Fall 2019

California State University, Northridge

Department of Electrical & Computer Engineering



Lab Experiment 3

*FPGA Features Part I:*

*Clock Management, DSP Blocks, DDR, & SRL*

Juan Silva & Tumen Khureltulga

(Group 16)

October 8, 2019

ECE 524L

**Table of Contents**

**I. Introduction and Problem Statements…….………………………...…Page 2**

**II. Procedure……..…………………………………………….…………...Page 2**

**III. Testing Strategy………………………………….....………………......Page 5**

**IV. Results…………....…………………………………....…………….......Page 6**

**V. Analysis…………....……………………………....……………….……Page 11**

**VI. Appendix……………...………………………....………………….….Page 14**

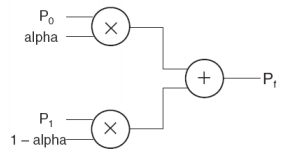
1. **Introduction and Problem Statements**

In this lab experiment, a stream interleaver is designed and several configurations are used to implement this function. Two streams of data inputs are received, multiplexed, and will output a combined stream running at twice the rate. This project can be designed using a direct implementation, an optimized implementation using Double Data Rate (DDR) with a clock manager, and a synchronized design using SRLs or BRAMs for delays. Although a direct implementation sufficient, the purpose of this experiment is to use several FPGA features to optimize the design. Applications of this function include fading between two images in image processing projects. Each implementation is designed, tested, and analyzed taking synthesizable hardware into account.

1. **Procedure**

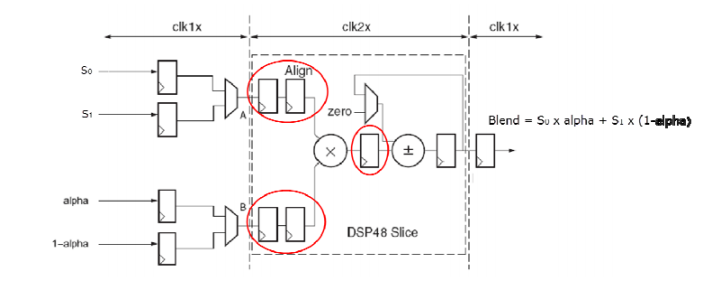
A signal stream blender block diagram in *Figure 3.1* and its mathematical representation in *Equation 3.1* are provided to implement the steam interleaver. Two stream inputs P0 & P1 are 8-bit inputs and alpha is a constant that determines which signal will be read. Based on the top level block diagram, each input is multiplied and the output results are summed together. This design will synthesize 3 DSP blocks: two for each multiplier and one for the adder. However, Vivado does not initially map each component to DSP blocks. Therefore, we force the Vivado tools to enable DSP block mapping. The results are shown in *Figure 3.10*.

**Equation 3.1 -** Mathematical representation of input frequency.

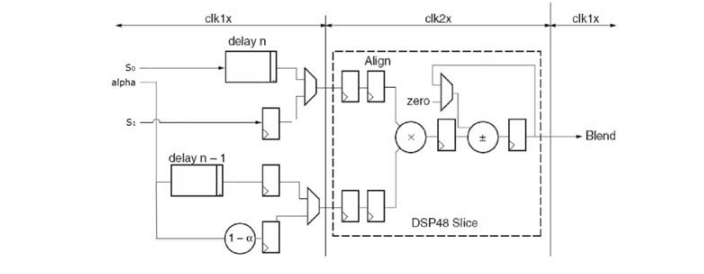


**Figure 3.1 -** High Level Block Diagram of Signal Stream Blender.

The direct implementation that uses DSP blocks can be optimized using a DDR approach. The arithmetic block can be split using varying clock signals where the MAC is done at half the clock speed. This is achieved by using a Digital Clock Manager (DCM) that runs a phase-aligned double-frequency clock. Therefore, one stream of data is sampled at 50 Mhz while the other is sampled at 100 Mhz. This top level diagram is shown on *Figure 3.3* By the time the next batch of data streams are sampled, the multiplier has completed two multiplication operations. The clock with a 50 Mhz frequency, Clk1x, controls the data to be sampled and when is sent out. The clock with a 100 Mhz frequency, Clk2x, serves as a control signal for MUX A and B. When it’s HIGH, inputs S\_0 and alpha are selected. When it’s LOW, the data S\_1 and 1-alpha are selected. Two registers are connected to align the data inputs before they’re multiplied. The product of the selected inputs are stored and a feedback is applied back to a MUX in the Clk2x process. Using Clk2x as a signal, it’ll pass the sum of the product when high or just the product when low. Data is sent out when clk1x is HIGH.

**Figure 3.3** - Optimized Implementation Block Diagram

In practical applications, the input signals are not synchronized so a delay must be applied to align each signal. The 64-bit delay can be designed using an SRL or with BRAM. The concept is to write to the memory and read after the desired time which can simulate a delay.  
 Each SRL component has eight 64 bit registers since the data is required after 64 clock cycles. SRL component takes in 8-bit data. Each bit is fed to eight registers bit by bit. For example, the first 64-bit register takes LSB from the input data, the second 64-bit register takes the next LSB bit. This cycle continues till MSB is fed to the last 64-bit register. The data is out after 64 cycles. The bits are put back together to form 8-bit data by concatenating them.  
 FPGA SLICE M is capable of creating a small-sized 64X1 RAM. This is small compared to BRAM. Two DRAM is required as well. Each DRAM has 64 arrays of 8-bit memory. The data is written to an address and read after 64 cycles. There are two ways to create a memory. The first way is to use the IP Catalog in Vivado. The second way is to write the VHDL code. A VHDL code is chosen because it is portable between designs.



**Figure 3.4** - Equalizing Delays Block Diagram

1. **Testing Strategy**

It is known from the equation that the output depends on the value of alpha and two signals have the same frequency of 8 MHz but varies in magnitude. The magnitude relationship between the two signals is = when two signals are in phase with each other. If alpha value is 0, the output should follow signal . If the alpha value is 1, the output should follow signal ,The table below, Table3.1, provides the expected values when alpha takes on different values. The experimental output value is compared with expected value for verification purposes.

| Alpha Value | Expected Output |
| --- | --- |
| 0.0 |  |
| 0.25 | 0.25+0.75=1.25 |
| 0.50 | 0.50+0.50=1.5 |
| 0.75 | 0.75+0.25=1.75 |
| 1.00 | =2 |

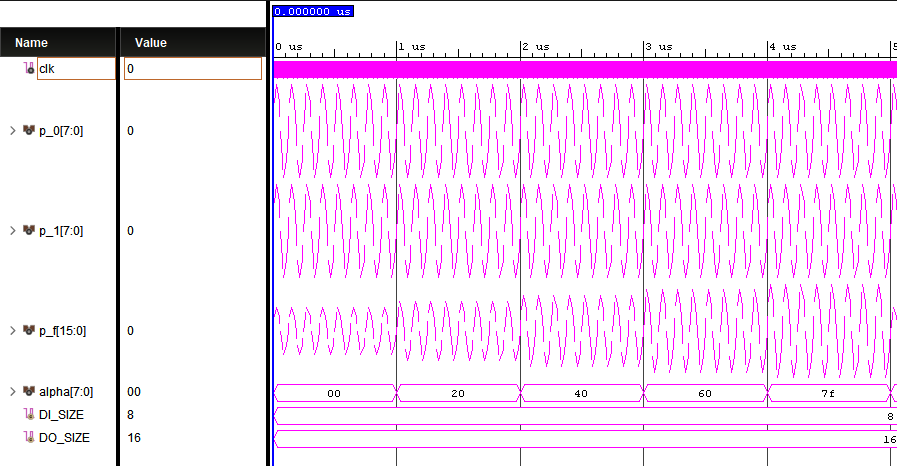
**Table 3.1** : Expected Output

Furthermore, the alpha value is varied from 0 to 1 during testing to further verify the designs. This was sufficient enough to conclude that the design is operational.

In part 3, there are delays. Thus, above Table 3.1 does not work. To verify this design, data are sampled and plugged in to Equation 1.1 and the result is compared with the sampled output. If the sampled output and expected values match, it is enough to say verification is complete.

1. **Results & Data**
2. Part 1:Direct Implementation

The following Figure1.1 shows the inputs, p\_0,p\_1, and alpha, and output p\_f. Alpha is type constant and its values changed every 1 us. Two signals are in phase with each other. They both start from a magnitude of zero. Alpha value takes on real values. Alpha had to be quantized to digital form. In other words, the alpha value is multiplied by 127 to form signed 8 bits.

****

**Figure 3.5** -Behavioral Simulation Waveform of Direct Implementation  
  
**Table 3.2** Alpha=0.0 and 0.25

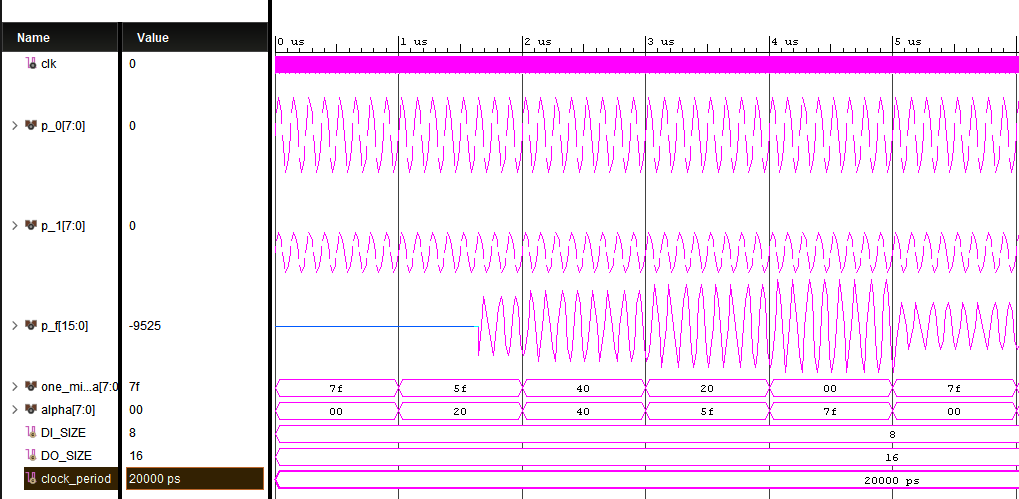
| Alpha=0.0 | Output Value, p\_f | Expected Output | Error % | Alpha=0.25 | Output Value,p\_f | Expected Output | Error % |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 4699/127=75 | 37\*1.25= | 0 | 1 | 9572/127= | 60\*1.25=75 | 0 |
| 2 | 7620/127=60 | 60 | 0 | 2 | 5915/127=46 | 37\*1.25=46 | 0 |
| 3 | 4699 | 60 | 0 | 3 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 4 | -5915/127 | 46 | 0 |
| 5 | -4699/127=-37 | -37 | 0 | 5 | -9572/127 | 75 | 0 |

**Table 3.3** Alpha=0.5 and 0.75

| Alpha=0.5 | Output Value,p\_f | Expected Output | Error % | Alpha=0.75 | Output Value, p\_f | Expected Output | Error % |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 11524/127=90 | 60\*1.5=90 | 0 | 1 | 13476/127=106 | 60\*1.75=106 | 0 |
| 2 | 7131/127=56 | 37\*1.5=56 | 0 | 2 | 8347/127=65 | 37\*1.75=65 | 0 |
| 3 | 0 | 60 | 0 | 3 | 0 | 0 | 0 |
| 4 | -7131 | -56 | 0 | 4 | -5915/127 | 46 | 0 |
| 5 | -11524/127 | --90 | 0 | 5 | -9572/127 | 75 | 0 |

Table 4.1 and 4.2 contains 5 sample outputs at different alpha values and it is compared with expected values. A case when alpha=1.0 is skipped because at this point it is sufficient enough to conclude that the design operates as designed.

1. Part 2: Optimized Implementation

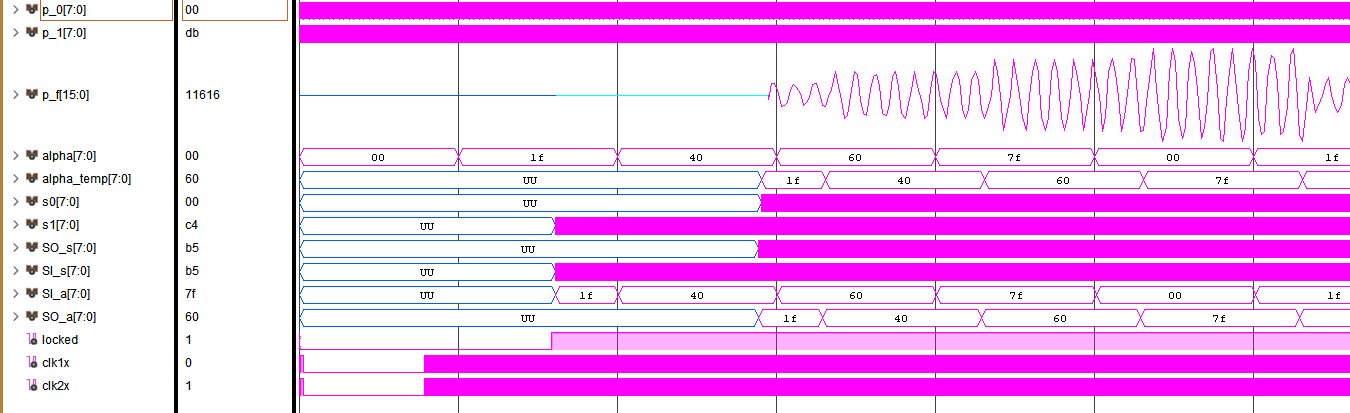
The following Figure 4.2 shows the simulated waveforms. The output is delayed because this is design used Digital Clock Manager to generate 50 and 100 MHz clock frequency. The DCM takes in 50 Mhz 

**Figure 3.6** - Behavioral Simulation Waveform of Optimized Implementation

clock frequency from an outside source such as an onboard oscillator, in this case from the test bench. It is capable of generating various clock frequencies. When the output clocks from DCM are stable, the module sends out an active high “locked” signal. This locked signal indicates that the output clocks are stable and ready to be used. This allows the design to be synchronous.

1. Part 3:Equalizing Delays
   1. SRL Component Implementation

Figure 5.3 shows the waveform of SRL based implementation of the design. The signal p\_1 and p\_2 are out of phase with each other. The signal p\_1 starts 12.5 ns before p\_0. At time=0, the initial values of p\_0 and p\_1 signals are 0 and -37, respectively



**Figure 3.7** - Behavioral Simulation Waveform of SRL Based Implementation

The design starts when locked is HIGH at time 1.501 us. The signal s0 goes to input port SI\_s of SRL and comes out from SO\_s after 64 clk1x clock cycles later. The input port SI\_s senses the incoming data at 1.61 us and SO\_s outputs the first data at 2.89 us. The difference is 1.28 us.

Alpha is delayed by 64 clock cycles as well. This means 1-alpha and alpha won’t be relational anymore. For example, by the time alpha is read. Let’s say alpha value read is 0.25. However, 1-alpha won’t be 0.75 because 1-alpha is not delayed. In Figure 5.4, at 3 us, alpha\_temp value from SRL is 31, 0.25 in real numbers. Notice that one\_minus\_alpha is 0.25 for a short time, not 0.75, then it is changed to 0.5.

Signals s0 and s1 are the inputs to multiplier. In other words, the output p\_f calculated from s0 and s1.It is shown that s0 gets available after a while compared to s1 signal.

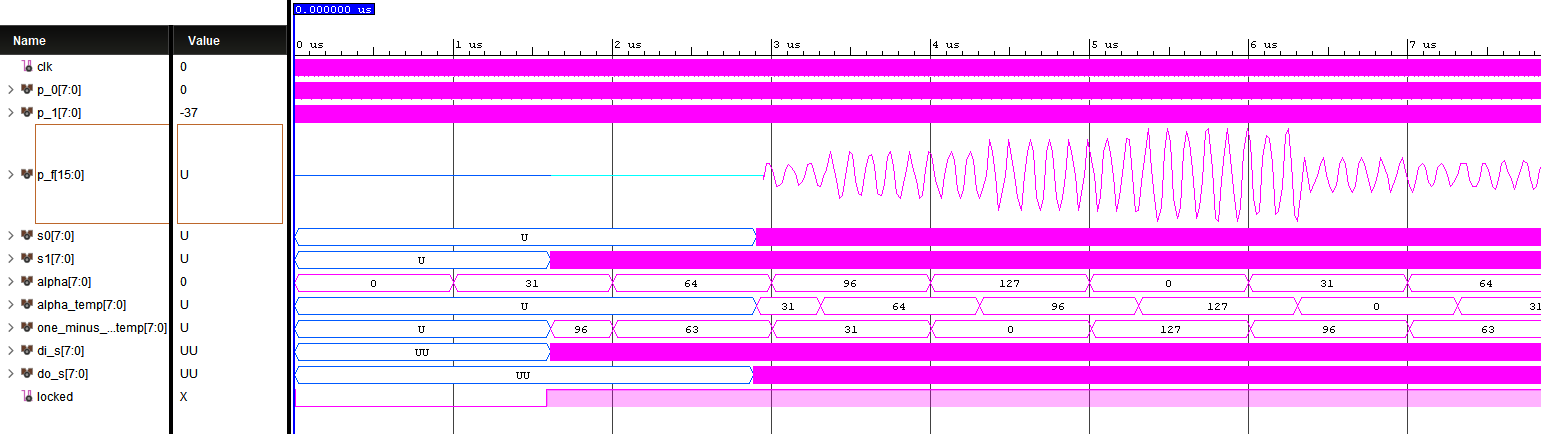
Since there are delays, the output p\_f values are different from the previous two designs. The magnitude relationship no longer holds true.Thus, alpha, 1-alpha, s0, and s1 are sampled and plugged into equation 1.1 to verify. Table 4.1 contains 10 sampled data. It is verified that the design function accordingly to equation 1.1. Automated testing procedure is recommended since one could only plug a few samples to the equation.

**Table 3.4** - Sampled Data

| sample | s0 | s1 | alpha\_temp | one\_minus\_alpha | Output, p\_f | Expected Output | %Error |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | -121 | 37 | 31=0.25 | 63=0.5 | -1420/127=-11 | 0.25(-121)+0.5(37)=-11 | 0 |
| 2 | 0 | 60 | 31 | 63 | 3780/127=30 | 0.5(60)=30 | 0 |
| 3 | 121 | 0 | 31 | 63 | 3751/127=29 | 0.25\*121=29 | 0 |
| 4 | 0 | -60 | 31 | 63 | U | U |  |
| 5 | -75 | -37 | 31 | 31 | -1420 | 0.25(-75)+0.25(-37)=-11 | 0 |
| 6 | -121 | 37 | 31 | 31 | 2604/127 | 0.25(-121)+0.25(37))=21 | 0 |
| 7 | 0 | 60 | 31 | 31 | 1860/127=15 | 0.25\*60=15 | 0 |
| 8 | 75 | 37 | 31 | 31 | 3472/127=27 | 0.25\*75+0.25\*37=27 | 0 |
| 9 | 121 | -37 | 31 | 31 | 2604/127=21 | 0.25\*121+0.25\*(-37)=21 | 0 |
| 10 | 75 | -60 | 31 | 31 | 465/127=3 | 0.25\*75+0.25(-60)=3 | 0 |

* 1. DRAM Implementation

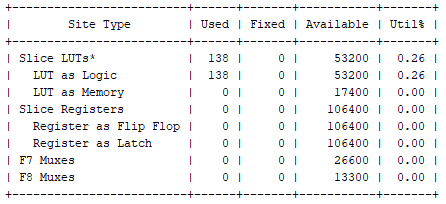
This design works the same as SRL implementation except delays are implemented using DRAM.



**Figure 3.8** - Behavioral Simulation Waveform of DRAM Based Implementation

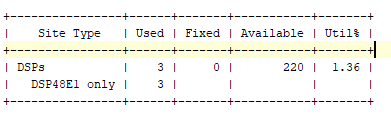
1. **Analysis**
2. Part 1:Direct Implementation

The implementation works perfectly. Table.4.1 shows that its output matched the expected value with a zero percent error. Its operation exactly follows the equation 1.1. Although it is the most direct implementation, it is the most carefree, in terms of hardware and resources. This design does not care whether it used 3 adders or one adder. Figure 5.1 shows that the direct implementation used 138 LUTs from SLICE L. The synthesis tool uses fabrics to do arithmetic by default.

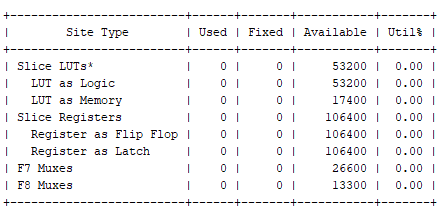


**Figure 3.9 -** Slice Logic Utilization Report before DSP forced.

After forcing the tool to use DSP for the whole entity, the design ended up using three DSP blocks, one for adder and two for multipliers.



**Figure 3.10** Utilization Report after DSP forced

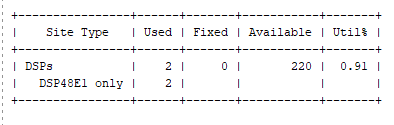


**Figure 3.11**  Slice LogicUlitization Report after DSP forced

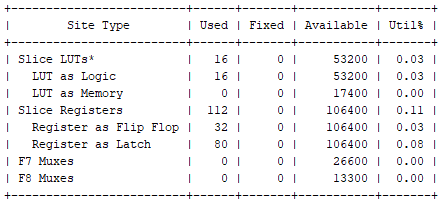
After DSP is forced, all the previously used LUTs are no longer used.

1. Part 2: Optimized Implementation

This design introduced to reduce the number of DSP used in the Direct Implementation method. After synthesizing the design, the tool generates following report. Figure 5.3 clearly shows that this design uses one less DSP.



**Figure 3.12** Utilization Report of DSP.

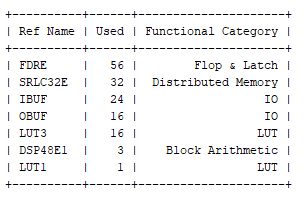


**Figure 3.13** Slice Logic Utilization Report

Figure 5.4 shows how many LUTs, Flip Flops, and Latches are used from the fabric. This design used registers, unlike the previous design.

1. Part 3:Equalizing Delays
   1. SRL Component Implementation

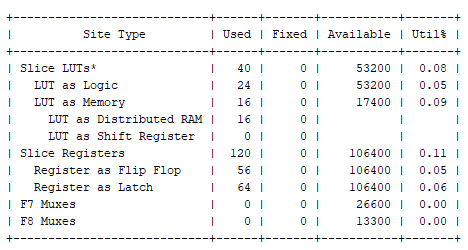
The latency of this design is 1360 ns. The design starts when locked gets HIGH, from this point to first output is the latency. It uses 3 DSP blocks because 1-alpha has to be calculated from alpha input unlike the Optimized Implementation where 1-alpha is input signal. From Figure 5.4, the design used 32 SRL components. One SRL component is 32 bit long. This design needs 8 rows of 64 bits. It takes 2 SRL to implement 64 bits. This has two delays. Each delay use 16 SRLs.



**Figure 3.14** Primitive Report

* 1. DRAM implementation

This design uses 5 DSP blocks. One DSP is added to generate 1-alpha signal and two to increment the address line. It takes 4 clock cycles of clk2x and 2 cycle of clk1x to generate output once data are read from DRAM. Total 16 DRAMs are used since each delay required 8 DRAM. 64 Flip-Flops are used. 32 FFs are used to sample the data and 32 FFs are used to send data out to output.



1. **Appendix**

**Source Codes**

| Direct Implementation | Optimized Implementation | DRAM Implementation | SRL Implementation |
| --- | --- | --- | --- |
| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use ieee.numeric\_std.all;  use ieee.std\_logic\_unsigned.all;  entity part\_1 is  generic(SAM\_SIZE:integer:=8;  OUT\_SIZE:integer:=16;  FAC\_SIZE:integer:=8);  Port ( p\_0, p\_1:in std\_logic\_vector(SAM\_SIZE-1 downto 0) ;  alpha:in std\_logic\_vector(FAC\_SIZE-1 downto 0);  p\_f:out std\_logic\_vector(OUT\_SIZE-1 downto 0)    );  attribute use\_dsp48:string;  attribute use\_dsp48 of part\_1:entity is "yes";  end part\_1;  architecture Behavioral of part\_1 is  signal temp1, temp2 : signed(15 downto 0);  begin  temp1 <= signed(p\_0) \* signed(alpha);  temp2 <= signed(p\_1) \* signed(127-alpha);  p\_f <= std\_logic\_vector( temp1 + temp2);  end Behavioral; | library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use ieee.numeric\_std.all;  entity part\_2 is  generic(SAM\_SIZE:integer:=8;  OUT\_SIZE:integer:=16;  FAC\_SIZE:integer:=8);  Port ( p\_0, p\_1:in std\_logic\_vector(SAM\_SIZE-1 downto 0) ;  alpha:in std\_logic\_vector(FAC\_SIZE-1 downto 0);  one\_minus\_alpha:in std\_logic\_vector(FAC\_SIZE-1 downto 0);  p\_f:out std\_logic\_vector(OUT\_SIZE-1 downto 0);  clk:in std\_logic  );    attribute use\_dsp48:string;  attribute use\_dsp48 of part\_2:entity is "yes";  end part\_2;  architecture Behavioral of part\_2 is  --CLK WIZARD  component clk\_wiz\_0  port  (-- Clock in ports  -- Clock out ports  clk\_out1 : out std\_logic;  clk\_out2 :out std\_logic;  -- Status and control signals  locked : out std\_logic;  clk\_in1 : in std\_logic  );    end component;  signal s0, s1 :std\_logic\_vector(SAM\_SIZE-1 downto 0);  signal locked,clk1x,clk2x:std\_logic;  signal temp2, temp3, temp4:signed(OUT\_SIZE-1 downto 0);  signal temp0, temp1:signed(SAM\_SIZE-1 downto 0);  signal var0, var1,var2:signed(SAM\_SIZE-1 downto 0);  signal var3,var4,var5:signed(OUT\_SIZE-1 downto 0);  begin  u1:clk\_wiz\_0 port map(clk\_out1=>clk1x, clk\_out2=>clk2x, locked=>locked, clk\_in1=>clk);  data\_in:process(clk1x)  begin  if ( locked='1') then  if rising\_edge(clk1x) then  s0 <=p\_0;  s1<=p\_1;  end if;  end if;  end process;  multiplication:process(clk2x)  begin  if locked='1' then  case clk2x is  when '1'=>  var0<=signed(s0);  temp0<=var0;  var1<=signed(alpha);  temp1<=var1;  var3<=temp0\*temp1;  temp2<=var3;  when '0' =>  var0<=signed(s1);  temp0<=var0;  var1<=signed(one\_minus\_alpha);  temp1<=var1;  var3<=temp0\*temp1;  temp2<=var3;  when others => null;  end case;  end if;  end process;  add:process(clk2x)  begin  if locked='1' then  case clk2x is  when '1' =>  temp4<=temp2+0;  when '0' =>  temp4<=temp4+temp2;  when others => null;  end case;  end if;  end process;  data\_out:process(clk1x)  begin  if locked='1' then  if rising\_edge(clk1x) then  p\_f<=std\_logic\_vector(temp4);  end if;  end if;  end process;  end Behavioral; | library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use ieee.numeric\_std.all;  entity part\_3 is  -- Port ( );  generic(SAM\_SIZE:integer:=8;  OUT\_SIZE:integer:=16;  FAC\_SIZE:integer:=8);  Port ( p\_0, p\_1:in std\_logic\_vector(SAM\_SIZE-1 downto 0) ;  alpha:in std\_logic\_vector(FAC\_SIZE-1 downto 0);  p\_f:out std\_logic\_vector(OUT\_SIZE-1 downto 0);  clk:in std\_logic  );  attribute use\_dsp48:string;  attribute use\_dsp48 of part\_3:entity is "yes";  end part\_3;  architecture Behavioral of part\_3 is  --CLK WIZARD  component clk\_wiz\_0  port  (-- Clock in ports  -- Clock out ports  clk\_out1 : out std\_logic;  clk\_out2 :out std\_logic;  -- Status and control signals  locked : out std\_logic;  clk\_in1 : in std\_logic  );  end component;    component rams\_dist  port(  clk : in std\_logic;  we : in std\_logic;  a : in std\_logic\_vector(5 downto 0);  di : in std\_logic\_vector(7 downto 0);  do : out std\_logic\_vector(7 downto 0)  );  end component;    signal alpha\_temp,one\_minus\_alpha\_temp,s0, s1,di\_s,di\_a, do\_s,d\_a,do\_a :std\_logic\_vector(SAM\_SIZE-1 downto 0);  signal locked,clk1x,clk2x:std\_logic;  signal temp2, temp3, temp4:signed(OUT\_SIZE-1 downto 0);  signal temp0, temp1:signed(SAM\_SIZE-1 downto 0);  signal a\_s,a\_a:STD\_LOGIC\_VECTOR(5 DOWNTO 0):=(others=>'0');  signal var0,var1,var2,var6,var7:signed(SAM\_SIZE-1 downto 0);  signal var3,var4,var5:signed(OUT\_SIZE-1 downto 0);    begin  u1:clk\_wiz\_0 port map(clk\_out1=>clk1x, clk\_out2=>clk2x, locked=>locked, clk\_in1=>clk);  u2:rams\_dist port map(clk=>clk1x,a=>a\_s, di=>di\_s,do=>do\_s,we=>locked);  u3:rams\_dist port map(clk=>clk1x,a=>a\_a, di=>di\_a,do=>do\_a,we=>locked);    multiplication:process(clk2x)  begin  if locked='1' then  case clk2x is  when '1'=>  var0<=signed(s0);  temp0<=var0;  var1<=signed(alpha\_temp);  temp1<=var1;  var3<=temp0\*temp1;  temp2<=var3;  when '0' =>  var0<=signed(s1);  temp0<=var0;  var1<=signed(one\_minus\_alpha\_temp);  temp1<=var1;  var3<=temp0\*temp1;  temp2<=var3;  when others => null;  end case;  end if;  end process;  add:process(clk2x)  variable var0 :signed(OUT\_SIZE-1 downto 0);  begin  case clk2x is  when '1' =>  temp4<=temp2+0;  when '0' =>  temp4<=temp4+temp2;  when others => null;  end case;  end process;  data\_out:process(clk1x)  begin  if(locked='1') then  if rising\_edge(clk1x) then  p\_f<=std\_logic\_vector(temp4);  end if;  end if;  end process;  sample:  process(clk1x)  variable temp:unsigned(5 downto 0);  variable temp1:signed(7 downto 0);  begin  if (locked='1') then  if (clk1x'event and clk1x='1') then  di\_s<=p\_0;  s0<=do\_s;  s1<=p\_1;  temp:=unsigned(a\_s);  temp:=temp+1;  a\_s<=std\_logic\_vector(temp);  end if;  end if;  end process;  alpha\_delay: process(clk1x)  variable temp:unsigned(5 downto 0);  variable temp1:signed(7 downto 0);  begin  if (locked='1') then  if (clk1x'event and clk1x='1') then  di\_a<=alpha;  temp:=unsigned(a\_a);  temp:=temp+1;  a\_a<=std\_logic\_vector(temp);  alpha\_temp<=do\_a;  one\_minus\_alpha\_temp<=std\_logic\_vector(to\_signed(127,8)-signed(alpha));  end if;  end if;  end process;  end Behavioral; | library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use ieee.numeric\_std.all;  entity part\_3b is  -- Port ( );  generic(SAM\_SIZE:integer:=8;  OUT\_SIZE:integer:=16;  FAC\_SIZE:integer:=8);  Port ( p\_0, p\_1:in std\_logic\_vector(SAM\_SIZE-1 downto 0) ;  alpha:in std\_logic\_vector(FAC\_SIZE-1 downto 0);  p\_f:out std\_logic\_vector(OUT\_SIZE-1 downto 0);  clk:in std\_logic  );  attribute use\_dsp48:string;  attribute use\_dsp48 of part\_3b:entity is "yes";  end part\_3b;  architecture Behavioral of part\_3b is  --CLK WIZARD  component clk\_wiz\_0  port  (-- Clock in ports  -- Clock out ports  clk\_out1 : out std\_logic;  clk\_out2 :out std\_logic;  -- Status and control signals  locked : out std\_logic;  clk\_in1 : in std\_logic  );  end component;  component shift\_registers\_0  port( clk : in std\_logic;  SI : in std\_logic\_vector(7 downto 0);  SO : out std\_logic\_vector(7 downto 0)  );  end component;    signal one\_minus\_alpha\_temp,alpha\_temp, s0, s1,SO\_s,SI\_s,SI\_a,SO\_a :std\_logic\_vector(SAM\_SIZE-1 downto 0);  signal locked,clk1x,clk2x:std\_logic;  signal temp2, temp3, temp4:signed(OUT\_SIZE-1 downto 0);  signal temp0, temp1:signed(SAM\_SIZE-1 downto 0);  signal var0, var1,var2:signed(SAM\_SIZE-1 downto 0);  signal var3,var4,var5:signed(OUT\_SIZE-1 downto 0);  begin  u1:clk\_wiz\_0 port map(clk\_out1=>clk1x, clk\_out2=>clk2x, locked=>locked, clk\_in1=>clk);  u2:shift\_registers\_0 port map(clk=>clk1x, SI=>SI\_s,SO=>SO\_s);  u3:shift\_registers\_0 port map(clk=>clk1x, SI=>SI\_a,SO=>SO\_a);      multiplication:process(clk2x)  begin  case clk2x is  when '1'=>  var0<=signed(s0);  temp0<=var0;  var1<=signed(alpha\_temp);  temp1<=var1;  var3<=temp0\*temp1;  temp2<=var3;  when '0' =>  var0<=signed(s1);  temp0<=var0;  var1<=signed(one\_minus\_alpha\_temp);  temp1<=var1;  var3<=temp0\*temp1;  temp2<=var3;  when others => null;  end case;  end process;  add:process(clk2x)  variable var0 :signed(OUT\_SIZE-1 downto 0);  begin  case clk2x is  when '1' =>  temp4<=temp2+0;  when '0' =>  temp4<=temp4+temp2;  when others => null;  end case;  end process;  data\_out:process(clk1x)  begin  if locked='1' then  if rising\_edge(clk1x) then  p\_f<=std\_logic\_vector(temp4);  end if;  end if;  end process;  sample:  process(clk1x)  begin  if locked='1' then  if (clk1x'event and clk1x='1') then  SI\_s<=p\_0;  s0<=SO\_s;  s1<=p\_1 ;  end if;  end if;  end process;  alpha\_delay: process(clk1x)  begin  if locked='1' then  if (clk1x'event and clk1x='1') then  SI\_a<=alpha;  alpha\_temp<=SO\_a;  one\_minus\_alpha\_temp<=std\_logic\_vector(to\_signed(127,8)-signed(alpha));  end if;  end if;  end process;  end Behavioral; |

**TestBench Code**

| library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use ieee.numeric\_std.all;use IEEE.STD\_LOGIC\_1164.ALL;  use ieee.std\_logic\_unsigned.all;  entity tb\_part\_1 is  generic (DI\_SIZE:integer :=8;  DO\_SIZE:integer :=16);  end tb\_part\_1;  architecture Behavioral of tb\_part\_1 is  component part\_1  Port ( p\_0, p\_1:in std\_logic\_vector(7 downto 0) ;  alpha:in std\_logic\_vector(7 downto 0);  p\_f:out std\_logic\_vector(15 downto 0));  end component;  component part\_2  Port ( p\_0, p\_1:in std\_logic\_vector(7 downto 0) ;  alpha:in std\_logic\_vector(7 downto 0);  one\_minus\_alpha:in std\_logic\_vector(7 downto 0);  p\_f:out std\_logic\_vector(15 downto 0);  clk:in std\_logic);  end component;  component part\_3  Port ( p\_0, p\_1:in std\_logic\_vector(7 downto 0) ;  alpha:in std\_logic\_vector(7 downto 0);  p\_f:out std\_logic\_vector(15 downto 0);  clk:in std\_logic);  end component;  component part\_3b  Port ( p\_0, p\_1:in std\_logic\_vector(7 downto 0) ;  alpha:in std\_logic\_vector(7 downto 0);  p\_f:out std\_logic\_vector(15 downto 0);  clk:in std\_logic);  end component;    constant clock\_period:time:=20 ns;  signal clk:std\_logic;  signal p\_0, p\_1:std\_logic\_vector(DI\_SIZE-1 downto 0);  signal p\_f :std\_logic\_vector(DO\_SIZE-1 downto 0);  signal one\_minus\_alpha, alpha:std\_logic\_vector(DI\_SIZE-1 downto 0);  begin  clock:process  begin  clk<='0';  wait for clock\_period/2;  clk<= not clk;  wait for clock\_period/2;  end process;  --sine\_wave\_inputs :process //For Part 1 and Part 2  --begin  --p\_0 <= std\_logic\_vector(to\_signed(0,DI\_SIZE));  --p\_1 <= std\_logic\_vector(to\_signed(0,DI\_SIZE));  --wait for 12.5 ns;  --p\_0 <= std\_logic\_vector(to\_signed(75,DI\_SIZE));  --p\_1 <= std\_logic\_vector(to\_signed(37,DI\_SIZE));  ----p\_1 <= std\_logic\_vector(to\_signed(37,DI\_SIZE));  --wait for 12.5 ns;  --p\_0 <= std\_logic\_vector(to\_signed(121,DI\_SIZE));  --p\_1 <= std\_logic\_vector(to\_signed(60,DI\_SIZE));  --wait for 12.5 ns;  --p\_0 <= std\_logic\_vector(to\_signed(121,DI\_SIZE));  --p\_1 <= std\_logic\_vector(to\_signed(60,DI\_SIZE));  ----p\_1 <= std\_logic\_vector(to\_signed(60,DI\_SIZE));  --wait for 12.5 ns;  --p\_0 <= std\_logic\_vector(to\_signed(75,DI\_SIZE));  --p\_1 <= std\_logic\_vector(to\_signed(37,DI\_SIZE));  --wait for 12.5 ns;  --p\_0 <= std\_logic\_vector(to\_signed(0,DI\_SIZE));  --p\_1 <= std\_logic\_vector(to\_signed(0,DI\_SIZE));  --wait for 12.5 ns;  --p\_0 <= std\_logic\_vector(to\_signed(-75,DI\_SIZE));  --p\_1 <= std\_logic\_vector(to\_signed(-37,DI\_SIZE));  --wait for 12.5 ns;  --p\_0 <= std\_logic\_vector(to\_signed(-121,DI\_SIZE));  --p\_1 <= std\_logic\_vector(to\_signed(-60,DI\_SIZE));  --wait for 12.5 ns;  --p\_0 <= std\_logic\_vector(to\_signed(-121,DI\_SIZE));  --p\_1 <= std\_logic\_vector(to\_signed(-60,DI\_SIZE));  --wait for 12.5 ns;  --p\_0 <= std\_logic\_vector(to\_signed(-75,DI\_SIZE));  --p\_1 <= std\_logic\_vector(to\_signed(-37,DI\_SIZE));  --wait for 12.5 ns;  --p\_0 <= std\_logic\_vector(to\_signed(0,DI\_SIZE));  --p\_1 <= std\_logic\_vector(to\_signed(0,DI\_SIZE));  --end process; | sine\_wave\_outofphase :process  begin  p\_0 <= std\_logic\_vector(to\_signed(0,DI\_SIZE));  p\_1 <= std\_logic\_vector(to\_signed(-37,DI\_SIZE));  wait for 12.5 ns;  p\_0 <= std\_logic\_vector(to\_signed(75,DI\_SIZE));  p\_1 <= std\_logic\_vector(to\_signed(0,DI\_SIZE));  wait for 12.5 ns;  p\_0 <= std\_logic\_vector(to\_signed(121,DI\_SIZE));  p\_1 <= std\_logic\_vector(to\_signed(37,DI\_SIZE));  wait for 12.5 ns;  p\_0 <= std\_logic\_vector(to\_signed(121,DI\_SIZE));  p\_1 <= std\_logic\_vector(to\_signed(60,DI\_SIZE));  wait for 12.5 ns;  p\_0 <= std\_logic\_vector(to\_signed(75,DI\_SIZE));  p\_1 <= std\_logic\_vector(to\_signed(60,DI\_SIZE));  wait for 12.5 ns;  p\_0 <= std\_logic\_vector(to\_signed(0,DI\_SIZE));  p\_1 <= std\_logic\_vector(to\_signed(37,DI\_SIZE));  wait for 12.5 ns;  p\_0 <= std\_logic\_vector(to\_signed(-75,DI\_SIZE));  p\_1 <= std\_logic\_vector(to\_signed(0,DI\_SIZE));  wait for 12.5 ns;  p\_0 <= std\_logic\_vector(to\_signed(-121,DI\_SIZE));  p\_1 <= std\_logic\_vector(to\_signed(-37,DI\_SIZE));  wait for 12.5 ns;  p\_0 <= std\_logic\_vector(to\_signed(-121,DI\_SIZE));  p\_1 <= std\_logic\_vector(to\_signed(-60,DI\_SIZE));  wait for 12.5 ns;  p\_0 <= std\_logic\_vector(to\_signed(-75,DI\_SIZE));  p\_1 <= std\_logic\_vector(to\_signed(-60,DI\_SIZE));  wait for 12.5 ns;  p\_0 <= std\_logic\_vector(to\_signed(0,DI\_SIZE));  p\_1 <= std\_logic\_vector(to\_signed(-37,DI\_SIZE));  end process;  --alpha\_input:process //For part 1 and part 2  --begin  --alpha <= std\_logic\_vector(to\_signed(0,8));  --one\_minus\_alpha<=std\_logic\_vector(to\_signed(127,8));  --wait for 1000 ns;  --alpha <= std\_logic\_vector(to\_signed(31,8));  --one\_minus\_alpha<=std\_logic\_vector(to\_signed(96,8));  --wait for 1000 ns;  --alpha <= std\_logic\_vector(to\_signed(64,8));  --one\_minus\_alpha<=std\_logic\_vector(to\_signed(63,8));  --wait for 1000 ns;  --alpha <= std\_logic\_vector(to\_signed(96,8));  --one\_minus\_alpha<=std\_logic\_vector(to\_signed(31,8));  --wait for 1000 ns;  --alpha <= std\_logic\_vector(to\_signed(127,8));  --one\_minus\_alpha<=std\_logic\_vector(to\_signed(0,8));  --wait for 1000 ns;  --end process;  alpha\_input:process  begin  alpha <= std\_logic\_vector(to\_signed(0,8));  wait for 1000 ns;  alpha <= std\_logic\_vector(to\_signed(31,8));  wait for 1000 ns;  alpha <= std\_logic\_vector(to\_signed(64,8));  wait for 1000 ns;  alpha <= std\_logic\_vector(to\_signed(96,8));  wait for 1000 ns;  alpha <= std\_logic\_vector(to\_signed(127,8));  wait for 1000 ns;  end process;  --u1:part\_1 port map(p\_0=>p\_0, p\_1=>p\_1, alpha=>alpha, p\_f=>p\_f);  --u2:part\_2 port map(p\_0=>p\_0, p\_1=>p\_1, alpha=>alpha,one\_minus\_alpha=>one\_minus\_alpha,p\_f=>p\_f, clk=>clk);  --u3:part\_3 port map(p\_0=>p\_0, p\_1=>p\_1, alpha=>alpha,p\_f=>p\_f, clk=>clk);  u3:part\_3b port map(p\_0=>p\_0, p\_1=>p\_1, alpha=>alpha,p\_f=>p\_f, clk=>clk);  end Behavioral; |
| --- | --- |