HUNT ENGINEERING
Chestnut Court, Burton Row,
Brent Knoll, Somerset, TA9 4BP, UK
Tel: (+44) (0)1278 760188,
Fax: (+44) (0)1278 760199,
Email: sales@hunteng.co.uk
http://www.hunteng.co.uk
http://www.hunt-dsp.com

HUNT ENGINEERING

VHDL Image Processing Source Modules

REFERENCE MANUAL

Document Rev B
J. Maddocks & R. Williams 11/12/2003

COPYRIGHT

WARRANTIES LIABILITY and INDEMNITIES

HUNT ENGINEERING warrants the hardware to be free from defects in the material and workmanship for 12 months from the date of purchase. Product returned under the terms of the warranty must be returned carriage paid to the main offices of HUNT ENGINEERING situated at BRENT KNOLL Somerset UK, the product will be repaired or replaced at the discretion of HUNT ENGINEERING.

Exclusions - If HUNT ENGINEERING decides that there is any evidence of electrical or mechanical abuse to the hardware, then the customer shall have no recourse to HUNT ENGINEERING or its agents. In such circumstances HUNT ENGINEERING may at its discretion offer to repair the hardware and charge for that repair.

Limitations of Liability - HUNT ENGINEERING makes no warranty as to the fitness of the product for any particular purpose. In no event shall HUNT ENGINEERING'S liability related to the product exceed the purchase fee actually paid by you for the product. Neither HUNT ENGINEERING nor its suppliers shall in any event be liable for any indirect, consequential or financial damages caused by the delivery, use or performance of this product.

Because some states do not allow the exclusion or limitation of incidental or consequential damages or limitation on how long an implied warranty lasts, the above limitations may not apply to you.

TECHNICAL SUPPORT

Technical support for HUNT ENGINEERING products should first be obtained from the comprehensive Support section www.hunteng.co.uk/support/index.htm on the HUNT ENGINEERING web site. This includes FAQs, latest product, software and documentation updates etc. Or contact your local supplier - if you are unsure of details please refer to www.hunteng.co.uk for the list of current re-sellers.

HUNT ENGINEERING technical support can be contacted by emailing support@hunteng.demon.co.uk, calling the direct support telephone number +44 (0)1278 760775, or by calling the general number +44 (0)1278 760188 and choosing the technical support option.

N.B. Technical support for the Image Processing VHDL source modules is provided for users of HUNT ENGINEERING hardware ONLY.

TABLE OF CONTENTS

HUNT ENGINEERING provides families of HERON modules that have FPGAs, often combined with some interface capability. The HERON-FPGA family in particular provides an FPGA along with a large number of signals routed to general-purpose connectors. These modules are suitable for connecting to digital cameras, where the control of the camera and image capture can be performed by the FPGA fitted to the module.

The HERON-FPGA range of modules is therefore highly suited to performing image processing on data directly received from a camera.

HUNT ENGINEERING also provide VHDL source to assist in the development of applications for HERON-FPGA modules. This VHDL source includes a set of VHDL Image Processing Source Modules that are intended to make the task of building an imaging system quicker and easier. As the VHDL source is made available to users of HUNT ENGINEERING boards, this allows the functions to be extended and modified as necessary.

The components provided in the Image Processing Source Modules are designed to be drop-in modules that use the Hunt Engineering pixel pipeline format to interface with adjacent components. They are intended to perform many of the common functions used in image processing such as image subtraction and convolution.

Due to the highly parallel nature of an FPGA and the large amount of concurrent IO provided by the HERON–FPGA modules, image processing functions can be performed at pixel rate.

The VHDL Image Processing Source Modules are ideal for image processing applications, providing optimised functions for many common operations in imaging. Each VHDL function provided is scalable and flexible, and provided as VHDL source that can be edited if required.

The VHDL imaging functions when mapped to the architecture of a Virtex-II FPGA allow image operations to be performed at a pixel rate. That is, the time taken to calculate a new value for one pixel in an image is no greater than the pixel clock period for that system. In comparison, when performing the same operation on a DSP several system clock cycles may be required to calculate each individual pixel result.

The VHDL Imaging Processing functions are organised as separate VHDL entities. The VHDL entities are grouped together in separate VHDL source files according to their function type. Each separate entity performs one particular image processing function.

For each image processing function, the HUNT ENGINEERING Pixel Pipeline Format is used. This enables components to be connected together regardless of the previous or subsequent operations. The data inputs and outputs are scalable so that no data loss in the pipeline is necessary.

To use the VHDL entities supplied all that is required is to add the appropriate source module to your VHDL project and instantiate the entity you wish to use. Having done the next step is to supply data to that entity in the format described in this document. The resulting output data from that entity should then be connected to the next stage in the image processing pipeline.

Function Classes

When processing an image, there are several different kinds of mathematical operation that could be performed, each with quite different requirements on how pixels are used.

The simplest type of operation is to perform one fixed operation equally across all pixels within an image. For example, to brighten an image one fixed value may be added to all pixels within the image.

The second type of operation is to perform one particular mathematical operation on each pixel but using a different value depending on the position of that pixel in the image. An example of this would be adding two images together, where the addition value for a pixel in image A is the value of the corresponding pixel in the same position in image B.

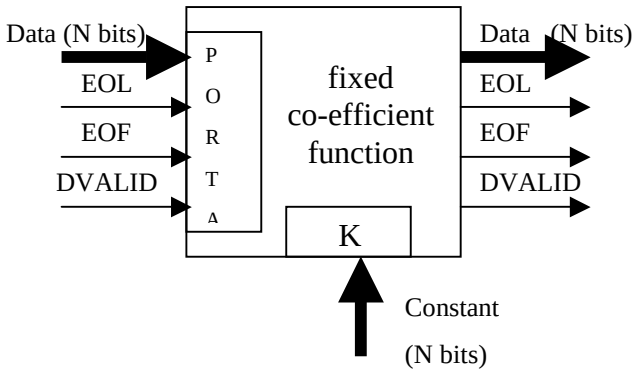
The third type of operation is convolution. This is a neighbourhood processing operation in that several pixels within a window all contribute mathematically to the result for one pixel. This type of processing would be used to perform functions such as edge detection or softening and blurring of an image for example.

With several different types of operation possible the VHDL Image Processing Source has been organised into three distinct function classes.

- `FIXED CO-EFFICIENT OPERATORS`: Functions that perform the same operation with a fixed constant for every pixel in a frame
- `VARIABLE CO-EFFICIENT OPERATORS`: Functions that perform the same operation with a varying value for every pixel in a frame
- `CONVOLUTION OPERATORS`: Functions that perform convolution on a frame

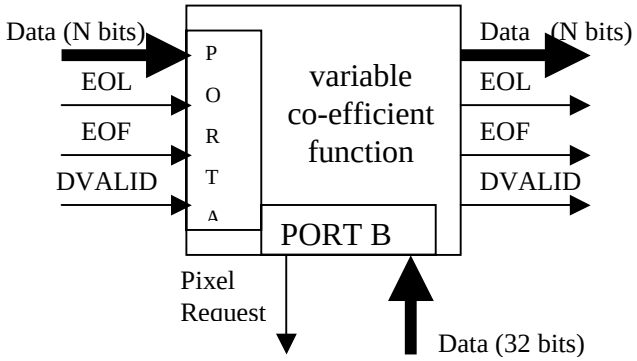
Fixed Co-Efficient Functions

The following diagram shows the architecture of each function in the Fixed Co-efficient class. For each function, pixel data is provided on Port A and operated against the fixed value supplied on the K input. The result is output as shown on the right hand edge of the diagram.



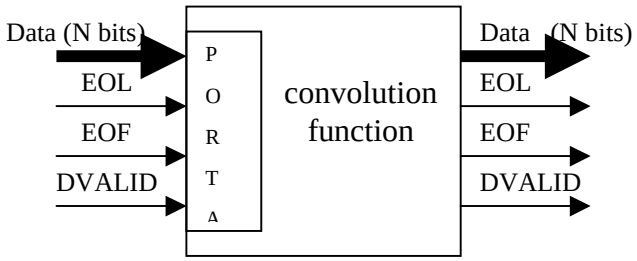
Variable Co-Efficient Functions

The following diagram shows the architecture of each function in the Variable Co-efficient class. For each function, pixel data is provided on Port A and operated against the data received on Port B. The result is output as shown on the right hand edge of the diagram.



Convolution Functions

The following diagram shows the architecture of each function in the Convolution class. For each function, pixel data is provided on Port A. The result is output as shown on the right hand edge of the diagram.



The input ports, Port A, for each component and the output ports work using the Pixel Pipeline Format.

The pixel pipeline format requires there be three control signals present, End Of Line (EOL), End Of Frame (EOF), and Data Valid (DVALID). The EOL signal is active high and asserted at a time following the last pixel of a line to indicate the end of that line. The EOF signal is active high and asserted at a time following the last pixel of a frame to indicate the end of that frame. The EOL and EOF signals must not be high at the same time. DVALID is held high while the data on the port is valid.

In addition to these three signals the pixel pipeline format requires a pixel strobe to be present. This is a free running clock driven by the camera pixel clock. All of the image processing functions work on the rising edge of the pixel clock.

For a more complete discussion of the Pixel Pipeline Format, please refer to the relevant section at the end of the RS422 and Camera-Link documents provided on the HUNT ENGINEERING CD and web-site.

Generic Options

The VHDL Image Processing functions have various generic options as discussed below.

Synchronous (Fixed co-efficient functions only): This is a Boolean option and is set to true by default. It controls whether a registered or unregistered version of the component is used.

N: This option is an integer that controls the width of the data bus. Its default value is 32.

StopAtZero (Subtract functions only): This is used to control whether when subtracting the answer will wrap around e.g. 0x00000002 – 0x00000004 = 0xFFFFFFFE or

0x00000002 – 0x00000004 = 0x00000000

It defaults to true.

FillWithOnes (Shift functions only): This Boolean option controls whether the shift operators introduce zeros or ones. When set to true, will fill with '1', when set to false will fill with '0'. This option defaults to false.

The following example shows an instantiation of the component SUBK. In this example the data width is set to 32, the asynchronous version will be used, and the component will not wrap around for subtraction.

```
iSUBK : SUBK
      generic map(
      Synchronous => False,
      StopAtZero  => True,
      N           => 32)
    Port map(
      DATA_IN     => DATA_A,
      DATA_OUT    => DATA_SUBK,
      DVALID_IN   => DVALID_A,
      DVALID_OUT  => DVALID_SUBK,
      CLOCK       => FCLK_G,
      RESET       => RESET,
      EOL_IN      => EOL_A,
      EOL_OUT     => EOL_SUBK,
      EOF_IN      => EOF_A,
      EOF_OUT     => EOF_SUBK,
      K_IN        => CONST);
```

Figure 1 - Example of setting generic options

Understanding Bus Widths

When using each of the image processing functions, you need to consider the appropriate widths of the data busses you connect to the inputs and outputs of each function.

All of the functions provide input data busses and output data busses that are the same size, with this size being set by the generic N.

For logical operations such as 'AND' and 'OR', the size of the result cannot change from the size of the original input data. For arithmetic operations however, such as 'ADD' and 'MPY', the internal result will grow. This internal result is then resized to match the output data bus size set by the generic N.

For example, using the addition functions with N set to 8, any internal result greater than 255 would be set to 255. This resizing is done because when you are creating an 8-bit data processing pipeline it is important to correctly resize the result at each stage in the pipeline.

For those cases where you would like to allow the result to grow you simply need to specify a larger value for N and pad the top of the input values with 0. For example, to allow an 8-bit addition to grow to a 9-bit result, you would simply need to set N to 9, and increase the size of the input data bus by one bit. The top bit of this new input data bus would then be set to 0. The output data results would become 9-bit values.

In addition to the input data and output data busses some components provide a K input port and some components provide a Port B.

For the Fixed Co-efficient functions, the K input port width will always match that set by the generic N.

For the Variable Co-efficient functions, the Port B data bus is always 32 bits as it is intended that these components source their Port B data from the 32-bit SDRAM interface of compatible HERON-FPGA modules. However, it would also be perfectly valid to source the second data stream for Port B from any other source such as a ROM made from internal Block RAM.

Component Latency and Output Controls

The control signals output by each function take into account the latency of operation inside the component. For example if the processing taking place on the data takes two clock cycles then the control signals will be delayed by two clock cycles within the component. The latency of each component differs and can be found towards the end of this document.

Data Format

For all functions included in the VHDL Image Processing Source Modules the input data busses and output data busses used unsigned data representation.

If there is a signed operation inherently performed by the component, for example where a signed Virtex-II hardware multiplier is used inside a component implementation, a sign conversion is performed on the input data bus and output data bus. When this is done the result is unsigned data busses on the ports of the function with signed data busses internally.

The VHDL Image Processing Source Modules provide a set of functions to perform many standard image processing operations on image data. For any application that is using the image processing functions there will also need to be some associated logic that is presenting data for processing and logic for storing or outputting the results.

The most obvious source for image data is a camera. Available on the HUNT ENGINEERING CD and web-site are several examples for digital RS422 and Camera-Link cameras. These examples provide all of the logic necessary to receive data from appropriate digital cameras. The examples also include a few other standard functions such as region of interest and frame control.

Other sources for camera data include external SDRAM interfaces available on several HERON-FPGA modules, or data provided over a HERON FIFO connection.

For the results of the image processing operation, again the external SDRAM interface may be used, or the data may be transmitted through a HERON FIFO to another HERON module or the Host interface.

When using the external SDRAM interface of certain HERON-FPGA modules, there are several key points to consider that include how images are organised in memory and how the memory is accessed in a way that suits the format of the Image Processing functions. All of these issues are discussed in the document 'Using the off-chip SDRAM for Image Processing on HERON-FPGA Modules' which can be found on the HUNT ENGINEERING CD or web-site.

Region of Interest

It is common for an image processing application to perform processing on a small window rather than on the whole image. In reducing the image processing to just a selected area of image, a region of interest is used to directly specify the area to work on.

A region of interest function is provided as part of the digital camera examples for RS422 and Camera-Link cameras. This region of interest function works with the same Pixel Pipeline Format as the Image Processing functions, and can be easily incorporated into the same VHDL project.

Image Capture and Image Processing

The most common situation for image processing with HERON-FPGA modules is to perform the camera interfacing and image processing together.

In fact, any of the standard RS422 or Camera-Link examples may be used as a starting point for development. The Pixel Pipeline Format being common to both sets of logic means that combining image capture functions with image processing functions is extremely straight-forward.

In doing so you will already have a project that is set up to correctly receive data from a digital camera. To this can be added the appropriate image processing functions to further process the camera data.

The only limitations that apply to the Image Processing Source Modules are that of available FPGA resources and their speed of operation.

FPGA Resources – Multipliers and Block RAMs

For each function a differing number of FPGA resources will be used and a different operating speed will be possible. For the majority of functions the resources used simply include Look-Up-Tables (LUTs) and registers. However, several key functions will potentially make use of the Virtex-II built-in hardware multipliers. Convolution functions in particular may also require the use of Virtex-II Block RAM.

For any particular Virtex-II device there will be a fixed, finite number of multipliers and Block RAMs available. When using the Image Processing functions it is important to know how many of each of these resources will be used for each operation. In order to calculate the resources that will be used you should refer to the table towards the end of this document for details of the resources used by each function.

FPGA Performance

All of the components supplied as part of the VHDL Image Processing Source Modules are intended to be run from one system-wide Pixel Clock. For each individual application the maximum frequency of the Pixel Clock will be governed by the slowest operation performed in the pipeline.

For many of the components, it is difficult to provide a complete set of performance figures as there are many issues that will affect the overall performance. For example, by varying the generic N of each component, the overall performance will drop as wider and wider data operations are performed.

In addition, many components may be pipelined together. When doing this the Pixel Clock rate will relate to the number of stages of maths performed between one register stage in the pipeline and the next. For this reason, the synchronous/asynchronous generic has been provided so that register stages can be introduced at key points in the pipeline in order to increase the operation frequency.

For a guide to the performance that can be expected from each function, please refer to the performance table at the end of this document.

Provided on the HUNT ENGINEERING CD and web-site is an Imaging Demo that takes data from a digital camera and performs several image processing functions before transmitting the new image data to the host for display on the PC monitor.

This example can be used as a starting point for the development of an image processing application. Alternatively, one of the standard digital camera examples (RS422, or Camera-Link) may be used as a starting point, with the required Image Processing functions instantiated as discussed in this document.

Fixed Co-efficient Functions

AddK	Add a constant to each pixel
SubK	Subtract a constant from each pixel
MpyK	Multiply constant with each pixel
AndK	AND a constant with each pixel
OrK	OR a constant with each pixel
XorK	XOR a constant with each pixel
RShK	Right shift each pixel a number of positions
LShK	Left shift each pixel a number of positions
Invert	Invert each pixel
Square	Square each pixel
FillK	Fill each pixel with same constant
FillRamp	Fill image with vertical ramps

Variable Co-efficient Functions

AddImage	Add two images
SubImage	Subtract two images
MpyImage	Multiply two images
AndImage	AND two images
OrImage	OR two images
XorImage	XOR two images

Convolution Functions

`Convolve`	Performs a convolution with a 3x3 window
`Convolve5x5`	Performs a convolution with a 5x5 window

Fixed Co-efficient Functions

AddK : Add constant to each pixel

```
Entity ADDK is
  Generic (
    Synchronous : boolean := True;
    N           : integer := 32
  );
  Port (
    DATA_IN     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT    : out std_logic_vector(N - 1 downto 0);
    DVALID_IN   : in  std_logic;
    DVALID_OUT  : out std_logic;
    CLOCK       : in  std_logic;
    RESET       : in  std_logic;
    EOL_IN      : in  std_logic;
    EOL_OUT     : out std_logic;
    EOF_IN      : in  std_logic;
    EOF_OUT     : out std_logic;
    K_IN        : in  std_logic_vector(N - 1 downto 0)
  );
end ADDK;
```

Using the ADDK Component

The AddK component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DATA_IN, DVALID_IN, EOL_IN and EOF_IN.

The constant to be added to each pixel value must be presented to the K_IN input.

The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT, and EOF_OUT.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

If the Synchronous generic is set to true (the default value) the result is registered internally before being output by the component. If the Synchronous generic is set to false the function will be implemented as a purely combinatorial function.

The data busses are all unsigned and scaled according to the generic N. The width of the DATA_IN and K_IN busses will equal the value N in bits. The DATA_OUT width will also be set to N.

Output data results are rounded to ensure data does not exceed the maximum output value for the DATA_OUT bus. For example, with N set to 8 all internal results greater than 255 would be set to 255.

SubK : Subtract constant from each pixel

```
Entity SUBK is
  Generic (
    Synchronous : boolean := True;
    StopAtZero  : boolean := True;
    N           : integer := 32
  );
  Port (
    DATA_IN     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT    : out std_logic_vector(N - 1 downto 0);
    DVALID_IN   : in  std_logic;
    DVALID_OUT  : out std_logic;
    CLOCK       : in  std_logic;
    RESET       : in  std_logic;
    EOL_IN      : in  std_logic;
    EOL_OUT     : out std_logic;
    EOF_IN      : in  std_logic;
    EOF_OUT     : out std_logic;
    K_IN        : in  std_logic_vector(N - 1 downto 0)
  );
end SUBK;
```

Using the SubK Component

The SubK component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs `DATA_IN`, `DVALID_IN`, `EOL_IN` and `EOF_IN`.

The constant to be subtracted from each pixel value must be presented to the `K_IN` input.

The result is output in the Pixel Pipeline Format on the outputs `DATA_OUT`, `DVALID_OUT`, `EOL_OUT`, and `EOF_OUT`.

The `CLOCK` input must be driven with the system wide pixel clock, and the `RESET` input driven by an active-high reset signal. All clock activity is on the rising edge of the `CLOCK` input.

If the `Synchronous` generic is set to true (the default value) the result is registered internally before being output by the component. If the `Synchronous` generic is set to false the function will be implemented as a purely combinatorial function.

The data busses are all unsigned and scaled according to the generic `N`. The width of the `DATA_IN` and `K_IN` busses will equal the value `N` in bits. The `DATA_OUT` width will also be of width `N`.

The `StopAtZero` generic option will dictate what will happen if the result of the subtraction is less than one. If the result of a subtraction operation is less than zero and this option is set to true then the answer will be zero. If the result of the subtraction is less than zero and this option is set to false then the answer will wrap around.

If `StopAtZero` is true then 0x0002 – 0x0004 will result in 0x0000

If `StopAtZero` is false then 0x0002 – 0x0004 will result in 0xFFFE

MpyK : Multiply constant with each pixel

```
Entity MPYK is
  Generic (
    Synchronous : boolean := True;
    N           : integer := 18
  );
  Port (
    DATA_IN     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT    : out std_logic_vector(N - 1 downto 0);
    DVALID_IN   : in  std_logic;
    DVALID_OUT  : out std_logic;
    CLOCK       : in  std_logic;
    RESET       : in  std_logic;
    EOL_IN      : in  std_logic;
    EOL_OUT     : out std_logic;
    EOF_IN      : in  std_logic;
    EOF_OUT     : out std_logic;
    K_IN        : in  std_logic_vector(N - 1 downto 0)
  );
end MPYK;
```

Using the MpyK Component

The MpyK component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DATA_IN, DVALID_IN, EOL_IN and EOF_IN.

The constant for multiplication must be presented to the K_IN input. The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT, and EOF_OUT.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

If the Synchronous generic is set to true (the default value) the result is registered internally before being output by the component. If the Synchronous generic is set to false the function will be implemented as a purely combinatorial function.

The data busses are all unsigned and scaled according to the generic N. The width of the DATA_IN, K_IN and DATA_OUT busses will equal the value N in bits. Output data results are rounded to ensure data does not exceed the maximum output value for the DATA_OUT bus. For example, with N set to 8 all internal results greater than 255 would be set to 255.

The multiplier component is a little more complex than the others in that the size of the input bus decides what sort of implementation will be used. The Xilinx Virtex II FPGA has built in 18 x 18 multipliers. If N is defined as being 18 or less then the MPYK component will use these built in multipliers. If however N is defined as being greater than 18 then COREGEN IP multipliers are used instead. The two multipliers included with these components are a 24 x 24 and a 32 x 32. The reason you would not automatically want to use the largest multiplier is down to the amount of resources used. As a guide, the 24 x 24 COREGEN IP multiplier using look up tables (LUT's) uses 481 LUT's and 614 Flip Flops, 4% and 6% respectively of the total resources available on a 1M gate part. The 32 x 32 COREGEN IP multiplier uses 1080 LUT's and 1235 Flip Flops, 10% and 12% respectively of the total resources available on a 1M-gate part.

AndK : AND constant with each pixel

```
Entity ANDK is
  Generic (
    Synchronous : boolean := True;
    N           : integer := 32
  );
  Port (
    DATA_IN     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT    : out std_logic_vector(N - 1 downto 0);
    DVALID_IN   : in  std_logic;
    DVALID_OUT  : out std_logic;
    CLOCK       : in  std_logic;
    RESET       : in  std_logic;
    EOL_IN      : in  std_logic;
    EOL_OUT     : out std_logic;
    EOF_IN      : in  std_logic;
    EOF_OUT     : out std_logic;
    K_IN        : in  std_logic_vector(N - 1 downto 0)
  );
end ANDK;
```

Using the AndK Component

The AndK component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs `DATA_IN`, `DVALID_IN`, `EOL_IN` and `EOF_IN`.

The constant for the logical operation must be presented to the `K_IN` input.

The result is output in the Pixel Pipeline Format on the outputs `DATA_OUT`, `DVALID_OUT`, `EOL_OUT`, and `EOF_OUT`.

The `CLOCK` input must be driven with the system wide pixel clock, and the `RESET` input driven by an active-high reset signal. All clock activity is on the rising edge of the `CLOCK` input.

If the `Synchronous` generic is set to true (the default value) the result is registered internally before being output by the component. If the `Synchronous` generic is set to false the function will be implemented as a purely combinatorial function.

The data busses are all unsigned and scaled according to the generic `N`. The width of the `DATA_IN` and `K_IN` busses will equal the value `N` in bits. The `DATA_OUT` width will also be `N` bits wide.

OrK : OR constant with each pixel

```
Entity ORK is
  Generic (
    Synchronous : boolean := True;
    N           : integer := 32
  );
  Port (
    DATA_IN     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT    : out std_logic_vector(N - 1 downto 0);
    DVALID_IN   : in  std_logic;
    DVALID_OUT  : out std_logic;
    CLOCK       : in  std_logic;
    RESET       : in  std_logic;
    EOL_IN      : in  std_logic;
    EOL_OUT     : out std_logic;
    EOF_IN      : in  std_logic;
    EOF_OUT     : out std_logic;
    K_IN        : in  std_logic_vector(N - 1 downto 0)
  );
end ORK;
```

Using the OrK Component

The OrK component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs `DATA_IN`, `DVALID_IN`, `EOL_IN` and `EOF_IN`.

The constant for the logical operation must be presented to the `K_IN` input.

The result is output in the Pixel Pipeline Format on the outputs `DATA_OUT`, `DVALID_OUT`, `EOL_OUT`, and `EOF_OUT`.

The `CLOCK` input must be driven with the system wide pixel clock, and the `RESET` input driven by an active-high reset signal. All clock activity is on the rising edge of the `CLOCK` input.

If the `Synchronous` generic is set to true (the default value) the result is registered internally before being output by the component. If the `Synchronous` generic is set to false the function will be implemented as a purely combinatorial function.

The data busses are all unsigned and scaled according to the generic `N`. The width of the `DATA_IN` and `K_IN` busses will equal the value `N` in bits. The `DATA_OUT` width will also be `N` bits wide.

XorK : XOR constant with each pixel

```
Entity XORK is
  Generic (
    Synchronous : boolean := True;
    N           : Integer := 32
  );
  Port (
    DATA_IN     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT    : out std_logic_vector(N - 1 downto 0);
    DVALID_IN   : in  std_logic;
    DVALID_OUT  : out std_logic;
    CLOCK       : in  std_logic;
    RESET       : in  std_logic;
    EOL_IN      : in  std_logic;
    EOL_OUT     : out std_logic;
    EOF_IN      : in  std_logic;
    EOF_OUT     : out std_logic;
    K_IN        : in  std_logic_vector(N - 1 downto 0)
  );
end XORK;
```

Using the XOrK Component

The XorK component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs `DATA_IN`, `DVALID_IN`, `EOL_IN` and `EOF_IN`.

The constant for the logical operation must be presented to the `K_IN` input.

The result is output in the Pixel Pipeline Format on the outputs `DATA_OUT`, `DVALID_OUT`, `EOL_OUT`, and `EOF_OUT`.

The `CLOCK` input must be driven with the system wide pixel clock, and the `RESET` input driven by an active-high reset signal. All clock activity is on the rising edge of the `CLOCK` input.

If the `Synchronous` generic is set to true (the default value) the result is registered internally before being output by the component. If the `Synchronous` generic is set to false the function will be implemented as a purely combinatorial function.

The data busses are all unsigned and scaled according to the generic `N`. The width of the `DATA_IN` and `K_IN` busses will equal the value `N` in bits. The `DATA_OUT` width will also be `N` bits wide.

RShK : Right shift pixel

```
Entity RSHK is
  Generic (
    Synchronous  : boolean := True;
    FillWithOnes : boolean := False;
    N            : integer := 32
  );
  Port (
    DATA_IN      : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT     : out std_logic_vector(N - 1 downto 0);
    DVALID_IN    : in  std_logic;
    DVALID_OUT   : out std_logic;
    CLOCK        : in  std_logic;
    RESET        : in  std_logic;
    EOL_IN       : in  std_logic;
    EOL_OUT      : out std_logic;
    EOF_IN       : in  std_logic;
    EOF_OUT      : out std_logic;
    K_IN         : in  std_logic_vector(N - 1 downto 0)
  );
end RSHK;
```

Using the RShK Component

The RShK component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DATA_IN, DVALID_IN, EOL_IN and EOF_IN.

The constant defining the number of places to be shifted must be presented to the K_IN input. This component can shift by up to 32 places.

The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT, and EOF_OUT.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

If the Synchronous generic is set to true (the default value) the result is registered internally before being output by the component. If the Synchronous generic is set to false the function will be implemented as a purely combinatorial function.

The FillWithOnes generic controls whether ones or zeros will be introduced to the left hand side. When set to true, ones will be introduced. When set to false, zeros will be introduced.

The data busses are all unsigned and scaled according to the generic N. The width of the DATA_IN and K_IN busses will equal the value N in bits. The DATA_OUT width will also be N bits wide.

This component takes the form of a barrel shifter. This allows the number of places to be shifted to be altered every cycle if it was so desired.

LShK : Left shift pixel

```
Entity LSHK is
  Generic (
    Synchronous  : boolean := True;
    FillWithOnes : boolean := False;
    N            : integer := 32
  );
  Port (
    DATA_IN      : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT     : out std_logic_vector(N - 1 downto 0);
    DVALID_IN    : in  std_logic;
    DVALID_OUT   : out std_logic;
    CLOCK        : in  std_logic;
    RESET        : in  std_logic;
    EOL_IN       : in  std_logic;
    EOL_OUT      : out std_logic;
    EOF_IN       : in  std_logic;
    EOF_OUT      : out std_logic;
    K_IN         : in  std_logic_vector(N - 1 downto 0)
  );
end LSHK;
```

Using the LShK Component

The LShK component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DATA_IN, DVALID_IN, EOL_IN and EOF_IN.

The constant defining the number of places to be shifted must be presented to the K_IN input. This component can shift by up to 32 places.

The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT, and EOF_OUT.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

If the Synchronous generic is set to true (the default value) the result is registered internally before being output by the component. If the Synchronous generic is set to false the function will be implemented as a purely combinatorial function.

The FillWithOnes generic controls whether ones or zeros will be introduced to the right hand side. When set to true, ones will be introduced. When set to false, zeros will be introduced.

The data busses are all unsigned and scaled according to the generic N. The width of the DATA_IN and K_IN busses will equal the value N in bits. The DATA_OUT width will also be N bits wide.

Output data results are rounded to ensure data does not exceed the maximum output value for the DATA_OUT bus. For example, with N set to 8 all internal results greater than 255 would be set to 255.

This component takes the form of a barrel shifter. This allows the number of places to be shifted to be altered every cycle if it was so desired.

Invert : Invert each pixel

```
Entity invert is
  Generic (
    Synchronous : boolean := True;
    N           : integer := 32
  );
  Port (
    DATA_IN     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT    : out std_logic_vector(N - 1 downto 0);
    DVALID_IN   : in  std_logic;
    DVALID_OUT  : out std_logic;
    CLOCK       : in  std_logic;
    RESET       : in  std_logic;
    EOL_IN      : in  std_logic;
    EOL_OUT     : out std_logic;
    EOF_IN      : in  std_logic;
    EOF_OUT     : out std_logic
  );
end invert;
```

Using the Invert Component

The Invert component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DATA_IN, DVALID_IN, EOL_IN and EOF_IN.

The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT, and EOF_OUT.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

If the Synchronous generic is set to true (the default value) the result is registered internally before being output by the component. If the Synchronous generic is set to false the function will be implemented as a purely combinatorial function.

The data busses are all unsigned and scaled according to the generic N. The width of the DATA_IN and K_IN busses will equal the value N in bits. The DATA_OUT width will also be N bits wide.

Square : Square each pixel

```
Entity SQUARE is
  Generic (
    Synchronous : boolean := True;
    N           : integer := 18
  );
  Port (
    DATA_IN     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT    : out std_logic_vector(N - 1 downto 0);
    DVALID_IN   : in  std_logic;
    DVALID_OUT  : out std_logic;
    CLOCK       : in  std_logic;
    RESET       : in  std_logic;
    EOL_IN      : in  std_logic;
    EOL_OUT     : out std_logic;
    EOF_IN      : in  std_logic;
    EOF_OUT     : out std_logic
  );
end SQUARE;
```

Using the Square Component

The Square component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DATA_IN, DVALID_IN, EOL_IN and EOF_IN.

The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT, and EOF_OUT.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

If the Synchronous generic is set to true (the default value) the result is registered internally before being output by the component. If the Synchronous generic is set to false the function will be implemented as a purely combinatorial function.

The data busses are all unsigned and scaled according to the generic N. The width of the DATA_IN and K_IN busses will equal the value N in bits. The DATA_OUT width will also be N bits wide.

Output data results are rounded to ensure data does not exceed the maximum output value for the DATA_OUT bus. For example, with N set to 8 all internal results greater than 255 would be set to 255.

The square component will instantiate an instance of the MpyK component so the same resource usage will apply for the Square component as for the MpyK component.

FillK : Fill pixel with constant

```
Entity FILLK is
  Generic (
    N          : integer := 32
  );
  Port (
    DATA_OUT   : out std_logic_vector(N - 1 downto 0);
    DVALID_IN  : in  std_logic;
    DVALID_OUT : out std_logic;
    CLOCK      : in  std_logic;
    RESET      : in  std_logic;
    EOL_IN     : in  std_logic;
    EOL_OUT    : out std_logic;
    EOF_IN     : in  std_logic;
    EOF_OUT    : out std_logic;
    K_IN       : in  std_logic_vector(N - 1 downto 0)
  );
end FILLK;
```

Using the FillK Component

The Fillk component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DVALID_IN, EOL_IN and EOF_IN. This component only uses the control signals from the pixel pipeline format data stream and replaces the data part of the data stream with the constant value.

This function is useful for testing.

The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT, and EOF_OUT.

The constant to be output must be presented to the input K_IN which is N bits wide.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

The data busses are all unsigned and scaled according to the generic N. The width of the DATA_IN and K_IN busses will equal the value N in bits. The DATA_OUT width will also be N bits wide.

FillRamp : Fill image with ramp

```
Entity FILLRAMP is
  Generic (
    N          : integer := 32
  );
  Port (
    DATA_OUT   : out std_logic_vector(N - 1 downto 0);
    DVALID_IN  : in  std_logic;
    DVALID_OUT : out std_logic;
    CLOCK      : in  std_logic;
    RESET      : in  std_logic;
    EOL_IN     : in  std_logic;
    EOL_OUT    : out std_logic;
    EOF_IN     : in  std_logic;
    EOF_OUT    : out std_logic;
    K_IN       : in  std_logic_vector(N - 1 downto 0)
  );
end FILLRAMP;
```

Using the FillRamp Component

The FillRamp component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs `DVALID_IN`, `EOL_IN` and `EOF_IN`. The data in the pixel pipeline format will be replaced by a ramp from zero up to the value dictated by the K value. This component is useful for testing.

The constant that is the limit value of the ramp must be presented to the `K_IN` input.

The result is output in the Pixel Pipeline Format on the outputs `DATA_OUT`, `DVALID_OUT`, `EOL_OUT`, and `EOF_OUT`.

The `CLOCK` input must be driven with the system wide pixel clock, and the `RESET` input driven by an active-high reset signal. All clock activity is on the rising edge of the `CLOCK` input.

The data busses are all unsigned and scaled according to the generic `N`. The width of the `DATA_IN` and `K_IN` busses will equal the value `N` in bits. The `DATA_OUT` width will also be `N` bits wide.

AddImage : Add two images

```
Entity ADDImage is
  Generic (
    N          : integer := 32
  );
  Port (
    DATA_A     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT   : out std_logic_vector(N - 1 downto 0);
    DVALID_A   : in  std_logic;
    DVALID_OUT : out std_logic;
    CLOCK      : in  std_logic;
    RESET      : in  std_logic;
    EOL_IN     : in  std_logic;
    EOL_OUT    : out std_logic;
    EOF_IN     : in  std_logic;
    EOF_OUT    : out std_logic;
    REQ_PIXEL  : out std_logic;
    DATA_B     : in  std_logic_vector(N - 1 downto 0)
  );
end ADDImage;
```

Using the AddImage Component

The AddImage component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DATA_A, DVALID_A, EOL_IN and EOF_IN.

The AddImage component also sources a second data stream from the DATA_B input. This input is intended to source its data from a FIFO or other similar storage. When a piece of data is required on the DATA_B input the REQ_PIXEL signal will be asserted. The data should be present on the DATA_B port the following cycle.

The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT and EOF_OUT.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

The data busses are all unsigned and scaled according to the generic N. The width of the DATA_A and DATA_B busses will equal the value N in bits. The DATA_OUT width will also be set to N bits.

Output data results are rounded to ensure data does not exceed the maximum output value for the DATA_OUT bus. For example, with N set to 8 all internal results greater than 255 would be set to 255.

SubImage : Subtract two images

```
Entity SUBImage is
  Generic (
    StopAtZero : boolean := True;
    N          : integer := 32
  );
  Port (
    DATA_A     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT   : out std_logic_vector(N - 1 downto 0);
    DVALID_A   : in  std_logic;
    DVALID_OUT : out std_logic;
    CLOCK      : in  std_logic;
    RESET      : in  std_logic;
    EOL_IN     : in  std_logic;
    EOL_OUT    : out std_logic;
    EOF_IN     : in  std_logic;
    EOF_OUT    : out std_logic;
    REQ_PIXEL  : out std_logic;
    DATA_B     : in  std_logic_vector(N - 1 downto 0)
  );
end SUBImage;
```

Using the SubImage Component

The SubImage component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DATA_A, DVALID_A, EOL_IN and EOF_IN.

The SubImage component also sources a second data stream from the DATA_B input. This input is intended to source its data from a FIFO or other similar storage. When a piece of data is required on the DATA_B input the REQ_PIXEL signal will be asserted. The data should be present on the DATA_B port the following cycle.

The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT, and EOF_OUT.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

The data busses are all unsigned and scaled according to the generic N. The width of the DATA_A and DATA_B busses will equal the value N in bits. The DATA_OUT width will also be N bits wide.

The StopAtZero generic (set to true by default) option will dictate what will happen if the result of the subtraction is less than one. If the result of a subtraction operation is less than zero and this option is set to true then the answer will be zero. If the result of the subtraction is less than zero and this option is set to false then the answer will wrap around.

If StopAtZero is true then 0x0002 – 0x0004 will result in 0x0000

If StopAtZero is false then 0x0002 – 0x0004 will result in 0xFFFE

MpyImage : Multiply two images

```
Entity MPYImage is
  Generic (
    N          : integer := 18
  );
  Port (
    DATA_A     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT   : out std_logic_vector(N - 1 downto 0);
    DVALID_A   : in  std_logic;
    DVALID_OUT : out std_logic;
    CLOCK      : in  std_logic;
    RESET      : in  std_logic;
    EOL_IN     : in  std_logic;
    EOL_OUT    : out std_logic;
    EOF_IN     : in  std_logic;
    EOF_OUT    : out std_logic;
    REQ_PIXEL  : out std_logic;
    DATA_B     : in  std_logic_vector(N - 1 downto 0)
  );
end MPYImage;
```

Using the MpyImage Component

The MpyImage component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DATA_A, DVALID_A, EOL_IN and EOF_IN.

The MpyImage component also sources a second data stream from the DATA_B input. This input is intended to source its data from a FIFO or other similar storage. When a piece of data is required on the DATA_B input the REQ_PIXEL signal will be asserted. The data should be present on the DATA_B port the following cycle.

The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT, and EOF_OUT.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

The data busses are all unsigned and scaled according to the generic N. The width of the DATA_A, DATA_B and DATA_OUT busses will equal the value N in bits. Output data results are rounded to ensure data does not exceed the maximum output value for the DATA_OUT bus. For example, with N set to 8 all internal results greater than 255 would be set to 255.

The multiplier component is a little more complex than the others in that the size of the input bus decides what sort of implementation will be used. The Xilinx Virtex II FPGA has built in 18 x 18 multipliers. If N is defined as being 18 or less then the MPYImage component will use these built in multipliers. If however N is defined as being greater than 18 then COREGEN IP multipliers are used instead. The two multipliers included with these components are a 24 x 24 and a 32 x 32. The reason you would not automatically want to use the largest multiplier is down to the amount of resources used. As a guide, the 24 x 24 COREGEN IP multiplier using look up tables (LUT's) uses 481 LUT's and 614 Flip Flops, 4% and 6% respectively of the total resources available on a 1M gate part. The 32 x 32 COREGEN IP multiplier uses 1080 LUT's and 1235 Flip Flops, 10% and 12% respectively of the total resources available on a 1M-gate part.

AndImage : AND two images

```
Entity ANDIMAGE is
  Generic (
    N          : integer := 32
  );
  Port (
    DATA_A     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT   : out std_logic_vector(N - 1 downto 0);
    DVALID_A   : in  std_logic;
    DVALID_OUT : out std_logic;
    CLOCK      : in  std_logic;
    RESET      : in  std_logic;
    EOL_IN     : in  std_logic;
    EOL_OUT    : out std_logic;
    EOF_IN     : in  std_logic;
    EOF_OUT    : out std_logic;
    REQ_PIXEL  : out std_logic;
    DATA_B     : in  std_logic_vector(N - 1 downto 0)
  );
end ANDIMAGE;
```

Using the AndImage Component

The AndImage component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DATA_A, DVALID_A, EOL_IN and EOF_IN.

The AndImage component also sources a second data stream from the DATA_B input. This input is intended to source its data from a FIFO or other similar storage. When a piece of data is required on the DATA_B input the REQ_PIXEL signal will be asserted. The data should be present on the DATA_B port the following cycle.

The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT, and EOF_OUT.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

The data busses are all unsigned and scaled according to the generic N. The width of the DATA_A and DATA_B busses will equal the value N in bits. The DATA_OUT width will be N bits wide.

OrImage : OR two images

```
Entity ORImage is
  Generic (
    N          : integer := 32
  );
  Port (
    DATA_A     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT   : out std_logic_vector(N - 1 downto 0);
    DVALID_A   : in  std_logic;
    DVALID_OUT : out std_logic;
    CLOCK      : in  std_logic;
    RESET      : in  std_logic;
    EOL_IN     : in  std_logic;
    EOL_OUT    : out std_logic;
    EOF_IN     : in  std_logic;
    EOF_OUT    : out std_logic;
    REQ_PIXEL  : out std_logic;
    DATA_B     : in  std_logic_vector(N - 1 downto 0)
  );
end ORImage;
```

Using the OrImage Component

The OrImage component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DATA_A, DVALID_A, EOL_IN and EOF_IN.

The OrImage component also sources a second data stream from the DATA_B input. This input is intended to source its data from a FIFO or other similar storage. When a piece of data is required on the DATA_B input the REQ_PIXEL signal will be asserted. The data should be present on the DATA_B port the following cycle.

The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT, and EOF_OUT.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

The data busses are all unsigned and scaled according to the generic N. The width of the DATA_A and DATA_B busses will equal the value N in bits. The DATA_OUT width will be N bits wide.

XorImage : XOR two images

```
Entity XORImage is
  Generic (
    N          : integer := 32
  );
  Port (
    DATA_A     : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT   : out std_logic_vector(N - 1 downto 0);
    DVALID_A   : in  std_logic;
    DVALID_OUT : out std_logic;
    CLOCK      : in  std_logic;
    RESET      : in  std_logic;
    EOL_IN     : in  std_logic;
    EOL_OUT    : out std_logic;
    EOF_IN     : in  std_logic;
    EOF_OUT    : out std_logic;
    REQ_PIXEL  : out std_logic;
    DATA_B     : in  std_logic_vector(N - 1 downto 0)
  );
end XORImage;
```

Using the XorImage Component

The XorImage component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DATA_A, DVALID_A, EOL_IN and EOF_IN.

The XorImage component also sources a second data stream from the DATA_B input. This input is intended to source its data from a FIFO or other similar storage. When a piece of data is required on the DATA_B input the REQ_PIXEL signal will be asserted. The data should be present on the DATA_B port the following cycle.

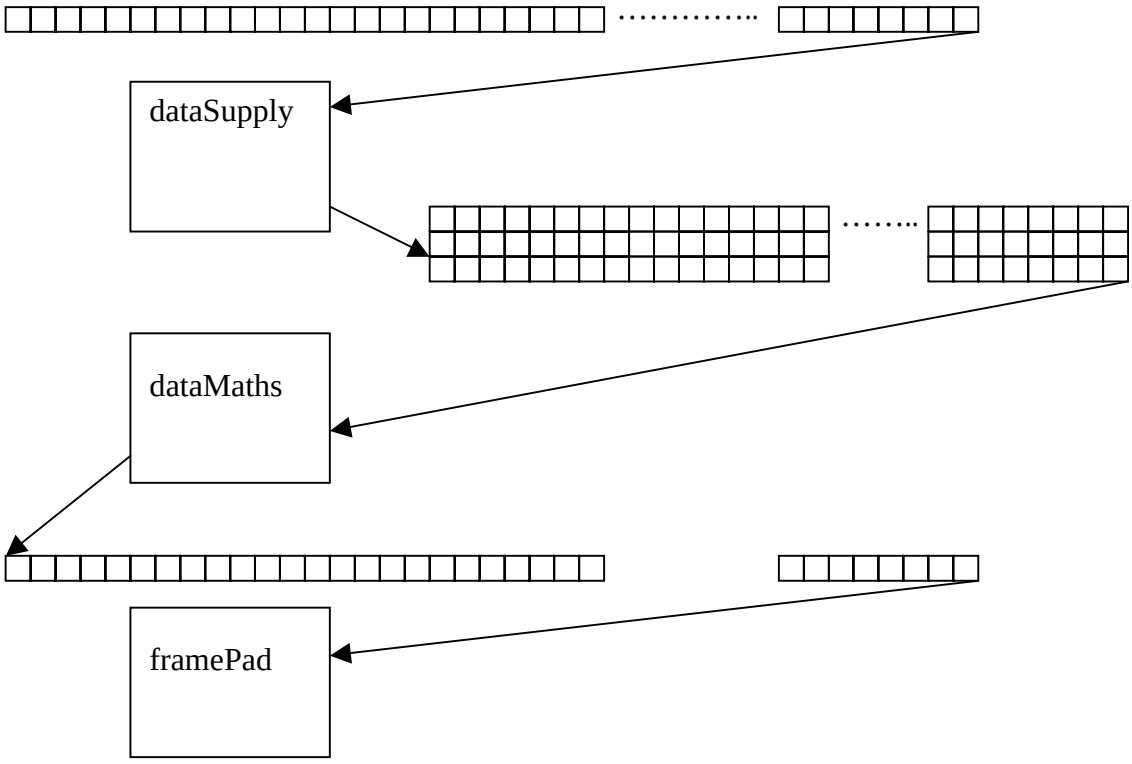
The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT, and EOF_OUT.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

The data busses are all unsigned and scaled according to the generic N. The width of the DATA_A and DATA_B busses will equal the value N in bits. The DATA_OUT width will be N bits wide.

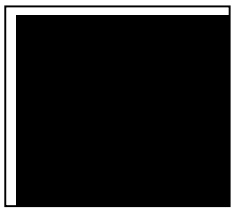
Convolution Functions

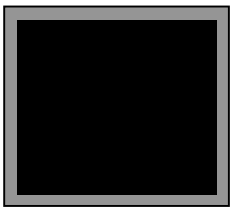
The convolution functions perform convolution on a frame of data presented to them. A frame is presented by a camera in a line-by-line format. This means that a way of storing lines of data is needed as we need at least 2 lines and 3 pixels worth of data before we are able to perform the first correct maths. The Virtex II FPGA's contain large amounts of Block RAM that is ideal for this purpose. The CONVOLVE component is made up of 3 sub components. These are called dataSupply, dataMaths and framePad. The function of dataSupply is to take a stream of data and present it correctly to the dataMaths component. The data will arrive at the dataSupply in the format Line1 pixel1, line1 pixel2…. The dataSupply component will then manipulate this data and output the appropriate nine pixel values at once.



The format of the data out of the dataMaths component will be back into a data stream of single pixels. Through the previous two components the control signals have been passed through unaffected other than being delayed by the latency of the component. Due to the nature of convolution a pixel will be lost from every edge of the frame. This means that the pixels whose value is valid looks like that shown on the left below where the outer box shows the frame indicated by the control signals but the inner filled box shows the position of the valid data. The framePad component shifts the valid data to that shown in the right hand diagram. It will also

add a pixel of value 0 to replace the lost pixels, indicated by the grey area in the diagram below right.





Convolve : Perform convolution with 3x3 window

```
Entity Convolve is
  Generic(
    N              : integer := 32;
    LINE_LENGTH    : integer := 32;
    RAM_ADDR_WIDTH : integer := 5;
    KERNEL         : string  := "GENERAL"
  );
  Port (
    DATA_IN    : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT   : out std_logic_vector(N - 1 downto 0);
    DVALID_IN  : in  std_logic;
    DVALID_OUT : out std_logic;
    EOL_IN     : in  std_logic;
    EOL_OUT    : out std_logic;
    EOF_IN     : in  std_logic;
    EOF_OUT    : out std_logic;
    K_A1       : in  std_logic_vector(17 downto 0);
    K_A2       : in  std_logic_vector(17 downto 0);
    K_A3       : in  std_logic_vector(17 downto 0);
    K_B1       : in  std_logic_vector(17 downto 0);
    K_B2       : in  std_logic_vector(17 downto 0);
    K_B3       : in  std_logic_vector(17 downto 0);
    K_C1       : in  std_logic_vector(17 downto 0);
    K_C2       : in  std_logic_vector(17 downto 0);
    K_C3       : in  std_logic_vector(17 downto 0);
    SHIFT      : in  std_logic_vector(3 downto 0);
    CLOCK      : in  std_logic;
    RESET      : in  std_logic
  );
end Convolve;
```

Using the Convolve Component

The Convolve component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs DATA_A, DVALID_IN, EOL_IN and EOF_IN.

The result is output in the Pixel Pipeline Format on the outputs DATA_OUT, DVALID_OUT, EOL_OUT, and EOF_OUT.

In the case of a general convolution (KERNEL="GENERAL") the constants are input via the K inputs K_A1, K_A2, K_A3, K_B1, K_B2, K_B3, K_C1, K_C2, and K_C3.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

The N generic option sets the data bus width used. The output data bus is the same size as the input data bus and that size will be N bits wide.

The generic LINE_LENGTH sets the number of pixels in a line.

RAM_ADDR_WIDTH is the generic that defines the number of bits used to address the Block RAM that is used to store lines of data. For example, if a line is 32 pixels long then 5 bits will be required so RAM_ADDR_WIDTH should be set to 5. If it were 512 pixels long then 10 bits would be required so RAM_ADDR_WITH should be 10. If however the number of pixels

in a line is not a power 2 then the number of bits to address the block RAM should be rounded up e.g. if the line length is 384 then this option should be set to 9. Setting this option too high will not cause the component to fail but will waste Block RAM resources.

The KERNEL option describes the operation that is required of the Convolve component. There are 8 options for this generic, GENERAL, vSobel, hSobel, vPrewitt, hPrewitt, Laplacian, LPFilter, and Sharpen.

Each option defines a specific kernel operator. For the GENERAL case, the co-efficients used are those supplied through the K_** component inputs. This option will take an input data bus up to 18 bits and use Virtex II hardware multipliers to multiply each cell by the constant given by the inputs K_A1 to K_C3. The pixels in the 3x3 Kernel are referenced as follows:

A1	B1	C1
A2	B2	C2
A3	B3	C3

The other Kernel options will perform convolution based upon optimised operations. That is, where the use of a dedicated multiplier can be saved, a simple shift will be used to implement the co-efficient operation. That is, multiplication can be performed by shifting data. For example, a shift left by one is the same as a multiplication by two.

Kernel Type			
General	K_A1	K_B1	K_C1
	K_A2	K_B2	K_C2
	K_A3	K_B3	K_C3
vSobel	1	0	-1
	2	0	-2
	1	0	-1
hSobel	1	2	1
	0	0	0
	-1	-2	-1
vPrewitt	1	0	-1
	1	0	-1
	1	0	-1
hPrewitt	1	1	1
	0	0	0
	-1	-1	-1

Laplacian	-1	-1	-1
	-1	8	-1
	-1	-1	-1
LP Filter (requires shift right by 4)	1	2	1
	2	4	2
	1	2	1
Sharpen (requires shift left by 3)	-1	-1	-1
	-1	16	-1
	-1	-1	-1

Some of these Kernels require a shift to correct any offset introduced. This can be achieved by setting the value of the shift input. The data can be shifted by up to 4 places either left or right.

The SHIFT input is a 2's complement signed value, where a positive number is a left shift, and a negative number is a right shift. Therefore, to shift left by one place, SHIFT would be set to +1 "0001". To shift right by one place, SHIFT would be set to –1 "1111".

Convolve Frame Size and Latency

Due to the nature of convolution two pixels will be lost from each dimension of a frame. To compensate for this the frameControl component within the Convolve component will add an extra line of pixels to the top and bottom of each frame and a pixel to the start and end of each line. These pixels have the value of zero. The end result of this operation is that the output frame is the exact same size as the input frame.

The latency of this component is 2 lines + 2 pixels + 1 clock cycle. This is because it is not possible to perform the necessary maths until this minimum amount of data has past.

Convolve K Values

The inputs to the convolve component named K_** must be in two's complement format.

Convolve Input and Output Data Values

The input data values must be unsigned. The Convolve component will present unsigned data values for the result of the convolution.

After the results are shifted according to the amount specified by SHIFT input, the result is rounded. All negative results are set to zero and all results greater than the output data bus maximum value are set to the maximum.

For example, with N set to 8 all internal results greater than 255 would be set to 255, and all negative results would be set to 0.

Convolve5x5 : Perform convolution with 5x5 window

```
Entity Convolve5x5 is
  Generic(
    N              : integer := 8;
    LINE_LENGTH    : integer := 384;
    RAM_ADDR_WIDTH : integer := 5;
    KERNEL         : string  := "GENERAL"
  );
  Port (
    DATA_IN    : in  std_logic_vector(N - 1 downto 0);
    DATA_OUT   : out std_logic_vector(N - 1 downto 0);
    DVALID_IN  : in  std_logic;
    DVALID_OUT : out std_logic;
    EOL_IN     : in  std_logic;
    EOL_OUT    : out std_logic;
    EOF_IN     : in  std_logic;
    EOF_OUT    : out std_logic;
    K_A1       : in  std_logic_vector(17 downto 0);
    K_A2       : in  std_logic_vector(17 downto 0);
    K_A3       : in  std_logic_vector(17 downto 0);
    K_A4       : in  std_logic_vector(17 downto 0);
    K_A5       : in  std_logic_vector(17 downto 0);
    K_B1       : in  std_logic_vector(17 downto 0);
    K_B2       : in  std_logic_vector(17 downto 0);
    K_B3       : in  std_logic_vector(17 downto 0);
    K_B4       : in  std_logic_vector(17 downto 0);
    K_B5       : in  std_logic_vector(17 downto 0);
    K_C1       : in  std_logic_vector(17 downto 0);
    K_C2       : in  std_logic_vector(17 downto 0);
    K_C3       : in  std_logic_vector(17 downto 0);
    K_C4       : in  std_logic_vector(17 downto 0);
    K_C5       : in  std_logic_vector(17 downto 0);
    K_D1       : in  std_logic_vector(17 downto 0);
    K_D2       : in  std_logic_vector(17 downto 0);
    K_D3       : in  std_logic_vector(17 downto 0);
    K_D4       : in  std_logic_vector(17 downto 0);
    K_D5       : in  std_logic_vector(17 downto 0);
    K_E1       : in  std_logic_vector(17 downto 0);
    K_E2       : in  std_logic_vector(17 downto 0);
    K_E3       : in  std_logic_vector(17 downto 0);
    K_E4       : in  std_logic_vector(17 downto 0);
    K_E5       : in  std_logic_vector(17 downto 0);
    SHIFT      : in  std_logic_vector(3 downto 0);
    CLOCK      : in  std_logic;
    RESET      : in  std_logic
  );
end Convolve5x5;
```

Using the Convolve5x5 Component

The Convolve5x5 component takes a stream of camera data in Pixel Pipeline Format. This stream must be presented to the inputs `DATA_A`, `DVALID_IN`, `EOL_IN` and `EOF_IN`.

The result is output in the Pixel Pipeline Format on the outputs `DATA_OUT`, `DVALID_OUT`, `EOL_OUT`, and `EOF_OUT`.

In the case of a general convolution (KERNEL="GENERAL") the constants are input via the K inputs K_A1, K_A2, K_A3, K_A4, K_A5, K_B1, K_B2, K_B3, K_B4, K_B5, K_C1, K_C2, K_C3, K_C4, K_C5, K_D1, K_D2, K_D3, K_D4, K_D5, K_E1, K_E2, K_E3, K_E4, K_E5.

The CLOCK input must be driven with the system wide pixel clock, and the RESET input driven by an active-high reset signal. All clock activity is on the rising edge of the CLOCK input.

The N generic option sets the data bus width used. The output data bus is the same size as the input data bus and that size will be N bits wide.

The generic LINE_LENGTH sets the number of pixels in a line.

RAM_ADDR_WIDTH is the generic that defines the number of bits used to address the Block RAM that is used to store lines of data. For example, if a line is 32 pixels long then 5 bits will be required so RAM_ADDR_WIDTH should be set to 5. If it were 512 pixels long then 10 bits would be required so RAM_ADDR_WITH should be 10. If however the number of pixels in a line is not a power 2 then the number of bits to address the Block RAM should be rounded up e.g. if the line length is 384 then this option should be set to 9. Setting this option to high will not cause the component to fail but will waste Block RAM resources.

The KERNEL option describes the operation that is required of the Convolve5x5 component. There are 4 options for this generic, GENERAL, vSobel, hSobel, and Laplacian.

Each option defines a specific kernel operator. For the GENERAL case, the co-efficients used are those supplied through the K_** component inputs. option will take an input data bus up to 18 bits and use Virtex II hardware multipliers to multiply each cell by the constant given by the inputs K_A1 through to K_E5. The pixels in the 5x5 Kernel are referenced as follows:

A1	B1	C1	D1	E1
A2	B2	C2	D2	E2
A3	B3	C3	D3	E3
A4	B4	C4	D4	E4
A5	B5	C5	D5	E5

The other Kernel options will perform convolution based upon optimised operations. That is, where the use of a dedicated multiplier can be saved, a simple shift will be used to implement the co-efficient operation. That is, multiplication can be performed by shifting data. For example, a shift left by one is the same as a multiplication by two.

Kernel Type					
General	K_A1	K_B1	K_C1	K_D1	K_E1
	K_A2	K_B2	K_C2	K_D2	K_E2
	K_A3	K_B3	K_C3	K_D3	K_E3
	K_A4	K_B4	K_C4	K_D4	K_E4
	K_A5	K_B5	K_C5	K_D5	K_E5
vSobel	1	2	0	-2	-1
	2	3	0	-3	-2
	3	4	0	-4	-3
	2	3	0	-3	-2
	1	2	0	-2	-1
hSobel	1	2	3	2	1
	2	3	4	3	2
	0	0	0	0	0
	-2	-3	-4	-3	-2
	-1	-2	-3	-2	-1
Laplacian	-1	-3	-4	-3	-1
	-3	0	6	0	-3
	-4	6	20	6	-4
	-3	0	6	0	-3
	-1	-3	-4	-3	-1

The SHIFT input is a 2's complement signed value, where a positive number is a left shift, and a negative number is a right shift. Therefore, to shift left by one place, SHIFT would be set to "0001". To shift right by one place, SHIFT would be set to "1111".

Convolve Frame Size and Latency

Due to the nature of 5x5 convolution four pixels will be lost from each dimension of a frame. To compensate for this the frameControl component within the Convolve5x5 component will add two extra lines of pixels to the top and bottom of each frame and two extra pixels to the start and end of each line. These pixels have the value of zero. The end result of this operation is that the output frame is the exact same size as the input frame.

The latency of this component is 4 lines + 5 pixels + 1 clock cycle. This is because it is not possible to perform the necessary maths until this minimum amount of data has past.

Convolve K Values

The inputs to the convolve component K_** must be in two's complement format.

Convolve Input and Output Data Values

The input data values must be unsigned. The Convolve5x5 component will present unsigned data values for the result of the convolution.

After the results are shifted according to the amount specified by SHIFT input, the result is rounded. All negative results are set to zero and all results greater than the output data bus maximum value are set to the maximum.

For example, with N set to 8 all internal results greater than 255 would be set to 255, and all negative results would be set to 0.

Implementing Larger Convolutions

Convolutions with a Kernel larger than 5x5 are unusual, but of course can easily be implemented in VHDL. Starting from the VHDL of one of the convolutions supplied you would need to change the following things:

The Convolve and Convolve5x5 components described above are each made up from three sub-components called `dataSupply`, `dataMaths` and `framePad`.

In implementing a larger convolution the `dataSupply` component must be changed to provide a larger amount of storage of lines of camera data. This component must provide enough storage for N lines of the image, where N is the depth of the Kernel. For example, the Convolve component (3x3) provides enough storage for 3 lines of an image, and the Convolve5x5 component provides enough storage for 5 lines. In controlling how this memory is used, a simple state machine is needed that automatically fills one Block RAM after another for each line of camera data passed into the component.

The `dataMaths` component is the heart of the convolution and needs to be modified to perform the same basic mathematical operation on a larger number of inputs. As the number of multiplies increases, you will need to pipeline the internal operation of the maths component to still meet your operating time constraints. Also, you may need to share dedicated multipliers as the number of multiplications increases. Of course, where you are able to perform the necessary multiplications with a simple shift operation this will allow you to save on dedicated multipliers.

The `framePad` component controls the size mismatch between the input data for the convolution and the output data. When a convolution is performed on an image, the output frame will have less lines and less pixels depending on the size of the kernel. For example, with a 3x3 kernel, the outside edge is lost around the entire image. This means that a 512x512 image would become a 510x510 image. For a 5x5 kernel, two pixels are lost along each edge, so a 512x512 image would become a 508x508 image. The framePad component however must pad out these edges so the final output image is the same size as the original.

This padding operation requires the framePad component to add black pixels (pixel value of 0) along all edges of the image to fill out the size to match that of the input image. This is done with state machine that adds black pixels, and ensures the correct generation of pixel-pipeline-format end-of-line markers and end-of-frame markers.

When to use off chip memory

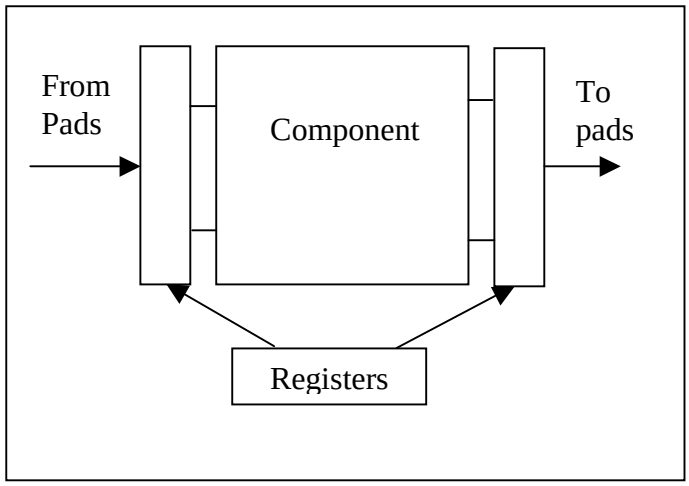
There is a separate application note “Using external SDRAM for image processing with FPGAs” that discusses when and how you might need to use external memory for convolutions.

The following tables give the Block RAM and hardware multiplier resources used by each component.

Component	Block RAMs Used	Hardware Multipliers Used
AddK	0	0
SubK	0	0
Invert	0	0
AndK	0	0
OrK	0	0
XorK	0	0
MpyK 18-bit i/ps	0	1
MpyK 32-bit i/ps	0	4
LShK	0	0
RShK	0	0
FillK	0	0
FillRamp	0	0
AddImage	0	0
SubImage	0	0
AndImage	0	0
OrImage	0	0
XorImage	0	0
MpyImage 18-bit i/ps	0	1
MpyImage 32-bit i/ps	0	4

Component	Block RAMs used	Hardware Multipliers Used
Convolve (General)	3 (up to 2K line length) 6 (up to 4K line length)	9
Convolve (vSobel)	3 (up to 2K line length) 6 (up to 4K line length)	0
Convolve (hSobel)	3 (up to 2K line length) 6 (up to 4K line length)	0
Convolve (vPrewitt)	3 (up to 2K line length) 6 (up to 4K line length)	0
Convolve (hPrewitt)	3 (up to 2K line length) 6 (up to 4K line length)	0
Convolve (Laplacian)	3 (up to 2K line length) 6 (up to 4K line length)	0
Convolve (LPFilter)	3 (up to 2K line length) 6 (up to 4K line length)	0
Convolve (Sharpen)	3 (up to 2K line length) 6 (up to 4K line length)	0
Convolve5x5 (General)	5 (up to 2K line length) 10 (up to 4K line length)	25
Convolve5x5 (vSobel)	5 (up to 2K line length) 10 (up to 4K line length)	0
Convolve5x5 (hSobel)	5 (up to 2K line length) 10 (up to 4K line length)	0
Convolve5x5 (Laplacian)	5 (up to 2K line length) 10 (up to 4K line length)	0

The performance of each component is extremely implementation specific. As a guide the components were each instantiated in a Virtex-II xc2v1000–4 FPGA with the surrounding architecture as shown below. The table contains the results obtained. These results give a good idea of the components performance relative to each other but the exact speed is an ideal for the specific silicon it was built for.



FPGA

Component	Asynchronous Speed (MHz)	Synchronous Speed (MHz)
AddK	570	513
SubK	570	513
Invert	570	513
AndK	570	513
OrK	570	513
XorK	570	513
MpyK	Restricted by mult18x18 delays	Restricted by mult18x18 delays
LShK	570	513
RShK	570	513
FillK	570	513
FillRamp	570	513

Component	Asynchronous Speed (MHz)	Synchronous Speed (MHz)
AddImage	N/A	453
SubImage	N/A	453
AndImage	N/A	453
OrImage	N/A	453
XorImage	N/A	453
MpyImage	N/A	Restricted by mult18x18 delays
Convolve (General)	N/A	Restricted by mult18x18 delays
Convolve (vSobel)	N/A	202
Convolve (hSobel)	N/A	202
Convolve (vPrewitt)	N/A	202
Convolve (hPrewitt)	N/A	202
Convolve (Laplacian)	N/A	202
Convolve (LPFilter)	N/A	202
Convolve (Sharpen)	N/A	202
Convolve5x5 (General)	N/A	Restricted by mult18x18 delays
Convolve5x5 (vSobel)	N/A	183
Convolve5x5 (hSobel)	N/A	183
Convolve5x5 (Laplacian)	N/A	183

The following tables give the default settings for each component, along with the latency through each component and whether an asynchronous version is supported.

Component	Latency (Synchronous)	Default Input Bus Width	Default Output Bus Width	Asynchronous Version
AddK	1	32	32	Yes
SubK	1	32	32	Yes
Invert	1	32	32	Yes
AndK	1	32	32	Yes
OrK	1	32	32	Yes
XorK	1	32	32	Yes
MpyK	2 (18x18) 6 (24x24) 6 (32x32)	18	18	Yes
LShK	1	32	32	Yes
RShK	1	32	32	Yes
FillK	0	32	32	No
FillRamp	0	32	32	No
AddImage	2	32	32	No
SubImage	2	32	32	No
AndImage	2	32	32	No
OrImage	2	32	32	No
XorImage	2	32	32	No
MpyImage	3 (18x18) 7 (24x24) 7 (32x32)	18	18	No
Convolve	2 lines + 3 pixels + 1 cycle	32	32	No
Convolve5x5	4 lines + 5 pixels + 1 cycle	32	32	No

Technical support for HUNT ENGINEERING products should first be obtained from the comprehensive Support section www.hunteng.co.uk/support/index.htm on the HUNT ENGINEERING web site. This includes FAQs, latest product, software and documentation updates etc. Or contact your local supplier - if you are unsure of details please refer to www.hunteng.co.uk for the list of current re-sellers.

HUNT ENGINEERING technical support can be contacted by emailing support@hunteng.demon.co.uk, calling the direct support telephone number +44 (0)1278 760775, or by calling the general number +44 (0)1278 760188 and choosing the technical support option.

If you are in North America, South America or Canada, contact our strategic partner Traquair Data Systems at www.traquair.com/company/support.html for support information and contact details.

N.B. Technical support for the Image Processing VHDL source modules is provided for users of HUNT ENGINEERING hardware ONLY.