Designing Application-Specific Processors for Image Processing

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

Master of Science in Computer Science

Massey University, Palmerston North,
New Zealand

Aaron Bishell

2008

Abstract

Implementing a real-time image-processing algorithm on a serial processor is difficult to achieve because such a processor cannot cope with the volume of data in the low-level operations. However, a parallel implementation, required to meet timing constraints for the low-level operations, results in low resource utilisation when implementing the high-level operations. These factors suggested a combination of parallel hardware, for the low-level operations, and a serial processor, for the high-level operations, for implementing a high-level image-processing algorithm.

Several types of serial processors were available. A general-purpose processor requires an extensive instruction set to be able to execute any arbitrary algorithm resulting in a relatively complex instruction decoder and possibly extra FUs. An application-specific processor, which was considered in this research, implements enough FUs to execute a given algorithm and implements a simpler, and more efficient, instruction decoder. In addition, an algorithms behaviour on a processor could be represented in either hardware (i.e. hardwired logic), which limits the ability to modify the algorithm behaviour of a processor, or "software" (i.e. programmable logic), which enables external sources to specify the algorithm behaviour.

This research investigated hardware- and software- controlled application-specific serial processors for the implementation of high-level image-processing algorithms and compared these against parallel hardware and general-purpose serial processors. It was found that application-specific processors are easily able to meet the timing constraints imposed by real-time high-level image processing. In addition, the software-controlled processors had additional flexibility, a performance penalty of 9.9% and 36.9% and inconclusive footprint savings (and costs) when compared to hardware-controlled processors.

Acknowledgements

Thanks go to my supervisors, Donald Bailey and Paul Lyons, for their patience and assistance in writing this thesis. It would not have been the same without their considerable knowledge in both image processing and formal writing. Their assistance has been greatly appreciated even though this may not have always been apparent.

Again, thanks to my friend Erica Jones for her friendship, support and writing assistance.

Table of Contents

List of Figures and Tables

1 Designing Application-Specific Serial Processors for Image Processing

Image processing involves the analysis of an image, or a stream of images, to derive information or to create new images that are more useful. It is becoming increasingly common with the advent of the digital camera and video camera surveillance, and has many applications including feature detection, facial recognition, medical imaging, computer vision, machine vision and image manipulation (e.g. red eye removal).

Many image-processing algorithms process input from a real-time video source. There is only a limited time available for processing each frame before the next frame arrives. For some applications, implementing a real-time image-processing algorithm on a serial processor is difficult because such a processor cannot cope with the volume of data [1]. Therefore, a parallel implementation is necessary because its ability to execute operations concurrently allows it to achieve the required throughput.

FPGAs (Field Programmable Gate Arrays) are suitable for implementing real-time image processing because they can exploit parallelism and do so with flexibility [2] since they are programmable. An FPGA comprises a matrix of logic blocks, interconnected with a switching network. The logic is programmable, allowing application-specific hardware to be constructed and altered with a relatively low cost. Being hardware, all of the circuits operate concurrently, allowing significant speedup over conventional serial processing. FPGAs can be seen as a compromise between inflexible but faster application-specific integrated circuits and flexible but slower general-purpose serial processors.

A typical image-processing algorithm involves various types of operations, which can be classified according to the volume of data they process [3]. Low-level operations such as contrast adjustment and edge detection are relatively simple, processing each pixel within an image and producing another image. These involve the highest volume of data and the resulting data is relatively simple (i.e. pixel coordinates and colours). Medium-level operations process each pixel within an image and derive features such as regions or lines. These involve less data than low-level operations and the resulting data is more complex (e.g. position and dimensions of a region). High-level operations are relatively complex, processing the derived features and inferring information. These involve the lowest volume of data and the resulting data is the most complex (e.g. position and orientation of an object in an image). Thus, as one moves from the low-level operations to high-level operations, the knowledge about an image increases and the volume of data that is processed decreases.

An approach based solely on a serial processor, such as shown as the top row in Figure 1.1, is unsuitable for implementing a complete image-processing algorithm because for many algorithms it is unable to meet timing constraints when processing low-level operations. The medium-level operations share many of the same characteristics as low-level operations (in that the input consists of a large array of pixels) so may also require parallel hardware to meet timing constraints. At the other extreme, a fully parallel implementation, shown as the second row from the bottom in Figure 1.1, requires an inordinately large amount of logic to implement the high-level operations. This is because the wider range of individual steps within the high-level operations would all need to be implemented in hardware and since each step is only used occasionally, it results in relatively low resource utilisation. High-level algorithms also tend to be dominated by serial operations, and are therefore less effective in exploiting the concurrency available in parallel hardware.

These factors suggest that when implementing an image-processing algorithm, parallel hardware is best used for the low- and medium-level operations and a conventional serial processor is best used for the high-level operations. There are three basic types of approach available on FPGAs that combine parallel hardware with a serial processor and these be categorised by their relative proportions of hardware and software. The two most software-oriented approaches, shown second and third from the top in Figure 1.1, use a general-purpose serial processor *core* (e.g. a soft “field programmable” processor such as the MicroBlaze or a “hardwired” processor such as the PowerPC 405) to implement the high-level operations. An alternative approach, which is considered in this research, is to use an application-specific serial processor to implement the high-level operations. This is shown third row from the bottom in Figure 1.1. Also shown on the bottom row of Figure 1.1 to put the other approaches in context is the dedicated hardware approach, using an application specific integrated circuit.

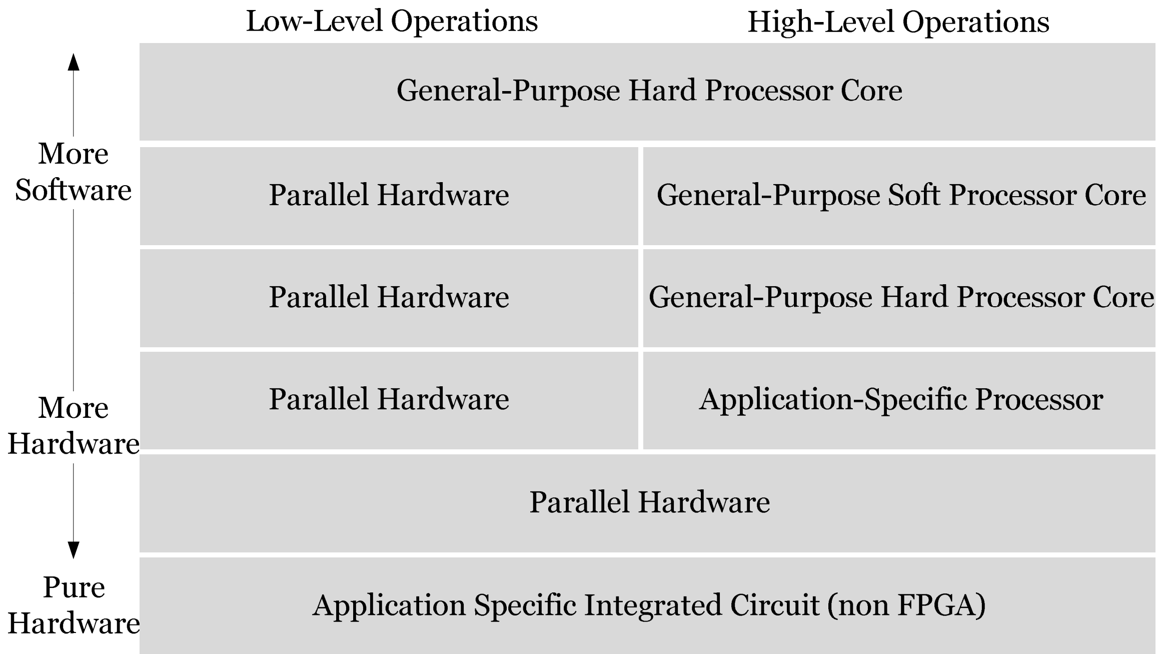

Figure 1.1: Implementation Approaches for an Image Processing Algorithm

We now consider in more detail the three basic approaches that combine parallel hardware and a serial processor on an FPGA.

The instruction set that a general-purpose serial processor requires in order to execute arbitrary algorithms is more extensive than the instruction set an application-specific serial processor needs for its single algorithm. This means a general-purpose processor requires a larger, more complex instruction decoder and possibly extra FUs (Functional Units). A FU is a component of a processor that performs a function, possibly using dedicated temporary registers and logic for arithmetic, and will implement an instruction in the processor. However, a general-purpose processor requires a smaller design cost because the hardware (e.g. the instruction decoder and FUs) has already been designed and implemented, leaving only the software to be programmed to support the execution of a given algorithm.

An application-specific processor contains fewer, but more specialised, FUs, and therefore may occupy a smaller footprint (a term that describes the resources used to implement the processor on the FPGA) than a general-purpose processor. The FUs of an application-specific processor are also tailored to the particular application, so they should provide better performance than the more general-purpose FUs of a general-purpose processor. With fewer instructions, the instruction decoding should also be simpler and more efficient. However, an application-specific processor requires a more extensive design effort since because both its software and hardware must be designed to implement an algorithm.

A general-purpose processor is more appropriate for the implementation of a relatively complex algorithm because the design cost to implement this on an application-specific processor would be prohibitively high. In addition, more of a general-purpose processor's instruction set would be utilised by a more complex algorithm, which would lessen the footprint and performance improvements that an application-specific processor could make when implementing the same algorithm. An application-specific processor may be more appropriate for implementing the high-level image-processing algorithm for a single specific application because such algorithms are more likely to be relatively simple. A simpler algorithm will increase the footprint savings and performance improvements relative to a general-purpose processor and the design cost to implement it on an application-specific processor will be minimal.

An algorithm's behaviour on a processor can be represented in either hardware (i.e. hardwired logic) or "software" (i.e. programmable logic). Hardware representation limits the ability to modify the algorithm behaviour of a processor because it is "hardwired" and if the algorithm is altered, the entire processor, including its behaviour, has to be re-synthesised on the FPGA. Software representation enables external sources, such as external inputs or memory, to specify the algorithm behaviour. This has the advantage of allowing algorithm behaviour to be modified by external sources and without a time-consuming re-synthesis of the processor to the FPGA.

Thus, this research investigates the suitability of software-controlled application-specific processors for the implementation of high-level image processing in terms of two contentions. The first contention is that, for high-level image processing problems, an application-specific serial processor is slower than parallel hardware, but is sufficiently fast to meet real-time timing constraints. The second contention is that, for high-level image processing problems a software-controlled application-specific processor provides the same performance as a hardware-controlled application-specific processor and allows the flexibility of software based algorithm representation. These contentions have been tested by seeking answers to the following research questions:

- Is the performance of an application-specific serial processor sufficient to meet real-time timing constraints?
- Is the footprint of a *software-controlled* application-specific processor smaller than that of a *hardware-controlled* application-specific processor?
- Does a *software-controlled* application-specific processor provide additional flexibility over a *hardware-controlled* application-specific processor?
- Is the performance of a *software-controlled* application-specific processor the same as that of a *hardware-controlled* application-specific processor when implementing a high-level image-processing problem?

The literature underpinning the design of an FPGA based application-specific processor is outlined in Chapter 2. Here, an appropriate hardware description language is chosen, the design space is constrained to several types of computer and processor architectures and finally a strategy for creating FUs is outlined. Chapter 3 derives a set of guidelines for implementing an image-processing problem with a hardware-controlled application-specific processor in a way that minimises footprint. In addition, custom hardware for performing multiplication, division and arctangent is created. Next, Chapter 4 develops a partial methodology to ease the creation of a software-controlled application-specific processor, including architectural guidelines, a FU partitioning strategy and a system for automation and programming. Chapter 5 evaluates hardware- and software- controlled application-specific processors and investigates the suitability of parallel hardware, application-specific processors (both hardware-and software- controlled) and general-purpose processors for implementing a specific high-level image-processing algorithm. Lastly, Chapter 6 draws conclusions based on the research questions posed above and identifies suitable areas for future work.

2 Related Research

This chapter investigates research related to the design of application-specific processors on FPGAs.

2.1 Hardware Description Languages

This section introduces two of the most popular HDLs (Hardware Description Languages) and justifies the decision to use Handel-C. It is important to discuss this before other issues related to implementing a processor because all HDLs introduce limitations that influence a processor's architecture and the design process in unique ways.

An HDL is a language capable of describing electronic circuits or "hardware". Originally, HDLs were used primarily for documentation and specification of digital circuits. However, more recently they have been used to synthesise logic circuits.

Many different HDLs exist with the two most common being VHDL (Very-high-speed-integrated-circuits-HDL) [4] and Verilog [5]. These HDLs have low-level constructs that allow developers to create circuit blocks at the gate level and these blocks can then be combined in a hierarchical manner to create complex circuits. However, these languages are verbose, requiring the developer to specify everything in minute detail. The low-level nature of much of this detail can distract developers from more important considerations such as the overall function of the hardware.

This has led to the development of a wide range of HDLs that allow a high-level description of an algorithm that is suitable for synthesis to hardware. Many of these high-level languages are based on software languages, such as Impulse C [6], JHDL [7-9], MATCH [10], SA-C [11, 12] and SystemC [13], that allow relatively compact representations of algorithms. In these cases, the compiler takes care of the low-level details required to map the algorithm onto gates, freeing the developer from this laborious task.

One such language is Handel-C [14]. It is based on a subset of the ISO C [15] programming language with additional extensions to facilitate the creation of efficient hardware. Consequently, it sits somewhere above the low-level HDLs because it enables a high-level algorithm description. It also provides low-level extensions for constructs that would be inefficient or clumsy to represent using standard C. This allows a developer in Handel-C to avoid many of the low-level distractions present in VHDL and Verilog but still create efficient hardware designs.

A significant disadvantage of Handel-C is that it does not support the use of buses and instead creates multiplexors to govern the inputs to blocks of hardware. Multiplexors can be costly in footprint when synthesised to an FPGA because every input to a shared block must be switched. This can inadvertently influence the design process because trying to minimise the footprint will favour architectures that minimise multiplexing and this possibly reduces the potential for hardware reuse. Multiplexing and hardware reuse are constraining characteristics of a processor's architecture because as more blocks of hardware are reused, there will be more inputs into those blocks and this will result in more and larger multiplexors being required to select the appropriate inputs. Despite this disadvantage, Handel-C is utilised as the HDL for this research for several reasons:

- It allows fast migration of concepts to hardware because its level of expressiveness is at a high level and therefore closer to the textual description of an image-processing problem.
- It allows low-level control design techniques [16], such as programmable logic arrays and microcode, to be avoided since it provides loop and branch constructs (discussed later) that implicitly compile to custom control circuitry. This makes it suitable for this research, which involves designing high-level architectures, as opposed to low-level circuitry.
- It has a small learning curve since it is based on the well-known C programming language.

- It allows complex functionality, typical of high-level image processing, to be expressed easily with complex data structures including: structures, pointers and functions.
- The development software for Handel-C was readily available.

2.2 Architectural Design Space

This section investigates the architectural design space based upon the decision to use Handel-C as the HDL. In this research, the architectural design space refers to the range of possible decisions that can be made when implementing a high-level image-processing algorithm with an application-specific processor.

This design space encompasses two aspects: the computer architecture, which is composed of memory, processor(s) and I/O (Input/Output) devices; and the processor architecture, which is composed of the registers, functional units and the controller. The design of both computer architecture and processor architecture without restriction leads to a prohibitively large architecture design space. Therefore, the following sections introduce constraints to restrict the design space to one that is smaller and more feasible.

2.2.1 Computer Architecture

The computer architecture characterises how the memory, processor(s) and I/O devices are connected and arranged. All computer architectures have these components as well as being capable of [17]:

- executing a program written in a language that is programmable
- transforming data from memory or I/O devices according to the program being executed, and
- potentially reading from and writing to memory and I/O devices

A computer architecture with multiple processors can exploit parallelism by giving each a different stream of data. However, implementing multiple processors is resource intensive since each processor incurs a large initial cost to implement the controller, irrespective of the complexity of the logic that transforms its data stream. The emphasis on reduction in footprint and the unimportance of exploiting parallelism in this research leads to the first constraint: the computer architecture will utilise only one processor.

Combined instruction and data memories provide flexibility since the program can be modified during execution. However, this is unnecessary (and generally regarded as bad practice) since the processors implemented in this research are algorithm specific and thus will not need to modify their behaviour during execution. Combining the instruction and data memory also has two disadvantages:

- It forces the width of the instruction and data memory to be the same rather than tailoring the width of each to what they require. Having the same width can potentially increase the overall memory footprint.
- It reduces the effective data memory bandwidth because data access is shared with instruction fetching. This is complicated by the fact that the hardware implementing the low-level operations is independent of the processor and will be executing concurrently to meet timing constraints. This further increases the chance that

memory bandwidth becomes constrained since the low-level hardware would also write to data memory.

The emphasis on performance rather than flexibility, in this research, leads to the second constraint: the computer architecture will use separate data and instruction memories.

These two constraints result in a single processor Harvard architecture, as shown in Figure 2.1, rather than the "von Neumann" architecture [18] which uses a single memory for both instructions and data.

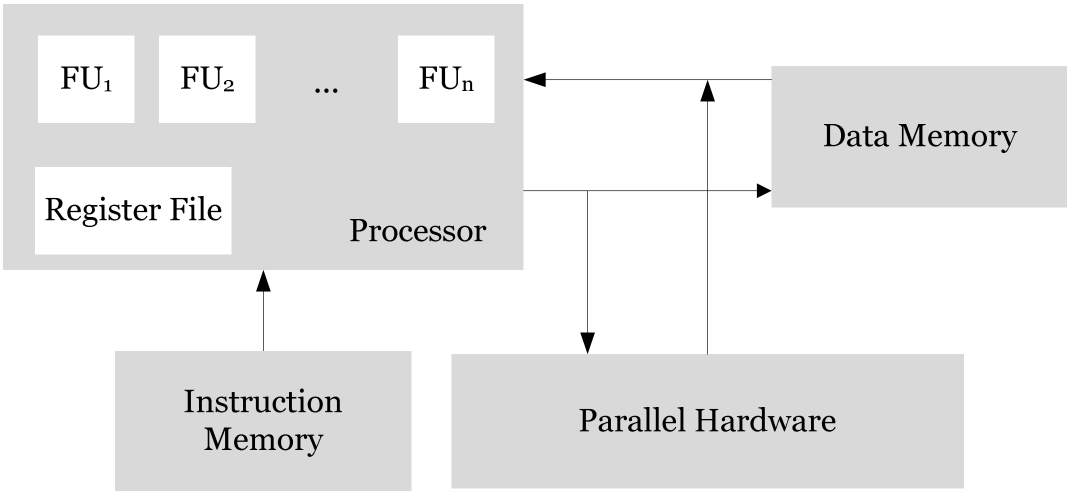

Figure 2.1: Harvard Architecture with a Single Processor

2.2.2 Processor Architecture

The previous section investigated how the memory, processor(s) and input/output (I/O) devices are connected and arranged whereas this section investigates the processor architecture itself. The design space for an unconstrained processor architecture is prohibitively large and only a small proportion of processor architectures are well suited to high-level image processing. Therefore, this section uses several classification systems to categorise the processor architecture design space, and then reduces the complexity of the processor design task by excluding architectures that are least suitable for high-level image processing.

The first classification system used is Flynn's taxonomy [19], which is shown in Figure 2.2. This classifies processor architectures into four categories based on the concurrency of instruction streams and the concurrency of data streams. SISD (Single Instruction stream, Single Data stream) is used to classify the "von Neumann" architecture mentioned previously. MISD (Multiple Instruction stream, Single Data stream) is often disregarded since its uses, such as task replication for fault tolerant computers, are rare. SIMD (Single Instruction stream, Multiple Data stream) and MIMD (Multiple Instruction stream, Multiple Data stream) are both suitable for exploiting data-level parallelism.

		Instruction Stream(s)	
		Single	*Multiple*
Data Stream(s)	*Single*	SISD	MISD
	Multiple	SIMD	MIMD

Figure 2.2: Flynn's taxonomy

A second classification is derived from the techniques available for increasing a processor's performance and builds on Flynn's taxonomy. It classifies architectures according to the following four components [17]:

- Issue rate (I)
  This represents the number of instructions that are issued per clock cycle and is associated with ILP (Instruction-Level Parallelism). MIMD and dataflow architectures have a relatively high issue rate.
- Number of operations specified per instruction (O)
  Instructions may contain more than one operation. VLIW (Very Long Instruction Word) architectures may have a high number of operations per instruction.
- Number of operands to which an operation is applied (D)
  If the same operation is applied to many operands (pieces of data) then the processor architecture will have a high D value. SIMD architectures typically have a relatively high D value.
- "Super pipelining degree" (S)
  This is calculated with:
  $S = \sum(frequency(instruction) x\ latency(instruction))$
  Where the latency is measured in clock cycles. This indicates how many delay slots have to be filled, on average, to keep the processor busy.

Given these two methods for classifying processor architectures, a more extensive taxonomy can be created which takes into account many aspects of processor architecture. Figure 2.3 [17] shows this by separating processor architectures based on their instruction stream, the type of parallelism they exploit (if any) and with the second classification scheme mentioned above. The CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer), shown in the right most branch, are instruction set architectures and not strictly types of processor architectures. However, they influence the processor architecture considerably. These are hereafter referred to as the *CISC-sequential* and *RISC-sequential* architectures respectively.

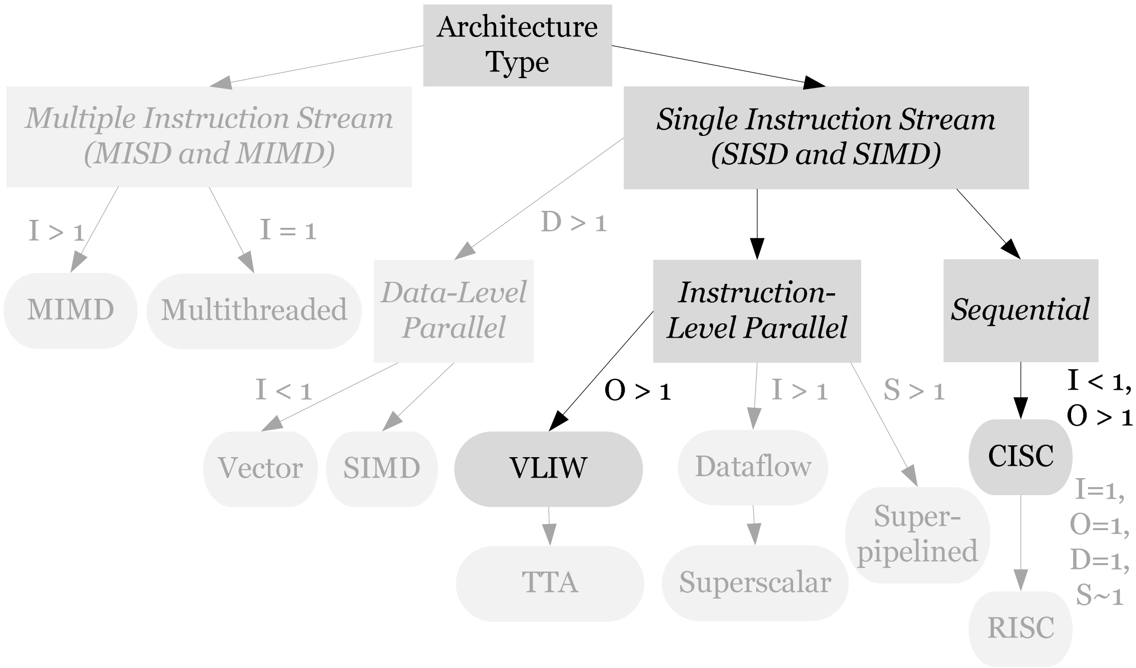

Figure 2.3: Processor Architecture Taxonomy

Architectures that exploit data or instruction-level parallelism are unnecessary for high-level image processing because parallelism is not required to meet timing constraints. This includes those that do so using a single instruction stream, shown as the VLIW (Very Long Instruction Word), TTA (Transport Triggered Architecture), dataflow and superscalar architectures, as well as those that do so using multiple instruction streams, shown as the MIMD and multithreaded architectures, from the figure above. Exploiting parallelism may also require additional hardware, which will increase the footprint of the implementation and thus is incongruent with the goals of this research. This introduces a third constraint: only single-instruction-stream-sequential processor architectures are considered since all others exploit some form of instruction or data level parallelism. However, there are two exceptions to this constraint, discussed below, the first being RISC-sequential architectures are avoided and the second being VLIW architectures are included. This is shown in the figure above where all unsuitable architectures have been greyed out.

RISC-sequential architectures are unsuitable because an algorithm-specific architecture is more likely to require the complex and algorithm-specific FUs that characterise the CISC-sequential approach rather than the simpler more general FUs that characterise the RISC-sequential approach. This concept is discussed in detail in Chapter 4.2.

An architecture that specifies multiple operations per instruction, such as the VLIW and TTA, has the advantage that the complexity of opcode decoding is reduced. These types of architectures require a wider instruction memory to accommodate specifying multiple operations per instruction word but the cost of this is likely to be offset by the savings made by simpler opcode decoding. Therefore, VLIW [20] and TTAs (Transport Triggered Architectures) [17] are considered.

A VLIW architecture is capable of specifying multiple operations in a single instruction word and is able to execute all FUs unconditionally and in parallel. This results in a relatively wide instruction word since the control data for multiple FUs (execute bits), indicating if they will execute, as well as sufficient fields for operand data is required. An example is shown at the bottom of Figure 2.4 where three execute bits and at least two fields are present. A TTA does not explicitly represent operations in the instruction word yet is able to perform multiple operations per instruction because it executes all FUs unconditionally, and in parallel, and uses the instruction word to specify one or more moves, which shift data between registers. This results in a relatively small instruction word since only the identity, in a specific order, of pairs of registers needs to be expressed so the shifting of data between those registers can be performed. An example is shown at the top of Figure 2.4 where two pairs of source ('s') and destination ('d') registers are specified.

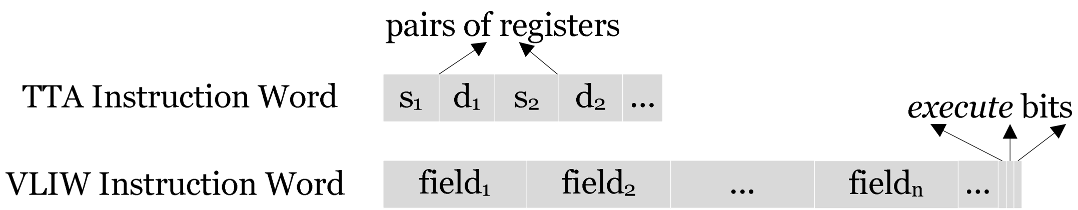

Figure 2.4: TTA and VLIW Instruction Words

The different instruction word structures differentiate the architectures in terms of programming. A VLIW instruction specifies operations explicitly and performs moves implicitly since they are "hardwired" into the FUs. The TTA specifies moves explicitly and performs operations implicitly since FUs always execute unconditionally. This implicit and explicit execution of moves and operations is shown in Figure 2.5.

	VLIW	*TTA*
Moves	Implicit (done by FUs)	Explicit (specified by programmer)
Operations	Explicit (specified by programmer)	Implicit (done automatically since all FUs execute unconditionally)

Figure 2.5: Differences between the VLIW and TTA Architectures

TTAs separate the design of the FUs and the interconnection network giving them an advantage over VLIW architectures since the interconnection network can be tailored [17]. This allows the interconnection network to be customised to the nature of communication required (e.g. creating a bus between a group of registers and a group of FUs which communicate often – see below) which leads to more optimal interconnection networks and thus superior scaling.

It is suggested in [17] that VLIW architectures require *3 * number of FUs* ports in the register file (in the worst case two read ports and one write port are required for each FU if every FU executes concurrently) which significantly restricts scalability as the number of FUs increase. However, this, as well as the proposed scaling advantages of the TTA, is negated in the context of this research for several reasons:

- Handel-C is used to implement the processors and does not support the use of buses. As a result, the interconnection network has a fixed organisation (connections from every FU to all registers that FU accesses) and cannot be tailored.
- Parallelism is not required in the current context, which means that FUs are able to share the read and write ports in the register file

because they do not require concurrent access. This significantly reduces the number of ports required to access the register file.

- The top-down FU partitioning approach (discussed in the following section) results in fewer but more complex FUs. Highly complex algorithms whose scalability is limited by the number of FUs required would be prohibitively expensive to design and would be implemented with general-purpose processors instead.

One way to characterise sequential, VLIW and TTAs is to determine the relative complexity of decoding that must be performed by hardware. Sequential architectures involve relatively complex decoding; in VLIW architectures, some of the decoding complexity is supplanted by an increase in software complexity; the shift to software is greater in TTAs.

This shift leads to TTAs being inherently complex to program because the programmer is responsible for more tasks and must schedule these correctly. A TTA must specify at least one, and frequently more, move per FU (e.g. if a FU accepts multiple operands as inputs then multiple moves are required) whereas a VLIW architecture specifies exactly one operation per FU. This leads to a more instructions in a TTA program and offsets its narrower instruction word resulting in it occupying approximately the same amount of instruction memory as a VLIW architecture. Therefore, due to the increased programming complexity and lack of advantages over a VLIW architecture, in the context of this research, the only ILP architecture considered is the VLIW.

The taxonomy shown in Figure 2.3 introduced several constraints that, along with several exceptions, narrowed the processor architectures down to either CISC-sequential or VLIW types. One additional consideration is the concept of scheduling [21], which is the activity of rearranging the instruction stream to exploit instruction-level parallelism and is therefore able to be exploited by the VLIW architecture.

Scheduling can be characterised as static or dynamic. Dynamic scheduling uses additional hardware to rearrange the instruction stream during execution, whereas static scheduling avoids this extra hardware cost by rearranging the instruction stream prior to execution. Dynamic scheduling is avoided since a minimal hardware footprint is emphasised in this research but static scheduling is allowed because, in this research, the instruction stream is known before execution and any parallelism that is present can be easily identified and exploited.

Instruction prefetching, a form of pipelining applied to the instruction cycle, is appropriate for all architectures since it can offer significant performance benefits at little hardware cost. The additional hardware is required to manage instances when control hazards occur (discussed in Chapter 4.1).

In summary, after applying the three previously mentioned constraints with the assistance of the processor architecture taxonomy shown in Figure 2.3 above, two types of architecture remain, CISC-sequential and VLIW, and these architectures are able exploit both static scheduling and instruction prefetching.

2.3 Hardware-Software Codesign

Hardware-software codesign, in general, is the simultaneous design of both hardware and software to implement a function. This is relevant when designing a software-controlled processor since it involves designing the FUs (the hardware) as well as programming (the software). Hardware-software codesign involves two tasks: hardware-software partitioning, which designs the set of FUs that will implement the application-specific behaviour, and determining the degree of control concurrency [22]. The degree of control concurrency was effectively determined in the previous section by constraining the processors, investigated in this research, to use either the CISC-sequential or VLIW architecture with static scheduling and instruction prefetching. Therefore, this section investigates the hardware-software partitioning approaches and considers which is most suitable for this research.

Hardware-software partitioning is traditionally the process of determining which portions of an algorithm will execute in hardware, and which will execute in software. However, in this research, the parallel hardware that performs the low-level image processing is assumed and the concept of hardware-software partitioning is applied solely to the design of the application-specific serial processor. The design of a processor is a hardware-software codesign problem because designing the FUs requires determining the parts of an algorithm to execute in hardware and calling those FUs requires defining the software. These activities are interrelated because the design of FUs dictates the instructions available in software.

Hardware-software partitioning can be performed using either a top-down or bottom-up approach [23]. In the context of a processor, these approaches are differentiated by the granularity of FUs at different times in the design process. Granularity is used to describe the size and number of FUs relative to each other in the processor architecture. The bottom-up approach begins with fine-grained FUs, each of which contains relatively little logic, and combines these to produce fewer but more complex FUs. The top-down approach begins with coarse-grained FUs, which contain relatively large amounts of logic, and decomposes these to produce more but less complex FUs.

These are shown in Figure 2.6 where each circle represents a FU and the size of each circle represents its complexity.

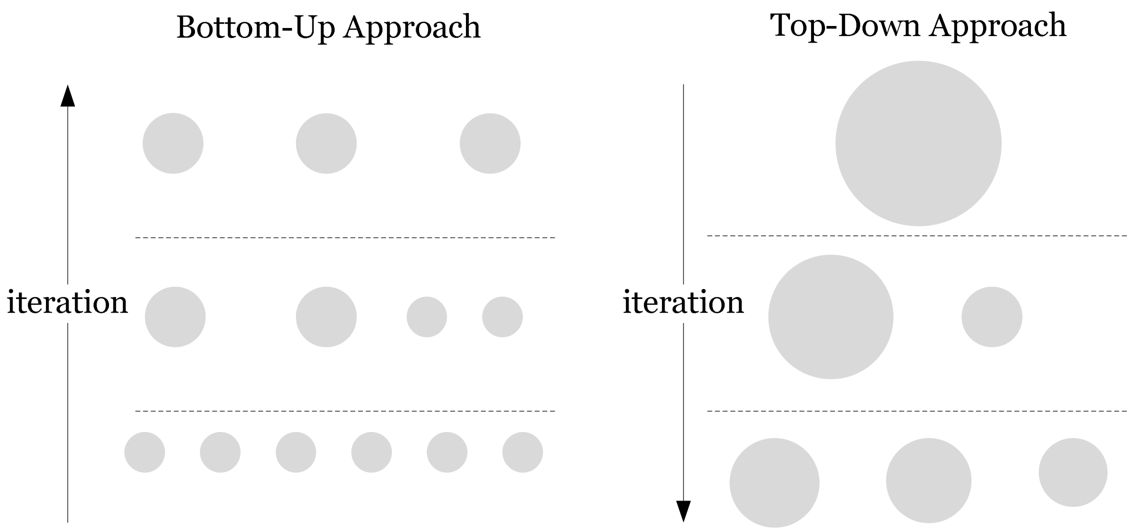

Figure 2.6: Top-Down and Bottom-Up Hardware-Software Partitioning Approaches

The bottom-up and top-down approaches above are described in their purest sense. If the initial FUs in the bottom-up approach are reused in many places in the algorithm, which the processor executes, then they will likely not be combined with other FUs. Similarly, the FUs in the top-down approach that are first decomposed are likely to be ones that are reused in multiple places in the algorithm. Overall, the goal of both approaches is to achieve a partitioning of FUs where the frequently performed tasks are represented as individual FUs and the less frequent tasks are grouped together in a single FU. Due to time constraints, exhaustive iteration is not possible and therefore the bottom-up approach finishes with more finely grained, simpler, FUs, and the top-down approach finishes with more coarsely grained, complex, FUs.

Common to both approaches is the concept of tuning the instruction granularity via design iteration [24]. This is beneficial because predicting the footprint based on changes to the architecture is inherently difficult and thus a trial-and-error approach is more appropriate.

The bottom-up approach is often applied by converting a high-level algorithm to a lower-level representation such as MIPS assembly [16, 25].

Generating assembly code using a RISC instruction set architecture, such as MIPS, is well suited to the bottom-up approach because it decomposes the high-level algorithm to one composed of simpler operations, which are more suitable for implementation as fine-grained FUs. These MIPS instructions are then grouped into basic blocks, which represent sequences of instructions where the control-flow is sequential, and are represented using data-flow graphs [20, 26-31]. Data-flow graphs are utilised because they allow common sequences of operations to be identified and implemented as FUs. This approach results in more general operations, which mask the algorithm-specific behaviour, and make it more difficult to identify application-specific optimisations.

The top-down approach examines the high-level algorithm directly and attempts to decompose it into blocks of hardware based on tasks that are performed frequently, and would therefore benefit from reuse as FUs. This approach is able to retain as much of the original algorithm behaviour as possible by having it execute as hardware within the FUs and is therefore suitable for *application-specific* processors. The result is reduced flexibility, since the FUs are less general, but also a smaller footprint and better performance since the application-specific optimisations can be more easily retained when compared to the bottom-up approach.

Therefore, the top-down approach is chosen instead of the bottom-up approach because it has the potential to create a processor with a smaller footprint and better performance, congruent with the goals of this research. This approach reduces the ability to execute a different algorithm on the same hardware but this is of little importance since the processors in this research are application-specific.

3 Development of Hardware-Controlled Processors

This chapter explores the design of hardware-controlled application-specific processors for high-level image processing. Creating hardware-controlled processors allows comparisons to be made with software-controlled processors and eases the creation of the more complex (to design) software-controlled processors by predefining much of the hardware logic that is incorporated into the FUs. Two real world image-processing algorithms, locating the robots within robot soccer and correcting for lens distortion, are analysed with the purpose of creating serial processors that occupy a minimal footprint. This is achieved primarily by minimising the amount of duplicate (or similar) hardware by maximising its sharing and reuse between different parts of the processor.

How reuse is implemented in Handel-C is important because it influences the hardware that is created. Reuse can be exploited both explicitly and implicitly. Explicit reuse represents logic as a Handel-C *function*, which creates a reusable block of hardware that can be called multiple times from anywhere in the algorithm. This is suitable when the logic being reused occurs in diverse places in the algorithm. Implicit reuse is achieved with loop constructs (e.g. *for*, *while* and *do-while*), which create a block of hardware and execute it repeatedly. This is suitable when the logic being reused is consecutive and occurs in one place in the algorithm.

A hardware-controlled processor is created in Handel-C by implementing the algorithm directly as opposed to defining the architectural components explicitly. In Handel-C, FUs are created explicitly from functions and implicitly from loop constructs. During compilation, Handel-C implicitly builds logic to control the execution of these FUs as well as other statements. This control logic is a form of FSM (Finite State Machine) with each state variable corresponding to a single executable statement within the program. Thus, a hardware-controlled processor implemented directly in Handel-C can be referred to as an IFCP (Implicit FSM-Controlled Processor).

The image-processing problems are defined textually with descriptions and mathematics. This means the creation of hardware-controlled processors do not just construct hardware but also define the algorithm. Handel-C is well suited to this since a direct implementation corresponds to an algorithmic representation. The image-processing problems are analysed from the top-down, which first identifies their major tasks. These are then broken into subtasks, and so on until the arithmetic operations and logic have been identified. Finally, once all tasks have been analysed and opportunities for reuse identified the algorithm behaviour is defined in Handel-C. The implementation details of arithmetic operations regarding their reuse and the choice (and design) of customised logic is discussed in a separate chapter since they are relevant to both image-processing problems.

Predicting the footprint savings or costs in a Handel-C implementation is inherently difficult due to the interrelated nature of the hardware and the behind-the-scenes optimisation Handel-C performs. Therefore, a heuristic trial-and-error approach has been taken in designing the IFCPs. This results in several observations that may assist in the implementation of other image processing problems on IFCPs in ways that minimise footprint.

3.1 Robot Soccer

The first real-world image-processing problem investigated in this research is to identify the members of two teams of robot soccer players from a plan-view video stream of the playing field as shown in Figure 3.1.

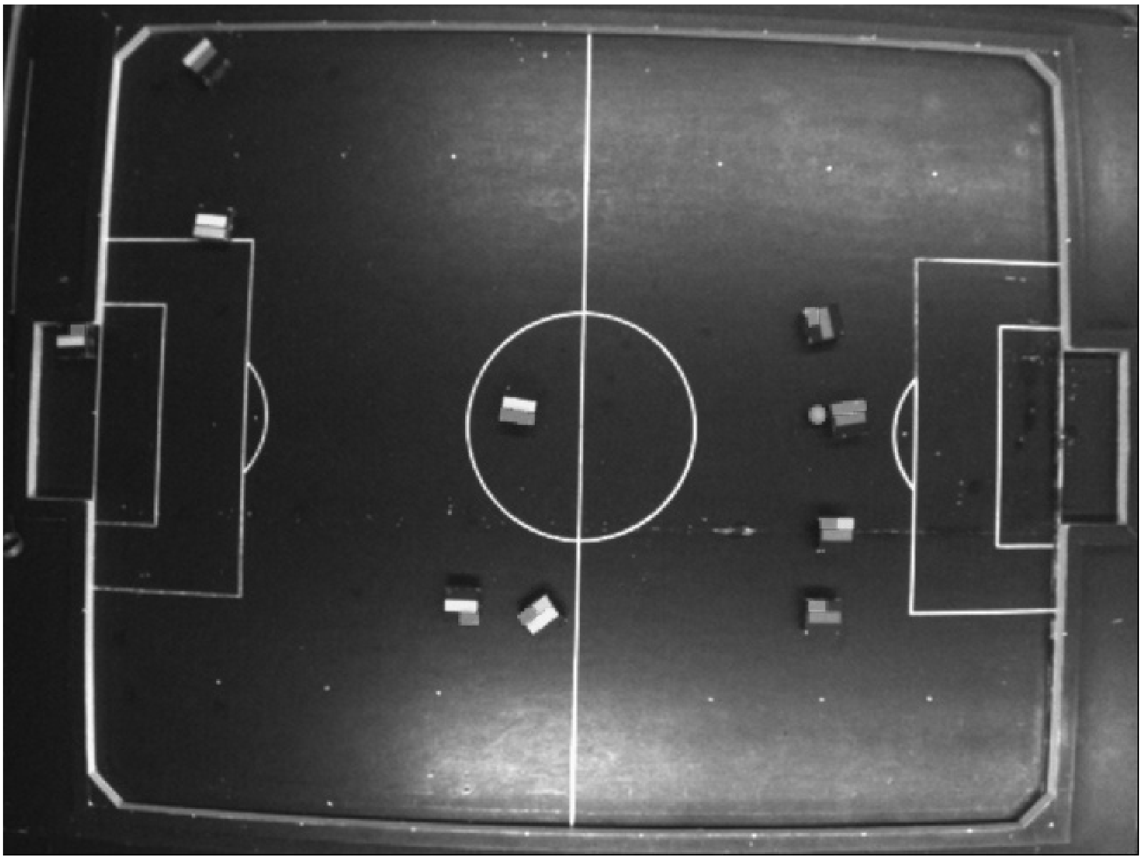

Figure 3.1: Example Frame from a Robot Soccer Video Stream

Each robot has a coloured jacket that can be used to identify the robot, and determine its location. While there are many different arrangements of patches used, for the purposes of the discussion in this research, it is assumed that the jacket is composed of four different coloured regions or patches, as shown in Figure 3.2. The direction patch, A, is always red. The team patch, T, is either blue or yellow, designating the team to which the robot belongs. The identity patches, B and C, can be any combination of green, purple, pink or cyan/aqua but never the same colour, allowing for differentiation of up to six robots per team.

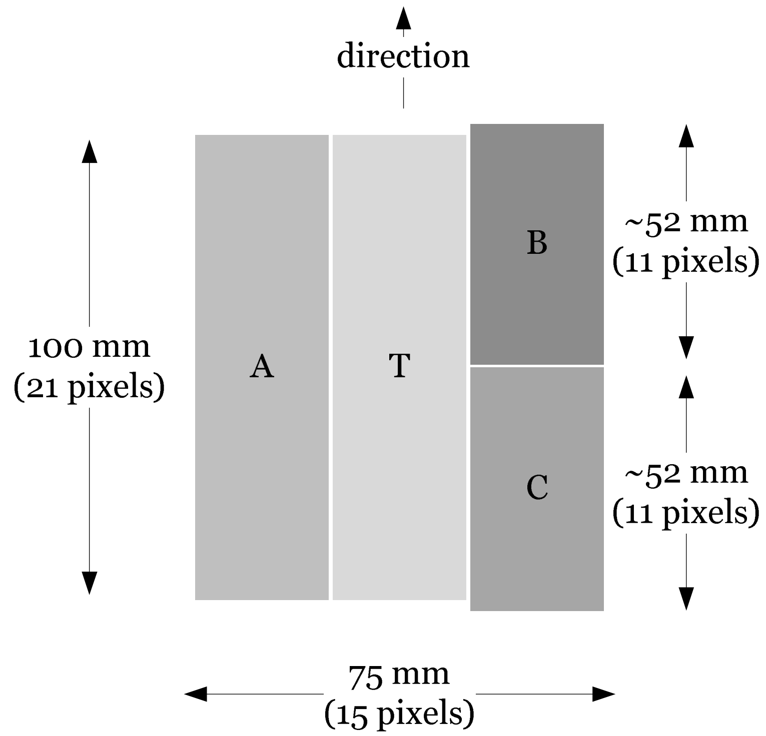

Figure 3.2: Plan View of a Robot

The algorithm for this task is composed of two stages. The first stage pre-processes the input video stream from the camera to detect coloured patches on the robots. The steps within the first stage include image filtering to reduce noise and enhance the edges, colour segmentation by classifying each pixel as belonging to a particular colour [32], and connected component analysis to identify groups of adjacent pixels that belong to the same colour [33]. These are high volume, pixel-level, low- and medium-level image-processing operations, which makes them suitable for implementation with parallel hardware as outlined previously.

In order to determine the location of each patch within the image, connected component analysis can produce the following data for each coloured patch [33]:

- the colour label associated with each region
- the number of pixels within the region
- the sum of x pixel coordinates within the region, and
- the sum of y pixel coordinates within the region

The output from the first stage is a region-data table, with one entry per coloured patch. The table does not group the patches associated with each robot, but rather provides the data in the order that the patches are first encountered during the processing. Since the implementation of parallel hardware is not considered in this research, the first stage was not actually implemented. To test the second stage, a sample of data from the first stage was derived manually as described in Appendix A.

The second stage analyses these colour patches to determine each robot's identity, orientation and position. It processes the region data table to derive a list of all robots each with the following information: team, identity (within the team), orientation (in degrees) and position (x and y components). This processing is significantly lower in volume since there are only 40 coloured patches (for five robots in each team) but the operations are considerably more complex (involving division and arctangent). These factors suggest that it is better to implement the second stage with a serial processor. Hereafter this second stage will be referred to as the *robot soccer algorithm*.

The robot soccer algorithm involves two major tasks. The first task groups the regions from the input table that belong to the same robot. The second task extracts output data for each the robot from the colour and relative positioning of those regions.

Each team patch will correspond to a single robot. An inexpensive method of associating each of the other coloured patches with a particular robot is to associate it with a robot's team patch. Since the team patch is in the centre of the robot, the distance between a patch and the associated team patch will always be under a predefined threshold for only one robot, enabling the patch to be uniquely associated with that robot. The position of each patch can be obtained from its centre of gravity as:

$$Centre\ of\ Gravity_X = Sum\ of\ X\ Values / Number\ of\ Pixels \tag{3.1}$$

$$Centre\ of\ Gravity_Y = Sum\ of\ Y\ Values / Number\ of\ Pixels \tag{3.2}$$

The first pass of the algorithm iterates over each region and calculates the centre of gravity for each patch. To avoid the need for temporary storage, which would increase the footprint of the implementation, the calculated centre of gravity values overwrite the corresponding *sum of X* and *sum of Y* values in the data table since these are not required again in the algorithm. Two divisions are required, one for each of the *X* and Y components, suggesting that division be implemented with reusable hardware in the form of a *function*.

To avoid having to store associations indicating the regions that belong together, a more iterative approach is used. This searches the region data table until a team patch is found, then searches the region data table again from the beginning looking for non-team patches associated with the team patch. Robot information is updated as non-team patches are found. The team identifier is assigned a zero or a one depending on whether the team patch is blue or yellow, respectively. The position of a robot is determined in the first pass when the centre of gravity of the team region is calculated and is simply copied to the robot's data. The identity of a robot is determined by the colour *combination* of the identity regions so when an identity region is first encountered (for a given robot) it must be stored until the second identity region is encountered, and thus requires temporary storage. This requires three bits of temporary storage (two bits to represent the 4 colours plus one bit to indicate whether a patch has been found). The robot's orientation is calculated from the position of the direction patch relative to the team patch region. This is calculated from the arctangent of the *X* and *Y* differences between the corresponding centres of gravity.

This algorithm involves much iteration since the region data table must be examined many times giving this portion of the second stage $O(n^2)$ computational complexity. Although this is relatively inefficient, it is offset by the relatively few entries in the region data table. In addition, it allows temporary storage, and therefore footprint, to be reduced by avoiding the need to store region association information.

Each pass in the two-pass approach repeats the same sequence operations multiple times suggesting the use of reusable hardware and since these sequences of operations occur consecutively these are implemented with loop constructs. The complete behaviour of the robot soccer algorithm, shown in

Figure 3.3, can be defined and implemented on an IFCP. Here each pass is implemented using a *for* loop (incrementing *i)* and within the second pass, which finds a team patch, a second *for* loop (incrementing j) is used to find non-team patches.

```
robotPtr = 0
for(i = 0 ; i != NUMBER_REGIONS ; i++)
    Regions[i].SumOfX = Divide(Regions[i].SumOfX, Regions.NumberPixels)
    Regions[i].SumOfY = Divide(Regions[i].SumOfY, Regions.NumberPixels)
for(i = 0 ; i != NUMBER_REGIONS ; i++)
    TempRegion1 = Regions[i]
    If TempRegion1.Colour == BLUE || TempRegion1.Colour == YELLOW
        Robots[robotPtr].Team = TempRegion1.Colour
        Robots[robotPtr].PositionX = TempRegion1.SumOfX
        Robots[robotPtr].PositionY = TempRegion1.SumOfY
        Execute in parallel
            OrientationFound = 0
            IdentityFound = 0
        for(j = 0 ; j != NUMBER_REGIONS &&
                                !(orientationFound && identityFound); j++)
            TempRegion2 = Regions[j]
            If TempRegion2.Colour != BLUE &&TempRegion2.Colour != YELLOW
                Calculate the patches distance from the team patch
                If it belongs to the current robot (is under the predefined threshold)
                    If TempRegion2.Colour == RED          // directional patch
                        Calculate the current robot's orientation with Arctan, the centre
                        of gravity of this patch and the robot's position
                    Else it must be an identity patch (green,purple,pink or cyan/aqua)
                        If a directional region has not been found for this robot
                            Remember this region for when the second one is found
                        Else
                            Set the robot's identity based on both identity regions found
        robotPtr++
```

Figure 3.3: Complete Robot Soccer Algorithm Behaviour

3.2 Lens Distortion

The second real-world image-processing problem investigated in this research was to characterise lens distortion. Such distortion results from a lens having a higher magnification in the centre of the image than at the edges [34], resulting in the barrel shape seen in Figure 3.4. Such radial distortion can be modelled by a radial magnification:

$$r' = r(1 + \kappa r^2) \tag{3.3}$$

where r is the radius from the centre of distortion in the distorted image, r' is the corresponding radius in the undistorted image, and κ is the lens distortion parameter.

While there are several approaches to estimating the distortion parameter (see for example [35-38]), the algorithm of Bailey [34] will be considered here. The input image contains a grid of horizontal and vertical lines as shown in Figure 3.4.

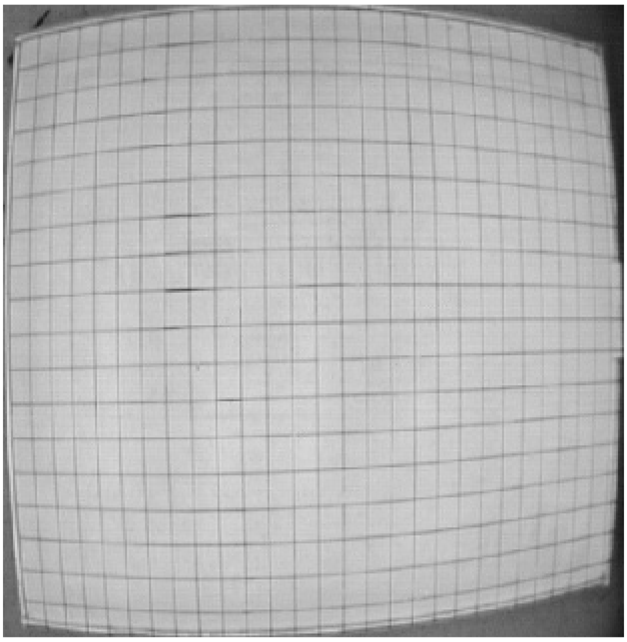

Figure 3.4: An Image with Severe Barrel Distortion

The low level pre-processing involves morphological filtering, thresholding and minimum filtering to detect each gridline within the image.

These are high-volume, pixel level operations making them suitable for implementation on parallel hardware as mentioned previously.

Minimum filtering detects the x and y coordinates of each gridline resulting in a gridline data table where each entry corresponds to the coordinates for a single gridline. Since the implementation of parallel hardware is not considered in this research, the first stage was not actually implemented. To test the second stage, a sample of data from the first stage derived manually as discussed in Appendix A.

The remainder of the algorithm fits parabolas to each of the detected gridlines and then uses the parabola coefficients to estimate aspect ratio distortion and centre of distortion. Finally, the distortion parameter, κ, of the lens is derived. Since this involves a relatively low volume of data, and within each gridline has many operations that are constrained to execute serially, it is indicative of high-level image processing operation and suggests the use of a serial processor. Due to time constraints, this research only implements the detection of the barrel distortion component, κ, and only for a single horizontal gridline, this is referred to as the *lens distortion algorithm*.

This simplified algorithm involves two major tasks. The first task fits a parabola, to a horizontal gridline's coordinates using the minimum least squares method, deriving its three coefficients, This can be decomposed into two subtasks: creating a system of linear equations and then solving those linear equations to ultimately determine the coefficients of the fitted parabola. In the second task, the lens distortion parameter is estimated from the parabola coefficients according to:

$$\kappa = \frac{-a}{c(3ac + 3b^2 + 1)} \tag{3.4}$$

3.2.1 Creating a System of Linear Equations

Each horizontal gridline in the distorted image can be represented by a parabola:

$$y = ax^2 + bx + c \tag{3.5}$$

Least squares techniques [39] can be used to determine the parabola coefficients that best fit the gridline using a set of linear equations:

$$\begin{matrix} \sum x^4 & \sum x^3 & \sum x^2 \\ \sum x^3 & \sum x^2 & \sum x \\ \sum x^2 & \sum x & \sum 1 \end{matrix} \quad \begin{matrix} a \\ b \\ c \end{matrix} = \begin{matrix} \sum x^2 y \\ \sum xy \\ \sum y \end{matrix} \tag{3.6}$$

where x and y represent the coordinates of the gridline. As this will be solved by Gaussian elimination [39], it is convenient to represent these equations as a single matrix:

$$\begin{matrix} \sum x^4 & \sum x^3 & \sum x^2 & \sum x^2 y \\ \sum x^3 & \sum x^2 & \sum x & \sum xy \\ \sum x^2 & \sum x & \sum 1 & \sum y \end{matrix} \tag{3.7}$$

This matrix is filled once during the lens distortion algorithm yet it contains a sequence of operations that is executed many times consecutively (for each point on the gridline). This suggests that these operations be implemented as reusable hardware with a loop construct. Where there are duplicate values in the unsolved matrix, such as the sum of x^2, they only need to be calculated once and copied to multiple locations. The gridline point that is retrieved from data memory at each iteration is stored in temporary registers (one for x and one for y) since it is used multiple times in different calculations. In addition, many calculations reuse other calculations (e.g. x^3 is composed of x multiplied by x^2) and to avoid repetition the results of these calculations are saved in temporary registers. The algorithm of the first task in the lens distortion algorithm can therefore be defined and is shown in Figure 3.5 (the matrix has already had all its values initialised to zero on creation). As per standard C notation the first column (or row) in the matrix is indexed with a zero, the second column (or row) with a one and so on for all locations in the matrix.

```
for(coordsIterator=0; coordsIterator<NUMBER_COORDS;coordsIterator++)
    TmpX = xCoordinates[coordsIterator];            // x
    TmpY = yCoordinates[coordsIterator];            // y
    TmpX^2 = Multiply(TmpX, TmpX);                  // x^2
    TmpX^3 = Multiply(TmpX^2, Tmp1);                // x^3
    Matrix[0][0] += Multiply(TmpX^3, TmpX);         // x^4
    Matrix[0][1] += TmpX^3;
    Matrix[0][2] += TmpX^2;
    Matrix[0][3] += Multiply(TmpY, TmpX^2)
    Matrix[1][2] += TmpX;
    Matrix[1][3] += Multiply(TmpX, TmpY);
    Matrix[2][2] += 1
    Matrix[2][3] += TmpY;
Matrix[1][0] = Matrix[0][1];
Matrix[1][1] = Matrix[0][2];
Matrix[2][0] = Matrix[0][2];
Matrix[2][1] = Matrix[1][2];
```

Figure 3.5: Filling the Matrix ready for Gaussian Elimination

Depending on the number of gridlines, and the number of points captured for each gridline, this subtask may have to be implemented with parallel hardware to meet timing constraints. However, for the purposes of this research, only a single gridline is analysed and relatively few points (11) are captured making this subtask suitable for serial implementation.

3.2.2 Solving the Linear Equations

The linear equations are solved using Gaussian elimination [39] since it involves repeated use of elementary row operations, which creates opportunities for reuse, and because it can be extended to support matrices of difference sizes (such as determining perspective distortion parameters). The goal of this is to convert the matrix to *reduced row echelon form*:

$$\begin{matrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{matrix} \rightarrow \begin{matrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \end{matrix} \tag{3.8}$$

The first operation, known as *eliminate*, sets a value to zero in the matrix. This is achieved with an elementary row addition:

$$R_i + vR_j \rightarrow R_i \tag{3.9}$$

The product of the basis row (R_j) and v is added to the target row (R_i) to derive new values for the target row. The v component is calculated such that a column in target row is set to zero:

$$v = -\frac{M_{R_iC}}{M_{R_jC}} \tag{3.10}$$

Eliminating a value involves determining v, multiplying this by each value in the basis row and then adding these products to the values in the target row. This repeated multiplication and addition, which occurs consecutively, suggests that the eliminate operation be implemented as reusable hardware with a loop construct. Distributed RAM, which is stored within the fabric of the FPGA, is used to store the matrix because it is cheaper than registers and a dual-port interface is implemented, which allows up to two read accesses concurrently. Registers are still necessary to store the calculated v as well as intermediate values from the matrix. The algorithm for the eliminate operation is shown in Figure 3.6. Here a 2-bit register is used to store the temporary value, i, so that once it has been incremented four times it will "roll over" to a value of zero. This results in the conditional expression in the *do-while* loop evaluating to false and the loop exiting.

```
i = 0
Temp1 = Matrix[target row][column]
If Temp1 != 0
    Temp2 = -Divide(Temp1, Matrix[basis row][column])    // calculate v
    Do
        Temp1 = Multiply(Matrix[basis row][i], Temp2)
        Matrix[target row][i] = Temp1 + Matrix[target row][i]
        i++
    While i != 0
```

Figure 3.6: Algorithm for Eliminating a Column

The second operation, known as *solve row*, sets the leading coefficient of a row to one (putting it into solved form) by dividing all values in the row by that of the leading coefficient. This operation is only performed on a row that is pre-solved (where all values except that in the fourth column and the leading coefficient have been eliminated) so that the number of division operations is minimised. In addition, the leading coefficient's value is not required again in the algorithm and therefore dividing this column is unnecessary. This leaves only a single division to apply to the fourth column in the row.

The matrix can only support two concurrent accesses, as mentioned previously, yet this operation requires three (two reads and a write for the division). Therefore, a register stores a matrix value temporarily. In addition, if the necessary eliminations are performed the column of a leading coefficient will always be the same as its row. Thus, only the location, which represents both the row and column, needs to be specified. This is shown by the second operation's algorithm in Figure 3.7.

```
Temp1 = Matrix[location][3]
Matrix[location][3] = Divide(Temp1, Matrix[location][location])
```

Figure 3.7: Algorithm for Solving a Row

The third operation, known as *back-substitute*, is essentially a row addition (as used in the first operation) but uses a solved row (where the column of the leading coefficient in the solved row must be the same as the column of the value that is to be eliminated from the target row) as the basis row to permit simplifications. This operation exploits the fact that the zeroes in the basis row have no effect on the target row and that the value to be eliminated in the target row is not required again in the algorithm. This results in a single calculation where the value to be eliminated and the value in the fourth column of the basis row are multiplied and their product is subtracted from the fourth column of the target row. Again, a temporary register is used since the matrix allows a maximum of two concurrent accesses. The algorithm for this operation is shown in Figure 3.8.

Temp1 = Multiply(Matrix[target row][column], Matrix[basis row][3])
Matrix[target row][3] = Matrix[target row][3] – Temp1

Figure 3.8: Algorithm for Back-Substitution

The first operation (eliminate) must be performed first since the second operation (solve row) requires rows to be pre-solved and the third operation (back-substitute) requires a solved row. Therefore, the first operation is performed on row two and three from column one and row three from column two:

$$\begin{matrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{matrix} \rightarrow \begin{matrix} ? & ? & ? & ? \\ 0 & ? & ? & ? \\ ? & ? & ? & ? \end{matrix} \rightarrow \begin{matrix} ? & ? & ? & ? \\ 0 & ? & ? & ? \\ 0 & ? & ? & ? \end{matrix} \rightarrow \begin{matrix} ? & ? & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \end{matrix} \quad \mathbf{(3.11)}$$

Row three is now in a pre-solved state and therefore the second operation is performed on this, which transforms it to a solved row:

$$\begin{matrix} ? & ? & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \end{matrix} \rightarrow \begin{matrix} ? & ? & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & 1 & c \end{matrix} \quad \mathbf{(3.12)}$$

Next, the third operation uses the third row to eliminate a column in the second row, transforming it to a pre-solved state. This second row is then solved using the second operation:

$$\begin{matrix} ? & ? & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & 1 & c \end{matrix} \rightarrow \begin{matrix} ? & ? & ? & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & 1 & c \end{matrix} \rightarrow \begin{matrix} ? & ? & ? & ? \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \end{matrix} \tag{3.13}$$

Finally, the third operation uses both the second and third rows to eliminate two columns from the first row, then this row is solved using the second operation.

$$\begin{matrix} ? & ? & ? & ? \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \end{matrix} \rightarrow \begin{matrix} ? & 0 & ? & ? \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \end{matrix} \rightarrow \begin{matrix} ? & 0 & 0 & ? \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \end{matrix} \rightarrow \begin{matrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \end{matrix} \tag{3.14}$$

The matrix is now in reduced row echelon form. Each of the operations is applied three times for the 3x3 matrix used in this algorithm, which suggests each be implemented with reusable hardware. The second and third operations occur in diverse places suggesting a function be used. The first operation is performed consecutively suggesting a loop construct yet to allow it to be executed non-consecutively, which may be useful for matrices of different dimensions, it is implemented as a function.

The algorithm for these three operations, for the 3x3 matrix used in the lens distortion algorithm, is shown in Figure 3.9. This uses explicit numbers to specify the parameters (rows and columns) of the operations yet can be extended easily for other matrices by using loop construct(s) with index variables.

Eliminate(0, 1, 0)	// basis row = 0, target row = 1, column = 0
Eliminate(0, 2, 0)	// basis row = 0, target row = 2, column = 0
Eliminate(1, 2, 1)	// basis row = 1, target row = 2, column = 1
SolveRow(2)	// row and column = 2
BackSubstitute(2, 1, 2)	// solved row = 2, target row = 1, column = 2
SolveRow(1)	// row and column = 1
BackSubstitute(2, 0, 2)	// solved row = 2, target row = 0, column = 2
BackSubstitute(1, 0, 1)	// solved row = 1, target row = 0, column = 1
SolveRow(0)	// row and column = 0

Figure 3.9: Algorithm for Gaussian Elimination

3.2.3 Calculating the Barrel Distortion Component

The third task calculates the barrel distortion parameter, κ, for the gridline from the parabola coefficients according to:

$$\kappa = \frac{-a}{c(3ac + 3b^2 + 1)} \tag{3.15}$$

This subtask is only performed once in the lens distortion algorithm, presented here, although in the complete algorithm this is calculated for each gridline and the results are averaged. Since no repetition is present, there is no opportunity for exploiting reuse suggesting this not be implemented with reusable hardware. Therefore, the algorithm of this, as shown in Figure 3.10, is carried out as sequential steps using temporary data to store the immediate results.

Tmp1 = Matrix[0][3];	// a
Tmp2 = Matrix[1][3];	// b
Tmp3 = Matrix[2][3];	// c
Tmp4 = Multiply(3, Tmp1);	// Tmp4 = 3 * a
Tmp4 = Multiply(Tmp4, Tmp3);	// Tmp4 = 3 * a * c
Tmp2 = Multiply(Tmp2, Tmp2);	// Tmp2 = b * b
Tmp2 = Multiply(3, Tmp2);	// Tmp2 = 3 * b * b
Tmp2 = Multiply(Tmp3, (Tmp4 + Tmp2 + 1));	// Tmp2 = c * ((3 * a * c) + (3* b * b) + 1)
κ = Divide(-Tmp1, Tmp2)	// κ = -a / (c * ((3 * a * c) + (3 * b * b) + 1)

Figure 3.10: Determining the Barrel Distortion Component

3.3 Implementing Arithmetic Operations

This section investigates the arithmetic operations present in the robot soccer and lens distortion problems that were identified as being suitable for implementation with reusable hardware. These operations are examined to determine if they should be implemented using custom logic or Handel-C's built-in operators, and with reusable hardware (based on if the savings from reuse outweigh the costs of multiplexing).

The implementation of these operations is investigated separately in this section because they are present in both problems. Implementing any logic as a block of reusable hardware isolates its behaviour from the rest of the algorithm also allowing it to be designed independently. However, if the precision differs between different algorithms this can justify the use of different methods to perform the same arithmetic. Although, this was not the case for the algorithms investigated in this research.

The operations that are identified as being suitable for implementation with reusable hardware are arctangent, division, multiplication, addition and subtraction. Addition and subtraction are implemented using Handel-C's built-in operators because their complexity is relatively insignificant. Division, multiplication and arctangent are implemented with custom logic since Handel-C's built-in operators for performing division and multiplication are relatively expensive to implement, and arctangent is not available in Handel-C.

Before the methods to perform arithmetic are investigated, the representation of real numbers (i.e. numbers with a fractional component) must be considered. Fixed-point representation was chosen over the more common floating-point [40] representation for several reasons:

- Fixed-point arithmetic can use the same hardware as standard integer arithmetic whereas floating-point arithmetic requires hardware that is more complex.
- The dynamic (and wider) range of numbers supported by the floating-point representation is less important for an application-specific processor whose behaviour is fixed.

A fixed-point representation treats a number as an integer number of units of a fraction rather than one. A fixed-point number format can be described using a notation involving a letter, which indicates the sign, and two numbers, the first expressing the total number of bits used, and the second expressing the number of bits used for the fractional component. For example, a fixed-point number format able to represent ten integer bits and six fractional bits is represented as U16.6 and is shown in Figure 3.11. The 16-bit integer is able to represent the number of 2^{-6}s.

$$2^9 2^8\ 2^7\ 2^6\ 2^5 2^4\ 2^3\ 2^2\ 2^1\ 2^0 .\ 2^{-1} 2^{-2} 2^{-3} 2^{-4} 2^{-5} 2^{-6}$$

Figure 3.11: Fixed Point Number Representation for U16.6

Determining the fixed-point number representation to use for the algorithms in this research requires determining the maximum absolute value, which influences the size of the integer component, and the desired precision, which influences the size of the fractional component.

The robot soccer algorithm involves two custom arithmetic operations, arctangent and division. These do not require negative values, which mean a sign bit can be avoided, and do not require a fractional component to achieve adequate precision. Its input data (*sum of x pixel coordinates* and *sum of y pixel* coordinates) can be represented with U16.0. This is because the algorithm supports an image resolution of 640x480 and does not involve operations that increase the maximum value. It is assumed the soccer field boundaries are visible at the edges resulting in the actual playing area for the robots having a resolution estimated to be approximately 620x460. Given this, and the maximum number of pixels in a patch being 105 (21 * 5), the maximum possible *sum of coordinates* value is 65100 (620 * 105 = 65100) requiring 16 bits (2^{16} = 65536). The robot soccer algorithm also involves other input data but this is not used for the arithmetic operations and is discussed in Appendix A where the complete input data is provided.

The lens distortion algorithm involves two custom arithmetic operations, multiplication and division. Its input data (gridline coordinates) can be represented with U10.0 since it specifies coordinates from an image with a resolution of 640x480 ($2^{10} > 640$). However, the use of repeated multiplication and the potential for up to 639 points per gridline leads to a maximum value of:

$$Maximum\ Value = \sum_{i=0}^{639} i^4 \approx 2^{45} \tag{3.16}$$

which requires 45 bits to represent. This means the lens distortion algorithm requires a relatively large U45. fixed-point number representation. This is expensive to implement because it requires many iterations when performing arithmetic since the number of iterations is proportional to the size of the data. Therefore, two techniques are applied to the input data to reduce the size of the maximum value.

The first translates all coordinates so the origin is at the centre of the image instead of being at the bottom-left. Thus, x values range from -320 to +319 and y values range from -240 to +239, this introduces the need for a sign bit. The maximum value is now:

$$Maximum\ Value = \sum_{i=-320}^{319} i^4 \approx 2^{41} \tag{3.17}$$

which requires 42 bits to represent when including the sign bit.

Next, normalisation is applied by dividing all values by 256 (8 bits) giving a range of -1.25 to +1.25 for x values and -0.9375 to +0.9375 for y values. The maximum value is now:

$$Maximum\ Value = \sum_{i=-320}^{319} (\frac{i}{256})^4 \approx 315 \approx 2^9 \tag{3.18}$$

which requires 10 bits to represent when including the sign bit yet to maintain precision a fractional component must also be included. The normalisation has essentially shifted the bits to the right of the decimal point by 32 places (since the maximum values are taken to the fourth power and a normalisation of 8 bits is applied, $4 * 8 = 32$) resulting in a number format of S42.32 instead of S42.0. Although there is no size difference between these two number formats, the truncation of the fractional component in S42.32 can be performed with a minimal loss to precision. In addition, many of the values calculated for the matrix shown in Equation 3.7 do not require the number of bits that the maximum possible values do and therefore modest truncation does not influence their precision.

During experimentation, a suitable level of truncation that delivered sufficiently accurately results was found to be 16 bits leaving a 16-bit fractional component and resulting in a number format of S26.16. However, one of the matrix values calculated in Equation 3.7 must represent the number of coordinates for a gridline, potentially as much as 640, and therefore the non-fractional component must be able to store a 10 bit unsigned value (since $2^9 < 640$ and $2^{10} > 640$). This resulted in the number format increasing slightly to S27.16.

Given the fixed-point number representations for robot soccer and lens distortion, the methods of arithmetic are now investigated.

The CORDIC (Coordinate-Rotation-Digital-Computer) [41] method is used to calculate arctangent and division because it ensures a relatively small footprint, at the cost of performance, which is congruent with the goals of this research.

CORDIC algorithms are capable of calculating hyperbolic and trigonometric functions as well as exponential, logarithm, square root, multiplication and division [41, 42]. The routines are iterative, performing a series of shifts and additions to achieve the desired result. Their ability to be implemented with relatively simple hardware is due to the use of shifters, which multiply and divide by powers of two [43].

The division and arctangent algorithms, implemented with the CORDIC method, are shown in Figure 3.12 and Figure 3.13 respectively. Some operations are performed in parallel for two reasons. The first is to improve performance when it does not incur any additional hardware complexity. The second is to avoid having to use temporary storage if a register needs to have its value read and a new value written to it.

```
Result = 1 << 10
While Result[29] != 1
    Execute in parallel
        Temp = Input_A << 1
        TempResult = Result << 1
    If Input_A < 0
        Execute in parallel
            Input_A = Temp + Input_B
            Result = TempResult
    Else
        Execute in parallel
            Input_A = Temp - Input_B
            Result = TempResult + 1
Return Result[26:0]                                  // S27.16
```

Figure 3.12: Division Algorithm

The division algorithm shown here is that used in the lens distortion algorithm and as such involves S27.16 representation. The version used in robot soccer is similar except for the conditional expression governing the while construct (*While !Result[20]*) and the returned data (*Return Result[18:3]*).

```
Result = 90 << 6                              // shifted to put into S13.6 form
For (i = 0 ; i < NUMBER_OF_ARCTAN_ITERATIONS ; i++)
    Execute in parallel
        Temp_A = Input_A << i
        Temp_B = Input_B << i
    If Input_B > 0
        Execute in parallel
            Input_B = Temp_B - Input_A
            Input_A = Temp_A + Input_B
            Result += ArctanLUT[i << 1]
    Else
        Execute in parallel
            Input_B = Temp_B + Input_A
            Input_A = Temp_A - Input_B
            Result += ArctanLUT[(i << 1) + 1]
Return Result[21:6]                           // U16.0
```

Figure 3.13: Arctangent Algorithm

The arctangent algorithm uses a lookup table filled with precalculated values according to:

$$Angle = Arctangent(2^{-i}) \qquad \textbf{(3.19)}$$

where i is the current iteration in the algorithm.

To minimise hardware complexity the same set of numbers are duplicated, made negative and interleaved with the original set so only the addition operation is necessary in the algorithm. This results in the lookup table storing the following values (in this order): +45.0°, -45.0°, +26.6°, -26.6°, +14.0°, -14.0°, and so on. These values are rounded to one decimal place here but are represented using a 6-bit fractional component (S13.6) to achieve an adequate precision (although the algorithm still returns U16.0).

The arctangent and division algorithms, used by robot soccer, are relatively similar, as supported by [44], and leveraging this similarity, by implementing both with the same block of hardware, may provide footprint savings. However, in this research, the additional control logic required to support the functionality of both algorithms does not offset the savings made through reuse and therefore these are implemented as separate blocks of hardware.

Multiplication, shown in Figure 3.14, was implemented with a method similar to CORDIC, by repeatedly performing additions and shifts, and subsequently had similar characteristics in that it occupies a relatively small footprint and requires a number of iterations.

```
Result = 1 << 17
While Result[44] != 1
    If most significant bit (but not sign bit) of Input_A == 1
        Result = Result + Input_B
    Execute in parallel
        Input_A <<= 1
        Result <<= 1
Return Result[43:17]                                        // S27.16
```

Figure 3.14: Multiplication Algorithm

The precision required by robot soccer and lens distortion algorithms depends on the number of iterations their CORDIC routines perform. In general, CORDIC routines return one bit of accuracy (including the bits in the integer component) for each iteration [43] implying that the number of iterations should be equivalent to the size of the fixed-point number involved. This suggests 16 and 27 iterations for the robot soccer and lens distortion algorithms respectively.

The arithmetic in the lens distortion algorithm must support the multiplication and division of negative values. The input parameters are converted to positive values to avoid having to implement the logic required to support negative values during multiplication or division. Then, the final calculated value is then converted back to its negative equivalent if appropriate. The algorithm for performing this sign inversion is shown in Figure 3.15. Note the difference between the unary negation operator (-), which inverts a values sign, and the bitwise *not* operator (~), which inverts all bits in a value.

```
signOfResult = 0
If Input_A < 0
    Input_A = -Input_A
    signOfResult = ~signOfResult
If Input_B < 0
    Input_B = -Input_B
    signOfResult = ~signOfResult
... execute division or multiplication with two positive inputs ...
If signOfResult == 1
    Result = -Result
... return the result as usual or save it somewhere ...
```

Figure 3.15: Handling Signed Arithmetic

This section has determined which arithmetic involved in the robot soccer and lens distortion algorithms is implemented using custom hardware. Then, a number representation for real numbers has been specified (fixed-point, U16.0 and S27.16 for robot soccer and lens distortion respectively) and finally the algorithms performing the arithmetic have been defined.

3.4 Summary

This chapter investigates how the IFCP implementations of robot soccer and lens distortion are created in a way that minimises the footprint. This involves balancing hardware reuse with the implicit increase in multiplexing (as blocks of hardware are reused, more data is input into such blocks requiring more and larger multiplexors to select from the appropriate inputs).

Both algorithms are composed of a set of tasks with subtasks underpinning them and at the lowest level the arithmetic operations. It is important to identify (and define) this hierarchy as the sequences of operations within the subtasks are implemented using reusable hardware. The arithmetic underpinning robot soccer and lens distortion is determined as the problems are analysed. Addition and subtraction are simple enough not to justify a custom implementation or reuse. This is because the savings made from their reuse would be offset by the multiplexing that is introduced. Multiplication, division and arctangent are both complex enough to justify a custom implementation and to be reused. It is important to identify the arithmetic required by an image-processing problem as it is being analysed so the numbers of diverse inputs, and therefore the amount of reuse it can exploit, can be determined gradually.

The robot soccer algorithm involves two major tasks: the first pass and the second pass, and two subtasks that support the second pass: finding a team region and then finding other regions. Although division and arctangent underpin several subtasks, the algorithm is composed primarily of loop and branch constructs and relatively little arithmetic.

The lens distortion problem involved three major tasks: filling the matrix, Gaussian elimination and determining the distortion parameter and three subtasks support that support Gaussian elimination task: eliminating columns, scaling the rows and back-substituting. Underpinning many of these tasks and subtasks is multiply and divide arithmetic. This algorithm reuses the arithmetic operations much more than robot soccer algorithm.

The implementation of the robot soccer and lens distortion algorithms on IFCPs carries with it several limitations. The first is that explicitly reusing any common functionality, such as blocks synthesised from functions or registers written to from multiple locations, requires the use of multiplexers to govern the inputs for the reused block. Multiplexing, which is present in large quantities in the lens distortion algorithm, is unavoidable because the more a block of hardware is reused, the more likely it is to involve diverse parameters and therefore require a multiplexer to govern the input of those parameters. The hardware constructed to perform multiplexing is sufficiently complex that it discourages such reuse. The second limitation is that the control logic, which is present in large quantities in the robot soccer algorithm, is not only "hardwired" but also is not reused and grows with the complexity of the algorithm. These limitations are addressed in the following chapter, which investigates software-controlled processors.

4 Development of Software-Controlled Processors

This chapter explores the development of software-controlled application-specific processors for high-level image processing. These represent the algorithm behaviour as a sequence of software instructions allowing it to be modified without a re-synthesis of the processor on the FPGA. Software-controlled processors also address the limitations of IFCPs (Implicit FSM-Controlled Processors) by using the same FUs (Functional Units), but scheduling these through a sequence of instructions defined in software. This minimises the use of expensive multiplexers because more operand data comes from instruction memory instead of diverse locations in the architecture. In addition, software-controlled processors involve a (mostly) constant overhead in terms of the control logic regardless of the complexity of the algorithm, and shift much of the control logic from the fabric of the FPGA to instruction memory.

Software-controlled processors are created by examining the architecture of the previously designed IFCPs and replacing their implicit FSM controller with an explicit software controller; and are therefore referred to as ESCPs (Explicit Software-Controlled Processors). Design repetition is minimised by reusing the same FUs from the IFCP in the ESCP but specifying these explicitly. This leaves the creation of the controller and instruction memory as well as the programming of instruction memory as the remaining design tasks. An ESCP may also require additional FUs to implement branch constructs (*if-else* and *switch*) and loop constructs (*for, while, do-while*) that are implicit within the IFCP. This chapter investigates these tasks in the following sections:

- Architecture and controller design
  This investigates the CISC-sequential and VLIW architectures, as well as instruction prefetching. An evaluation of these architectures is performed and the most appropriate combination is chosen as the basis of the ESCPs.

- Functional unit partitioning
  This investigates the problems associated with recognising the explicit and implicit reuse exploited in the IFCPs, and associated issues. A partial methodology is developed to assist the creation of ESCPs for other image-processing problems.
- Tuning the architecture
  Here the reuse of fields in the instruction word is explored as well as the combining and decomposing of FUs.
- Programming
  This explores ways to simplify programming ESCPs by creating symbolic abstractions similar to assembly languages.

As was explained in the previous chapter, the prediction of footprint differences between implementations is inherently difficult and therefore a heuristic trial-and-error approach has been taken in designing the ESCPs as well.

4.1 Architecture and Controller Design

This section defines the architecture, investigates instruction prefetching and selects from one of the two architecture types that are considered in this research, CISC-sequential and VLIW.

The CISC-sequential architecture is shown in Figure 4.1 with a single FU (in practice there are many FUs with each implementing an instruction) to illustrate communication between the components in the architecture. Here, an IR (Instruction Register) and PC (Program Counter) are used to store the currently executing instruction word and its address respectively. The IR buffers the next instruction for instruction prefetching (discussed below) and the PC, like all registers in the register file, is accessible by all FUs. In addition, the design space is further limited by constraining operands that specify instruction addresses to use absolute addressing. This simplifies programming and the hardware required to implement instruction prefetching (discussed below).

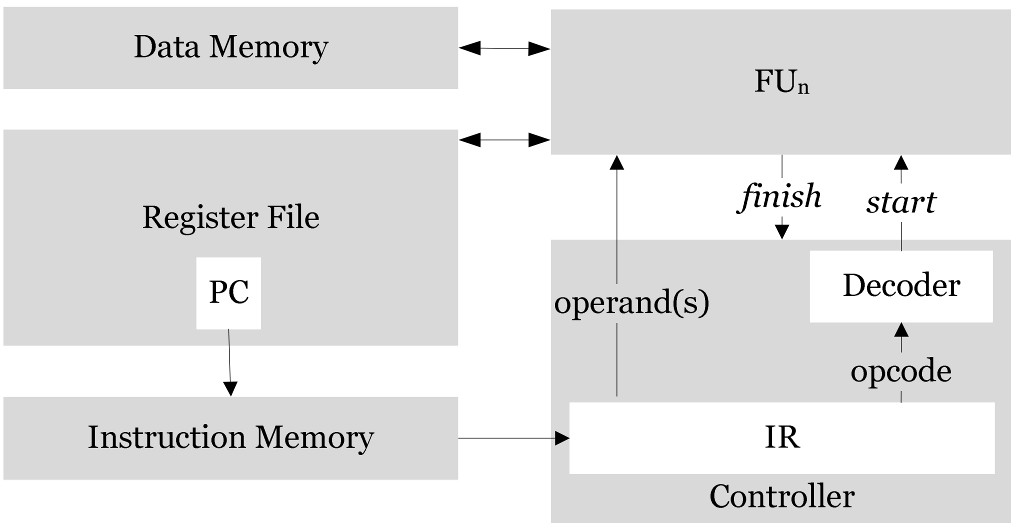

Figure 4.1 CISC-sequential Architecture

In Handel-C, the software controller is implemented with two loop constructs that iterate continuously until the *terminate* register is set to true (with the terminate instruction). The FUs are implemented as *macro procedures*, which compile to a block of hardware that accept a *start* signal, input values and return a *finish* signal. During each instruction cycle, a *switch* statement decodes the *opcode* field from the instruction register and determines which FU to execute. A major disadvantage with the CISC-sequential architecture is that a switch statement is implemented with a decoder, which is relatively costly in terms of footprint. This is shown in Figure 4.2 where a 2-bit *opcode* is input into the decoder and the corresponding *start* signal is sent to one of four FUs.

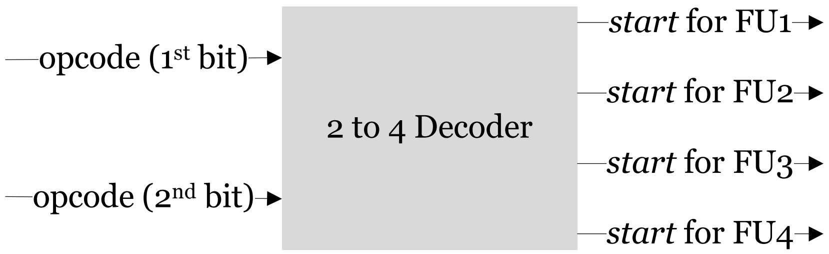

Figure 4.2: Example Decoder

Whilst the CISC-sequential architecture originates from an architectural style where a single general purpose FU is used (the arithmetic logic unit), the VLIW architecture originates from an architectural style where multiple FUs are used (one for each separate function) [17]. This suggests the VLIW architecture may be more appropriate since the processors involved in this research implement multiple FUs to be able to exploit reuse.

The VLIW architecture simplifies the decode phase by allocating one bit, known as the *execute* bit, of the instruction word to each FU to determine if that FU will execute. An example is shown in Figure 4.3 where the CISC-sequential instruction word has a single, relatively large, opcode whereas the VLIW instruction word uses many execute bits. The VLIW instruction word refers to the bit ranges that store operand data, as fields. The boundaries of these fields can differ for each instruction. However, due to time constraints, this is not explored and each field uses a fixed bit range for all instructions.

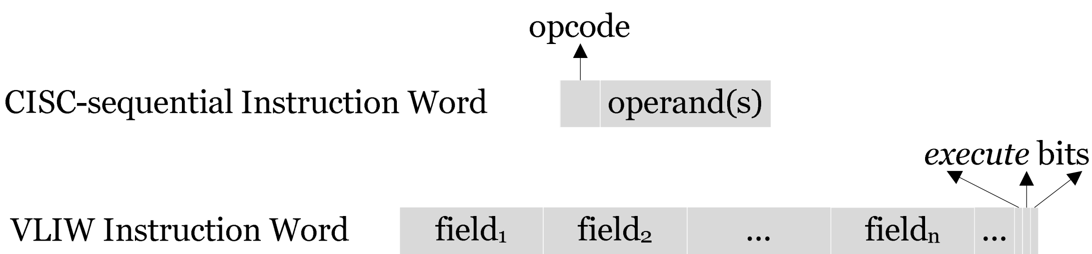

Figure 4.3: CISC-sequential and VLIW Instruction Words

The VLIW architecture also allows instruction-level parallelism to be exposed since concurrent execution of FUs can be achieved by simply setting two or more *execute* bits to be true in the same instruction word. However, the VLIW architecture requires more width in the instruction word, which results in instruction memory incurring a larger a footprint. As an example, a VLIW architecture containing 16 FUs requires 16 bits in the instruction word to control their execution whereas the corresponding CISC-sequential architecture requires only 4 bits ($\log_2 16 = 4$).

The number of fields in a VLIW instruction word represents the total amount of data that can be concurrently passed to FUs during the execute phase. Since ILP (Instruction-Level Parallelism), and therefore concurrent execution of FUs, is not explored in this research (due to time constraints), the number of fields in the instruction word need only be sufficient to specify the operand data for a single FU. However, even if ILP is exploited many of the fields in the instruction word can still be shared because many FUs would use the same operand data.

An example of the VLIW architecture with a single FU (again, in practice there are many FUs with each implementing an instruction) is shown in Figure 4.4. This illustrates how the VLIW architecture avoids the decoder present in the CISC-sequential architecture by simply passing the *start* signal directly to the FSM that controls each FU. It is up to the controller to determine which fields are passed as operands to each FU.

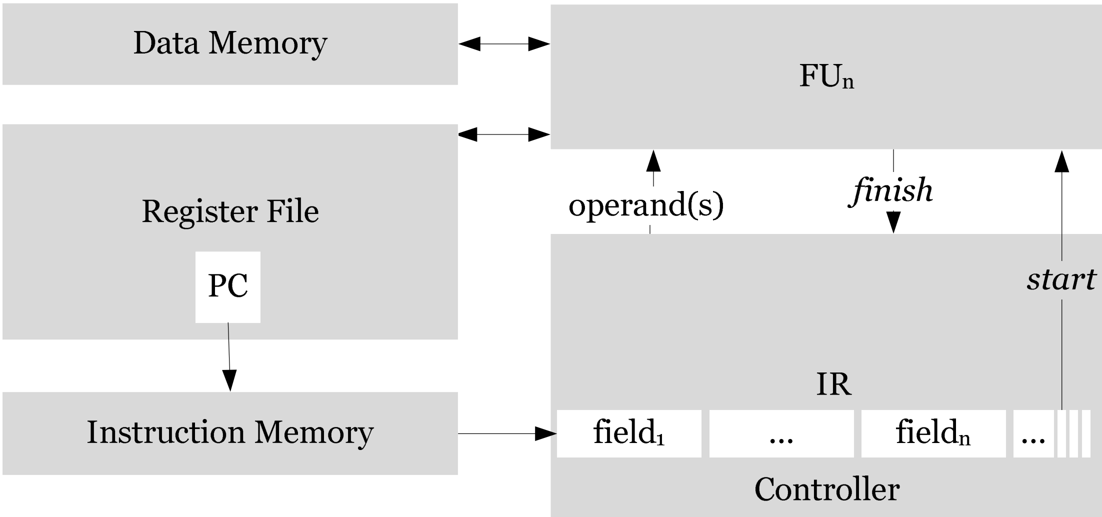

Figure 4.4: VLIW Architecture

The hardware savings a VLIW architecture makes by avoiding the decoder may offset the hardware costs of the wider instruction word and (as mentioned above) the VLIW architecture may be more suitable than the CISC-sequential architecture when multiple FUs are involved (likely to be the case in this research). In addition, the VLIW architecture can exploit ILP with only minimal changes to hardware (e.g. adding extra fields to the instruction word), which may be useful if explored in the future. Therefore, the VLIW architecture is chosen rather than the CISC-sequential architecture for the implementation of ESCPs in this research.

The software controller is only responsible for executing FUs and passing operand data, whereas the FUs themselves will likely involve considerable complexity, especially due to the use of the top-down approach, which produces relatively complex FUs. This means the hardware within the FU is still controlled implicitly by an FSM and therefore the ESCPs essentially use a hybrid of explicit software and implicit FSM control. This is beneficial because the explicit software control still provides the desired flexibility at the algorithm level, and the implicit FSM control avoids the need for low-level control design within each FU.

An ESCP involves a performance overhead because each instruction must be fetched before it can be executed. Instruction prefetching reduces this overhead by carrying out the fetch phase of the next instruction while the execute phase of the current instruction is under way. If all FUs execute in a single clock cycle then instruction prefetching can reduce the overall execution time by up to 50% since an instruction fetch phase also takes a single clock cycle. However, in practice this will not happen for two reasons. Firstly, the use of the top-down hardware-software partitioning approach creates relatively coarse-grained FUs and these are likely to require multiple clock cycles to execute. Secondly, the next instruction cannot always be prefetched without causing control hazards [16, 45] since the PC can be written to by loop and branch FUs. Potential solutions for dealing with control hazards while still exploiting instruction prefetching are:

- Speculative execution
  This uses branch prediction to determine which branch to take and if the prediction turns out to be incorrect then any subsequent writes are undone. The simplest type of speculative execution is trivial prediction, which simply assumes a branch is never taken and fetches the next instruction word from PC + 1.
- Concurrent execution
  This concurrently fetches and executes both of the two potential instructions and may require the duplication of hardware if the instructions (and corresponding FUs) are the same or if the FUs share the same temporary registers. These FUs are executed in parallel while the branch destination is being calculated but only the results for the instruction that is the eventual branch destination are written.
- Stalling.
  This stalls the instruction fetch pipeline when the currently executing instruction is a control instruction, able to write to the PC. Only when instructions do not write to the PC is the next instruction fetch performed.

Although stalling offers the smallest potential performance benefits of the three ways to deal with control hazards it is the simplest, involving the smallest hardware cost, and therefore is used in the ESCPs in this research.

The first step in implementing instruction prefetching is identifying control instructions, which can potentially modify the PC, and non-control instructions, which cannot. The next instruction should only be fetched when all control instructions have finished executing. Handel-C code that implements a partial controller is shown in Figure 4.5. Here, two non-control instructions, *NON_CONTROL_FUNC1* and *NON_CONTROL_FUNC2*, may execute in parallel along with a sequential block. The sequential block may also execute two control instructions, *CONTROL_FUNC1* and *CONTROL_FUNC1*, in parallel. This ensures that all control instructions finish executing before the next instruction is fetched and it gives the programmer the freedom to execute any functional units concurrently (although it would be bad practice to execute control instructions concurrently since both may write to the PC).

```
par
{
      if(IR[OPCODE_NON_CONTROL_FUNC1]) { NON_CONTROL_FUNC1(); }
      if(IR[OPCODE_NON_CONTROL_FUNC2]) { NON_CONTROL_FUNC2(); }

      seq    // Instructions which may change the PC (control instructions)
      {
            par
            {
                  if(IR[OPCODE_CONTROL_FUNC1]) { CONTROL_FUNC1(); }
                  if(IR[OPCODE_CONTROL_FUNC2]) { CONTROL_FUNC2(); }
            }
            par
            {
                  IR = instructionMemory[PC];
                  PC++;
            }
      }
}
```

Figure 4.5: Instruction Prefetching for Multiple Instructions

If an algorithm involves no control instructions then the instruction fetch and PC increment will be executed in parallel with the other non-control instructions.

Two experiments were used to evaluate the effect instruction prefetching has on footprint and performance. The first is a contrived experiment that executes a set of 32 FUs that implement no logic and consume a single clock cycle. This exploits a larger proportion of control instructions (50%) than what would be expected in real world image-processing problems, which has the effect of decreasing the significance of performance savings from instruction prefetching, and giving it unrealistic performance benefits. However, this is offset by it implementing FUs that consume a smaller number of clock cycles (one) than what would be expected, which increases the significance of performance savings made from instruction prefetching. Nevertheless, the performance differences of the contrived experiment should not be considered representative of high-level image processing and it is created primarily to investigate the footprint differences with and without instruction prefetching. The second experiment is a preliminary implementation of the robot soccer algorithm. This is used because its FUs are from a real-world image-processing algorithm, therefore its performance and footprint differences are more likely to be representative of high-level image processing than that of a contrived experiment. Table 4.1 gives the results showing the percentage differences for both implementations, relative to the version without instruction prefetching. The complete tabulated results can be found in Table B.1 from Appendix B.

	Footprint	*Performance*	
		Execution Time	*Clock Cycles*
Contrived Experiment	-18.8%	-44.5%	-23.4%
Intermediate Robot Soccer Implementation	-0.5%	-23.2%	-19.2%

Table 4.1: Instruction Prefetching Evaluation Differences

These results show a decrease in the footprint of both experiments when instruction prefetched is implemented. Instruction prefetching requires additional control logic and therefore the reduction in footprint may be attributed to a reduction in the size of hardware (albeit with an increase in its complexity).

The difference in execution time, which takes into account the maximum attainable clock frequency and clock cycles consumed, is significant. However, the maximum attainable clock frequency is not necessarily what will be used by a processor (for example, the pixel clock frequency may be used) and therefore it is more accurate to compare based on clock cycles. Nevertheless, the difference in performance, in terms of clock cycles, is still significant. Therefore, due to both smaller hardware and significant performance benefits, instruction prefetching is used with the VLIW architecture by all ESCPs in this research.

Given this, the first step in creating an ESCP is implementing a skeleton VLIW architecture that supports instruction prefetching using stalling. The controller components (PC, IR, instruction memory and terminate register) are first implemented followed by two nested loops, which perform the instruction cycle. The exterior loop unconditionally executes the inner loop, which performs the instruction cycle until the terminate register is set. Once the terminate register has been set the processor stops executing the inner loop but the outside loop will keep executing. This allows an ESCP to be reset after termination by resetting the terminate register and PC. The widths, depths and bit positions of the controller components are left undefined until the software for the processor is finalised, and are described in Chapter 4.4.

Two FUs are also implemented during the creation of the architecture skeleton because they are usually independent of the algorithm the ESCP will execute. The first of these is to support the *terminate* instruction which halts the instruction cycle by setting the *terminate register* to true. The second implements a *jump* instruction that allows the PC to be set to a specified value. Most high-level image-processing algorithms involve branches or loops, which would necessitate such an instruction.

The data memory is identical to that used by IFCPs because its format and content is determined by the algorithm, which remains the same regardless of whether the processor is hardware- or software- controlled.

An IFCP implicitly specifies a “register file” through its collection of registers (made from the flip-flops in the logic blocks), distributed RAM (using the lookup table in the logic block as memory) and block RAM. This same register file is used in the ESCP with operand data used to differentiate between the data items used in an FU. If additional temporary data is introduced in the ESCP (e.g. for the results returned from functions implementing FUs) this is stored in additional registers since the amount of additional data is likely to be small.

4.2 Functional Unit Partitioning

The previous section described the basic architecture whereas this section investigates the partitioning of an actual algorithm, represented by the IFCP implementation of the image-processing problem, over the FUs that comprise the ESCP and create what is referred to as the initial ESCP. Overall, the goal of this section is to replicate the same reuse that is present in the IFCP but to do so using explicit FUs.

In an IFCP, explicit reuse is found in functions and therefore these are immediately implemented as FUs. In congruence with the top-down approach, loop and branch constructs are implemented in their entirety in FUs (instead of being decomposing to multiple, more primitive, FUs). However, a major complication is that if a loop or branch contains a function call, and this is implemented in its entirety as a single FU, it will result in a FU executing another FU. FUs are not permitted to call each other because this diverges from the principles of a processor architecture, which uses FUs to implement instructions and controls these FUs through the programming of instruction memory. Therefore, loop and branch constructs from the IFCP that contain function calls must be decomposed over multiple FUs. This implements the logic before and after the function call as individual FUs allowing the function call in between to be separate, as an instruction in software.

The loop and branch constructs are decomposed to multiple, and by implication simpler, MIPS [25] instructions. MIPS is a RISC ISA (Instruction Set Architecture) and therefore its instructions, and FUs to implement these, are relatively simple, which is incongruent with the top-down approach. However, it is a simple and familiar ISA, which helps to create an initial implementation quickly, and is intended for use as a starting point to be refined rather than a final partitioning of FUs. The FUs created to support decomposed constructs are not reused for other constructs at this stage since the combining of these is investigated during architecture tuning (discussed in the next section).

Each loop and branch construct requires a minimum of two FUs, one to execute the conditional expression (essentially a conditional *jump* instruction) and one to execute its block of code within the construct. In addition to these, *for* loops may require an additional FU for incrementing or decrementing a register (or other data item) that is used by their conditional expression. FUs to initialise registers (or other data items) may also be required.

An additional concern is that an IFCP often makes use of a function's ability to pass return data back to where the function was called. In an ESCP, the controller executes a FU and does not have the ability to handle return data. Thus, all FUs corresponding to functions in the IFCP that return data are modified to write their result to a temporary register. This requires an additional instruction to shift the data from the temporary register to the desired location. This decreases performance due to the extra instruction and modestly increases hardware complexity due to the extra register and instruction control logic.

Applying this implementation strategy to robot soccer is complicated because it involves multiple nested loops and branches and at the innermost level calls a function. This results in the inner most construct having to be decomposed, which in turn necessitates the next outer loop having to be decomposed, and so on for all nested constructs. Overall, the robot soccer algorithm requires a total of 24 FUs, including the sole function (division), as well as the *terminate* and *jump* FUs.

The lens distortion algorithm involves fewer nested loops or branches than the robot soccer algorithm but it is complicated by functions that execute other functions (e.g. the function implementing back-substitution uses the function implementing multiplication). Such functions require decomposing into multiple simpler FUs. The lens distortion algorithm requires a total of 22 FUs, including five functions (division and multiplication, and three for Gaussian elimination), as well as the *terminate* and *jump* FUs.

FU partitioning is a process of identifying and implementing the reuse from an IFCP in an ESCP. The strategy employed is to implement the explicit reuse (functions) initially and then to implement the implicit reuse (loop constructs). Loops and branches may require decomposition to multiple FUs if they contain function calls. This is likely to be the case for all high-level image-processing problems because their execution is generally unordered, involving significant reuse (implemented implicitly with loop constructs) and control logic (implemented with branch constructs).

4.3 Tuning the Architecture

This section investigates creating what is referred to as tuned ESCPs from an initial ESCP (discussed in the previous section). Tuned ESCPs are created by reusing fields in the instruction word and refining the FUs that implement the loop and branch constructs from the IFCP to achieve a smaller footprint.

When FUs are partitioned for the initial ESCP, a separate field in the instruction word is allocated for every operand required for each FU. This simplifies development but results in a wide instruction memory. A field can be shared if it never specifies multiple independent operands concurrently. Concurrent access to different operands is required in two instances: when a single FU requires more than one operand, or when multiple FUs, each of which accepts at least one operand, execute concurrently. The reuse of fields in the instruction word restricts the flexibility of the processor to support instruction-level parallelism by limiting the amount of concurrent operand data that can be supported.

The initial ESCP implementing the robot soccer algorithm has two fields in its instruction word. Field_1, which is six bits wide, allows a PC address for use by control instructions. Field_2, which is one bit wide, allows the division FU to have the choice of one of its parameters specified in software. As these are not used concurrently, they may be merged field into a single 6-bit wide field. As a result, the instruction word is reduced from a total size of 31 bits (six bits for field_1, one bit for field_2 and 24 execute bits) to 30 bits (six bits for the merged field and 24 execute bits). The effects of merging the fields are shown in Table 4.2, with instruction memory stored in both block RAM and distributed RAM. The complete tabulated results can be found in Table B.2 from Appendix B.

Location of Instruction Memory	*Footprint Difference*					
	Total	*Other Logic*	*RAM*			
			Dual-port	*32x1*	*16x1*	*Block*
Block RAM	-1.9%	-2.2%	0.0%	0.0%	0.0%	0.0%
Distributed	-2.0%	-2.3%	0.0%	-1.2%	0.0%	N/A

Table 4.2: Reductions in Footprint for Instruction Word Field Merging for Robot Soccer

The distributed RAM version shows a small decrease in 32x1 RAM from the reduction in instruction word width. The block RAM version shows no reduction in RAM because the available width of block RAM means that savings cannot be made. Both versions also show a reduction in footprint from the registers and other logic as a consequence of field merging.

The initial ESCP implementing the lens distortion algorithm has considerably more fields in its instruction word: field$_1$ (seven bits), field$_2$ (two bits), field$_3$ (two bits), field$_4$ (five bits) and field$_5$ (five bits) as shown in Figure 4.6.

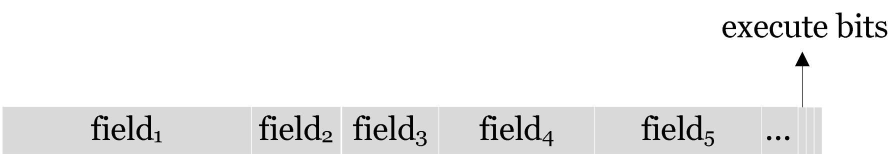

Figure 4.6: Lens Distortion Instruction Word before Merging

Field$_2$ is required at the same time as field$_3$, and field$_4$ is required at the same time as field$_5$, both due to FUs requiring multiple operands. This allows the following combinations: field$_{1-2}$, field$_{1-3}$, field$_{1-4}$, field$_{1-5}$, field$_{2-4}$, field$_{2-5}$, field$_{3-4}$, field$_{3-5}$. Due to the requirement that a merged field be large enough to support the largest operand, it is reasonable to merge fields that are closest in size and thus minimise amount of wasted space in the instruction word. The fields closest in size that can be merged are field$_1$ and field$_4$ and therefore these are combined into field$_{1-4}$ (seven bits). This now allows the following possible combinations: field$_{1-4-2}$, field$_{1-4-3}$, field$_{2-5}$, field$_{3-5}$. Field$_2$ and field$_5$ were the most similar in size and as such, these are combined into field$_{2-5}$ leaving only one possible combination: $field_{1\text{-}4\text{-}3}$. This resulting instruction word, shown in Figure 4.7, represents a reduction of nine bits (from 43 to 34 bits).

Figure 4.7: Lens Distortion Instruction Word after Merging

The results before and after the merging of the fields are shown in Table 4.3. This gives the percentage differences for the lens distortion algorithm, with instruction memory stored in both block RAM and distributed RAM, relative to the version that has not had its fields merged. The complete tabulated results can be found in Table B.3 from Appendix B.

Location of Instruction Memory	*Footprint Difference*					
	Total	*Other Logic*	*RAM*			
			Dual-port	*32x1*	*16x1*	*Block*
Block RAM	+2.8%	+2.9%	0.0%	0.0%	0.0%	0.0%
Distributed	+1.3%	+2.1%	0.0%	-14.0%	0.0%	N/A

Table 4.3: Reduction in Footprint for Instruction Word Field Merging for Lens Distortion

Again, the block RAM version showed no decrease in RAM usage because the bit widths available for the block RAM could not take advantage of this reduced width. The distributed RAM version showed a decrease in 32x1RAM usage suggesting the reuse of fields did indeed reduce the footprint of instruction memory. Both versions incur additional extra logic which offset any RAM savings, and results in an overall increase in footprint.

The next step is to investigate the combining and decomposing of FUs to achieve a smaller footprint. This is a process of balancing reuse against multiplexing.

The FUs that are examined for combining or decomposing exclude FUs that implement functions from the IFCP. This is because those FUs have already been deemed (in Chapter 3) to be suitable allocations of logic to achieve a minimal footprint (because they maximise reuse) and this process is not repeated. Two types of FUs are created that are appropriate for combining or decomposing. The first type is those that are created when a loop or branch construct in an IFCP has to be decomposed to multiple FUs. These are usually relatively primitive and the sets of FUs implemented for each construct tend to be similar. These similarities make the FUs that perform similar functions (e.g. incrementing a register) good candidates for merging. The second type is those that are created when a loop or branch construct is implemented in its entirety in a single FU. These are usually relatively complex, application-specific, and as such are good candidates for decomposition to simpler, more general, and therefore more reusable FUs.

The process of refinement involves either combining multiple FUs into a single FU, or by decomposing an existing FU into multiple FUs. If FUs are simple and many then the expensive multiplexing of registers begins to offset the savings made from FU reuse. Conversely, if FUs are complex and few, less behaviour is represented in software and more is represented in the FUs, leading to less multiplexing of registers but also less FU reuse.

All FUs from both algorithms were decomposed individually and it was found that this decomposition always resulted in increases in footprint. Similarly, the combining of FUs usually resulted in an increase in footprint except for when the two original FUs are used to increment the *same* register (i.e. implement identical logic).

The conclusion is that combining or decomposing FUs to new, more general FUs, results in footprint savings only when parameterisation (i.e. operands to control their behaviour) is not introduced. This is because such parameterisation introduces multiplexing, which, in the robot soccer and lens distortion problems, always offset the savings achieved from reuse. An example is reusing an adder for incrementing two different registers; using an operand to specify which register to increment results in a multiplexer that uses more logic blocks than the adder and as such causes the overall footprint to increase. In the two image processing problems in this research the only FUs that can be combined without introducing parameterisation are those that modify (i.e. increment or decrement) the same register.

4.4 Programming

An ESCP involves defining the binary machine code within instruction memory, setting the characteristics of the controller components (the width of the IR and PC) and passing the appropriate bit fields to the FUs. These are complex, tedious and prone to error. Thus, several techniques are investigated to reduce the design cost when programming an ESCP:

- Automating aspects of the controller based on known characteristics such as the length of the program and the number of FUs.
- Creating a symbolic assembly language rather than programming in machine code.
- Logical addressing to allow addresses to be specified symbolically rather than physically.
- Automatic addressing to avoid the process of manually assigning physical addresses.

These are investigated in the following sections.

4.4.1 Automating Aspects of the Controller

The Handel-C preprocessor, through macro expressions, is used to automate some of the processor design. Since each program instruction occupies one word in instruction memory, the width of the PC depends on the size of the program according to:

$$width\ of\ PC = ceiling(log_2(program\ size)) \tag{4.1}$$

The width of the IR and instruction memory are calculated from the sum of the widths for the execute bits and fields for operand data. The number of execute bits corresponds to the number of unique instructions or FUs. The number and width of all fields in the instruction word are defined during the creation of the ESCP. An exception to this is the field used by control instructions since its width depends on the number of instructions in the program (it must be capable of addressing the entire program since absolute addressing is used). The width of the instruction register is calculated:

$$width\ of\ IR = number\ of\ instructions + \sum_{i=1}^{number\ of\ fields} width(field_i) \tag{4.2}$$

Determining the portion of the IR to pass to an FU requires identifying the bit range each field occupies in the instruction word. This can be automated by representing the bit range for each field symbolically based on their relative positions within the instruction word. An example, given in Figure 4.8, shows $field_2$, which is six bits wide, $field_1$, which is five bits wide, and three execute bits all of which comprise an instruction word that is 14 bits wide.

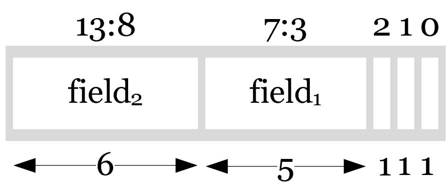

Figure 4.8: Instruction Word Structure Example

This demonstrates the implicit positioning that results in $field_2$ being situated further towards the more significant bits in the instruction word (for big-endian ordering) compared to $field_1$. Assuming such an implicit ordering allows the bit range for all fields to be determined solely from the width of every field. This is shown symbolically in Figure 4.9 in which Section A defines the positions of the three execute bits, Section B determines the limits of the 5-bit field, $field_1$, and Section C determines the limits of the 6-bit field, $field_2$, as well as defining the width of the IR.

Section A

```
enum
{
    OPCODE_1, // The bit positions for the execute bits
    OPCODE_2,
    OPCODE_3,
    NUM_INSTR // Automatically keeps track of size of exec field
};
```

Section B

```
macro expr WIDTH_FIELD1 = 5;     // Definition for field 1
macro expr LSB_FIELD1 = NUM_INSTR;
macro expr MSB_FIELD1 = LSB_FIELD1 + WIDTH_FIELD1 – 1;
macro expr IR_FIELD1 = IR[LSB_FIELD2-1:LSB_FIELD1];
```

Section C

```
macro expr WIDTH_FIELD2 = 6;     // Definitions for field 2
macro expr LSB_FIELD2 = MSB_FIELD1 + 1;
macro expr MSB_FIELD2 = LSB_FIELD2 + WIDTH_FIELD2 – 1;
macro expr IR_FIELD2 = IR[WIDTH_IR-1:LSB_FIELD2];

macro expr WIDTH_IR = MSB_FIELD2 + 1;
```

Figure 4.9: Example Expressions Representing the Bit Ranges of Two Fields

Figure 4.10 shows how this symbolic representation of field bit ranges is used in the controller to pass the appropriate operand data to FUs.

```
par
{
      if(IR[OPCODE_1]) FU_1();
      if(IR[OPCODE_2]) FU_2(IR_FIELD_1);
      if(IR[OPCODE_3]) FU_3(IR_FIELD_1, IR_FIELD_2);
}
```

Figure 4.10: Example Controller Logic for Passing Operand Data to an FU

The conditional expression governing the *if* statement simply checks the execute bit for the corresponding instruction and executes the corresponding FU.

Representing the bit ranges of instruction fields as expressions allows changes to the field widths without having to manually readjust the bit ranges of every operand passed to every FU. This eases the burden of controller development especially during the refinement phase of ESCP design.

The automation of controller specifications is a relatively minor issue but useful during implementation for two reasons. Firstly, it reduces the possibility of bugs being introduced into the controller because the regular structure reduces the errors that may be introduced by adding new opcodes and changing the fields. Secondly, it simplifies development because a change made to the program only requires changing the width expressions, which implement the automations, rather than many parts of the controller.

4.4.2 Creating the Instruction Set Architecture

Programming in binary machine code is also tedious and prone to error and therefore this section shows how this may be abstracted by creating an ISA for the ESCP. This system, like in the previous section, is implemented in the Handel-C preprocessor to avoid unnecessary hardware being created within the ESCP.

Instruction memory contains a sequence of instruction words; where each instruction word is composed of a finite number of operand fields and a finite number of execute bits. An example of two identical instruction words where each has two operand fields and three execute bits is shown in Figure 4.11.

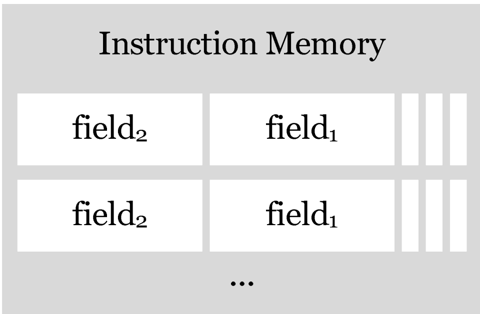

Figure 4.11: Example Structure of Instruction Memory and Instruction Words

The structure of an instruction word (i.e. the size, number and position of both the operand fields and execute bits) must be the same for all instruction words to simplify the design of the controller. Therefore, an initial step to simplify programming an ESCP is to automate this structure by defining an instruction expression, which accepts operand data and an opcode and creates a correctly structured instruction word. The instruction expression corresponding to the previous example is shown in Figure 4.12. This uses the concatenation operator, @, to join the values of the operands after extending them to the correct width. The opcode is used to set the appropriate execute bit.

```
macro expr INSTRUCTION(opcode, operandValue1, operandValue2)
      = (unsigned WIDTH_FIELD2) operandValue2
      @ (unsigned WIDTH_FIELD1) operandValue1
      @ (unsigned NUM_INSTR) (1 << opcode);
```

Figure 4.12: Instruction Expression Example

Since instructions use the same structure for every instance of reuse it is logical to create a separate expression for each instruction. This is useful because it allows instructions to be specified during programming by only their name and the operands they require. An example of the *jump* instruction is shown in Figure 4.13 where the programmer need only specify *INS_Jump(pc)* and the appropriate instruction word will be returned with the absolute *pc* value assigned to the correct field in the instruction word. If instructions do not require operand data to be specified in all fields in the instruction word, zero values are passed to the instruction expression.

```
macro expr INS_Jump( pc ) = INSTRUCTION ( OPCODE_Jump, pc, 0 );
```

Figure 4.13: Instruction Signature for Jump

The program executed by the ESPC is specified by statically initialising the instruction memory with a series of calls to instruction expressions with the correct operand values. As these are static (all based on compile time constants), the Handel-C compiler will reduce the instruction expressions to the corresponding sequence of instruction words in instruction memory.

New FUs are supported by adding the opcode to the list of definitions, and creating a new instruction expression. A change to the instruction word format can be made by changing the instruction expression. This forms the basis of an ISA, which is tailored to a particular ESCP, and helps the programmer abstract away from the intricacies of the instruction word format and focus on the logical purpose of each instruction. It simplifies the development on an ESCP by minimising the manual changes that are required when fine-tuning the underlying architecture. While this particular mechanism only allows a single instruction per instruction word, it can be readily extended to exploit instruction-level parallelism.

4.4.3 Logical Addressing

The instructions require absolute instruction memory addresses to be specified, which introduces significant potential for bugs as it is tedious to adjust all of the addresses each time a program is modified. Therefore, logical addressing was implemented whereby symbolic names could be used in place of physical addresses.

This is achieved using a series of macros to map symbolic names to their corresponding physical locations in instruction memory. The symbolic names are then used in the program and their values are defined once the programming is complete. This slightly simplifies the programming because each unique address need only be mapped once even though it may be used in multiple instructions.

An example symbol table composed of three logical addresses with a corresponding program utilising these is shown in Figure 4.14. This shows eight instructions where three involve a memory address that is expressed symbolically.

```
macro expr j1 = (unsigned 6) 3;
macro expr j2 = (unsigned 6) 6;
macro expr j5 = (unsigned 6) 1;
```

```
ram unsigned WIDTH_IR instructionMemory[...] =
{
            INS_XLoopSetup(),
/*j5*/      INS_XLoopBreakDetection(j1),
            INS_CordicDivisionLoopSetup1(),
/*j1*/      INS_CordicDivisionLoopBreakDetection(j2),
            INS_CordicDivisionLoopIteration(),
            INS_Jump(j5),
/*j2*/      INS_CordicDivisionLoopReturn(),
            INS_XFirstPassLoopIteration1(),
            ...
};
```

Figure 4.14: Assembly Code Example using Logical Addressing

4.4.4 Automated Addressing

The previous addressing system still requires memory addresses to be specified manually. However, automated addressing is beyond the capabilities of the Handel-C preprocessor because it requires looking ahead to see where a symbolic address is located in terms of its line number within the program. Therefore, an external command line tool was created that parses the ESCP program searching for symbolic addresses and replacing them with their physical equivalents. This performs string replacement (replacing symbolic source addresses, $..$, with the line number of the corresponding symbolic target addresses, #..#). An example of a program prior to and after having its symbolic addresses replaced with physical equivalents is shown in Figure 4.15.

```
     INS_XLoopSetup(),
#j5# INS_XLoopBreakDetection($j1$),
     INS_CordicDivisionLoopSetup1(),
#j1# INS_CordicDivisionLoopBreakDetection($j2$),
     INS_CordicDivisionLoopIteration(),
     INS_Jump($j5$),
#j2# INS_CordicDivisionLoopReturn(),
     INS_XFirstPassLoopIteration1(),
```

```
     INS_XLoopSetup(),
     INS_XLoopBreakDetection(3),
     INS_CordicDivisionLoopSetup1(),
     INS_CordicDivisionLoopBreakDetection(6),
     INS_CordicDivisionLoopIteration(),
     INS_Jump(1),
     INS_CordicDivisionLoopReturn(),
     INS_XFirstPassLoopIteration1(),
```

Figure 4.15: Assembly Code Example for Automated Addressing

Unfortunately, it is not possible to execute the command line tool prior to the Handel-C preprocessor in the software used in this research (Celoxica DK Design Suite v4.0); therefore, it must be run manually before compiling a project in Handel-C. This is clumsy but still useful especially for larger, more complex, programs.

4.4.5 Exploiting Instruction-Level Parallelism

This section investigates the previously mentioned ability of VLIW architectures to exploit instruction-level parallelism, which executes multiple FUs concurrently, solely with changes to software. For FUs to be able to execute concurrently they must not share any data-dependencies (i.e. not depend on the data produced by the other FU). To ensure this the following constraints must be enforced for groups of instructions executed concurrently:

- At most only one control instruction, which may modify the PC, is allowed with the remainder being non-control instructions.
- If a control instruction conditionally modifies the PC, the conditional expression must not depend on data written to by the other instruction(s).
- If a control instruction is present it must occur sequentially after the non-control instructions in the original instruction stream. This is because the result of a control instruction may change the path of execution influencing the execution of all following instructions.
- The different operand data required by each instruction must be represented in separate fields in the instruction word. This may require widening the instruction word, and thus result in changes to hardware, if the available fields are insufficient.

Once these constraints have been met by a group of instructions they can be combined into a single instruction word by setting the *execute* bits for each instruction to true.

5 Evaluation

This chapter investigates the suitability of hardware- and software- controlled application-specific processors for implementing high-level image processing algorithms. The different approaches are evaluated by measuring the footprint and performance of these processors for the robot soccer and lens distortion algorithms. Other implementation approaches, including parallel hardware and general-purpose processors, are also investigated.

The different versions of software-controlled processors (initial and tuned) have different footprint and performance characteristics. Those that exhibit the smallest footprint are used for the final evaluation shown below, with the results of all versions given in Appendix B.

The footprint of an implementation is measured using LUTs (Look Up Tables), where a logic block consists of four LUTs, since this is the basic building block of the FPGAs used in this research. The performance of an implementation is measured by the number of clock cycles required to complete the algorithm rather than the total execution time. This is because the minimum execution time will depend on the maximum attainable clock frequency, which may not be the clock frequency used by a processor (e.g. the pixel clock frequency may be used) and the implementations have not been optimised to minimise propagation delay (which will affect the maximum clock frequency).

The footprints for the robot soccer and lens distortion implementations can be found in Table B.2 and Table B.3 (from Appendix B) respectively and are compared in Figure 5.1.

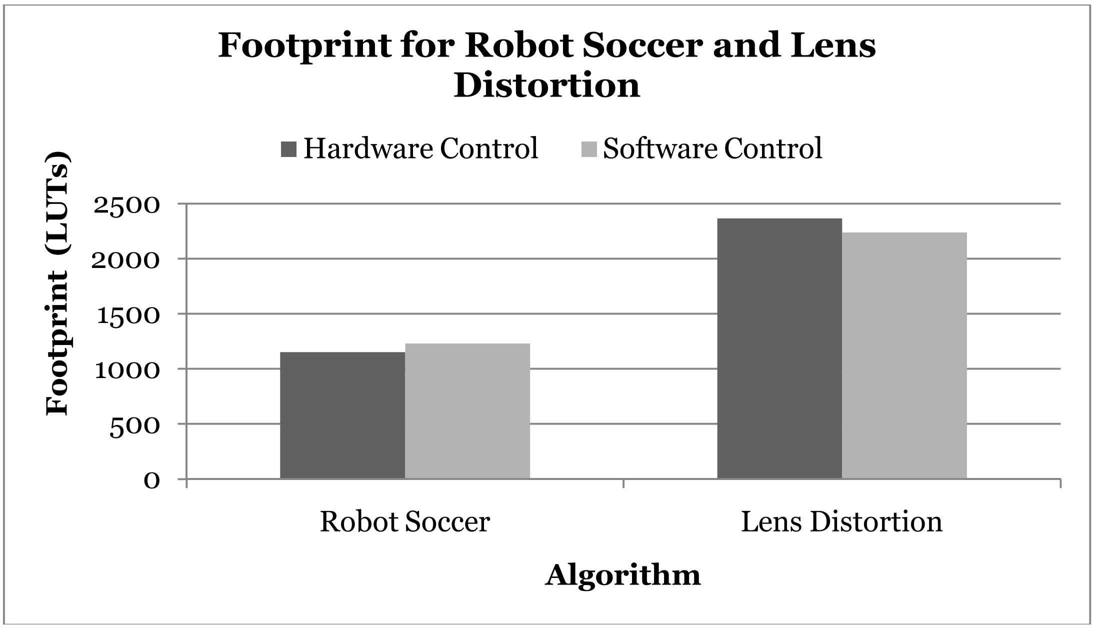

Figure 5.1: Footprint Comparison of Hardware- and Software- Controlled Processors

These results show the robot soccer algorithm incurs a 6.8% larger footprint when implemented as a software-controlled processor (from 1153 to 1231 LUTs) whereas the lens distortion algorithm occupies a 5.3% smaller footprint (from 2365 to 2239 LUTs), relative to a hardware-controlled processor.

The hardware-controlled processor that implements the robot soccer algorithm involves significant control logic and implicit reuse with its multiple nested loop and branch constructs. However, only two different calls are made to its division function resulting in relatively little explicit reuse being exploited from diverse locations in the algorithm. The increase in footprint indicates that the explicit reuse of control logic, of which robot soccer has substantial amounts, does not result in footprint savings; or at least such footprint savings are insufficient to offset the costs associated with overhead of a software-controlled processor.

The hardware-controlled processor that implements the lens distortion algorithm involves much less control logic and implicit reuse than the robot soccer algorithm. However, it makes 24 different calls to its five functions resulting in significantly more explicit reuse being exploited from diverse locations in the algorithm. Its decrease in footprint using software control can be attributed to this explicit reuse from diverse locations. This supports the assertion that avoiding multiplexing by shifting FU operands to instruction memory will result in footprint savings and that these footprint savings are sufficient to offset the overheads of a software-controlled processor.

The performance of the robot soccer and lens distortion implementations can be found in Table B.2 and Table B.3 (from Appendix B) respectively and are compared in Figure 5.2.

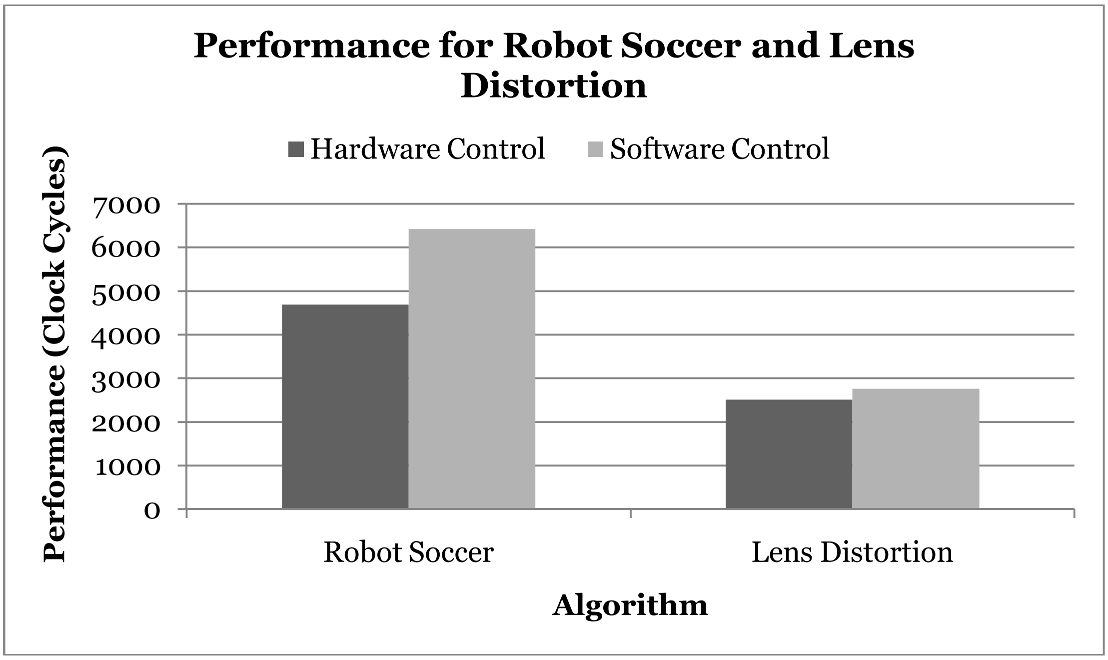

Figure 5.2: Performance Comparison of Hardware- and Software- Controlled Processors

The performance results show that for both image-processing algorithms, the software-controlled versions incur a performance penalty of 36.9% for robot soccer (from 4688 to 6419 clock cycles) and 9.9% for lens distortion (from 2513 to 2762 clock cycles).

This increase in the number of clock cycles can be attributed to three factors. The first is the inherent overheads of a software-controlled processor's instruction cycle, which requires an instruction fetch phase. This is mitigated by instruction prefetching, which concurrently performs the fetch phase during FU execution, yet this has limitations since it can only be exploited when non-control FUs, which do not modify the PC, are executed. The second factor is that the data that is normally returned directly by functions in the hardware-controlled processor must be stored temporarily and then shifted to the correct location with another instruction in the software-controlled processor, since the controller is unable to handle such data. This adds an additional clock cycle every time an FU implementing one of these functions is executed. The third is that a software-controlled processor implements the loop and branch constructs from the hardware-controlled processor as explicit FUs. In the hardware-controlled processor, many of these constructs do not consume clock cycles (but may increase the propagation delay). However, when implemented as FUs in a software-controlled processor, each of these will consume up to two clock cycles (one for the comparison and one for the jump). This explains the more significant performance penalty that robot soccer, which involves more loop and branch constructs, incurs when implemented with software control.

In summary, both software-controlled implementations incur a relatively significant performance penalty in comparison with the hardware-controlled implementations. The robot soccer algorithm's footprint increases when it is implemented on a software-controlled processor, whereas the lens distortion algorithm's footprint decreases. The relatively few algorithms investigated limit the significance that can be placed on these results yet they still provide insight into the differences of hardware- and software- controlled processors for high-level image processing algorithms.

Due to time constraints, the other possible approaches for implementing a high-level image-processing problem - parallel hardware and general-purpose processors - were not implemented. However, a discussion regarding general-purpose processors is provided and the footprint of parallel hardware for both algorithms is estimated so it can be evaluated against hardware- and software-controlled processors.

To approximate the footprint of parallel hardware a direct implementation is created. This replicates the hardware blocks that are explicitly reused in the hardware-controlled processors for each instance of reuse. This is not a completely accurate strategy because implicit reuse is still exploited and a parallel implementation will not necessarily replicate all hardware and avoid all forms of reuse. However, it provides the basis of an estimate with which to compare parallel hardware against the other implementation approaches.

The footprint for parallel hardware is estimated in Table B.4 from Appendix B. The parallel hardware implementation is expected to require an 11.7% increase in footprint (from 1153 to 1288 LUTs) for the robot soccer algorithm, and a 214.8% increase (from 2365 to 7446 LUTs) in footprint for the lens distortion algorithm, compared to the corresponding hardware-controlled processors. These estimates show, not surprisingly, that the lens distortion algorithm, which involves the most explicit reuse, incurs the most significant footprint penalty when implemented directly. This is in contrast to robot soccer, which involves relatively little explicit reuse, and shows a relatively modest footprint penalty.

In order to investigate general-purpose processors, the following soft processor cores are considered: LatticeMico8 [46], LatticeMico32 [47], PicoBlaze [48], MicroBlaze [49], Nios[50] and the Nios II [51]. Generally, these are either 8-bit or 32-bit processor architectures. Investigating these is inherently difficult because they are optimised for specific FPGAs (FPGAs can differ significantly in how they implement logic) and allow many different configurations, which influences both footprint and performance. In addition, most soft processor cores allow application-specific FUs to be “tacked on” to the processor, blurring the line between application-specific and general-purpose processors.

The reuse exploited by a soft processor core’s general purpose RISC FUs and its high level of optimisation results in relatively small footprints. The 8-bit and 32-bit cores occupy approximately 200 and 2000 LUTs respectively. The 8-bit architectures would impose severe constraints on high-level image processing algorithms, as the restricted bit widths would require multiple instructions to implement the operations that require more than 8 bits. The 32-bit architectures would mitigate these constraints somewhat yet, like their 8-bit counterparts, still use a RISC ISA, requiring a larger and more complex program. It would be prohibitively expensive to design an application-specific processor for a complex algorithm and a more complex algorithm would utilise more of the idle logic and controller overheads a general-purpose processor implements. These make a general-purpose soft processor core more suitable for complex algorithms. This is discussed in more detail in the following chapter.

6 Conclusion and Future Work

Application-specific serial processors were investigated as alternatives to parallel hardware and general-purpose serial processors for the implementation of high-level image processing. In addition, hardware and software control were compared for the application-specific processors. These experiences have provided considerable insight into the implementation of high-level image processing algorithms, the influence HDLs can have on an implementation and the different characteristics between hardware- and software-controlled processors. Guidelines have also been derived for creating future hardware- and software-controlled processors for high-level image processing tasks. Several conclusions are now summarised and the research questions posed in Chapter 1 are answered.

In terms of the question: does a software-controlled application-specific processor provide additional flexibility over a hardware-controlled application-specific processor? The answer is yes. A software-controlled processor is more flexible than a hardware-controlled processor, enabling instruction memory to be stored in distributed RAM, and block RAM as well as on I/O devices (although external storage devices were not investigated).

In terms of the question: is the performance of a software-controlled application-specific processor the same as that of a hardware-controlled application-specific processor when implementing a high-level image-processing problem? The answer is no. A software-controlled processor incurs a performance penalty compared to a hardware-controlled processor for both image-processing algorithms implemented in this research.

In terms of the question: is the footprint of a software-controlled application-specific processor smaller than that of a hardware-controlled application-specific processor? The footprint of software-controlled processors is smaller than that of hardware-controlled processors for one of the two image-processing algorithms investigated. However, it is difficult to know if these results are indicative of hardware- and software-controlled processors in general, or are because of the limitations introduced by Handel-C. Therefore, to obtain conclusive results it is necessary to implement more algorithms on both types of processors investigated in this research. In addition, other HDLs need to be investigated to isolate the effects of limitations imposed by Handel-C. These would be appropriate areas for future work.

The footprint of an application-specific processor is smaller than that of parallel hardware due to it reuse of blocks of hardware. The footprint of a RISC general-purpose processor is likely to be smaller than that of an application-specific processor. This is because the application-specific processors created in this research use the top-down approach, which results in relatively complex FUs, whereas the more RISC general-purpose processors discussed previously use relatively simple FUs. The footprint of a more CISC general-purpose processor may be larger than that of an application-specific processor because it will contain many unnecessary and complex FUs.

The performance of an application-specific processor is the same as that of parallel hardware when all operations are constrained to execute serially. However, in realistic applications, even high-level image processing will involve some operations that can be executed in parallel and therefore parallel hardware is able to exceed the performance of a purely serial application-specific processor. This can be mitigated by making a processor more parallel with pipelining or concurrent execution of FUs (discussed below). The performance of an application-specific processor is likely to better that of a general-purpose processor. This is because a general-purpose processor uses more general RISC style FUs than an application-specific processor and therefore requires more instructions to perform the same task.

The three approaches (parallel hardware, application-specific processor and general-purpose processor) can be compared in terms of their emphasis on maximising performance or minimising footprint. Parallel hardware sacrifices a minimal footprint to achieve maximum performance whereas a general-purpose processor sacrifices maximum performance to achieve a minimal footprint. Application-specific processors occupy the middle ground, striving for a balance between both footprint and performance. This correlates with their mixture of hardware and software shown previously in Chapter 1 (Figure 1.1). The more hardware-oriented approaches place more emphasis on maximising performance and less on minimising footprint. The more software-oriented approaches, place more emphasis on minimising footprint and less on maximising performance.

The design cost and initial fixed cost must be considered along with algorithm complexity to determine the approach most suitable for implementing a particular algorithm. Simple algorithms are best implemented directly in (possibly parallel) hardware because they are too simple to offset the overhead associated with a processor. Complex algorithms are best implemented on a general-purpose processor to avoid the prohibitively expensive design cost of an application-specific processor. In addition, such complex algorithms may utilise more of the logic in a *general-purpose* processor, which sits idle when executing a simple algorithm. Algorithms of medium complexity may be worth the effort to implement on application-specific processors because they are both complex enough to justify the overheads of a processor yet simple enough that their design cost is not prohibitive.

The two-image processing algorithms that were implemented on application-specific processors in this research are certainly complex enough and exhibit sufficient reuse to suggest a processor-based implementation. In addition, the performance results indicate that application-processors can easily meet timing constraints although their footprints are larger than some general-purpose processors. As long as performance is adequate to meet real time timing constraints, this is a secondary consideration to footprint.

In terms of the question: is the performance of an application-specific serial processor sufficient to meet real-time timing constraints? As an example, a real time stream of 25 frames per second allows 40 ms for each frame to be processed. The slowest application-specific processor implemented in this research requires only 183.4 µs to complete execution; this is approximately 1/200th of the time available. Although other time-consuming tasks such as delays caused by the hardware implementing the low-level operations could occur there is still likely to be a substantial amount of time available. The *complete* lens distortion algorithm may require significantly more time to execute and a useful area for future work would be to investigate its performance in regards to real time image processing.

Therefore, the ease with which application-specific processors meet real-time timing constraints and their comparatively large footprint suggests the algorithms investigated in this research would be better implemented on general-purpose processors. Assuming these algorithms are typical of high-level image processing this would lead to the conclusion that general-purpose processors are more suitable for implementing high-level image processing. This is because they are likely to be able to meet real time timing constraints, do so with a smaller footprint and have a smaller design cost than application-specific processors.

In addition to the areas of future work already identified, a suitable avenue of further investigation would be the implementation of the robot soccer and lens distortion algorithms, as well as further high-level image processing algorithms, on both parallel hardware and general-purpose processors to allow more accurate conclusions to be drawn.

A serial processor was considered suitable for implementing high-level image processing because of its ability to reuse hardware. A software-controlled application-specific processor had the additional advantage of flexibility. Such a processor may be suitable for implementing a combination of both medium-level and high-level image processing because it would be able to apply the advantages of flexibility to both. It would need to exploit some form of parallelism to meet real time timing constraints of the medium-level operations and this could be achieved via instruction-level parallelism. An investigation into the exploitation of concurrency within an application-specific processor would be useful not only for other potential high-level image processing algorithms that have stricter timing constraints and require parallel execution to meet those constraints but also for the potential execution of medium-level operations.

References

[1] C. T. Johnston, K. Gribbon, and G. D. Bailey, "Implementing Image Processing Algorithms on FPGAs," in *11th Electronics New Zealand Conference (ENZCon04)*, Palmerston North, New Zealand, 2004, pp. 118-123.

[2] K. T. Gribbon, C. T. Johnston, and D. G. Bailey, "A Real-time FPGA Implementation of a Barrel Distortion Correction Algorithm with Bilinear Interpolation," in *Image and Vision Computing New Zealand (IVCNZ)* Palmerston North, New Zealand, 2003, pp. 408-413.

[3] A. Downton and D. Crookes, "Parallel Architectures for Image Processing," *Electronics & Communication Engineering Journal,* vol. 10, pp. 139-151 1998.

[4] "IEEE Standard for VHDL Language Reference Manual", IEEE Standard 1076-2000, 2000.

[5] "IEEE Standard for Verilog Hardware Description Language", IEEE Standard 1364-2005, 2005.

[6] Impulse C, visited on 3rd June 2008, http://www.impulsec.com/

[7] P. Bellows and B. Hutchings, "Designing Run-Time Reconfigurable Systems with JHDL," *The Journal of VLSI Signal Processing,* vol. 28, pp. 29-45, 2001.

[8] JHDL, visited on 3rd June 2008, http://www.jhdl.org/

[9] P. Bellows and B. Hutchings, "JHDL-An HDL for Reconfigurable Systems," *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 175-184, 1998.

[10] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, and A. Nayak, "A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems," *8th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '00),* pp. 39-48, 2000.

[11] J. Hammes, B. Rinker, W. Bohm, W. Najjar, B. Draper, and R. Beveridge, "Cameron: High Level Language Compilation for Reconfigurable Systems," in *Proceedings of International Conference on Parallel*

Architectures and Compilation Techniques (PACT '99), Newport Beach, Calif, USA, 1999, pp. 236–244.

[12] R. Rinker, J. Hammes, W. A. Najjar, W. Bohm, and B. Draper, "Compiling Image Processing Applications to Reconfigurable Hardware," in *IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, Boston, Mass, USA, 2000, pp. 56-65.

[13] "IEEE Standard for SystemC Language Reference Manual", IEEE Standard 1666-2005, 2005.

[14] Celoxica Limited, "DK4", in *Handel-C Language Reference Manual*, 2005

[15] "ISO Standard for Programming Languages - C", ISO/IEC Standard 9899:1999, 1999.

[16] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface 3rd Edition*, 3rd ed.: Morgan Kaufmann, 2004.

[17] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*. New York: John Wiley & Sons, Inc., 1997.

[18] J. von Neumann, "First Draft of a Report on the EDVAC," *IEEE Annals of the History of Computing,* vol. 15, pp. 27-75, June 30 1993.

[19] M. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers,* vol. C-21, pp. 948-960, 1972.

[20] J. A. Fisher, "Very Long Instruction Word architectures and the ELI-512," in *10th annual international symposium on Computer architecture*, Stockholm, Sweden, 1983, pp. 140-150.

[21] M. Flynn, *Computer Architecture: Pipelined and Parallel Processor Design*: Jones and Bartlett Publishers, 1995.

[22] D. E. Thomas, J. K. Adams, and H. Schmit, "A Model and Methodology for Hardware-Software Codesign," in *IEEE Design & Test of Computers*. vol. 10, 1993, pp. 6-15.

[23] "IEEE Standard Glossary of Software Engineering Terminology", IEEE Standard 610.12-1990, 1990.

[24] R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Cosynthesis for Microcontrollers," in *IEEE Design & Test*. vol. 10, 1993, pp. 64-75.

[25] G. Kane, *MIPS RISC Architecture*: Prentice-Hall, 1989.

[26] X. Chen, D. L. Maskell, and Y. Sun, "Fast Identification of Custom Instructions for Extensible Processors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* vol. 26, pp. 359-368, February 2007.

[27] M. Arnold and H. Corporaal, "Designing Domain-Specific Processors," in *9th international symposium on Hardware/software codesign* Copenhagen, Denmark: ACM Press, 2001.

[28] C. Liem, T. May, and P. Paulin, "Instruction-Set Matching and Selection for DSP and ASIP Code Generation," in *European Design and Test Conference (EDAC)*, Paris, France, 1994, pp. 31-37.

[29] K. Atasu, L. Pozzi, and P. Ienne, "Automatic Application-Specific Instruction-Set Extensions Under Microarchitectural Constraints," *International Journal of Parallel Programming,* vol. 31, pp. 411-428, 2003.

[30] P. Ienne, L. Pozzi, and M. Vuletic, "On the Limits of Automatic Processor Specialisation by Mapping Dataflow Sections on Ad-hoc Functional Units", Technical Report 01/376, Swiss Federal Institute of Technology Lausanne (EPFL), Science Department (DI), Lausanne, Switzerland, 2001.

[31] P. Brisk, A. Kaplan, and M. Sarrafzadeh, "Area-Efficient Instruction Set Synthesis for Reconfigurable System-on-Chip Designs," in *Design Automation Conference (DAC)*, San Diego, California, USA, 2004, pp. 395-400.

[32] C. T. Johnston, K. T. Gribbon, and D. G. Bailey, "FPGA based Remote Object Tracking for Real-time Control," in *International Conference on Sensing Technology* Palmerston North, New Zealand, 2005, pp. 66-72.

[33] D. G. Bailey and C. T. Johnston, "Single Pass Connected Components Analysis," in *Image and Vision Computing New Zealand IVCNZ'07* Hamilton, New Zealand, 2007, pp. 217-222.

[34] D. G. Bailey, "A New Approach to Lens Distortion Correction," in *Image and Vision Computing New Zealand*, Auckland, New Zealand, 2002, pp. 59-64.

[35] G. P. Stein, "Lens Distortion Calibration using Point Correspondences," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1997, pp. 602–608.

[36] B. Prescott and G. F. McLean, "Line-Based Correction of Radial Lens Distortion," *Graphical Models and Image Processing,* vol. 59, pp. 39-47, 1997.

[37] M. Ahmed and A. Farag, "Non-Metric Calibration of Camera Lens Distortion," in *International Conference on Image Processing*. vol. 2, 2001, pp. 157-160.

[38] S. W. Park and K. S. Hong, "Practical Ways to Calculate Camera Lens Distortion for Real-Time Camera Calibration," *Pattern Recognition,* vol. 34, pp. 1199-1206, 2001.

[39] D. C. Lay, *Linear Algebra and Its Applications*, 3rd ed.: Addison-Wesley, 2006.

[40] "IEEE Standard for Binary Floating-Point Arithmetic", IEEE Standard 754-1985, 1985.

[41] J. E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Transactions on Electronic Computers,* vol. 8, pp. 330–334, September 1959.

[42] J. S. Walther, "The Story of Unified Cordic," *VLSI Signal Processing,* vol. 25, pp. 107-112, 2000.

[43] R. Andraka, "A Survey of CORDIC Algorithms for FPGA Based Computers," in *International Symposium on Field Programmable Gate Arrays* Monterey, California, United States: ACM Press New York, NY, USA, 1998, pp. 191-200.

[44] J. S. Walther, "A Unified Algorithm for Elementary Functions," *Spring Joint Computer Conference,* vol. 38, pp. 379-385, 1971.

[45] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed., 2007.

[46] LatticeMico8, visited on 4th June 2008, http://www.latticesemi.com/products/intellectualproperty/referencedesigns/8bitmicrocontrollermico8.cfm

[47] LatticeMico32, visited on 8th May 2007, http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/index.cfm

[48] Chapman, K., "PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IIE Devices", in *Xilinx Application Notes*, 2003

[49] Xilinx Incorporated, "MicroBlaze Processor Reference Guide", in *Embedded Development Kit*, 2007

[50] Nios Processor, visited on 1st June 2008, http://www.altera.com/products/ip/processors/nios/nio-index.html

[51] Nios II Processor, visited on 1st June 2008, http://www.altera.com/products/ip/processors/nios2/ni2-index.html

[52] Xilinx Incoporated, "Spartan-II 2.5V FPGA Family", in *Complete Data Sheet*, 2004

Glossary

CISC	Complex Instruction Set Computer
CORDIC	COordinate Rotation DIgital Computer
ESCP	Explicit Software-Controlled Processor
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit
HDL	Hardware Description Language
IFCP	Implicit FSM-Controlled Processor
I/O	Input/Output
ILP	Instruction-Level Parallelism
IR	Instruction Register
ISA	Instruction Set Architecture
LUT	Lookup Table
MIMD	Multiple Instruction stream, Multiple Data stream
MISD	Multiple Instruction stream, Single Data stream
PC	Program Counter
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction stream, Multiple Data stream
SISD	Single Instruction stream, Single Data stream
TTA	Transport Triggered Architecture
VLIW	Very Long Instruction Word

Appendices

Appendix A Data Generation

The image processing problems implemented in this research depend on data produced by the low-level operations yet the implementation of these (with parallel hardware) is not explored in this research and as such, this data must be manually created. Matlab, which is a numerical computing environment, was used to derive this data.

The data derived for the robot soccer algorithm is shown in Table A.1. Here the first four columns represent the data input by the ESCP (Explicit Software-Controlled Processor), the region type column is there for reference, and the robot information column shows the data that is calculated by the ESCP. As mentioned previously, the *sum of x pixel coordinates* and *sum of pixel* coordinates for the robot soccer algorithm are represented using U16.0. Based on the plan view of a robot, shown in Figure 3.2 previously, its fixed-point representations for the other input data is:

- U3.0 for the patch colour since seven colours (red, blue, yellow, green, purple, pink and cyan/aqua) must be represented requiring 3 bits ($2^3 = 8$).
- U7.0 for the *number of pixels* in a patch since the largest region consists of 105 pixels (21 * 5 = 105) requiring 7 bits ($2^7 = 128$).

<table>
<tr><th colspan="4">Region Data</th><th rowspan="2">Region Type</th><th rowspan="2">Robot Information</th></tr>
<tr><th>Colour Label (U3.0)</th><th>Number of Pixels (U7.0)</th><th>Sum of X Pixel Coordinates (U16.0)</th><th>Sum of Y Pixel Coordinates (U16.0)</th></tr>
<tr><td>6</td><td>105</td><td>34908</td><td>12203</td><td>A</td><td rowspan="4">Team: 4
Identity: 4
Orientation: 17°
PositionX :121
PositionY: 331</td></tr>
<tr><td>4</td><td>105</td><td>34755</td><td>12705</td><td>T</td></tr>
<tr><td>2</td><td>55</td><td>18388</td><td>6998</td><td>B</td></tr>
<tr><td>7</td><td>55</td><td>17862</td><td>6838</td><td>C</td></tr>
<tr><td>6</td><td>105</td><td>3094</td><td>37534</td><td>A</td><td rowspan="2">Team: 4
Identity: 5</td></tr>
<tr><td>4</td><td>105</td><td>3465</td><td>37905</td><td>T</td></tr>
</table>

2	55	2204	19855	B	Orientation: 315° PositionX: 361 PositionY: 33
1	55	1815	20244	C	
6	105	18432	25748	A	Team: 4 Identity: 2 Orientation: 287° PositionX: 177 PositionY: 250
4	105	18585	26250	T	
3	55	9552	14093	B	
2	55	10078	13933	C	
6	105	37055	13120	A	Team: 4 Identity: 3 Orientation: 169° PositionX: 348 PositionY: 124
4	105	36540	13020	T	
1	55	18923	6498	B	
7	55	18818	7037	C	
6	105	20378	6306	A	Team: 4 Identity: 1 Orientation: 232° PositionX: 191 PositionY: 64
4	105	20055	6720	T	
3	55	10119	3567	B	
1	55	10552	3906	C	
6	105	35915	14563	A	Team: 5 Identity: 5 Orientation: 8° PositionX: 347 PositionY:138
5	105	36435	14490	T	
2	55	19396	7824	B	
1	55	19319	7279	C	
6	105	18170	38567	A	Team: 5 Identity: 4 Orientation: 352 PositionX: 178 PositionY: 368
5	105	18690	38640	T	
2	55	10024	20551	B	
7	55	10101	20006	C	
6	105	57970	45269	A	Team: 5 Identity:3 Orientation:215° PositionX: 548 PositionY: 434
5	105	57540	45570	T	
1	55	29757	23802	B	
7	55	30072	24253	C	
6	105	37672	21006	A	Team: 5 Identity: 2 Orientation: 261 PositionX: 358 PositionY: 205
5	105	37590	21525	T	
3	55	19375	11504	B	
2	55	19919	11590	C	
6	105	53490	40841	A	Team: 5 Identity: 1 Orientation: 336° PositionX: 514
5	105	53970	41055	T	
3	55	28409	21868	B	

1	55	28633	21366	C	PositionY: 391

Table A.1: Robot Soccer Algorithm Data

The data used for the lens distortion algorithm is shown in Table A.2. Here two sets of data are used, each representing the coordinates of a different gridline. The parabola coefficients are shown for reference (and are used for debugging) with the barrel distortion component, κ, in the right most column. As described in Chapter 3, the shifting of origin and normalisation resulted in a range of -1.25 to +1.25 for x coordinates, -0.9375 to +0.9375 for y coordinates and are represented using S27.16. All values are rounded to three decimal places for readability.

X Coordinates (S27.16)	*Y Coordinates (S27.16)*	*Parabola Coefficients*			*κ (S27.16)*
		a	*b*	*c*	
-1.25 -1 -0.75 -0.5 -0.25 0 0.25 0.5 0.75 1 1.25	0.684 0.750 0.777 0.766 0.715 0.625 0.496 0.328 0.121 -0.125 -0.410	-0.313	-0.438	0.625	0.506
-1.25 -1 -0.75 -0.5 -0.25 0 0.25 0.5 0.75 1 1.25	-0.211 -0.188 -0.180 -0.190 -0.211 -0.250 -0.305 -0.375 -0.461 -0.563 -0.680	0.125	0.188	-0.250	-0.417

Table A.2: Lens Distortion Algorithm Data

Appendix B Implementation Details, Evaluation Metrics and Results

This section investigates the implementation details relevant for synthesising IFCPs (Implicit FSM-Controlled Processors) and ESCPs (Explicit Software-Controlled Processors) on FPGAs, explains the methods for calculating an implementation's footprint and performance and provides the results for the implementation of the robot soccer and lens distortion algorithms on IFCPs, ESCPs (both initially and when tuned).

A Xilinx Spartan-II FPGA [52] was used as the target platform for synthesising the implementations. This device was introduced in 1999, making it relatively obsolete, but was utilised because a suitable development board was available for this research project. The basic building block on this FPGA is a four-input LUT (Look Up Table) where each logic block contains four LUTs. The footprint measurement determined from a synthesised implementation is composed of several different elements:

- Actual logic (measured in LUTs).
- Route-through (measured in LUTs),
- Distributed RAM (measured in LUTs).
- Shift registers (measured in LUTs).
- Block RAM (measured in the number used).

The route-through, which is required to connect the other components, depends on the mapping of the logic onto the LUTs of the FPGA, and can vary from one compilation to the next. Distributed RAM is given in two values representing the 32-bit and 16-bit variations. Block RAM is external to the fabric of the FPGA and therefore not measured in LUTs.

If a footprint measurement is specified as a single value, it is calculated from the sum of all of the above values excluding the route-through and the block RAMs. This reflects the footprint used for logic and memory on the fabric of the FPGA. If block RAMs are applicable for a given implementation their number is specified separately.

Performance is specified with two measured values, the number of clock cycles required to execute the algorithm and the maximum attainable clock frequency, and one derived value (execution time). The number of clock cycles required to execute the algorithm is output by Handel-C after running a simulation whereas the maximum attainable clock frequency is determined from the maximum timing delay output by Handel-C during compilation. The execution time is calculated:

$$Execution\ Time\ (s) = \frac{Number\ of\ Clock\ Cycles\ Required}{Maximim\ Attainable\ Clock\ Frequency} \qquad \textbf{(B.1)}$$

Several different application-specific processors are implemented. The IFCP refers to the hardware-controlled processor, the initial ESCP refers to the software-controlled processor after initial FU partitioning, the tuned ESCP refers to the software-controlled processor after instruction fields have been merged to maximise their reuse and FUs combined or decomposed to minimise footprint.

The first set of data, shown in Table B.1, illustrates the footprint and performance of two experiments, a contrived experiment and an intermediate robot soccer implementation (using block RAM for instruction memory), to evaluate ILP (Instruction-Level Parallelism) in Chapter 4.

	Instruction Prefetching	*Footprint (LUTs)*	*Performance (μs)*
Contrived experiment	No	186	1.208 (64 clock cycles @ 53Mhz)
	Yes	151	0.671 (49 clock cycles @ 73Mhz)
Intermediate Robot Soccer implementation	No	1301	391.315 (14870 clock cycles @ 38Mhz)
	Yes	1294	300.375 (12015 clock cycles @ 40Mhz)

Table B.1: Instruction Prefetching Evaluation Results

Table B.2 and Table B.3 show the footprint and performance data for the robot soccer and lens distortion algorithm respectively. The rows bolded are those referred to for evaluation in Chapter 5. The first column is the type of implementation, followed by the type of RAM (distributed or block) used for instruction memory. The total footprint represents the entire footprint of the implementation, including all forms of RAM except block RAM.

	Instruction Memory Type	*Footprint (LUTs)*						*Performance (μs)*
		Total	*Other Logic*	*RAM*				
				Dual-port	*32x1*	*16x1*	*Block*	
IFCP	**N/A**	**1153**	**977**	**0**	**100**	**76**	N/A	**106.5 (4688 clock cycles @ 44Mhz)**
Initial ESCP	Block	1255	1077	0	102	76	2	183.4 (6419 clock cycles @ 35Mhz)
	Distributed	1349	1108	0	164	77	N/A	
Tuned ESCP	**Block**	**1231**	**1053**	**0**	**102**	**76**	**2**	**173.5 (6419 clock cycles @ 37Mhz)**
	Distributed	1322	1083	0	162	77	N/A	

Table B.2: Implementation Results of Processors for the Robot Soccer Algorithm

<table>
<tr><td rowspan="3"></td><td rowspan="3">Instruction Memory Type</td><td colspan="6">Footprint (LUTs)</td><td rowspan="3">Performance (µs)</td></tr>
<tr><td rowspan="2">Total</td><td rowspan="2">Other Logic</td><td colspan="4">RAM</td></tr>
<tr><td>Dual -port</td><td>32x1</td><td>16x1</td><td>Block</td></tr>
<tr><td>IFCP</td><td>N/A</td><td>2365</td><td>2313</td><td>52</td><td>0</td><td>0</td><td>N/A</td><td>104.7 (2513 clock cycles @ 24Mhz)</td></tr>
<tr><td rowspan="2">Initial ESCP</td><td>Block</td><td>2239</td><td>2187</td><td>52</td><td>0</td><td>0</td><td>3</td><td rowspan="2">110.5 (2762 clock cycles @ 25Mhz)</td></tr>
<tr><td>Distributed</td><td>2468</td><td>2292</td><td>52</td><td>114</td><td>10</td><td>N/A</td></tr>
<tr><td rowspan="2">Tuned ESCP</td><td>Block</td><td>2302</td><td>2250</td><td>52</td><td>0</td><td>0</td><td>3</td><td rowspan="2">115.1 (2762 clock cycles @ 24Mhz)</td></tr>
<tr><td>Distributed</td><td>2501</td><td>2341</td><td>52</td><td>98</td><td>10</td><td>N/A</td></tr>
</table>

Table B.3: Implementation Results of Processors for the Lens Distortion Algorithm

Table B.4 shows the footprint results of the direct implementations of robot soccer and lens distortion. Performance is not shown because it cannot easily be estimated.

<table>
<tr><td rowspan="3">Implementation</td><td colspan="6">Footprint (LUTs)</td></tr>
<tr><td rowspan="2">Total</td><td rowspan="2">Other Logic</td><td colspan="4">RAM</td></tr>
<tr><td>Dual-port</td><td>32x1</td><td>16x1</td><td>Block</td></tr>
<tr><td>Robot Soccer</td><td>1288</td><td>1112</td><td>0</td><td>100</td><td>76</td><td>0</td></tr>
<tr><td>Lens Distortion</td><td>7466</td><td>7414</td><td>52</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Table B.4: Footprint Results for the Robot Soccer and Lens Distortion Direct Hardware Implementations