

Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

# **GateOS: A minimalist Windowing Environment and Operating System for FPGAs**

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

**Master of Engineering  
In  
Computer Systems Engineering**

At Massey University, Palmerston North,  
New Zealand

**Andreas Buhler**

**2007**

# ABSTRACT

In order to debug and tune stand-alone FPGA image processing configurations, it is necessary for a developer to also create the required debug tools and to implement them on the FPGA. This process takes both time and effort that could be better spent on improving the image processing algorithms. The Gate Array Terminal Operating System (GateOS) is proposed to relieve the developer of the need to construct many of these debugging tools. In GateOS we separate the image processing algorithms from the rest of the operating system. GateOS is presented to the developer as a Handel-C library, which can be customised at compile-time, to facilitate the creation of windows and widgets. Several types of widgets are described that can manipulate the parameters of image processing algorithms and enable the end-user to dynamically rearrange the position of a window on the VDU. An end user is able to interact with GateOS with both a keyboard and a mouse.

# ACKNOWLEDGEMENTS

I would like to thank my supervisor, Associate Professor Donald Bailey, for all the guidance given to me over the previous year. Without his sound advice, GateOS would not be what it is today. I thank him for being brutally honest with me whenever I became sidetracked and deviated from dealing with the core issues of GateOS. I am also grateful for the considerable amount of time and patience expended on his behalf proofreading this thesis. Personally, I am astounded at how he was able to do this on his very tight schedule. I am thankful to Donald and Massey University for allowing me to use and take home various FPGA development boards. I have appreciated all the advice given to me by Chris Johnston and Kim Gribbon during the year. Their ideas for GateOS, recorded in a conference paper (Bailey et al, 2006) have proved invaluable in formulating the requirements of GateOS and to some extent, its design. I would like to thank Massey University for allowing me to complete the research for my thesis using their equipment in the computer labs.

I also acknowledge the role of Xilinx in providing me with licences to the ISE suite of software tools, which were required for the implementation of GateOS.

Finally, I would like to thank Roger Gook of Celoxica who has provided me with licences for the DK suite of software, so that I was able to use Handel-C to implement GateOS. He has also been very helpful by providing me with the source code for various board level drivers.

# TABLE OF CONTENTS

ABSTRACT.....	i
ACKNOWLEDGEMENTS.....	ii
TABLE OF CONTENTS .....	iii
LIST OF FIGURES .....	vi
LIST OF TABLES.....	viii
CHAPTER One INTRODUCTION & LITERATURE REVIEW.....	1
CHAPTER Two REQUIREMENTS ANALYSIS & HIGH LEVEL DESIGN .....	7
2.1 Introduction.....	7
2.2 Proposal .....	8
2.3 Hardware Environment .....	9
2.4 System Overview.....	11
2.5 Non-Volatile Storage Manager.....	14
CHAPTER Three WINDOW MANAGEMENT.....	15
3.1 Introduction.....	15
3.2 Representing a window's size and location .....	16
3.2.1 Virtual Coordinate Space .....	17
3.2.2 Window extent .....	18
3.3 Overlap of windows.....	19
3.4 Window States .....	20
3.5 Window Regions .....	21
3.6 Managing and displaying windows.....	23
3.6.1 Parallel Incidence Test.....	23
3.6.2 Window Transition Method.....	26
3.7 Discussion.....	34
CHAPTER Four WIDGET MANAGEMENT .....	36
4.1 Introduction.....	36
4.2 Widget Display Layers .....	37
4.3 Widget Details.....	38
4.3.1 Label Widget.....	38

4.3.2	Button Widget.....	39
4.3.3	Text Edit Widget.....	40
4.3.4	Slider Widget .....	43
4.3.5	Imaging Widgets .....	46
4.4	Data-Structures and Interactions.....	50
4.4.1	Imaging widgets.....	52
4.5	Scheduling and Display of Widgets .....	53
4.5.1	Image Layer.....	53
4.5.2	Algorithm Control Layer .....	53
4.5.3	Window Control Layer .....	56
4.6	Discussion.....	56
CHAPTER Five MANAGEMENT & DISPLAY OF TEXT .....		58
5.1	Introduction.....	58
5.2	Requirements and Intended Usage.....	59
5.2.1	Window Labels.....	59
5.2.2	Widget Annotations.....	60
5.2.3	Output Text Box .....	61
5.2.4	Image annotations.....	61
5.2.5	Text editing .....	61
5.3	Analysis and Design .....	62
5.3.1	Image-Table Lookup Method.....	63
5.3.2	Font Table Lookup Method .....	66
5.4	Implementation.....	71
5.5	Discussion.....	73
CHAPTER Six INPUT MANAGEMENT.....		75
6.1	Introduction.....	75
6.2	Cursor Layer.....	76
6.3	Keyboard Input.....	79
6.4	Mouse Input .....	79
6.5	Discussion.....	81
CHAPTER Seven DISCUSSION & CONCLUSIONS.....		82

REFERENCES .....87

# LIST OF FIGURES

Figure 2-1 Timing Regions within a video frame, active, H-blanking and V-blanking .	10
Figure 2-2 GateOS IP Core and its relationship to the rest of GateOS.....	12
Figure 3-1 Windowing System architectural overview .....	16
Figure 3-2 Choices of origin within virtual coordinate space.....	17
Figure 3-3 Virtual coordinate space based on the two encoding schemes.....	19
Figure 3-4 Changing the z-index of a window.....	20
Figure 3-5 Final layer with a grip for resizing windows.....	22
Figure 3-6 Detecting whether a window is present at a screen coordinate.....	24
Figure 3-7 Updated window identification that uses less hardware per window .....	25
Figure 3-8 Selecting the window with the highest z-index.....	25
Figure 3-9 How WLL entries for each scan-line are related to window positions .....	28
Figure 3-10 Worst case scenario with each windows vertical edges displayed.....	28
Figure 3-11 Edge list lookup-table.....	30
Figure 4-1 Content window layer split into three widget sub-layers.....	37
Figure 4-2 A simple Label Widget .....	38
Figure 4-3 The background of a Button Widget can be one of two possible colors.....	40
Figure 4-4 An Edit Widget displaying the string “The brown fox jumped” .....	41
Figure 4-5 Vertical and horizontal slider widgets.....	43
Figure 4-6 The offset of a horizontal Slider Bar from the left border.....	44
Figure 4-7 Two zoom buttons, one in and the other out .....	46
Figure 4-8 Slider widget is used to manipulate denominator of zoom fraction.....	47
Figure 4-9 Sample Histogram Content .....	48
Figure 4-10 B_VALUE and I_VALUE data structures.....	51
Figure 4-11 The data structures necessary for each widget type .....	51
Figure 4-12 Aligning common properties between widgets .....	52
Figure 4-13 Valid and invalid widget layouts.....	54
Figure 4-14 Order of widget data structures is determined by row then column .....	55
Figure 5-1 A simple textual label displayed on a window.....	60
Figure 5-2 String annotations on label and button widgets.....	60
Figure 5-3 The construction of the String LUT at compile-time.....	64
Figure 5-4 The use of an external utility to construct Image Table.....	65



Figure 5-5 Possible implementation strategies .....	66
Figure 5-6 Using bitmaps for individual characters.....	67
Figure 5-7 The string id and how it relates to the string, offset, and length tables.....	69
Figure 5-8 The Text Manager design.....	70
Figure 5-9 GUI Utility that manipulates the String Table .....	72

# LIST OF TABLES

Table 4-1 Cursor movements .....	42
Table 4-2 Available numeric string conversions .....	43
Table 7-1 Slices used on a Xilinx ML402 (Virtex 4) for 8, 16, 32 and 64 windows.....	84
Table 7-2 Breakdown of resources required, as estimated by the Celoxica build tools .	85

# CHAPTER ONE

## INTRODUCTION & LITERATURE REVIEW

### Chapter Outline

This chapter introduces GateOS, a windowing environment for FPGAs. It outlines how image processing algorithms implemented on an FPGA can be debugged using either a hosted or stand-alone development model, and in addition, it describes the advantages and disadvantages of each model. Prior research on the application of the hosted development model is reviewed. Very little of existing research relates directly to the stand-alone development model, and it is this disparity that GateOS attempts to remedy.

Recently, there has been significant research into the applications of Field Programmable Gate Arrays (FPGAs), for example: (Chan et al., 2007; Hemmert & Underwood, 2007; Tahoori, 2006; Tessier et al., 2007; Yiannacouras et al., 2007). This increased interest is largely sparked by innovations in semiconductor manufacturing, thus resulting in a reduction in IC die sizes and an increased gate count. As a result, FPGAs are now more powerful and less expensive than they were previously. Consequently, new medium level hardware description languages (HDLs) such as JHDL, Handel-C and SAP have evolved to program these newer generations of FPGAs. Companies such as Xilinx, Altera and Lattice Semiconductor lead the field when it comes to the design and manufacture of FPGAs. Celoxica is a good example of a company that provides the software to program these FPGA such as Handel-C and System-C.

A common application of FPGAs is in the field of image processing (Benitez, 2002; Yano et al., 1999; Crookes et al., 1998; Gribbon et al, 2006; Hsiao et al., 2005; Johnston et al, 2005; Uzun & Bouridane, 2003) due to the high levels of parallelism that are available to the developer. The hardware programming language Handel-C is commonly used by such developers for the task of implementing image processing algorithms on FPGAs (McCurry et al., 2001; Ramdas et al., 2004; Vitabile et al., 2004).

The focus of my research is to develop tools that can be used to debug image processing algorithms that are implemented on an FPGA. For the development and subsequent debugging of an FPGA solution, a developer may adopt one of two possible development models; the hosted or standalone. The hosted model describes the situation where a host computer or device directly controls a hardware program running on a slave FPGA. The FPGA is effectively acting as a co-processor to the host system, thus accelerating the computational tasks assigned to it. This model is by far the most popular with developers, because the development and testing of the software running on the host computer is easier and is more user-friendly since the host system provides the tools for interactive debugging of the algorithms. The stand-alone model is where the complete system, including control and debug logic, is implemented directly on the FPGA itself. Developing the control and debug logic onboard the FPGA requires more time and effort on behalf of the developer, since it is necessary to first create the visual environment before constructing the debug tools. This fact has resulted in a preference for the hosted model.

Very little has been written on debugging applications using the hosted development model. One paper (Tomko & Tiwari, 2000) describes the use of the read-back capabilities of FPGAs (involves the serial transmission through the JTAG<sup>1</sup> interface of check bits available on each CLB<sup>2</sup>) to debug applications

---

<sup>1</sup> Joint Test Action Group

<sup>2</sup> Configurable Logic Block

targeted towards reconfigurable computing. Such an approach may be beneficial in some situations, but due to the large volume of data involved in image processing, it is unsuitable for debugging these types of algorithms.

ChipScope Pro<sup>3</sup> is a commercial product by Xilinx that offers low-level debugging of FPGA software (using test vectors). Test vectors are used to validate the outputs of a system with regards to a series of known inputs. A large number of test vectors are commonly used to thoroughly test all aspects of an FPGA design, both in simulation and in the implementation. A significant number of test vectors would be required for image processing algorithms and may require some time to construct, which has the add-on effect of increasing the overall complexity of the task. Transmitting such a large number of test vectors between the FPGA and the host PC would require considerable bandwidth (which is limited). Because of this, the use of test vectors is perhaps less appropriate for the debugging of image processing algorithms on FPGAs. For these reasons, in image processing, it can get difficult to completely test each aspect of an algorithm. However, if the image processing algorithm is first developed and tested on a standard software platform, all that really needs to be debugged is the correctness of the individual operations, which is a simpler task.

For most types of application, the hosted development model is the most suitable choice. However, for debugging image processing applications, this model imposes several restrictions that are not apparent when using the stand-alone development model. The first major issue is the latency of the test and debug signals (when transmitting complete images or test vectors) between the host PC and the client FPGA, and the second is the limited bandwidth with which to transmit the intermediate images which are the output of the intermediate stages of an image processing algorithm. This problem can be partially overcome by using memory that is shared between the host system

---

<sup>3</sup> [http://www.xilinx.com/ise/optional\\_prod/cspro.htm](http://www.xilinx.com/ise/optional_prod/cspro.htm)

and the FPGA for transferring images. The stand-alone development model suffers from none of these restrictions; however, to implement the debugging tools directly on the FPGA requires significantly more hardware resources and will increase the complexity of the FPGA design.

From an image processing perspective, it is important to be able to view images at different stages of the algorithm, and also to see the effects of adjusting various algorithm tuning parameters on those images. In tailor making a solution for a single application, the images and controls can be displayed at fixed locations on an output display. However, a more flexible arrangement would be to provide a windowing environment that also manages all end-user interactions (keyboard and mouse). Currently, the developer must also construct all the necessary debugging and tuning tools for each image processing design. Thus, we propose a windowing operating system to facilitate the tuning and debugging of image processing (IP) algorithms directly on an FPGA.

An FPGA does not have any operating system to speak of. All peripheral board drivers must either be developed by the user or be provided in libraries by board vendors. There is no task scheduling (as compared to scheduling software processes in other operating systems) or memory management except that which occurs at compile time. Access to restricted hardware resources can be controlled with hardware semaphores and such like.

Several attempts have been made to construct low-level hardware operating systems such as (Tomko & Tiwari, 2000) and (Wigley & Kearney, 2001), but these are targeted towards reconfigurable computing and deal primarily with the run-time scheduling of hardware processes rather than direct user interaction.

At present, no such generic tool exists that caters for the stand-alone hosted model. The Gate Array Terminal Operating System (GateOS) described in this thesis is the first step towards addressing this need.

**CHAPTER Two** describes what is required of the GateOS windowing environment. We give an overview of the FPGA hardware environment that will host GateOS and the image processing algorithms. Also discussed is the need for a form of window management and display system, as well as a widget management and display system. A text management system is also necessary to display textual annotations. An overview is given on how an end-user might expect to be able to use GateOS to tune and debug resident image processing algorithms.

**CHAPTER Three** describes the window management subsystem of GateOS. We discuss its purpose and the context within which it is used. The need for a virtual coordinate system is discussed and how it is used to position and display windows on a visual display unit (VDU). We also describe a layered approach to effectively manage and display each window's content.

**CHAPTER Four** discusses the need for a widget management subsystem in GateOS. We describe a number of different widgets and what they can be used for. Also, we demonstrate the benefits of using a layered approach for the management and display of widgets. We also discuss how widgets can be used to control various aspects of GateOS in addition to facilitating the tuning and debugging of image processing algorithms.

**CHAPTER Five** describes the text management subsystem and the context in which it is used. We discuss several alternative designs that implement this functionality on the FPGA. Also discussed are the interactions between this subsystem and the rest of GateOS.

**CHAPTER Six** describes how an end-user can use either a keyboard or a mouse to interact with both windows and widgets in GateOS. We reveal the design challenges involved with displaying both keyboard and mouse cursors. Also discussed is how GateOS is able to manage each device type and the relationships between them.

**CHAPTER Seven** is a reflection on the work completed on GateOS. We outline the benefits of using GateOS to tune and debug image processing algorithms on FPGAs. We describe the limitations of this approach and discuss where future research on GateOS could be advantageous.



# CHAPTER TWO

## REQUIREMENTS ANALYSIS & HIGH LEVEL DESIGN

### Chapter Outline

This chapter describes the various requirements of GateOS. We discuss why a windowing approach is appropriate for the tuning and debugging of image processing algorithms. We give an overview of the hardware environment within which GateOS is to be developed and discuss how the image processing algorithms can be managed and kept separate from the rest of GateOS. We also cover the requirements for the windowing, widget and control subsystems in GateOS.

## 2.1 Introduction

For some time now, the hosted development model has allowed end-users to debug and control sections of time-critical FPGA hardware from a host PC. The developer is able to choose which aspects of an algorithm are implemented on an FPGA and which are implemented on the PC. This type of flexibility is suitable for applications that require less real-time interactive debugging and where the relatively high latency of test signals between the FPGA and the host PC is not vital.

In order to debug image processing algorithms, it would be beneficial if the developer was able to view the outputs of the algorithm on the VDU in real-

time. Since the output images are updated continuously (otherwise there is no point in implementing the IP algorithm on the FPGA), the hosted development model is somewhat less attractive (the bandwidth between the host PC and the FPGA is a bottleneck). The alternative is to operate the FPGA in standalone mode with all the debug and tuning tools located on the FPGA along with the IP algorithms. The standalone development model is also more portable than the hosted development model as the host PC is not required (one can configure the FPGA from flash or other local non-volatile memory).

Very little research has been conducted on tuning and debugging image processing algorithms using the standalone development model. Currently, it is necessary for a developer to construct the necessary tools to facilitate debugging on the FPGA for each IP algorithm developed. Such tools may use onboard LEDs, multi-line character displays, or a VDU to display debugging information. End-users can tune the IP algorithms via onboard DIP switches, push-button switches, a keyboard or a mouse (if a VDU is used). The construction of each debugging and tuning facility requires a certain amount of development time and hardware resources on the FPGA.

## **2.2 Proposal**

Constructing a hardware operating system could resolve many of the aforementioned problems by providing resources for an end-user to interactively debug and tune IP algorithms when using the standalone development model. A developer should be able to focus on the development of IP algorithms and not on the underlying tuning and debugging framework. An end-user should be able to visualise the results of IP algorithms in real-time on a VDU.

Essentially, the proposed GateOS is a restricted form of windowing operating system. It would be responsible for the provision and management of GUI tools

that facilitate the real-time debugging and configuration of algorithms on the FPGA. The requirements of these GUI tools are discussed in subsequent sections of this chapter. The restrictions on the scope and functionality of GateOS are required to cope with the limited resources available on a FPGA. Where possible, it is desired that GateOS be independent of a particular FPGA or FPGA family, and should also be independent of the resources available on the particular development board used (apart from the provision of video output and mouse input).

A successful development of GateOS, may be gauged by the provision of source codes that compose GateOS. These source-codes should be able to be compiled without alteration so as to provide a live demonstration of GateOS. This would constitute a proof of concept. Time-willing, user evaluation of GateOS would also be useful, although, this isn't essential to fulfil the goals of this thesis.

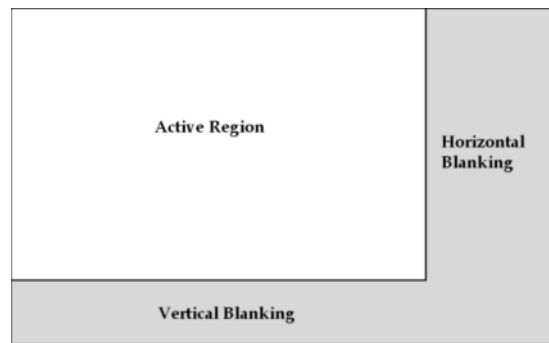
## 2.3 Hardware Environment

Use of the Handel-C programming language has become more popular in recent years for the development and implementation of image processing algorithms on FPGAs. As its name implies, Handel-C has a similar syntax to the C language but with various extensions that cater specifically for the construction of hardware<sup>4</sup>. When compared with lower level languages such as VHDL or VERILOG, Handel-C is the preferred option for developing GateOS, due to its ability to incorporate both low-level and medium-level constructs. This simplifies the development process by allowing most of GateOS to be developed at a relatively high level, while still allowing lower level design where necessary. The Handel-C language is not in itself an object-oriented language, but the object oriented paradigm can still be useful in the design of the various systems and data-structures within GateOS.

---

<sup>4</sup> <http://www.celoxica.com/products/dk/default.asp>

Since much of GateOS is concerned with the GUI, it is very much dependent on the underlying technique that is used to generate and display content on a VDU. In a standard VGA video output frame there are three main regions of interest in terms of timing: the active region, the horizontal blanking region, and the vertical blanking region. The blanking regions (see Figure 2-1) are necessary for CRT displays to rescan the internal electron beam to the start or the top of the display respectively. Modern TFT displays still support these blanking regions; however, this is merely for backward compatibility reasons. The video driver only needs to output display pixels in the active region, as pixels generated in either blanking region are ignored.



**Figure 2-1 Timing Regions within a video frame, active, H-blanking and V-blanking**

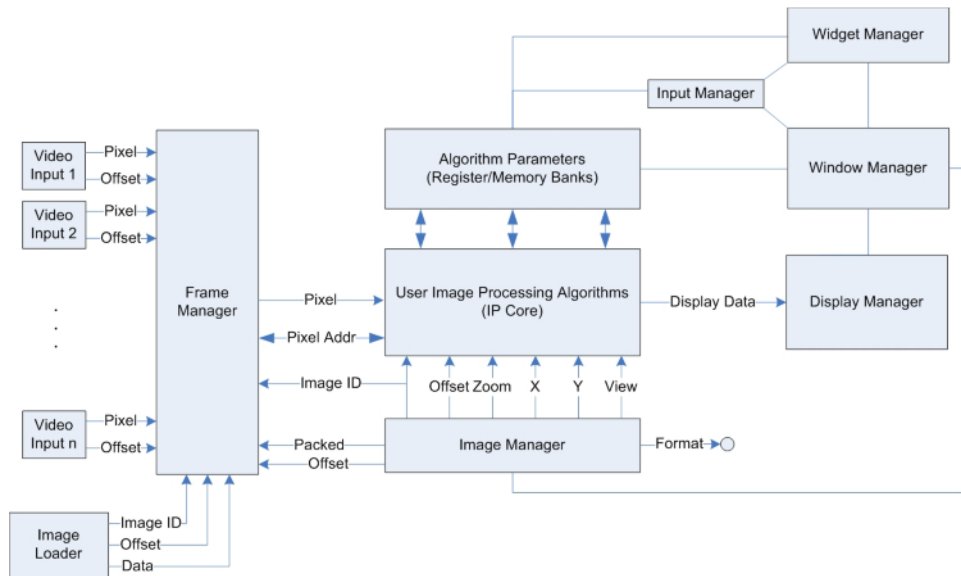
GateOS must therefore generate output pixels in synchronization with the video clock. There are two main techniques for providing this steady stream of pixels: a frame-buffer approach and generating pixels on-the-fly. The frame-buffer approach stores the pixels to be displayed in memory (the frame buffer), which is then read sequentially as each pixel is required for the display. It requires a large block of memory dedicated to the frame-buffer. This effectively decouples output generation from the display process. The problem with this approach is the limited bandwidth of the frame-buffer. Unless it is dual-ported, only a single read or write can be performed on the frame buffer memory per clock cycle; which becomes a problem when it is necessary to display live video feeds.

The alternative is to use the on-the-fly approach to provide the stream of pixels for display. The pixel situated at each display coordinate is generated on-demand when it needs to be displayed. No frame-buffer memory is required for this approach and the bandwidth problems experienced when using live video feeds are for the most part resolved.

For GateOS, we have elected to adopt an on-the-fly approach. This choice does however come at the cost of increased complexity of the window manager in GateOS. Since timing is critical, major design challenges (pipelining display pixels as a part of video scheduling) are introduced with the on-the-fly approach that is not present when using a frame-buffer. These issues form a significant part of the discussions in this thesis.

## 2.4 System Overview

GateOS should be able to interface with the IP algorithms as well as providing an interactive user interface. From an engineering perspective, decoupling the IP algorithms from GateOS will almost always result in a simpler design. GateOS should be independent of a particular IP algorithm - the algorithm should be developed within the context of an OS, not the other way around. A modular design allows only the features necessary in a particular application to be incorporated. Keeping this in mind, the architecture of GateOS might look similar to that shown in Figure 2-2. All the user algorithms must be decoupled from GateOS and thus are located within an IP core container and communicate with GateOS through a well-defined set of interfaces.



**Figure 2-2 GateOS IP Core and its relationship to the rest of GateOS**

Figure 2-2 is an attempt to identify the various sub-systems within GateOS and give some indication as to their relationship with one another. The IP Core (and its constituent algorithms) shown in Figure 2-2, is managed entirely by the various subsystems in GateOS, clearly qualifying it as an Operating System, albeit a cut-down one. The inclusion of a non-volatile storage manager, discussed briefly in Section 2.5, further reinforces this principle.

One of the major considerations in the design of the OS will be managing the timing constraints imposed by the run-time environment of the FPGA. Since we are using on-the-fly pixel generation, it will almost certainly be necessary to pipeline areas of GateOS to ensure that pixels at particular display coordinates are generated on time. While it is true that modern FPGAs have more resources and higher clock speeds than was available previously, a desirable feature of GateOS will be to support operation on small and low cost FPGAs if possible.

GateOS should be presented to a developer as a set of libraries. Developers can then interface with these libraries when it becomes necessary to tune or debug their IP algorithms. The behaviour of GateOS is, therefore, customized by the developer to fit the particular application by constructing the required windows, widgets and so forth. In this way, only the hardware needed for a

particular application is built. For instance, if a mouse button is never used in a project, then the hardware for it will not be included in the final configuration.

Since GateOS will use a windowing environment, it will require some form of window manager. Each window in GateOS is just a visual container, which can be used to display content such as images, histograms and widgets. The window manager is responsible for maintaining any window data-structures, as well as scheduling the display content of each window at the appropriate location on the VDU. A key reason for having a windowing based OS is that the user can hide, restore, reposition and resize individual windows on the VDU. This can typically be done with the use of particular mouse gestures. In addition, the user could also wish to clone or destroy particular windows.

Widgets form an important part of any windowing environment and because of this, GateOS should also incorporate a widget management subsystem. Widgets are useful for displaying and manipulating Boolean (buttons) and integer (sliders) variables, as well as displaying textual annotations (labels). Widgets are only ever displayed within the content area of a window, so the widget manager requires constant communication with the window manager. Support for the following widget types should be incorporated into GateOS: labels (textual annotations), text edit areas, sliders (horizontal and vertical) and buttons (both momentary action and toggle buttons). The widget manager is responsible for managing all the widget data-structures and displaying each widget within of its associated window. A special class of widgets is required for scheduling the display of image/video data and statistical information (such as histograms) within windows.

In terms of user interaction, we need to be able to tune and debug IP algorithms. This tuning may involve the manipulation of buttons, sliders and text edit widgets that are linked to the IP algorithm. Using a keyboard and a mouse to do this manipulation is appropriate in this situation and is familiar to

users. Using a mouse and keyboard necessitates the building low-level drivers on the FPGA to communicate with each device. Also, as a part of the display, there are keyboard and mouse cursors that may be visible. An additional input management system interprets the inputs and input gestures and passes the corresponding information to the appropriate widget or window. As the widget and window parameters are adjusted, this affects a window's appearance or position on the display, as well as control variables that may be used by the IP algorithm.

An essential component of any interactive operating system (windowing or command-line based) is the management and display of textual annotations. A textual annotation is a string of characters displayed somewhere on the VDU. A text manager is necessary to manage and display strings of characters on widgets and windows in GateOS. The text manager may also be required to annotate areas of interest on an image at run-time. The text manager is thus tightly integrated with the window and widget managers as well as the IP core, since it can be called upon by any of these to display an annotation.

## **2.5 Non-Volatile Storage Manager**

A developer may need to store data in non-volatile RAM (i.e. flash RAM). The most obvious approach for fixed size data is to use preset addresses for each data item. For variable sized data, a more flexible approach would be better, such as modelling the nv-RAM as a flat file system (no directory structure). There are several different types of file system that could be used (such as FAT, EXTFS, UDF...). Incorporating such a file system into GateOS would allow the developer to specify a filename or id as the unique handle for an item of data and then read or write data to this file via the storage manager. One would need to consider if the extra hardware needed to implement such a manager is worthwhile.



# CHAPTER THREE

## WINDOW MANAGEMENT

### Chapter Outline

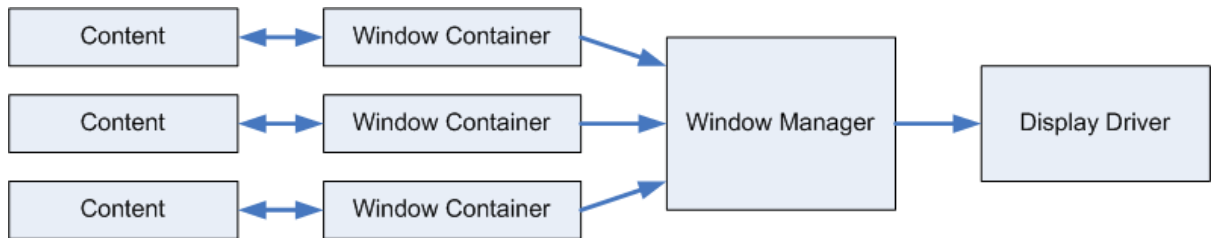
This chapter discusses how windows are represented and implemented in GateOS. We discuss how a window is represented on a screen in terms of dimensions, logical position and status. Mechanisms for managing and displaying overlapping windows are described using 'on-the-fly' pixel generation.

### 3.1 Introduction

From the user's point of view, a window is a container for content displayed on the screen. The content may be images, user-interface controls or debugging information, and it is usually more convenient to group related components within a single window. Usually, there is more information available than can conveniently (or aesthetically) fit onto a single display. A common approach to handling this problem is to separate the content into windows and to allow the user to control which windows are visible and which are not. For convenience, the user can rearrange the window positions and where they overlap, and can determine which window is visible.

From a designer's perspective, we define a window to be a rectangular container displayed on a VDU that can be moved, resized, overlapped, created, and destroyed. A user should be able to distinguish a window from all others in

the windowing environment. A developer should be able to specify the content that is displayed within the window, either at compile-time or in the case of dynamic data, at run-time through the image processing core.



**Figure 3-1 Windowing System architectural overview**

A window manager determines which window (if any) is visible at a particular screen location, and routes the window's content to the display driver as shown in Figure 3-1.

## 3.2 Representing a window's size and location

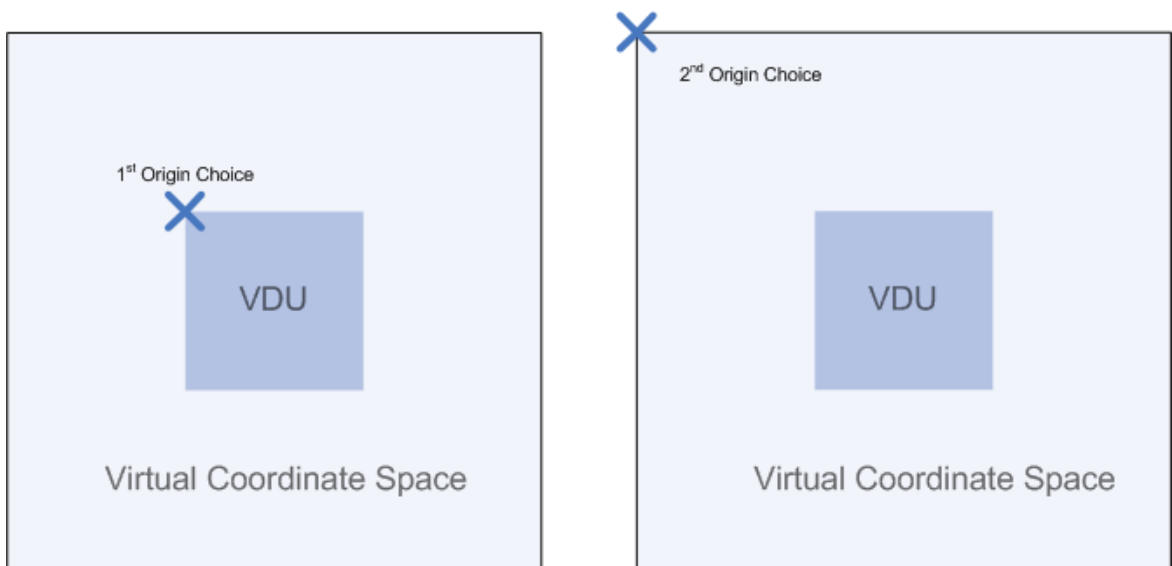
It is common knowledge that a window's position and size on screen can be characterized with four properties; x position, y position, width and height. However, there are design questions that may be raised that must be resolved. The first issue concerns the location of the origin and the second is concerned with how to represent a window's extent on screen.

It is assumed that the system has only a single VDU. This simplifies considerably the logic and scheduling compared to managing multiple windows across several VDUs. A constraint imposed on windows is that their size (width and height) must be less than that of the VDU. This ensures that the content of each window can always be displayed in its entirety on the VDU. Also, the implementation of the window manager in Handel-C can be

simplified somewhat if a designer can assume that these constraints are enforced. There are several ways in which the four window parameters can be represented and interpreted in GateOS, which leads to the use of a virtual coordinate space.

### 3.2.1 Virtual Coordinate Space

If windows were restricted in their movements, such that each window must reside fully within the bounds of the VDU, some end-users would find this both frustrating and counter-productive. To allow window's to extend past the edge of the display, a virtual coordinate space can be defined.



**Figure 3-2 Choices of origin within virtual coordinate space**

The constraints described earlier mean that the virtual coordinate space only needs to be triple that of the display resolution of the output VDU, with the VDU occupying the central region. The origin in the virtual coordinate space can be located at either the top-left corner of the VDU or the top-left corner of the virtual coordinate space, as shown in Figure 3-2. The first choice would entail the use of signed variables to represent a windows position on the VDU.

Since the output video driver defines the origin of the VDU to be the same as that used in the virtual coordinate space, mapping a window's position relative to the VDU requires no conversion (except signed to unsigned).

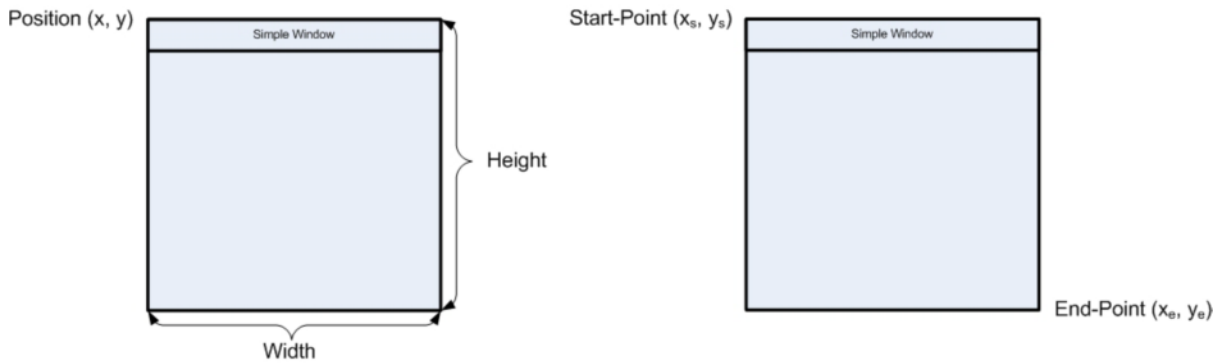
The second choice would use only unsigned variables. However, a disadvantage of this approach is that display coordinates from the video driver must be translated to obtain the virtual coordinates that the window manager can use in provide the correct output pixel.

### **3.2.2 Window extent**

With regards to representing the extent of a window within the virtual coordinate space, the position and size parameters of a window can be encoded in several different ways. Two potential encoding schemes have been identified that could be used in GateOS, as shown in Figure 3-3. The first scheme uses size and position coding, while the second scheme uses point coding.

The coordinate space for this coding schema uses unsigned numbers to represent its width and height dimensions. To find a window's end-point (required in the implementation for calculating particular offsets), one has to perform two separate additions ( $x + \text{width}$ ,  $y + \text{height}$ ). If this end-point value is frequently used by GateOS on multiple occasions from different locations in the window manager design, additional hardware may be necessary to multiplex the different hardware requests for the endpoint value.

The alternative is to represent a window's extent on the VDU is to use a point based coding schema. Rather than representing a window by its width and height, it explicitly encodes the start and end points of the window.



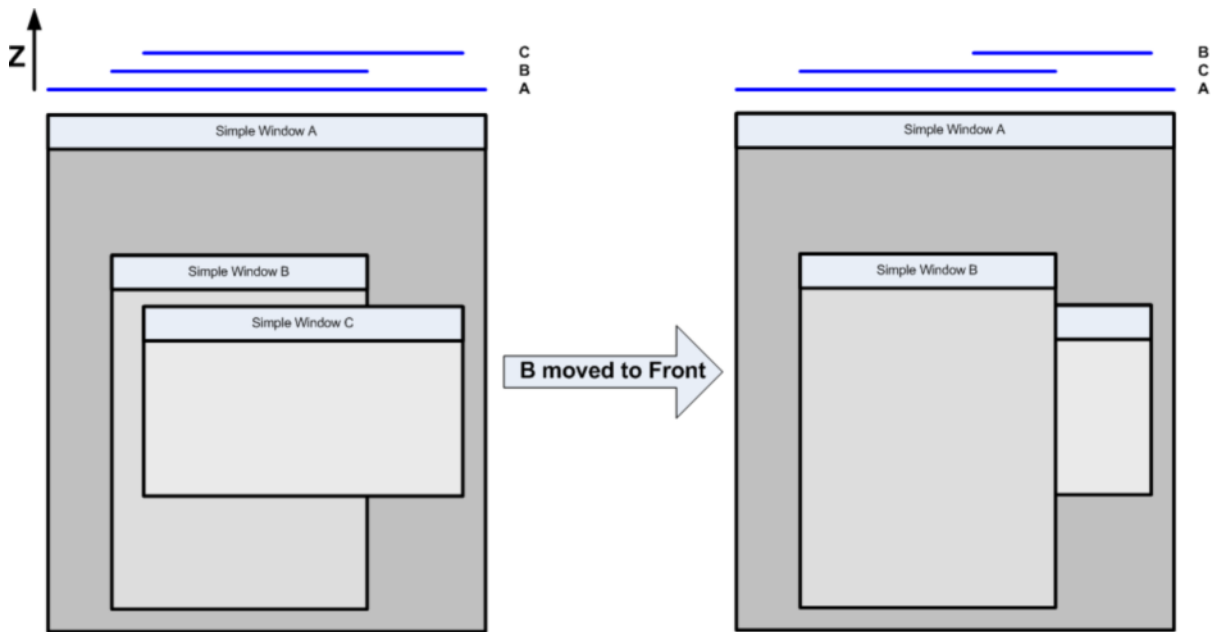
**Figure 3-3 Virtual coordinate space based on the two encoding schemes**

The end-point  $(x_e, y_e)$  defines the location of the bottom-right corner of the window. The width of the window can be obtained at run-time with a subtraction  $(x_e - x_s)$  as can its height  $(y_e - y_s)$ .

This approach could be beneficial in situations where the start or end points of a window are required more often by GateOS than the width or height. This is the primary motivation for the use of this schema in the current implementation of GateOS.

### 3.3 Overlap of windows

Overlapping occurs when a window is positioned such that its bounding rectangle intersects the bounding rectangle of one or more other windows. In the intersection, the window manager must be able to identify which window is visible on screen and which window(s) are obscured or occluded. One can think of the window's as being stacked one behind the other in layers. A window's position in this stack determines whether it is in front of, or behind another window. This depth adds a third dimension,  $z$  with a windows  $z$ -index effectively identifying the priority with which it is displayed as shown in Figure 3-4.



**Figure 3-4** Changing the z-index of a window

A left mouse click within a window can be used to bring that window to the front. This is achieved by setting the z-index of that window to the z value of the current front window and decrementing the z-indices of the windows in-between to move them back a layer. A generic sorting algorithm can be used to perform this task. To enable the window manager to identify the id of a window in a single clock cycle - given its z-index - it is necessary to maintain an inverse z-table; which is a list of windows sorted by z-index.

### 3.4 Window States

A window in GateOS can be in one of four possible states; null, hidden, inactive or active. The null state is used if one or more of the window's core properties are incomplete. This will be the case if a window has been defined, but is not currently in use by GateOS. A null window should be concealed from view.

The active window is situated in front of all other windows and can receive both keyboard and mouse input. Inactive windows may be occluded and can receive no input from a keyboard. The only mouse input accepted by an

inactive window is the gesture associated with making the window active. As the name implies, the hidden state is used for a window whose properties are valid, yet the window and its contents are fully hidden from view. The status of a window may be toggled between the hidden and visible states by clicking an associated button on the window activation bar. The window activation bar is a special widget window that provides a set of buttons reflecting the visibility of each window in GateOS. The window activation bar must always remain in the visible state, thus no mechanism is provided to change it (there is no button reflecting its visibility). The buttons within the window activation bar can be aligned vertically or horizontally. A window is created in the visible state when its properties have been initialized. In the visible state, a window and its contents are viewable on the VDU (unless it is occluded). The window with the highest z-index is considered to be the active window; otherwise it is inactive.

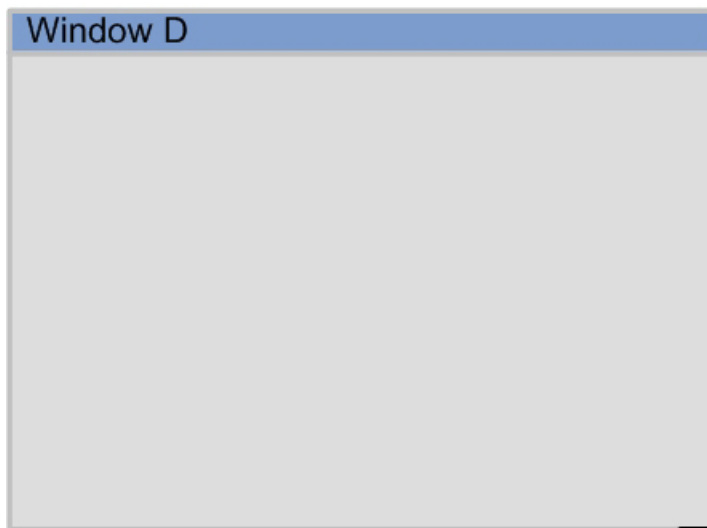
## 3.5 Window Regions

Each window consists of three overlapping layers: a background layer, a content layer and a border layer.

The background layer is split vertically into two separate regions; the title-bar and the content area background. The title-bar stretches the full width across the top of the window and has a height of 16 pixels. The value of 16 pixels is chosen because it is aesthetically pleasing (to me), and due to the fact that offset calculations are somewhat simplified (if the four LSBs of the row relative to the top of the window are dropped, the remaining value is zero within the title-bar, otherwise it's within the content area). The text management system, as detailed in CHAPTER Five, is used to add a textual label (as a part of the background layer) for the window. The label is aligned to the left side of the title-bar.

The colour of the title-bar is used to indicate whether or not a window is active; with pale-blue used for the active window and dark blue used for inactive windows. The background colour of the content area can be customized for each window.

The second (content) layer is where the window's content resides. Widgets that control a window's behaviour (see CHAPTER Four) are right-aligned within the title-bar. It is within this content layer of the main part of the window that image and video data are displayed, along with any widgets (see CHAPTER Four) used to tune algorithm parameters and display debugging information. The content layer of a window has its own local virtual coordinate space with the default origin located in the top left corner of the content area. If the window content (e.g. a large image) is larger than the available display area of the window, then the window's contents can be scrolled by offsetting the local origin of the window's virtual coordinate space; either horizontally or vertically.



**Figure 3-5 Final layer with a grip for resizing windows**

The third layer is used to display a window's internal and external bounding borders. The external borders are two pixels deep and are situated on all four



sides of a window, while the internal border vertically separates the title-bar from the content area with a horizontal line on the bottom two rows of the title bar. These borders are present on all windows in GateOS without exception. If a window can be resized, then a resize grip is also displayed in the bottom-right corner of the external window border, as shown in Figure 3-5.

## 3.6 Managing and displaying windows

In CHAPTER Two, we specified that on-the-fly pixel generation should be used in GateOS. This requires the window manager to be able to identify at any given clock cycle, the foremost visible window at the output display coordinate  $(px, py)$  as provided by the video driver. This task is simplified somewhat by the fact the pixels are required in a scan order; so adjacent pixels are likely to come from the same window. Two potential techniques are to simultaneously test all windows in parallel with a priority encoder to select the front-most window, or to determine the locations of the transitions from one window to the next along each raster.

### 3.6.1 Parallel Incidence Test

The start and end points of each window are compared with the VDU display coordinates (provided by the video driver) to identify which windows are present for the current output pixel. From these, the window with the highest  $z$  value is selected for display.

For a screen coordinate  $(x_p, y_p)$  to be within a window, it must satisfy three conditions. The pixel must be between the left and right sides of a window ( $x_s \leq x_p < x_e$ ). It must also be between the top and bottom of the window

$(y_s \leq y_p < y_e)$ . The window must also be visible (status = active or inactive). Refer to Figure 3-6.

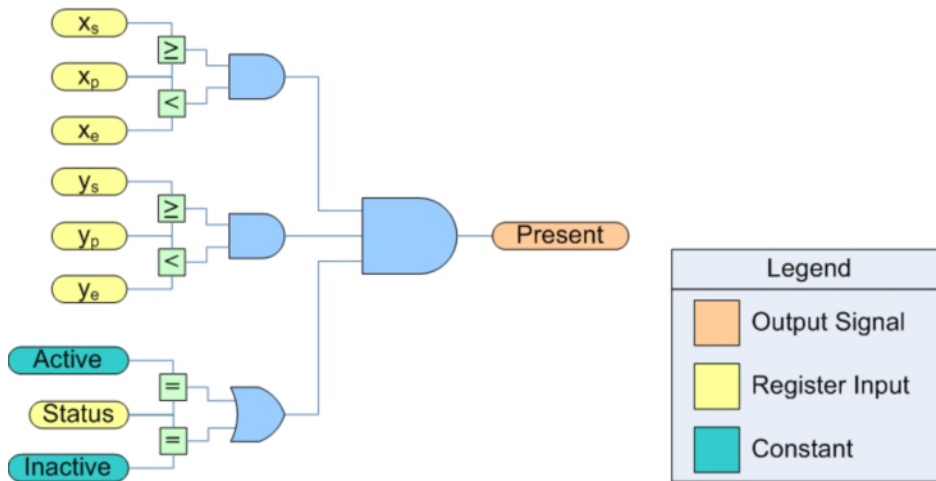


Figure 3-6 Detecting whether a window is present at a screen coordinate

To determine this for every window in a single clock cycle requires separate hardware for each window, and that the window parameters to be stored in registers (rather than memory) because they must all be accessible every clock cycle. The primary weakness of this approach is that it requires a significant amount of hardware for each window in GateOS.

We can optimize this design by eliminating some of this hardware. This is possible because the pixels must be produced as a series of left to right scan-lines. The horizontal blanking also provides an opportunity for the window manager to perform a limited amount of processing before each line is displayed. The visibility status of each window is active or inactive. Since the row number is constant for a scan-line, the condition  $(y_s \leq y_{p+1} < y_e)$  only needs to be evaluated once per line, and this can be performed during the horizontal blanking period. Similarly the visibility status of a window can be evaluated once per line. The combined result of these evaluations can be stored in a single binary register per window (see valid in Figure 3-7). Since the timing is

less critical during the blanking period, these evaluations may be performed sequentially. Concurrent access to the status or the  $y_s$  and  $y_e$  components are no longer required, thus allowing these three properties to be stored in a RAM and be indexed by window id. During horizontal blanking, a serial process can then iterate through the windows and evaluate the validity condition.

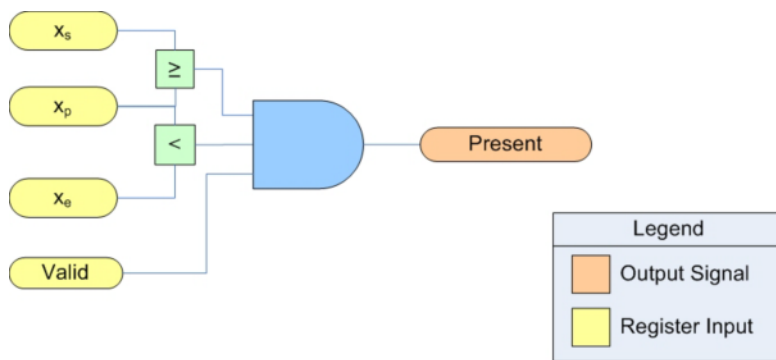


Figure 3-7 Updated window identification that uses less hardware per window

The real-time hardware, therefore, only needs to check the horizontal extent of each window ( $x_s \leq x_p < x_e$ ) and ensure that the window is valid on the current scan-line as shown in Figure 3-7. This can be further optimised by replacing the comparison tests with equality tests (which use less hardware), this is because pixels are scanned sequentially (i.e. set a flip-flop when  $x_s=x_p$ , and clear it when  $x_s=x_e$ ).

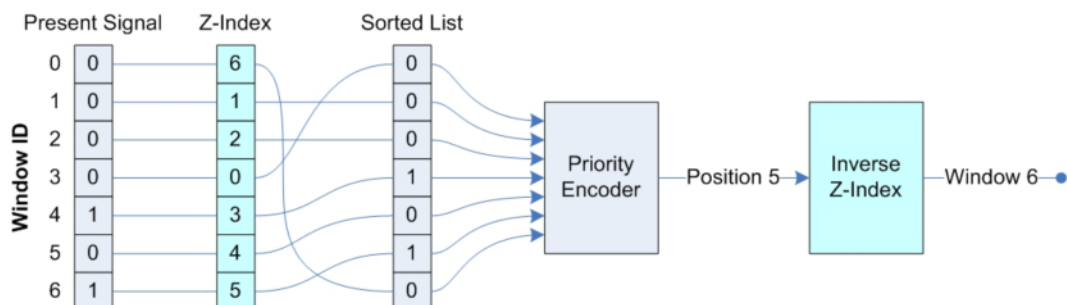


Figure 3-8 Selecting the window with the highest z-index

At a given screen coordinate there may be several windows present, but only the window with the highest z-index is selected for display. The present signal of each window is routed to an input of the priority encoder based on the z-index of the window, as shown in Figure 3-8. The priority encoder then identifies the index of the highest set bit, which is looked up in an inverse z-index table. The resulting value is the id of the window to be displayed at the particular screen coordinate.

This input sorting based on the z-index is, in fact, a crossbar multiplexor, which is very expensive (in terms of hardware resources) to implement. The RMO macro (returns the offset of the rightmost set bit in an expression), which is provided by the Handel-C standard library (see the Handel-C PDK Reference Manual<sup>5</sup>), can be used to implement the priority encoder.

## 3.6.2 Window Transition Method

Previously, we have discussed how a display frame is output to the VDU as progressive scan-lines and how a horizontal-blanking period is situated at the end of each scan-line. In section 3.6.1, we were able to exploit this horizontal blanking period to make the real-time window selection process independent of the subsequent scan-line. This can be similarly exploited by a memory-based technique.

This technique uses the horizontal blanking period for each scan-line to construct a look-ahead table that lists windows in the order (left to right) that they are to be displayed on the current scan-line. When the VDU is in the active region, a selector iterates through this list of windows, displaying each in turn until the last entry is reached. The look-ahead table must then be rewritten for each scan-line as the order of the windows can change.

---

<sup>5</sup> [http://www.celoxica.com/support/view\\_article.asp?ArticleID=578](http://www.celoxica.com/support/view_article.asp?ArticleID=578)

This approach makes it possible to store all the core properties that determine the window's position, dimensions and z-order to be stored in fabric-RAM. This is preferred because fabric-RAM requires fewer hardware resources than register banks to store the same amount of data. However, the use of fabric-RAM restricts the number of concurrent accesses to one (or two if the RAM is dual-port). The fabric-RAM bandwidth limitations mean that only a single window's properties can be accessed in a particular clock cycle, thus necessitating the search for an alternate approach, such as the use of look-ahead tables. These tables are populated in advance and are used to assist in the real time selection process. Whilst there may be other solutions to this problem, the requirements of this thesis are such, that only a minimal amount of FPGA resources should be used whenever possible. Using look-ahead tables satisfies these requirements.

The horizontal blanking period (approx 20 percent of total horizontal scan time) of each scan-line can be used to perform a limited amount of processing, whereas the longer vertical blanking period (several scan-lines) can be used for more time-intensive processing. The existence of these two timing intervals can also be exploited to populate the look-ahead tables.

### **3.6.2.1 Window List Look-ahead table (WLL)**

The WLL lists the windows in the order that they are displayed on a particular scan-line. Each entry is a window id (integer) that the window manger can use as the index for accessing those core window properties that are stored in fabric-RAM. The table partitions the active portion of a scan-line into discrete line segments, such that each segment represents an unbroken stream of pixels belonging to an individual window. Each window entry must be displayed on the current scan-line for at least one clock-cycle, as shown in Figure 3-9.

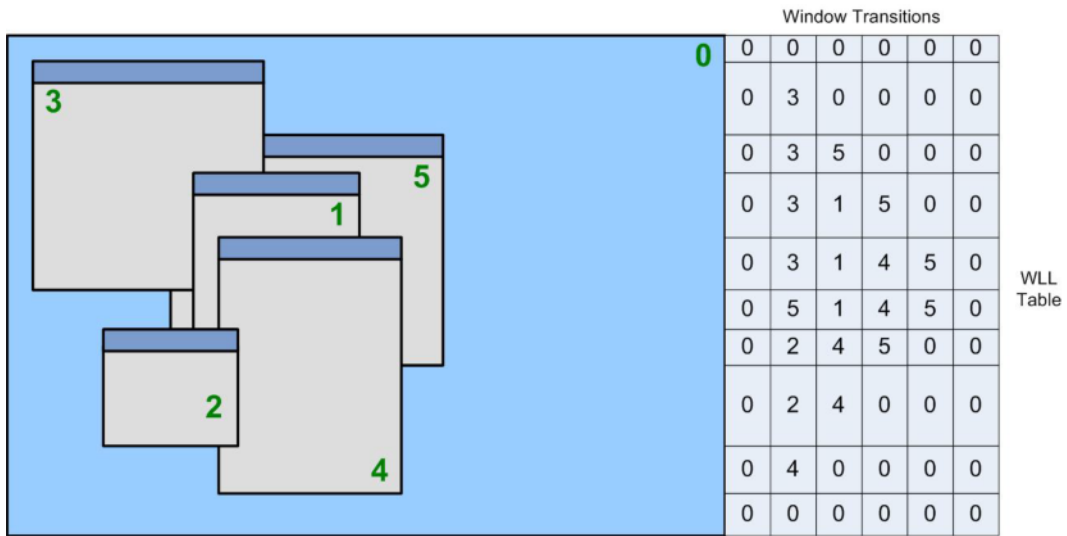


Figure 3-9 How WLL entries for each scan-line are related to window positions

The size of the WLL is determined by the maximum number of window transitions that can occur on a scan-line, as shown in Figure 3-10.

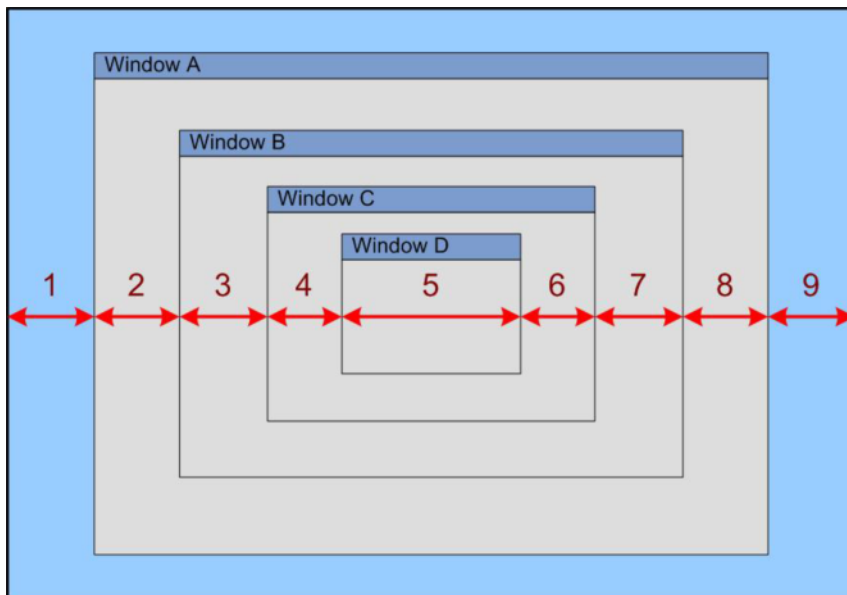


Figure 3-10 Worst case scenario with each windows vertical edges displayed

Each window has two vertical edges. If both edges of  $N$  windows are visible in a scan-line, then there are a maximum of  $2N$  transitions (or  $2N+1$  counting the

special background window). The WLL can be stored as an array in fabric-RAM.

### 3.6.2.2 The real-time window selection process

For each entry in WLL, the window selector must establish how many clock cycles the current window is displayed for, before transitioning to the next window. If the current window has the higher z-index (it is in front of the next window), the transition occurs when the current window's right edge is encountered ( $x_{e,current} = x_p$ ). Otherwise, the next window is in front and is displayed when its left edge is encountered ( $x_{s,next} = x_p$ ). In a worst-case scenario, the above technique should be able to switch the display window with every clock-cycle, requiring the properties of the current window to be cached.

To implement this process, relatively few hardware resources are required. Registers can be used to store the id of the current window, its z-index, the transition coordinate (either  $x_{e,current}$  or  $x_{s,next}$ ) and the WLL iterator. Additional information can be retrieved on demand from fabric-RAM using the window's id as an index.

### 3.6.2.3 Building the window-list look ahead table (WLL)

The WLL table is potentially different for every scan-line, so must be rebuilt for every line that is displayed. Since WLL is stored in RAM, the only time it can be accessed to construct the table is during the horizontal-blanking. For a 640x480 screen resolution, the horizontal-blanking period lasts approximately 120 clock cycles. The maximum number of windows that GateOS can manage is

directly dependent on how fast we can generate entries for the WLL. Since we can only write to WLL once every clock cycle and each window has 2 edges; GateOS can manage at most 60 windows at VGA resolution (more than enough).

It is clear that serial search techniques cannot be used to create the WLL. To assist in this process, an additional table, the edge-list lookup (ELL) table is required. This lists the vertical window edges of all visible windows, sorted in order from left to right. For each entry in this list, we store the windows id and the polarity of referenced edge (left or right) as shown in Figure 3-11. If the edges of two or more windows coincide, the windows are ordered by z-index with the highest z-index placed first. This data-structure can also be placed in fabric-RAM.

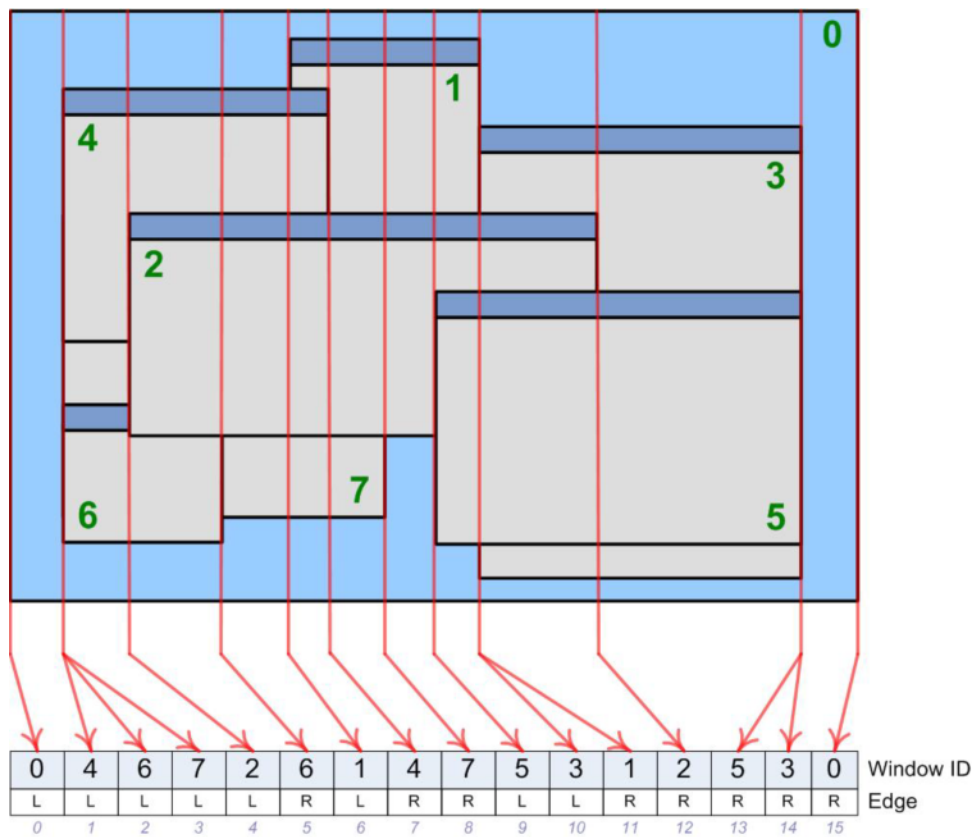


Figure 3-11 Edge list lookup-table



The first and last entries in ELL always refer to the special desktop window as it covers the entire virtual coordinate space and is always visible on the VDU, albeit with the lowest z-order.

When the right edge of a window is reached, a priority register ( $R_s$ ) is used by this algorithm to select the next topmost window. Each bit in  $R_s$  is associated with a z-index. A priority encoder (the RMO macro in Handel-C) and inverse-z lookup is used to identify the topmost window.

The algorithm starts out clearing  $R_s$  to zero. The window id from the first entry in ELL (the desktop window) is copied to the WLL and is set as the 'current' window. The window id of the next entry in ELL is read, and denoted as the 'next' window (only if the scanline is within the window). Depending on the z-indexes of both 'current' and 'next' windows and the polarity of the 'next' windows edge (left or right), one of three actions is performed.

- 1) If the next window is a left edge, the  $z_{\text{next}}^{\text{th}}$  bit in  $R_s$  is set (the start of a window), otherwise it is cleared (the end of the window).
- 2) If the left edge has a higher z-index than the current window, it replaces the current window and is added to the WLL.
- 3) If the right edge of the current window is encountered, the highest priority window in  $R_s$  is found, and it replaces the current window and is added to the WLL.

This process is repeated for each entry in the ELL data-structure. A moderate amount of FPGA hardware resources are required to implement this algorithm. Since several conditions need to be evaluated in a single clock cycle and actions are performed based on these results, the combinatorial delay could be quite long, adversely influence the maximum clock speed of the FPGA design. The current window id, its z-index,  $R_s$ , the WLL and the ELL iterators need to be stored as registers.

### 3.6.2.4 Building the Edge-list look ahead table (ELL)

The contents of this data-structure will not change unless an end-user repositions, resizes, hides, clones or destroys a window. Furthermore, since keyboard and mouse input is processed serially during vertical blanking; only a single window in ELL will be affected at any given time. This fact simplifies any required updates to ELL into two separate tasks: insertion and removal. The removal process completely removes from ELL any references to a particular window and the insertion process subsequently reinserts the two edges. In the current design of GateOS, the status of a window is considered to be null if it is not represented in the ELL data-structure (eliminates the need to have a separate data structure to indicate if a window has been constructed or not). An independent variable is used to maintain a running total of all visible windows in GateOS.

<i>Clks</i>	<i>Source</i>
1	<i>// prepare the ELL iterator register</i> <i>i = 1;</i>
1	<i>// we need to remove references to the left and right window</i> <i>// edges from the ELL (two references)</i> <i>step = 0;</i>
1	<i>// iterate forwards through ELL until these two references have</i> <i>// been removed</i> <i>do {</i> <i>// Extract the current window index</i> <i>window index = ELL[i];</i>
1	<i>// compare the ids of the two windows</i> <i>if(window_index\1 == window_id)</i> <i>step++;</i> <i>else</i> <i>ELL[i - (0@step)] = window index</i>
1	<i>// compare the ELL iterator register</i> <i>i++;</i> <i>}while(i != (WINDOW COUNT VISIBLE &lt;&lt; 1));</i>
1	<i>// there is one less valid window in GateOS</i> <i>WINDOW COUNT VISIBLE--; //running window total</i>

**Listing 3-1 Removal routine**

As shown in Listing 3-1, this Handle-C routine performs a forward scan through ELL, removing each invalid window reference as it progresses. After the first removal, successive entries are shifted down by one place to occupy the vacated space. Similarly, after the second removal, successive entries are shifted

down by two places. Upon completion, the variable holding the current visible window count is decremented.

Clks	Source
1	<i>// we only need to do this twice</i> <i>step = 2;</i>
1	<i>// get the z-index of the window we need to insert</i> <i>compare_z = Window Z[window id];</i>
1	<i>// get the right edge value of the window we need to insert</i> <i>compare_x = Window XE[window id];</i>
1	<i>// prepare the iterator</i> <i>i = (WINDOW COUNT ACTIVE &lt;&lt; 1) - 1;</i>
1	<i>// Perform a reverse search through the ELL table, and insert the</i> <i>// proper entries representing both left and right edges of the</i> <i>// window</i> <i>do {</i> <i>    //by default we decrement the ELL iterator register</i> <i>    perform_dec = 1;</i>
1	<i>// Extract the indexed window id</i> <i>window_index = ELL[i];</i>
1	<i>// extract the indexed windows z-index</i> <i>current_z = Window Z[window_index\1];</i>
1	<i>// extract the indexed windows right or left edge (the 0th bit</i> <i>// of the windows index determines the edge)</i> <i>current_x = (window_index[0]) ? Window_XE[window_index\1]</i> <i>:Window_XS[window_index\1];</i>
3 or 1	<i>//Should an edge (left or right) of the window we want to insert</i> <i>//be stored at this location in the ELL</i> <i>if(compare_x &gt; current_x    (compare_x == current_x &amp;&amp;</i> <i>    compare_z &gt; current_z))</i> <i>{</i> <i>    //prepare the new window index (append the 1-bit flag</i> <i>    //that represents the edge</i> <i>    window_index = window_id@step[1];</i> <i>    //we only need this one</i> <i>    compare_x = Window_XS[window_id];</i> <i>    //don't decrement the iterator this time</i> <i>    perform_dec = 0;</i> <i>    }else delay; //otherwise just wait</i>
1	<i>//insert the new or old entry</i> <i>ELL[i + (0@step)] = window_index;</i>
1	<i>//perform the decrement operation</i> <i>if(perform_dec)</i> <i>    i--;</i> <i>else</i> <i>    step--;</i>
1	<i>}while(step != 0);</i> <i>// an extra window has been inserted, therefore increment the</i> <i>// total number of valid windows in the system</i> <i>WINDOW COUNT ACTIVE++;</i>

**Listing 3-2 Insertion routine**

The insertion routine as shown in Listing 3-2 is slightly more complicated than the previous removal procedure, as we need to insert the two new window

references in into ELL. To do this, we scan backwards through ELL. Until the first insert location is found, the entries of ELL are shifted up by two places (to make room for the new window entries). On encountering the location where the right edge of the window belongs, the window id is inserted and flagged as a right edge. From this point, entries are shifted up one place until we reach the location of the left window edge. The current visible window count is subsequently incremented.

### 3.7 Discussion

For the implementation of GateOS, the designer is able to choose which coding scheme is used to represent a windows position and dimensions. In using a virtual coordinate space, we have simplified the process of mapping a window's coordinates to those used when generating pixels on the VDU.

The register technique used to schedule window's for display is functional but not very scalable. The hardware resources required to represent each additional window in GateOS is directly related to the number of bits needed to represent each window's ID. This means there is virtually no resource penalty (except for RAM storage space) incurred with changing the number of windows from 5 to 8; however, a small increment in resources is required to increase the number of windows from 8 to 9 and so on. The crossbar switch required for priority encoding is the main reason for these increases. The alternative technique, which assumes that core windowing properties are stored in fabric-RAM, attempts to resolve many of these issues with a considerable amount of success. For this memory-based technique, the hardware resources required for each additional window are relatively fewer than those required for the register-based technique.

In the future, it would be useful to construct techniques that would make it possible to clone particular types of windows (Image, Video) at runtime as well

as destroy them (revert to a null state). This could be used to clone an image window associated with a particular IP algorithm. A widget window could then tune an IP property associated with this cloned window and compare the resulting image with the original. The implementation of this could be the focus of much future research.

# CHAPTER FOUR

## WIDGET MANAGEMENT

### Chapter Outline

This chapter describes the properties of the various widgets used within GateOS. An in-depth discussion on the data-structures required for retaining each widget's properties is also provided. It covers the three widget display layers (window boarder, content and background) and the required processing attributed to each of them, in order that widgets can be managed and displayed within a host window on the VDU.

### 4.1 Introduction

The term 'widget,' as used in GateOS, describes a single graphical control located within a window. In this chapter, we define several different types of widgets - the most important of which have been implemented in the current version of GateOS while others remain as potential designs.

The types of the widgets detailed within this chapter include; label, button, edit box, slider, histogram display, image display and video display. The last three may be grouped into a separate category called the image processing (IP) widgets, as they rely directly on the IP core for their display content.

In terms of GateOS, widgets are useful for graphically tuning and debugging image processing algorithms in addition to controlling the behaviour of

GateOS. More specifically, interactive widgets are able to manipulate Boolean or integer variables while the image processing widgets display useful debugging information in a graphical form.

## 4.2 Widget Display Layers

Three sub-layers (within the window's content layer) are used to display widgets. These sub-layers (see Figure 4-1) are used for the display of image content, algorithm control and window control.

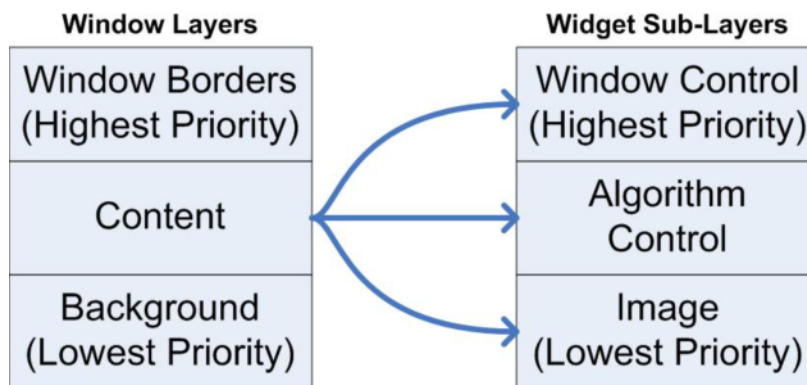


Figure 4-1 Content window layer split into three widget sub-layers

The first 'image' layer occupies the complete window's content area. Since the content region uses its own coordinate space, this 'image' layer may be scrolled both horizontally and vertically so that all of its content is accessible. Only a single IP widget (histogram, image or video) may be displayed in any window. If there is no IP widget associated with the host window, then this image layer is transparent and the underlying window background layer is displayed.

The second 'algorithm control' layer is also positioned within the window content area. This layer shares its vertical and horizontal offsets with the image layer, so scrolling the window will move these controls with the image. Several widgets can be positioned within a window, with the underlying image layer

visible where there are no widgets. Widgets within this layer are used to tune algorithms in the IP core and display debugging information.

The third 'window control' layer contains the widgets responsible for manipulating the various properties and behaviours of a window and its layers. The location of widgets in this layer is fixed relative to the window at compile-time as these conceptually form part of the window rather than the contents. Buttons on the title-bar are all 16 pixels in size (requires less hardware resources) as are the sliders (except for the length which is variable and depends on the window size). Again, this layer is transparent; if no widget is present at a particular point, then the underlying layers are displayed in order of priority (window control layer, then image layer).

## 4.3 Widget Details

### 4.3.1 Label Widget

The purpose of a label widget (see Figure 4-2) is to provide a textual annotation within a window. An example would be to identify areas of interest within an image.

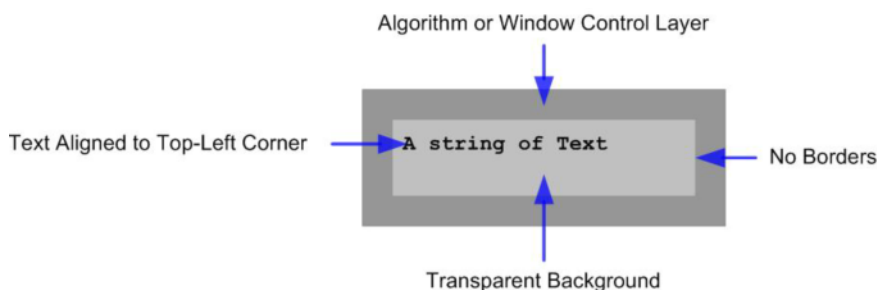


Figure 4-2 A simple Label Widget



A label widget is stateless and has no mouse or keyboard interaction. Each label widget has a transparent background. The only purpose of a label widget is to display a textual annotation; therefore, a single string is displayed within each label. This string is aligned with the top-left corner of the widget and any text that extends past the bottom and right boundaries of the widget is truncated. As detailed in CHAPTER Five, the actual text string is stored in a string table and referenced by its id.

### **4.3.2 Button Widget**

If a button widget is situated within the algorithm control layer, then it can be used to manipulate a Boolean parameter of an algorithm. Alternatively, if this widget is used within the window control layer, then it may be used for a variety of window management tasks. Such tasks may include hiding/restoring a window from view, creating clone windows and destroying them, or pausing/resuming a live video feed.

A button widget has a border displayed on each of its four sides (top, bottom, left and right), so that it is clearly visible. This border is one pixel deep and is dark-grey in colour (common to all buttons in GateOS).

A button widget has two valid states, ON and OFF, as shown in Figure 4-3. The state reflects the state of a Boolean parameter within the system, and the button can be used to manipulate that state. Two visual aids are used to denote the current state of a button and its purpose; a text string and the background colour. Each button widget has associated two strings, one for each state of the button. A user should be able to establish the purpose of each button, if not its state, simply by reading its label string. The string is aligned to the top-left corner of the widget and has a black foreground colour.

The background colour of the button also reflects the button state, with the ON and OFF colours defined globally.

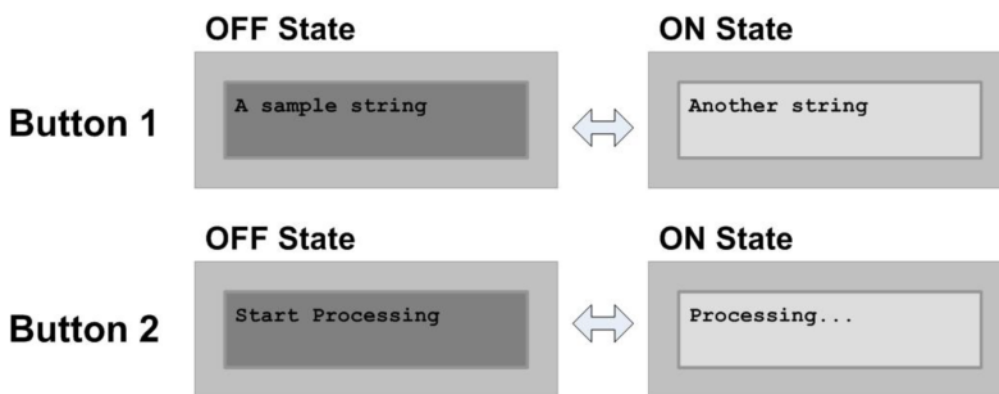


Figure 4-3 The background of a Button Widget can be one of two possible colors

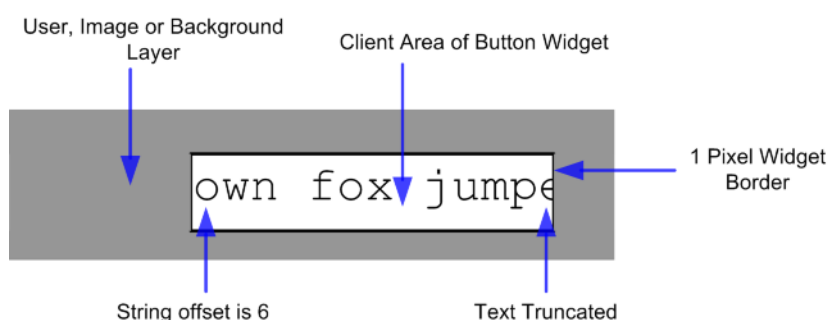
A keyboard or mouse can be used to interact with a button widget. Each button can be assigned a character, such that when this character is input from the keyboard, GateOS can set, reset or toggle the button's state. If the key is held down, no further state changes occur until that key is released (see CHAPTER Six for further details on managing keyboard input).

When using a mouse to interact with a button widget, only left or right mouse button clicks are accepted. A left click on the button widget will toggle the button's state. Alternatively, a click by the right mouse button will toggle the button's state only temporarily (until it is read and reset by the IP core), thus exhibiting a push-button like behaviour.

### 4.3.3 Text Edit Widget

The purpose of a text edit widget is to allow the user to enter a string. What the programmer does with the string is independent of the widget.

A text string needs to be displayed within an edit widget. Like the label and button widgets, this string is aligned to the top-left corner of the widget. Characters in the string that extend past the bottom or right boundaries will not be displayed. The widget must also be able to provide access to all the characters within its text string. This is enabled by scrolling the string on a character-by-character basis, with an integer variable to store the offset of the first character displayed. The colour of the text in each text edit widget is black.



**Figure 4-4 An Edit Widget displaying the string “The brown fox jumped”**

A text edit widget has a 1 pixel dark-grey border on all four sides to clearly delineate its bounding rectangle, as shown in Figure 4-4. The background colour of the text edit widget is white to distinguish it from other widget types. Only a single text edit widget can be accepting input (active) at a time, so one is also able to distinguish between active (where the text cursor is) and inactive text edit widgets (no text cursor).

A simple graphical cursor is used to indicate the current insertion point. The cursor is a black vertical line that is 1 pixel wide and 16 pixels high, and is always situated between two characters in the text string. Either a left or right click on an edit widget is used to select the widget and set the initial cursor position. Once attached to an edit widget, the text cursor can be manipulated using the arrow keys on the keyboard (see Table 4-1). If a keyboard cursor is

moved to either the left or right borders of the widget, then the text string is scrolled by one character to keep the cursor in view.

Key	Cursor Action
Right Arrow	Move to next character
Left Arrow	Move to previous character
End	Move to last character
Home	Move to first character

**Table 4-1** Cursor movements

The 'Backspace' key removes the character directly to the left of the cursor. Using any other keyboard key (excepting the backspace and the four cursor control keys) will result in the insertion of the associated character into the text string at the location of the cursor. Unprintable characters are replaced by spaces.

To extend the usefulness we have added the automatic conversion of a numeric text string into an integer. A 2-bit flag as shown in Table 4-2, is required to designate whether the input is interpreted (takes place after each character is entered) as binary, decimal, hexadecimal or plain text. Only characters in the particular number based are allowed.

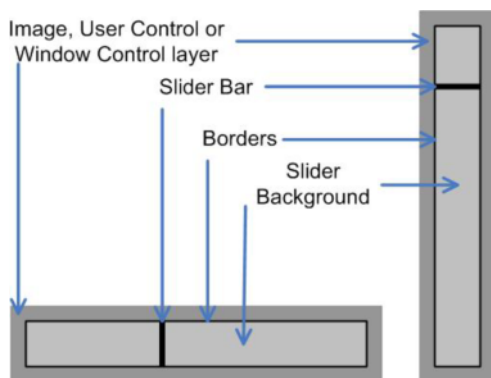
Flag	Operation
0	plain text
1	binary
2	decimal
3	hexadecimal

**Table 4-2 Available numeric string conversions**

### 4.3.4 Slider Widget

A slider widget is similar in function to that of a scroll bar commonly used by other operating systems. However, its primary purpose (as opposed to a scroll bar) is to allow a user to manipulate one or more integer variables using a variety of mouse gestures. The definition of a slider widget has been further modified to fit the design goals of this thesis, namely the use of a minimal amount of resources.

Each slider widget has a one pixel dark-grey border on each of the four sides to delineate its bounding rectangle, as shown in Figure 4-5. In some applications it may be useful to use the background of the slider to convey meaning about the slider position (threshold for example). A flag is used to indicate whether a standard or user-generated background is used.

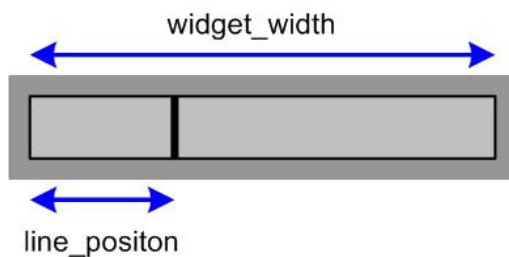


**Figure 4-5 Vertical and horizontal slider widgets**

The slider indicates and controls an integer variable that can have a value between zero and *max\_val* inclusive. The minimum value is thus constant and never negative. The position of the slider bar within the slider is therefore:

$$line\_position = \frac{value \times (widget\_length)}{max\_val}$$

The slider bar is one pixel thick and has a black colour.



**Figure 4-6** The offset of a horizontal Slider Bar from the left border

Note that there may not be an exact representation of every possible value if *max\_val* is larger than the widget length. A slider widget may also be configured to manipulate two separate variables by adding a second slide bar with its own *value* and *max\_val* variables.

The use of a single line to represent a slider's value (see Figure 4-6) instead of the rectangular block favoured by other operating systems has benefits as well as some drawbacks. The principal benefit is that the position of the slide bar is easier to manage and display. The calculations required to determine the left and right boundaries of a rectangular slider are avoided. Conversely, using a rectangular block for the slide bar can convey additional information, and the concept of applying step sizes to slider movements may have more meaning when using a rectangular block.

A mouse gesture is used to reposition individual slider bars within a slider widget; we have considered two alternative methods of doing this. Both are

unconventional but functional and are designed to use a minimum amount of resources.

The first method associates the left and right mouse buttons with the first and second sliders respectively. When the mouse button is pressed within the slider, the corresponding slider bar is repositioned directly under the mouse cursor. The value associated with the slider widget is then updated as:

$$value = \frac{line\_position \times max\_val}{widget\_length}$$

The slider bar will continue to move with the mouse cursor while the button is pressed. If both left and right mouse buttons are depressed concurrently, then both slider bars will follow the mouse cursor.

With the second method the mouse click does not immediately move the cursor, but requires the user to drag the mouse past the slider bar (with the button down) to select it (in a left-right or right-left sweep). After selection, that slider bar will move with the mouse cursor as before. While the slider bar is tracking the cursor, it must remain within the boundaries of the slider widget. When the mouse button is released, the slider will stay in its last location.

Either of the two methods can be selected at compile time, although it must be noted that the first is only able to manipulate up to two bars within the slider. The slider bars are aligned along the widget and their movement constraints (whether they can be dragged through each other) can be configured at compile time. All three widget types, namely the button, text and slider widgets have been implemented, but are yet to be subjected to user evaluation, due to time constraints.

## 4.3.5 Imaging Widgets

Each window in GateOS may have a single histogram, image or video widget displayed within its content layer. An IP widget always starts at (0,0) in the virtual window coordinates. Its length and width properties are determined by the IP core, but are also cached by GateOS.

Since the dimensions of an IP widget may be larger than the content area of a window, a scrolling mechanism is provided. A vertical and horizontal slider is added to the window control layer whenever the widget is too large for the content area.

GateOS should allow a user to zoom the display area of IP widget both IN and OUT. A rudimentary zoom capability would be to allow zoom sizes that are only powers of 2 (...  $\frac{1}{4}$ ,  $\frac{1}{2}$ , 1, 2, 4, 8, 16 ...). In this case, determining the correct display data would be trivial. Two square button widgets situated on a window's title-bar could provide the necessary zoom interfaces as shown in Figure 4-7. A mouse click on either button would adjust the zoom factor to the next step in the corresponding direction.

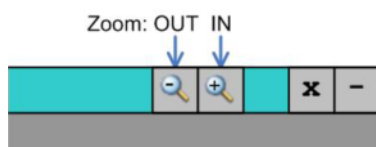
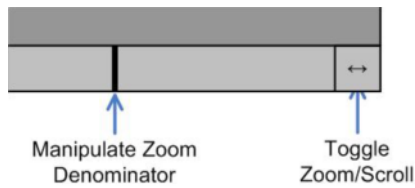


Figure 4-7 Two zoom buttons, one in and the other out

The biggest limitation with power of 2 zooming is the constraint on the zoom factor. A better approach would be to allow a user to specify the zoom factor as a fractional value. The denominator of this fraction would be fixed by the IP core with the numerator adjusted by the user. This approach would give good zoom in but limited zoom out.





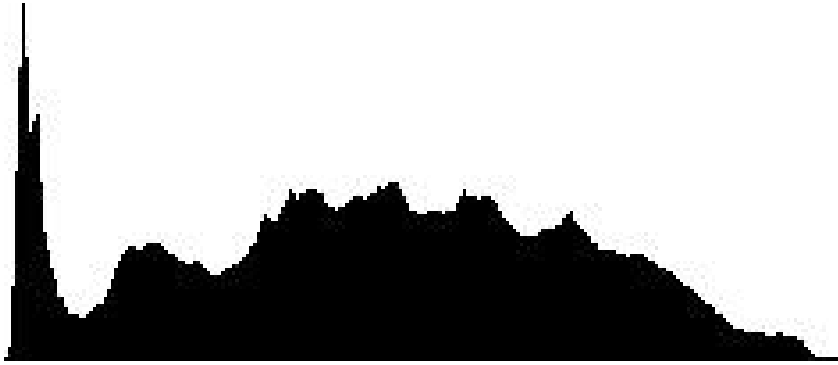
**Figure 4-8 Slider widget is used to manipulate numerator of zoom fraction**

The numerator could be manipulated with the help of a slider widget. One can reuse the horizontal slider used for horizontal scrolling to also manipulate the zoom factor, as shown in Figure 4-8. A button widget, located next to this slider in the global layer, can select which parameter the slider would manipulate. The alternative might be to have two slider bars on the same scroll widget; one for horizontal panning and the other for controlling the zoom. Ideally, one would want the natural zoom size (1:1) to be in the centre of the scroll bar.

Unfortunately, due to time limitations, I was unable to investigate the three IP widgets in any great detail. Therefore, only a brief description of the widgets and their possible implementation are provided in the following sections.

### **4.3.5.1 Histogram Widget**

The purpose of this widget is to display a histogram or other similar graphical data, as shown in Figure 4-9. Usually the dataset represents some statistic derived from an image, and consists of an array of integers. Since display pixels are generated 'on-the-fly' from this data, only a single element of the data-set needs to be accessed per clock.



**Figure 4-9 Sample Histogram Content**

A user does not interact directly with the histogram, other than through zooming and scrolling via the appropriate mouse gestures.

### **4.3.5.2 Image Widget**

The purpose of an image widget is to display the images after various stages of an image processing algorithm. The image widget refers to the source image by an integer id. The IP core is responsible for indicating the correct display pixel format, which is forwarded by GateOS to the display.

The latencies inherent within the IP algorithm may require GateOS to start fetching pixels for particular images columns or even rows in advance. Dealing with these latencies can be quite complex, especially when windows overlap. The display of some image widgets may be starved of pixel data (unless some form of buffering is used). This resolution for this issue is discussed further in section 4.5.1.

An IP algorithm may on occasion wish to annotate an image with text, labelling objects or regions of interest. To do this, a number of label widgets can be reserved for that particular window. The IP core would be responsible for generating the text strings and the locations within the window at which the widgets should be positioned.

### 4.3.5.3 Video Widget

In terms of real-time applications, there is no difference between an image and a video widget. In most algorithms, the image will be changing dynamically; otherwise there is little need for the FPGA. From a user's perspective, a video widget is quite similar to an image widget, except that the displayed frame content on the video widget is subject to change at the underlying rate. This rate is specified by the developer at compile-time.

A user should be able to freeze the video widget, such that its source stream of images is frozen and the most recent frame or image is continuously displayed. This is implemented with a 'pause-play' button. Additionally, a user may wish to review individual frames of the input video stream. This would require capturing a limited number of frames when the video stream is frozen, thus requiring significant memory resources to buffer the frames. A user may then step forward or back through these frames. These frames would be discarded when the user indicates that the video widget should resume displaying the live video feed. Again two buttons can provide the necessary interface for a user to step back or forward respectively.

GateOS does not directly manage the image or video frames used by the algorithms within the IP core. The primary reason for this is that it is very hard for GateOS to cope, in a generic fashion, with the demands of each IP algorithm with respect to image data being input/output. This is best illustrated with an example IP core that requires six separate video streams as input to various IP algorithms. Depending on the quantity and types of memory available on the FPGA board, the required frame management system may need to perform bank interleaving, skip frames and other such tasks. Therefore, it is up to the developer of the IP core to construct a suitable frame management system for

their application. However, we do require that the developer conform to all interfaces between the IP Core and GateOS (defined in section 4.4).

## 4.4 Data-Structures and Interactions

Several data-structures are required to store the properties and configuration information of all the widgets in GateOS, as shown in Figure 4-11. Properties that are common between widgets are stored within the same data structures. The  $x$ ,  $y$ , width and height properties are common to all widgets. There are several ways in which these four properties can be stored:

- 4 arrays, one for each of  $x$ ,  $y$ , width, height
- 1 array but with separate entries for  $x$ ,  $y$ , width and height
- 1 array of structures containing  $x$ ,  $y$ , width and height.

The first approach is the preferred option as it is more flexible with regards to accessing properties of different windows concurrently. In order to conserve hardware resources, fabric-RAM instead of registers should be used to implement the data-structures.

The  $x$  and  $y$  widget properties will never change (except for label widgets used to annotate images). With the exception of the histogram, image and video widgets, the height and width properties of all other widgets remain constant. The height and width properties of the three IP widgets may on rare occasions need to be refreshed as the IP core may change the dimensions of their content at run-time.

An interactive widget (a button, text edit or slider) should be able to directly configure the parameters of an IP algorithm managed by the IP core. A suitable interface is required to link the values of button, text edit and slider widgets to the corresponding algorithm parameters.

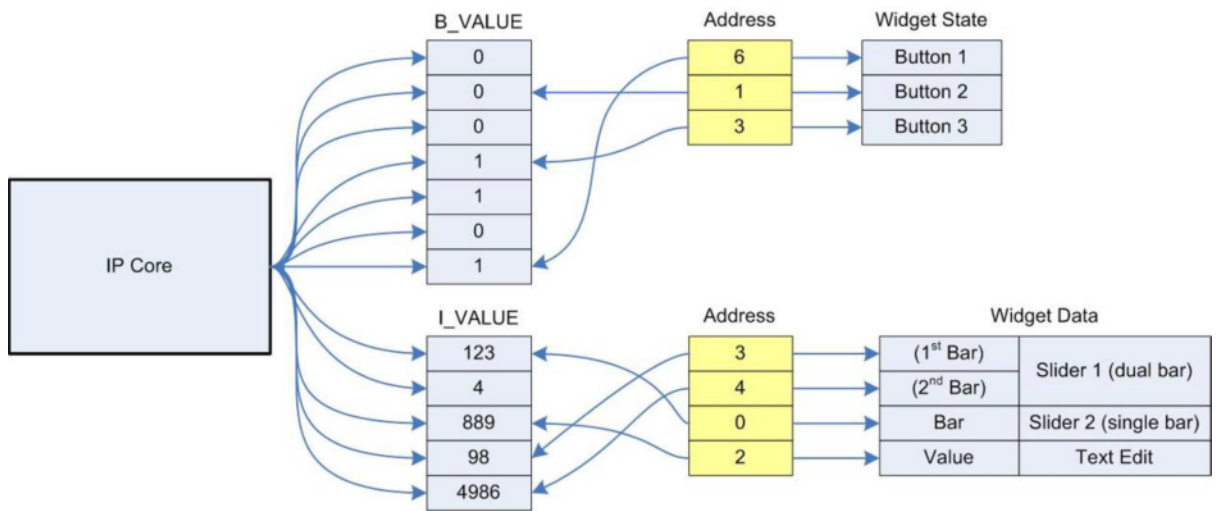


Figure 4-10 B\_VALUE and I\_VALUE data structures

A possible approach would be to use a set of value registers as the interface between GateOS and the IP core. To simplify the management, these would be separated into two arrays - one of Boolean values (B\_VALUE), and one of integer values (I\_VALUE) as illustrated in Figure 4-10. Algorithms in the IP core are able to read from and occasionally write (push buttons) to the B\_VALUE and I\_VALUE arrays at run-time. The corresponding widget stores the index to the parameter, and accesses the value indirectly.

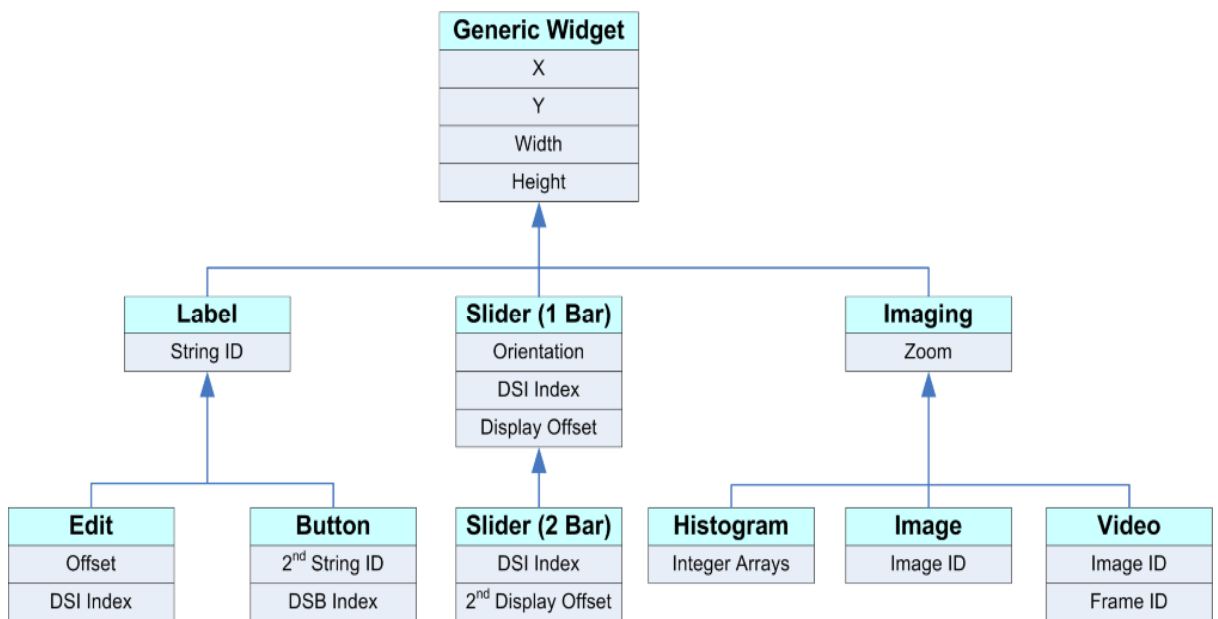
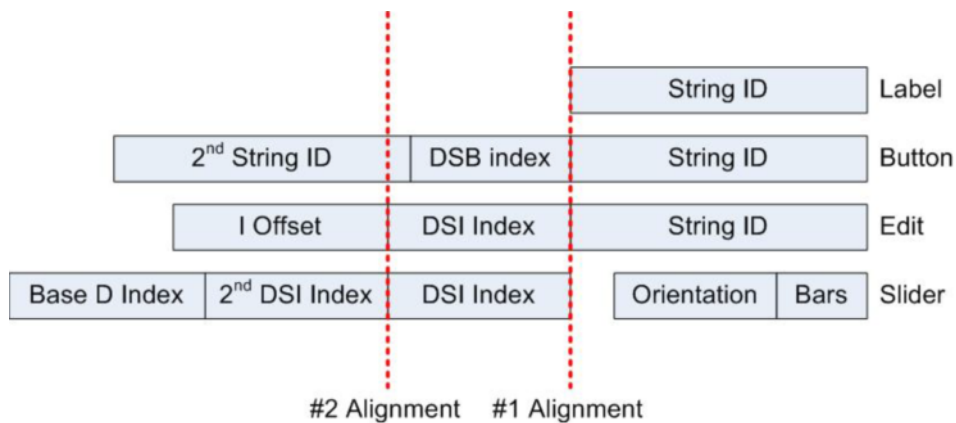


Figure 4-11 The data structures necessary for each widget type



**Figure 4-12 Aligning common properties between widget's**

The remaining properties for each widget can be concatenated into a single variable. To reduce the decoding logic required, the string id property and the value properties are aligned within each widget variable (see Figure 4-12). The complete variable that retains each widgets properties are stored as an array in fabric-RAM, using the widget's unique id as an index.

### 4.4.1 Imaging widgets

Further data structures may be required by the IP core to provide the source data used by the histogram widget and to buffer image and video frames.

The histogram widget needs to have access to an array of integers in order to display its content. The IP core may need to randomly read/write to each array of integers as the IP algorithm is executed. Since only a single read access per clock is required from these arrays, they should be stored as FabricRAM or even BlockRAM instead of registers, thus saving a significant amount of hardware resources.

## 4.5 Scheduling and Display of Widgets

### 4.5.1 Image Layer

Only a single IP widget may be displayed on the 'image' layer. We can construct a table that associates the image source with the window.

Some IP algorithms may impose both horizontal and vertical latencies between the input and output streams of pixels. Thus, for a scan-line the content of an IP widget may need to be scheduled well in advance of the display of its associated window. A large vertical latency may require the content to be scheduled even though the window is not visible on a scan-line. The complications arising from this mean that one cannot guarantee that the content of each IP widget will be fully displayed, unless extensive buffering is used.

Since a window has a 2-pixel border and a 16-pixel title-bar, it would be relatively easy to support IP algorithms with less than 2 clocks horizontal and 16 lines vertical latency respectively. In such environments, the IP window content would be scheduled in step with the regular window scheduling faculties. Longer horizontal latencies are more likely; however, this would require modifying the existing window handler to provide trigger signals to the IP core with the required latency.

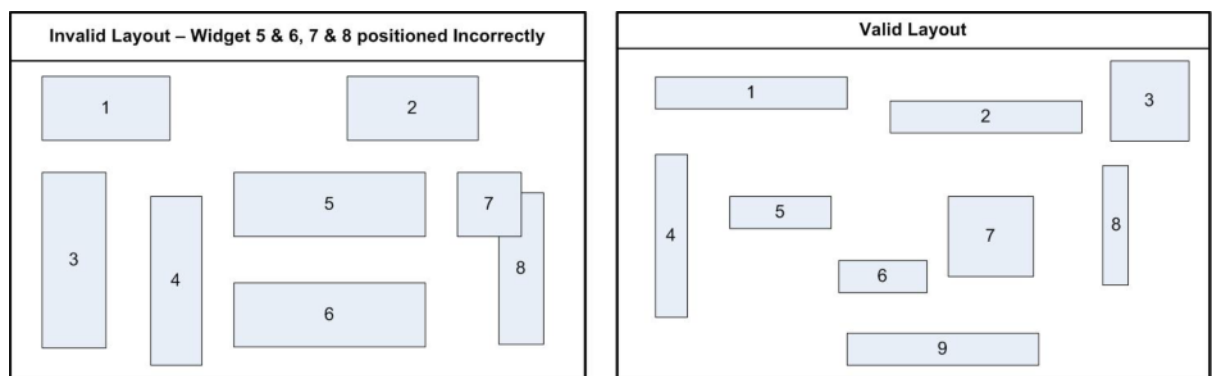
### 4.5.2 Algorithm Control Layer

For this layer, we need to be able to schedule the display of widgets that are located within each window for a particular scan-line. Since several window's may contain multiple widgets and overlap of windows is allowed, this scheduling is somewhat more complicated than that required for the display of

windows. This is further complicated by the fact that some widgets may be partially or fully occluded from view due to being positioned elsewhere in the windows virtual coordinate space (which can be vertically and horizontally scrolled).

The scheduler also needs to be able to transition from displaying a particular widget to another within a single clock-cycle (this can occur when a widget's left edge is adjacent to the right edge of another widget within the same window). Without assistance of expensive look-ahead tables, this clock cycle is insufficient for a scheduler to ascertain the next widget to display. A possible solution is to restrict the positioning of widgets such that look-ahead tables are unnecessary. Imposing these restrictions (see below) would then allow the widget scheduler to iterate through a list of widgets that are organised in the order in which they are displayed on the window.

The first restriction is that a widget cannot overlap with another widget located within the same layer of any window. This is easy to arrange, except perhaps with labels used to annotate an image.



**Figure 4-13 Valid and invalid widget layouts**

The other restriction is that widgets must be grouped in consecutive horizontal strips that are subject to the following conditions:

- The horizontal strips are not overlapping.



- Widgets must fit completely within a strip.
- Horizontally, within a strip, only a single widget may occupy any given x coordinate.

This is very restrictive, especially where label widgets are used to annotate images. However, this approach is the only one that currently works, hence its usage in the current implementation of GateOS.

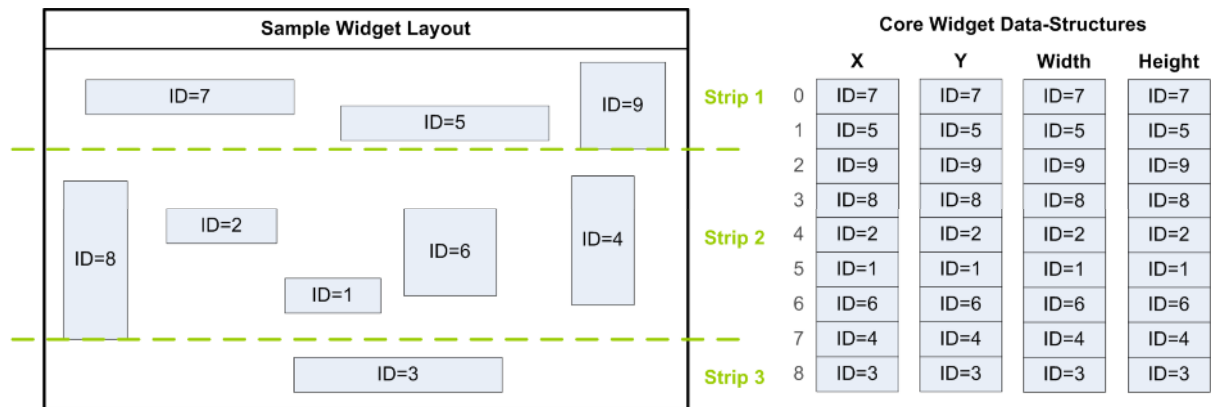


Figure 4-14 Order of widget data structures is determined by row then column

Each entry in the four core widget data-structures (x, y, width and height) should be ordered first by horizontal strip then by the x coordinate (see Figure 4-14). While the algorithm control layer is being displayed, the widget manager scans through the four core widget data structures and compares each entry with the current display coordinates. If a widget entry has already been displayed, then the widget manager proceeds to the next entry. This process continues until the last widget in the horizontal strip is encountered.

During h-blanking, we check whether all widgets in the current horizontal strip have been displayed (current display row > bottom edge of bottommost widget in horizontal strip). If so, the real-time part of the algorithm resumes with the next widget entry (in the ordered data-structures), which is the first entry of the next horizontal strip. This requires that the index of the current widget of each window be stored in a data-structure. The same is also true for the location of the first entry in the current row of each window. Conversely, if one or more

widgets in the current horizontal strip are yet to be fully displayed, the widget manger proceeds to the first entry in the current row. This process continues until the last widget has been displayed on the algorithm control layer of the current window.

The hardware resources required to implement this approach are considerable, especially when compared to the system used for window management (refer to the preliminary results in CHAPTER Seven). The current implementation of GateOS uses a system of bitmasks to assist in the real-time identification of the last widget in a horizontal strip (using the RMO Handel-C macro) as well as the bottommost. In the absence of a suitable alternative, this approach seems to work well even though it is expensive (due to the large multiplexors required).

### **4.5.3 Window Control Layer**

The position and size of each window control widget relative to the window is predefined. The display routine need only check which region of the window is being displayed to identify the correct control widget to display. Since a control widget may be visible on some windows and not others, an array of 1-bit flags is used for each window to indicate whether a particular widget is visible or not. This means that there only needs to be a single instance of each window control widget that is shared across all windows.

## **4.6 Discussion**

In the future, additional types of widgets could be used to perform more advanced tasks. A good example of this may be the capability to consolidate a set of buttons into a group. A block-based slider could be useful to give a more

visual representation of step size. Improvements could also be made with the IP widgets so that there are possibilities for more user interaction (interactive histograms that have movable ranges, images that pan or zoom when mouse gestures are performed directly on them).

Displaying widgets in real-time can be a complex process since we are required to display widgets on multiple windows that frequently overlap. This means that the display of a particular widget can be interrupted due to a portion of that widget being occluded from view. Coping with the horizontal and vertical latencies of IP algorithms is a complex task; worthy of future research.

An alternative algorithm would be desirable for the scheduling and display of widgets within the algorithm control layer. It should allow widgets to be positioned anywhere within the layer, and not just in tabular rows. It may be possible to adapt the approach used to manage and position windows to do the same for widgets. Again further research in this area could be very beneficial to GateOS.

# CHAPTER FIVE

## MANAGEMENT & DISPLAY OF TEXT

### Chapter Outline

This chapter discusses the need for a text manager in GateOS. We identify the requirements for this text manager and the context it will be used within GateOS. Several potential designs are proposed to fulfill these requirements, which include an image table approach and an alternate approach that incorporates the use of a font table.

### 5.1 Introduction

One of the primary design goals of GateOS is to provide an interactive user environment that facilitates the real-time debugging of image processing algorithms on FPGAs. Invariably, this requires a user environment to display text as an essential part of the Graphical User Interface (GUI). Text has long been a standard method of conveying useful information to an end user. In fact, ever since the introduction of the text-based console, the display of text has been the primary method of communication between the user and any operating system. With the advent of the GUI, there has been a movement away from pure text to the use of icons and other visual cues by an operating system. In spite of this move, however, text is still an integral part of any communication between the operating system and a user.

Given that the primary function of GateOS is to provide tools that simplify debugging and algorithm tuning, a fully-fledged text manipulation system

(multiple fonts, underlining...) would be overkill. Therefore, within GateOS, text management is defined to be the system responsible for the storage and retrieval of strings of characters that may be displayed in real time on the VDU. These strings of characters can be either static (characters cannot be modified at runtime) or dynamic (limited modification of characters is possible at runtime).

## **5.2 Requirements and Intended Usage**

Text can be used to annotate windows, so as to explain each window's purpose to the user. Also, text may be used to annotate widgets (button widgets, text edit widgets and stand-alone labels) in order to visually describe its current state or display other useful information. Textual labels situated on images can be used to indicate regions of interest to the user. Text can also be input from a keyboard device and displayed within a text edit widget. It therefore follows that the text management system will be required to display strings of characters at particular locations on the VDU. The start or end of the displayed strings may have pixels truncated, for example if a widget or window border is encountered.

### **5.2.1 Window Labels**

It is necessary for the text manager to be able to manage and display individual textual labels for each window in GateOS. This label would enable the easy identification of a window and provide some indication as to its purpose. Such a label should be positioned directly below the topmost border of a window, and should also be left-aligned. Doing this simplifies display offset calculations (when compared to text being right aligned) and offers a familiar look to the user (see Figure 5-1). In most instances the window label is defined at compile-time, although for dynamic windows it may also be useful to allow the label to

be modified at run-time. Since the label is a simple textual annotation, the use of a fixed-pitch font with a limited palette of colours is considered enough to fulfil the basic requirements of a window label in GateOS.

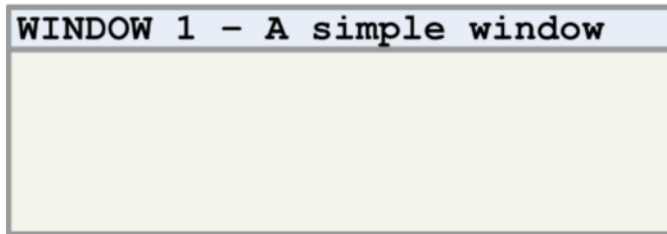


Figure 5-1 A simple textual label displayed on a window

## 5.2.2 Widget Annotations

The text manager is also required to be able to manage and display textual labels for widgets. Such labels are similar to those used by windows in GateOS, as shown in Figure 5-2. Generally, the content of each textual label is unique to each widget (except window buttons).

Normally, the label associated with a widget is constant. Some widgets (e.g. particular types of buttons) may require two or more strings in order to properly represent the state. At runtime, it is necessary for the widget to use the current state to identify the correct string to be displayed.

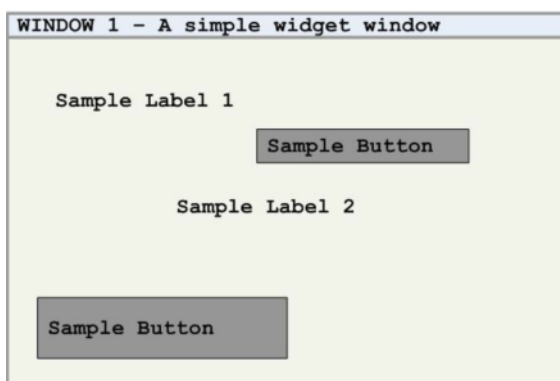


Figure 5-2 String annotations on label and button widgets

### **5.2.3 Output Text Box**

Output text boxes positioned within windows can be used to convey static (help and useful tips) or dynamic (during debugging) information to the end-user. The text manager would be required to manage and display the text within the content area of this widget type. It should also provide support for the runtime modification of the string.

### **5.2.4 Image annotations**

From time to time, the developer may wish to annotate regions of interest on an image. Such annotation consists of a short string that can be dynamically instantiated and positioned anywhere on the image. The maximum number of annotations per image needs to be determined at compile-time by the developer. Since they appear on the widget layer (annotations are just label widgets), the widget manager is responsible for their display. The visibility of each annotation is controlled by the developer's algorithms in the IP core (using a bitmask).

### **5.2.5 Text editing**

The text manager is also used to display the text in a text input widget. The content of the string is edited at runtime by the user via the keyboard. The text manager will be required to modify, insert or remove stored characters in response to this user input.

## 5.3 Analysis and Design

The three key functions of the text manager are to provide storage for strings (both static and dynamic), facilitate editing of dynamic strings and display the appropriate strings on the display as requested by windows and widgets.

At a minimum, the text management system needs to provide support for at least one font in order to visually represent characters on screen. For the sake of simplicity, a fixed pitch font is used with the width and height of each character being  $2^m$  and  $2^n$  respectively. This makes it easier to calculate the character offsets on the display. Without these restrictions it would be necessary to involve either using multipliers (which can get expensive) or calculating the positions of each character by adding the widths as each character is displayed. Hence, the current implementation of GateOS uses a single fixed-pitch font that is 8 pixels wide and 16 pixels high for each character. The justification for this is that a width of 4 pixels is too narrow to effectively represent the characters, while a width of 16 pixels is too wide for most applications. A character width of 8 pixels on a 640 by 480 screen gives up to 80 characters per scanline. A similar argument can be made for the height, giving 8 or 16 as suitable heights. It is also less important for the height to be a power of two unless we have contiguous blocks of text. Finally, a single font (and size) was considered sufficient for the requirements of GateOS. Supporting multiple fonts or sizes would considerably expand the resources required to implement the text manager on the FPGA.

In order to display text on the VDU, we have developed two separate techniques. The first technique (section 5.3.1) uses bitmaps of entire strings of characters and can only be used for displaying static text, while the second (see section 5.3.2) uses bitmaps of individual characters that can display static and dynamic text.



## 5.3.1 Image-Table Lookup Method

The simplest design for displaying text in GateOS is to represent each string by a separate bitmap. All of the character strings used in GateOS would be stored in a lookup table as a series of bitmaps. At run-time, the text manager, once properly scheduled, provides a continuous stream of pixels obtained from this table onto the output display.

There are two possible techniques to produce the data content to populate this lookup table. The first technique makes extensive use of Handel-C's pre-processor macro capability. The second technique requires the construction and use of an external tool, which manages all the character strings in GateOS by creating the bitmaps and representing them in the form of a lookup table that can be loaded by or included into Handel-C.

### 5.3.1.1 Construction using Pre-processor Macros

In this method we define a pre-processor constant that contains the bitmap for each character used in GateOS. The total number of bits needed per character is  $m \times n$  bits, where  $m$  is the width of the character and  $n$  is its height. The individual character constants are then combined to construct the bitmap for the complete string, as shown in Listing 5-1 and in Figure 5-3. It must be noted that the following example, which uses a 4x5 font for the sake of simplicity in this thesis, may in fact make addressing quite awkward.

```
#define _A      0,1,0,0,
               1,0,1,0,
               1,1,1,0,
               1,0,1,0,
               1,0,1,0

#define _B      1,1,1,0,
               1,0,1,0,
               1,1,0,0,
               1,0,1,0,
               1,1,1,0

#define _C      1,1,1,0,
               1,0,0,0,
               1,0,0,0,
               1,0,0,0,
               1,1,1,0

unsigned 1 LUT Text [] = { A , C , B , B , A , B , C , B};
```

Listing 5-1 Compiling an image table using bitmaps for the characters A, B and C

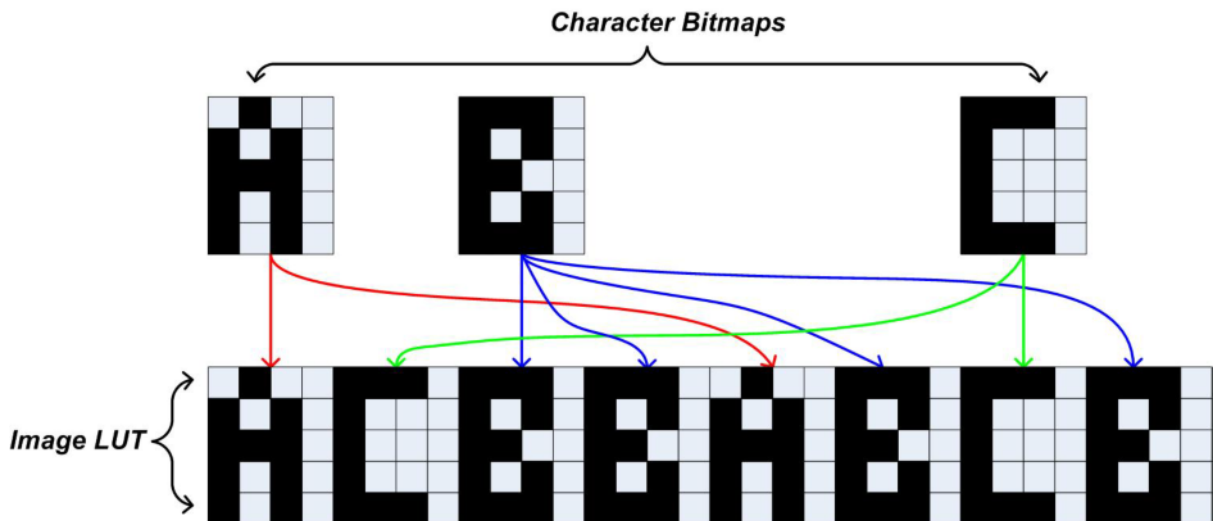


Figure 5-3 The construction of the String LUT at compile-time

The addressing is simplified if the bitmaps are stored column-wise rather than row-wise. The steps between successive bits on a row would be a constant (the character width) which is fixed at compile-time. The construction of the lookup table then involves the repeated use of the macro character definitions. It must be stated that all the bitmaps representing each string are combined into a single table. The text manager could then access the bitmap data for a particular string by its offset in the table. While this approach allows the strings to be constructed within Handel-C, representing a string by a sequence of individual letters is both clumsy and unnatural (as Handel-C lacks the capability to manipulate and scan strings at compile-time),

### 5.3.1.2 Building string bitmaps with an external program

To make the manipulation and bitmap construction a little more natural, an external program could be created that converts characters strings into bitmaps. This program would allow the strings to be more easily manipulated and result in the semi-automatic generation of data content for the look-up table (see Figure 5-4). The developer would still need to perform the supplementary step of copying the bitmap data into the Handel-C source file.

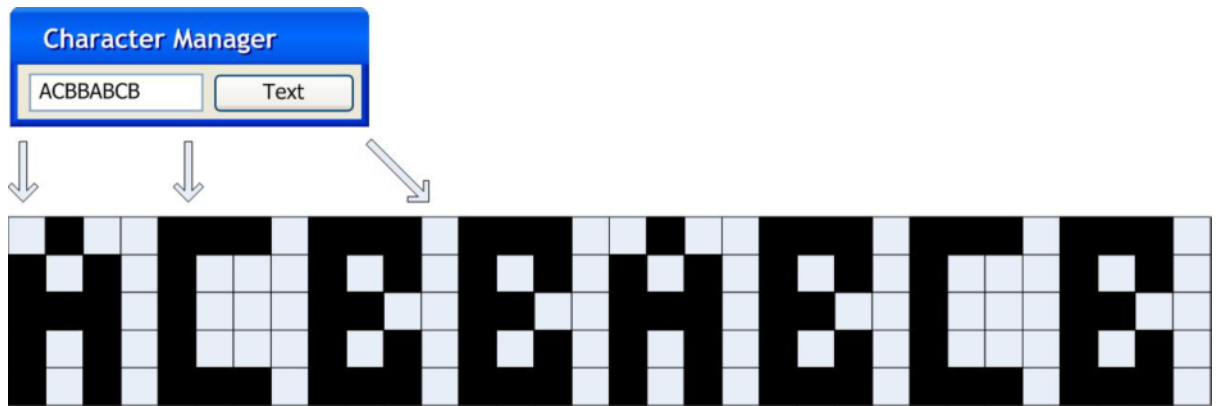


Figure 5-4 The use of an external utility to construct Image Table

Changing the font style and the dimensions of characters can now be done with relative ease, depending, of course, on how well the external program is coded. The overall time required of the user to manage strings using this method would be less than using Handel-C macros; however, some applications may require many strings resulting in a lot of copying and pasting.

### 5.3.1.3 Discussion of both methods

Use of bitmaps to represent entire strings is unlikely to be useful in GateOS because it is incapable of supporting dynamic text. The need to construct the bitmap data for each string also makes it quite clumsy. The advantage of using string bitmaps, however, is that they can offer a reduced resource count if only a few static characters are required. Conversely, the look-up table may be inefficient when using a large number of characters. There are no resource compensations for repeated characters and due to the real-time constraints imposed by GateOS, advanced image compression of the source images in the lookup-table is not a feasible option. Run-length compression cannot easily be used because it may be necessary to begin displaying part way through the string (if the left of the string is covered by another window for example).

Three parameters are required to schedule the display of a string on-screen: a string's starting display offset, and the string's starting and finishing offsets within the bitmap look-up-table. An extra level of indirection can be applied when there are a large number of strings in a GateOS. This involves storing both the string start and stop offsets in a table and referencing strings indirectly with a unique id as shown in Figure 5-5.

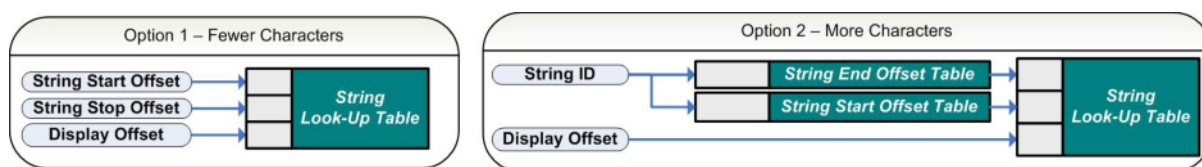
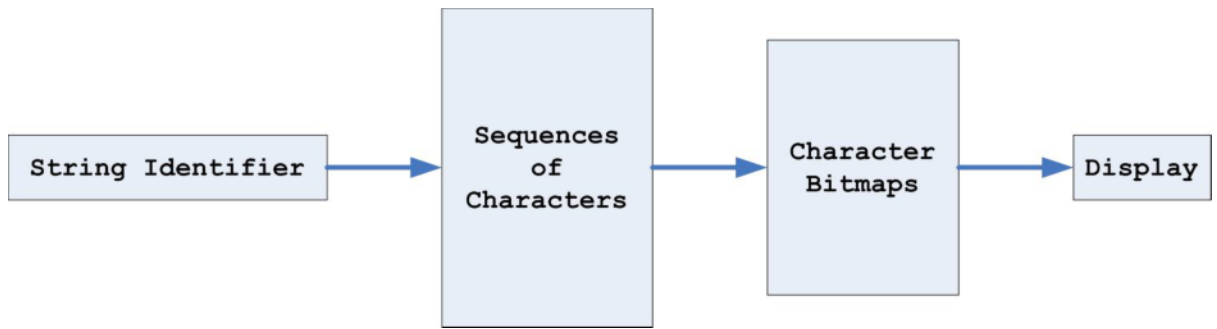


Figure 5-5 Possible implementation strategies

As a final note, storing this look-up table in Block RAM would be preferable where the size of the lookup table is sufficiently large (BlockRAM is more efficient in such situations); however, this depends on the size and availability of BlockRAM.

### 5.3.2 Font Table Lookup Method

An alternative to using bitmaps to represent whole strings is to use the bitmaps to represent individual characters. This requires an extra level of indirection with a table used to store the characters of each strings, and these characters are then used to index the character bitmaps in the font table, as shown in Figure 5-6.



**Figure 5-6 Using bitmaps for individual characters**

A standard ASCII character code requires 7 bits, thus the font table would need to contain the bitmaps for 128 unique entries. To reduce the address decoding logic to access a particular bit in the bitmap, the width and height of the characters should each be a power of two. This then allows an address of a bit in the font table to be formed by concatenating the character code, the row, and the column. A single read of the font table could either return a single bit or a row of pixels, depending on how the table is configured. Reading a row of data would incur an additional resource penalty to buffer the output and shift it out as a pixel stream. However, power savings may be possible with fewer reads from the font table. Currently, a data-width of 1-bit has been selected to minimise the resource requirements.

### **5.3.2.1 Character Scheduling**

To accurately display a string of characters on screen, the text-manager needs to be informed on which character to display next. A crude but simple method of doing this is to manually schedule a new character with the text manager every 8 clock cycles (assuming a fixed character width of 8 pixels). Thus, the Handel-C code to display the character string “Hello World” may resemble that in Listing 5-2.

<i>Clks</i>	<i>Source</i>
	<pre>macro proc DoDelay(t){seq(i=0;i&lt;t;i++){delay;}} par {</pre>
88	<pre>    seq {         CharGen_newchar = 'H'; DoDelay(7);         CharGen_newchar = 'e'; DoDelay(7);         CharGen_newchar = 'l'; DoDelay(7);         CharGen_newchar = 'l'; DoDelay(7);         CharGen_newchar = 'o'; DoDelay(7);         CharGen_newchar = ' '; DoDelay(7);         CharGen_newchar = 'W'; DoDelay(7);         CharGen_newchar = 'o'; DoDelay(7);         CharGen_newchar = 'r'; DoDelay(7);         CharGen_newchar = 'l'; DoDelay(7);         CharGen_newchar = 'd'; DoDelay(7);         CharGen_newchar = 0;     }</pre>
8	<pre>    while(CharGen_newchar != 0){         Load = 1; //signal that a new character to be displayed         Load = 0; //reset signal         DoDelay(6); //wait for six cycles     } }</pre>

**Listing 5-2** The code required to schedule characters for the simplest text manager design

Such an approach can be quite cumbersome to code. The use of a loop that iterates through a table of characters would be easier to code and use fewer hardware resources. The string table contains the sequence of ASCII character codes that complete a string. The text manager is then provided with the indices of both the first and final characters in the string as well as the row number. A display offset is also required to specify the exact location at which the text manger starts generating character pixels. The text manager then iterates through each character in the string and feeds them to the character generator until the final character is encountered. The revised Handel-C program to schedule and display one line of pixels from the text 'Hello World' on the screen is shown in Listing 5-3.

<i>Clks</i>	<i>Source</i>
	<pre>macro proc DoDelay(t){seq(i=0;i&lt;t;i++){delay;}} ram char StringTable[]={ 'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd' }; macro expr IDC_STRING_1_START = 0; macro expr IDC_STRING_1_END = 10;</pre>
1	<pre>for(i = IDC_STRING_1_START; i &lt; IDC_STRING_1_END; i++) par{ //execute concurrently     Index = i; //the current character width     Offset = 0; //the starting character column is always zero     DoDelay(7); //wait for 7 clock cycles (assuming a fixed char width of 7) }</pre>

**Listing 5-3** The use of a dedicated String Table

One can eliminate the need to specify the final index of a string with two modifications. The first involves the definition of a terminating character token.

NULL (0x00) is typically used for this. The second adjustment involves the creation of two additional tables. The start index of each string is stored in the first table and the full length of each string is stored in the second as shown in Figure 5-7. For the current implementation of GateOS, we need to establish this length of a string in a single clock cycle, in order to perform certain boundary checks so as to avoid display overflow (since strings are stored sequentially in the string table separated by a single null character token).

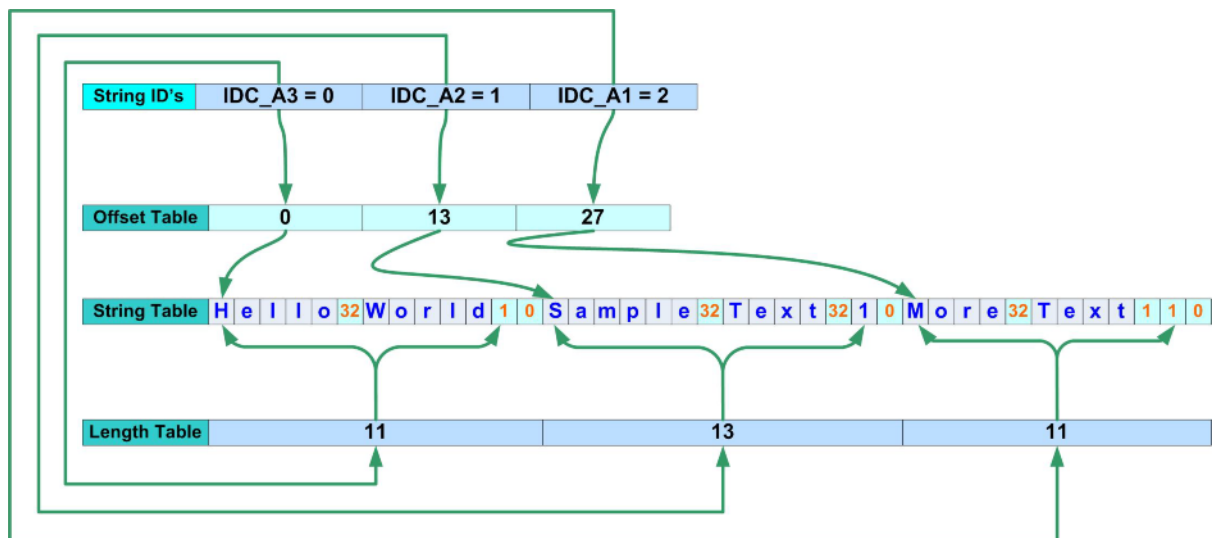


Figure 5-7 The string id and how it relates to the string, offset, and length tables

Thus, to properly schedule the display of a string with the text manager, one need only specify the unique id of the string and the character offset within the string. The character offset is needed for displaying text in an edit widget and for resuming the display of partially occluded text. The revised Handel-C code to display the text "Hello World" is shown in Listing 5-2. The internal architecture that incorporates these changes is shown in Figure 5-8.

Clks	Source
	<pre> ram char StringTable[]={'H','e','l','l','o',' ',' ','W','o','r','l','d'}; ram char StringOffsetTable [] = {0}; ram char StringSizeTable [] = {13}; macro expr IDC_STRING1 = 0; par {     Start = IDC_STRING1;     Offset = 0; } </pre>
1	

Listing 5-4 Scheduling a string with its unique ID

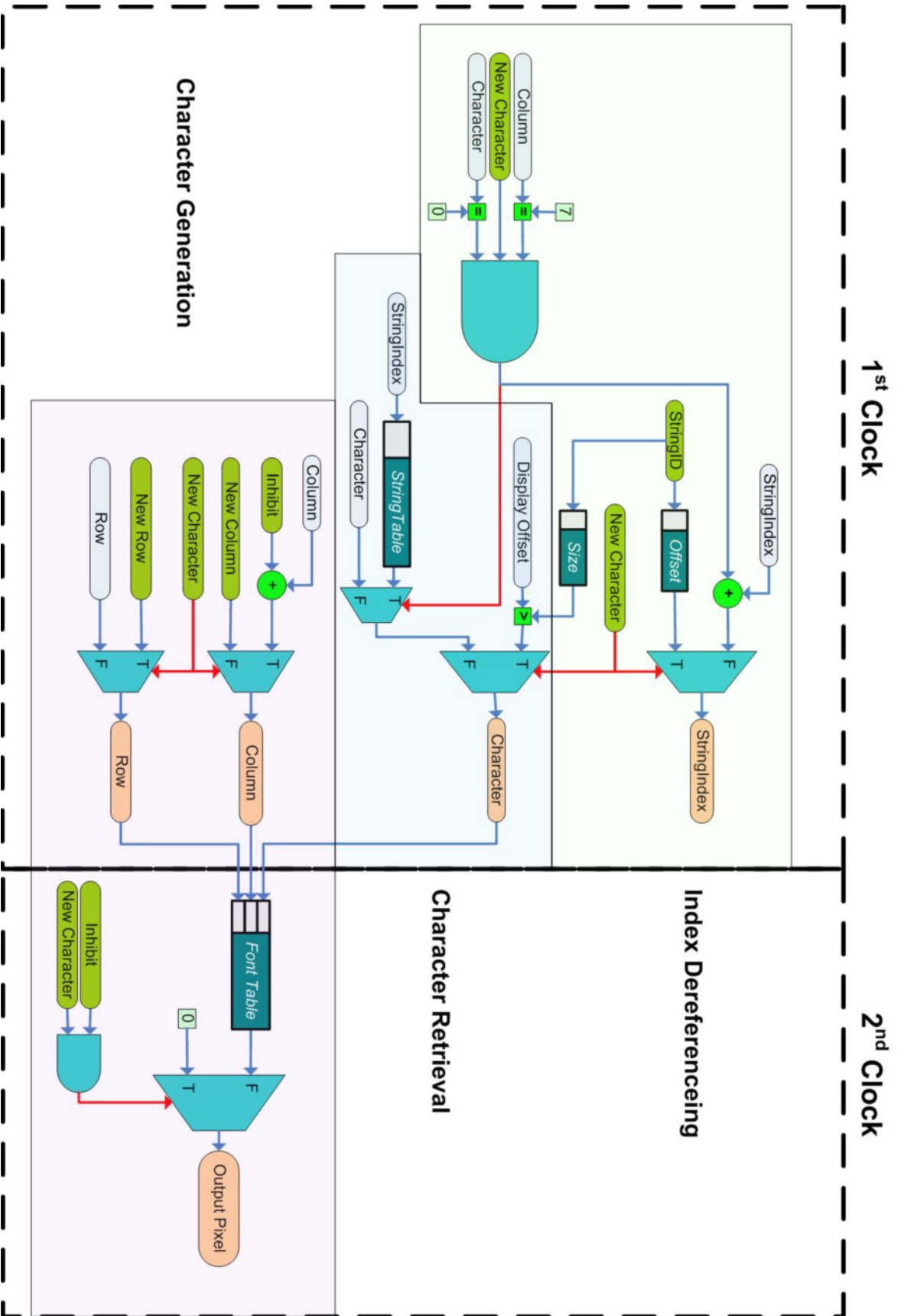


Figure 5-8 The Text Manager design



The preceding design uses a minimalist approach where registers are used to store the results for each particular computation. The reason for using a multi-cycle approach is influenced by two main factors; block-RAM timing considerations and the desire to reduce the control path latency.

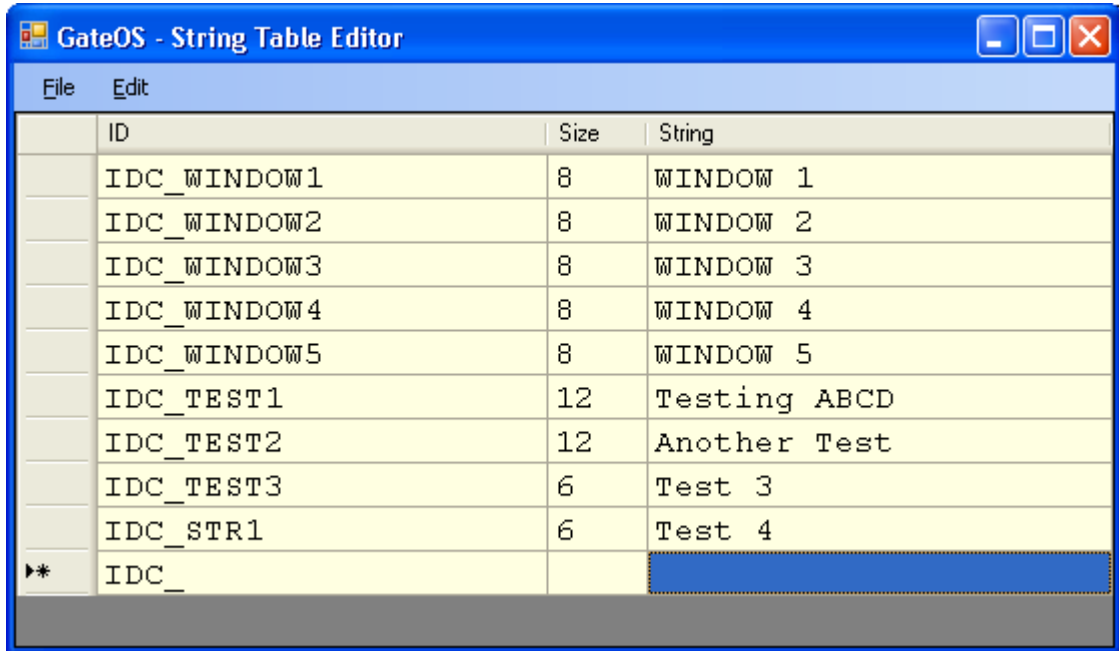
When a string is scheduled to be displayed, the text manager retrieves the total length of the string from the size table and performs a check to guarantee that the display offset is valid (it is not past the end of the string). If the display offset is invalid, the text manager will suspend the output of characters to the VDU, preventing the next string in the string table from being displayed. If the string is confirmed to be valid, the text manager will then iterate through each character in the string until the NULL token is encountered; at which time it will halt. To prematurely flush the output of the text manager, it can be scheduled with an empty string (which contains only a NULL token). The row of the output character is specified by an input.

This system has a latency of two clock cycles from when a string is scheduled (its ID is provided) until the pixels for that string appear at the output. For this implementation, both the string and font tables are stored in separate Block RAMs. If there are a large number of strings, then it may be necessary to place both the size and offset tables in Block RAM; otherwise distributed RAM should be sufficient.

## 5.4 Implementation

I have constructed two external tools to manage the tables used by the text manager. The purpose of the first tool is to convert a simple bitmap file containing the font table into a Handel-C compatible declaration. This tool is a command-line program that was written in Visual Basic .Net. The second tool is also written in Visual Basic .Net, and offers a GUI to allow users to directly

manipulate the strings used in GateOS. A screenshot of this program is provided in Figure 5-9.



ID	Size	String
IDC_WINDOW1	8	WINDOW 1
IDC_WINDOW2	8	WINDOW 2
IDC_WINDOW3	8	WINDOW 3
IDC_WINDOW4	8	WINDOW 4
IDC_WINDOW5	8	WINDOW 5
IDC_TEST1	12	Testing ABCD
IDC_TEST2	12	Another Test
IDC_TEST3	6	Test 3
IDC_STR1	6	Test 4
▶* IDC_		

Figure 5-9 GUI Utility that manipulates the String Table

The GUI automatically generates the string, offset, and size tables in addition to generating all the necessary Handel-C macro expressions that represent the various parameters required by the text manger.

GateOS is able to provide a limited support for dynamic strings. A NULL token is used in the string table to separate strings of characters. An additional token, EMPTY, can be used to pad strings. The use of this token allows for a string to expand or contract at runtime. For display purposes, EMPTY is also treated as a string terminator.

Externally, one is able to schedule a string by providing the text management system with that string's unique id (as well as the starting character offset and the current row) two clock cycles before it is to be displayed. This id is stored as a property in the data structure representing the corresponding object.

## 5.5 Discussion

Currently, there are several limitations with regards to the management and display of textual annotations in GateOS. Only a single font size and type is allowed in order to reduce the FPGA resources used. It is not possible to store and display variable width characters in the current implementation. Also, characters are not allowed to overlap on the VDU.

In addition, only a single line of text can be displayed for each individual annotation. The total number of strings is constant (each string is referenced by a unique id); however, it is possible to resize the length of strings at run-time due to the added level of indirection (the start and stop offset of each string within the string table is stored in a separate data-structure). The latency (two clocks) between the scheduling of a string and its subsequent display is managed through pipe-lining.

The final approach taken fully satisfies the basic set of requirements for a text manager in GateOS. Externally, it allows a developer to display annotations on windows, widgets and images. Support for dynamic (runtime generated) text has been proposed within this chapter, but due to time constraints, has yet to be implemented in GateOS.

We have described two different methods of displaying strings of characters on a VDU. One of which is capable of displaying only static text, while the other method can display both static and dynamic text. We have attempted to minimise the resource requirements while ensuring that the design is as flexible as possible. This paves the way for future updates and new features, which may improve a user's interaction with GateOS. Such features may include support

for additional fonts, variable character sizes, Unicode alphabets and dynamic text. However, for the present time, these features are not considered crucial.

# CHAPTER SIX

## INPUT MANAGEMENT

### Chapter Outline

This chapter focuses on describing how GateOS interacts with both keyboard and mouse input hardware. We discuss how keyboard and mouse cursors displayed on a VDU can be used to represent input from these devices. We describe keyboard and mouse interactions as well as those between a particular input device and the various windows or widgets featured in GateOS.

### 6.1 Introduction

Interaction with GateOS may involve the use of both mouse and keyboard input devices. The keyboard can be used to input characters into text widgets and manipulate the state of button widgets, while a mouse can interact with button, slider and edit widgets in addition to repositioning and resizing a window.

An FPGA development board may incorporate one or more PS/2 ports in order to accommodate keyboard and mouse devices. Each port uses two lines, clock and data, for communication with the device. The FPGA, as the host, controls both these lines and the low bit-rate serial data communications through them.

A device driver is required to manage the communication and provide a programming interface between the device and the user's code on the FPGA. If a keyboard is connected to the PS/2 port, the driver must receive the keyboard

scan-codes and convert them to their associated ASCII character codes. If a mouse is connected then the driver buffers the mouse button status and integrates the movement vectors to obtain the mouse's current position. If the development board vendor does not provide appropriate board level drivers or libraries, these must be developed as part of GateOS. We used the Platform Abstraction Layer (PAL) library provided by Celoxica<sup>6</sup>.

## 6.2 Cursor Layer

The cursor layer is used by GateOS to display both the mouse and keyboard cursors. This layer covers the entire screen area and has the highest display priority of all layers within the system. Within this layer, the mouse cursor is given priority over the keyboard cursor.

The keyboard cursor is a vertical, 1 pixel wide line of the same height as the character font used by GateOS. Since the cursor will only interact with an edit widget, it is only displayed if the widget's window is active. The keyboard cursor in the widget is situated in one of three possible locations; between characters, before the first character or after the last character.

The mouse cursor can be positioned anywhere on the input layer. While in principle, the cursor shape could be anything; we have implemented a white arrow with a black border (the border enables the cursor to be clearly distinguished against a white background) using a small (16x16) bitmap. The bitmap therefore requires 3 values: black, white, and transparent.

The location of the keyboard or mouse cursor is represented by the coordinates (x, y) of the top left pixel of the cursor. The keyboard cursor's location is set when the mouse cursor attaches it to a text edit widget. The keyboard cursor is

---

<sup>6</sup> <http://www.celoxica.com/products/dk/default.asp>

repositioned to the nearest gap between characters in that widget. If the cursor has a height of  $h$  pixels then it is displayed when the following conditions are met:

$$\begin{aligned} \text{ScreenX} &\equiv \text{KeyboardCursorX} \\ \text{ScreenY} &\leq \text{KeyboardCursorY} < \text{ScreenY} + h \end{aligned}$$

A simple Handel-C implementation of the previous condition checks may look like:

```
(ScreenX == KCursorX) && (ScreenY ≥ KCursorY) && (KCursorY < ScreenY + n)
```

There are at least three addition/subtraction operations required here; which in most situations, have a longer critical path than equality comparisons. We can avoid using these by rewriting the previous statement as:

<i>Clks</i>	<i>Source</i>
1	<pre>//during v-blanking Counter = 0; //reset counter for next frame</pre>
1	<pre>//during active region if (KCursorX == ScreenX) par { //execute concurrently     if (KCursorY == ScreenY) //first pixel in vertical line         Counter = h;     else if(counter != 0) //subsequent pixels vertical line         Counter--;     else         delay; //wait for 1 clock cycle     Tflag = 1; //the cursor is displayed the next clock } else par { //execute concurrently     Cursor = Tflag &amp; (Counter != 0); //Should the cursor be displayed ?     Tflag = 0; //reset Cursor output }</pre>

The preceding code now has only a single decrement operation, which should result in slightly less hardware being constructed. This is in spite of an additional register for the counter and a flip-flop for the flag. By pipelining the control path, it has a significantly shorter critical path with regards to timing

To display the mouse cursor, we need to know, both the row( $x$ ) and the column( $y$ ) of the current mouse cursor pixel being output on the VDU and ensure that the following conditions are met:

$$\begin{aligned} \text{ScreenX} &\leq \text{MouseCursorX} < \text{ScreenX} + x \\ \text{ScreenY} &\leq \text{MouseCursorY} < \text{ScreenY} + y \end{aligned}$$

Written in Handel-C, this check condition may look like:

```
(ScreenX ≥ MCursorX) && (MCursorX < ScreenX + m) &&
(ScreenY ≥ MCursorY) && (MCursorY < ScreenY + n)
```

From the above statement we can see that at least six addition/subtraction operations are required. A similar process to that used to display the keyboard cursors can then be applied for the mouse cursor like so:

<i>Clks</i>	<i>Source</i>
2	<pre>//during v-blanking BaseOffset = 0; ShiftReg = 0;</pre>
3	<pre>// during h-blanking if(ShiftReg[0]){     BaseOffset += m; //this is to avoid doing a multiplication (see below)     ShiftReg &gt;&gt;= 1; // while the 0th bit is set, the cursor is displayed     x_cntr = m; // reset this } else delay</pre>
1	<pre>//during active region par {     if (KCursorX == ScreenX) par{         if (KCursorY == ScreenY)             ShiftReg = ~ (0); //start displaying the cursor         else             delay; //delay for a clock cycle         x_cntr = 0; //reset the column counter     }     else if (ShiftReg[0] &amp;&amp; x_cntr != m) par{         Output = CursorBitmap[BaseOffset + x_cntr] //output correct pixel         x_cntr++; //increment column counter     }     else delay; }</pre>

The preceding Handel-C code requires only two addition operations, although this is at the expense of some extra hardware logic. Several further optimisations can be applied, such as eliminating the need for a base offset by restricting the width of a mouse cursor to powers of two, or reading all the pixels required for the next scan-line of the mouse cursor during horizontal blanking.



## 6.3 Keyboard Input

Characters from the keyboard must first be filtered to remove unprintable characters before being stored in the string table (for text edit widgets) or interpreted as a command (for button widgets). The relatively slow input rate allows characters to be processed serially during the vertical blanking period of the output display driver. This also avoids resource access conflicts with the string table, which is potentially being used by the text management system during the display periods.

If either the 'control' or 'alt' keys are depressed when a character is entered, then that input is treated as a command. Otherwise, the current input is treated as a stream of input characters to an edit widget.

Each button in GateOS may have a 'hotkey' associated with it. If the keyboard cursor is not attached to a text edit widget, then when characters are entered, they are first compared against each button widget's hotkey (located in the widget data-structures). If a match is found (globally, not just the active window), then the button's state is toggled, otherwise the input character is discarded.

The id of the currently active edit widget is stored (in a register), as well as the offset of the keyboard cursor from the start of the string that is displayed within the widget.

## 6.4 Mouse Input

When a mouse button is pressed, it is necessary to identify the window and any associated widget at that screen location. The most obvious solution involves iterating through each window in order of priority to determine the window,

and then iterating through each widget contained by that window to identify the widget. This approach would involve significant hardware resources.

Since GateOS uses on-the-fly pixel generation, the display process must determine which window and widget is associated with each pixel on the display in order to display it (this was covered in CHAPTER Two). Therefore the hardware already exists to locate a window and widget from a pixel position. When the display driver is at the mouse cursor position, the window and widget currently being displayed can simply be recorded.

A mouse button click within an inactive window will make that window active (and move it to top priority in the display order). A window can be repositioned anywhere on screen by dragging it with the left mouse button within that window's title-bar. During such a drag operation, when the mouse is moved, the window position is updated to maintain the same position relative to the mouse. When the mouse button is released, the window will no longer be repositioned. A window can be resized by dragging the right-angled resize grip located on the bottom right window border. The coordinates of the window's bottom right position are then be continuously updated to reflect the mouse's drag movements.

A mouse cursor may also interact with a button widget. A right click anywhere within the button widget's content area will toggle its state value. In some applications a push-button behaviour may be required. To accomplish this, IP core simply resets the button to the default state after reading the associated flag.

The mouse can also be used to reposition the keyboard cursor, and to attach it to an edit widget. A mouse cursor can also be used to reposition up to two slider bars within a slider widget. The mouse gestures involved with these tasks are detailed in CHAPTER Four.

## 6.5 Discussion

The preceding material only describes GateOS as supporting a single mouse and keyboard. For debugging and tuning image processing algorithms, this is sufficient, however in principle there is little stopping multiple mice or keyboards from being used. Supporting extra input devices would require more PS/2 ports and associated driver hardware. The user interactions with GateOS would also be more complicated since multiple windows would be active simultaneously. One would have to resolve which active windows are topmost or (possibly) occluded, as well as handling scheduling and management issues that occur when two mice interact with the same window or widget.

# CHAPTER SEVEN

## DISCUSSION & CONCLUSIONS

### Chapter Outline

This chapter reflects on the research completed on GateOS. Some of the more important problems and limitations apparent in the current implementation of GateOS are briefly explained. We also highlight potential areas that can be targeted by future research.

The current approach taken towards the design of GateOS is both functional and effective and is targeted towards supporting as many small and medium scale FPGAs as possible. We have attempted to use a minimal amount of resources to meet all of the requirements of GateOS. In this regard we have succeeded in reusing hardware (using a single block of hardware and multiplexing it for multiple items) for windowing, widgets and the control of image processing algorithms. Overall, this approach has contributed substantially to using FPGAs in stand-alone mode, this by providing tools for tuning and debugging IP algorithms.

In reflection, some of the goals mentioned at the start of the thesis, seem to have been somewhat optimistic rather than realistic. As the design of GateOS has evolved, many compromises were necessary in order to fulfil the core goals of the thesis; namely those pertaining to the use of a minimal amount of resources for the implementation of each sub-system. Towards the end of the thesis, when time was running short, sub-systems such as the non-volatile storage manager and the frame-manager had to be deferred in favor of providing a working

demonstration as proof of concept for GateOS. In time, these sub-systems will be incorporated into GateOS, thus fully qualifying it as proper Operating System, instead of the cut-down hybrid it is now.

The string management system in GateOS could be improved to support the display of variable pitch fonts. The current design of the text management system is capable of displaying any ASCII-coded character on the VDU. The window management and display system has been the primary focus of much of my research. The resulting algorithms and techniques achieve their objective of displaying windows when using on-the-fly pixel generation. The approaches used have most of the core window's structures stored in RAM instead of registers, thus saving a significant amount of FPGA resources. The widget system currently functions in a limited capacity (widgets can only be displayed in rows) and could benefit from additional work, particularly with regard to finding an algorithm that would enable widgets to be positioned anywhere in a window instead of being subject to tabular layout restrictions.

The text edit widget has been designed, but has not been implemented at this stage, as it is considered to be less important than the other widgets. More advanced keyboard input and gestures (such as scroll wheels manipulating slider bars, or multiple mice controlling different windows concurrently) could be used to perform additional tasks to enhance end-user interactions. Character input into the text widget could conceivably be used by a developer at runtime to identify, and subsequently save, images as files (for example in flash RAM).

A proper file system for nv-RAM storage that uses either some form of file allocation table (FAT), or node based structures (EXTFS), could provide more flexible storage of images than a block based approach (a file is stored as a continuous block from a start offset to an end offset) . File-system operations are inherently serial in nature, thus the use of a serial processor may be more efficient (in terms of hardware resources) for this.

In the requirements section (see CHAPTER Two), the possibility of using a scripting language to configure the various aspects of GateOS at compile-time was considered. This concept could be extended by constructing a GUI that would let the developer graphically configure the properties and behaviours of windows, user widgets and IP widgets in addition to specifying their relationships with the IP core (i.e. associate button widget six to index 4 in IP core Boolean register bank). The developer could then directly position a widget within a control window using a mouse on the host PC. This GUI would then automatically generate the necessary Handel-C code that would configure the appropriate sections of GateOS at compile-time.

Currently, the developer needs to construct and manage any image storage areas required by their image processing algorithms. It may be possible for GateOS to manage the memory management for simpler IP algorithms.

The next step for GateOS is to support the display of live video feeds. GateOS could benefit from being able to display multiple overlapping video feeds concurrently on the same VDU; so as to provide a better end-user experience with debugging multiple algorithms or steps in an algorithm. The number of concurrent video feeds is limited only by the storage memory available for buffering the input video frames before they are processed by the IP algorithms. Additional research is needed to support other types of histograms. Supporting the display of 2-D histograms could be quite useful for particular IP algorithms.

Configuration	Slices Used (out of 15,360)			
	8	16	32	64
Simple Configuration (no widgets)	1253 slices	1339 slices	1462 slices	1759 slices

Table 7-1 Slices used on a Xilinx ML402 (Virtex 4) for 8, 16, 32 and 64 windows

Since the implementation of the windowing system in GateOS has undergone several revisions, it is relatively stable. Because of this, the number of slices used by the windowing system in GateOS has also undergone some reductions. From the preliminary results shown in Table 7-1, it is evident that a small increment in FPGA slices is required to support an increased number of windows in GateOS.

The hardware requirements to implement the windowing system are listed in Table 7-1. Adding additional windows requires only modest extra hardware. The window manager itself does not change. Therefore, this extra hardware is that required to store the window parameters and the window list and edge list tables. The size of each of these tables is proportional to the number of windows, and as these are stored in FabricRAM, the requirements for each additional window are modest. Note that as more windows are used, the number of bits required to represent a window ID also increases.

The algorithms for displaying and manipulating widgets are still being refined, and as such a relatively large number of slices are used (3,275 slices for 6 windows and 16 widgets).

<b>Components</b>	<b>LUT</b>	<b>FF</b>	<b>Mem</b>	<b>Other</b>
Windows + Widgets (6 Windows + 16 Widgets)	3807	1407	6032	1757
Strings (Character Tables)	18	0	18720	9
Graphics (Mouse cursors)	0	0	18432	0
Fonts (Font table for Character Generator)	0	0	16384	0
Math (multi-cycle multiplication and division macros)	251	91	0	99
Display (selects the correct layer and pixel to display)	165	168	0	32
Mouse (one mouse)	100	100	0	20
Character Generator (generates characters for display)	76	46	0	34
Core (where everything starts)	3	13	0	0
<b>TOTAL</b>	<b>4420</b>	<b>1825</b>	<b>59568</b>	<b>1951</b>

**Table 7-2 Breakdown of resources required, as estimated by the Celoxica build tools**

To summarize, we have created a reusable, configurable windowing environment for FPGAs in the form of GateOS. Preliminary results (see Table

7-1 and Table 7-2) indicate that GateOS could be a viable solution for tuning and debugging image processing algorithms when operating an FPGA in the stand-alone model. This thesis demonstrates that a windowing operating system can be practically built on an FPGA. Although it is quite basic, even minimalistic in its current form, it has served its purpose and can already be used for the configuration and graphical debugging of image processing algorithms on an FPGA. The source code for the current version of GateOS is included on the CD attached to this thesis.



## REFERENCES

- Bailey, D.G., Gribbon, K, Johnston, C (2006). GATOS: A Windowing Operating System for FPGAs. Proceedings of the third IEEE International Workshop on Electronic Design, Test, and Applications (Delta 2006), Kuala Lumpur, Malaysia, (pp 405-409)
- Benitez, D. (2002). Performance of remote FPGA-based coprocessors for image-processing applications. *IEEE Euromicro Symposium on Digital System Design* (pp. 268-275).
- Chan, S. C., Tsui, K. M., Yeung, K. S., & Yuk, T. I. (2007). Design and Complexity Optimization of a New Digital IF for Software Radio Receivers With Prescribed Output Accuracy. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* , 54 (2), (pp 351-366).
- Crookes, D., Alotaibi, K., Bouridane, A., Donachy, P., & Benkrid, A. (1998). An environment for generating FPGA architectures for image algebra-based algorithms. *IEEE International Conference on Image Processing*. 3, (pp. 990-994).
- Gribbon, K.T., Johnston, C.T., Bailey, D.G. (2006). Formalizing Design Patterns for Image Processing Algorithm Development on FPGAs, *Proceedings of the third IEEE International Workshop on Electronic Design, Test, and Applications (Delta 2006)*, Kuala Lumpur, Malaysia, (pp 47-53)
- Hemmert, K. S., & Underwood, K. D. (2007). Floating-Point Divider Design for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* , 15 (1), (pp 115-118).
- Hsiao, P.-Y., Li, L.-T., Chen, C.-H., Chen, S.-W., & Chen, S.-J. (2005). An FPGA architecture design of parameter-adaptive real-time image processing system for edge detection. *IEEE Emerging Information Technology Conference* (p 3).

- Johnston, C.T., Bailey, D.G., Gribbon, K.T.(2005). Optimisation of a color segmentation and tracking for real-time FPGA implementation, *Image and Vision Computing New Zealand*, Dunedin, New Zealand, (pp 422-427)
- McCurry, P., Morgan, F., & Kilmartin, L. (2001). Xilinx FPGA implementation of an image classifier for object detection applications. *IEEE International Conference on Image Processing*. 3, (pp 346-349).
- Ramdass, T., Ang, L.-m., & Egan, G. (2004). FPGA implementation of an integer MIPS processor in Handel-C and its application to human face detection. *IEEE Region 10 Conference*. 1, (pp 36-39).
- Tahoori, M. B. (2006). Application-Dependent Testing of FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* , 14 (9), (pp 1024-1033).
- Tessier, R., Betz, V., Neto, D., Egier, A., & Gopalsamy, T. (2007). Power-Efficient RAM Mapping Algorithms for FPGA Embedded Memory Blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* , 26 (2), (pp 278-290).
- Tomko, K. A., & Tiwari, A. (2000). Hardware/software co-debugging for reconfigurable computing. *IEEE High-Level Design Validation and Test Workshop*. (pp. 59-63).
- Uzun, I., & Bouridane, A. (2003). FPGA implementations of fast fourier transforms for real-time signal and image processing. *IEEE International Conference on Field-Programmable Technology (FPT)* (pp 102-109).
- Vitabile, S., Gentile, A., Siniscalchi, S., & Sorbello, F. (2004). Efficient rapid prototyping of image and video processing algorithms. *IEEE Euromicro Symposium on Digital System Design* (pp 452-458).
- Wigley, G., & Kearney, D. (2001). The First Real Operating System for Reconfigurable Computers. *IEEE Computer Systems Architecture Conference Gold Coast, Qld., Australia* (pp 130-137).
- Yano, Y., Hashimiyama, T., & Okuma, S. (1999). On-line filter generation for binary image processing using FPGA. *IEEE International Conference on Systems, Man, and Cybernetics*. 5, (pp 565-570).

Yiannacouras, P., Steffan, J. G., & Rose, J. (2007). Exploration and Customization of FPGA-Based Soft Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* , 26 (2), (pp 266-277).