

This paper has been published as

**A Visual Environment for Real-Time Image Processing in Hardware
(VERTIPH)**

by

C. T. Johnston, D. G. Bailey, and P. Lyons

in

**EURASIP Journal on Embedded Systems
Volume 2006 (2006), Article ID 72962, 8 pages**

The paper is online at <http://tinyurl.com/y8sopz>
The journal is online at <http://tinyurl.com/wl6nm>

A Visual Environment for Real Time Image Processing in Hardware (VERTIPH)

C. T. Johnston, D. G. Bailey, P. Lyons
Institute of Information Sciences and Technology,
Massey University, Private Bag 11222, Palmerston North, New Zealand
C.T.Johnston@massey.ac.nz, D.G.Bailey@massey.ac.nz, P.Lyons@massey.ac.nz

Abstract

FPGAs offer an excellent platform for use in real-time video processing applications. There are several modes of operation that can be used to implement image processing algorithms; streamed, offline or a hybrid of the two. We discuss these and consider the suitability of the present types of languages for the implementation of image processing algorithms on FPGAs. Examples of Hardware Description Languages, Language Extensions, and Hardware compilers are discussed along with their strengths and weaknesses. We propose VERTIPH, a new multiple-view visual language that is designed for image processing on FPGAs avoid these weaknesses. Three views show different parts of the system; an overall architectural view, a computation view and a scheduling view.

Keywords: FPGA, Visual Language, Image processing

1 Introduction

With the continual growth in size and functionality of FPGAs (Field Programmable Gate Arrays) there has been increasing interest in their use as implementation platforms for image processing applications, particularly real time video processing [1]. Due to their structure with a large array of parallel logic and registers, FPGAs can exploit the data parallelism found in images. They can either performing all required operations or perform a subset of operations to reduce data before passing processing to a standard DSP or microprocessor. However the programming model is different from normal software design. It cannot be treated as a single processor with a large amount of memory. Instead an FPGA is a system with a large number of simple processors which competition for memory access. In designing an appropriate algorithm for the FPGA you need to take into account the limited memory bandwidth and the related constraints that this imposes on the system. There are three main processing models used in designing image processing on FPGAs, offline, stream and hybrid processing.

Offline processing is commonly used in hosted system configurations. As a co-processor, the FPGA's role in the embedded system is to complement as host computer by accelerating certain tasks. Offline processing virtually eliminates all constraints with the constraint on bandwidth is eliminated because random access to shared memory is possible and desired pixel values can be obtained over a number of clock cycles.

In stream processing, data is serially presented as a one-dimensional pixel stream by means of a suitable

access pattern [2]. Typically the access pattern is raster order in which pixels are presented left to right for each image row beginning with the top row. Effectively, this converts the spatial distribution to a temporal stream and is the mode often used for video rate processing. Processing is performed "on-the-fly" as the data is streamed through the system. This type of processing is more suited to the stand-alone configuration where the FPGA is connected directly to a video source that is producing a continuous stream of data. A good example is as when the FPGA is used as a "front end" in a smart camera processing the image from a sensor before storing the result into memory.

The strict time constraints involved with stream processing are dependent on the video capture rate and image size (for example PAL 25 frames per second and a 768 by 576 colour image). At the minimum clock rate, which is desirable for embedded systems, stream processing constrains the design into performing all of the required calculations for each pixel at the pixel clock rate. If this cannot be managed then some pixels in the stream will be missed and so will not be processed

Producing pixel outputs with high clock rates for non-trivial applications, such as lens distortion correction [3, 4], is difficult because for each pixel complex expressions must be evaluated. These can introduce significant propagation delay, which may easily exceed a single pixel clock cycle. A pipelined approach is thus needed that accepts an input pixel value from the stream and outputs a processed pixel value every clock cycle with several clock cycles of latency, equal to the number of pipeline states, between the input and output. This allows several

pipeline stages each for the evaluation of complex expressions and functions. Pipelining is an important technique for exploiting the temporal parallelism inherent in stream data.

In stream processing, memory bandwidth constraints dictate that as much processing as possible is performed on the data as it arrives. For some operations the order that the pixels are required for processing does not directly correspond to the raster order in which they are input. This requires that the image be partly or wholly buffered. There are complications associated with this but they may be unavoidable: Buffering requires memory which is limited on an FPGA. Frame buffering requires large amounts of memory (such as for image warping), typically off-chip memory and introduces additional latency. Data is placed behind limited bandwidth and serialized connections which can make it difficult to retrieve desired pixel values when multiple accesses are required such as for bilinear interpolation [ref kims]

Hybrid processing is a mixture of stream and offline processing. For example, stream processing can be used for image capture and display while offline processing can be used in order to provide random access to a region of interest in the captured image.

FPGAs allow increased performance by freeing the implementation from the fixed architecture of standard processors. However, they make the design and implementation of the algorithm more difficult for most image processing practitioners as they are not familiar with FPGA-related issues of concurrency, pipelining, priming and bandwidth. Offen [5] has stated that the classical serial architecture is so central to modern computing that the architecture-algorithm duality is firmly skewed towards this type of architecture.

FPGAs, with their small size, low power consumption, large number of I/O ports and large number of computational logic blocks are ideal for many embedded systems. Image processing which can support both spatial parallelism in the images with multiple processor blocks and the pipelining of their algorithms are a good fit for the structure of FPGA devices.

2 Present Languages

Schematic entry is too low-level as a design tool for image processing as it does not capture the algorithmic nature of image processing functions adequately. As the complexities of digital designs increased Hardware description languages (HDL) were developed allowing designs to be expressed at a higher level of abstraction from schematic entry. HDLs were designed to express both the temporal behaviour and circuit structure of electronic systems. Verilog [6], and VHDL [7] are industry standard HDLs. They can be thought of as the assemblers of

hardware programming, providing great flexibility at several levels from gate level through to Behavioural. As they offer similar functionality, we will concentrate on VHDL. The constructs that are supported by VHDL are very closely related to hardware and can be at a very low level making it a poor choice for implementing complex image processing algorithms. As a general purpose language there is no specific support for image processing operations. While HDLs offer a great deal of flexibility in terms of the control logic it is up to the designer to construct any state machines required to control the system. This can be advantageous, allowing very efficient control over the execution path. However this burdens the designer with designing both the required logic and the state machine to drive it. HDLs are very powerful and flexible but with this the high level algorithm can be lost amongst the details required to program them.

The need for more high level design tools has led to several different methodologies. One is to modify an existing software programming language to add in the constructs required for building hardware. Handel-C takes this approach with the aim of making hardware design more accessible for software engineers [8]. In most conventional programming languages, statements are executed sequentially following the order of assignment statements, and branches specified by flow-of-control statements (**while**-, **if**-statements, etc). In general, they do not offer the ability to run processes in parallel, although some may support process threads. The lengths of data types are defined by either the fixed architecture of the processor (ANSI C) or by the language (Java). These languages are not designed to be compiled into hardware, so they lack hardware-oriented constructs such as ways to define communication between different processes, to create RAMs and to assign I/O pins.

There are five main areas in which conventional programming languages need to be extended in order to support hardware design. It should be possible:

- to specify that operations occur concurrently and to specify the timing or clock speed of processes
- to define communication between processes running at different speeds
- to define data types in terms of their bit length as there is no fixed architecture to conform to
- to build architectural components such as RAMs, ROMs, WOMs, channels,
- to create low level structures such as wires along with bit level operations such as bit concatenation.

Handel-C is a language that compiles algorithms written in a high-level C-like language directly into gate-level netlists. It is based on a subset of ANSI-C with syntax extensions for hardware design such as

variable data widths, parallel processing and channel communication between parallel processing blocks. The language is designed to allow software engineers to express an algorithm without any knowledge of the underlying hardware [8].

Assignment statements in Handel-C take exactly one clock cycle. All other language statements, such as control logic constructs, add zero additional clock cycles although they can increase combinatorial delays to the extent that the system clock cycle may need to be lengthened [8].

Apart from the introduction of architectural constructs and bit level operations the only main differentiation from ANSI-C and Handel-C is the introduction of the **par** construct. All statements within a **par** block run in parallel. In Handel-C the default is sequential operation.

Handel-C provides a good level of abstraction from hardware design. However, its textual nature makes it difficult to understand the data flow in a parallel design. As illustrated below there is almost no difference in terms of textual representation between sequential and parallel code. This is common to all text based HDLs.

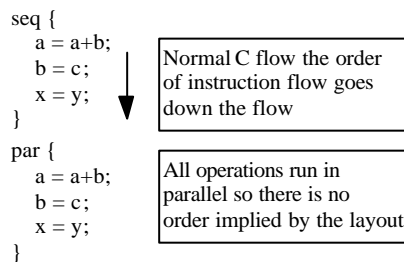


Figure 1: Logical flow of instructions

The increased level of abstraction from a hardware definition to a high level language enables the developer to concentrate on algorithm development. With these approaches, the designer has less control over the control flow (Handel-C builds an implied state machine), but gains an increase in design abstraction.

Another approach taken has been to move all of the hardware design away from the designer and to a hardware compiler. Normally there is a need to have some compiler directives for data type lengths. This approach has been taken by SA-C [9, 10] and MATCH [11]. SA-C is aimed at image processing and makes some changes to the normal C model. The language does not include pointers but it does incorporate common image processing functions such as array summing for histograms, and window loops. The first step in converting to hardware is to unroll loops. Then, where possible, consecutive loops are combined into one. This is followed by standard CSE (common sub-expression elimination) and temporal CSE (replacing a computation in one loop iteration

with a result computed in a previous iteration). It is through the loop unrolling that parallelism is exploited. In SA-C, if timing needs to be improved, a further step of breaking up complex expressions into pipelined sub-expressions can be invoked through compiler options.

These systems take all control away from the designer. They can achieve real-time operation using an offline design model. However they can only optimise an algorithm through pipelining the sequential algorithm.

While image processing algorithms are inherently parallel their implementation tends to be serial. A filter is parallel in its design, but is normally implemented as loops. Most image processing applications involve several steps which can each run concurrently as pipelined processes. It is therefore desirable to have a development tool which allows this parallelism to be captured at an appropriate level of abstraction.

When implementing algorithms on to FPGA we have used the following design flow:

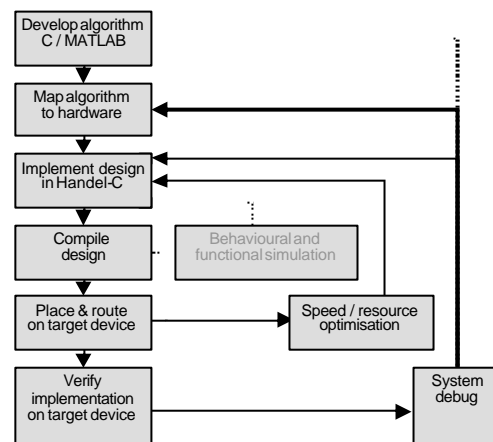


Figure 2: Image Processing on FPGA design flow

Thus when doing this we have spent most of the time mapping an algorithm into a form suitable for FPGA implementation, generally using a stream processing model. The aim of this is to make the implementation as efficient as possible. We do this by course gain pipelining (between operations), fine grain pipelining (breaking up operations), combining operations into one, utilising look up tables, CORDIC functions and redesigning a standard algorithm for one pass implementation.

This high level design is then implemented on to hardware using a hardware language in our case Handel-C. There is a large gap between our design mapping and the hardware languages used to implement the design. A high-level language for expression of image processing algorithms in hardware should aim to facilitate this. It should

- allow a mixture of parallel and sequential design
- make it clear to the designer what runs in parallel and what forms part of a pipeline
- be able to detect when concurrent processes may access a shared resource such as a RAM and manage this accordingly, by informing the designer and giving some suggestions as to how to resolve the issue
- be able to handle stream, offline and hybrid processing models
- have some of the common image processing functions and data-types as primitives. Examples include row and pixel buffering, window filters, and look up tables (LUT)
- be intuitive and easy to use
- provide multiple views onto the design

No present system incorporates these features, and this paper is based on providing a system which meets these requirements.

Visual design tools can aid in the specification and development of image processing algorithms. There have been a number of different visual image processing languages for use on a serial computer including Khoros [12] and OpShop [13]. There are also several general purpose visual languages which can be used for image processing including LabView [14] and Simulink [15]. Khoros, LabView and Simulink now have extensions that allow them to be used for FPGA design, although this was not their original purpose. Khoros offers a high level view for algorithm development, but it was not designed to support the implementation of novel image processing operations, and so it does not include lower level design capabilities. Recently other IP based systems such as Celoxica's PixelStreams [16] and Xilinx's DSP block sets [17] have been developed to provide faster development time for projects and provide similar functionality to Khoros.

These languages all follow a form of the dataflow paradigm where streams of data flow through a network of nodes which performs a computation on the tokens within the stream before passing the output data to the next function block [18]. It has been noted [19] that dataflow graphs (the natural visual representation of this programming paradigm) are an effective representation for problems in digital signal processing (DSP), both because it is natural for many DSP researchers and because it exposes parallelism in

the algorithm with limited constraints on evaluation order.

3 VERTIPH

As discussed in section two textual languages lack the ability to represent concurrency and complex scheduling of operations. We believe that a visual representation is better at representing the parallel design of image processing algorithms than textual languages. It will use multiple views of the algorithm and resources required for data storage, capture and processing. A comparison of VERTIPH to other HDLs and its required features was presented in [20].

At present there are three defined views of the VERTIPH system; a top level architectural view, a computational view and a scheduling and resource view.

3.1 Architecture View

This view aims to provide the designer with an overall system view from which the other views can be used to specify the low level tasks required to perform the desired function. As image processing algorithms are broken up into blocks which perform very specific processing tasks, they can be developed independently using test images to make sure the function is correct. An image processing algorithm is normally constructed by having several blocks operating one after another. This view is to aid in the design of the algorithm at this high block level and functions much like OpShop or Khoros.

The use of component blocks allows for the encapsulation of resources (such as frame buffers) and for the logical grouping of related computational processor. A frame buffer component will have both an input stream and an output stream, and contained within it will be two RAM banks. Other components which communicate with this only see address and data lines and the switching between memory banks can be done within the component.

Processors which are logically related to each are also encapsulated. An example of this part of a colour segmentation and tracking algorithm detailed in [21, 22] a bounding box is required. This has a data structure to hold bounding boxes for the different colour class, a processor which uses this with the inputted current selected colour and a processor which calculates the results for all the bounding boxes detected. These are logically related and should therefore be kept together. This idea of encapsulation

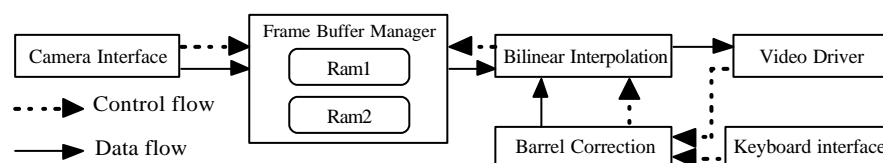


Figure 3: Architectural view of a Barrel distortion correction system showing: components, control and data flows

borrowed from object oriented software engineering. However we do not advocate the use of inheritance for image processing on hardware as most components follow a *has-a* rather than the *is-a* relationship required for objects. For example a filter has a data structure and an operator; however different filters can have very different implementations.

A filter might be constructed like the one described in [3] or it might be separable, offering less need for row buffering and simpler arithmetic operators as the one used in [21, 22]. Encapsulation also allows components to keep logically related processors together in one place. This can simplify the sharing of data and resources and it becomes clear which processor can access them and for what purpose. This can in turn make the scheduling of these processors easier as the developer does not need to remember all the parts of the system which are related to the resource or data structure being used.

Recursive encapsulation can allow for very complex IP blocks to be built. With one block and interfaces representing a complex system of data structures, resources, processes and their scheduled operations or response to events. This also allows for a hierarchy of state machines to be used, with each component within a component having its own state machine which may or may not then be controlled by a higher level of the design.

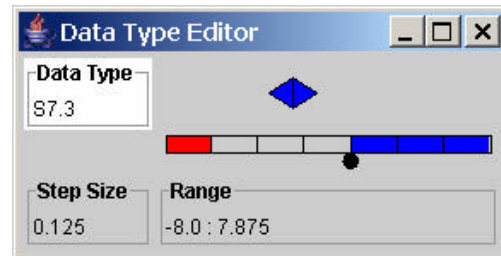
The aim of this view is to allow logical separation of operators and the encapsulation of data and processors related to that operation.

3.2 Data types

As we are having high level blocks with connections between them, there is a need to check that the output from one block is acceptable as an input to another. This leads to the need for strict type checking and this requires data types. For image processing there needs to be types for both 16- and 32-bit colour, 8- and 16-bit grey scale and any length integer and fixed point numbers both signed and unsigned. As an FPGA is a concurrent systems then there also needs to be an indication on how the data is communicated. This could be by a channel, register or as a wire (no storage). Once this is done type checking can be performed between connections to check they are of the same type.

Floating point numbers have not been included within the system for several reasons: using 32- or 64-bit IEEE standard 754 floating point numbers requires large amounts of resources and are power intensive. Image processing operations generally do not require the dynamic range which floating point offers. Although varying word length floating point numbers can be selected, fixed point numbers offer better overall noise performance when the probability density function of the signals is uniform [23]. As long as appropriate fixed point word lengths are

chosen almost all standard image processing operations can be implemented (with some degree of rounding error). Fixed point operations have a small footprint in hardware and lead to lower power drain making them the best choice for most applications[23]. Below is the dialogue for specifying the size and range of fixed point numbers in VERTIPH.



Another advantage to having strict type can be the automatic alignment of types allow for easier arithmetic manipulation. When adding two fixed point numbers they need to be aligned and the resulting register needs to be of the correct width. Often this is left up to the designer to do, Handel-C does simple type checking, when the operands and result are not of the same type they must be manually cast to that type. This can be done automatically in most cases. When doing this, the order of operations becomes important as performing operations in one order can give a different sized result to another order. Below is an example showing an operation with a multiplication, addition and subtraction. Depending on the order of operations the temporary registers can be different although in this case the final result register is the same size.

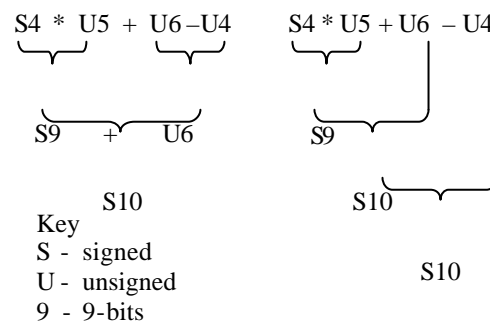


Figure 4: Different temporary register sizes depending on arithmetic order

3.3 Operators

The main operator that needs to be added in VERTIPH is a window filter. This is because it is such a common low-level image processing operator. There are several forms that a window operator can take in hardware [24] therefore a design wizard type approach for constructing this operator has been developed for VERTIPH.

3.4 Computation View

Developers who never design their own algorithms, can assemble pre-defined library modules into a high-level overview like the one shown in Figure 2. Similar to how other IP based systems such as Celoxica's PixelStreams and Xilinx's DSP block sets.

However, to allow the developer to design their own operations and help with buffering, pipeline priming and synchronisation, a lower level timing view is needed. To accomplish this we have modified the Gantt chart notation [25]. In this notation, time flows from left to right, so in Figure 6 (a), operation **x** is followed by operation **y**, which is followed by operation **z**. In Figure 6(b), the operations occur concurrently, and in Figure 6(c) they are pipelined. This representation is an abbreviation of Figure 6(d) which explicitly shows the parallel repeating processes that feed data from one to the other, and that each process is active in succeeding phases.

Of course, these basic types can be used together as is shown in Figure 4 which is the pipeline for the barrel distortion algorithm. This figure also shows VERTIPH's **if-** and **while-** control structures provided by the language which are based on the control structures used in Nassi-Shneiderman diagrams [26]. This pipeline view graphically conveys to the developer the time required to prime and flush

the pipeline.

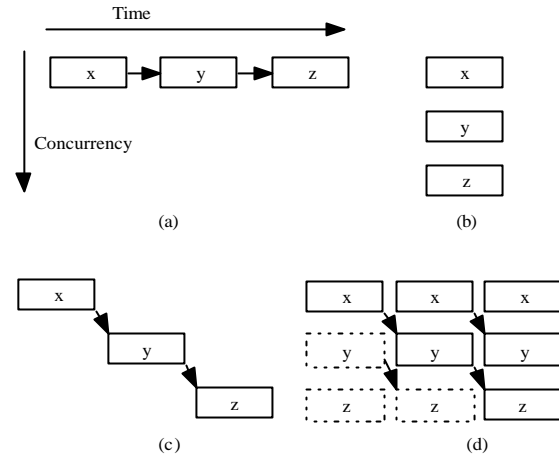


Figure 6: Process representations: (a) Sequential, (b) Parallel, (c) Pipelined, (d) actual pipeline structure

Operations can be registered or unregistered with unregistered operations needing feed to a register before a clock cycle can finish. To save space on the screen only the operation or register name is shown, a operations key has the instructions for the block in a Handel-C type syntax. This view shows the same information as a textual language but the layout

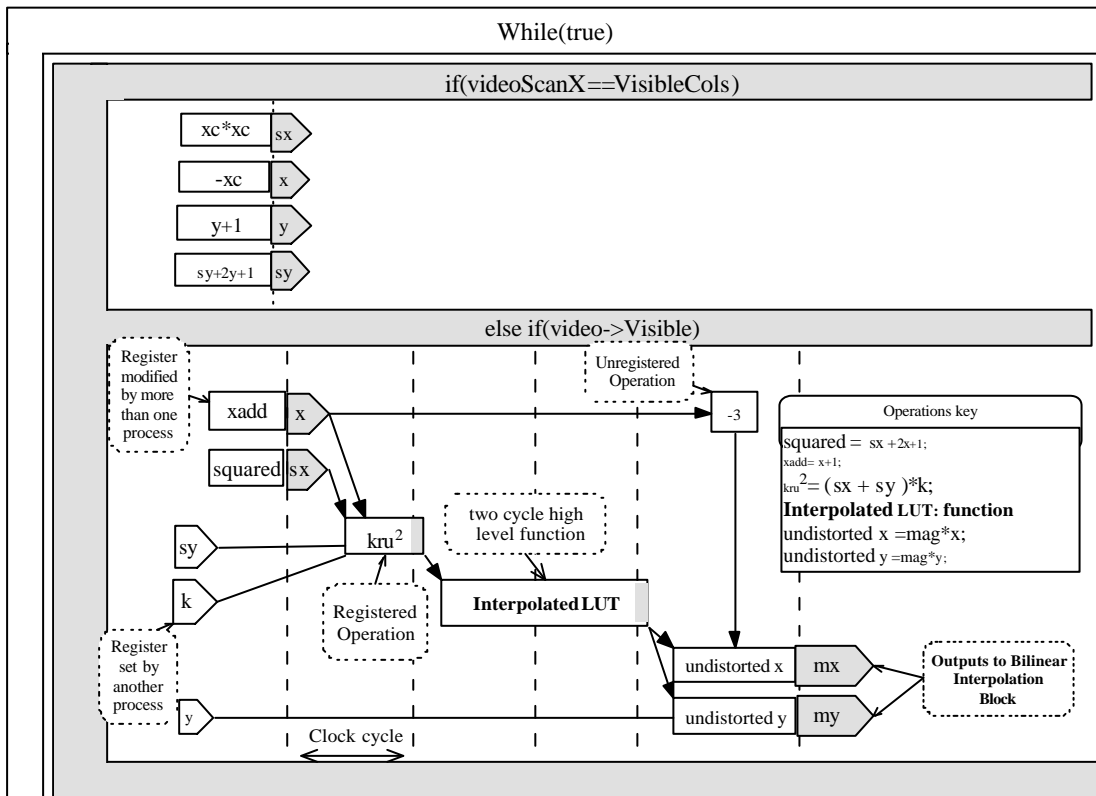


Figure 5: Low level view of Barrel distortion block showing control functions, timing and operation representation

makes the structure of the algorithm easier to visualise. For example it is immediately obvious that the x value must be offset by 3 before it is used in the calculation of the undistorted x value.

The language should where possible automatically generate structures to handle pipeline priming, stalling and flushing and it should prompt the developer when their design might be using values from a different stage of the pipeline.

This view aims to improve the visualisation of the concurrent aspects of the low level computations. This is done through the use of control structures that highlight clearly what parts of the code they control through vertical bars.

3.5 Scheduling View

In an embedded image processing FPGA based system there are a large number of processors competing for access to a limited number of resources. There are also processors which can only run after certain events have occurred, such as an external trigger or another processor finishing. These competing and co-operative processors need to be managed and scheduled. The encapsulation of resources and the associated processes which act on them allows the processes to be scheduled to take into account this resource sharing. This also allows for both global scheduling for processors and for local scheduling within components.

To help the designer to avoid resource conflicts such as two parallel processes accessing an external RAM at the same time, a resource usage view should be incorporated. This should work like standard Gantt software packages and identify when resources are used more than once in a time period. It can then suggest changes in the ordering of events. In the case of a multi process design, this would involve either modifying start conditions for processes (to ensure they do not run together) or the use of semaphores to block access to the resource. For a time-critical design such as stream processing from a video camera, the blocking approach is not desirable as it can cause data to be lost, such as when writing from a pixel stream to a frame buffer. Fortunately, blanking periods or pixel buffering can often be used to allow changes in the scheduling of competing processes. This view can also help in the scheduling of processes which run only at specific times. This could be used such as when a new frame is received, or for identifying where caching of pixels would be more appropriate than memory access, such as when a RAM access occurs when another process is using it and no rescheduling is possible.

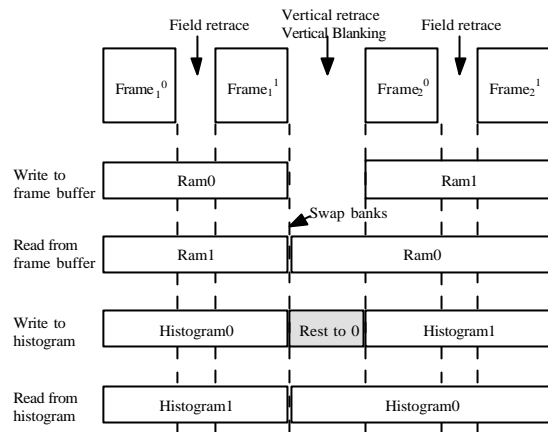


Figure 7: Timing of processes and resources used for a streamed histogram function

For example this type of resource conflict can occur when a histogram is being constructed and displayed. It is desirable to construct the histogram while the video stream is buffered into one RAM. At the same time, in a different clock domain, both the last full image and its histogram are being displayed. Keeping one of these processes from trying to write to one RAM while the other is reading can be accomplished with a simple condition test. The problem occurs due to the need to reset the histogram values in each bin before the histogram construction algorithm is run, as shown in Figure 7. While this requires a more complex passing of control of resources from process to process, it can also lead to error.

When a number of conditions have to be met before a process can execute, the nested logic can become difficult to interpret. This is especially true with expressions which become true when several counter or external events become true. It can be easier for the designer to have a flag which is set when the conditions are met. Such flags may be used to give the conditions for the control structures similar to how state machines and events are expressed in the ladder logic used in Programmable Logic Controllers [27].

4 Discussion

This work has identified several existing languages which are used for image processing on FPGAs, and commented on both their benefits and limitations.

A new visual language VERTIPH has been presented. This is based around making sequential, concurrent and pipelined operations clear to the developer. It also aims to break up the design into three parts to aid in its implementation; A block level architectural view similar to many other DSP block set systems, a computational view expressing the operations required in each block and based on Nassi-Shneiderman diagrams, and a scheduling view to aid in the development of the complex state machines that

are required to respond to events and aid in avoiding resource contention between processors. At present the block level design view and data type implementation is nearing completion, with the rest of it still to be implemented.

This system is only one of several approaches that can be taken when developing image processing systems on FPGAs, it is not a final solution but a step towards better tools and methodologies that will make FPGAs more usable and useful for image processing applications.

5 References

- [1] J. Villasenor and B. Hutchings, "The flexibility of configurable computing," *IEEE Signal Processing Magazine*, vol. 15, no. 5, pp. 67-84, 1998.
- [2] V. M. J. Bove, M. Lee, C. McEniry, T. Nwodah, and J. Watlington, "Media Processing with Field Programmable Gate Arrays on a Microprocessor's Local Bus," *Proceedings of SPIE Media Processors*, vol. 3655, no., 1999.
- [3] C. T. Johnston, K.T. Gribbon, and D. G. Bailey, "Implementing Image Processing Algorithms on FPGAs," *Proceedings of the Eleventh Electronics New Zealand Conference, ENZCon'04*, Palmerston North, pp. 118-123, November. 2004.
- [4] K.T. Gribbon, C.T. Johnston, and D. G. Bailey, "A Real-time FPGA Implementation of a Barrel Distortion Correction Algorithm with Bilinear Interpolation," *Proceedings of Image and Vision Computing New Zealand*, Massey University, Palmerston North, New Zealand, pp. 408-413, Nov. 2003.
- [5] R. J. Offen, "VLSI Image Processing," 1 ed. London: Collins, 1985, pp. 326.
- [6] IEEE, "IEEE Standard Verilog Hardware Description Language," 2004, <http://www.verilog.com/IEEEVerilog.html>, visited on August 2004/.
- [7] J. Bhasker, *A VHDL Primer*, Third ed. New Jersey: Prentice-Hall, 1999.
- [8] I. Alston and B. Madahar, "From C to netlists: hardware engineering for software engineers?," *Electronics & Communication Engineering Journal*, vol. 14, no. 4, pp. 165-173, 2002.
- [9] R. Rinker, J. Hammes, W. A. Najjar, W. Bohm, and B. Draper, "Compiling image processing applications to reconfigurable hardware," *Proceedings. IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 56-65, 2000.
- [10] J. Hammes, B. Rinker, W. Bohm, W. Najjar, B. Draper, and R. Beveridge, "Cameron: high level language compilation for reconfigurable systems," *Proceedings. International Conference on Parallel Architectures and Compilation Techniques*, pp. 236-244, 1999.
- [11] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, "A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems," *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pp. 39-48, 2000.
- [12] K. Konstantinides and J. R. Rasure, "The Khoros software development environment for image and signal processing," *Image Processing, IEEE Transactions on*, vol. 3, no. 3, pp. 243-252, 1994.
- [13] P. M. Ngan, "The Development of a Visual Language for Image Processing Applications," PhD Thesis in Computer Science, Massey University, Palmerston North, 1992.
- [14] "LabVIEW," 2005, www.ni.com/labview, visited on 16 February/2005.
- [15] The MathWorks, "Simulink 6.1," 2005, www.mathworks.com/products/simulink/, visited on 16 February/2005.
- [16] Celoxica, *PixelStreams Manual*, 1 ed: Celoxica, 2005.
- [17] Xilinx, "Xilinx System Generator for DSP Blockset," 2005, <http://www.xilinx.com/products/software/sysgen/blockset.htm>, visited on November/2005.
- [18] J. T. Buck, "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model," PhD Thesis in Electrical Engineering and Computer Sciences, University of California, Berkeley, 1993.
- [19] J. T. Buck and E. A. Lee, "Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model," *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, Minneapolis, pp. 429-432, 1993.
- [20] C. T. Johnston, D. G. Bailey, P. Lyons, and K. T. Gribbon, "Formalisation of a visual environment for real time image processing in hardware (VERTIPH)," *IVCNZ*, Akaroa, N.Z., pp. 291-296, 2004.
- [21] C. T. Johnston, D. G. Bailey, and K.T. Gribbon, "Optimisation of a colour segmentation and tracking algorithm for real-

- time FPGA implementation," *Image and Vision Conference New Zealand*, Dundie, pp. 2005.
- [22] C. T. Johnston, K.T. Gribbon, and D. G. Bailey, "FPGA based Remote Object Tracking for Real-time Control," *To appear at International Conference on Sensing Technology*, Palmerston North, New Zealand, pp. 2005.
- [23] A. S. L. Bainbridge-Smith, "Real Number Representation for Image Processing on FPGAs," *Image and Vision Computing*, Dunedin, pp. 471-475, 2005.
- [24] K. T. Gribbon, C. T. Johnston, and D. G. Bailey, "Formalizing Design Patterns for Image Processing Algorithm Development on FPGs," *IEEE Tencon*, Melbourne, Australia, pp. 21-24 November. 2005.
- [25] J. R. Schermerhorn, *Management*, Sixth ed. New York: John Wiley & Sons, 2001.
- [26] Nassi I and Shneiderman B, "Flowchart techniques for structured programming," *ACM SIGPLAN Notices*, vol. 8, no. 8, pp. 12-26, 1973.
- [27] T. R. Kuphaldt, "Volume IV - Digital," in *Lessons In Electric Circuits*, 4th ed: Tony R. Kuphaldt, 2005.