VERTIPH:
A Visual Environment for Real-Time Image Processing on Hardware

A thesis presented in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

in

Computer Systems Engineering

at Massey University, Palmerston North,
New Zealand.

Christopher Troy Johnston

2009

Abstract

This thesis presents VERTIPH, a visual programming language for the development of image processing algorithms on FPGA hardware. The research began with an examination of the whole design cycle, with a view to identifying requirements for implementing image processing on FPGAs. Based on this analysis, a design process was developed where a selected software algorithm is matched to a hardware architecture tailor made for its implementation. The algorithm and architecture are then transformed into an FPGA suitable design. It was found that in most cases the most efficient mapping for image processing algorithms is to use a streamed processing approach. This constrains how data is presented and requires most existing algorithms to be extensively modified. Therefore, the resultant designs are heavily streamed and pipelined.

A visual notation was developed to complement this design process, as both streaming and pipelining can be well represented by data flow visual languages. The notation has three views each of which represents and supports a different part of the design process. An architecture view gives an overview of the design's main blocks and their interconnections. A computational view represents lower-level details by representing each block by a set of computational expressions and low-level controls. This includes a novel visual representation of pipelining that simplifies latency analysis, multiphase design, priming, flushing and stalling, and the detection of sequencing errors. A scheduling view adds a state machine for high-level control of processing blocks. This extended state objects to allow for the priming and flushing of pipelined operations.

User evaluations of an implementation of the key parts of this language (the architecture view and the computational view) found that both were generally good visualisations and aided in design (especially the type interface, pipeline and control notations). The user evaluations provided several suggestions for the improvement of the language, and in particular the evaluators would have preferred to use the diagrams as a verification tool for a textual representation rather than as the primary data capture mechanism.

A cognitive dimensions analysis showed that the language scores highly for thirteen of the twenty dimensions considered, particularly those related to making details of the design clearer to the developer.

To my family

Acknowledgements

Thanks to my supervisors Donald and Paul for all the advice and encouragement that you gave me.

Thanks to my office mates for the interesting discussions, especially Kim for being a sounding board for ideas.

Thanks to my friends and flatmates for keeping me sane and making sure I got out once in a while, especially Andy, Jess, Øyvind, Reuben and Sarah

I would like to thank my family for everything, without you I would not be the person I am today. Though you did not live to see me achieve this work thanks to Grandma and Granddad for the Lego that encouraged me into engineering and to Nana for encouraging me to think and go as far as I could. Thank you Pop for letting me build things and take them apart and for all the wood over the past years to keep me warm in winter.

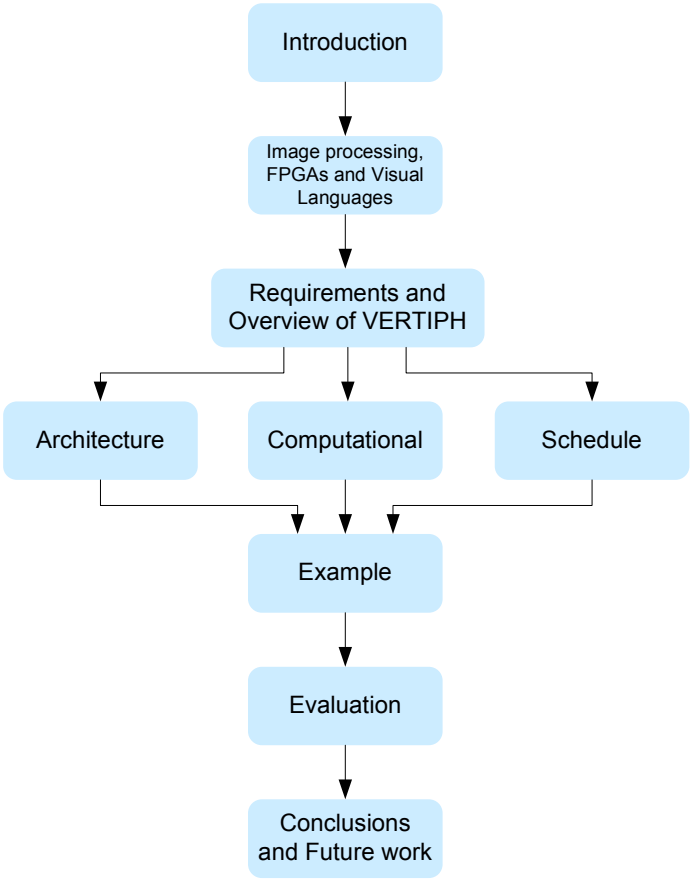
To Nic and Bridget thanks for the dinners and proof reading. AJ for just being you.

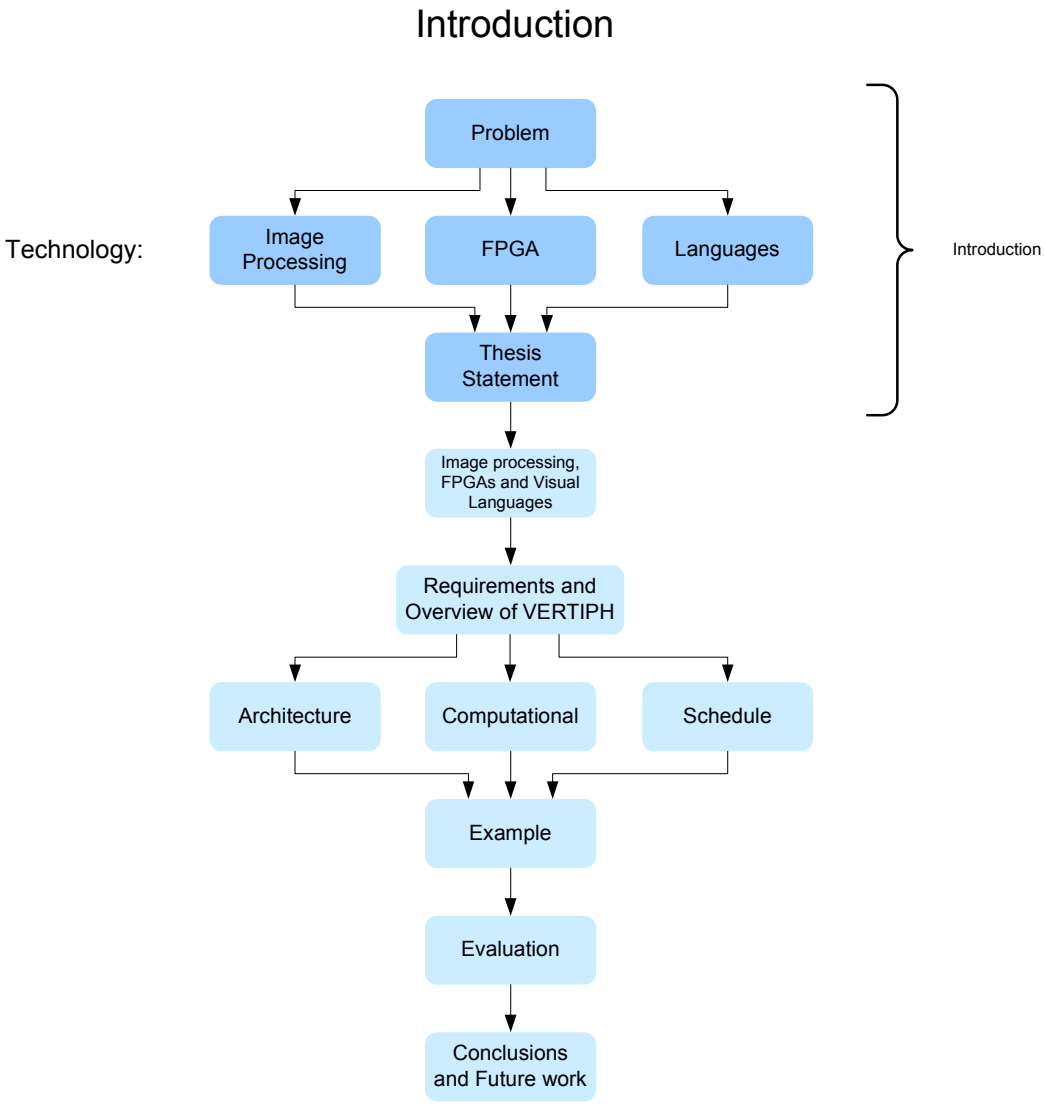
To Mum and Dad thank you for all the support and encouragement with out it I would not have got this far.

Table of Contents

List of Figures

Thesis Structure




Introduction
Technology:
Problem
Image Processing
FPGA
Languages
Thesis Statement
Image processing, FPGAs and Visual Languages
Requirements and Overview of VERTIPH
Architecture
Computational
Schedule
Example
Evaluation
Conclusions and Future work
Introduction

1 Introduction

The motivation for this thesis is derived from my earlier experiences with implementing image processing algorithms on FPGAs (Field Programmable Gate Arrays). Firstly, I felt that existing design tools seemed to be quite difficult to use. Present tools can hinder the translation of a conceived design into a practical implementation by obscuring the basic design with a lot of low level details. Secondly, I felt that it would be possible to invent an alternative type of design tool that would make it easier to map and understand the design than existing tools. This thesis will examine my experiences with implementing a range of image processing algorithms onto FPGAs and use these to highlight and explain some of the problems and pitfalls that can occur with algorithm design on FPGA. This work concentrates on synchronous designs not asynchronous (without a global clock) designs as in most embedded systems there is a data clock for arriving pixel data and for the output (often to a screen). Though these clocks might be different and the intermediate steps could be asynchronous the increased difficulty of designing an asynchronous system has lead to only synchronous designs being considered. Insights from these examples are used to specify the requirements for a visual language to express image processing on FPGAs.

An engineer who is designing image processing algorithms on FPGAs needs to work out a number of design issues. Details will not be provided in this introduction but will be expanded on in later sections. However, some of the key issues include mapping of image processing algorithm onto hardware, modular design, and meeting timing constraints.

Mapping an image processing algorithm on to FPGA hardware is not straightforward. Most image processing algorithms are expressed in a serial manner which does not exploit the parallelism available in an FPGA implementation. The experience of the research group I work in, and through my own work suggests that the best way to implement many real-time image processing algorithms is to use a streamed processing approach. This leads to algorithms which are capable of being pipelined and require limited or no random access to image data. Present languages do not support this approach in a clear manner. This thesis aims to develop a language which does.

A design needs to be broken up into modular blocks which can be worked on and tested independently. This is common for most image processing and hardware design problems. Breaking up a design in an appropriate way is a skill that is learned

over time. However, having a tool that encourages a modular design approach can encourage the designer to consider modular design while in the planning stages.

As we are implementing algorithms onto physical hardware we need to conform to timing constraints. The propagation delay of the logic needs to be shorter than the clock period for synchronous designs. In practice, this requires image processing (and other complex) algorithms to be pipelined. Pipelining can be difficult and error prone, particularly when managing timing issues where data generated in one part of the algorithm is used in multiple places. To aid in the design and debugging of my final year project and other image processing algorithms a number of timing diagrams were used to express the algorithm dataflow and the pipeline on paper. This thesis is about formalising those diagrams as a visual language.

This thesis is at the interface of three major topics: image processing, FPGA or hardware design, and human computer interaction (HCI) specifically visual language design. In particular, this work looks at how a visual language can aid in the design of suitable image processing algorithms for FPGAs. It therefore covers both the process of how to develop image processing on FPGAs and the design of VERTIPH, a visual language designed to support this process.

1.1 Image Processing

Digital image processing is a large subject area and has different meanings and area of scope for different people. It can be considered either a subset or superset of signal-processing. Russ defines image processing as being used for two different purposes (Russ, 2002):

1. Improving the visual appearance of images to a human viewer
2. Preparing images for measurement of the features and structures present.

Computer vision can be defined as the application of image processing algorithms within a vision capture and analysis system, including the camera, image capture and storage system and processing device. Computer vision normally includes the extraction of features from the image and an attempt to make “sense” of the image. This is then used for some action including control, animation, as a control for a user interface, and surveillance. Machine vision can be defined as application of image processing and computer vision in industrial settings.

Real-time processing can be subdivided into hard and soft real-time systems. For hard real-time systems the completion of an operation after a deadline is

considered a failure while soft real-time allows lateness to be tolerated, and may result in degradation of service (dropping of frames) or - if input data is buffered until processing can be done - increasing the system latency. In general the longer it takes to process the image, the less time is available to perform other tasks by the system as a whole. Most image processing systems are classified as real-time if they are able to process the image (or frame in video stream) at the rate it arrives. The image may be buffered; this will increase the latency to a frame delay, plus the processing time, but if the throughput of the system is not reduced, its classification as real-time remains valid.

For this research the term image processing includes all the computational steps in analysing images, but excludes the lighting set up, camera selection (including resolution and colour depth) and control of external devices.

Therefore, this thesis defines image processing to include these three alternatives:

- improving the visual appearance of images to a human viewer
- preparing images for measurement of the feature and structures present.
- extracting some desired information from an image or video stream

1.2 Field Programmable Gate Arrays

1.2.1 What are they?

FPGAs (field programmable gate arrays) are reconfigurable hardware devices. They allow a wide variety of digital logic to be designed and implemented. However, this is subject to the availability of resources on the FPGA. Field programmable gate arrays combine the speed of hardware with the flexibility of software programming. They fit between dedicated hardware, such as ASICs (application specific integrated circuits), and highly parallel multi-core processors.

As shown in Figure 1.1 an FPGA features an array-like architecture with a matrix of configurable logic linked with interconnections by programmable switches. These programmable switches and user configurable logic allow for almost any digital logic to be implemented. Other parts of the device are used for dedicated RAM, and for accessing I/O pins. On some higher end FPGA devices, a microprocessor might be implemented on part of the device instead of configurable logic.

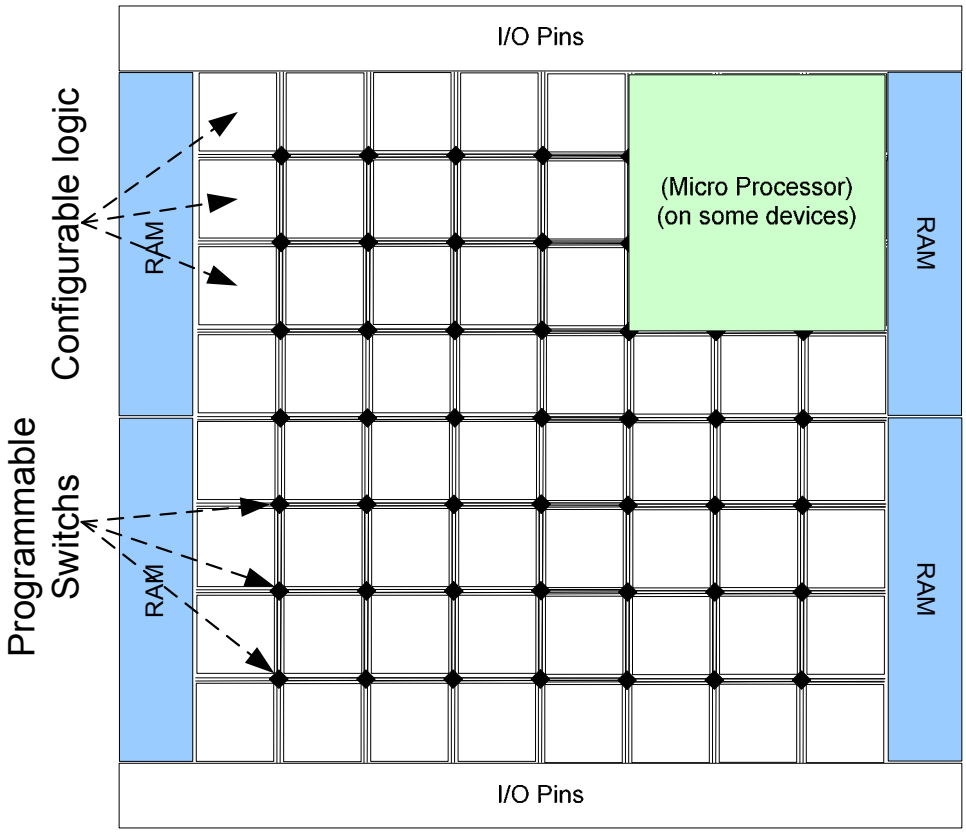


Figure 1.1: Block representation of a FPGA layout

Xilinx was the first to introduce these devices in 1985 and offer a number of families of re-programmable, static memory based FPGAs. The blocks of configurable logic (CLBs) have a lookup table for implementing logic (in most devices this has 4 or 6 inputs), carry and control logic and a flip flop as shown in Figure 1.2

Block RAM comprises small blocks of RAM that can be used as on chip RAM or ROM for use as memory elements, buffers, registers or tables. Memory functions can also be constructed out of the logic units. Memories constructed from the different blocks have different characteristics. Logic based memory (fabric RAM) allows for multiple access to elements and has short propagation delays. These benefits are paid for by a reduction on the number of logic blocks available for computation. On most products the block RAM is dual-port, allowing one read and one write or two reads per clock cycle.

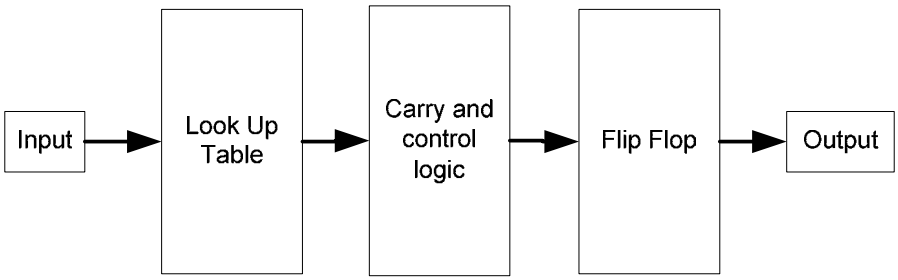


Figure 1.2: Common parts of a block of Configurable Logic

The desired hardware is produced on an FPGA by configuring the logic functions and their interconnections using programmable internal static memory cells. The logic also needs to be mapped onto the CLBs. Simple functions might be implemented in a single CLB but for complex functions like adders or multipliers, the logic may occupy several CLBs. Finally, I/O pins need to be configured for input, output or tri-state operation, and the appropriate signals need to be routed from the pins to the logic.

Once this hardware is mapped and routed for the FPGA, a file is created to configure the device (called a bitfile). The bitfile is stored into the static RAM of the FPGA, and controls the functions of the CLBs, IOBs, and the connections made by the interconnecting switch matrix.

Most of this low level mapping is performed by the FPGA vendor's design tools. FPGAs are designed to be flexible to support a large range of hardware tasks and operations and they are thus very complex devices.

1.2.2 Appropriate for Image Processing

Image processing is computationally intensive. Most useful image processing algorithms require a large number of operations; each operation can require a large number of arithmetic or logic steps.

Image processing is also data intensive. A real-time image processing algorithm will have to operate on a large amount of pixel data. At video rates of 25 frames per second a single operation performed on every colour component of every pixel of a 768 by 576 colour image (a PAL frame) equates to 33 million operations per second. This does not take into account the overhead of storing and retrieving pixel values. Many image processing applications require that several operations be performed on each pixel in the image, resulting in an even larger number of operations per second. On a modern PC with a high clock-speed it is possible to perform many complex image processing operations at real-time rates, although the

processor is often stalled as the large data throughput and the often large instruction window are not suited to modern superscalar processors (Ratha and Jain, 1999).

Embedded systems often have to be small and low power. Lower speed processors are used, and the complexity of the operations that can be preformed decreases correspondingly. Embedded systems with gigaHertz processors are available but, due to their high clock speed and the size of the transistor features they use a large amount of power, much of which is radiated as heat. FPGA hardware can operate at much lower clock frequencies by performing more tasks in parallel; this results in lower power requirements (although not as low as can be achieved with dedicated ASICs). An FPGA can also be used to perform preprocessing to reduce the volume of data by performing some of the data-intensive and or computationally intensive tasks before passing the result on to a microprocessor (either externally or built on the FPGA). This works based on the idea that in image processing, operations can in general fit into three categories or levels.

(Downton and Crookes, 1998) have suggested three categories of image processing operations:

- *low-level* operations take a pixel image and generate another pixel image. Examples include contrast adjustment, edge detection and filtering. Data is typically geometric, regular and mainly local.
- *medium-level* operations take a pixel image and generate a set of features. Examples include histograms, detection of connected (single-colour) regions, Hough transforms. Feature data is usually symbolic and non-local.
- *high-level* operations. These take a set of features and generate a set of objects. Data is symbolic and complex and the processing is often model-driven.

This idea is illustrated in the pyramid, Figure 1.3, where as the image moves from low- to high-level processing, the amount of data to process reduces and the knowledge about the image increases.

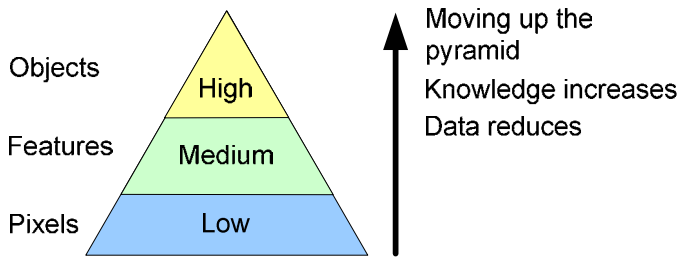


Figure 1.3: Image processing pyramid

As we move from low- to high-level, the level of data parallelism that can be exploited from the algorithm reduces, due to the decrease in the amount of data to process, and the fact that the data becomes less regular in its structure. At the higher levels the amount of data to be processed is reduced but the algorithms used to analyse the data are more complex.

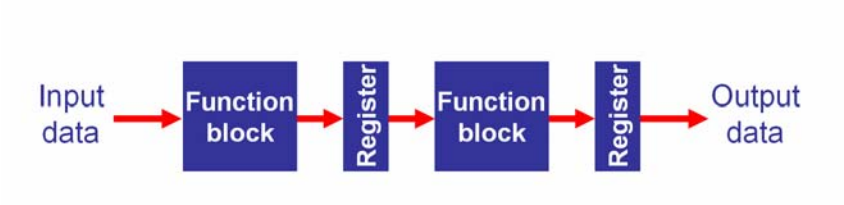


Figure 1.4: Temporal parallelism creates a pipeline

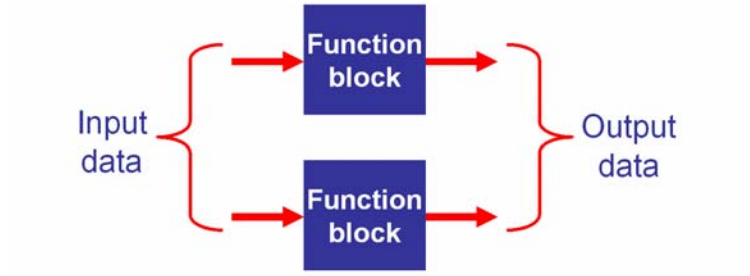


Figure 1.5: Spatial parallelism same operations on different data

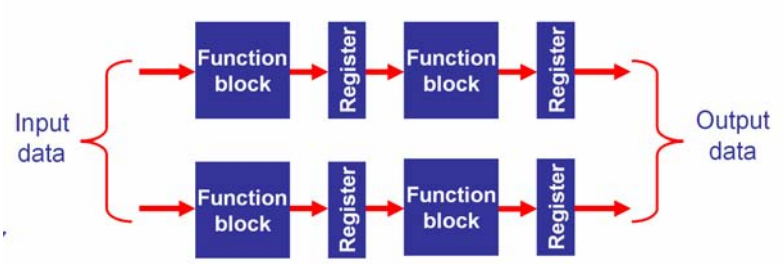


Figure 1.6: A hybrid of temporal and spatial parallelism

High speed operation can be achieved either by using pipelined architectures or by using multiple function blocks that perform the same function on different data sets. Assuming that new data arrives every clock cycle; Figure 1.4 illustrates pipelining (otherwise it has an equivalent structure to sequential processing) and Figure 1.5 repeated function blocks. Note that pipelining increases the latency of the operation, the number of clock cycles taken to produce an output. These two types of parallel design can be combined with multiple pipelines operating on several different data sets as shown in Figure 1.6.

FPGAs are particularly well suited to both low-level and medium-level image processing. Due to the large volume of data to be processed and the regular access

pattern that image data requires, low-level algorithms can be converted into pipelined operations with the image data accessed in a raster manner. Low-level operations may require some limited form of caching (such as row buffering) to access local pixel values on previous rows. Performing a raster scan through the image leads to a streamed processing approach, which is discussed in chapter 2. Medium-level operations can also be implemented with great success on FPGAs. These operations are normally computationally complex and, although the data accesses are not always uniform, they can be accommodated with caching. High-level operations are not normally suited to implementation on FPGAs, as they would require complex hardware which is significantly harder to reuse and difficult to programme. The lower data volume relaxes the timing constraints, making it more efficient to offload these parts of the algorithm to a microprocessor. Higher level operations are also not suitable for streamed processing, as discussed in the second chapter, although streaming can be performed at the feature level, and the higher level operations can be pipelined.

FPGAs offer a great degree of flexibility; almost any architecture can be built. This allows for data widths to be selected to give the optimal trade-off between the size of the design and precision required. The paper by (Bainbridge-Smith, 2005) looks at how to get the optimal size and therefore power consumption for particular requirements. This is an improvement over fixed architectures, where the word size cannot be changed. FPGAs can also be reprogrammed in the field, making them more flexible than ASICs with a penalty in terms of slower speed and increased power consumption. FPGAs also have a shorter development cycle and time to market than ASICs.

1.3 Languages

Image processing development requires an interactive development environment to allow algorithms to be tested. This will often have a large number of library functions and the ability to construct new operations either out of existing operations or by programming new functions which are added to the library. Image processing development environments include MATLAB (MathWorks, 2005a), Khoros (Konstantinides and Rasure, 1994), VIPS (Bailey and Hodgson, 1988) and many others. Once an algorithm has been developed for a particular application, it can be used directly in the development environment, compiled from that environment to an executable or library function, or converted by a developer to another language for target implementation.

The main reasons for converting from a development tool to another language are to allow optimisations to be made, or to move from one hardware platform to another. At present there is no development environment that can be directly converted to run on FPGA hardware (AccelChip (AccelChip, 2005) though, does have a product to take MATLAB code). As FPGAs also have a longer development and test cycle than software (code, test, simulation, place a route, configure, test) most designs are developed in an interactive development environment on a PC. They are then converted via a hardware description language (HDL) to configure the FPGA. More detail on the design cycle and different image processing languages will be presented in the next chapter.

"Programming" hardware is quite different from programming software, in spite of the syntactic similarities between hardware description languages and software languages. All instructions will result in a piece of hardware being built for that instruction even if it is used only once, unlike in software where the instruction is stored in memory. Like parallel software languages, HDLs need to have constructions to deal with parallelism (although they often differ from those in parallel software languages). HDLs also need to have constructions which are not needed in software, to set the sizes of operands and registers and many other hardware specific tasks which are not part of software design (in software these are set implicitly by the hardware architecture or virtual machine the programme is compiled to).

There is a range of hardware development tools, such as schematic capture tools, intellectual property hardware blocks, and HDLs. Hardware circuits may be drawn using a schematic editor. The schematic is visual and familiar to engineers, but it is not algorithmic and can be very difficult to understand and change because the algorithm is implicit in the hardware. It can also be error prone and slow to develop; most people do not think at the logic gate level. Hardware blocks can help with making more manageable designs. Hardware representations based on schematic capture or state transition diagrams are not well suited for expressing complex algorithms.

Current HDLs can be broken down into low-level and higher-level HDLs. Low-level HDLs like VHDL were developed to capture the high-level temporal behaviour of complex digital designs as well as their circuit structure. Industry standard HDLs such as Verilog, and VHDL (Accellera, 2008) can be thought of as the assemblers of hardware programming, providing great flexibility from gate level up to the behavioural level. While HDLs offer considerable flexibility in terms of the control logic, it is up to the designer to explicitly specify the logic required to control

the system. This can allow very efficient control of the execution path, but the developer must design both the data flow and the control flow. Higher-level HDLs such as Handel-C (Alston and Madahar, 2002), ImpulseC (ImpulseC, 2008) and SA-C (Rinker et al., 2000) are closer to software languages and are more abstract from hardware. This makes such languages easier to program in general but they are less flexible. These HDLs either add hardware constructs to a software language, requiring the developer specify hardware related concepts, or incorporate a hardware compiler for a subset of C, which hides parallelism and other hardware design decisions from the developer.

For example Handel-C is a subset of ANSI-C with hardware-oriented extensions for specifying variable data widths, parallel processing and channel communication between parallel processing blocks. However, its textual nature obscures data flow in parallel designs. Sequential and parallel code look virtually identical (see Figure 1.7). This is common in text based HDLs. These languages, and others, will be discussed in more detail in the next chapter.

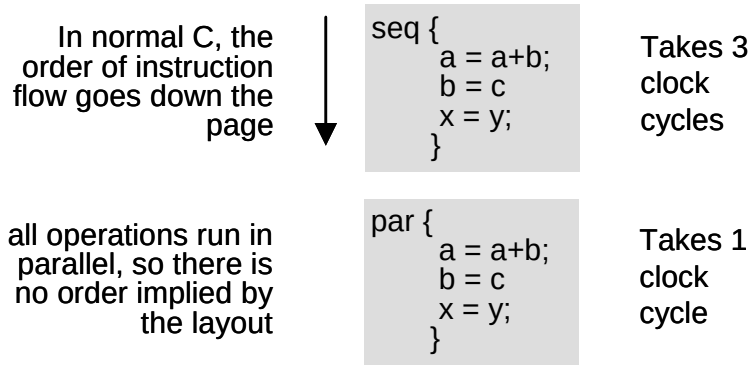


Figure 1.7: Logical flow of instructions in Handel-C

1.4 Thesis Statement

As discussed earlier, the motivation for this work came from my earlier experiences with implementing image processing algorithms on FPGAs. I felt that existing design tools seemed to be quite difficult to use. Present tools can hinder the translation of a conceived design into a practical implementation by obscuring the basic design with a lot of low level details. Therefore, I contend that:

> *it should be possible to invent an alternative type of design tool that would make it easier to understand and map the design of image processing algorithms onto FPGAs than existing tools.*

There are five supplementary research questions which test this statement:

- Can the design process for implementing image processing algorithms on FPGAs be characterised well enough to act as the inspiration for a software tool?
- Is it possible to produce a detailed and complete syntax for a visual design tool based on such a characterisation?
- Is it possible to build an alternative tool based on these analyses?
- Is this tool easier to understand than existing tools?
- Is it easier to map a design onto an FPGA with this tool than with existing tools?

Various practical activities have been undertaken to establish the answers to these questions.

Several published image processing algorithms have been analysed with a view to establishing whether their design processes can be characterised well enough to act as the basis for a software tool[1].

These analyses have also acted as a starting point for the design of a detailed and complete syntax for a visual design tool[2]. The design and syntax of this visual tool

[1] Chapter 2

[2] Chapter 3

is outlined in detail looking at each of the three main components, the Architectural[3], the Computational[4] and the Scheduling views[5].

Two out of three components of an alternative tool based on these analyses have been implemented[6].

At this stage, because the complete tool has not been implemented, it is not possible to test definitively whether or not it is easier to understand or easier to use to map a design onto an FPGA than existing tools. However, the results of a paper-based user evaluation of the design and a qualitative evaluation of the implementation are presented, and a cognitive dimensions analysis of the language is presented[7].

1.5 Thesis Contributions

This thesis presents a number of novel contributions and also builds on other work which the author and others have contributed in other contexts. This includes the analysis of several implementations of image processing algorithms on FPGA (the algorithms have been published previously but the analysis is new material). These algorithms are:

- A barrel distortion correction algorithm (Johnston, 2003, Johnston and Bailey, 2003, Gribbon et al., 2003). This algorithm was developed as a project during the final year project work, in cooperation with Donald Bailey (supervisor) and Kim Gribbon (PhD candidate).
- An object tracking algorithm (Johnston et al., 2005b, Johnston et al., 2005a), with Donald Bailey and Kim Gribbon
- A connected components algorithm (Bailey and Johnston, 2007, Johnston and Bailey, 2008), with Donald Bailey and an improved algorithm (Ma et al., 2008, Bailey et al., 2008) with Donald Bailey and Ma Ni (Intern).

The analyses of these were used to produce a list of requirements for VERTIPH in consultation with my supervisors. It updates and expands the work in (Johnston et al., 2004a, Johnston et al., 2006b).

[3] Chapters 4

[4] Chapters 5

[5] Chapters 6

[6] Chapters 4, 5, and 6

[7] Chapter 8

The thesis contains an extension of the design process originally outlined (Johnston et al., 2004b) and further developed by the other members of our research group in (Gribbon et al., 2007, Bailey, 2007, Bailey and Gribbon, 2005). This is a summary and extension of the information presented at (Bailey and Gribbon, 2005) and in (Bailey, 2007), developed with Donald Bailey and with input from Kim Gribbon.

There are a number of novel contributions presented in this thesis. Each of these are listed below where the contribution followed by a parenthesised list that describes the contributors to the work, the stage of the development and any publications that have resulted.

- VERTIPH, a visual language (VL) for designing image processing algorithms on FPGAs (*designed in consultation with my supervisors Donald Bailey and Paul Lyons; partially implemented; partially evaluated) (Johnston et al., 2004a, Johnston et al., 2006b, Johnston et al., 2006a, Johnston et al., 2008).* The VL approach has not previously been explored in this context. The language treats a design as three separate aspects (summarised in chapter three):
    - an architectural view (*designed in consultation with my supervisors, implemented, evaluated (paper-based and implementation).* This includes a box-and-wire representation (Johnston et al., 2006b) of an algorithm's modules. Unique features of this view include the data type interface (a modification of the one developed by the author and described in (Johnston et al., 2006b)) and the junction box interface (described in chapter four).
    - a computational view (*designed, in consultation with my supervisors; implemented; evaluated (paper-based and implementation)*). The control structures used are based on a modified version of Nassi-Shneiderman diagrams (Nassi and Shneiderman, 1973); extensions that have been developed in this research include explicit representation of timing, separation of pipelines from parallel design and event triggers. In addition the computational view incorporates a newly-designed notation intended to facilitate the design pipelines including the correct control for multiphase pipelines (*designed, in consultation with my supervisors; implemented; evaluated; updates and expands the work in (Johnston et al., 2004a, Johnston et al., 2006b, Johnston et al., 2006a))* (described in chapter five).

- a resource and scheduling view (*designed, in consultation with my supervisors; evaluated (paper-based evaluation); previously presented in* (Johnston et al., 2008). This presents a design for a high-level control system. It allows processors to be scheduled to run during an event state. These processors controlled by the state can then be scheduled to run in sequence, in parallel or as a pipeline with respect to each other. It also helps to highlight the shared resources used (described in chapter six).

The prototyped parts of VERTIPH have been evaluated in the user evaluation. A paper on this evaluation has been accepted for publication and is to appear at CHINZ 2009 (Johnston et al., 2009), and it is described in more detail in chapter eight.

A cognitive dimensions analysis of VERTIPH following the procedures outlined in (Green and Petre, 1996, Green, 1996, Blackwell and Green, 1998) is presented. This work has not previously been published, and is described in chapter eight.

In addition to these practical developments, the thesis contains a summary of image processing languages, hardware design languages and visual languages. The grouping of the hardware design languages based on the level of hardware design control that they give to a developer, and their qualities for aiding image processing implementations, gives a unique insight into the way they can be used (described in chapter two).

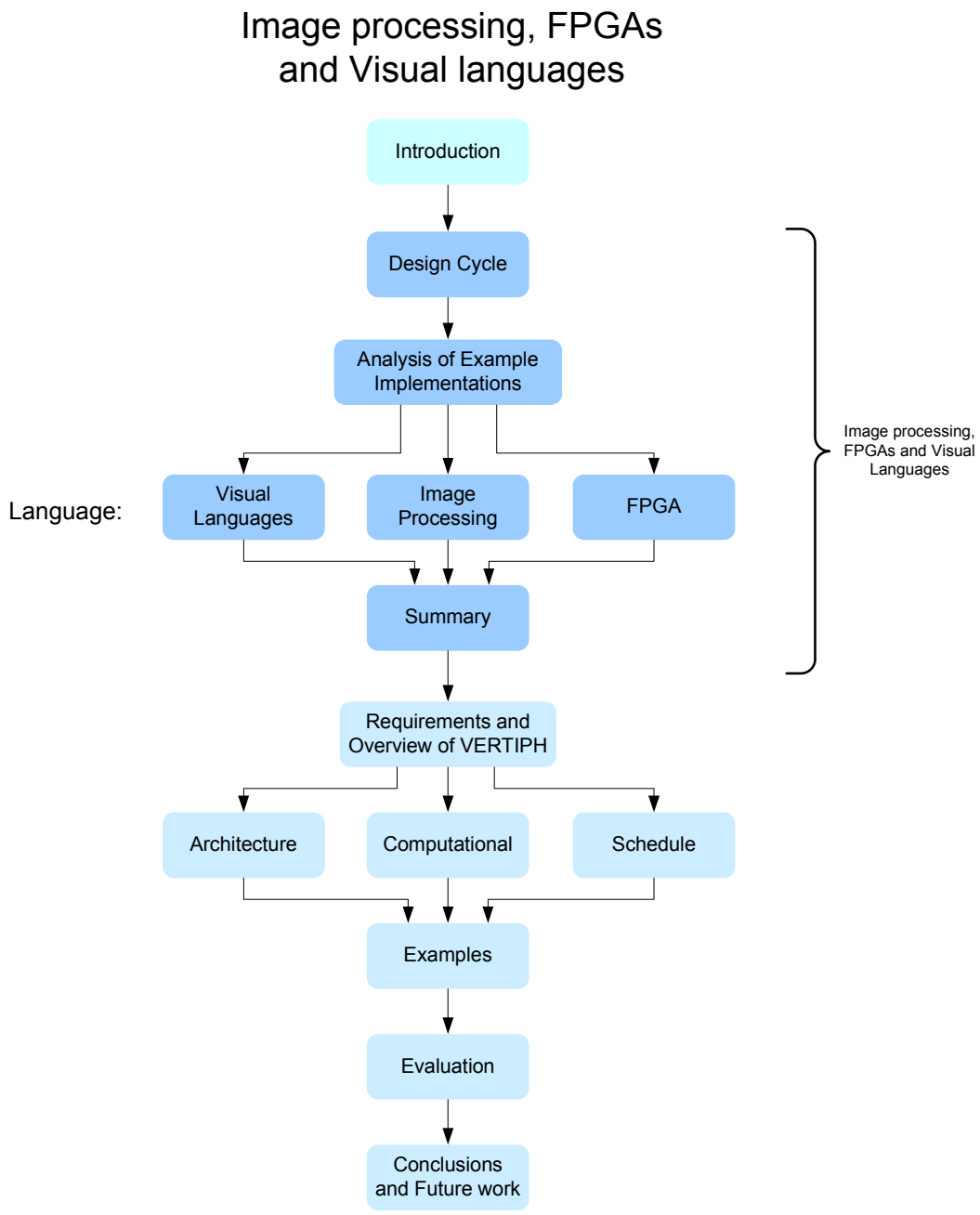


Image processing, FPGAs and Visual languages
Introduction
Design Cycle
Analysis of Example Implementations
Language:
Visual Languages
Image Processing
FPGA
Summary
Requirements and Overview of VERTIPH
Architecture
Computational
Schedule
Examples
Evaluation
Conclusions and Future work
Image processing, FPGAs and Visual Languages

2 Image processing, FPGAs and Visual languages

When image processing algorithms on FPGAs are presented in the literature, the papers usually only describe the final implementation. The development process, if covered at all, is usually given only a brief mention. Therefore, a number of image processing operations and applications were implemented on an FPGA so that experience with the development process might provide insight into suitable language features. These are discussed in the first part of this chapter. Insights gleaned from the literature are also summarised later in this chapter.

Traditionally, image processing algorithms have been primarily implemented in software. As a result, most of the skills pertain more to software engineering than to hardware engineering. There have been many attempts to make implementation of complex algorithms in hardware more accessible to software engineers. This chapter will also review languages for development of algorithms and in particular image processing algorithms on FPGAs. Several visual languages have also been designed specifically for image processing. These will be reviewed at the end of this chapter.

2.1 Image Processing Design Process

Before looking at the examples, it is important to consider the design process for implementing image processing algorithms onto an FPGA. This section presents the design process that the Image and Signal Processing Research Group at Massey University has developed for implementing embedded image processing onto FPGA devices. This section is based on material developed for and presented in (Bailey, 2007).

2.1.1 Design Stages

There are four key design stages (Bailey, 2007, Gribbon et al., 2007, Bailey and Gribbon, 2005):

- *Problem specification* – firstly, the developer defines the problem, provides the context and outlines the set of engineering and functional constraints that need to be met for a proposed solution to be successful.
- *Algorithm development* - next, the developer determines the sequence of image processing operations that transforms the input image to the desired output, fulfilling the image processing requirements in the problem specification.

- *Architecture selection* - then the developer selects the conceptual design and operational structure of the system, including memory architecture, processor architecture, data widths, and hardware interfaces.
- *System implementation* - finally, the developer maps the algorithm onto the selected architecture.

Problem specification is the same for both FPGA and software-based systems. It is necessary to translate the often vague problem description into an engineering requirement specification which can be used to specify and then evaluate the design. This requires determining the image processing requirements and how these fit into the overall system design (including lighting, camera setup, and other engineering parts of the system such as holding an object for inspection). Selection of the image features to be measured is quite task-dependent. This requires a comprehensive knowledge of the problem to ensure valid assumptions are made and a representative set of test images selected for development.

Algorithm development is a problem-solving task aimed at finding a sequence of image processing operations which will transform the image into the desired result. Commonly a heuristic design approach is used for algorithm development (Bailey and Hodgson, 1988, Brumfitt, 1984). An interactive design environment is needed, where the results of applying an operation can be quickly evaluated to see if it is moving towards the final goal of the algorithm. It is also beneficial to have a large number of image processing operations to choose from. This makes interactive systems desirable. MATLAB with the image processing toolbox (MathWorks, 2005a) is popular in the engineering community as it offers a wide range of image processing operations and also allows developers to code their own operations. MATLAB is also matrix-based which maps well to images as they can be treated as a matrix. Other tools are more specialised and contain a large number of image processing operations that can be combined together to find the correct result, either inline or by writing scripts, such as VIPS (Bailey and Hodgson, 1988). As these tools involve the combining of many operations in a sequence, several visual design tools have been developed to allow the linking of operations using a box and wire metaphor; these include OpShop (Ngan, 1992) and Khoros (Williams and Rasure, 1990). One of the more interesting and recent systems to be developed is T-STAR (Cinque et al., 2007) where image processing operations are treated as trees of operations. In this system the different image processing operations the developer proposes to achieve the goal are remembered at each stage. This leads to a system which remembers the choices made (in the heuristic design) and each can be evaluated against later designs so that

the best one can be selected. These different tools will be discussed later in this chapter.

In the context of FPGA based image processing, several points about both FPGA and image processing design need to be made. Firstly, the FPGA development environment is not very interactive. There are several reasons for this. The simulation of a parallel architecture on a serial computer is relatively slow (depending on the complexity of the design, thousands or millions of times slower), especially if clock-accurate simulation is required. Also, the time taken to compile and map the design to hardware is significantly longer (hours, for designs that are close to the capacity of the device) than for software based systems. Secondly, once the design has been mapped it needs to be tested. This introduces problems with the need to develop a "ground truth" to test the algorithm against the image captured and recorded. Evaluating the effectiveness on a continuous real-time video stream is difficult, due to noise and the difficulty of viewing the results of different stages within the algorithm. Such issues require the incorporation of additional diagnostic hardware into the design, which must also be debugged.

This makes using an FPGA design tool for the initial algorithm development undesirable, although it is required for later stages. Instead, the initial design should use an interactive software based image processing system.

A typical sequence of operations used by many algorithms includes:

- Preprocessing - enhancing the required information while suppressing irrelevant information (filtering, contrast expansion, thresholding etc).
- Segmentation - finding regions of interest within an image for further analysis.
- Classification - measuring image features that enable identification of the objects and extracting key data.
- Post-processing - formatting the extracted data into the required form.

Once the type and sequence of operations has been defined, the mapping between the software implementation and the hardware operations can begin.

"*A simple mapping of a software implementation into hardware often falls short of the potential benefits offered by an FPGA solution as they* (sic) *do not tend to leverage concurrency*" (Gribbon et al., 2007). Sequential algorithms must be redesigned to take into account the opportunities for parallelism which can be exploited on FPGAs. Otherwise the design is just a hardware copy of a software algorithm, which will not be as efficient as it could be, and may not perform as well

on an FPGA as an optimised software version on a modern desktop computer or high end microprocessor.

The Image and Signal Processing Research Group at Massey University has found it necessary to design the hardware architecture in the light of the initial algorithm, and then map the algorithm onto that architecture. Mapped operations have the same (or similar) result but with different implementations. The operations need to be compatible with the architecture and the final algorithm must be developed with the computational architecture in mind.

Architecture selection can be considered at several different levels. The *system level architecture* considers the FPGA in relation to peripheral devices (video input etc) and the external memory architecture. The *computational architecture* considers the partitioning of the design between hardware and software implementation. This is important, as in a hardware design the underlying architecture must be designed to match the algorithm being implemented. Unlike a conventional serial processor the computational architecture is not fixed, but needs to be designed and optimised with a particular algorithm in mind. The processing mode also needs to be selected for the parts of the algorithm that are implemented in hardware. There will also be constraints imposed on the algorithm due to real time processing. These will also be affected by the processing mode and the architecture. There are also system level architecture issues which will affect the lower level. These problems occur whether the configuration of the FPGA is a hosted or standalone application as it will affect the mapping of operations.

Processing Configurations

The type of image processing algorithm is affected by the system which it is implemented on. An FPGA can be part of a computing system which has both software and hardware elements. This leads to the need for hardware software partitioning. This is an architecture decision which will affect the algorithm. The FPGA can be in a hosted configuration (Figure 2.1) where the host computer configures the FPGA, stores the image in memory and tells the FPGA to start processing it. In this case the computer is using the FPGA to accelerate the operation (speed up complex tasks so the computer can do other tasks). The host also normally handles I/O and user interaction.

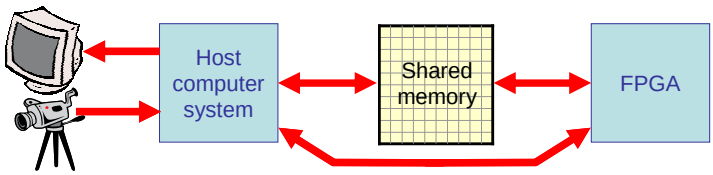


Figure 2.1: Hosted configuration

In a stand alone design (Figure 2.2) the FPGA performs all of the processing; image capture, processing and display and user interaction. There is no operating system which requires hardware to be designed to allow for turning, calibration, user interaction and debugging. This makes the design more complex. Modern FPGA devices often allow the implementation of soft-core microprocessors on the device which can also allow for a fine-grain software/hardware partitioning. These devices may also be capable of running optimised operating systems.

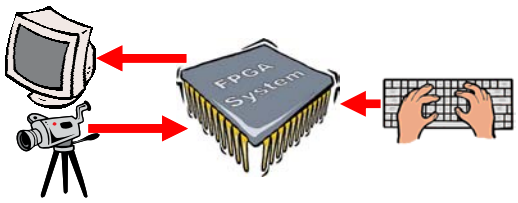


Figure 2.2: Stand alone configuration

For the applications in this thesis we will consider only standalone hardware implementations. This has allowed us to concentrate on the hardware design of the algorithm rather than the communication and partitioning between hardware and software.

Processing Modes

We have identified a number of different processing modes for implementing image processing algorithms on FPGAs (Johnston et al., 2004b, Gribbon et al., 2005, Gribbon et al., 2006). These are *streamed processing*, *random access processing*, and *image partitioning*.

Streamed processing processes the image as a sequential stream of pixels. As seen in Figure 2.3 the image (top of diagram) is accessed by scanning through it in a raster fashion. This converts the spatial parallelism in the image into temporal parallelism within a data stream (bottom of diagram). This is well suited for processing the image on the fly as it is input to the FPGA or output to a display. As one pixel must be consumed or produced every clock cycle, streamed processing constrains memory access. However, the data is presented to the algorithm in a

structured and known format. This processing model is suited to low-level processing which performs point operations and for local filters (with appropriate data caching). For other operations, the standard software algorithms may need to be significantly redesigned to match the constraints of streamed processing. We look at an example of this with connected components analysis. When redesign is necessary, there is often the need to tightly couple the algorithm steps with specialised caching. The acceleration is gained through low level pipelining.

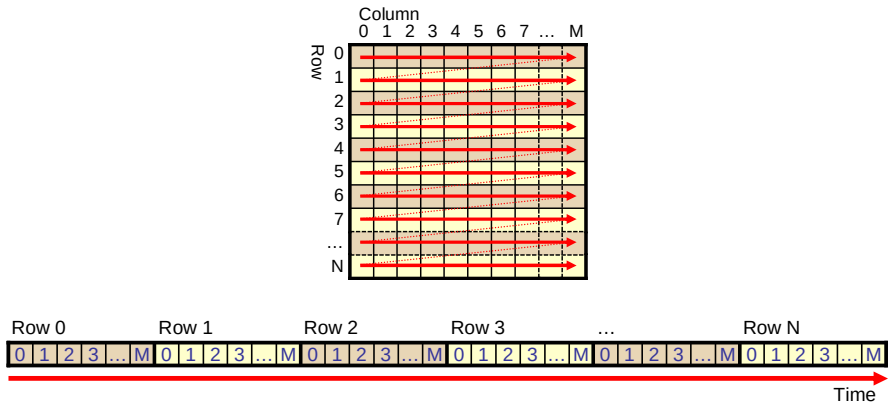


Figure 2.3: Conversion from spatial parallelism to temporal based on a stream processing model

Random access processing relaxes the hard timing constraint imposed by stream processing. Pixels can be accessed randomly as needed from a frame buffer. As the pixel access is not deterministic and access can take time, pixel access may take several clock cycles but this can be taken into account by the algorithm design. This configuration more closely follows the software model, which makes the direct mapping of software algorithm easier. This type of design aims to maximise clock speed using low-level pipelines.

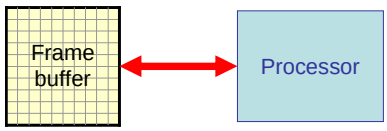


Figure 2.4: Random access processing

Stream and random access modes can be combined. Data can be stream processed as it is loaded into a frame buffer. This can then be accessed by a random access process. Stream processing can then also be used on the output as shown in Figure 2.5. Frame buffers must generally be inserted between stream and random access processors to decouple the image access requirements. Each frame buffer will introduce a latency of one or more frames.

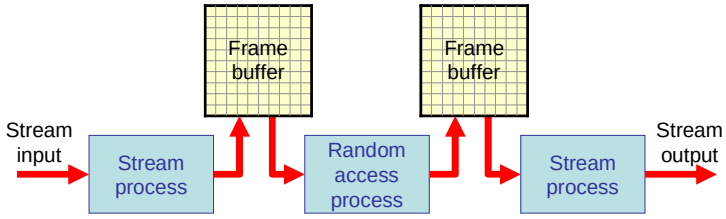


Figure 2.5: Hybrid of stream and random access processing

Another way to use multiple frame buffers is image partitioning. In this design the image is either duplicated, or split up into parts in multiple buffers. Each of these is then processed with separate duplicated hardware blocks. This allows the same function to be performed on the whole image; it is best used on point operations as extra hardware is required to combine area based operations. These combination operations can be costly in terms of the hardware required.

Constraints

As well as FPGAs having very different configurations, real-time image processing imposes constraints (Johnston et al., 2004b), which depend on the processing mode used. Streamed processing imposes the strictest constraints. Constraints include *timing constraints*, *memory bandwidth constraints* and *resource constraints*. The requirement to operate on an image at real-time rates, either at the input or output produces a *timing constraint*. For example, if operating at a VGA output rate, a new pixel must be produced every 40 nano-seconds. Achieving this for non-trivial applications, such as lens distortion correction (Gribbon et al., 2003) 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 can solve such problems. Given enough resources, almost any desired throughput can be achieved by pipelining at the expense of added latency. The need for long pipelines introduces control complications, including the priming, stalling and flushing of pipelines. There are also synchronisation issues which can be solved with global scheduling, token based control, channels (Burns and Davies, 1993), or buffered channels (channels with FIFO buffers).

Memory bandwidth constraints come from the need to buffer all or large parts of the image. Frame buffering requires a large amount of memory. Buffering is required when operations require pixel data in a different order from the order the pixels arrive, or need more than one pixel at once. The required size of the buffer depends on the operation. Potentially, the whole frame must be buffered. For example to rotate an image rotation by 90 degrees. This requires more memory than

is typically available on an FPGA. Off-chip RAM typically only allows a single access per clock cycle. This can be a problem for the many operations that require simultaneous access to more than one pixel from the input image. For example, bilinear interpolation (discussed later, or in (Gribbon and Bailey, 2004)) requires simultaneous access to four pixels from the input image. Possible alternatives to deal with this problem are to use multiport RAM, multiple RAM banks in parallel, a faster RAM clock to read multiple locations in a single pixel clock cycle, or packing several pixels into a single memory location. Caching reduces the memory bandwidth by storing data that will be needed again in on-chip memory. Data from a local cache can be read in parallel with data from the frame buffer. These caches are typically small with respect to image size. The type of caching required is normally dependent on the algorithm, requiring the cache and algorithm to be developed together.

There are two types of resource contention. Resource conflicts due to concurrent access and resource contention arising due to the finite resources available. Potential resource conflicts can occur when several parts of the algorithm share a common resource, such as memory. Normally, only one process can access a resource at a time. The other processors need to be denied this access. If they carry on regardless, dangerous side effects can occur. A processor may wait for the resource to be free, which can affect processor timing, or perform some other task while waiting, which is hard to implement. This means that the algorithm must be designed to handle potential conflicts, either by incorporating sharing mechanisms or by reducing the coupling between parallel sections. Scheduling access works for synchronous systems but is difficult to handle for asynchronous access. Semaphores (Burns and Davies, 1993) built in hardware avoid conflicts by giving exclusive access to the resource. However, processors will block if the semaphore blocks access to the resource they require, and this can affect the timing of the algorithm. The resource being accessed can be coupled with the logic to share its access to produce a resource manager.

The resource constraint due to finite system resources, is not limited to the on-chip resources of logic blocks and BlockRAM. It includes I/O pins and off-chip memory. The developer also needs to make efficient use of logic blocks as these are normally used by arithmetic operations. Multiplication, division, square root and trigonometric functions normally use large amounts of resources. These can be reduced by the use of a number of techniques some of which will be discussed in the next examples. These include look-up tables, CORDIC arithmetic, and incremental calculation.

Summary of Design Cycle

The selection of the architecture will affect the mapping of the algorithm onto that architecture. In particular, it will limit the types of operations which can be implemented easily and efficiently. For example, although many low-level image processing operations map well onto a stream-based processing model, many medium and high level operations do not. As another example, if an operation requires a lot of RAM access, it will be necessary to use an architecture which has a wide memory bandwidth. For example, by mapping or caching external memory into multiple parallel BlockRAMs within the FPGA. If this is not possible (or practical) then the alternative is to modify the algorithm to match the architecture. A poor match between the architecture and the algorithm can result in limited acceleration of the design when implemented on hardware. It can also result in inefficient use of the resource which can lead to a larger (more expensive) FPGA than needed.

The actual mapping takes place during the system implementation stage, and is based on the constraints placed on the design by the architecture. This is often the most difficult task. It involves the mapping of operations onto the hardware resources. This may require the modification of the algorithm, which must again be tested to ensure the design still meets the application specifications. At this stage it is often necessary to develop device drivers for the connected peripherals so that the FPGA can communicate with its inputs and outputs.

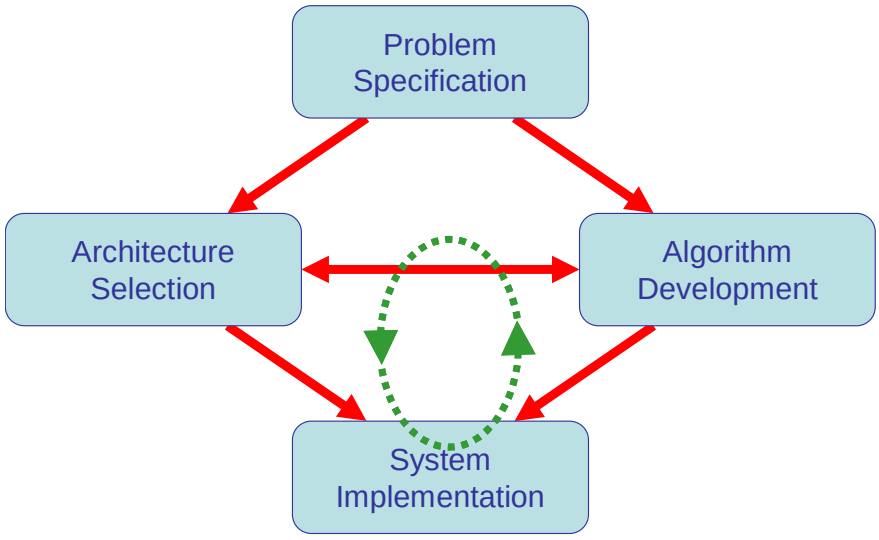


Figure 2.6: FPGA design cycle (Bailey, 2007)

The stages in the design cycle are interdependent, as illustrated in Figure 2.6 (Bailey, 2007). The problem specification will affect design decisions for both the selection of the architecture and the algorithm development. The architecture and the algorithm affect each other and these in turn affect the final system implementation.

This leads to a development cycle where there may be several iterations modifying both the image processing algorithm and the computational architecture to produce a final system implementation that meets the problem specification. This design cycle can be further refined, as in Figure 2.7, by treating the development of the algorithm and its mapping to the computer architecture as an iterative process where both need to be considered at the same time, with one feeding into the other.

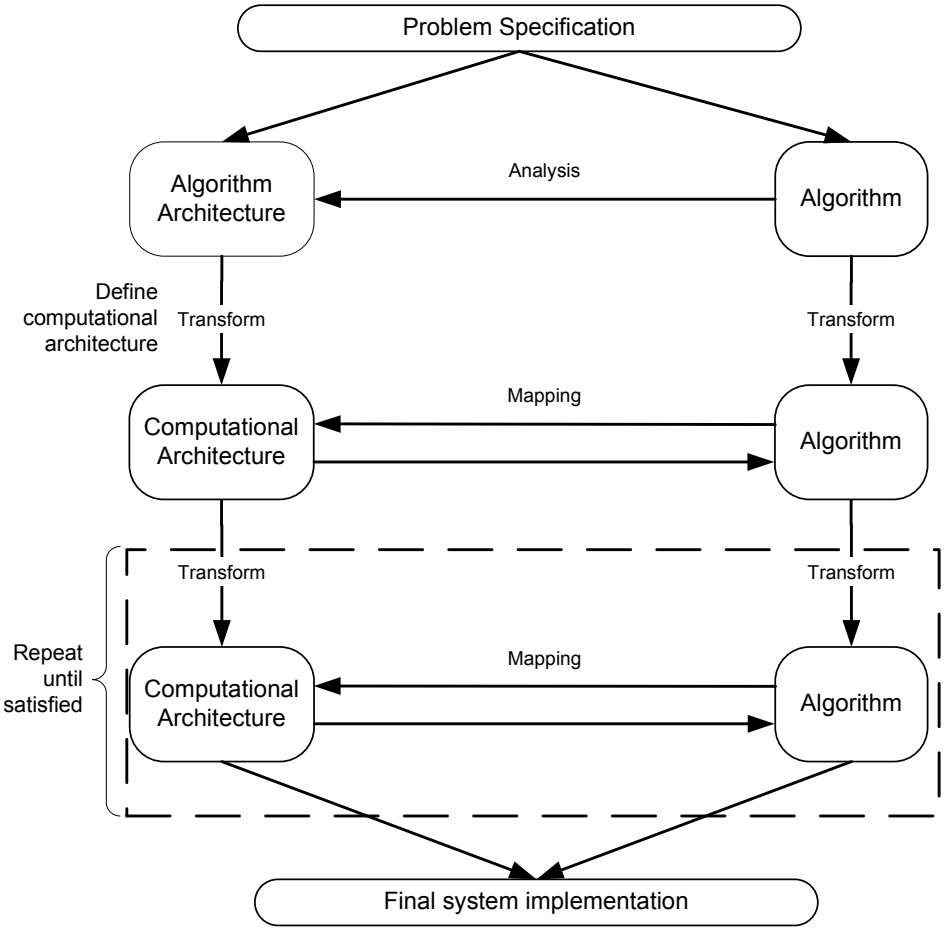


Figure 2.7: Explicit design cycle

2.1.2 Analysis of Example Implementations

To determine the essential features that a language for image processing on FPGA hardware needs to support, it is perhaps best to examine in detail the implementation of a range of algorithms. During the development of each algorithm, the following issues are of particular interest:

- What are the problems encountered? Many of the problems encountered in one algorithm will also be present in others. The language should help the developer solve the problems, either explicitly by providing problem specific tools, or generally by providing features that facilitate a solution.

- What are the particular hardware constructs used? If the constructs are particular to an application, then the language should make it easy to design or develop the constructs needed. If the constructs can be generalised, and are applicable to a wide range of problems, then it may be appropriate to provide parameterised versions of those constructs that may be instantiated into a particular design.
- What are the key differences between hardware and software development? The language needs to ease the developers into the process of hardware design, as they port well established algorithms from software.

Many of the observations will be specific to a particular application or implementation. Attempts will be made to generalise from these observations. The overall goal is to design a language whose constructs are (a) specific to the design of image processing algorithms on FPGA hardware and (b) capable of being used in more general situations than those encountered in the examples.

The examples of image processing algorithms that we will look at are:

- Lens distortion correction
- Production of a histogram of pixel values in an image
- Object tracking
- Identification of connected components (single-coloured regions) in an image

These examples cover a subset of image processing problems, concentrating on low to medium level image processing operations. This selection is a reasonable representation of image processing including point operations, local filters, geometric transformation, segmentation, object labelling and tracking and data extraction. The focus on low- to medium-level image processing operations is primarily because high-level operations are generally less able to exploit parallel architectures as they are often serial and process relatively low volumes of data. Consequently, high-level operations usually do not make efficient use of hardware, although the high-level operations can be pipelined to form a specialised pipeline processor.

2.1.3 Lens Distortion Correction

Lens distortion occurs when the magnification of the lens is not uniform across its field of view. Barrel distortion is the most prevalent, and is usually associated with wide angle lenses. The characteristic barrel shape results because the magnification at the centre of the lens is greater than at the edges, as illustrated in

Figure 2.8. While it may be avoided by use a higher quality multi-element lens system, this comes at considerable additional cost which is not warranted in lower cost commodity consumer devices such as web cameras, surveillance cameras, and cell phone cameras. Barrel distortion is primarily radial in nature and for many lenses, a relatively simple one-parameter model can account for most of the distortion (Mengxiang and Lavest, 1996). A cost-effective alternative to using an expensive lens is to algorithmically correct for the distortion using the model. In this application, the goal was to correct the distortion on-the-fly as the image is being displayed from a frame buffer.

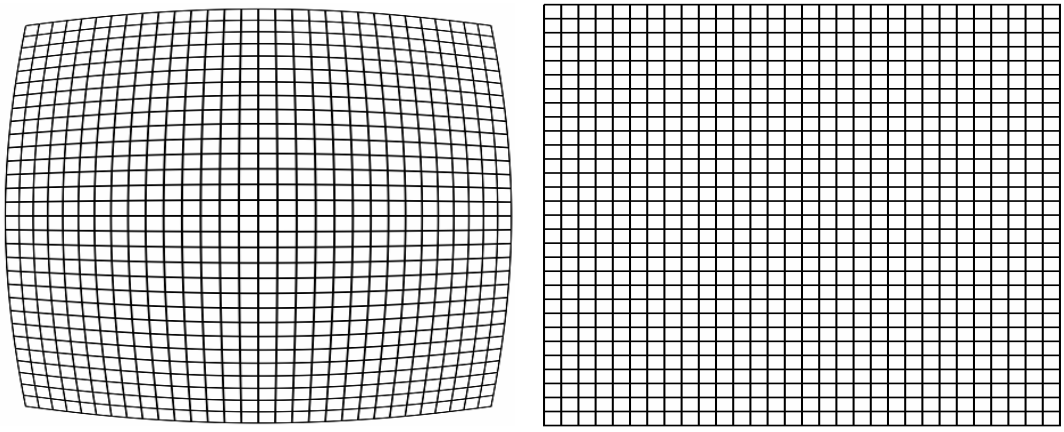


Figure 2.8: Distorted captured image of a regular image on the left and desired image on the right

The relationship between the distorted and undistorted image coordinates may be modelled by (Bailey, 2002)

$$r_u = r_d(1 + kr_d^2) \qquad (2.1)$$

where r_u and r_d are the radial distances from the centre of distortion and k parameterises the distortion. For barrel distortion, a positive value of k accounts for the decrease in magnification with increasing radius.

The problem with this model is that the correction is in terms of the distorted image. To correct for the distortion as the image is being displayed, it is necessary to determine the corresponding pixel in the distorted image for each undistorted pixel. The display requires that the undistorted pixels be produced in a raster order at a rate of 1 pixel per clock cycle.

Observe that the term in parentheses in (2.1) represents the radially dependent magnification function. What is desired is the equation in the form of

$$x_d = x_u M(k, r_u^2)$$
$$y_d = y_u M(k, r_u^2) \qquad (2.2)$$

where $M(k,r_u^2)$ is a magnification factor that depends on the distortion and position in the image. The reason for using r_u^2 rather than r_u is that $r_u = \sqrt{x_u^2 + y_u^2}$ and the square root function is expensive to implement in hardware. Equation (2.1) can be rearranged to give:

$$M = \frac{r_d}{r_u} = \frac{1}{1 + kr_d^2} \qquad (2.3)$$

The dependence of the magnification on r_d^2 can be removed by substituting (2.2) to give:

$$M = \frac{1}{1 + kM^2 r_u^2} \qquad (2.4)$$

This can be solved either directly (by solving the underlying cubic polynomial) or iteratively (by initialising M to 1, and then iteratively substituting into equation (2.4) until it converges). The mapping depends on the product kr_u^2, which avoids having to create a separate mapping for each k. The direct solution is quite complicated. The iterations cannot be performed in real time, so the mapping $M(kr_u^2)$ is precalculated and stored in a lookup table.

For a particular level of distortion, both r_u^2 and k depend on the size of the image. r_u^2 is very large, and k very small. Rather than use floating point, both are scaled by powers of 2 to bring into the range 0 to 1, and simpler fixed point arithmetic is used, reducing the area and power used. This has the added advantage that the magnification table does not need to be changed for different sized images.

Algorithm

The algorithm can be broken up into a number of subsections. The system can be driven entirely from the current output raster scan position as shown in Figure 2.9. The current scan position needs to be offset by the centre of distortion to get the undistorted image coordinates. Then r_u^2 can be calculated, multiplied by k and looked up to obtain the correcting magnification factor. This is multiplied by the undistorted coordinates to calculate the address of the corresponding distorted pixel that is located in video RAM. This pixel is read from the RAM and displayed to the screen.

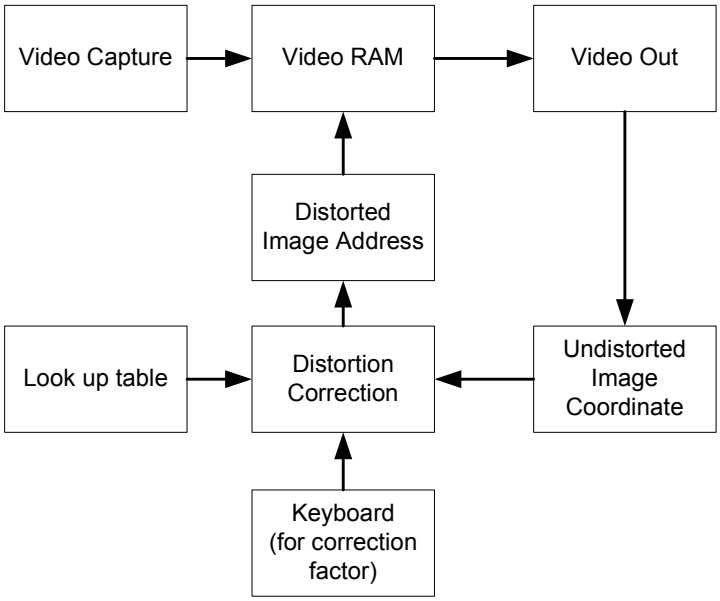


Figure 2.9: System diagram

The implementation details of the algorithm are summarised in the following sections; for more detail on the algorithm refer to (Gribbon et al., 2003, Johnston and Bailey, 2003, Johnston, 2003) with particular attention to the optimisations required to reduce the resources and for the system to run in real time.

In the design there are several functions running in parallel. These are:

- the keyboard interface - used to allow the user to change the correction factor *k*.
- the video capture – which captures the image from the camera and places it into video RAM.
- the video output – which provides the desired pixel coordinate which is translated to the corrected pixel address which it displays.
- the distortion correction.

The ability to have several modules operating in parallel increases the flexibility of the system but there needs to be communication between cooperating hardware blocks. In this system, the video capture and the display sections share access to the video RAM via a register based communication flag. The distortion parameter, *k*, from the user is stored in a register with the correction factor used in the next iteration of the algorithm.

Screen Coordinate Calculation

The output is required in a raster scan, where x increments by one for each pixel output. This enables x^2 to be calculated incrementally making use of the expansion:

$$(x+1)^2 = x^2 + 2x + 1 = x^2 + x@1 \qquad (2.5)$$

The second form makes use of the fact $2x+1$ is equivalent to multiply x by 2 (a left shift by 1 bit) and setting the least significant bit to 1. Using Handel-C notation equation (2.5) reduces both the logic depth of the equation and the required hardware by replacing the multiplication with a single addition.

Magnification Calculation

The calculated x^2 and y^2 are then added to give r_u^2 which is multiplied by k to produce kr_u^2. This is then used to index a lookup table to give the magnification. The number of bits required to sufficiently represent the magnification depends on the later stages. If the nearest neighbour pixel is used we throw away any fractional bits so there is no need to calculate them and we can use fewer bits than if some form of interpolation between pixels is used.

Pipelining

Not all of the calculations can be performed in a single clock cycle. Therefore, to meet the real-time processing constraint, the operations need to be implemented as a pipeline.

In designing such a system, it is the sequence of operations that matters, not their simultaneity. The same is true of data being processed in a hardware pipeline; as far as the data is concerned, the pipeline is sequential, transforming the data from input to output through a sequence of operations. Each processing element in the pipeline, like its production line counterpart, operates sequentially on each input data element before passing the results on to the output. It is only the correct synchronization between successive (serial) stages that allows each stage in a pipeline to run in parallel with all the others.

Once the pipeline has been primed an output is generated every clock cycle, but until then, the output will not be valid. The length of the pipe depends on the number of operations that it contains.

Figure 2.10 shows the pipeline for this process. The bold boxes indicate where data is registered between clock cycles. The total pipeline latency is five clock cycles.

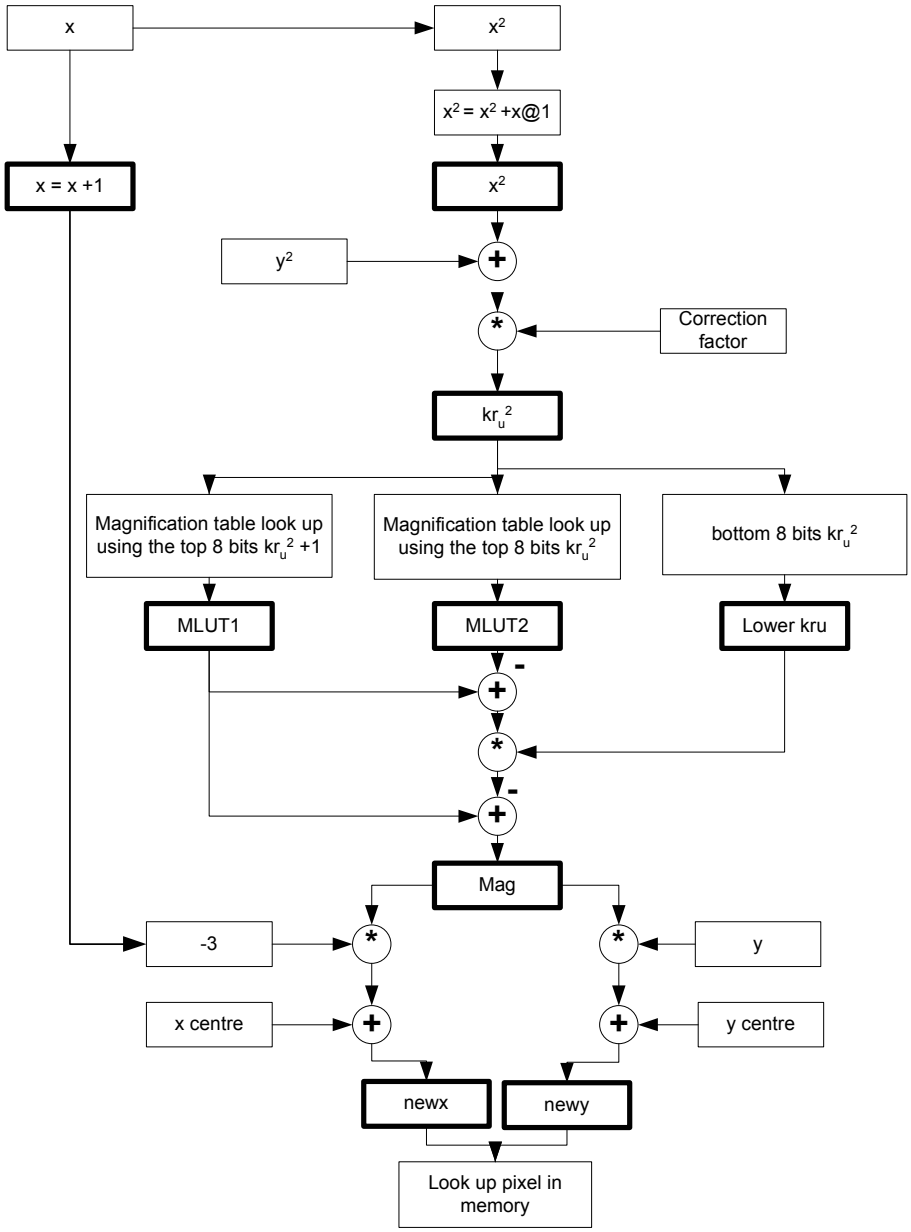


Figure 2.10: Pipeline for coordinate calculation

One of the limitations of this implementation is that any subpixel information is truncated. This can be improved by adding an offset of 0.5 before truncation to give rounding or by the use of interpolation. Bilinear interpolation requires reading 4 pixels from the video memory. This results in a bandwidth bottleneck as only one pixel may be read from video memory per clock cycle. Overcoming this requires a custom caching technique such as that described in (Gribbon and Bailey, 2004), to increase the likelihood that cached pixels are ones that will be used later for interpolation. This results in a complicated pipelined algorithm where many parallel operations are running in each stage of the pipeline. Combining the interpolation stage with the distortion correction leads to a 9-stage pipelined algorithm (Gribbon et al., 2003). (Ngo and Asari, 2005) have also investigated implementing barrel

distortion correction on an FPGA. Their approach is to map a software distortion correction algorithm almost directly onto a pipelined hardware implementation. Consequently, their design is less efficient requiring a 91 stage pipeline.

Insights Gained from Algorithm

From these two examples of a barrel distortion algorithm, we can gain some insights into both the design process and the operations that are used.

As we are performing a geometric transformation of an image, the whole image needs to be buffered into memory. This leads to the need for a frame buffer. Since the image capture must take place simultaneously, the frame buffer is composed of two memory banks (to simultaneously allow one to be written by the image capture and one read by the distortion correction hardware). Such shared access requires logic to swap the banks between input and output. The frame buffer can be thought of as a resource and the logic to share access as its corresponding resource manager.

The iterative calculation of the magnification factor ($M = \frac{1}{1 + kM^2 r_u^2}$) is very expensive to implement in hardware, both in the area needed and in the time delay it would impose. The calculation contains two squaring functions and a division, both expensive operations to implement in hardware. Being iterative, it is also expensive in terms of the time to get a result. This led to precalculating the magnification factor and storing it in a LUT (lookup table). Many complex operations can be converted to a lookup based operation, giving a constant operation time of 1 clock cycle. However, lookup tables are not free. The table size will affect the resolution (precision) of the result. The trade off between the size of the table and the size of implementing the operation needs to be analysed. There is no point to replace an operation with a LUT only to have it use more resources than the operation would use directly in hardware to get the required resolution. To improve the resolution, an interpolated lookup table can be used, as in this example. The use of LUTs is an important design tool when implementing image processing on to FPGAs. Such tables would be candidates for an advanced building block within the language. Though this example used split tables and interpolation using a multiplier, bipartite and multipartite (Boullis et al., 2001, de Dinechin and Tisserand, 2001, Mencer and Luk, 2004) LUTs are more often used as they avoid this multiplication and require only lookups, additions and subtractions.

This design has used a mixture of different width operands. Most of these were fixed-point or had inputs that had different fixed-point arrangements in Handel-C, the language used to implement this example. As with other hardware definition languages, there is no fixed-point data type. This requires the operands to be manually padded so that their binary point positions match. Having a fixed-point data type as a native type would allow the design tool to automatically align operands, saving developers from having to keep track of manually mapping of operand widths and binary point position.

Bit manipulation operations were also used to take parts of registers and results (such as for the LUT interpolations and appending bits). These bit manipulation operations (appending, truncating and selecting) should be incorporated in any hardware description language.

The use of fine-grained pipelining is a critical part of implementing both of the algorithms for distortion correction on FPGA. In both cases the main emphasis has been on the pipeline path. Errors are easy to introduce as the misalignment of parallel operations which can result in incorrect results. Such errors can be hard to detect and debug. It would therefore be advantageous to have some way to make these errors more visible to the developer. Another difficulty is dealing with the priming, flushing and stalling (or pausing) of pipelines. A pipeline has several operation phases: priming where the pipeline is being filled and invalid data will be output, running where the pipeline works as normal, and flushing where the data in the pipeline is flushed out by inputting dummy data until the remaining valid results exit the pipeline. In image processing, pipelines can also be stalled while waiting for new data to arrive (this is common when interfacing with a camera which uses blanking periods). These different phases require conditional statements to control the execution of the pipeline. Usually, when developing algorithms most attention is initially given to the normal operation of the pipeline. However, the control aspects often require more design effort. If many of these aspects of pipeline can be at least partially automated by the language, this would relieve the developer of some of this complex process.

The design also has different operations taking place at different times. In this case they are during the image scanning, horizontal blanking and the vertical blanking periods. This type of design requires some form of event triggering or scheduling so that each component of the design can be controlled and activated at the required time.

It is beneficial to create an algorithm appropriate for the FPGA implementation rather than directly porting a software design.

2.1.4 Histogram Example

Intensity histograms are constructed by counting the number of pixels in an image which have an intensity value and plotting a frequency distribution. Figure 2.11. shows a grey scale image with its resulting histogram. Histograms are normally constructed for grey scale images but can also be accumulated for colour images by treating each of the red, green and blue components as a separate grey scale image. Histograms are useful for setting threshold values, for providing data for contrast expansion, histogram equalization or other intensity-based enhancement techniques and for gathering of basic image features.

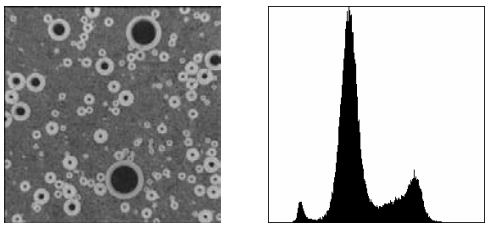


Figure 2.11: Image and its histogram

A histogram is a good example for illustrating resource conflicts. These can occur when a histogram is being constructed and displayed. The actual processing for constructing a histogram is quite simple, consisting of reading the pixel count for the current pixel intensity, incrementing the value, and storing this back in memory. If the histogram is constructed while the video stream is buffered into a frame buffer (one of two banks of RAM), the histogram data can be kept synchronised with the image it relates to. At the same time, both the previous full image and its histogram are being processed for other operations (or in our test implementation, 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. Problems can occur due to the need to reset the values in each histogram bin before constructing the histogram for the next image. This requires a more complex passing of control of resources from process to process (than a simple conditional test), which can increase the likelihood of an error. The sequence for processing video sequences is shown in Figure 2.12. In this, two full frames are shown with the corresponding resource usage. It can be seen that none of the read or writes to the RAMs or histogram storage (which are stored as block RAM) overlap and the histogram can be reset during the blanking period between new frames.

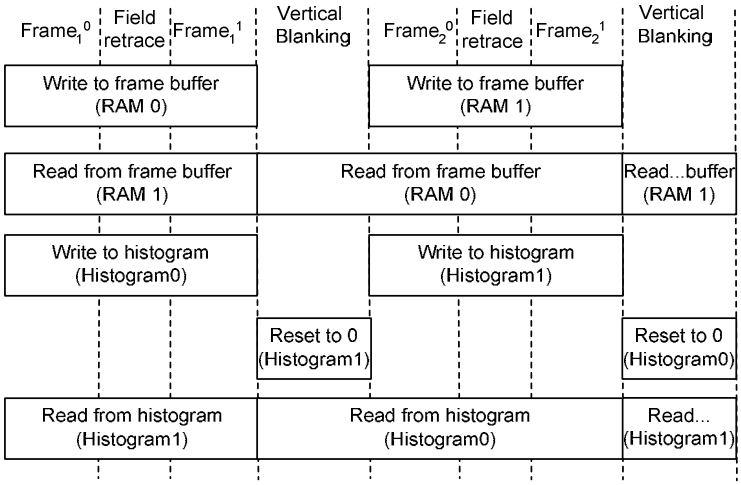


Figure 2.12: Timing of processes and resources used for a streamed histogram function

Insights from the Histogram Example

There are two main insights to be gained from the histogram processing example. Again, it is necessary to buffer images and data so they can be read and written concurrently without the risk of overwriting required information. Secondly, it is necessary to be able to schedule processes at a level where the impact of shared resources is taken into account. In this example, the different operations on a data structure have been scheduled to avoid data access conflicts.

2.1.5 Object tracking

This section is an analysis of the work presented in (Johnston et al., 2005a, Gribbon et al., 2005).

Object tracking often involves the use of a camera to provide scene data from which the motion of real-world objects is mapped to system controls (Baumela and Maravall, 1995). Object tracking for control-based applications must be real-time as sensing delays in the input can cause instability in closed-loop control. This is particularly important if the user must receive sensory feedback from the system.

The design emphasis we have taken in our implementation has been on minimising the logic and resource utilisation, leaving resources for additional functionality that will use the tracking information in the desired application. Design decisions to reduce hardware requirements place a limit on the type of object tracking algorithms that can be implemented (Johnston et al., 2004b).

Environment

Our aim is to work in a relatively unstructured environment. To allow this we restricted the target objects to saturated and distinctive colours to enable them to be distinguished from the unstructured background. This enables a much simpler segmentation algorithm to be used. In the context of a gesture based input, the user is required to wear or hold fluorescent (highly saturated and intense) colour "markers". Each marker has a different colour to allow the tracking algorithm to differentiate between them. Each object to be identified and tracked must have a unique uniform colour associated with it.

Image Processing System

The required algorithm needs to work on a stream based image. Stream-based processing limits the type of algorithms that can be easily implemented (Johnston et al., 2004b) so the algorithm is unlikely to be able to be based directly on a standard image processing algorithm.

Object tracking first requires the target objects to be segmented from the rest of the scene in the captured image. The objects are then tracked by considering their change of position in successive frames.

Segmentation partitions the captured image into several disjointed object regions based on common uniform feature characteristics (Castleman, 1996). One simple method is to segment the image based on colour by applying thresholds to each pixel. This is ideal for stream processing because thresholding is a point operation which can be implemented easily on the FPGA. A block diagram of the complete system is shown in Figure 2.13. We capture a scene with a camera; this is converted to a digital pixel stream which the FPGA reads. The tracking system detects the target objects, whose positions are then fed to another application on the FPGA; this can be used to control an external system.

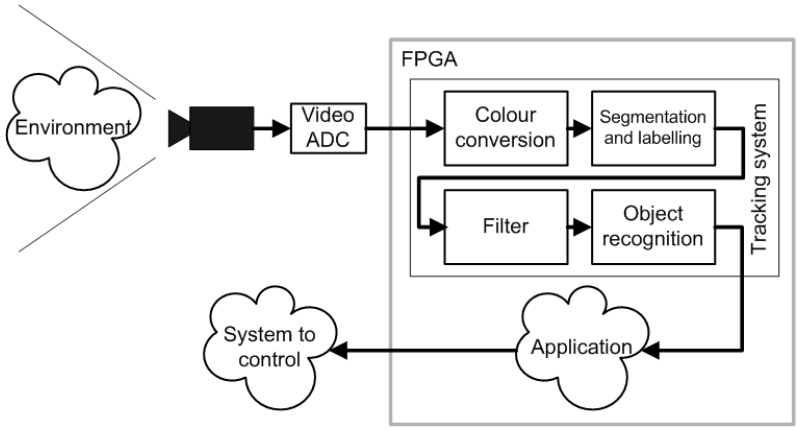


Figure 2.13: Block diagram of tracking system

The tracking algorithm is broken into four stages: colour conversion, segmentation and region labelling, morphological filtering, and object detection. The pixel stream from the image capture sub-system is converted to a YUV colour space to remove the inherent interdependence between luminance and chrominance in the RGB colour space. Segmentation is performed using colour thresholding to associate each pixel with a particular colour class. A morphological filter is then used to remove any noise pixels that are not part of the object regions. Finally, objects are detected by constructing a bounding box which encloses all of the pixels within a colour class. For each object, region parameters such as position, size, and orientation may be determined.

A strict timing constraint is imposed with pixels arriving at a rate of 13.5 MHz, allowing approximately 74 ns per pixel to perform all four operations in Figure 2.13 . This inevitably requires applying both coarse-grain (between operations in the processing chain) and fine-grain pipelining (within operations) to ensure meeting of timing constraints.

Image Capture

The input from the camera is a stream of 16-bit RGB (5:6:5) pixels. The stream is interlaced with successive fields providing the odd and even lines of the PAL frame. There are two clock domains used: one is for the communication with the video ADC, and the other for communication with the video output chip. The ADC needs two clock cycles to transfer the pixel data to the FPGA. To save storage space and reduce the channel size needed for data transfer between clock domains the image processing is performed on the input side (with no image buffering).

Colour Space Transformation

The simplest form of colour segmentation is to independently threshold the red, green and blue components of each pixel. Those pixels that are within all three ranges are classified as belonging to the corresponding colour class. Unfortunately, such segmentation is frequently unreliable because any change in the intensity of a colour results in a diagonal movement within RGB space, with a significant effect on all three components (Sen Gupta and Bailey, 2004). To allow for this, the colour region has to be quite large with the significant likelihood of background pixels being misclassified into each colour class. This intensity interdependence may be reduced by transforming the image to another colour space which separates the chrominance from the intensity or luminance, such as HSI or YUV. The transformation from RGB to YUV colour space consists of a coordinate rotation to map the RGB cube onto the *Y*, *U* and *V* axes. Transforming from RGB to HSI involves mapping the RGB cube to a cone and is computationally more expensive, although it gives better intensity independence (Foley and Van Dam, 1982).

The YUV colour space is widely used in video and broadcasting (Russ, 2002). The standard RGB to YUV transformation matrix involves several floating point multiplication operations which are computationally expensive for an FPGA implementation:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.6}$$

A more efficient alternative is to replace equation (2.6) with the modified YUV colour space transform proposed in (Sen Gupta and Bailey, 2004). This removes the multiplications, replacing them with simple addition, subtraction and shift operations:

$$\begin{bmatrix} Y' \\ U' \\ V' \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 0 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.7}$$

Note that this transformation is only similar to the standard YUV transformation in that it separates the luminance and chrominance components. The two chrominance components are orthogonal. However, in this application, we do not need true YUV (as YUV is based around human perception), so this simplification is valid. Although the *R*, *G*, and *B* components have different precisions, as far as

equations (2.6) and (2.7) are concerned, they can be treated as fractions between 0 and 1. Equation (2.7) will give 7 bits of precision for each component.

As equation (2.7) is a coordinate rotation, any change to the intensity will also affect the values of U' and V'. Therefore, to make the U' and V' less sensitive to illumination, we want to normalise them by the intensity, Y'. This caused problems because for the saturated colours chosen for the markers, the value of Y' can be lower than that of V'. This can shift the normalised values outside the range -1 to 1. This may be avoided by normalising by

$$Y'' = \max(R, G, B) \qquad (2.8)$$

instead of Y'. Y'' has 6 bits precision.

The FPGA implementation calculates Y'' from equation (2.8), and U' and V' from equation (2.7) in parallel for each pixel in the input stream. This makes it possible for the whole conversion to occur in one clock cycle.

Colour Thresholding

For each colour class, the normalised components are independently thresholded using

$$Y'' > Y''_{\min},\ U'_{\min} < \frac{U'}{Y''} < U'_{\max} \text{ and } V'_{\min} < \frac{V'}{Y''} < V'_{\max} \qquad (2.9)$$

Only a single threshold is required for Y'' because the target colours are bright (using problem knowledge to simplify hardware). A pixel belongs to a colour class only if it is within all three Y'', U', and V' ranges, and it is assigned the label corresponding to that particular class. Any pixel not within any colour class is labelled 0, and is considered part of the background.

Equation (2.9) includes a division operation which can be costly to implement in hardware. Divisions introduce long combinatorial delays if not pipelined (Johnston et al., 2004b), although there are hardware efficient pipelined algorithms (some are described in (Bailey, 2006)) that can be used. The division can be removed algebraically by multiplying the U', and V' components of equation (2.9) through by Y'' to give

$$Y''U'_{\min} < U' < Y''U'_{\max} \text{ and } Y''V'_{\min} < V' < Y''V'_{\max} \qquad (2.10)$$

To perform segmentation and labelling of N colour classes, equation (2.10) requires $4N$ multiplications and $5N$ comparisons. For real-time operation each

colour class must have separate hardware because all N sets of comparisons must be performed per clock cycle. These operations can therefore consume significant resources on the FPGA if performed in parallel.

A lookup table (LUT) can be used to perform the thresholding, normalisation and labelling in a single step. The LUT method involves pre-calculating the result of equation (2.10) for all valid input values. On initialisation, the resulting values are loaded into local memory on the FPGA (in Block RAM). Since the U' and V' thresholds are independent, they may be separated, requiring 2 lookup tables of 2^{13} entries each. Each entry would produce a single bit result indicating whether or not the colour is within the thresholded range. This is still expensive in terms of resources. By reducing the precision of Y'', U' and V', a compromise can be made between precision and resource utilisation. As the RGB inputs only have 5 or 6 bit precision, reducing the precision of U' and V' to 6 bits is reasonable. The Y'' is used primarily for normalisation, and some normalisation will take place even when relatively low precision is used. Therefore the Y'' component was reduced to 3 bits. The net effect is that the $Y''U'$ and $Y''V'$ tables requires 2^9 bits each.

Multiple colour classes may be tested simultaneously because several tables in parallel would use the same address (the actual colour of the pixel being classified), but have 1 bit for each colour class. This arrangement is illustrated in Figure 2.14.

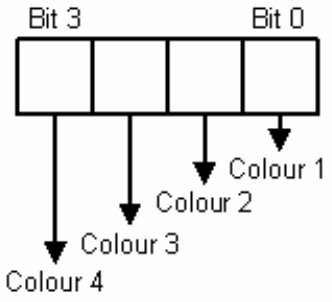


Figure 2.14: Representation within a LUT element

In this implementation, 4 colours are tested simultaneously in this way. Both $Y''U'$ and $Y''V'$ for 4 colours may be combined into a single Block RAM. The RAM is dual-ported, allowing both tables to be accessed simultaneously.

The two 4-bit colour class mask results are bit-wise ANDed to test that both bounds are satisfied. If the result is 0000 the pixel belongs to the background. A single 1 in any bit position indicates the corresponding colour class. The case of multiple ones is by definition invalid, because a pixel may belong to at most one class.

This only occurs when the colour space regions overlap, and is a result of incorrect threshold settings.

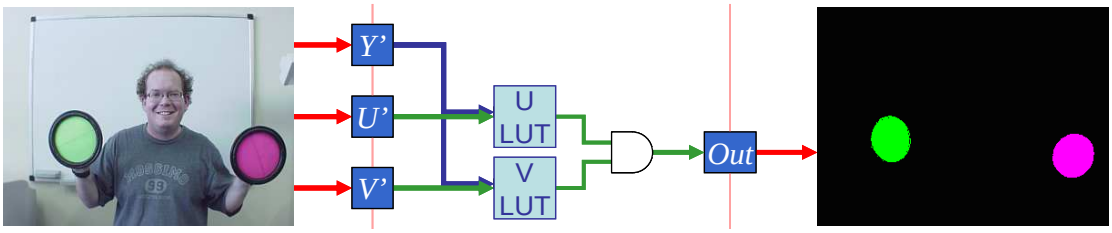


Figure 2.15: Result of colour conversion then LUT

Figure 2.15, shows the operation of the colour segmentation algorithm. The LUT approach to colour classification works with any number of colour classes (subject to resource limitations), and has a constant processing time of 1 clock cycle per pixel.

Object Recognition

A bounding box (Russ, 2002) provides a simple method for calculating the position, size and aspect ratio of a labelled region. With a little additional processing, the bounding box can also be employed to give the orientation of a non-symmetric object. A bounding box was chosen because it is a very simple and computationally efficient way of localising an object.

Each target object and thus colour class has its own bounding box. Thus, all bounding boxes must be calculated in parallel. Since each pixel can belong to at most one colour class, only a single bounding box is adjusted for any pixel. This implies that a single instance of the bounding box update hardware can be multiplexed between the bounding box data structures containing information for each colour class.

Data on the target object may be extracted from the bounding box during the vertical blanking period and be used to control any application that can map this information to some useful function. This can be the centre of the bounding box, the size, the aspect ratio and the orientation.

Morphological Filtering

After testing the initial algorithm, it was found that even with adequate tuning there were isolated noise pixels associated with each colour class. Any stray pixel will cause the box to grow to encompass the noise pixel, leading to the erroneous calculation of bounding boxes and consequently the derived tracking information. Pixels can be labelled incorrectly for several reasons including objects of similar

colour to the target in the environment, noise introduced by the image capture sub-system, specular reflections, and high contrast edges (Gribbon et al., 2004). To remove these mislabelled pixels a 2x2 morphological erosion filter was employed.

The two by two filter is separable which means that filtering can be performed independently using a two element window in each of the horizontal and vertical directions. Thus, the current pixel is first compared with the previous pixel (its left neighbour), by using a bitwise AND of the colour class masks. If the previous pixel belongs to the same class, it will be unchanged, but if the class is different, the result will be 0. The result is stored into the corresponding location in a line buffer. The result is also compared with the class label of the pixel from the previous line, obtained from the line buffer.

Insights

To achieve a compact design, it is necessary to balance algorithm optimisations and their effect on the target environment. All such optimisations rely on assumptions made about the environment and may restrict the operation of the system. The colour transformation has been simplified to remove the multiplications found in the standard YUV transform. The LUT-based segmentation and region labelling combine two costly operations into one step, eliminating the need for a large number of parallel comparators. To keep the lookup table small, it was necessary to reduce the resolution of the input data, limiting the degree of lighting independence. The use of a bounding box over more complex connected component based region labelling makes the algorithm sensitive to isolated noise points.

Like the other examples, pipelining is necessary to implement the design. Streaming converted the spatial parallelism in the image to temporal parallelism. The clock for the image capture codec was twice the input pixel clock. This required the system to use a multiphase pipeline (which is discussed in detail in the computational view chapter).

The design also operates over multiple clock domains. This requires the ability to assign an operation to run in a specific clock cycle and communication methods between clock domains.

This example also required the sharing of resources (in this case hardware) and the scheduling of operations to allow this.

Filters are a standard function in image processing and most preprocessing applications would require some form of filtering. It would be advantageous for the

language to provide the caching necessary for window scanning, and automatically manage edge effects (where the window extends past the edge of the image).

Again, this design used a mixture of fixed-point data types and widths.

The amount of hardware required was reduced by using a lookup table. This requires a degree of problem knowledge about what tradeoffs could be made.

The whole design followed our design process and we have had to make many decisions as we have mapped the design to hardware. A tool needs to encourage and aid developers to make good design decisions, and assist with evaluating the tradeoffs.

2.1.6 Connected Components

This section summarises and analyses the work published in (Johnston and Bailey, 2008, Bailey and Johnston, 2007) (for the original implementation) and (Ma et al., 2008, Bailey et al., 2008) for an improved algorithm. This section will discuss the design decisions and the mapped implementation, for resource utilisation and examples refer to the papers.

In the object tracking example we used a bounding box to track objects. This has limitations. If there is more than one object in the scene which is classified with the same label then the bounding box will expand to include all such objects. A connected components operation would allow multiple objects with the same colour class to be segmented and separately labelled. Connected components labelling is, therefore, an important step in many image processing algorithms.

The classic connected components algorithm (Rosenfeld and Pfaltz, 1966) requires two raster-scan passes through the image. In the first pass, when an object pixel is encountered, the 4 neighbours that have already been processed (see Figure 2.16) are examined. If one of those is already labelled, the label is copied to the current pixel. If none are labelled, a new label is assigned to the current pixel. However, a complication occurs when a “U” shaped object is encountered. Each of the branches of the “U” will have a different label, and when they join at the bottom, these labels must be merged. One of the two labels will continue to be used, and all instances of the other label need to be replaced with the label that was retained. Since many mergers may occur in processing an image, the relabelling process is usually deferred so that all of the merged labels may be changed at once. A merger table is therefore used to record such mergers. The merger table is usually constructed as a look-up table, with the label recorded for the previously processed pixels looked up to

derive the correct (merged) label for a connected component. In the second pass through the image, the merger table is used to relabel all of the pixels.

The preprocessing operations are ideally suited for stream-based processing without image buffering. This makes them an ideal candidate for implementation using an FPGA. If the connected components algorithm could be implemented within a stream-processing framework then it may be possible to eliminate the need for image buffering altogether. This would allow an FPGA based system to process a progressively scanned image directly from the camera without any off-chip buffering.

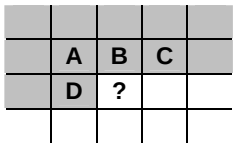


Figure 2.16: A label is assigned to the current pixel based on already processed neighbours.

While several high-speed parallel algorithms exist for connected components labelling (see for example the review in (Alnuweiti and Prasanna, 1992)), such algorithms are very resource-intensive, requiring massively parallel processors. This makes them less suitable for FPGA-based implementation because of the bandwidth bottleneck in reading in the image data. A resource-efficient iterative algorithm has been implemented on an FPGA (Crookes and Benkrid, 1999, Benkrid et al., 2003). This uses very simple processing but requires an indeterminate number of passes to completely label the image. This makes such an algorithm unsuited for real-time processing. There is also the requirement to buffer the intermediate image between passes.

Jablonski and Gorgon (Jablonski and Gorgon, 2004) have implemented the classic two-pass connected component labelling on an FPGA. In doing so, they were able to take advantage of the parallelism offered by FPGA-based processing to gain considerable processing efficiencies over a standard serial algorithm. However, their two-pass algorithm still requires the image to be buffered for the second pass.

Newer work (Appiah et al., 2008) presents a run-length based connected component algorithm that has been implemented on an FPGA and runs in real time. They convert connections on lines to runs. These are labelled and an equivalence table is created. In a second pass, the run labels are combined and translated to produce connected components.

To achieve a single pass streamed algorithm, the features of interest for each component must be extracted while performing the connected components analysis

(Bailey, 1991). This removes the need for producing a labelled image and saves having to perform the second relabelling pass. It is also necessary to perform merging and relabelling on the fly to ensure that consistent results are obtained.

Single Pass Algorithm

The initial single-pass connected components algorithm had four main blocks, as shown in Figure 2.17:

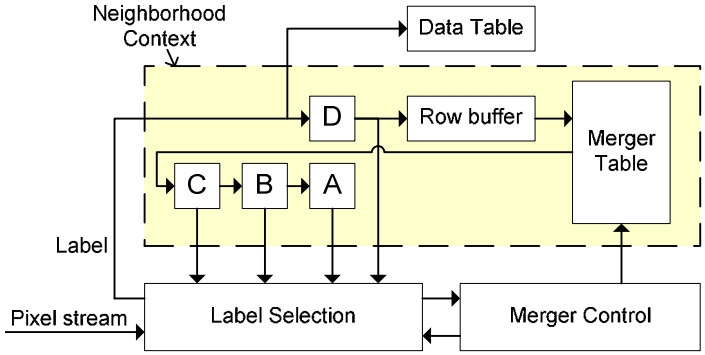


Figure 2.17: Basic architecture of the single pass algorithm.

1. The neighbourhood context block provides the labels of the four pixels connected to the current pixel being processed. This was implemented in much the same manner as a window filter. The neighbouring pixel labels are stored in registers A, B, C, and D. These are shifted along each clock cycle as the window is scanned across the image. Since the resultant labels are not saved, the labels from the previous row must be cached using a row buffer. The labels from the cache must also be looked up in the merger table to correct the label for any mergers since the pixel was cached.

2. The label selection block selects the label for the current pixel based on the labels of its neighbours. This used the same logic as the software algorithm, selecting the minimum label when a merger occurs. The selection logic may be simplified by considering the neighbourhood pixels in a particular order (Wu et al., 2005). A merger will only occur if pixel B is background and the label of C is different from either A or D. In all other cases, the labels will already be equivalent from prior processing.

3. A merger control block is required to update the merger table whenever two objects are merged. Whenever a new label is created, a new entry is added to the merger table pointing to itself. This avoids the need for initialising the merger table prior to processing. To keep the merger table up to date, whenever a merger occurs, the label that is replaced should now point to the merged label. The

merger table also uses a stack to resolve chains of successive mergers at the end of each row (as described in (Johnston and Bailey, 2008, Bailey and Johnston, 2007)).

4. The data table accumulates the raw data from the image required for calculating the features of each connected component. Since the image is not retained, data must be accumulated for each connected component as the image is scanned.

Mapping the Single Pass Algorithm

For the hardware implementation it was decided to break the design into five main blocks which could be progressively implemented and tested. These were the neighbourhood context, label selection, merger control, chain resolution and data table, as shown in Figure 2.18. The merging and chain resolution were separated into two parts.

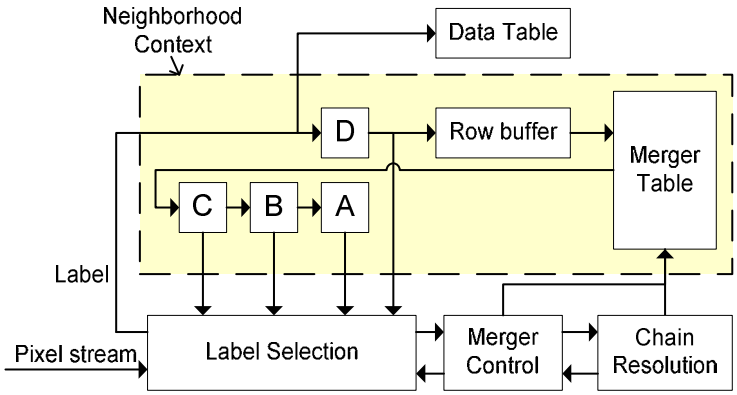


Figure 2.18: Architecture of the single pass algorithm.

This algorithm was implemented onto FPGA hardware in three stages. This enabled the main blocks of the algorithm to be developed and tested independently. In the first stage, the main labelling feedback loop, including the merger table, was implemented. The second stage added the stack for handling label chaining. The third stage introduced the data table, initially to enable "blob" counting, and then to measure the area of each region.

Data Table

The data table implementation depends significantly on the data required for each region. To avoid having to do two reads and a write when a merger occurs (to combine the areas for the two merged regions), the label and partial data for the current region are cached in registers. The area is accumulated for the different scenarios as follows:

- When a new label is created, it is copied into the cached label, and the cached area is set to 1.
- If a merger occurs, the data table entry for the region with the higher label is read and added to the cached area. That data table entry is also cleared to indicate that this data is not valid. The cached label is set to the lower label, as this is the label now associated with the data.
- Otherwise if the current label is the same as the cached label, the cached area is incremented.
- Finally, when a background pixel or the end of the row is encountered, the cached label is used to read the area from the data table. This is added to the cached area, and written back into the data table. The cached label is then set to the background.

At the end of each frame, the data can be read from the data table for subsequent processing. Since the merged regions have been reset to 0, the merger table is not needed to determine whether or not the label is valid.

2.1.7 Improved Algorithm

This algorithm was refined and modified in (Ma et al., 2008, Bailey et al., 2008). This was to reduce the memory requirements for the tables. To do this, the number of labels being tracked needs to be reduced. This can be done by re-labelling connected segments on each line and sending unconnected labels (ones which are finished) for further processing. This allows the reuse of those labels and produces the system shown below.

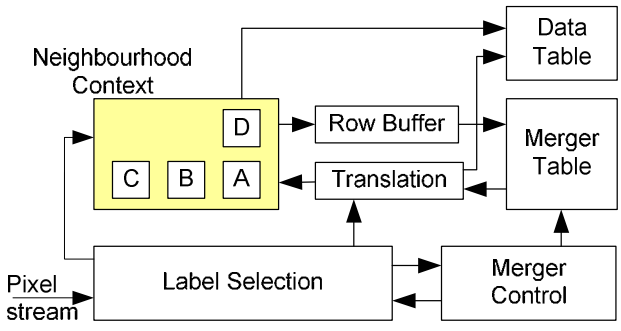


Figure 2.19: Improved algorithm block diagram

The design trades off an increase in complexity (more logic and extra tables) for smaller tables as they only need to have elements for ½ the image width.

Insights

The conversion of an existing algorithm to one that can be streamed is a non-trivial task. The mapping of this algorithm onto the FPGA is also non-trivial with the architecture design being a major factor in whether it will meet timing constraints, or fit within the resources available on the FPGA. Both logic and memory type need to be carefully considered in order to meet timing requirements.

The hardware required for this algorithm again needs pipelining and some stages of the pipeline have parallel operations within them.

Scheduling is important in this as different sections run during different events. The chain resolution section is scheduled to run only during the horizontal blanking period. The data table also needs to be read out for external algorithms during the vertical blanking period. The other operations run when there are pixels to be processed.

The logic used to do both the label selection and the merging is condition based. There is a need for nested conditions and these need to be carefully checked to make sure the logic is correct. This can be an error prone task and could be aided by the use of automatically generated `else` Boolean expressions and by making it clearer what operations are contained within a condition.

Row buffering structures were required to allow access to the surrounding pixels. Tables were an important part of both algorithms; they were used for both keeping data and for keeping the labels up to date.

These two algorithms clearly illustrate our design process (illustrated by Figure 2.7) of recursive mapping between the architecture and the algorithm. The first mapping of the algorithm considered the architecture and the need to remove the frame buffer by the removal of the second pass. This involved a significant redesign of the algorithm to enable it to be used with a streamed processing architecture, where any data needed for future processing was cached locally. This resulted in a novel algorithm where the labelling was managed through the use of a dynamically updated merge table. For this to work the problem of chaining had to be overcome which required both the architecture and the algorithm to be modified to have a pass through the merge table at the end of the line.

This algorithm was satisfactory for our needs but we realised that the table's size grew with the size of the image due to the need to keep the labels unique. This lead to a second major refinement by reusing the labels through a translation set. This again needed modifications to both the architecture, with new tables, and the

algorithm, with new steps. The final result has a smaller area and allows the objects detected to be output for further processing once they are no longer connected.

2.2 Insights Summary

The task of converting an image processing algorithm to a real-time FPGA implementation is not straight forward and involves an iterative approach. To achieve a low resource, high speed design, stream processing is one of the best approaches. All of the designs used stream processing for the majority of their design.

Image processing algorithms can benefit from extensive use of pipelining at both the operation and expression level. This needs to be distinguished from the purely parallel elements, which are often used in conjunction with pipelines. All of the operations used pipelines with two designs requiring multiphase pipelines. Some of the designs used more than one clock domain, and required data to be transferred between the domains (Celoxica, 2005, Jablonski and Gorgon, 2004).

Resource management and the scheduling of processors are also required. This can be helped through the use of processor scheduling at a global level.

Fixed-point operations are common and it would be desirable to have this as the default data type. The use of the correctly sized data types also has the advantage of reducing the amount of power and hardware required (Bainbridge-Smith, 2005).

Some operations, such as filtering are very common. These tend to have similar caching requirements but different operations on the window data. Row buffers are also often used. Lookup tables are also used to reduce the hardware computation required.

It is also important to have clear control structures (`if else` etc.) which make the nesting clear to the developer.

2.3 Other work on Image Processing on FPGAs

With the continual growth in size and functionality of FPGAs there has been increasing interest in their use as implementation platforms for image processing applications, particularly real time video processing, some of which we summarise in this section.

FPGAs allow increased performance by freeing the implementation from the fixed architecture of standard processors. However, this makes the design and implementation of the algorithm more difficult for most image processing practitioners, who are not familiar with parallel hardware issues of concurrency,

pipelining, and off-chip bandwidth. Many of the algorithms developed for image processing do not map well to FPGAs. Offen (Offen, 1985) 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. It is the mapping from a standard algorithm to a custom architecture which presents the greatest challenge to the novice FPGA designer. Serot el al (Serot et al., 1995) have also noted that programming parallel architectures for real-time image processing is often difficult and error-prone. This mainly results from the fact that image processing algorithms typically involve several distinct processing level and data representations (as illustrated in the introduction). This results in complex hardware being needed for handling this processing under real-time constraints.

FPGAs features map well to many image processing applications, as such applications traditionally process large amounts of data and involve complex algorithms (Sanderson and Shand, 2005). Image processing applications often have parallelism which can be exploited in the algorithms. There is also data parallelism due to the regular nature of images, which can allow different parts of the image to be processed separately with the results combined at a later stage. Filters which are regular in nature can be easily implemented onto FPGA devices using simple row buffers and registers. From this general structure any local filter can be implemented.

Most early papers looked at the implementation of specific image processing operations on FPGAs, either looking at applications involving filtering (including morphological operations) as these are very processing intensive operations. The authors in (Chanussot et al., 1999, Benkrid et al., 2002a, Swenson and Dimond, 1999) looked at image filtering and all three use a window structure similar in design to that in Figure 2.20.

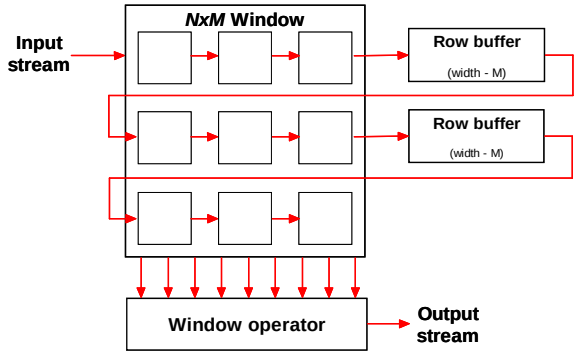


Figure 2.20: Window Filter Structure

Binary morphological operations were implemented in (Velten and Kummert, 2002) using binary logic and a variable window structure which was used as the

structuring element for the image. Alves and Akil (Alves de Barros and Akil, 1994) presented a range of low level image processing operations and architectures and developed a cost function for each method based on the FPGA resources each one used. Draper et al (Draper et al., 2003) created designs for Prewitt, Canny, Wavelet and Dilation operations. Most papers on image processing algorithms and systems implemented on FPGAs focus on the speedup of existing algorithms through the use of hardware. They often gloss over the process which was followed to convert an algorithm to an FPGA design.

3D image processing is possible on FPGAs in real-time, such as 3D image preprocessing for image-guided medical interventions (Dandekar et al., 2007). The paper identifies that a pipelined implementation improves the system throughput. They look at what parts of the algorithm are the key producers and consumers. To save on hardware and increase speed they use fixed-point numbers. LUTs were used to implement the diffusion function to save on resources. The trades-offs identified in converting the design are similar to those which we have identified as are the techniques used to solve them.

A real-time motion estimation FPGA architecture (Babionitakis et al., 2008) looks at a motion estimation architecture for block-matching search techniques. As motion estimation requires access to large amounts of data, much of the paper is dedicated to the design of the data cache.

The paper on reconfigurable computing: design methodology for real-time image processing (Kessal et al., 2008) takes a different approach to most of the other reviewed work. They compare a number of systems for dynamic reconfiguration. The idea behind this is to have each operation of the image processing system swapped in as it is needed. This is an interesting idea and more complicated designs can be implemented onto the FPGA. It allows for real time processing but would have a higher latency compared to a dedicated FPGA system. The thesis by Sedcole (Sedcole, 2006) looks at the architectures required to allow the reconfiguration of platforms in FPGAs for the use with video processing.

The Journal of Real-Time Image Processing had a special issue on FPGA technology (Constantinides, 2007) to quote from this overview: *"One application area that has been quick to reap the benefits of FPGA-based computation is the field of real-time image and video processing, where the type of algorithms commonly seen can be highly parallelised, and where frame-rates impose real-time constraints."*

Of the papers in this special issue (Constantinides, 2007, Quinn et al., 2007, Dias et al., 2007, Chrysos et al., 2007, Lindoso and Entrena, 2007, Fahmy et al., 2007, Smach et al., 2007, Sosa et al., 2007, Ramirez-Agundis et al., 2007, Zhang et al., 2007, Hassan and Carletta, 2007, Saegusa and Maruyama, 2007, Trieu and Maruyama, 2007, Kerhet et al., 2007), many take a different approach from that taken in this thesis by developing customised microprocessors rather than specialised hardware designs.

Lindoso et al (Lindoso and Entrena, 2007) present several options for image correlation, using both FFT based and direct approaches. They have used an FPGA with specialised DSP blocks for their implementation. This means that DSP operations can be mapped to these instead of general purpose logic blocks. This paper presents similar trade offs required to mapping an algorithm to hardware from software to those we have identified.

Fahmy et al (Fahmy et al., 2007) have taken the system architecture into account when designing the algorithm for the trace transform. There is a top-level control block that controls the other blocks and also manages the external RAMs used for image buffering. The implemented function blocks are quite complex and they used BlockRAM as lookup tables to implement a number of functions.

In their paper on change-driven data flow image processing architecture for optical flow computation (Sosa et al., 2007) the authors have developed a change driven pipeline where each pipeline stage has a control stage which has a FIFO. The stage will run when there is data in the FIFO. This is a pipelined architecture with specialised token based control.

In their paper on hierarchical pipelining architectures for lifting-based 2-D DWT (Zhang et al., 2007) have created what they call hierarchical pipelining. They use a FSM (finite state machine) as a scheduler that is used to choreograph the overall data flow and processing. There are five states: reset, idle, horizontal transform, vertical transform and configuration. Each of these states contains more phases of operation (other states). This use of a hierarchical state machine is one approach that could be taken to scheduling complex designs if appropriate tools were available.

In their paper (Hassan and Carletta, 2007), the authors present an FPGA-based architecture for local tone mapping of grey scale high dynamic range images. They have studied the peak signal-to-noise ratio and have found that the fixed-point hardware approximation produces results similar to a floating-point original.

In their paper on real-time image segmentation based on a parallel and pipelined watershed algorithm (Trieu and Maruyama, 2007) the authors present a watershed algorithm on an FPGA which takes a connected component based approach. The algorithm is pipelined with four main stages. Like many of the other implementations discussed in this section, the pipeline, memory bandwidth and buffering are all major design features. The design has been significantly modified from the standard algorithm to allow an efficient FPGA implementation.

The paper by (Ratha and Jain, 1999) on computer vision algorithms on reconfigurable logic arrays is an early paper that looked at implementing the whole pyramid of image processing operations on to FPGAs. It identifies that FPGAs can be used for different levels of vision algorithms. They also illustrate that different styles of parallel programming are possible. These are not all directly applicable to modern FPGAs as most of these designs deal with having an array of interconnected FPGAs. However, the use of partitioned images, parallel hardware blocks and pipeline processing can be applied to larger, more modern, FPGAs.

More recent work looked at using a dataflow based mapping approach for implementing computer vision algorithms onto FPGAs (Sen et al., 2007). They present a conceptual tool for use by the designer to aid in the design process for implementing an image processing algorithm on FPGAs, rather than relying on automated translation (as discussed in language section). This is an important front end step for exploiting an FPGA's architecture as modelling computer vision application using dataflow graphs can lead to useful formal properties, such as bounded memory requirements and efficient synthesis (Bhartacharyya et al., 2000). The (Sen et al., 2007) paper advocates thinking about the design in terms of dataflow, but then use current HDL tools to implement this design, with the implementation being improved as it is dataflow based. This is similar to creating a streamed processing based design, which makes the dataflow more explicit and visible.

There are many ways to implement image processing algorithms on to FPGA devices. This is unsurprising as the flexibility of FPGAs allows for many different approaches to be taken. However, there are common trends of the use of pipelining, data access strategies which reduce memory access (caching and streamed processing), fixed-point data types and algorithm redesign.

2.3.1 More General FPGA Design Papers

The paper "Achieving high performance with FPGA-Based Computing" (Herbordt et al., 2007), gives a very good overview of the compromises required to

achieve a high performance on FPGAs. The paper presents twelve methods to avoid the generation of *implementational heat*. They can be categorised by the type of support required to implement them: programming tools, libraries, programmer awareness of the target architecture, or no support at all. They can also be categorised by the type of method: *Application restructuring,* restructuring the application to enable substantial FPGA acceleration, *Design and implementation,* logic or FPGA specific design issues, *Arithmetic operations,* these are methods to deal with arithmetic operations, and *system and integration issues,* issues to do with flexibility and scalability. The methods are summarised in the table below.

Table 2-1 Methods for avoid generating implementational heat (adapted from (Herbordt et al., 2007))

Type of support required	Methods supported	Type of method
Electronic design automation: languages and synthesis	Use rate-matching to remove bottlenecks	Design and implementation
	Take advantage of FPGA-specific hardware	Design and implementation
	Use appropriate arithmetic precision	Arithmetic Operations
	Create families of application, not point solutions	System and integration issues
	Scale application for maximal use of FPGA hardware	System and integration issues
Function/arithmetic libraries	Use appropriate FPGA structures	Application restructuring
	Use appropriate arithmetic mode	Arithmetic Operations
Programmer/designer FPGA awareness	Use an algorithm optimal for FPGA structure	Application restructuring
	Use a computing mode appropriate for FPGAs	Application restructuring
	Hide latency of independent functions	Design and implementation
	Minimise use of high-cost arithmetic operations	Arithmetic Operations
None	Living with Amdahl's law	Application restructuring

This implies that the FPGA awareness methods, the use of appropriate FPGA structures and the use of appropriate arithmetic precision are important requirements for a development environment.

The paper on reconfigurable computing architectures and design methods by (Todman et al., 2005) is an overview paper of both the design process for implementing algorithms on to FPGAs, the possible system architectures, and the languages. The paper identifies five classes of system architectures, stand-alone processing unit, attached processing unit (like co-processor but connect to CPU via cache), co-processor, reconfigurable function unit in a CPU, and a large FPGA with the CPU embedded in it as opposed to the three main types we have identified. These different types of design can be thought of as extensions to the three we have identified.

2.4 Languages for Hardware Design

Hardware was traditionally designed through the use of schematic entry. Schematic diagrams can capture a design at its lowest level and be used to build more complex functions. However, this approach does not lend itself to the design of algorithm, with many tedious steps required to layout the logic. Instead, more flexible languages which could allow a more algorithmic design approach have been developed.

2.4.1 Hardware Description Languages

The present approach for the design of both FPGA and ASIC circuits is by specifying the functionality using a hardware description language. The two main standards are Verilog (IEEE, 2008), and VHDL (Accellera, 2008) both of which offer similar design features to digital hardware engineers. HDLs can be thought of as the assemblers of hardware programming, providing great flexibility at several levels from gate level through to register transfer level (RTL).

VHDL

VHDL (Accellera, 2008) can be used to model any digital system from a simple gate to a complete digital electronic system, with support for both top down and bottom up design methodologies. Bhasker (Bhasker, 1999) describes VHDL as an integrated amalgamation of several languages: sequential language, concurrent language, net-list language, timing specifications, and a waveform generation language.

VHDL therefore, has constructs that enable the concurrent or sequential behaviour of a digital system to be expressed with or without timing. It also allows the system to be modelled as an interconnection of components, and tested using waveforms generated using the same constructs. Bhasker also states that "*Because*

VHDL provides an extensive range of modelling capabilities, it is often difficult to understand." He also notes that a subset is usually sufficient to model most applications.

One of the key abilities of VHDL is synthesis, where behavioural models that conform to a certain synthesis description style are capable of being synthesized to gate level descriptions or to programming files for FPGA implementation (Bhasker, 1999). The flexibility of the language means that the designer can make a system which can be modelled but not built because it is un-synthesisable. Therefore, the designer is required to learn which parts of the language cannot be used to build real hardware, a confusing process, especially for someone learning the language.

In VHDL, the basic abstraction for a digital circuit is called an *entity*. An entity comprises a declaration (which describes how it is accessed using ports) and an architecture body (which specifies the operation to be performed). As a general purpose language it provides no image-processing specific operations, although these could be constructed as libraries.

Verilog

Verilog was developed in the early 1980s as a hardware modelling language, and has since become an IEEE standard 1364-2001 (IEEE, 2008). Like VHDL there are both sequential and parallel constructs. It is also based on a hierarchy of modules. Modules are defined with their inputs and outputs, after which statements, wires and registers are defined. Within each module execution blocks define the behaviour of the module. Only a subset of Verilog language statements are synthesizable.

Like VHDL the constructs provided by Verilog are low level and do not provide domain specific operations or modelling tools.

Others

AHDL (Altera, 2008a) a proprietary language, developed by Altera, similar to VHDL and Verilog, where all language constructs are synthesizable. This language has a limited use as it can only target Altera devices.

JHDL (Bellows et al., 1999, Bellows and Hutchings, 2001, Bellows and Hutchings, 1998) is a set of FPGA CAD tools developed at Brigham Young University's Configurable Computing Laboratory that allows the user to design the structure and layout of a circuit, debug the circuit in simulation, compile the net list and do for bit-stream synthesis. It is an exploratory attempt to identify the key features and functionality of good FPGA tools (JHDL, 2005). JHDL has many

powerful features including support for in-circuit debugging and partial reconfiguration all from within a single IDE. JHDL is a completely integrated design environment which presently only supports Xilinx FPGAs.

Quartz (Pell and Luk, 2005) is a structural hardware description language which is designed to make structural hardware designs more compact. The authors contend that using fewer lines of code to describe an operation can make the design clearer and easier to understand. This may not be the case, as it depends on the expressiveness of the lines that are used. It also has a compiler which optimises for power.

Hardware description languages have been developed to be very general purpose and allow a great degree of control. Due to the control structures offered in HDLs, they tend to have verbose and low-level design constructs. This means that HDLs in general are very powerful and flexible but can be difficult to program. The need to specify both dataflow and the operations on the data make HDLs an unappealing option for the majority of image processing designers. Overall, HDLs do not offer the features which image processing developers require for developing real-time applications for FPGAs.

2.4.2 High-level Languages

There has been a growth in high-level languages to reduce the design gap between traditional HDLs and the problems which developers are trying to solve. A recent review (Todman et al., 2005), covers many different languages. They separate the languages into annotated/constraint driven and source-driven compilation languages. This thesis has broken up the languages based on a programmer's perpective which results in a similar segmentation. The thesis groupings include parallel languages extensions, languages which have extensions for parallel operations, and serial language extensions, which are hardware compilers from source code (which may require some additional information or restrictions on the code).

Many of the languages which have been developed have been based on C. Edwards (Edwards, 2005, Edwards, 2006) looks at the challenges of hardware synthesis from C-like languages. Edwards notes that concurrency needs to be added to the language. This has been done by either adding concurrency with parallel constructs (about half of the reviewed languages did it this way) or retaining a sequential model and relying on the compiler to extract the parallelism. Adding

parallel constructs modifies the language substantially, so requires the developer to modify their thinking.

2.4.3 Parallel Language Extensions

Many programming languages have been extended to allow the implementation of hardware in a higher level language. SystemC (Swan, 2001) and Handel-C (Celoxica, 2004) will be discussed in detail.

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 support process threads. The lengths of data types are defined by either the fixed architecture of the processor (ANSI C (Schildt, 1997)) or by the language (Java).

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.

Both Handel-C and SystemC offer this type of functionality.

SystemC

SystemC is a modelling language based on C++. It is intended to enable system-level design at multiple levels of abstraction (Swan, 2001). It can be used for software hardware co-design and partitioning, and for modelling a complex system. It extends C++ with classes for concurrency, hardware data types, clocks, reactive behaviour and communication. SystemC is like VHDL, it includes the concept of modules and ports for transferring data into and out of modules. As in VHDL, modules contain processes where functionality is described. Processes run concurrently, but code inside a process is sequential (Ganguly, 2001). SystemC also provides a rich set of data types, allowing the user to set widths and to define fixed point data types for DSP operations.

As SystemC is a general purpose hardware design language, it does not provide any specific support for image processing. The way that ports, methods and processes need to be defined is very closely related to hardware and it is not intuitive for someone used to writing C++ code. That said, SystemC provides excellent tools for the simulation of systems at both high-level and RTL levels.

Handel-C

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. It has parallel key words and features similar to Occam. The language is designed to allow software engineers to express an algorithm without any knowledge of the underlying hardware (Alston and Madahar, 2002).

Assignment statements in Handel-C take exactly one clock cycle and there can be no side effects. 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 (Alston and Madahar, 2002). Handel-C controls the execution sequence using a one-hot state control similar to that described in (Page and Luk, 1991).

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 standard sequential operation.

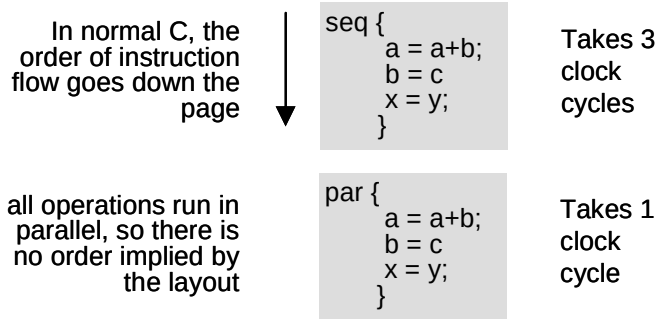


Figure 2.21: Logical flow of instructions, and time to run

Handel-C provides a good level of abstraction from hardware design. However, as Figure 2.21 shows, there is almost no textual difference between sequential and parallel code. This makes errors in design easier to make and harder

to spot, leading to errors in program execution. Handel-C has been used for many image processing implementations on FPGAs (Celoxica, 2005, Jablonski and Gorgon, 2004, Appiah et al., 2008, Fahmy et al., 2005, Gribbon et al., 2004) .

The increased level of abstraction from a hardware definition to a high level language enables the developer to concentrate on algorithm development. Overall both SystemC and Handel-C offer an improvement in terms of abstraction over lower level hardware description languages. However, they do not have any features which aid the designer to visualise the effects of concurrency. Their textual nature as illustrated in Figure 2.21 does not allow for a good representation of concurrent operations, and by extension more complex concurrent processes.

Other Languages

There are many other languages which have been implemented, some of which are discussed below. In general the criticisms of Handel-C and SystemC can apply to these.

Streams-C for FPGA design (Gokhale et al., 2000) uses an extension to C using special constructs provided via a library. The design can be simulated with a functional simulation or compiled for FPGA hardware via VHDL. This language creates a controlling state machine for the execution of the data streams as they move through the processors.

Sea Cucumber (Jackson et al., 2003) is an extension to the Java language using a library for special functions. This language uses the data-transfer scheme CSP (Hoare, 1985). This can be represented in both hardware and software. Both software and hardware can be created. It only offers two communication primitives put() and get() which communicate via a channel. Like Handel-C, primitives can either be placed in a parallel group or sequential group.

SPARK (Gupta et al., 2003) is an extension to C, which provides the developer with a toolbox that allows them to heuristically select and control code transformations. It does this by producing a representation of the program in the form of hierarchical task graphs which can be analysed to produce optimised code.

2.4.4 Serial Language Extensions

These aim to convert “standard” software code to hardware. Many of these hardware “compilers” have been developed, although we will concentrate on two of these that have been aimed at image processing: SA-C (single-assignment C, from the Cameron project (Hammes et al., 1999a, Rinker et al., 2000, Draper et al., 2000,

Bohm et al., 2002a, Rinker et al., 2001, Draper et al., 2003, Bohm et al., 2002b, Najjar et al., 2003, Bohm et al., 2001, Draper et al., 2002, Hammes et al., 2001a, Draper et al., 2001, Hammes et al., 2001b, Hammes et al., 2000, Hammes et al., 1999b, Bins et al., 1999)) and Match (Banerjee et al., 2003, Banerjee, 2003, Haldar et al., 2001b, Banerjee et al., 2000, Haldar et al., 2001a, Nayak et al., 2001, Haldar et al., 2001c, Haldar et al., 2000) (derived from MatLab (MathWorks, 2005a)),. The aim of these systems is to move the extraction of program parallelism away from the designer to an intelligent compiler.

Match

Match needs to convert from the vector operation in Matlab to `for` loops unless there is an optimized core which can be used for the operation (such as a fast Fourier transform). It then converts complex expressions to single operation statements, in a process called *levelization*. Pipelining is done by finding data dependences within the loops and loop unrolling.

Match also performs a large number of optimization steps to allow it to convert MatLab code including lookup table generation and optimization of both space and access time (Nayak et al., 2001). The Match system has the advantage that an algorithm can be designed and tested in MatLab. Once tested, it is then compiled to the target FPGA using VHDL code. Since Match was developed, its technology has been transferred to a start-up company AccelChip (AccelChip, 2005), and is now part of the Xilinx tool set.

SA-C

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 common sub-expression elimination and temporal common sub-expression elimination (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. SA-C has been used to implement common image processing functions (Draper et al., 2002, Bohm et al., 2001, Draper et al., 2003).

With both Match and SA-C the designer only needs a very limited understanding of the underlying hardware that implements the algorithm. The designers of these languages have the philosophy that developers will write and test their algorithms on a PC. Once this is done they will recompile to hardware to get a speed up in processing. Converting an algorithm from software to hardware can result in significant speed improvements. However, only some of the potential benefits are realised. Match and SA-C can only express algorithms sequentially, where there may be a more efficient concurrent algorithm. These design environments also limit the designer to working in a random access processing mode where the desired image is loaded into memory and then processed until finished.

Other Languages

There are many other serial language extensions. In general what has been said about Match and SA-C applies to them.

DIMEtalk (Sanderson, 2004) is a tool and IDE which allows the encapsulation of VHDL and DIME-C (Genest et al., 2007) code into blocks which can then be connected together graphically. It aims to simplify the communications infrastructure required for the FPGA computational components. The visual component of the IDE is a networked visualisation based on the block and wire metaphor and allows the connection of computational blocks via well defined interfaces. Code can be imported into the tool and the communications interface is warped around this to allow it to work with the DIMEtalk design tool. DIME-C (Genest et al., 2007) is a C to VHDL compiler that is integrated into the DIMEtalk IDE. DIME-C attempts to be ANSI compliant but removes side-effect expressions.

ASC (a stream compiler) (Mencer et al., 2003) is a hardware compiler based on C++ using a class library to provide timing and architecture constraints to C++. The developer needs to write the design using the stream class library and follow particular conventions. It uses a stream processing model. Hardware types and modules are generated from the code. Like other languages, loop unrolling is performed by the compiler. It allows the exploration of the design space through looking at the trade offs between area, latency and throughput for the hardware design.

Haydn-C (Coutinho and Luk, 2003) is a hardware compiler language that is based on C. It extends Handel-C by allowing the developer to apply design constraints which can be modified to allow a finer degree of control when optimising the design.

Bach (Yamada et al., 1999) is a hardware compiler language based on C with extensions for parallelism and channel communication. It is designed to facilitate automated high-level design and synthesis of communicating parallel processes. Validation of the design from the processing is done both in software (functional) and hardware (clock accurate), allowing the same test benches to be used for both.

(Weinhardt and Luk, 2001) introduces SPC, a "hardware compiler" which takes C and creates an EDIF file. By producing an optimised pipeline circuits from high level C programs. Minor changes to the C code may be needed to enable vectorization and pipeline synthesis. The authors state that their system is better than other C based design tools as the parallel sections do not have to be specified by the developer and it synchronises the parallel and pipeline components of the design. The author of this thesis disagrees with this approach of hiding all these design choices, because this approach can only speed up a serial algorithm, and cannot exploit all of the parallelism available. Transforming the algorithm into one designed for an FPGA implementation can result in a better solution than a direct mapping (even if optimised) of a serial algorithm.

The thesis (Benkrid, 2000) and paper (Benkrid et al., 2002c) aimed to produce a high level programming language for implementing image processing on FPGAs and hide all complexity from the user's point of view. This should produce an efficient FPGA configuration directly from the language to improve the design cycle time. To do this, they created a language based on HIDE which is based on an earlier HDL design based on Prolog. This new language HIDE4K allows users to develop applications in an application unit developer. This design is then translated to automatically generate the hardware blocks. This is done by producing a number of different directed acrylic graphs (DAGs) of the design which are compared by a cost function and selected by the user to produce the required netlist. These DAGs are created from the user's design using a library of optimised basic building blocks (such as line buffers, adders, multipliers etc) and user parameterisable hardware skeletons (user parameterisable hardware blocks). Hardware skeletons have been created as part of this thesis to allow for efficient and parameterisable code to be produced while hiding the complexity of the design from the user. These are created in an architecture builder in either EDIF or VHDL and then stored in a library for future use. As more library functions are added, the number of possible DAGs to solve the solution increases. HIDE4k and the concept of hardware skeletons have allowed some of the complexity of hardware design to be hidden from the developer of image processing application in hardware. This is done at a higher level than systems such

as SA-C and is similar to the replacement scheme used by Match. The limitation is that the user needs to implement a design that can be mapped to the available DAGs. However, the developers have more flexibility than SA-C and Match as they can select a design that meets their needs.

2.4.5 Hardware Software Co-Design Languages

The Dynamo system provides a runtime environment for mapping image processing applications to hardware/software platforms that contain a mix of software processors and reconfigurable hardware (Quinn et al., 2007). Dynamo has been developed to allow an image analyst to create designs with no knowledge of hardware or software design. The analyst specifies the algorithms by constructing a processing pipeline from a library of operations. These operations have had their performance modelled and this is used to develop a mixed software-hardware architecture with the performance costs traded off to meet the desired operation requirements. This is done at run time. There are several limitations of this approach. Firstly the execution time is not known until run time. Secondly, the analyst is restricted to a subset of image processing operations which have been developed. Dynamo requires the algorithm to be implemented to be able to be represented as a pipeline, which not all standard operations can be. This system produces a pipeline processor, running different sections on hardware or in software as appropriate.

A recent experimental language hardware join Java (David and John, 2006) aims to allow a transparent hardware software interface and an integrated expression of dynamic reconfiguration. It supports a static approach where the hardware is designed as a single configuration, and a dynamic version which supports partial dynamic reconfiguration of the FPGA at run time. This runs code on a java virtual machine running on the FPGA and configures the hardware elements. These are controlled by a state machine which can execute them and stores the current state of the hardware so that it can be scheduled by the software part of the system. In contrast with most systems, which require separate development of the hardware and software, this is a tightly coupled approach to hardware/software co-design.

2.5 Languages for Image Processing

There are there are many tools, and libraries which cater to image processing tasks. These include libraries, and templates for C and C++ (such as CImg (Tschumperlé, 2008) and OpenCV (Intel, 2006), both of which are popular in image processing due to their speed and portability).

Interactive design tools are often more useful for development of image processing systems and algorithms. Once a design is finalised it can either be used as is, or ported to a language such as C.

Matlab (MathWorks, 2005a) has an image processing toolbox which provides a large number of operations and tools to aid in the design of image processing systems and algorithms. VIPS (Bailey and Hodgson, 1988) is an image processing design tool. It allows interactive design using operations in a command line, or through the use of scripts. There are many other design tools that offer similar functionally to these two (including another system also called VIPS (Cupitt et al., 2009))

There has also been a large amount of work on developing an image algebra by the University of Florida, beginning in the 1980s (the history of image algebra is summarised in (Wilson, 2009)). This work was extended to parallel machines in (Wilson et al., 1998), and has been used as the basis of an Image Algebra Language (IAL) (Crookes et al., 1989, Crookes et al., 1990) and the Tulip Language (Crookes, 2009, Steele, 1994) for use on Transputers and standard microprocessors. Such languages enable a compact representation of low to medium level image processing operations that is implicitly parallel at the pixel level. Such a representation can be readily mapped to a streamed or other parallel implementation, as was explored by (Benkrid, 2000). An FPGA implementation would still require the underlying support structures to be implemented, for example as hardware skeletons (Benkrid et al., 2002b).

As image processing algorithms are often a collection of operations which pass data between each other there have been several tools which allow a box and wire design approach.

2.6 Image Processing Visual Tools and Languages

There have been a number of different visual image processing languages for use on a serial computer including Khoros (now Cantata) (Williams and Rasure, 1990, Konstantinides and Rasure, 1994), OpShop (Ngan, 1992) (Figure 2.22), and others (Atkins et al., 1994, Del Bimbo, 1993, Cinque et al., 2007, Tanimoto, 1990). There are also several general purpose visual languages which can be used for image processing including LabView (Instruments, 2005) and Simulink (MathWorks, 2005b). LabView and Simulink now have extensions that allow them to be used for FPGA design. There have also been several research projects on converting Khoros diagrams to FPGAs (Hoisko et al., 1996) and CHAMPION (Levine et al., 1999). These

both use HDL libraries of Khoros functions. Finally, the PixelStreams (Celoxica, 2005) graphical interface which maps stream based image processing operations to Handel-C code, could be considered as a visual language.

These languages all follow a form of the dataflow paradigm. Streams of data flow through a network of nodes which perform a computation on the tokens within the stream before passing the output data to the next function block (Buck, 1993). It has been noted (Buck and Lee, 1993) 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. This allows pipelines and limited hardware resources to be exploited. Many compilers for pipelined or parallel machines rely on this type of analysis. SA-C uses a dataflow graph back end so that it can analyse and construct the VHDL code used to programme the FPGA (as described in (Hammes et al., 1999a, Rinker et al., 2000, Rinker et al., 2001)).

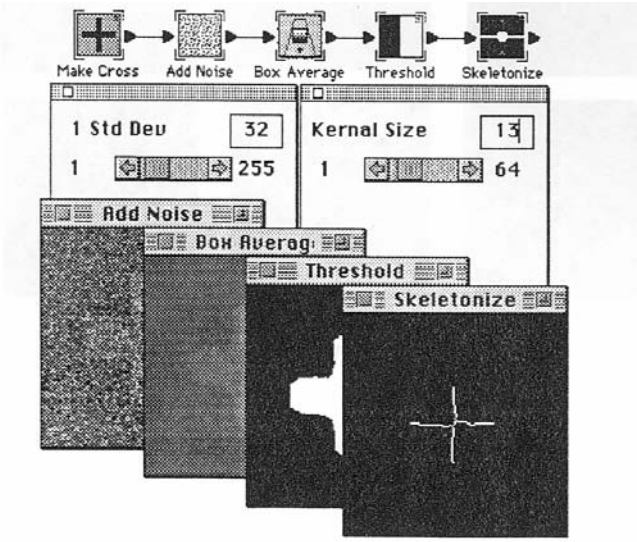


Figure 2.22: Example of IP-Core based design, OpShop algorithm for Abingdon cross benchmark showing blocks, their parameters and their effects. (Ngan, 1992)

The visual languages above can be broken up into several types. Some are aimed at sequential operation, while others are designed for or aimed at parallel operation on FPGA devices. Although an important differentiator, the generality of programming is a more appropriate classification as sequential programmes can be implemented on an FPGA, although usually with limited speed up. Of these languages there are IP-core based and low-level design based approaches. IP-cores are predefined, often parameterisable blocks which can be connected together to

form the required function. If a required block is not available then it needs to be created using another language. This is the case with Khoros, Opshop and the PixelStreams GUI. Simulink and LabView offer a mixture of IP-core functions and low-level operations which can be use in conjunction with the IP blocks or used to build customised function blocks. This makes these two systems far more flexible, as all of the design and testing can be performed in the same system.

CHAMPION (Levine et al., 1999) was expanded on in (Sze-Wei et al., 2001). In this paper they expand on the system overview, compilation and library cell development and verification. CHAMPION uses the Khoros graphical language to produce configuration code for a number of different FPGA development boards. The design is developed and tested in Cantata, using library cells which have both Cantata and VHDL code associated with them. When the design is to be implemented the logic cells have their data types matched (using padding and truncation) then they are synchronised so that longer operations have the same delay as sorter parallel ones. The VHDL code for each cell is then connected together and the design created. Adding new logic cells to the CHAMPION library requires their operation to be described in fixed-point C/C++ and in VHDL code. These need to be functionally equivalent otherwise when compared the system will reject them from being added, this is done by comparing the results from test vectors applied to the C/C++ and VHDL code. To reduce this double overhead, a commercial C to VHDL converter has been incorporated into the system. The system is limited by the need to design in C for a software system, then have this converted into hardware. This allows both the software and hardware implementations to be equivalent and have the same source. However, the software to hardware compiler approach limits opportunities for hardware design optimisation. This approach does have a lower time to produce a system than with a pure hardware design approach.

Another approach to visual dataflow programming for real-time parallel vision is taken by (Serot et al., 1995). Their goal was to implement real-time image processing algorithms on a computer made from 3-D interconnected data-driven processing elements. Rather than manipulating dataflow graphs, a functional language is interpreted on the fly to produce a dataflow graph which is used to visualise the system. This graph is also the programme and it is this which is mapped to the computer system. With the ease of display of graphics on modern computers there seems little need to go from a textual representation to a graphic when manipulation of a graphic can be so much more intuitive to the designer.

Celoxica has identified that *"The most difficult aspect of creating pipelined hardware is often handling flow control correctly: the PixelStreams architecture effectively eliminates these issues by providing re-usable flow control components."* (Celoxica, 2005). PixelStreams achieves this through a library of parameterisable IP blocks (or filters). These filters can then be assembled to form a video processing system. This is done either using the PixelStreams GUI or directly in Handel-C. Celoxica recommends that *"Although the GUI is useful as a training tool, we strongly recommend graduating to a programmatic style as this greatly increases the flexibility of the system."* The GUI is very simple and allows for the connection of blocks with wires and for the parameters of the blocks to be edited. The GUI only supports the provided blocks and requires users to develop and integrate custom filters in Handel-C. PixelStreams uses "Streams" which are containers for passing data, giving details on whether the data is valid, or if a process should halt. This is a token based approach. Although the PixelStreams library is extensive, for many applications new blocks would need to be built, eliminating the ability to use the GUI. This limits the visualisation of the system to a collection of Handel-C function calls.

As discussed earlier, the authors in (Cinque et al., 2007) have a novel approach to both image processing development and have created a visual tool to aid in image processing development. They aim to provide transparent visualizations for states. This is done by making the internal dataflow and relationships clearer so that humans can more easily verify the correctness of the system's computations. The developed toolkit aims to assist the designer in formulating, testing, and keeping track of alternative sequences. It does this in an interesting way by allowing multiple options for operations to be constructed in a tree structure. Each of these is then evaluated and a scaled down version of the result is shown. The tree can be expanded as needed. An example of this is shown in Figure 2.23, where several different approaches to detect a face in an image are evaluated against each other. As image processing is often developed in a heuristic nature where many options are investigated, this type of view can allow multiple options to be evaluated at once. This is a good way to develop image processing applications. However, the visualisation requires a large set of operations to work from and therefore is not suited to low-level algorithm design.

Del Bimbo looked at the design and implementation of an iconic environment for image processing operations (Del Bimbo, 1993). The work created a graphical language which sits on top of a standard image processing language. The

environment allows an image to be selected and the results of operations applied shown.

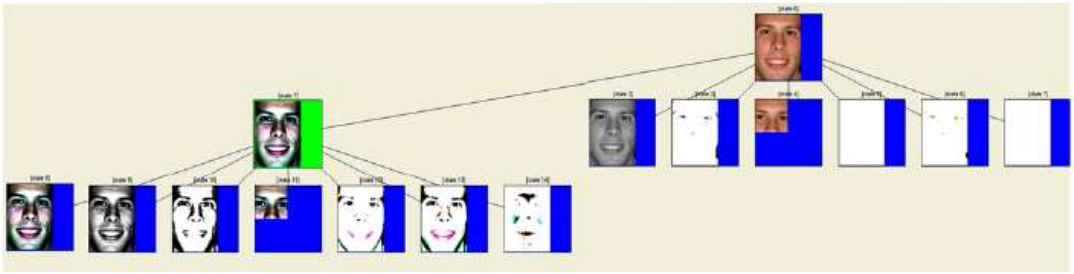


Figure 2.23: TRAIPSE example for a three-level tree of stages constructed using face image processing operations from (Cinque et al., 2007)

Tanimoto (Tanimoto, 1990) investigates the use of visual programming languages for image processing. VIVA is designed to provide a lot of feedback to the programmer during the construction of the program and the paper introduces the concept of liveness which measures how interactive the design environment and the programs execution is.

2.7 Visual Languages

This section will concentrate on visual languages in general. What is a visual language? Shu (Shu, 1986) presents a good definition of what makes a visual programming language (VPL) and how they are separate from visual environments. According to the paper *a VPL can be informally defined to be a language which uses some visual representation (in addition to or in place of words and numbers) to accomplish what would otherwise have to be written in a traditional one-dimensional programming language.* This is a broad but good definition as it is not restrictive in the mixture of text and graphical representation.

The paper "The Future of Visual Languages" (Chang et al., 1999) further developed the ideas of what a VPL is. The authors believe that *"VPLs are not about graphics, are not about minimal usage of text, and are not synonyms for "icons on strings" -- VPLs are about multidimensionality. Examples of multidimensionality are the use of icons, the use of spatial relationships, or the use of the time dimension to demonstrate "before-after" semantic relationships."*

The author also states that VPLs have had three main goals:

"(1) to aim to make programming more accessible to some particular audience,

(2) to aim to improve the correctness with which people perform programming tasks, and/or

(3) to aim to improve the speed with which people perform programming tasks."

The sections author also states that *"most VPL designers have included in their languages one or more of the following attributes, all of which are facilitated by multiple dimensions:*

a) Concreteness: Concreteness is the opposite of abstractness, and means expressing some aspect of a program using particular instances.

b) Directness: Directness means that there is a small distance between a goal and the actions required of the user to achieve the goal.

c) Explicitness: An aspect of semantics is explicit if it is directly stated, without the requirement that the programmer infer it.

d) Immediate Visual Feedback: In the context of visual programming, immediate visual feedback refers to automatic display of the semantic effects of program edits."

However, the paper comments that just having a VPL does not make programming easier. Visual is not always better. There needs to be an investigation into what should and should not be visual in the language.

There has been research on the use of visual languages for concurrent programming, including (Biancardi et al., 1995, Usher and Jackson, 1998, Krazlmuller et al., 1999, Exman and Citron, 1994, Sheehan, 2003, Glinert and Stotts, 1992, Loyall and Kaplan, 1992, Glinert and Norton, 1992, Roman et al., 1992, Miriyala et al., 1992, David Stotts and Furutat, 1992). Unfortunately, most of these assume an underlying architecture which makes them less relevant to FPGA design.

One of the reasons for the application of visual languages to concurrent programming is that many are dataflow based. The original motivation for dataflow languages was to exploit massive parallelism. (Johnston et al., 2004c) review the original usage of dataflow languages and then follow the development over the next three decades. Dataflow hardware was believed to avoid the bottlenecks of von Neumann hardware (global program counter and memory access). It was found over time that efficient dataflow architectures never took over from von Neumann concepts. The move away from fine-grained parallelism to coarse-grained parallelism, with groups of instructions executed in sequences, began in 1990 (Bic,

1990). This has lead to the current hybrid model for dataflow programming where dataflow programmes are implemented on sequential processors. During the nineties most of the primary motivation for dataflow programming moved from the exploitation of parallelism to looking at visual data flow languages and the advantages these have in terms of software engineering. The key advantage of dataflow is that if several instructions become fireable at the same time, they can be executed in parallel, providing the potential for parallel execution at the instruction level. Dataflow also allows for pipelining to be utilised, especially in loops, by utilising the data dependencies to locate parallelism. Like many hardware description languages, dataflow languages do not allow side effects. Although dataflow hardware did not take off, the use of dataflow concepts in von Neumann architectures was investigated. This allows for a hybrid approach with dataflow used for control and medium- to large-grained parallelism exploited. Though dataflow hardware did not become popular, FPGAs could be thought of as allowing dataflow hardware designs. This is because the hardware design is not limited by any existing architecture and the designer is free to develop the logic how they wish. This can allow for massively parallel instructions to be implemented.

With the advent of visual dataflow programming languages, dataflow moved into the area of software engineering aspects. Graphs were seen as a good way to easily communicate ideas (Johnston et al., 2004c). Human computer interaction engineering became the principal motivation for developing dataflow languages. We believe that dataflows can be used to achieve both gaols; aiding in the human and parallel design aspects. (Johnston et al., 2004c) present several conclusions about visual dataflow programming languages:

- *"they allow the developer to proceed with design and implementation in their own order, thus making the design process freer and easier;*
- *that secondary notation (comments and other annotations) could be utilized much more than it currently is;*
- *that more work needs to be conducted on incorporating control-flow constructs;*
- *that the effectiveness of program editors remains to be investigated in literature;*
- *that the problem of screen real estate is not as major as many assume it to be."*

Its also has the following conclusions concerning visual dataflow programming environment.

- "*In a DFVPL there is a blur in the distinction between language and environment.*
- *The design phase benefits the most from the use of DFVPLs over textual languages.*
- *The animation offered by a DFVPL environment is vitally important to its usefulness.*
- *The library of functions included with a DFVPL is not a major factor in productivity.*
- *Key areas requiring work include the use of secondary notation, and control flow constructs.*

One important observation on visual languages is (Chang et al., 1994):

"Metaphors play the key role in mapping knowledge from a domain in which the user is conversant into a new one, the one of the application so as to help him/her in understanding the system, its features and commands."

Therefore, it is necessary to make sure that the metaphors used to develop a new language help the developers to understand the design.

When developing a programming language, it is important to keep in mind the different mental representations that novices and experts have (Vikki et al., 1993). The paper identified *"five abstract characteristics that an experts' mental representation has which are often absent from novice representations:*

1. *It is hierarchical and multi-layered;*
2. *It contains explicit mappings between the layers;*
3. *It is founded on the recognition of basic patterns;*
4. *It is well connected internally;*
5. *It is well grounded in the program text."*

These concepts need to be taken into account when designing a language as it is necessary to aid the mappings of novices, experts and those in between. In conclusion, the paper found that experts extract many different kinds of information from a program which become a part of their mental representation. Novices do not build these representations and therefore have more difficulty with understanding

and modifying code. There are skills in understanding the program structure and the links between separate program models that need to be developed (Vikki et al., 1993). It is necessary to encode and express or enhance the developers' representations.

2.7.1 Visual Hardware Languages

There has been much work in visual language design and on the use of symbols in aiding in understanding (Shu, 1986, Chang et al., 1999, Ware, 1993, Myers, 1986, Shneiderman, 1983, Johnston et al., 2004c, Chang et al., 1994). However, there has been surprisingly little work on the use of visual languages for hardware design. Besides schematic entry, PixelStreams (Celoxica, 2005) and (Serot et al., 1995) discussed in the visual image processing language section, there are few visual HDLs. Several of these are summarised below:

The vVHDL (Miller-Karlow and Golin, 1992, Golin et al., 1993) language is a visual hardware description language. It works to construct icons to represent the different types of sequential and parallel control and processing blocks available in VHDL. The authors have constructed icons to represent different processes; concurrent ones have a rounded boarder while serial ones have square borders. This is similar to Pixelstreams and CHAMPION.

Smedley (Smedley, 1995) describes a high-level visual language for digital circuits, based on Prograph (Cox et al., 1989). This takes a Prograph-type design and uses it to produce digital circuits. Like Prograph, designs can require many levels of encapsulated design to design the system.

PICSIL (Pearson, 1992) is a dataflow-diagram-based HDL with graphical components and communication. The components can be expanded to show the logic level description in HardwareC (Ku and DeMicheli, 1990). PICSIL also introduces the idea of using complex data types represented by one wire to reduce the number of connections and screen space required.

This is similar to many modern FPGA IDEs (such as Quartus II (Altera, 2008b) and Xlinx ISE (Xilinx, 2008)) which allow modules or entities to be visualised and have their ports and connections edited graphically with the internal logic provided by an HDL. Quartus II (Altera, 2008b) also allows an RTL diagram of AHDL code to be viewed but this cannot be edited.

2.7.2 Concurrent Visual Languages

The Journal of Visual Languages and Computing had a special issue on visual languages and concurrent computing (Glinert and Stotts, 1992). *"An important goal*

of research related to visual environments is to find ways to enable highly skilled, professional programmers more easily to handle domains where traditional languages are inadequate or cumbersome" and that concurrent programming was an area where this occurred. There was a mixture of visual programming and program visualisation papers (Glinert and Norton, 1992, Loyall and Kaplan, 1992, Miriyala et al., 1992, Roman et al., 1992, David Stotts and Furutat, 1992). As programming an FPGA is similar to programming a concurrent system several of these papers were relevant.

The paper by Miriyala et al (Miriyala et al., 1992) looks at using an actor model for concurrent computation. This model allows the programming of systems without any specific assumptions about the underlying concurrent hardware. As there are at least three distinct models of concurrency, data parallelism, shared memory, and distributed memory. An actor model is used to combine procedural and data abstraction with concurrency. Actors are programs which are encapsulated in visual objects. Tokens are passed between actors to allow for the activation and control of the program. The actor-based model is an interesting way to abstract the control of the algorithm from the computation by embedding the computation in actors, and then controlling the actors.

The final paper of interest (David Stotts and Furutat, 1992) looks at hypertextual concurrent control of a Lisp kernel. This is intended to support human-directed parallel computations. This visualization tool uses a mixture of text and visual representation with the visual information mainly encapsulating communication. It aims to help with debugging and understanding the program. It presents a good argument for having multiple views of the system in development.

2.7.3 Criticism and Evaluation of Visual Languages

Green and Petre (Green and Petre, 1992) present a number of criticisms of VPLs including the fact that VPLs can be harder to read than textual programming languages. They found that for their somewhat simplistic example programs and example languages, the visual languages were slower to edit and understand than text. They found that the gate structure shown in Figure 2.24 was hard to work with. In this case a logical expression was represented with gates. Subjects found it difficult to keep their working memory updated and it was harder to understand than the textual representation. This problem can be avoided in newer versions of LabView through the use of expression blocks.

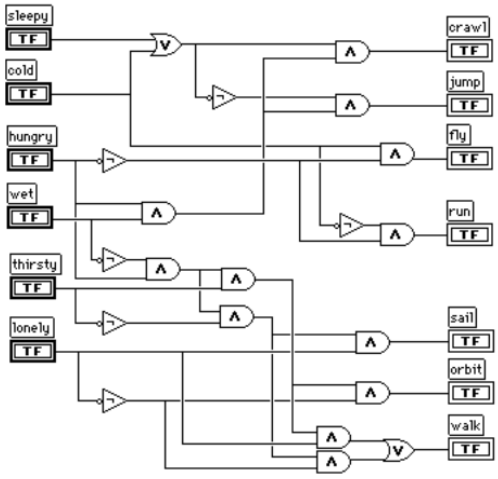


Figure 2.24:Example of Gates notation in Labview, from Green and Petre (Green and Petre, 1992)

They suggest that the slowness in using VPLs could be due to the need to *"trace programme behaviour"* and *"that not all graphical structures are equivalent"*. *" 'Knots' which force working memory load on the reader"* can be produced. They found that *"the text structures used in this study do not contain similar knots; they make a better use of spatial topography than the graphical notations!"* From these comments it can be seen that it is important when designing VPLs to achieve a balance between textual representations and visual representations, and to use each when appropriate to avoid the disadvantages which can be caused by a purely graphical representation.

UML (Unified Modelling Language) (Object Management Group, 2009) is by far the most common visual language. It is used to describe and model complex software systems. The main problem with UML is that it is too general purpose. This research aims to make a tool for a specific use, describing pipelined processing to enable the implementation of image processing on FPGAs.

2.8 Summary of Languages

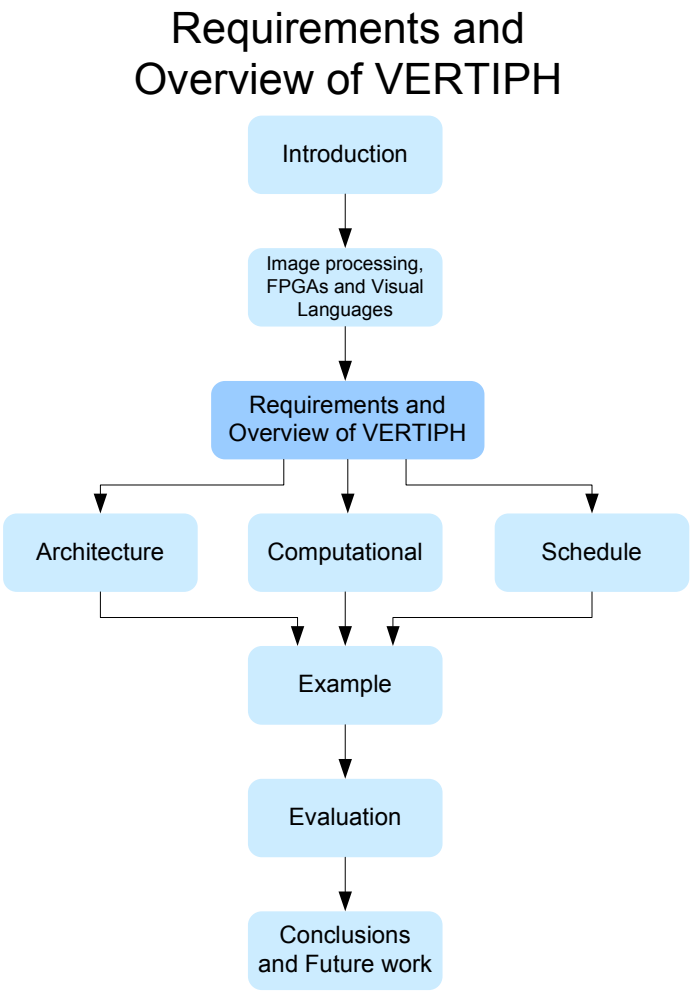
FPGAs have been found to provide an ideal target for the implementation of image processing algorithms. The ability to generate a custom computer to implement the algorithm allows real-time implementation of image processing algorithms without the cost of dedicated hardware. Programming FPGAs for image processing is not trivial, limiting their more general use.

There has been a move away from low level HDLs to higher level programming languages. Some, such as SA-C and Match, constrain the developer to a

particular design methodology. Others, such as Handel-C and SystemC do not give a good representation of the parallel nature of the design due to their textual representation. The current image processing visual languages are good at specifying the steps of an algorithm, but they do not allow for the detailed lower level design to be done in the same system. They instead rely on a large set of pre-built operations and either do not allow the VPL to be used to develop new ones or require them to be made up of the library functions. This is not ideal for FPGA implementation due to the difficulty in optimising the design. Nor do current languages separate the design of parallel and pipelined operations.

A new integrated development tool is required to allow image processing developers to move to FPGAs and exploit the benefits in speed found in using FPGAs over conventional processors. Hardware description languages currently take two different design paths. Languages such as VHDL and Handel-C give the designer a great deal of control over the design, but require detailed knowledge of hardware. Languages such as SA-C hide the complexities from the designer either through hardware compilers or through hiding the complexities of the design through intellectual property blocks or library blocks connected graphically (such as PixelStreams). Image processing design tools are also split between intellectual property blocks or library based design tools (such as Khoros and VIPS) and the use of general purpose software languages for design.

Currently, there is no language which fits between the two extremes of hardware design. They are either lower level text based or high level library based languages. Nor is there one that has specialised image processing tools which are not block based. There is a requirement for a language which has the flexibility of traditional hardware languages but that provides the developer with better feedback on the design. The language outlined in this thesis aims to make programming on FPGA hardware more accessible to the image processing community (goal 1 of Chang et al. (Chang et al., 1999)) and to make errors less common (goal 2 of Chang et al.). A visual language can use the spatial relationship to aid the designer. Making the timing information, pipelining, parallel and sequential design more explicit and making this more direct is an important aim of a visual language for image processing on FPGA. There is a requirement that the language clearly separates sequential and parallel processors and, more importantly, that parallel and pipelined operations can be easily distinguished. There is also the need for good control structures both at the low-level and for high-level state based control.


Requirements and
Overview of VERTIPH
Introduction
Image processing, FPGAs and Visual Languages
Requirements and Overview of VERTIPH
Architecture
Computational
Schedule
Example
Evaluation
Conclusions and Future work

3 Requirements Analysis and Overview of VERTIPH

FPGAs allow designers to construct fully customised hardware for their applications. Purely sequential designs can be developed, and both fine-grained and coarse-grained parallelism are available. Fine-grained parallel processes can be used as parts of a larger, coarse grained, parallel process, or sequential processes. This makes it important to have clear and distinct representations of the parallel and sequential parts of the design so that they can be quickly identified and understood. The fact that there can be multiple levels of parallelism implies that some form of hierarchal design approach is needed.

Pipelining is one form of parallelism that is often used in image processing to obtain the necessary throughput. Current hardware description languages, including VHDL (Accellera, 2008) , Verilog (IEEE, 2008) and Handel-C (Alston and Madahar, 2002), represent pipelining in exactly the same way as other parallel operations. This causes problems because the data dependencies between the parallel operations in a pipeline are not always obvious, particularly when first reading a program. This is especially apparent if there is a mixture of parallel and pipelined design within the same block of code. Therefore, although pipelines are one form of parallel operation, from a developer's perspective they should be distinguished and treated as a special case.

Another common problem in designing parallel systems is managing resource contention between different parts of the design. As discussed in the introduction, some resources have constrained access, such as memory, I/O pins, and peripherals. It is common for several parts of a complex system to share a resource. The accesses may be interleaved, or the processes may be in direct competition. This access could be simple such as two processors wanting to access a table in off-chip RAM; one may need to write to the table and the other read from it. As long as their access is managed, conflicts can be avoided. More complex resource contention can occur, so any aid to the designer can be helpful to them. This requirement was one of the main reasons for developing the resource and scheduling view, introduced later in this chapter.

As discussed in chapter 2 three main processing models for image processing in hardware have been identified. These are the streamed, random access and hybrid processing models. A language should allow all of these processing models to be expressed.

Some image processing functions are so common and are used so often that they need to be included in any specialised image processing language as primitives. Operations that would fit into this category include row and pixel buffering, window filters, and lookup tables. Row and pixel buffering are used in many operations (Johnston et al., 2004b) to reduce memory bandwidth by caching pixel values that are going to be needed in subsequent clock cycles. Window filters are used in many applications and are a common design pattern structure with only the filter function or pixel weights different from one application to another (Johnston et al., 2004b). LUTs are also used in hardware implementations as they can reduce the hardware required to compute complex functions, effectively trading memory for combinatorial logic.

As the language developed in this thesis is targeted for image processing experts who are not necessarily experts in hardware design, the language should be intuitive and easy to use. Any language features that are difficult to use will reduce productivity and will diminish the language's usefulness.

Developers need to see information relevant to their present task or goal. This information is different depending on where they are in the development cycle and what they are trying to achieve. The type of detail that needs to be displayed for an overview of the program can be very different from what is required for the low-level pipelining of operations. High-level scheduling also needs different control representations from the control structures used at lower-levels of design.

A high-level language for expressing image processing algorithms in hardware should minimise the semantic gap between design mapping and implementation by:

- allowing a mixture of parallel and sequential design;
- making it clear to the designer what runs in parallel and what forms part of a pipeline;
- being able to detect when concurrent processes may access a shared resource such as a RAM, and manage this by informing the designer of potential conflicts and provide suggestions as to how to resolve them;
- being able to handle stream, random access, and hybrid processing models;
- including some of the common image processing functions and data types as primitives. Examples include row and pixel buffering, window filters, and look up tables;

- providing multiple views onto the design, each view designed to highlight some distinct aspect of the design.

3.1 Three Graphical Representations

As image processing algorithms often involve a sequence of relatively independent processing blocks, a high-level view would allow the designer to specify the data flow through a sequence of modules. This is then augmented with lower-level views that define the parallel computations that make up each higher-level module. Finally, a further representation is required to give the designer an overview of the system events, resource usage, and enable potential conflicts to be identified and managed. These are the three defined views of an algorithm within the VERTIPH system: the top-level *Architecture View*, the *Computational View*, and the *Scheduling View*.

A version of VERTIPH and screencasts of it in operation can be found on the attached DVD or at www.johnston.geek.nz/vertiph.html.

3.1.1 Architecture Graphical Representation

The architecture view aims to provide the designer with a perspective on the overall system. Its primary focus is on the broad decomposition of the high-level algorithm, and the flow of data between the blocks which perform specific image processing tasks. Two other image processing systems that have similar data-flow models are Khoros (Konstantinides and Rasure, 1994) and OpShop (Ngan, 1992).

The use of component blocks allows resources such as frame buffers to be encapsulated and related computational processes to be logically grouped. For example, a bank-switched frame buffer component will have both an input stream and an output stream, and it will contain two RAM banks. Other components which communicate with this only see address and data lines and the details associated with switching between memory banks can be hidden within the component.

Processors which are logically related to each other can also be encapsulated in a single module. For example, a colour segmentation and tracking algorithm detailed in (Johnston et al., 2005a, Johnston et al., 2005b) represents each uniquely coloured detected object as a bounding box. It stores the bounding boxes for each colour class in a data structure, and it incorporates processors for tracking-related bounding boxes between frames and for calculating the position of all the bounding boxes that have been detected. The data structure and the processors are logically

related and should therefore be kept together. This idea of encapsulation borrows from object-oriented software engineering.

Encapsulation simplifies the sharing of data and resources and it becomes clear which processor can access them and for what purpose. This can, in turn, make it easier to schedule these processes, 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.

Hierarchical encapsulation can allow very complex 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. It 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 the architectural view is to allow logical separation of image processing operators, to show the data flow through the operators, and to encapsulate data and processors related to each operation.

The Problem Justification

The main problem with implementing image processing on FPGAs is complexity. As discussed in the introduction, there are two levels of image processing algorithms. Application-level algorithms which produce an output based on a number of chained image processing operations - these often producing an abstract result from an image, such as a licence plate number. Each image processing operation is in turn defined by an operation-level algorithm, which represents the sequence of computational steps required to implement a self-contained operation, such as an edge detection filter. Each image processing operation can be quite complex. When these are combined to produce an image processing algorithm, the program as a whole becomes very complex. It has been shown that as a program reaches a certain size and level of complexity it becomes difficult to understand (Schildt, 2002, Green and Petre, 1996). This is true for both textual and visual languages. There is therefore a need to manage the complexity of the program.

One of the reasons for using a graphical representation for both control and data flow visual representations to make the relationships between the components of an algorithm clearer and more explicit. The connections between different parts can be quite complex and may include the passing of multiple data and control flows. This leads to a requirement for a mechanism to represent multiple data flows with a single connection. The barrel distortion correction algorithm (Gribbon et al., 2003)

discussed in the second chapter involves complex data and control paths. By being able to collapse the flows into the main paths the complexity of the design is simplified making it easier to understand.

Two or more different image processing operations may have the same (or very similar) functional output. Laplacian and Sobel edge detection filters are an example of this. Although, they produce similar results (on good images) they have very different implementations (Castleman, 1996). It is therefore beneficial if the language allows the design to be separated into modules each of which communicates with the outside world through a standardised external interface. This makes it possible to isolate any changes in the algorithm, making it easier to change from one operation-level algorithm to another. Each module may be complex and contain other parts so it is desirable to be able to separate between the steps, the operators and the modules they make up.

Characterisation

Providing an overview of the design allows the complexity of the system to be managed. This allows the program to be decomposed into logical units, to create a block level diagram. Block diagrams are used extensively as they provide a view of the data flow and allow the specific details of the process and implementation to be defined later. The complexity is managed by allowing the modularisation of the design, letting blocks contain other block diagrams.

High Level Design

The architectural graphical representation consists of computational blocks, allows encapsulation and hierarchical design, and manages data and control flow links through the use of complex data types. The overall goal of the architectural graphical representation is to provide better comprehension of the 'big picture'.

The 'wires' which connect the blocks represent data or control information. As more than one data value is passed between blocks, at this level each connection can represent a hierarchical complex data type. By collecting the data objects into one connection the diagram becomes simpler to understand as it reduces the number of connections between blocks. Blocks are provided with ports to which the connections are attached. This visual component acts as an interface at which the single-wire external representation of a data-flow can be broken out for internal use inside the block; this will be discussed in the architecture chapter.

Each architectural block either encapsulates a collection of other architectural blocks or represents a single computational block. A hierarchical representation allows a top-down programming approach where main functional blocks can be successively decomposed into smaller processes. Such a block hierarchy can allow for very complex designs to be built without overloading the developer with too much detail. At the lowest level of the hierarchy is a computational block which is opened with the computational graphical representation editor.

3.1.2 Computational Graphical Representation

It would be possible for developers who never design their own algorithms to use the architecture view's editor to assemble predefined library modules into a high-level overview. This is similar to the way that other intellectual-property-based systems such as Celoxica's PixelStreams and Xilinx's DSP block sets operate. However, VERTIPH is intended to allow the user to create their own processing blocks.

Using a low level dataflow diagram to implement the algorithms for image processing operations has several limitations. Firstly, control flows are hard to separate from data flows. The data flow of the algorithm is expressed more clearly; however the control flow can become difficult to understand and modify. The second issue is one of scope, or the level of detail required to develop a fully working image processing system. This requires a number of interconnected image processing operations. Using a single data flow graph for the whole system results in one large complex diagram. To address these issues it was decided that views that allowed for hierarchical encapsulation, computational data flow with both low-level control, and high-level control were required.

The Problem Justification

As designs on FPGAs produce hardware units which can run in parallel, programming is more difficult than with sequential software systems. FPGA designs are made up of a network of hardware units, which in most cases do not implement a purely sequential data flow. In most cases at the lower level, operations either run in parallel or form pipelines. It is only in rare cases, when timing is not critical or when data can not be processed in a pipelined or parallel manner, that sequential data flow operations are used.

Timing is important in pipelined processes; developers need to specify when operations occur in relation to each other. There is a need to be able to visualise the data dependence especially with complex pipelines that are composed of parallel

operations. This information can be more clearly expressed graphically than using textual HDLs. The extra dimension of space can be used to show the timing relationship which otherwise needs to be conveyed via special instructions or conventions in textual languages.

In many common HDLs (such are VHDL, Veralog, Handel-C) there is no distinction between parallel and pipelined operations. For example in Handel-C both parallel and pipelined operations are contained in the *par{}* statement and it is up to the developer to step through the expressions looking for the data dependencies to construct a mental representation of the pipeline and any other parallel processes. This can be a time-consuming and error-prone task. From the author's experience, as pipelines become larger, with more than one data path, errors become exponentially harder to identify. This becomes worse when the data path splits and joins within a pipeline.

The fact that pipelining, parallel and sequential operations combine to produce a network of interdependent hardware units suggests that a visual language would be a good choice of representation. The main justification for this is that textual languages do not represent networks as clearly as visual languages. Text, by its nature, is sequential, whereas visual languages can readily show two dimensional relationships. However, in addition to being expressive and clear, any representation needs to be compact, and visual languages can take up more screen space than textual languages for equivalent problems (Green and Petre, 1992, Green and Petre, 1996). Therefore, an optimal language will involve a mixture of both visual and textual representations, as discussed in the next section. To have a clear compact representation, it is also desirable to encapsulate complex multistep functions into hierarchal computational blocks like the architectural graphical representation. This can limit the number of objects on the screen and improve the understanding of the programming by hiding lower level details.

Characterisation

To reduce the difficulty of distinguishing between pipelining and parallelism in standard HDLs, a separate graphical representation is used for each.

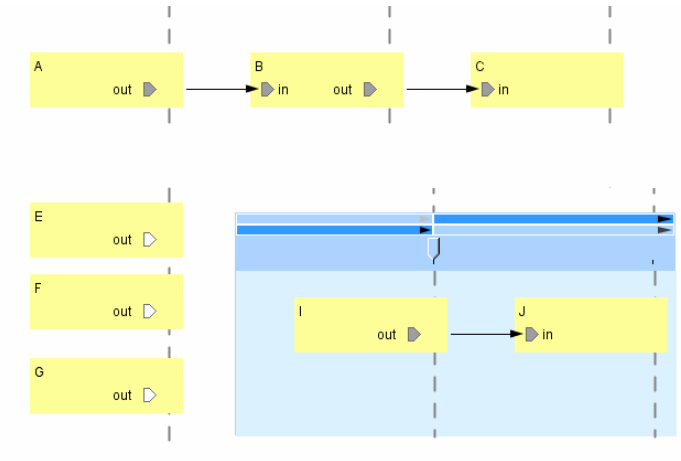


Figure 3.1: Sequential, parallel, and pipelined arrangements of modules in a design

Figure 3.1 illustrates this. Operations A, B and C run in sequence; E,F,G run in parallel, and I and J are a pipeline. Pipelined operations are enclosed in a control structure but are otherwise treated as a special case of sequential operations.

Image processing algorithms may be complicated by the need to cope with a data rate that differs from the processor clock rate. In this circumstance, the pipeline must stall until more data arrives or different parts of the pipeline must be manually scheduled to run on different clock cycles. One solution is to use a multiphase design as in (Johnston et al., 2005a) to deal with situations in which the data rate is an integer multiple of the processor clock rate. This makes it possible to break the pixel processing pipeline into parts.

Visual languages are good at expressing relationships between operations but are often not succinct. For this reason, text is used for mathematical and logical operations and a graphical representation is used for connections between those operations. Textual representations for mathematical operations have been developed and improved for centuries and are now very succinct, and textual representations for mathematical expression on computers are also well developed. Visual representations have been criticised for their representations of expressions (Green and Petre, 1996). A mixed mode representation allows the benefits of both textual and graphical representations to be exploited.

As it is describing a computation, control structures are needed. At the computational level, fine-gain control is available. Different actions often need to be taken depending on what data is being passed through the network. Data dependent branches in execution and the ability to run loops are required. This leads to the need

for the standard *if else*, *for*, *while* and *do while*, structures although these are represented graphically.

High Level Design

The designs of the computational representation has drawn on existing graphical notations for pipelining, parallel and sequential operations. Gantt charts (Schermerhorn, 2001) have been used for project planning, event scheduling and critical path determination for many years. In a Gantt chart, sequential tasks follow each other horizontally in a time line; any tasks which can be performed concurrently are stacked vertically. In this way several concurrent tasks and task sequences can be scheduled so that they can all be completed in the shortest time possible.

In VERTIPH, this idea of representing time in the horizontal direction and independent concurrent processes in the vertical direction is used. This allows parallel and sequential process to be separated as shown in Figure 3.1. The three tasks A, B, C each follow each other in a sequence taking three clock cycles to complete and tasks E, F and G occur in parallel and take one clock cycle.

However, the Gantt chart does not have a representation for pipelining. Data passes through a pipeline sequentially, with processors at different stages of the pipeline operating concurrently as new data arrives. This led to a sequential representation. Figure 3.2 is augmented with bars to indicate when new data arrives into to the pipeline and how long the pipeline delay for the data is. Both this pipelined representation and the sequential view representation flows left to right.

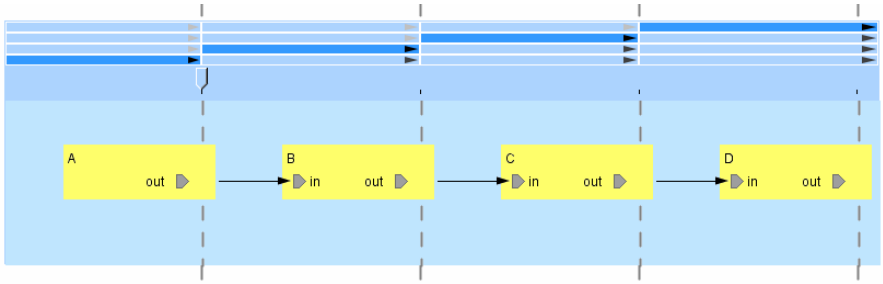


Figure 3.2: Pipeline representation, for a four stage pipeline

For the control structures, a modification of the Nassi-Shneiderman representation (Nassi and Shneiderman, 1973) has been employed. As pipeline and sequential representations use a large amount of screen space in the horizontal direction the standard Nassi-Shneiderman style of control is inefficient (Figure 5.23 chapter 5). Furthermore, the Nassi-Shneiderman control is inconsistent with the VERTIPH convention that data flows from left to right, so that the horizontal dimension of the diagram represents time. If more than one action is possible at any

given time the actions are stacked vertically. (Note that in parallel–processing, more than one action *will* happen at a time, and in a choice, only one of the actions that *may* happen at the time *will* happen at that time). The Nassi-Sheniderman *if-else* control structure has been replaced by an F shaped structure to make it more consistent with this convention. This also allows for a switch-type structure, which can be read down the screen.

3.1.3 Scheduling Graphical Representation

In an embedded image processing system using FPGAs there may be a large number of parallel processors competing for access to resources. The main resources in contention are off-chip devices, such as RAM, video input and output and other I/O based access. On-chip RAM also limits the number of concurrent accesses to a single read or write per port in a clock-cycle.

There are also processors which should only run after certain events have occurred, such as an external trigger or an internal trigger generated by another processor when it has finished its task. VERTIPH facilitates resource sharing by encapsulating resources and the processes that act on them, so that the processes can be scheduled. The resource and scheduling view allows for both global scheduling between processors and for local scheduling within components.

The Problem Justification

The processing used in embedded image processing is often either source-driven or sink-driven. A sink-driven system, such as producing an output image on a VGA screen, requires the output data to be produced with specific timings. For a VGA screen, data will need to be produced during the visible region and the processes can either stall or do book-keeping operations during the horizontal and vertical blanking. Conversely, in a source-driven system, the data timing (and processing) is controlled by the source, for example, processing images from a video camera. Being source-driven or sink-driven encourages the adoption of an event driven control approach.

It is necessary to be able to specify the conditions under which particular architectural blocks or sub-blocks will be running. Consider, for example, constructing a cumulative histogram required for histogram equalisation. A grey scale histogram of an image can be constructed as pixels arrive. Then, during the vertical blanking this can then be converted to the output cumulative histogram, while at the same time resetting the input histogram bins. In this case the cumulative histogram block has three sub-blocks, a histogram construction block, and a

cumulative construction and reset block. Each of these needs to run when different conditions occur.

There is also a requirement for processes which are triggered by the same event to be scheduled with relation to each other. An example would be a series of image processing blocks which should operate when pixel data is available to process. However, these blocks form a high level pipeline. In this case, blocks controlled by the same event need to be scheduled as a pipeline while the priming and flushing of the pipeline is managed by the controller. There are also cases for sequential and parallel scheduling of blocks.

One of the key reasons for scheduling processes at the block level is to reduce resource conflicts. A high level view of the program's scheduling makes it easier to identify resource conflicts. It may be possible to automatically identify when resource accesses will conflict with each other.

Characterisation

After considering several different representations for event driven scheduling (discussed in chapter 6) state machine diagrams (Samek, 2002) were chosen. State diagrams allow for the specification of transitions from one state to another state based on externally and internally generated events.

As processes need to be scheduled within states, a split view allows a selected state to have a list of processes that operate in that state. By incorporating the modified Gantt view from the computational view these processes may be scheduled to run sequentially, in parallel, as a pipeline, or some combination of these.

3.2 Tying it back together

As discussed in the introduction textual languages poorly represent both concurrency and complex scheduling. Data flow visual languages are a natural way to express concurrent systems. It has also been noted (Buck and Lee, 1993, Buck, 1993) that data flow graphs are an effective representation for problems in digital signal processing (DSP), both because they are a natural representation for many DSP researchers and because they expose parallelism in the algorithm with limited constraints on evaluation order. This insight led to the selection of a data-flow based graphical representation for expressing image processing algorithms in hardware.

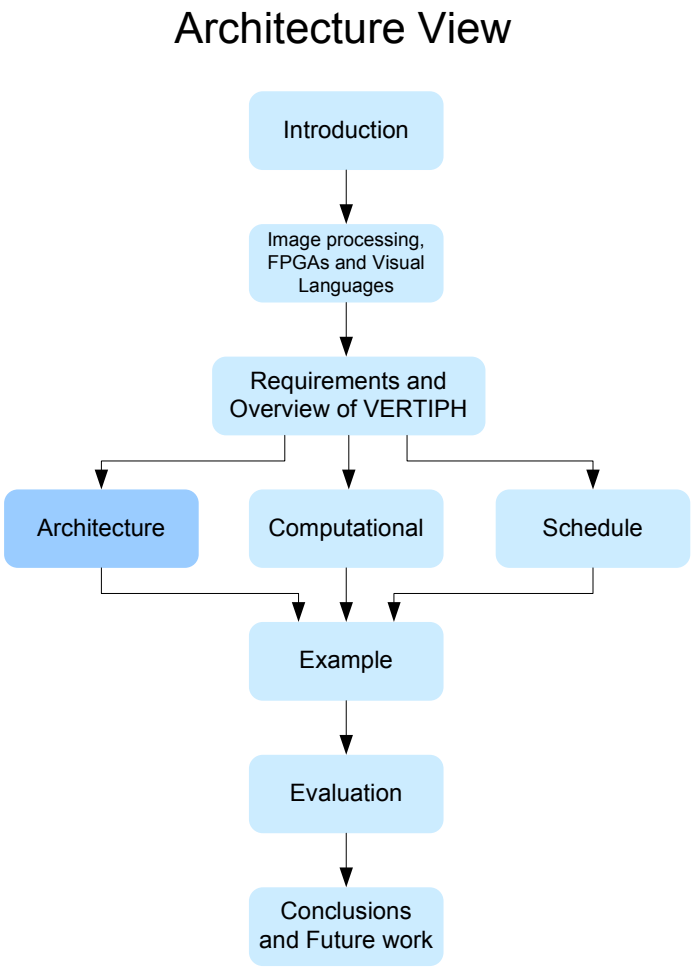
The other main advantage of visual languages is that they are better at expressing networks than textual languages. Text is a serial medium which needs to be read in order; this is not a good representation for networks which can have many

interconnections and sub systems. As a hardware-based system is normally built using a number of interconnected specialised processors which communicate with each other, designs can quickly form quite complex networks.

As diagrams get overly large and complex, a person's ability to understand a system is reduced (Green and Petre, 1996, Green, 2000) in the same way that large text programs with functions longer than a page of text are difficult to understand (Green and Petre, 1996). This highlights the need for hierarchical encapsulation and leads to the need for a block level system overview, the architectural graphical representation.

The need to clearly specify parallel, pipelined and sequential operations and low level data dependent control flow has led to the development of the computational graphical representation.

The requirement for event driven running of processors and high level state based control has lead to the need for a scheduling graphical representation.


Architecture View
Introduction
Image processing, FPGAs and Visual Languages
Requirements and Overview of VERTIPH
Architecture
Computational
Schedule
Example
Evaluation
Conclusions and Future work

4 Architectural view

VERTIPH's architecture view is a graphical editor which provides the top level of a hierarchical decomposition design methodology. Image processing systems often comprise a sequence (or network) of highly specific processing tasks and I/O modules which can be developed independently and validated using test image data. It is therefore appropriate to provide a design environment that allows a designer to start by specifying the major modules and the data flows between them, leaving decisions about low level details until the high-level data flow structure has been determined.

The architecture view thus gives the designer an early perspective on the overall system, and is at the same time a portal for accessing other editors for the low level tasks. Khoros (Konstantinides and Rasure, 1994) and OpShop (Ngan, 1992) are other systems that provide a design environment at a similar level.

There are many ways to achieve the same image processing goal. An example would be an operation for counting the number of similarly coloured blocks in an image. One approach is to segment the image using a threshold function for each colour, resulting in a separate image of each different coloured block. Each image can then be processed independently to count the number of objects it contains. Another approach is to scan through the image once for each threshold level to do the thresholding or to do all the threshold tests concurrently, feeding the data into a data processor for each colour class. Just selecting the threshold value to use requires a selection of the colour space that will be used of which there are many (YUV, HSV, HIS etc.). Once the different coloured objects have been separated it is necessary to uniquely identify them so that they can be counted. In most cases, each object will be made up of more than one pixel. Connected components labelling and analysis can be used for the labelling. There are many approaches and the analysis is often done separately but can be combined producing a stream-based algorithm (Johnston and Bailey, 2008, Ma et al., 2008, Bailey et al., 2008, Bailey and Johnston, 2007). Or bounding boxes could be used as the method of object detection as could region growing and edge based methods. Each of these different methods can have multiple implementations (such as the various two- and one-pass connected component labelling and analysis methods), each requiring a different set of sub-operations and steps. An ideal design will fit within the hardware limitations imposed by the available FPGA device. However, in many cases a trade-off between the algorithm complexity and hardware usage needs to be made. A too-complex algorithm that implements the desired operation at the expense of heavy resource usage on the

FPGA should be replaced by a less resource-intensive alternative. If the operation forms part of a larger system we want to be able to change its internal operation without redesigning the rest of the system. This suggests a system in which modules' internal operations are decoupled from the rest of the system by well-defined interfaces.

VERTIPH's architectural view uses the common visual metaphor of blocks and wires to give the developer an overview of the whole programme. It is hierarchical, with each block able to hold other architectural block views or computational blocks. This aims to address the complexity of the design by presenting the developer with only the required information for the task. It also encourages the developer to use a top-down programming methodology, breaking the design down into logical function blocks.

A hierarchical design not only hides complexity but also encourages component reuse, as smaller sub-blocks of a system can often be reused in other projects. This is not as common when large complex blocks are used, as they tend to have more specialised functions. Processors which are logically related to each function are also encapsulated into one module. For example, a colour segmentation and tracking algorithm detailed in (Johnston et al., 2005b, Johnston et al., 2005a), uses bounding boxes for object detection. This utilises 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-oriented software engineering. Each architecture block can be considered as an object. It has a well defined interface and each computational block that it contains can be considered a method. Encapsulation is specifically designed to reduce the external influence of the internal implementation details.

Encapsulation also groups 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.

The use of blocks allows the graphical notation to be broken up into a number of separate graphical modules thereby avoiding the problem of too many icons on a screen at once, which can lead to information overload. Graphical modularisation also allows the developer to develop a design that is similar to the common mental

mapping of the design. Each operation is its own module, which can be thought of separately from the others, but the modules are connected together at the highest level of abstraction to form a fully functional design. A modular design also means that when changes are made to the block at a later stage, they are localised to that block. Localisation is important as it can prevent changes in one part of a design from affecting another.

To achieve a localised design it is important to consider the communications between the different modules. Like other visual languages, VERTIPH uses a system of external and internal names and types. Data type and names will be discussed later in this chapter. A well defined external interface can aid in keeping changes localised to the module. It is only when the interface needs to change that the other modules are affected.

Recursive encapsulation can result in very complex intellectual property blocks being built, with one block and its interface representing a complex system of data structures, resources, processes and their scheduled operations or response to events.

4.1 The Design of the Architectural View

As discussed earlier, the architecture view comprises a hierarchical block–and-wire graphical representation.

The hierarchy of operations is detailed in Figure 4.1. A project consists of a root architecture view. Within this view, other architecture blocks can be added and connected with wires via connection points called terminals, which have an internal connection point and an internal name and an external connection point and an external name. Each architecture block can then contain either a network of connected architecture blocks (deriving its inputs from the internal connection points of the terminals of the parent architecture block and producing outputs attached to the internal connections points of the terminals of the parent architecture block) or a collection of encapsulated computational blocks. The architectural view allows the designer to edit a network of components. The computational view allows the designer to edit an algorithm; this is discussed in detail in chapter 5. Each computational block contains a computational view. As in the architectural block view, the internal terminal connections are shown in the computational block view to allow connections to other blocks at a higher level using internal names and types.

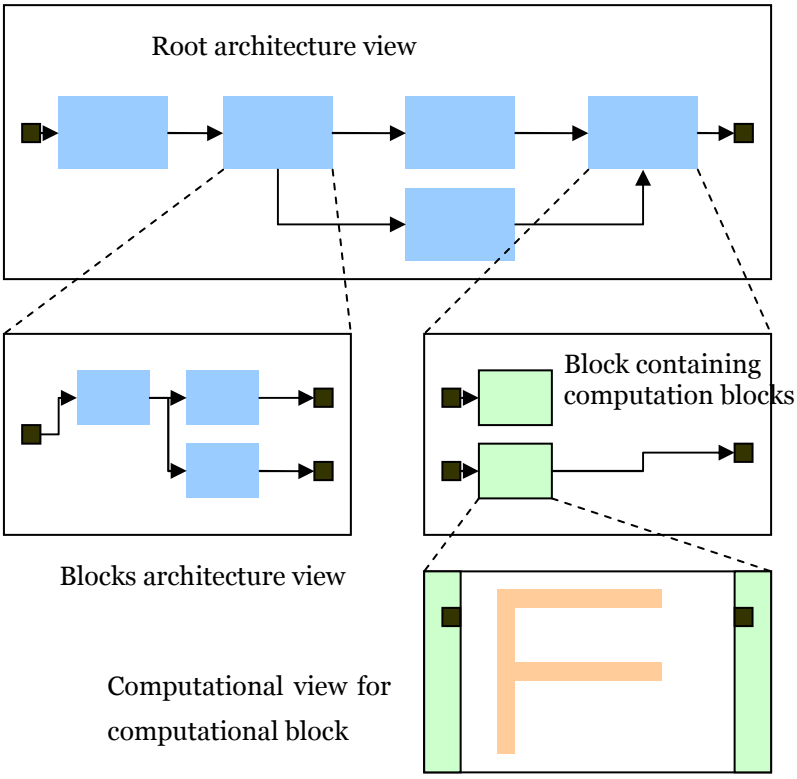


Figure 4.1 : Overview of the architecture view

In any box and wire design there can be a large number of wires showing information transfer paths between the blocks. In this design, there is no visual distinction between control and data connections. All connections have a type and they can transfer data or control information, as control is just another type of data. The aim is to encourage developers to use the scheduling view for control, where possible, so there is no visually distinct control wire.

4.2 Architecture Blocks

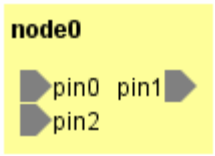


Figure 4.2: Architectural block with 2 input terminals and one output terminal

Each architecture block has a name and can have input and output terminals as shown in Figure 4.2. Blocks can have comments added to them to describe their function and these can be viewed with a mouse-over or when the block is expanded to give its internal view.

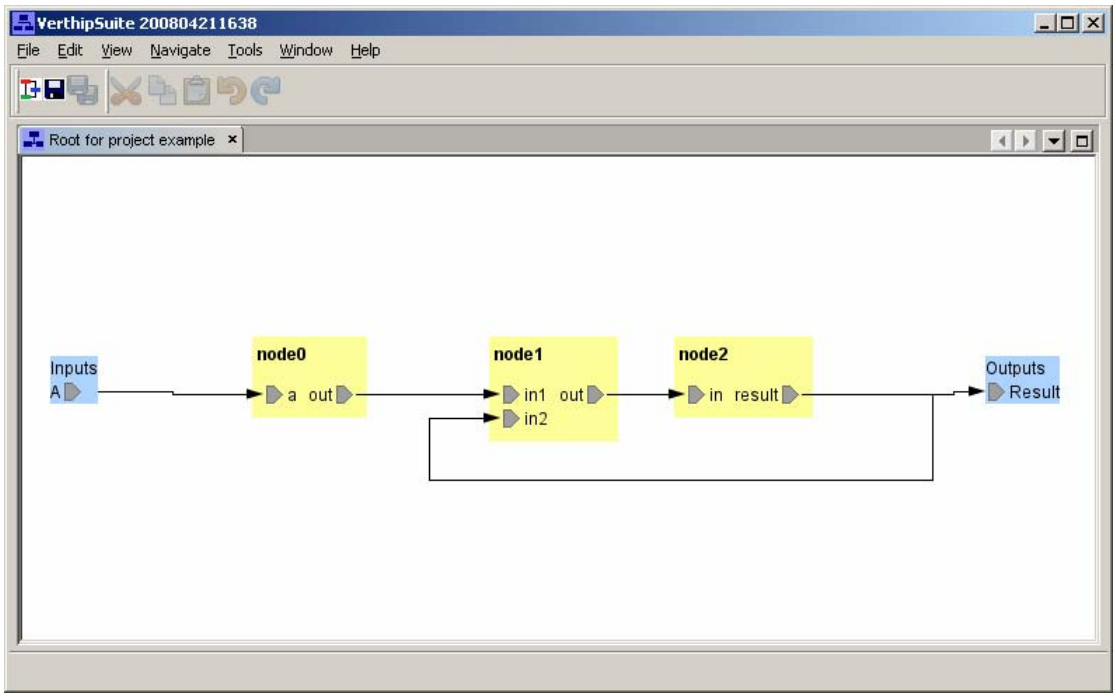


Figure 4.3 : Architecture view with blocks, terminals and wires

4.2.1 Terminals

Terminals can either be sink (input) or source (output). Tri-state connections are not allowed in the current design. Sink terminals can have only one input wire; source terminals can have an infinite fan-out. To reduce the number of connections between blocks, terminals (and the wires attached to them) can have a complex data type, which amalgamates a collection of related data fields into a single type. This reduces the clutter on the screen, as fewer wires must be routed between blocks but it does so at the cost of more complex data types. This leads to the need for type conversions or splitters between an input or output terminal and its internal representation. If only one field in a data structure is required by the block then that field needs to be broken out so that it can be accessed. If the type conversion was performed by separate blocks in the architecture view, this would increase the number of blocks on the screen. Such blocks would not aid in the understanding of the design. Instead the breaking out of encapsulated data is done between the external connection and the internal connections by the terminals through the use of a junction box.

Terminals can be thought of as having an external type and internal type. Externally a sink terminal will have a public type, this is mapped to one or more internal names and data types via a junction box. Junction boxes (Figure 4.4) can be

used to extract an encapsulated data type and give it an internal name. It can also be used to provide data type conversion.

Terminals are strictly typed and will only allow connections from wires of the same type. Wires pick up their type from the source terminal which they are attached to. They may not be connected to a terminal of a different type – except that if a wire is connected to an untyped terminal, that terminal will pick up the wire's type. Strict typing also requires the use of data conversion. The developer can explicitly cast incompatible types, removing the ambiguity that can occur with implicitly typed languages.

4.2.2 Junction Boxes

When strict typing is used, it is often necessary to do some form of type conversion as operators need to be of the same type for an operation to be allowed. Other dataflow languages do this by using special data conversion blocks. These blocks can be used to convert from one basic type to another and also to break out basic or complex types from a complex data type in a wire.

Rather than creating special type converter blocks it was decided to create a mapping between the terminal and the inside of the architecture blocks. This has two benefits. Firstly, it allows any data conversion and breaking out of encapsulated data types at the edge for the block. Secondly, it allows for a mapping between internal and external names. Junction boxes can also be used to combine several data flows into a single data flow to be delivered to an output terminal.

As seen in Figure 4.4, nodes can be connected to other nodes or the individual bits in each basic data type can be mapped individually to the internal data type. In this example an outer terminal of type RGB is mapped to three separate terminals; Red (8-bit), Green (10-bit) and Blue (8-bits). As Red and Blue are both 8-bit the 8-bit subtypes are attached directly. The 10-bit Green is mapped bit by bit using the top bits from G and the bottom bit from the constant '1' and second–least-significant bit from the constant '0'. This junction box mapping can also be used for the mapping from the internal terminals to the outside terminals.

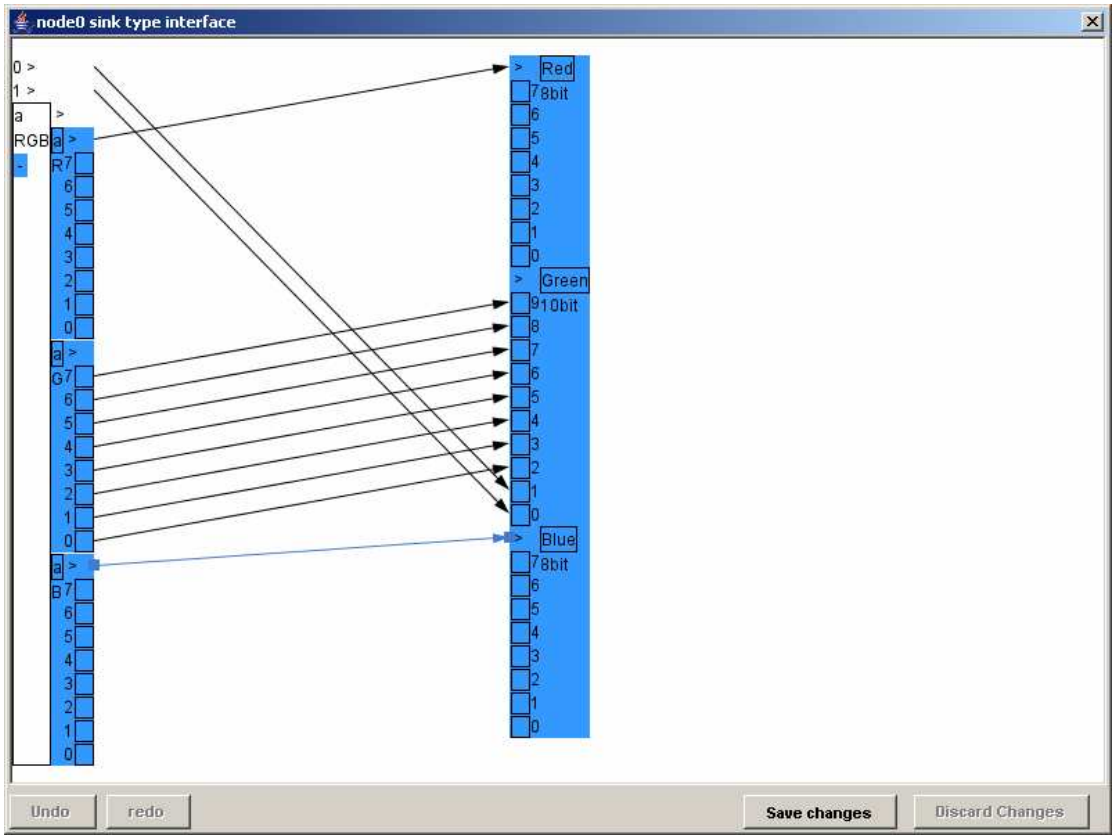


Figure 4.4: Junction box

4.2.3 Data Types

In VERTIPH there is only one native basic type – fixed-point numbers, which can be either signed or unsigned. All other types are derived from this basic type. Fixed-point numbers have been chosen over floating-point numbers for several reasons: 32- or 64-bit IEEE standard 754 floating-point numbers are expensive to implement in terms of memory, circuit size, and power consumption. Image processing operations generally do not require the dynamic range which floating-point offers. Fixed-point numbers offer better overall noise performance when the probability density function of the signals is uniform (Bainbridge-Smith, 2005). 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 consumption, thus making them the best choice for embedded applications (Bainbridge-Smith, 2005).

Fixed-point numbers can also be used to represent integer values as integers are equivalent to a fixed-point number with the fractional part of the number set to 0. One of the advantages of using fixed-point numbers for all types including integers is

that the binary points of operands are automatically aligned and the size and binary point of the result of an operation can also be calculated.

Other types can be derived from the basic type. For example a 16 point grey scale pixel can be derived from an unsigned 16 bit fixed-point number with the binary point at zero. Multiple basic and complex types can be grouped together into more complex types, like *structs* in *C*. A good example for a more complex type is a colour pixel. This can encapsulate three 8 bit colour channels, red, green and blue, into one data type.

The type editor shown in Figure 4.5 illustrates the hierarchical tree structure used to produce more complex types. In this example the "root node" type is constructed of two basic types and one complex child. Each part of this complex diagram will be explained throughout this section. In the far left there is a root name which contains two basic types 'Basic$_1$' and 'Basic$_2$' and a complex type 'complex_node' which contains two basic types. Each basic type has different data widths and types. The total bits required are given by the length field in the complex types.

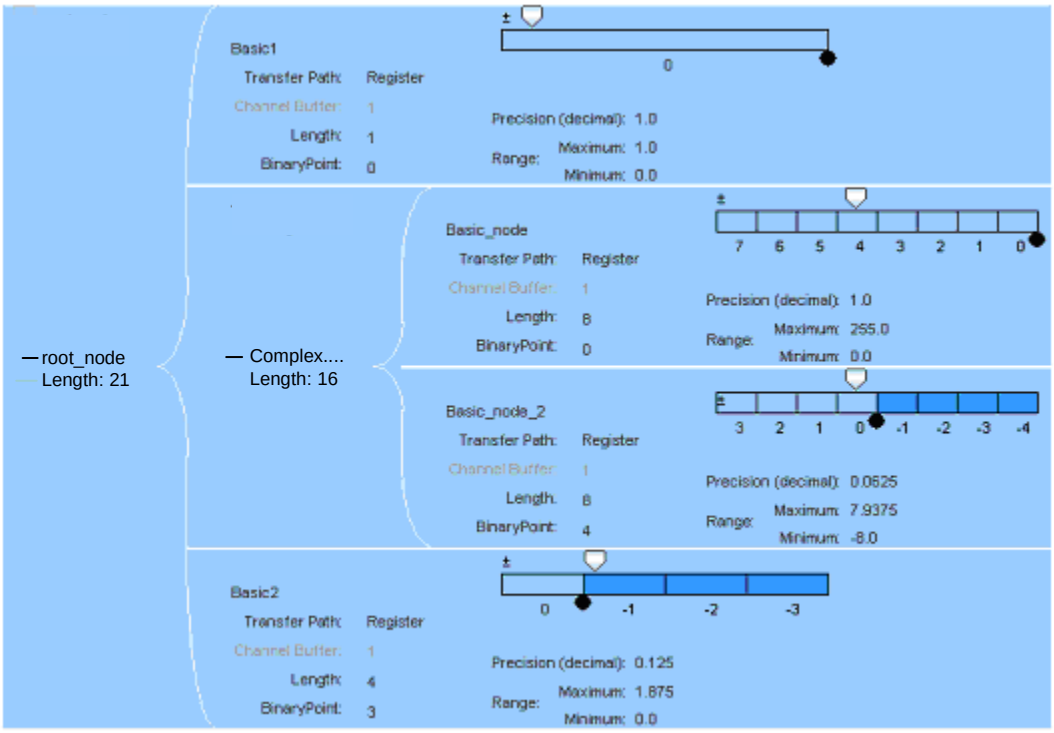


Figure 4.5 : Type editor

Data types can be associated with one of three communication methods: *register*, *wire*, and *channel*. A registered data type will be written to a register (logic on the FPGA) at the end of each clock cycle. A channel, which may be buffered, will

cause the sink operation to stall until there is an output from the source and vice versa. If the channel is buffered, any data placed on it will enter a FIFO queue equal to the length of the buffer, and the sink will process until the buffer is empty. If the channel buffer is full, then writing to the channel will block and the source will have to block.

4.2.3.1 Basic type editor panel

The basic type editor panel has several key features. In the top right corner it has a graphical representation of the number showing the size, sign and binary point position for the number. The size can be controlled via the upper slider and the location of the binary point is also under slider control as illustrated in Figure 4.6.

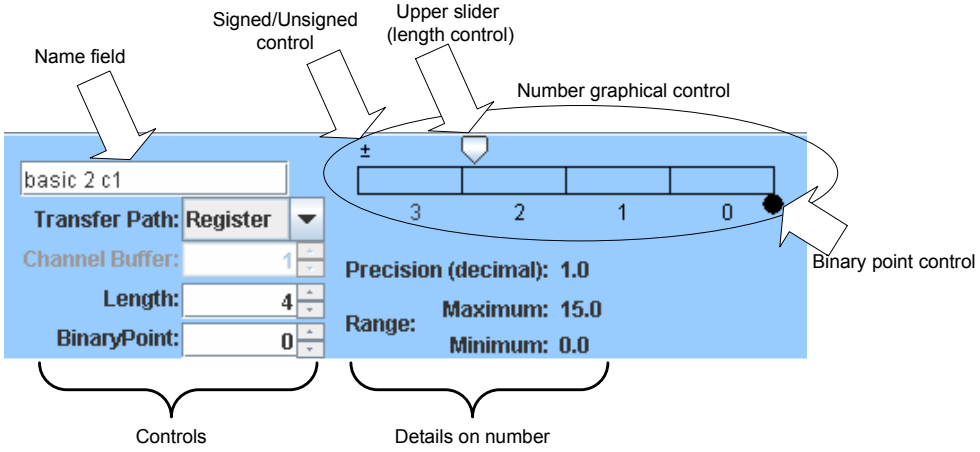


Figure 4.6: Parts of Basic type panel

As shown in Figure 4.6 and Figure 4.7 the panel has a number of editable controls. The name field can be set via a text box. The transfer path can be set via a drop down combo box to *Register*, *Wire* or *Channel*. If *Channel* is selected, the channel buffer spinner box can be edited; otherwise it is disabled as shown. The Length parameter can be set via either the spinner box or by dragging the upper control of the number's graphical representation. The binary point can also be set via either the slider or spinner control box. When the cursor is not located over the parameters, they appear as text labels as shown in Figure 4.8. This was done to reduce the amount of non-relevant information presented to the developer, who only sees the edit controls when using them.

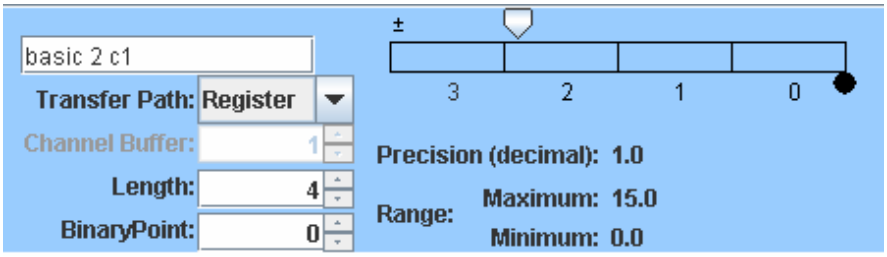


Figure 4.7 : Type editor panel showing editable controls

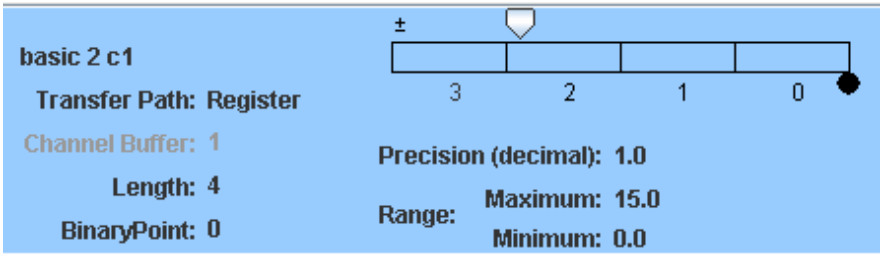


Figure 4.8 : Type editor panel normal view

The next four figures illustrate the information presented to the developer about the type that is being created, with several of the different options selected. There are several outputs given to the developer: the precision, which gives the smallest decimal step available, and the range of numbers that can be represented. A 2's complement representation is used for signed numbers. All of these examples are based on a 7 bit registered number. A 7 bit integer is shown in Figure 4.9; the implied binary point is at zero, the precision is one and the range spans from zero to 127 or 2^7-1. It can also be seen that each bit location is labelled with its significance in base-2. This can aid developers who need to do bit manipulation work. Bit manipulation can be error prone and a simple number reference for each bit can help.

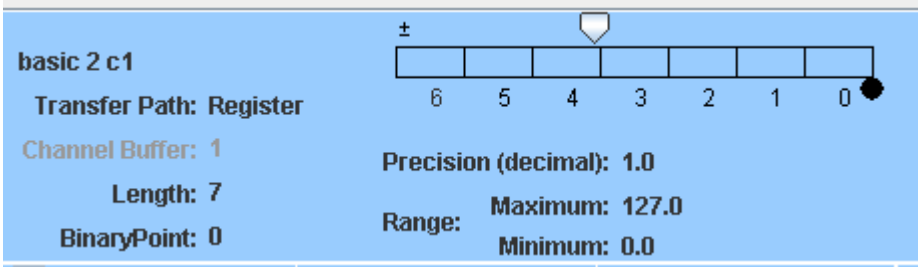


Figure 4.9 : 7 bit integer

In Figure 4.10 the number is signed. This is indicated by the position of the ± inside the box. The user toggles the location of this mnemonic symbol by clicking on it. As the sign takes up one bit, the range of possible values becomes -64 to 63, as for 2's complement numbers the range is -2^{N-1} to $+2^{N-1}$-1.

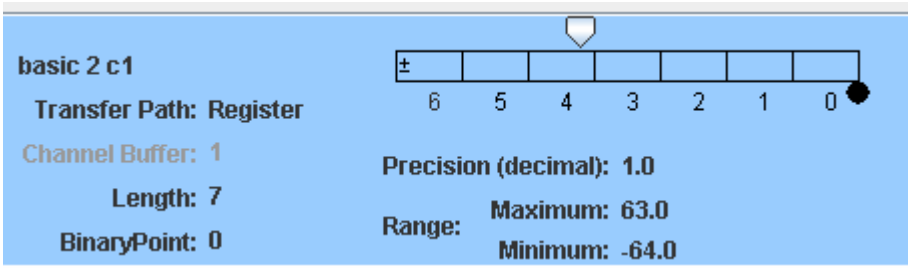


Figure 4.10 : Signed 7 bit integer

When the binary point is moved away from zero, the bits after the binary point are coloured to indicate that they form part of a fraction. Here, the main benefit of showing the precision becomes apparent; the increment is 2^{-4} (0.0625). As the number is unsigned, the maximum number is 7.9375 and the minimum is 0.

In Figure 4.12 the number is changed to signed which affects the range, now from -4 to 3.9375.

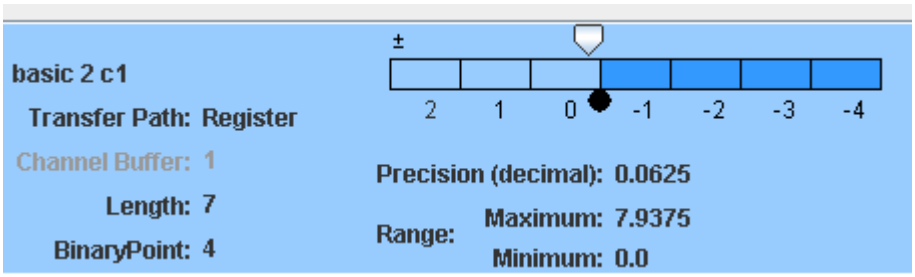


Figure 4.11 : Unsigned 7 bit fixed point with binary point at 4

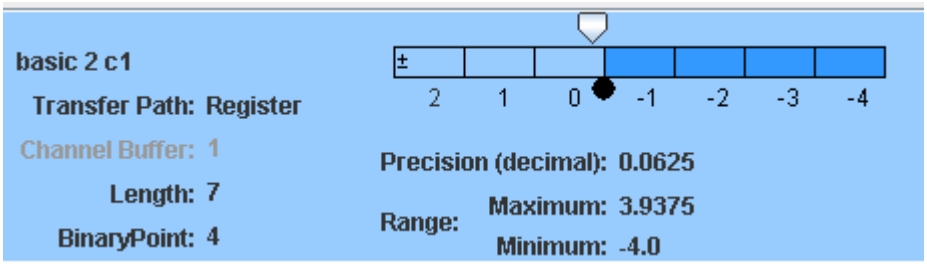


Figure 4.12 : Signed 7 bit fixed point with binary point at 4

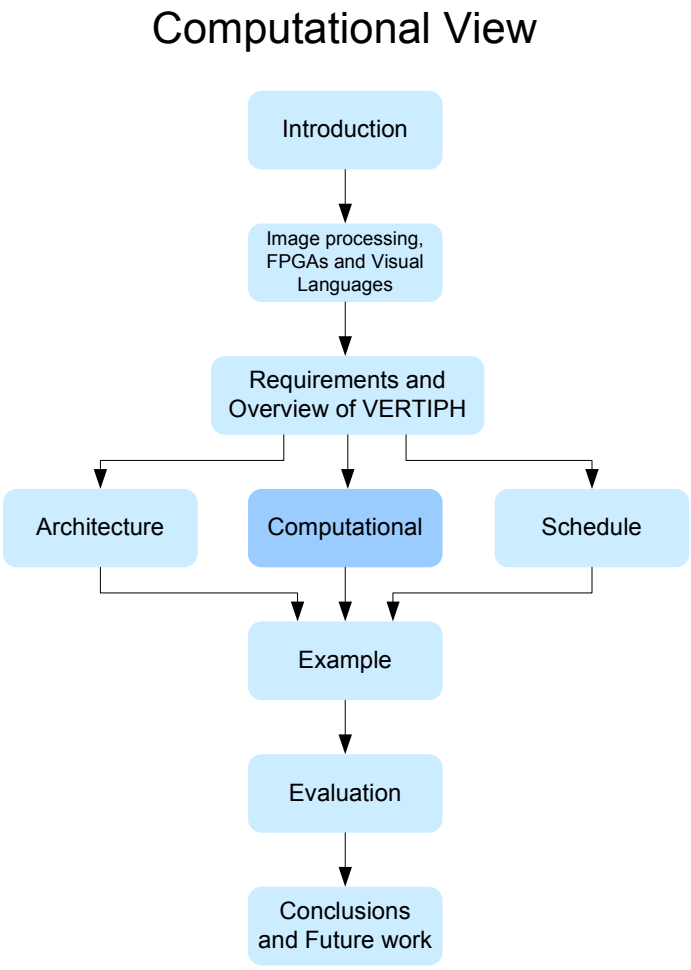
Having several different forms of feedback is useful to the developer when editing the data type. For example, having the range and precision immediately available decreases the burden on the developer because they do not have to calculate it manually.

4.3 Summary

The architectural view is designed to allow the representation of image processing algorithms as block diagrams which are a common and popular notation

used in image processing. This view is primarily designed to illustrate the dataflow between different parts of an algorithm.

As image processing algorithms can be very complex, this view also allows multiple levels of decomposition, producing a hierarchical view. This is designed to encourage top down development. It also gives the developer the ability to group related processors together.


Computational View
Introduction
Image processing, FPGAs and Visual Languages
Requirements and Overview of VERTIPH
Architecture
Computational
Schedule
Example
Evaluation
Conclusions and Future work

5 Computational View

The architectural view allows a designer to break a design into manageable blocks. The algorithm for each block is then specified in the computational view. The computational view has been designed to make it simple to specify the arithmetic and logical operations required to produce functional hardware.

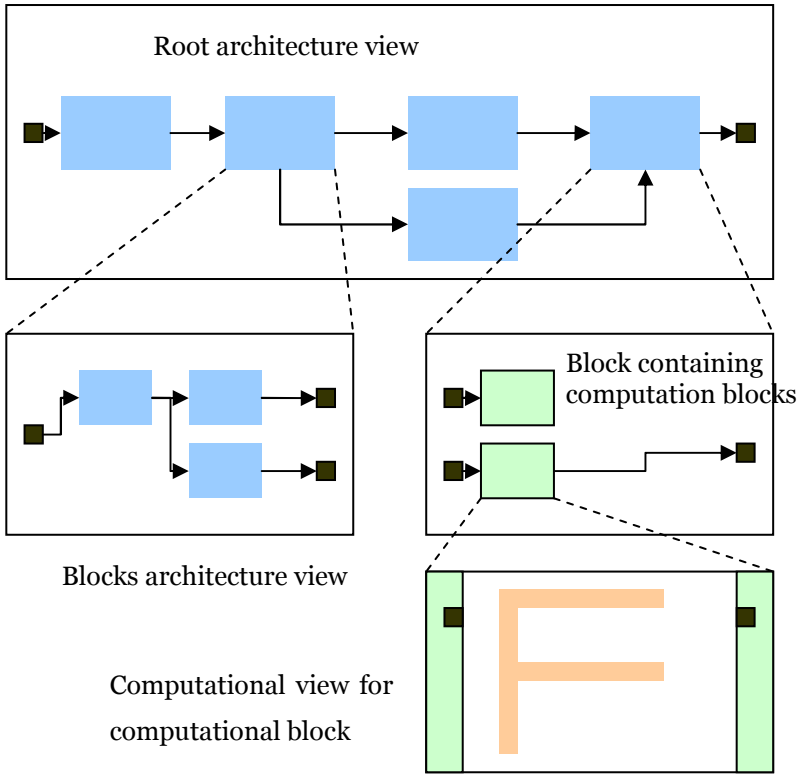


Figure 5.1: Hierarchy of the Architectural and Computational views

To encourage functional decomposition, the computational view, like the architectural view, is hierarchical. This is illustrated in Figure 5.1 where computational processing blocks can contain computational expressions. The use of modularization allows common operations or operation sequences to be designed for one project and added to a library for use in later projects. A hierarchical representation may also lead to a reduction of the number of instructions on the screen, as the developer is only presented with the top most representation and only has to dig into the detail when needed.

The hardware units at the lowest level of the design (arithmetic and logic units, look-up tables, caches and other memory units) are not organised as a sequential data flow. Instead, they are organised as a network whose components

interchange data and which need to be scheduled with relation to each other. Visual languages are good at showing complex networked structures such as these.

Visual languages also have the ability to show timing relationships. This is important as one of the main issues when working at the lower level of design is keeping track of the timing relationship between different hardware units. Some will operate concurrently and independently; others will operate sequentially and others will form pipelines. As the stream of data arrives, it is operated on and passed between different processing blocks. It is important to schedule these data transfers to maintain synchronisation and to prevent data loss.

Unlike architectural blocks, computational processing blocks need to have timing information. For example, a parallel operation block will take as long as its slowest component operation – generally one clock cycle. The duration of a sequential computational block will be the total time required to complete all the operations. It also cannot accept new data until it has finished all the operations. A block containing a pipeline will also have a timing length – a latency – equal to the sum of the durations of the individual operations, but it will accept new data during its operation.

This timing information is needed when a hardware function is placed into other expressions because some combinations of operations have incompatible timing. A sequential function cannot, in general, be added into a parallel design without altering its duration. In general, multi-clock cycle operations are not suitable for parallel or pipeline operations. Multi-clock cycle operations which operate sequentially cannot be implemented in pipelines without having multiple copies of the hardware that can be multiplexed to allow for the required throughput. If the block is a pipeline then it can be incorporated directly into a pipeline block. The multiphase section of this chapter goes into this in more detail.

While many image processing algorithms are inherently parallel, they are commonly implemented on a serial processor and are most often thought of and expressed serially. For example, a filter is inherently parallel because it applies the same operation independently to each input pixel. Many algorithms can be converted to a hardware design by pipelining a serial algorithm. There may also be limited access to memory, requiring the use of pipelining due to the serial nature that the data is presented. Furthermore, most image processing applications involve several steps which can run concurrently as pipelined processors. It is therefore desirable to have a development tool which allows pipelining to be captured at an appropriate level of abstraction.

5.1 Operations (expression blocks)

Textual languages are by nature single-dimensioned. Visual languages are two-dimensional and this gives them the ability to highlight the timing relationships in the computational view. This section will look at some of the deficiencies of using text for programming FPGAs, looking at some of its limitations, the ways that these can be reduced using secondary notations, and begins to look at our visual notation for the computational view.

In text, words are written together conventionally as runs (sentences and groups of sentences). Below the level of paragraphs, text does not represent structure visually. To aid in the understanding of programs it helps to add structure to the text. Structured programming languages have added ways to group text, the most obvious being the *begin* and *end* key words (Pascal) or {} brackets (C-based languages) to contain procedures or conditional statements. These parsing keys make the program more understandable by breaking up the design into blocks of code. However, to read and understand a program more structure is needed. This leads to conventions such as putting each statement on a separate line and indenting statements. This can be especially useful in nested conditional statements, although it can result in increased maintenance when any changes are made. Other conventions include the colouring of special text such as comments, variables and keywords. Some developers go to some lengths to structure their code, such as creating templates to box and enclose procedures and class declarations, like the example code below where a Delphi procedure declaration is highlighted by boxed it in comments to highlight its start and end. These informal conventions have been labelled as secondary notations by Green and Petre (Green and Petre, 1996).

```
{'==========='==========="==========='==========='}
{}              PROCEDURE procName;           {}
{'==========='==========="==========='==========='}
begin
  showmessage('procName' has been called but has not been
implemented');
end {of PROCEDURE procname};
```

Even though textual notations make it possible to highlight structure with slightly unwieldy boxes, they do not highlight differences between data manipulation code and control code. Text based languages treat them the same, as characters to be manipulated. Most IDEs will highlight keywords (either with colour or by font) but they do not highlight what the control structure controls other than with the *begin*

and *end* block terminators. This can make it harder to implement an intelligent cut and paste operation that automatically selects complete structures such as a *while* and its initialisation statement. Conventional IDEs cut only the selected text, not the loop operation as a whole. In a visual language the control, initialisation and statements can be more clearly identified, and instead of being syntactic sugar, the boxes used to group things together can be implemented as selectable objects. The notation has been constructed so that it makes the sequences of data manipulation, control flow and declarations all distinguishable from each other.

These constructs can be dispersed on the screen in two dimensions; whereas the linear structure of text hides more complex relationships between functional blocks and has led to the use of variables to link value-creation and value usage, a 2D graphical layout allows the user to draw lines that indicate relationships and data transfers between the functional blocks explicitly.

Figure 5.2 shows how sequential operations (a) and parallel operations (b) are represented. Sequential operations are placed one after another using the horizontal axis to represent time in clock cycles. Parallel operations are placed under each other similar to the manner that parallel tasks are represented in Gantt charts

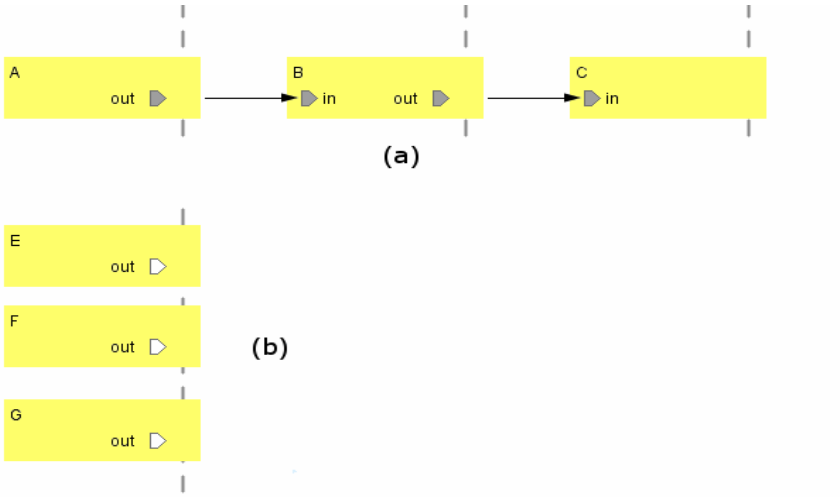


Figure 5.2: Process representations: (a) Sequential, (b) Parallel

5.2 Pipelines

Pipeline concepts are simple to describe and understand but the implementation of all but trivial pipelines can be difficult and error prone. This is generally due to errors in placing individual computations in the correct pipeline stage, not registering the results of processors correctly between stages, or the incorrect design of a pipeline. Particular attention must be paid to data synchronisation and scheduling issues when designing pipelines.

Developers used to instruction pipelines in conventional processors can also have problems visualising hardware pipelines. In a conventional processor, an instruction is fetched, then decoded at the same time as data is accessed, executed and, when necessary, the result is written back to memory. In modern processors, especially embedded processors, this fetch-decode-execute-write-back architecture may have 20 or more pipeline stages to achieve the desired operation frequency (Stokes, 2007) . However, as the length of the pipeline increases, the cost of pipeline flushing when an incorrect branch prediction occurs increases also, so pipelines are often limited to 7 to 15 stages.

Hardware pipelines do not have this problem, as the "instructions" are implemented as fixed hardware units (and therefore do not have the setup overhead of a conventional processor), with data being selectively multiplexed between different paths. This allows for the construction of very long pipelines. This is desirable as one main reason for the use of pipelining in an FPGA design is so that the designs can operate at the desired clock frequency and splitting operations into simpler tasks can decrease propagation delay (and therefore increase frequency) but produce longer pipelines. Nevertheless there are still problems. For example, the priming and flushing of pipelines needs to be scheduled and there are issues with scheduling tasks and the stalling of pipelines to wait for data or resources.

To ground this discussion in a particular situation, consider some of the different timing control options for a geometric calculation - determining the distance between two points given their separation in the X and Y dimensions. Handel-C would allow the whole calculation, $Z = SQRT(X^2 + Y^2)$, to be performed in a single clock cycle. However, this arrangement would have a large propagation delay, because it would require the data to pass through a large number of logic elements. Consequently, the FPGA would run at a slow clock rate. Breaking the calculation up into stages would reduce the propagation delay for an individual stage, and increase

the clock frequency, but increase the latency, because the clock rate for all the stages would be limited by the propagation delay of the slowest stage.

It is easy to see that the problem can naturally be broken down into four separate calculations, X^2, Y^2 , Z^2 and Z.

```
Given X and Y
X^2 = X * X
Y^2 = Y * Y
Z^2 = X^2 + Y^2
Z = SQRT(Z^2)
```

Assuming that each operation takes only one clock cycle, and X and Y are provided at the same time, there are several different configurations available for these four instructions. Each option may take a different number of clock cycles to compute the final result. The first option is to use a purely sequential (non-pipelined) design as shown in Figure 5.3. The boxes represent the operations and registers used to store the results. In this, X^2 is calculated in the first clock cycle, Y^2 is calculated in the second, Z^2 in the third using the results from the first two clock cycles, and Z in the fourth clock cycle. Therefore, for each X and Y value it takes four clock cycles to calculate Z. A new X and Y can only begin calculation once every 4 clock cycles. Assuming X = 3 and Y = 4, the corresponding calculation is shown in Table 2, where the clock cycle, the active operations, and their results are shown.

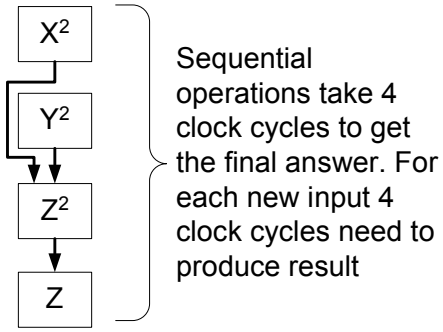


Figure 5.3: Sequential flow of operations

Table 2: Effects of timing on operations sequential operations

Clock cycle	Active operation	Result
1	$X^2 = X \times X$	9
2	$Y^2 = Y \times Y$	16
3	$Z^2 = X^2 + Y^2$	25
4	$Z = SQRT(Z^2)$	5

The next option to consider is a parallel implementation. As there are data dependencies between Z^2 and X^2, between Z^2 and Y^2 and between Z and Z^2, not all of

these operations cannot occur in parallel. However, X^2 and Y^2 can be calculated in parallel. The data dependency for this is shown in Figure 5.4. In this case three clock cycles are needed to calculate the final result (X^2 and Y^2 are calculated in the first clock cycle, Z^2 in the second and Z in the third clock cycle as shown in Table 3), and new data can be introduced once every three clock cycles.

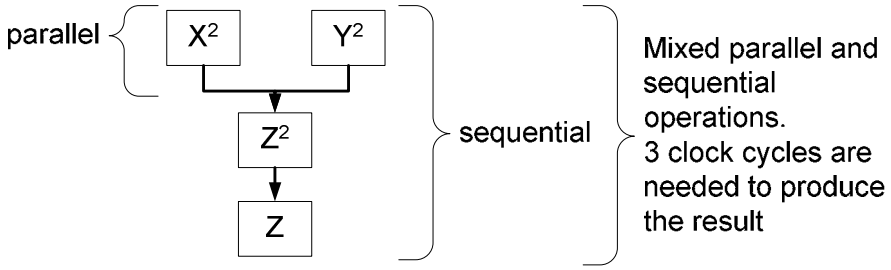


Figure 5.4: Mixed parallel and sequential operations

Table 3: Effects of timing on operations parallel operations

Clock cycle	Active operation	Result
1	$X^2 = X \times X$	9
	$Y^2 = Y \times Y$	16
2	$Z^2 = X^2 + Y^2$	25
3	$Z = SQRT(Z^2)$	5

There are two choices for implementing a pipelined design: one using a pipeline based on the sequential design and one based on the parallel design. The data dependency for a pipeline based on a sequential design is shown in Figure 5.5. It looks just like the sequential architecture; however, extra registers have been added. The use of registers between successive operations separates the processing of one instruction from the processing of the next. This allows new data to enter the pipeline with each clock cycle. It will take four clock cycles before valid data will exit. This is referred to as priming the pipeline. Once the pipeline is primed a new valid result will be produced every clock cycle as long as new data is fed into the input. This means that the latency of the design, the time taken from when data enters until it exits, is four clock cycles. The pipeline also needs to be flushed when no more new data is entering the pipeline, to ensure that all the valid results have left. This can be done by adding dummy data to the pipeline until the valid results have left the pipeline. As X and Y are made available in parallel and X^2 is calculated before Y^2, Y either needs to be placed in a temporary register or passed to the pipeline one clock cycle later than X. Also, as both Y^2 and X^2 are needed at the same time to calculate Z^2 the result from the X^2 calculation needs to be registered twice (as X^2 and X^{2reg}), otherwise the correct

answer will be overwritten by the calculation for the next input before it is used and an incorrect result produced. Table 4 shows the state of the pipeline at four successive clock cycles, assuming that during clock cycle 0, Y^{reg}, X^2, X^{2reg}, Y^2, Z^2 and Z receive the values 1, 0, 1, 1, 3 and 3 and that new X and Y values enter the pipeline on each clock cycle, with Y being buffered. Note that the active operations related to the (X, Y) data pair introduced into the pipeline on clock cycle 1 are shown in bold.

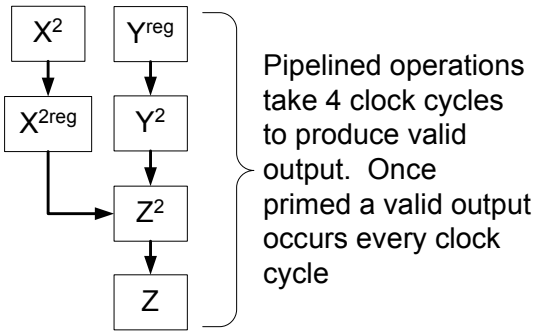


Figure 5.5: Sequential based pipelined operations

Table 4: Effects of timing on operations for sequential based pipeline implementation

Clock cycle	X and Y	Active operations	Results
1	X = 3, Y =4	**$X^2 = X \times X$**	**9**
		$Y^{reg} = Y$	**4**
		$X^{2reg} = X^2$	0
		$Y^2 = Y^{reg} \times Y^{reg}$	1
		$Z^2 = X^{2reg} + Y^2$	2
		$Z = SQRT(Z^2)$	1.73
2	X = 5, Y = 12.	$X^2 = X \times X$	25
		$Y^{reg} = Y$	12
		$X^{2reg} = X^2$	**9**
		$Y^2 = Y^{reg} \times Y^{reg}$	**16**
		$Z^2 = X^{2reg} + Y^2$	3
		$Z = SQRT(Z^2)$	1.41
3	X =6, Y = 8.	$X^2 = X \times X$	36
		$Y^{reg} = Y$	8
		$X^{2reg} = X^2$	25
		$Y^2 = Y^{reg} \times Y^{reg}$	144
		$Z^2 = X^{2reg} + Y^2$	**25**
		$Z = SQRT(Z^2)$	1.73
4	X =6, Y = 8.	$X^2 = X \times X$	16
		$Y^{reg} = Y$	3
		$X^{2reg} = X^2$	36
		$Y^2 = Y^{reg} \times Y^{reg}$	64
		$Z^2 = X^{2reg} + Y^2$	169
		$Z = SQRT(Z^2)$	**5**

It can be seen from Table 4 that the flow for the "sequential" pipeline is not purely sequential, as the result of an operation needs to be registered twice (as X^2 and X^{2reg}), so that the X^2 and Y^2 results are both accessible for the Z^2 calculation. This pipeline design is not as efficient due to these extra registers. The design also introduces the possibility of errors due to not registering the output and the complicated way the (X, Y) pairs are entered into the pipeline. A better approach is to have parallel X^2 and Y^2 calculations. This allows both X and Y to be operated on at the same time and the correct results available simultaneously for the Z^2 calculation. This results in the data flow illustrated in Figure 5.6, giving a three stage pipeline; that is, one that has a latency of three clock cycles. The outputs of each stage for this are shown in Table 5, again the operations for the data values of interest shown in bold. The initial value of the (X^2, Y^2, Z^2) triplet is (1, 3, 3).

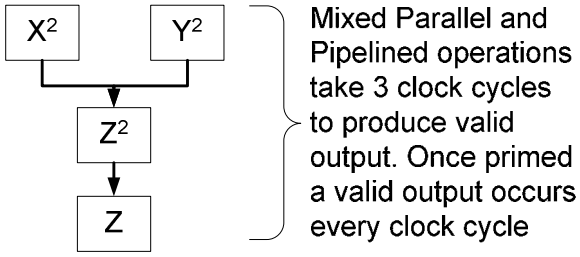


Figure 5.6: Parallel pipeline operations

Table 5: Effects of timing on operations for parallel

Clock cycle	X and Y	Active operations	Results
1	**X = 3, Y =4**	**$X^2 = X \times X$**	**9**
		$Y^2 = Y \times Y$	**16**
		$Z^2 = X^2 + Y^2$	2
		$Z = SQRT(Z^2)$	1.73
2	**X = 5, Y = 12.**	$X^2 = X \times X$	25
		$Y^2 = Y \times Y$	144
		$Z^2 = X^2 + Y^2$	**25**
		$Z = SQRT(Z^2)$	1.41
3	**X = 6, Y = 8.**	$X^2 = X \times X$	36
		$Y^2 = Y \times Y$	64
		$Z^2 = X^2 + Y^2$	169
		$Z = SQRT(Z^2)$	**5**

As can been seen from this example, it is important to have a distinction between sequential and parallel operations. Different HDLs do this in different ways. Some use text block functions (such as *par{}* and *seq{}* in Handel-C (Alston and Madahar, 2002)) and some use different assignment statements (such as := and <=

in VHDL (Bhasker, 1999)). An illustration of some of the difficulty with interpreting such languages occurs later in this chapter. The goal of the notation is to make parallel and sequential operations distinct so that it is immediately obvious to the developer which is which, to avoid possible errors and confusion.

In text notations, pipelined operations are treated as a special case of parallel operations and are not differentiated from them. This notation highlights an aspect of the pipeline that involves few design problems – the fact that all of the operations within a pipeline are executed simultaneously – but ignores an aspect that requires considerable mental gymnastics – keeping track of data dependencies. For a simple design with a single operation per pipeline stage, the timing and actions can be easily managed. As more operations are added per stage and parallel and pipelined operations are mixed, the problem of keeping track of all the different execution times becomes quite challenging. Designers commonly schedule operations at the wrong time or incorrectly place delays that are intended to synchronise different parts of the pipeline.

This suggests that it is more appropriate to treat pipelines as a special case of sequential operations, where the data dependencies are explicit, than as a special case of parallel operations. Pipelines have certain similarities with, and share certain problems with production lines. An automotive production line could be used to assemble a single car at a time, but works far more efficiently if all its assembly stations are active simultaneously. This simultaneity of operations does not apply to individual vehicles – an individual car's body panels are not being formed, welded in place, and painted at the same time. Instead, each car is assembled in a sequence of discrete operations. In order to maximize the throughput of the production line, its designers have to characterize the individual operations carefully, so that each can be completed in some maximum time. The slowest operation determines the throughput of the whole production line. If, as is common, a variety of options can be specified for an individual vehicle, then the steps required and therefore the time that a vehicle spends at a particular station on the assembly line may vary, and the design becomes correspondingly more complex.

In designing such a system, it is the sequence of operations that matters, not their simultaneity. The same is true of data being processed in a hardware pipeline. As far as the data is concerned, the pipeline is sequential, transforming the data from input to output through a sequence of operations. Each processing element in the pipeline, like its production line counterpart, operates sequentially on each input data element before passing the results on to the output. It is only the correct

synchronisation between successive (serial) stages that allows each stage in a pipeline to run in parallel with all the others.

Thus, from the developer's perspective it is less important to visualize the parallel nature of the operations than to visualize how the data progresses through the design. A pipeline will only ever operate correctly in parallel if the data supply is correctly synchronized with data consumption. If the parallelism of the operations is the chief feature of the notation, the synchronization between stages is implicit (for example through shared variables or registers), requiring secondary notations (comments) to explain the timing. For this reason, I consider that it is more important to show the algorithm as a sequential process, but highlighting the fact that the operations run in parallel to allow different data to be active in different stages of the design.

Algorithms' pipelines can become more complex when there is more than one datapath, such as when pipeline stages contain their own internal parallel operations. In HDLs that make no distinction between parallel and pipeline architectures, confusion can occur between what forms part of a sequential pipeline and what is a parallel stage in that pipeline. When some of the processing elements need data from different stages, then the possibility of introducing synchronization errors as a result of data dependency increases. The following Handel-C code is an example of this.

```
par{
     a = a +1; //stage 1
     b = a << 1;    //stage 2
     c = b - 3;     //stage 3 two parallel
     d = b + 3;     //    operations
     e = d << 1;    //stage 4 two parallel
     f = d + 2;     //    operations
     g = e * f + c  //stage 5
}
```

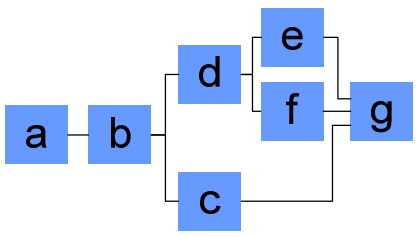


Figure 5.7: Graphical representation of the pipeline

The code example shows an algorithm with seven operations organised into five stages. Stages 3 and 4 operate in parallel and the final stage receives data from the two stages before it. The Handel-C code in this example can take some time to

understand fully. For example, this code actually contains an error: the output from block *c* must be temporarily stored in a register so that its entry to operation *g* is synchronised with the data from operations *e* and *f*. Comments can help the reader (or the designer) to understand the pipeline, but even if a comment explaining the need for a register were added to this textual code, it would still be difficult to understand. On the other hand, when a simple dataflow diagram of the stages is constructed, as shown in Figure 5.7, it becomes easy to see that the output from block *c* needs to be registered because it is required by block *g* two clock cycles later, and not in the next clock cycle. Though the configuration without the register may be what the developer intended, it would not generally be considered a proper pipeline as it uses results derived from different data inputs. In all but a few cases this configuration can be considered an error. A dataflow diagram allows this to be identified quickly and makes the required fix of adding a register obvious.

Furthermore, the sequence of events implied by the textual representation is lost with designs where the processing clock rate is different from the input data rate (usually an integer multiple). Such multiphase pipeline designs will be discussed in more detail in section 5.2.1. Multiphase pipelines can also allow the sharing of resources. It is possible to improve the sequential pipeline presented earlier to avoid the need for extra registers. It is possible to take a two phase approach, as illustrated in Figure 5.8 where X^2 and Y^2 are calculated over two phases of the data clock cycle (each processing clock cycle is a separate phase of the data clock) and Z and Z^2 are calculated over the two phases in the next clock cycle. An advantage of doing this is that the hardware used to calculate X^2 and Y^2 can be shared.

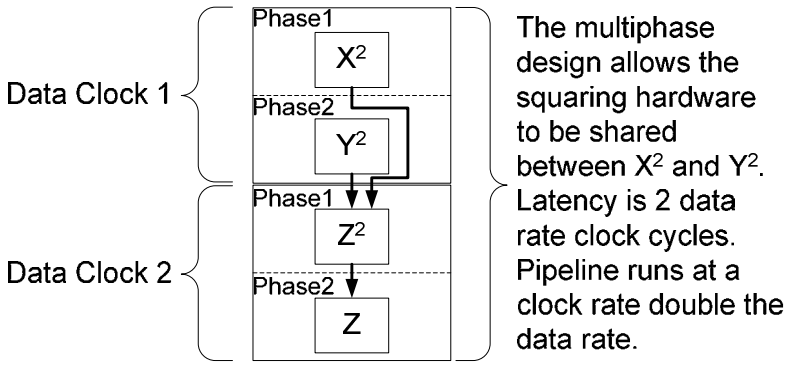


Figure 5.8: Two phase shared hardware example

Shared hardware can be used in a design to minimise the amount of hardware that is required for the design. This occurs when a computationally expensive hardware function, such a multiplier, is shared through the use of a multiplexor on the input and the ANDing of the control with the clock enable of the corresponding output

register, as is Figure 5.9. There is a trade off with hardware sharing. If the shared function is too simple relative to the multiplexors required to switch the hardware, there will be no overall hardware saving.

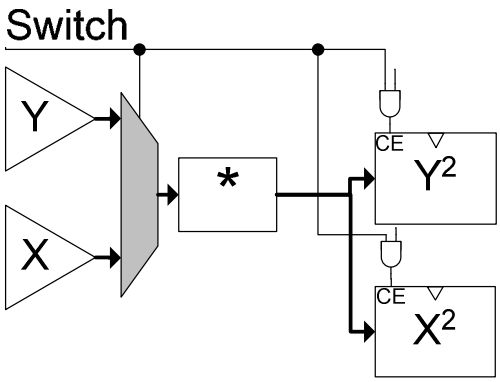


Figure 5.9: How can be hardware shared

5.2.1 Multiphase Pipeline Representation

Conventional HDLs like VHDL (Accellera, 2008), JHDL (Bellows and Hutchings, 1998), Verilog (IEEE, 2008), Handel-C (Alston and Madahar, 2002) have constructs for parallel and sequential operations but not specifically for pipelining, which they treat as a special case of parallel operation. For a multi-phase pipeline, this approach is flawed. This can be illustrated with an example of an algorithm (here written in Handel-C) that is considered first as a single phase and then as a multiphase pipelined design. This example is part of an object tracking system described in (Johnston et al., 2005a). Consider the system architecture, shown in Figure 5.10, which has a video interface, FPGA, RAM and an output for results. It is unlikely that the video decoder, RAM and output will all operate at the same clock rate, so in this system it would be appropriate to have the FPGA operating with two or more clock domains. An RC100 (Celoxica, 2004) board has this configuration.

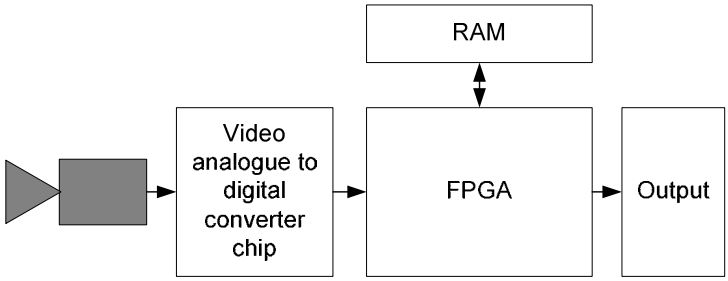


Figure 5.10: FPGA system architecture

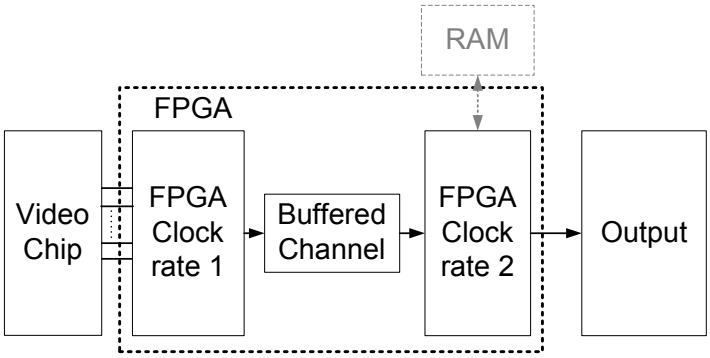


Figure 5.11: Detailed FPGA view with different clock domains

The dashed lines in the more detailed architecture (Figure 5.11) enclose the two clock domains in the design - one with a clock rate synchronised to the camera pixel clock (via the video chip), the other with a clock rate synchronised to the display pixel clock (which also drives the off-chip RAM). Communication between these two clock domains uses a buffered channel. There are two ways to implement a tracking algorithm using this architecture, either by operating on the image in the output clock domain or by operating on the image in the input (camera) clock domain. The first requires the image to be buffered in RAM. This buffering is done using the channel to send the pixel data to the other clock domain then store this in the RAM (the channel needs to be buffered as the clock rates don't match, so some small storage is needed to hold pixels until they are moved from one domain to the other), and then process the data as it is read out of the RAM to be displayed. Using this approach, the data rate is equal to the clock rate used by the RAM (Clock rate 2). This will result is a pipeline which has only one stage per clock cycle. A five stage algorithm for this algorithm is shown below. Expressions have been replaced by function calls to make the logic easier to follow:

```
if(pixels_data) par {
  rgb_to_yuv( rgb, yuv );       //stage 1
  threshold( yuv, label );      //stage 2
  filter_vert (label, vf);      //stage 3
  filter_horz( vf, filtered );  //stage 4
  bounding_box( filtered );     //stage 5
}
```

In this example, the five functions each take one clock-cycle and each forms a stage in the pipeline. In Handel-C the functions are inside a `par` (Gribbon and Bailey, 2004) statement block to indicate that they operate in parallel, but, because of the data dependencies, they will form a pipeline; each function accepts data from the previous function and passes data to the next function block.

Once the pipeline has been primed, all the functions will, as the textual code implies, operate in parallel, but there is no explicit indication of sequential transfer of the data along the pipeline. The developer must infer this aspect of the design from the data dependencies.

For a single phase pipeline of moderate length, this type of representation, although hardly ideal, is workable as long as care is taken and incremental testing is used. But with multiphase pipelines, where the clock rate is a multiple of the input data rate, the situation is more complex. Moving the processing to the camera clock domain will avoid the need to buffer the image, and only the object positions will need to be passed between clock domains. For the RC100 development board, the video ADC only has 16 data lines, so to transfer a full 24 bit colour pixel takes two clock cycles, with a full pixel available every second clock cycles. The pipeline can either operate only when data arrives, using the same operations as before, or create a two-stage design. A single phase design would waste half the clock cycles. A two phase design would reduce the latency of the algorithm as all of the available clock cycles can be used. To implement a multiphase pipeline, the different stages need to be separated either by conditional statements (Handel-C) or by incorporating separate modules with differently phased clocking (VHDL, Verilog). A two-phase Handel-C version of the object tracking algorithm is shown below. Note the order of the stages, and that the implied dataflow is quite unclear because each phase is only clocked on every second clock cycle.

```
if(Pixels_to_process){
  if(phase1) par {
    rgb_to_yuv(rgb, yuv)                //stage 1
    filter_vertically(label,vf)         //stage 3
    bounding_box(filtered)              //stage 5
  } else par {
    threshold(yuv, label)               //stage 2
    filter_horizontally(vf, filtered)   //stage 4
}
```

This makes it much harder to follow the flow of data and makes modification and debugging very difficult. Developers have to rely on comments or stepping through the algorithm each time a modification or correction is needed, to avoid introducing data dependency errors. This is especially the case when adding a stage early in the pipeline; the phasing of all subsequent stages must be adjusted.

As illustrated in the textual code segment above, it is not clear that the latency of the pipeline has changed. Instead it needs to be inferred from the use of a

conditional statement and the comments. Other HDLs can also achieve this result but with different code, either through the use of conditional statements or by breaking the statements into multiple processes which are clocked on different clock signals.

As was explained earlier, a higher processing clock rate can be used to reduce the latency. However, the clock rate cannot be increase indefinitely, as the shorter clock period can restrict the operations that can be performed within each clock period. Operations with a propagation delay longer than a clock period will need to be split over two or more clock cycles, producing a longer pipeline. The optimum trade-off between latency and clock frequency is quite algorithm dependent. It is important that the language makes it easy to experiment with the timing by modifying the phasing of the design since in many applications; the propagation delay is difficult to estimate before the algorithm is mapped onto a particular FPGA device.

The representation used for pipelines should make it easy to manage the data dependency, which is a consequence of the sequential characteristics of the pipeline and not the pipeline's ability to process two or more items of data in parallel. With multiphase pipelines an ideal representation would also show when new data arrives into the pipeline, so that the pipeline phasing can be determined. Programming such systems with a textual notation is difficult; as pointed out above, textual notations can be extended to provide a somewhat strained representation of parallel systems, but they are quite inadequate for representing systems that have some sequential components, some parallel components and some components that combine some aspects of both. Visual notations can be used to represent these three processing modes. They can also allow multiphase pipelines to be expressed more clearly.

5.3 Pipeline Visualisations

The parallel and sequential representations described are distinct. It is desirable to find a clear, informative representation for pipelines. Given the constraints of time progressing horizontally and the vertical dimension being used to allow simultaneously active processors to be distinguishable on the diagram, there is no obvious representation which would achieve this. This section investigates a number of possible representations, and comments on their benefits and limitations. After this evaluation, the most suitable representation is described in detail.

5.3.1 The Diagonal Gantt

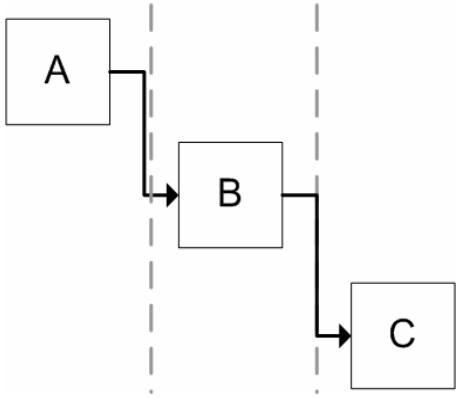


Figure 5.12: In the Diagonal Gantt each pipeline stage is translated across and down one space from the previous stage.

Gantt charts have been used as a tool for project planning for many years. They allow for a sequential group of processes to be planned and the project timing estimated. The Gantt chart metaphor can equally apply to computational statements. In a Gantt chart, parallel activities occur under each other, therefore it can represent both sequential and concurrent processors. However, Gantt charts were not designed to visualise computer operations. While they can allow the visualisation of parallel and sequential processes, they do not directly have the facility to represent pipelines. 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. Figure 5.2 had parallel operations displaced vertically and sequential ones displaced horizontally. Extending this idea leads to a diagonal Gantt representation for pipelines. Figure 5.12 shows such a three-stage pipeline. A pixel enters processor A on clock pulse 1 and moves to processor B 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. B is displaced vertically with respect to A to indicate that the pixel in B is being processed in parallel with another operation (processor A, working on the next pixel, and after the pipeline has been fully primed, processor C working on the previous pixel).

5.3.2 The Staggered Gantt

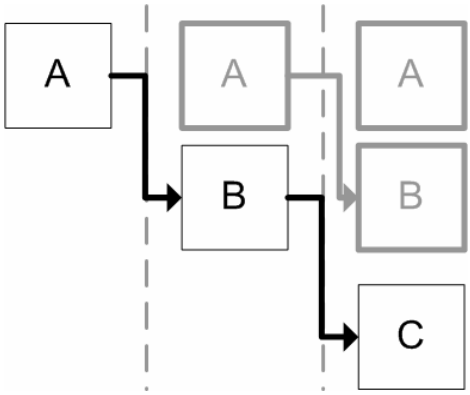


Figure 5.13: The Staggered Gantt illustrates that the pipeline is active on successive clock pulses.

Although it is possible to interpret a processor's vertical position in the Diagonal Gantt as an indication of the number of processors with which it is concurrently active (at least until the pipeline starts flushing), 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 Staggered Gantt, shown in Figure 5.13, is an attempt to make this parallelism explicit. 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.

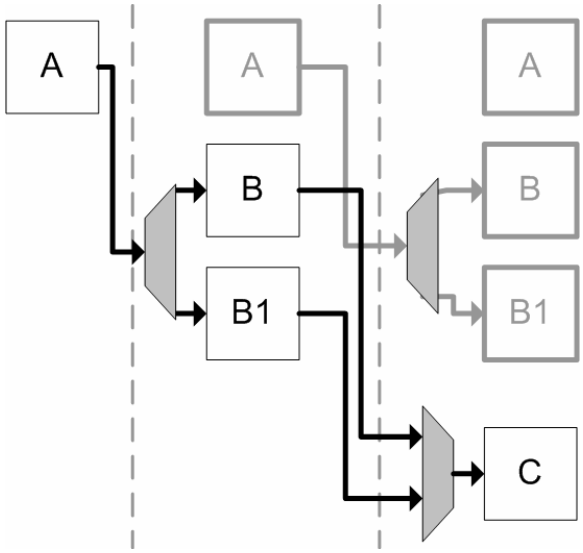


Figure 5.14: The Staggered Gantt with conditional branching.

The Staggered Gantt view also has limitations. Figure 5.14 shows a design in which multiplexors and demultiplexors are used to select between different branches in the pipeline. It is apparent that the repeated sets of branching paths make the diagrams in the Staggered Gantt notation comparatively complex and difficult to understand.

There is a further complication. Process A is shown in columns 1, 2, and 3, which signifies that it receives new pixel data on clock cycles 1, 2, and 3. A multiphase design can be indicated within the notation by moving the second box representing process A to column three, thereby indicating that it gets new data on clock cycle three. The notation is capable of representing this, but the diagrams become more complex and unwieldy.

5.3.3 The Sequential Pipeline

To resolve these issues I abandoned the idea that the data must always be shown entering the pipeline on the top row of the diagram (which leads to the diagonal datapaths seen in the first two visualisations). Instead, I experimented with a notation in which the pipeline is treated 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. As was discussed in the introduction to this chapter, it is better to treat a pipeline as a special case of sequential operations rather than parallel operations. Visually, this change in emphasis results in a more intuitive pipeline that extends from left to right across the diagram – a diagrammatic layout that is consistent with the convention that time increases from left to right (see Figure 5.2) rather than diagonally. To distinguish a pipeline from a set of sequential processors, highlighting is used in this notation to show when new data arrives, with a change in background colour indicating successive data items. A slider is used to set the pixel clock rate relative to the data clock. Figure 5.15 shows the Sequential Pipeline with data arriving every clock cycle and Figure 5.16 shows the view with data arriving every second clock cycle.

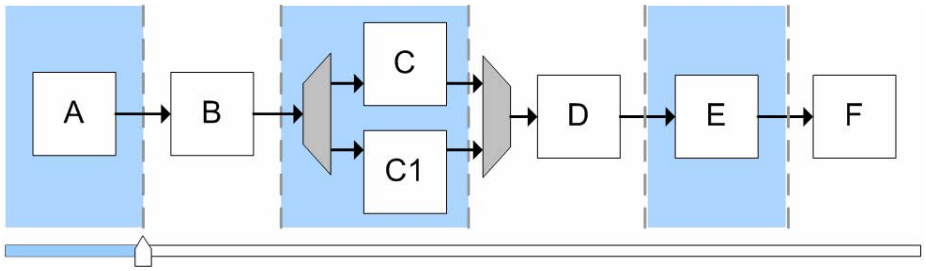


Figure 5.15: The Sequential Pipeline view uses coloured bars to show the "extent" of each pixel. Here data arrives every clock cycle

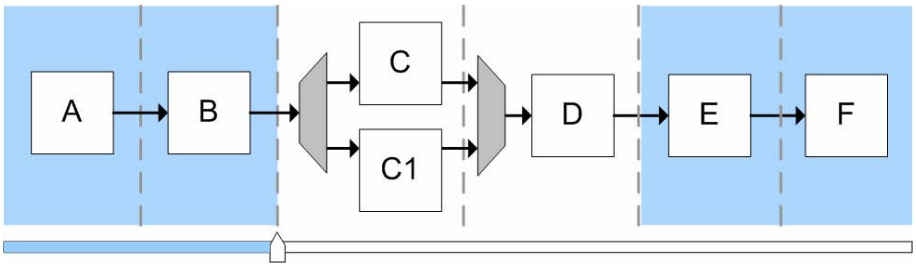


Figure 5.16: The user drags the pointer one clock cycle to the right to design a pipeline in which data arrives every second pipeline clock cycle. The coloured regions denoting the amount of processing that occurs in a single pixel clock cycle automatically double in width

Although this view is compact, and involves a more intuitive representation of the position (and temporal separation) of successive pixels in the pipeline, it is less intuitively obvious that the processors are operating concurrently.

5.3.4 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 schematic representations of the pipeline as coloured 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. 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 a pipeline's maximum data throughput. Figure 5.17 illustrates this view for data with an inter-pixel delay of 1 and 2 clock cycles respectively.

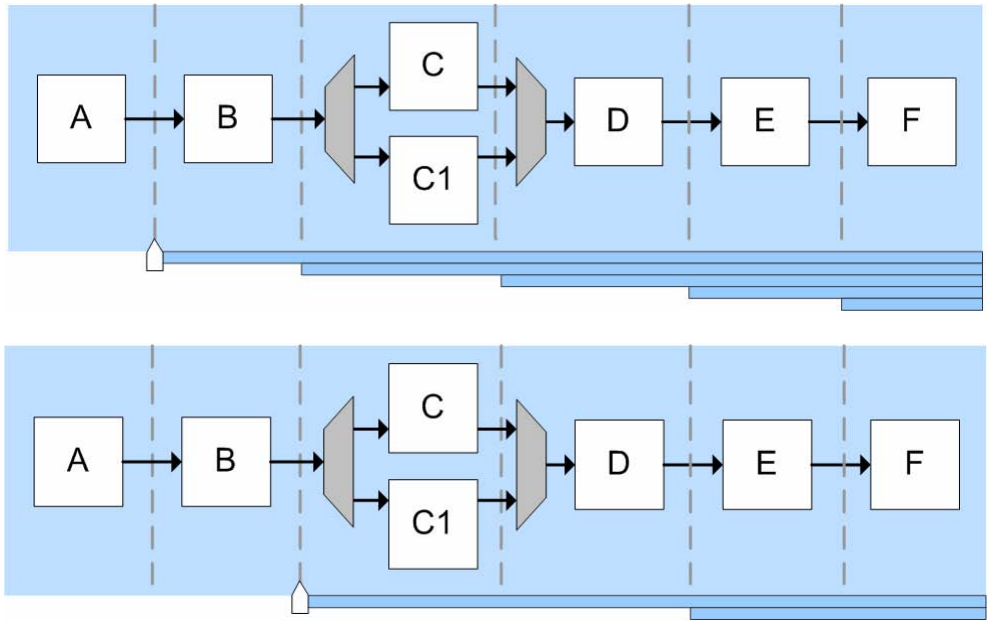


Figure 5.17: The Sequential Pipeline with Staggered Bars clarifies the relationship between the pipeline clock and the pixel interarrival delay: The top view shows data arriving every pipeline clock cycle, whereas in the bottom view, the control has been dragged to the right to indicate that data arrives every second pipeline clock cycle.

5.3.5 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 5.18, restores this property to the interface.

This 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 specify which phases those will be. The arrangement of the bars also shows how long it takes to prime the pipeline with valid data (and, by symmetry, how long it takes to flush it) and gives an indication of throughput. It also allows for more complex pipelines by allowing conditional execution of processors through multiplexors, and it shows that later pixels are being processed by the same hardware as the first pixel.

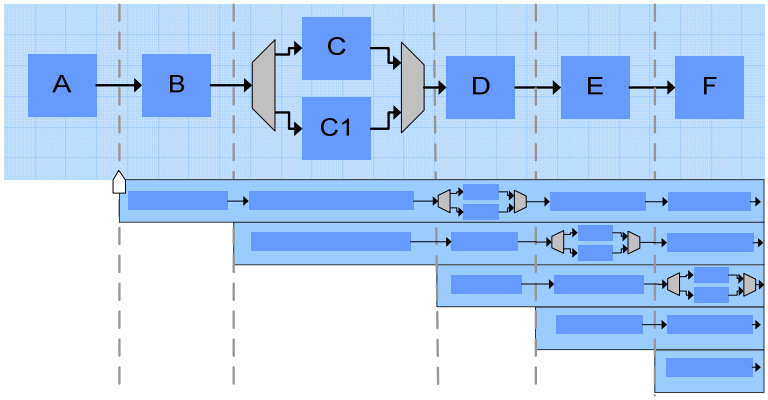


Figure 5.18: The Staggered Sequential Pipeline with Detailed Bars shows how the pipeline operates on successive pixels

5.3.6 Data Explicit Pipeline Representation

There are a number of limitations to the graphical representation shown in Figure 5.18. During an informal user evaluation it was found that users sometimes calculated the latency by counting the number of bars under the architecture. They omitted the full architecture diagram at the top, which is, from this point of view, just another bar. This led to an underestimate of the latency by one data rate clock period. A second problem was that users did not always understand that the bars indicated when next data pixel was to start processing; they thought that it showed that the hardware was replicated, not that new data is being fed into the same pipeline hardware. This confusion was compounded by having a representation of the hardware architecture, albeit a reduced one, repeated in the hardware bars. It was felt important to clarify the movement of the data through the pipeline. The pipeline latency also needs to be more clearly illustrated. The final pipeline representation aims to make the temporal data flow more apparent.

This notation retains the idea of bars to represent the processing of data through the pipeline. However, there are some key differences. Firstly, the section that shows the pipeline's internal architecture is not used as one of the bars, so a five stage single phase pipeline will show five similar-looking bars. The design of the bars has also changed. Now they show, from top to bottom, the progression of the first piece of data till the latest piece of data to enter the pipeline (as labelled in Figure 5.19). This represents a fully primed pipeline in operation. Arrows to show the movement of the data through the pipeline, light grey arrows show the previous stages the data has moved through. Black arrows show the next stage the data moves into. Dark grey arrows show what stages the data is yet to pass through. This both

allows the latency to be quickly determined and shows how data progresses through the pipeline illustrated below it. The bars could be placed either above or below the pipeline. However, having them above the pipeline seems to aid in disconnecting the bars from the pipeline architecture. Figure 5.20 shows the representation as implemented in the VERTIPH prototype.

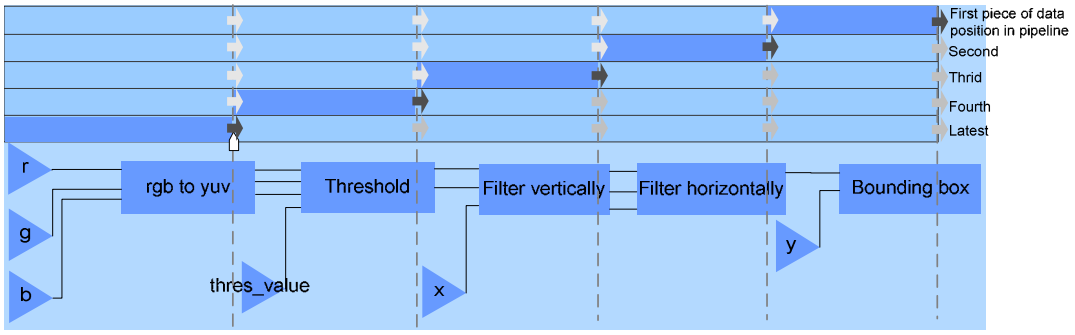


Figure 5.19: Data explicit pipeline view for single phase operations

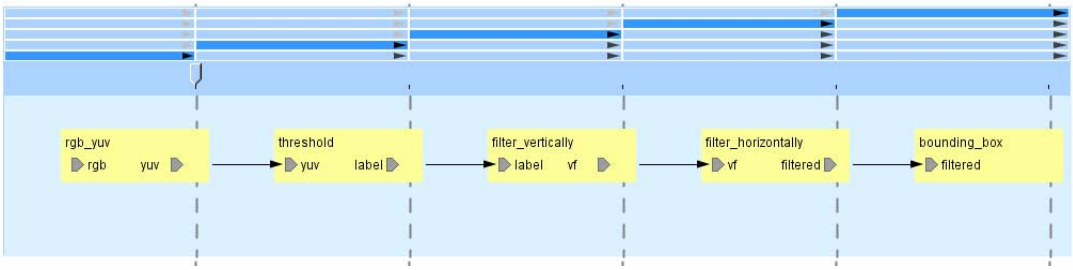


Figure 5.20: Single phase pipeline representation, real example of the Data Explicit Pipeline representation

In Figure 5.21 the two phase version of the pipeline is shown. In this, the latency is 2½ data rate clock cycles. Again the arrows show when the data moves between different phases of the pipeline stage. Based on the uneven data rate caused by the 5 stages in the pipeline it might be decided to buffer the result to keep the timing consistent with the data clock. This representation makes it clear that some action needs to be undertaken to achieve this.

Figure 5.22 shows the same pipeline as a three phase design. In this the latency with respect with to the data rate is $1^{2}/_{3}$ clock cycles. Again, any buffering which may be required is obvious from the representation.

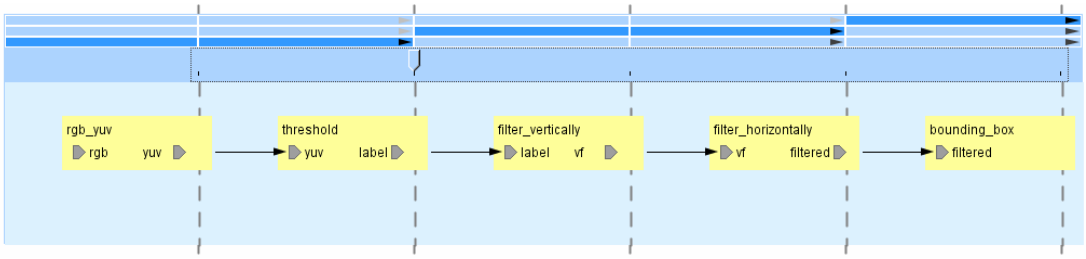


Figure 5.21: Two phase explicit pipeline view

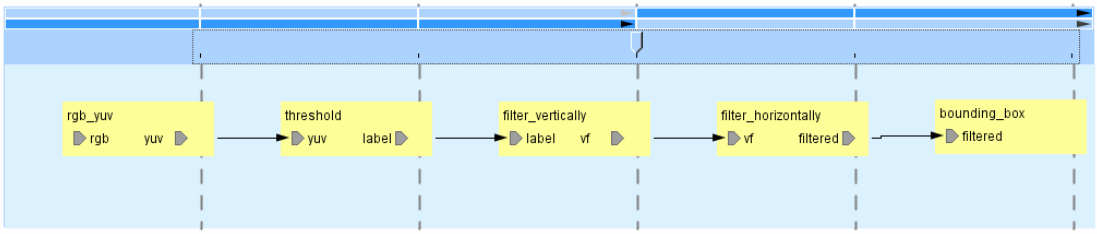


Figure 5.22: Three phase explicit pipeline view

5.3.7 Pipeline Control

Having the pipeline represented explicitly can also aid in representing the control of the pipeline. As discussed earlier, pipelines have special control phases. First a pipeline needs to be primed. This involves filling it with data so that it will produce valid output. When the end of the data stream is reached, the pipeline needs to be flushed. That is, it needs to continue operating until all the input data has been processed. Otherwise the data is stuck in the pipeline and will not exit until new data arrives. This flushing is best done by inputting constants which will reset the pipeline to a known state which can be used when priming the pipeline. Stalling in a conventional processor can lead to the introduction of bubbles into the pipeline due to a lack of instructions or data. In a hardware design as the instructions are built in hardware stalling occurs due to a lack of data. In streamed (real-time) images, the frame is broken up into pixel data with blanking periods between rows and frames. At the end of a row or frame, the incoming data will run out, but more data can reliably be expected to arrive soon. Stalling the pipeline does not delay the output of the last pixels in the row or frame for a significant period. This depends on whether the processing is input driven or output driven. If it is output driven then stalling is not an option and data needs to be buffered beforehand to ensure this is not the case.

There are other common control cases. For example, when there is no input data, it may be appropriate to stall the pipeline, or to set a flag on the output to denote that it is invalid, or simply to let the developer select and implement the best

option for their application. Defining these activities is exacting work, and is often associated with errors. Flushing and priming control logic can be generated automatically by the compiler, but only if the hardware description language has provision for representing pipelines explicitly. The HDL could also allow the developer to specify what action is required to control the pipeline at its different stages; priming, flushing and stalling.

The representations shown in Figure 5.20, Figure 5.21 and Figure 5.22, show the normal operation of the pipeline once it is loaded with data. Further control logic is required to handle the boundary cases (when data starts or stops). In stream based image processing, these typically occur at the start and end of each row of data, and at the start and end of each frame.

The particular control mechanisms depend on whether the timing is source driven (for example data streamed from a camera) or sink driven (for example data being streamed to a display). Defining these control activities is a fruitful source of errors. Again, some of the control logic can be generated automatically by the compiler if the hardware description language explicitly represents pipelines.

It is necessary to ignore the invalid data that the pipeline produces while it is being primed. For source driven pipelines, this may simply be a matter of producing a data-valid token when valid data reaches the end of the pipeline. However, the process is a little more complicated when pipelining filters (as discussed in section 5.5.2), which require data from several rows of an image before they can produce valid output (the filter has row buffers to cache input data from previous rows). Designs that require caching, like filters, need a two dimensional pipeline, as data is required over rows and columns in the image and not just from the current row. This implies that each block may require additional control signals to indicate the start and end of line and start and end of frame to enable correct initialisation and determine when output data is valid. For straightforward cases, such as filters, the logic may be created automatically by the compiler, but more complex cases will require the user to provide additional pipeline control logic.

For sink-driven pipelines, it is necessary to preload the pipeline with valid data so that data will be ready on the output when required. For simple cases, this involves starting the pipeline at some predetermined time before the output data is required. In more complex cases, it may be necessary to start the pipeline as soon as convenient when data is available at the input, and then stall the pipeline once it has been primed and valid data is ready on the output.

At the end of each row or frame, additional control logic is required to flush the pipeline. For source-driven processes, this requires running the pipeline for several clock cycles after the last valid data has entered until all the valid data has passed through the pipeline. Again, with filters the control becomes more complex because they may require running the pipeline for several rows after the last valid data is input.

For sink driven processes, it is only necessary to continue running the pipeline while output data is required (assuming that the width of the output image is less than or equal to the width of the input data). When the end of row is reached, priming may begin immediately for the next row. By having an explicit representation for a pipeline, it may be possible for much of the logic to control the pipeline to be inferred automatically by the compiler. This is particularly the case with the normal operation of both standard and multi-phase pipelines. For one-dimensional pipelines, the control logic for priming and flushing both source and sink driven pipelines can also be inferred.

With two dimensional pipelining, the control logic is more complex because it must also take into account cache management (for keeping row or column data over multiple lines). Caching may require a high-level event-triggered and state-based controller (such as that described in (Johnston et al., 2008) and the scheduling chapter) to provide the auxiliary control signals. While the developer must still design the complex priming and flushing logic, this will be made easier by having an explicit representation of the pipeline that models its sequential behaviour.

5.4 Control Structures.

Like any programming language, HDLs need control structures. In textual languages, data manipulation code and control statements are not well distinguished and the controls' moderation of the data manipulation operation may be obscured. Nested control is hard to show as the control can be spread over a considerable physical extent. In chapter 2 this thesis looked at a number of different visual languages. Some data flow languages use control structures similar to the multiplexor (for `if-else` control), for example, schematic entry tools for digital design. Others like Labview use special boxes for loops and a box for conditional control where only one path, either the true or false is visible at once. Other grid type languages use containing controls. Some visual languages, such as Subtext (Edwards, 2007), have abandoned `if-else` structures completely. The notation which this thesis has developed was inspired by N-S (Nassi-Shneiderman) diagrams (Nassi and

Shneiderman, 1973) and these have been used as a starting port for the design. This notation also aims to tap into the brain's perceptual processing capabilities, so that analysis of the program structure is transferred from conscious cognitive processing to the lower neural-processing levels of the brain. To do this colour coding has been used for control operations. Although the language uses a visual representation that highlights its control structures, and, in particular, the *extent* of its control structures, it does not abandon the conventional (Algol-based) layout of control. Thus, it should tap into users' familiarity with such layouts. N-S control structures use lines and boxes to separate and control processors.

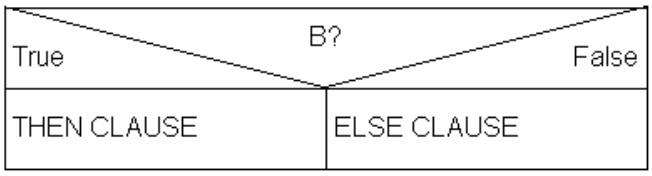


Figure 5.23: Conditional branch selection (from (Nassi and Shneiderman, 1973))

Branching blocks are represented as a downwards-pointing triangle with the action that is performed when the condition evaluates to TRUE shown under the left hand side of the triangle. An action that is performed when the condition evaluates to FALSE is shown under the right hand side of the triangle. Figure 5.23 shows a conditional branch which is equivalent to an **`if-then-else`**. In this example if B is TRUE the **`then`** clause processes will run. If B is FALSE the **`else`** clause becomes active. This control structure is not directly suitable for use with VERTIPH. Firstly, as the developed notation uses the horizontal direction to show sequential operations, having the TRUE condition followed by the FALSE condition implies that they run in sequence one after the other. This N-S representation also uses up too much room although parallel operations will fit nicely into it. Sequential and pipelined operations will result in a doubling of the horizontal space required. Instead VERTIPH's **`if`** and **`switch`** statements are based on an F-shaped representation. This has the advantage (discussed later) that the TRUE and FALSE conditions are above and below each other which implies they run in parallel, but with only one path active based on the condition. The expressions are contained visually within the control bars. The vertical bars can also be used for developer's comments. Figure 5.24 illustrates this construct. The user editable Boolean expression (labelled (a)) encloses the area which contains the expression that run when it evaluates true (labelled (b)). A horizontal bar (c) contains the else not condition. This combined with the vertical bar and the

light shading contains the code that this executed if the Boolean expression evaluates to false.

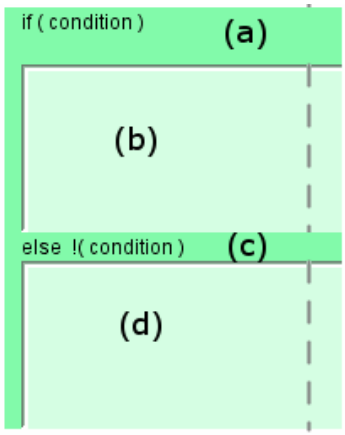


Figure 5.24: 'F' based `if-else`

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 N-S diagrams. Instead, the clauses are aligned vertically, as they would be in conventional textual languages. This similarity should make the language a little more familiar and easier to use. This arrangement also allows time to increase from left to right across the diagram, with the parallel alternatives stacked vertically The control structures use vertical and horizontal bars to show the scope of the operators. It is planned to allow comments to be written in the vertical bars, taking advantage of the trend to monitor screens with a 16:9 aspect ratio.

In N-S a loop constructs are based on an 'L' shape such as the **`do-while`** shown in Figure 5.25. In this diagram, the loop encloses the expression it controls by horizontal and vertical bars. In the VERTIPH N-S inspired notation the **`while`** construct and the other loop structures (**`for`** and **`until`**) very similar to those in N-S diagrams.

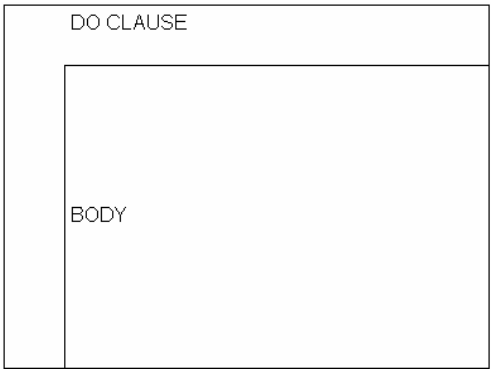


Figure 5.25: While or For construct, from (Nassi and Shneiderman, 1973)

Figure 5.26 shows the **`while`** control, a Boolean expression that is evaluated (a) at the start of the loop, and after each iteration. The loop contents (b) are executed while the Boolean expression is TRUE. The **`until`** control is similar, however, in this case the Boolean expression (a) is evaluated after each iteration.

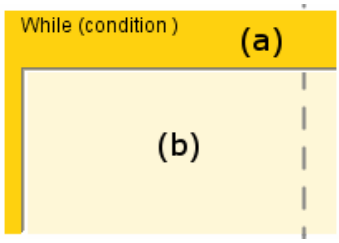


Figure 5.26: While loop

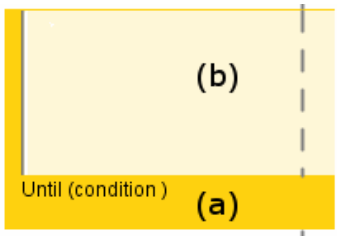


Figure 5.27: Until loop

The computational view's visual control structures all incorporate a vertical bar that extends the height of the controlled constructs and is intended to indicate clearly the extent of the code that is controlled.

This thesis has identified the need for control structures at two levels. The first is low-level data-dependent control, where the choice of program action depends on the data that is received. For example, in Chapter 2, an algorithm for identifying a uniformly coloured regions in an image was discussed (Johnston et al., 2005a). This would incorporate an if-statement controlled by a Boolean expression such as a pixel-

value less than A and a pixel-value greater than B. It is this type of control which the VERTIPH control structures described above are primarily designed to represent.

The second is high-level control, which operates at a processor or global level. In many algorithms, some or all of the processors should only run during certain defined periods. A good example of this is an algorithm for building a histogram of pixel values in an image from a video source. One processor is used to build the frequency histogram of pixel intensities by incrementing the count into the correct bin. Another processor is used to reset the histogram at the end of the frame before a new image arrives. These two blocks need to be scheduled to run at different times, as they access the same memory resource and one runs as valid pixel data is arriving, and the other when the frame ends. This higher level type of control in not intended to be implemented within the computational view, and will be described in the scheduling chapter.

5.4.1 Dataflow and Pipeline Branching

In the previous examples, a multiplexor and de-multiplexor have been used for branching operations. These were used as they are common in many dataflow languages and are used in hardware design. There is already a selection control, the *if-else* or *switch* construct, so having two equivalent controllers is redundant. The path selection with a multiplexor and demultiplexor is unclear. The multiplexors acts like a pair of switches, controlled by a single shared condition. In Figure 5.28 a process has two different data paths. Process ‘A’ feeds either into ‘B’ or ‘B1’ based on the state of the condition. The data path then feeds back into ‘E,’ based on the same condition. It should be notated that this illustration is ambiguous; it assumes that the condition is constant through out the processing. To ensure this the case the control signal needs to be registered at each clock edge. For the sequential case here this may not be required as new data cannot be processed until the processing is finished. For a pipelined operation the condition must be registered otherwise the data may not be directed correctly. It may be what the developer intended (no buffering) but it can be interpreted in more than one way, which can lead to errors.

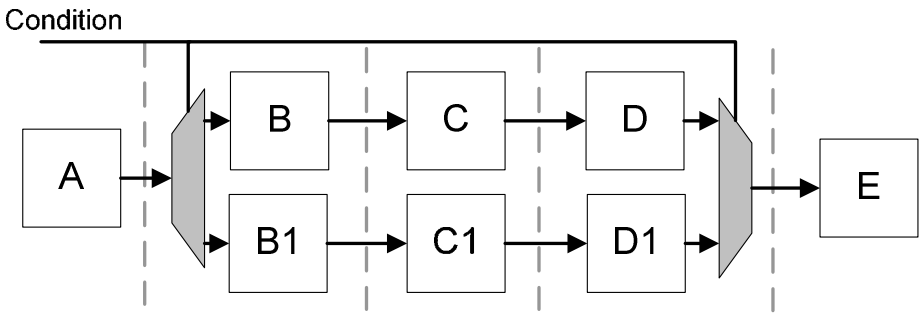


Figure 5.28: Path selection with control based on multiplexors

This figure can be decomposed into two different figures if the multiplexors and demultiplexors are treated as switches. Figure 5.29 when the condition is TRUE, shows that data moves from 'A' to 'B' to 'C' to 'D' to 'E' and Figure 5.30 shows that when the condition is FALSE it moves from 'A' to 'B1' to 'C1' to 'D1' to 'E'. A key insight which can be obtained from this is that the arrangement can be considered as 'A' writing into a register which is shared by both 'B' and 'B1'. The middle processors are then either scheduled to run when the condition is TRUE. That is, in hardware, rather than modifying the flow of data, operations that are allowed to run are selected and this selects which registers are written to and read from.

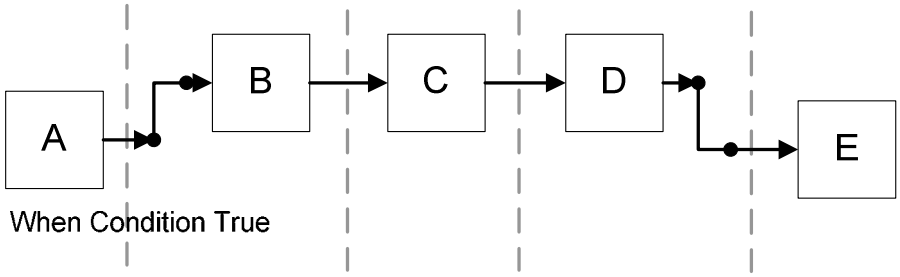


Figure 5.29: Data path for a true condition

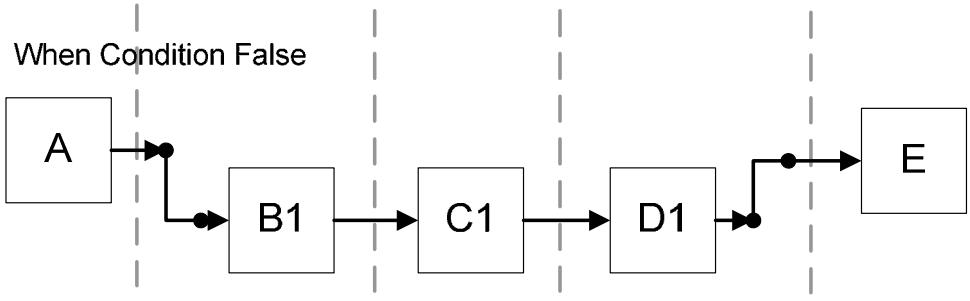


Figure 5.30: Data path for a false condition

Using this idea, rather than using multiplexors, an **`if-else`** control structure becomes a much better representation as it reinforces the fact that only one path is selected for execution. The fact that data needs to be shared *via* registers is only implied, as it is not essential; both processing tasks have the same input and output registers. As the branches can collapse later in the execution path, the control can be thought of as a combined multiplexor de-multiplexor, with inputs and output connected to the **`if`** statement rather than the internal operations. This allows the input to be split and then merged in the output so that input and output operations are always connected to a register. That is, the operations internal to the **`if`** control share input and output registers but that this is hidden from the developer. This type of control is illustrated in Figure 5.31 where 'A' connects to the control structure so that both paths can access its data. The results also connect to the control structure which results in a single output for 'E' to connect to. The **`if`** control can also operate over multiple clock cycles and in multi-stage pipelines, with the compiler building the necessary control signal buffers automatically and unambiguously from the control structure.

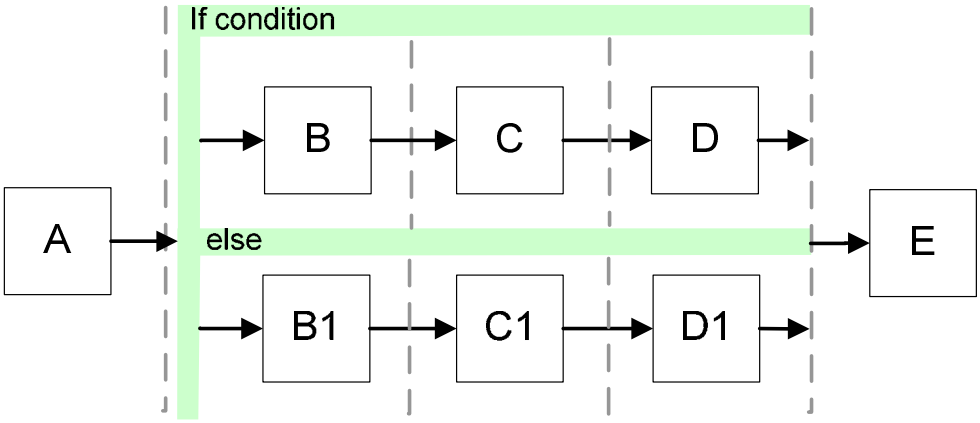


Figure 5.31: If-else based conditional branching control

5.5 Expression Editor

Operation blocks contain expressions which need to be edited. In a visual language, equations can be expressed graphically or as text. Based on the long use of algebraic notation in computer languages and its familiarly to developers, therefore the notation uses a C-like text representation. It is similar to the notation used by Handel-C which extends the standard C syntax with operations to handle bit manipulation. This also has the advantage of reducing the screen real estate of the expressions. While visual languages are good at representing relationships, visual

representations of algebraic expressions are not very succinct. As pointed out in (Green and Petre, 1992) following a wire and expression model is very slow compared to editing text notations for expressions.

Complex expressions will often take up a large amount of screen space. As much of the expression as possible is shown in the block and the whole equation is shown when the user hovers the cursor over the block.

I found in the user evaluation that participants would rather edit text than have the equations within the boxes (see chapter 9). To achieve this, a split screen could be used, with the operations in the selected computational block highlighted in an editor panel in the bottom screen. This links to the top panel which shows the relationship layout of the instructions. Selecting an object in one screen highlights the operation in the other screen. This might give a better balance between text and diagrammatic notations.

Another proposal for the editing of expression supported by the user evaluations was the automatic creation of inputs and outputs for expression blocks when the expression is edited. Both of these need to be implemented and their usefulness compared as part of future work.

5.5.1 Number Representation

The expressions are based on C, as it is a common language for implementing real-time and embedded image processing, but as VERTIPH is a HDL there is no fixed data width (like int, Boolean, or long etc). Instead there are only signed and unsigned numbers whose lengths can be set explicitly. The notation also allows the designer to specify bit selection, bit append and casting operations so that incompatible types can be truncated or padded to compatible lengths.

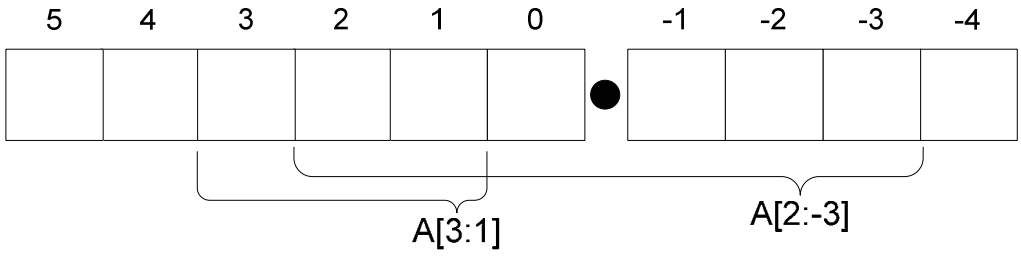


Figure 5.32: Fixed-point number line with bit selection operations

As discussed in chapter 4, VERTIPH uses fixed-point numbers, as it allows for the desired of resolution without the power and computational expense of floating point numbers. The bits in a fixed-point number are identified by their significance.

In the 10 bit number shown in Figure 5.32, with 6 bits above and 4 bits below the binary point, the significance of the bits to the left of the binary point ranges from 2^5 down to 2^0, and the significance of the bits to the right of the radix point ranges from 2^{-1} down to 2^{-4}. The numbers are binary. Therefore, the bit positions can be identified solely from the exponents of their significance. These exponents are shown in the diagram above the bit positions to which they apply, and are used to denote the extremes of the range when selecting a bit range. That is, A[3:1] denotes the three-bit field whose bits have significances 2^3, 2^2 and 2^1, and A[2:-3] denotes the 6-bit field whose bits have significances 2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, and 2^{-3}.

5.5.2 Filter Editor

As discussed in chapter two, filters are an important part of many image processing systems. Therefore, it is proposed to integrate domain specific tools such as a filter editor.

The filter editor, Figure 5.33, contains five parts. The editor uses a drop combo bock for specifying the shape of the filter's window (square, cross, "circular", line). In image processing, the designer must often specify the size of windows explicitly, so an N × M selection is provided to facilitate this. Within the representation of the window, the weight for each pixel can be user-edited (for linear filters) and the pixel register name is shown. The edge mode (specification of the operation to perform at the edges of the image) can be specified and a computational view editor is available, to allow the pixels to be accessed and the operations applied to them to be specified. Any shape can be constructed by selecting a square window then setting the weights of unwanted pixels to zero.

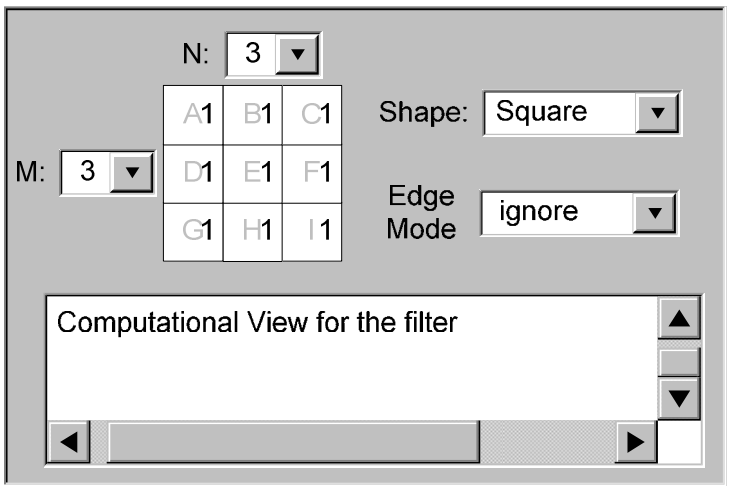


Figure 5.33: Main parts of a filter editor

The filter will have a main processing path to perform the normal filtering operation within the image. However, it will also need supporting control and operations for dealing with the case when the filter is not completely within the image (one such edge case can be. seen in Figure 5.34). Much of this additional control is associated with priming, flushing of pipelines, resetting tables and performing other initialisation tasks.

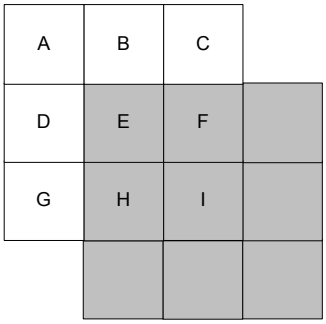


Figure 5.34: Filter overlaid for edge pixels of an image

This has been designed so that the filter editor can automate much of this extra design, hiding it from the developer. However, it is useful to discuss each option and its trade-offs both in terms of hardware and the effects on processing an image. Different options for dealing with edges result in different image processing considerations and will require different hardware to be created to control the priming and flushing of the window's registers and the row buffers.

To provide the window with pixel information, buffering is required. If pixel data is not available then the pipeline needs to be paused and any data in it flushed. To restart the processing, the buffers need to be primed by filling the window with pixel data before a valid output is produced. Assuming a streamed based processing model there are two basic architectures which can be used for designing the filter buffers. The window may either be in series or in parallel with the row buffers. In the series configuration, data from the registers is propagated into a row buffer (which will have a length equal to the image width minus the window width), as shown in Figure 5.35 (a). In the parallel configuration, the input goes straight through to the row buffer (with a length of the image width) and also propagates through the window registers Figure 5.35 (b). The series option requires a smaller row buffer while the parallel approach does not need to have its row buffer modified if the window size or shape changes. In either case the registers need to be accessed in parallel so that the window operation can be preformed.

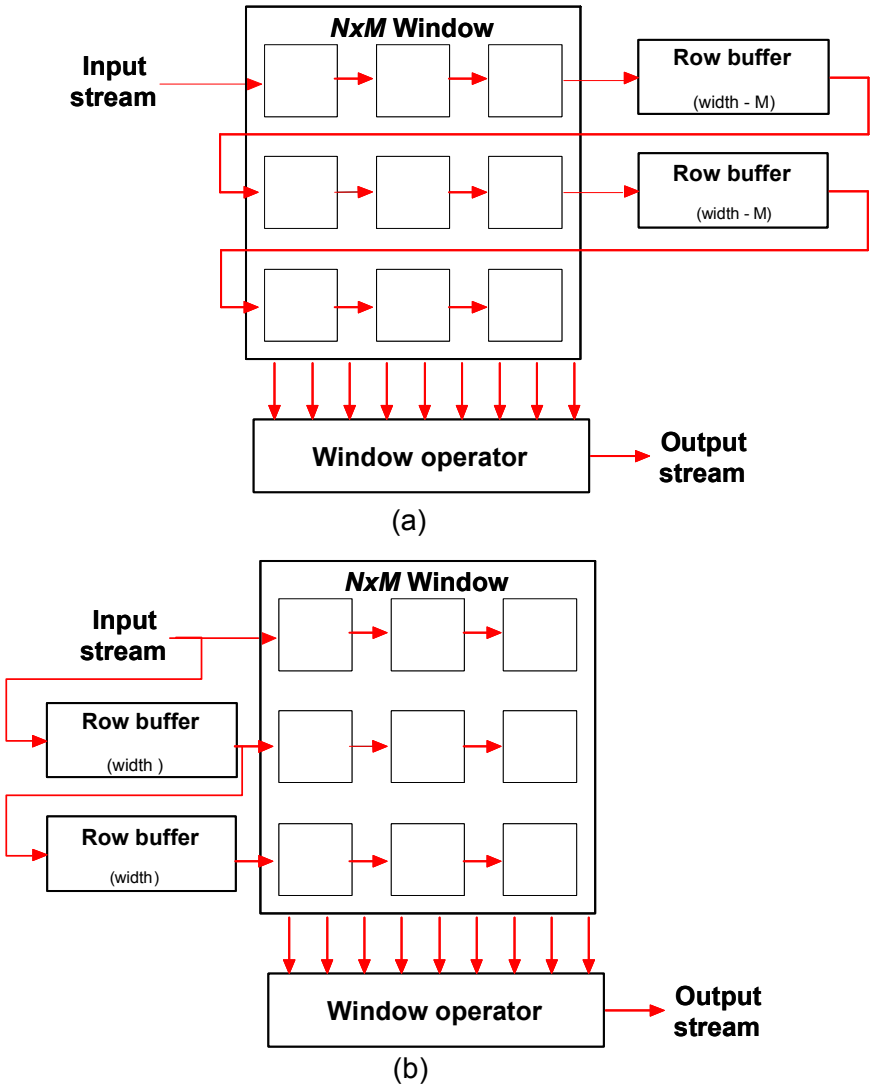


Figure 5.35: Filter architectures

For a filter, the standard processing pipeline takes the window registers and processes them using the operations which the developer has specified. First the pixels are multiplied by the weights. These pixels are then combined or compared with a mathematical or logical operation to produce a single output pixel to represent the centre pixel in the filtered image. The operations may take one or many clock cycles with multi clock cycle designs normally implemented as a pipeline.

To process the image around the edges, there is a need to generate pixel data for the pixel positions outside the image. Without the addition of pixel data around the edges, the result from the filter would be invalid. To add the edges for the image to be filtered, pixels need to be added to the first and last lines. This requires a line of pixel data to be put into the filters buffer before the start of the image and following the last line of the image. When processing the edges of image rows, the window registers need to have pixels ahead of the new line and at the end of the line.

The simplest option (in terms of hardware required) is to ignore the edge pixels and the effects these will have on the image. This is the simplest because whatever pixel values happen to be in the row buffers are used as though they are valid data. This effectively wraps the end of one row onto the beginning of the next row, and the last row of one frame onto the beginning of the next frame. This requires no logic for priming or flushing the pipeline, apart from accounting for the latency of the filter operation. It will require logic to stall the filter operation if there is a delay between rows and frames. While the simplest, this can introduce artefacts around the edges of the image resulting from spatially separated pixels.

The next simplest approach is to truncate the output image so that output pixels are produced only when the filter window is completely within the image. This requires the later stages to be informed when the output data is valid. Two approaches to this are to use a token-passing design to indicate data validity, or to incorporate some simple conditional control logic that prevents invalid data from being passed on to the next stage. The image processing trade off is that the image size is reduced. This will affect the later stages of the design as any downstream caching will need to take account of this the new size of the image. The larger the filter's size, or the more filters that are cascaded in the image processing system, the smaller the output image becomes.

To retain the whole image size at the output, either the input image or the output image needs to be padded. Padding the output using pixel replication or mirroring will increase the filter latency because the first valid row is not produced until the whole window is within the image. This replicated first row is output multiple times, requiring additional row buffers to delay the subsequent lines. The replicated output produces a flat appearance around the edges of the image. An alternative to replication is mirroring where the valid data is reflected to fill in the invalid output. Since the first row output with mirroring may be several rows further down the image, this has even greater latency and more row buffers. If there is a gradient in the output image near the edge, a false ridge can appear around the border of the image.

A better choice is to pad the input image. The simplest approach is to pad outside the image with a constant pixel value. This can be done by filling the row buffer from the edge pixels with the constant. It is necessary to know how the constant will affect the result of the filter. The constant should be chosen to suit the particular filter operation. The hardware to achieve constant padding requires that the row buffer and registers in the window be primed with the pixel constant as

shown in Figure 5.36. If there is a delay between frames, this can be accomplished by priming the row buffer with the constant value at the end of the frame until the buffers are filled, then pausing the operation. This has the advantage that the bottom edges are handled and that the buffer is also primed for the start of the next image. If the row buffer is larger than the image to account for the buffer pixels, the logic can be reduced, as pixels only need to be added as the line is streamed through the window. This reduces the number of special cases which are required and therefore, the logic needed. This edge handling scheme is quite simple to implement in hardware and requires only modest hardware increases, enlarged buffers and some conditional logic. More complex is the case where there is no delay at the end of the row or between frames. This requires considerable additional logic to determine which pixels are constants, and which come from the input data. From an image processing perspective, the fixed constant can result in undesired effects. This will depend on the filter type and the constant selected. One example would be an averaging filter that takes a weighted average of a square window. If the constant is black then the resulting average will be darker.

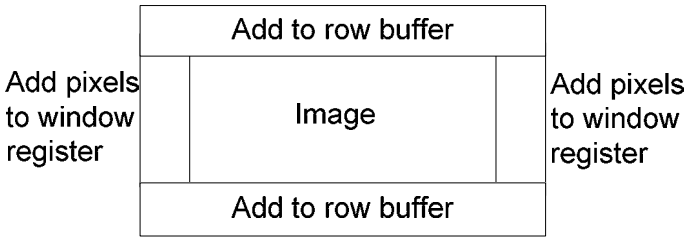


Figure 5.36: Where pixels are stuffed into

An alternative padding scheme is to replicate or mirror the input pixels from the edge of the image. In this the "closest" pixel to the edge pixel is used, which is a copy edge pixel. As shown in Figure 5.37 the corner pixels are used to mirror 3 pixels (assuming a 3×3 square window) while the other edges only mirror the current pixel using the pixels next to it to fill the rest of the pixels.

A	A	B	C	C
A	A	B	C	C
D	D	E	F	F
D	D	E	F	F

Figure 5.37: Mirroring of pixels

To do this the hardware needs to allow the registers and row buffers for multiple lines to be written to at once. By allowing the first line and last line to be written to the buffer and window twice, the data is mirrored. For the left and right pixels, two approaches can be taken. One is to expand the size of the buffer and fill the extra positions with the start and end pixels. This also removes the difficulty of a special case for corner pixels. Otherwise the pixel data needs to be added to the registers. This results in conditional loading (the need for extra control logic to do the load) for all the side pixels and the corner pixels.

Depending on how it is implemented, the hardware requirement is only slightly greater than for the constant pixel option. Again, the logic is simpler if there is a delay between rows and between the frames, but it is possible to filter continuous streams without such gaps in the data with additional control logic. Replication or mirroring can also introduce artefacts around the edge of the output image, although they are generally less noticeable than replicating or mirroring the output image.

Finally, the filter function can be modified around the edges of the image. This requires the construction of special cases for the processing of the images with a special case needed for each possible overlap of the filter window with the image. This results in considerable hardware repetition and therefore is the most resource intensive method, although if carefully designed can give the least artefacts around the edge of the output image.

The best trade-off is application-dependent, which is why the developer must be allowed to select the design most appropriate for their application. From a hardware perspective, the simplest case is ignoring the edge effect. Mirroring on the input is only slightly more complex than extending by a constant value but produces much better results, and is usually the best compromise for most applications.

Having a filter block is a desirable feature. Though this section has described the requirements of such a filter block it is not yet implemented in the prototype. However, filters can be implemented manually through the use of row buffers and registers, with the developer designing and entering the specific control logic.

5.6 Design Example

Let's consider the design of a simple pipeline. Firstly, starting with the operation shown below (Figure 5.38), this expression is relatively complex with both additions and a multiplication. To increase the speed of the design the expression can be broken up into simpler parts to form a pipeline.

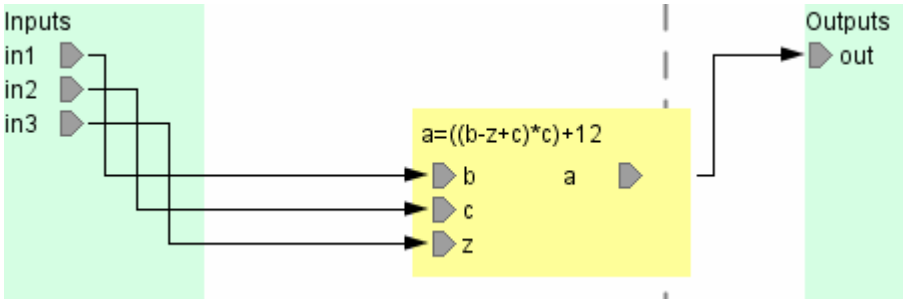


Figure 5.38: A complex expression

This expression can be broken up into the four parts shown in Figure 5.39 by separating the subtraction and addition, then doing the multiplication and then the final addition. This sequential operation will take four clock cycles and there will be a shorter propagation delay for each operation so the design should be able to run at a higher clock speed. In the serial design, a one-hot state machine is produced which will turn on the operations in each clock one after the other.

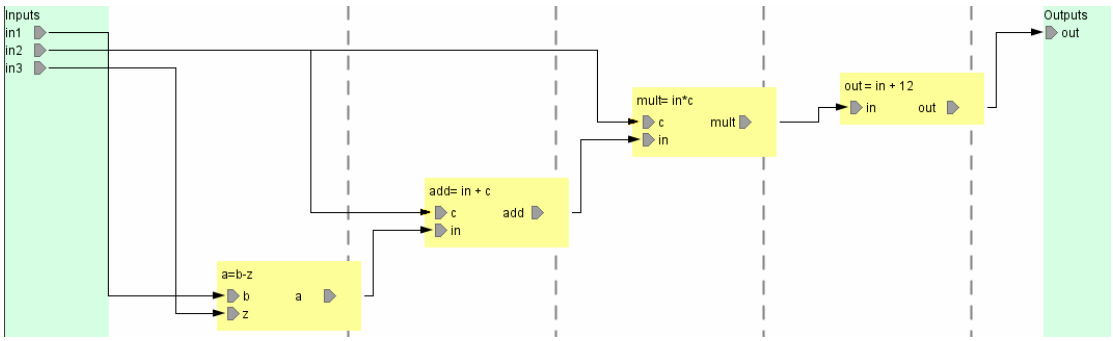


Figure 5.39: Converted to a sequential design

This design can be enclosed by the pipeline control structure to make it a pipelined operation (Figure 5.40). As 'c' is used twice in two different clock cycles there is a warning on the wire connections. If the designer knows that 'c' is a constant value then they should be given an option to suppress this warning (planned but not in the current prototype).

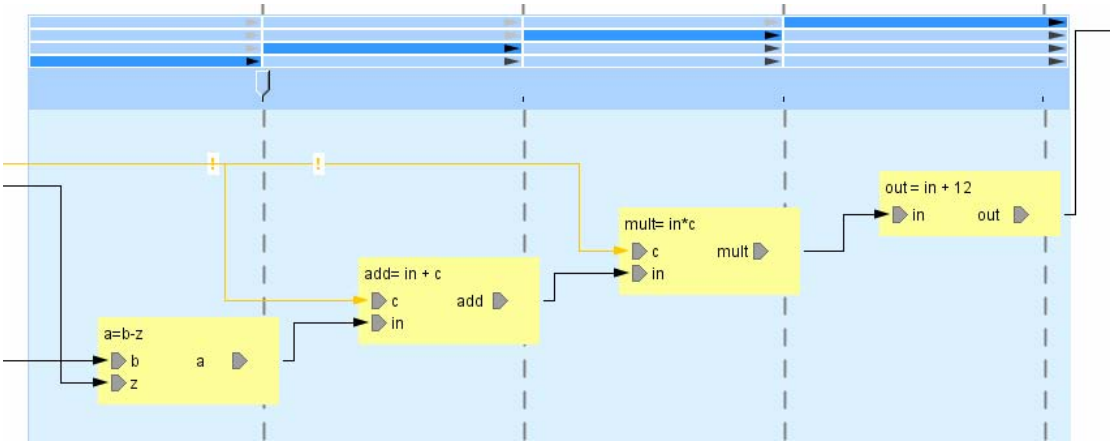


Figure 5.40: Conversion to a pipeline

Though registering the outputs is not required for a serial design Figure 5.39, in a pipeline any changing variables must be registered. The appropriate registers can be added as in Figure 5.41. The design can now accept new data every clock cycle and has a latency of four clock cycles. In the pipelined designs, the state machine sets all the stages active at once.

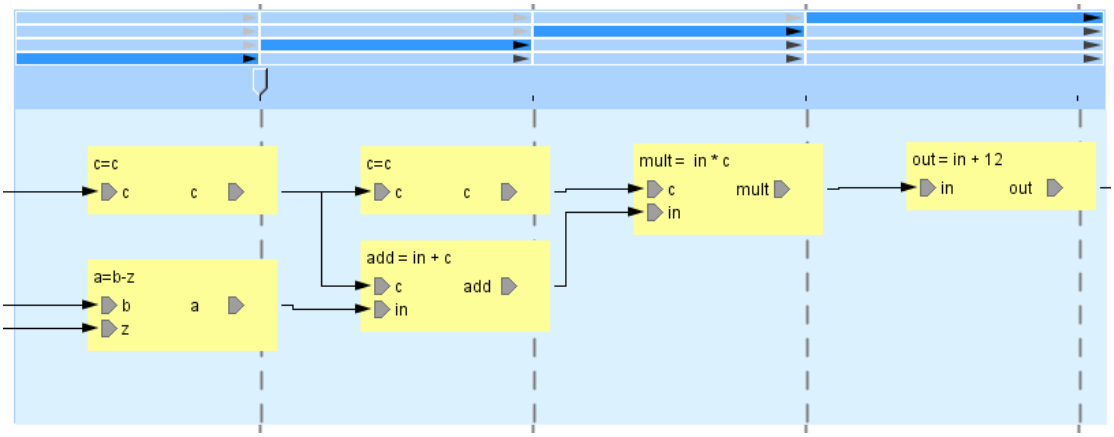


Figure 5.41: Pipelined Operation

To convert to a two phase design the slider can be dragged, Figure 5.42. This signifies that the data arrives every two clock cycles, either because the global clock is twice the data clock rate or because the computational block has been defined as running at twice the clock rate. This design now has a latency of two data clock cycles. The first register is no longer required as the input will not change until after the second pipeline clock cycle. In this design the state machine activates the odd clock cycle operations then the even ones.

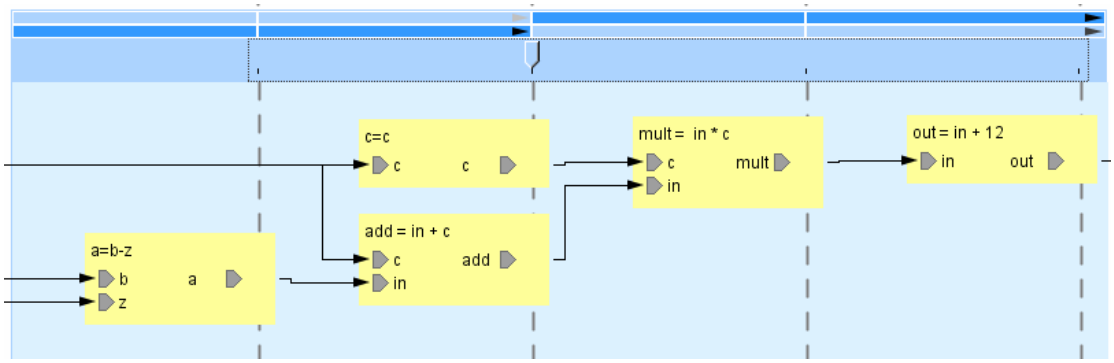


Figure 5.42: Two phase pipeline

Now consider putting the serial design into an **if-else** control structure. Assume that if any input is negative then the output is zero, this will result in the design below, Figure 5.43. In this design the operations are only clocked if the conditions are met.

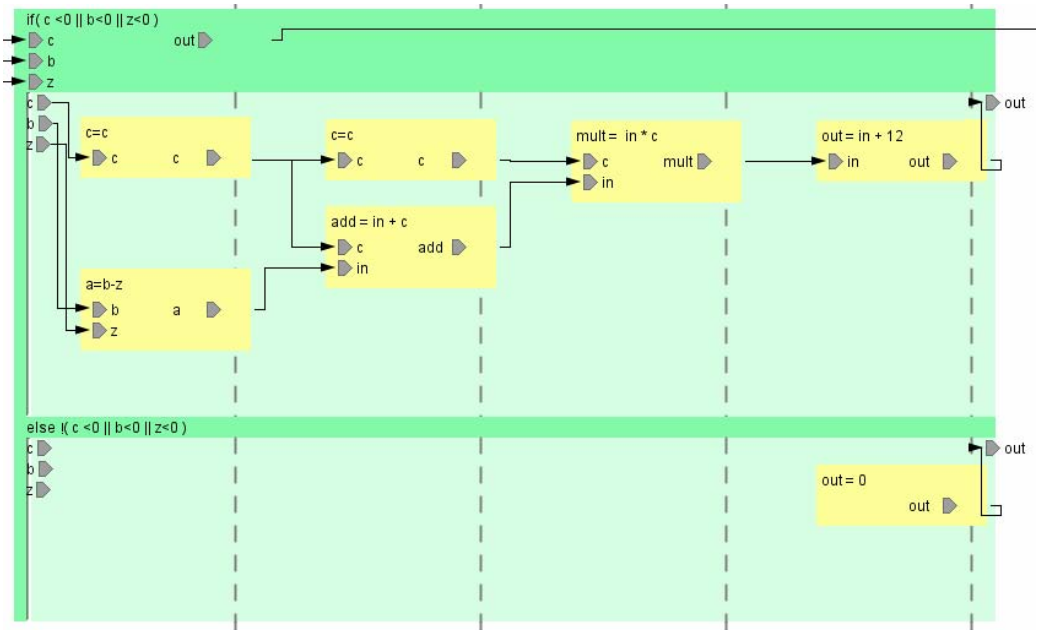


Figure 5.43: If else

This can be put into a pipeline which will result in an output every clock cycle; again with a latency of four clock cycles. In this design the condition is tested at the start and is then used to control which of the true or false clocks is made active for the first clock. This is passed like a token and stored in a register. This register is then ANDed with the driving clock signal. This is illustrated in Figure 5.45.

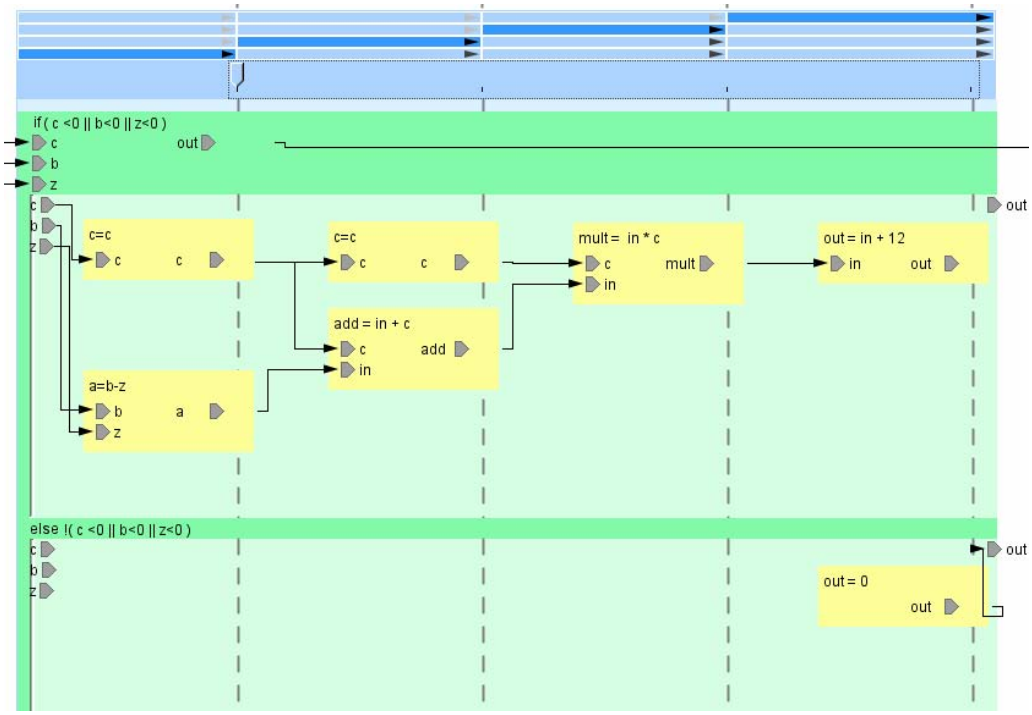


Figure 5.44: Pipelined if else

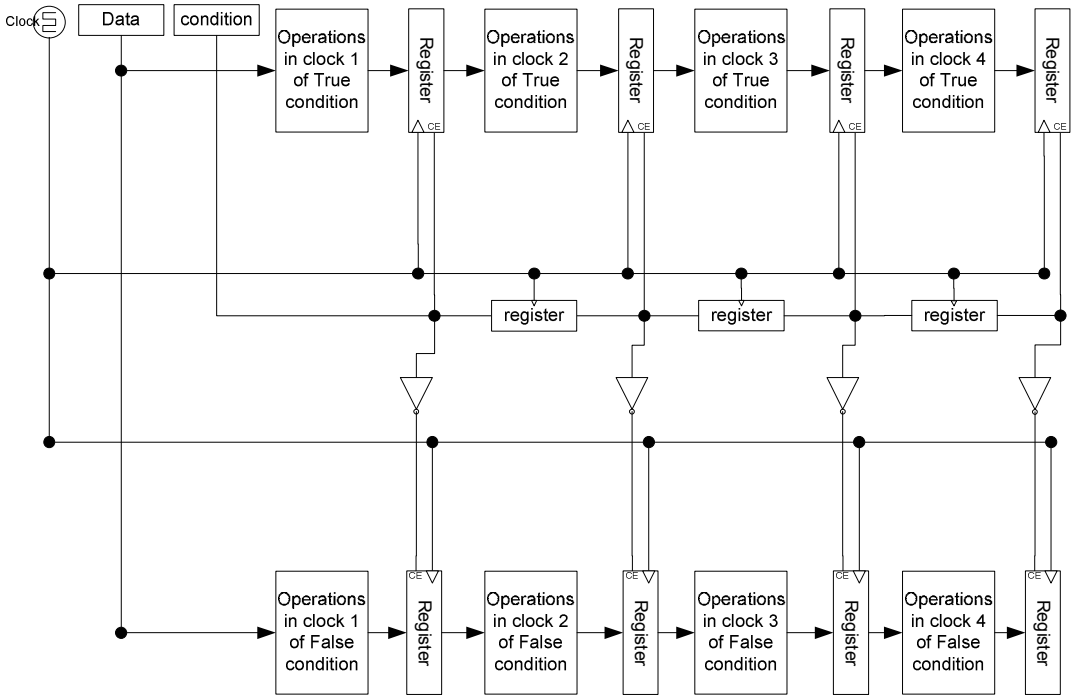


Figure 5.45: Control of pipelined if else

This can also be made a two phase pipeline, Figure 5.46. The state machine to control this is the same as the single phase operation but the clock runs at twice the data rate. As this is now a two phase design the first C register can be removed from the design.

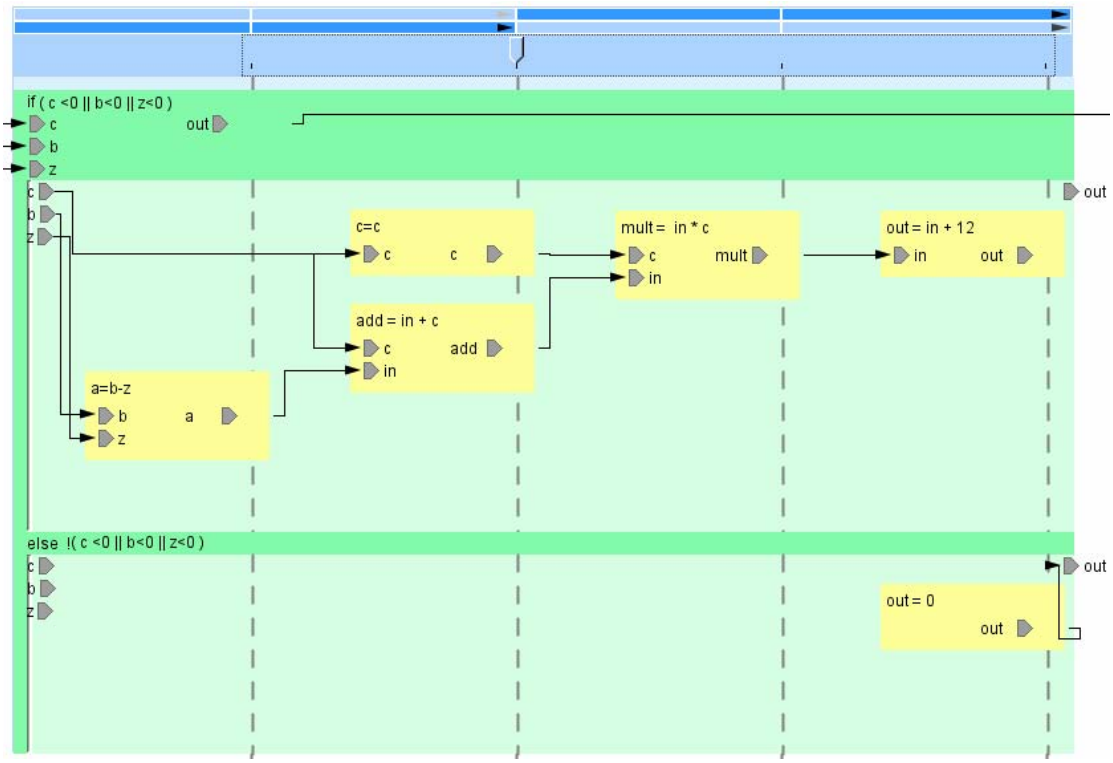
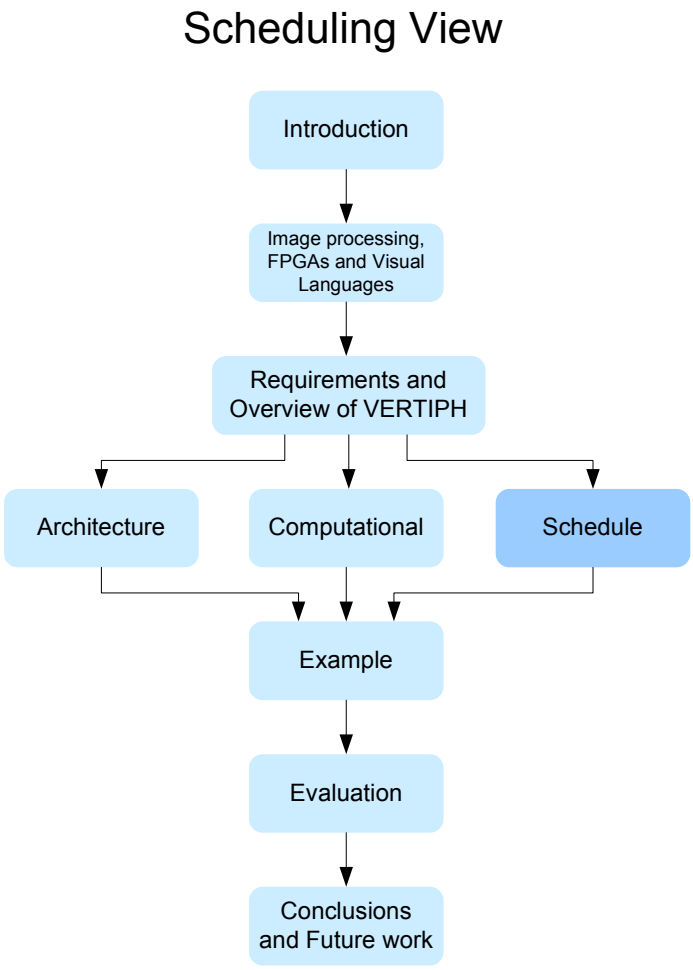


Figure 5.46: Two phase pipelined if else

5.7 Summary

This chapter has looked at the design of the computational view. It has concentrated on pipeline representation. The separation of pipelining from parallel operations and the representation which this section has developed are the main novel aspect of this thesis. The pipeline notation can aid in understanding of the design particularly when a pipeline has stages with multiple parallel operations, reducing the need for secondary notations. The notation also allows multiphase designs to be more easily described and modified.

This section has also discussed expression editing, number representation, the design of control structures and filter editor and associated complications.


Scheduling View
Introduction
Image processing, FPGAs and Visual Languages
Requirements and Overview of VERTIPH
Architecture
Computational
Schedule
Example
Evaluation
Conclusions and Future work

6 Scheduling View

(This chapter is an expansion and extension of the work presented in (Johnston et al., 2008)).

As discussed in earlier chapters, in a complex image processing system implemented on an FPGA, the processors which make up the system need to be scheduled to run at particular times to avoid resource contention, to ensure correct program execution, and to respond by processing data as it arrives or producing data as it is requested. The scheduling view should also show the resources used by the processors and any conflicts that might occur.

Several approaches can be used to allow processors to run only when required. One approach is to pass control tokens between processors, signalling that they can execute. Celoxica's PixelStreams (Celoxica, 2005) passes tokens with the standard data flow to indicate whether the data received is valid and therefore the processor can run. This approach is common in many visual languages for image processing, including Khoros (Konstantinides and Rasure, 1994), Opshop (Ngan, 1992) and PICSIL (Pearson et al., 1996), where only the data flow is explicit and not the control flow. This data-flow approach to program control, as discussed by (Green and Petre, 1996), can highlight the data flow of a program but this is at the expense of obscuring the control flow from the developer. In some cases (for example lens distortion correction – figure 2 of (Johnston et al., 2004a)) the control flow is different from the data flow.

Another approach is to use a controlling processor to direct other processors to run during well defined "epochs". These epochs can be predefined (based on clock cycle counters) or can be generated by other external or internally generated triggers. This can allow for both source- and sink-driven processes to be scheduled efficiently. This programming model makes the control flow more explicit to the developer.

As discussed in the control section of the computational chapter, the signal processing group at Massey University experience suggests treating control at two levels. The first type of control operates at a low level, choosing program actions depending on data received; for example, selecting an identifying label for a uniformly coloured region depends on pixel values (Johnston et al., 2005a). This approach is used in the computational view. The processing is low-level as the immediate effect of the control flow is local to that pixel and is dependent on the data that arrives.

The second type of control is high-level, and operates globally. In many algorithms, some or all of the processors should only run during certain epochs, with a predefined duration or with external or internal triggers. An algorithm which builds a histogram of pixel values in an image from a video source, as discussed in chapter 2, contains a good example of this type of control. One processor is used to build the frequency histogram by incrementing the tally of the bin associated with the input pixel value. Another processor is used to reset the histogram at the end of the frame before a new image arrives. These two blocks need to be scheduled to run at different times as they access the same memory resource. One runs as valid pixel data is arriving and the other when the frame ends. The processing is high-level control as the processing is driven by events and runs at a time defined by those events. In this case, the events are external but they could equally be events driven by other logic on the device.

6.1 Requirements

Image processing algorithms on FPGAs naturally map to a network of interdependent hardware units (Johnston et al., 2004a, Johnston et al., 2006b, Johnston et al., 2006a). Such a structure is suitable for representation in a visual language, because visual languages can represent networks more clearly than textual languages, which are inherently sequential. This chapter presents the development of the visual representation for high-level control of the network of processors – the processing blocks – from which a hardware algorithm is constructed.

In image processing, it is common to have both source- and sink-driven processing modes; each imposes its own requirements on the visual representation. For example, when data from a camera is streamed through an FPGA, the processors need to be source-driven, as the FPGA hardware has no direct control over the data arrival. The system needs to respond to the data and control events as they arrive. For video data, the events of concern are new pixels, new-line and new-frame events. The event determines which part of the algorithm needs to run; for example, when a new frame is received, registers and tables may need to be reset. That is, the processors in source-driven systems are usually triggered by external events.

Sink-driven processing is also common; a good example of this type of processing occurs when displaying images and other graphics on a screen. In this case, processors need to be scheduled to provide the correct control signals and data to drive the screen. This imposes strict time constraints on the processes, as there is a limited number of clock cycles before the pixels for the next screen need to be

produced. Therefore, the processors in a sink-driven system can be triggered to run at specified times, using internal events.

VERTIPH therefore needs to handle both externally triggered events and internally triggered events. External triggers are used to signal new data arrival and data requests from external devices. Internal triggers are generated by other processors to signal that they have completed their task or that they require new data. The visual language therefore, needs a notation for showing the trigger events that cause processors to start running.

Another important consideration in the design of concurrent real-time systems is contention for on- and off-chip resources such as memory, and access to I/O. In image-processing systems, contention can occur within an operation, where different processors use the same memory, or between different image processing operations, when the processors are sharing data via memory. In synchronous systems, contention is easier to manage, as the designer explicitly controls when processors are scheduled. However, in systems where events are not synchronised with the normal processing, scheduling cannot be used and if more than one processor requires a resource, contention can occur. This cannot be predetermined, so other strategies such as semaphores and channels need to be used. The visual language should therefore, provide mechanisms for the scheduling of processors and processor sub-blocks where possible, and show where potential resource conflicts may occur.

Defining the activation order of processors within an epoch should be facilitated by the visual language. Processors may run in parallel, form a pipeline, or run sequentially. Parallel processors all run at the start of the epoch. When running sequentially, processors need to run after the epoch starts and after the previous processor has finished. A processor within a pipeline runs after the epoch start, and once the previous process is producing valid output.

As discussed in earlier chapters, pipeline processing has the complications of priming, flushing and other control, including stalling. Pipeline priming for image processing operations is often difficult and error prone because of the need to deal with “special cases”. Priming involves inputting data into the pipeline to ensure that it will operate correctly. In image processing there are a number of different priming strategies. Consider the processing of an image assuming that the processing requires a window. For the first line, the pipeline needs to be primed with a line of data either before (if the priming can be achieved without the data, for example padding an image with a constant) or as the initial data arrives. The pipeline then operates as

normal until the end of the line; an edge data pad needs to be added at the end of the line, the pipeline then flushes and stalls for the blanking period. When the next line is reached (or before) the pipeline needs to be primed with an edge pixel. It is these different special cases and the need to often prime before data arrives, or to buffer data so the priming can be done that makes the design of such systems difficult, and the complexity can lead to errors. Flushing can similarly be difficult due to special cases and the need to keep the pipeline running until all the data has been processed. This may be after the transition to a new epoch has occurred.

To summarize, the graphical representation for control constructs in such an environment should:

- allow a mixture of sequential and event driven control of processors
- allow processors and processor sub-blocks to be scheduled
- show where resource conflicts may occur
- schedule processors from the same epoch to run sequentially, in parallel or as a pipeline
- simplify the design of pipeline priming, flushing and control.

The rest of this chapter will evaluate several possible solutions. Finally, a representation is selected that is effectively a trade-off of several competing features.

6.2 Graphical Representation

The first graphical representation (the linear epoch based model) represents segments of the algorithm as a linear sequence of epochs. A synchronous scheduler is responsible for issuing trigger events to start the epochs. The epochs run for a fixed time, based on the run times required by the processors they contain and then the controller generates the trigger for the next epoch. The designer specifies the resources required by the processors in an epoch, which allows the controller to schedule the processors so that they do not conflict for resources.

In this representation, processors can be scheduled to run concurrently or sequentially within an epoch. This allows independent processors to run freely within an epoch and allows developers to move resource-conflicted processors. Processors can span epochs and run in more than one epoch.

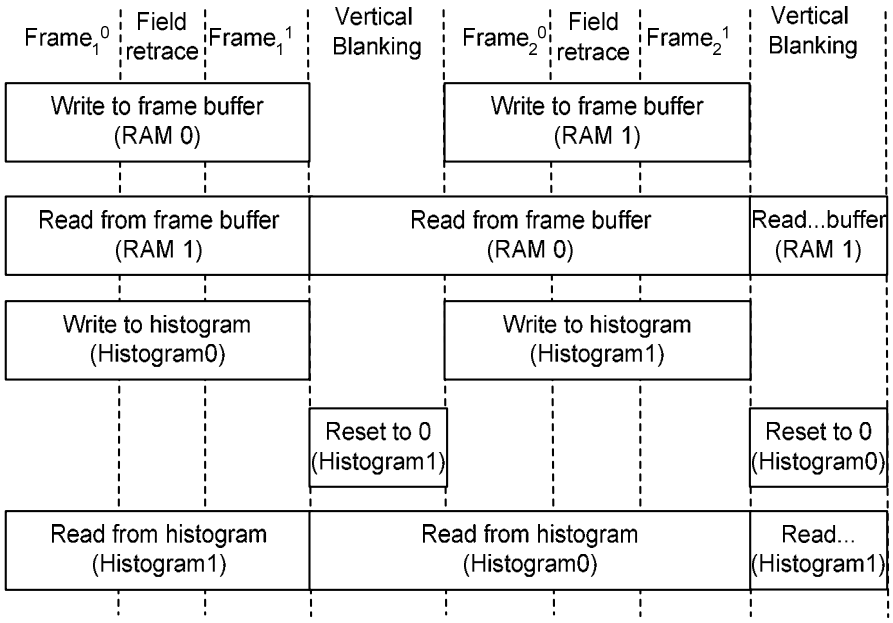


Figure 6.1: Linear epoch-based graphical representation for a histogram algorithm

Figure 6.1 is an example of the proposed visual syntax to support the linear epoch-based model. In this example five different processors (*Write to frame-buffer*, *Read from frame-buffer*, *Write to histogram*, *Reset to 0* and *Read from histogram*) operate on four resources (RAM_0 and RAM_1, $Histogram_0$ and $Histogram_1$) used in the histogram algorithm detailed in chapter 2. In this algorithm there is a frame buffer using two RAM blocks; these are used alternately by two processors, one writing the image to memory and one reading from the memory. Three processors require access to two histogram buffers. Two processors write to the histograms; one resets them to zero and the other builds the histogram. The final processor reads the histogram information and processes it for other processors to use.

The horizontal direction represents time (in epochs) moving from left to right. Each epoch is represented by a dotted vertical line and the label at the top shows the epoch name. Processors are represented by rectangular blocks. Each block is labelled with the processor's name and the resource (or resources), in brackets, that it requires. Processors are laid out in a grid, with processors overlapping the epochs they run in. The vertical direction has no particular significance except that it serves as a way of separating diagram components graphically.

This epoch-based graphical representation is a manifold; it should really wrap around to connect the epochs, because the operations at the start of the processing (those active in the first epoch) are reactivated at the end for the final epoch when the first epoch becomes active again. In this example, two of the three processors which

occur in the final vertical blanking epoch continue to run during the next epoch ($Frame_1^0$). In order to produce a more intuitive representation of the repetitive nature of epochs, we can make the epoch display circular. This notation, illustrated in Figure 6.2, more clearly shows that the processors repeat forever. Transitions from one epoch to the next are triggered by the events, with the epoch labelled in the centre of the diagram. Dotted lines radiating from the centre extend the epoch area to the processors. Processors are layered around the centre as overlapping sectors. In this representation, rather than using brackets to indicate resources, a darker grey background and white text is used to label the processor's required resources. Time progresses in a clockwise direction and, as in the first view, it is in epochs.

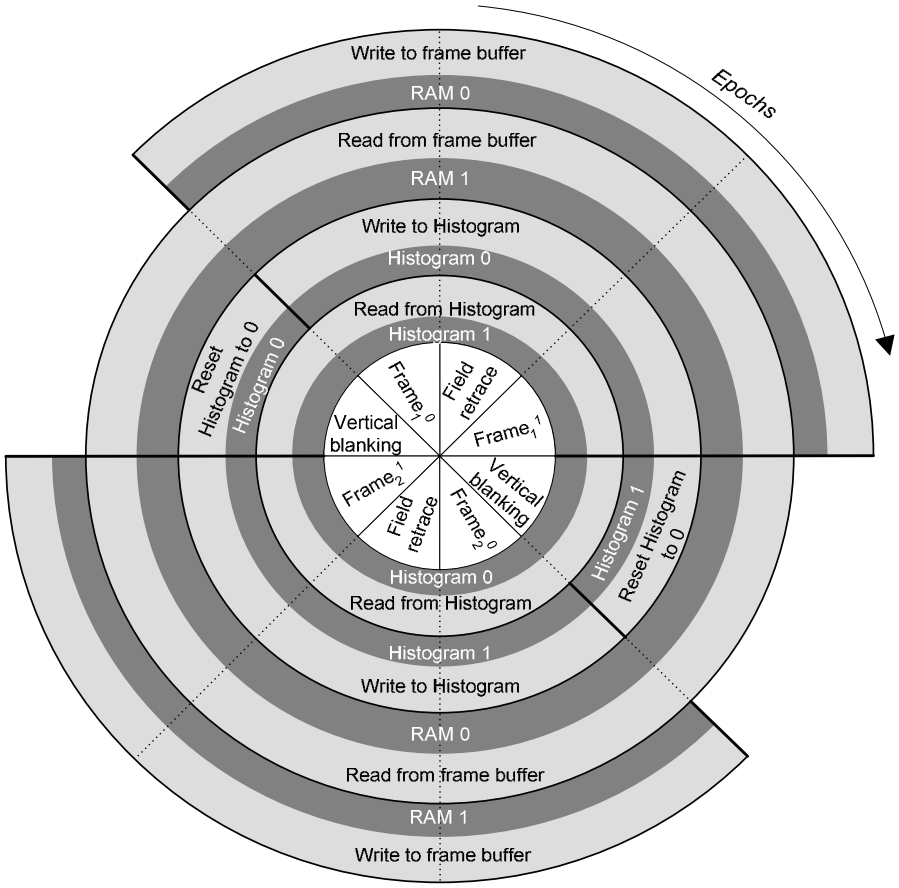


Figure 6.2: Circular Epoch-based representation

There are several limitations to this view. Informal user tests suggested that designers may have some difficulty with reading and understanding the notation. Also as the degree of concurrency increases, a new ring is required and each additional ring of processors uses more screen real estate. The processor's screen area is therefore not proportional to the processing required, but its size depends on when it was added to the representation.

These two views clearly show when processors are running simultaneously and make it clear what resources the processors use. However, both views require events to be synchronous. They also do not allow a mixture of sequential and event driven control of processors and the mixing of source and sink driven processing.

These two views cannot support situations where there is more than one possible exit from an epoch. This produces a non-linear, networked control structure which is better represented as a directed graph than the linear representations used in the two previous views. Graph-based notations allow for many connections between states to be represented.

There are many different visual representations of mathematical graphs. This work developed a representation based on state charts (or state machine diagrams) (Harel, 1987), which are familiar to the engineering and computer science community and have been found to be useful to describe event-driven embedded systems (Samek, 2002).

The goal is to be able to schedule processors that operate in the same epoch to run sequentially, in parallel or as a pipeline. The processors will already be connected in the architecture or computational node view. This information can be used to automatically put the processors in the right order; this allows the user to select whether they are run as a pipeline or sequentially. A standard state chart only uses the present active state to select processors to run. This requires the extension of the state machine concept to allow for the scheduling of processors within a state, without the state having to include a full state machine to do the scheduling. This would lead to a low-level control structure similar to that which can be created with the low-level control structures already offered by the computational view and would not add any benefit.

In a notation based on standard state charts, each state would correspond to an epoch, and process activation would occur as a result of the state machine reaching a particular state. Control in an FPGA environment is more complex than this, so we extended the notation to make it possible to schedule an epoch's processors to run sequentially, in parallel, or as a pipeline.

To represent this the state chart notation has been extended with a modified version of Gantt charts based on those described in (Johnston et al., 2006a). Figure 6.3 shows the Gantt chart component of the notation; time progresses from left to right, sequential processors (i), are drawn from left to right on a single line, and parallel processors (ii), are stacked vertically. As a pipeline comprises a combination

of sequential and parallel processing the pipeline is drawn, labelled (iii), using the same notation as in the computational view. Unlike the computational view, where the boundaries take one clock cycle, the data processing times will vary depending on the each block's latency (or processing time).

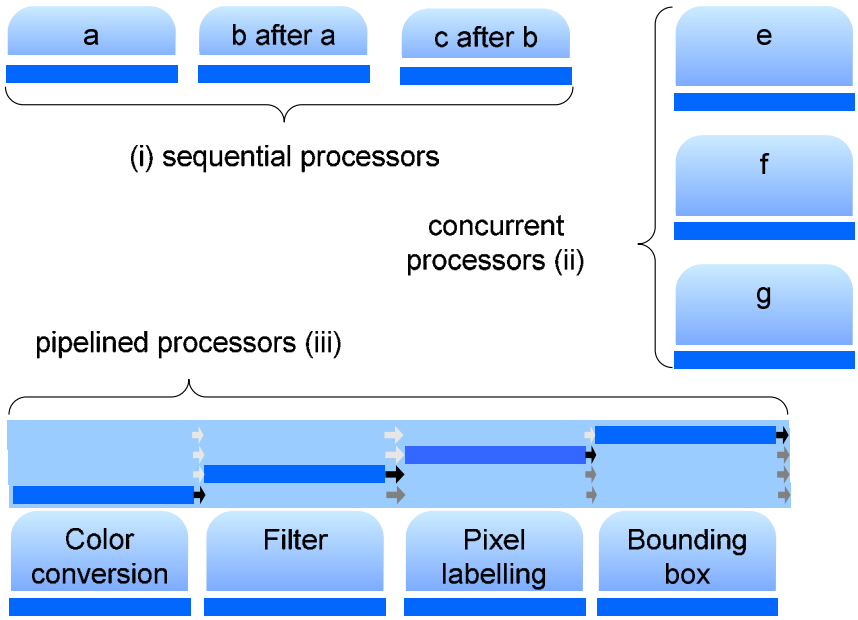


Figure 6.3: The three process representations: (i) Sequential, (ii) Parallel, (iii) Pipelined

Processes are separated into a dark horizontal bar and a lighter coloured "flag" that contains its name. This flag can be hidden when not required, saving on screen real estate. The resulting diagram clearly illustrates pipeline priming and flushing phases, and the resources that processors require are represented as flags in a different colour and under the processor bar (see Figure 6.4). Figure 6.4 shows a representation of the histogram example using a combination of this Gantt and the standard state-machine notations. In this example, two state machines are shown: one for the main algorithm and another for controlling an encapsulated resource (discussed later). When a state is selected in the editor, the internal processors are shown in a split window. In the lower panel, processors can be added to a state, and their execution order can be edited. This makes it possible to implement complex scheduling if required. Hierarchical state machines can be constructed by adding another state machine inside a state. This allows for the creation of very complex state transitions.

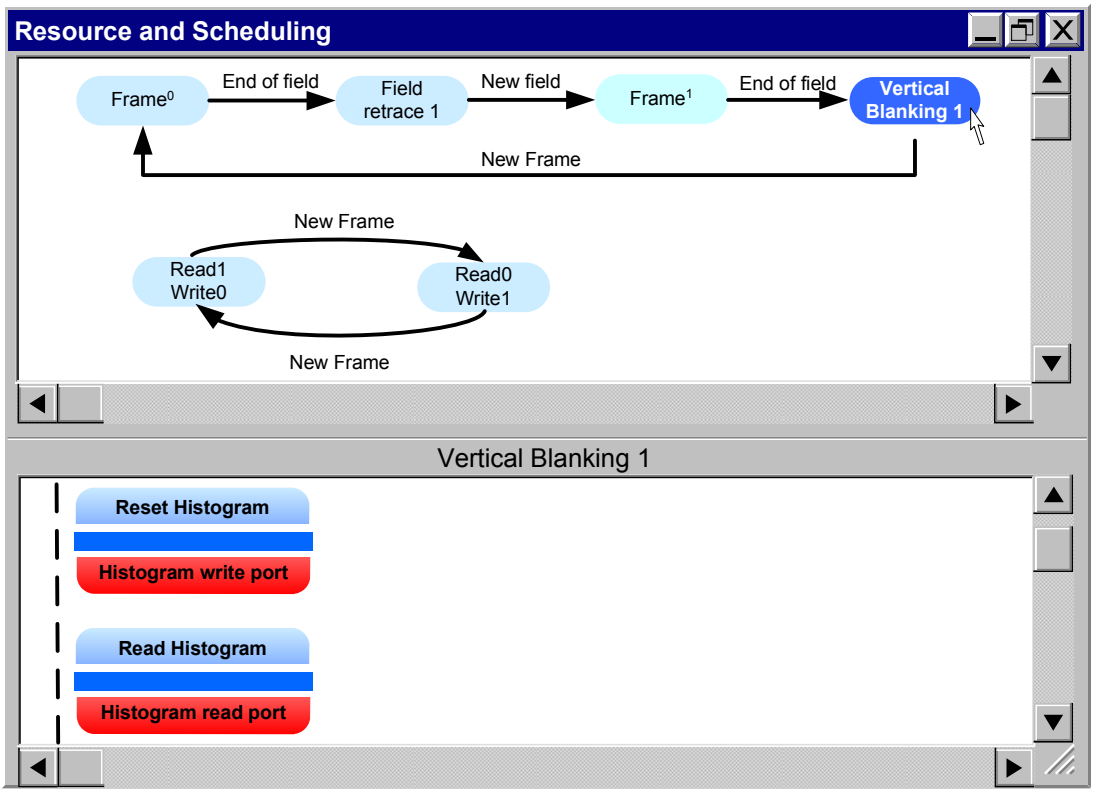


Figure 6.4: Extended state machine editor, Vertical Blanking selected. The three concurrent processors associated with the state are shown in the lower split screen

State transitions are triggered either by external events or by triggers from other states. For a state machine driving another processor or an external device such as a screen, it is common for a state that has been triggered to run for a predetermined time and then trigger the next state. Therefore, states can be both triggered and triggering. All states will have a triggered property which will cause a transition to this state. States may also have a triggering property that causes them to transit to other states either after a set number of clock cycles, or after all the state's processors are finished, or by events internal to the state's processors.

A state-based representation does not fulfil all of the requirements. The combined Gantt-state notation provides multiple state machines and asynchronous event-driven processing, but it hides resource conflicts. This is a trade-off, making one type of information, the scheduling, more visible to the developer, while reducing the visibility of the other, resource usage. The linear and circular epoch views made this resource contention explicit, but they relied on synchronous processors. It is possible to detect resource conflicts occurring within the state as these processors are scheduled and run in the same clock domain. However, as we allow multiple asynchronous state machines, a search must be made to check if processors in other

states might also need the resource. It is possible for two or more asynchronous states to require a resource simultaneously. Resolution of such a situation is left to the developer; therefore the developer needs to be notified. If processors that appear to be competing for resources cannot be scheduled to run at different times, then explicit resource-sharing mechanisms, such as semaphores and channels, should be used.

In concurrent systems, deadlock can occur if two or more of the processors cannot move to another state. The four conditions for deadlock are (Coffman et al., 1971):

1. that the processors involved need exclusive access to the same resource;
2. that processors hold onto resources while awaiting the granting of additional ones that are being held onto by other processors;
3. that the system cannot preempt resources once they have been allocated;
4. that a circular chain of requests and allocations exists.

Within a single state machine, deadlock cannot occur, as only one state is active and processors within a state are scheduled so that processors which require mutual exclusion cannot run at the same time.

However, in multiple state machine configurations, deadlock can occur as processors from different state machines can compete for resources. By performing a search of the resources used by processors in the different state machines we can notify the developer in this case. When multiple resources are required, it is possible to avoid both the second and fourth condition of deadlock by requesting all required resources simultaneously before a state can start. If they cannot all be allocated, none are allocated. This can result in poor resource utilization, as a state may not require a resource for the whole period (Burns and Davies, 1993). Blocking a processor from running to satisfy resource contention issues can cause a real-time image processing application to fail. As failure is not desirable, the developer will need to redesign the algorithm to avoid this (use a different algorithm, allocate more resources, prioritise access, etc.)The IDE should therefore alert the developer when the above condition might occur so that she or he can take any necessary action.

Many of these conflicts can be simplified by encapsulating a set of resources with a manager. The manager then uses a separate state machine to control access to the resource through predefined ports. Use of such managers can simplify the overall system design. When the resource is used by multiple asynchronous processors, the

arbitration mechanism may be incorporated within the manager to allow more efficient use of shared resources. Resource managers can be illustrated by a simple example. A frame buffer requires two or more memory banks to allow data to be written to memory by one process while another process reads it, without corrupting the data due to overwrites or not allowing access due to memory bandwidth constraints. The sharing of the resource between the two processors needs to be managed to provide a consistent interface to the resource for the processors. In this simple case, the sharing can be achieved with suitable *if-else* logic, but for more complex sharing or resource configurations, a state machine can be used to control the sharing of resources.

When the sub-processors which form a pipeline are examined, their individual priming lengths can be determined. The sum of these individual lengths is the time necessary for priming the pipeline to provide the output data at the correct time. When the processors are driving a source, a state can suggest to the developer how to modify its start condition to run early. Unfortunately, this is not an option when data is requested asynchronously by a sink. In this case, the designer can set a property called run-to-prime, which forces the state to run as data is produced and then stall in a primed state until the data is requested. Neither of these strategies can be used when the processors are source-driven, as the pipeline cannot be primed until data arrives from the source.

For flushing, once a state's exit conditions have been met, the associated processors need to keep running while valid data remains. This is relatively simple in cases when there are no resource contention issues, as the state can run for the number of clock cycles to flush the pipeline while the other state begins processing. However, the situation is more complicated when the states share the same resources. In this case, the preceding state needs to either delay the transition to the new state until the pipeline has been flushed, or exit with data still in the pipeline. This is a design decision and should be a state-based option that the designer can select according to the algorithm's requirements. The pipeline notation, Figure 6.3 (c), can aid the designer by showing the number of processor epochs required to flush the system and the developer can select the best option.

These different pipelining options are represented in Figure 6.5. Option (a) shows a processor which needs to be primed before running and will pause until the state's start event is triggered; (b) shows a pipeline which will continue to run after the state has transited until it has flushed; (c) shows a pipeline that will delay the

transition to a new state until it has flushed. Figure 6.7 shows an example which uses both priming and flush after transition.

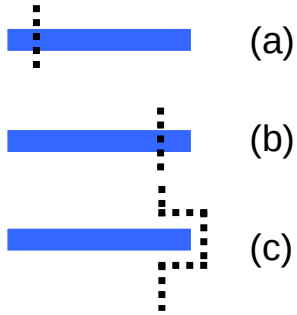


Figure 6.5: Three pipeline controls

(a) prime, (b) keep running to flush, (c) flush before state transition

The state-based notation allows the developer to get a quick overview of the high-level control states and the trigger events which cause their transitions. Multiple state machines can be displayed on the screen with the developer able to see more detail about the processors running in a state by selecting it. This allows the developer to concentrate on the controller design while hiding the details of what is running in those states. This can help to reduce information overload. Visual programming languages have advantages over textual languages in this respect, as in text the separation of controller and action normally does not allow for linkages to be made between the two.

"Processors" are either architectural blocks or computational processing blocks. The scheduling of an architectural block will control when all the operations within that block will run, so it is more appropriate in most cases to schedule the computational blocks which make up the operations. The blocks which are scheduled need to be identified with a control notation, either in the architectural or computational node view. To do this a new control structure, *when,* has been added. This is like the *if* control structure, but is controlled by the state machine. That is, the processor will run when the controlling state is active, or for more complex control, when state X and state Y (etc) are active. This control will be shown on the controlled node (Figure 6.6 which has two architecture blocks controlled by different states) and as a flag inside the controlled node's computational expression. By clicking on the control within the other views, the developer will be taken to the controlling state (or state machine if controlled by more than one state).

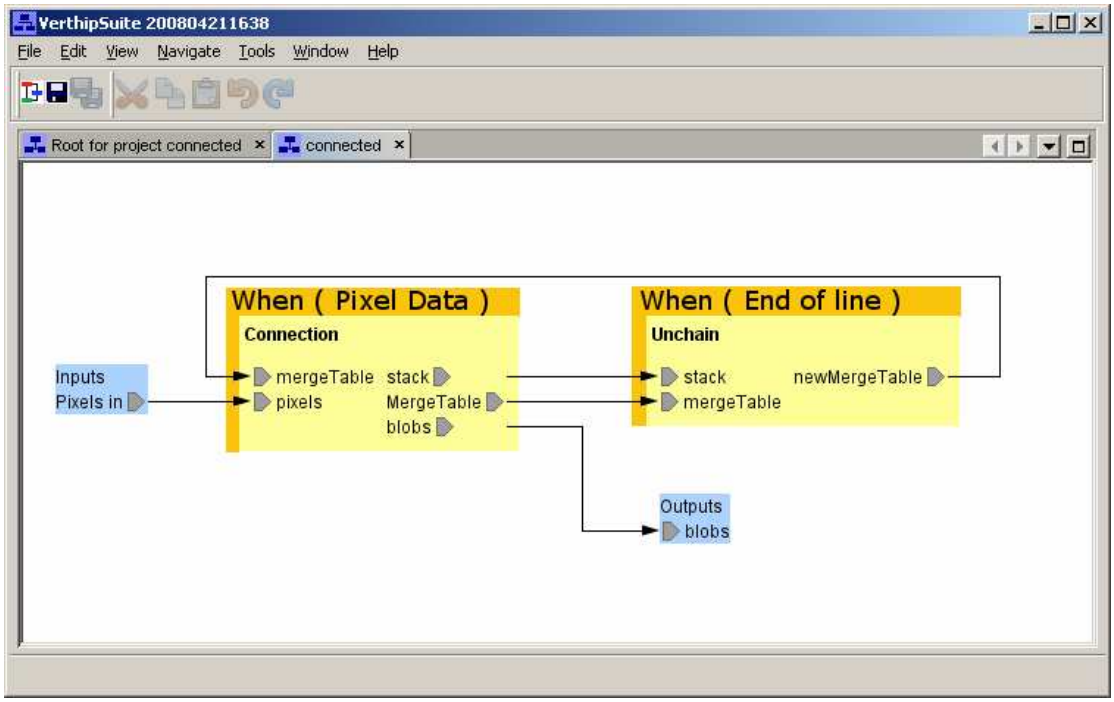


Figure 6.6: *When*s **controlling architecture nodes in connected components analysis**

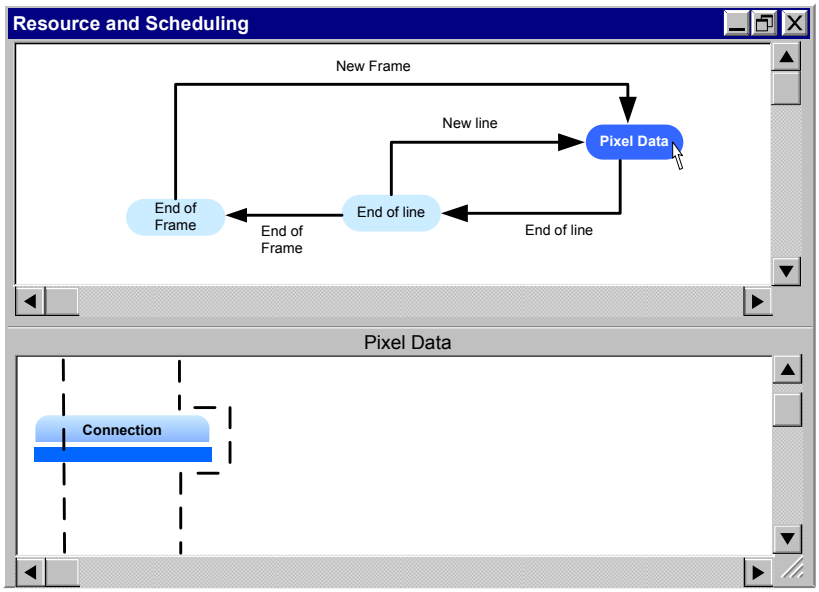


Figure 6.7: State machine, showing *when* for Connection

Control logic is required to handle the boundary cases (when data starts or stops). In stream-based image processing, these typically occur at the start and end of each row of data, and at the start and end of each frame. *When*s, and the state machine conditions, can be used to handle these cases. In Figure 6.7 the *when* controlling the main pipeline (connection block) is shown. This control has been

scheduled to prime and then wait for data and to continue running to flush the buffer before exit from state due to a transition event. In this example, the framing of the data can be automated as the data is requested in a regular pattern that can be predicted. Flushing before a transition is required as the merger table and stack need to be complete before the unchain operation can run.

6.3 Summary

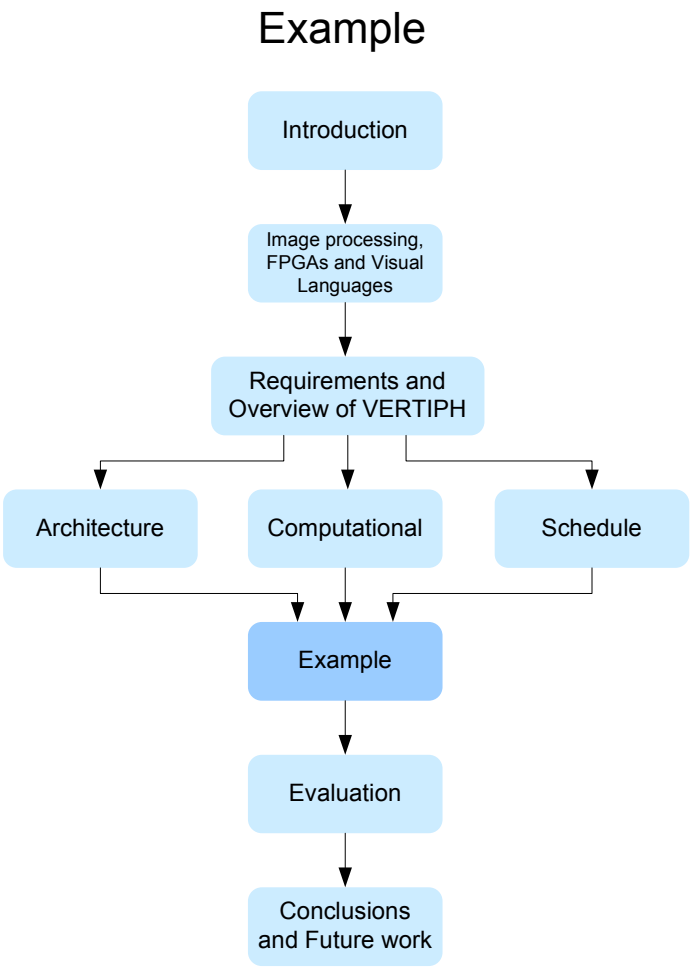
This chapter has evaluated several different graphical representations for high level control of image processing algorithms on FPGAs. We identified five requirements that any representation should have. It should

- allow a mixture of sequential and event-driven control of processors
- allow processors and processor sub-blocks to be scheduled
- show where resources conflicts may occur
- schedule processors from the same epoch to run sequentially, in parallel or as a pipeline.
- aid in the priming, flushing and control of pipelines.

The first two epoch-based views allowed processors to be scheduled and were very good at showing resource conflicts. They were both based on synchronous control but could not be extended to incorporate event driven asynchronous control. This limitation led to a combined Gantt chart/state machine diagram notation, which has the advantage of allowing for asynchronous event-driven processor scheduling. To accommodate the need for processors to run sequentially, to run in parallel or to form pipelines, basic state chart concept has been extended with an internal state scheduling view. The states allow for some limited automation of the control of pipelines.

This view is good at illustrating and avoiding resource conflicts within a state, but it does not show resource conflicts which may occur between multiple concurrent state machines. This means that to detect possible deadlocks, the editor needs to check what resources the other states are using and evaluate where conflicts may occur. The system attempts to prevent deadlock conditions by requiring states to request all resources concurrently at the start of the state; if not all are available, then the transition to that state is blocked until the resources are available. This is not an efficient use of resources, and the developer needs to be informed when this may possibly occur.

The proposed representation illustrated in Figure 6.3 and Figure 6.4 meets all of the requirements that were identified for the representation of image processing algorithms on FPGAs. It allows both sequential and event driven control of processors. It allows processors and processor sub-blocks to be scheduled. It can display where resource conflicts may occur within a state and, with editor support, it can find where processors belonging to different states may conflict. The resources used by processors are not as clear in the chosen solution as the first two views and further work is needed to make this more explicit, and to automatically detect potential resource conflicts. Processors within a state can be scheduled to run sequentially, in parallel or as part of a pipeline. The design has added controls to aid in pipeline priming, flushing and control that are often needed. This design needs to be implemented and will probably require further refinement as the interaction between this view and the other views become apparent with use.


Example
Introduction
Image processing, FPGAs and Visual Languages
Requirements and Overview of VERTIPH
Architecture
Computational
Schedule
Example
Evaluation
Conclusions and Future work

7 Example

This section will look at an implementation of the barrel distortion correction algorithm described in the second chapter. Barrel distortion occurs when the magnification of the lens is not uniform across its field of view and is usually associated with wide angle lenses. The characteristic barrel shape results because the magnification at the centre of the lens is greater than at the edges. The algorithm illustrated in this chapter uses a reverse map (Wolberg, 1990), where the address in the distorted input image is determined from the address in the corrected output image. The output pixel value is determined by selecting the nearest neighbour. The design for the key parts of the algorithm will be shown in VERTIPH.

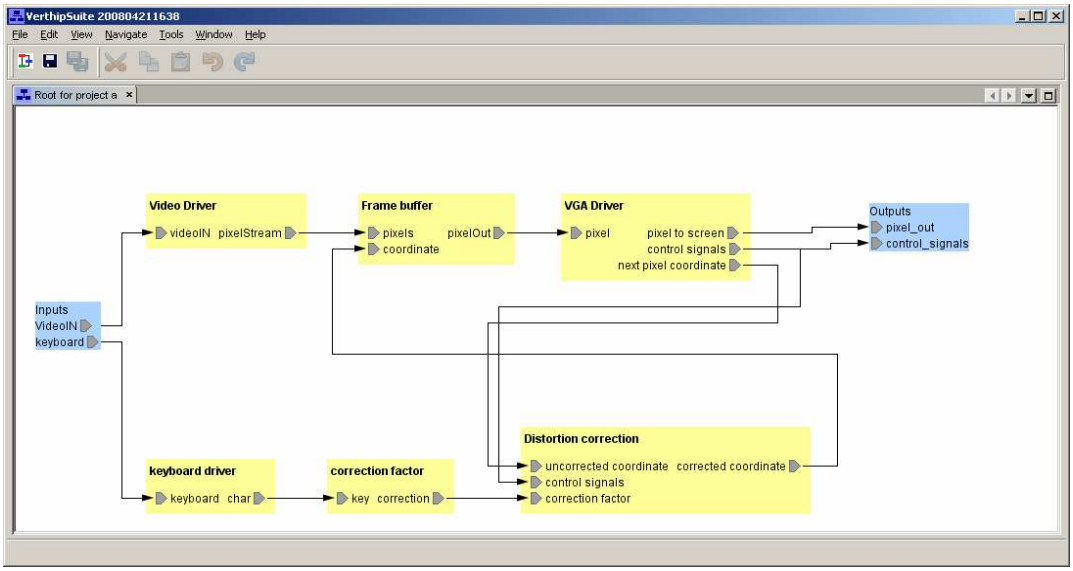


Figure 7.1: Root architectural view for Barrel distortion

The design can be decomposed into the six main architecture blocks shown in Figure 7.1: an input video driver, a frame buffer to hold the input image, a video output controller to display the corrected image, a keyboard driver, a correction control and a distortion correction block. This top-down decomposition allows each of the device's major functions to be treated as an independent design task, with an associated functional block.

The video driver handles the communication between the video chip and the FPGA, producing pixels for an off-chip frame buffer. It needs to handle the pixel data and control the buffering of it from the input to output clock domains before transferring it to the frame buffer.

The frame buffer module handles the storing of the pixels into the off-chip RAM used for the frame buffer. This module also produces the output pixel based on

the requested pixel address. This means the module has two inputs, *videoIN*, a complex data type containing the pixel, its coordinate and control signals and secondly the *coordinate* which is the address of the next pixel to be output. This block acts as a resource manager for the frame buffer.

The VGA driver produces the control and data signal for driving the display; it also produces the next pixel and the control signals for the distortion correction section. The algorithm is display driven, because output pixels must be produced as they are needed for the display.

A keyboard architecture block is added for communicating with the keyboard. Characters from the keyboard are used to adjust the correction factor used by the distortion correction block.

The distortion correction block generates the address of the source pixel in the distorted image stored in the frame buffer, using the correction factor, control signals from the display driver and the address of the pixel required by the display.

A number of different data types are required for the design. *VideoIN* needs to be connected to several external pins which are a mixture of control and data lines. *PixelStream* contains complex pixel, coordinate and control information. The *keyboard* input needs to have a clock and data line input. This is then converted to characters which are used to update the 10 bit correction factor (Figure 7.2). The pixel type is a complex type with red, green and blue values. The control signals from the VGA driver are shown in Figure 7.3 below with a one-bit flags for visible region, horizontal blanking or vertical blanking. The coordinate type (Figure 7.4) is also a complex type which has 10-bit types for each of the x and y positions.

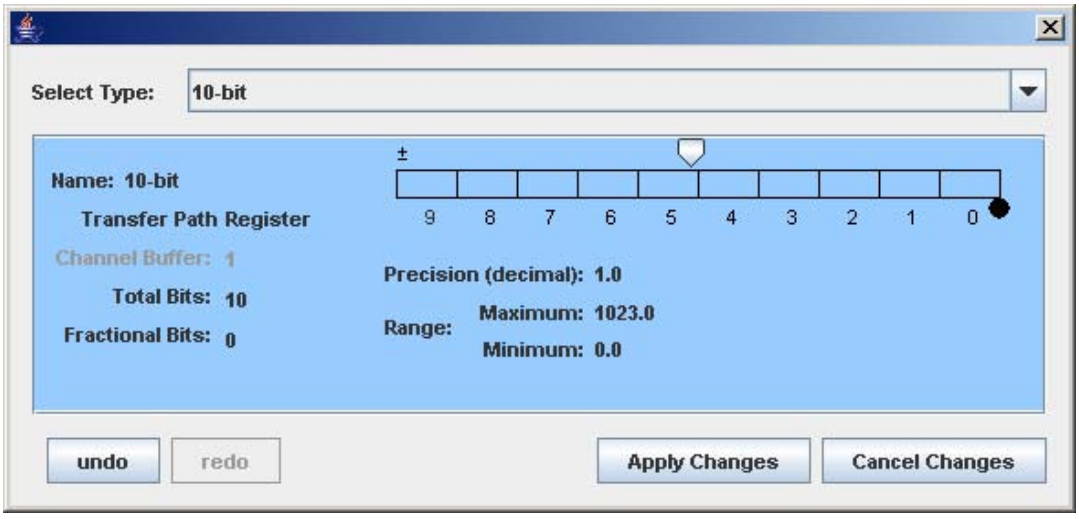


Figure 7.2: a generic 10-bit type used for correction factor and other internal variables.

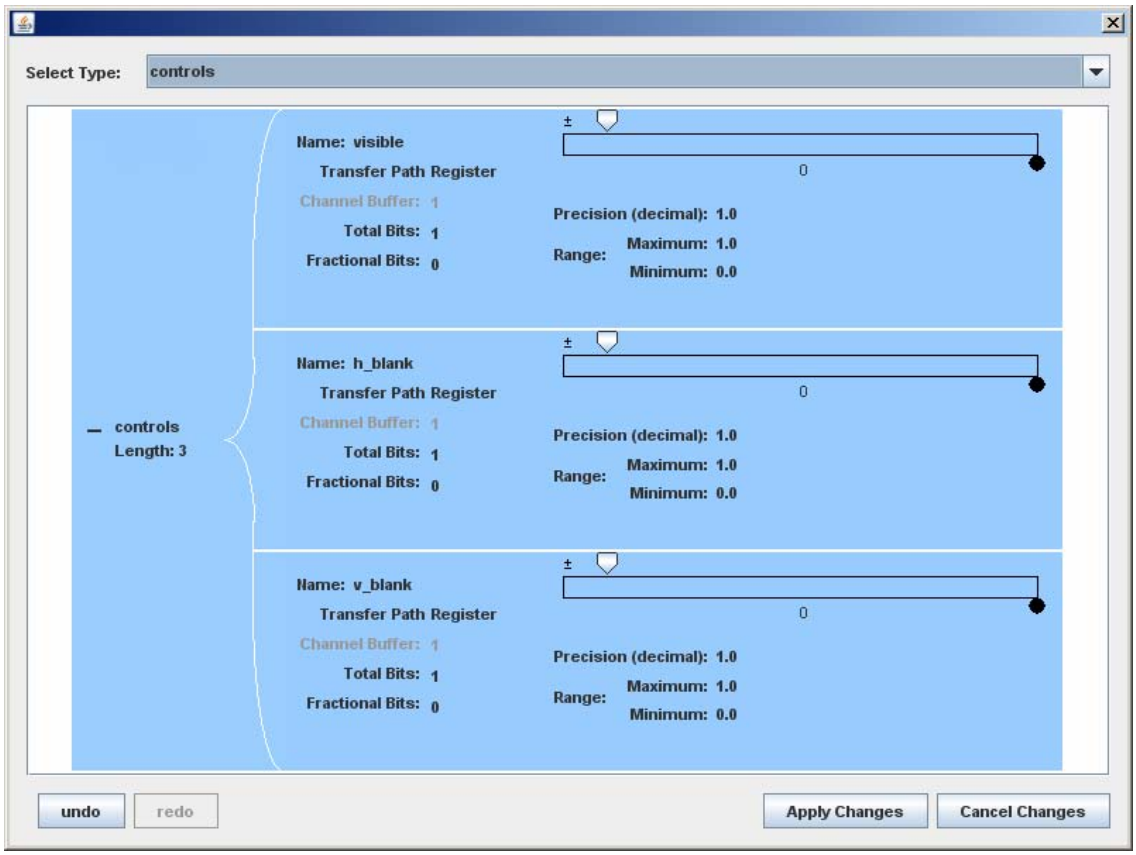


Figure 7.3: Control data type

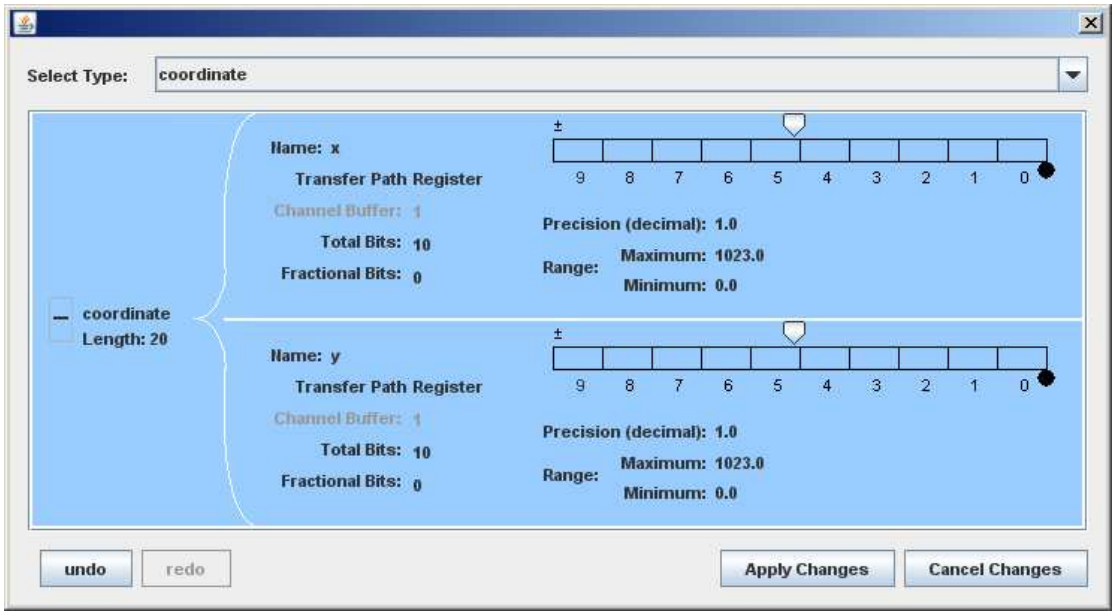


Figure 7.4:Coordinate type

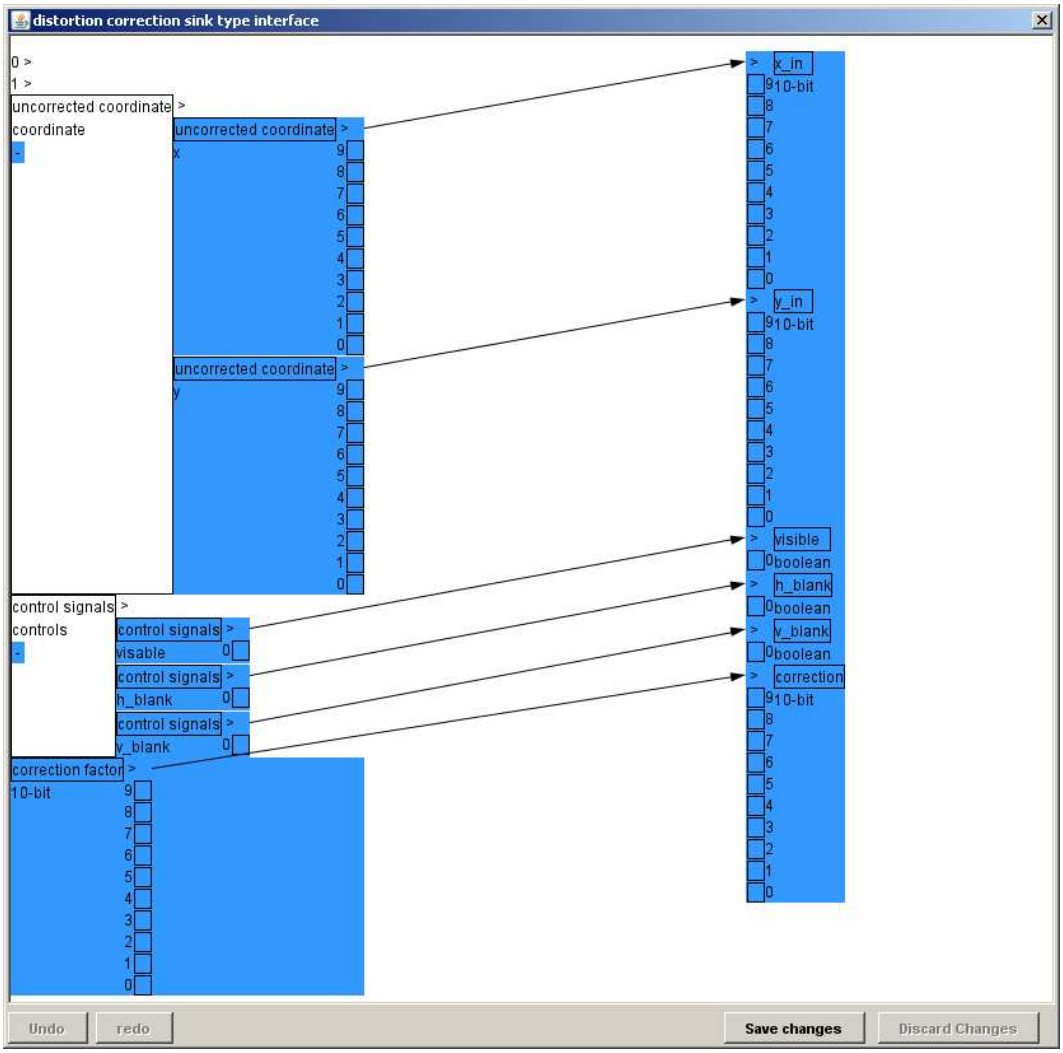


Figure 7.5: Input junction box for distortion correction block

The distortion correction block has the input junction box (Figure 7.5) which breaks out several of the complex data types into basic types. This allows the individual elements which are used by different parts of the computational block to be accessed individually. The coordinate type is separated into its 10-bit x and y components. The control signal is also broken up into component flags, and the correction factor is passed unchanged through the junction box. The output junction box (Figure 7.6) combines the x and y out into one coordinate type for ease of routing in the architecture view.

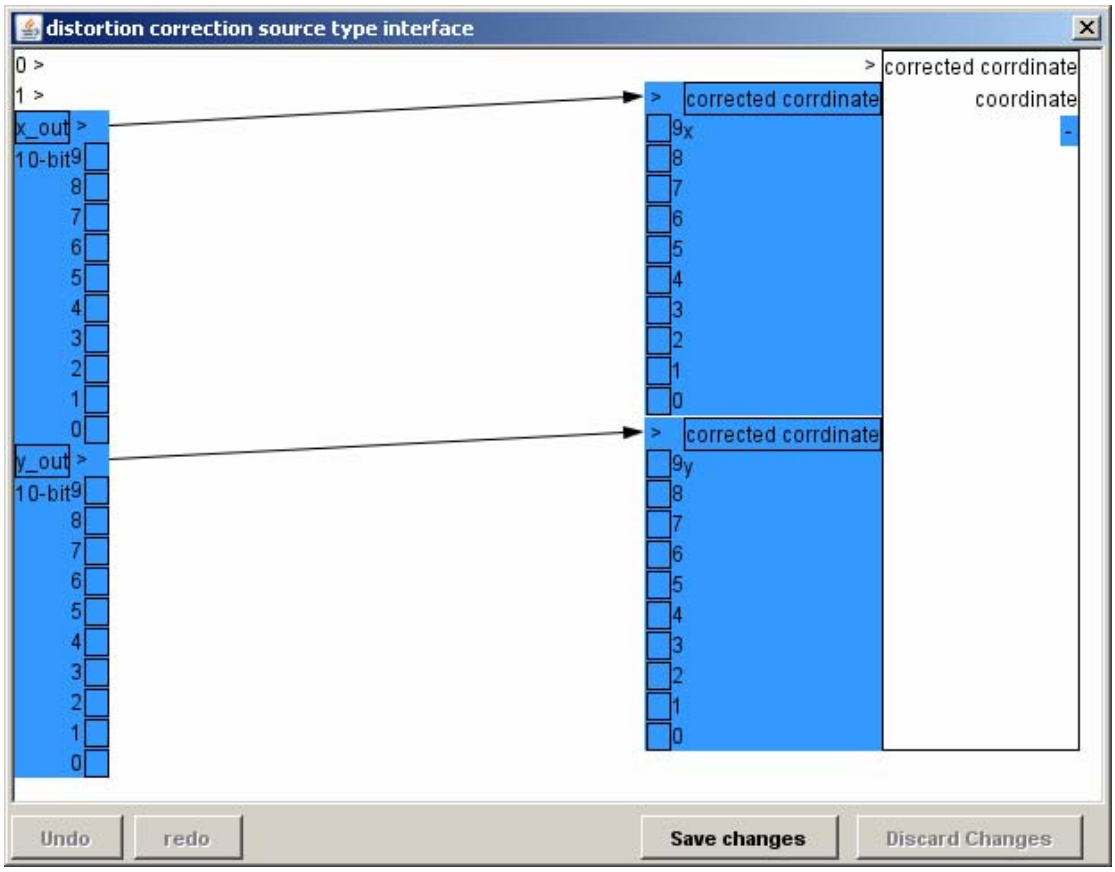


Figure 7.6: Output junction box for distortion correction block

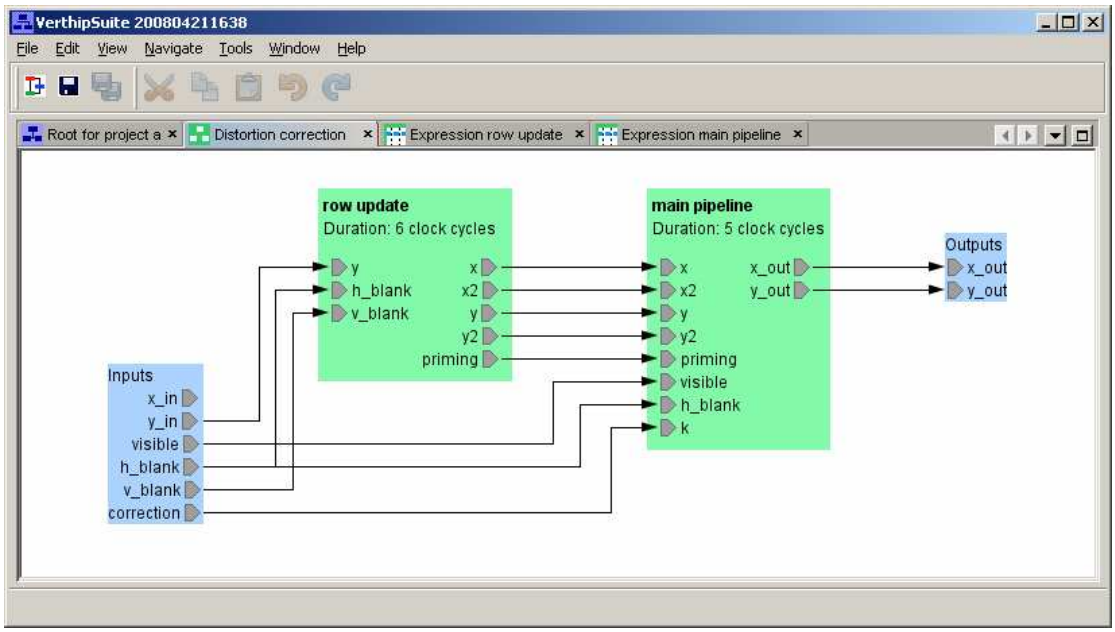


Figure 7.7: Distortion correction computational node view

The distortion correction block in the architecture view contains the computational node view, shown in Figure 7.7. This node view contains two separate

components which operate independently. There is a block to update y^2 on each line and prime the pipeline (row update) and the main pipeline for correcting for the distortion. As illustrated on the node by the duration indicator, the main pipeline takes 5 clock cycles for data to exit and the y update takes 6 clock cycles to update and prime the pipeline. y^2 needs to be updated when the horizontal blanking occurs (once a line) otherwise y^2 stays the same, x and x^2 need to be reset and then the main pipeline can run for 5 clock cycles to prime. When the vertical blanking is reached y also needs to be reset. A possible approach to accomplishing this is shown in Figure 7.8.

If the scheduling view was used to control the main pipeline block then the priming and flushing of the 5-stage pipeline shown Figure 7.9 could be controlled by a *when*. Instead, an *if-else* control is used, with another two *if-else* blocks used to control the flushing and priming. The condition for the main pipeline will be true if processing visible pixels; it will also run if the priming or flushing conditions are met. Notice the warnings about inputs being used several clock cycles after their output (the orange wire with the ! mark). In the design the warnings for y, y^2 and k can be ignored as they are not updated each clock cycle. However, x is updated each clock cycle and is used 4 clock cycles later which is why 4 is subtracted from the $X_{corrected}$ equation. Future versions of VERTIPH will need to allow developers to suppress such warnings if they consider them to be too conservative.

The priming is achieved as a result of the row function (Figure 7.8). Once x and x^2 have been updated the loop will run for 5 clock cycles which will make the main function run (Figure 7.9).

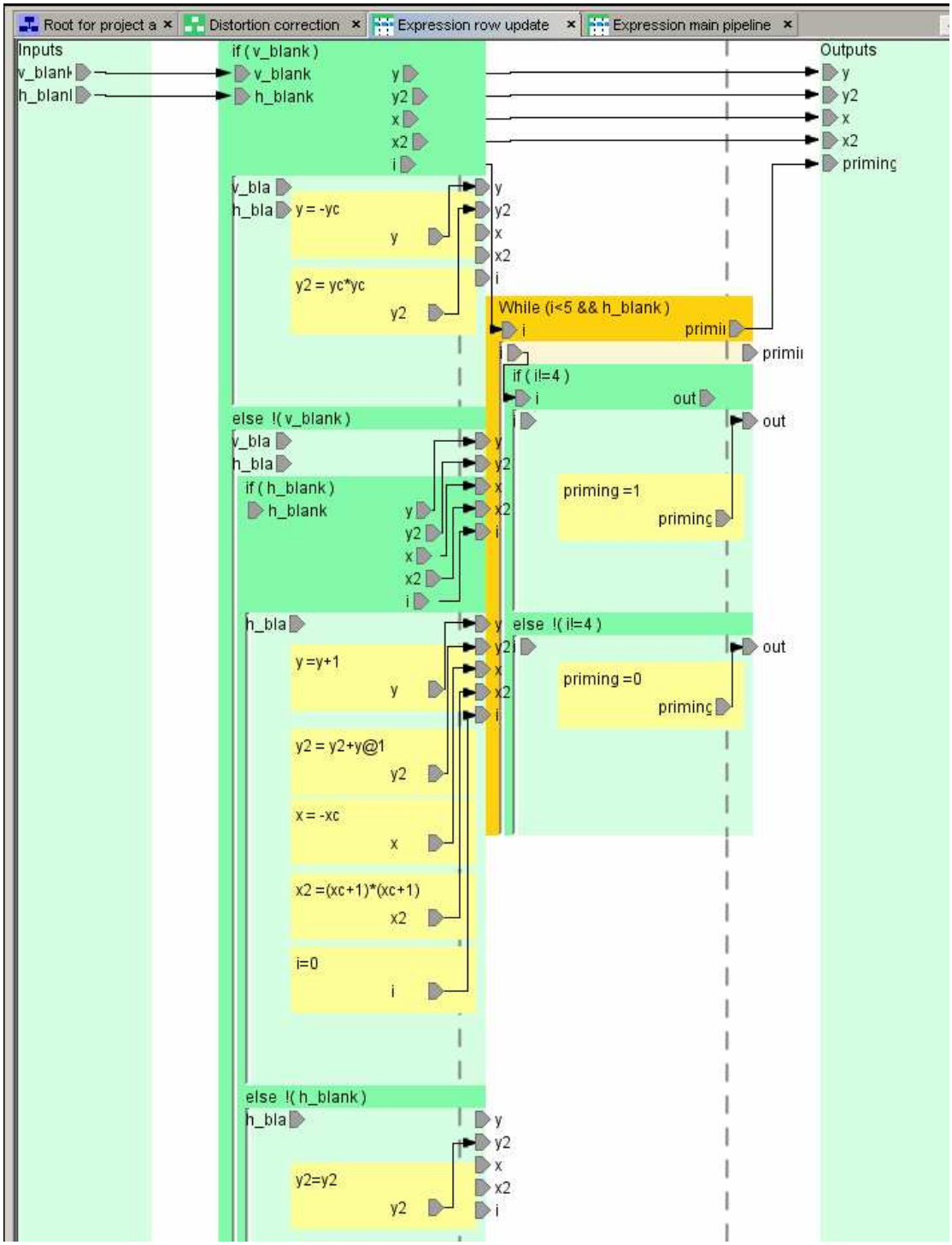


Figure 7.8: Row update expression view

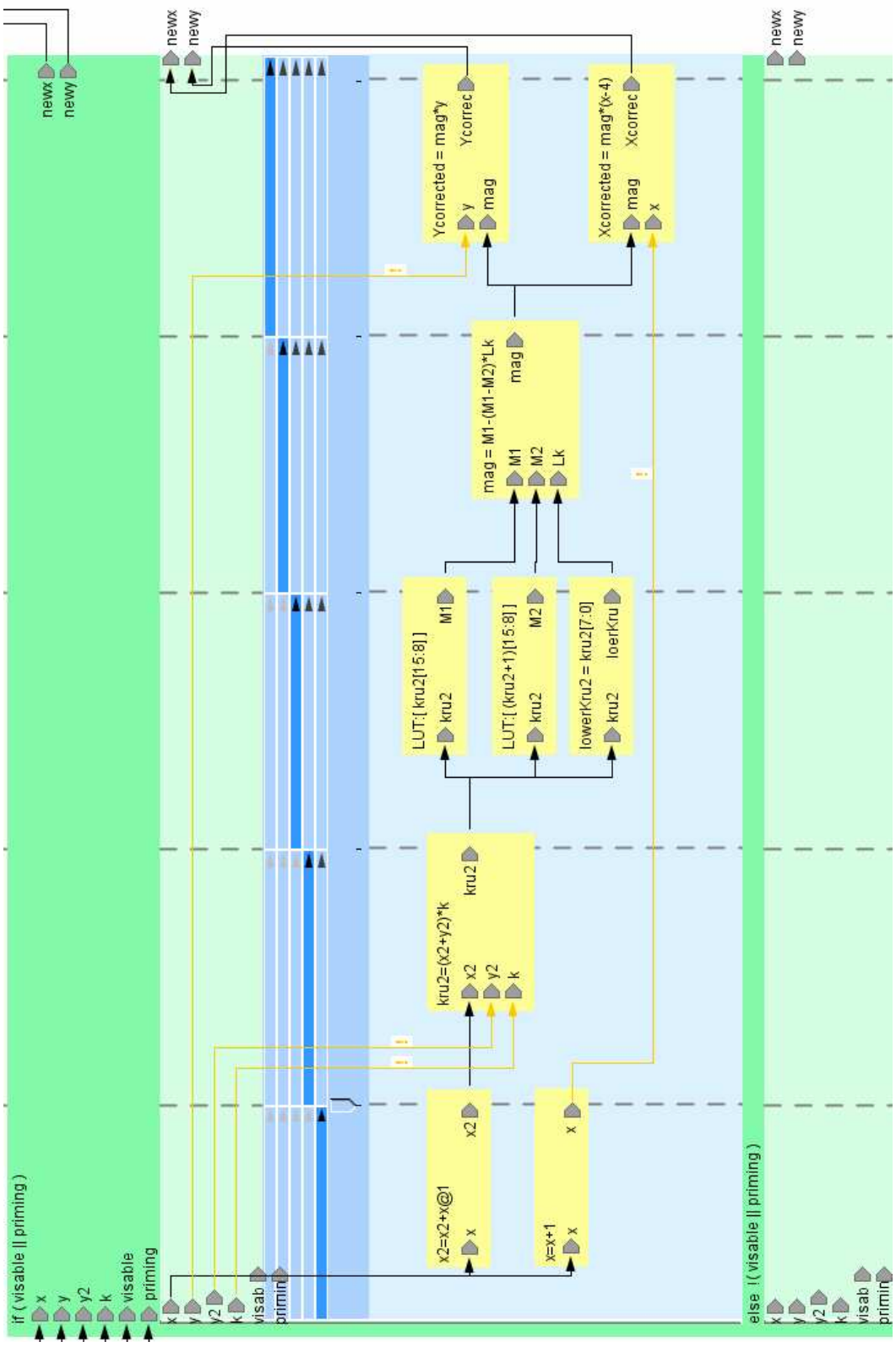


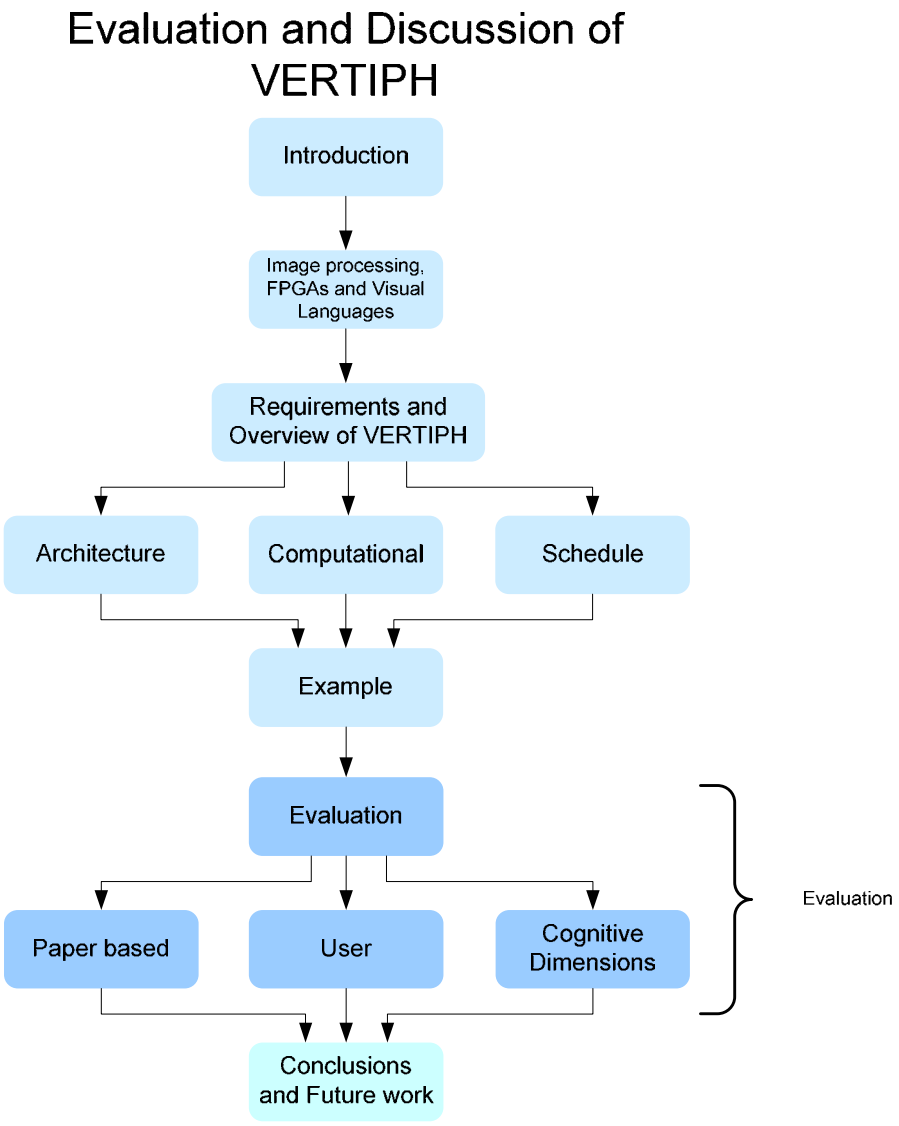
Figure 7.9: Pipeline controlled by if else

7.1 Summary of Example

This chapter has shown how the key elements of a barrel distortion algorithm can be implemented with VERTIPH. The presented solution is not the only solution that can be created but is one that illustrates a number of features of VERTIPH. The design starts with a top level architectural view of the algorithm. Both a complex and a basic data type are shown for use in the distortion correction block, as is its junction box for this block. In the computational view the design is decomposed into two related parts: a row update (active once per line) and the main pipeline (updated when a pixel is visible or priming). As the scheduling view has not been implemented at this stage, these are both controlled by low level *if-else* control structures with the priming and flushing controlled manually.

A comparison of the differences between the textual solution and the VERTIPH solution can be made. The VERTIPH notation this takes up more space on the screen. The operations are the same (it still takes the same amount of code to work), however this increased space is used to add more information by making the pipeline clearer than the text version. It is also easier to change this design to a multiphase design if the clock rate was increased with respect to the data rate.

The VERTIPH solution also has the required code to interface with external devices. In VERTIPH these hardware blocks and the connections between them are shown in the architectural view. This compares to text based designs where each is shown in a separate file with a main code block linking them. VERTIPH connects the separate parts of the design in a more intuitive way which gives an overview of the design.


Evaluation and Discussion of VERTIPH
Introduction
Image processing, FPGAs and Visual Languages
Requirements and Overview of VERTIPH
Architecture
Computational
Schedule
Example
Evaluation
Paper based
User
Cognitive Dimensions
Evaluation
Conclusions and Future work

8 Evaluation and Discussion of VERTIPH

VERTIPH has been evaluated in several ways. Once the main elements of the different views had been designed, a paper-based user evaluation was conducted to test the system so that any major design flaws could be found and fixed before implementation.

Later, when key parts of VERTIPH had been implemented, a user evaluation was conducted using participants who were knowledgeable about image processing or signal processing on FPGAs. This was to identify what users considered to be good and bad features of VERTIPH and to give insights for future development of the design.

Finally, a CD (cognitive dimensions) analysis of VERTIPH was undertaken. Although, there was some attempt to make the design of VERTIPH a steady progress through predefined sequence of steps, inevitably the process was somewhat informal, as good ideas do not always suggest themselves on cue, and later ideas may support or discount earlier ones. During this process, principles of good practice received considerable attention but, once the language had been designed, it seemed appropriate to subject the design to the thorough, formal evaluation afforded by a CD analysis. The analysis was undertaken once the design had been developed and partly implemented and covered both the completed parts of the system and on the proposed scheduling view. The CD results complement the less formal opinions from the user evaluations.

8.1 Paper Based User Evaluation of VERTIPH

A small scale paper-based user evaluation was undertaken with 8 participants, including engineering undergraduates, masterate and PhD students. Each participant was provided with a paper mock-up of the interface and paper templates to simulate interface objects. The author played the part of the IDE. Participants had a tray of different cardboard cut-outs to represent the different objects (*if-else, loop, pipeline* etc) from which they could choose. These were assembled by the participants and expressions were parameterised by pencilling in the operation. These cut-outs were available in a number of different sizes and could be extended. Terminals were added manually and wires between objects drawn in by either the author or the participant. Each participant was given three tasks: the first was to represent the architectural view for an algorithm to count grains of rice in a greyscale image; in the second, the participants specified the high-level scheduling for this grain-counting problem; the final task was to implement the low-level

operations of a given algorithm for correcting lens distortion. Participants were allowed as long as they needed to complete the tasks. They were observed during the evaluation process and could ask any questions relating to the problem or development with VERTIPH. When they had finished all three tasks, they answered a questionnaire (see appendix I) on their experiences. Following this there was an open discussion with me where they could indicate which aspects of the notation they found intuitive and helpful, and which parts needed modifying.

Most of the participants had a low to medium level of knowledge of both image processing and FPGA design. Two of the participants had a high degree of knowledge of both image processing and FPGA design. It was found that these 'expert' participants had the best grasp of the concepts, both those required for solving the tasks and those associated with the goals behind VERTIPH's design.

All the participants said that when explaining a system they would draw a diagram to aid them in the description. This suggests that diagrams are a useful tool in capturing the structure of a design. They all found that the architecture view was useful in that it encouraged top level design and was an intuitive notation. This is not surprising, as block diagrams are common in most forms of engineering and are also used in programming.

The participants were split equally on the question of whether VERTIPH is too graphical or is well balanced. Those who found it too graphical found the block level design used in the Architectural View useful for describing the system but found the Computational View too graphical for expressions and would have preferred to use text only. One solution to this would be to incorporate a split view with the assignment statements editable as text in one of the panels. Indeed, this was considered at the design stage, but there was only enough time available to implement one alternative. It was therefore decided to restrict the implementation to a diagram-only notation, as this alternative allowed us to gain useful insights. We were familiar with, and already had some idea of the strengths and weaknesses of, textual notations (see the earlier discussion relating to Figure 1.7); therefore it seemed more interesting to investigate the graphical notation.

The pipelining notation was useful and aided in the understanding of the design. However, many of the participants had difficulties solving the pipelining problem they were given, because they appeared to have difficulties understanding the problem. They lacked understanding of how to transform the set of equations into a pipeline. Most of the participants said that, with practice, it would be easier to use this notation.

The control constructs (from chapter 5 and (Johnston et al., 2006a, Johnston et al., 2009)), based on N-S control notations, were considered to be useful and helped to associate functions with the corresponding controls. All but one of the participants found the diagrams to be intuitive to both read and use.

The Resource and Scheduling View was considered to be disconnected from the other views and several participants stated that they would not use it. Other than this view being disconnected from the other views there are two possible reasons for this. With a paper based evaluation, there is no linkage between the Architectural View and the Resource and Scheduling View, so participants had to represent architectural blocks twice, which they reported as being time consuming and pointless. The second possibility is that the problem they were required to solve was too simple to really test the usefulness of the view. The problem needed to be simple so it could be solved in a reasonable time but this meant that it did not require the more complex high-level control structures offered by the state based view. The participants felt that they could define the control at the lower level. One of the more experienced participants noted that, for more complex problems, the usefulness of the high level view of the Resource and Scheduling View would become more apparent.

The developer most familiar with VHDL had several observations on both VERTIPH and its design methodology. She correctly completed all tasks but would have preferred to be able to build the state machine with the states first and then put the operations controlled by them into it. From an implementation point of view, the components of the state machine could be built in either order. We have encouraged the Architecture View as the first view so that developers think about the system as a whole first. This developer would like to have had the option for bottom-up programming; rather than having to define the top level architecture first. This would let her create low level operations independently and then stitch them together at the higher level. While this is possible in VERTIPH, it is not encouraged because a top-down approach can generally lead to more maintainable designs (Juan et al., 1993).

The developer with the most experience with implementing image processing on FPGAs also took the longest to complete the task, spending most of the time on the architectural view. The reason for this is interesting; he said in the discussion afterwards that he wanted to get all the architectural blocks correct before committing to use them. Generalizing this implies that the architectural view requires forward commitment (discussed in CD section) - that a choice made at the start would affect the outcome, and can lead to reimplementation if incorrect choices are

made at the start. The benefit is that a top-down approach encourages more thought about the problem at the design phase, which can reduce errors later in the design.

Some of the participants did not have the required detailed knowledge of both image processing and FPGA design (despite having done papers on both). Their comments were not as beneficial as those of the more knowledgeable participants; however, they still provided some useful feedback.

No major changes to VERTIPH were made following the paper–based user evaluation as the key parts (pipelining, computational control structures and use of a top level architectural view) were seen as useful and beneficial by most participants. The interface between the scheduling view and the other views needs to be clear to improve the feeling of connectivity between them. This can be done through the linking of controlled objects with the state machine and vice versa. The editing of equations did elicit some comments and may need some modification, as some participants would like to be able to edit equations outside the expression blocks. One proposed solution is to use a split panel to allow this. However, this would result in a two step process of editing equations in a text area then moving the objects which represent them into the correct sequence in the equation flow window. It was decided to implement expression editing within an expression object and to evaluate whether a split view would be desirable in the user evaluation of the prototype.

8.2 User Evaluation of VERTIPH

To answer the research questions about how easy the tool is to use and understand, a user evaluation of the prototype implementation of VERTIPH was undertaken. The version of VERTIPH and screencasts of it in operation used in this evaluation can be found on the attached DVD or at www.johnston.geek.nz/vertiph.html.

The paper-based user evaluation discussed above was conducted earlier in the project to identify parts of the notation that needed to be improved before it was implemented. As many of the participants did not have a high level of knowledge of both FPGA and image processing, the information gained from this was limited.

These lessons have been considered when developing this user evaluation on the two implemented views of VERTIPH. Originally, a user evaluation involving several tasks and a questionnaire was designed to investigate the research question of "whether VERTIPH is better than existing tools". This research question resulted in tasks and a questionnaire which compared VERTIPH designs with design in other HDLs. This is not a fair compassion, as it is expected that participants would prefer

the tool that they are most familiar with. It also resulted in tasks which did not reflect real life or allow the participants to use the tool as they wanted. As participants were unlikely to have implemented the same algorithms in a different tool it was unfair to ask them to make comparisons. Once the questionnaire and the tasks were developed it was piloted by one of my supervisors and a member of our research group. This resulted in the questionnaire being updated and the task clarified.

Once the screening form was developed, questions designed to extract the information about the good and bad features of VERTIPH were developed. Based on these questions, I could work out the task that the participant must do to answer the questions, and there was feedback between these tasks and the questions asked. The screening questions will now be presented, this includes the tasks and evaluation questions.

8.2.1 Screening of Participants

As image processing on FPGAs is a niche area, the evaluation tasks and questions would only be suitable for an expert, and therefore only a few participants would be able to take part. Participants familiar only with image processing (and not FPGA design) would not know the design conditions involved in the FPGA domain. Asking them to compare VERTIPH to other tools would be pointless; it would be like asking a cyclist to drive a car and then asking about how it compared to other cars they have driven. Therefore, only people with hardware design experience should be considered, preferably those with good knowledge of image or signal processing. Participants filled out a detailed screening questionnaire to determine whether or not they had sufficient experience to complete the design. The screening questionnaire also aimed to identify past experience and design preferences which could colour their design decisions and evaluation.

There needs to be a balance between being selective enough to get “perfect” participants and being so selective that no participants qualify. The screening questionnaire was designed to extract information about the participants’ past experience and knowledge. The questionnaire also asked about how they would solve a design problem so that we could see whether or not their preferred design process affects their later responses.

After the screening, test participants who did not meet the requirements might be excluded, though lack of hardware design experience did not exclude someone if they had a good understanding of the concepts, and were willing to participate.

The participants that were invited were known either to me or to my supervisors; this allowed us to target people who we were confident had the required experience. Initially ten participants were contacted, three did not respond or declined due to time constants, and three read the screening questionnaire and felt they did not meet the FPGA design experience requirement. This left four participants, three with experience in implementing image processing algorithms on FPGAs and one with experience in signal processing algorithms on FPGAs. This has led to an evaluation by experts, which has benefits (in terms of understanding the problem domain) and drawbacks (in terms of being used to a particular methodology or tool to solving the problem). Three participants were most familiar with lower level HDLs (mainly VHDL), though they also had some experience with Handel-C. One participant had only used Handel-C and instead of using it for algorithms, had developed specialised CPUs for high level image processing operations. All were used to using a mixture of parallel and pipelined design to speed up algorithms when mapping to FPGAs. Most used a mixture of top-down and bottom-up programming, using a top-down approach to get the key parts of the design then building these up from the bottom by creating smaller functions. All but one had a good understanding of the use of pipelining.

8.2.2 Introduction to VERTIPH

The post-screening group of participants needed to learn how to use VERTIPH. They learnt how to use its interface and the notations needed for the rest of the evaluation through two introductory tasks which were explained by means of a screencast and screen shots.

The first task (detailed in appendix II) introduced the participants to the architectural view. The second task (detailed in appendix II) introduced them to the computational view.

Following these introductory tasks, the participants completed a short questionnaire (appendix II). This served two purposes:

1) to identify any problem with the understanding or use of VERTIPH which could affect the later evaluation stages, so that these can be rectified.

2) to gauge the participants' initial reaction to VERTIPH.

All but one of the evaluations were preformed remotely and therefore this information was only gathered after the participants had completed the evaluation.

This limited the usefulness for identifying any problems before they could occur but did enable me to see when participants were confused. The initial reaction was generally that they need to use VERTIPH more before they could give an opinion. The design flows used by participants were generally similar to ours, but influenced by the tools they were familiar with. Generally, the design flow followed by the participants was to develop and test the algorithm in software (such as MATLAB) and then port the design to hardware. The porting generally involved developing the top level design before constructing the low-level detailed design.

8.2.3 Evaluation Tasks

After the participants had become comfortable using VERTIPH, they completed the evaluation tasks. These tasks have been developed to allow the developer to test key parts of the system without taking too much time to complete. They were selected by first working out what information was required and then using this to generate the questions needed.

Overall, the author wanted to find which aspects of VERTIPH were useful for aiding in the design process and which aspects were detrimental. This allowed key features of the language to be evaluated and also future improvements developed. There was also a desire to see how users develop their designs and how they implement a given design.

The parts of VERTIPH that the author wanted to test were:

- the multi-view design process, both the enforced top-down design process and the separation of the high-level block design and the low-level computational design.
- the computational view's separation of parallel and sequential operations.
- the pipeline control structure
- the *if-else* and *loop* control structure
- the type editor and junction box editor

These are the novel parts of the language compared to other HDLs, so evaluating these will help to answer the research questions of whether it is easy to understand the design and map it to FPGAs.

The evaluation process was constrained by need to allow the participants to explore the design space but still guide them towards solutions that used the above

features of the language that were being evaluated. The time that the participants can be expected to volunteer is also constrained; a too-lengthy evaluation process will limit participation and annoy participants. Therefore, it was necessary to design tasks that could be completed within a reasonable time and still expose the participants to VERTIPH so that they can answer the required questions. This meant that the design tasks and the questions had to be developed hand in hand.

To accomplish this, the evaluation consisted of two further tasks (after the two introductory tasks). Both had an image processing focus but were general enough so that a non-expert could complete them. They also had the majority of their focus on the computational view as this was the most complex and also where most of the time would be spent in a normal design.

The third task involved implementing an image processing algorithm to correct for barrel distortion. The problem was given to the participants in the form of a poster with the output and a diagram and in the form of pseudo code (shown in appendix II). The participants were asked to implement this design in VERTIPH. They needed to identify that a pipeline was the most sensible approach, although they were free to modify the algorithm to produce a similar algorithm. Though they were guided through a design in this task, it allowed them to concentrate on the VERTIPH design rather than solving the problem itself.

The final task (a min-max filter, appendix II) was more open-ended, and although it was image processing-based, it could be implemented in many different ways, and had several simple solutions which should be familiar to signal processing practitioners. It also had the added dimension of handling edge effects. It was assumed that most participants would not consider the edge effects but gives the opportunity for design decision questions in the questionnaire and interview, which may test knowledgeable participants. Participants were not required to complete all of the task as it could become quite time consuming.

8.2.4 Analysis of the Questionnaires

All of the participants took a lot longer to complete the tasks than expected; this led to most of them not finishing the final task. There were several reasons for this: the learning required to become sufficiently proficient using VERTIPH for completing the tasks, and the amount of work required for the final task being greater than anticipated.

One criticism from all of the participants was that the junction box interface was not easy to use (Figure 8.1). They found that the need to define the internal and

external terminals and then make a connection was cumbersome. They would prefer that any internal or external terminal created would have the corresponding terminal and connection automatically. This, with hindsight, is what should happen; as most of the time the internal and external terminals will be the same with the breaking out or combining of data types only required with some operations and data types.

Participant A was experienced in the implementation of signal processing algorithms on FPGA, mainly in VHDL. Overall the participant liked the top-down approach of VERTIPH and found the use of the spatial dimension for separating parallel and sequential operations very useful and very easy to understand. The pipeline notation and the control structures were also a useful and natural notation and did not need any obvious improvement. Compared to VHDL, VERTIPH was simple to use.

The participant found several areas of the design that needed improvement. They would like a quick method for selecting types when connecting two untyped terminals. They also would like the auto generation of types for output of expressions based on the input type and the expression (where possible). The most interesting suggestion was that they would like the computational view to be split-screened. This would augment the computational view's current graphical area for expressions with a text area for equations. This would, in their view, give the ability to write expressions quickly and then have these automatically appearing in the graphical computational view section. The computational view section would then be used to show the timing information by laying out the equations. This would also require that variable names be associated with the terminals and wires, creating a need for a name space. A similar idea to this was discussed in the early phases of the design of VERTIPH and in the paper-based user evaluation but was discounted as it involved creating expressions, then manipulating the visual objects used to represent the timing. However, the participant wanted to be able to write the expressions they needed to do the operation and then work out the timing relationship. This aspect of design needs to be investigated further. A modification which allows both approaches needs to be developed and tested as this would let the developer work in the way they most prefer.

Participant A would also like to see how optimised the compiled output (logic utilisation and structure) would be before passing judgement on whether or not he would use the tool. Overall the participant found that VERTIPH *"helped with my design process"*, but would need more use to say whether it could replace his current HDL.

Participant B did the evaluation remotely and in his own time. He had worked with FPGAs, building complex embedded systems, including image processing systems. His past experience was with VHDL and Verilog, but also with the EDK tools from Xilinx. He had implemented simple image processing algorithms from published descriptions and had not developed any from scratch. In the screening questionnaire he identified that *"Designing visually is good for showing structure and the flow of data. It is particularly good for the top levels of a design hierarchy. It does not capture the low-level details of a design so well*". When commenting on their main language he stated that VHDL is a verbose language. This also makes it difficult to debug as the code size increases, as there is a lot of code to go through. Verilog is more compact than VHDL, and it seems to have better support for parameterisation and generation. He liked the ability in both languages to describe a variety of different objects, such as FSMs, FIFOs, filters, in a "generic" way. Both languages get unwieldy as the design increases in size, and the major problems come when you have to make a major change to the structure or function of the design. Because of the parallel nature of VHDL and Verilog, it can be difficult to determine what effects a change to one part of the code will have. At the top-level of a design, which is often mainly wiring, both VHDL and Verilog are tedious to produce and difficult to read. Participant B only likes a top-down design approach for a first-time design. Participant B thinks a mixture is better for designs with a number of components already designed to avoid "reinventing the wheel". This criticism is not quite valid, as top-down design does not preclude the usage of existing components and common modules can be reused.

His development process was to "*model and explore an algorithm in MATLAB, since it is easy to develop, there are many things you don't need to think about (such as the type of variables), there are many built-in features / toolboxes which reduces the development time, the development is interactive and data analysis is easy. Once I have fixed the algorithm, I make a model of the way I expect it to operate in hardware, so I have a standard to compare outputs against when implementing the hardware. This model includes things like fixed-point representation. Next, I write HDL, and test it using simulation – matching the results of simulation with the software-based model*". This is quite similar to the design process that our research group follows.

Due to time constraints, participant B did not complete the final task or the questionnaire and instead made a summary of his comments on VERTIPH.

"The GUI for VERTIPH is fairly self-explanatory, and does not take much effort to learn how to use, which is good. I think the use of a graphical notation to express connectivity and dataflow, especially at a high-level of a design, is useful. The attempt to show computation in terms of the clock cycles is interesting, and has promise. The ability to have a hierarchy is good. I think the principle of defining types is probably a good one, since it means that a major change can be achieved by updating the type."

The junction box interface (Figure 8.1) was *"tedious"* as many wires were needed for connection and he did not like the fact that connectivity seemed to be able to be "hidden" in the interface. He thought this would be better if the interface was incorporated into the inputs and outputs columns of the node view when descending the hierarchy and would also like the ability to select whole groups of bits for connection.

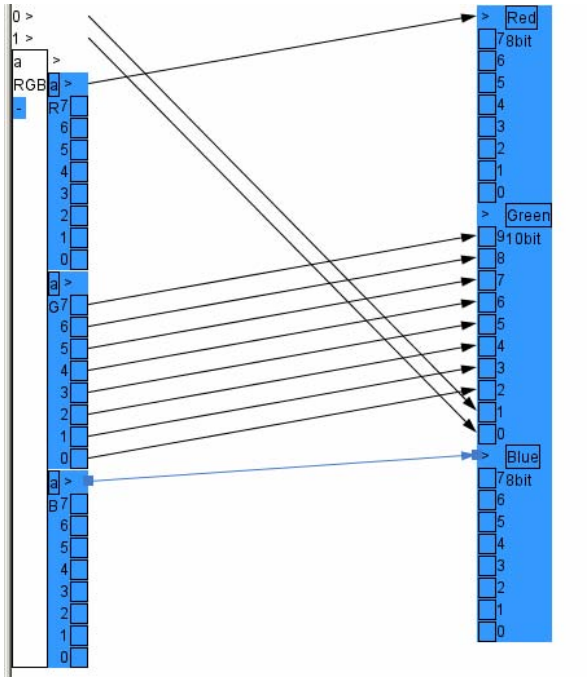


Figure 8.1: Junction box view with RGB components

They would like to see a synthesizable output from the tool (such as synthesizable RTL). They also thought it "*would be really useful if the GUI could be incorporated into a simulator somehow, so that you get feedback in the GUI about the functionality of the design"*. They also thought that "*at the moment, the graphical representation is very much dataflow orientated. This is fine, but I think it would be interesting if the system could be designed in a dataflow method but then mapped to some other architecture, such as a microprocessor-based system where communication between objects uses buses. In the GUI, you could offer different*

views of the same system, one showing the dataflow and one showing the physical architecture. The interaction between the performance of the dataflow view as one changes how it is mapped to an architecture would be interesting". These are both interesting ideas and are worth investigating. Incorporating a simulator into VERTIPH, either at the functional simulation or a clock-accurate simulation for validating complex timing issues is a complex task and outside of the scope of this thesis. The mapping between a dataflow design and a microprocessor-based system is a question of hardware-software co-design. This is well outside the scope of this thesis but would be an interesting topic for a future PhD. As there is a mixture of pure dataflow parts of the design, parallel and sequential elements, the segmentation of the architecture developed from the design could be tweaked through a user feedback system showing the effects of moving elements of the design (probably computational nodes) between dedicated hardware, hardware which can be reconfigured on the fly and software on a microprocessor. The hardware and software code could then both be generated from the VPL.

Participant B found the pipelining *"a little confusing"*. He concluded that *"I guess I therefore do not understand the difference between the pipeline structure and placing expressions as normal within the computational view."* He appeared confused about what the clock divisions signified and he assumed that there was an implicitly pipelined structure (as they have a strong background in parallel HDLs). This confusion could have been avoided if the evaluation was not done remotely (and in a different time zone) as it could have been identified and corrected early in the evaluation. They would also have liked to be able to have unregistered expressions. This is a simple extension to the language, which is in the design but not the implementation, as it was deemed non-critical for a prototype. Ideally he would allow the tool to work out where the expression should be broken up and registered. This would make it closer to some of the software-to-hardware design automation tools, extending the idea of using a split screen, so that not just the expressions are shown but also ways to further pipeline (break up the expressions) could be added by adding pipeline stages. In this view, the developer would type out the initial equations, then edit the computational expression visualisation. They could then be shown how the speed of the design could be increased through the use of low-level pipelining. This would be a significant extension of the project as it would be necessary to evaluate a number of segmentations.

In general, Participant B thought that he would prefer to use an HDL for the low-level design, as it gives finer control over the computation being performed. He

also felt that *"for the low-level details of the design, a visual interface is less productive than a textual one. However, some of the visual methods you have used could be very useful to help design the HDL. I could imagine that if you are writing in HDL and have a visual interpretation of the HDL design in terms of the clock cycles and the scheduling of computation, this would be very useful*".

This participant also came up with some good questions about the ease of designing an FSM in VERTIPH. As the prototype does not have the FSM-based scheduling view, it is not surprising to have this limitation identified in the current implementation. They also recommended creating a *"stallable object"* as they are very useful when it comes to design reuse and system integration, since you do not have to worry so much about making sure latencies match exactly. Although this is one of the properties of a block in the scheduling view, it would be worth investigating having an explicit stallable control within the computational block. At present channels can be used for communication between architecture blocks which can in part mimic a stallable object.

In summary, like participant A, participant B liked some parts of VERTIPH and disliked others. The junction box interface was again a problem and he would also like to have more text at the lower level of design. Participant B also made many useful comments for ways to improve the design and areas of possible future work.

Participant C's evaluation was also done remotely, in this case with some interactive discussions during the early stages. She had designed a few image processing algorithms and implemented many others on FPGAs. Most of this work had been in VHDL with some work in Handel-C. She completed all of the tasks, although the last task was not quite correct and did not take into account edge effects. This participant thought that her main design tool *"VHDL has strict syntax: which is good for control but frustrating at times"*. She normally follows a mixed top-down and bottom-up design approach. Like the others she did not like the number of steps required to connect the inside types with the outside types using the junction box interface. She liked the *if-else* and *loop* control structures. She found the controls clear and wished her design tool had this *"colourful and automatic control structure"*. She felt there was a good balance between text and visual notations. She thought that the best part of VERTIPH was the *"architecture view, this is what is missing with VHDL and software that aids the design for them. For example the RTL viewer in Quartus (Altera) is meant to present a similar view from the VHDL project, but most times it shows a mess of views that spans multiple pages which is completely useless."*

Participant C did not like having to go down a number of levels to get to the expression level, and she also would like to be able to simulate the design in VERTIPH. She suggested that it would be useful to label the 'wires' with their type. This could be easily implemented through a pop-up-on-mouse-over or other similar action to avoid the clutter that would come with permanent labels. She would like to be able to control lower level structures. Originally she stated that the pipeline notation was not useful, as she *"did not like using pipelines."* Further discussion in the interview was required to establish what this meant, as it conflicted with the positive statements that VERTIPH *"make me think more structurally, it's clearer to see where the data is"* - a main issue when defining pipelines. It was found that even though she had had problems in the past with incorrect data paths, Participant C does not separate parallel design from pipelined operations and treats them as being the same. The consequent errors were the reason for her dislike for using pipelines. This is exactly the sort of confusion that VERTIPH's pipeline notation is intended to avoid.

She thinks that with refinement, VERTIPH would be a useful HDL for her work even though she has not used another tool like this for HDL design. She found that *"there are parts that will definitely make my normal design practice faster and easier. Maybe it can be used for only sections of the design, so I can have the best from the two worlds."* Like the other participants, she would like to be able to use our visualisation with her existing HDLs. Overall the evaluation was positive, with most of the criticisms relating to the 'wiring' and junction box.

The fourth evaluation was also performed remotely, also without completing the final task. Participant D has a degree of experience with implementing image processing on FPGAs. This has mainly been in designing specialised micro processors (on FPGAs) for high-level image processing algorithms. He liked the top-down design approach and thought it would be good for reusing code. He found the pipeline notation useful and thought it was *"quite nice to use"*. He also liked the distinction between sequential, parallel and pipelined operations. He found the control structures useful, especially as the he was familiar with N-S diagrams which VERTIPH diagrams are derived from. His main criticism was the number of steps required to assign pins in the junction box interface. In general they found VERTIPH useful and easy to use.

8.2.5 Summary of User Evaluations

The parts of VERTIPH that the author wanted to test were:

- the multi-view design process, both the enforced top-down design process and the separation of the high level block design and the low level computational design.
- the computational view's separation of parallel and sequential operations.
- the pipeline control structure
- the *if-else* and *loop* control structure
- the type editor and junction box editor

*Multi-view design proces*s: The multi-view design process was found to be "different" but could be useful. Enforcing a top-down design approach did not seem to be a major problem with the participants. The architectural view was viewed positively, with most participants preferring it to the module wiring tools they were used to in VHDL.

Computational view: The computational expression view had the most mixed reviews. Generally it was thought that the visualisation was useful. However, the participants would like to be able to edit the expression in text and then either have this view automatically generated or edit the placement of the expressions in the view. Generally the separation of parallel and sequential operations and the pipeline control structures was considered useful. However, they would like to use this more as a visualisation rather than for editing directly. The control structures were viewed very positively.

Pipeline control: There was also some confusion over how the pipeline control was used, with several participants assuming that sequential operations were forming an implicit pipeline, when operations need to be inside the pipeline control structure for this to be a case. This is perhaps due to confusing our dataflow diagram and the RTL diagrams which some HDL IDEs can create. With more training and use this confusion should be reduced.

Control structures: The participants who commented on the control structures found them useful and intuitive. One participant was very complimentary about them and wished their design tool had such a control structure.

Type editor and Junction box: The data type editor was viewed positively. All of the user evaluations commented that the junction box interface needs to be improved by adding both internal and external terminals and connecting them by default when a new terminal is created, rather than having to create the internal connections manually.

In addition to these questions, all the participants thought that the incorporation of a simulator would be desirable. There was also a desire for a number of different specialised hardware objects.

The majority of participants involved with the evaluation thought that they would consider using VERTIPH after further refinement, and if it produced optimised FPGA designs.

In summary, the participants found most parts of VERTIPH useful, even if they would prefer to use some parts as a visualisation rather than an editing tool. This implies that VERTIPH provides a good balance between textual and graphical representation, although further refinement is still needed. The evaluations resulted in many useful suggestions for other improvements and future work. Unlike the paper-based user evaluation, the implementation evaluation provoked no comments about the perceived problem of forward commitment.

8.3 Cognitive Dimensions Analysis of VERTIPH

Cognitive dimensions analysis is another way to analyse and evaluate a programming language or user interface (Green and Petre, 1996). It is a broad-brush approach in which the language or tool is assessed against a number of different concepts related to its usability. Unlike (Shu, 1986) classification in which visual languages are assessed in terms of three independent dimensions (visual extent, scope and language level) and which produces graphical visualisations that allow languages to be compared with one another, the cognitive dimensions are deliberately not independent, in an effort to illustrate differences and identify trade-offs between the choices made in the language design. Other existing fine-grained user interface evaluation systems look at the low-level details, such as the actions involved in performing a task (Green and Petre, 1996). This makes them ill-suited for evaluating a programming language, where such conventional HCI concerns relate more to the IDE used to capture the program than to the language itself. CD analysis gives the opportunity to analyse the design both during the design phase and once the design is complete, without user evaluation.

There are trade-offs between the different dimensions. The features that score well along one dimension can have implications or effects that reduce the score in other dimensions. Although some of these trade-offs have been identified in (Green, 1996), have not been formally analysed (Green, 2000). Some of the common relationships can be identified in Figure 8.2.

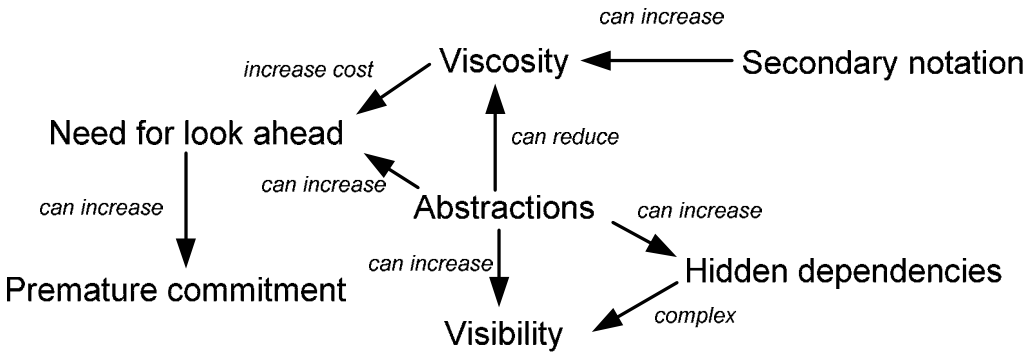


Figure 8.2:Trade-offs adapted from (Green, 1996)

Creating the correct *abstractions* to reduce *viscosity* (the notation's resistance to change) requires *look ahead* which can cause *premature commitment* (decisions that must be made before all the necessary information is available). The cost of

premature commitment is increased if the *abstractions* are *viscous*, so that if the user makes a wrong choice, it is hard to fix. *Abstractions* can also introduce problems of *hidden dependencies* (where not all the dependencies between entities are shown), because one *abstraction* is defined using another. *Secondary notations* (comments, layout etc related to code) can increase *viscosity* as these need to be updated to reflect any changes.

(Petre, 2006) identifies that there is a task sensitivity, in that different notations are more important at different times of the design phase. This depends on what the user's goals are, and on what information needs to be emphasized for the current task. For example, *juxtaposability* (the ability to view two or more parts of the design at once) is not particularly relevant for very short programs but is important for understanding and debugging of very large programs. Trade-offs are not static and may change over time, as the goals of the programmer change as they move through different phases of the program implementation.

It is not ideal to have the developers of a system analyse it, as they can be blind to errors or bad decisions which may be more obvious to independent evaluators. This is illustrated, in the case of VERTIPH, by the problems related to I/O pins and the junction box that seemed obvious in hindsight, but were only identified through the user evaluations. Consequently, the evaluation that follows – which was undertaken by the developer of the system – is limited. However, the cognitive dimensions framework is used to evaluate the design decisions that were made and the trade-offs which resulted.

8.3.1 CD 1: Abstraction Gradient

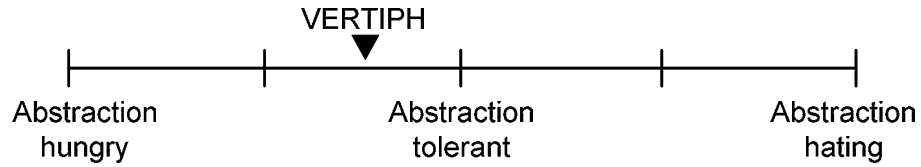
This deals with the range of abstractions that are required to use a system. An abstraction allows entities to be grouped (encapsulated) to either lower the viscosity (the notation's resistance to change) or allow the developer to make the notation closer to how they perceive the system. Systems may have an *abstraction barrier* if the number of new abstractions that must be understood to use a system exceeds some minimum. Systems can be *abstraction-hungry* if they need a large number of abstractions to be defined before use, *abstraction-tolerant* if they permit abstractions but do not require them, and *abstraction-hating* if they do not allow user-defined abstractions.

In VERTIPH each architecture block is an abstraction; it encapsulates the computational logic to perform an action. This requires developers to create an abstract object – the architecture block – before they implement any of the associated

computational functions. This approach was taken because the abstraction of relatively complex algorithms as blocks is common in the image processing domain. It allows an operation to be defined with an interface that is independent of its internal computational elements, and this can be reused. Such a block view also enables a high-level view of the overall algorithm to be developed. Being somewhat abstraction-hungry at this higher level is not too much of a difficulty for image processing applications on hardware because the design process generally requires that the image processing algorithm first be developed in an interactive environment before it is mapped to hardware (Gribbon et al., 2007). Therefore, the use of abstraction at this level aids the porting of the algorithm to VERTIPH because the algorithm has already been developed and the abstractions that have already developed during the design process can be reused.

VERTIPH also allows, but does not require, that data types may be combined (hierarchically) to form data structures. The basic type is a variable-width fixed-point number. The developer can group a number of these to create a complex data type and these can also be incorporated into more complex data types. Such data abstraction was provided for two reasons. First it reduces the number of "wires" needed to connect different architecture blocks. Secondly it is envisaged that in future developments it will be desirable to overload standard operations so that they apply to the complex data type. An example of this is adding two colour pixels directly rather than breaking them down into their fundamental components (red, green, blue) and adding those.

VERTIPH fits somewhere between being abstraction-hungry and abstraction-tolerant.



CD1: Abstraction Gradient

8.3.2 CD 2: Closeness of Mapping

Closeness of mapping is used to describe the mapping between the problem world and the program world. The closer the programming world is to the problem world, the easier the problem-solving should be.

In VERTIPH, the language is mapping from two disparate problem worlds, the image processing domain and the digital hardware design domain. Both worlds

need to be considered when designing the language features. The mappings may not be mutually compatible and design decisions need to be made about what is most important to the developer for aiding in designing their system.

The image processing problem world is difficult to define. Image processing was originally performed in the optical domain (analogue world) using a sequence of lenses and filters to operate on the image. With the advent of digital image capture systems, image processing development moved to computer hardware, both as dedicated hardware (in ASIC and specialised processors such as those in (Duff, 2001)) and as software running on PCs and microprocessors. The move to digital processing allowed for new, non-filter based operations to be created. Image processing now uses a mixture of filter, algorithmic, artificial intelligence and other techniques from computer science, statistics, mathematics and engineering to produce the desired result. However, in most cases, operations still follow a sequence, with data moving from one operation to the next in order. This implies that image processing is strongly dataflow orientated. Other visual image processing systems such as Khoros (Konstantinides and Rasure, 1994) have exploited this through a box-and-wire design flow. The design space for digital image processing is similar to other programming systems, although usually at a higher level, as it dealing with image processing operations rather than the low-level algorithms required to implement those operations.

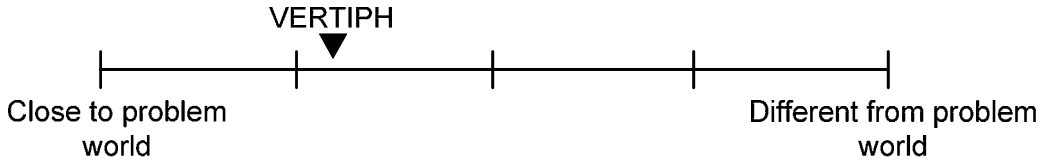
In the digital hardware design problem domain there are different problems to express. Many of the key issues are at the low level and for FPGAs include propagation delay, synchronisation, state machine design, logic minimisation, layout, routing, and clock distribution. Many of these are taken care of by the FPGA mapping tools. When looking at the higher level design (what the developer is most often faced with) concurrent operations and pipelines are the main aspects of the digital design space. Propagation delay (and therefore clock speed), memory access (bandwidth, sharing) and inter-processor communication are also important. These physical factors are difficult to concentrate on while also designing the algorithm to achieve the desired result.

One common way of mapping between a digital design and the real world is to use schematics. Schematic entry provides a one-to-one correspondence between the digital design and the implementation. Schematics allow for the gate level functions of the design to be represented pictorially. These are then implemented in the hardware, as they would be using the discrete components.

To make the mapping as close as possible, there needs to be a good metaphor for pipelining; as discussed in earlier chapters, implementing a pipeline is challenging and can be error prone. Notations for pipelining are poor in current text-based languages, as they require the developer to keep track of the data dependency through variable names. Often there is a need to use secondary notations to clarify the stages in the pipeline to the user. As a pipeline is a dataflow-dependent design the pipeline notation was based on a standard sequential visual dataflow notation. This has been augmented with extra notational information to offer information to the developer about the latency of the pipeline.

VERTIPH is attempting to map from the image processing domain to the digital hardware design domain. One of the problems here is that algorithms are conventionally thought of in terms of software rather than hardware. VERTIPH maps image processing onto a representation of the hardware that is relatively transparent, reflecting the lower level image processing algorithm.

As VERTIPH offers a high level architectural view that closely maps to the image processing perspective and a computational view close to the operation level image processing algorithm and FPGA concurrency requirements, VERTIPH is close to the problem world but does not map to it directly.



CD2: Closeness of Mapping

8.3.3 CD 3: Consistency

Consistency aims to measure the extent to which similar functions within a language are represented by similar forms. In a consistent language a user who has mastered the rules that govern one part of the notation will be able to use the same rules in similar situations throughout the rest of the language.

Though each of the three views of the system are different, in that they show different relationships, the author has tried to keep the notation similar or the same when there is a similar relationship or concept that needs to be represented in more than one view. The representations of sequential, parallel and pipelined operations are the same for the computational and scheduling views. Both the architecture and computational views use the same terminal (Figure 8.3 and data type (Figure 8.4) representations.

Figure 8.3: Architectural and Computational nodes showing terminals

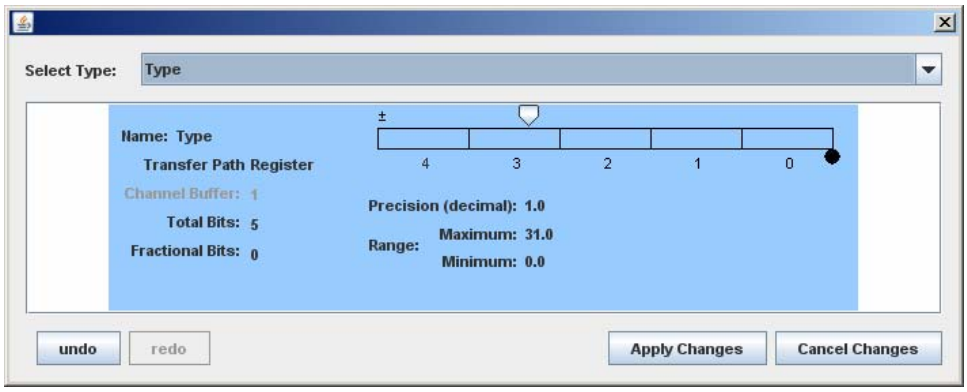
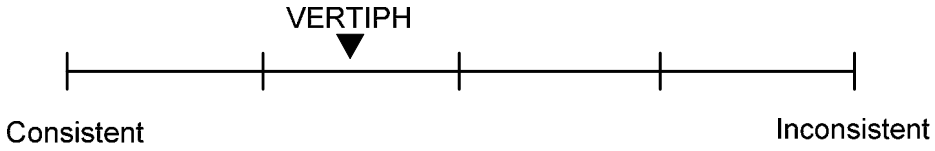


Figure 8.4: Type editor

The notations within a view also need to be consistent; the architectural view only has one notation so is highly consistent. The computational view has many notations specifying different functions. These can be grouped into expressions and functions (expressions which run for more than one clock cycle), looping controls and conditional controls. Compared to conventional textual languages the consistency for loops is improved. There is only one loop construct with variations for control at the start and the finish instead of separate constructs for *while*, *for* and *until* loops. Conditional controls also only have one notation for *if*, *if-else* and nested statements.

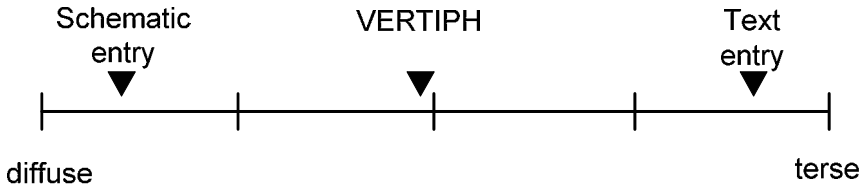


CD3: Consistency

8.3.4 CD 4: Diffuseness / Terseness

Some notations use a large number of symbols and or take up a lot of space to achieve the same results as more compact notations. Diffuse notations bury the functional specifications amongst non-functional specifications, which are identical from one program to another, so that it is difficult to see what a program does without detailed analysis. For example Cobol is diffuse and APL is famously terse.

Our representation has attempted to balance the number of items on the screen with the amount of information they convey. The computational view can be diffuse particularly when developing large pipelines. The pipeline view is diffuse because the same functionality could be encoded in a more compact text notation. The author considers that the diffuseness of the notation is justified because it maps more closely onto the user's mental model of pipelining than other, terser, notations. In VERTIPH, the pipeline control structure has been made more diffuse in an attempt to make the complex design more understandable. Where this representation is unnecessary, text is used for its compactness, such as in the expression where text is used in preference to diagrams. This use of text within the computational view improves the terseness. Using a mixture of the two allows the developer to use the best features of both the representations.



CD4: Deffuseness/Terseness

8.3.5 CD 5: Error-proneness

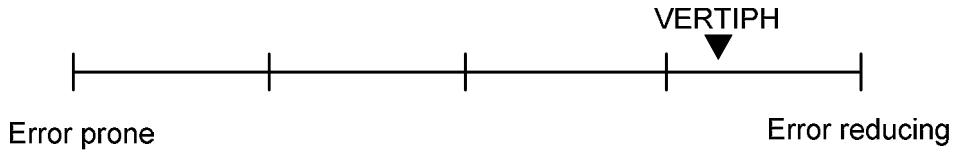
This is a measure of whether the notation introduces any features which increase the likelihood of making mistakes.

The author has identified pipeline design, resource sharing and scheduling as common sources of errors in current languages. By specifically addressing these aspects of the design, the author has aimed to reduce the errors in these aspects of FPGA design. A more detailed user evaluation will be necessary to identify any other sources of errors introduced by the language.

As discussed in earlier chapters, pipelining can be a difficult and error prone task. The problems can arise at the design stage (a pipeline that is badly designed) or during the implementation. Both the terseness of present languages and their emphasis on parallel processing foster the generation of "silly" errors during implementation. Even experienced developers (including the author) can make "silly" mistakes when implementing a pipeline. Common errors include not registering outputs that are used at a later stage and running an operation in the wrong stage of the pipeline. One of the reasons that these errors occur frequently is that present languages require the developer to manually keep track of the data dependencies

through register names. The failure of current languages to differentiate between parallel and pipelined operations leads to control and scheduling issues. VERTIPH's pipeline notation encloses the operations in a formal control structure, separating the pipeline from the rest of the design and reducing the chance of these errors by making the pipeline a sequential diagram. This makes it clear at what stage an operation occurs so that developers do not have to manually build a timing diagram of their design to identify errors and it makes connections between different stages more obvious. As discussed in earlier chapters, marking the operations that form a pipeline as a pipeline also simplifies the control and scheduling of the pipeline.

In image processing on FPGAs, the system is either controlled at the low level, with the information for the control decisions being extracted from the data that is arriving, or at a higher level, where the activation of processing blocks is based on event conditions. Having to do the high level control with low level constructs can cause errors, as control constructs and conditions need to be repeated in different sections of the code. To address this, a state-machine-based control was designed to allow high level control to be implemented in a clearer global way which may reduce errors. There is no empirical verification of this at present as this view has not been implemented. The theoretical argument for the reduction in errors is based on the regular control structures used to implement image processing operations. Often the same control structure was used on many different parts of the system, so each had to be modified when a change in the condition was made. Making the condition global and allowing a group of processors to be controlled by this state should reduce errors by ensuring that the controls are consistent. In situations with multiple control conditions, the number of variables which have to be passed between modules is also reduced as tokens do not need to be passed between objects to control the processing. This should also help with the scheduling of resources as discussed in earlier chapters.



CD5: Error-proneness

8.3.6 CD 6: Hard Mental Operations (HMO)

(Green and Petre, 1996) use the phrase *hard mental operations* in a somewhat counterintuitive sense. They mean it to refer to operations or

combinations of operations whose representation makes them mentally difficult to construct, independent of their underlying semantic complexity.

A good indication that a language ranks high for hard mental operations is that developers often need to resort to following code using their finger, by added annotation or external diagrams to keep track of what is happening. (Green et al., 2006) comment that HMO is often misunderstood and that it is meant to cover the overload of the cognitive processing. A good example of this is conditional control logic. The logic created when *if-else* statements are nested can quickly become complex and difficult to understand with the developer left to work out the required Boolean logic for false conditions based on multiple true conditions.

One of the key features of VERTIPH is how pipelining is represented. Pipeline implementation is an inherently challenging task. It is quite easy to specify small pipelines, but as the design gets more complex it can become challenging and is arguably an explosive mental process. This does not mean that a good representation cannot make pipelines easier to understand. To measure the quality of the notation, it is useful to compare it to existing languages. Both VHDL (a lower level HDL) and Handel-C (a modified high-level language) have parallel and sequential expressions. In VHDL all operations occur in parallel by default; operations within a procedural block run in parallel or sequentially depending on the assignment symbol used (<= and :=). Handel-C has seq{} and par{} blocks to determine whether expressions occur in parallel or sequentially. In both languages a pipeline is constructed by writing parallel expressions. A pipeline is therefore represented as a set of parallel statements linked by variable names which the developer must keep track of. It is very easy to make mistakes for pipelines of any complexity. If a pipeline contains stages with parallel elements then the mental mapping becomes even harder for the developer. In both these languages, the mental mapping can be thought of as hard. Hardware compilers such as SA-C fully hide the pipelining of operations. This removes pipelining from being a hard mental operation completely, but at the cost of loss of control over the design. Therefore, hardware compilers have the least hard mental operations, but do not provide the same level of control over the design.

VERTIPH can be rated on the continuum of hard to simple by comparing VERTIPH's pipeline notation against the options available in other languages. As VERTIPH uses a visual dataflow design, the dataflow is linked by wires and not variable names. This reduces the need for designers to explicitly keep track of the implicit linkages between operations.

Unlike text based languages, where all the parallel operations are grouped together, the VERTIPH notation separates out the specification of a pipeline so that the temporal relationship between the operations within a pipeline are made clearer to the developer without having to rely on secondary notations. The notation was designed this way to make it easier to detect the lack of buffering between stages. Unlike the text based notations cited above, the different stages of the pipeline are separated into their relative clock cycles by the notation, with the linkages made explicit through the connections between operations. The Handel-C code shown in Figure 8.5 has an error in its pipelining.

```
par{
     a = a +1;       //stage 1
     b = a << 1;     //stage 2
     c = b – 3;      //stage 3 two parallel
     d = b + 3;      //operations
     e = d << 1;     //stage 4 two parallel
     f = d + 2;      //operations
     g = e * f + c   //stage 5
}
```

Figure 8.5; A Handel-C pipeline with an error involving stages 3, 4, and 5

The code example shows an algorithm with seven operations organised into five stages. Stages 3 and 4 both contain a pair of parallel operations and the final stage receives data from those two stages. The output from block *c* is not used in stage 4, so it should have been temporarily stored in a register so that its entry to operation *g* in stage 5 is synchronised with the data from stage 4 operations *e* and *f*.

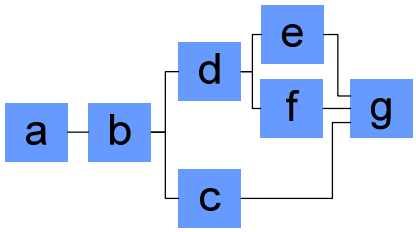


Figure 8.6: Graphical representation of the pipeline

With a simple dataflow diagram of the stages, as shown in Figure 8.6 it becomes easy to see that the output from block *c* needs to be registered (stored in a register) because it is required by block *g* two clock cycles later, and not in the next clock cycle. A dataflow diagram allows this to be identified quickly and makes the required fix of adding a register obvious. VERTIPH flags these errors by changing the wire colour and adding a '!' to it (Figure 8.7).

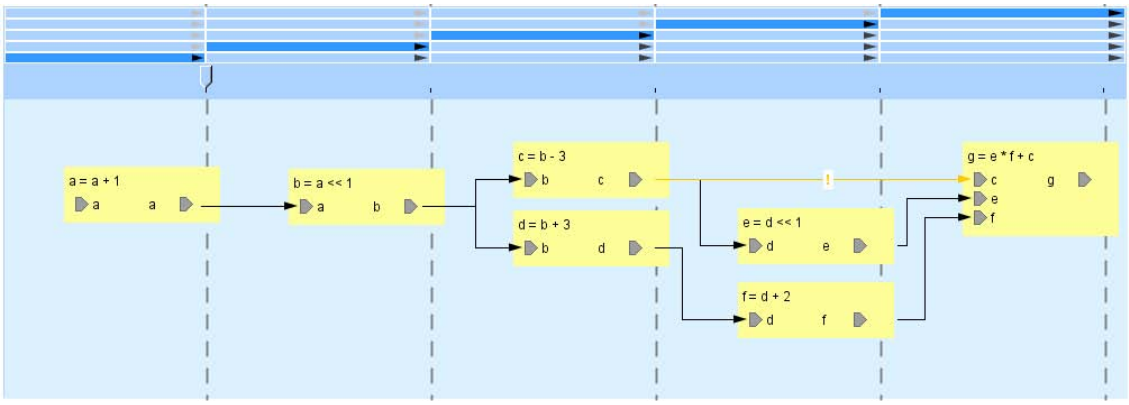


Figure 8.7: VERTIPH representation of the pipeline

The pipeline notation (Figure 8.7) also makes latency explicit. The hardness of the mental operation of determining the relative position of operations in the pipeline is less with this notation than it is with the other two languages cited above. Since designing a pipeline is an inherently challenging task, no notation can make pipeline design trivial. However, the pipeline notation can make pipelines easier to define.

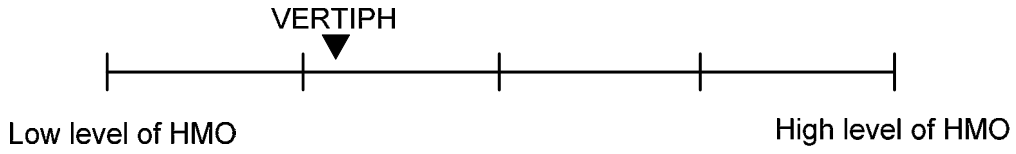
There are a number of considerations to take into account when designing multiphase pipelines. The Handel-C example used in chapter 5 illustrated the difficulty of breaking up the design into multiple phases with conditionals; the same problem arises with VHDL. By using a sequential representation and abstracting the phasing away from the developer VERTIPH makes the mental operation easier than in current languages. It also allows multi-clock cycle operations to be added to a pipeline and checked to see that they can operate within the multiphase pipeline.

Pipelining is not the only image-processing-on-FPGAs operation that many languages make mentally hard. Scheduling is another area where the syntax of many languages creates an HMO. The semantics of this task - like pipelining - are inherently complex. However, most languages propagate the difficulty unnecessarily into their syntax, by only providing low-level control structures via conditionals (although these are often augmented with semaphores, channels and other parallel communication mechanisms). Working with low-level control structures can lead to harder mental operations; this was one of the main reasons for the construction of the scheduling view which allows operation to be scheduled globally (the other being to make it easier to identify resource conflicts).

The scheduling view can lead to hidden dependencies. For example the computational block can be flagged as controlled by a *when* condition which is active during a particular state (or states). In this case, the state machine needs to be inspected to see how the different computational blocks interact with each other, as it

is not obvious from just inspecting the computational block view. The view can also lead to premature commitment, as the breaking up of the design into schedulable parts needs to be made early in the design.

VERTIPH therefore has a relatively low level of HMO.



CD6: Hard Mental Operations

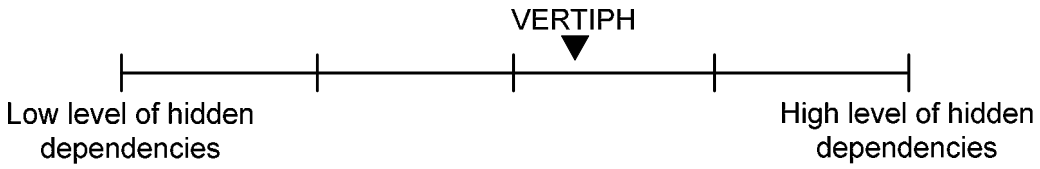
8.3.7 CD 7: Hidden Dependencies

A hidden dependency occurs where there is a relationship between two components such that one is dependent on the other, but the dependency is not fully visible. This can lead to unexpected side effects when a change in one part of the notation leads to a change in another part. To avoid this, both components should show the dependency or the user should be warned when one change will affect something else that it is not visibly connected to.

In VERTIPH, type declarations are inherited by the variables. A change to the type of one variable will affect all other variables that have this type. As the type does not list the variables that have that type or, in the case of complex data types, are based on it, there is a hidden dependency. This hidden dependency also affects junction boxes, which connect the inside of architecture blocks to the outside terminals. When a variable's type is changed, the junction boxes through which it passes will need to be modified to take into account the type change, as the mapping between types will now be incorrect.

As discussed above, the use of the scheduling view with computational blocks does not fully expose the dependency.

Hidden dependencies do exist in VERTIPH, though where possible the author has tried to avoid creating them or attempted to mitigate their effects. In the case of the scheduling view, linking is allowed between a controlled operation and its state machine via the *when* control structure and vice versa. The hidden dependencies in type declarations can be fixed with the use of a search to find affected junction boxes and variables. This is similar to a refactoring in modern OO IDEs. This is yet to be implemented in the prototype.



CD7: Hidden Dependencies

8.3.8 CD 8: Premature Commitment

This occurs when the developer is forced to make a decision before all the information is available to make it. This occurs when (a) the notation contains internal dependencies, (b) the environment constrains the order in which information is defined and (c) that order is not appropriate. The effects of premature commitment can be mitigated if decisions can be reversed or corrected later. In visual programming languages, premature commitment includes:

- Commitment to layout

  the programmer cannot move or it is difficult to move a component once it is placed;

- Commitment to connections

  the programmer cannot change connection locations or types;

- Commitment to creation order

  the interpretation of a program depends on order in which entities are created;

- Commitment to choice of construct

  the programmer cannot change an construct, such as changing a *while* to a *for*.

Developers can move components once they have been placed. This means that commitment to layout is not a problem in VERTIPH.

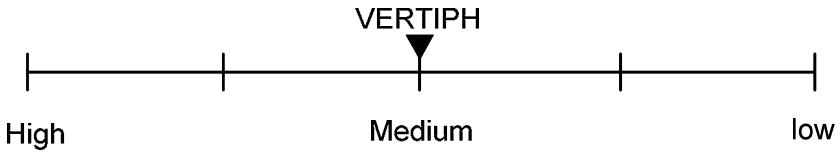
There is a commitment to connections as there is a fixed side that ports occur, inputs on the left, outputs on the right.

Creation order does not alter how the design operates. However, as identified during the paper-based user evaluation, there is the possibility of breaking up the design of the system incorrectly at the architecture level, requiring it to be re-implemented. This can be resolved later on, but it is more difficult to do so. There is also an issue with the location at which image processing operations are defined

within the design. For example the object-tracking program pipelined many of the operations at the computational level. If these operations were defined as separate architecture blocks then the pipeline notation provided by the computational view could not be used, because each operation would be defined in a different computational block. The user is therefore committed to perform the pipelining at the resource and scheduling level, which may be more complex (as this view does not operate at a clock level pipeline), and this is especially true for complex multiphase pipelines which are not supported in the scheduling view.

There is a commitment to the construct being used. To change the type of construct, a new one needs to be created and the operations from the old one copied to it.

VERTIPH therefore has a medium level of premature commitment.



CD8: Level of premature commitment

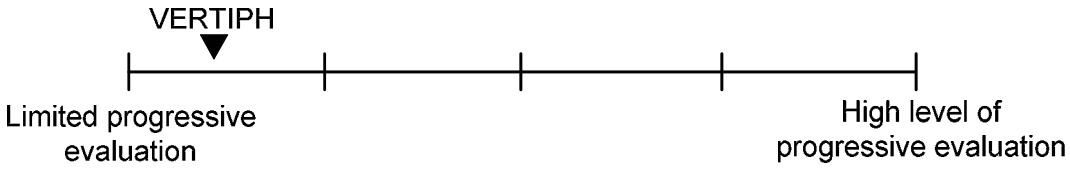
8.3.9 CD 9: Progressive Evaluation

This considers how the environment supports the evaluation of partially finished programs so that feedback can be obtained on the incomplete solution.

As discussed earlier, the design process for FPGAs is not the same as for software design. The progressive evaluation phase is quite different, as the design will be mapped to hardware. There are several choices, functional simulation, clock-accurate simulation and testing the design on the FPGA. Functional simulation is quick but may not be accurate. Clock-accurate simulation is very slow, typically operating at Hertz speeds on a modern processor, making it almost impossibly slow. Testing the design on an FPGA (assuming stand-alone operation) is difficult as there is no operating system to support debugging operations (Bailey et al., 2006). Not only does the part of the design that needs to be tested have to be created but it is also necessary to construct code to test it and to display the result.

This all makes progressive evaluation less valuable than with conventional programming languages. The output and test requirements will be different depending on the stage of the design. The feedback available is also limited compared to what is available on a PC.

It is possible to construct part of a design, compile it and implement it on an FPGA to verify that the design so far has been implemented correctly. A developer can use an incremental develop-and-test approach to perform progressive evaluation. However, it is over to the user to decide appropriate points to test, and also provide any additional tools required for the evaluation, such as test generation and testing and displaying of results.

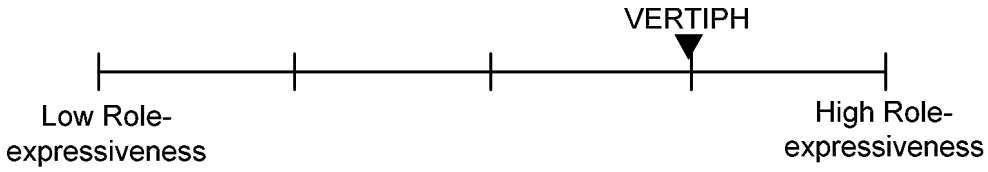


CD9: Progressive Evaluation

8.3.10 CD 10: Role-expressiveness

This is intended to describe how easy it is to understand the role of each language component and how it fits into the solution as a whole. This definition is vague, as identified in (Green et al., 2006), as it is not possible to make a simple cognitive model of role expressiveness; it is easy to illustrate good and bad role expressiveness but hard to give guidelines to spot them. It should also not be confused with closeness of mapping.

As discussed in the relevant chapters, each of VERTIPH's three graphical representations is different from the others. Though they have some similarities, they each contain components which are only relevant to the particular aspects of the design in that view. Each of these components has been designed for the particular task and each view is designed to complement the other ones. For example, the computational view has a very different layout structure from the architectural view to reflect parallel and sequential components and the specialised control structures which exist in only in the computational view. The architectural view ties all the different parts into one whole solution. This makes VERTIPH quite highly role expressive.



CD10: Role Expressiveness

8.3.11 CD 11: Secondary Notation and Escape from Formalism

Secondary notation can be used to carry extra information that is not part of the formal syntax. This includes layout, indentation and statement grouping, colour, commenting and naming conventions. This is true in text and visual diagrams. In schematic entry, a good layout (with clear directions of flow, conventionally from left to right and top to bottom) is easier to understand than a bad one.

Unlike some other visual systems, in its computational view and in the scheduling panel of the scheduling view, VERTIPH constrains the developer as to where components can be placed, as their position conveys semantic information about their timing, and whether they occur sequentially or in parallel with each other. Using the position to provide more information reduces the need somewhat for secondary notation. For example, it is used in the computational view to provide more information about pipelines. The other views allow for a more free-form visual structure which open the way for secondary notations.

Each view has its own commenting mechanisms (such as allowing comments in the vertical bars of control structures), and developers are free to adopt naming conventions for data connections.

CD11: Secondary Notation and Escape from Formalism

8.3.12 CD 12: Viscosity

Viscosity is a measure of the notation's resistance to change, the amount of effort that is required to make a change to a program expressed in the notation.

This dimension can be further classified into the following types :

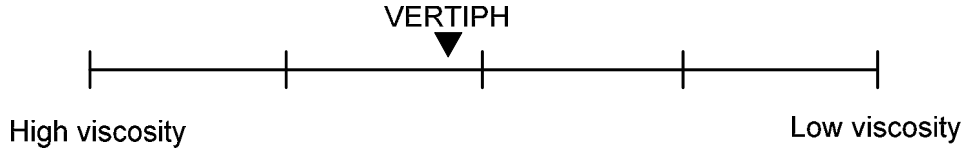
- 'Knock-on viscosity': a change in the code violates internal constraints in the program, whose resolution may violate further internal constraints. An example given by (Blackwell and Green, 1998) is the insertion of a new figure in a document, requiring all later figures and their cross-references to be updated.
- 'Repetition viscosity': a single action within the user's conceptual model, such as matrix multiplication, requires many repetitive device actions.

- 'Scope viscosity': a change in the size of the input data set requires changes to the program structure itself. Programs that are set up to store a data set in a fixed size array suffer from scope viscosity.

Scope viscosity is minor in VERTIPH; if the image size changes then some changes may be required, (buffer sizes, and some word widths), to reflect the new image size, but there should be no change needed in the program's structure. There may be flow on effects that relate to the efficiency of the design or the use of resources. A small change may have a large effect on resources. An example of this would be increasing the size of a row buffer from 512 pixels, which fits perfectly into a block RAM, to 513 which would require a new block of RAM (double the space for a small change). While this is not a language or image processing issue, it is a hardware design issue. Issues such as this will depend on the type and size of the FPGA to be targeted so the developer also needs to consider them when making the design.

With this design the author has attempted to reduce repetition viscosity. The scheduling view aims to remove the need to change several low-level control structures by having a higher-level event-driven control which can schedule whole processing blocks. Repetition and knock-on viscosity do occur in the architectural terminal types and junction boxes; changing a type of a terminal will require all the connected types, and the junction boxes related to these, to be modified.

As previously described, changes in type affect multiple variables. One of the reasons for using types is to reduce the knock-on viscosity associated with changing the data width of a variable. This is more of an issue with hardware design than software because the bit width is more likely to be optimised to the function, rather than being one of a small set of standard sizes. Appropriate use of types can allow a single change to be made (to the type) and the rest of the design automatically adjusts to the new bit width. By not allowing connections between different types in the interfaces between architecture blocks, knock-on viscosity is reduced. A change in one part of the design will only affect the other parts of the interface with a change in how the data is processed or presented. If the function of a block is changed then, as in other languages, this change will have a knock-on effect on the rest of the design.



CD12: Viscosity

8.3.13 CD 13: Visibility

Visibility is a measure of whether required information is accessible without cognitive work; it measures how many steps are required to make an item visible. This includes how readily the notation elements can be identified and how parts can be accessed and made visible.

The example of nesting used in (Green and Petre, 1996) where it can be hard to see what is nested within what and to identify alternative steps. This has been improved by the use of horizontal and vertical bars which can show the nesting more clearly as illustrated in Figure 8.8, where one *if-else* control is enclosed by the true condition of another *if-else* control.

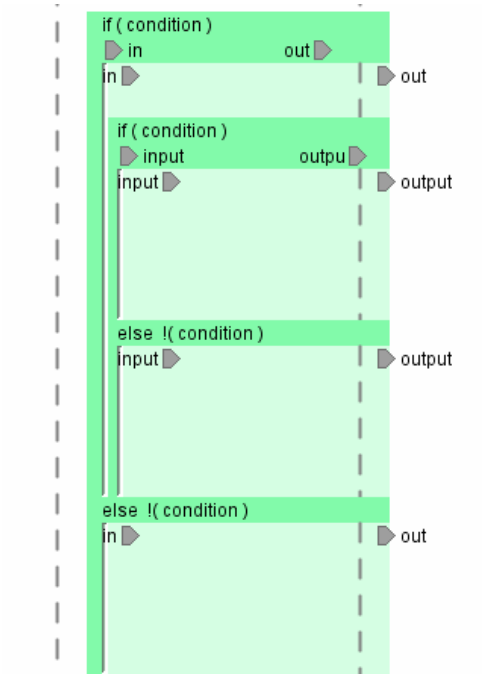


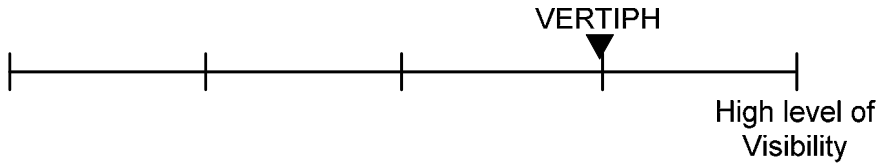
Figure 8.8: Nested *if-else*

The different control structures and expressions have different shapes and colours as illustrated in the computational view chapter.

The pipeline view has been designed to make the important information, latency, the pipeline stage of an operation and the data flow, visible to the developer.

The architectural view can have many sub–blocks, both architectural and computational. This can lead to situations where many windows need to be opened before the part that the developer wants to see or edit is reached. This reduces visibility. However, VERTIPH trades off reduced visibility for greater abstraction. Currently it is easy to navigate from a parent to a sub-block, but not from a sub-block to its parent. This deficiency can be addressed (as part of future work) with the

addition of a tree view similar to the project–package–class tree views that are found in many modern IDEs.

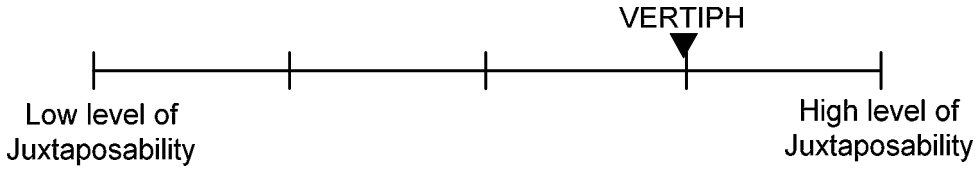


CD13: Visibility

8.3.14 CD 14: Juxtaposability

This evaluates whether two of more parts of the notation can be compared side-by-side at the same time.

VERTIPH rates high in this dimension as its tabbed windowing system is derived from the NetBeans IDE (NetBeans, 2008); in this IDE it is possible to put two or more parts of the notation side-by-side, dock and undock tabs and rearrange them in many different ways. However, there are some minor restrictions on juxtaposability; some of the elements cannot be viewed side by side. For example, it is not possible to view more than one type editor panel at a time.



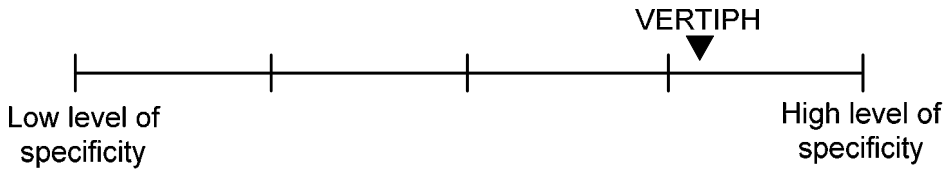
CD14: Juxtaposability

8.3.15 CD 15: Specificity

The notation uses elements that have a limited number of potential meanings (irrespective of their defined meaning in this notation), rather than a wide range of conventional uses (Blackwell et al., 2001).

VERTIPH uses elements which have a limited number of possible meanings to make their use more intuitive; however, it is also desirable to avoid too may different objects in our notation. This has led to using one graphics object to represent several similar objects. The control structure notations use the same notation for a *while* and a *for* loop, as they are quite closely related, although the *for* loop has both the condition and the step (counter) function when the *while* loop only has the condition. The *if-else* and *switch* elements are closely related (a *switch* can be seen as an extension of a *if-else,* one that provides more options without the need for nesting), so they share a similar object. The notation does not use many elements that

have a meaning other than those unique to VERTIPH, with the exception of the state machine diagrams of the scheduling view. This makes VERTIPH rate high for specificity.

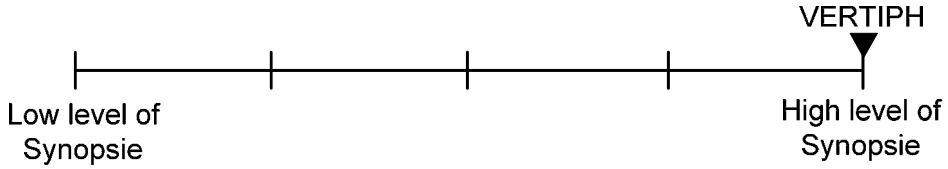


CD15: Specificity

8.3.16 CD 16: Synopsie (originally "grokkiness")

The notation provides an understanding of the whole when you "stand back and look" (Blackwell et al., 2001).

VERTIPH's notation was designed to enforce a top-down design process, with each phase of design in a different view. The top level architectural view provides a high level overview of the complete design. However, as with any complex design, many of the important details are hidden. Having a hierarchy of architectural objects allows the developer to get a good overview of parts of a complex design. In this regard, VERTIPH rates high in the synopsie dimension.

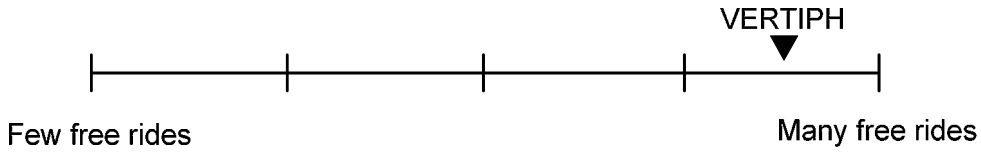


CD16: Synopsie

8.3.17 CD 17: Free Rides

New information is generated as a result of following the notational rules (Blackwell et al., 2001).

This is relevant to the computational view; by following the placement rules for sequential, parallel and pipelined operations, information about the latency, total operation time and possible resource contention becomes significantly more visible than in conventional HDLs. The example for this is the pipeline notation and the layout expression in it.



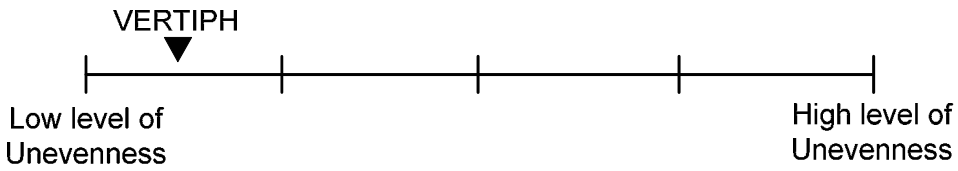
CD17: Free Rides

8.3.18 CD 18: Unevenness

A language that forces the developer in a particular direction is bad, because the developer may not consider certain possible types of solution, which might be better than the one chosen, because it is easier to program a suboptimal solution (Stacey, 1995).

Conventional HDLs are uneven, because they provide poor support for designing pipelines, whereas VERTIPH has overcome this unevenness by making pipelines as easy to design as, say, sequential datapaths. The multi-phase notation is particularly relevant to this. In most HDLs, multiphase pipelining is difficult to define. When a change such as inserting a stage is made in the early stages of the pipeline, it can require all downstream tasks to be rescheduled. In VERTIPH's notation, the complexity of controlling the pipeline's stage scheduling is removed from the developer. This makes multiphase design easier and therefore more likely to be used.

This means that VERTIPH has a low level of unevenness.



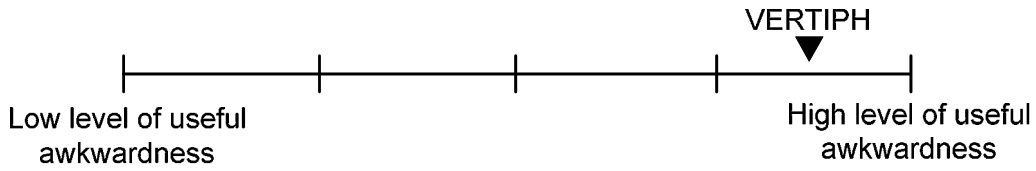
CD18: Unevenness

8.3.19 CD 19: Useful Awkwardness

It's not always good to be able to do things easily. Awkward interfaces can force the user to reflect on the task, with an overall gain in efficiency. This is the other side of unevenness (Blackwell et al., 2001).

VERTIPH aims to encourage developers to use a design methodology rather than make things more awkward. However, the architecture view makes it awkward to design a system from the bottom up, as it requires the root architecture block(s) to be defined before their computational instructions can be defined. It was found during the paper-based user evaluation that this feature was either hated or liked by

developers. The most experienced developer thought it was very good as it made him think about how best to break up the design into sensible components. Others found it restrictive as they wanted to explore the design space first, getting the low-level functionality worked out before creating the higher level components. This needs further investigation.



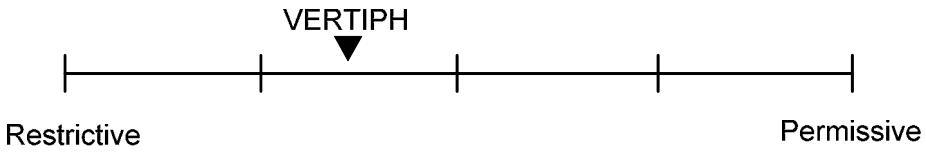
CD19: Useful Awkwardness

8.3.20 CD 20: Permissiveness

Permissiveness is a measure of the number of different ways the notation allows to do different things (Blackwell et al., 2001).

Much of VERTIPH is not very permissive. However, the types and their representations are quite flexible, as are the control structures. Types can be defined as either complex or basic. Complex types reduce the number of wires needed between architecture blocks, although this often comes at the cost of the need to break them up in the junction box to reduce them back to basic types for processing in the computational view. Links do not need to be complex, so if the developer wishes, several wires can be used instead of the one combined complex type.

The operations can be scheduled several ways depending on the control structures used. One way is to use state machines in the scheduling view to respond to events at a global level and control the operation of components. The other way of achieving the same end is to use the low level structures (*if-else*, *while* etc) within the computational view. As these were designed for data dependent decisions, the events need to be passed between all relevant architecture blocks. This may be fine for simple applications but does not scale well for large designs. However, the final choice is left up to the developer. Care is required when mixing the two types of control (for high level control) as this can potentially lead to conflicts if, for example, one control turns a block on while the other is turning it off.



CD20: Permissiveness

8.3.21 Cognitive Dimensions Summary

This detailed evaluation was conducted after VERTIPH was fully defined and the prototype implemented.

CDA picks out and focuses on different aspects of the language's design. Trade-offs between different cognitive dimensions exist and these trade-offs were considered when developing VERTIPH. In VERTIPH having a top-down design approach requires developers to look ahead when making a design, which can increase premature commitment. Creating different views of the design increases the number of abstractions required in the design. This has made parts of the design more visible by having a specialised view tailored for a specific part of the design. Having multiple views has allowed VERTIPH to have a high level of role-expressiveness.

Having a defined data types which can be modified in one place and affect all usages has the advantage of reducing the knock-on viscosity, but it can also introduce a hidden dependence. The developer only needs to change a data type in one place which is good but they might need to change many junction box interfaces as a consequence.

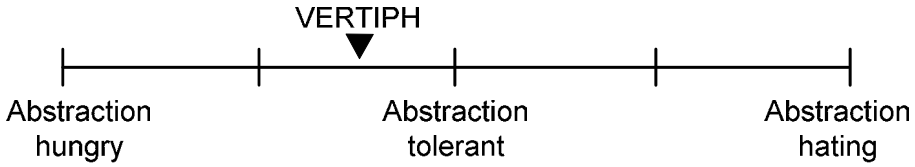
VERTIPH requires abstractions but is not abstraction hungry, as only a small number of abstractions is required, such as architecture blocks and data types.

VERTIPH's computational view is close to the low level image processing algorithm design. This allows the pipelining and control of the algorithm to be shown more explicitly. Its visual representations are, where possible, consistent and it tries to a create compact and easier to view design (a balance between being diffuse and terse).

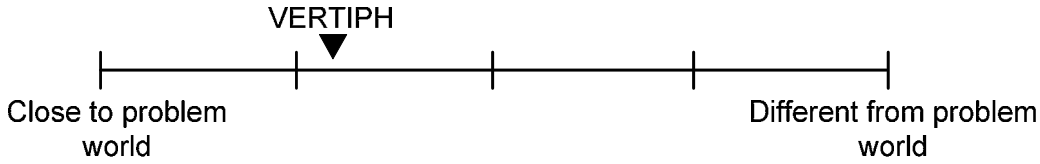
Pipelining has been identified as one of the main parts of image processing on FPGA design that can lead to errors; another is control expressions. Both of these have been improved over existing tools in VERTIPH's notation by providing more information to the developer. Pipelining is also a hard mental operation in most HDLs and it has been made less so in VERTIPH.

When evaluating VERTIPH, most parts can be viewed side by side (the juxtaposablity for most parts of VERTIPH is good) except for data types and junction boxes; these need refinement. When looking at the visibility of VERTIPH it was found that there was no way to go back up the tree hierarchy from the lower levels of the design. Neither of these was commented on by user evaluators even though they are limiting omissions in the design.

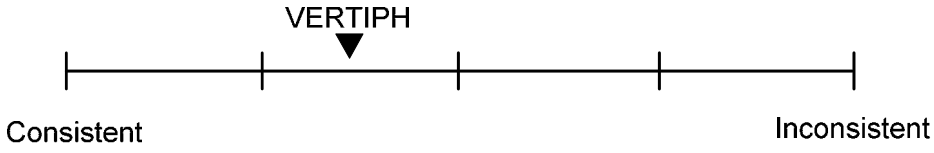
The computational view encourages the use of pipelines and parallel structures due to useful awkwardness which pushes the developer towards the streamed processing (pipelined) design approach that was developed. This was not a conscious intent but is a consequence of the design of the language being based around our research groups design process, which concentrated on stream processing due to the efficient mapping onto FPGAs it can produce.



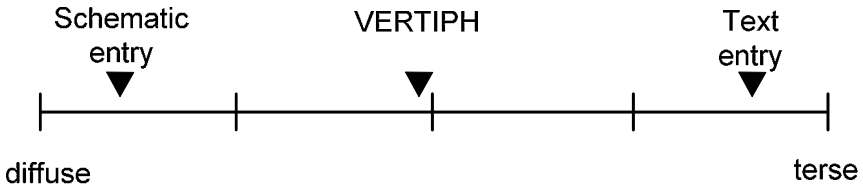
CD1: Abstraction Gradient



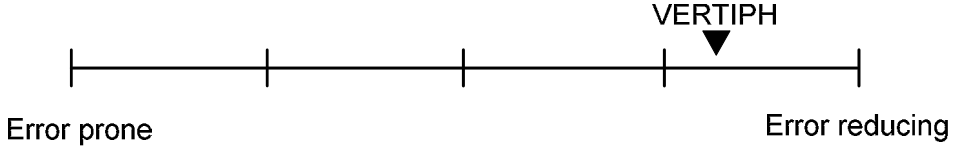
CD2: Closeness of Mapping



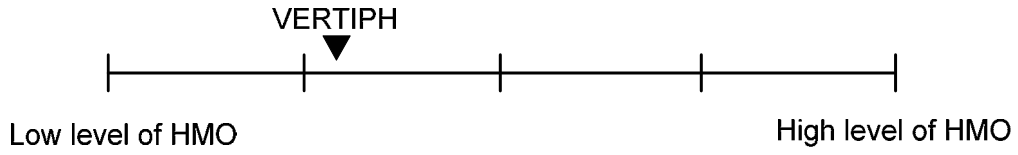
CD3: Consistency



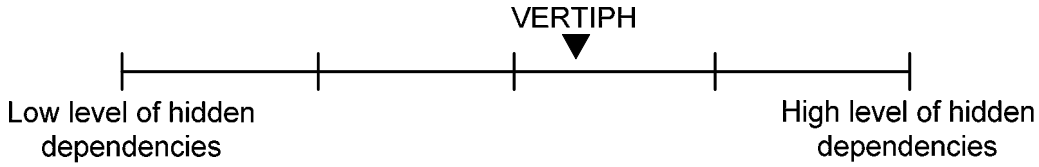
CD4: Deffuseness/Terseness



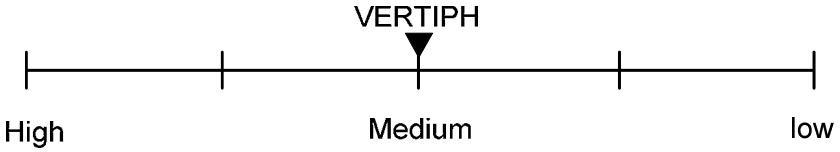
CD5: Error-proneness



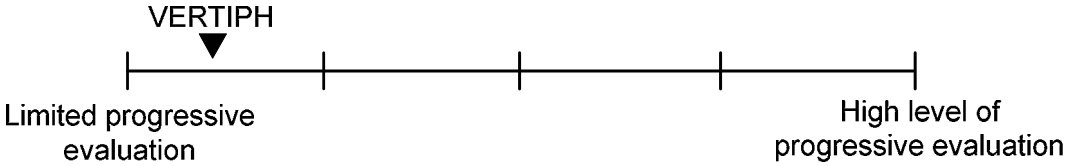
CD6: Hard Mental Operations



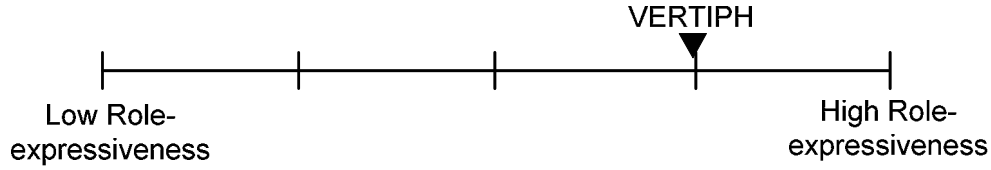
CD7: Hidden Dependencies



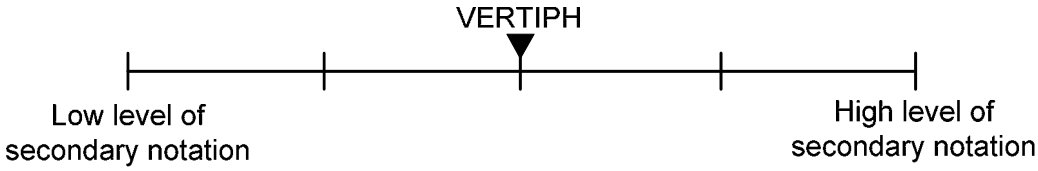
CD8: Level of premature commitment



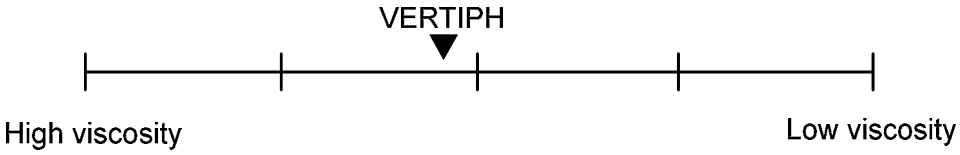
CD9: Progressive Evaluation



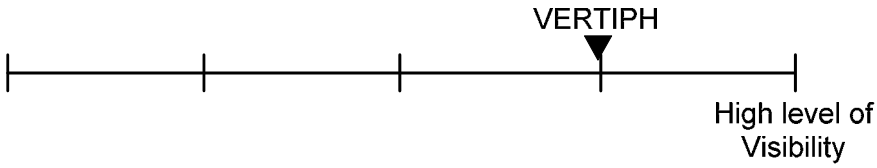
CD10: Role Expressiveness



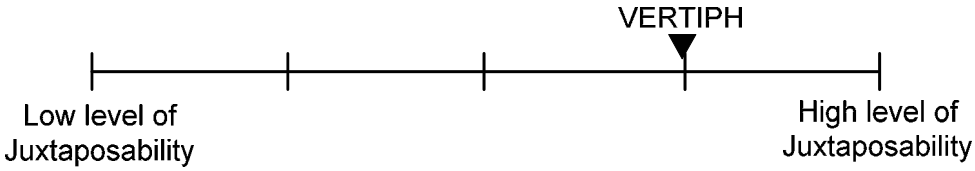
CD11: Secondary Notation and Escape from Formalism



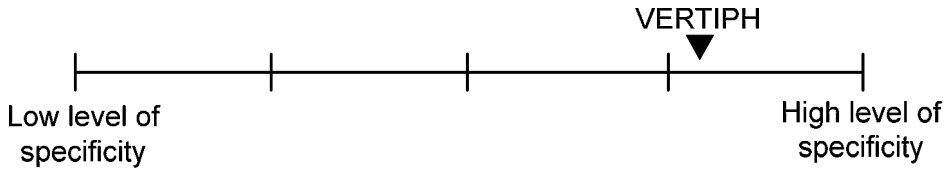
CD12: Viscosity



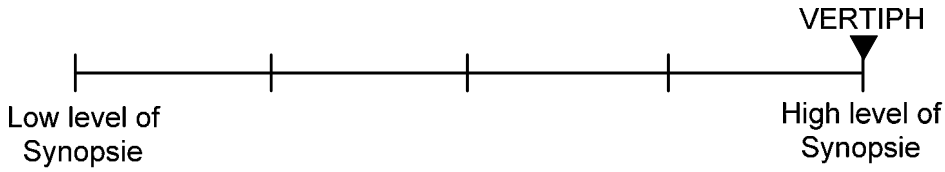
CD13: Visibility



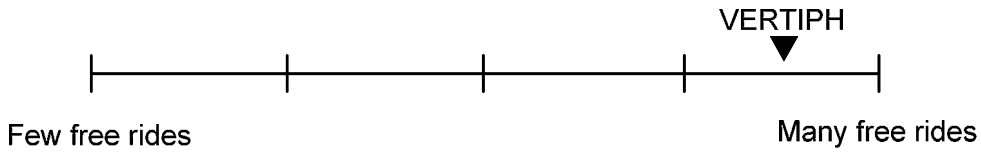
CD14: Juxtaposability



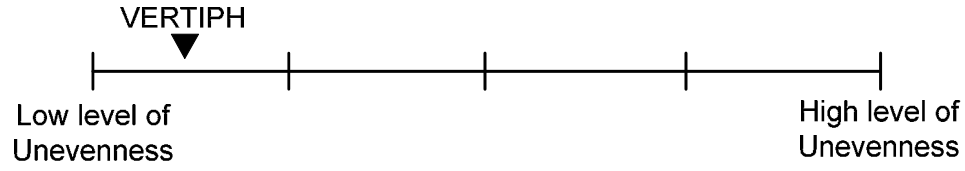
CD15: Specificity



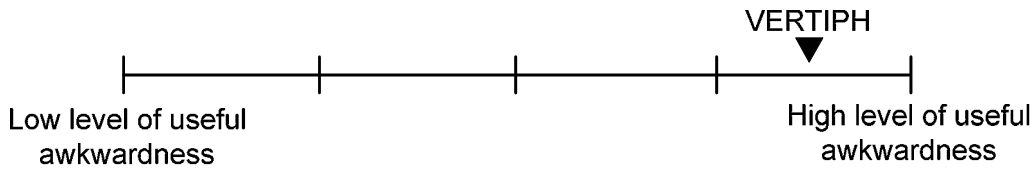
CD16: Synopsie



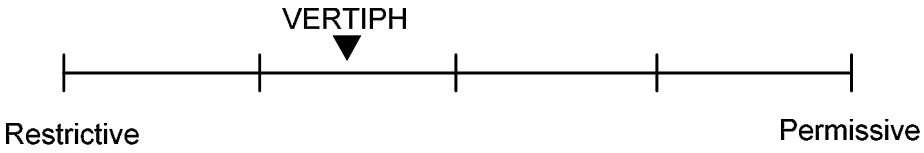
CD17: Free Rides



CD18: Unevenness



CD19: Useful Awkwardness



CD20: Permissiveness

8.4 Discussion

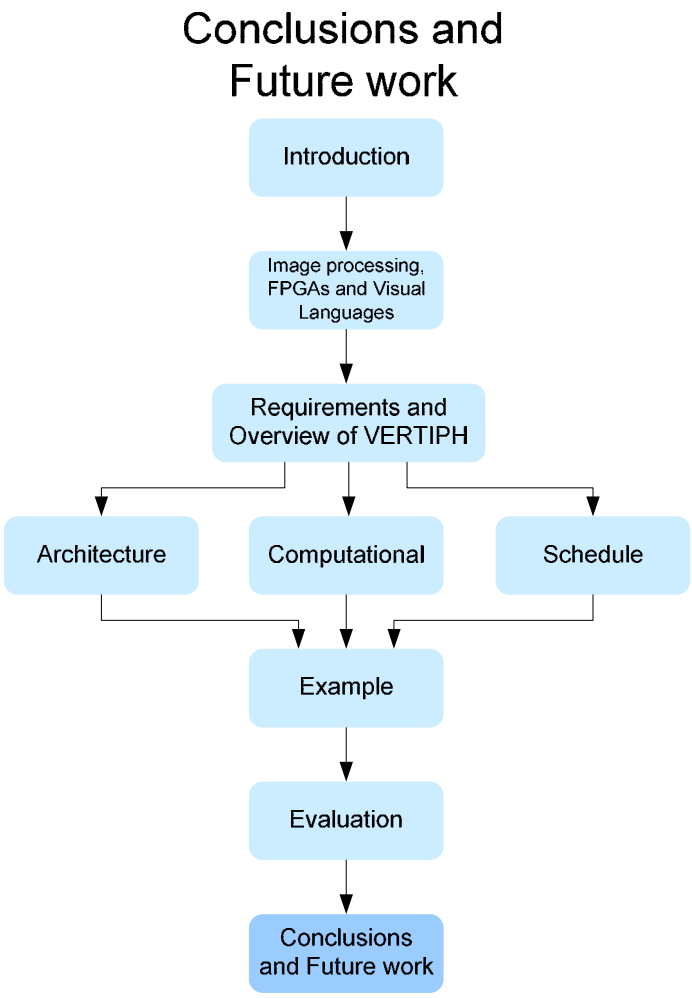
The user evaluation feedback was valuable. The most experienced developer had many useful suggestions for extending VERTIPH. These will be discussed in more detail in future work. The architecture block view was viewed very positively. However, at the lower level developers would like to use more text, possibly because this is what they are used to or it could be a more efficient means of entering the details than editing each expression block. They would however, like to keep the computational expression view as a visualisation of the expressions. Most of the participants found that the control structures were useful and they liked them. The clock layout and pipelining was also considered useful, though generally they would prefer this as a visualisation, rather than editing the diagram directly. The major criticism was of the junction box and this can be fixed by automating the creation of matching I/O terminals between the levels.

As the scheduling view has not been implemented, the CD analysis has been the primary evaluation of this part of the language. The analysis showed that it is necessary to minimise the impact of potential hidden dependences between the scheduling view and the other views. The CD analysis also identified some problems not shown in the user evaluation. Though users did not comment on it, there needs to be a way to navigate up the hierarchy (from a sub view) as easily as down. The ability to view and edit more than one junction box or data type at a time (juxtaposability) also needs to be added.

For the many of the dimensions, VERTIPH was not at either extreme. The only dimension where VERTIPH was at a negative extreme was the low level of progressive evaluation, which is more to do with the nature of FPGAs and the lack of a simulator design, than the language itself. There are many dimensions where VERTIPH is near a positive extreme. These tended to be related to making details of the design clearer to the developer. These dimensions are the high levels of: free rides, synopsise, visibility, role-expressiveness, error reduction, specificity, juxtaposability, useful awkwardness and low levels of hard mental operations. This is not surprising as making information more visible to the developer especially about

timing, parallelism pipelining, scheduling and resources was one of the main design goals of VERTIPH.

VERTIPH's current design and partial implementation comprise a good first step at exploring the use of VPLs for use in "programming" modern FPGAs. The majority of participants involved with the evaluation thought that VERTIPH would be useful with further refinement, including improvements with the interaction (especially the balance between text and visual notations at the lower levels) and once it can generate optimised FPGA "code".


Conclusions and Future work
Introduction
Image processing, FPGAs and Visual Languages
Requirements and Overview of VERTIPH
Architecture
Computational
Schedule
Example
Evaluation
Conclusions and Future work

9 The Final Countdown

9.1 Conclusions

In order to explore the proposition that

> it would be possible to invent an alternative type of design tool that would make it easier to understand and map the design of image processing algorithms onto FPGAs than existing tools.

this research has attempted to answer the following five questions:

- Can the design process for implementing image processing algorithms on FPGAs be characterised well enough to act as the inspiration for a software tool?
- Is it possible to produce a detailed and complete syntax for a visual design tool based on such a characterisation?
- Is it possible to build an alternative tool based on these analyses?
- Is this tool easier to understand than existing tools?
- Is it easier to map a design onto an FPGA with this tool than with existing tools?

Can the design process for implementing image processing algorithms on FPGAs be characterised well enough to act as the inspiration for a software tool?

There are many ways to implement image processing applications on FPGAs, and the design flow is often directed by the tool chain used. Some of the high-level compiler systems have no user input into the design of the architecture.

Chapter 2 described the design process employed within the Image and Signal Processing Research Group at Massey University, which is aimed either at low-level HDLs or at HDLs that extend high-level languages. In this design process, the designer starts by choosing and implementing, in software, an algorithm that satisfies the requirements of the image processing task. Next, a hardware architecture tailor-made for this implementation is designed and the algorithm and architecture are transformed into an FPGA design. This is then optimised for speed, hardware usage and functionality. It was found that in most cases the most efficient mapping for image processing algorithms (in terms of memory bandwidth and hardware) is to use

a streamed processing approach, as this reduces the need for reading to and from memory. However, it constrains how data is presented to the algorithm and this requires most existing algorithms to be extensively modified. The connected component algorithm is an example of this. Therefore, the resultant designs are heavily streamed orientated, which implies the need for pipelining (to meet timing constraints). Both streaming and pipelining can be well represented by data flow visual languages which lead to the design for the computational view.

As image processing algorithms normally contain a number of complex steps which pass data between each other, it is necessary to be able to see the algorithm as a whole. Algorithms also need to be decomposed into logical units to group related processors or to allow processors to be scheduled together. These requirements lead to the design of the hierarchical architectural view. As many algorithms require both low-level control based on pixel data, and high level control (scheduling) based on events the scheduling view was developed.

In chapter two it was found that there has been a move away from low level HDLs to higher level programming languages. Some, such as SA-C and Match, constrain the developer to a particular design methodology. Others, such as Handel-C, and SystemC do not give good representation of the parallel nature of the design due to their textual representation. Current image processing visual languages are good at specifying the steps of an algorithm, but they do not allow the lower level details to be designed in the same system. They instead rely on a large set of pre-built operations and either do not allow the VPL to be used to develop new one or require them to be made up of available library functions. This is not ideal for FPGA implementation due to the difficulty in optimising the design. Nor do current languages separate the design of parallel and pipelined operations. This gives a justification for the work on developing a new VPL for image processing on FPGAs.

This research has investigated a design process for implementing image processing algorithms (within the embedded image processing domain) onto FPGAs and then analysed the process for a given image processing algorithm to characterise what a tool should have to support this design process.

Is it possible to produce a detailed and complete syntax for a visual design tool based on such a characterisation?

This design process and an earlier unrefined process were used to develop a number of image processing algorithms (see chapter two). These implementations

were used as case studies in developing and refining the design process and to exemplify what is required in a visual design tool to implement image processing on FPGAs. In chapter two, the detailed requirements of a number of problems were analysed and several syntactic features were identified. In particular, a tool 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 by informing the designer of potential conflicts and provide suggestions as to how to resolve the issue;
- be able to handle stream, random access, and hybrid processing models
- include some of the common image processing functions and data types as primitives. Examples include row and pixel buffering, window filters, and lookup tables (LUT);
- provide multiple views on the design.

The prototype of VERTIPH allows for a mixture of parallel and sequential design and makes it clear to the designer what runs in parallel and what forms a pipeline through the computational view. It can be used to design streamed, random access and hybrid processing models. It provides multiple views on the design: the high level architectural view to give a global overview and the computational view for operation details. The computational view is based on a synchronous register transfer language approach to logic design. The required control is implied by the loop, branch and pipeline constructs enabling the developer to focus on the data transformation. Being a synchronous design tool, it would not be able to easily programme an asynchronous (without a global clock) design.

The prototype currently has row and pixel buffering implemented and simple LUTs; an editor for window filters has been designed but needs to be implemented. The management of resource sharing between concurrent processors is part of the scheduling view, which is designed but not implemented in the current VERTIPH prototype.

Is it possible to build an alternative tool based on these analyses?

I have built and tested VERTIPH, a data flow VPL which provides the features described above except for the filter editor and the high level scheduling and resource

sharing detection which are incorporated into the scheduling view. This view was not implemented because of lack of time.

Although the scheduling view has useful features, not having an implementation of this view does not compromise the conclusion greatly as the features of the view can be replicated by using the control structures in the computational view. This does requires more effort for the developer as the tasks are not automated. It also uses a visual metaphor similar to state machines which is augmented with scheduling features modified from the implemented computational view.

The notation used in VERTIPH has been used by the author when developing image processing algorithms which were then implemented using another HDL. This tested the notation to ensure that it could express all that was required of it. Once the prototype was constructed several of the algorithms which have been described in this thesis were implemented in it. The barrel distortion algorithm (as seen in the examples chapter) and the colour object tracking (used as an example in the evaluation) where both implemented in VERTIPH as was the second evaluation task; the rank filter. VERTIPH was able to implement these algorithms with more than one solution possible.

It is possible to build an alternative design tool for implementing image processing on FPGAs.

Is this tool easier to understand than existing tools?

and

Is it easier to map a design onto an FPGA with this tool than with existing tools?

These questions related to VERTIPH's usability were explored via a user evaluation and a cognitive dimensions analysis and the results were presented in detail in the evaluation section.

In the user evaluation the participants viewed the architecture view positively, finding it better than the wiring tools they were used to. They also liked the type-declaration interface. They found the control structures both easy to understand and use, with two of the evaluators preferring them to their existing tools. The computational view and pipeline control structure provided a good visualisation of the lower level design (with several evaluators preferring it to the RTL diagrams

which some IDEs can produce). The sequential, parallel and pipeline metaphors with some explanation were found to be easy to understand, if a little difficult to use at first as it was a "different way of working".

Two problems emerged with the VPL's ease of use as opposed to its comprehensibility (question 4). These were associated with the junction box and the editing of the lower level details. The junction box requires too many steps to connect between the upper and lower levels; this can be easily fixed by the auto generation of I/O pins between levels. The other was with the editing of the computational view. Most of the evaluators found that having to create equations in place and then edit them disruptive to their work flow or time consuming. The editing of the lower level details, equations, buffers etc. can be improved, though the use of a split screen to allow the editing of the equations as text or in the visualisation.

My thesis statement was that:

> *it would be possible to invent an alternative type of design tool that would make it easier to understand and map the design of image processing algorithms onto FPGAs than existing tools.*

VERTIPH is an alternative type of design tool as it uses a visual programming model to express image processing algorithm for FPGA hardware. It makes the separation of parallel, sequential and pipelined expressions clearer and easier to understand than existing tools. The architectural and control structures were also seen as easier to use. There are still several improvements that need to be made to VERTIPH, including the implementation of a compiler.

Therefore, this thesis statement is partially valid as the mapping from the language to a working FPGA design is not yet implemented.

Though VERTIPH's core application area is in the presentation of pipelined algorithms for use on FPGAs it has possible wider application. It can be used in many data streaming applications. This includes implementation on high end parallel computing, Cell processors and DSPs. Although the focus of this thesis was within the domain of image processing, VERTIPH could readily be used for more general signal processing. Although not explored here, VERTIPH would also be well suited to represent solutions to hardware implementation of parallel algorithms, especially algorithms that can be represented at a high level with a block structure.

9.2 Future Work

Due to time limitations, it has been necessary to stop with only two of VERTIPH's three major components implemented. Currently VERTIPH is at a stage where it can be considered as a prototype for testing key features of the language. The future work can be broken up into three man aspects: implementation of already designed but unimplemented features, features which were identified and planned for but were not expected to be implemented within this thesis research, and design and implementation of features identified as desirable by participants of the user evaluation.

The designed features that are still to be implemented include an improved expression editor, a filter editor, a look-up table designer, the scheduling view, and a compiler. Completing all of these features would result in a fully working version of VERTIPH. The expression editor requires the syntax checking of expressions and the automatic generation of inputs and outputs for expression blocks and the automatic generation of the output type given the input type and the expression. This is a desirable feature but was not required for a prototype.

As filtering can be accomplished by the use of the provided features (expressions and row buffers) it has not been implemented. However, the proposed filter editor has a number of features which would be desirable for an image processing practitioner, particularly to do with the effects associated with the edges of the image.

The scheduling view is designed for high-level control. However, low-level control can be used to control processing and this was all that was needed in the prototype. The scheduling view will probably require refinement as it is implemented and tested, as the paper based user evaluation provided limited feedback. The scheduling view was not originally planned in the initial thesis proposal and it was not until several algorithms had been implemented that the desirability and usefulness of a scheduling view was identified. This design (as described in the scheduling chapter) was then revised significantly as other examples were considered and through discussions with other members of our research team.

For any language to be practically useful, a compiler needs to be constructed. As this thesis focused on the design of a language from the developer's perspective, and due to time constraints, a compiler has not been implemented. However, the data structure that is created to support the visual components of the language has been designed to facilitate a recursive descent compilation. Building a compiler to translate

a high level language to an FPGA design is a complex task; a basic compiler for converting VERTIPH to an intermediate language such as VHDL is probably enough work for a masters project, and an optimised compiler would make a good topic for a PhD.

It was identified at the start of my thesis that a simulator would greatly speed up the design process and would allow VERTIPH to be used as a software implementation and test tool as well as a HDL. It was thought that this would take a considerable amount time to implement and was discarded as being a nice to have but unlikely to be implemented within the time required. Making a good simulator based on VERTIPH which can be used both for functional and clock accurate simulation would be a good future research topic.

As identified in the cognitive dimensions analysis, there is currently no way to go back up the hierarchy from a sub-block. Even though no user participants identified this as a problem it needs to be added to VERTIPH.

Several participants thought that the editing of expressions by typing them in as text then using the graphical computational timing editor to move them to the correct locations would be useful. A similar idea to this was discussed earlier on in the design stage but was discarded as it would require two separate tasks to complete a design and it was felt that having the editing and location in one step was more efficient. However, many participants stated that they would like to be able to quickly type out the equations they needed then think about when they occur; they felt it was more efficient to do it this way. This would also involve the automatic wiring of expressions based on I/O names. This change would require some new features to be added to the computational view, allowing developers to use either the current method or the text and visual method. A user evaluation would also need to be conducted to compare the two methods. This is an interesting idea and allowing both methods is probably desirable to allow developers to use whichever method they prefer.

There is also further work to do on the development process for the mapping of image processing to FPGAs. The process described in this thesis is interactive but requires a human in the loop with knowledge of both the possible algorithms and possible architectures. Humans currently are better at the higher level design decisions but as shown with (Banerjee et al., 2000, Benkrid et al., 2002c, Najjar et al., 2003) lower level and some higher level optimisations can be made by an intelligent compiler. It would be interesting to attempt to formally characterise the process that is followed by humans and attempt to replicate it semi-automatically or even

automatically. This is a difficult task as the tool needs to understand not just the steps but the design as a whole, substituting operations which have the same or similar (a subjective task) function but different hardware mappings. Currently, humans need to rely on experience, skill and looking at how they can modify the design as a whole (not just the individual steps) to produce an optimal result. The efficient transformation of algorithms into parallel computational architectures is a challenging topic that would provide material for a number of PhD theses.

10 Appendix I: Paper based evaluation

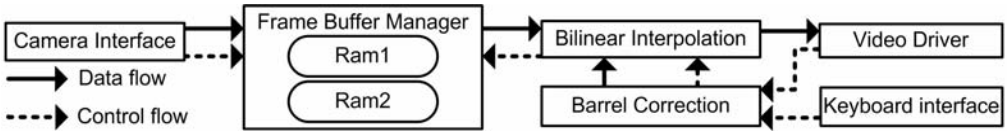
This appendix contains the introduction to VERTIPH, tasks and the questionnaire that was used in the paper based user evaluation.

10.1 VERTIPH paper based user evaluation

VERTIPH is a visual programming language designed to aid in the development of image processing algorithms on Field Programmable Gate Arrays (FPGA).

VERTIPH has three views that are used to specify a programme.

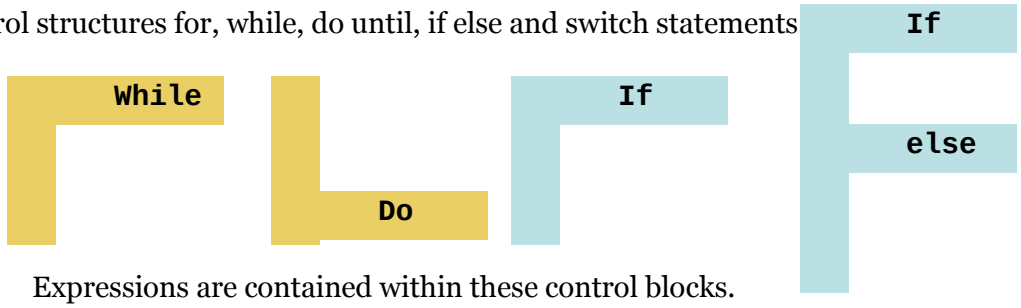
The first is the Architectural view this is designed to be a high level block diagram showing the parts of the system required to achieve the desired image processing task. It has blocks and the data paths (and control) between them. Each block in this view can contain either another Architectural diagram or a Computational diagram.



Example of an Architectural diagram

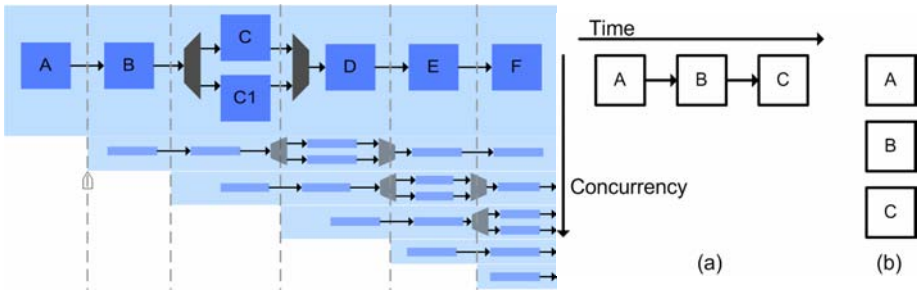
This from of encapsulation allows for a top down design methodology to be followed.

The Computational diagram shows what steps need to be taken to perform the desired operation. Modified Nassi-Shneiderman diagrams are used to represent the control structures for, while, do until, if else and switch statements



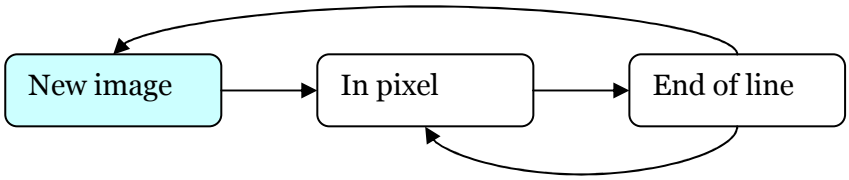
Expressions are contained within these control blocks.

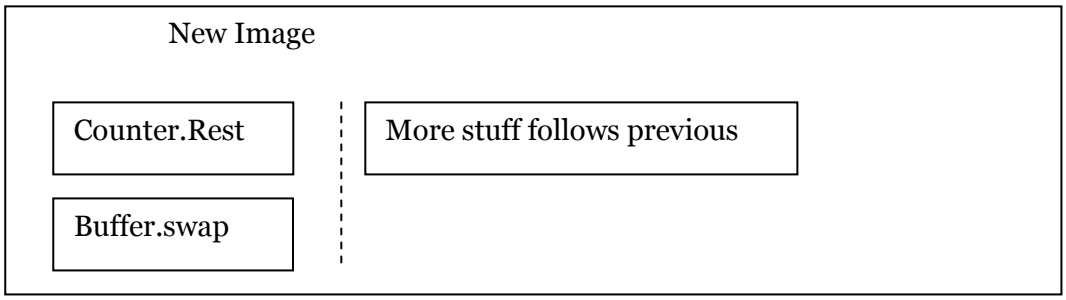
Expressions and control structures within a computational diagram can occur: concurrently (b) with each other, sequentially (a) or form a pipeline.



Pipeline sequential concurrent

The third view is based on a state chart diagram and allows architectural blocks and computational segments to be controlled using a when control. Making them only run when the state they are in is active.





1) First the problem is to design a system to count the number of rice in an image.

You only need to specify the Architectural blocks and sub blocks you require. You can assume that any block you need can be found or designed with out you needing to specify the computational design.

2) You then need to construct a Resource and Scheduling view to specify when the different blocks can run.

3) The second problem is to implement an algorithm for colour blob tracking.

Use VERTIPH in any way you wish to implement the supplied pseudo-code algorithm or another algorithm with a similar function

Steps:

A bounding box (bb) data structure is require:

```
bb{
     minx
     maxx
     miny
     maxy
}
Colour space transform:
```

$$\begin{bmatrix} Y' \\ U' \\ V' \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 0 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

```
two table look ups (to find required regions of intrest)
only one table look up can occur per table. You don't
need to specify the table contents
Utest = Table(Y, U);
Vtest = Table (Y, V);

Work out the colour class that the pixel corresponds to.
Colour_class = Utest&Vtest;

Filter parallel implementation:
filterX = colour_class & filter_old; //horizontal compare
filter_old = colour_class;    //save to compare next time
rowBuff(x)=filterX;           //buffer result
bb.i = filterX & rowbuff(x); //used buffed and current to
                              get answer

Update the bounding box
xmin = x < bb.minx  // Check if lower x limit needs
                       adjusting
xmax = x > bb.maxx; // Check if upper x limit needs
                       adjusting
init = bb.maxy; == 0;    // Have any pixels been added to
                            this region yet?
bb.maxy= y;              // Current pixel is always
                              bottom so far
if (init) {                   // First pixel found
bb.minx[bb.i] = x;
     bb.miny[bb.i] = y;       // Initialise structure to
                                   pixel position
     bb.maxx[bb.i] = x;
}
else if (xmin) par {              // New minimum x
     bb.minx[bb.i] = x;
```

```
}
else if (xmax) par {                    // New maximum x
     bb.maxx[bb.i] = x;
}
```

Having assumed that new data arrives every clock cycle you find out that new data arrives ever second clock cycle, make the pipeline you have made a two phase pipeline. You can re arrange parts of the algorithm if required.

10.2 Question

10.2.1 Experience questions

Age:

Industry Experience:

Experience in digital design (summarise and list):

Experience in programming FPGAs (summarise and list):

When explaining a system would you be more likely to draw a diagram or write a description?

Have you used a visual programming language like:

LabView:

ProgGraph:

Others please list:

How would you asses your experience and confidence in digital design

How many if any image processing algorithms have you implemented?

None A few lots it is what I do as a job

10.2.2 Questions on VERTIPH

Does VERTIPH conform to your conception of what a Visual Programming Language is? (Explain your answer)

How do you find the balance between pictures and text?

Good too graphical too textural bad

If there were an area where you could increase the amount of textual notation were would it be?

If there were an area where you could increase the amount of visual notation were would it be?

Does the structure of the design seam like a suitable structure for the problem you solved?

Did the structure imposed by VERTIPH help or hinder you?

Does the structure required by VERTIPH seam natural?

Did you have to redesign parts of your solution to fit VERTHIPH?

(if so how and why)

How easy was it to make changes?

Did you find the pipelining notation

helpful useful ok annoying confusing

Did the pipeline notation seam natural?

Yes somewhat not really no

Did you find the control structures available to you expressive enough?

Do the graphical symbols seam intuitive?

What did you find as the most difficult part of the task and how did VERTIPH help or hinder you?

Is the graphical syntax easy or hard to use?

Do you think the Resource and Scheduling view helps in your design or do you find it confusing?

Do you think that this mode of construction of the digammas with paper effect what you think of the system?

What would you use the different views for?

If you could break your design any why you liked how would you (into different view etc)?

Does the division into 3 views inhibit or is it natural for you design process?

How difficult did you find the tool to use?

How much easier would you find the tool to use:

After this session?

With practise?

What are the good parts VERTIPH for you?

What improvements could be made to VERTIPH?

Given the choice would you use a tool like VERTIPH in preference to:

VHDL

Verilog

Handle-C

SystemC

Matlab

Others (list):

Open ended discussion

11 Appendix II: Evaluation of prototype

This appendix contains the introduction to VERTIPH, tasks and the questionnaire that was used in the user evaluation of the VERTIPH prototype.

11.1 VERTIPH user evaluation questionnaire

The purpose of this evaluation is for you to test and review VERTIPH giving feedback.

The evaluation comprises two sections. First, to familiarise you with the system, we've provided a couple of worked examples which are presented as a screen cast and screen shots. The second section comprises two implementation tasks. Each section of the evaluation is followed by a questionnaire and discussion

The evaluation has been developed to be as quick and simple as possible. Therefore, the example tasks focus on the features required to perform the later tasks, and do not require a full problem to be solved. The first evaluation task involves constructing an architectural diagram for a problem and specifying some of its computations in the computational view. The second evaluation task involves implementing an image processing task given a detailed description of the task and some pseudo-code of one possible design.

Please read the introduction and overview of VERTIPH sheet before continuing.

11.2 A short introduction to VERTIPH

This is a short introduction to VERTIPH focusing on the things that you need to know about VERTIPH to be able to use it during the evaluation.

This version of VERTIPH has two editing environments, one for designing the major architectural components of a device and the information flows between them (the Architecture Editor), and one for designing the detailed computations associated with an architectural module (the Computation Editor). A third environment, for scheduling computations (the Resource and Scheduling Editor) will be added later

A VERTIPH project is arranged as a tree, with computations confined to the the leaf nodes; all ancestor nodes are thus architectures. (Figure 11.1 illustrates this arrangement.) The internal design of an architectural node comprises a network; the nodes of the network represent the architecture's children and the links represent information flows between those children. Links are connected to *terminals* which have an internal name and an external name, so that generic functions can be created and used in more than one context.

Each architecture block can then contain either a network of connected architecture blocks or a collection of encapsulated computational blocks. Within a block (whether architectural or computational), the terminals through which inputs and outputs arrive and depart so that it can communicate with higher level components are shown at the boundaries, along with their internal names. Note that these terminals are not shown in Figure 11.1.

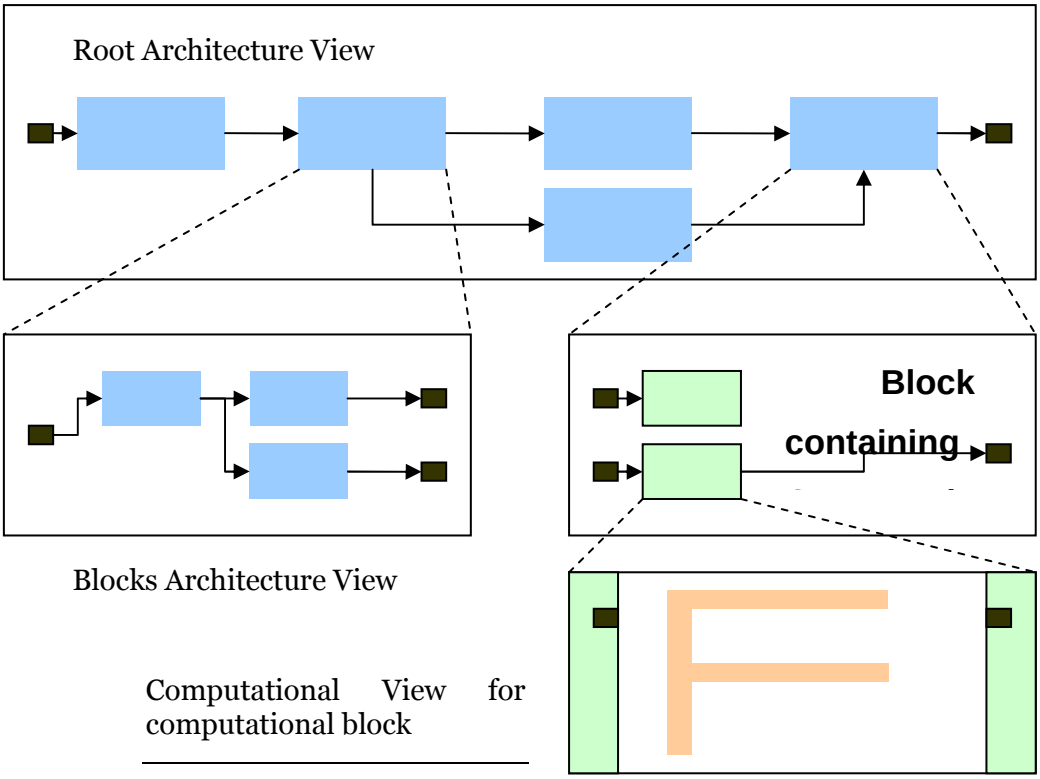


Figure 11.1: Architectural and Computational views

The primary focus of the architectural IDE is therefore on the broad decomposition of the high-level algorithm, and the flow of data between the high level operations. It allows specific tasks within an image processing algorithm to be allocated to blocks which can be developed independently and validated using test image data.

Detailed operations are specified in the computational IDE. Unlike HDLs like VHDL, which uses different assignment symbols to distinguish between parallel, sequential and pipelined operations, or Handel-C, which uses sequential and parallel control structures, VERTIPH represents the degree of parallelism in a design visually.

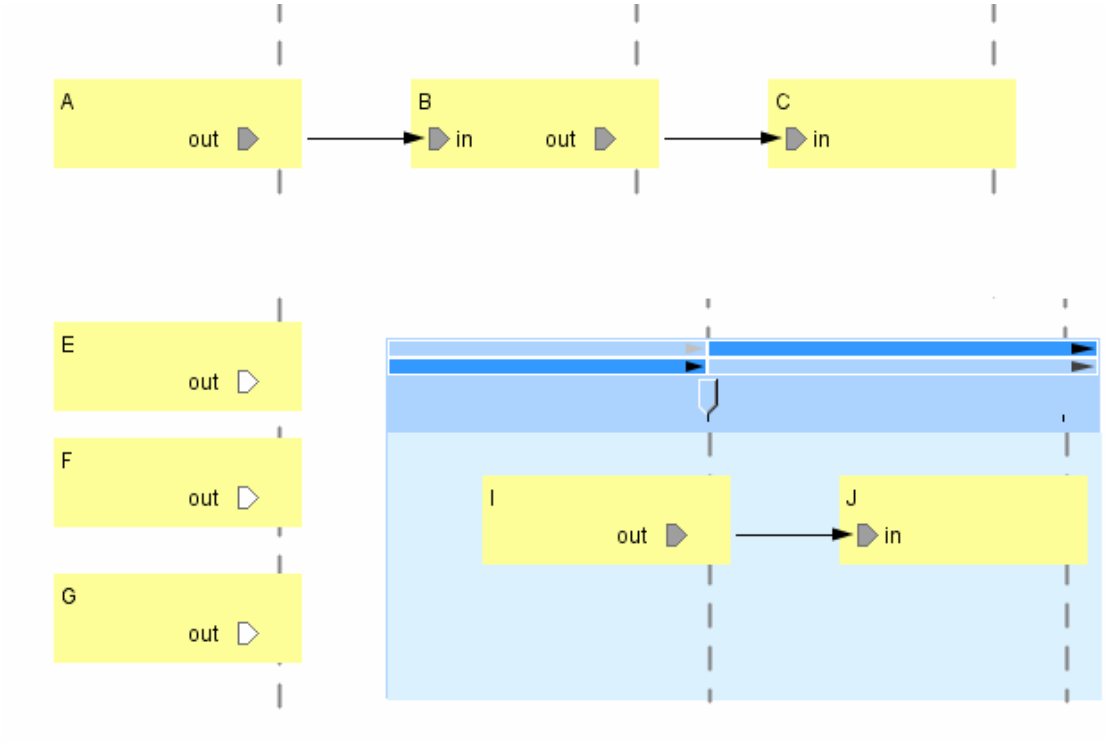


Figure 11.2: Sequential, parallel, and pipelined arrangements of modules in a design

Figure 11.2 illustrates this. Operations A, B and C run in sequence; E,F,G run in parallel, and I and J are a pipeline. Pipelined operations are enclosed in a control structure but are otherwise treated as a special case of sequential operations.

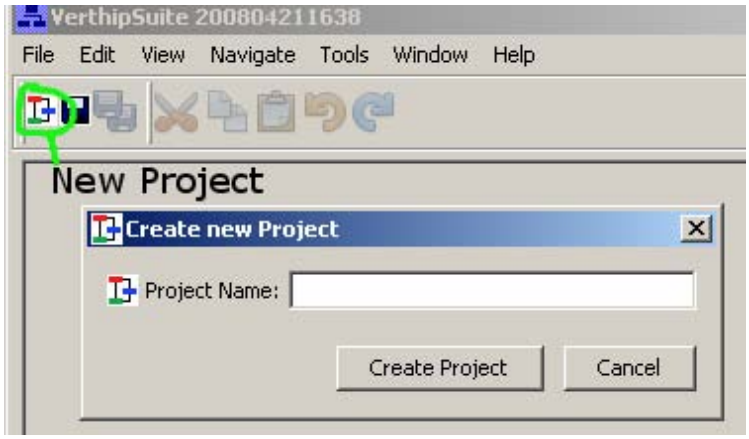
The following is a quick getting-started guide.

11.3 Getting Started

Open VERTIPH.

Figure 11.3 New Project

Click on the *New Project* icon.



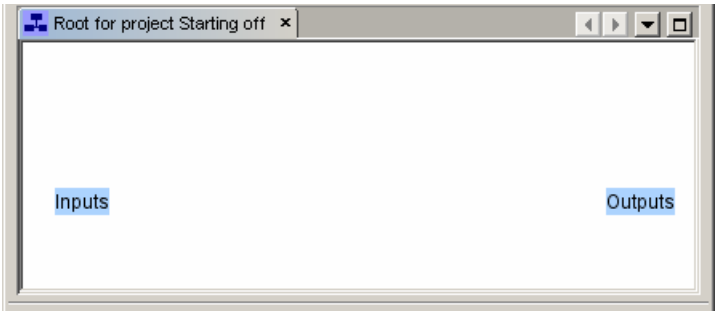


Figure 11.4 Root

This will create a new root architecture view.

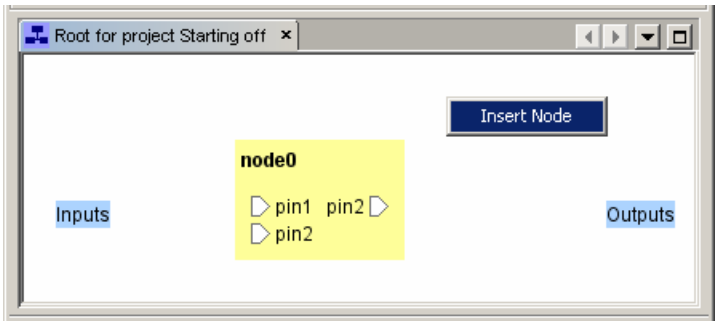


Figure 11.5 New Node

Add a new node by right-clicking on the background and selecting *Insert Node*

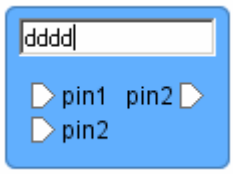


Figure 11.6 Name

You can change a node's name or a pin's name by double-clicking on the text.

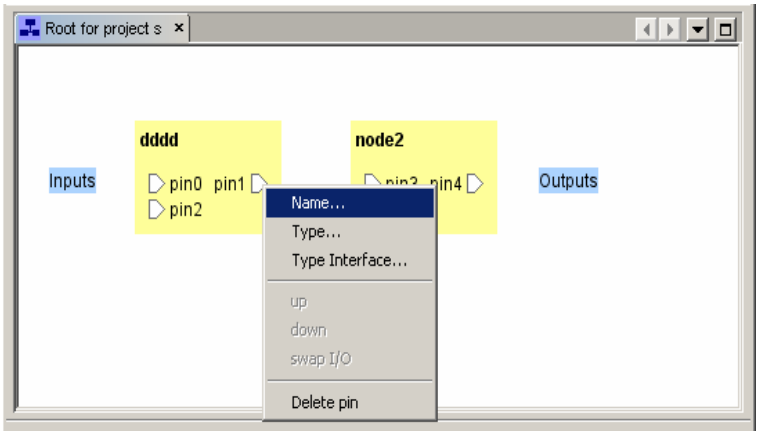


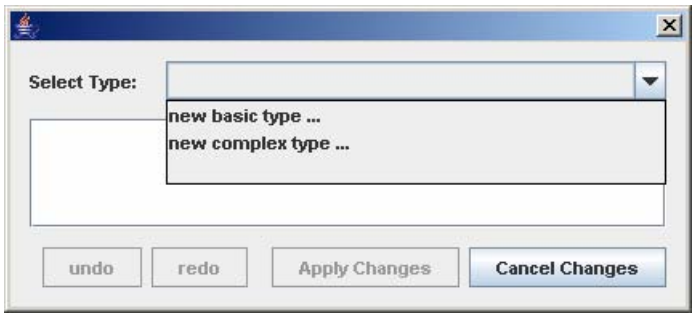
Figure 11.7 Pin

Right clicking a pin displays a menu for creating the type, editing the type interface (the interface between the external

pins and types and the internal pins and types) or deleting the pin.

Figure 11.8 Type

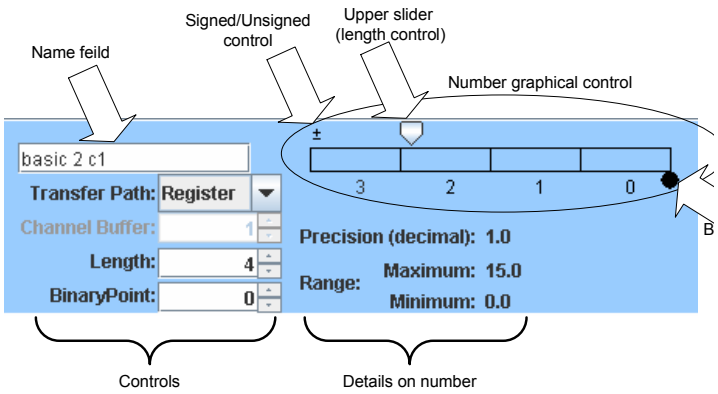
Select *Type...* to bring up a type editor.



You can use this menu to create a new basic or complex type, or to select any predefined types from the drop down menu.

Figure 11.9 Type Basic

Select *Type...* to bring up a type editor.



. Figure 11.10 Type complex

A complex type with a root node having three children one of which is a complex type with two basic types can children

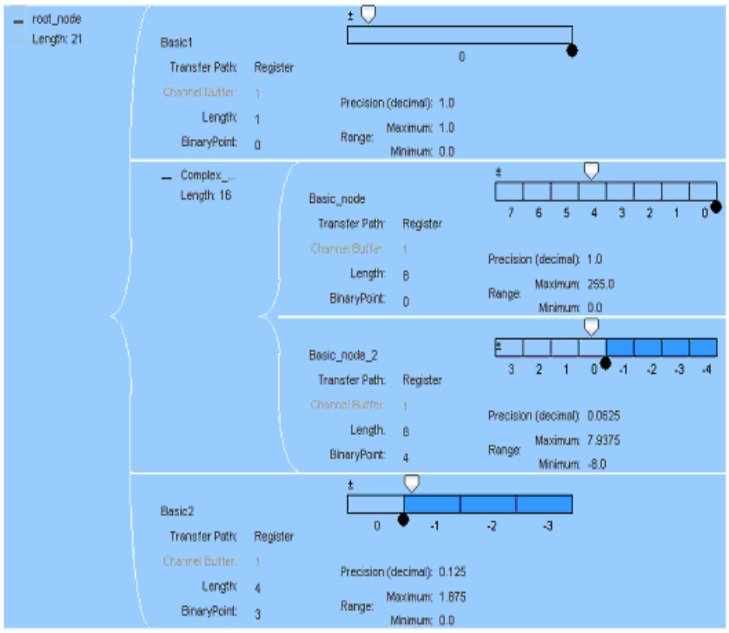


Figure 11.11 Mouse over

Once a pin's type has been defined, its colour becomes grey. Thereafter, moving the cursor over the port will display the type.

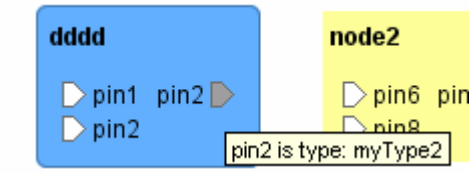
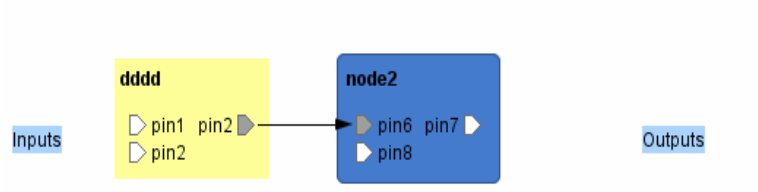
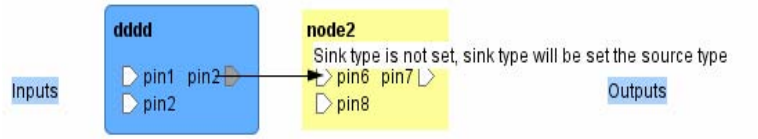


Figure 11.12 Connection

To connect ports, drag a wire from an output to an input.

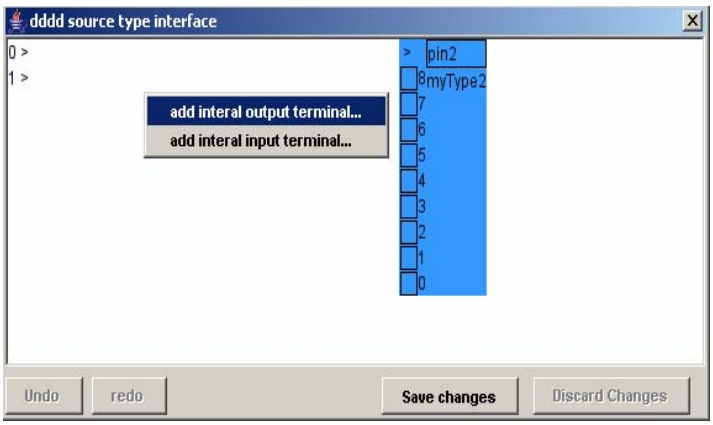


At least one of the ports needs to have its type defined. The



other will take this type. If there is a type mismatch between the ports the connection is not made.

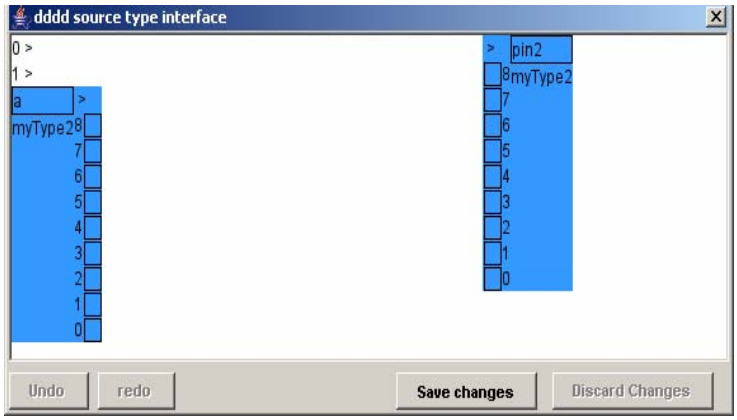
Figure 11.13
Junction Box



Type interface:

Looking at the source side of node *dddd* we see pin_2 with its type and the bits of the type.

Figure 11.14
Junction Box 2



When you add a new type, the IDE will request a name and then pop up the type dialogue.

Figure 11.15 Connections

You can drag wires between the bits of two pins to indicate how they are connected, and the ‘0’ and ‘1’ can be used as constants for padding bits

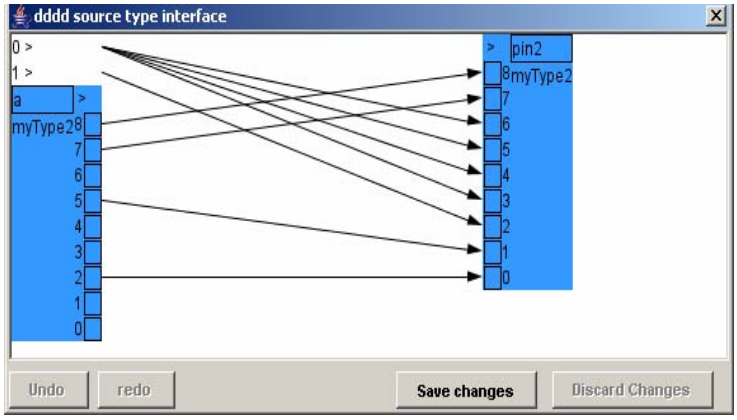


Figure 11.16 Connections

...or if types are the same length then you can connect them without dealing with the individual bits (the binary bit position will be ignored when doing this)

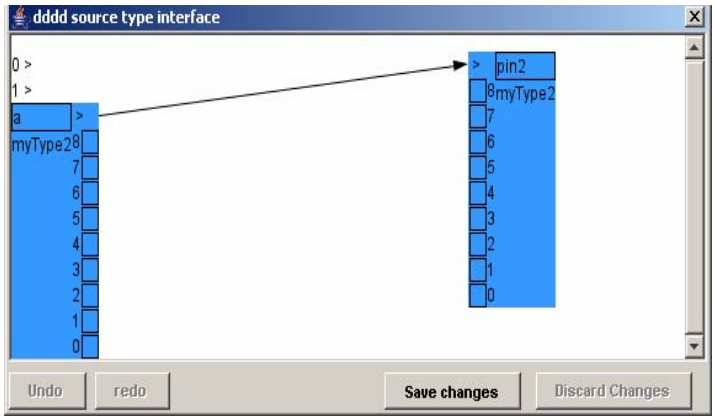
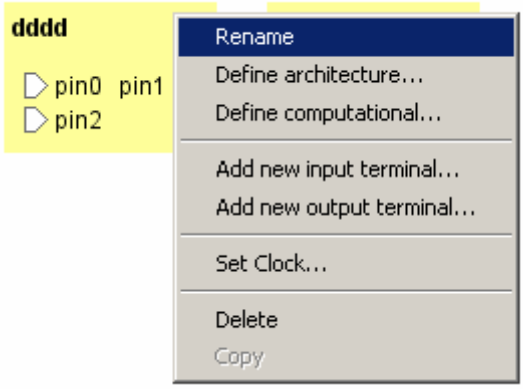


Figure 11.17 Menu

Right-click on a node to display a menu of node-related operations;

to rename it,

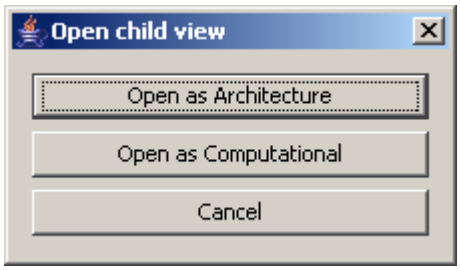
to define its contents



or its interface, to set its clock domain or to delete it.

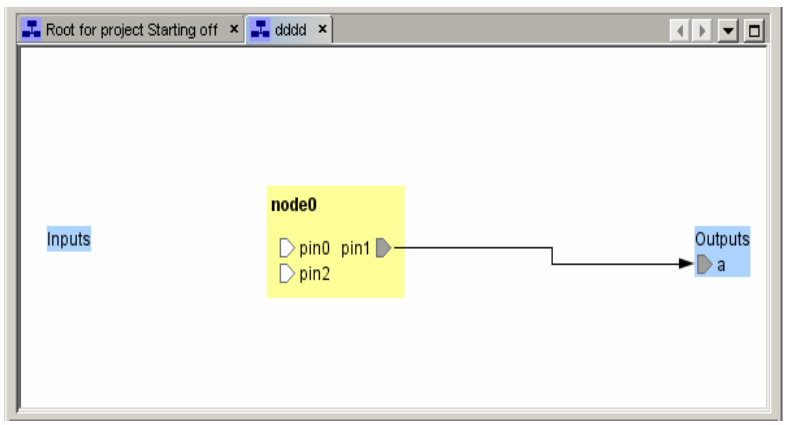
The *Define Architecture* option will set the child of the current node to be an architectural module, *Define Computational* will set the child to be a computational module

Figure 11.18 Dialog



You can also double click on the node to open it, and (if its type is not already defined) the dialogue at right will appear, allowing you to set its type.

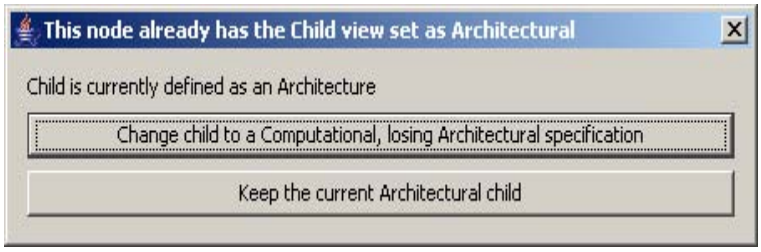
Figure 11.19 Open as



If you open the child as an architecture, the view at right will appear. (Note we have added a new node to this and connected it to

an output.)

Figure 11.20 Change



You can close this and redefine the node *dddd* as a computational view. This will give the message at right.

Figure 11.21 Comp View

The view at right is created. We have added a sample computational node to this which is similar to a architectural node, but also includes the duration in clock cycles of the node. Like the architectural nodes these computational nodes have internal and external pins. Defining an in and out internal pin gives the computational view shown below.

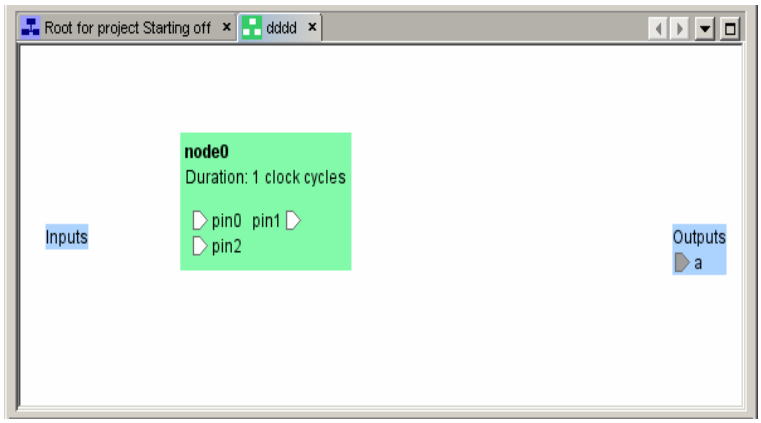
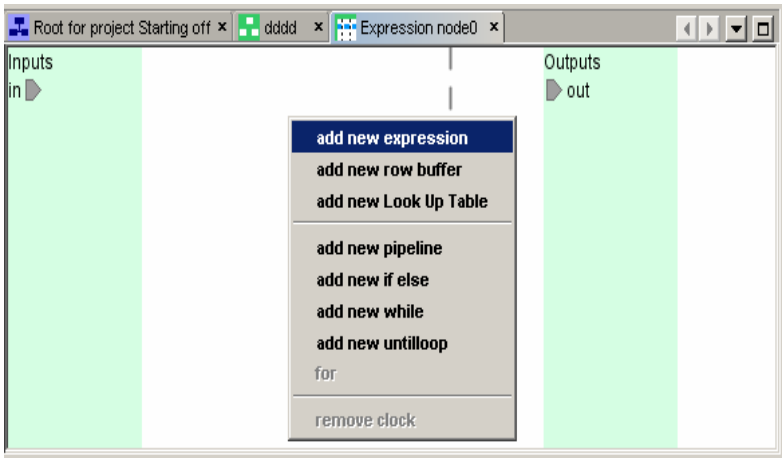
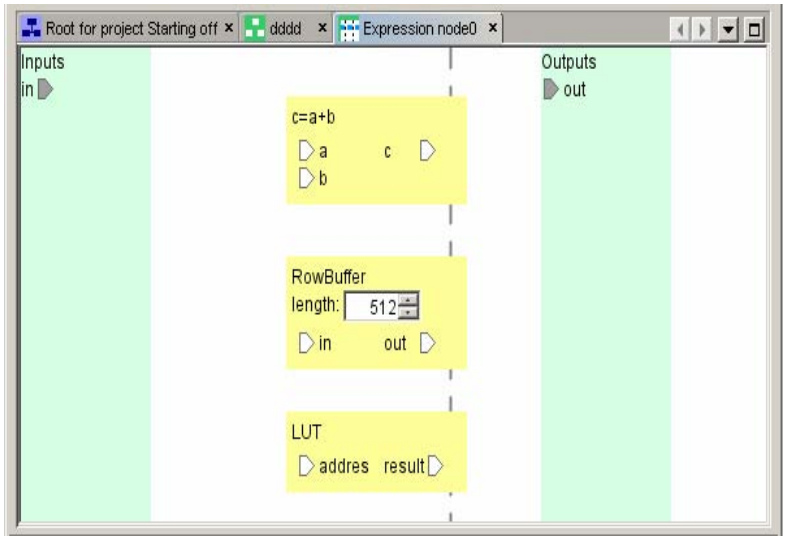


Figure 11.22 Add Menu



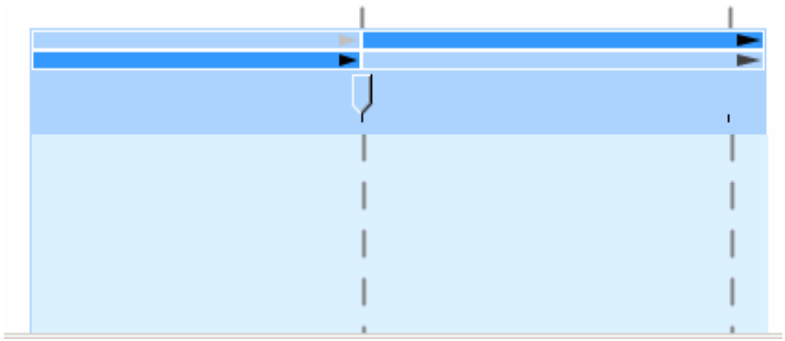
You can display a menu of the possible operations and controls by right-clicking on the clock.

Figure 11.23 Operations



Here the designer has used the menu in Figure 11.22 to add an arithmetic expression, a row buffer with 512 elements and a look up table.

Figure 11.24 Pipeline



When you use the menu shown in Figure 11.22 to create a pipeline, it is initialised with two clock cycles as shown at right. In the form shown here, new data

arrive every clock cycle.

Figure 11.25 Pipeline 2

To change the data arrival time to every second clock cycle, move the slider to the second clock cycle, as shown below

Figure 11.26 Pipeline two phase

Figure 11.27 Node

Changing the duration of this computational node to four clock cycles will automatically update the duration of the parent node.

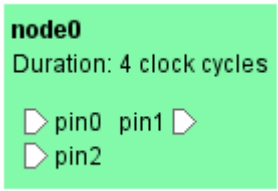


Figure 11.28 Controls

The control structures, *if else*, *while* and *until* can be selected from the menu shown in Figure 11.22 are shown at right.

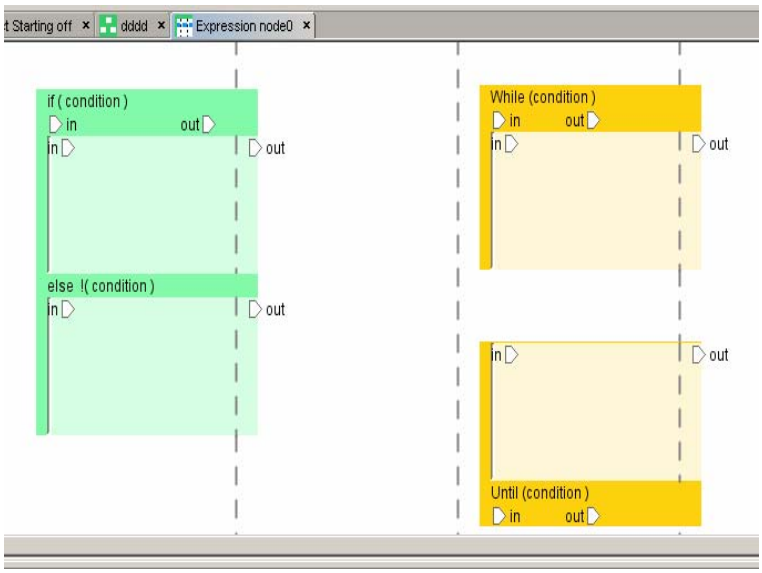
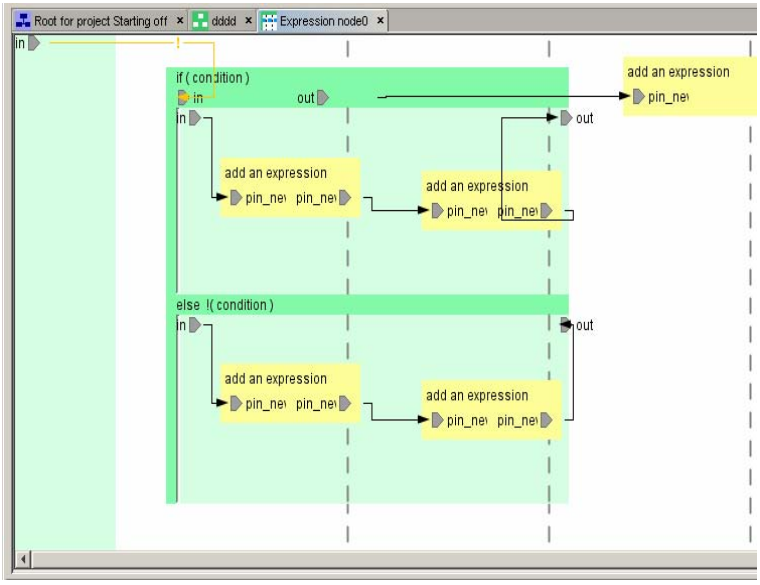


Figure 11.29 Controls 2

A connected *if-else* control



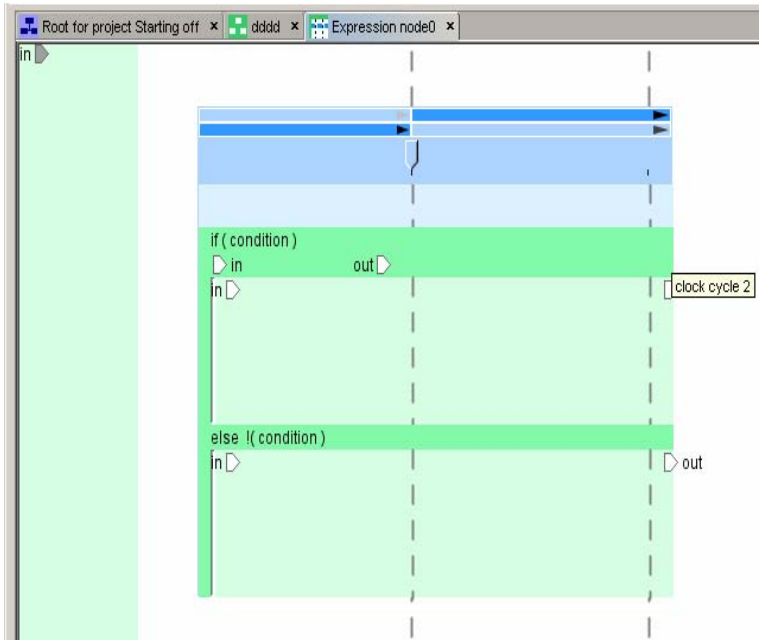


Figure 11.30
Controls 3

An *if-else* control in a pipeline

11.4 Activities

11.4.1 Activity 1

Purpose

To develop an understanding of how to use the architectural view of VERTIPH for an image processing task, learning about block diagrams, the data types, terminals and junction boxes for the design.

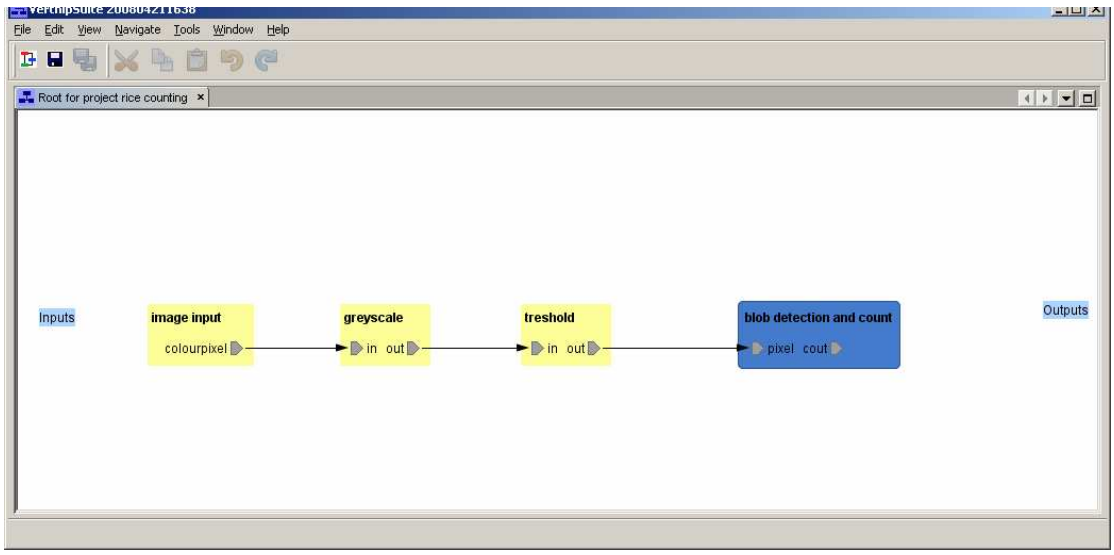
Task

To segment the objects in an image (grains of rice on a dark background) and count them.

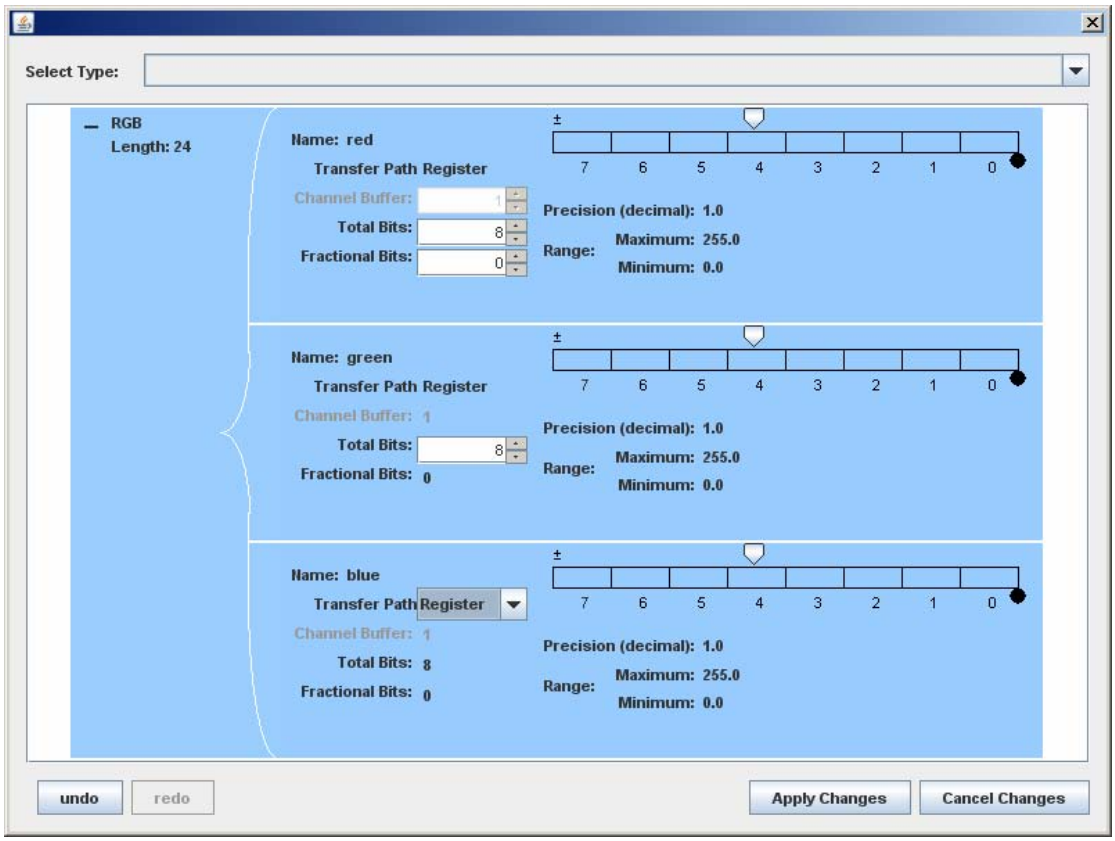
Method

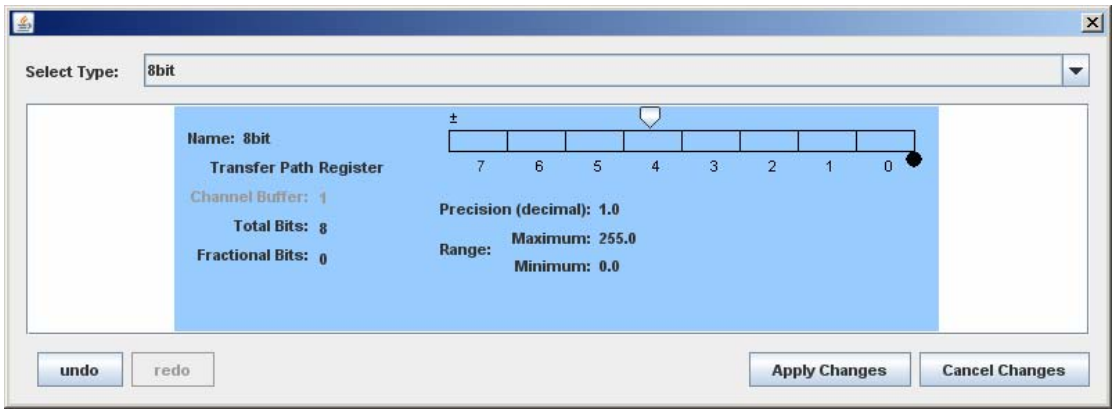
An existing solution will be presented in a screen cast video and screen captures can also be provided if required. You will use VERTIPH to replicate the design, first by creating the architectural blocks with the correct input and outputs, then setting the input and output types. You will also have to define the internal I/O connections via the junction box view. For this task, the actual implementation (in the computational view) is not required.

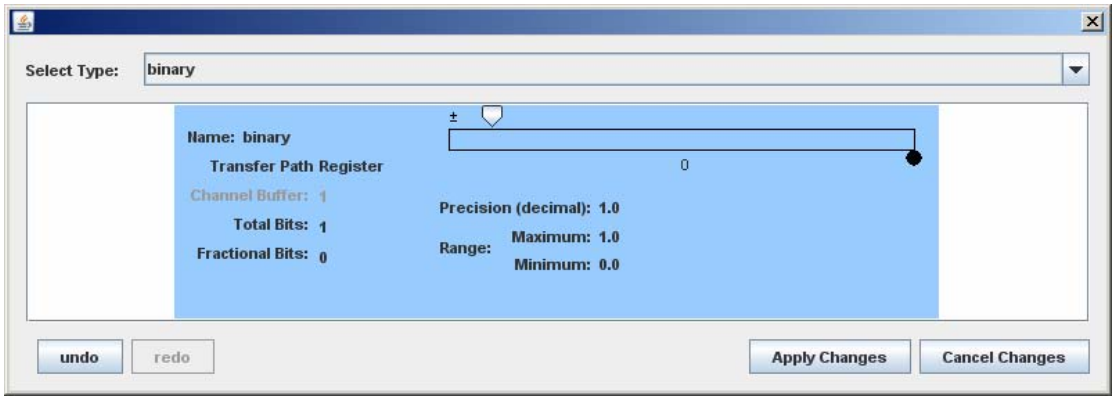
Final architecture view:

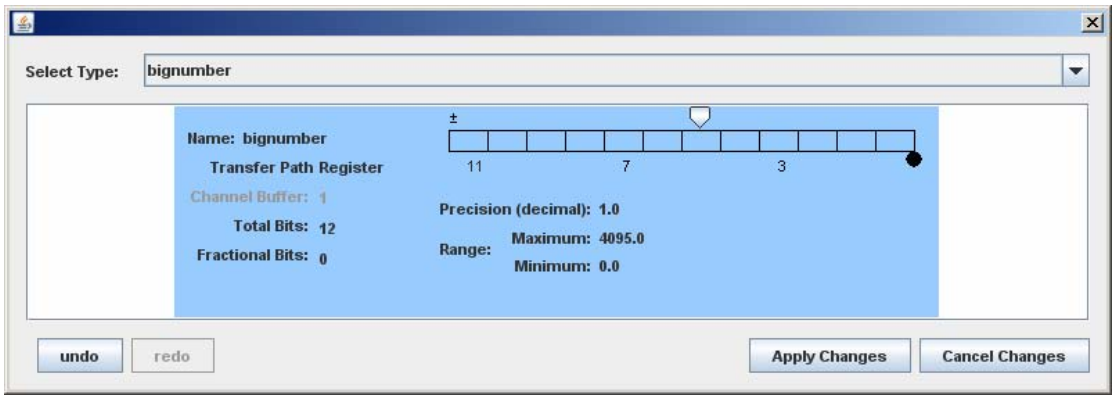


Types:


Select Type:
RGB
Length: 24
Name: red
Transfer Path Register
Channel Buffer: 1
Total Bits: 8
Fractional Bits: 0
Precision (decimal): 1.0
Range:
Maximum: 255.0
Minimum: 0.0
7 6 5 4 3 2 1 0
Name: green
Transfer Path Register
Channel Buffer: 1
Total Bits: 8
Fractional Bits: 0
Precision (decimal): 1.0
Range:
Maximum: 255.0
Minimum: 0.0
Name: blue
Transfer Path Register
Channel Buffer: 1
Total Bits: 8
Fractional Bits: 0
Precision (decimal): 1.0
Range:
Maximum: 255.0
Minimum: 0.0
undo
redo
Apply Changes
Cancel Changes

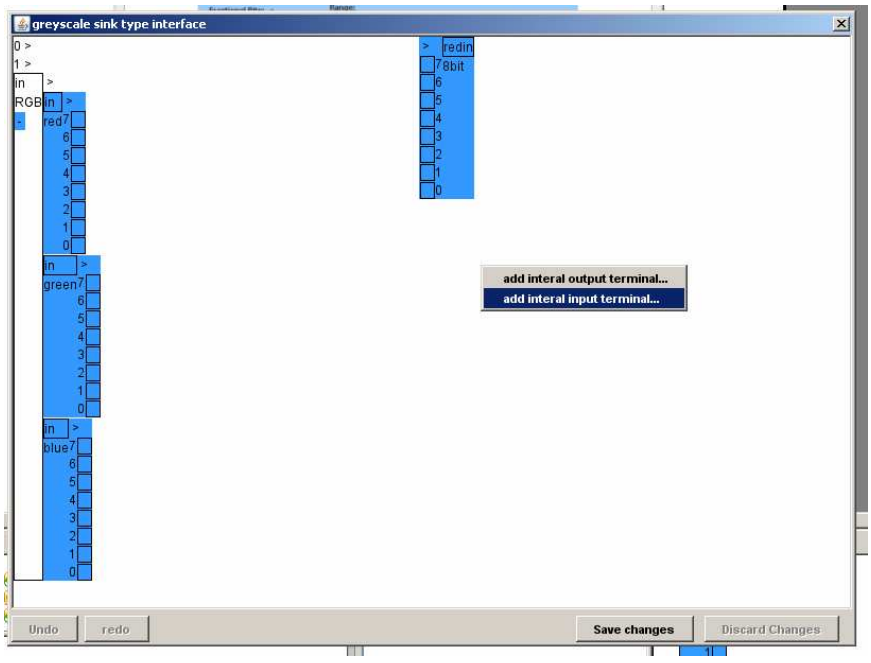

Select Type:
8bit
Name: 8bit
Transfer Path Register
Channel Buffer: 1
Total Bits: 8
Fractional Bits: 0
Precision (decimal): 1.0
Range:
Maximum: 255.0
Minimum: 0.0
7 6 5 4 3 2 1 0
undo
redo
Apply Changes
Cancel Changes



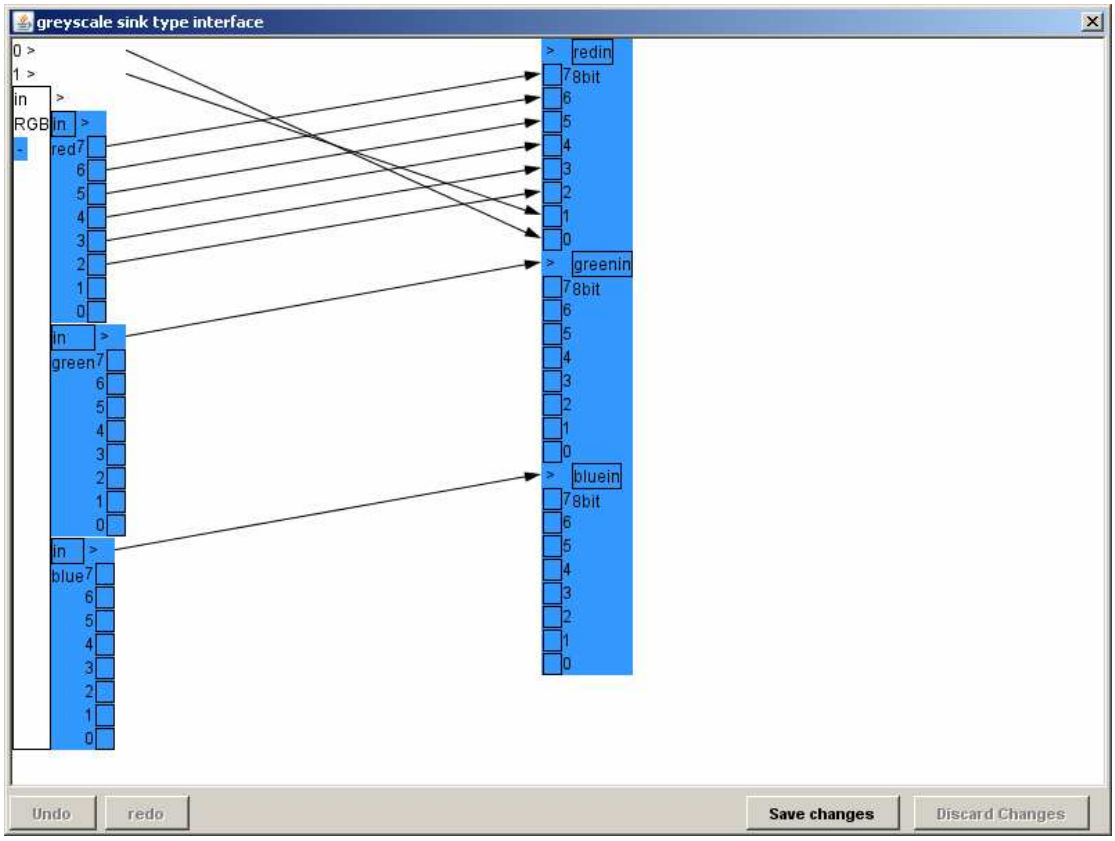


Greyscale input type interface:

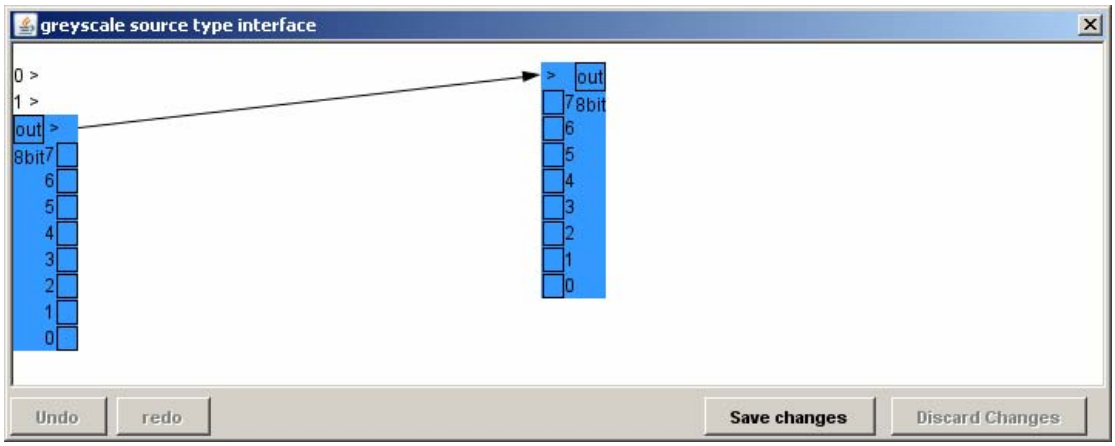
Create new internal output.



greyscale input type interface final :



final output type interface:



11.4.2 Activity 2

Purpose

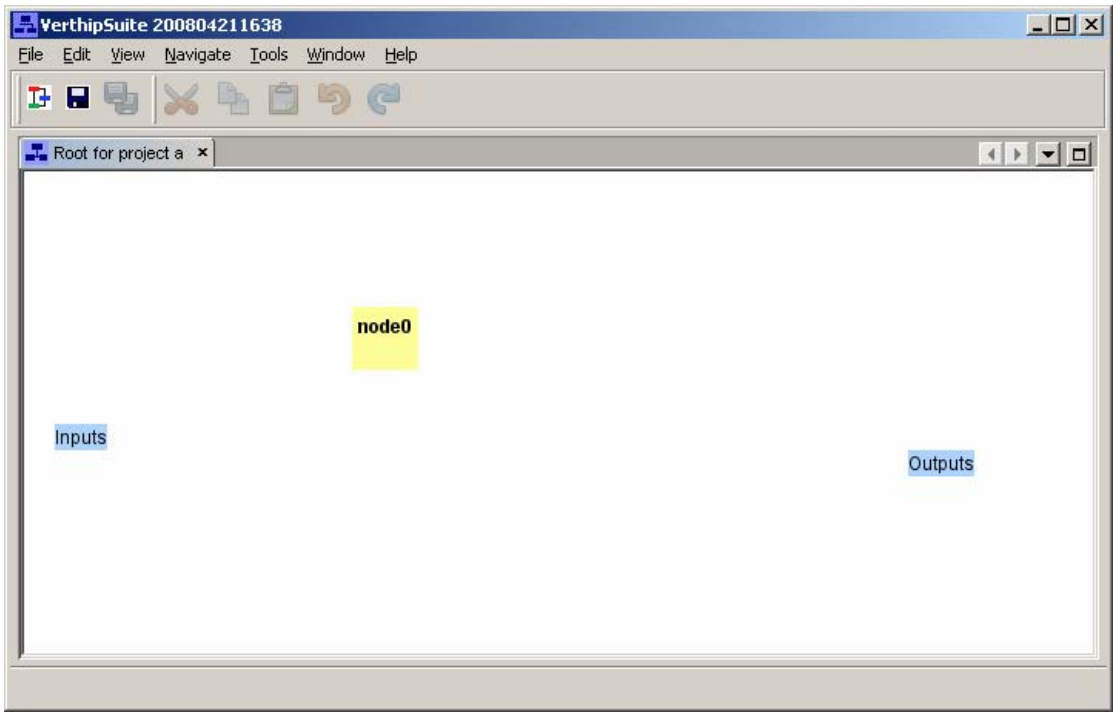
To familiarise you with the computational view components, pipelines and control structures.

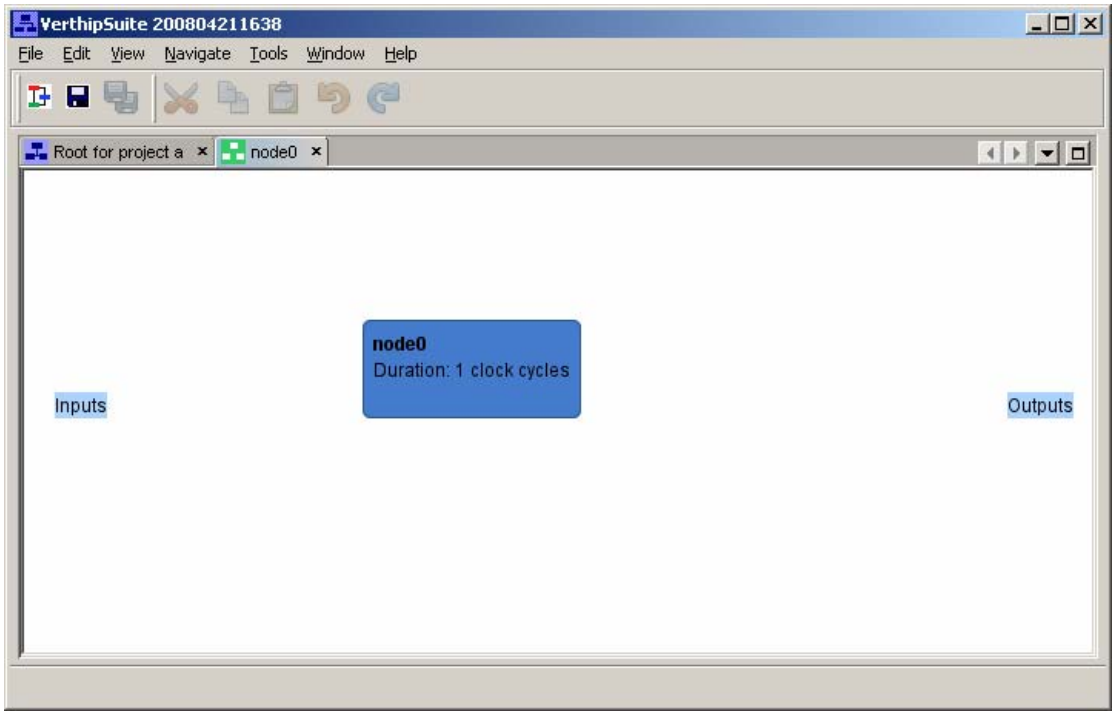
Task

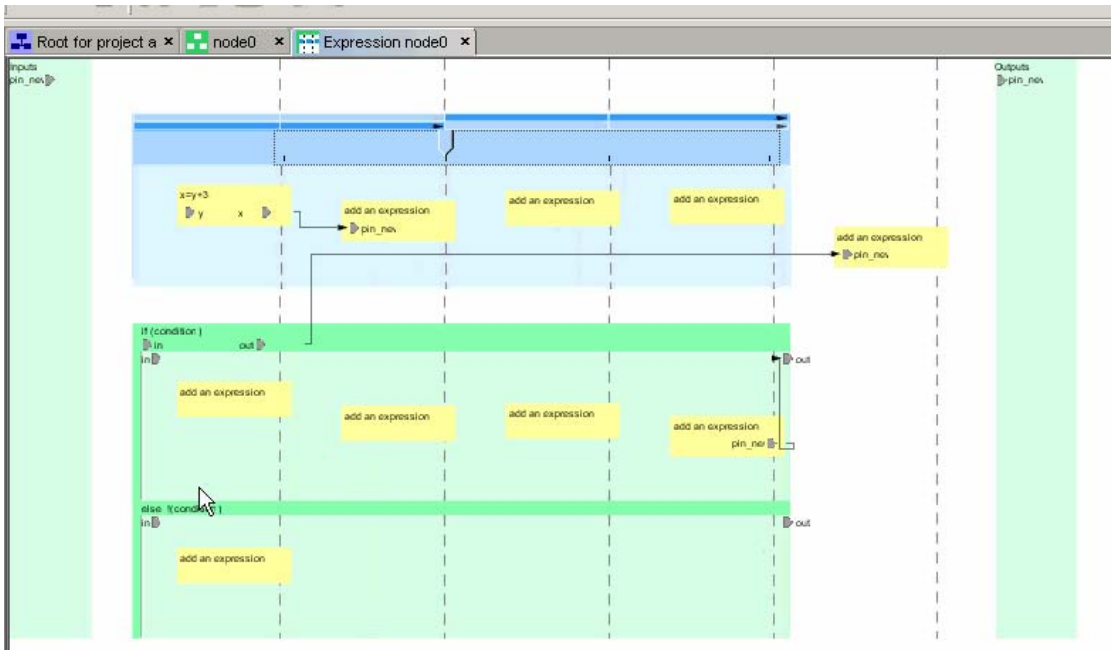
To construct a control structure involving a pipeline and a *if else* control

Method

A computational view will be presented in a screen cast video and screen captures can also be provided if required. You will use VERTIPH to replicate the design which involves constructing a pipeline with an *if else* control structure. The operations to be implemented are contrived and perform no useful function but represent all the operations required to complete the later tasks.




VerthipSuite 200804211638
File Edit View Navigate Tools Window Help
Root for project a
node0
node0
Duration: 1 clock cycles
Inputs
Outputs


Root for project a
node0
Expression node0
add an expression
if (condition)
else

11.4.3 Activity 3

Purpose

To elicit user feedback about using the VERTIPH editor

Task

To implement an image processing algorithm for the tracking of a coloured object.

Method

Use VERTIPH in any way you wish to implement the supplied pseudo-code algorithm or another algorithm with a similar function

Please use the code below and the attached poster description of the algorithm t implement the algorithm in VERTIPH.

Steps:

A bounding box (bb) data structure is require:

```
bb{
     minx
     maxx
     miny
     maxy
}
Colour space transform:
```

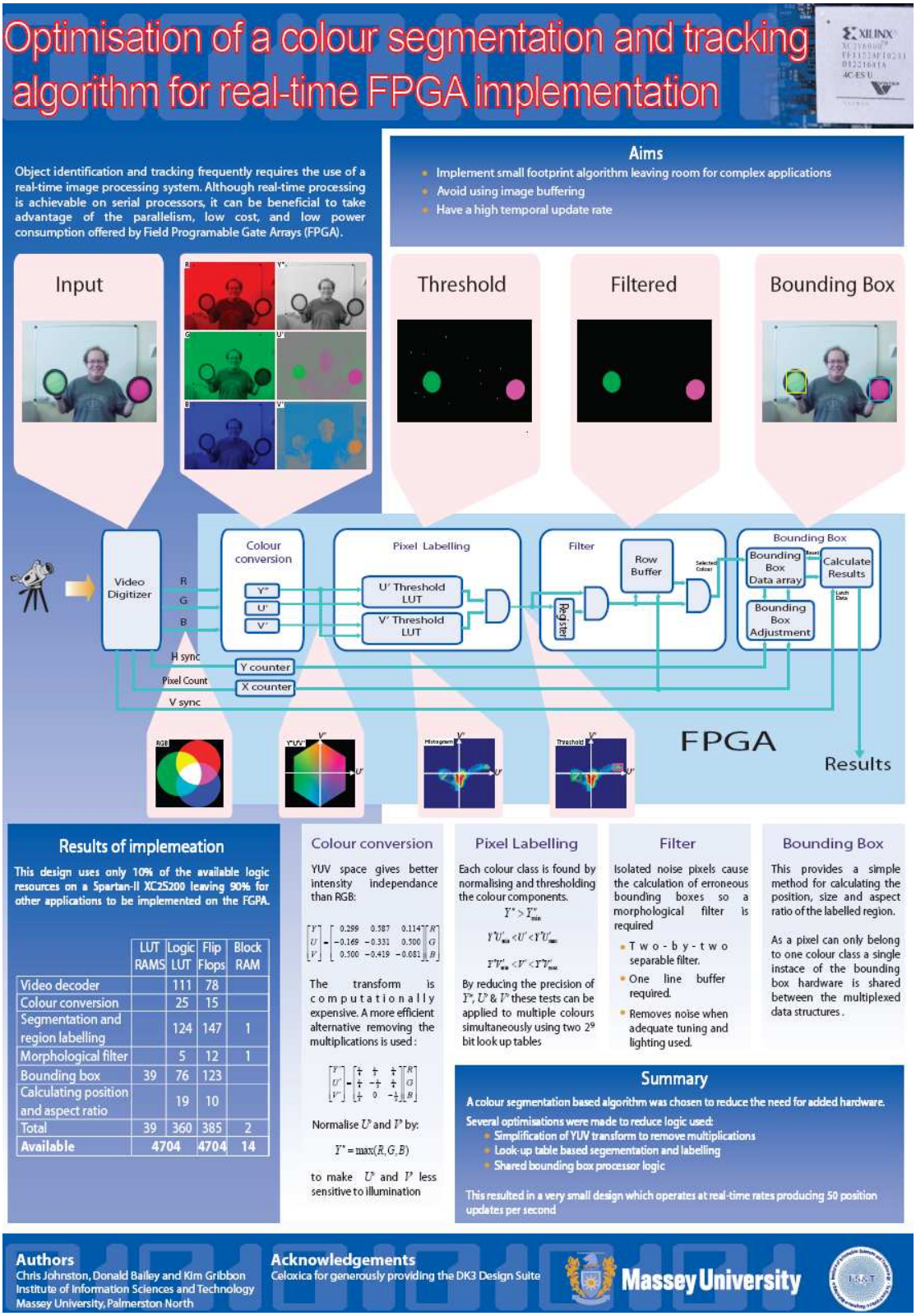
$$\begin{bmatrix} Y' \\ U' \\ V' \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 0 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

```
two table look ups (to find required regions of intrest)
only one table look up can occur per table. You don't
need to specify the table contents
Utest = Table(Y, U);
Vtest = Table (Y, V);

Work out the colour class that the pixel corresponds to.
Colour_class = Utest&Vtest;

Filter parallel implementation:
filterX = colour_class & filter_old; //horizontal compare
filter_old = colour_class;    //save to compare next time
rowBuff(x)=filterX;           //buffer result
bb.i = filterX & rowbuff(x); //used buffed and current to
                               get answer
```

```
Update the bounding box
xmin = x < bb.minx  // Check if lower x limit needs
                       adjusting
xmax = x > bb.maxx; // Check if upper x limit needs
                       adjusting
init = bb.maxy; == 0;    // Have any pixels been added to
                            this region yet?
bb.maxy= y;              // Current pixel is always
                              bottom so far
if (init) {                   // First pixel found
bb.minx[bb.i] = x;
     bb.miny[bb.i] = y;       // Initialise structure to
                                   pixel position
     bb.maxx[bb.i] = x;
}
else if (xmin) par {              // New minimum x
     bb.minx[bb.i] = x;
}
else if (xmax) par {              // New maximum x
     bb.maxx[bb.i] = x;
}
```

having assumed that new data arrives every clock cycle you find out that new data arrives ever second clock cycle, make the pipeline you have made a two phase pipeline. You can re arrange parts of the algorithm if required.

Algorithm summary if needed.



	LUT RAMS	Logic LUT	Flip Flops	Block RAM
Video decoder		111	78	
Colour conversion		25	15	
Segmentation and region labelling		124	147	1
Morphological filter		5	12	1
Bounding box	39	76	123	
Calculating position and aspect ratio		19	10	
Total	39	360	385	2
Available	4704		4704	14

11.4.4 Activity 4

Purpose

To elicit feedback about using VERTIPH as a design environment

Task

To design a min/max filter for every 3 pixel x 3 pixel block in a 512 pixel x 512 pixel greyscale image. Each pixel is represented as a 16-bit value, and one pixel arrives per clock cycle. End of line and end of frame signals will be provided.

The design should aim to minimise both logic and processing time.

Method

Use VERTIPH to develop the architectural block diagram and the computational view for the main processing elements. A full solution is not required due to the time limitations but if you do not complete the task can you give reasons why (such as unable to solve problem, taking too much time, found VERTIPH could not express what I wanted, etc).

11.5 Screening Questions

1. Summarise your experience in FPGA "programming" or digital design

Determine experience in digital and FPGA design

2. Summarise your experience in algorithm development

Determine if used in algorithm development

3. Have you done any image processing on FPGA or ASIC devices?

None ☐

Tried it ☐

Done a little ☐

Done a lot ☐

Find if familiar with image processing on FPGA

4. Have you done any signal processing on FPGA or ASIC devices?

None ☐

Tried it ☐

Done a little ☐

Done a lot ☐

5. Have you used any visual programming languages?

Simulink	None	tried it	use it a little	use it a lot
Labview	None	tried it	use it a little	use it a lot
Prograph	None	tried it	use it a little	use it a lot

Others (list):

How familiar they are with the visual languages and visual programming

6. What do you believe characterises a visual programming language?

Find what they believe a visual programming language should be

7. How would you speed up an algorithm for FPGA implementation?

Would they use a pipeline, parallel hardware units or another method. This may affect how they perceive VERTIPH as it is mainly pipeline orientated.

8. What HDLs have you used for algorithmic digital design?

VHDL	None	tried it	use it a little	use it a lot
Verilog	None	tried it	use it a little	use it a lot
Handel-C	None	tried it	use it a little	use it a lot
ImpulseC	None	tried it	use it a little	use it a lot
CUPL	None	tried it	use it a little	use it a lot
SystemC	None	tried it	use it a little	use it a lot

Others (list):

Try to find out what HDL and design process they are use to

9. What are the good and bad features of the HDLs you have used?

Find a prior bias

10. What is you understanding of pipelining?

Find what they consider is meant by pipelining.

11. What considerations would you make when designing an FPGA algorithm

1) for speed?

2) to reduce area used?

Consider all aspect of the process, from algorithm selection through to implementation summarise your thought process including algorithm selection and design through to implementation.

Find how they usually to design a solution.

12. Do you prefer a top down or bottom up approach, a mixture of the two or does it depend on the application?

As VERTIPH encourages a top down approach someone who favours a bottom up approach are likely to become frustrated.

11.6 Question after introduction

1. Did you have any problems with the tasks themselves? If so, what were they?

2. Did you have any problems with expressing your solutions in VERTIPH? If so, what were they?

3. Do you need any parts of VERTIPH explained?

Do the participants need more information or have any difficulties

4. What features of VERTIPH do you like or dislike? (please explain)

5. What do you think of:

(a) the pipeline notation

(b) the control structures

(c) the type junction box

find what initial reaction are

6. How closely does VERTIPH's top-down design flow match your own mental approach to design?

Similar different not sure

7. When developing and implementing an algorithm what is your preferred design flow?

Find if the design flow will that is imposed on them will effect what they think of VERTIPH.

11.6.1 Questionnaire following evaluation tasks

11.6.2 Questions about the first task

1. Did you have any problems with or difficulties completing this task?

2. Did you make changes the algorithm you were given? (if so how and why)

See what was thought of the algorithm and how it was implemented

3. Did the VERTIPH notation allow you to implement the algorithm how you wanted? If not, how not?

Find out expressiveness of the notation

4. If you used it, what did you think of the pipeline notation? If you didn't, was it because there was something about it that you didn't like or understand?

Specifics on the pipeline notation

5. What did you find the most difficult part of the task, and how did VERTIPH help or hinder you?

Did VERTIPH make it easier or harder to do the design

11.6.3 Questions about the second task

1. The task you were given was open ended; describe your solution.

Find what they thought they did so can compare to solution in VERTIPH

2. What design changes would you make if you were designing for speed?

3. What design changes would you make if you were designing to conserve area?

These aim to get the participant to give information on their design by commenting on what changes they would make. They might of made a one clock cycle design (nested if else) which is slow but works or they might have implemented a pipeline design.

4. How did you account for the lack of image data and the edges of the image for the filter (edge effects)? If you did not, discuss how you would modify your design. (If you don’t know what is meant by edge effect please ask for an explanation)

Edge effects are cased by having no data around the boarder of the image. There are many possible solutions, it is useful to see what was done and why or what could be done to modify the normal operation.

5. What did you find the most difficult part of the task and how did VERTIPH help or hinder you?

Did VERTIPH make it easier or harder to do the design

6. What parts of the VERTIPH notation helped or hindered you?

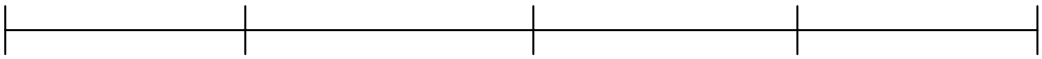
Did VERTIPH make it easier or harder to do the design

11.6.4 Questions related to both tasks

1. Overall, did VERTIPH help or hinder your design process?

Get a general comment

2. Was the top down design approach enforced by VERTIPH



helpful with design | neutral | a hindrance

Why?

If they don't like top down will not rate this high, links back to earlier questions. Want to eliminate bias towards VERTIPH due to use of top-down design process.

3. How did you find the balance between text and graphical notations in VERTIPH?

Far too graphical ☐

A little too graphical ☐

Good ☐

A little too textual ☐

Far too textual ☐

Is there too much text or not enough

4. If you could increase or decrease the amount of graphical notation what would you change?

What parts do they want more or less graphical notation used.

5. If you could increase or decrease the amount of textural notation what would you change?

What parts do they want more or less text used.

6. How did you find the pipeline notation?

Very useful ☐

Somewhat useful ☐

Neutral ☐

Somewhat annoying ☐

Very annoying ☐

Why?

What do they (dis)like about the pipeline notation.

7. Did the pipeline notation seam natural to you?

Yes ☐

Somewhat ☐

Neutral ☐

Not really ☐

Not at all ☐

Why?

Does the notation make sense or are they having some alien idea imposed on them.

8. The (commercial) automatic layout algorithm does not always lay wires out in an appropriate way; do you think that this affected your evaluation of VERTIPH?

Yes, because it interfered significantly with my ability to do the job ☐

Yes, but it was only a minor distraction ☐

No, it was acceptable in a prototype ☐

I didn't notice it ☐

How badly did the errors in wire layout effect their design

9. What did you think of the loop control structures?

useful ☐

somewhat useful ☐

neutral ☐

somewhat hard to use ☐

hard to use ☐

10. How would you like to see the loop control structures improved?

What was good or bad about the loop control structures

11. What did you think of the *if-else* control structure?

useful ☐

somewhat useful ☐

neutral ☐

somewhat hard to use ☐

hard to use ☐

12. How would you like to see the *if-else* control structure improved?

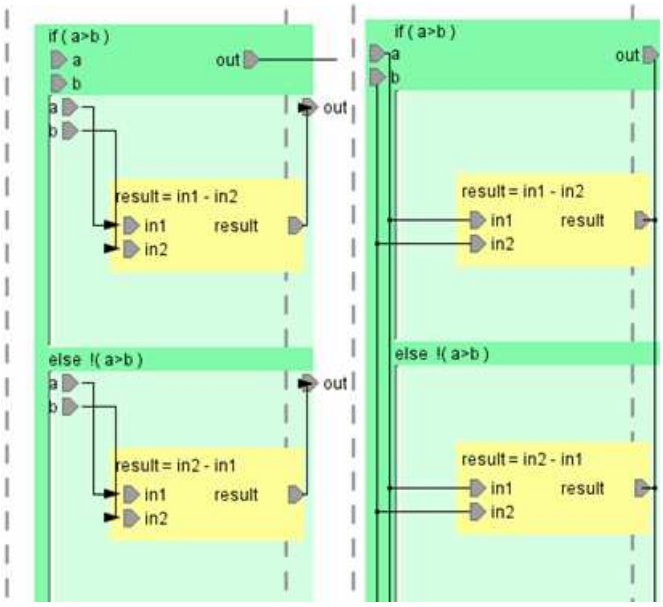
What was good or bad about the if-else control structure

13. You used the *if else* control labelled A below; there are several other *if else* controls being considered. Please rank these different views (shown on the next page) and comment on their advantages and disadvantages.

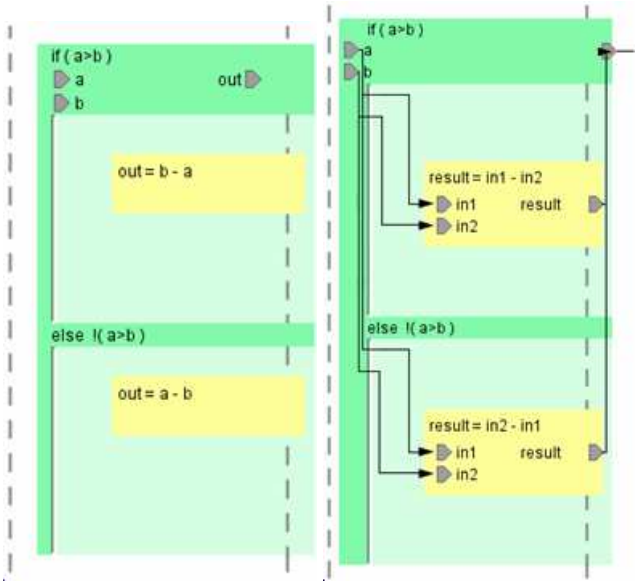
Rank: Best

Worst

Comments:



Current Wire-based



Name-based 2nd wire-based

Do the connections need improvement and which one of the possible solutions is best.

14. What did you think of the multi view design concept?

Comments on the use of multi view.

15. How easy or difficult did you find VERTIPH to use and why? (and which parts)

How easy was VERTIPH to use.

16. Did you find VERTIPH easier to use as you gained more experience?

17. Do you think it would be easier to use with practice?

Do they think it will get better using VERTIPH later

18. What do you believe are the good parts of VERTIPH?

General comment on what parts of VERTIPH are good.

19. What improvements could be made?

20. Do you think with refinement VERTIPH could be a useful HDL for your use?

21. How would you compare this tool to other you have used?

General comparison of between VERTIPH and other tools

22. What did you think about the use of the spatial dimension to separate parallel and sequential operations?

Very useful ☐

Somewhat useful ☐

Neutral ☐

Somewhat annoying ☐

Very annoying ☐

Very easy to use and understand ☐

Somewhat easy to use and understand ☐

Neutral ☐

Somewhat hard to use and understand ☐

Very hard to use and understand ☐

Was using layout for parallel and sequential operations useful and was it easy to use and understand.

23. Basing the pipeline structure on sequential design is somewhat different from other languages; what do you think of the pipeline approach used?

Very useful ☐

Somewhat useful ☐

Neutral ☐

Somewhat annoying ☐

Very annoying ☐

Very easy to use and understand ☐

Somewhat easy to use and understand ☐

Neutral ☐

Somewhat hard to use and understand ☐

Very hard to use and understand ☐

Was the pipeline structure useful and was it easy to use and understand.

24. How well do you think VERTIPH would scale to more complex designs?

As the examples they worked on were relatively simple to save on the time required to complete the evaluation, do they think VERTIPH would scale to more complex designs.

Open discussion.

12 References

ACCELCHIP (2005) AccelChip Products, www.accelchip.com, visited on February 2005.

ACCELLERA (2008) EDA Industry Working Groups - VHDL, http://www.vhdl.org/, visited on March 2008.

ALNUWEITI, H. M. & PRASANNA, V. K. (1992) "Parallel architectures and algorithms for image component labeling", *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 14, no. 10, pp. 1014-1034

ALSTON, I. & MADAHAR, B. (2002) "From C to netlists: hardware engineering for software engineers?" *Electronics & Communication Engineering Journal,* vol. 14, no. 4, pp. 165-173

ALTERA (2008a) AHDL, http://www.altera.com/support/examples/ahdl/ahdl.html, visited on December 2008.

ALTERA (2008b) Quartus II http://www.altera.com/literature/lit-qts.jsp, visited on February 2008.

ALVES DE BARROS, M. & AKIL, M. (1994) "Low level image processing operators on FPGA: implementation examples and performance evaluation", *Proceedings of the 12th IAPR International Conference on Pattern Recognition, Conference C: Signal Processing*, Jerusalem, Israel, October 9-13, vol. 3, pp. 262-267.

APPIAH, K., HUNTER, A., DICKINSON, P. & OWENS, J. (2008) "A Run-Length Based Connected Component Algorithm for FPGA Implementation", *International Conference on Filed-Programmable Technology (FPT'08)*, Taipei, Taiwan, 7-10 December, pp. 177-184.

ATKINS, M. S., ZUK, T., JOHNSTON, B. & ARDEN, T. A. A. T. (1994) "Role of visual languages in developing image analysis algorithms", *Proceedings of the IEEE Symposium on Visual Languages.*, St. Louis, MO, USA, pp. 262-269.

BABIONITAKIS, K., DOUMENIS, G., GEORGAKARAKOS, G., LENTARIS, G., NAKOS, K., REISIS, D., SIFNAIOS, I. & VLASSOPOULOS, N. (2008) "A real-time motion estimation FPGA architecture", *Journal of Real-Time Image Processing,* vol. 3, no. 1, pp. 3-20

BAILEY, D. G. (1991) "Raster based region growing", *Proceedings of the 6th New Zealand Image Processing Workshop*, Lower Hutt, 29-30 August, pp. 21-26.

BAILEY, D. G. (2002) "A new approach to lens distortion correction", *Image and Vision Computing New Zealand*, Auckland, New Zealand, 26th - 28th November pp. 59-64.

BAILEY, D. G. (2006) "Space efficient division on FPGAs", *Electronics New Zealand Conference (EnzCon'06)*, Christchurch, NZ, 13-14 November, pp. 206-211.

BAILEY, D. G. (2007) "Image Processing Using FPGAs", *Short Course offered in conjunction with the IEEE International Conference on Image Processing (ICIP 2007)*, San Antonio, Texas, USA, 16 September.

BAILEY, D. G. & GRIBBON, K. T. (2005) *Image Processing using FPGAs*. Tutorial offered in conjunction with the IEEE TENCON conference. Melbourne, Australia. 21-24 November, 2005.

BAILEY, D. G., GRIBBON, K. T. & JOHNSTON, C. T. (2006) "GATOS: A Windowing Operating System for FPGAs", *3rd IEEE International Workshop on Electronic Design, Test, and Applications (Delta 2006)*, Kuala Lumpur, Malaysia, 17-19 January, pp. 47-53.

BAILEY, D. G. & HODGSON, R. M. (1988) "VIPS - a digital image processing algorithm development environment", *Image and Vision Computing,* vol. 6, no. 3, pp. 176-184

BAILEY, D. G. & JOHNSTON, C. T. (2007) "Single Pass Connected Components Analysis", *Image and Vision Computing New Zealand*, Hamilton, New Zealand, 5-7 December, pp. 217-222.

BAILEY, D. G., JOHNSTON, C. T. & MA, N. (2008) "Connected components analysis of streamed images", *International Conference on Field Programmable Logic and Applications*, Heidelberg, Germany, 8-10 September, pp. 679-682

BAINBRIDGE-SMITH, A. S. L. (2005) "Real Number Repesentation for Image Processing on FPGAs", *Image and Vision Computing*, Dunedin, 28 - 29 November, pp. 471-475.

BANERJEE, P. (2003) "An overview of a compiler for mapping MATLAB programs onto FPGAs", *Proceedings of the ASP-DAC, Asia and South Pacific Design Automation Conference*, Kitakyushu City, Japan, January 21-January 24, pp. 477-482.

BANERJEE, P., BAGCHI, D., HALDAR, M., NAYAK, A., KIM, V. & URIBE, R. (2003) "Automatic conversion of floating point MATLAB programs into fixed point FPGA based hardware design", *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2003*, Napa, CA, USA 8-11 April, pp. 263-264.

BANERJEE, P., SHENOY, N., CHOUDHARY, A., HAUCK, S., BACHMANN, C., HALDAR, M., JOISHA, P., JONES, A., KANHARE, A., NAYAK, A., PERIYACHERI, S., WALKDEN, M. & ZARETSKY, D. (2000) "A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems", *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, California, April 17-19, pp. 39-48.

BAUMELA, L. & MARAVALL, D. (1995) "Real-time target tracking", *Aerospace and Electronic Systems Magazine, IEEE,* vol. 10, no. 7, pp. 4-7

BELLOWS, P. & HUTCHINGS, B. (1998) "JHDL-An HDL for Reconfigurable Systems", *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 15-17 April, pp. 175 -184.

BELLOWS, P. & HUTCHINGS, B. (2001) "Designing Run-Time Reconfigurable Systems With JHDL", *Journal of VLSI Signal Processing Systems for Signal Image and Video Technology,* vol. 28, no. 1-2, pp. 29-45

BELLOWS, P., HUTCHINGS, B., HAWKINS, J., HEMMERT, S., NELSON, B. & RYTTING, M. (1999) "A CAD Suite For High Performance FPGA Design", *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, California, USA, April 17-April 19, pp. 12-24.

BENKRID, K. (2000) *Design and Implementation of a High Level FPGA Based Coprocessor for Image and Video Processing*, PhD Thesis in Department of Computer Science, The Queen's University of Belfast, Belfast.

BENKRID, K., CROOKES, D. & BENKRID, A. (2002a) "Design and implementation of a novel algorithm for general purpose median filtering on FPGAs", *IEEE International Symposium on Circuits and Systems, ISCAS 2002.*, Phoenix, Arizona, 26-29 May, vol. 4, pp. 425-428

BENKRID, K., CROOKES, D. & BENKRID, A. (2002b) "Towards a general framework for FPGA based image processing using hardware skeletons", *Parallel Computing,* vol. 28, no. 7-8, pp. 1141-1154

BENKRID, K., CROOKES, D., BENKRID, A. & BELKACEMI, S. (2002c) "A Prolog-based hardware development environment", *International Conference on Field Programmable Logic and Applications*, Montpellier, France, 2-4 September, vol. LNCS 2438, pp. 370-380.

BENKRID, K., SUKHSAWAS, D., CROOKES, D. & BENKRID, A. (2003) "An FPGA-Based Image Connected Component Labeller," in *Field-Programmable Logic and Applications*, vol. 2778, *Lecture Notes in Computer Science*: Springer Berlin, 2003, pp. 1012-1015.

BHARTACHARYYA, S. S., LEUPERS, R. & MARWEDEL, P. (2000) "Software synthesis and code generation for signal processing systems", *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, [see also IEEE Transactions on Circuits and Systems II: Express Briefs],* vol. 47, no. 9, pp. 849-875

BHASKER, J. (1999) *A VHDL Primer*, Third ed. New Jersey, Prentice-Hall.

BIANCARDI, A., MOSCONI, M. & RUBINI, A. (1995) "Papilio: A Visual Environment for Multi-SIMD Programming", *Journal of Visual Languages & Computing,* vol. 6, no. 4, pp. 349-366

BIC, L. (1990) "A process-oriented model for efficient execution of dataflow programs." *Journal of Parallel and Distributed Computing,* vol. 8, no. 1, pp. 42-52

BINS, J., DRAPER, B., BOHM, W. & NAJJAR, W. (1999) "Precision vs. Error in JPEG Compression", *SPIE Parallel and Distributed Methods for Image Processing III*, Denver, CO, 22 July, vol. SPIE Volume 3817, pp. 76-87.

BLACKWELL, A., BRITTON, C., COX, A., GREEN, T., GURR, C., KADODA, G., KUTAR, M., LOOMES, M., NEHANIV, C., PETRE, M., ROAST, C., ROE, C., WONG, A. & YOUNG, R. (2001) "Cognitive Dimensions of Notations: Design Tools for Cognitive Technology," in *Cognitive Technology: Instruments of Mind*. Warwick, UK, 2001, pp. 325-341.

BLACKWELL, A. F. & GREEN, T. R. G. (1998) "Design for usability using Cognitive Dimensions", *Tutorial session at British Computer Society Conference on Human Computer Interaction HCI'98*.

BOHM, A. P. W., DRAPER, B., NAJJAR, W., HAMMES, J., RINKER, B., CHAWATHE, M. & ROSS, C. (2001) "One-step Compilation of Image Processing Applications to FPGAs", *IEEE Symposium on Field-programmable Custom Computing Machines*, Rohnert Park, CA, April 30 -2 May, pp. 209-218.

BOHM, W., BEVERIDGE, R., DRAPER, B., ROSS, C., CHAWATHE, M. & NAJJAR, W. (2002a) "Compiling ATR probing codes for execution on

FPGA hardware", *Proceedings 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* Napa, California, USA, September 22-24, pp. 301-302.

BOHM, W., HAMMES, J., DRAPER, B., CHAWATHE, M., ROSS, C., RINKER, R. & NAJJAR, W. (2002b) "Mapping a Single Assignment Programming Language to Reconfigurable Systems", *The Journal of Supercomputing,* vol. 21, no. 2, pp. 117-130

BOULLIS, N., MENCER, O., LUK, W. & STYLES, H. (2001) "Pipelined function evaluation on FPGAs", *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, Rohnert Park, California, USA, 30 April -2 May, pp. 304-306.

BRUMFITT, P. J. (1984) "Environments for image processing algorithm development", *Image Vision Computing,* vol. 2, no. 4, pp. 198-203

BUCK, J. T. (1993) *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, PhD Thesis in Electrical Engineering and Computer Sciences, University of California, Berkeley.

BUCK, J. T. & LEE, E. A. (1993) "Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model", *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, Minneapolis, USA, 27-30 April, vol. I, pp. 429-432.

BURNS, A. & DAVIES, G. (1993) *Concurrent Programming*, 1 ed. England, Addison-Wesley Publishers Ltd.

CASTLEMAN, K. R. (1996) *Digital Image Processing*, 1 ed. New Jersey, Prentice-Hall.

CELOXICA (2004) Handel-C Software-Compiled System Design, http://www.celoxica.com/methodology/handelc.asp, visited on August 2004.

CELOXICA (2005) *PixelStreams Manual*, 1 ed, Celoxica.

CHANG, S. K., BARNETT, M. M., LEVIALDI, S., MARRIOTT, K., PFEIFFER, J. J. & TANIMOTO, S. L. (1999) "The future of visual languages", *Proceedings IEEE Symposium on Visual Languages.*, Tokyo, Japan, 13-16 September, pp. 58-61.

CHANG, S. K., KORFHAGE, R. R., LEVIALDI, S. & ICHIKAWA, T. (1994) "Ten years of visual languages research", *Proceedings IEEE Symposium on Visual Languages*, St. Louis, Missouri, USA, October 4-7, pp. 196-205.

CHANUSSOT, J., PAINDAVOINE, M. & LAMBERT, P. (1999) "Real time vector median like filter: FPGA design and application to color image filtering", *Proceedings International Conference on Image Processing, ICIP 99*, Chicago, Illinois, USA, 24-28 October, vol. 2, pp. 414-418.

CHRYSOS, G., DOLLAS, A. & BOURBAKIS, N. (2007) "Architecture and design of an embeddable system for SCAN-based compression, encryption and information hiding", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 207-222

CINQUE, L., CANIZARES, S. S. & TANIMOTO, S. (2007) "Application of a transparent interface methodology to image processing", *Journal of Visual Languages & Computing,* vol. 18, no. 5, pp. 504-512

COFFMAN, E. G., ELPHICK, M. & SHOSHANI, A. (1971) "System Deadlocks", *ACM Computing Surveys,* vol. 3, no. 2, pp. 67-78

CONSTANTINIDES, G. (2007) "Special issue on Field-Programmable Technology", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 177-178

COUTINHO, J. G. F. & LUK, W. (2003) "Source-directed transformations for hardware compilation", *Proceedings IEEE International Conference on Field-Programmable Technology (FPT)*, Tokyo, Japan, July, pp. 278-285.

COX, P. T., GILES, F. R. & PIETRZYKOWSKI, T. (1989) "Prograph: a step towards liberating programming from textual conditioning", *IEEE Workshop on Visual Languages*, Rome, Italy, 4-6 October, pp. 150-156.

CROOKES, D. (2009) TULIP - A Language for Image Processing, http://www.cs.qub.ac.uk/~D.Crookes/index2.html#TULIP, visited on September 2009.

CROOKES, D. & BENKRID, K. (1999) "FPGA implementation of image component labelling", *Reconfigurable Technology: FPGAs for Computing and Applications*, Boston, Massachusetts, USA, 20-21 September, vol. SPIE 3844, pp. 17-23.

CROOKES, D., MORROW, P. J. & MCPARLAND, P. J. (1989) "An algebra-based language for image processing on transputers", *Third International Conference on Image Processing and its Applications*, Warwick, UK, 18-20 July, pp. 457-461.

CROOKES, D., MORROW, P. J. & MCPARLAND, P. J. (1990) "IAL: a parallel image processing programming language", *IEE Proceedings Communications, Speech and Vision,* vol. 137, no. 3, pp. 176-182

CUPITT, J., MARTINEZ, K. & PADFIELD, J. (2009) VIPS, http://www.vips.ecs.soton.ac.uk/, visited on January 2009.

DANDEKAR, O., CASTRO-PAREJA, C. & SHEKHAR, R. (2007) "FPGA-based real-time 3D image preprocessing for image-guided medical interventions", *Journal of Real-Time Image Processing,* vol. 1, no. 4, pp. 285-301

DAVID, K. & JOHN, H. (2006) "Hardware join Java: a unified hardware/software language for dynamic partial runtime reconfigurable computing applications", *IEEE International Conference on Field Programmable Technology, FPT 2006* Bangkok, Thailand, December, pp. 277-280.

DAVID STOTTS, P. & FURUTAT, R. (1992) "Hypertextual concurrent control of a Lisp Kernel", *Journal of Visual Languages & Computing,* vol. 3, no. 2, pp. 221-236

DE DINECHIN, F. & TISSERAND, A. (2001) "Some improvements on multipartite table methods", *Proceedings 15th IEEE Symposium on Computer Arithmetic*, Vail, Colorado, USA 11-13 June, pp. 128-135.

DEL BIMBO, A. (1993) "An Iconic Environment for Image-processing Operations", *Journal of Visual Languages & Computing,* vol. 4, no. 3, pp. 267-282

DIAS, T., ROMA, N., SOUSA, L. & RIBEIRO, M. (2007) "Reconfigurable architectures and processors for real-time video motion estimation", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 191-205

DOWNTON, A. & CROOKES, D. (1998) "Parallel architectures for image processing", *Electronics & Communication Engineering Journal,* vol. 10, no. 3, pp. 139-151

DRAPER, B., BEVERIDGE, J. R., BOHM, A. P. W., ROSS, C. & CHAWATHE, M. (2002) "Implementing Image Application on FPGAs", *International Conference on Pattern Recognition*, Quebec City, August 11-15, vol. 3, pp. 265-268.

DRAPER, B., BOHM, A. P. W., HAMMES, J., NAJJAR, W., BEVERIDGE, R., ROSS, C., CHAWATHE, M., DESAI, M. & BINS, J. (2001) "Compiling SA-C Programs to FPGAs: Performance Results", *International Conference on Vision Systems*, Vancouver, 7-8 July, pp. 220-235.

DRAPER, B., NAJJAR, W., BOHM, W., HAMMES, J., RINKER, B., ROSS, C., CHAWATHE, M. & BINS, J. (2000) "Compiling and optimizing image processing algorithms for FPGAs", *Proceedings Fifth IEEE International Workshop on Computer Architectures for Machine Perception*, Padova, Italy, September 11-13, pp. 222-231.

DRAPER, B. A., BEVERIDGE, J. R., BOHM, A. P. W., ROSS, C. & CHAWATHE, M. (2003) "Accelerated image processing on FPGAs", *IEEE Transactions on Image Processing,* vol. 12, no. 12, pp. 1543-1551

DUFF, M. J. B. (2001) "Thirty Years of Parallel Image Processing", *4th International Conference on Vector and Parallel Processing*, Porto, Portugal, 21-23 June, vol. LNCS 1981, pp. 419-438.

EDWARDS, J. (2007) "No Ifs, Ands, or Buts: Uncovering the Simplicity of Conditionals", *Proceedings of the 22th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, Montreal, Canada, 21-25 October, pp. 639-658.

EDWARDS, S. A. (2005) "The challenges of hardware synthesis from C-like languages", *Proceedings Design, Automation and Test in Europe*, Messe Munich, Germany, 7-11 March, vol. 1, pp. 66-67.

EDWARDS, S. A. (2006) "The Challenges of synthesizing hardware from C-like Languages", *IEEE Design & Test of Computers,* vol. 23, no. 5, pp. 375-383

EXMAN, I. & CITRON, D. (1994) "Parallel roadmaps: discrete to continuous", *Proceedings IEEE Symposium on Visual Languages*, St. Louis, Missouri, USA, 4-7 October, pp. 186-188.

FAHMY, S., BOUGANIS, C.-S., CHEUNG, P. & LUK, W. (2007) "Real-time hardware acceleration of the trace transform", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 235-248

FAHMY, S. A., CHEUNG, P. Y. K. & LUK, W. (2005) "Novel FPGA-based implementation of median and weighted median filters for image processing", *International Conference on Field Programmable Logic and Applications*, Tampere University of Technology, Finland, 24-26 August, pp. 142-147.

FOLEY, J. D. & VAN DAM, A. (1982) *Fundamentals of Interactive Computer Graphics*. Reading, Massachusetts, Addison-Wesley.

GANGULY, M. (2001) SystemC Overview, www.systemc.org, visited on August 2004 2003.

GENEST, G., CHAMBERLAIN, R. & BRUCE, R. (2007) "Programming an FPGA-based Super Computer Using a C-to-VHDL Compiler: DIME-C", *Second NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2007*, Palo Alto, USA, 13-15 July, pp. 280-286.

GLINERT, E. P. & NORTON, C. D. (1992) "Novis: a visual laboratory for exploring the design of processor arrays", *Journal of Visual Languages & Computing,* vol. 3, no. 2, pp. 135-159

GLINERT, E. P. & STOTTS, P. D. (1992) "Visual languages and concurrent computing", *Journal of Visual Languages & Computing,* vol. 3, no. 2, pp. 105

GOKHALE, M., STONE, J., ARNOLD, J. & KALINOWSKI, M. (2000) "Stream-oriented FPGA computing in the Streams-C high level language", *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, Napa Valley, USA, 17-19 April, pp. 49-56.

GOLIN, E. J., FENG, A. C., HUANG, L. & HUGHES, E. A. H. E. (1993) "A visual design environment", *IEEE/ACM International Conference on Computer-Aided Design, ICCAD-93*, Santa Clara, California, USA, 7-11 November, pp. 364-367.

GREEN, T. R. G. (1996) "An Introduction to the Cognitive Dimensions Framework", *MIRA workshop*, Monselice, Italy, November.

GREEN, T. R. G. (2000) "Instructions and descriptions: some cognitive aspects of programming and similar activities", *Proceedings of Working Conference on Advanced Visual Interfaces (AVI 2000).* Palermo, Italy, 23-26 May pp. 21-28.

GREEN, T. R. G., BLANDFORD, A. E., CHURCH, L., ROAST, C. R. & CLARKE, S. (2006) "Cognitive dimensions: Achievements, new directions, and open questions", *Journal of Visual Languages & Computing,* vol. 17, no. 4, pp. 328-365

GREEN, T. R. G. & PETRE, M. (1992) "When Visual Programs are Harder to Read than Textual Programs", *Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics) Human-Computer Interaction: Tasks and Organisation* pp. 167-180.

GREEN, T. R. G. & PETRE, M. (1996) "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework", *Journal of Visual Languages and Computing,* vol. 7, no. 2, pp. 131-174

GRIBBON, K. T. & BAILEY, D. G. (2004) "A Novel Approach to Real Time Bilinear Interpolation", *The IEEE International Workshop on Electronic Design, Test, and Applications*, Perth, Australia, 28-30 January, pp. 126-131.

GRIBBON, K. T., BAILEY, D. G. & BAINBRIDGE-SMITH, A. S. L. (2007) "Development Issues in Using FPGAs for Image Processing", *Image and Vision Computing New Zealand*, Hamilton, New Zealand, 5-7 December, pp. 282-287

GRIBBON, K. T., BAILEY, D. G. & JOHNSTON, C. T. (2004) "Colour edge enhancement", *Proceedings of Image and Vision conference New Zealand*, Akaroa, N.Z., 21-23 November, pp. 297-302.

GRIBBON, K. T., JOHNSTON, C. T. & BAILEY, D. G. (2003) "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, 26-28 November, pp. 408-413.

GRIBBON, K. T., JOHNSTON, C. T. & BAILEY, D. G. (2005) "Formalizing Design Patterns for Image Processing Algorithm Development on FPGAs", *IEEE Tencon*, Melbourne, Australia, 21-24 November.

GRIBBON, K. T., JOHNSTON, C. T. & BAILEY, D. G. (2006) "Using Design Patterns to Overcome Image Processing Constraints on FPGAs", *3rd IEEE International Workshop on Electronic Design, Test, and*

Applications (Delta 2006), Kuala Lumpur, Malaysia, 17-19 January, pp. 405-409.

GUPTA, S., DUTT, N., GUPTA, R. & NICOLAU, A. (2003) "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations", *Proceedings 16th International Conference on VLSI Design*, New Delhi, India, 4-8 January, pp. 461-466.

HALDAR, M., NAYAK, A., CHOUDHARY, A. & BANERJEE, P. (2001a) "Automated synthesis of pipelined designs on FPGAs for signal and image processing applications described in MATLAB(R)", *Proceedings of the ASP-DAC, Asia and South Pacific Design Automation Conference*, Bangalore, India, 21-24 January, pp. 645-648.

HALDAR, M., NAYAK, A., CHOUDHARY, A. & BANERJEE, P. (2001b) "A system for synthesizing optimized FPGA hardware from Matlab(R)", *International Conference on Computer Aided Design, ICCAD*, San Jose, CA, USA, 4-8 November pp. 314-319.

HALDAR, M., NAYAK, A., KANHERE, A., JOISHA, P., SHENOY, N., CHOUDHARY, A. & BANERJEE, P. (2000) "Match virtual machine: an adaptive runtime system to execute MATLAB in parallel", *Proceedings International Conference on Parallel Processing*, Toronto, Canada, 21-24 August, pp. 145-152.

HALDAR, M., NAYAK, A., SHENOY, N., CHOUDHARY, A. & BANERJEE, P. (2001c) "FPGA hardware synthesis from MATLAB", *Fourteenth International Conference on VLSI Design*, Bangalore, India, 3-7 January pp. 299-304.

HAMMES, J., BOHM, A. P. W., ROSS, C., CHAWATHE, M., DRAPER, B. & NAJJAR, W. (2001a) "High Performance Image Processing on FPGAs", *Los Alamos Computer Science Institute Symposium*, Santa Fe, NM, USA, 15-18 October.

HAMMES, J., BOHM, W., ROSS, C., CHAWATHE, M., DRAPER, B., RINKER, B. & NAJJAR, W. (2001b) "Loop Fusion and Temporal Common Subexpression Elimination in Window-based Loops", *IPDPS 8th Reconfigurable Architectures Workshop*, San Francisco, CA, 27 April, pp. 1433-1440.

HAMMES, J., RINKER, B., BOHM, W., NAJJAR, W., DRAPER, B. & BEVERIDGE, R. (1999a) "Cameron: high level language compilation for reconfigurable systems", *Proceedings International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, California, USA, 12-16 October, pp. 236-244.

HAMMES, J., RINKER, R., BOHM, W. & NAJJAR, W. (1999b) "Compiling a High-level Language to Reconfigurable Systems", *Compiler and Architecture Support for Embedded Systems (CASES)*, Washington, DC, October.

HAMMES, J., RINKER, R., NAJJAR, W. & DRAPER, B. (2000) "A High-level, Algorithmic Programming Language and Compiler for Reconfigurable Systems", *The 2nd International Workshop on the Engineering of Reconfigurable Hardware/Software Objects (ENREGLE), part of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, NV, 26-29 June.

HAREL, D. (1987) "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, vol. 8, no. 3, pp. 231-274

HASSAN, F. & CARLETTA, J. (2007) "An FPGA-based architecture for a local tone-mapping operator", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 293-308

HERBORDT, M. C., VANCOURT, T., GU, Y., SUKHWANI, B., CONTI, A., MODEL, J. & DISABELLO, D. (2007) "Achieving High Performance with FPGA-Based Computing", *Computer,* vol. 40, no. 3, pp. 50-57

HOARE, C. A. R. (1985) *Communicating Sequential Processes* London, UK, Prentice-Hall.

HOISKO, S., HAKKARAINEN, H., VIHAVAINEN, K. & ISOAHO, J. (1996) "Specification, hardware implementation and prototyping environment for image processing algorithms", *IEEE International Symposium on Circuits and Systems, ISCAS '96.*, Atlanta, USA, 12-15 May vol. 4, pp. 834-837.

IEEE (2008) IEEE Standard Verilog Hardware Description Language, http://www.verilog.com/IEEEVerilog.html, visited on August 2008.

IMPULSEC (2008) ImpulseC, http://www.impulsec.com/, visited on March 2008.

INSTRUMENTS, N. (2005) National Instruments LabVIEW, www.ni.com/labview, visited on 16 February 2005.

INTEL (2006) opencv, http://www.intel.com/technology/computing/opencv/overview.htm, visited on December 2006.

JABLONSKI, M. & GORGON, M. (2004) "Handel-C implementation of classical component labelling algorithm", *Euromicro Symposium on Digital System Design (DSD 2004)*, Rennes, France, 31 August - 3 September, pp. 387-393.

JACKSON, P. A., HUTCHINGS, B. L. & TRIPP, J. L. (2003) "Simulation and synthesis of CSP-based interprocess communication", *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2003*, Napa, California, 9-11 April, pp. 218-227.

JHDL (2005) JHDL, www.jhdl.org, visited on 21 February 2005.

JOHNSTON, C. T. (2003) *Real-time FPGA Implementation of a Barrel Distortion Correction Algorithm*, Honours Project Thesis in Engineering, IIST, College of Science, Massey University, Palmerston North.

JOHNSTON, C. T. & BAILEY, D. G. (2003) " Real-time FPGA Implementation of a Barrel Distortion Correction Algorithm", *Projects,* vol. 12, pp. 91-96

JOHNSTON, C. T. & BAILEY, D. G. (2008) "FPGA implementation of a Single Pass Connected Components Algorithm", *IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2008)*, Hong Kong, 23-25 January, pp. 228-231.

JOHNSTON, C. T., BAILEY, D. G. & GRIBBON, K. T. (2005a) "Optimisation of a colour segmentation and tracking algorithm for real-time FPGA implementation", *Proceedings of Image and Vision Computing New Zealand*, Dunedin, 28-29 November, pp. 422-427.

JOHNSTON, C. T., BAILEY, D. G. & LYONS, P. (2006a) "Towards a visual notation for pipelining in a visual programming language for programming FPGAs", *7th International Conference of the NZ chapter of the ACM's Special Interest Group on Human-Computer Interaction*, Christchurch, New Zealand, 6-7 July, pp. 1-9.

JOHNSTON, C. T., BAILEY, D. G. & LYONS, P. (2006b) "A Visual Environment for Real-Time Image Processing in Hardware (VERTIPH)", *EURASIP Journal on Embedded Systems, (Article ID 72962),* vol. 2006, pp. 1-8

JOHNSTON, C. T., BAILEY, D. G., LYONS, P. & GRIBBON, K. T. (2004a) "Formalisation of a visual environment for real time image processing in hardware (VERTIPH)", *Proceedings of Image and Vision Computing New Zealand*, Akaroa, N.Z., 21-23 November, pp. 291-296.

JOHNSTON, C. T., GRIBBON, K. T. & BAILEY, D. G. (2004b) "Implementing Image Processing Algorithms on FPGAs", *Proceedings of the Eleventh Electronics New Zealand Conference, ENZCon'04*, Palmerston North, 15-16 November, vol. 11, pp. 118-123.

JOHNSTON, C. T., GRIBBON, K. T. & BAILEY, D. G. (2005b) "FPGA based Remote Object Tracking for Real-time Control", *International Conference on Sensing Technology*, Palmerston North, New Zealand, 21-23 November, pp. 66-72.

JOHNSTON, C. T., LYONS, P. & BAILEY, D. G. (2008) "A Visual Notation for Processor and Resource Scheduling", *IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2008)*, Hong Kong, 23-25 January, pp. 296-301.

JOHNSTON, C. T., LYONS, P. & BAILEY, D. G. (2009) "User Evaluation and Overview of a Visual Language for Real Time Image Processing on FPGAs", *10th Annual Conference of the NZ ACM Special Interest Group on Human-Computer Interaction (CHINZ 2009)*, Auckland, New Zealand, 6-7 July, pp. 85-92.

JOHNSTON, W. M., HANNA, J. R. P. & MILLAR, R. J. (2004c) "Advances in dataflow programming languages", *ACM Computing Surveys* vol. 36, no. 1, pp. 1-34

JUAN, H.-P., HOLMES, N. D., BAKSHI, S. & GAJSKI, D. D. (1993) "Top-down modeling of RISC processors in VHDL", *European Design Automation Conference, 1993, with EURO-VHDL '93*, Hamburg, Germany, 20-24 September, pp. 454-459.

KERHET, A., MAGNO, M., LEONARDI, F., BONI, A. & BENINI, L. (2007) "A low-power wireless video sensor node for distributed object detection", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 331-342

KESSAL, L., ABEL, N., KARABERNOU, S. & DEMIGNY, D. (2008) "Reconfigurable computing: design methodology and hardware tasks scheduling for real-time image processing", *Journal of Real-Time Image Processing,* vol. 3, no. 3, pp. 131-147

KONSTANTINIDES, K. & RASURE, J. R. (1994) "The Khoros software development environment for image and signal processing", *IEEE Transactions on Image Processing,* vol. 3, no. 3, pp. 243-252

KRAZLMULLER, D., STANKOVIC, N. & VOLKERT, J. (1999) "Debugging parallel programs with visual patterns", *Proceedings IEEE Symposium on Visual Languages*, Tokyo, Japan, 13-16 September, pp. 180-181.

KU, D. & DEMICHELI, G. (1990) *HardwareC -- A Language for Hardware Design (Version 2.0)*, Stanford University.

LEVINE, B., NATARAJAN, S., TAN, C., NEWPORT, D. & BOULDIN, D. (1999) "Mapping of an automated target recognition application from a graphical software environment to FPGA-based reconfigurable hardware", *Proceedings. Seventh Annual IEEE Symposium on Field-*

Programmable Custom Computing Machines, FCCM '99, Napa Valley, Napa, California 20-23 April, pp. 292-293.

LINDOSO, A. & ENTRENA, L. (2007) "High performance FPGA-based image correlation", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 223-233

LOYALL, J. P. & KAPLAN, S. M. (1992) "Visual concurrent programming with [Delta]-grammars", *Journal of Visual Languages & Computing,* vol. 3, no. 2, pp. 107-133

MA, N., BAILEY, D. & JOHNSTON, C. (2008) "Optimised single pass connected components analysis", *International Conference on Field Programmable Technology*, Taiwan, 8-10 December, pp. 185-192.

MATHWORKS (2005a) Matlab and Simulink, http://www.mathworks.com/, visited on 12 February 2005.

MATHWORKS (2005b) Simulink 6.1, www.mathworks.com/products/simulink/, visited on 16 February 2005.

MENCER, O. & LUK, W. (2004) "Parameterized high throughput function evaluation for FPGAs", *Journal of VLSI Signal Processing,* vol. 36, no. 1, pp. 17-25

MENCER, O., PEARCE, D. J., HOWES, L. W. & LUK, W. (2003) "Design space exploration with A Stream Compiler", *Proceedings IEEE International Conference on Field-Programmable Technology (FPT)*, Tokyo, Japan, 15-17 December pp. 270-277.

MENGXIANG, L. & LAVEST, J. M. (1996) "Some aspects of zoom lens camera calibration", *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 18, no. 11, pp. 1105-1110

MILLER-KARLOW, D. L. & GOLIN, E. J. (1992) "vVHDL: a visual hardware description language", *IEEE Workshop on Visual Languages.*, Seattle, Washington, USA, 15-18 September pp. 133-139.

MIRIYALA, S., AGHA, G. & SAMI, Y. (1992) "Visualizing actor programs using predicate transition nets", *Journal of Visual Languages & Computing,* vol. 3, no. 2, pp. 195-220

MYERS, B. A. (1986) "Visual programming, programming by example, and program visualization: a taxonomy", *Proceedings of the SIGCHI conference on Human factors in computing systems*, Boston, Massachusetts, United States, pp. 59-66.

NAJJAR, W. A., BOHM, W., DRAPER, B. A., HAMMES, J., RINKER, R., BEVERIDGE, J. R., CHAWATHE, M. & ROSS, C. (2003) "High-level language abstraction for reconfigurable computing", *IEEE Computer,* vol. 36, no. 8, pp. 63-69

NASSI, I. & SHNEIDERMAN, B. (1973) "Flowchart techniques for structured programming", *ACM SIGPLAN Notices,* vol. 8, no. 8, pp. 12-26

NAYAK, A., HALDAR, M., CHOUDHARY, A. & BANERJEE, P. (2001) "Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs", *Proceedings Design, Automation and Test in Europe*, Munich, Germany, 12-16 March pp. 722-728.

NETBEANS (2008), www.netbeans.org, visited on December 2008.

NGAN, P. M. (1992) *The Development of a Visual Language for Image Processing Applications*, Ph.D. dissertation Thesis in Computer Science, Massey University, Palmerston North, New Zealand.

NGO, H. T. & ASARI, V. K. (2005) "A pipelined architecture for real-time correction of barrel distortion in wide-angle camera images", *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 15, no. 3, pp. 436-444

OBJECT MANAGEMENT GROUP (2009) UML, http://www.uml.org/, visited on September 2009.

OFFEN, R. J. (Ed.) (1985) *VLSI Image Processing,* London, Collins.

PAGE, I. & LUK, W. (1991) "Compiling Occam into field-programmable gate arrays", *Proceedings of the Field Programmable Logic and Applications*, Oxford, UK, 4-6 September, pp. 271-283.

PEARSON, M. W. (1992) *PICSIL: Design and Synthesis of Digital ICs from Data Flow Diagrams*, PhD Thesis in Computer Science, Massey University, Palmerston North.

PEARSON, M. W., LYONS, P. J. & APPERLEY, M. D. (1996) "High-level Graphical Abstraction in Digital Design", *VLSI Design,* vol. 5, no. 1, pp. 101-110

PELL, O. & LUK, W. (2005) "Quartz: a framework for correct and efficient reconfigurable design", *International Conference on Reconfigurable Computing and FPGAs, ReConFig 2005*, Puebla City, Mexico, 28-30 September pp. 8 pp.

PETRE, M. (2006) "Cognitive dimensions 'beyond the notation'", *Journal of Visual Languages & Computing,* vol. 17, pp. 292-301

QUINN, H., LEESER, M. & SMITH KING, L. (2007) "Dynamo: a runtime partitioning system for FPGA-based HW/SW image processing systems", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 179-190

RAMIREZ-AGUNDIS, A., GADEA-GIRONES, R., COLOM-PALERO, R. & DIAZ-CARMONA, J. (2007) "A wavelet-VQ system for real-time video compression", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 271-280

RATHA, N. K. & JAIN, A. K. (1999) "Computer vision algorithms on reconfigurable logic arrays", *Parallel and Distributed Systems, IEEE Transactions on,* vol. 10, no. 1, pp. 29-43

RINKER, R., CARTER, M., PATEL, A., CHAWATHE, M., ROSS, C., HAMMES, J., NAJJAR, W. A. & BOHM, W. (2001) "An automated process for compiling dataflow graphs into reconfigurable hardware", *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* vol. 9, no. 1, pp. 130-139

RINKER, R., HAMMES, J., NAJJAR, W. A., BOHM, W. & DRAPER, B. (2000) "Compiling image processing applications to reconfigurable hardware", *Proceedings. IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, Boston, MA, 10-12 July, pp. 56-65.

ROMAN, G.-C., COX, K. C., WILCOX, C. D. & PLUN, J. Y. (1992) "Pavane: a system for declarative visualization of concurrent computations", *Journal of Visual Languages & Computing,* vol. 3, no. 2, pp. 161-193

ROSENFELD, A. & PFALTZ, J. (1966) "Sequential Operations in Digital Picture Processing", *Journal of the ACM,* vol. 13, no. 4, pp. 471-494

RUSS, J. C. (2002) *The Image Processing Handbook*, Fourth ed. Boca Raton, CRC Press.

SAEGUSA, T. & MARUYAMA, T. (2007) "An FPGA implementation of real-time K-means clustering for color images", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 309-318

SAMEK, M. (2002) *Practical Statecharts in C C++: An Introduction to Quantum Programming with CDROM*, CMP Books.

SANDERSON, C. (2004) *Simplify FPGA Application Design with DIMEtalk. Xcell Journal.*

SANDERSON, C. & SHAND, D. (2005) *FPGAs Supplant Processors and ASICs in Advances Imaging Application. FPGA and Programmable Logic Journal.*

SCHERMERHORN, J. R. (2001) *Management*, Sixth ed. New York, John Wiley & Sons.

SCHILDT, H. (1997) *Programmer's Reference C/C++*. Berkeley, Osborne McGraw-Hill.

SCHILDT, H. (2002) *Java2: The Complete Reference*, Fifth ed. Berkeley, McGraw-Hill/Osborne.

SEDCOLE, N. P. (2006) *Reconfigurable Platform-Based Design in FPGAs for Video Image Processing*, Thesis in Department of Electrical and Electronic Engineering, Imperial College of Science, Technology and Medicine, University of London, London.

SEN GUPTA, G. & BAILEY, D. G. (2004) "A new colour-space for efficient and robust segmentation", *Proceedings of Image and Vision conference New Zealand*, Akaroa, N.Z., 21-23 November, pp. 315-320.

SEN, M., CORRETJER, I., HAIM, F., SAHA, S., SCHLESSMAN, J., LV, T., BHATTACHARYY, S. S. & WOLF, W. (2007) "Dataflow-Based Mapping of Computer Vision Algorithms onto FPGAs", *EURASIP Journal on Embedded Systems,* vol. 2007, no. Article ID 49236, pp. 12

SEROT, J., QUENOT, G. & ZAVIDOVIQUE, B. (1995) "A visual Dataflow Programming Environment for a Real Time Parallel Vision Machine", *Journal of Visual Languages and Computing,* vol. 6, pp. 327-347

SHEEHAN, R. (2003) "Parallelism in the Icicle programming environment", *Proceedings IEEE Symposium on Human Centric Computing Languages and Environments*, Auckland, New Zealand 28-31 October pp. 53-55.

SHNEIDERMAN, B. (1983) "Direct Manipulation: A Step Beyond Programming Languages", *Computer,* vol. 16, no. 8, pp. 57-69

SHU, N. C. (1986) "Visual Programming Languages: A Perspective and a Dimensional Analysis," in *Visual Languages*. New York, USA: Plenum Publishing Corporation, 1986, pp. 11–34.

SMACH, F., MITERAN, J., ATRI, M., DUBOIS, J., ABID, M. & GAUTHIER, J.-P. (2007) "An FPGA-based accelerator for Fourier Descriptors computing for color object recognition using SVM", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 249-258

SMEDLEY, T. J. (1995) "A high-level visual language for the graphical description of digital circuits", *Proceedings 11th IEEE International Symposium on Visual Languages* Darmstadt, Germany, 5-9 September pp. 77-82.

SOSA, J., BOLUDA, J., PARDO, F. & GÓMEZ-FABELA, R. (2007) "Change-driven data flow image processing architecture for optical flow computation", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 259-270

STACEY, M. (1995) "Distorting Design: Unevenness as a Cognitive Dimension of Design Tools", *The 10th Annual Conference of the British HCI Group Adjunct Proceedings of HCI'95: People and Computers*, University of Huddersfield, Huddersfield, UK, 29 August - 1 September, pp. 90-95.

STEELE, J. A. (1994) *An abstract machine approach to environments for image interpretation on transputers*, Thesis in Computer Science, The Queen's University of Belfast, Belfast.

STOKES, J. (2007) *Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture*, No Starch Press.

SWAN, S. (2001) *An Introduction to System Level Modeling in SystemC 2.0*, Open SystemC Initiative.

SWENSON, R. L. & DIMOND, K. R. (1999) "A hardware FPGA implementation of a 2D median filter using a novel rank adjustment technique", *Seventh International Conference on Image Processing And Its Applications (Conf. Publ. No. 465)*, Dublin, Ireland, 14-17 July vol. 1, pp. 103-106.

SZE-WEI, O., KERKIZ, N., SRIJANTO, B., CHANDRA, T., LANGSTON, M., NEWPORT, D. & BOULDIN, D. (2001) "Automatic Mapping of Multiple Applications to Multiple Adaptive Computing Systems", *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '01.* , Rohnert Park, California 29 April - 2 May, pp. 10-20.

TANIMOTO, S. L. (1990) "VIVA: A Visual Language for Image Processing", *Journal of Visual Languages & Computing,* vol. 1, pp. 127-139

TODMAN, T. J., CONSTANTINIDES, G. A., WILTON, S. J. E., MENCER, O., LUK, W. & CHEUNG, P. Y. K. (2005) "Reconfigurable computing: architectures and design methods", *Computers and Digital Techniques, IEE Proceedings -,* vol. 152, no. 2, pp. 193-207

TRIEU, D. & MARUYAMA, T. (2007) "Real-time image segmentation based on a parallel and pipelined watershed algorithm", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 319-329

TSCHUMPERLÉ, D. (2008) The CImg Library, http://cimg.sourceforge.net/, visited on December 2008.

USHER, M. & JACKSON, D. (1998) "A concurrent visual language based on Petri nets", *Proceedings IEEE Symposium onVisual Languages*, Nova Scotia, Canada, September 1-4, pp. 72-73.

VELTEN, J. & KUMMERT, A. (2002) "FPGA-based implementation of variable sized structuring elements for 2D binary morphological operations", *The First IEEE International Workshop on Electronic Design, Test and Applications*, Christchurch, New Zealand, 29-31 January, pp. 309-312.

VIKKI, F., SUSAN, W. & JEAN, S. (1993) "Mental representations of programs by novices and experts", *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, Amsterdam, The Netherlands, 24-29 April, pp. 74-79.

WARE, C. (1993) "The Foundations of Experimental Semiotics: a Theory of Sensory and Conventional Representation", *Journal of Visual Languages and Computing,* vol. 4, pp. 91-100

WEINHARDT, M. & LUK, W. (2001) "Pipeline vectorization", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 20, no. 2, pp. 234-248

WILLIAMS, C. S. & RASURE, J. R. (1990) "A visual language for image processing", *Proceedings of the IEEE Workshop on Visual Languages*, 4-6 October, pp. 86-91.

WILSON, J. N. (2009) Theoretical Impact of Image Algebra, Gainesville, Florida, http://www.cise.ufl.edu/~jnw/IA/ia-theory.html, visited on September 2009.

WILSON, J. N., RITTER, G. X. & FISCHER, R. (1998) "Implementation and use of an Image Processing Algebra for Programming Massively Parallel Machines, " *IEEE 2nd Symposium of the Frontiers of Massively Parallel Computation*, George Mason Univ., Fairfax, VA pp. 12-31.

WOLBERG, G. (1990) *Digital Image Warping*, first ed. Los Alamitos, IEEE Computer Society Press.

WU, K., OTOO, E. & SHOSHANI, A. (2005) "Optimizing Connected Component Labeling Algorithms", *Medical Imaging 2005: Image Processing,* vol. 5747, pp. 1965-1976

XILINX (2008) ISE, http://www.xilinx.com/products/design_resources/design_tool/index.htm, visited on December 2008.

YAMADA, A., NISHIDA, K., SAKURAI, R., KAY, A., NOMURA, T. & KAMBE, T. (1999) "Hardware synthesis with the Bach system", *Proceedings of the 1999 IEEE International Symposium on Circuits and Systems, ISCAS '99.* , Orlando, Florida, USA, May 30 - June 2, vol. 6, pp. 366-369

ZHANG, C., LONG, Y. & KURDAHI, F. (2007) "A hierarchical pipelining architecture and FPGA implementation for lifting-based 2-D DWT", *Journal of Real-Time Image Processing,* vol. 2, no. 4, pp. 281-291