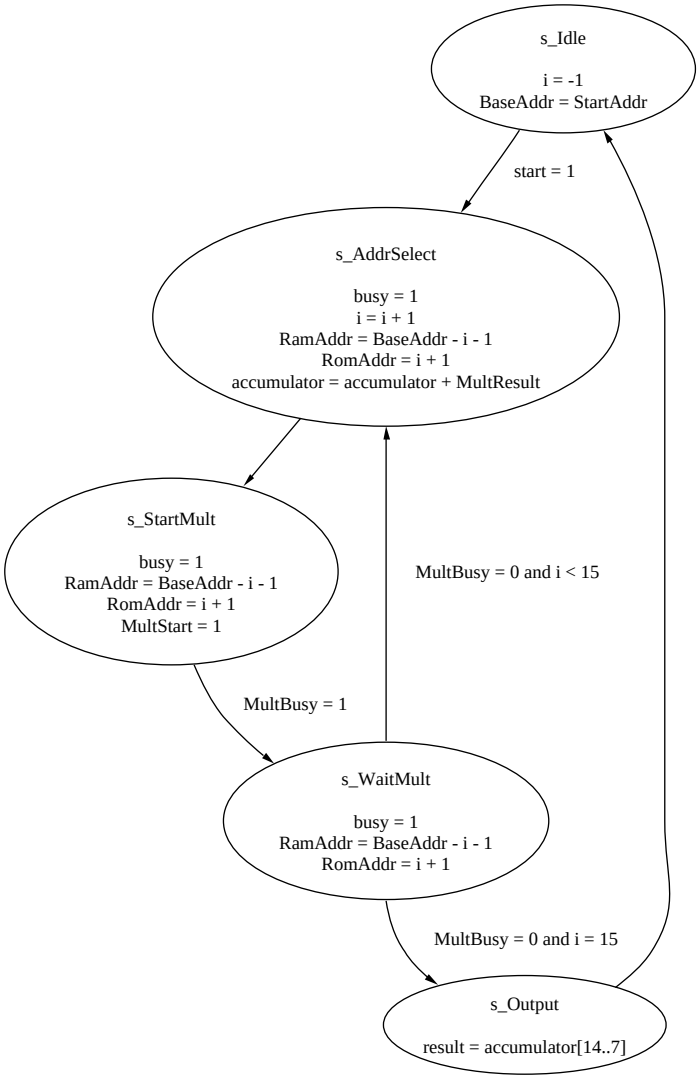

Figure 5: Convolver FSM state diagram

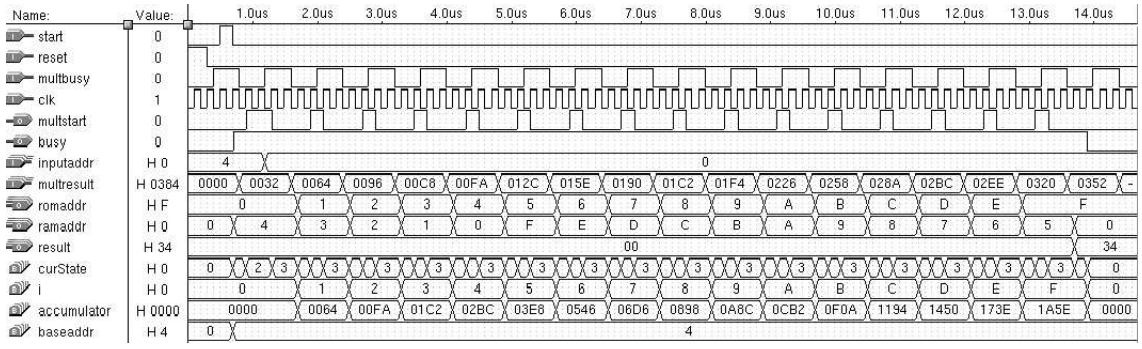

Figure 6: Timing diagram for Convolver FSM

3.5 Multiplier

An 8-bit sequential multiplier is used to generate the partial products in the convolution. The multiplier takes in two eight-bit integers in sign-magnitude format (1 sign bit followed by 7 magnitude bits), and generates a 15-bit sign-magnitude result. Each multiplication requires eight cycles to perform. The multiplier is triggered by a start signal from the convolver FSM, and outputs a busy signal whenever it is currently multiplying. When the busy signal returns to its inactive low state, the result is available.

Internally, the multiplier operates by maintaining two fifteen-bit shift registers, SREG and HREG, and a fifteen-bit accumulator. When the start signal is received, the A input is copied into the SREG register and the B input is copied into HREG. Each clock cycle, SREG shifts to the left and HREG shifts to the right. If the least significant bit of HREG is 1, SREG is added to the accumulator. After 8 cycles, the accumulator contains the product of A and B (ignoring sign). The sign bit is generated by taking the XOR of the sign bits from A and B.

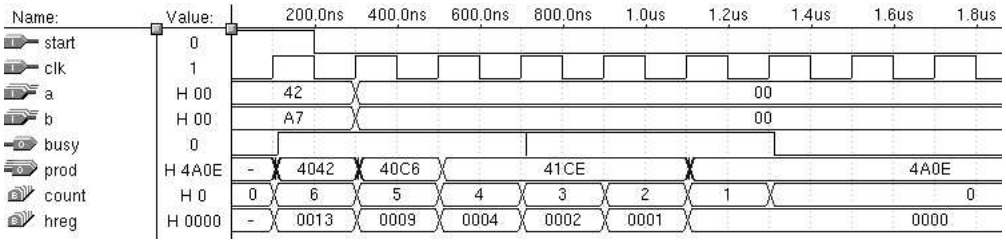

Figure 7: Multiplier timing diagram

3.6 Divider

The purpose of the divider is to generate a 44.287 kHz sample timer signal from the 10 MHz clock signal. The output of the divider is a signal that is high for one pulse of the system clock, each time a sample needs to be performed. This is done by maintaining an 8-bit counter that is incremented each clock pulse. When the counter reaches 225, the output is set high. On the next clock pulse (when the counter is 226), the counter is reset to zero and the output is set low again.

The divider also has a reset signal that allows the counter to be reset to zero. This is connected to the RESET input.

The divider was originally designed for use in Lab 2; it was adapted for use in this lab.

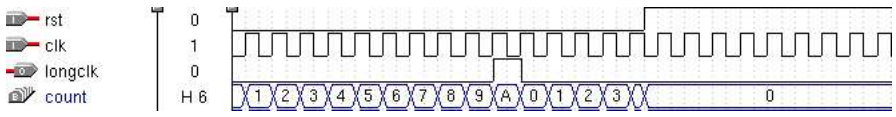

Figure 8: Divider timing diagram (ratio changed from 226:1 to 10:1)

3.7 Numeric Conversion

While the ADC and the convolver perform their computations with two's complement representations of numbers, the multiplier uses sign-magnitude representations for its input and output because this makes multiplication easier to implement. Hence, it is necessary to convert between the two representations. This is the purpose of the numeric conversion module. This module converts two's complement numbers to sign-magnitude, and vice versa.

Conversion is performed asynchronously by inverting all bits of the number and adding one, if the sign bit (most significant bit) is one; otherwise, the output is the same as the input. A VHDL generic construct is used to allow the width of the number to be specified.

An 8-bit numeric conversion module is used to convert the two's complmenet numbers stored in RAM into sign-magnitude representations to be input into the multiplier, and a 15-bit module is used to convert the sign-magnitude output of the multiplier to two's complement. The impulse coefficients stored in ROM are already in sign-magnitude format, so no conversion is necessary.

Figure 9: Numeric conversion timing diagram)

3.8 Analog-to-Digital Converter

An analog-to-digital converter (ADC) is required to convert the incoming analog signal to a digital representation. The Analog Devices AD670 chip is used to fulfill this requirement. The chip accepts a differential voltage input, and is configured for a bipolar ±1.28V range. The two's complement output format is selected, using the wiring configuration shown in Figure 2.

The control FSM handles the task of starting ADC conversions when appropriate. Because an ADC conversion requires over a hundred clock cycles to perform, it is necessary for the convolution arithmetic to be performed at the same time as the ADC conversion.

3.9 Digital-to-Analog Converter

The digital-to-analog converter (DAC) converts the computed output value to an analog voltage. This is implemented with an Analog Devices AD558 chip. The V_{out}, V_{out} Sense, and V_{out} Select outputs are tied together to select the +2.56V range. The DAC is powered by the lab kit's +5 Analog supply, and referenced to ground. The output range is 0 – +2.56V, so the zero value has a 1.28V offset relative to ground.

The DAC and ADC share a data bus. Because both chips are controlled by the Controller FSM, only one is active at any given time, and bus contention can never occur.

3.10 RAM

A 16 location by 8 bit memory is used for storing the 16 most recent input sample values from the ADC, in two's complement representation. A single port asynchronous RAM is used, using the Altera lpm_ram_dq predefined component. The D input is connected to the Controller FSM via an 8-bit bus, and the Q output is connected to the multiplier's B input via a numeric conversion module that converts the two's complement number read from RAM to sign-magnitude.

The write-enable (WE) line is connected to the RAMWE output on the Controller FSM. The WE line is gated by taking the logical AND of the RAMWE output from the Controller FSM with the inverse of the clock. Thus, the RAM is only enabled for writing when the clock is low and the Controller FSM has requested a write. This ensures that glitches on the RAMWE line caused by static hazards in the implementation of the Controller FSM do not cause spurious writes to overwrite data in the RAM.

A multiplexer is used to control access to the 4-bit RAM address bus. When the Convolver FSM's BUSY output is low, the multiplexer selects the Controller FSM's RAMADDR output; when the convolver is busy, the multiplexer instead selects the Convolver FSM's RAMADDR output. This ensures that whichever FSM is currently active controls the RAM address input. This is an effective scheme because only one FSM ever accesses the RAM at the same time; moreover, the Controller FSM only writes to the RAM and the Convolver FSM only selects addresses for the Multiplier module to read.

A dual-port RAM was originally selected in order to simplify this implementation. However, the FLEX10K10 FPGA could not implement a dual-port RAM without synthesizing it from discrete latches, and there was not enough space on the chip for this implementation. Hence, the RAM was replaced with a single-port RAM, and the multiplexer described above was added.

The Controller FSM maintains a counter to keep track of which location is to be filled next, and passes this offset to the Convolver FSM for use in computation. Because a 4-bit counter is used, it allows only 16 values. The counter is allowed to overflow, so address 15 is followed by address 0. This makes the memory function essentially as a circular buffer, beginning with the offset specified by the counter. In a similar manner, 4-bit integer overflow is allowed when the Convolver FSM calculates RAM addresses, so that the correct address will be determined.

3.11 ROM

A 256 location by 8 bit ROM is used to store the impulse response coefficients used in convolution calculations. These predefined values are provided by a file of hex values, in sign-magnitude format. The ROM operates asynchronously, with the address lines connected to the convolver's ROMADDR output, and the data bus connected directly to the multiplier's A input.

The ROM is implemented on the FPGA using Altera's lpm_rom library.

4 Testing and Debugging

All modules of the system were tested first in simulation using MAX+PLUS II software, then in hardware once the FPGA was programmed.

The numeric conversion module and multiplier were tested by providing them with a representative sample of input values, including both positive and negative numbers, and verifying that they produced the correct output valus.

To simplify testing the divider, it was modified slightly so that it generated a pulse every 10 cycles of the clock rather than every 226 (as in Figure 8). This made the behavior easier to visualize and validate in simulation. A clock signal was applied to the input, and the output and state of the internal counter were verified. The reset signal was also tested.

The two FSMs were tested in simulation using a waveform that tested the possible states and transitions. The values of the state variables and internal counters were used to verify that the transitions were being performed correctly, in addition to verifying that the FSMs produced the correct outputs at the correct times.

Prior to testing the arithmetic modules, the analog-to-digital and digital-to-analog converters and the control FSM and divider required to operate them were tested for the analog checkoff. This testing used a simplified version of the control FSM (listed in the appendix), and no convolver or other computational components. The FSM simply inverted the input signal and output it on the data bus to the DAC. The ADC was configured in offset binary mode rather than two's complement; this simplified implementation because the DAC accepts an offset binary input. The input signal was inverted to ensure that the FSM was actually controlling the DAC, and the data bus was not simply holding its value due to parasitic capacitance on a tristate bus.

The top-level structure combining all of the modules was next tested in simulation to verify that it produced the correct sequence of output values. After this simulation testing was completed, the FPGA was programmed and tested in hardware.

In addition to verifying the correct operation of the FSMs through the logic analyzer, testing included connecting a function generator to the input and monitoring the input and output on the digital oscilloscope. Filters were tested with sine and square waves of various frequencies. In particular, the first four impulse responses were used for testing, along with a square wave generated by the function generator. The input and output of these filters is shown in Figures 10, 11, 12, and 13.

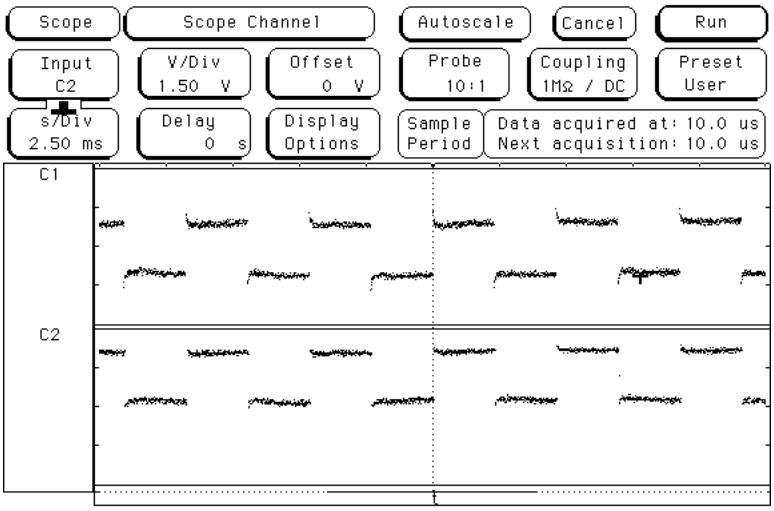

Figure 10: Input and output of test filter 0 (single positive impulse)

During testing, a small bug was discovered that caused occasional output samples to have radically different values than expected. Testing revealed that this discrepancy was caused by a bug in the numeric conversion module that caused invalid results to be generated when the sign of a sign-magnitude number was negative but the magnitude was zero (the "minus-zero" case). Logic was added to identify this case and produce the correct output value. Once this fix was implemented, the signal processor produced the correct values.

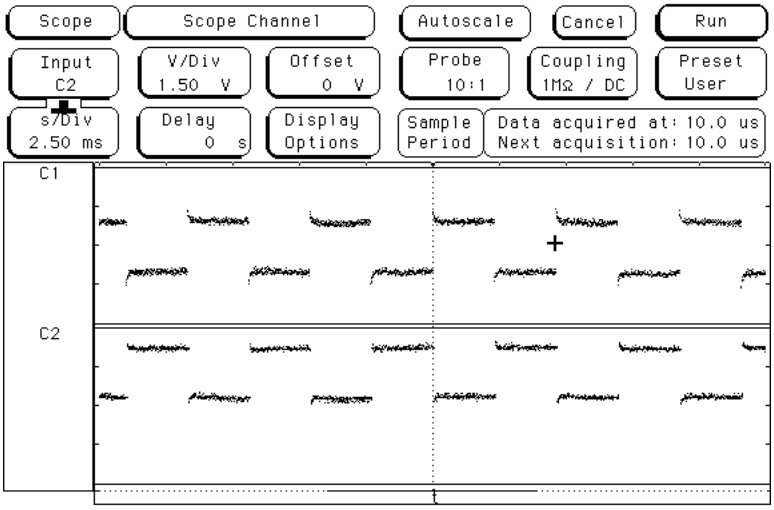

Figure 11: Input and output of test filter 1 (single negative impulse)

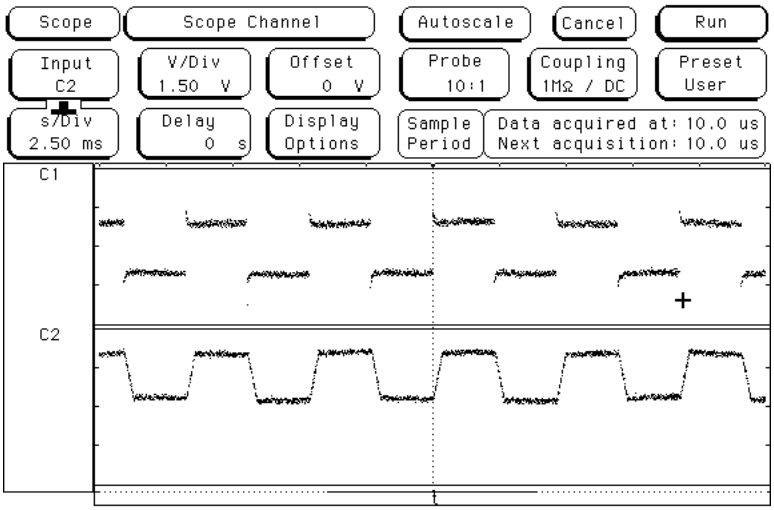

Figure 12: Input and output of test filter 2 (boxcar filter)

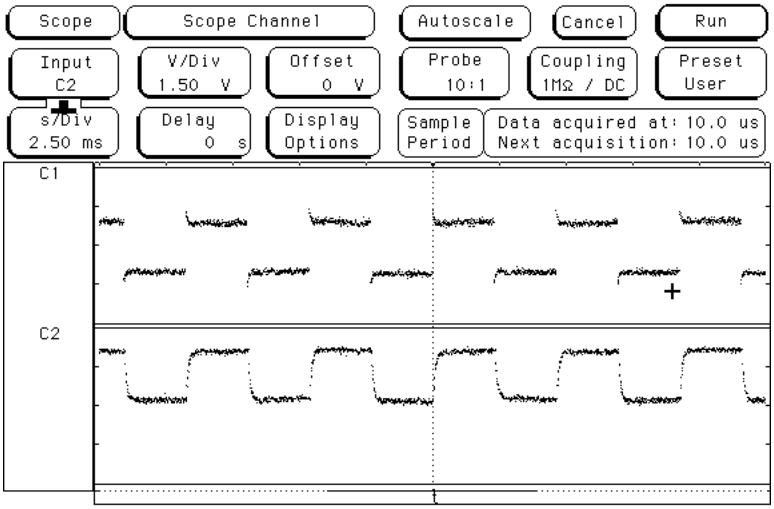

Figure 13: Input and output of test filter 3 (exponential filter)

5 Conclusion

A digital signal processor can be implemented using the method of finite impulse response convolution. It can be built using an analog-to-digital converter, digital-to-analog converter, two synchronous finite state machines, and a sequential multiplier, implemented using a FPGA and AD558 and AD670 chips. The processor described above was successfully implemented and tested.

A Appendix: VHDL Source

The following code was compiled with MAX+PLUS II and used to program the FPGA to implement this signal processor, along with the pin assignments represented in Figure 1.

A.1 top.vhd

```
-- top.vhd: top-level structural definition
-- Dan R. K. Ports  <drkp@mit.edu>
-- 6.111 Lab 3, 2003/10/22

library ieee;
use ieee.std_logic_1164.all;
library lpm;

entity top is
  port (
    n_ADCEnable, ADCRead : out   std_logic;  -- ADC control
    ADCStatus            : in    std_logic;  -- ADC status
    n_DACEnable          : out   std_logic;  -- DAC control
    reset                : in    std_logic;
    clk                  : in    std_logic;
    impswitches          : in    std_logic_vector(3 downto 0);
    bypass               : in    std_logic;
    D                    : inout std_logic_vector(7 downto 0));
                                 -- ADC/DAC data bus

end top;

architecture structural of top is

  component rom
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
  end component;

  component ram
        PORT
        (
                data            : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                address         : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                we              : IN STD_LOGIC  := '1';
                q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
  end component;

  component divider
    port (
```

```
    clk, rst : in  std_logic;
    longclk  : out std_logic);
end component;

component convolver
 port (
  clk               : in  std_logic;
  start             : in  std_logic;
  reset             : in  std_logic;
  busy              : out std_logic;
  inputaddr         : in  std_logic_vector(3 downto 0);
  multstart         : out std_logic;
  multbusy          : in  std_logic;
  multresult        : in  std_logic_vector(14 downto 0);
  romaddr, ramaddr  : out std_logic_vector(3 downto 0);
  result            : out std_logic_vector(7 downto 0));
end component;

component fsm
  port (
  n_ADCEnable, ADCRead : out   std_logic;  -- ADC control
  ADCStatus            : in    std_logic;  -- ADC status
  n_DACEnable          : out   std_logic;  -- DAC control
  reset                : in    std_logic;
  clk                  : in    std_logic;
  timerClk             : in    std_logic;
  convolverstart       : out   std_logic;
  convolverbusy        : in    std_logic;
  convolverresult      : in    std_logic_vector(7 downto 0);
  convolverstartaddr   : out   std_logic_vector(3 downto 0);
  ramaddr              : out   std_logic_vector(3 downto 0);
  ramdata              : out   std_logic_vector(7 downto 0);
  ramwe                : out   std_logic;
  bypass               : in   std_logic;
  D                    : inout std_logic_vector(7 downto 0));
                              -- ADC/DAC data bus

end component;

component multiplier
 port (
  a, b       : in  std_logic_vector(7 downto 0);  -- sign/mag
  prod       : out std_logic_vector(14 downto 0); -- sign/mag
  clk, start : in  std_logic;
  busy       : out std_logic);
end component;

component numconv
 generic (
  width : integer := 8);
```

```
    port (
      input  : in  std_logic_vector(width-1 downto 0);
      output : out std_logic_vector(width-1 downto 0));
  end component;

  signal timerClk : std_logic;
  signal convolverstart, convolverbusy : std_logic;
  signal convolverstartaddr : std_logic_vector(3 downto 0);
  signal convolverresult : std_logic_vector(7 downto 0);
  signal ramwe, ramwren : std_logic;
  signal ramwdata, ramrdata, ramdatasm, romdata
                                      : std_logic_vector(7 downto 0);
  signal ramwaddr, ramraddr, ramaddr : std_logic_vector(3 downto 0);
  signal romaddr : std_logic_vector(7 downto 0);
  signal romaddrl : std_logic_vector(3 downto 0);
  signal multstart, multbusy : std_logic;
  signal multresultsm, multresulttc : std_logic_vector(14 downto 0);

begin  -- structural

  controller : fsm port map (
    n_ADCEnable => n_ADCEnable,
    ADCRead     => ADCRead,
    n_DACEnable => n_DACEnable,
    ADCStatus   => ADCStatus,
    reset       => reset,
    clk         => clk,
    timerClk    => timerClk,
    D           => D,
    convolverstart => convolverstart,
    convolverbusy => convolverbusy,
    convolverstartaddr => convolverstartaddr,
    convolverresult =>  convolverresult,
    ramwe => ramwe,
    ramdata => ramwdata,
    bypass => bypass,
    ramaddr => ramwaddr);

  div : divider port map (
    clk     => clk,
    rst     => reset,
    longclk => timerClk);

  conv : convolver port map (
    clk                 => clk,
    start               => convolverstart,
    reset               => reset,
    busy                => convolverbusy,
    inputaddr           => convolverstartaddr,
    multstart           => multstart,
    multbusy            => multbusy,
```

```
    multresult          => multresulttc,
    romaddr             => romaddrl,
    ramaddr             => ramraddr,
    result              => convolverresult);

  mult : multiplier port map (
    a      => romdata,
    b      => ramdatasm,
    prod   => multresultsm,
    clk    => clk,
    start => multstart,
    busy   => multbusy);

  ramnc : numconv generic map (
    width => 8)
    port map (
      input  => ramrdata,
      output => ramdatasm);

  multnc : numconv generic map (
    width => 15)
    port map (
      input  => multresultsm,
      output => multresulttc);

  raminst : ram port map (
    data       => ramwdata,
    q          => ramrdata,
--    wraddress => ramwaddr,
    address => ramaddr,
--    clock        => clk,
    we       => ramwren);

  rominst : rom port map (
    address => romaddr,
    q       => romdata);

  ramaddr <= ramraddr when convolverbusy = '1' else ramwaddr;
  ramwren <= ramwe and not clk;
  romaddr <= impswitches & romaddrl;

end structural;
```

A.2 fsm.vhd

```
-- fsm.vhd: finite state machine controller
-- Dan R. K. Ports  <drkp@mit.edu>
-- 6.111 Lab 3, 2003/10/22

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

entity fsm is

  port (
    n_ADCEnable, ADCRead : out   std_logic;  -- ADC control
    ADCStatus            : in    std_logic;  -- ADC status
    n_DACEnable          : out   std_logic;  -- DAC control
    reset                : in    std_logic;
    clk                  : in    std_logic;
    timerClk             : in    std_logic;
    convolverstart       : out   std_logic;
    convolverbusy        : in    std_logic;
    convolverresult      : in    std_logic_vector(7 downto 0);
    convolverstartaddr   : out   std_logic_vector(3 downto 0);
    ramaddr              : out   std_logic_vector(3 downto 0);
    ramdata              : out   std_logic_vector(7 downto 0);
    ramwe                : out   std_logic;
    bypass               : in    std_logic;
    D                    : inout std_logic_vector(7 downto 0));
                           -- ADC/DAC data bus

end fsm;

architecture behavioral of fsm is

  type StateType is (s_WaitForTimer, s_DACWrite,
                     s_ADCEnable, s_ADCWait, s_ADCRead, s_RAMWrite);
  signal curState, nextState : StateType := s_WaitForTimer;

  signal i, nexti : std_logic_vector(7 downto 0);
                            -- iterator for multi-cycle waits
  signal addr : std_logic_vector(3 downto 0);
  signal inputdata,outputdata : std_logic_vector(7 downto 0);
begin  -- behavioral


  -- purpose: determine next state
  -- type    : combinational
  -- inputs : curState, i, ADCStatus
  -- outputs: nextState
  newState: process (curState, i, ADCStatus)
  begin  -- process newState
```

```
    if reset = '1' then
      nextState <= s_WaitForTimer;
    else
      case curState is
        when s_WaitForTimer =>
         if timerClk = '1' then
           nextState <= s_DACWrite;
           nexti <= "00000010";
         else
           nextState <= s_WaitForTimer;
         end if;
        when s_DACWrite =>
          if i = 0 then
            nextState <= s_ADCEnable;
          else
            nexti <= i-1;
            nextState <= s_DACWrite;
          end if;
        when s_ADCEnable =>
          if ADCStatus = '1' and convolverbusy = '1' then
            nextState <= s_ADCWait;
            nexti <= "00000011";
          else
            nextState <= s_ADCEnable;
          end if;
        when s_ADCWait =>
          if ADCStatus = '0' and convolverbusy = '0' then
            if i = 0 then
              nextState <= s_ADCRead;
            else
              nextState <= s_ADCWait;
              nexti <= i - 1;
            end if;
          else
            nextState <= s_ADCWait;
            nexti <= "00000011";
          end if;
        when s_ADCRead =>
--          nextState <= s_ADCDisable;
--          when s_ADCDisable =>
          nextState <= s_RAMWrite;
        when s_RAMWrite =>
          nextState <= s_WaitForTimer;
        when others =>
          nextState <= s_WaitForTimer;
      end case;
    end if;

  end process newState;
```

```
-- purpose: grab input value
-- type    : combinational
-- inputs : curState, D
-- outputs: inputdata
getinput: process (curState, D)
begin  -- process getinput
  if curState = s_ADCRead then
    inputdata <= D;
    outputdata <= (not (convolverresult(7))
                   & convolverresult(6 downto 0));
  end if;
end process getinput;

-- purpose: update current state and i on clock
-- type    : combinational
-- inputs : clk, nextState, nexti
-- outputs: curState, i
state_clocked: process (clk)
begin  -- process state_clocked
  if rising_edge(clk) then
    curState <= nextState;
    i <=  nexti;
    if nextState = s_DACWrite and nexti=0 then
      addr <= addr+1;
    elsif reset = '1' then
      addr <= (others => '0');
    end if;
  end if;
end process state_clocked;

-- purpose: generate output values
-- type    : combinational
-- inputs : curState, D
-- outputs: ADCRead, n_ADCEnable, n_DACEnable, D
gen_outputs: process (curState)
begin  -- process gen_outputs
  case curState is
    when s_WaitForTimer => ADCRead <= '1';
                           n_DACEnable <= '1';
                           n_ADCEnable <=  '1';
                           D <= (others => 'Z');
                           convolverstart <= '0';
                           ramwe <= '0';
                         ramaddr <= (others => '0');
                         ramdata <= (others => '0');
    when s_DACWrite => ADCRead <= '1';
                       n_DACEnable <= '0';
                       n_ADCEnable <= '1';
                       if bypass = '0' then
                         D <= outputdata;
                       else
```

```
                    D <= inputdata;
                  end if;
                  convolverstart <= '0';
                  ramwe <= '0';
                ramaddr <= (others => '0');
                ramdata <= (others => '0');
      when s_ADCEnable => ADCRead <= '0';
                    n_DACEnable <= '1';
                   D <= (others => 'Z');
                    n_ADCEnable <=  '0';
                    convolverstart <= '1';
                    convolverstartaddr <= addr - 1;
                    ramwe <= '0';
                  ramaddr <= (others => '0');
                  ramdata <= (others => '0');
      when s_ADCWait => ADCRead <= '1';
                  n_DACEnable <= '1';
                  D <= (others => 'Z');
                  n_ADCEnable <=  '0';
                  convolverstart <= '0';
                  ramwe <= '0';
                  ramaddr <= (others => '0');
                  ramdata <= (others => '0');
      when s_ADCRead => ADCRead <= '1';
                  n_DACEnable <= '1';
                  D <= (others => 'Z');
                  n_ADCEnable <=  '0';
                 -- Dbuf <= D;
                  convolverstart <= '0';
                  ramwe <= '0';
                  ramaddr <= (others => '0');
                  ramdata <= (others => '0');
      when s_RAMWrite => ADCRead <= '1';
                    n_DACEnable <= '1';
                    n_ADCEnable <= '0';
                    convolverstart <= '0';
                    ramwe <= '1';
                    ramaddr <= addr;
                    ramdata <= inputdata;
      when others => ADCRead <= '1';
                  n_DACEnable <= '1';
                  n_ADCEnable <=  '1';
                  D <= (others => 'Z');
    end case;

  end process gen_outputs;


end behavioral;
```

A.3 convolver.vhd

```
-- convolver.vhd: convolver finite state machine
-- Dan R. K. Ports  <drkp@mit.edu>
-- 6.111 Lab 3, 2003/10/22

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

entity convolver is

  port (
    clk               : in  std_logic;
    start             : in  std_logic;
    reset             : in  std_logic;
    busy              : out std_logic;
    inputaddr         : in  std_logic_vector(3 downto 0);
    multstart         : out std_logic;
    multbusy          : in  std_logic;
    multresult        : in  std_logic_vector(14 downto 0);
    romaddr, ramaddr  : out std_logic_vector(3 downto 0);
    result            : out std_logic_vector(7 downto 0));

end convolver;

architecture behavioral of convolver is

  type statetype is (s_idle, s_addrselect, s_startmult,
                     s_waitmult, s_output);
  signal curState, nextState : statetype;
  signal baseaddr : std_logic_vector(3 downto 0);

  signal i : std_logic_vector(3 downto 0);
  signal accumulator : std_logic_vector(14 downto 0);

begin  -- behavioral

  -- purpose: generate next state
  -- type    : combinational
  -- inputs : curState, multbusy, start, i
  -- outputs: nextState
  gennextstate: process (curState, multbusy, start, i)
  begin  -- process nextstate

    if reset = '1' then
      nextState <= s_idle;
    else

    case curState is
```

```
    when s_idle =>
      if start = '1' then
        nextState <= s_addrselect;
      else
        nextState <= s_idle;
      end if;
    when s_addrselect => nextState <= s_startmult;
    when s_startmult =>
      if multbusy = '1' then
        nextState <= s_waitmult;
      else
        nextState <= s_startmult;
      end if;
    when s_waitmult =>
      if multbusy = '0' then
        if i = 15 then
          nextState <=  s_output;
        else
          nextState <=  s_addrselect;
        end if;
      else
        nextState <=  s_waitmult;
      end if;
    when s_output => nextState <= s_idle;
    when others => nextState <= s_idle;
  end case;
end if;

end process gennextstate;

-- purpose: perform state transitions and outputs
-- type    : combinational
-- inputs : clk
-- outputs: curState, i, accumulator, romaddr, ramaddr
state_clocked: process (clk)
begin  -- process state_clocked
  if rising_edge(clk) then
    curState <= nextState;

    case nextState is
      when s_idle => i <= (others => '0');
                     accumulator <= (others => '0');
                     multstart <= '0';
                     busy <= '0';
                     ramaddr <= (others => '0');
      when s_addrselect =>
        multstart <= '0';
        busy <= '1';
        if curState = s_waitmult then
          i <= i + 1;
          ramaddr <= baseaddr - i - 1;
```

```
            romaddr <= i + 1;
            accumulator <= accumulator + multresult;
          else
            i <= (others => '0');
            baseaddr <= inputaddr;
            romaddr <= (others => '0');
            ramaddr <= inputaddr;
          end if;
        when s_startmult => multstart <= '1';
                            busy <= '1';
        when s_waitmult => multstart <= '0';
                           busy <= '1';
        when s_output => multstart <= '0';
                         busy <= '1';
                         result <= accumulator(14 downto 7);
                       ramaddr <= (others => '0');
        when others => null;
      end case;
    end if;
  end process state_clocked;

end behavioral;
```

A.4 multiplier.vhd

```
-- multiplier.vhd: 8-bit synchronous multiplier
-- Dan R. K. Ports  <drkp@mit.edu>
-- 6.111 Lab 3, 2003/10/22

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

entity multiplier is

  port (
    a, b        : in  std_logic_vector(7 downto 0);  -- sign/mag
    prod        : out std_logic_vector(14 downto 0); -- sign/mag
    clk, start  : in  std_logic;
    busy        : out std_logic);

end multiplier;

architecture behavioral of multiplier is

  signal sreg, hreg : std_logic_vector(14 downto 0);
  signal accumulator : std_logic_vector(14 downto 0);
  signal count : std_logic_vector(3 downto 0);
  signal sign : std_logic;

begin  -- behavioral

  -- purpose: perform multiplication
  -- type    : sequential
  -- inputs : clk, start, a, b
  -- outputs: accumulator, count
  multiply: process (clk, start)
  begin  -- process multiply

    if rising_edge(clk) then
      if start = '1' then
        count <= "0110";
        sreg <= "0000000" & a(6 downto 0) & '0';
        hreg <= "000000000" & b(6 downto 1);
        sign <= a(7) xor b(7);
        if b(0) = '1' then
          accumulator <= "0000000" & a;
        else
          accumulator <= (others => '0');
        end if;
      elsif count > 0 then
        count <= count - 1;
        if hreg(0) = '1' then
          accumulator <= accumulator + sreg;
```

```
        end if;
        hreg <= '0' & hreg(14 downto 1);
        sreg <= sreg(13 downto 0) & '0';
      end if;
    end if;

  end process multiply;

  busy <= '0' when count = 0 else '1';
  prod <= sign & accumulator(13 downto 0);

end behavioral;
```

A.5 numconv.vhd

```
-- numconv.vhd: numeric conversion between two's complement
--   and sign-magnitude
-- Dan R. K. Ports  <drkp@mit.edu>
-- 6.111 Lab 3, 2003/10/22

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

entity numconv is

  generic (
    width : integer := 8);

  port (
    input  : in  std_logic_vector(width-1 downto 0);
    output : out std_logic_vector(width-1 downto 0));

end numconv;

architecture behavioral of numconv is

begin  -- behavioral

  -- purpose: convert input between sign/magnitude and two's complement
  -- type   : combinational
  -- inputs : input
  -- outputs: output
  convert: process (input)
  begin  -- process convert
    if input(width-1) = '0' then
      output <= input;
    else

      if input(width-2 downto 0) = 0 then
        output <= (others => '0');
      else
        output(width-1) <= '1';
        output(width-2 downto 0) <= (not input(width-2 downto 0)) + 1;
      end if;

    end if;
  end process convert;

end behavioral;
```

A.6 divider.vhd

```
-- divider.vhd: 10 MHz to 44.248 kHz clock divider
-- Dan R. K. Ports  <drkp@mit.edu>
-- 6.111 Lab 2, 2003/10/01
-- Modified for 6.111 Lab 3, 2003/10/15

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

entity divider is

  port (
    clk, rst : in  std_logic;
    longclk  : out std_logic);

end divider;

architecture behavioral of divider is

  signal longclkstate : std_logic := '0';
  signal count : std_logic_vector(24 downto 0);

begin  -- behavioral

  -- purpose: generate output pulse when counter reaches 1000000
  -- type   : sequential
  -- inputs : clk, rst
  -- outputs: longclk
  process (clk, rst)
  begin  -- process
    if rst = '1' then           -- asynchronous reset (active high)
      count <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      count <= count + 1;
    if count = 225 then
      longclkstate <= '1';
    elsif count = 226 then
      count <= (others => '0');
      longclkstate <= '0';
    end if;
    end if;


  end process;

  longclk <= longclkstate;

end behavioral;
```

A.7 ram.vhd

```
-- megafunction wizard: %LPM_RAM_DQ%
-- GENERATION: STANDARD
-- VERSION: WM1.0
-- MODULE: lpm_ram_dq

--============================================================
-- File Name: ram.vhd
-- Megafunction Name(s):
--                      lpm_ram_dq
--============================================================
-- ************************************************************
-- THIS IS A WIZARD GENERATED FILE. DO NOT EDIT THIS FILE!
-- ************************************************************


--      Copyright (C) 1988-2002 Altera Corporation

--      Any megafunction design, and related net list (encrypted or decrypted),
--      support information, device programming or simulation file, and any other
--      associated documentation or information provided by Altera or a partner
--      under Altera's Megafunction Partnership Program may be used only to
--      program PLD devices (but not masked PLD devices) from Altera. Any other
--      use of such megafunction design, net list, support information, device
--      programming or simulation file, or any other related documentation or
--      information is prohibited for any other purpose, including, but not
--      limited to modification, reverse engineering, de-compiling, or use with
--      any other silicon devices, unless such use is explicitly licensed under
--      a separate agreement with Altera or a megafunction partner. Title to
--      the intellectual property, including patents, copyrights, trademarks,
--      trade secrets, or maskworks, embodied in any such megafunction design,
--      net list, support information, device programming or simulation file, or
--      any other related documentation or information provided by Altera or a
--      megafunction partner, remains with Altera, the megafunction partner, or
--      their respective licensors. No other licenses, including any licenses
--      needed under any third party's intellectual property, are provided herein.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY ram IS
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                we              : IN STD_LOGIC  := '1';
                data            : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
END ram;
```

```
ARCHITECTURE SYN OF ram IS

        SIGNAL sub_wire0          : STD_LOGIC_VECTOR (7 DOWNTO 0);



        COMPONENT lpm_ram_dq
        GENERIC (
                lpm_width                 : NATURAL;
                lpm_widthad               : NATURAL;
                lpm_indata                : STRING;
                lpm_address_control               : STRING;
                lpm_outdata               : STRING;
                lpm_hint                  : STRING
        );
        PORT (
                        address : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
                        q       : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
                        data    : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                        we      : IN STD_LOGIC
        );
        END COMPONENT;

BEGIN
        q    <= sub_wire0(7 DOWNTO 0);

        lpm_ram_dq_component : lpm_ram_dq
        GENERIC MAP (
                LPM_WIDTH => 8,
                LPM_WIDTHAD => 4,
                LPM_INDATA => "UNREGISTERED",
                LPM_ADDRESS_CONTROL => "UNREGISTERED",
                LPM_OUTDATA => "UNREGISTERED",
                LPM_HINT => "USE_EAB=ON"
        )
        PORT MAP (
                address => address,
                data => data,
                we => we,
                q => sub_wire0
        );



END SYN;

-- ============================================================
-- CNX file retrieval info
-- ============================================================
-- Retrieval info: PRIVATE: WidthData NUMERIC "8"
```

```
-- Retrieval info: PRIVATE: WidthAddr NUMERIC "4"
-- Retrieval info: PRIVATE: RegData NUMERIC "0"
-- Retrieval info: PRIVATE: RegAdd NUMERIC "0"
-- Retrieval info: PRIVATE: OutputRegistered NUMERIC "0"
-- Retrieval info: PRIVATE: BlankMemory NUMERIC "1"
-- Retrieval info: PRIVATE: MIFfilename STRING ""
-- Retrieval info: PRIVATE: UseLCs NUMERIC "0"
-- Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"
-- Retrieval info: CONSTANT: LPM_WIDTH NUMERIC "8"
-- Retrieval info: CONSTANT: LPM_WIDTHAD NUMERIC "4"
-- Retrieval info: CONSTANT: LPM_INDATA STRING "UNREGISTERED"
-- Retrieval info: CONSTANT: LPM_ADDRESS_CONTROL STRING "UNREGISTERED"
-- Retrieval info: CONSTANT: LPM_OUTDATA STRING "UNREGISTERED"
-- Retrieval info: CONSTANT: LPM_HINT STRING "USE_EAB=ON"
-- Retrieval info: USED_PORT: address 0 0 4 0 INPUT NODEFVAL address[3..0]
-- Retrieval info: USED_PORT: we 0 0 0 0 INPUT VCC we
-- Retrieval info: USED_PORT: q 0 0 8 0 OUTPUT NODEFVAL q[7..0]
-- Retrieval info: USED_PORT: data 0 0 8 0 INPUT NODEFVAL data[7..0]
-- Retrieval info: CONNECT: @address 0 0 4 0 address 0 0 4 0
-- Retrieval info: CONNECT: @we 0 0 0 0 we 0 0 0 0
-- Retrieval info: CONNECT: q 0 0 8 0 @q 0 0 8 0
-- Retrieval info: CONNECT: @data 0 0 8 0 data 0 0 8 0
```

A.8 rom.vhd

```
-- megafunction wizard: %LPM_ROM%
-- GENERATION: STANDARD
-- VERSION: WM1.0
-- MODULE: lpm_rom

--============================================================
-- File Name: rom.vhd
-- Megafunction Name(s):
--                         lpm_rom
--============================================================
-- ************************************************************
-- THIS IS A WIZARD GENERATED FILE. DO NOT EDIT THIS FILE!
-- ************************************************************


--        Copyright (C) 1988-2002 Altera Corporation

--        Any megafunction design, and related net list (encrypted or decrypted),
--        support information, device programming or simulation file, and any other
--        associated documentation or information provided by Altera or a partner
--        under Altera's Megafunction Partnership Program may be used only to
--        program PLD devices (but not masked PLD devices) from Altera. Any other
--        use of such megafunction design, net list, support information, device
--        programming or simulation file, or any other related documentation or
--        information is prohibited for any other purpose, including, but not
--        limited to modification, reverse engineering, de-compiling, or use with
--        any other silicon devices, unless such use is explicitly licensed under
--        a separate agreement with Altera or a megafunction partner. Title to
--        the intellectual property, including patents, copyrights, trademarks,
--        trade secrets, or maskworks, embodied in any such megafunction design,
--        net list, support information, device programming or simulation file, or
--        any other related documentation or information provided by Altera or a
--        megafunction partner, remains with Altera, the megafunction partner, or
--        their respective licensors. No other licenses, including any licenses
--        needed under any third party's intellectual property, are provided herein.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY rom IS
        PORT
        (
                address         : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
END rom;


ARCHITECTURE SYN OF rom IS
```

```
        SIGNAL sub_wire0          : STD_LOGIC_VECTOR (7 DOWNTO 0);



        COMPONENT lpm_rom
        GENERIC (
                lpm_width               : NATURAL;
                lpm_widthad             : NATURAL;
                lpm_address_control             : STRING;
                lpm_outdata             : STRING;
                lpm_file                : STRING
        );
        PORT (
                        address : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                        q       : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
        END COMPONENT;

BEGIN
        q    <= sub_wire0(7 DOWNTO 0);

        lpm_rom_component : lpm_rom
        GENERIC MAP (
                LPM_WIDTH => 8,
                LPM_WIDTHAD => 8,
                LPM_ADDRESS_CONTROL => "UNREGISTERED",
                LPM_OUTDATA => "UNREGISTERED",
                LPM_FILE => "/mit/6.111-nfs/handouts/labs/lab3.f2003/impulses.hex"
        )
        PORT MAP (
                address => address,
                q => sub_wire0
        );



END SYN;

-- ============================================================
-- CNX file retrieval info
-- ============================================================
-- Retrieval info: PRIVATE: WidthData NUMERIC "8"
-- Retrieval info: PRIVATE: WidthAddr NUMERIC "8"
-- Retrieval info: PRIVATE: RegAdd NUMERIC "0"
-- Retrieval info: PRIVATE: OutputRegistered NUMERIC "0"
-- Retrieval info: PRIVATE: MIFfilename STRING "/mit/6.111-nfs/handouts/labs/lab3.f2003/
-- Retrieval info: CONSTANT: LPM_WIDTH NUMERIC "8"
-- Retrieval info: CONSTANT: LPM_WIDTHAD NUMERIC "8"
-- Retrieval info: CONSTANT: LPM_ADDRESS_CONTROL STRING "UNREGISTERED"
-- Retrieval info: CONSTANT: LPM_OUTDATA STRING "UNREGISTERED"
```

```
-- Retrieval info: CONSTANT: LPM_FILE STRING "/mit/6.111-nfs/handouts/labs/lab3.f2003/i
-- Retrieval info: USED_PORT: address 0 0 8 0 INPUT NODEFVAL address[7..0]
-- Retrieval info: USED_PORT: q 0 0 8 0 OUTPUT NODEFVAL q[7..0]
-- Retrieval info: CONNECT: @address 0 0 8 0 address 0 0 8 0
-- Retrieval info: CONNECT: q 0 0 8 0 @q 0 0 8 0
```

A.9 impulses.hex

```
:080000007F0000000000000079
:0800080000000000000000F0
:08001000FF00000000000000E9
:0800180000000000000000E0
:0800200008080808080808098
:0800280008080808080808090
:080030002018120E0A08060454
:0800380003020201010100B5
:08004000000082838109182 4ED
:0800480024180981838200 00E5
:0800500083030482 8A82183147
:080058003118828A820403833F
:0800600000000000000004058
:08006800C000000000000000D0
:0800700000010084050AA5331C
:08007800A50A05840001000047
:0800800081040087070BA43185
:08008800A40B078700048100AE
:08009000030B0486918E03149A
:0800980014038E9186040B0392
:0800A00006038C021087920E8A
:0800A8000E928710028C030088
:0800B000008A0A8587139408F9
:0800B80008941287850A8A00F2
:0800C0008888000D11068C6315
:0800C8008C06110D0088880070
:0800D00086070A8B8712036307
:0800D8000312878B0A07860062
:0800E0000801880F8D010E5983
:0800E8000E018D0F88010806CE
:0800F0000087891D0508034586
:0800F8000308051D89870000C3
:00000001FF
```

B Appendix: VHDL Source for Analog Checkoff

The following two additional source files were used to program the FPGA for the analog checkoff, in which the digital-to-analog and analog-to-digital converters were tested, along with a simple control FSM in the FPGA. Note also that the ADC was wired in the offset binary rather than two's complement configuration for the analog checkoff.

B.1 top-analog.vhd

```
-- top-analog.vhd: top level structural definition for analog checkoff
-- Dan R. K. Ports  <drkp@mit.edu>
-- 6.111 Lab 3, 2003/10/15

library ieee;
use ieee.std_logic_1164.all;

entity top is
  port (
    n_ADCEnable, ADCRead : out   std_logic;  -- ADC control
    ADCStatus            : in    std_logic;  -- ADC status
    n_DACEnable          : out   std_logic;  -- DAC control
    reset                : in    std_logic;
    clk                  : in    std_logic;
    D                    : inout std_logic_vector(7 downto 0));
                           -- ADC/DAC data bus

end top;

architecture structural of top is

  component fsm
     port (
    n_ADCEnable, ADCRead : out   std_logic;  -- ADC control
    ADCStatus            : in    std_logic;  -- ADC status
    n_DACEnable          : out   std_logic;  -- DAC control
    reset                : in    std_logic;
    clk                  : in    std_logic;
    timerClk             : in    std_logic;
    Dout                 : out   std_logic_vector(7 downto 0);
    D                    : inout std_logic_vector(7 downto 0));
                           -- ADC/DAC data bus
  end component;

  component divider
    port (
      clk, rst : in  std_logic;
      longclk  : out std_logic);
  end component;

  signal timerClk : std_logic;
```

```
begin  -- structural

  controller : fsm port map (
    n_ADCEnable => n_ADCEnable,
    ADCRead     => ADCRead,
    n_DACEnable => n_DACEnable,
    ADCStatus   => ADCStatus,
    reset       => reset,
    clk         => clk,
    timerClk    => timerClk,
    D           => D);

  div : divider port map (
    clk     => clk,
    rst     => reset,
    longclk => timerClk);

end structural;
```

B.2 fsm-analog.vhd

```
-- fsm-analog.vhd: controller finite state machine for analog checkoff
-- Dan R. K. Ports  <drkp@mit.edu>
-- 6.111 Lab 3, 2003/10/15

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

entity fsm is

  port (
    n_ADCEnable, ADCRead : out   std_logic;  -- ADC control
    ADCStatus            : in    std_logic;  -- ADC status
    n_DACEnable          : out   std_logic;  -- DAC control
    reset                : in    std_logic;
    clk                  : in    std_logic;
    timerClk             : in    std_logic;
    Dout                 : out   std_logic_vector(7 downto 0);
    D                    : inout std_logic_vector(7 downto 0));
                          -- ADC/DAC data bus

end fsm;

architecture behavioral of fsm is

  type StateType is (s_WaitForTimer, s_DACWrite,
                     s_ADCEnable, s_ADCWait, s_ADCRead);
  signal curState, nextState : StateType := s_WaitForTimer;

  signal i, nexti : std_logic_vector(7 downto 0);
                            -- iterator for multi-cycle waits
  signal Dbuf : std_logic_vector(7 downto 0);
begin  -- behavioral

  Dout <= Dbuf;

  -- purpose: determine next state
  -- type   : combinational
  -- inputs : curState, i, ADCStatus
  -- outputs: nextState
  newState: process (curState, i, ADCStatus)
  begin  -- process newState

    if reset = '1' then
      nextState <= s_WaitForTimer;
    else
      case curState is
        when s_WaitForTimer =>
          if timerClk = '1' then
```

```
            nextState <= s_DACWrite;
            nexti <= "00000010";
          else
            nextState <= s_WaitForTimer;
          end if;
        when s_DACWrite =>
          if i = 0 then
            nextState <= s_ADCEnable;
          else
            nexti <= i-1;
            nextState <= s_DACWrite;
          end if;
        when s_ADCEnable =>
          if ADCStatus = '1' then
            nextState <= s_ADCWait;
            nexti <= "00000011";
          else
            nextState <= s_ADCEnable;
          end if;
        when s_ADCWait =>
          if ADCStatus = '0' then
            if i = 0 then
              nextState <= s_ADCRead;
            else
              nextState <= s_ADCWait;
              nexti <= i - 1;
            end if;
          else
            nextState <= s_ADCWait;
            nexti <= "00000011";
          end if;
        when s_ADCRead =>
--          nextState <= s_ADCDisable;
--          when s_ADCDisable =>
          nextState <= s_WaitForTimer;
        when others =>
          nextState <= s_WaitForTimer;
      end case;
    end if;

  end process newState;

  -- purpose: grab input value
  -- type   : combinational
  -- inputs : curState, D
  -- outputs: Dbuf
  getinput: process (curState, D)
  begin  -- process getinput
    if curState = s_ADCRead then
      Dbuf <= D;
    end if;
```

```
end process getinput;

-- purpose: update current state and i on clock
-- type   : combinational
-- inputs : clk, nextState, nexti
-- outputs: curState, i
state_clocked: process (clk)
begin  -- process state_clocked
  if rising_edge(clk) then
    curState <= nextState;
    i <=  nexti;
  end if;
end process state_clocked;

-- purpose: generate output values
-- type   : combinational
-- inputs : curState, D
-- outputs: ADCRead, n_ADCEnable, n_DACEnable, D
gen_outputs: process (curState)
begin  -- process gen_outputs
  case curState is
    when s_WaitForTimer => ADCRead <= '1';
                           n_DACEnable <= '1';
                           n_ADCEnable <=  '1';
                           D <= (others => 'Z');
    when s_DACWrite => ADCRead <= '1';
                       n_DACEnable <= '0';
                       n_ADCEnable <= '1';
                       D <= Dbuf;
    when s_ADCEnable => ADCRead <= '0';
                        n_DACEnable <= '1';
                        D <= (others => 'Z');
                        n_ADCEnable <=  '0';
    when s_ADCWait => ADCRead <= '1';
                      n_DACEnable <= '1';
                      D <= (others => 'Z');
                      n_ADCEnable <=  '0';
    when s_ADCRead => ADCRead <= '1';
                      n_DACEnable <= '1';
                      D <= (others => 'Z');
                      n_ADCEnable <=  '0';
                     -- Dbuf <= D;
    when others => ADCRead <= '1';
                   n_DACEnable <= '1';
                   n_ADCEnable <=  '1';
                   D <= (others => 'Z');
  end case;

end process gen_outputs;
```

```
end behavioral;
```