Bilinear Filtering

Recall that the blend equation was:

Cnew = Ca * f + Cb * (1-f)

Where Ca, Cb were two 8-bit colors, and Cnew was a blend of these two colors using the blend factor 'f' (a 9-bit value).

A similar operation is performed when a texture is mapped onto an object in 3D graphics, except that 2 blend factors and four colors are used:

$T_{new} = (1-v)*(1-u)*T_{00} + (1-v)*u*T_{01} + v*(1-u)*T_{10} + u*v*T_{11}$

$T_{00}, T_{01}, T_{01}, T_{11}$ are 8-bit color values as before, with two 9-bit factors v, u used to determine T_{new} .

Bilinear Filtering (cont)

We will use 9-bits to represent u, v as with the blend equation in order to represent 1.0 accurately.

Sample calculations:

u=1.0, v=1.0, then Tnew = T_{11}

u=0.0, v=1.0, then Tnew = T_{10}

u=1.0, v=0.0, then Tnew = T_{01}

u=0.0, v=0.0, then Tnew = T_{00}

u = 0.5, v=0.5 then

Tnew = $0.25*T_{00} + 0.25*T_{01} + 0.25*T_{10} + 0.25*T_{11}$

The Problem

- Use Synopsys Behavioral Compiler to create three different implementations
  - Minimum resource implementation (1 adder, 1 multiplier), no-overlapped computations. New output is produced every ???? clock cycles??
  - 2 Multiplier implementation – no overlapped computations. New output is produced every ????? clock cycles.
  - Overlapped computation implementation in which input bus is always busy and a new output is produced every 4 clock cycles.
- Will use '+','*' , *oneminus* operations from *dwdsp_arith_unsigned* and modules from *DWDSP.sl* synthetic library.
  - Must have completed *DWDSP_mult_csa.vhd* from previous assignment.
- Only keep 8 most significant bits of each multiplication operation.

bifilt Entity

```
entity bifilt is
   port ( clk,reset: std_logic;
          din: in std_logic_vector(8 downto 0);
          coeff_rdy: out std_logic;
          irdy: out std_logic;
          ordy: out std_logic;
          dout: out std_logic_vector(7 downto 0) );
end bifilt;
```

din – input bus for *u*,*v*, *Txx* values

coeff_rdy asserted when input u,v after synchronous reset.

irdy asserted when ready for input of successive Txx value – T00, T01, T01, T11 on successive clock cycles.

ordy asserted when *dout* has valid output value.

bifilt_behv.vhd architecture – *reset* states

```
library ieee,dwdsp;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use dwdsp.dwdsp_arith_unsigned.all;

architecture behv of bifilt is
begin
  main:process
  variable u, v: std_logic_vector(8 downto 0);
  begin
    reset_loop: loop
    ordy <= '0';
    irdy <= '0';
    coeff_rdy <= '0';
    wait until clk'event and clk = '1';
    if (reset = '1') then exit reset_loop; end if;
```

All handshaking lines negated on reset.

bifilt_behv.vhd architecture – *reset* states (cont)

u, v values input on successive clocks after assertion of *coeff_rdy* .

```
    coeff_rdy <= '1';
    wait until clk'event and clk = '1';
    if (reset = '1') then exit reset_loop; end if;
    coeff_rdy <= '0';
    u := din;
    wait until clk'event and clk = '1';
    if (reset = '1') then exit reset_loop; end if;
    v := din;
    wait until clk'event and clk = '1';
    if (reset = '1') then exit reset_loop; end if;
    irdy <= '1';
    wait until clk'event and clk = '1';
    if (reset = '1') then exit reset_loop; end if;
l1: loop -- sample loop
```

bifilt_behv.vhd architecture – *sample_loop*

```
l1: loop -- sample loop
-- fill this in.
   end loop; -- L1
  end loop;  -- reset_loop;
 end process;
```

Fill in the sample_loop. Must input T00, T01, T10, T11 in successive super states (you can compute with a *Txx* value in the same super state in which you input the value).

For non-pipelined implementation, *irdy* and *ordy* must be negated in the first super state, and asserted in the super state in which the output is ready.

For pipelined implementation, *irdy, ordy* are never negated after its initial assertion in the reset states (new Txx value input every clock, new Tnew value every 4 clocks)

bifilt_test.zip Archive

This expands to a *bifilt_test/* directory that provides a testbench for your bifilt implementations. Install this a modelsim library. Files are:

bifilt_behv.vhd -- behavioral model for bifilt implementation, will be used for Synthesis with Behavioral Compiler.

bifilt_mult1.vhd, bifilt_mult2.vhd, bifilt_pipe.vhd – replace these with 3 synthesized gate level implementations (1 multiplier, 2 multipliers, pipelined).

tb.vhd – testbench for use with 'behv', 'mult1', 'mult2' implementations (provides configurations for each).

tb_pipe.vhd – testbench for use with 'pipe' implementation

bifilt_behv.log – log file that has golden output results – output files all implementations should match these outputs.

dsp_dware.zip Archive

This archive unpacks to a *dsp_dware/* directory (same as previous assignment). This only contains three files:

behv/bifilt.vhd -- edit the architecture to contain the architecture you created in *bifilt_test/bifilt_behv.vhd*. Synthesize the mult1 and mult2 implementations via this file.

bifilt_mult1.script -- a dc_shell script that uses *behv/bifilt.vhd* to synthesize a minimum resource implementation using Behavioral compiler. Create new versions of this script (*bifilt_mult2.script*, *bifilt_pipe.script*) to synthesize two multiplier pipelined implementations.

behv/bifilt_pipe.vhd -- replace this with the architecture that will be used for the pipelined implementation – the only difference is that *irdy, ordy* are never negated after its initial assertion (new Txx value input every clock, new Tnew value every 4 clocks)

Procedure

- Complete the *bifilt_test/bifilt_behv.vhd* architecture and simulate in modelsim using the *cfg_behv* configuration provided in *bifilt_test/tb.vhd*
  - The results must match the bifilt_test/bifilt_behv.log results
- Place the architecture from *bifilt_test/bifilt_behv.vhd* into the *dsp_dware/behv/bifilt.vhd* file. Use *dc_shell* and the *dsp_dware/bifilt_mult1.vhd* script to synthesize a gate level implementation
  - Gate level implementation will be placed in the *./gate* directory. Copy this to the *bifilt_test* directory and simulate using modelsim – verify that the output results match the *bifilt_behv* simulation results.

Procedure (cont)

- Create a new version of the *bifilt_mult1.script* such that a two multiplier implementation is synthesized
  - Call new script *bifilt_mult2.script*, write the gate level output to *gate/bifilt_mult2.vhd*.
  - Synthesize using dc_shell; look at the report file and verify that two multipliers are used
  - Copy the *gate/bifilt_mult2.vhd* file to the *bifilt_test* directory and simulate with modelsim – verify the output results match the *bifilt_behv* results.

Procedure (cont)

- Create a new version of the *bifilt_mult1.script* such that a pipelined implementation is synthesized that inputs a new *Txx* value every clock with outputs produced every four clocks
  - Call the new script *bifilt_pipe.script*, must read the file *behv/bifilt_pipe.vhd*.
  - Must create a new file called 'behv/bifilt_pipe.vhd' that is only a slight modification of the original 'behv/bifilt.vhd' – *irdy* is never negated after its assertion
  - Use the configuration named *cfg_pipe* provided in *bifilt_test/tb_pipe.vhd* to verify that the output results match the *bifilt_test/bifilt_behv* results.

Required Files for Submission

- All files placed in directory called sim7
- *./bifilt_mult2.script* - script for synthesizing 2 multiplier implementation; must read file *behv/bifilt.vhd* and produce file *gate/bifilt_mult2.vhd*
- *./bifilt_pipe.script* - script for synthesizing pipelined implementation; must read file *behv/bifilt_pipe.vhd* and produce file *gate/bifilt_pipe.vhd*
- *./behv/bifilt.vhd* - file read by *bifilt_mult2.script*
- *./behv/bifilt_pipe.vhd* - file read by *bifilt_pipe.script.*

Comments on Testbench (*tb.vhd*, *tb_pipe.vhd*)

- Testbench computes 8 Tnew values using 8 sets of Txx values read from a 32-location memory (8 x 4 = 32).
- 6 different values of u,v used for each set of 8 Tnew values
  - V=1.0, u = 0.0
  - V=0.0, u = 1.0
  - V=0.0, u = 0.0
  - V=1.0, u = 1.0
  - V = 0.5, u = 0.5
  - V = 0.75, u = 0.25
- For the last two cases, might get a different value in the LSB than my provided golden file depending on the order of the multiplications (1-v|v * 1-u|u * Txx)
  - Difference is due to dropping the least significant 8 bits.