Streaming colour interpolator

Team Name: Logicians

Team members:

  Rana Prathap : 140050068
  Abhijeet      : 140050052
  Srinath      : 140050080
Overview

The project is implementing “Streaming colour interpolator in a digital image” using VHDL.

It can be implemented using software also but it will be much faster if implemented using hardware.

High level Architecture

Inputs and outputs of overall system:

- clock :: (1 bit) The clock of the system
- reset :: (1 bit) Used to reset the system when it is ‘1’ for 2 clock cycles
- start :: (1 bit) Indicates start of input streams when it is ‘1’ for 2 clock cycles
- r :: (16 bit) Takes value of red component of a pixel
- g :: (16 bit) Takes value of green component of a pixel
- b :: (16 bit) Takes value of blue component of a pixel
- Output_Ready :: (1 bit) Indicates that the output is ready ie. started to stream
- r_out :: (16 bit) Holds output value of processed red component of a pixel
- g_out :: (16 bit) Holds output value of processed green component of a pixel
- b_out :: (16 bit) Holds output value of processed blue component of a pixel

Algorithm implemented:

Variables used :

A,B,C: We are using three Arrays A,B,C each of size 3x100 to store the pixels.
At any point of time a window of 3x100 is stored in the memory, each entry A[i] internally contains three pixel values (r,g,b) of 16 bits each, which can be accessed as A[0][i], A[1][i], A[2][i] respectively. Similar is the case with B and C.

`local_clk_cycles`: Counts the number of clock cycles finished from the time of entering at each block as shown in flowchart

`line_no_taken`: Indicates the line number till which input is taken

`order_read`: Indicates the order in which the input should be read and order in which processing to be done, 0→[0---->99], 1→[99---->0]

If `order_read=0`, read from column 0 to 99
If `order_read=1`, read from column 99 to 0

Process:

1) Initialise all the variables that are mentioned above all to ‘0’
2) Read the 1st line and the 2nd line of the image into B, C respectively in given order
   i.e., B[0--99], C[99--0]. This step creates the delay between input and output.
   This is done till 200 clock cycles, increment ‘line_no_taken’ by 2.
3) For the next 100 clock cycles process elements of B in order specified by ‘order_read’ and take input to A in same order. Increment ‘line_no_taken’ by 1, flip ‘order_read’.
4) If ‘global clock cycles’ reaches 10000, stop process else go to step 5
5) For the next 100 clock cycles process elements of C in order specified by ‘order_read’ and take input to B in same order. Increment ‘line_no_taken’ by 1, flip ‘order_read’.
6) If ‘global clock cycles’ reaches 10000, stop process else go to step 7
7) For the next 100 clock cycles process elements of A in order specified by ‘order_read’ and take input to C in same order. Increment ‘line_no_processed’ by 1 and ‘line_no_processed’ by 1, flip ‘order_read’.
8) If ‘global clock cycles’ reaches 10000, stop process else go to step 3
Note:

In steps 3, 5, 7 if ‘line_no_taken’ is already 100 do not take any inputs, simply initialize all elements of the array (which is inputted if line_no_taken != 100) to 0.

Since the number of pixels, rows and columns are known priorly (100, 100), we can calculate exactly where the 10000th global clock cycle reaches (i.e., in which step) and so can stop the process there. This is one of the optimizations performed.

Flow chart indicating sequence of steps to be performed:
Overview of blocks:

If reset is made 1 for 2 clock cycles at any state, system goes to idle state.

State s0: (Idle state)
This is the idle state. Here variables and registers (Arrays used here) are in their default values. If start = 1 for 2 clock cycles and reset = 0 go to s1

Block s1: (Read B,C)

The first 100 pixels (i.e., r,g,b values of each of the 100 pixels) are read from index(j) 0 to 99 into a 3x100 array B[3][100], next 100 pixels from j=99 to 0 into 3x100 array C[3][100]. Increment ‘local_clock_cycles’ by 1 at each pixel input.

If local_clock_cycles = 200,
set local_clock_cycles = 0, order_read = 0; go to block s2

Block s2: (Processing 1)

Based on ‘order_read’ decide whether to process from 0 to 99 or 99 to 0. Process each pixel of B using pixels of A and C which are above and below B respectively. Keep track of local_clock_cycles taken, and finally line_no_processed. Whenever a processed pixel of B is output, take element of A(pixel of next row) with same index from input.

If ‘line_no_processed’ reaches 100, go to state s0, by setting everything to their default.
If ‘local_clock_cycles’ reaches 100 and line_no_processed < 100, flip ‘order_read’,
set local_clock_cycles = 0, go to s3. Make Output_ready to 1 just before first value is being output.

Block s3: (Processing 2)

Based on ‘order_read’ decide whether to process from 0 to 99 or 99 to 0. Process each pixel of C using pixels of B and A which are above and below C respectively. Keep track of local_clock_cycles taken, and finally line_no_processed. Whenever a processed pixel of C is output, take element of B(pixel of next row) with same index from input.

If ‘line_no_processed’ reaches 100, go to state s0, by setting everything to their default.
If ‘local_clock_cycles’ reaches 100 and line_no_processed < 100, flip ‘order_read’,
set local\_clock\_cycles = 0, go to s4.

**Block s4:** (Processing 3)

Based on ‘order\_read’ decide whether to process from 0 to 99 or 99 to 0. Process each pixel of A using pixels of C and B which are above and below A respectively. Keep track of local\_clock\_cycles taken, and finally line\_no\_processed. Whenever a processed pixel of A is output, take element of C(pixel of next row) with same index from input.

If ‘line\_no\_processed’ reaches 100, go to state s0, by setting everything to their default.

If ‘local\_clock\_cycles’ reaches 100 and line\_no\_processed <100 , flip ‘order\_read’, set local\_clock\_cycles = 0, go to s2.

**Note:**

While taking input into arrays in s2,s3,s4 , if already 100 lines are taken, input 0 into the arrays. At any state if reset is held for 2 clock cycles. Here each block has many states.

**Description of functionalities of blocks:**

**Processing:**

Processing each pixel of any array using the other two arrays is defined such that if:

A : array of pixels above the processing array's pixels

B: array of pixels which are to be processed

C: array of pixels which are below the processing array's pixels

x: index of the pixel which is being processed

Thus each pixel of B is processed as shown

\[
\begin{align*}
\text{r\_sum} &= A[x][0] + B[x-1][0] + B[x+1][0] + C[x][0] \\
\text{r\_out} &= B[x][0]*(B[x][0]-\text{r\_sum})/65536 + \text{r\_sum} \\
\text{g\_sum} &= A[x][1] + B[x-1][1] + B[x+1][1] + C[x][1] \\
\text{g\_out} &= B[x][1]*(B[x][1]-\text{g\_sum})/65536 + \text{g\_sum} \\
\text{b\_sum} &= A[x][2] + B[x-1][2] + B[x+1][2] + C[x][2]
\end{align*}
\]
\[ b_{\text{out}} = B[x][2]*(B[x][2]-b_{\text{sum}})/65536 + b_{\text{sum}} \]

This process is done for all the pixels i.e., for \( x \) ranging from 0--99.

Here since the values of indices go out of bound when \( x = 0, 99 \), we assume that when
the process is actually implemented, the array to be processed is padded with two
columns on the two extremities of that array. This functionality is accomplished by
having the actual pixels in arrays \( A, B, C \) of sizes 102 in the indices between 1-100 and the
remaining two indexes contain the dummy values which are zero.

Process \( B[i] \) Block Level:

All the processing is done using a single processing block in which processing goes via
multiple processes simultaneously

**Processing step**

At each clock cycle just after first 200 clock cycles, we get the data required to process
some pixel from the arrays which store them and send these to a separate processing
block, in which each of the output values of R,G,B are calculated in three different
processes which run simultaneously to give the required output,

We have tested the processing block separately by giving input pixels and creating
a separate testbench for it and found that the output comes at each clock cycle correctly

**Assumptions and constraints**

**Assumptions:**

Pixels are inputted for one clock cycle each i.e., 10000 continuous clock cycles for
100x100 pixels.

Order of inputting pixel values changes alternatively i.e., for the first row it is from left to
right and for the second row it is from right to left and so on.

**Constraints:**

Initial processing delay cannot be avoided because, processed values can only be
obtained if we have all the five pixel values where two of them occur in two other rows
making it inevitable.

Maximum storage capacity of the hardware limits the no. of pixels which can be
processed in a single row.
Applicable for any mxn arrays of pixel values, wherein if some of the pixel values are missing, input should pad them with 0 pixel value thus still making it a rectangular input pattern.

Testing and Verification

Here the number of pixels actually needed to store the pixel values are 100 but for the reason of easy processing, we define arrays of sizes 102 where the first and last pixel values are used to pad the actual pixel values with zeroes.

For the pixel values occurring above and below the required pixel when they go out of bound, since initially all arrays are initialised with 0 and finally when line_no_read=100 the arrays are again into aliased with zero thereby the array values are not out of bound and give the aimed results.

Testing Framework

We are using VHDL testbench for testing our design, The input is taken from 3 text files On for each color r,g,b. and at each rising clock the input pixel is converted to 16 bit vector and fed to inputs of our main design, the outputs are written down to 3 output text files as long as the output streams out.

We tested the correctness of outputs using an exact replica of the image processing in c++, and

Time analysis:

Overall input streaming : 10000 clock cycles
Output starts at : 201 clock cycles------
Delay : 201 clock cycles ------
Output stops at : 10201 clock cycles

Assuming input starts at time 0.

When simulated on vhdl it looks

Used clock period 10 ns.
Input starts at 145 ns
When an image is given as input to the algorithm the image obtained as output is shown below.

Input image :-

![Input Image]

Final image generated :-

![Final Image]

Clearly the output image is expected to be blurred or the red, blue and green values appears to be spread around. This validates or expresses the logic inherited by the algorithm.
**Work done by individuals**

**Rana Prathap** : Processing (logicians_worker.vhd), Optimizing use of hardware resources, debugging top module and test bench, interpolate.cpp to test output

**Abhijeet** : Processing(logicians_worker.vhd) and testing implementation in VHDL, documentation

**Srinath** : parts of main block code and processing block code, part of testbench, testbench checking correctness for logicians_worker.vhd, debugging, python codes for Testing and getting images, documentation

**Mugilvannan** : Testing implementation in VHDL, miscellaneous debugging in certain parts of the code including writing certain c++ files, writing parts of documentation.