Lab Exercise 3

Contents

1 Overview

1.1 Multiplication

1.1.1 The Spartan-3 18 × 18 multiplier

1.1.2 Sequential multiplier

1.1.3 Serial-parallel multiplier

1.1.4 Bit-serial multiply-and-add

1.1.5 The MAC test entity

1.2 Discarding bits and rounding

1.3 Bit-serial finite impulse response filter

1.3.1 The bit=serial FIR entity

1.3.2 The bit-serial FIR stage entity

1.4 Test filter coefficient values

2 The Exercise

2.1 The MAC entity

2.2 Implementing convergent rounding

2.3 Using more bits in the MAC

2.4 Bit-serial FIR implementation

A MAC test source code listings

A.1 MAC test entity source code listing

A.2 The MAC entity VHDL

B FIR_test_V01 listings

B.1 FIR_test_V01.vhd

B.2 SerialFIR

B.3 FIRstage.vhd

C The serial-parallel multiplier entity VHDL listing

D One-bit full adder entity VHDL listing
1 Overview

Exercise 1 introduced the Spartan-3 Starter Board, its peripherals and closed doing parallel and bit-serial addition/subtraction. Exercise 2 took up waveform generation, digital to analog conversion and analog to digital conversion. Exercise 3 continues with bit-serial multiplication, truncation and rounding and closes with the implementation of bit-serial finite impulse response (FIR) filter. The filter combines the bit-serial addition and multiplication operations and is visually tested using analog input and digital output.

This exercise brings to a close the FPGA portion of the Workshop’s structured exercises. Next week we start on a series of three exercises using the TI TMS320F28027 Piccolo ControlSTICK.

1.1 Multiplication

Multiplication implemented combining bits entirely combinatorially requires lots of gates. Bit serial multiplication needs, relatively speaking, much fewer gates. This exercise focuses on bit-serial multiplication. The bit-serial multiplier architecture that we will be working with is termed serial-parallel.

When doing DSP calculations using a $B$ bit word size it is very common to use values in $Q(B - 1)$ form. Multiplying two $Q(B - 1)$ values results in a $Q(2B - 2)$ result. Discarding the low $B - 1$ bits gives back a $Q(B - 1)$ result. In order to minimize a bias in the result, the $Q(2B - 2)$ values should be rounded prior to truncation.

There are many applications where one wants to position the binary point differently but the basic ideas and means are the same as used in this exercise.

1.1.1 The Spartan-3 18 × 18 multiplier

The Spartan-3/1000 possesses 24 high speed 18-bit bit-parallel multipliers. These are located off-fabric and connect into the fabric.

We will not be making use of these in the lab exercises. Feel free to explore them on your own.

1.1.2 Sequential multiplier

Figure contains the block diagram of a sequential multiplier. Multiplies the multiplicand by the multiplier one bit position at a time in the fashion that we normally do decimal multiplication using paper and pencil. The main difference
is that the process shown in Figure 1 sums up the rows as they are generated rather than after they have all been generated.

The \( p \)-register holds the partial products. Once a bit position result has been generated the lower bits no longer enter into the sum. Thus the partial product values are shift one bit right per iteration.

The \text{and} gates implement binary (0 or 1) multipliers.

The \text{add/subtract} control bit is set as needed when the most significant bit of \( b \) is multiplies \( a \). The most significant bit of a two's complement number contains the sign information, as well as a binary weight.

Product of two 16-bit values is formed in 16 clock cycles. The adder and \( p \)-register accumulate the rows one at a time. As a row is accumulated the lower bits of the sum no longer add in into the result so a shift register is used to hold the partial sum bits as they are generated.

We will not be working with this architecture in the Exercise. It is mentioned in order to bring it to your awareness and as a place holder for possible expansion of the exercise. Feel free to investigate further.
1.1.3 Serial-parallel multiplier

The multiplier configuration shown in Figure 2 is based on *On a Bit-Serial Input and Bit-Serial Output Multiplier*, R. Gnanasekaran, IEEE Transactions on Computers, Vol. C-32, No. 9, September 1983.

This configuration has the advantage that it multiplies two signed two's complement values without the need for any special initialization or add/sub control signals.

The longest delay is that of a one-bit adder. The maximum clock rate can be much higher than that usable with the circuit in Figure 1. On the down side, 32 clock cycles are needed to obtain the product of two 16-bit two's complement values.

1.1.4 Bit-serial multiply-and-add

Adding a bit-serial adder and a shift register to the multiplier in Figure 2 results in the multiply-and-accumulate (MAC) unit shown in Figure 3. The $a$ value is the multiplicand and the $b$ value is the multiplier. The unit does signed multiplication. Only four bits are shown, this exercise implements 8.

The logic for the bit-serial accumulator is on the right side of Figure 3. The output of the multiplier is bit-serial, least significant bit first. This is added bit serial wise to the shifted contents of the accumulator register. A single full adder with a delay in the carry is sufficient to do the addition. The extra delay in between the adder output and the accumulator is present for use when rounding (to be added). With this delay present it requires 17 clock ticks to add the multiplier output to the accumulator. This is for the circuit as shown.

![Diagram of Signed serial-parallel multiplier block diagram.](image-url)
Figure 3: Signed serial-parallel multiplier MAC entity block diagram (4-bit word size shown).

The adder and accumulator require minimal logic making minimal demands on use of FPGA fabric. The cost is higher execution time compared to a parallel implementation. However, if the time is available the use of bit-serial logic can allow use of smaller FPGA devices or increased computational ability for a given size FPGA as compared to parallel logic.

1.1.5 The MAC test entity

This section documents the MAC entity port signals and the use of the Spartan-3 board switches and display. A Digilent four slide switch Pmod module is expected to be attached to PMod port A.

**MAC entity port signals**

```vhdl
entity SP_MAC is
  Port ( a : in STD_LOGIC_VECTOR (7 downto 0);
        b : in STD_LOGIC_VECTOR (7 downto 0);
        go : in STD_LOGIC;
        clear : in STD_LOGIC;
        round : in STD_LOGIC;
        ready : out STD_LOGIC;
        ac : out STD_LOGIC_VECTOR (15 downto 0);
        reset : in STD_LOGIC := '0';
        clk : in STD_LOGIC);
  end SP_MAC;
```

Signal a is the multiplicand. b is the multiplier. go is a one clock period pulse which triggers the operation of the MAC. clear is a one period pulse which sets...
the MAC accumulator to zero, round is a level indicating whether or not the current MAC operation is to have its result rounded.

Signal ready is a one when the MAC entity is not working, a zero otherwise. There is no need for this application to check the ready. Operation is very fast and we, as observers, are, relatively speaking, glacially slow. The ac lines connect to the accumulator. The reset and clk are as normally.

1.2 Discarding bits and rounding

Many DSP calculations involve using 16-bit (or some other word size) multiplications and additions. Multiplications result in word size bit growth. Generally somewhere during or at the end of a series of operations it is necessary/desirable to store a 32-bit (or whatever) value back into a 16-bit word.

An example of the need to round in real life is when paying school and property taxes. These are typically figured in terms of mils, i.e., one thousandths of a dollar. Our coinage and banking system works in units that are a multiple of one-hundredths of a dollar, commonly referred to as a cent or penny.

Figuring interest payments is another real-life application where the rounding and discarding of digits is done.

In grade/high school we have been taught that when throwing digits away one should round first. We can round to the nearest cent, or the nearest dime, or the nearest quarter, etc. Lots of options. Whatever we and whoever we are dealing with are willing to agree to.

When computing there are times where it is decided that bits have more significance that others. Then we can discard the less significant ones and round based on what is discarded what’s left.

There are a quite of number of ways one can when rounding. The choice among them migh be conditioned on the statistical properties of the data values to be rounded. The two rounding methods that we will consider are referred to as two's complement and convergent rounding (aka rounding to the nearest even, ...).

Two's complement rounding is relatively easy to implement but does result in a small DC bias in the data. In many applications this bias is not important. Deciding whether this bias is important or not is the system designer's decision. Convergent rounding is more complicated than is two’s complement but generally results a unbiased result. There are DSP applications (e.g., when working
with CIC filters in high factor sample rate conversions) where convergent rounding is considered necessary.

First a quick review of rounding.

Consider an 8-bit value held in two 4-bit words, xxxxxxxx where the x are arbitrary bit values. It is desired to discard the low 4 bits. Before doing so we round the value by adding 00001000. then truncating by retaining only the top four bits of the result.

It is reasonable to ask that any two values that added to zero prior to rounding and truncation (or chopping) add to zero afterwards. This is claimed to be the case except for values of the form xxxx1000.

For example, 01101000 and 10011000 add to zero. Their two’s complemented truncated bit patterns are easily seen to be 0111 and 1010 which do not sum to zero.

A rule than can be used in this case that if the right most bit just left of the cut is a 0 then no value is added prior to discarding bits. If the right most bit just left of the cut is a 1 then then 00001000 is added. The effect is to round to the nearest remaining even value. This is also called convergent rounding.

Using the example from above we see the convergent rounded values are 0110 and 1010 which add to zero.

Given a bit pattern xxxx | xxxxxxxx where | indicates the cut point (where the bits to the right of the | are to be discarded) it can be established that we can do convergent rounding followed by cutting by adding 0000 | 1000000 and discarding the low 7 bits, unless we have a bit pattern of the form xxxx0 | x000000. In this situation, simply cut and discard. It should be clear how to generalize this procedure to any word size and cut position.

Convergent rounding can be thought of as being two’s complement rounding followed by truncation where there a single special case to be dealt with.

Figure 6 illustrates a logic that can be used to add convergent rounding to Figure 3.

The detection of the low bit all zero string could have been accomplished by anding high bits in the accumulator. The shown zero string detector works independent of the number of bits being checked. This allows easy later modification of where to round.

The round_now signal is a one only when the bit to be rounded is contained in the delay immediately following the left most adder.
1.3 Bit-serial finite impulse response filter

![Diagram](image)

Figure 5: Bit-serial FIR block diagram.

1.3.1 The bit-serial FIR entity

The filter coefficients, the input data and the output values are all 16-bit Q15. The intermediate sum is 31 bits. The low 15 bits are discarded (truncation).

Future work or a possible student exercise:

- Saturate the result for $-1 \times -1$ to be positive maximum.
- Rounding prior to discarding the low product bits.
1.3.2 The bit-serial FIR stage entity

Adds one stage of delay to the full adder and serial-parallel multiplier entities to create a single stage of FIR filter.

1.4 Test filter coefficient values

The test FIR is a four point (value) sliding average. The coefficient values are Q15 and equal to $1/4 = 0x2000$.

The transfer function is

$$H(f) = \frac{\sin(\pi Pf/f_s)}{P \sin(\pi f/f_s)}$$

where $P$ is the number of values averaged and $f_s$ is the sample rate. This is plotted in Figure 7 for values of $P = 2, 4, 8$.

This choice of coefficient values has been found to facilitate testing and verification of basic filter operation.

2 The Exercise

2.1 The MAC entity

You will be supplied with a non-rounding version of the MAC entity VHDL, see Appendix A.2 along with associated support files and test top (Appendix A.1). All modules should be operational.

Create a project using the supplied files and generate the bit file. You will need to generate the required UCF file.
Figure 7: Sliding average filter transfer function. Top: two values. Middle: four values. Bottom: eight values.
Load the bit file into the Spartan-3 board.

As might be expected successive MAC operations sum into the accumulator. Push button 3 can be used to clear the accumulator. Run a few numbers through in order to gain familiarity with the buttons and switches as well as to build confidence.

Eight bit values are entered via the 8 slide switches. Sixteen bit values are shown on the seven-segment display.

Push button function:

PB 0 do a MAC operation.
PB 1 load slide switch value into a-register.
PB 2 load slide switch value into b-register.
PB 3 clear the MAC accumulator.

PMod slide switch functions:

SW 1 show a-register in seven-segment display.
SW 2 show b-register in seven-segment display.
SW 3 round and truncate 16-bit MAC retaining the top 8-bits on PB0 operations. (After this functionality has been added.
SW 4 not used.

The contents of the MAC accumulator are shown in the seven-segment display when all switches are in the down position. The truncation operation might or might not (the more likely implementation) zero the low 8 bits of the MAC accumulator. Try it and see.

Run some test cases to verify multiplier and adder operation. It should work, it is supposed to. The operative word is “should”.

Test various sign value combinations (pos times pos, neg times pos, pos times neg, etc.) Do the largest values work. Do the smallest. Build your confidence that the units as advertised. If it doesn’t please tell and demonstrate to the GSI.

Include you test values and results in the report.
2.2 Implementing convergent rounding

Copy your MAC project into a different folder and use that as your starting point for adding the convergent rounding logic. Modify the MAC VHDL so that when the round line is high the value being shifted into the accumulator is convergent rounded. The rounding is to be accomplished by, in effect, adding (or not) a one in bit 7 of the accumulator. Because this a bit serial adder is being used the to be added needs to occur at the right time.

Convergent rounding was discussed both in lecture and earlier in this write-up. Basically one wants to place small amount of logic just prior to the MSB input of the accumulator shift register.

One possible logic design is shown in Figure 6.

Your GSI can give you some test cases to use for verifying correct operation. Note that the way the supplied logic works only the bits at the bit position to be rounded are modified. All of the bits to the right are unchanged.

Once you have your Spartan-3 correctly rounding, demonstrate to your GSI. Include your modified source code in you lab report. Also document the test cases that you used and explain why you choose as you did, what the expected results were and what your implementation produced.

2.3 Using more bits in the MAC

Read but not do.

We eventually expect to be using either a 12-bit or a 16-bit MAC entity in future lab exercises. Can you produce a generic MAC with rounding for arbitrary word sizes? Another feature that one might consider is to supply the index of the MAC bit where the rounding 1 is being added just to its right. In our case the rounding position was bit 8. The rounding one was added into bit position 7.

The input values are saved in registers. The round control like should be saved as well. Doing so would allow operation to proceed without requiring the input data lines being held fixed. Something to do on the next revision iteration.

The design included here does not have any overflow detection support nor the ability to saturate. The need for these abilities will be application dependent. Adding their support adds to the complexity of the logic. The more general a design, the more complex it is likely to be and the more difficult to understand and use.
Xilinx has a MAC logic core available. It is available to us via the Core Generator included in WebPACK. The documentation is available on Xilinx’s web site. Having done this exercise studying Xilinx’s MAC implementation should be informative.

### 2.4 Bit-serial FIR implementation

Using the following VHDL files:

- FIR_top.vhd,
- FIR_V01.vhd,
- pmod_ad1.vhd,
- pmod_da2.vhd,
- DCMxx.

Provide the needed VHDL in FIR_top to take samples at a 1 MHz sample rate, filter them with the FIR code and send the filtered values to the DA2 module. A very good starting point is the Exercise 2’s ad_da_01_top.vhd.

The A/D input should be from AD1 channel. DA2 channel 1 should be the filtered output. DA2 channel 2 should be the A/D samples. I used my level shifting amplifier to interface my signal generator to the A/D input. My generator does not have level shifting capability. The lab units do.

A reasonable sample rate to use is 10 kHz.

Test to see if the filter works as advertised. For example:

- Check for the existence of the transfer function zero at \( f = f_s/4 \).
- Use two scope channel of the scope to verify the approximate filter gain at a few frequencies.

Next in the top level implement a cascade of four FIR filters. The magnitude of the resulting transfer function should be the fourth power of on FIR filter. Use the signal generator to verify that the cascade works as expected.

Compare the maximum operating frequency determined by the ISE synthesizer for one FIR and for the cascade of four. Can this be used to get a rough idea of the per FIR section delay?

Compare the logic utilization for the single FIR and the cascade of four. Can this be used to get a rough idea of the per FIR section logic needs?

In lecture it was claimed that aliasing can be modeled by shifting the spectrum of signal multiples of \( f_s \) until the result lies in the range \([-f_s/2, f_s/2)\).
Check this out. Use a 10 kHz sample rate. Set the signal generator to 1500 Hz. Observe the output waveform on the oscilloscope as the generator frequency is increased by 10 kHz steps. Repeat for a sine wave at 2.5 kHz and multiples of 10 kHz.

What would you expect to observe for a square wave input at 1.5 kHz? At 2.5 kHz? This would be a good exercise for when/if we get a spectrum analyzer running in the Piccolo. Save it for then. Though one might take a stab at modeling in MATLAB and see if the waveform matches the model.
A  MAC test source code listings

A.1  MAC test entity source code listing

This entity uses the two process state machine approach.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity SP_MAC_test is
    Port ( btn : in STD_LOGIC_VECTOR (3 downto 0);
          -- led : out STD_LOGIC_VECTOR (7 downto 0);
          swt : in std_logic_vector(7 downto 0);
          pmod_a : in std_logic_vector(3 downto 0);
          ssg : out std_logic_vector(7 downto 0);
          an : inout std_logic_vector(3 downto 0);
          mclk : in STD_LOGIC);
end SP_MAC_test;

architecture Behavioral of SP_MAC_test is

signal a : std_logic_vector(7 downto 0);
signal b : std_logic_vector(7 downto 0);
```
signal ac : std_logic_vector(15 downto 0);
signal display : std_logic_vector(15 downto 0);
signal go : std_logic;

signal next_a : std_logic_vector(7 downto 0);
signal next_b : std_logic_vector(7 downto 0);
signal next_go : std_logic;
signal ready : std_logic;
signal reset : std_logic := '0';

signal pb_db : std_logic_vector(3 downto 0);
signal pb_clear : std_logic_vector(3 downto 0);
signal next_pb_clear : std_logic_vector(3 downto 0);

type t_state is (s_idle, s_go, s_done);
signal state : t_state := s_idle;
signal next_state : t_state := s_idle;
signal dp : std_logic_vector(3 downto 0);
signal sel : std_logic_vector(3 downto 0);
signal clk : std_logic;

begin

clk <= mclk;
dp <= "0000";
 sel <= "1111";

process (clk, reset) begin -- state machine updating
if reset = '1' then
elsif rising_edge(clk) then -- the actual updates
    pb_clear <= next_pb_clear;
    a <= next_a;
    b <= next_b;
    go <= next_go;
    state <= next_state;
end if;
end process;

process(state, pb_db) begin -- state machine sequencer
    next_pb_clear <= "0000";
    next_a <= a;
    next_b <= b;
    next_go <= go;
    next_state <= state;

    case state is
    when s_idle =>
        if pb_db(0) = '1' then -- pb0 causes p to be calculated
            next_pb_clear <= "0001";
        end if;
end case;
end process;
next_state <= s_go;
if pb_db(1) = '1' then -- pb1 loads a from the switches
next_pb_clear <= "0010";
next_a <= swt;
next_state <= s_idle;
elsif pb_db(2) = '1' then -- pb2 loads b from the switches
next_pb_clear <= "0100";
next_b <= swt;
next_state <= s_idle;
elsif pb_db(3) = '1' then
next_pb_clear <= "1000";
end if;
when s_go => -- state to initiate multiply
next_go <= '1';
next_state <= s_done;
when s_done => -- doesn't bother to wait for ready
next_go <= '0';
next_state <= s_idle;
end case;
end process;

multiplier: entity work.SP_MAC -- preliminary version -- signed
port map (
  a => a,
  b => b,
  ac => ac,
  go => go, -- pulse to start
  clear => pb_clear(3),
  ready => ready, -- ready also means done
  reset => reset,
  clk => clk
);

push_buttons: entity work.pb_debounce -- push button debouncer
port map (
  pb_in => btn, -- input actual push buttons
  pb_out => pb_db, -- debounced push button
  pb_clear => pb_clear, -- clears the button state
  reset => '0',
  clk => clk
);

display <= ("00000000" & a) when pmod_a = "0001" else
  ("00000000" & b) when pmod_a = "0010" else
  ac;

SSD02_unit: entity work.SSD02 -- Chih-Wei's seven-segment support
port map (
  ssd0 => display(3 downto 0),
  ssd1 => display(7 downto 4),
  ssd2 => display(11 downto 8),

ssd3 => display(15 downto 12),
ssd => ssg,
dp => dp,
se1 => sel,
an => an,
clk => clk);
end Behavioral;

A.2 The MAC entity VHDL

This unit uses the one-process state machine approach. Required combinational
logic can be added in the location where the line

\[
\text{sum\_rounded} \leftarrow \text{sum\_delayed};
\]

is located. The two signals used here bridge the place where the rounding logic is
to go. The signal input to the rounding is \text{sum\_delayed} and the signal generated
by the rounding logic is \text{sum\_rounded}.

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
entity SP_MAC is
  Port ( a : in STD_LOGIC_VECTOR (7 downto 0);
        b : in STD_LOGIC_VECTOR (7 downto 0);
        go : in STD_LOGIC;
        clear : in STD_LOGIC;
        round : in STD_LOGIC;
        ready : out STD_LOGIC;
        ac : out STD_LOGIC_VECTOR (15 downto 0);
        reset : in STD_LOGIC := '0';
        clk : in STD_LOGIC);
end SP_MAC;

architecture Behavioral of SP_MAC is

signal acc : std_logic_vector(15 downto 0);
signal ar : std_logic_vector(7 downto 0);
signal br : std_logic_vector(7 downto 0);
signal pbit : std_logic;
signal sum_carry_delay : std_logic := '0';
signal sum_carry_out : std_logic;
signal sum_out : std_logic;
signal sum_delayed : std_logic;
signal sum_rounded : std_logic;
signal counter : std_logic_vector(4 downto 0);
signal spm_go : std_logic := '0';
signal spm_reset : std_logic := '0';
signal my_ready : std_logic := '1';

type t_state is (idle, start, running, done);
signal state : t_state := idle;

begin

  ac <= acc;
  ready <= my_ready;

  spmult : entity work.SP_Mult
    port map (a=>ar(0), b=>br, p=>pbit, go=>spm_go,
             reset=>spm_reset, clk=>clk);

  psum : entity work.FullAdder01
    port map (a=>acc(0), b=>pbit, sum=>sum_out, cin=>sum_carry_delay, cout=>sum_carry_out,
              sum_rounded <= sum_delayed; -- remove when adding rounding support
  controller : process(go, reset, clk)
  begin
    if reset = '1' then
      my_ready <= '1';
  end process controller;
acc<=(others=>'0');
spm_go='0';
state<='idle';
elif rising_edge(clk) then
  case state is
    when idle =>
      if clear='1' then
        acc<=(others=>'0');
      end if;
      if go='1' then
        my_ready<='0';
        spm_reset<='1';
ar<=a; ar<=ar(7) & ar(7 downto 1); -- sign extend right shift
        br<b; br<=br(7) & br(7 downto 1);
        sum_delayed<='0'; sum_delayed<=sum_out;
        sum_carry_delay<='0'; sum_carry_delay<=sum_carry_out;
        counter<="00000";
        state<='start';
      end if;
    when start =>
      spm_reset<='0';
      spm_go='1';
      state<='running';
    when running =>
      sum_carry_delay<=sum_carry_out; sum_carry_delay<=sum_carry_delay;
      sum_delayed<=sum_out; sum_delayed<=sum_delayed;
      acc<=sum_rounded & acc(15 downto 1); acc<=acc;
      ar<=ar(7) & ar(7 downto 1); ar<=ar;
      counter<=counter+1; counter<=counter;
      if counter=16 then
        spm_go<='0';
        state<='done';
      end if;
    when done =>
      my_ready<='1';
      state<='idle';
  end case;
end if;
end process;
end Behavioral;
B  FIR_test_V01 listings

B.1  FIR_test_V01.vhd

Consider using Exercise 2’s ad_da_01_top.vhd as your starting point. Rename it and the associated UCF file as FIR_test_V01.

B.2  SerialFIR

----------------------------------
-- Company:  Doing DSP Workshop Summer 2009
-- Engineer:  Kurt Metzger
--
-- Create Date:  15:27:32 05/22/2009
-- Design Name:
-- Module Name:  SerialFIR - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- very simple FIR filter
-- presently truncates
-- no overflow checks
-- no saturation

entity SerialFIR is
  Port ( xin : in  STD_LOGIC_VECTOR (15 downto 0);
        yout : out STD_LOGIC_VECTOR (15 downto 0);
        req_in : in  STD_LOGIC;
        ack_out : out STD_LOGIC;
        clk : in  STD_LOGIC;
        reset : in  STD_LOGIC);
end SerialFIR;

architecture Behavioral of SerialFIR is

signal d_in : std_logic_vector(15 downto 0);
signal d_out : std_logic_vector(15 downto 0);
signal out0, out1, out2 : std_logic;
signal req_in_old : std_logic := '0';
signal req : std_logic_vector (1 downto 0);
signal ps0, ps1, ps2 : std_logic;  -- partial sums
signal multiply_en, shift_en : std_logic;
signal end_sum : std_logic;
signal fir_reset : std_logic := '0';
signal ctr : std_logic_vector(5 downto 0) := (others => '0');  -- 6 bit counter
type t_state is (idle, start, multa, multb, finish);
signal state : t_state := idle;

begin

  req <= req_in_old & req_in;

  process(req_in, clk, reset)
  begin
    if reset = '1' then
      ack_out <= '0';
      shift_en <= '0';
      multiply_en <= '0';
      state <= idle;
    elsif rising_edge(clk) then
      req_in_old <= req_in;
      if req = "10" then
        ack_out <= '0';
      end if;

      case state is
      when idle =>
        if req_in = '1' then
          shift_en <= '0';
          multiply_en <= '0';
          fir_reset <= '1';
          d_in <= xin;
          ack_out <= '1';
          ctr <= (others => '0');
          state <= start;
        end if;
      when start =>
        fir_reset <= '0';
        shift_en <= '1';
        multiply_en <= '1';
        state <= multa;
      when multa =>
        ctr <= ctr+1;
        d_in <= d_in(15) & d_in(15 downto 1);
        d_out <= end_sum & d_out(15 downto 1);
if ctr = 14 then
    shift_en <= '0';
    state <= multb;
end if;
when multb =>
    ctr <= ctr+1;
    d_out <= end_sum & d_out(15 downto 1);
if ctr = 30 then
    shift_en <= '1';
    multiply_en <= '0';
    state <= finish;
end if;
when finish =>
    shift_en <= '0';
yout <= d_out;
    state <= idle;
end case;
end if;
end process;

stage0 : entity work.FIRstage
  port map ( serial_in => d_in(0),
              serial_out => out0,
              sum_in => '0',
              sum_out => ps0,
              coefficient => X"2000",
              shift_en => shift_en,
              multiply_en => multiply_en,
              reset => fir_reset,
              clk => clk);

stage1 : entity work.FIRstage
  port map ( serial_in => out0,
              serial_out => out1,
              sum_in => ps0,
              sum_out => ps1,
              coefficient => X"2000",
              shift_en => shift_en,
              multiply_en => multiply_en,
              reset => fir_reset,
              clk => clk);

stage2 : entity work.FIRstage
  port map ( serial_in => out1,
              serial_out => out2,
              sum_in => ps1,
              sum_out => ps2,
coefficient => X"2000",
shift_en => shift_en,
multiply_en => multiply_en,
reset => fir_reset,
clk => clk);

stage3 : entity work.FIRstage
port map (
  serial_in => out2,
  --serial_out => out3,
  sum_in => ps2,
  sum_out => end_sum,
  coefficient => X"2000",
  shift_en => shift_en,
  multiply_en => multiply_en,
  reset => fir_reset,
  clk => clk);

end Behavioral;

B.3 FIRstage.vhd

-- Company: Doing DSP Workshop Summer 2009
-- Engineer: Kurt Metzger
-- Create Date: 15:17:19 05/22/2009
-- Design Name:
-- Module Name: FIRstage - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
-- Dependencies:
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;
entity FIRstage is
  Port ( serial_in : in STD_LOGIC;
         serial_out : out STD_LOGIC;
         sum_in : in STD_LOGIC;
         sum_out : out STD_LOGIC;
         coefficient : in STD_LOGIC_VECTOR (15 downto 0);
         shift_en : in STD_LOGIC;
         multiply_en : in STD_LOGIC;
         reset : in STD_LOGIC;
         clk : in STD_LOGIC);
end FIRstage;

architecture Behavioral of FIRstage is

  signal product_out, cin, cout, ser_out : std_logic;

begin

  serial_out <= ser_out;

  update : process(clk, multiply_en)
  begin
    if reset = '1' then
      cin <= '0';
    else if rising_edge(clk) then
      if multiply_en = '1' then cin <= cout; end if;
    end if;
  end if;
end process;

  shift : SRL16E generic map(init => X"0000")
    port map (d=>serial_in, a0=>'1', a1=>'1', a2=>'1', a3=>'1',
               clk=>clk, ce=>shift_en, q=>ser_out);

  s_mult : entity work.SP_Mult
    generic map (N=>16)
    port map (a=>ser_out, b=>coefficient, p=>product_out, go=>multiply_en,
              reset=>reset, clk=>clk);

  sum : entity work.FullAdder01
    port map (a=>sum_in, b=>product_out, cin=>cin, sum=>sum_out, cout=>cout);

end Behavioral;

C  The serial-parallel multiplier entity VHDL listing

This module can be customized to the required word size by specifying the value of N in the instantiation call. The default size is 8 bits. Generate statements were
used to generate the necessary VHDL statements. The generic keyword was used to make the module parameterizable.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity SP_Mult is
  Generic ( N : integer := 8);
  Port ( a : in STD_LOGIC;
          b : in STD_LOGIC_VECTOR (N-1 downto 0);
          p : out STD_LOGIC;
          go : in STD_LOGIC;
          reset : in STD_LOGIC;
          clk : in STD_LOGIC);
end SP_Mult;

architecture Behavioral of SP_Mult is

signal asr : std_logic_vector(N-1 downto 0);
signal delayin : std_logic_vector(N-1 downto 1);
signal delayout : std_logic_vector(N-1 downto 1);
signal carryin : std_logic_vector(N-1 downto 0);
signal carryout : std_logic_vector(N-1 downto 0);
signal ab : std_logic_vector(N-1 downto 0);
```

```
begin

ab <= b when a = '1' else (others=>'0');

sp_mul: for i in 0 to N-1 generate
begin

  right : if i = 0 generate
  sr : entity work.FullAdder01
  port map (a=>delayout(i+1), b=>ab(i),
    cin=>carryout(i), cout=>carryin(i), sum=>p);
  end generate right;

  mid : if (0 < i) and (i < N-1) generate
  sr : entity work.FullAdder01
  port map (a=>delayout(i+1), b=>ab(i),
    cin=>carryout(i), cout=>carryin(i), sum=>delayin(i));
  end generate mid;

  left : if i = N-1 generate
  sr : entity work.FullAdder01
  port map (a=>delayout(i), b=>ab(i),
    cin=>carryout(i), cout=>carryin(i), sum=>delayin(i));
  end generate left;

end generate sp_mul;

process(clk, go)
begin
if reset = '1' then
  delayout <= (others=>'0');
  carryout <= (others=>'0');
elsif rising_edge(clk) and go = '1' then
  delayout <= delayin;
  carryout <= carryin;
end if;
end process;

end Behavioral;
```

D One-bit full adder entity VHDL listing

It was found easiest to work with the full adder directly and not use an entity specifically written for a one-bit serial adder (full adder with a register between the carry out and the carry in).
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FullAdder01 is
  Port ( a : in STD_LOGIC;
         b : in STD_LOGIC;
         cin : in STD_LOGIC;
         sum : out STD_LOGIC;
         cout : out STD_LOGIC);
end FullAdder01;

architecture Behavioral of FullAdder01 is
begin

  sum <= a xor b xor cin;
  cout <= (a and b) or (cin and a) or (cin and b);

end Behavioral;