Massey University

Institute of Information Sciences & Technology

A REAL-TIME FPGA IMPLEMENTATION OF A BARREL DISTORTION CORRECTION ALGORITHM

Christopher Johnston

2003

Supervisor
Dr Donald Bailey

Contents

Abstract

This final year Information and Telecommunications Engineering project report presents a novel FGPA implementation of a barrel distortion correction algorithm. In order to perform real-time correction in hardware the undistorted output pixels must be produced in raster order for display to a VGA screen. To do this the implementation uses the current scan position in the undistorted image to determine which pixel in the distorted image to display. The implementation employs the use of a look-up table with interpolation to reduce the complexity of the hardware design, without significant loss of precision. The report details the background of the hardware and design environment used, and then barrel distortion and the model selected for correcting it. The implementation aspects are discussed and the results of the implemented design are shown. This leads to a discussion on future work and conclusions.

Keywords: FPGA, reconfigurable hardware, lens distortion, camera calibration

Acknowledgements

The Celoxica University Programme for generously providing the DK1 Design Suite.
Dr Donald Bailey my supervisor for his guidance with this project.
The Institute of Information Sciences and Technology for providing the Image Processing lab and the RC100 development board.

1 Background

1.1 FPGA

Field Programmable Gate Arrays combine the speed of hardware with the flexibility of software programming. FPGA devices feature a gate array like architecture with a matrix of logic cells surrounded by I/O cells. The logic cells called configurable logic blocks (CLBs) are linked together using segments of metal interconnects which can be linked together in an arbitrary manner by programmable switches, figure 1.1. This means that almost any routing combination is possible.

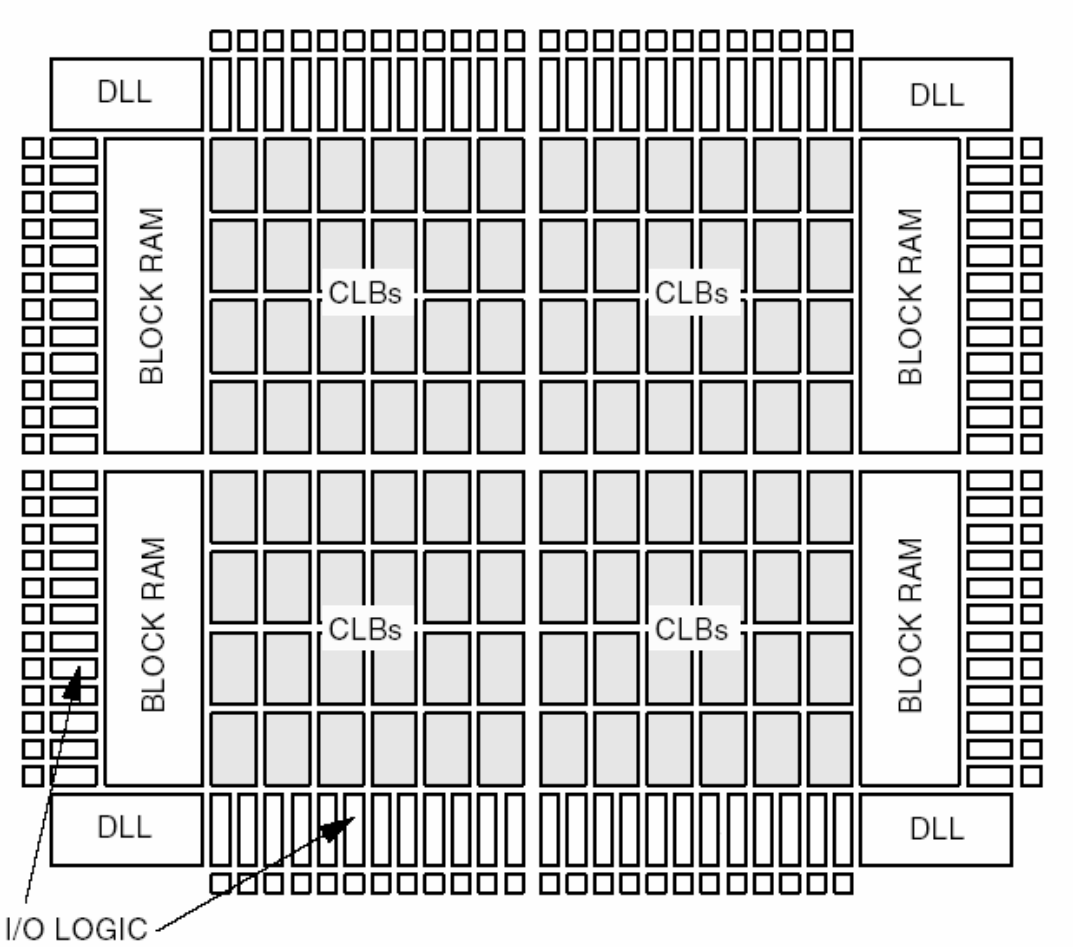

Figure 1.1 FPGA layout taken from [2]

Xilinx was the first to introduce these devices in 1985 and offer a number of families of re-programmable, static memory based FPGAs including the one used in my project the SPARTAN II. There are two main building block in Xilinx FPGAs; CLBs which provide

the functional elements of constructing the required logic (figure 1.2), and input/output blocks (IOBs) which provide the interface between the pins of the package and the internal signal lines.

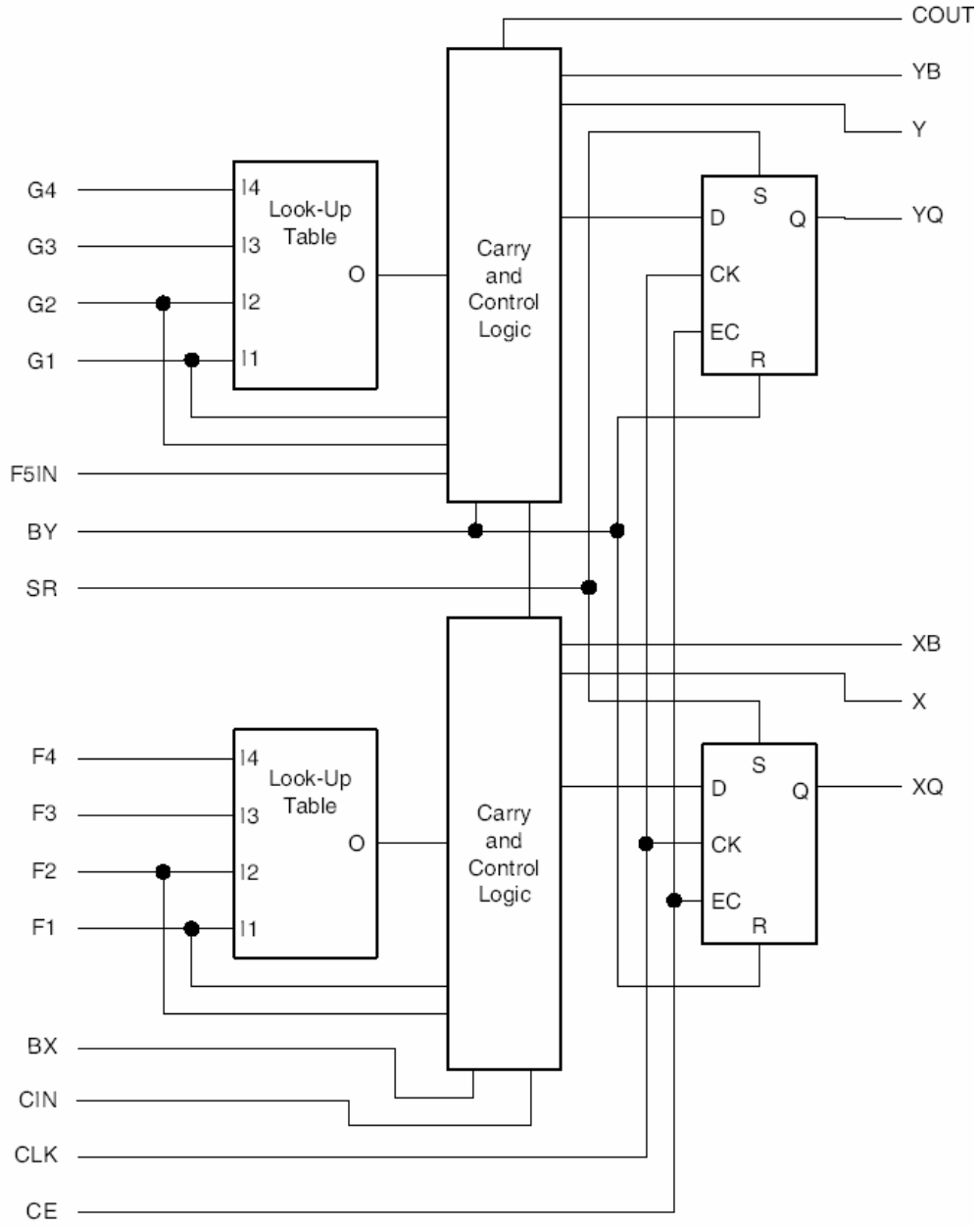

Figure 1.2 Spartan II Slice, there are two slices in a Configurable Logic Block, taken from [2]

Block RAM is also provided on some families, including the SPARTAN II. These are small blocks of RAM that can be declared as on chip RAM or ROM for use with look up tables or buffering. Memory functions can also be constructed out of CLBs that allow for all elements to be accessed at a time however this reduces the complexity of logic design that can be implemented due to the reduction in CLBs for logic. On Xilinx products the Block RAM is dual port where one read and one write or two reads can be performed in a

single block at the same time. The functionality of each circuit bock is customised during configuration by programming internal static memory cells. The values stored in these memory cells determine the logic functions and interconnections implemented in the FPGA.

There are a number of different lengths of interconnects including; direct between CLBs, single, double and long lines. The reason for this is that when a signal is passed in to a switching matrix there is a delay caused by the programmable pass transistors which are used to establish connections between lines. Thus to improve the speed of communication between CLBs which are located a distance apart the longer lines are used to reduce the number of switching matrix passed through. Single lines provide the greatest interconnect flexibility and offer fast routing between adjacent blocks.

Double-length lines run past two CLBs before entering a switching matrix. These lines are grouped in pairs with one of the pairs going into the switching matrix while the other bypasses it. These then swap for the next switching matrix. Double-length lines are used to provide fast intermediate interconnects while still retaining a level of routing flexibility, figure 1.3.

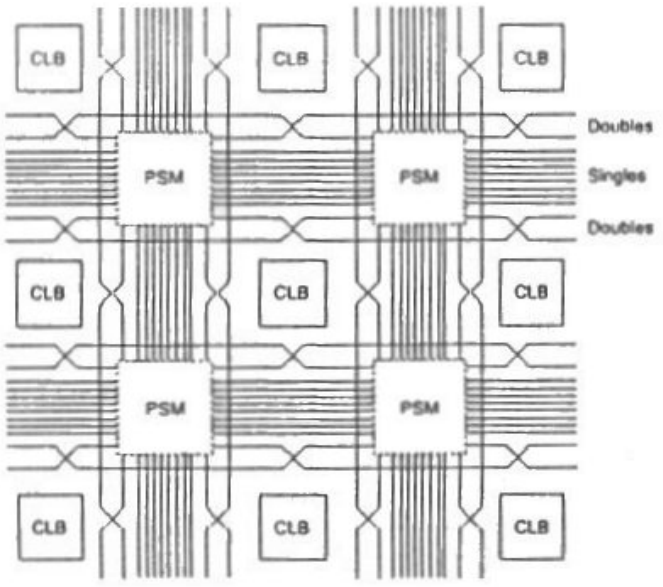

Figure 1.3 Interconnects and Switching Matrix taken from [1]

Long-lines form a grid of interconnects that run the length or width of the FPGA, figure 1.4. These are intended for time-critical signals or nets that are distributed over a large distance. These do not pass though the switching matrixes.

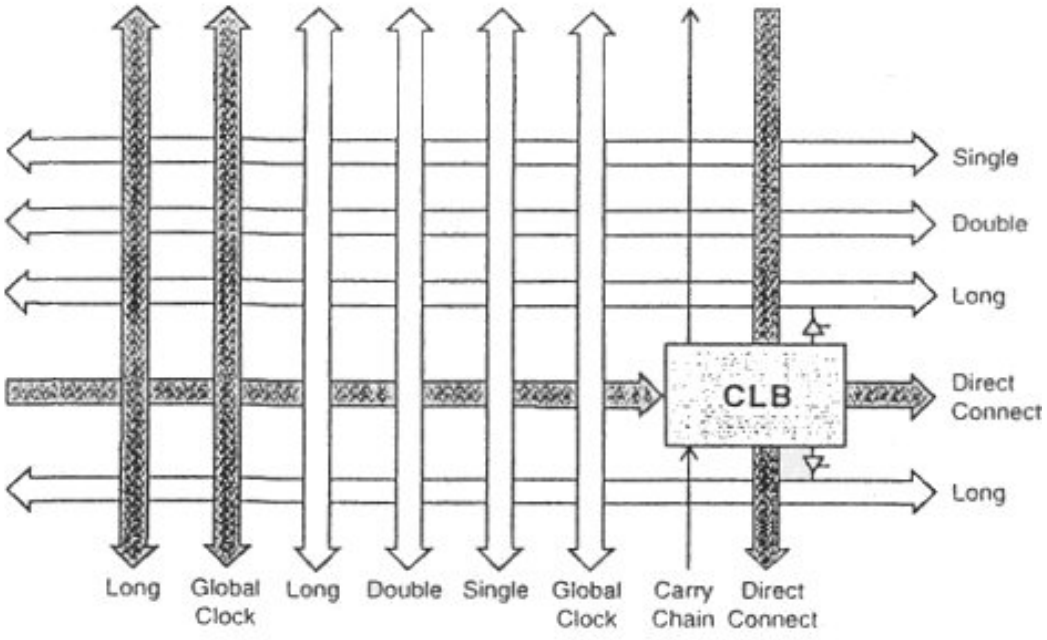

Figure 1.4 Signal Lines, taken form [1]

Due to the complex nature of the routing it is normal to use an automatic routing algorithm. For this to work effectively in the design of the logic signals that are time critical need to be identified and maximum timing constraints found. These constraints can then be used by the routing algorithm to find a configuration of interconnects which can achieve the desired result. This routing algorithm also needs to be able to move the logic functions implemented in one CLB to another CLB to enable faster signalling.

Mapping of described logic to CLBs also needs to be done. Simple functions might be implemented in a single CLB but if the function is complex, like an adder or multiplier, several CLBs might need to be used, with the logic spread between them.

Finally I/O pins need to be configured for input, output or tri-state operation. And the appropriate signals need to be routed form the pins to the logic.

Once this hardware is mapped and routed for the FPGA a bit file in a PROM file is created to configure the device [1]. The bit file is stored into the static RAM of the FPGA, and controls the functions f the CLBs, IOBs, and the connections made by the switch matrix in the interconnects.

1.2 Celoxica

The design was prototyped on the Celoxica RC100 development board using Handel-C. Handel-C is designed to be a high-level design language that can be compiled to hardware. By using FPGAs in the design of a system, all or part of the design can be implemented in hardware rather than software. By doing this there can be significant speed improvement by removing the normal need in a CPU to split the functionality up into individual instructions than need to be fetched, decoded and the executed. Any parallelism in the algorithm can also be exploited by having hardware functions running concurrently. Up until recently any gains from doing this were achieve by converting a high level algorithm in to a hardware description language (HDL) such a VHDL. This is a complex process and higher-level languages are needed to allow integration between current design methods, this is what Handle-C is.

1.2.1 Handel-C

Handel-C is a language with an associated development environment DK1 developed by Celoxica, which was formed out of the University of Oxford in 1996 to commercialise its research into Handel-C. Handel-C is a C based language with extensions for hardware; it is aimed at compiling high-level algorithms directly to gate level hardware. It can also compile to VHDL for incorporation into an existing system.

Handel-C can be used to design sequential programs but to gain speed improvement parallel constructs need to be used. Handel-C allows for the algorithm to be written without any knowledge of the underlying architecture which makes it a true high level programming language. Handel-C generates the required logic gates from the source code, however it works at the register transfer level. This means that each assignment is clocked into a register after calculation. This enables Handel-C to ensure that all assignments are performed in a single clock cycle. However the complexity of the assignment will affect the speed at which the FPGA can be run due to the combinatorial delays created by deep logic. Handel-C also ensures that the control logic of the language constructs adds no additional clock cycles to the implementation, again they add to the length of the system clock cycle.

The main language extension in Handel-C is that of the PAR statement, this enables several statements to be run in parallel. There is also the inclusion of I/O pin constructs, and port and channel constructs to enable communication between external interfaces and parallel process respectively. This is needed to allow the passing of data from one thread or process to another. Channels also allow parallel processes to synchronise with each other. Data types such as the signal have been added which act like wires in a hardware design. There are also extensions for bit manipulations including bit selection and concatenation of variables.

Although much of ANSI-C is supported there are some restrictions:

- No floating-point support, due to the increased gate count that floating-point instructions involve.
- No recursive functions, this is due to the lack of stack and the fact that the same hardware would be used to perform the operation. As the depth of the function is not known at compile time the correct amount of hardware cannot be built.
- Statements cannot cause side effects. This means that:
    - Local initialisations are not supported.
    - The initialisation and iteration phase of loops must be statements and not expressions.
    - Shortcut assignments such as +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=, ++ must be stand alone statements and not part of more complex expressions

There is also a limited standard library, but for the RC100 board there is a library which give assess to the peripherals. Handel-C also only supports integers either signed or unsigned, there is library functions provided to do fixed point operations, however compile times are faster if integers are used with the binary point dealt with by the programmer.[3]

Handel-c does optimisation on code when compiling. Since compiling is from software to hardware benefits can be gained both from traditional software compiler optimisation techniques and logic optimisation techniques. This is a multi-step process involving both technology independent optimisation and technology specific optimisation such as; dedicated multiplier circuitry, fast carry chains, etc. The first step is to compile to a high-level netlist before it is expanded to a technology specific lower-level netlist, this is illustrated in figure 1.1. [4]

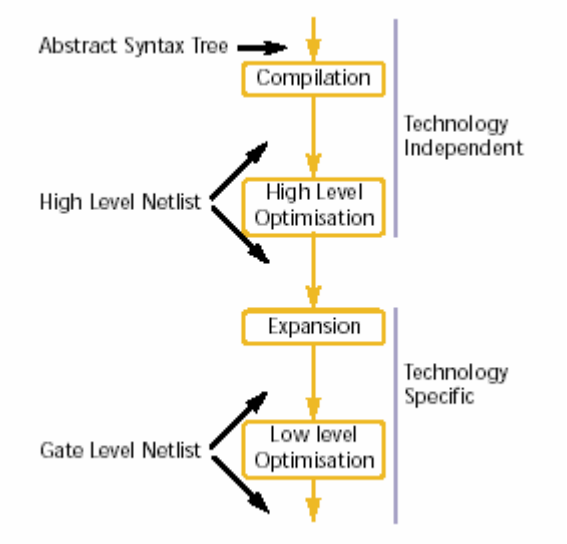

Figure 1.1Compaliton flow after parsing, taken from [4]

The compiler can perform a number of automated optimisations such as re-writing, conditional rewriting and common sub-expression elimination.

Re-writing involves changing the gate level netlists to logical equivalents. Logic that has one or more constant inputs can be simplified, and hardware that is executed but never connected to an external pin can be eliminated because it has on affects on the outputs.

Conditional re-writing is an extension of re-writing where test patterns are applied to gates to find impossible conditions, as there conditions cannot exist the compiler can remove any circuitry that contributes to these.

Common sub-expression elimination is a classic optimisation compilation technique where the same output is used by several expressions, rather than building two sets of hardware the output is routed to the two outputs.

Although the Handel-C complier can make a number of other optimisation techniques it will not do some optimisations. This includes no attempt to dynamically reuse hardware. This means that it is up to the user to explicitly declare hardware for reuse by making it a shared function. [4]

1.2.2 RC100 Development Board

The RC100 development board has a number of devices that can be controlled or accessed by the Spartan II FPGA. The FPGA is a 200,000 gate equivalent device with 14 block RAMs. The devices available are listed below

- 1 Flash RAM - for storing bitfiles to program the FPGA, 64Mbits in size
- 2 SSRAM - for storing user data, 36 bit with 256k locations
- 1 Video output system: Video DAC & VGA connector
- 1 Video decoding circuitry: SAA7111 decoder, s-video, composite video input
- 1 CPLD for communicating with the parallel port
- 7-Segment displays
- 3 LEDs
- 1 Main Clock Crystal
- PS/2 Ports for mouse and keyboard interface
- 1 Parallel Port - for connecting to the host computer

The FPGA device is the centrepiece of the board and it is the main piece of reconfigurable logic that users can target. The FPGA has direct connections to the; two SSRAM banks, Flash RAM, Video DAC, Video Input Decoder, PS/2 connectors, LEDs, two 7 segment displays and the expansion header. The FPGA also has access to the parallel port through the CPLD.

In my design I use the Video output system for displaying the corrected image to the screen, the composite input to capture images, the SSRAM for storing frames and one of the PS/2 ports for a keyboard that is use for adjusting variables. The parallel port is used to load the programs onto the FPGA. This process is done using the file transfer program provided with the DK1 suite, this program can be used to load programmes in to either the FPGA or the Flash RAM and can load images or any other data in the Flash RAM. By using the Flash it is possible to write code that can be loaded and run when the board is turned on.

1.3 Hardware Resources

Besides the limitation on the number of CLBs in the FPGA there are other constrains. The other main constraint is memory access. Only one off chip memory address can be written to or read from in a clock cycle. However as the RAM address are 36 bits in length two pixels can be stored in one address, which reduces the required number of memory access. The range of functions that can be implemented on the FPGA is limited unless buffering is used to allow access to more than two pixels at a time.

If the CLBs are configured into memory blocks they can all be access at the same time, the cost of this is reducing the amount of logic that can be implemented. When using Block RAM dual port memory can be constructed, allowing for one read and one write, or two read per clock cycle. This can increase the amount of on chip storage available without reducing the number CLBs available fro logic functions

The corrected image will be displayed directly to a VGA screen, meaning the output will be in a raster fashion. This affects how the algorithm is developed and adds the constraint that a pixel needs to be outputted for every clock cycle of the FPGA. The block diagram in figure 1.2 illustrates how the captured image data will flow through the system.

The FPGA must perform scan rate conversion because the input signal is composite PAL with a frame rate of 25Hz this must be processed and outputted to the VGA display at 60Hz. To perform this conversion two banks of single port off chip RAM on the RC100 board are used. The video decoder stream is deinterlaced and written to one RAM bank while two pixel values are read from the other bank and processed. When a frame has been written the RAM banks swap.

The distortion in the image is corrected for in the FPGA as it is outputted to the display. This approach was taken, as opposed to writing a corrected image into memory, due to video decoder library producing output in two-pixel chunks that are then stored into one memory location. The pixels could be broken up and correction applied but they might need to go to different memory address adding to the complexity of the memory write function and requiring pixels to be buffered before writing

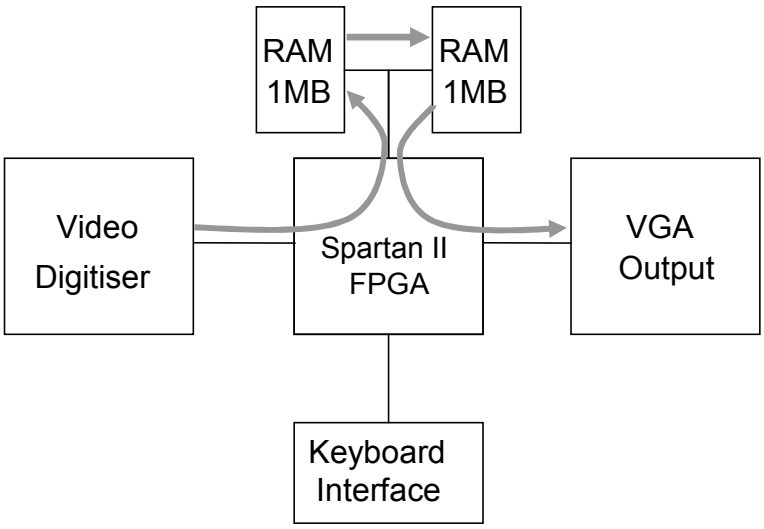

Figure 1.2 Block diagram of data flow

Due to the physical limits, placed on the number of logic elements, time needs to be spent on reducing the logic required for operations. Where possible logic functions that require a large number of gates have been avoided by the use of bit manipulation simplification and look up tables. This has included the avoiding of multiplications where possible. This has been done by the implementation of a lookup table in block RAM. Other reductions can be made such are converting multiplications by powers of twos to right shifts.

Due to the limited number of CLBs developing an algorithm for hardware is different to developing one for a software implementation. Although the underlying algorithm is the same, the way in which is structured is very different. In software to create speed improvements the code is optimised reduce the number of instructions. In programming for hardware implementation reducing the amount of logic to perform each operation is desired.

Due to lack of a shared arithmetic logic unit (ALU) used in microprocessors, algorithms can take advantage of the true parallel nature that comes about through hardware. These makes true concurrent processes possible and enables the programmer to build pipelines of any length to improve algorithm speed.

1.4 Number representation

When doing mathematical calculations on a PC normally the IEEE Standard 754 for floating-point arithmetic is used. This contains three components: a sign bit, a fraction field, and an exponent field, similar to scientific notation. There are single (32-bit) or double (64-bit) precision floating-point values.

For representation of sign in digital hardware it is usual to use twos complement numbers however in the IEEE standard the sign bit 0 denotes a positive number and 1 denotes a negative number. The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For double precision, the exponent field is 11 bits, and has a bias of 1023. The mantissa represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits. The implicit leading bit comes from the fact that floating-point numbers are stored in normalized form. This basically puts the binary point after the first non-zero digit. [7]

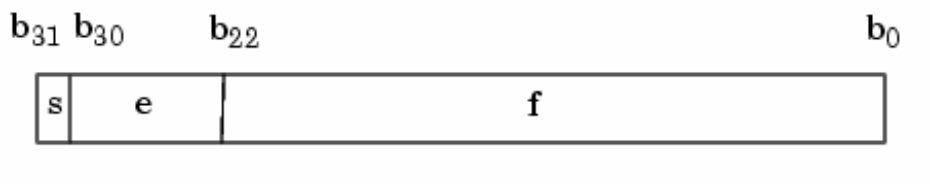

Figure 1.3 IEEE 754 Floating Point Number construction for 32 numbers

This standard can represent a large range of decimal numbers. However it is costly in terms of resources to implement due to the width bit length and in most DSPs and hardware-based logic this format is not used. Either a variable floating point notation where the exponent and fraction bit lengths are changed to suit the implementation ore fixed-point notation is used. In this the bits are treated as normal integers but there is an indication of were the binary point is placed within the number. This method has the advantage that normal arithmetic operations can be used the bits need to be shifted to be aligned to the

right position before calculations are done. The main disadvantage of fixed-point is that it has a fixed window so cannot represent very large or very small numbers.

Due to the complications and large number of logic gates needed to perform floating-point operations in hardware, a fixed-point representation was chosen. Originally the Handel-C fixed-point library was used for performing arithmetic operations, however When using only two fixed-point operands and one fixed-point function from the Handel-C library the compile time was in the terms of hours rather than minutes, this made an incremental approach to development impossible. Therefore all variables were represented as signed or unsigned integers and were commented to indicate the position of the binary point. When arithmetic operations were performed operands were shifted to ensure alignment. Treating each fixed point number as a simple integer and using comments to specify what shifting was required though harder to debug was a lot quicker to compile.

By using fixed-point numbers in Handel-C the bit length of each step can be defined to be different. This gives a great deal of flexibility in design and can save on the amount hardware used for registered outputs, as smaller registers can be constructed. The bit lengths used will be discussed in the implementation section.

1.5 Barrel distortion

Image processing is often used to make non-contact measurements for real-time measurement applications. It is therefore vital to ensure that accurate measurements can be made from the captured images. If an analogue camera is used it must often be of greater resolution and quality than is needed for the particular application in order to compensate for losses incurred before digitisation of the image [8]. A digital camera facilitates early digitisation and therefore it is less crucial to compensate for these losses. This can lead to substantial cost savings since a digital camera with lower resolution can be used. However, the inexpensive and wide-angle lenses often used in low cost digital cameras are susceptible to barrel distortion, which can introduce significant errors into any measurements [5].

Barrel distortion occurs when the magnification at the centre of the lens is greater than at the edges. It is possible to use a higher quality multi-element lens system to correct for this but this comes at considerable additional cost to the image capture system. As this project is aimed at low cost consumer products such as web cameras and PXT phones this cost cannot be justified. Barrel distortion is primarily radial in nature, a relatively simple one parameter model can account for most of the distortion [6]. A cost effective alternative to using an expensive lens is to algorithmically correct for the distortion using the model. Barrel distortion is illustrated in figure 1.4 below.

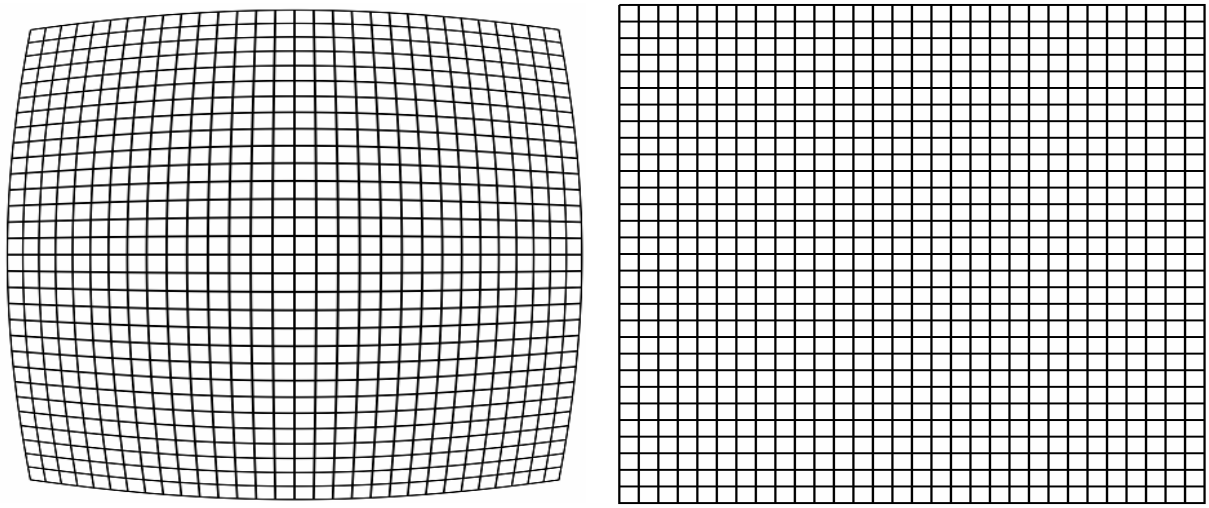

Figure 1.4 Distorted captured image on the left and desired image on the right.

2 Implementation

2.1 Barrel Distortion Correction

Barrel Distortion as described in the introduction is caused by a larger magnification at the centre of the lens than at the edges. The relationship between the distorted and undistorted image coordinates may be modelled by

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

where r_u and r_d are the radial distances from the centre of the image as shown in figure 2.1 and k parameterises the distortion. For barrel distortion a positive value of k accounts for the decrease in magnification with increasing radius from the centre of the image.

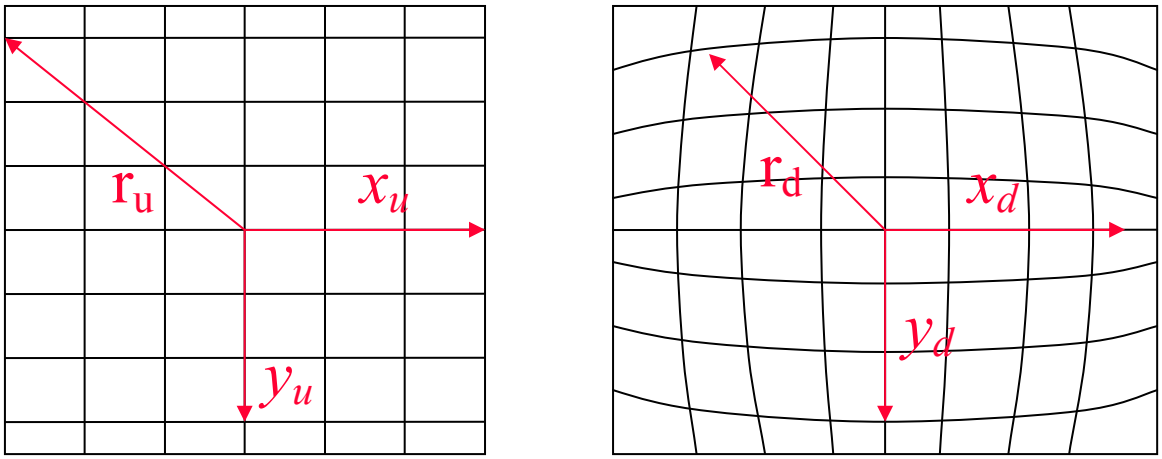

Figure 2.1 Undistorted and Distorted image coordinate systems

This model only corrects for radial distortion and does not correct for any other distortion caused by the lens system, such as perspective distortion.

The problem with this is that the correction is in terms of the distorted image. The position of the pixel in the distorted image is used to calculate were the pixel should be stored in a corrected image. This form is unsuitable for real-time correction because it is necessary to produce the undistorted output pixels in raster order. The coordinates in the undistorted image must be used to determine which pixel in the distorted image should be displayed. Instead it is desirable to have the equation in the form of

$$\begin{aligned} x_d &= x_u M(k, r_u^2) \\ y_d &= y_u M(k, r_u^2) \end{aligned} \qquad (2)$$

Where $M(k,r_u^2)$ is a magnification factor that depends on the distortion, and position in the image. The reason for requiring r_u^2 rather than r_u is that as $r_u = \sqrt{x_u^2 + y_u^2}$ and the square root function that would be required uses a large number of resources. Equation (1) can be rearranged to give

$$M = \frac{r_d}{r_u} = \frac{1}{1 + kr_d^2} \qquad (3)$$

However this has the magnification depending on r_d^2 rather than r_u^2 by substituting $r_d = r_u M$, $M(k,r_u^2)$ is given by

$$M = \frac{1}{1 + kM^2 r_u^2} \qquad (4)$$

This can be solved iteratively by first setting M to 1, and substituted into equation (4). This gives a revised value of $M(kr_u^2)$, which is again substituted into the equation. This is iterated until $M(kr_u^2)$ converges to the desired precision. It can be shown that the equation does converge for values within the range required (Appendix A). Figure 2.2 shows the resultant magnification function. As the mapping depends on the product kr_u^2, this avoids having to have a separate mapping for each k. Having a single mapping allows $M(kr_u^2)$ to be precalculated and stored in a lookup table. The MATLAB code for calculating M is given in appendix B.

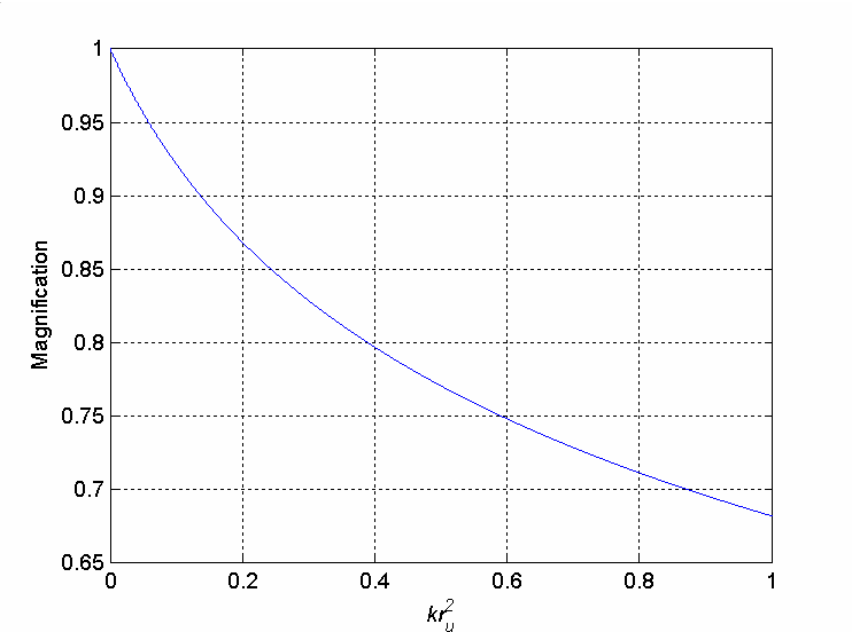

Figure 2.2: Magnification factor from pixel radius

The coordinate system that is used of the image differs from a normal one. First the origin of the image is at the centre of distortion. This can be approximated to be the centre of the image. As the image is has its origin at the centre the magnification of the four quadrants will be the same. The image is then scaled to be a fixed width, by making r_u^2 can be zero and one within a quadrant. As the correction factor k is also between zero and one, kr_u^2 will also be between zero and one. By doing this normalisation the required magnification does not need to be recalculated when the image size changes. This is normalisation is archived by shifting the x^2 and y^2 bits so that they are fixed point numbers less than one.

By having kr_u^2 between zero and one the magnification table does not need to be changed for different sized images, this allows the precalculation to be preformed.

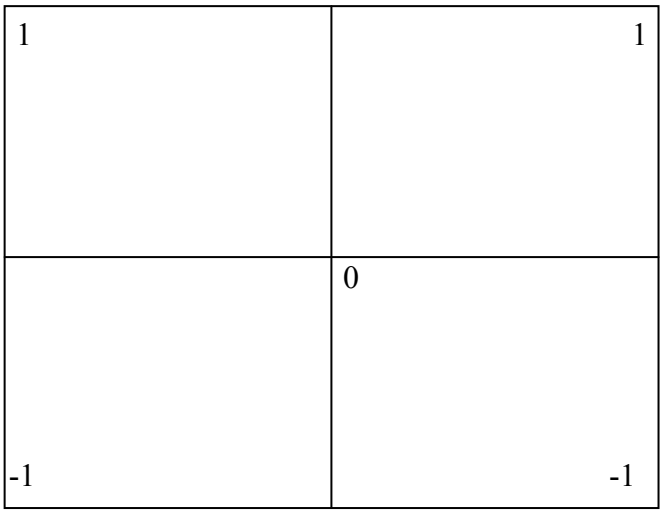

Figure 2.3 Normalised image coordinates in terms of r_u

2.1.1 Algorithm

The algorithm can be broken up in to a number of subsections. The system is driven entirely from the present scan position of the display. Distortion correction is performed by using the current scan position and a magnification factor held in a look-up table to calculate the address of the corresponding distorted pixel that is located in video RAM. This pixel is read from the RAM and displayed to the screen. Figure 2.4 show a system diagram of how the correction algorithm interacts with the components.

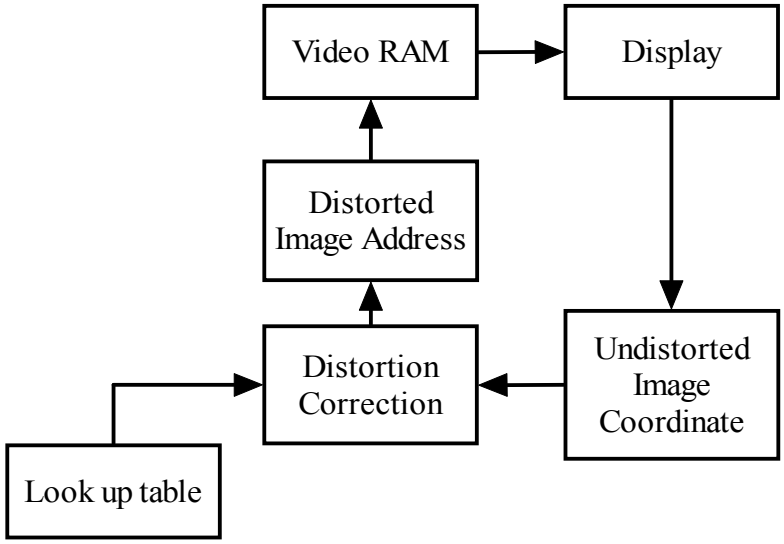

Figure 2.4 System diagram

The details of how the correction algorithm is implemented are given in the following sections. It describes how the correction is calculated and the step required to design a system that can run in real time.

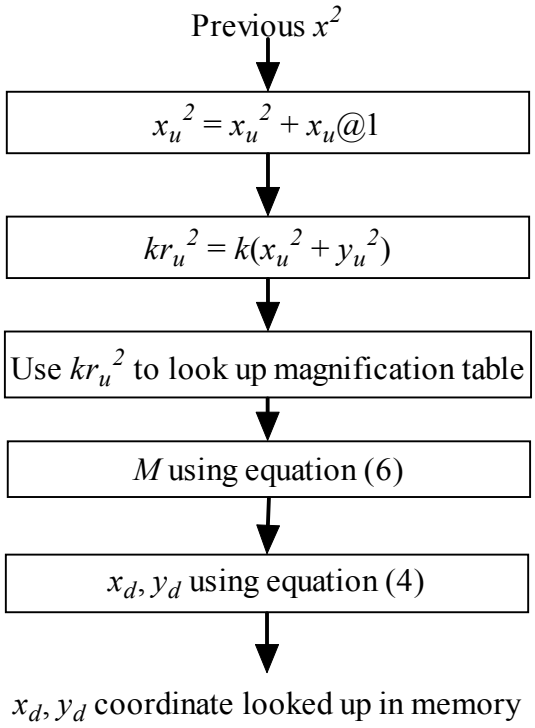

Figure 2.5 Barrel Correction Algorithm

Figure 2.5 shows the algorithm that does the correction, this works on one pixel at a time. Only x^2 is calculated in this section, as the output is raster based y^2 only needs to be calculated once per line. The next step is to calculate kr_u^2, this is used to look up the magnification is the look up table. The actual magnifications then estimated. The magnification is multiplied by the x and y coordinates and this is used as the address for the pixel to be displayed to the screen. These steps will be explained in more detail in the following sections.

In the design there are several functions running in parallel. These are the keyboard interface, the video capture and the display sections. The ability to program parallel functions can increase the flexibility of the system but there needs to be communication between cooperating hardware blocks. In my case the video capture and the display sections were run in parallel but they communicated via a register to identify, which memory block is being written to and which was being read from. The keyboard interface is used to allow the user to change the correction factor *k*, this is done via a register and will update the correction factor used in the algorithm for its next iteration. Figure 2.6 shows how the three parallel processes communicate with each other.

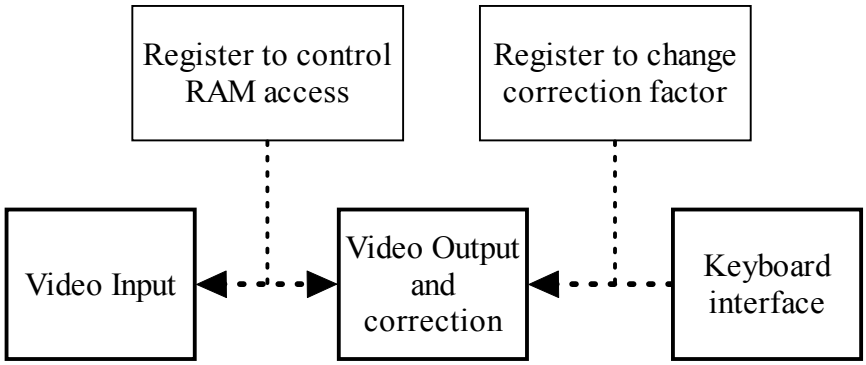

Figure 2.6 Parallel processes communication

2.1.2 Screen Coordinate Calculation

Using the RC100 development board the corrected image is to be displayed to a VGA screen in real time. The output is therefore required to be a raster scan. Therefore each row in the image will have a constant y and as this is constant so will y^2. Due to this y^2 only needs to be calculated one per row. This calculation can be done during the horizontal blanking period.

As we scan across the screen x increments by one for each pixel in the output. This can be used to enable us to calculate x^2 incrementally making use of the expansion:

$$(x+1)^2 = x^2 + 2x + 1 \qquad (5)$$

therefore x^2 only needs to be calculated at the start of each row, and after that the incremental formulate (5) can be used to calculate x^2. This can be further reduced in terms of hardware logic by making use of the fact that multiplying by 2 is equitant to a left shift by 1 bit position. As one is then to be added shifting with a 1 can be done, which is taking x and appending 1 as the lowest bit can reduce the logic further. Substituting x@1, x with one appended to the end, for 2x+1 in equation 5:

$$(x+1)^2 = x^2 + x@1 \qquad (6)$$

This equation reduces both the logic depth of the equation and the required hardware.

As y^2 and x^2 are never calculated at the same time the hardware for this can be share, with the cost of the multiplexes required to do this. It was found that doing this had no effect on the number of gates that Handel-C built.

The calculated x^2 and y^2 are then used to give r_u^2 which is calculated by

$$r_u^2 = x^2 + y^2 \qquad (7)$$

this must then be multiplied by k to produce kr_u^2.

As the image size is 503 by 480 pixels (due to the FPGA clock not being able to be divided to the correct frequency for 512 by 460) only 9 bits are required for x and y position. This in turn means that 18 bits are required for x^2 and y^2.

2.1.3 Magnification Calculation

As described earlier the magnification is found using a look up table, the first step in doing this is to find the present x^2 and y^2. These are summed and multiplied by the correction factor k (set by the user using the keyboard) to produce kr_u^2. To give adequate resolution for the required correction 10 bits are required for k. kr_u^2 can then be used to find the value in the look up table. The main problem with this is that the look up table can only hold a limited number of magnification values. By using only one block RAM 256 16-bit entries can be stored, and the top 8 bits of kr_u^2 can be used to address these. This will only give 8 bits of resolution, which can be improved by applying an appropriate offset; figure 2.7 graphs the error in magnification against kr_u^2. Using all the available block RAM to create a 2304 entry look up table does not significantly increase the resolution. This can be improved by interpolation. A number of instructions must be implemented to find the magnification and then this must be used to calculate the pixel to be displayed. To achieve an output of one pixel per clock cycle a pipeline needs to be used. Both interpolation and pipelining are now discussed.

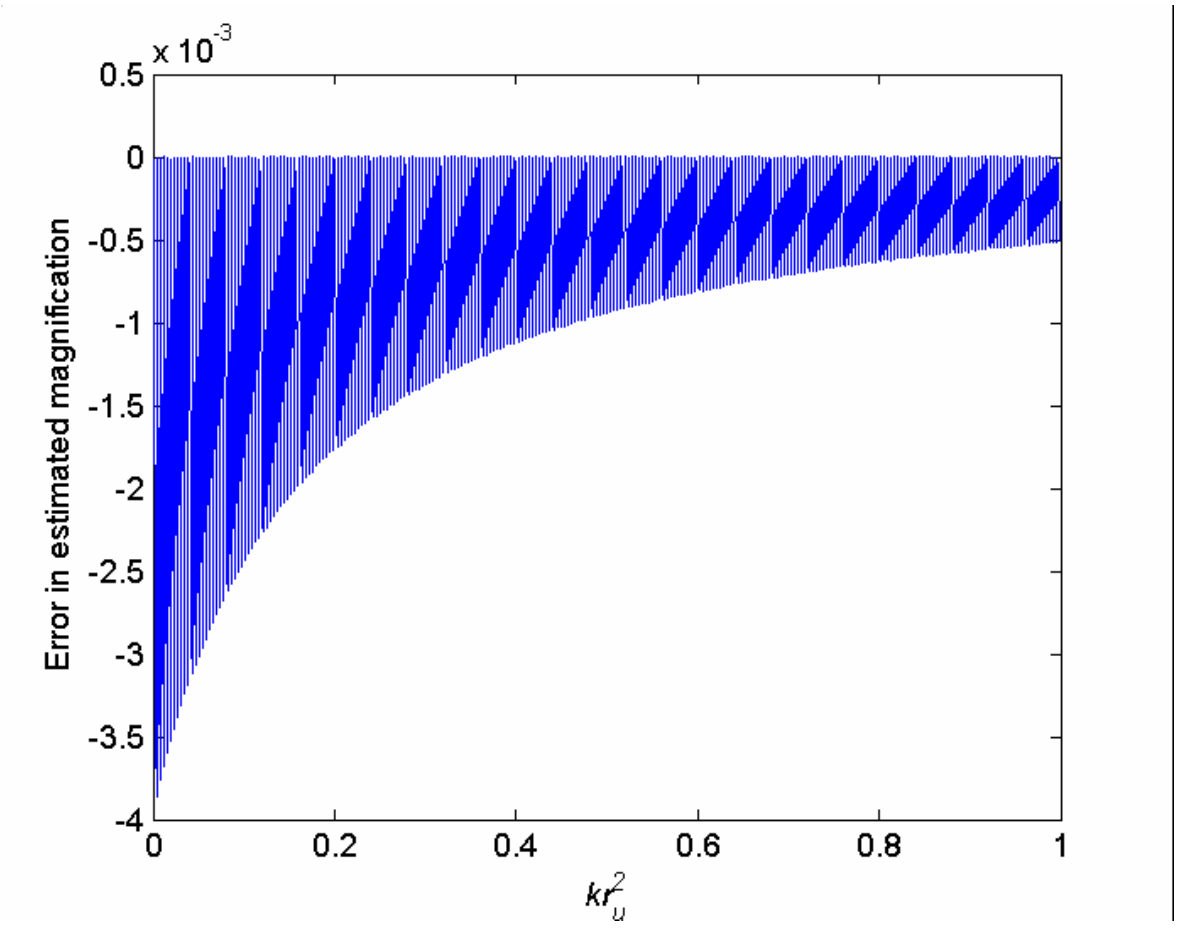

Figure 2.7 Look up table magnification error

To improve the resolution, several steps can be taken. The first is to notice that all the magnification values in the area of interest are between ½ and one. This means that the 16 bits available in the RAM can be used to represent the bottom 16 bits of a 17-bit number with the top bit of 1 being appended before any calculation. This does not improve the resolution of the value significantly because there are only 256 entries. This means that only the top 8 bits of kr_u^2 can be used. To improve the accuracy, linear interpolation between adjacent samples may be used. This is effective because the magnification function is smooth. To perform the interpolation, both the required magnification table entry and the next entry are looked up. The slope of the line between the two values is calculated and the lower bits of kr_u^2 can then be used to estimate the magnification at the intermediate point:

$$M(kr_u^2) \approx M([kr_u^2]_{MSB}) + (M([kr_u^2]_{MSB} + 1) - M([kr_u^2]_{MSB})) \times [kr_u^2]_{LSB} \times 2^{-8} \qquad (8)$$

and illustrated in figure 2.8.

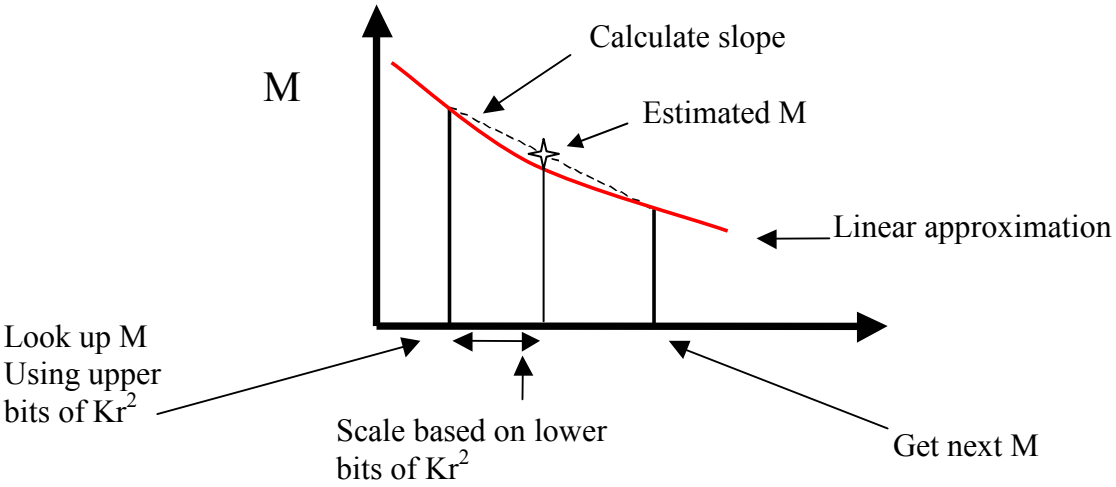

Figure 2.8 Magnification using interpolation

When the 17-bit magnification value is combined with interpolation, a resolution of 15.5 bits can be achieved for a 16-bit precision of kr_u^2. This is calculated by finding the maximum error between the real magnification value and the estimated, converting it in to the equivalent number of bits require to represent the error. A plot of the error against kr_u^2 is given in figure 2.9.

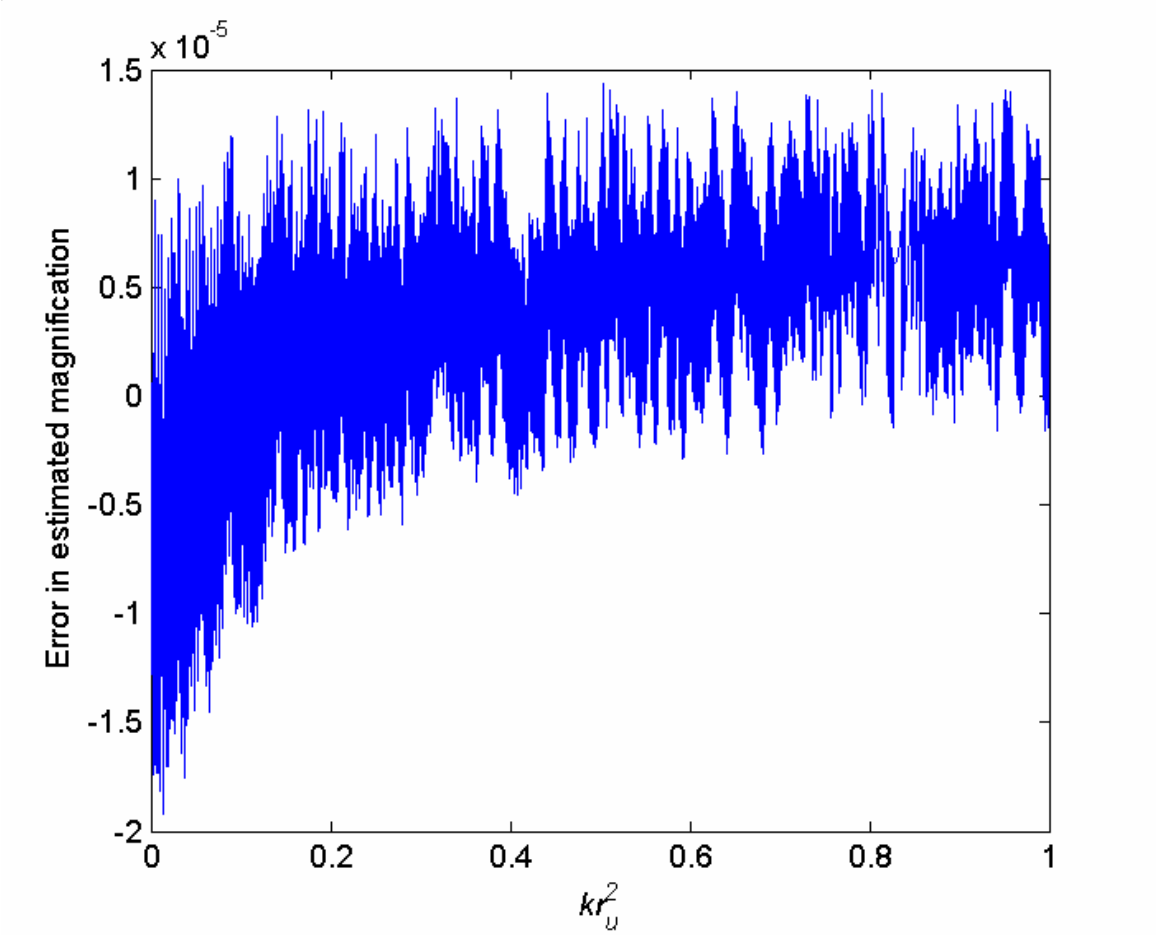

Figure 2.9 Error in magnification using interpolation

We then need to use equation (2) to determine coordinates in the distorted image. These are used as the address into RAM to read the pixel value to be displayed.

2.1.4 Coordinate Truncation

When calculating the coordinates for the pixel location to be displayed the result is seldom an integer number. The corrected *x* and *y* are 10 and 9 bits in length. This is due to the top 9 bits of *x* and the 9 bits of *y* giving the address in memory, which contains two pixels; the bottom bit of *x* selects which of these pixels is displayed.

The approach that has been taken for this implementation is to truncate, so that the fractional component is discarded. Truncation or the alternative rounding can introduce substantial error in pixel location. This intern distorts lines in the image by producing jagged edges. Figure 2.10 shows the effect of the use of truncation, the circles represent actual pixels, in (a) there is only a small error, however in (b) the pixel is a lot closer to the lower right pixel then the resulting top left pixel.

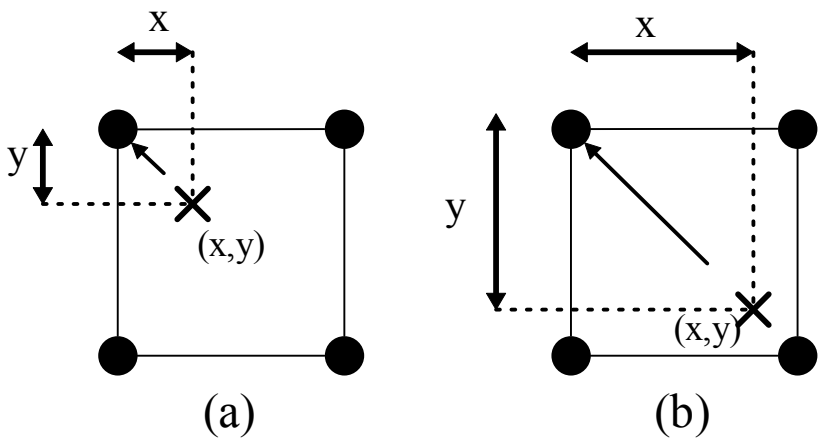

Figure 2.10 Effect of truncation

2.1.5 Pipelining

Due to the constraint of real time processing and the need to produce one pixel for each clock cycle, a pipeline must be constructed. This works by having all the required operations running in parallel each one using the result from the previous clock cycle to perform its operations on. Once the pipeline has been primed then one output is generated every clock cycle. While the pipeline is being primed, the output will not be valid. The length of the pipe depends on the number of operations that are required to run in parallel.

As y^2 is constant for a line it can be calculated in the vertical blanking period, x^2 is calculated using the previous x^2 and the fact that it is changing in a raster fashion. kr_u^2 can then be calculated and this used to find the magnification interpolation is done and the address of the required pixel is found. This process will have a delay of five clock cycles before there is output. This requires that data must be fed into the pipeline five clock cycles before the output is required. Starting the process 5 clock cycles before the end of the horizontal blanking period achieves this. Figure 2.11 shows the algorithm written for a parallel approach. The bold boxes indicate registers and there names. In this each operation will occur on the registered output from the operation that occurred in the previous clock cycle. As x will have incremented by four by the time it is used to calculate the required pixel address either four needs to be subtracted from it, or the original x needs to also be delayed with four registers. The subtraction was the chosen approach as the registers use more resources than a subtraction.

The notations in figure 2.8 on the connecting arrows indicate the data widths of the operands, whether it is signed and were the binary point is. As described earlier by using a variable bit length approach the amount of hardware needed can be reduced. This can be achieved by truncating the results of operations before they are registered. The length of each operand depends on the level of precision required.

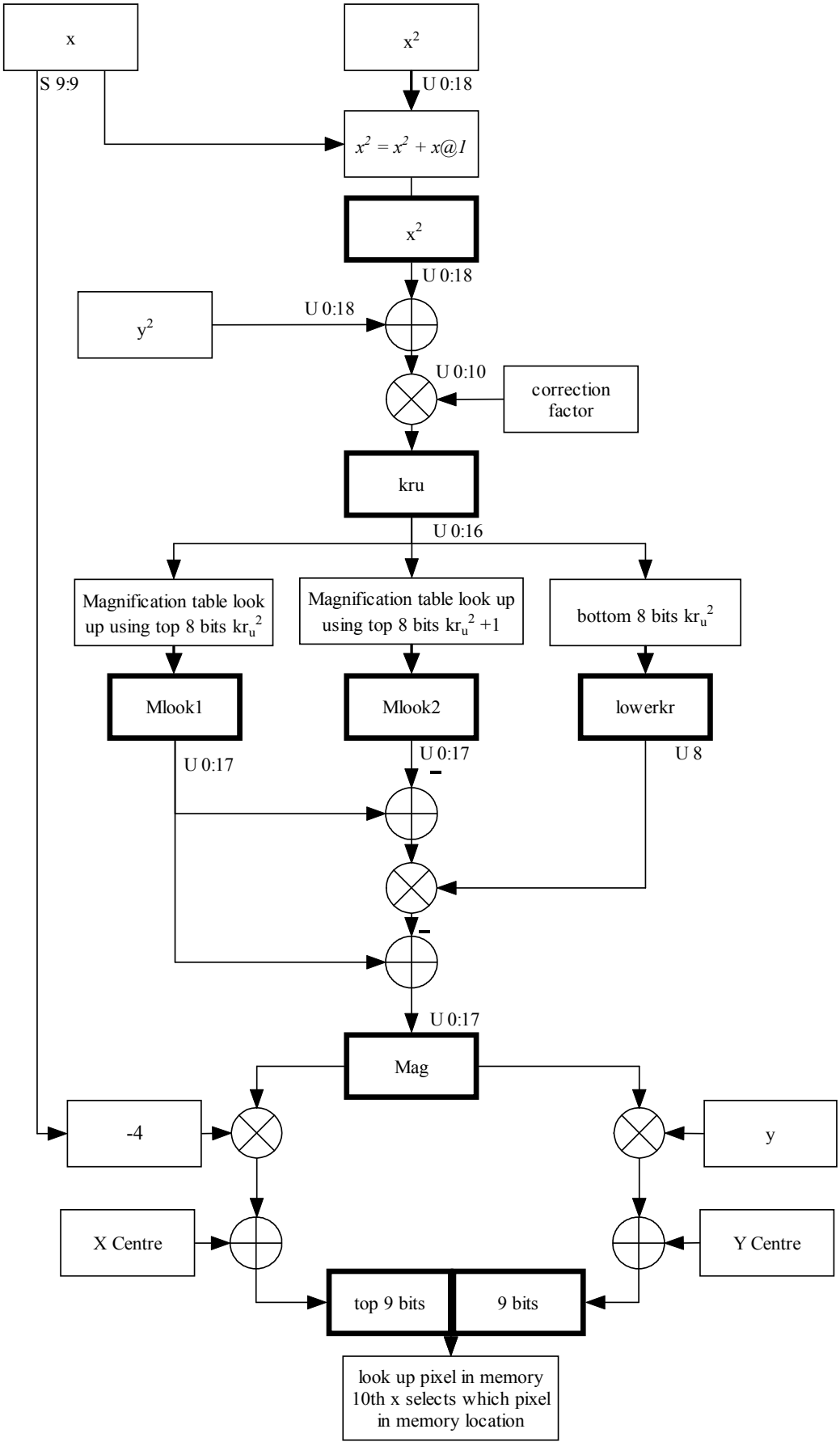

Figure 2.11: Pipeline for coordinate calculation

3 Results

The distortion correction algorithm has been successfully implemented and tested on the RC100 development board, figure 3.1 shows a distorted image and figure 3.2 the corrected image. The utilisation of the FPGA used on the RC100 development board is shown in table 3.1.

Table 3.1: Resource utilisation of device (XC2S200)

	CLBS (1172 total)	**Block RAM (14 total)**
Keyboard interface	129 (11%)	1
Video decoder / VGA	235 (20%)	4
Correction algorithm	270 (23%)	1
Total	642 (54%)	6 (42%)

The correction algorithm uses 23% of the logic resources of the FPGA and one of the block RAM for the look up table. The majority of the logic resources are used to implement the large multipliers used. If this were implemented on an FPGA such as the Virtex-II that incorporates embedded multipliers the resource utilisation due to the multipliers would be significantly reduced.

The other resources used are not directly part of the correction algorithm but are used to support it. The keyboard interface is only used to allow the correction factor to be changed so if a fixed lens configuration was to be used this could be removed, saving 11% of the resources. The Video decoder is required for capture and storing to memory. The VGA section is required to control the display.

Although it cannot be seen clearly in figure 3.1b there are some jagged edges in the image, this is cased by the truncation processes and contouring caused by a change from one magnification value to the next. The algorithm used was also implemented in MATLAB and produced similar results.

Figure 3.1. Distorted Image

Figure 3.2. Corrected Image

4 Future work

It can be seen form the resource utilisation table that almost half of the FPGA is not utilised. This space can be used to implement bilinear interpolation between pixels to remove the effects of coordinate truncation, and improve the quality of the corrected image. This requires the used of row buffering to allow a number of pixels to be accessed in one clock cycle and is discussed in [7].

An investigation in to whether a CMOS optical sensor could be used as the memory element for the distorted image need to be done. If this can be successfully implemented an FPGA could be placed between the sensor element and a memory bank with the barrel distortion being corrected for before the image is stored. This would then enable other image processing functions to be done on the corrected image by either another FPGA or a microprocessor. If this is feasible then for the application that this is aimed at, low cost web and PXT cameras the presented algorithm with only minor changes could be implemented on an FPGA that could be incorporated into the device to ensure corrected images before they were stored in memory.

As the barrel distortion has been corrected other image processing operations could be investigated, due to the probable need for line buffering the size of the present Spartan-II chip on the RC100 board is likely to limit the size and type of operations that can be implemented.

Handel-C tries to enable hardware to be programmed like software, however due to the nature of hardware design a software approach is almost impossible. Although a C program may be written to implement an image processing algorithm it cannot easily be converted to a Handel-C program when real time operation is required. Most of the effort involved in this project was in designing the real time hardware required, such as pipeline design. This means that at present an image processing expert with limited hardware knowledge would have difficulty in developing FPGA systems. Work needs to be done on making the process of converting from a software design to hardware a less complex task.

5 Conclusions

The barrel correction algorithm has been successfully implemented on a FPGA using the RC100 development board and Handel-C.

To allow the algorithm to be preformed buffering of the image is required, for simplicity a whole frame is buffered in to RAM

The conversion from a software algorithm to one that runs in hardware in real-time presents a number of difficulties. The main one is the inability to do offline processing but this is inherent in all real time applications. Due to the construction of the RC100 board only one off chip memory access is possible per clock cycle (per RAM), this means that if any complex pixel manipulation is to be done buffering of the image is required. In my algorithm this could be avoid as only the address of the pixel is calculated then retrieved and directly displayed.

The real-time requirement makes the construction of a pipeline architecture essential, reconfigurable hardware allows this to be customised to the desired length and complexity. For the hardware to be able to be mapped to the target device there is a need to minimise logic gate count.

The use of a look-up table with interpolation can reduce the complexity of the hardware design without significant loss of precision compared to calculating values with hardware at run-time.

Even though Handel-C was designed to raise the level of abstraction for hardware design and shift the focus to algorithmic design, it has been beneficial to maintain a data flow approach at the register transfer level. The data flow approach has many advantages when it comes to the design of pipelines as it makes the breaking up of an algorithm into separate stages of a pipeline conceptually simpler by using registers.

Appendix A: Convergence of Magnification

The magnification factor is calculated iteratively. We need to show that this iteration converges. The condition for the iteration to converge is that the slope of the line for M must be between –1 and 1. This is illustrated graphically in figure 1, case (a) is when the slope of the line is positive and less than one and case (b) is when the slope of the line is less than negative one:

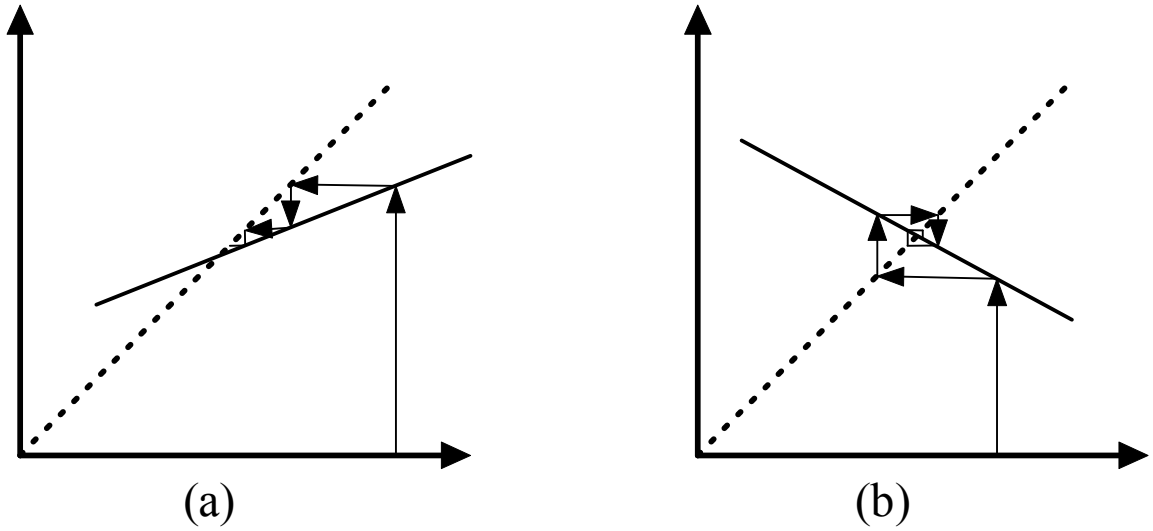

Figure 1.1 Graphical Convergence

The magnification required to corrected for barrel distortion can be shown to be

$$M = \frac{1}{1 + M^2 k r_u^2} \qquad \text{(a1)}$$

where k is the required correction and r_u^2 is the radial distance from the centre of the image.

The slope can be found by differentiating with respect to M:

$$\frac{d\dfrac{1}{1 + M^2 k r_u^2}}{dM} = \frac{d\left(1 + M^2 k r_u^2\right)^{-1}}{dM} = \frac{-2Mkr_u^2}{\left(1 + M^2 k r_u^2\right)^2} \qquad \text{(a2)}$$

Equation (a1) can be substituted into (a2) resulting in

$$\frac{d\dfrac{1}{1+M^2kr_u^2}}{dM} = -2M^3kr_u^2 \qquad \text{(a3)}$$

For convergence $-1 < -2M^3kr_u^2 < 1$ rearranging (a1) gives

$$M(M^2kr_u^2 + 1) = 1$$

$$M^3kr_u^2 + M = 1$$

$$\therefore M^3kr_u^2 = 1 - M$$

Therefore, for convergence

$$-1 < -2(1 - M) < 1$$

$$\therefore -\frac{1}{2} < M - 1 < \frac{1}{2}$$

$$\therefore \frac{1}{2} < M < \frac{3}{2}$$

Since $kr_u^2 = \dfrac{1-M}{M^3}$, the iteration will converge for $-\dfrac{4}{27} < kr_u^2 < 4$ provided the starting point is sufficiently close to the final value.

This will be satisfied if the slope is also between –1 and 1 at the starting point
The iteration is always stated at M =1. Substituting this into equation (a2) gives

$$-1 < \frac{-2kr_u^2}{\left(1+kr_u^2\right)^2} < 1 \qquad \text{(a4)}$$

This is plotted in figure 1.2 between –4/27 and 4,showing that this condition is meet.

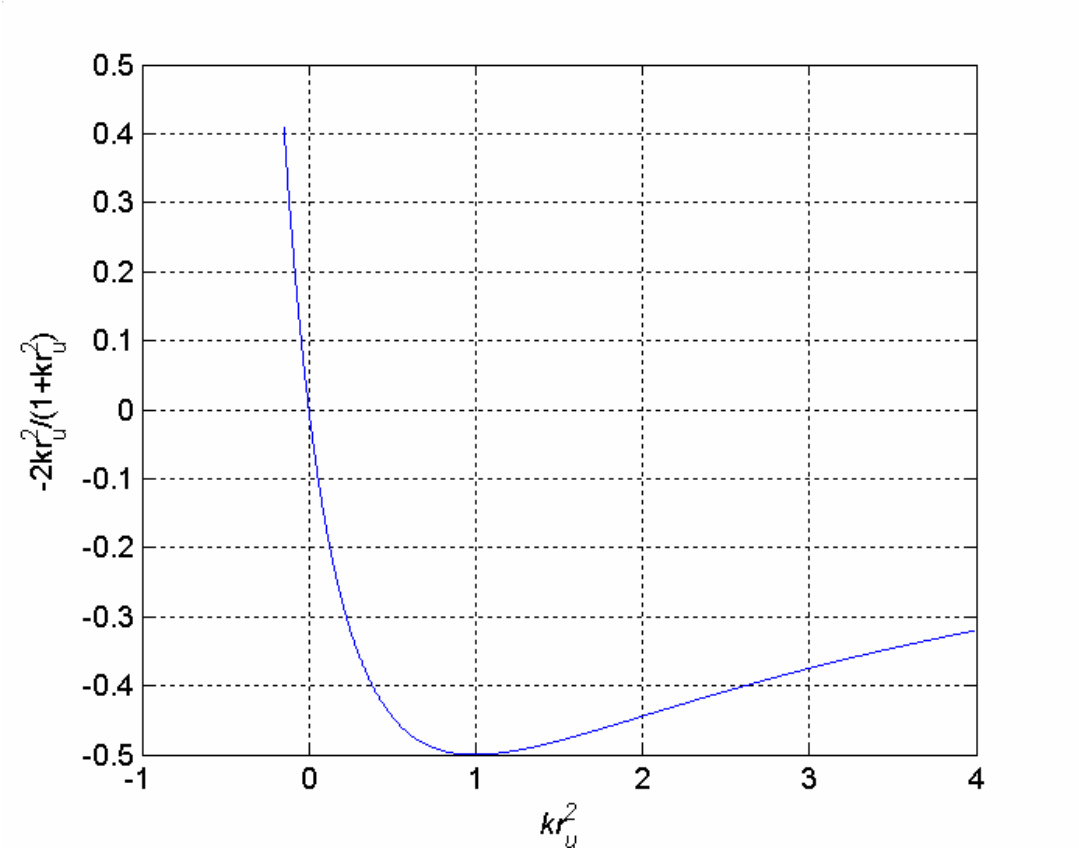

Figure 1.2 Convergence of M

The full condition for graphical converges to be meet also requires that the function is continuous, which it is, and that the slope for all points visited is between –1 and 1, not just the start and finish points. This means that there should be no turning points, the second derivative can be found for (a1)

$$\frac{d-2M^3kr_u^2}{dm} = -6M^2kr_u^2 \qquad \text{(a5)}$$

Since $kr_u^2 = \frac{1-M}{M^3}$, equation (a5) becomes

$$\frac{d-2M^3kr_u^2}{dm} = -6M^2kr_u^2 = \frac{6(M-1)}{M} \qquad \text{(a6)}$$

which means that there is a turning point at M=1 or $kr_u^2 = 0$, for M <1 it has a negative concavity meaning that for $0 < kr_u^2 < 1$, our area of interest there are no turning points indicating that the slope of all visited points are between –1 and 1.

Appendix B: MATLAB Magnification Table

```
clc
close all
k =1;
m =[];
rd2 =[];
space = 1/256; %set steps to 256

ru2 =[0:space:1]; %getting ru from 0 to 1 with 256 steps

m= 1./(1 + k.*ru2) ; %calculate m

rd2 = m.^2 .* ru2;
%rd2old =rd2;
i =1;
%always run if first interation
diff =2^16;
%run until the min and max differece of old to newless than 2^-20
while (max(diff) >=2^-20)
   m = 1./(1 + k*rd2);
   rd2old = rd2;
   rd2 = m.^2 .* ru2;
   diff = abs(rd2 -rd2old);
   i = i + 1;
end
x =i;
ru2;
m;
rd2 ;
mcoder =floor(m*2^17);
ms = floor(m*2^17)-2^16;
```

Appendix C: MATLAB Magnification Error Code

```
clc
close all
k =1;
m =[];
rd2 =[];
space = 1/256; %set steps to 256

ru2 =[0:space:1]; %getting ru from 0 to 1 with 256 steps

m= 1./(1 + k.*ru2) ; %calculate m

rd2 = m.^2 .* ru2;
%rd2old =rd2;
i =1;
%always run if first interation
diff =2^16;
%run until the min and max differece of old to newless than 2^-20
while (max(diff) >=2^-20)
   m = 1./(1 + k*rd2);
   rd2old = rd2;
   rd2 = m.^2 .* ru2;
   diff = abs(rd2 -rd2old);
   i = i + 1;
end
x =i;
ru2;
m;
rd2 ;

ms = floor(m*2^17);

 rd2b = [0:1/2^16:1];
 ru1 = sqrt(rd2b) .* (1 + rd2b);
 ru = ru1(find(ru1<=1));
 ru2b = floor(ru.^2*2^8);
 ru2s = floor((ru.^2 * 2^8 - ru2b)*2^8);
 mreal = (sqrt(rd2b)./ru1);
 mreal2 = mreal(1:length(ru));

 mlook = ms(ru2b+1);
 mdiff = ms(ru2b+1) -ms(ru2b+2);
 mtimes = floor(mdiff .* ru2s)/2^8;
 minter = floor(mlook - mtimes);
 diff = mreal2 - minter/2^17;
 nondiff = (mreal2-mlook/2^17);

  ms2 = floor(m*2^17) -2^16;
 mcoder = floor(m*2^17);
 mr = mreal2;%*2^17;

 error = (mr - mlook/2^17);

 error1 = (mr - minter/2^17);

 sumerror1=0;
 sumerror=0;
 i=1;
```

```
 while i<length(error)
    i=i+1;
    sumerror = sumerror + abs(error(i));
    sumerror1 = sumerror1 + abs(error1(i));
 end
 maxer = max(abs(error))
totale = sumerror
avge = sumerror/length(error)
avge1 = sumerror1/length(error1)
maxer1 = max(abs(error1))
totale1 = sumerror1

LUTbits_resolution = 17 - log2(maxer*2^17)
Interplatebits_resolution = 17 - log2(maxer1*2^17)

figure(1)
axes('FontSize', 12)
plot(ru.^2,error)
ylabel('Error in estimated magnification','FontSize', 12)
xlabel('kr_u^2','FontSize', 12,'FontAngle','italic' )

figure(2)
axes('FontSize', 12)
plot(ru.^2,error1)
ylabel('Error in estimated magnification','FontSize', 12)
xlabel('kr_u^2','FontSize', 12,'FontAngle','italic' )
```

Appendix D: MATLAB Simulation of Implementation

```
% Magnification Table must be run first to get ncoder array
I=imread('WARPED.BMP');% disbuild.bmp');%
imshow(I)
[xmax,ymax]=size(I);
xa=[];
B=[];
k =1.28
y =-1*(ymax/2)
xc = xmax/2;
yc = ymax/2;
for yi = 1 :ymax
   x = -1*(xmax/2);
     y2 = y^2;
   y = y +1;
   for xi = 1 :xmax
      x2 = x^2;                    %pipe 1
      x =x+1;
      kr2 = (k*(x2 +y2)/2^12);             %pipe 2
      krtop8 = floor(kr2/2^8);
      krbot = (kr2 -krtop8*2^8);

      %pipe 3
      if krtop8+1 >length(mcoder)
         m1 =89434;
      else
         m1 =mcoder(krtop8+1);
      end
           if krtop8+2 >length(mcoder)
         m2 =89334;
      else
         m2 = mcoder(krtop8+2);
      end

      %pipe 4
      m = (m1 - ((m1 -m2)*krbot))/2^17;

     %pipe 5
      xf = floor((m*x) + xc);
      yf = floor((m*y) + yc);

      %check if out of array index
      if xf >xmax
         xf =xmax;
      end
      if xf <1
         xf =1;
      end
      if yf <1
         yf =1;
      end
      if yf>ymax
         yf=ymax;
      end
      B(xi,yi) =I(xf,yf);
   end
end

% make array 8 bit
X8 = uint8(round(B - 1));
figure(2)
imshow(X8)
```

Appendix E: Handel-C Implementation Code

Main Code

```
/* Set the clock divisor and board type
   Clock divide of 4 is 503 by 480 display  use 251
   Clock divide by 3 is 670 by 480 */
#define RC100_CLOCK_DIVIDE 4
#define RC100_BOARD

/* Include the relevant headers */
#include <RC100.h>
#include <stdlib.h>
#include "ASCIIChars.hch"
#include "tables.hch"

//define the centre of the image distortion
#define xc 251
#define yc 240

/* Function Prototypes */
macro proc WriteBackBuffer( LineCount, PixelCount, Value, ActiveRAM);
macro proc DrawVideoOutputToRAM(Decoder, ActiveRAM);
macro proc Display(Video, ActiveRAM, Keyboard);

//uses fact that (x+1)^2 = x^2 + 2x +1 = x^2 + x<<1 +1
macro expr squred( sa,a) = (unsigned)((signed)sa + (adjs(a,width(sa)-1)@(signed 1) 1));
macro expr myadd(a1,a2) = a1 + a2;

void main(void)
{
        // video stuff
        SAA7111_DECODER Decoder;
        RC100_VGA_DRIVER Video;

        RC100_PS2_KEYBOARD Keyboard;
```

```
	unsigned 1 ActiveRAM;		/* Flag to indicate which RAM bank is the front buffer*/

	do
	{
		par
		{
			RC100PS2KeyboardDriver(&Keyboard, RC100_KEYBOARD_PORT,

	(RC100_K_ASCII_CODES|RC100_K_TRAP_KEY_RELEASE|RC100_K_USE_LEDS));

			RC100VideoDriver(&Video);
			RC100VideoDecoder(&Decoder);
			DrawVideoOutputToRAM(&Decoder, &ActiveRAM);
			Display(&Video, &ActiveRAM, &Keyboard);
		}
	}while(1);
}

/* Write the Video camera output to the back buffer*/
macro proc DrawVideoOutputToRAM(Decoder, ActiveRAM)
{
	unsigned 10 LineCount, PixelCount;
	unsigned 33 Value;


	macro expr Command(x) = x[32];
	macro expr Token(x) = x[31:30];
	macro expr Location = ((LineCount)<-9)@(PixelCount\\1);

	par
	{
		do
		{
			Decoder->VideoOutput ? Value;
			if(!Command(Value))
			{
				/* Write it to the correct RAM and update the pixel counter*/
				par
```

```
		{
			if (LineCount[9])
			{
				delay;
			}
			else
			{
				par
				{

				// data arrives in two pixel chunks
				if(*ActiveRAM == 0)
				{
				RC100WriteSSRAM1( Location, 0 @ Value);
				}
				else
				{
				RC100WriteSSRAM0( Location, 0 @ Value);
				}

				PixelCount = PixelCount+2;
				}
			}
		}
}
else
{
		/* Set the internal registers and swap buffers according to
		* which token arrived. The cases are defined in the RC100.h
		* header file */
		switch(Token(Value))
		{
			case SAA7111_START_LINE_TOKEN:
				par
				{
					PixelCount = 0;
					LineCount = Value[9:0];
				}
				break;
```

```
                                        case SAA7111_START_FRAME_TOKEN:
                                                par
                                                {
                                                        PixelCount = 0;
                                                        LineCount = 0;
                                                        *ActiveRAM = ~(*ActiveRAM);
                                                }
                                                break;

                                        default:
                                                delay;
                                                break;
                                }
                        }
                }while(1);
        }
}


/*
* VGA Display Generation
*/
macro proc Display(Video, ActiveRAM, Keyboard)
{
        //varables used in pipline
        unsigned 36 TempRegister;
        unsigned 18 Location;


        unsigned 11 Input;          //keyboard input
        unsigned 10 correctionfactor; //k

        unsigned 16 kr2;
        unsigned 16 mag ;
        unsigned 16 interpmag1,interpmag2;
        unsigned 8 interscale;
        unsigned 18 sx;
        unsigned 18 sy;
```

```
signed 10 x,y;
unsigned 10 scalex;
unsigned 9 scaley;


macro expr CurrentOutput = (scalex[0] ? TempRegister[15:0]:TempRegister[31:16]) ;

correctionfactor = 0;
par
{
/* Assign to the video output every cycle*/
do
{

        Video->Output = RC100Convert565to888(CurrentOutput);

}while(1);

do
{

                /*      Calcualte the x^2 x^2 use this to work out the
                 *      r^2 and the kr2 */
                if ( Video->ScanX == RC100VisibleCols)
                {
                        par
                        {
                //this changes cordinate system
                        sx = xc*xc;         //at the end of each line reset x^2 to max value
                        x = -xc;  //and make x the value at start of line

                        }
                        if (!Video->VBlank){
                                //update the y value but only not in the vblack line

                                par
                                {       // 2 lines equlivant to y*y
                                        sy = squred(sy,y);
                                        y = myadd(y,1);//y = y +1;
```

```
                }
        }else
        if (Video->ScanY == RC100VisibleLines)
        {
                par
                {       // at the bottom of screen reset y^2 and y
                        sy = yc*yc;
                        y = -yc;
                }
        }else delay;

}
else if (Video->Visible)
{       par
        {
                //pipeline 1

                sx = squred(sx,x);
                x = myadd(x,1);//x = x +1;



                //pipeline 2
                //calculate kr2
                kr2 = ((0@(sx + sy))*(0@correctionfactor))\\12;



                //pipeline 3
                //get mag values
                interpmag1 = Mtable.read1[kr2[15:8]];
                //include case to protect if outside array index
        interpmag2 =((kr2[15:8]+1)!=0)Mtable.read2[kr2[15:8]+1]:23928;
                        interscale = kr2[7:0]; //kr2[19:12]


                        //pipeline 4
                        //******Magnifiaction estimation
```

```
        mag = (interpmag1 - ((0@(interpmag1-interpmag2))*(0@interscale))\\8);



                                        //pipeline 5
                                        //calculate the x and y to look cu in memory
scalex =(unsigned)(xc + ((((0@(signed)((unsigned 1)1@mag))*adjs(x-4,27))\\17)+19);
scaley =(unsigned)(yc + ((0@(signed)((unsigned 1)1@mag))*adjs(y,27))[25:17]);
// taking in to accoun the fact that only bottom bits used for y(unsigned)(yc + y)<-9;

                                        //use the fact that the bottom bit of x selects the
                                        //pixel from the memory address to be displayed
                                        Location = scaley@(scalex\\1);

                                        // Display the front buffer
                                        if(*ActiveRAM == 0)
                                        {
                                        RC100ReadSSRAM0(Location, TempRegister);

                                        }
                                        else
                                        {
                                        RC100ReadSSRAM1(Location, TempRegister);
                                        }
                                }
                        }else delay;
}while(1);

do
{
        // wait until a character can be read from the input
        *Keyboard->ReadChannel ? Input;

        // chage the correction depending on input
        switch(Input)
        {

                case ASCII_RIGHT_ARROW:
                correctionfactor = correctionfactor + 10;
```

```
                                    break;

                                    case ASCII_LEFT_ARROW:
                                    correctionfactor =correctionfactor -10;
                                    break;

                                    case ASCII_r:
                                    correctionfactor = 0;
                                    break;


                                    case ASCII_f:
                                    correctionfactor = 1023;
                                    break;
                                    default:
                                    delay;
                                    break;
                                }
                        }while(1);
                }
}
```

Look up table

```
//this containes the magnification look up table
//Christopher Johnston modified 11/9/2003



static mpram Mtype
{
        ram <unsigned 16> read1[256];
        rom <unsigned 16> read2[256];
}Mtable ={
65535,65029,64535,64051,63578,63115,62661,62217,
61781,61355,60936,60525,60122,59726,59337,58955,
58580,58211,57848,57491,57140,56794,56454,56119,
55790,55465,55145,54830,54519,54213,53911,53613,
53319,53029,52743,52461,52182,51907,51636,51368,
51103,50842,50583,50328,50076,49826,49580,49336,
49096,48857,48622,48389,48159,47931,47705,47482,
47262,47043,46827,46613,46401,46191,45984,45778,
45574,45373,45173,44975,44779,44585,44392,44202,
44013,43825,43640,43456,43274,43093,42914,42736,
42560,42385,42212,42040,41870,41701,41534,41367,
41203,41039,40877,40716,40556,40398,40240,40084,
39929,39776,39623,39471,39321,39172,39024,38877,
38731,38586,38442,38299,38157,38016,37876,37737,
37599,37461,37325,37190,37056,36922,36789,36658,
36527,36397,36267,36139,36011,35885,35759,35634,
35509,35385,35263,35140,35019,34898,34779,34659,
34541,34423,34306,34190,34074,33959,33844,33731,
33618,33505,33393,33282,33172,33062,32952,32844,
32736,32628,32521,32415,32309,32204,32099,31995,
31891,31788,31686,31584,31483,31382,31281,31181,
31082,30983,30885,30787,30690,30593,30497,30401,
30305,30210,30116,30022,29928,29835,29742,29650,
29558,29467,29376,29285,29195,29106,29016,28928,
28839,28751,28664,28576,28490,28403,28317,28231,
28146,28061,27977,27893,27809,27725,27642,27560,
27477,27395,27314,27233,27152,27071,26991,26911,
26831,26752,26673,26595,26516,26439,26361,26284,
26207,26130,26054,25978,25902,25827,25751,25677,
25602,25528,25454,25380,25307,25234,25161,25089,
25016,24945,24873,24801,24730,24659,24589,24519,
24449,24379,24309,24240,24171,24102,24034,23966

}with {  block =1,
                westart =2,
                welength = 1,
                rclkpos ={1.5},
                wclkpos ={2, 3},
                clkpulselen = 0.5
                };
```

Keyboard Support Library

```
/*
************************************************************** *
*                                                              *
* Project   :  RC100 Support Libraries                         *
* Date      :  16 JAN 2001                                     *
* File      :  ASCIIChars.h                                    *
* Author    :  SC                                              *
*                                                              *
* Description:                                                 *
* This header contains RC100 key definitions for a UK Keyboard *
* when used in the ASCII mode.                                 *
*                                                              *
* Date        Version  Author  Reason for change               *
* 16 JAN 2001  1.00    SC     Created.                         *
* 10 APR 2001  1.01    SC     Added more definitions.          *
* 26 APR 2001  1.02    SC     Removed Function keys.           *
* 11 MAR 2003  2.00      CJ                Make it work on RC100 & US keys*
* modified by Christopher Johnston on the 11/3/2002            *
* adding of letters and the correction of arrows               *
**************************************************************/

#ifndef __CELOXICA_ASCII_LIBRARY_HEADER__
#define __CELOXICA_ASCII_LIBRARY_HEADER__


/*
 *letters
 */
#define ASCII_A                                 0x41
#define ASCII_B                                 0x42
#define ASCII_C                                 0x43
#define ASCII_D                                 0x44
#define ASCII_E                                 0x45
#define ASCII_F                                 0x46
#define ASCII_G                                 0x47
#define ASCII_H                                 0x48
#define ASCII_I                                 0x49
#define ASCII_J                                 0x4a
#define ASCII_K                                 0x4b
#define ASCII_L                                 0x4c
#define ASCII_M                                         0x4d
#define ASCII_N                                 0x4e
#define ASCII_O                                 0x4f
#define ASCII_P                                 0x50
#define ASCII_Q                                 0x51
#define ASCII_R                                 0x52
#define ASCII_S                                 0x53
#define ASCII_T                                 0x54
#define ASCII_U                                 0x55
#define ASCII_V                                 0x56
#define ASCII_W                                         0x57
#define ASCII_X                                 0x58
#define ASCII_Y                                 0x59
#define ASCII_Z                                 0x5a

#define ASCII_a                                 0x61
```

```
#define ASCII_b                                 0x62
#define ASCII_c                                 0x63
#define ASCII_d                                 0x64
#define ASCII_e                                 0x65
#define ASCII_f                                 0x66
#define ASCII_g                                 0x67
#define ASCII_h                                 0x68
#define ASCII_i                                 0x69
#define ASCII_j                                 0x6a
#define ASCII_k                                 0x6b
#define ASCII_l                                 0x6c
#define ASCII_m                                 0x6d
#define ASCII_n                                 0x6e
#define ASCII_o                                 0x6f
#define ASCII_p                                 0x70
#define ASCII_q                                 0x71
#define ASCII_r                                 0x72
#define ASCII_s                                 0x73
#define ASCII_t                                 0x74
#define ASCII_u                                 0x75
#define ASCII_v                                 0x76
#define ASCII_w                                 0x77
#define ASCII_x                                 0x78
#define ASCII_y                                 0x79
#define ASCII_z                                 0x7a

/*
 * Control Codes
 */

#define ASCII_SPACE                             0x20
#define ASCII_BACKSPACE         0x08
#define ASCII_TAB               0x09
#define ASCII_ENTER             0x0d
#define ASCII_ESCAPE            0x1b

#define ASCII_ALT_BACKSPACE     0x108
#define ASCII_ALT_TAB           0x109
#define ASCII_ALT_ENTER         0x10d
#define ASCII_ALT_ESCAPE        0x11b

#define ASCII_CTRL_BACKSPACE    0x208
#define ASCII_CTRL_TAB          0x209
#define ASCII_CTRL_ENTER        0x20d
#define ASCII_CTRL_ESCAPE       0x21b

#define ASCII_CTRL_ALT_BACKSPACE  0x308
#define ASCII_CTRL_ALT_TAB      0x309
#define ASCII_CTRL_ALT_ENTER    0x30d
#define ASCII_CTRL_ALT_ESCAPE   0x31b



/*
 * Arrow Keys
 */
#define ASCII_LEFT_ARROW        0xda
#define ASCII_DOWN_ARROW        0xdb
#define ASCII_RIGHT_ARROW       0xdc
```

```
#define ASCII_UP_ARROW              0xdd

#define ASCII_ALT_LEFT_ARROW        0x5da
#define ASCII_ALT_DOWN_ARROW        0x5db
#define ASCII_ALT_RIGHT_ARROW       0x5dc
#define ASCII_ALT_UP_ARROW          0x5dd

#define ASCII_CTRL_LEFT_ARROW       0x6da
#define ASCII_CTRL_DOWN_ARROW       0x6db
#define ASCII_CTRL_RIGHT_ARROW      0x6dc
#define ASCII_CTRL_UP_ARROW         0x6dd

#define ASCII_CTRL_ALT_LEFT_ARROW   0x7da
#define ASCII_CTRL_ALT_DOWN_ARROW   0x7db
#define ASCII_CTRL_ALT_RIGHT_ARROW  0x7dc
#define ASCII_CTRL_ALT_UP_ARROW     0x7dd


/*
 * Editing Keys
 */
#define ASCII_INSERT                0x4d0
#define ASCII_HOME                  0x4d1
#define ASCII_END                   0x4d2
#define ASCII_PAGE_UP               0x4d3
#define ASCII_PAGE_DOWN             0x4d4
#define ASCII_DELETE                0x7f

#define ASCII_ALT_INSERT            0x5d0
#define ASCII_ALT_HOME              0x5d1
#define ASCII_ALT_END               0x5d2
#define ASCII_ALT_PAGE_UP           0x5d3
#define ASCII_ALT_PAGE_DOWN         0x5d4
#define ASCII_ALT_DELETE            0x17f

#define ASCII_CTRL_INSERT           0x6d0
#define ASCII_CTRL_HOME             0x6d1
#define ASCII_CTRL_END              0x6d2
#define ASCII_CTRL_PAGE_UP          0x6d3
#define ASCII_CTRL_PAGE_DOWN        0x6d4
#define ASCII_CTRL_DELETE           0x27f

#define ASCII_CTRL_ALT_INSERT       0x7d0
#define ASCII_CTRL_ALT_HOME         0x7d1
#define ASCII_CTRL_ALT_END          0x7d2
#define ASCII_CTRL_ALT_PAGE_UP      0x7d3
#define ASCII_CTRL_ALT_PAGE_DOWN    0x7d4
#define ASCII_CTRL_ALT_DELETE       0x37f


/*
 * Special Keys
 */
#define ASCII_PRINT_SCREEN          0x4d6
#define ASCII_SCROLL_LOCK           0x4d7
#define ASCII_NUM_LOCK              0x4d8
#define ASCII_BREAK                 0x4d9

#endif //__CELOXICA_ASCII_LIBRARY_HEADER__
```

References

[1] S. Demidenko, Design for Computer and communication systems notes of FPGAs 2002

[2] Xilinx®, *Spartan-II 2.5V FPGA Family: Complete Data Sheet,* (September 3, 2003)

[3] Alston, I., Madahar, B., "From C to netlists: hardware engineering for software engineers?", *IEE Electronics & Communication Engineering Journal*, pp 165-173 (August 2002).

[4] Celoxica Ltd., Programming in Handel-C Frequently Asked questions (2001), www.celoxica.com

[5] Bailey, D.G., "A new approach to lens distortion correction", *Proceedings Image and Vision Computing New Zealand 2002*, pp 59-64 (2002).

[6] Li, M., Lavest, S.M., "Some aspects of zoom lens camera calibration", *IEEE Trans on PAMI*, 18(11): 1105-1110 (1996).

[7] Hollasch, S, "IEEE Standard 754 Floating Point Numbers", (2003-Jan-02) http://research.microsoft.com/~hollasch/cgindex/coding/ieeefloat.html (visited October 2003)

[8] Bainbridge-Smith, A. & Dunne, P., "FPGAs in computer vision applications", *Proceedings Image and Vision Computing New Zealand 2002*, pp 347-352 (2002).

[9] K.T. Gribbon, C.T. Johnston, and D.G. Bailey, *"A Real-time FPGA Implementation of a Barrel Distortion Correction Algorithm with Bilinear Interpolation"* accepted for IVCNZ 2003 conference.