XAPP953 (v1.1) September 21, 2006

Two-Dimensional Rank Order Filter

Author: Gabor Szedo

Summary

This application note describes the implementation of a two-dimensional Rank Order filter. The reference design includes the RTL VHDL implementation of an efficient sorting algorithm. The design is parameterizable for input/output precision, color standards, filter kernel size, maximum horizontal resolution, and implementation options. The rank to be selected can be modified dynamically, and the actual horizontal resolution is picked up automatically from the input synchronization signals. The design has a fully synchronous interface through the ce, clk, and rst ports.

Introduction

Rank order filtering is a class of operators that use neighborhood pixels to perform comparisons and ranking. The median filter, a sub-class of the rank order filter [Ref 1][Ref 2] [Ref 3], sorts the pixels in a region by luminance, finds the median value and replaces the central pixel with that value. Used to remove noise from images, this operation completely eliminates extreme values from the image. Rank operations also include the maximum and minimum operators, which find the brightest or darkest pixels in each neighborhood and place that value into the central pixel. By loose analogy to the erosion and dilation operations on binary images, these are sometimes called grey scale erosion and dilation [Ref 4].

One important variable in the use of a rank operator is the size of the neighborhood. Generally, rectangular (for convenience of computation) or circular (to minimize directional effects) shapes are used. As the size of the neighborhood is increased, however, the computational effort in performing the ranking increases rapidly. Also, these ranking operations cannot be easily programmed into specialized hardware, such as array processors, or programmable DSP processors [Ref 5][Ref 6][Ref 7][Ref 8].

Rank order filtering or Median filtering is used extensively in smoothing and de-noising applications for images and video [Ref 9]. It is a cost-effective solution used predominantly in video pre- and post-processing systems. It is also deployed extensively in real-time vision systems and automatic target recognition (ATR) systems [Ref 10].

Pinout

Figure 1 shows the rank_2d symbol. Table 1 provides the symbol pinout and descriptions.

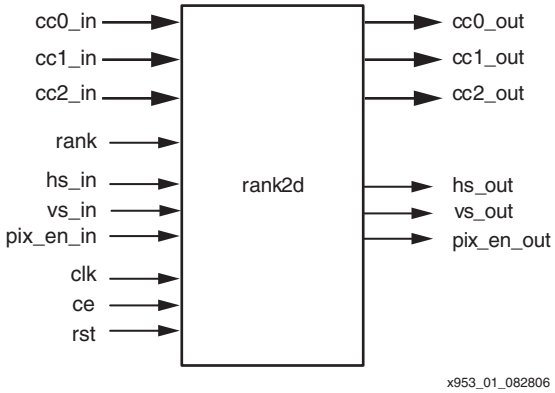

Figure 1: **rank_2d Symbol**

Table 1: **Symbol Pinout**

Port Name	Port Width	Direction	Description
cc0_in	DATA_WIDTH_CH0	Input	Color Channel input 0 (R for RGB, Y for YUV or YCrCb)
cc1_in	DATA_WIDTH_CH1	Input	Color Channel input 1 (G for RGB, U for YUV, Cr for YCrCb)
cc2_in	DATA_WIDTH_CH2	Input	Color Channel input 2 (B for RGB, V for YUV, Cb for YCrCb)
cc0_out	DATA_WIDTH_CH0	Output	Color Channel output 0 (R for RGB, Y for YUV or YCrCb)
cc1_out	DATA_WIDTH_CH1	Output	Color Channel output 1 (G for RGB, U for YUV, Cr for YCrCb)
cc2_out	DATA_WIDTH_CH2	Output	Color Channel output 2 (B for RGB, V for YUV, Cb for YCrCb)
rank	$\log_2(WH_{virt}*WW)$[1]	Input	Designates which sample to select from the ordered list
hs_in	1	Input	Horizontal Sync input
vs_in	1	Input	Vertical Sync input
pix_en_in	1	Input	Pixel Enable input
hs_out	1	Output	Horizontal Sync output
vs_out	1	Output	Vertical Sync output
pix_en_out	1	Output	Pixel Enable output
clk	1	input	System clock
ce	1	input	Clock Enable
rst	1	input	Synchronous Clear Input

Notes:

1. WW is the width, $WH_{virt}(7)$ is the vertical size of the virtual filter kernel. See Figure 6.

Signal Descriptions

rst - Synchronous Clear

Pulling rst High results in resetting all internal registers and keeps pix_en_out Low until valid samples are available on the output channels (cc0_out, cc1_out, and cc2_out). Output channels (cc0_out, cc1_out, and cc2_out) are not cleared. Previous pixels might appear on the output; however, these pixels are invalidated by pix_en_out = 0.

ce - Clock Enable

Pulling ce Low suspends all operations of the design. Input signals are not sampled, except for reset (rst takes precedence over ce).

cc0_in, cc1_in, cc2_in - Data inputs

Input pixels are presented to the reference design through the color channel inputs (cc0_in, cc1_in, cc2_in). The reference design caters to RGB, YUV, and YCrCb color representations. The width of channel inputs should be positive (non-zero) integers. For RGB representation, connect channel R to cc0_in, channel G to cc1_in, and channel B to cc2_in. For YUV or YCrCb signals connect channel Y to cc0_in, U or Cr to cc1_in, V or Cb to cc2_in.

pix_en_in - Pixel Enable Input

Input pixels are validated by pix_en_in. When pix_en_in is Low and ce is High, the sorting core of the filter keeps working. However, no new pixels are latched into the line buffer. Pix_en_in basically facilitates working with a core clock rate higher than of the pixel clock rate, as the filter core might need multiple clock cycles to process input pixels. The duty cycle of pix_en_in has to be set so the filter core has sufficient extra cycles to perform sorting.

hs_in - Horizontal Sync Input

A strobe on this input signals the beginning of a new line. As the two-dimensional filter operates on multiple lines, it is crucial that pixels in the same column are aligned. Pixel rows are stored in programmable length line-buffers, which are concatenated at the end of each line, deduced from hs_in signal.

vs_in - Horizontal Sync Input

A strobe on this input signals the beginning of a new frame. This signal is necessary to avoid carrying forward information from one frame to the next. After a pulse is detected on vs_in, the contents of the line buffer are invalidated.

cc0_out, cc1_out, cc2_out - Data Outputs

Output pixels are presented on the color channel outputs (cc0_out, cc1_out, and cc2_out) in a fashion similar to the data inputs. The width of channel outputs equal those of the corresponding input channels.

pix_en_out – Pixel Enable Output

Output pixels are validated by pix_en_out.

hs_out – Horizontal Sync Output

A strobe on this output indicates the beginning of a new output line. This signal is a delayed version of hs_in.

vs_out – Horizontal Sync Output

A strobe on this output indicates the beginning of a new output frame. This signal is a delayed version of vs_in.

Generic Parameters

The design parameters are listed in Table 2.

Table 2: **Design Parameters**

Name	Type	Range	Description
DATA_WIDTH_CH0	Integer	1 to 16	Bit width of color channel 0.
DATA_WIDTH_CH1	Integer	1 to 16	Bit width of color channel 1.
DATA_WIDTH_CH2	Integer	1 to 16	Bit width of color channel 2.
DATA_WIDTH_FILTER	Integer	4 to 24	Width of the magnitude value, generated from the 3 color. channel values, on which sorting is performed.
WINDOW_WIDTH	Integer	3 to 9	Horizontal size of filter kernel.
WINDOW_HEIGHT	Integer	3 to 9	Vertical size of filter kernel.
NEW_INPUTS	Integer	1 to WINDOW_HEIGHT	Number of pixels entered into the filter kernel from the line buffers per clk cycle.
MAX_HORIZONTAL_RES	Integer	2048	Maximum length of a scan line. This value controls line buffer memory allocation. The actual horizontal resolution is controlled by the horizontal sync signals.
Y_GENERATOR_TYPE(1)	Integer	0 to 2	0: magnitude value = cc0+cc1+cc2. 1: magnitude value = 0.51*cc0+cc1+0.19*cc2 2: magnitude value = cc0
FAMILY	String	-	Spartan™-3, Spartan-3E, Virtex™-II, Virtex-II Pro, Virtex-4, Virtex-5

Notes:

1. See "Basic Filter Architecture" for more information.

Theory of Operation

Rank order filtering is a non-linear filtering technique which orders the contents of a filter kernel and selects the sample indexed by rank from the magnitude ordered samples. In the two-dimensional (2D) case, contents of a two-dimensional window (which slides across the image) are filtered. Every time the window is shifted by one pixel, a set of obsolete pixels are discarded and a set of new pixels are inserted. The samples within the window are sorted and the element with the specified rank replaces the center element of the window in the output. Most typical ranks are median, minimum, and maximum.

Compared to linear filters, such as FIR or IIR, rank filters can effectively remove impulse-like noises while preserving the edges of the original image. This can be very useful for various applications, such as pre-processing before edge detection or removing certain types of transmission noises.

The hardware architecture presented here is tailored to high performance color video processing.

Architecture

Let *TAP=WW·WH* denote the number of taps, where *WW (*WINDOW_WIDTH*)* and *WH* (WINDOW_HEIGHT) are the vertical and the horizontal size of the filter window. Also, let *DW* (DATA_WIDTH) denote the width of the complete per-pixel information (e.g., R, G, and B values), and *DWF* (DATA_WIDTH_FILTER) denote the width of the data used for ordering. This value can be any function of the complete pixel information. For instance, many applications may find using luminance (Y) useful for ordering pixels. This is trivial when the input is in the YC_bC_r or YUV color space; otherwise, it can be easily derived from RGB components.

The most important difference between 2D and 1D rank filtering beyond larger tap numbers is *WH* input image samples are inserted into the 2D filter core for each new output sample. This reference design targets applications with *rectangular filter kernel*s, scanning the image left to right, top to bottom. Figure 2 presents the sliding window moving across the input image from left to right. Smaller, grey squares represent pixels. The light grey block outlined with white correspond to the current filter window; the black dotted line outlines the next window. Darker squares illustrate new pixels entering the kernel.

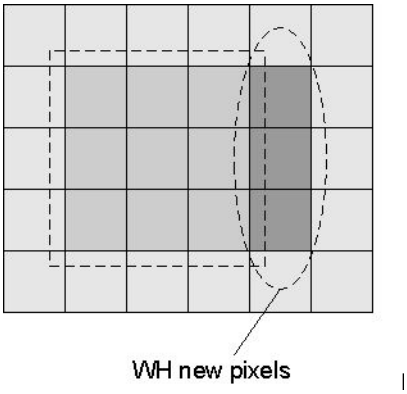

Figure 2: **2D Filtering Window**

1D filters can be trivially extended to 2D by operating the filter at *WH* multiple of the pixel clock, reading new input pixels every clock cycle, but generating valid output pixels only once in every *WH* clock cycle. Depending on the filter size and the targeted FPGA family, this solution is viable for a wide range of applications.

If pixel clock frequencies are prohibitively high to run the filter core at a multiple of the pixel clock frequency, parallel instances of some key filter components can be used so the filter may accept *WH* number of new input samples every clock cycle. However, for most applications a fully parallel implementation is suboptimal due to inefficient resource utilization.

Hybrid solutions spanning between fully parallel (*WH* input samples per clock cycle) and word serial (one input sample per clock cycle) allow tuning the filter core to the maximum clock frequency allowed by the target chip while minimizing resource counts. From the vertical size of the filter window (*WH*), the sampling (pixel) frequency of the input (*FS*), and the number of new input samples (*NI*) the required operating frequency of the filter core (FO_{max}) can be determined:

$$FO_{max} = FS\frac{WH}{NI} \qquad \text{Equation 1}$$

Basic Filter Architecture

The architecture consists of five main components, illustrated on Figure 3. The Line Buffer stores *WH*-1 lines of the input frame. If required, the Y Generator computes magnitude values, such as luminance, from RGB for magnitude ordering. The Delay Line block stores full pixel information (all 3 color components) for the pixels currently being processed by the Filter Core, which does the actual rank filtering. The Control block generates optional data switching, masking and output valid signals.

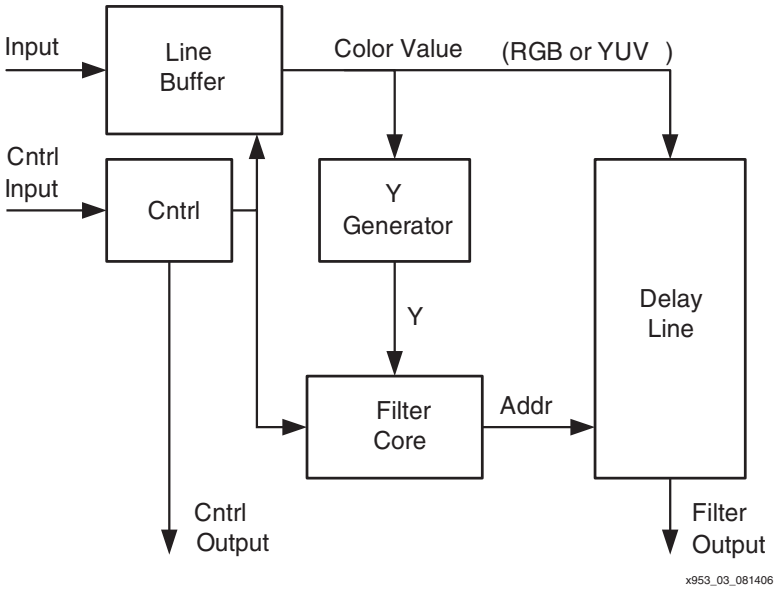

Figure 3: **Filter Architecture**

The Line Buffer is implemented using block RAMs. For example, HDTV resolution and 7x7 filter window size requires:

$$1920*(7 - 1)*3 = 34560 \qquad \textit{Equation 2}$$

bytes to be stored (assuming 3 bytes/pixel), which can fit into 17 block RAMs.

Color frames are usually not filtered using the full RGB information, but a function of the RGB values, typically luminance, which allocates fewer bits than the RGB values. Therefore, color information for the pixels being processed by the Filter Core is stored in a FIFO, so data paths within the Filter Core can be streamlined. The Filter Core produces an appropriate address for this FIFO to access full pixel (RGB) information. The number of pixels the FIFO stores is the sum of the filter size (*TAP*) and the latency of the Filter Core. SRL16/SRL32 primitives in Xilinx FPGAs offer an efficient way to implement this addressable FIFO with minimal hardware resources.

The Y Generator is an optional hardware module required only when the input format is not suitable for direct magnitude filtering. For YC_bC_r or YUV input representations, this module can be omitted as the Y component lends itself well for magnitude ordering. For RGB input, luminance, a typical magnitude value can be calculated as

$$Y' = 0.299\ R + 0.587\ G + 0.114\ B \qquad \textit{Equation 3}$$

As the Y information is used only to order the pixels, to simplify calculations (i.e., to save a multiplier) the actual Y value used is

$$Y = 0.50989\ R + G + 0.19421\ B \qquad \textit{Equation 4}$$

The complexity and latency of this module does not change the filter architecture at all, so arbitrary algorithms can be used from simple summation of the RGB component values to true color space conversion. The number of Y Generator modules used is the same as the number of new input samples (NI).

Filter Core

The operation of the Filter Core is based on observations introduced in [Ref 11]. To illustrate the theory of operation assume the filter contains *TAP* number of different samples. Each sample in the filter core is coupled with an index value representing the number of samples not smaller than the corresponding sample. When a new sample is inserted into the window, the samples already in the filter are compared to the new sample. Based on the comparisons, the index values are updated, resulting in *TAP* distinct values ranging from 1 (smallest sample) to *TAP* (largest sample) at all times. As new samples enter the filter, samples already in the filter shift along with their corresponding index values.

The architecture illustrated in Figure 4 presents the above algorithm for *TAP*=5. There are *TAP* - 1 = 4 registers storing previous samples (D[3…0]) and a register for the new input sample (ND).

Every older sample is compared with the new sample. C[3…0] register the results of these comparisons, which in turn supply the LSB bits of *TAP* bit wide registers CR[4…1]. The remaining bits of CR[4...1] are propagated from CR[3...0] registers, such that

$$CR[k] = \{CR[k-1]\ (TAP - 2{:}0),\ C[k]\} \qquad \textit{Equation 5}$$

where (:) denotes bit selection, and {} denotes concatenation.

Consequently, at any given time data register D*[k]* with its associated CR*[k]* register stores an input sample and all the comparison results of this input sample with other samples residing in the filter. Therefore, calculating the sum of bits in the CR registers generates the index information.

The CR register update mechanism of the new sample is different, updated with the inverted comparison results. Bit *b* of CN is updated with the inverted result of comparator *b*. Bit 0 of CN is initialized with 1.

Figure 5 illustrates the algorithm by presenting one cycle of the five tap example above as a new sample enters from the right. Table cells show the contents of the data register (ND, D) and the corresponding comparator result registers (CR and CN).

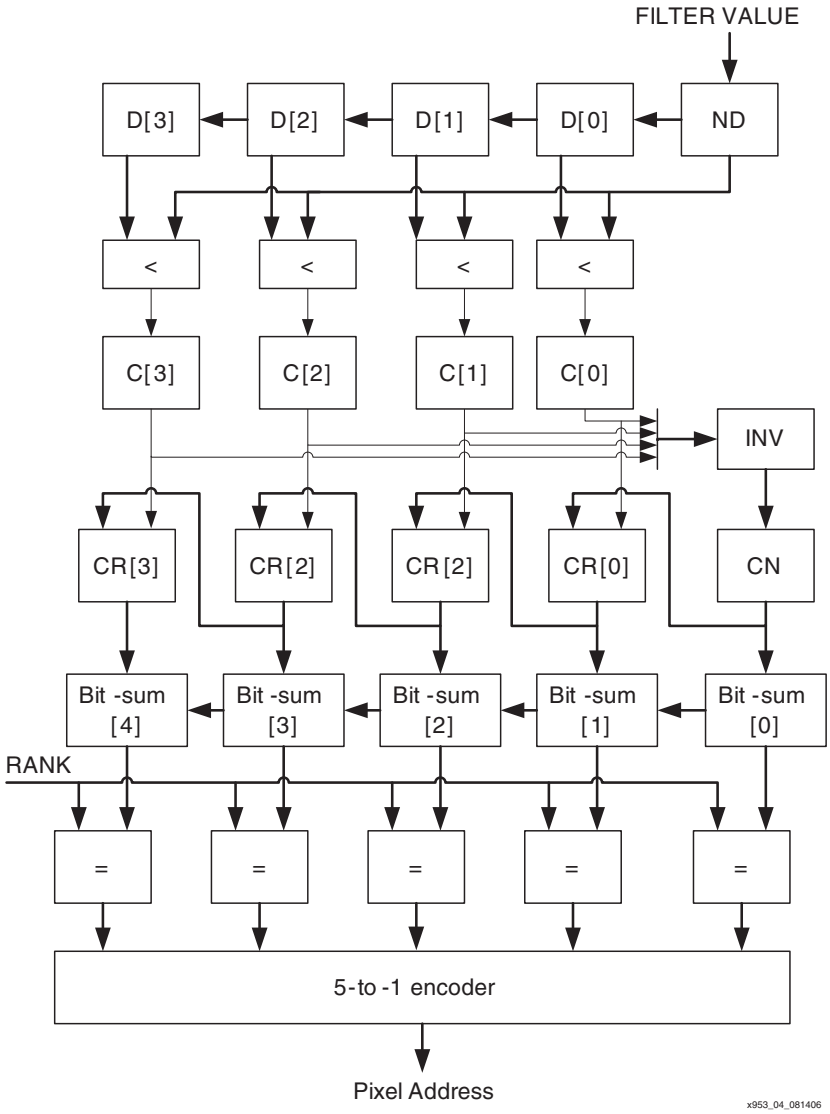

Figure 4: **Filter Core Architecture**

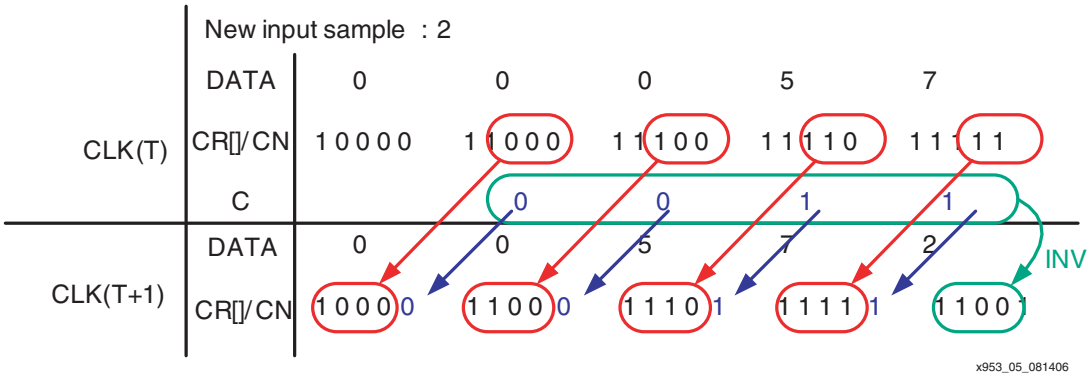

Figure 5: **Filter Core Operation Example**

Bit-Sum Operation

The design contains two files `cntr1_new.vhd` and `cntr1_old.vhd` describing two different architectures.

Cntr1_new uses an adder tree with *TAP*-1 number of 2-bit adders. For the new sample, this is the only solution because contents of *CN* can change arbitrarily between subsequent clock cycles. For summing the bits of CR[*TAP*-2 ... 0], an incrementer/decrementer structure is utilized taking advantage of correlation between subsequent CR values. For every sample, the appropriate counter described in cntr1_old.vhd should be

- incremented when the comparison result with the oldest sample (CR[*k*][*TAP*-2]) is 0 and the comparison result with the new sample is 1
- decremented when the comparison result with the oldest sample is 1, and the comparison result with the new sample is 0
- otherwise, kept unmodified.

The performance of the 1D filter plays a key role in classifying 2D implementation options. Table 3 summarizes maximum operating frequencies achieved for certain *TAP* numbers and FPGA families.

Table 3: **1D Filtering Performance(1)**

Family	*TAP* = 9	*TAP* = 25	*TAP* = 49
XC4V -10	400 MHz	400 MHz	350 MHz
XC2V -5	235 MHz	225 MHz	215 MHz
XC3S -4	200 MHz	185 MHz	185 MHz

Notes:

1. Post synthesis results, using Synplify Pro 8.5.

2D Extension

To process images (two-dimensional data), the filter core has to process *WH* new samples and generate one new output sample in every clock cycle. The 75 MHz pixel frequency of HDTV 1080i commercial video format allows 3x3 tap filters processing one input sample per CLK cycle to be used for image/video processing. On Virtex-5 devices, even 5x5 tap filters can be implemented using over-clocked 1D architectures. However, for larger filters, multiple samples have to be processed in each clk cycle to increase throughput.

In such cases, the filter core in the reference design is extended to process multiple samples per clock cycle. The number of samples processed per clock cycle is controlled by generic parameter *NEW INPUTS (NI)*. The data and comparator result registers shift by *NI* data positions. The number of comparators is multiplied proportionally as old samples should be compared with all new samples and new samples should be compared with each other. If *WH* is not an integer multiple of *NI*, the throughput of the filter core input supersedes that of the input stream, so in some clock cycles the number of valid new data samples is going to be less than *NI*. Therefore, the actual number of available new input samples might change at each CLK cycle. Processing dynamically changing number of new samples would require inserting numerous multiplexers into the data paths.

Instead, padding samples are inserted as necessary such that *NI* new samples enter the filter core every clock cycle.

$$\left\lceil \frac{WH}{NI} \right\rceil \times NI - WV \qquad \text{Equation 6}$$

padding samples are added to every filter column, so the actual filter uses a virtual filter window with WH_{virt}**WW* size, where

$$WH_{vin} = \left\lceil \frac{WH}{NI} \right\rceil \times NI \qquad \text{Equation 7}$$

Figure 6 illustrates such a virtual window for the *WH* = 3, *NI* = 2 case. Valid samples in the window are marked with light grey, padding samples are marked with dark grey. Figure 7 presents the contents of the data registers clock by clock using the example from Figure 6. New inputs are inserted from the right as the filter window is moved horizontally.

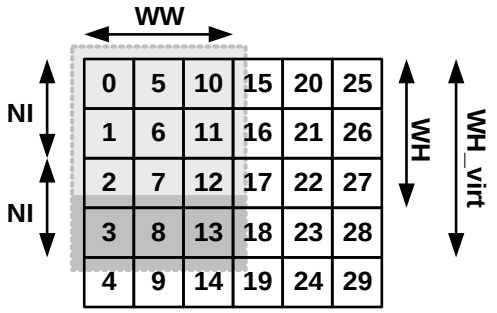

Figure 6: **Virtual Filter Window**

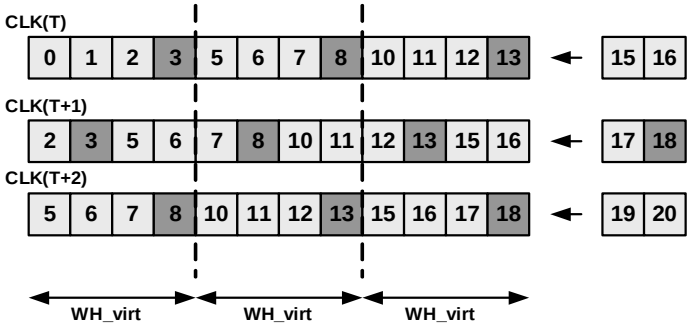

Figure 7: **Data Registers Contents**

All data registers may contain both valid and padding samples during operation. Comparisons are done using all data registers irrespective of whether the actual sample is valid or not. Therefore, the design size scales with the size of the virtual filter window. Padding samples are masked out at the shift registers before summing up the bits of comparator results. For older samples, masking is done for 2*NI* bits (*NI* bits for masking the comparison results with the *NI* new samples, and another *NI* bits to mask the comparison results of the oldest (discarded) samples CR*(TAP-1:TAP-NI)*). Masking values change according to the validity of new samples. In Figure 6, the masking value is 11 for CLK(T) and 10 for CLK(T+1). Generally, the masking value is all ones 11...11, except for the last cycle of a filter column, where it can be generated by:

$$mask\_old = \{C1(NI - NZ, C0(NZ)\} \qquad \text{Equation 8}$$

$$mask = \{mask\_new << NI, mask\_old\} \qquad \text{Equation 9}$$

where C1(*k*) denotes a *k* bit wide set of ones and C0(*j*) denotes a *j* bit wide set of zeros. All bits of the *TAP* wide registers of new samples can be masked, as register contents can change from clock to clock. The mask value is periodic with *WH_virt* and can be generated by a shift register.

Apart from bit masking, bit-summing for the new samples is the same as in the 1D case (adder tree). However, counter-based bit summing for the least recent samples become more complex

as *NI* increases, because the modification value is extended from the (-1, 0, +1) to the (-NI….+NI) range. The modification value for data index can be computed by:

$$\sum_{b=0}^{NI-1} C(I)(b) - \sum_{i=0}^{NI-1} SH(I)(TAP-1-t) \qquad \text{Equation 10}$$

Depending on the number of padding samples inserted certain filter configurations can become prohibitively large. The complexity of the filter core is proportional to *NI* and the virtual filter size (*TAPv=WW* WH_virt*). *Figure 8* presents the number of comparators, comparing old and new samples, as a function of *TAPv*. The number of comparators, as well as the resource requirements of the Y Generator modules, scale with *NI*.

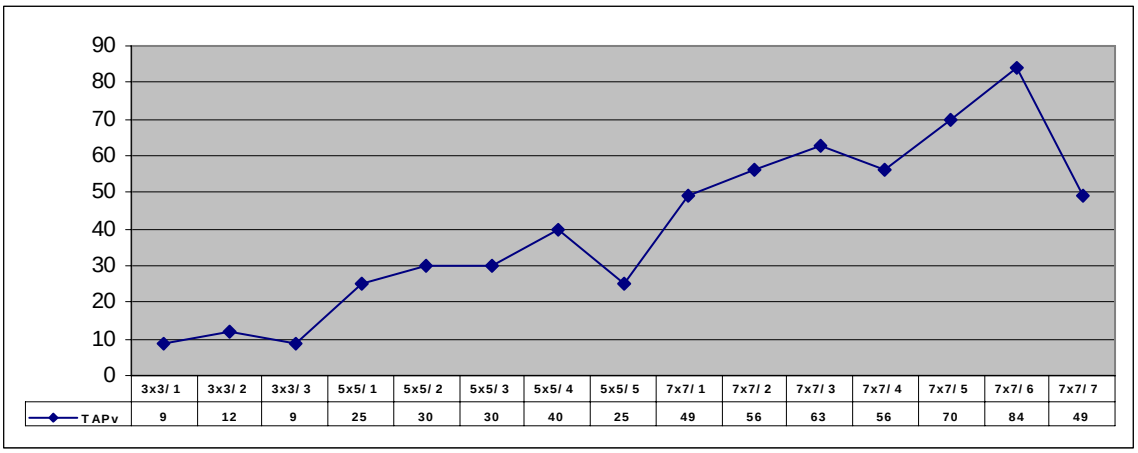

Figure 8: **Virtual Filter Size (TAPv) for Different Filter Configurations**

Performance and Resource Characterization

The design was implemented using Synplicity Synplify Pro 8.5 and ISE 8.1.03 tools with the following tool options:

- Synplify: resource sharing turned on, other options turned off
- map –cm speed –k 8
- par –ol high –t 9 (for Virtex-4 and some Spartan-3 results)
- par –ol high –t 8 (for Spartan-3 results)

Filter *configuration* for Table 4 through Table 10 is defined by $WH \times WW / NI$. Other than WINDOW_HEIGHT, WINDOW_WIDTH, and NEW_INPUTS, default values were used for generic parameters.

Characterization was performed on a Virtex- 5 XC5LX30-1 device, Virtex-4 XC4VSX35-10 device, and on a Spartan-3 XC3S1000-5 device. For characterization, a top-level wrapper module was used which registers all input and output signals of the *rank2d* module. Tests were performed with two CE options. In one scenario, CE was tied to logic 1, resulting in some logic optimizations and simplified CE routing. In the second scenario, CE was driven by a logic-fabric FF simulating the environment when the design is driven by other modules within the FPGA.

The last rows of the characterization tables contain expected performance for 75 MHz 1080i (HDTV) filtering. The actual clock frequency required to perform 2D filtering is a function of the virtual vertical size of the filter kernel (WH_{virt}) [Ref 12] and the number of new input samples/CLK (*NI*), as defined by [Ref 1].

Table 4: **Performance Requirements for HDTV 1080i Filtering (F_{CLK} HDTV [MHz])**

	***NI* =1**	***NI* =2**	***NI* =3**	**NI=4**
WH = 3	225	150	75	---
WH = 5	375	225	150	150
WH = 7	525	300	225	150

Based on Table 4, configurations highlighted in green are suitable for HDTV 75 MHz 1080i (HDTV) filtering.

Table 5: **Performance and Resource Numbers for Virtex-5, CE Tied High**

Configuration	**3×3/1**	**5×5/**	**7×7/1**	**7×7/2**
FFs	579	1039	1736	2521
LUTs	289	750	1446	2657
Block RAMs	3	6	9	9
$F_{CLK\ max}$ [MHz]	460	420	400	340

Table 6: **Performance and Resource Numbers for Virtex-5, CE from Fabric Register**

Configuration	**3×3/1**	**5×5/1**	**5×5/2**	**7×7/1**	**7×7/2**
FFs	548	998	1413	1643	2443
LUTs	274	680	1222	1306	2331
Block RAMs	3	6	6	9	9
$F_{CLK\ max}$ [MHz]	410	355	320	320	320

Table 7: **Performance and Resource Numbers for Virtex-4, CE Tied High**

Configuration	**3×3/1**	**5×5/1**	**7×7/1**	**7×7/2**
FFs	544	1169	2211	3416
LUTs	363	812	1556	2749
Block RAMs	6	12	18	18
$F_{CLK\ max}$ [MHz]	400	375	355	300

Table 8: **Performance and Resource Numbers for Virtex-4, CE from Fabric Register**

Configuration	**3×3/1**	**5×5/1**	**5×5/2**	**7×7/1**	**7×7/2**	**7×7/3**
FFs	595	1188	1702	2223	3342	4142
LUTs	344	780	1454	1534	2771	4153
Block RAMs	6	12	12	18	18	18
$F_{CLK\ max}$ [MHz]	330	300	270	260	245	235

Table 9: **Performance and Resource Numbers for Spartan-3, CE Tied High**

Configuration	**3×3/1**	**5×5/1**	**5×5/2**	**5×5/3**	**7×7/1**	**7×7/2**	**7×7/3**	**7×7/4**
FFs	683	1088	1832	2044	2666	3872	4683	5132
LUTs	323	851	1420	1913	1545	2747	4195	4664
Block RAMs	6	12	12	12	18	18	18	18
$F_{CLK\ max}$ [MHz]	245	195	180	165	175	160	150	150

Table 10: **Performance and Resource Numbers for Spartan-3, CE from Fabric Register**

Configuration	3×3/1	5×5/1	5×5/2	5×5/3	7×7/1	7×7/2	7×7/3	7×7/4
FFs	768	1470	2061	2288	2642	3839	4639	5076
LUTs	363	804	1439	1952	1544	2756	4206	4660
Block RAMs	6	12	12	12	18	18	18	18
$F_{CLK\ max}$ [MHz]	225	190	175	180	170	160	150	150

If the input is in RGB format and Y_GENERATOR_TYPE is set to 0 or 1, additional resources are required for the Y Generator blocks, for channel (R, G, B) summation (Y_GENERATOR_TYPE=0), and true luminosity computation (Y_GENERATOR_TYPE=1) (Equation 4). In these cases, the number of Y Generator modules instantiated is equal to *NI*.

System Generator Token

To facilitate easy integration of the reference design into a complex system developed using System Generator, a black-box token (Figure 9) encapsulating the VHDL code is supplied.

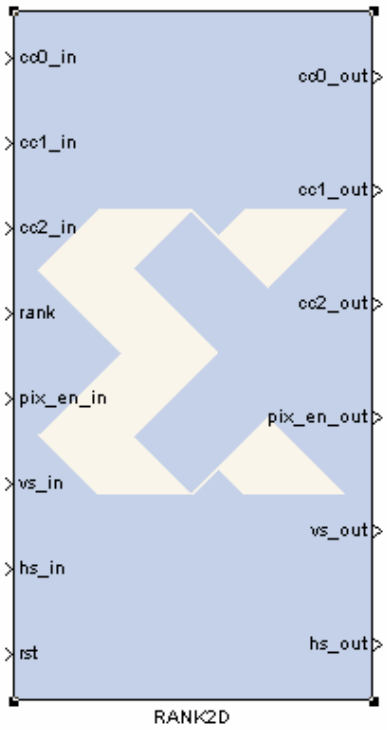

Figure 9: **System Generator Token**

The System Generator instance can be parameterized through a Graphical User Interface (GUI) invoked by double clicking the token. The GUI variables are the same as the generic variables for the top-level VHDL code (see Table 2).

To ensure that the black box component works correctly with the rest of the design, file *rank2d_top_config.m*, which resides under the `/testbench/matlab/sysgen` directory, *has to be copied into the project (MATLAB working) directory*. Also, the configuration utility has to refer to the source VHDL files. Edit the bottom portion of the code, such as

```
this_block.addFile('../../../GenXlib_utils.vhd');
```

so the source file locations are correctly defined relative to the directory where *rank2d_top_config.m* is located.

System Generator Test Bench

To help prototype, test, and verify the 2D rank order filter, a System Generator test bench is included with the reference design. To open the test bench, change your MATLAB directory to `testbench/matlab/sysgen`, and load `rank_2d.mdl` (Figure 10).

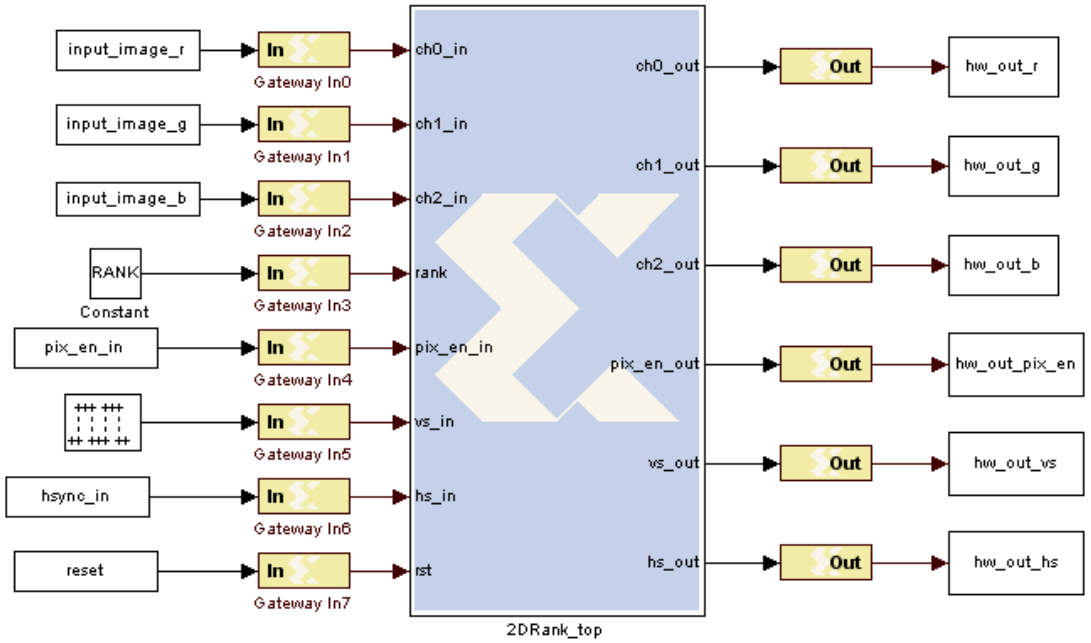

Figure 10: **System Generator Test Bench**

Double clicking the black-box icon invokes the GUI, which assigns generic variables and workspace variables sharing the same name as the generic variables in the HDL code. Variables get initial values assigned when simulation starts. To see the initial workspace variable assignments, select Model Properties from the file menu. From the Callbacks tab, pick InitFnc, which reveals the initial assignments. This script invokes `rank2d_init_mdl_2.m`, which loads a test image and sets up the workspace variables driving the ports of the token, such as input_image_r, input_image_g, input_image_b, the enable, reset, and sync signals.

Running the Test Bench

The test bench can be executed using ISE simulator ISIM, external simulator ModelSim®, or using hardware co-simulation. See the System Generator documentation for more information on hardware co-simulation. By default, the test bench uses ModelSim, taking advantage of the option to leave the ModelSim simulation window open after the simulation has completed. To switch between simulators, right click on the 2Drank_top token, and select Look under Mask from the context menu. Double click on the RANK2D token, and select the Simulation Mode of your choice in the block properties dialog box displayed. ModelSim specific options can be set by double-clicking the ModelSim token. Loading a macro file before the simulation starts enables displaying additional (internal) VHDL signals during simulation.

Click on the Start simulation (*) icon to run the simulation. After the simulation is finished, function `rank2d_post_proc_o.m` is invoked, which reformats VHDL output for visual verification (Figure 11).

A fixed-point MATLAB model (`rank2d_matlab.m`) is included in the bundle to facilitate bit-true verification of VHDL results. By definition, the rank order filter selects a sample from the magnitude ordered list of samples. If the kernel contains pixels with colors that have similar Y values, the actual pixel selected may be application specific. In other words, if the kernel contains pixels with different colors but similar Y values, the choice of Y at the output is unique, but the choice of colors corresponding to the particular Y value is up to the application. For that

reason, some errors between the MATLAB model and VHDL output results are tolerable, as long as the differing color values share the same Y value.

The test bench displays the output images of the VHDL and the MATLAB model outputs, as well as the differences between them (if any). The difference is amplified so the difference image utilizes the available brightness dynamic range of the display.

Figure 11 shows the input image. Figure 12 presents the corresponding VHDL simulation results.

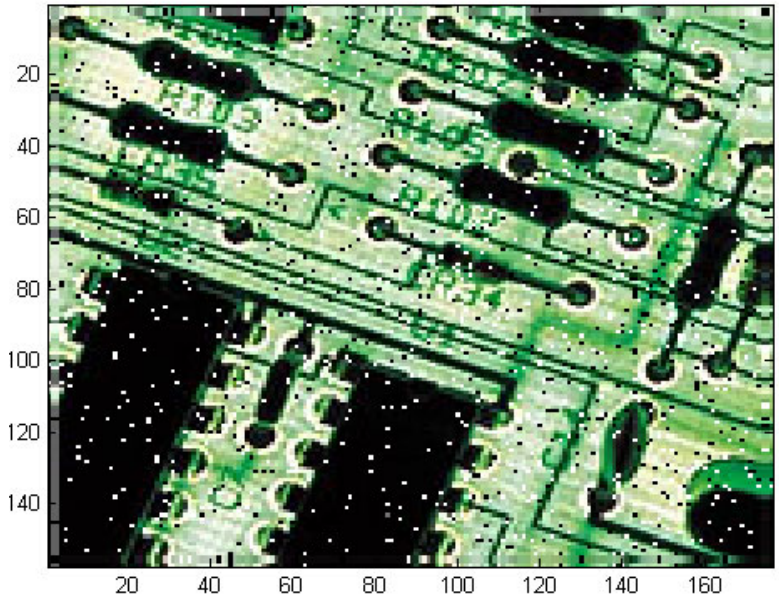

Figure 11: **R, G, B Stimulus**

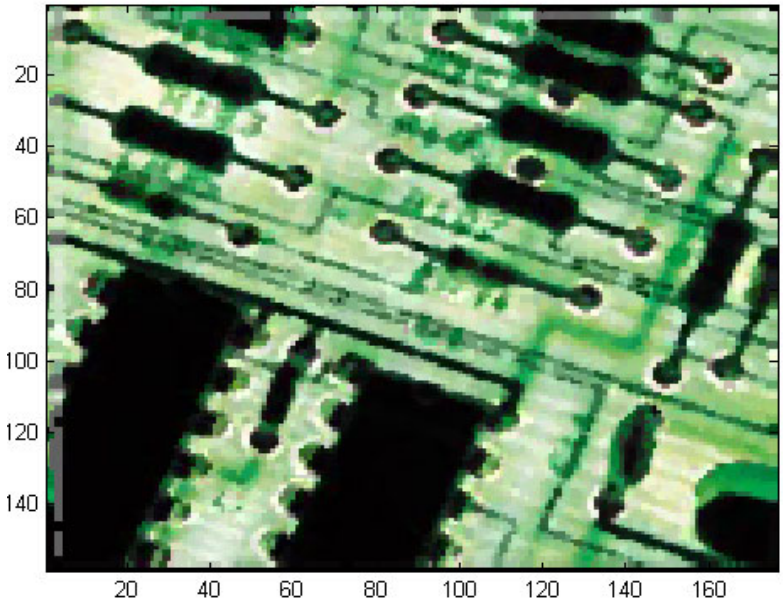

Figure 12: **Output Filtered with a 3x3 Median Filter**

Reference Design Files

The reference design files can be downloaded from: xapp953.zip

Source files to be inserted into an ISE project, in compilation order, are:

- `genXlib_util.vhd`
- `genXlib_arch.vhd`
- `rank2d_utils.vhd`
- `rank2d_latency.vhd`
- `cntr1_new.vhd`
- `cntr1_old.vhd`
- `delay_line.vhd`
- `delay_line_srl16.vhd`
- `delay_line_srl32.vhd`
- `lbuff_mem.vhd`
- `comp_module.vhd`
- `sub_inst.vhd`
- `sub_inst_v2.vhd`
- `sub_inst_v4.vhd`
- `fvg_sum.vhd`
- `fvg_y.vhd`
- `filter_core.vhd`
- `rank2d_top.vhd`

A System Generator token encapsulating the HDL code is also available for System Generator users. A System Generator test bench is provided to visually inspect output results.

References

1. R. Roncella, R. Saletti, P. Terreni, *70-MHz 2-mm CMOS Bit-Level Systolic Array Median Filter*, IEEE Journal of Solid State Circuits, vol 28, 1993.
2. L. Chang and J. Lin, *Bit Level Systolic Arrays for Real Time Median Filters*, International Conference on Acoustics, Speech and Signal Processing, 1990.
3. B. K. Kar and D. K. Pradhan, *A New Algorithm for Order Statistic and Sorting*, IEEE Transactions on Signal Processing, vol 41, August 1993.
4. H. Heijmans (1991). *Theoretic Aspects Of Grey-Scale Morphology*. IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI) 13(6):568-582.
5. M. Karaman, L. Onural and A. Atalar, *Design and Implementation of a General-Purpose Median Filter in CMOS VLSI,* IEEE J. Solid-State Circuits, vol. 25, April 1990.
6. C. Chakrabarti, *Sorting Network-Based Architectures for Median Filters*. IEEE Trans. on Signal Processing, March 1994.
7. C. Chakrabarti and L. Wang, *Novel Sorting Network-Based Architectures For Rank Order Filters*, IEEE Trans. On VLSI Systems, vol. 2, December 1994.
8. F. A. Suhaib, P. Y. K. Cheung, L. Wayne, Novel *FPGA-Based Implementation of Median and Weighted Median Filters for Image Processing*, Field-Programmable Logic and Applications (FPL 2005), 2005.
9. V. Fischer, R. Lukac, and K. Martin, *Cost-Effective Video Filtering Solution for Real-Time Vision Systems*, EURASIP Journal on Applied Signal Processing 2005:13, 2036-2042.
10. J. B. Wilburn, *A Statistical Methodology for Automatic Target Recognition in Satellite Imagery*.

11. C. Chakrabarti, *High Sample Rate Array Architectures for Median Filters*, IEEE Transactions on Signal Processing, vol. 42, March 1994.
12. F. A. Suhaib, P. Y. K. Cheung, L. Wayne, Novel *FPGA-Based Implementation of Median and Weighted Median Filters for Image Processing*, Field-Programmable Logic and Applications (FPL 2005), 2005.

Acknowledgements

- Peter Szanto, Budapest University of Technology and Economics, Department of Measurement and Information Systems, for contributions to the final architecture and implementation of the modules in HDL.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
09/06/06	1.0	Initial Xilinx release.
09/21/06	1.1	Added “Acknowledgements.”