Techniques for Designing an FPGA-Based Intelligent Camera for Robots

Miguel Contreras, Donald G Bailey and Gourab Sen Gupta

School of Engineering and Advanced Technology
Massey University, Palmerston North, New Zealand
M.Contreras@massey.ac.nz, D.G.Bailey@massey.ac.nz, G.SenGupta@massey.ac.nz

Abstract. This paper outlines useful techniques to design and develop an intelligent camera on a Field Programmable Gate Array (FPGA). Some of the development and testing issues of porting a software algorithm onto a hardware platform are discussed, as well as ways to avoid the corresponding problems. To demonstrate the importance of these techniques, an intelligent camera designed to calculate the position, orientation and identification of soccer playing robots is used as a case study.

1 Introduction

Intelligent cameras can be an integral part of many robotic or automated systems. Bailey et al. [1] describes an intelligent camera as an extension of the smart camera by directly processing the pixels as they are streamed from the camera. As such they are well suited for applications where limited space, limited energy, and fast and reliable image processing is required in a self-contained unit. Field Programmable Gate Arrays (FPGAs) are ideal for this as they offer parallel processing, compared to serial processing from a conventional computer CPU. Combined with streamed processing it is possible to begin processing each frame directly as the pixels are read from the sensor, without the need for large latency memory units. In some instances, it is possible to calculate all of the useful information even before the frame is fully captured.

There is a lot of research outlining the use of FPGAs utilising image processing algorithms to create smart cameras [2-4]. However there is very little literature on design techniques for the development of FPGA-based intelligent cameras. Furthermore, there is even less literature which explicitly identifies the development issues and pitfalls. Designing an intelligent camera is a complex task, and there are numerous pitfalls. During the design of our own intelligent camera we have encountered many of them. This paper attempts to address this lack of literature by explicitly identifying these pitfalls, in an effort to reduce the learning curve so that others may learn from our mistakes. This paper is primarily aimed at early developers of intelligent cameras. Nevertheless, some of the discussions will benefit even those who have experience with image processing and FPGA algorithm development. The techniques discussed will help simplify the design of the intelligent camera and decrease the development time.

Section 2 identifies many of the pitfalls and techniques used to develop an intelligent camera. These techniques have been divided into three categories

developing an algorithm testing scheme, developing the initial software algorithm, and developing the resulting hardware algorithm. Each of these sections will outline techniques to overcome the pitfalls, decrease the development time and simplify the development.

The techniques discussed in this paper will be explained using the FPGA-based intelligent camera currently being developed at Massey University. This intelligent camera is designed for robot soccer to identify individual robots using colour patches and calculate their positions and orientation on the field. As described by Contreras et al. [5] the algorithm follows a modular design of blocks or components. Each block is responsible for applying a filter or a process from the overall algorithm, and passing the results onto the subsequent blocks. Further information on the functionality and design on the algorithm can be found in [1, 5]. The software algorithm was developed using Matlab and the hardware description language (HDL) used to program the FPGA was Handel-C. However the techniques discussed in this paper are universal and can be used with any other software package or HDL.

2 Issues with Intelligent Camera Design

Despite all of the advantages that an intelligent camera can offer to projects requiring image processing applications they are not commonly employed. This is due to the greater complexity and longer development time of an FPGA-based camera compared to more conventional software approaches. This paper will look at a few of the more common issues that can hinder the development process.

- Design of parallel systems is complex
- Timing constraints must be explicitly handled due to parallelism
- Limited hardware debugging resources at run-time
- Long compile times for hardware algorithms

Designing a hardware algorithm is not straightforward. Unlike software algorithms which follow a serial command structure FPGAs allow for parallel algorithms to be executed. Parallel algorithms allow for faster processing as many functions can be calculated simultaneously. Because of this the design can become very complex, especially as the algorithm becomes more intricate.

Timing in particular becomes an issue as different operations have different latencies. This is not as much of an issue in software as timing is implicit in the sequence of commands. However, hardware must handle the timing explicitly the ensure information is processed correctly. In many cases errors can occur when implementing the correct logic but at the wrong instance.

Debugging is not as simple for hardware algorithm development as it is for software algorithms. Although compilers check for syntax errors, finding logic and timing errors is much more difficult in hardware. There are two main difficulties here. First, in software, only one thing happens at a time. So the changes resulting from a single clock cycle are small. In hardware, many things happen simultaneously. A lot can change in a single clock cycle, making it significantly more difficult to identify errors and track down there root causes. Many development environments provide a

software simulation package meant to emulate the functionality of the FPGAs resources. These provide a systematic overview of the functionality as the parallel image processing algorithm is executed. Even so, finding the causes of unwanted or unanticipated changes in data can be time-consuming. Second, when interfacing the algorithm with real-time hardware, it is necessary for the algorithm to process data at real-time rates. For simulation, it is also necessary to simulate the operation of hardware external to the FPGA. Therefore, additional simulation models must be developed, debugged, and tested before even simulating the algorithm. For an intelligent camera the simulation model for the camera will need to stream pixel data from a file. Any errors in this model (for example timing errors) can invalidate perfectly working algorithm code. Due to the complexity of most image processing algorithms, simulation can be a very laborious and time consuming method of debugging. Furthermore once the algorithm is encoded onto the FPGA there are limited mechanisms for debugging errors.

Code compilation is an inevitable part of any programming and image processing project. The more complex the algorithm is the longer it takes to compile. This is true for both software and hardware compilers. However it takes longer to compile using HDL compilers as more optimization and timing passes must be processed. Hardware has an added place and route phase where logic is mapped to specific resources on the FPGA. The very large number of possible mappings makes the process poorly defined and difficult to optimise, especially when approaching the capacity of the FPGA. This drastically increases the development time. This is especially true when parts of the algorithm are dependent on the surrounding environment, and must be adjusted frequently.

During the development of our own intelligent camera many techniques were explored in an attempt to minimize these issues. This paper will outline some of the more important and successful principles.

- It is faster to test theories in a software environment than in hardware.
- A modular algorithm design can make programming and testing simpler.
- It is important to clearly define the interfaces between modules.
- Built-in functions in an image processing suite (such as Matlab) can both help and slow down algorithm development
- Some form of communication with the intelligent camera is critical.
- A parameterized hardware design can help make it easier and faster to make changes.
- There are many resources available to help debug during run-time in hardware.

Designing an image processing algorithm in hardware is very difficult. These compilers are not designed to easily show the effects of different filters or techniques compared to a software development environment such as Matlab. Even if the filter is relatively trivial it is better to first test its behaviour in a software environment to investigate how it affects the algorithm as a whole. Using a software environment can save a lot of development time as it allows for easy interaction and fast prototyping of different algorithms.

Numerous image processing environments also come with built-in functions and filters that can further decrease development time. Although great care should be taken when utilising these built-in functions and filters. Even though they may return quick and desirable results, they may be too complex to run efficiently on an FPGA. Once an algorithm has been developed it is possible to use the software environment to modify the design to closer resemble the functionality of the FPGA. Even though the parallelism will not be duplicated, the algorithm can be adapted to loosely mimic its functionality. This makes converting the algorithm to a hardware language much simpler and helps reduce its complexity as a whole.

Both a modular and parameterised algorithm design can be very useful when developing and testing in both software and hardware. A modular design allows each filter or block to be tested individually and simplifies the algorithm into smaller blocks. With each module compartmentalised it is important to clearly define and minimize the transfer of information between each module. This makes design, debugging, and testing much simpler as each module becomes localised. Similarly making each block parameterised can simplify changes to the interaction of the blocks without changing their functionality. However this can add some complexity to the design of each block. Even with the added complexity it still benefits the development process by eliminating the possible of accidentally introducing errors.

Communication between the FPGA and a computer is important. Many off-the-shelf FPGA development boards come with a variety of communication ports, such as USB, RS232, and general purpose IO. These interfaces are essential for transferring captured images from the sensor to begin the development of an algorithm. They can also be used to send commands to the intelligent camera to execute different functions, such as adjusting thresholds. This allows for settings to be changed during run-time without the need to recompile, thus reducing the total time spent compiling the algorithm. Results and processed images can also be transferred from the FPGA in order to test the functionality of the algorithm, in order to measure performance and debug errors. A solid form of communication can be one of the most useful ways to test and develop the intelligent camera during run-time.

There are several other resources available to help debug an FPGA at run-time other than a solid form of communication. Many off-the-shelf FPGAs come with resources such as LEDs, switches and buttons that allow for interaction with the intelligent camera. These can be used in many different ways, from displaying information to enabling and disabling filters. This can help identify faults without the need for extensive simulation examinations.

3 Developing an Algorithm Testing Scheme

When developing any kind of image processing solution, it is very important to follow a procedural development plan. In a very basic form this includes gathering test images, algorithm design, and algorithm testing. This is usually done using a software image processing suite, such as Matlab. This allows for quick prototyping and testing of algorithms with visual and numeric results. Since the image processing algorithm development process is largely heuristic, it involves a lot of trial and error. The long compilation times of HDLs means that making even small changes to an algorithm is no longer interactive, hampering the design exploration process.

Therefore it is best to first design the algorithm in a software environment, and then optimize the resulting working algorithm into an HDL. This allows us to fully test the algorithm to see how it works and behaves under different circumstances before any difficult hardware programming begins. **Fig. 1** illustrates the process used to fully develop an intelligent camera on an FPGA.

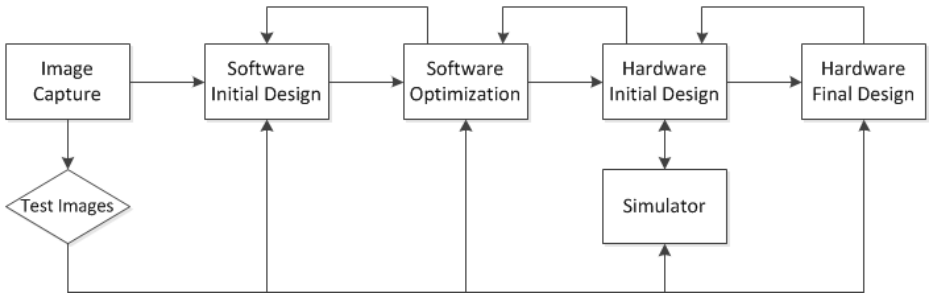


Fig. 1. Development and testing process for an FPGA-based intelligent camera

The first step in the design of any image processing algorithm is to capture a set of test images from the camera. A basic skeleton code will need to be developed on the FPGA to allow images to be taken from the camera and transferred to a computer, shown in **Fig. 2**.

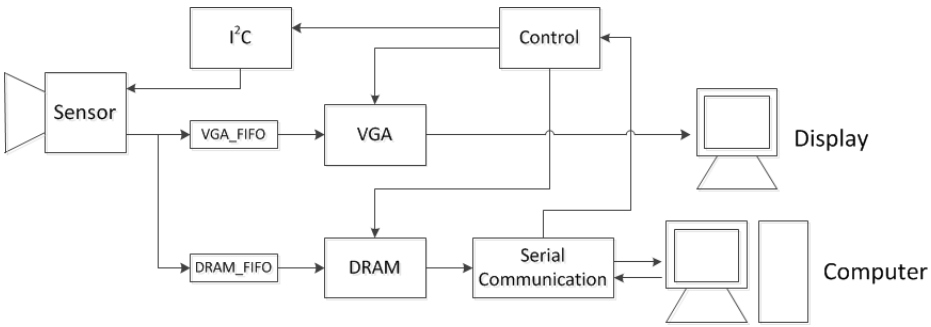


Fig. 2. Block diagram for a basic image capture algorithm on an FPGA

During initialization the control unit initialises the camera resolution, pixel skipping, gain and other camera parameters via the I^2C controller. Under normal operation the captured pixels are loaded into the VGA_FIFO only. The VGA_FIFO acts as a buffer to load the data into the VGA module which controls displaying the pixels onto a monitor. The serial communication module receives the commands coming from the computer and controls sending the test images to the computer. Basic serial commands include camera adjustment via I^2C and commands to initiate a test image capture. When an image capture command is received the controller waits

until the start of the next frame, to ensure a full frame is captured. At the start of the next frame the pixel data is loaded into the DRAM via the DRAM_FIFO. Once the image has been saved into memory it can then be sent to the serial module where it will be sent to a computer. Using an RS232 connection to transfer an image is very slow, taking over one and a half minutes to transfer a single 640x480 image at 57600 baud rate. This is why the image must first be saved to memory so that no information is lost.

The next stage in the development is to start designing the algorithm in a software environment. The desired characteristics of a good image processing algorithm development package are for it to be interactive and to allow fast prototyping of algorithms. Matlab is one such package. This allows various design decisions to be effectively explored to get the desired results from processing the image. **Fig. 1** incorporates two software design processes, the initial software design, and the software optimization.

The aim of the first software implementation is to design the algorithm and make sure it works as intended, as quickly as possible. At this stage it may be easier and quicker to use built-in image processing functions to get the basic functionality of the algorithm. An example of this could be implementing standardized Bayer interpolation or YUV transforms which would be built into Matlab. This saves development time and allows these filters to be tested without needing to program them from scratch.

Once the initial design is completed, the algorithm needs to be optimized to more closely follow the hardware functionality. This is used to help test the effectiveness of the algorithm using the advantages and limitations of the hardware environment. It also provides a standard to test the hardware implementation against, as the results from the hardware modules should be identical to the software implementation when testing the same image. Matlab generally uses double precision floating point in its calculations, whereas most FPGA designs are implemented using fixed point or integer arithmetic. Firstly all equations must be converted or rounded to integer numbers otherwise deviations will occur when comparing test images. Similarly all built-in functions will need to be replaced with limited precision functions that match those which will operate on the FPGA. At this point, reduced precision can be explored to optimize word lengths. Planning the hardware implementation is made easier as the functionality of each algorithm block can be examined and optimized. Consequently any potential implementation problems or limitations can be identified before trying to develop a hardware solution.

Once the algorithm is designed from a software aspect it can be adapted for a hardware implementation. Parallel algorithm design is not straightforward. Because there are multiple processes running simultaneously, not only does the logic and functionality need to be correct but they must also be synchronised. Hardware adds a temporal aspect that must be explicitly handled, unlike serial coding which handles this implicitly. This can make the debugging process very intricate with multiple modules running in parallel. In software debugging each step executes one instruction. This makes error identification relatively easy as only one thing changes each step. The debugging process for hardware becomes more complicated as the parallelism allows for multiple processes to occur each clock cycle, generating numerous changes at once. This can make isolating an error very difficult.

The first step to debug a hardware algorithm is to create a test bench within the simulator to emulate the functionality of the camera. It must emulate capturing pixels, streamed from a text file, and pass them to each block for testing. Using this test bench it is possible to test each algorithm module individually or the complete algorithm using the test images captured earlier. To speed up the simulation, smaller sub-images can be used if the algorithm has been designed appropriately. This simulation is implemented on a computer so it operates serially, however it will simulate parallel operations when intended. This will accurately simulate the operation of the algorithm though it will take much longer to do so. Another advantage of the simulation is that it allows a step by step procedural overview of each block, such as when a register is read or written to, or what values are present in a FIFO etc. This can be very helpful to identify timing or synchronisation errors. The main purpose is to test the modules and compare the results to the software algorithm. The aim is to make the results of the simulation identically match the software results. Any differences present at this time could mean an error in the adaptation of the algorithm. Subsequent testing and debugging will need to be done to find the error and correct it.

A disadvantage of the test bench is that it must be programmed from scratch. This can introduce unintended errors or limitations that may not be applicable in the actual hardware implementation. Special care should be taken when creating this test bench to ensure that any faults that occur are not being caused by the simulator itself.

We made this mistake ourselves in the first revision of our test bench. The camera requires a blanking period at the end of each row before transmitting the next one. In an attempt to compress the simulation time smaller case-specific test images were used with shorter blanking periods. Normally the blanking period would last for several hundred clock cycles, but was shortened to a few dozen. During the testing of the connected component module several errors were reported and it was assumed that the module was not programmed correctly. After much investigation into the logic and timing no problems could be found that could explain the incorrect results. It was not discovered until later that the errors were being caused by the test bench itself and not the module. The connected component module required several clock cycles in the blanking period to finish calculating and assembling each blob. Normally there would be plenty of time to complete them on the FPGA; however the blanking period was too small in the simulator. Although this limitation was relatively easy to correct, it shows that not all errors are necessarily caused by the modules.

Finally once the algorithm is adapted and tested in simulation it is time to run some real-world tests on the hardware itself. This is done by comparing test images processed on the FPGA with identical images processed by software. There are two ways to test the algorithm on the FPGA. The first is to upload a previously captured test image onto the FPGA, processing the image and then downloading the results back to the computer for comparison with the ones calculated by the software implementation. The second method captures a new test image, stores it in memory, and then processes it, avoiding the need to upload a test image. Then both the test image and results are downloaded to the computer and compared with the software implementation. The first method allows for faster overall processing as the software algorithm has already calculated the results. Another advantage is that the camera is not actually required. Hardware emulates the camera by providing the pixel stream

from memory. This can be helpful if the camera will be operating in areas where it may not be possible to have a computer for a long period of time, such as outdoors. However, an advantage of the second method is it can help find limitations in the algorithm that were not initially considered by the initial test images.

4 Techniques for Designing in Software

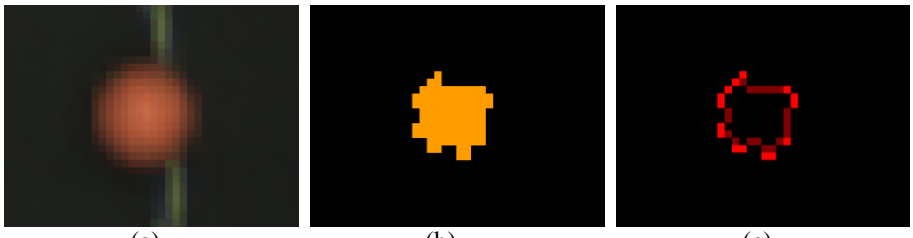
Before the algorithm can be implemented onto an FPGA it must first be designed. HDL compilers are not designed to allow for easy image processing algorithm development. So it is necessary to first design and test the algorithm in software. Even when the algorithm has been developed complications can occur with converting into an HDL. This is where a software application can be used to quickly test changes to the design or operation of the algorithm.

Software algorithm development

The initial design of the algorithm in software is a very important step that should not be overlooked. Even filters that may seem trivial should be tested and designed in software to see how they react with the rest of the algorithm. A software package, such as Matlab, is interactive and allows for quick prototyping of concepts to get the functionality correct. An example of this from the development of our own intelligent camera, was trying to identify each robot. It was theorized that the colour patches could be identified using shape recognition, using compactness [6].

$$Compactness = Perimeter^2/Area \quad (1)$$

This is a unitless value that can distinguish between circular, square, and rectangular patches on the basis of shape. However, dealing with discrete images means that many measurements are only approximations of their continuous counterparts. It is well known that measuring a perimeter can be difficult [7, 8]. Therefore it is important to first test the theory in software to identify any possible limitations or inaccuracies before developing the hardware algorithm and discovering it does not work.



(a) (b) (c)

Fig. 3. Images from case study. (a) Image of the ball after the Bayer interpolation. (b) Image after colour thresholding. (c) Perimeter outline of the ball.

Fig. 3 shows the ball from one of the test images. The ball looks circular after the Bayer interpolation. However, the thresholding modules removed a lot of the edge pixels. This left the circle very misshaped. Using the compactness equation it was very hard to distinguish the difference between the circles and squares.

In this case several different methods of estimating the perimeter were compared before deciding that none were sufficiently reliable, and a different shape factor was used. By using the software development platform we were able to quickly prototype this theory and visually observe its effects. This allowed us to make quick decisions and explore other options.

Another advantage of creating a software algorithm is it creates a gold standard for which to test the hardware implementation against. If the software algorithm is capable of calculating correct results, then the goal is to make the FPGA return identical results. The software algorithm can therefore be used to create case-specific test images by partially processing test images captured from the camera.

When to use built-in image processing functions

Matlab like many other image processing suites come with built-in functions and filters used to support image processing. Examples are Bayer pattern demosaicing and various morphological noise reduction filters. In the first stage of algorithm design an argument can be made whether or not these built-in functions should be used. On one hand the aim of the first algorithm design is to devise a solution to the problem as quickly as possible. On the other hand these filters exploit techniques and algorithms that may not function on an FPGA, therefore making the results less useful. Though, this may not be a problem when applying simpler, standardised filters such as an RGB to YUV conversion. Special care should be taken when applying filters such as noise reduction or Bayer interpolation, as there are many different types. A lot of papers have been written comparing the difference of image quality between different Bayer interpolations[9, 10]. For example, a gradient-corrected linear interpolation as described by Malvar et al. [11], similar to the algorithm used in Matlab, gives better results than a simple nearest-neighbour interpolation. Nevertheless the nearest-neighbour interpolation would be computationally simpler to implement on an FPGA. It is good practice, especially in this case, to try the simplest algorithms first. A complex Bayer interpolation could take a long time to develop and test compared to a simpler solution. Why spend weeks or months developing a complex solution when a simple one will return adequate results? Also, if the nearest-neighbour interpolation is likely to be implemented on the FPGA then it should be used in the software development; otherwise any results calculated would be less applicable.

5 Techniques for Designing and Testing in Hardware

There are many difficulties with programming an HDL. One of the most common, and at times hardest to debug, is timing and synchronisation errors [4, 12]. Commonly these errors will cause a register to be overwritten before the data has been read, causing the algorithm to fail. These errors become increasingly hard to find as the algorithm becomes larger and more complex. In a software implementation this is not a problem as there is plenty of memory to assign each variable an individual register.

In a hardware solution this can be impossible or certainly impractical to do especially as the size and complexity of the algorithm increases. With pixel processing, for example, the same hardware processes all the pixels, just at different times. This is because memory and resources are limited on FPGAs. There are some techniques that can help simplify algorithms, and thus make it easier to debug and resolve timing errors.

Modular algorithms

A modular algorithm breaks up the entire algorithm into smaller interconnected blocks. Each block is responsible for applying a specific filter or process and passes the results to the subsequent block. Each block operates independently from each other but is kept synchronised by signals passed between them. By reducing the algorithm into smaller blocks each module becomes less complex and can be tested individually.

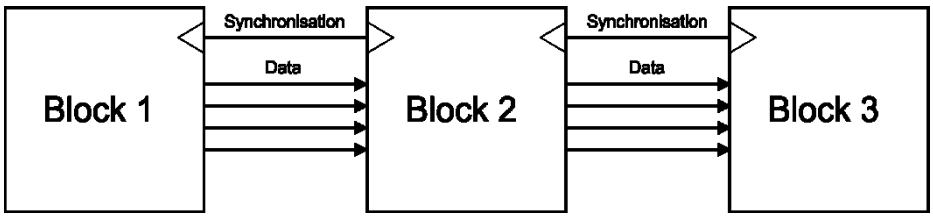


Fig. 4. Block diagram showing a basic modular design

The modular design utilises the hardware's parallel properties to run each block concurrently on streamed data. Unlike software image processing where entire frames are captured and processed, this design processes each pixel individually. This greatly decreases the latency as each frame is processed as it is captured. Therefore there is no need to store each frame into memory. However, since each block requires a different amount of time to process (latency) a data synchronous control signal is used to keep the algorithm synchronised. The processed data is passed between each block though signal carriers, and there is no limit to how many there can be. In fact in more complex situations a single block can pass multiple synchronisation and data signals to various different blocks. For example, **Fig. 5** shows the output of the connected component module sending data to various other blocks.

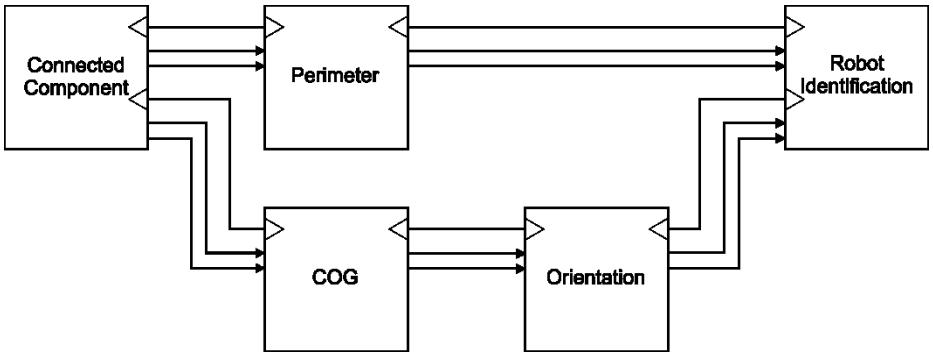


Fig. 5. Block diagram illustrating the functionality of multi input/output modules

The connected component module is responsible for detecting blobs from thresholded pixels and gathering data about them. This information is needed by multiple blocks to calculate the robot position, orientation, and identification. Therefore the blob detection block outputs separate data and synchronisation signals to the various blocks down stream.

Parameterised algorithms

When programming in any HDL, register widths and data paths must be statically defined. This means that an 8 bit value can only fit in an 8 bit register, unless specific concatenation or splitting is used to change the width of the value. The natural instinct when programming an HDL is to hard-code the lengths of these registers. Most of the time, this creates a simple and easy to follow design that works perfectly. The problem comes if the design changes and several of the data paths change width. It can be a tedious and error-prone process going through the design making all the changes. However, a technique that can help simplify the development, especially when used with a modular design, is to parameterise each block. This allows adjustments to be made quickly without changing the functionality of each block. Example parameters may be to adjust the width of data pixels, change the position of words within concatenated strings, or changing the latency of synchronisation signals. Each change on its own could be responsible for dozens of register changes, and could introduce errors.

There are a number of ways to parameterise an algorithm, but the simplest is to have all register widths point back to global variables instead of hard-coding them. By adjusting the global variables the changes will propagate throughout each block without changing their functionality. The disadvantage of this method is that it may take longer to encode each block. Nevertheless it can make the development of the algorithm simpler and faster should any changes be necessary.

A variation of the first method requires utilising control variables inside each block as well as global variables. Utilizing all of the global variables in one position can create a very large and complex group. Particularly if some of the values are only used by a small number of blocks. By removing some of the less common global variables and only implementing them within their required blocks it is possible to

reduce the number of global variables. The drawback to this method is that each control variable will need to be changed individually, although this is still simpler than changing dozens of individual registers.

An example of where a parameterised algorithm would be useful can be examined from the case study. The original algorithm for the intelligent camera was designed to use an 8-bit data stream from the image sensor. It was later discovered that there were problems distinguishing some colours. This was resolved by increasing the camera pixel resolution to the full 12-bits per pixel available from the camera. This required massive changes to the hardware algorithm to allow for the extra data to be processed. This took considerable time to manually change each block and debug any errors that were inadvertently introduced. However, with a parameterised design only a few parameters would need to be changed and the blocks would not need to be retested as they would still function as before, only processing wider data words.

Techniques to help debug at run-time on an FPGA

One of the hardest parts of developing and testing the algorithm on an FPGA is the lack of a run-time debugger. Once the FPGA is running the design will either work or not, and when it fails often there will be no indication as to the cause. There are some ways to troubleshoot this however. Many off-the-shelf FPGA development boards have features, such as LEDs, LCD displays, 7-segment number displays, switches, and buttons. These can all be used to help debug the code or relay what process is being executed. For example the 7-segment number display can be utilised to act as a frame counter, or to display threshold values. LEDs can be used to indicate whether a communication command has been sent or received, or when a command finishes executing. Switches can be used to activate or deactivate specific filters to see the effect they have. Similarly, buttons can be used to make adjustments to filter characteristics and thresholds.

An example from the case study shows how many of these features where used together to help capture test images. When a serial command is sent from the computer to the FPGA to initiate an image capture command an LED will light up. This is to indicate that the command has been received and the process will begin. As the image is transferred from DRAM to the serial module the 7-segment counter is used to keep track of which row is being processed. This helped identify issues with data loss, whether it was caused due to FPGA or connection error. Finally when the test image was fully sent another LED would light up to signal the end of the process.

Some FPGA manufacturers will make basic code available to operate these hardware features. However it is usually only available for a limited number of HDLs. This means that drivers will first need to be created before these features can be used. This is not too much of a problem as these are relatively simple to write.

6 Summary & Conclusion

Using an FPGA to implement image processing is not a new idea. FPGA-based intelligent cameras can be very powerful, allowing for fast and accurate processing from a smaller and more power efficient platform, compared with conventional computers. Even though intelligent cameras offer many advantages for different robotic applications, they are difficult to develop. With this in mind, it is surprising to find a lack of literature detailing the development process for intelligent camera design. The techniques discussed in this paper are meant to as a guide to help others new to this field to quickly and efficiently develop their own projects. To an expert some of these techniques may seem obvious; however it is likely that at some point, like us, they have made the same mistakes.

Because the development of any image processing algorithm can be quite complex it is best to first design it using a software package, such as Matlab. Once an optimized algorithm has been developed it needs to be redeveloped to operate efficiently on an FPGA. The design and testing of the hardware implementation can be simplified by making the algorithm modular. Also, by parameterising each module, changes can be effected easily without changing the functionality of each block. Overall, a solid testing scheme must be present to ensure both software and hardware algorithms operate as designed, thus aiding in the development of the intelligent camera.

Acknowledgements

This research has been supported in part by a grant from the Massey University Research Fund (11/0191).

References

[1] Bailey, D., Sen Gupta, G. and Contreras, M., Intelligent Camera for Object Identification and Tracking. In: Robot Intelligence Technology and Applications 2012. Advances in Intelligent Systems and Computing, vol. 208. Springer Berlin Heidelberg, pp 1003-1013 (2012)

[2] Johnston, C., Gribbon, K. and Bailey, D., Implementing image processing algorithms on FPGAs. In: Eleventh Electronics New Zealand Conference (ENZCon'04), Palmerston North, New Zealand, pp 118-123, (2004)

[3] Dias, F., Berry, F., Serot, J. and Marmoiton, F., Hardware, Design and Implementation Issues on a FPGA-Based Smart Camera. In: First ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC '07) Vienna, Austria, pp 20-26, (2007)

[4] Lim, Y., Kleeman, L. and Drummond, T., Algorithmic Methodologies for FPGA-Based Vision. In: Machine Vision and Applications, vol. 24, pp 1197-1211, (2013)

[5] Contreras, M., Bailey, D. and Sen Gupta, G., FPGA Implementation of Global Vision for Robot Soccer as a Smart Camera. In: Robot Intelligence Technology

and Applications 2013. Advances in Intelligent Systems and Computing, vol. 274. Springer International Publishing, pp 657-665 (2013)
[6] Davies, E., Machine Vision: Theory, Algorithms, Practicalities (3rd ed), Morgan Kauffmann, San Francisco, USA, pp 193-194 (2005)
[7] Kulpa, Z., Area and Perimeter Measurement of Blobs in Discrete Binary Pictures. In: Computer Graphics and Image Processing, vol. 6, pp 434-451, (1977)
[8] Ellis, T., Proffitt, D., Rosen, D. and Rutkowski, W., Measurement of the Lengths of Digitized Curved Lines. In: Computer Graphics and Image Processing, vol. 10, pp 333-347, (1979)
[9] Ramanath, R., Snyder, W., Bilbro, G. and Sander, W., Demosaicking methods for Bayer color arrays. In: Journal of Electronic imaging, vol. 11, pp 306-315, (2002)
[10] Jean, R., Demosaicing with The Bayer Pattern. Department of Computer Science, University of North Carolina, (2010)
[11] Malvar, H., Li-Wei, H. and Cutler, R., High-Quality Linear Interpolation for Demosaicing of Bayer-Patterned Color Images. In: IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '04), Montreal, Canada, pp 485-488, (2004)
[12] Gribbon, K., Bailey, D. and Johnston, C., Design Patterns for Image Processing Algorithm Development on FPGAs. In: IEEE TENCON of Region 10, Melbourne, Australia, pp 1-6, (2005)