# Towards a Visual Notation for Pipelining in a Visual Programming Language for Programming FPGAs

**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**
VERTIPH is a visual language designed to aid in the development of image processing algorithms on FPGAs (Field Programmable Gate Arrays). We justify the use of a visual language for this purpose, and describe the key parts of VERTIPH. One aspect of importance is how to clearly and efficiently represent a pipeline of processors, and in particular distinguish a pipeline from the simpler serial or parallel structures. This paper develops a number of pipeline representations, discussing the rationale behind and limitations associated with each representation. The culmination of this development is the Sequential Pipeline with Detailed Bars, visually an efficient and unambiguous representation.

**Author Keywords**
Visual programming language, pipelining, FPGA.

**ACM Classification Keywords**
B.7 INTEGRATED CIRCUITS, B.7.2 Design Aids, C.3 SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS, D.2.2 Design Tools and Techniques, D.3 PROGRAMMING LANGUAGES, D.3.3 Language Constructs and Features, H.5 INFORMATION INTERFACES AND PRESENTATION (HCI).

## 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. Programming an FPGA is significantly different from writing software for conventional systems with a single processor. In designing an appropriate algorithm for FPGAs, it is necessary to take into account limited memory bandwidth, parallel operations, pipelining and resource conflicts [2].

Stream processing can be used for image processing. Data arrives as a one-dimensional pixel stream with a suitable access pattern [3], typically raster order (in which pixels are presented left to right for each image row, beginning with the top row). This converts the spatial distribution to a temporal stream and is often used for processing video data in real-time as it streams through the system. This type of processing is well suited to stand-alone configurations - for example, an FPGA "front end" for a smart camera can be fed directly by a continuous stream of data from a sensor, processing the image before storing the result in memory.

The strict time constraints involved in stream processing depend on the video capture rate and image size (for example each of the 25 frames that PAL produces per second contains a 768 by 576 colour image). Stream processing constrains the design to perform all of the required calculations for each pixel at the pixel clock rate. If this is not possible, then some pixels in the stream will be missed and so will not be processed

In some non-trivial applications, such as lens distortion correction [4] these high data rates are difficult to achieve, because each pixel requires complex calculations, producing a combinatorial delay that may easily exceed a single pixel clock cycle. In such situations it is common to break the calculation down into several phases, and to implement the hardware algorithm as a pipeline, with one clock cycle allocated to each stage. At any instant, successive stages of the pipeline will contain pixels at successive stages of processing. The overall rate of output will be one pixel per clock cycle, but there may be a latency of several clock cycles between inputting a raw pixel and outputting the processed result. Pipelining is an important technique for exploiting the temporal parallelism inherent in stream data.

FPGAs increase performance by allowing customised processor architectures. However, most image processing practitioners find it difficult to optimise a design in terms 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.

Textual HDLs (Hardware Description Languages) have traditionally been used for programming FPGAs, but they do not differentiate well between pipelined, parallel and sequential operations. In order to address these problems, the VERTIPH visual programming language incorporates several views of the system being designed. One view, derived from the Gantt chart [6], is designed for

expressing parallel and sequential operations. Here we describe how pipelining was incorporated into this view.

## PRESENT LANGUAGES

Schematic entry is too low-level for designing image processing hardware as it does not capture the algorithmic nature of image processing functions adequately. HDLs (Hardware Description Languages) were developed to allow designers to capture the high-level temporal behaviour of complex digital designs as well as their circuit structure. The industry-standard HDLs Verilog [7] and VHDL [8] can be thought of as the assemblers of hardware programming, providing great flexibility from gate level up to behavioural level. The low level constructs supported by HDLs make them a poor choice for implementing complex image processing algorithms. Being general purpose, HDLs offers no specific support for image processing operations.

HDLs require the designer to explicitly specify any state machines that control the execution path. This offers great flexibility, and can control the execution path very efficiently, but the designer can easily lose sight of the high-level algorithm when dealing with the details of controlling it.

The need for higher-level design tools has led to several different methodologies. One approach is to incorporate hardware-oriented constructs into an existing software programming language. The designers of one popular HDL, Handel-C, have taken this approach, with the aim of making hardware design more accessible to software engineers [9]. However, most conventional programming languages lack constructs for specifying efficient hardware. For example, in most conventional programming languages, assignment statements are executed sequentially and it is not possible to run processes in parallel (although some support process threads). They are also poor at defining low-level data paths. The lengths of data types are defined either by 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.

Handel-C is designed to allow software engineers to compile an algorithm written in a high-level C-like language directly into gate-level netlists, without knowing anything about designing hardware [9]. It is based on a subset of ANSI-C with hardware-oriented extensions such as variable data widths, parallel processing and channel communication between parallel processing blocks. Apart from the introduction of architectural constructs and bit level operations the only significant difference between ANSI-C and Handel-C is the introduction of the **`par`** construct. All statements within a **`par`** block run in parallel rather than the default sequential operation.

Handel-C automatically builds a state machine to control the execution path; this is both a benefit and a drawback; the designer can concentrate exclusively on algorithm development, but loses some control over details such as control flow.

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 in Figure 1, there is almost no visual difference between sequential and parallel code. This is common to all text based HDLs.

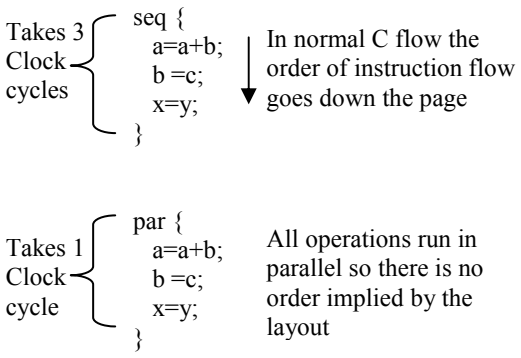


**Figure 1: Logical flow of instructions**

Besides modifying standard software programming languages, HDL designers have also produced hardware compilers. These take all the hardware design decisions except data type lengths away from the designer. This approach has been taken in SA-C [10, 11] and MATCH [12]. 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. However they can only optimise an algorithm though pipelining the sequential algorithm.

While many image processing algorithms are inherently parallel, they are commonly expressed serially, for implementation on a serial processor. For example a filter is parallel in its specification, but is normally implemented as loops. Most image processing applications involve several steps which can each run concurrently as pipelined processors. It is therefore desirable to have a development tool which allows this parallelism to be captured at an appropriate level of abstraction. This leads to the need for multiple views to capture; the design of processors, the interaction of processors with each other and the local and global scheduling of processors.

A high-level language for expression of image processing algorithms in hardware should aim to reduce the difficulty of bridging this gap. 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 processors may access a shared resource, and suggest how to resolve the issue;
- be able to handle stream, offline and hybrid processing models;

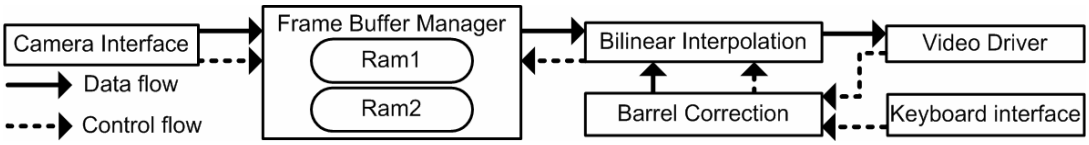


**Figure 2: Architectural view of a Barrel Distortion correction system showing: components, control and data flows**

- include 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.

Currently no system incorporates all of these features; this paper describes aspects of a visual language which would meet 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 [13] and OpShop [14] There are also several general purpose visual languages which can be used for image processing, including LabView [15] and Simulink [16]. 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 systems such as Celoxica's PixelStreams [17] and Xilinx's DSP block sets [18] have been developed to provide faster development time for projects and provide similar functionality to Khoros. PICSL also allows for the design and synthesis of digital ICs from data flow diagrams [19, 20].

These languages all follow a form of the dataflow paradigm where streams of data flow through a network of nodes, each performing a computation on the tokens in the stream before passing the result to the next node [21]. It has been noted [22] that dataflow graphs (the natural visual representation of this paradigm) are an effective representation for DSP (Digital Signal Processing) problems, because they are a natural representation for many DSP researchers, and because they expose parallelism in the algorithm with limited constraints on evaluation order. None of these languages deal with pipelining. As they do not allow the editing of their IP blocks, the pipelining is hidden from the developer.

## VERTIPH

As discussed previously, textual languages represent concurrency and complex scheduling poorly. Being Image processing developers who design for FPGA devices, we have developed a system which improves on the limitations we have identified with present tool. We have developed the visual programming language VERTIPH, which provides multiple views of parallel image processing algorithms. An *architectural* view provides a natural representation of the top level data flow aspects of the language. This is augmented with a *computational* view to aid in expressing the parallel nature of the computation within an operation. Finally a *scheduling and resource* view is provided, for specifying inter-operation timing, and for controlling access to resources. A comparison of VERTIPH with other HDLs was presented in [23]. This work expands on VERTIPH's features, concentrating on its recently developed representation of parallel processing and pipelining.

### The Architectural View

The Architectural View (Figure 2) shows the data flow within the overall algorithm, and gives access to the lower-level functional views. Image processing algorithms are broken up into blocks which perform very specific processing tasks, and it is common for these blocks to be developed independently, and validated using test image data. This view allows the designer to construct an image processing algorithm as several blocks that operate sequentially on the image data. Khoros and OpShop act at a similar level.

Processors which are logically related to each other are also encapsulated. For example, a colour segmentation and tracking algorithm detailed in [24, 25] uses bounding boxes for object detection. It incorporates a data structure to hold bounding boxes for each colour class, a processor which uses this to update the bounding box associated with the current input colour label, 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 borrows from object orientated software engineering. However we do not advocate the used 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.

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 in turn can make the scheduling of these processors easier as the developer does not need to remember all the parts of the system that are related to the resource or data structure being used.

Hierarchical encapsulation can allow for very complex image processing blocks to be built, with one block and interfaces representing a complex system of data structures, resources, processors and their scheduled operations or response to events.

The aim of the Architectural View is to allow the architecture of the algorithm to be represented as a data-flow graph. It provides logical separation of operators and

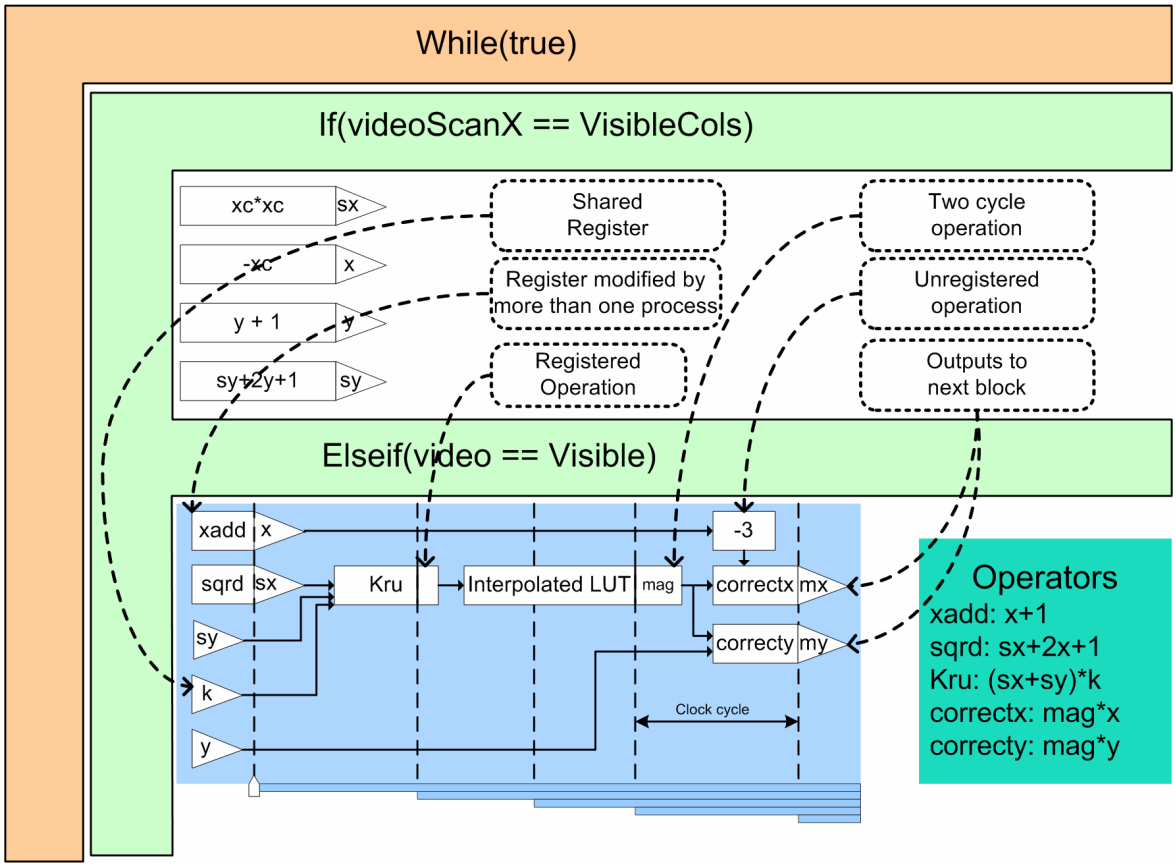


**Figure 3: Computational view of Barrel distortion block showing control functions, timing and operation representation. Note that text in dashed boxes are comments added to the figure for clarification. They do not form part of the language**

hides the lower level details by the encapsulation of data and processors related to that operation.

VERTIPH developers who never design their own algorithms can use the Architectural View to assemble pre-defined library modules into a high-level overview like the one shown in Figure 2. This is similar to the way that other image processing based systems such as Celoxica's PixelStreams and Xilinx's DSP block sets operate.

### Scheduling View

In an embedded image processing system using FPGAs 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. VERTIPH facilitates resource sharing by encapsulating resources and the processors that act on them, so that the processors and their access to resources can be scheduled. This also allows for both global scheduling for processors and for local scheduling within components. This view is discussed in [23] and is an extension to the control structures found in the Computational View.

### Computational View

To allow developers 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 [6]. In this notation, time flows from left to right, so Figure 4 (a) shows a sequential set of operations; operation **`A`** is followed by operation **`B`**, which is followed by operation **`C`**. In Figure 4 (b), the operations execute concurrently.

Of course, these basic types can be used together as shown in Figure 3, which is the pipeline for row processing used by the barrel distortion algorithm [26]. This figure also shows the **`if-`** and **`while-`** control structures provided by VERTIPH, which are based on the control structures used in NS (Nassi-Shneiderman) diagrams [27].

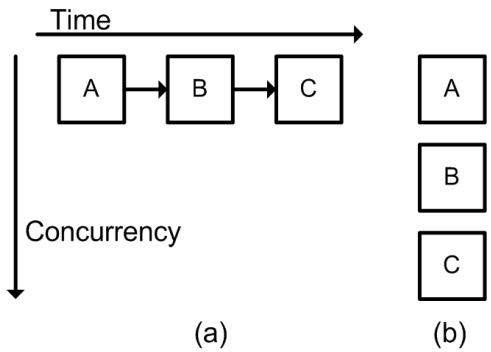


**Figure 4: Process representations: (a) Sequential, (b) Parallel**

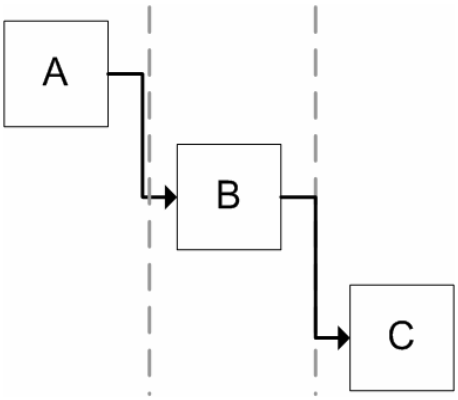


**Figure 5: In the Diagonal Gantt each pipeline stage is translated across and down one space from the previous stage.**

These visualisations of control structures closely follow standard textual layout, but they highlight the extent of loops and choice constructs while they dispense with the need for explicit **begin-end** brackets. The **if-** construct does not have two horizontally adjacent fields containing the **then-** clause and the **else-** clause, as in NS diagrams. Instead, the clauses are aligned vertically, as they would be in conventional textual languages. This is for familiarity and to allow time to increase from left to right across the diagram. The top bar in Figure 3 displays the control expression for the structure, with the vertical bar enclosing the controlled processors. This pipeline view graphically conveys to the developer the time required to prime and flush the pipeline.

When a hardware component has performed a calculation, the resulting value is sometimes stored in a register, and sometimes transferred directly to the next component. These modes of operation are respectively termed registered and unregistered. To save space on the screen, only the operation or register name is shown; an operation's key has the instructions for the block in a C-type syntax. This view shows the same information as a textual language but the layout is intended to make the structure of the algorithm easier to visualise. For example it is easy to see that the x value must be offset by 3 before it is used in the calculation of the undistorted x value.

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

## PIPELINE VISUALISATIONS

### The Diagonal Gantt

A Gantt chart is a natural representation for sequential and concurrent processors. However, it needs to be extended to handle pipelining. A pipeline contains both serial and parallel aspects. Data is processed serially by a set of operations or processors, however each processor is working in parallel on data at different stages of processing. Extending the ideas shown in Figure 4 led to a diagonal Gantt representation.

Figure 5 shows such a three-stage pipeline. A pixel enters processor A on clock pulse 1, and moves to processor B

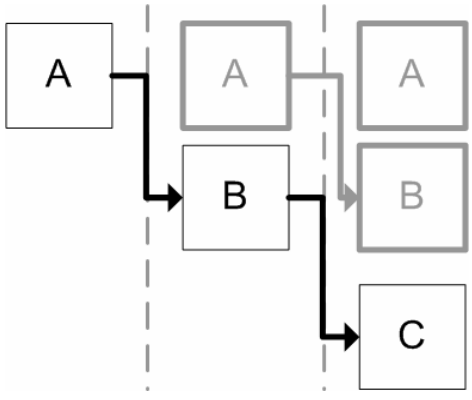


**Figure 6: The Staggered Gantt illustrates that the pipeline is active on successive clock pulses.**

on clock pulse 2, as a new pixel enters processor A. On the third clock pulse, a third pixel enters processor A, the second pixel moves to processor B, and the first pixel moves to processor C. In the overall diagonal structure of the representation, B is displaced horizontally with respect to A to indicate that the data reaches component B after it reaches component A, and B is displaced vertically with respect to A to indicate that the pixel in B is being processed in parallel with another operation (A working on the next pixel).

### The Staggered Gantt

This Diagonal Gantt is not very compact and although it uses the vertical dimension to indicate concurrency, it is not intuitively obvious that on clock cycle 2, A and B are processing data simultaneously, and that on clock cycle three, all three processes are active. The next visualisation dealt with the latter problem, but not the former. Figure 6 shows this representation, the Staggered Gantt, in which the data flow is again shown as a diagonal path through the diagram, but concurrent processors are explicitly stacked above each other, as they start operation. Processors associated with later pixels are shown in grey, so that the data path followed by the first pixel is clearly identified, and to emphasise that the pipeline does not comprise two or three separate sets of hardware.

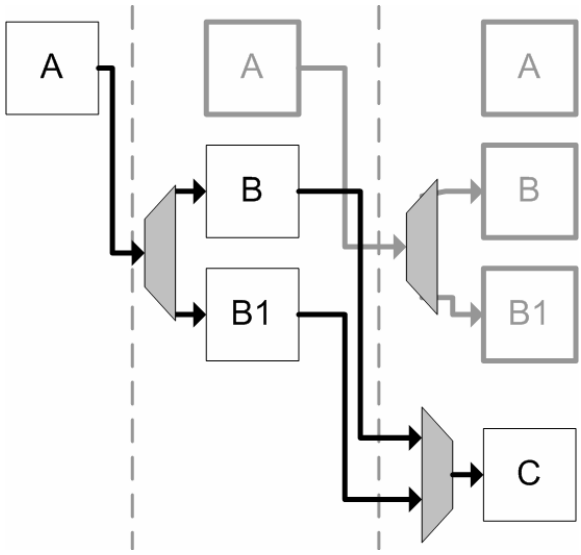


**Figure 7: The Staggered Gantt with conditional branching.**

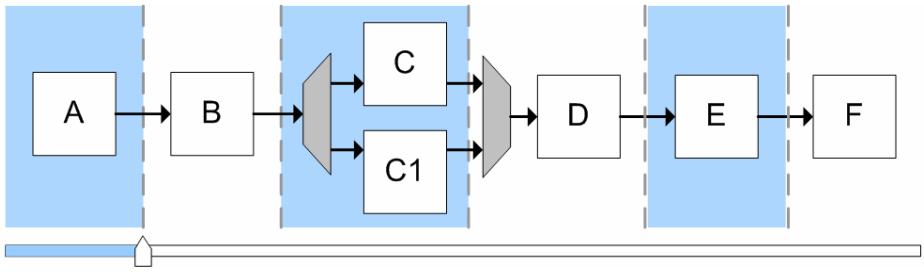


**Figure 8: The Sequential Pipeline view uses coloured bars to show the "extent" of each pixel. Here data arrives every clock cycle**

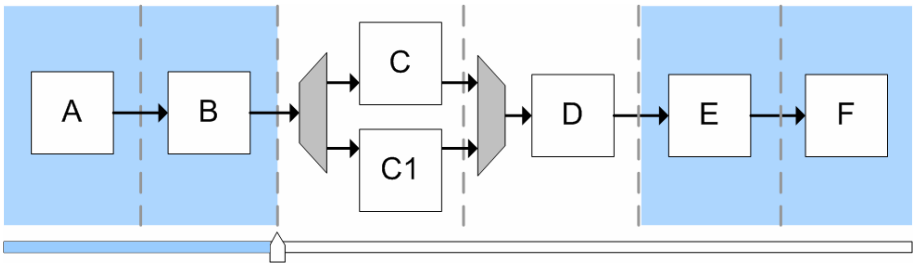


**Figure 9: The user drags the pointer one clock cycle to the right to indicate that data arrives every second clock cycle**

The Staggered Gantt view also has limitations; it does not display pipelines with complex branching very well, as illustrated by the interleaved processors in Figure 7. In this figure multiplexors and demultiplexors are used to select between different branches in the pipeline to be executed. This makes the diagram very complex and hard to understand. It is also implied that new pixel data arrives on successive clock cycles. It can be desirable to operate the FPGA at a clock rate faster than the data rate, because this increases the throughput of the pixel data with respect to the pixel data clock. This can be done by moving the second A processes to the next column, thereby indicating that it gets new data on clock cycle three. This however also adds to the complexity of the diagram.

### The Sequential Pipeline

To resolve these issues we abandoned the idea that the data must always enter the pipeline in the top row of the diagram (which leads to the diagonal datapaths seen in the first two visualisations). Instead, we treated the pipeline as a sequence of processors into which data is injected at regular intervals and in which – once the pipeline has been primed – the processors run in parallel. Visually, this change in emphasis results in a pipeline that extends from left to right across the diagram rather than diagonally. To distinguish a pipeline from a set of sequential processors, highlighting is used to show when new data arrives, with a changes in colour indicating successive data items. A slider is used to set the pixel clock rate relative to the data clock. Figure 8 shows the Sequential Pipeline with data arriving every clock cycle and Figure 9 shows the view with data arriving every second clock cycle.

Although this view is compact, and shows the position (and temporal separation) of successive pixels in the pipeline, the fact that the processors are operating concurrently is not intuitively obvious.

### The Sequential Pipeline with Staggered Bars

The next view once again uses the vertical dimension to suggest simultaneously active processors. It shows one fully detailed diagram of the pipeline in a coloured region, and a number of replicates of the pipeline as coloured, staggered bars, devoid of detail, underneath. These are staggered to indicate the inter-pixel arrival delay. When the user drags the first bar to the right, increasing the inter-pixel arrival delay, the number of bars that can fit into the available horizontal space reduces, and the height of the vertical stack of bars provides a visual indication of the number of pixels that exist in the pipeline simultaneously; that is, it indicates pipeline's data throughput. It also gives a more intuitive indication of inter-pixel delay than the coloured shading in the sequential pipeline. Figure 10 illustrates this view for data with an inter-pixel delay of 1 and 2 clock cycles respectively.

### The Sequential Pipeline with Detailed Bars

Although the Sequential Pipeline with Staggered Bars has several desirable features, it does not provide as good a visual metaphor for simultaneously active processors as the Staggered Gantt. The Staggered Pipeline with Detailed Bars (Figure 11) restores this property to the interface.

This final view achieves a balance between complexity and expressiveness. The bars show that the pipeline will accept new data, allow the user to specify that data will arrive on only some phases of the processing clock and to specify which phases those will be. This was not possible in the first two views where data arrived every clock cycle. The bars also show when all of the processors will have valid data and give an indication of throughput. It also allows for more complex pipelines by allowing conditional execution of processors though multiplexors, and it shows that later pixels are being processed by the same hardware as the first pixel.

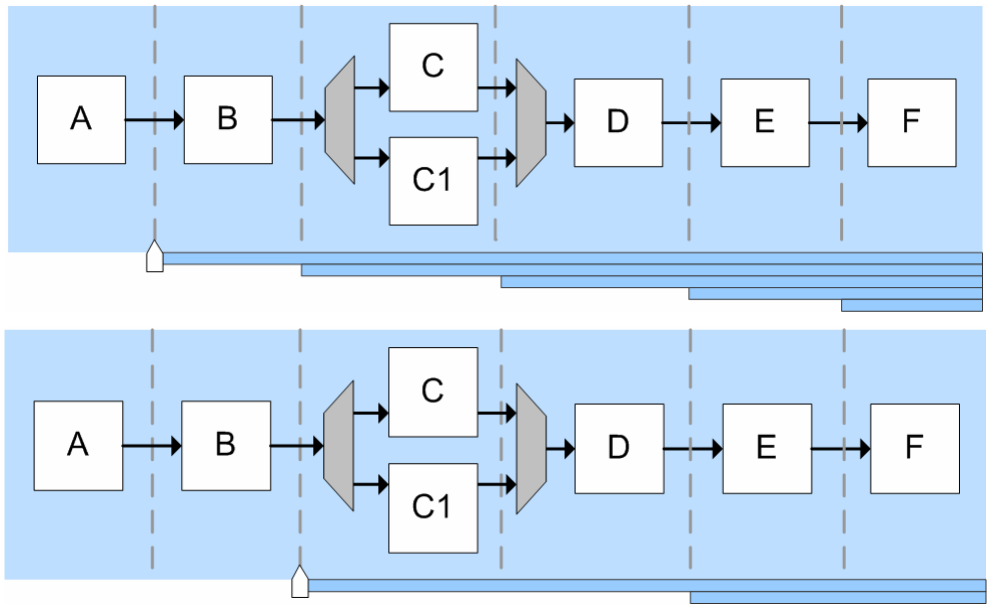


**Figure 10: The Sequential Pipeline with Staggered Bars clarifies the relationship between the pipeline clock and the pixel interarrival delay: The top view shows data arriving every clock cycle, whereas in the bottom view, the control has been dragged to the right to indicate that data arrives every second clock cycle**

## DISCUSSION

We have presented a number of possible solutions to graphically representing pipelining in a static manner for a visual programming environment. The staggered Gantt and the Sequential Pipeline were expensive in terms of screen real-estate and could not adequately describe all possible pipelining options.

The Sequential Pipeline used the fact that a pipeline is a special case of a sequential design. This is a much more compact representation and can clearly show pipelines with branching. However, in this view it is hard to visualise when new data enters the pipeline.

We solved this with the Sequential Pipeline with Staggered Bars, which has bars that show the start points of the pipeline. This has the added benefit of giving an indication of the data throughput. The Sequential Pipeline with Detailed Bars adds internal structure to the bars and makes it clear that they represent a minimised view of the same hardware as the fuller diagram above.

As this work is still in development no user evaluation has presently been done. This will be preformed once implementation of the design is complete.

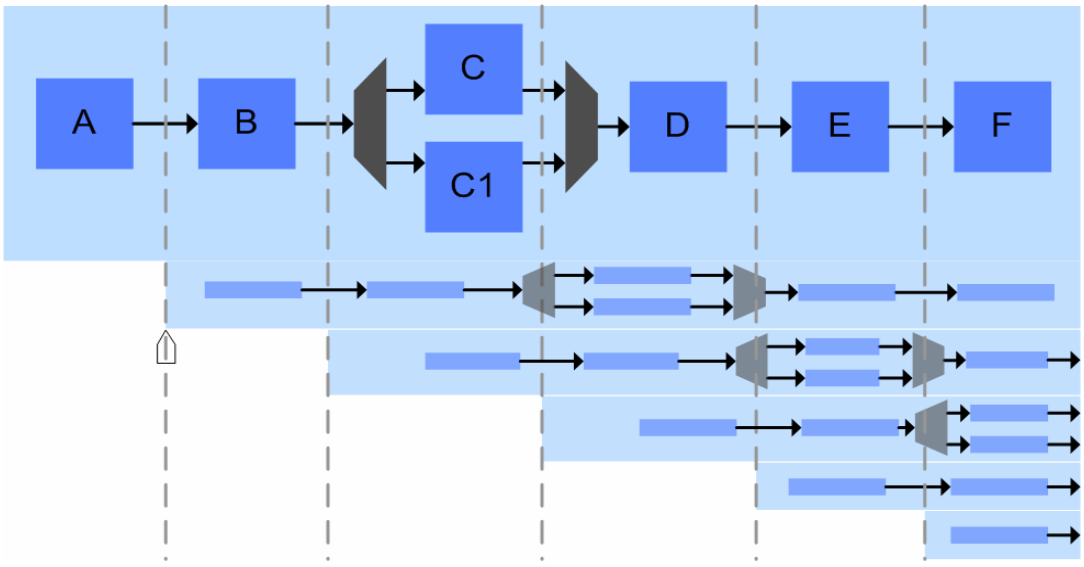


**Figure 11: The Staggered Sequential Pipeline with Detailed Bars shows how the pipeline operates on successive pixels**

## REFERENCES


1. J. Villasenor and B. Hutchings, The flexibility of configurable computing. *IEEE Signal Processing Magazine*, **15,** 5 (1998), 67-84.
2. K. T. Gribbon, C. T. Johnston and D. G. Bailey. Using Design Patterns to Overcome Image Processing Constraints on FPGAs. In *3rd IEEE International Workshop on Electronic Design, Test, and Applications (Delta 2006)* (2006).
3. 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*, **3655,** (1999), 12-20.
4. C. T. Johnston, K. T. Gribbon and D. G. Bailey. Implementing Image Processing Algorithms on FPGAs. In *Proceedings of the Eleventh Electronics New Zealand Conference, ENZCon'04* (2004), 118-123.
5. R. J. Offen, *VLSI Image Processing*. 1985, Collins: London. p. 326.
6. J. R. Schermerhorn, *Management*. Sixth ed. New York: John Wiley & Sons, 2001.
7. IEEE Standard Verilog Hardware Description Language, http://www.verilog.com/IEEEVerilog.html, visited on August 2004
8. J. Bhasker, *A VHDL Primer*. Third ed. Modern Semiconductor Design Series. New Jersey: Prentice-Hall, 1999.
9. I. Alston and B. Madahar, From C to netlists: hardware engineering for software engineers? *Electronics & Communication Engineering Journal*, **14,** 4 (2002), 165-173.
10. R. Rinker, J. Hammes, W. A. Najjar, W. Bohm and B. Draper. Compiling image processing applications to reconfigurable hardware. In *Proceedings. IEEE International Conference on Application-Specific Systems, Architectures, and Processors* (2000), 56-65.
11. J. Hammes, B. Rinker, W. Bohm, W. Najjar, B. Draper and R. Beveridge. Cameron: high level language compilation for reconfigurable systems. In *Proceedings International Conference on Parallel Architectures and Compilation Techniques* (1999), 236-244.
12. 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. In *2000 IEEE Symposium on Field-Programmable Custom Computing Machines* (2000), 39-48.
13. K. Konstantinides and J. R. Rasure, The Khoros software development environment for image and signal processing. *IEEE Transactions on Image Processing*, **3,** 3 (1994), 243-252.
14. P. M. Ngan, *The Development of a Visual Language for Image Processing Applications*, PhD in *Computer Science*, Massey University: Palmerston North. 1992.
15. National Instruments LabVIEW, www.ni.com/labview, visited on 16 February 2005
16. Matlab and Simulink, http://www.mathworks.com/, visited on 12 February 2005
17. Celoxica, *PixelStreams Manual*. 1 ed. Platform Developer's Kit: Celoxica, 2005.
18. Xilinx System Generator for DSP Blockset, http://www.xilinx.com/products/software/sysgen/blockset.htm, visited on November 2005
19. M. W. Pearson and P. J. Lyons. Using Object Flows to specify Communication in a Visual HDL. In *The Fourth Asia-Pacific Conference on Hardware Description Languages (APCHDL '97)* (1997), 32-37.
20. M. W. Pearson, P. J. Lyons and M. D. Apperley, High-level Graphical Abstraction in Digital Design. *VLSI Design*, **5,** 1 (1996), 101-110.
21. J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, PhD in *Electrical Engineering and Computer Sciences*, University of California: Berkeley. 1993.
22. J. T. Buck and E. A. Lee. Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing* (1993), 429-432.
23. K. T. Gribbon, C. T. Johnston and D. G. Bailey. Formalizing Design Patterns for Image Processing Algorithm Development on FPGs. In *IEEE Tencon* (2005).
24. C. T. Johnston, D. G. Bailey and K. T. Gribbon. Optimisation of a colour segmentation and tracking algorithm for real-time FPGA implementation. In *Image and Vision Conference New Zealand* (2005), 422-427.
25. C. T. Johnston, K. T. Gribbon and D. G. Bailey. FPGA based Remote Object Tracking for Real-time Control. In *International Conference on Sensing Technology* (2005), 66-72.
26. K. T. Gribbon, C. T. Johnston and D. G. Bailey. A Real-time FPGA Implementation of a Barrel Distortion Correction Algorithm with Bilinear Interpolation. In *Proceedings of Image and Vision Computing New Zealand* (2003), 408-413.
27. Nassi I and Shneiderman B, Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, **8,** 8 (1973), 12-26.