DOCTOR OF PHILOSOPHY

FPGA techniques for algorithm acceleration

Lewis, Emlyn

Award date:
2007

Awarding institution:
University of Wales, Bangor

FPGA Techniques for Algorithm Acceleration

Emlyn Lewis

Thesis submitted in candidature for the degree of Doctor of Philosophy

October 2007

School of Informatics
University of Wales, Bangor

Summary

The techniques necessary for the hardware implementation of systems which would traditionally be implemented in software are investigated, with regard to two systems: an image processor and an electronic neuron model. The latter is developed in detail and it is shown that a simplified and space-efficient model can perform the functions of more complex models. Interesting results are shown and novel methods of building with these models are demonstrated.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Dr Iestyn Pierce, my supervisor, for his help and support during this project, and my family and friends.

Chapter 1: Introduction

This work is an investigation into the processes involved in implementing systems in custom hardware which have previously been implemented in software. In particular, the aim was to investigate Field-Programmable Gate Array (FPGA) implementations, looking at the particular challenges inherent in the use of these devices. Two major projects were undertaken, the first was an implementation of an image filtering system, and the second was an implementation of digital neuron models. The image processing system was a reimplementation of a software-based system, and its development provides an insight into the process of converting a software algorithm to hardware, resulting in a hardware-based system which performs the same sequence of operations. The neuron models are different in that they were not based on software implementations but instead designed for hardware from the start. In both cases, various issues and difficulties were encountered and overcome, and these are discussed, along with extensive test result analysis.

1.1: Structure of the Thesis

Chapter 2 provides an introduction to FPGAs and their current uses, both in general-purpose logic implementation and prototyping, and in more specialised processing systems. The internal structure of the FPGA is discussed, looking specifically at Altera's Apex series FPGAs, which were used for the subsequent work. The use of hardware description languages and some of the issues associated with hardware compilation are also discussed. This chapter summarises many of the issues encountered during the development of the hardware described in chapters 3, 4 and 5.

Chapter 3 describes the first body of experimental work, an investigation into implementing a simple image processing function in an FPGA, looking at the issues involved in the implementation and the possible performance of the system. Current implementations in this field are discussed at the start of the chapter, with the new work being introduced from section 3.2 onwards.

Two filtering systems are developed and optimised for efficient operation in the FPGA. A range of test results is presented to show the operation of the filters and

the differences between the two types. The specific issues associated with making efficient and effective use of the hardware resources available in the FPGA are revealed and discussed, as are the issues associated with implementing fast and accurate arithmetic circuitry in the hardware. The trade-off between hardware size and processing speed is demonstrated and discussed.
Following from the development of the image processor, a cellular automaton processor using very similar hardware is also presented, demonstrating the flexibility of the design.

Chapter 4 describes the development and implementation of a simple microprocessor which was intended to provide good performance with low hardware cost. The use of the processor is further explored in chapter 5. The design is shown to be versatile and capable, even though it lacks many of the features of conventional microprocessors, and to have a high performance for its size. The optimisation of the design to make the hardware cost as low as possible is also discussed.

Chapter 5 describes the major body of work, looking at implementations of digital spiking neuron models in the FPGA. The existing work in this field is discussed in section 5.4, with the project work beginning in section 5.5. Various neuron models are introduced, along with simple networks in which the neurons are tested. The response of the neurons to input spikes is demonstrated and a range of test results are presented covering the two major models constructed. Some simple neural circuits are presented, showing interesting functions being performed by the neuron models in novel ways, and a network of these neurons is demonstrated showing complex dynamics due to feedback.
This chapter also includes some discussion of the trade-offs between the size and complexity of the models and their abilities, and analysis of the importance of choosing the register sizes and parameter ranges carefully to avoid erroneous operation.
The conclusions from the previous chapters are finally summarised in chapter 6.

1.2: Contributions

The major contributions in this thesis are summarised below.

Image processing systems have been developed and it has been shown that there is a definite trade-off between performance and hardware complexity. The image processors presented show a large difference in both performance and complexity, with the faster version showing a good performance / area ratio.

The issues associated with the efficient implementation of hardware image processing systems have been discussed and methods for making the most efficient use of the FPGA hardware have been considered.

The extension of the image processing hardware to cater for different algorithms and processing of cellular automata has been shown, and a versatile cellular automaton processor has been demonstrated.

A simple microprocessor for FPGA implementation has been developed, and has been shown to be versatile and capable despite lacking much of the complexity of conventional microprocessors. Its simple hardware is shown to yield a relatively high performance – area ratio.

Hardware-based digital spiking neuron models have been developed and these have been shown to be capable of performing complex functions with relatively simple internal hardware.

Networks of simplified neuron models have been demonstrated and have been shown to be capable of complex and possibly chaotic dynamics, replicating the oscillatory behaviour seen in small networks of neurons in biological tissue.

Novel neural circuits have been developed which can perform functions not normally associated with neurons. These building blocks have shown promising results and the potential for future development.

Chapter 2: FPGA Technology

This chapter is intended to give a brief overview of the FPGA, its hardware and its uses in accelerating signal processing. Detailed reviews of the use of FPGAs in the three fields of image processing, microprocessor implementation and neuron modelling can be found in sections 3.1, 4.1 and 5.5 respectively.

2.1: Reconfigurable Logic – The FPGA

The Field Programmable Gate Array (FPGA) allows large-scale digital circuitry to be implemented without the cost of producing a fully-custom VLSI device. The configurability of the FPGA architecture means that prototype circuits can be built, tested and debugged rapidly, without requiring experience in VLSI layout on the part of the designer. The reconfigurability of the device can provide a useful benefit for consumer appliances such as set-top boxes, where the hardware digital signal processor which performs the decoding of the signals can be upgraded in much the same way as firmware is upgraded, without having to physically replace the part. In fact, the circuit can be considered to actually be firmware, as the FPGA will usually read its configuration data from external storage into internal SRAM at power-up, therefore if the data held in this external storage is changed, the next time the appliance is powered up, its hardware will be updated.

The FPGA is typically composed of a number of logic cells connected together by a switching and routing matrix. In a fine-grained FPGA, each logic cell will typically have a small number of inputs feeding a look-up table based logic block, which in turn feeds a register, usually implemented with a D-type flip-flop. A coarse-grained device, such as a CPLD (Complex Programmable Logic Device) will tend to consist of a smaller number of much larger cells, called macrocells, each of which may have 30 – 40 inputs feeding an AND-OR array. In either case, a series of SRAM cells hold the configuration data which in turn sets the state of switches – MOS transistors in most cases, which configure the internal operation of the block. A four input logic cell's logic block can perform one of $2^4 = 16$

different operations, so the block can be implemented as a 16 x 1bit SRAM where the inputs to the block form the address.

The structure of the logic element of an Altera Apex 20K series FPGA [2.1] is shown in Figure 1. The look-up table has additional carry and cascade logic attached, which allows wider functions to be implemented with a smaller speed penalty than would be the case if the function was implemented with a number of LEs in series. The registered part of the LE consists of a flip-flop and logic to allow synchronous or asynchronous operation.

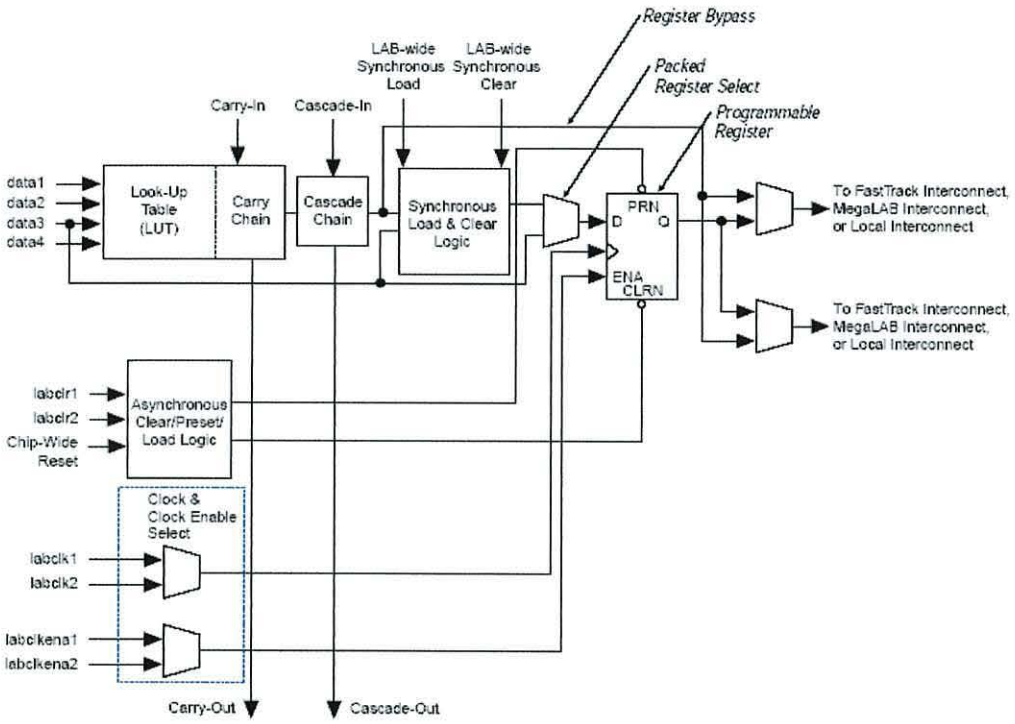

Figure 1: Logic Element layout of the Apex 20K FPGA (From Altera's Apex datasheet)

It can be seen from the figure that since the LE has two outputs, it is possible to use the register and the LUT separately, with the input 'data3' feeding the register, which feeds one output, and the other three inputs feeding the LUT, which can then feed the other output. Thus, it is possible to make more efficient use of the logic space in the device than it would be if the LUT's output had to go through the register, and it is also possible to obtain higher speeds than if the register's input had to come through the LUT.

The LEs in the Apex series are grouped into larger blocks called Logic Array Blocks (LABs) which each contain 10 LEs. Within the LAB connections between LEs are very fast, allowing small logic structures to operate efficiently. Any processing or register structure which require more than 10 bits will need to use more than one LAB, which can reduce speed slightly, but the LABs are joined together to form MegaLABs, which vary slightly across the range of devices within the Apex 20K series. The 300,000 gate 20K300E device used for the experiments described in the following chapters has 16 LABs to each MegaLAB, with 72 MegaLABs in total. Some of the larger devices in the series have 24 LABs per MegaLAB, while the smallest one, the 20K30, has 10.

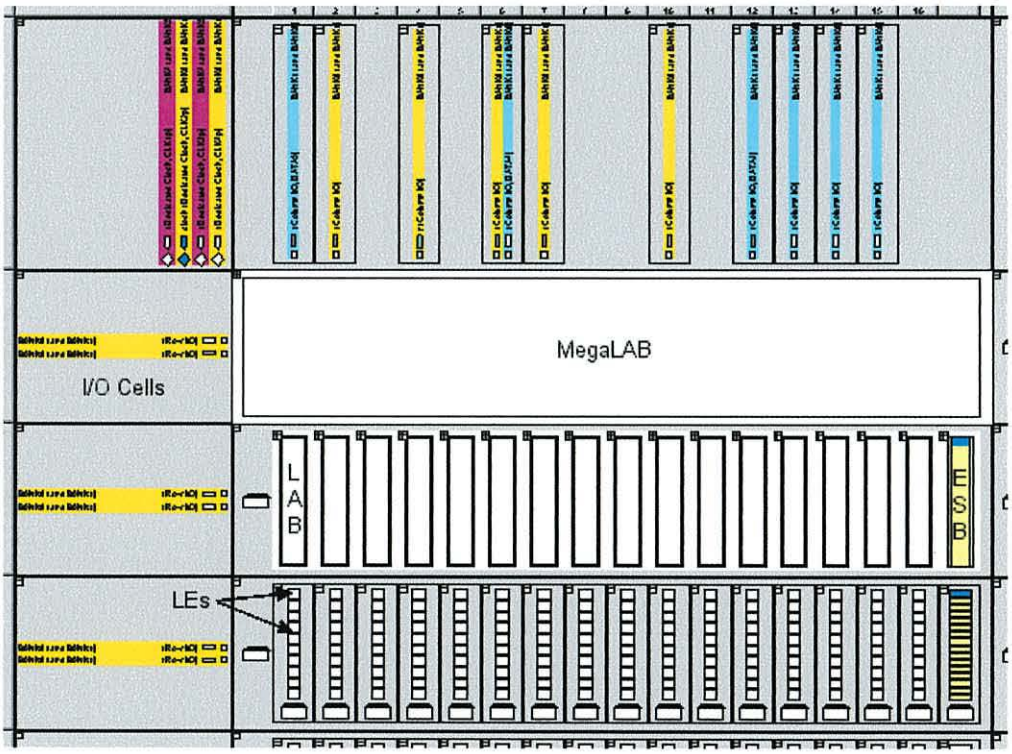

Figure 2: Section of Quartus floorplan view showing FPGA structures

These structures are shown graphically in Figure 2, which shows a screenshot of a section of the Quartus software's floorplanner window, with the various structures either open or closed. The coloured blocks are those which have been used by the compiled design, the white blocks are unused. For completeness, Figure 3 shows the full chip view, with the section in Figure 2 visible in the top left hand corner. Although the scale of the image makes it difficult to see the coloured LEs, the

long carry chains used in counter and adders can be seen in the lower right hand section of the chip, shown as dark lines. When joining two or more LABs in a carry chain, the fitter usually skips alternate LABs, so a chain starting in LAB 1 will continue in LAB 3 rather than LAB 2.

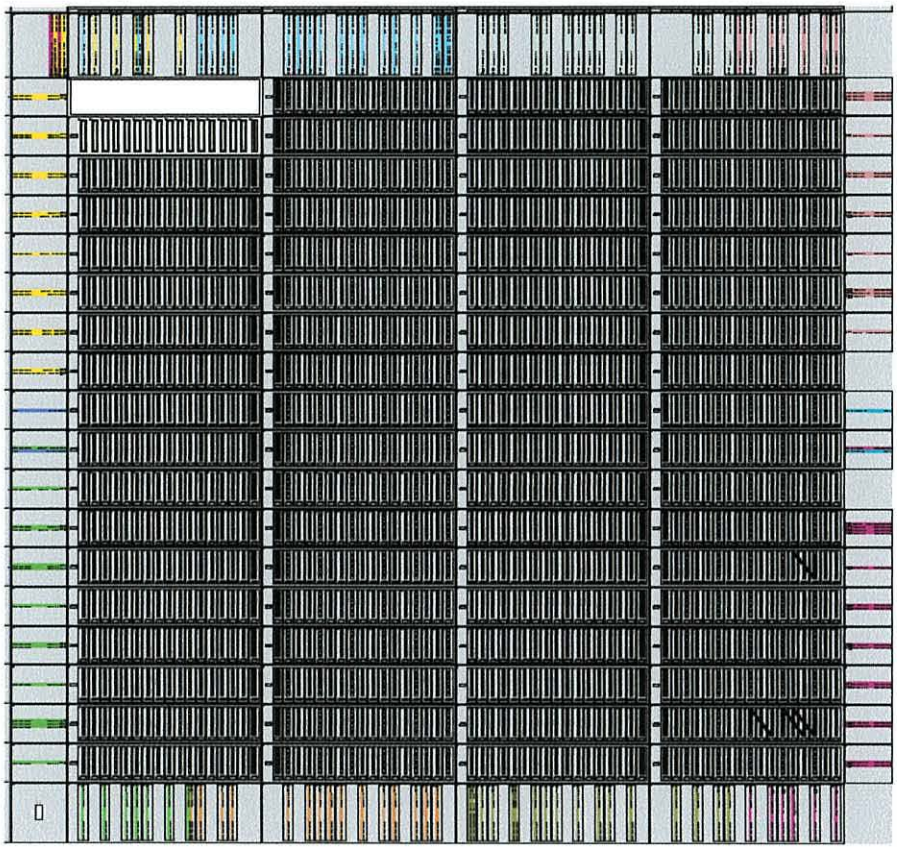

Figure 3: Quartus floorplan view of entire FPGA

Each MegaLAB also contains an Embedded System Block (ESB), which in the Apex 20K devices is a 2048 bit RAM block that can be configured as RAM or ROM. The use of these blocks is further discussed in the microprocessor design of chapter 4 and the neuron designs in chapter 5.

For comparison purposes, the Virtex series from Xilinx [2.2], being approximately equivalent to the Apex series in terms of age and capabilities, uses a slightly different arrangement. The basic block is still the Logic Cell, consisting of a LUT and flip-flop, but these are grouped in pairs to form 'slices', with each slice containing extra circuitry which can allow the LUTs to be grouped together efficiently to implement functions of up to 9 inputs. The slices are then grouped in

pairs to form Configurable Logic Blocks (CLBs), which are arranged in a regular lattice. Although the basic Logic Cell is similar to the LE of the Apex device, the slight differences mean that in many cases a similar circuit implemented in the two FPGA families will have quite different logic cell requirements. Some of the more advanced Altera devices have multiple LUTs and registers in each LE, and this will be noted where necessary, but the convention adopted in this thesis for comparisons between the two FPGA types is that a CLB and slice in the Xilinx devices are equivalent to 4 LEs and 2 LEs respectively in the Altera Apex device.

2.2: Hardware Compilation

The mapping of a design to the logic-cell structure of the FPGA is generally performed by software, usually software provided by the device manufacturer. For Altera's FPGAs the software is Altera's own Quartus II software [2.3], which contains a sophisticated logic synthesiser and fitter, allowing the design to be entered as a schematic or in some form of hardware description language (HDL). The use of a HDL allows a much more complex system to be generated easily, as a single line of code could potentially represent a great deal of hardware, especially where mathematical functions are required. The language used for the systems in this work was VHDL [2.4], an industry standard which allows the hardware to be defined in a manner similar to software, though with some important differences. Firstly, the hardware description does not define an algorithm, but rather a series of independent pieces of circuitry, all of which will operate in parallel. If an algorithm is to be implemented in hardware, it must be broken down into a sequence of operations, so that the operations can be implemented as blocks of hardware, and the sequence implemented by a state machine or similar. The convolution-based image processor presented in chapter 3 was originally based on a simple software implementation, and the state machine used to control it shows the flow of events that would take place if software was used.

In some cases the logic synthesiser can be put to use to avoid having to design a complex piece of hardware, as demonstrated in chapter 3 where the image processor developed there required a divide-by-81 circuit. The fastest arithmetic circuit is one that is built from purely combinatorial logic, rather than using any form of repetitive bit-by-bit algorithm, and this can be made even faster if built as the hardware form of a look-up-table. The divider was therefore defined by a long block of VHDL that explicitly stated the desired output for all possible inputs. With a 15-bit input, this necessitated over 32,000 lines of code, which demonstrates another useful feature of a hardware description language such as VHDL, in that it can be written by machine, in this case being generated by a simple program written in QBasic, which calculated the desired output for each possible input, and created the VHDL code accordingly. The logic analyser and synthesiser were then able to simplify this code to produce something relatively

small. This is a useful aspect of the hardware compilation which can save a lot of time.

In addition to producing VHDL descriptions of simple blocks such as a divider or look-up-table, much work has been done in allowing implementations of complete DSP functions in software packages such as MATLAB to be converted automatically to VHDL [2.5][2.6], which can then be processed with the FPGA vendor's own tools to produce an FPGA implementation. It is also possible to compile from programming languages directly to hardware. One such language is Handel-C [2.7], which is much like C apart from its output. In either of these cases the compiler will automatically generate the necessary hardware to perform the coded algorithms, and the language contains some extensions to make use of the parallelism possible with a hardware implementation.

While this approach may be useful in converting existing implementations, the work presented in this thesis was coded either in VHDL or in a mixture of VHDL for the functional blocks and schematic entry for their interconnections.

2.3: FPGA Performance

With the signals passing through a series of routing switches between logic cells, there is always a speed penalty associated with using programmable logic rather than full-custom devices. As is shown in the later chapters, if a design contains a lot of logic spread out over many LABs this can limit the possible clock rates to a few tens of megahertz, whereas a well laid-out design implemented in full-custom VLSI could achieve many times this speed, as is apparent in the modern generation of PC processors.

It is this reduction in performance that makes parallelism so important. In addition to this, a hardware algorithm can often outperform a software algorithm without using parallelism, as it doesn't have to fetch and decode instructions, and intermediate results can be stored in registers rather than being written back to memory for later retrieval. Local parallelism can also be used, for instance if complex numbers are used, the real and imaginary parts must be handled separately by a software algorithm, while hardware can handle them in parallel. Complex multiplication requires four multiplications, an addition and a subtraction, and performing these in parallel could provide a significant performance increase. [2.8]

2.4: Digital Signal Processing with FPGAs

Since it is possible in many cases to gain a performance increase over traditional software by using hardware, the usefulness of FPGAs in this field has been known for some time. [2.9][2.10] In particular, the reconfigurable aspects of FPGAs [2.11] and their ability to mix hardware and software [2.12] have been shown to be useful for DSP acceleration. A method for mixing hardware and software in the FPGA with a novel low-footprint embedded processor is discussed in chapter 4. In DSP applications, the FPGA can be useful either in implementing the entire DSP system on a single chip, or as a co-processor in conventional PC-based systems. [2.13] In systems such as these, the functions which can be adapted to make use of parallelism are implemented on the FPGA, with the rest of the processing performed by the software. The high-performance GPUs found on modern PC video cards are a good example of how custom hardware can provide a significant performance boost compared with a software implementation.

Many of the more complex modern FPGA families contain additional DSP blocks alongside the logic elements and embedded memory. The DSP blocks are designed to support the implementation of digital filters, FFTs and DCTs, and other similar functions. These functions tend to require multiplication, and so the DSP blocks contain configurable multipliers which are usually much faster than the logic elements. Taking Altera's Stratix 2 series [2.14] as an example, there are up to 96 DSP blocks available (in the largest device), and each block can be configured to support either a single 36 bit x 36 bit multiplier, four 18 x 18 multipliers or eight 9 x 9 bit multipliers. Theoretically then, the largest device in the series could potentially provide the capacity to perform 768 9-bit multiplications in parallel, before any of the logic elements are used. It has been shown that high performance can be achieved with relatively low clock rates if parallelism is exploited [2.8] and it can be assumed that the DSP blocks will be able to perform much faster than the equivalent blocks built with logic elements as their internal structure does not suffer from the reduction in speed due to the configurable routing circuitry.

2.5: The Experimental Hardware

The hardware implementations of the systems discussed in this thesis were done with a Digilab 20Kx240 FPGA development kit, from El Camino [2.15]. The Digilab was fitted with an Altera Apex 20KE series FPGA, of type EP20K300EQC240-1X, providing the equivalent of approximately 300,000 logic gates.

The Digilab board pictured in Figure 4 provides the means for powering and configuring the FPGA, and also provides a range of additional devices, many of which were used in the prototype systems. There are two banks of high-speed asynchronous static RAM, arranged as 512K x 16-bit, along with a serial EEPROM for non-volatile storage. Four push buttons provide simple on-board command inputs, while four bi-colour LEDs and a four digit 7-segment display provide output capability. In addition to this, facilities are provided to attach external hardware to the FPGA, allowing additional I/O devices to be used. The use of such devices is shown in chapter 3, with a communications system, and chapter 5, with a variety of input and output systems.

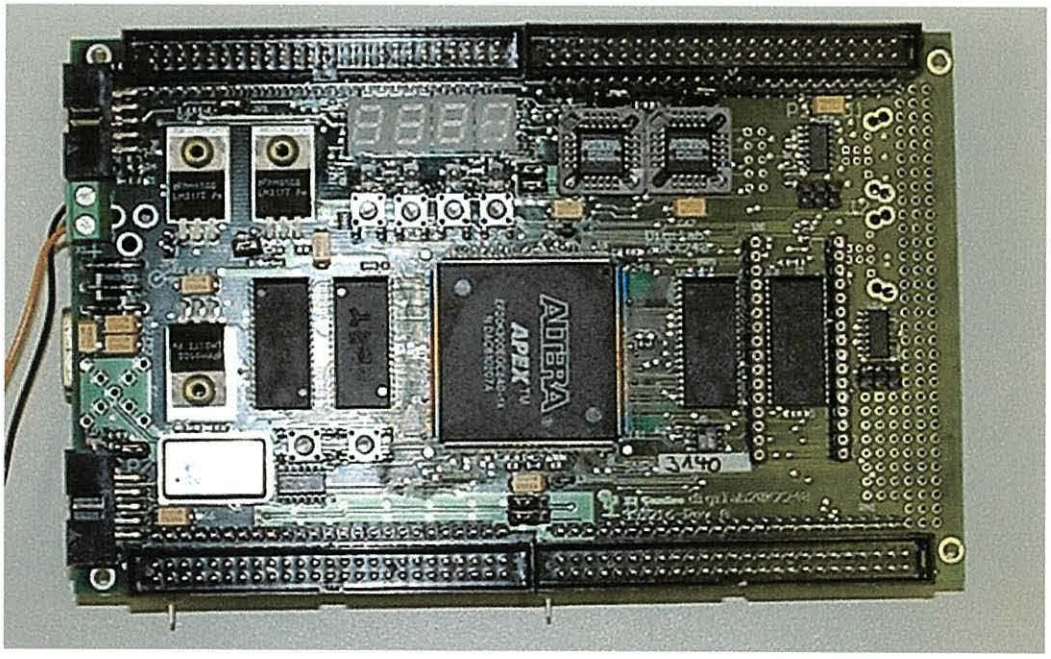

Figure 4: Photograph of Digilab 20Kx240 FPGA development kit

There is a clock oscillator on the board, providing a 48MHz clock, and the –1X speed grade FPGAs have PLL-based frequency synthesisers on board which can be used to multiply or divide the frequency to provide alternative clocks. These PLLs were used to generate a 25MHz clock for the VGA timing logic used in some of the experimental work. It was found that a video display could be useful when a large quantity of low bandwidth status signals needed to be displayed, and because the reconfigurable logic made it a simple process to tailor a display controller to fit the exact needs of the system, VGA video displays were used on some occasions to provide a user interface.

The Altera Apex 20K series were state-of-the-art when the work was started, though technology moved on rather quickly, and just a couple of years later it was possible to fit at least ten times the amount of logic into an FPGA as the 300k-gate 20K300 device allows. However, unless otherwise specified, throughout this document it is assumed that any references to the architecture of the FPGA or its logic cell functions refer to the Apex 20K series. The results obtained and the architectures presented are still valid, as the systems presented in this thesis are all implemented in VHDL and can be migrated to newer devices with little modification.

Chapter 3: Image Processing with an FPGA

This chapter describes a series of image processing systems implemented in an FPGA, for the purpose of performing high-pass filtering on an image in order to extract details for further processing. Two main processing algorithms are presented, along with a cellular automaton processor based on a reworking of the same hardware.

3.1 Theory and Review

It has been shown [3.1][3.2] that an analysis of the surface texture of a skin lesion can be useful in determining whether the lesion is benign or malignant. The skin has a natural pattern of lines, which generally tend to flow in one overall direction. As new skin develops and replaces the old, it will usually conform to this same pattern, preserving the skin line structure. However, an uncontrolled growth of cells, such as that displayed by cancerous cells, will not follow the same pattern and will lead to a disruption of the pattern on the skin above. Therefore, as shown in [3.1], it is possible to gauge the likelihood of a skin lesion being cancerous by examining the skin lines. Figure 5 shows a comparison of a malignant lesion and a benign one, and their skin line patterns.

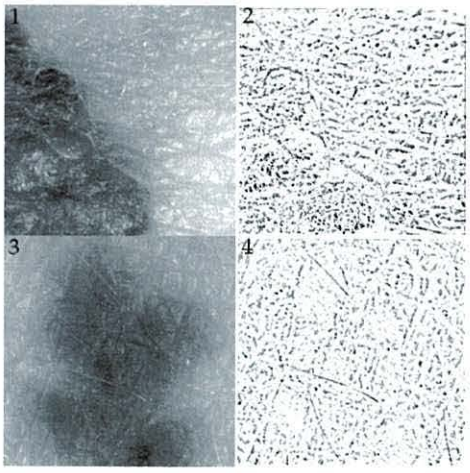

Figure 5: Comparison of lesions and their skin line patterns

Sections 1 and 2 show the malignant lesion, and it can be seen that the pattern in the lower left hand corner of section 2, corresponding to the inside of the lesion, is

much more irregular than in the upper right hand corner. By contrast, sections 3 and 4 show a benign lesion, and it can be seen that the skin lines do not noticeably change direction between the inside and outside of the lesion.
This skin line analysis can be done by computer, but it is necessary to first extract the pattern of skin lines from an image of the lesion before they can be analysed. This can be done by a number of methods, which all essentially involve a high-pass filtering operation. This operation removes the more gradual changes in intensity, leaving just the fine details, such as the skin lines. With good enough contrast in the source image, an edge detection algorithm could also be used.

A widely used method of filtering in image processing is convolution, where each pixel in the output image is computed from the pixels within a mask centred on the corresponding pixel in the source image. Each cell within the mask has a weight by which the pixel under that cell is multiplied, and then the weighted pixels are summed, usually divided by the sum of the mask weights if this is non-zero, and the new pixel is stored in the result image. This is a simple and configurable process, as the mask weights can be changed to implement a range of different functions. The function of interest, based on the work in [3.1, 3.2], is a high-pass filter operation implemented by isolating the low-frequency components via a low-pass averaging filter and subtracting these from the original image. The averaging filter is a simple filter to implement with convolution, as all mask weights are 1, and therefore no multiplication is required. In the case of this particular type of filter, the size of the mask determines the cut-off frequency of the filter, with a larger mask allowing coarser details through.
With convolution being a widely used image processing function, many VLSI implementations of the process have been made, in both custom VLSI and reconfigurable logic.

Vega-Rodriguez et al. present an optimised architecture for image convolution [3.3], focusing on convolving an image with a fixed 3x3 mask. The implementation presented accelerates the process by performing the processing for four pixels in parallel, with pipelining. On each clock cycle four rows of the image data under the mask are read, allowing three pixels to be partially computed

and the fourth to be fully computed. Each cycle also finishes the three partial computations from the previous cycle.

The filter mask is implemented as 9 parallel multipliers and an adder tree, with optimisations applied to the latter such that each adder is built with only as many bits as are required, thus reducing hardware usage. The multiplications are decomposed into a series of summed multiplications by powers of two, implemented with shifters and an adder tree. In the case of a low-pass filter, where the centre column of the mask has coefficients which are twice that of the other columns, the same hardware is used for all columns, with the result of the computation of the centre partial result being multiplied by two, again by shifting, thus reducing the hardware usage still further. It is shown that with a clock of 16MHz the system is capable of processing 30 images in less than 900ms, and is therefore capable of real-time processing of video information.

The approach of simplifying multiplications by restricting them to powers of two is also adopted by Hsiao et al. in the implementation of an edge detection system [3.4] which incorporates a noise removal filter with a Gaussian mask. The mask is approximated with powers of two, such that each mask coefficient is either a power of two or the sum of no more than two such numbers.

Torres-Huitzil et al. present another architecture for convolution in which processing is accelerated by re-using partially computed pixels [3.5]. This design recognises that each pixel in the source image will be used in the computation of several output pixels, and so for each pixel read a series of processors compute that pixel's contribution to each of the output pixels dependant on it. A mask size of 7 x 7 pixels is used, and so 49 processors compute partial results for the same pixel, each processor using the mask co-efficient for a different position within the mask. Thus, each pixel need only be read once. The partial results are collated and used to generate the output pixels once enough input pixels have been read. The system is stated to take 8.35ms to process a 512x512 pixel greyscale image with a 7x7 mask.

Bosi et al. present an architecture for a convolution co-processor intended to work alongside a conventional DSP, which handles the high-level transfer of image data. The intent of this implementation is to increase the speed of processing, and

it is stated that a TMS320C40 DSP requires around 20 instruction cycles per pixel with a 3x3 mask. This co-processor [3.6] is based on shift-registers, which hold the previously read pixels, so that each pixel need only be read once. The processor holds two complete image lines plus 3 pixels of a third line, starting at the pixel under the top-left corner of the mask and extending across and down the image to the opposite corner of the mask. For the example presented, in which the processor is handling subsections of the image 68 pixels wide, a total of 139 pixels are held in the shift register. With the register full, an output pixel is produced on every clock cycle. The principle of storing pixels so that they can be used in subsequent mask operations without being re-read from memory is a simple and widely used way of speeding up the process [3.7][3.8][3.9]. The reduction in logic usage brought about by implementing some of the shift register stages with the FPGA's RAM rather than logic elements is also shown in the work by Hsiao et al. [3.4] This usage of the internal RAM as a shift-register is possible on some of the more modern FPGAs.

It is also shown in [3.6] that convolution with a larger mask can be broken down into a series of convolution operations using 3x3 masks. A 3x3 'elementary' convolution engine architecture is shown, along with the method by which four of these can be used to perform a convolution with a 5x5 mask. This elementary convolver is one which can take in partial sums from other convolvers, thus allowing several to be used together. This method has the advantage of reducing the complexity of the design, as all that is required is a simple 3x3 elementary convolver repeated several times, but it does increase hardware usage, as many operations are carried out twice where the convolution windows overlap. A third alternative shown is to break the convolution down into a series of 1-D convolutions, which has the benefit of being easier to scale as the image or mask size is changed, though with the drawback of increased hardware usage over a single 2-D convolver as partial sums must be passed between stages. For a fixed mask size a single 2-D convolver is the optimum choice, but the more complex methods involving breaking the convolution up into smaller processes allows for better scalability when the mask is of variable size. Consideration is also given to multiplexing the processing hardware between two shift-registers, allowing two image lines to be processed simultaneously with the same hardware, though reducing the throughput from one result pixel per clock cycles to one per two

cycles. However, this is a reduction in the quantity of multipliers only, and a large number of shift register stages are required for large images. A total of 962 CLBs in a pair of Xilinx XC4013s are required for the implementation, the majority being used to implement the shift registers. This is equivalent to 1924 LEs in an Apex device.

Addressing the issue of shift-register requirement, Cardells-Tormo et al. present a series of alternative shift-register-based architectures [3.10], intended for processing the large quantities of image data required for high-resolution printing. The first of the architectures presented holds as many pixels within the shift-register as there are pixels in the mask, and thus allows the output pixel to be computed using only the data from the shift-register, while not requiring any more shift register stages than the minimum necessary. More complex architectures are also presented, which use several shift registers operating in parallel, or move a larger block of image data through the register with each cycle, to further enhance the speed of the processing. It is shown that there is a trade-off between performance and hardware cost; although the methods presented use fewer shift register stages than the earlier work [3.6] the performance decreases and it is no longer possible to produce a new result on each clock cycle, as to produce a row of output data, 2s+1 complete rows must be read from the source image, where s is the mask size.

In order to further reduce the amount of hardware required, Zhang et al. noted that many of the most frequently used convolution kernels have some degree of symmetry, and indeed many are quadrant-symmetric, so a reduction in hardware can be achieved by implementing only a quarter of the kernel, and using this to process the pixels from all four quadrants, with suitable co-ordinate transformation [3.11]. A reduction of 75% in the number of multipliers and nearly 50% reduction in the number of adders is claimed, when compared with a hardware system in which all kernel cells are processed by separate hardware.

Taking the other route of performance over hardware cost, Perri et al. have shown that if minimising the hardware is not an issue, greatly improved performance and flexibility can result from employing an array of 3 x 3 convolvers, each built with

its multipliers implemented in parallel [3.12]. This implementation allows processing at different word lengths depending on configuration signals, and can use arbitrary mask coefficients, in contrast with the architecture in [3.6] where the mask coefficients were restricted to certain values to save hardware. This design takes 4.6ms to process an image of 1024 x 1024 pixels with a 5 x 5 mask, operating at around 28MHz.

The benefits arising from parallel processing are demonstrated in work by Rosas et al. where a convolution engine designed to implement edge detection with a Sobel mask operation is implemented with 30 parallel processing elements [3.13]. The overall processor works on a 32 pixel wide column of the source image, repeating as necessary across the image, and stores the read pixels in memory internal to each of the processing elements, which produce their outputs in parallel. The resultant processor can process a 640x480 image in 23ms, fast enough for real-time processing. This is compared with 3.6 seconds for the same processing carried out on a SPARC-20 CPU.

In a similar fashion, Saldana et al. present an image processor using 49 parallel processing elements [3.14], which also makes use of local caching of pixels to reduce the number of source pixel accesses, as has been seen to be the case in many works. In this case around 200 images of 640x480 pixels can be processed each second at 66MHz, an increase in speed of 8 x compared with a software implementation on a 1.5GHz Pentium 4. This is indicative of the power of parallel processing, as most if not all FPGA implementations operate at significantly lower clock speeds than their software counterparts, and still achieve a performance increase.

While many of the above implementations are of general-purpose convolution systems, much work has been done specifically on edge detection [3.15][3.4][3.16][3.17], which, though less flexible than full convolution, is perhaps suited to the task at hand, i.e. the extraction of the skin lines from the image. Indeed, edge detection is widely used enough that Altera produce an IP block for this purpose [3.18], which can be implemented through a plug-in to the Quartus software. This is a convolution-based system operating with the Sobel

masks, and in the reference design is connected to the Nios processor core (see section 4.1), which handles the transfer of data between the core and memory. The vast majority of the edge detection systems use the Sobel masks, as these require just a 3x3 convolution with coefficients which are powers of two, and as the previous work shows, can be implemented efficiently.

Convolution is the simplest method of implementing image filtering, but there are other more complex methods such as FFTs which can provide more flexibility or better performance. The FFT, and its close relative the Discrete Fourier Transform, have been widely implemented in hardware. [3.19] An alternative, the Discrete Hartley Transform, which uses only real numbers, has also been implemented. [3.20]

Uzun et al. describe a system using an FPGA as a co-processor [3.21] in a frequency-domain image filtering environment. The FFT in this case is performed by several parallel processing elements, with each processing element containing a 7 stage pipeline and implementing a 1-D FFT. The 2-D FFT for image processing is composed of separate 1-D FFTs performed on each row and column of the image. The system is built using the Handel-C language, and shows a performance increase when compared with software implementations.

The FFT hardware does however tend to be very complex, due to the large number of multiplications required when computing with complex numbers, especially when using floating-point number representation. An example given in [3.21] uses 45% of the logic space in a Xilinx XCV2000E, which equates to 4320 CLBs, functionally equivalent to 17,280 LEs in an Altera Apex device.

This large-scale hardware usage is typical of many of the systems presented in this section, as in general high performance is more important than efficient logic cell usage in these image processing applications.

3.2: Image Processor Implementation

It was decided that the initial implementation would be a direct hardware translation of the basic flow of the operation, i.e. reading the pixels in turn and computing the result when all the required pixels had been read. The aim of this initial implementation was to determine if any speed-up could be achieved without optimising the architecture for hardware implementation. Once a working system was developed, future implementations would be more highly optimised to achieve maximum performance.

It was clear that an implementation using shift-registers to hold entire image lines would be significantly more costly in terms of logic element usage than one which holds only the data required to compute one output pixel.
The shift register design as used by Bosi [3.6] which holds M image lines for a mask size of M x M is feasible when a 3 x 3 mask is used, but becomes unfeasible for FPGA implementation with a 9 x 9 mask, as the number of shift register cells exceed the number of LEs available in the FPGA. Since a 9 x 9 mask was required, as this was the size used in the original work, a shift-register design was not used. The implementation was based on the flow of the software algorithm for the filter, reading the pixels one at a time and keeping a running sum, before dividing.

A memory-to-memory design was used, constrained by the features of the FPGA development kit introduced in section 2.5. Two banks of memory were used, one for the source image and the other for the result image. The image size was set to 256 x 256 pixels (65536 pixels total), though the hardware could easily be changed to accommodate larger or smaller images, with 512 x 512 being the largest square power-of-two dimensioned image which could fit into the memory on the board. If a square image was not required, it could be expanded to 512 x 1024 or 1024 x 512, to use all available memory. The dimensions do not necessarily need to be powers of two, but this makes the most efficient use of the memory, as the X and Y co-ordinates each fill a certain number of address bits completely, and the complete address can be produced merely by joining the X and Y addresses without needing to perform addition.

The overall form of the processor is shown in Figure 6. There are three main functional elements: the data processor (named DATAPATH in the VHDL code), the address processor (ADDRESSPROC) and the control unit (CSM). These are explained in detail below.

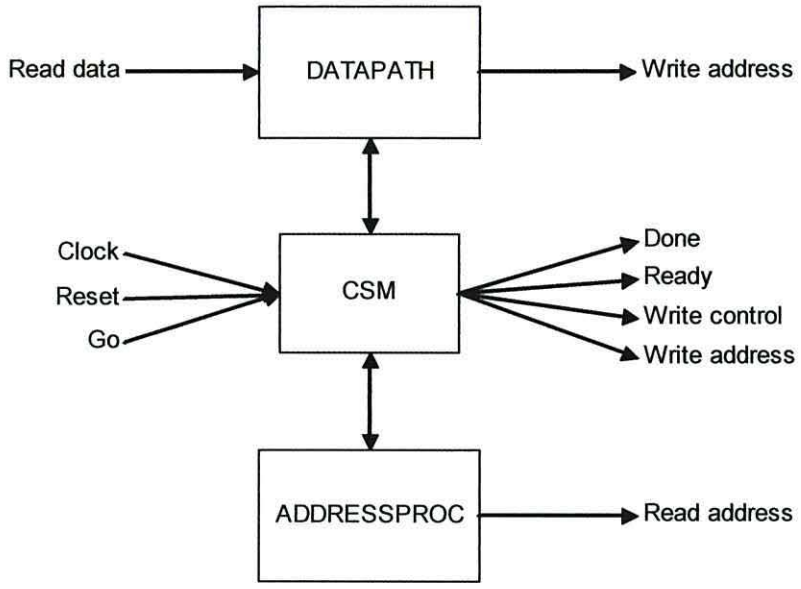

Figure 6: Block Diagram of the Convolution Image Processor

3.2.1: Data processor

This section performs the arithmetic on the image. Recalling the basic description of the process, the required processing consists of summing the pixel values over the mask area, dividing the sum by the number of pixels in the mask, and subtracting this value from the original pixel value. Making the assumption that an image of skin will not usually contain extremely sharp changes in brightness, we can see that this will usually yield a low value for the output pixel, so the image is then brightened using a gamma correction process. Figure 7 shows the layout of this hardware.

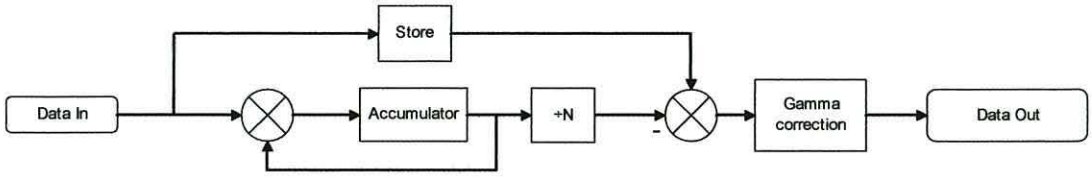

Figure 7: Block diagram of the data processing section

The incoming data is 8 bits wide, but the accumulator and adders must be wider to accommodate the accumulated values. The maximum possible input value will be 255, and since the original test system used a mask of 9 x 9 pixels, up to 81 such pixels can be accumulated, yielding a maximum possible accumulator value of 20655. This can be accommodated with 15 bits, so the accumulator, the first adder and the divider were built 15 bits wide. The output of the divider, which divided by 81 in this case, would never be greater than 255, so the subtractor and gamma corrector were built 8 bits wide.

Incoming 8-bit pixel data is padded to 15 bits by filling the upper seven bits with zeroes, and fed to the adder. This outputs the sum of the current accumulator value and the incoming data, which is clocked into the accumulator for each incoming pixel. A special case is when the pixel at the centre of the mask is read, in which case it is also clocked into a second storage register. Once all required pixels have been read, the result is available at the output of the data path without any further clocking. Only a short delay of 30-50ns is required to allow the data to propagate through the chain of processing elements.

The divider was built for speed rather than compactness, and as such is simply a large combinatorial logic circuit with 15 inputs and 8 outputs, whose VHDL definition was computer-generated. The code which generated this is shown in appendix A.1. Some code space and compilation time was saved by exploiting the fact that the maximum number which the divider would have to divide is less than the maximum number supported by 15 bits, in the case of the 9 x 9 mask this was 20655 vs. 32767, saving 12112 lines of code.

A physically smaller divider could have been used in place of this, though at the expense of speed, since the smallest dividers are generally those which use an iterative process or repeated shift-and-subtract.

These dividers would require several clock cycles to complete a division, although a pipelined system would be able to perform these while reading in and integrating the source pixels for the next output pixel, saving time.

Gamma correction is provided by a look-up table (LUT) consisting of a 256x8 bit ROM. The data from the subtractor is fed to the address inputs of the ROM, which provides the corrected data at its outputs. This allows quick changes of the correction curve during testing, as even though the ROM contents were fixed at compile-time, changing the ROM required only that the final assembly of the programming file was carried out, with no new logic synthesis or fitting being necessary.

The correction curve is shown in Figure 8. Input values are shown on the X axis, output value on the Y axis. This curve was determined experimentally, using Jasc Software's Paint Shop Pro [3.22] to apply gamma correction to an uncorrected image produced by an early test system. Once a correction curve had been found which produced an image in which the skin lines were clearly visible, the curve was programmed into the ROM, and was then used for all subsequent tests.

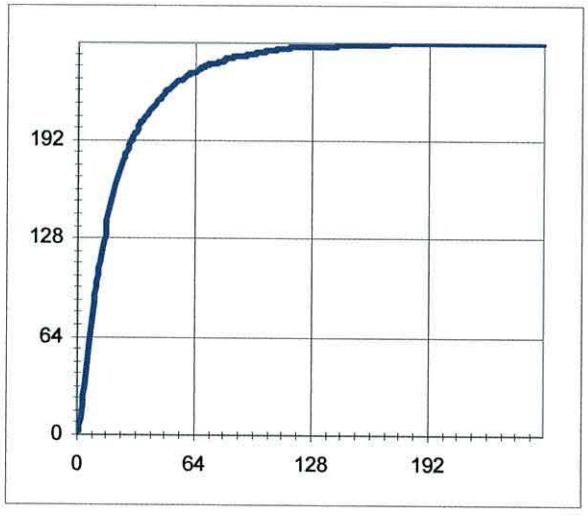

Figure 8: Gamma correction curve.

3.2.2: Address Processor

The address processor generates the read address from the pixel address and the mask count. The read address is calculated by adding an offset to the pixel

address, based on the currently selected mask pixel. For a 9 x 9 mask, the top left corner's co-ordinates will be (x – 4, y – 4), where (x,y) is the target pixel co-ordinate. Since the pixel address is a single binary number, it must first be split into X and Y co-ordinate components. The image size was chosen to be a power of two to make this split very simple. With an image of 256 x 256 pixels, or 2^8 pixels on each axis, the two bytes which make up the 16-bit address contain the two co-ordinates. In order to retain compatibility with standard computer image formats, the image is scanned horizontally, line by line, therefore the high order byte is the Y co-ordinate and the low order byte is the X co-ordinate.

For performance reasons, the original approach chosen to perform the offset was to split the incoming address into X and Y, and generate all offsets in parallel, selecting the appropriate pair with a pair of multiplexers, as depicted in Figure 9. This approach has the advantages of speed and coding simplicity, but generates a large amount of logic.

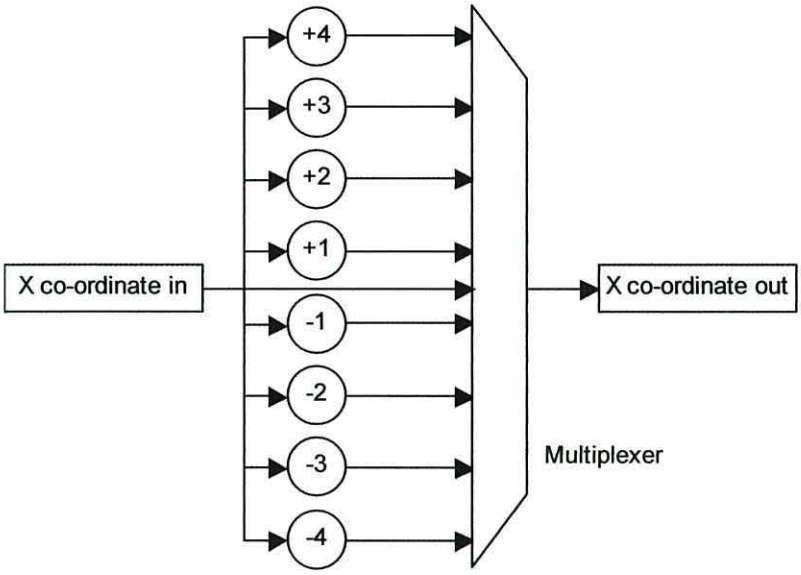

Figure 9: Example of a co-ordinate transforming multiplexer

A simpler approach which was used later replaced the multiple adders with a single 16-bit adder, which adds a single offset onto the address to produce the same result. Since the image size is fixed at 256 x 256, we can see that the pixel immediately above the target pixel will have an address which is 256 lower than the address of the target pixel. More generally, the pixel at (x + a, y + b) will have

an address of $P+(W \times b)+a$, where W is the image width and P is the address of the pixel at (x, y). For an image 256 pixels wide, with a 9 x 9 mask, the full set of mask offsets is shown in table n.

-1028	-1027	-1026	-1025	-1024	-1023	-1022	-1021	-1020
-772	-771	-770	-769	-768	-767	-766	-765	-764
-516	-515	-514	-513	-512	-511	-510	-509	-508
-260	-259	-258	-257	-256	-255	-254	-253	-252
-4	-3	-2	-1	0	1	2	3	4
252	253	254	255	256	257	258	259	260
508	509	510	511	512	513	514	515	516
764	765	766	767	768	769	770	771	772
1020	1021	1022	1023	1024	1025	1026	1027	1028

Table 1: Mask offsets for a 256 pixel wide image

The offset is generated by a look-up table (LUT) which is implemented as a small block of VHDL-coded ROM. The address for the ROM is the mask co-ordinate generated by the mask counter in the control unit. It should be noted that the two versions of the address processor perform slightly differently when used at the very edges of the image. Processing the X and Y co-ordinates separately means that if the mask moves off the edge of the image, for example if the centre pixel is on the right-hand edge of the image, the section of the mask which has left the image area will in fact 'wrap' around to the opposite edge, remaining aligned vertically with the rest of the mask. This is because there is no carry from the X co-ordinate to the Y co-ordinate, so as the X co-ordinate rolls over through zero, the Y co-ordinate is unchanged. Adding a single offset to the pixel address will cause the mask cells which have moved off the edge of the image to reappear on the opposite edge one row down (assuming it is the right-hand edge into which the mask is moving). This difference doesn't matter in the case of the convolution filters, as the mask is never placed anywhere where it would have pixels outside the image boundary, but for the cellular automaton processor in section 3.8 correct wrapping at the edges is essential and so two individual co-ordinate transformers must be used.

An additional logic circuit checks the current pixel address for validity. An invalid pixel is defined as one which is close enough to the edge of the image that the

mask could not be placed around it without some of the mask pixels being outside the image boundary. These pixels are not processed by the system, and so the output from this validity-checker is used by the state machine to decide whether to proceed with the inner processing loop. If the pixel is invalid, it is skipped to save processing time and a black pixel is written to the processed image.

This checker splits the pixel address into X and Y co-ordinates and checks that both co-ordinates are greater than 3 and less than 252, for a 9 x 9 mask.

3.2.3: Control Unit

The control unit consists of a state machine and a pair of counters which generate the memory addresses and control signals for the system. The master pixel counter generates the address of the target pixel which is to be processed. The mask counter is used to count mask pixels, so for a 9 x 9 mask this counts from 0 to 80. The mask count is used by the address processor to modify the pixel address to fetch the correct group of source pixels.

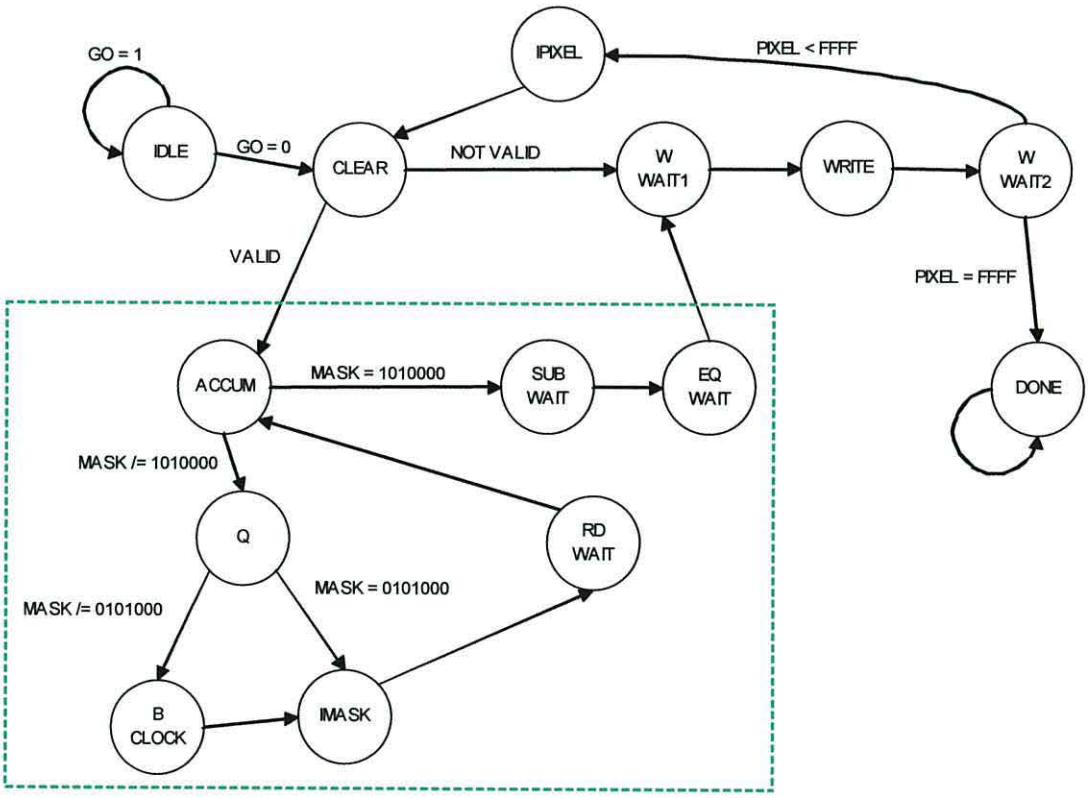

Figure 10: State transition diagram for the control unit

Figure 10 shows the state transition diagram for the control unit. When reset is asserted, the system enters the idle state, where it will remain until the active-low GO input is asserted. The supporting logic issues a clear signal to the address counter when the state machine is in the idle state, to ensure that the processing operation begins at the correct point.

Upon receiving the GO signal, the system enters the main processing loop with a transition to the CLEAR state. While in this state signals are issued to clear the accumulator and mask counter, and the next state is chosen by looking at the VALID signal from the address processor. If the pixel is valid, the next state will be ACCUM, starting a new integration cycle. For invalid pixels the next states will be the chain consisting of WWAIT1, WRITE and WWAIT2, which write the accumulator to memory at the currently selected pixel address.

After writing a pixel, the state machine will either enter the DONE state, if the pixel address is $FFFF_{16}$, or go to the IPIXEL state, during which the pixel address is incremented. After IPIXEL the state machine returns to the CLEAR state.

The pixel processing is performed by the states inside the dashed box in Figure 10. Starting in the ACCUM state, with the accumulator cleared and the mask count at zero, the state machine passes through states Q, IMASK and RDWAIT until the mask count reaches 80 and all pixels in the mask have been read and accumulated. The exception occurs when the mask count reaches 40, when an additional state, BCLOCK, is used to write the centre pixel into the centre pixel register.

3.3: Alternative Filter Masks

3.3.1: Second – order High-Pass Filter

The high pass filter is an effective method of extracting the skin lines, but it can be costly in terms of processing time and logic element usage due to its large mask size and the need for a divider. Therefore some speed increase could theoretically be obtained through the use of a filter requiring fewer source pixels to be read. One such simpler filter is the second-order or Laplacian filter [3.9], which uses a mask of 3 x 3 pixels, of which only five need to be read.

	1	
1	-4	1
	1	

Figure 11: Second-order filter mask

Since the mask weights are either 1 or -4, this is a simple function to implement in digital hardware. A multiplication by four can be performed with a left - shift of two bits while the negation can be performed by using a subtractor rather than an adder. The remaining hardware is very similar to the averaging filter. The pixels read from the source image are either added to the accumulator or multiplied by four and subtracted from the accumulator. No division is required as the sum of the weights is zero.

This filter essentially performs a high-pass or edge detection operation, and whereas after performing the low-pass filtering with the averaging filter, it is necessary to subtract these filtered components from the original image, there is no such requirement with the second-order filter. Also, while the averaging filter, by subtracting one image from an essentially similar one, will generally produce low pixel values, the lack of such subtraction in the second order filter means that the gamma-correction is not necessary and can be removed. These simplifications mean that this filter has the advantage of requiring much less hardware than the filter of section 3.2, while also exhibiting higher performance.

3.3.2: Gaussian Blur filter

The Gaussian blur filter is a low-pass filter that performs the same function as the averaging filter, but produces a weighted output average of the neighbourhood of each pixel. This tends to preserve edges more than the averaging filter, and has a gentler response for a given mask size.

The Gaussian coefficients are shown in Table 2. For a mask size of N x N, row N is taken from the table, and the outer product is found with the transpose.

1								
1	1							
1	2	1						
1	3	3	1					
1	4	6	4	1				
1	5	10	10	5	1			
1	6	15	20	15	6	1		
1	7	21	35	35	21	7	1	
1	8	28	56	70	56	28	8	1

Table 2: Gaussian coefficients

It can be seen from the values in the table that these approximate a Gaussian distribution for large N. [3.23]

From the table of coefficients, we see that for a 3 x 3 mask, the final mask values will be:

$$\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

The mask coefficients sum to 16, so this must be followed by a division by 16. This is trivial to implement in hardware, as a division by 2^n is a right shift of n bits. It is similarly easy to implement the multiplications by two and four required when weighting the pixels according to the mask values using left-shifts. For a 5 x 5 mask, the coefficients are:

$$\begin{bmatrix}1\\4\\6\\4\\1\end{bmatrix}\begin{bmatrix}1 & 4 & 6 & 4 & 1\end{bmatrix}=\begin{bmatrix}1 & 4 & 6 & 4 & 1\\4 & 16 & 24 & 16 & 4\\6 & 24 & 36 & 24 & 6\\4 & 16 & 24 & 16 & 4\\1 & 4 & 6 & 4 & 1\end{bmatrix}$$

The sum of these coefficients is 256, which is again simple to implement, but the multiplications by the mask values are more difficult to implement, as not all are powers of two. However, these multiplications, by 24 or by 36, can be broken down into multiplications and additions which can be achieved as follows:

$$24n = 8n + 16n$$
$$36n = 4n + 32n$$

Reducing these multiplications to bit-shifts and additions usually results in a faster or less complex logic circuit, especially in cases where only one addition is required. In those cases when several shifted terms must be added, the adder will have to be broken down into a tree of two-input adders, which will multiply the propagation delay of a single adder by $\log_2 N$, for N inputs, where $N > 2$.

The final divisor required for a mask of size m x m is 2^{2m-2}, so for the 9 x 9 mask used by the test system the divisor will be 65536. This implies that the data buses carrying the sum of the weighted pixels will need to be at least 17 bits wide, compared with the 15-bit bus used with the averaging filter. However, the division by a power of two is simple to implement and removes the need for the hugely complex divider as used in section 3.2.

3.3.3: Median Filter

The median filter produces the average of the pixel values within each pixels neighbourhood by taking the median of these values, i.e. by taking the middle value when the pixels are sorted by brightness. This has the advantage of preserving edges more accurately, but is much more computationally intensive than the simpler averaging functions, as it requires that the pixels under the mask be sorted. In a hardware implementation, this sorting would be carried out by a

sorting network [3.24] consisting of several 2-input compare-and-swap blocks, each of which can sort two values. These sorting networks quickly grow in size as the number of inputs grows. For example the odd-even transposition sort, which is one of the simpler ones, requires n stages for n inputs, as shown in Figure 12 for an 8-input network. For a 9 x 9 mask this network would be 81 stages long, though as the stages repeat it would be possible to implement a partial network of two stages and apply this 41 times, feeding the outputs back to the inputs each time.

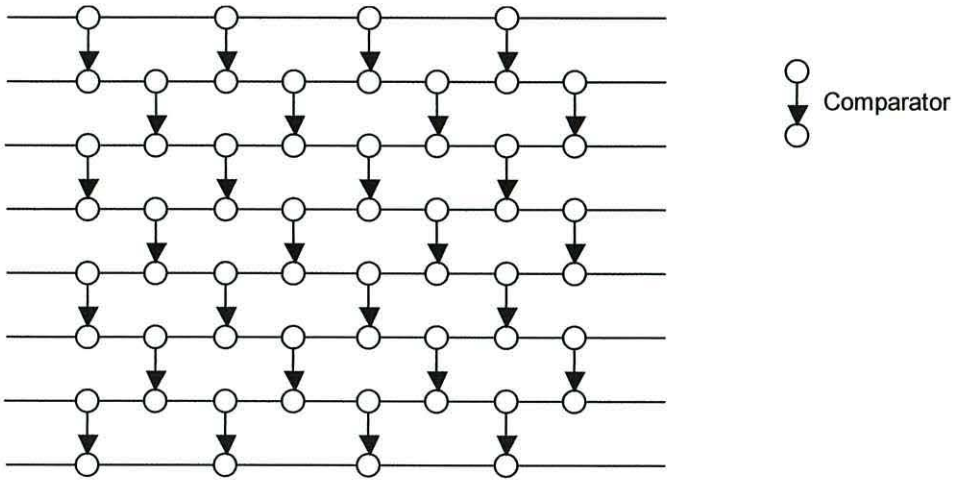

Figure 12: Odd-Even Transposition Sorting Network

3.4: Alternative Implementation: Second-order Filter

The second-order filter described above was implemented as an alternative to the standard blur filter. It was decided to implement the second-order filter because the processing is very simple, with only 5 pixels being read per output pixel, and there being only two values of mask weights and no requirement for a divider. This makes the second-order filter much simpler than the Gaussian filter or Median filter implementations would be, as the arithmetic requirements for the second order filter are minimal.

The complete system is depicted in Figure 13. This was implemented as a single unit which is externally compatible with the convolution filter, and can replace it in a test system without any further changes being necessary.

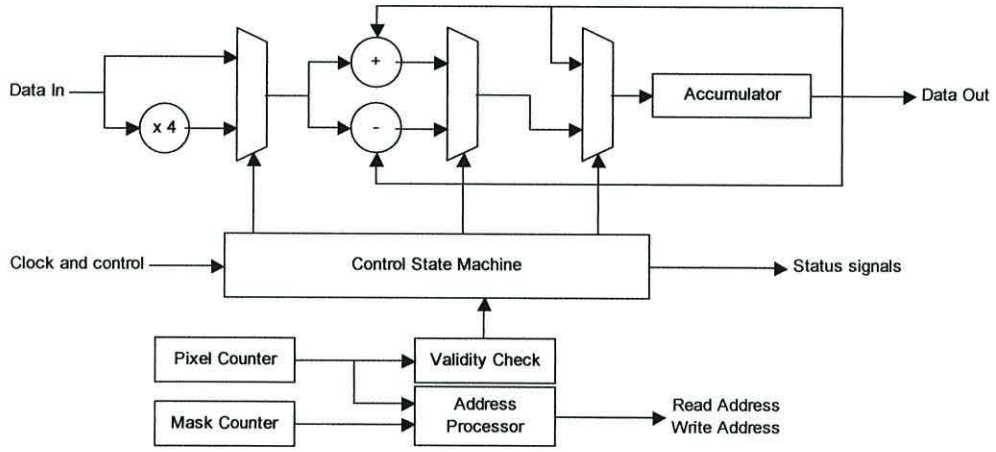

Figure 13: Layout of the second order filter processor

The simplified processing section consists of a series of simple functions joined by multiplexers. The first of these multiplexers selects either the raw input data, or the output of a multiply-by-four circuit that shifts the incoming data left by two bits. The output of this multiplexer is fed to an adder and a subtractor, both of which take their other input from the accumulator. The second multiplexer thus selects whether the output from the first is to be added to or subtracted from the accumulator. The third multiplexer is used to select whether the accumulator is to be updated or not, which is required because the accumulator is clocked on every cycle, to avoid having to feed its clock through logic, ensuring glitch-free operation. The state machine is clocked on the opposite edge of the clock from the

registers, so that its multiplexer controlling outputs are stable when the registers are clocked.

The pixel and mask counters are also built synchronous, and are clocked at the same time as the accumulator. An additional pair of multiplexers are used to ensure that these are only incremented when required.

The state machine controlling this filter system is also a little simpler than the convolution controller. The state diagram is shown in Figure 14.

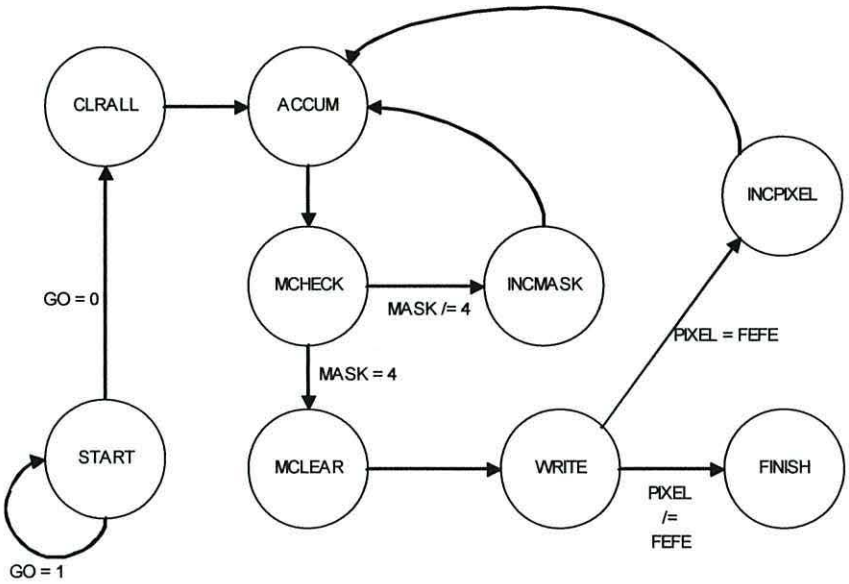

Figure 14: Control State Machine for the Second-Order Filter Processor

Processing begins in the CLRALL state, which clears the registers and sets the pixel counter to 0101_{16}. This is the top-left-hand corner of the valid area of the image. During the ACCUM state the accumulator is updated, and the arithmetic circuitry before it ensures that the correct operation is performed on the incoming data, either an addition or a subtraction, and either shifted or not, based on the mask counter value. The mask is checked in the MCHECK state, and if there are still pixels to be read, the process repeats. When the mask counter reaches 4, the last of the required pixels have been read, and so the mask counter is reset, the result pixel is written, and if the current pixel isn't the last valid one, the pixel counter is incremented in state INCPIXEL. If the validity checking logic in the

address processing section determines that the pixel is invalid, the system remains in state INCPIXEL until a valid pixel is reached. Thus, with a border of 1 invalid pixel down each edge of the image, two extra cycles are required at the end of each row, to skip the border pixels at the end of the row and the start of the next one. For pixels at the end of a row, 19 cycles are required, for all others, 17 are required.

3.5: A Test-bed System

In order to test the image processors, a system was required which could place a source image in one memory bank, allow the processor to perform its operation, and then retrieve the processed image from the second memory bank. Originally the reconfigurability of the FPGA was exploited and these three jobs were performed by three individual designs. The FPGA's internal circuit could be changed without disturbing the images in the RAM, as the RAM is external to the chip. However, it was quickly found that this was a time-consuming way to process the images, and the three processes were combined into a single test-bed system.

An overview of the system is shown in Figure 15.

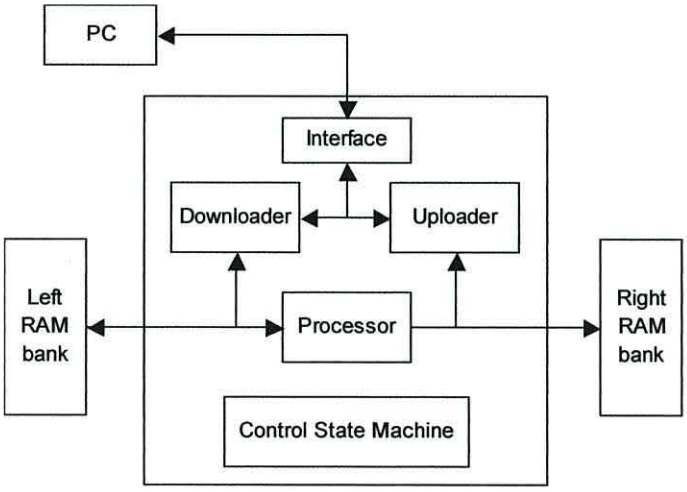

Figure 15: Overview of the image processing test bed

The test bed was built to fit the arrangement of components on the Digilab board. The two RAM banks on the board itself are 512K x 16 bits, wired as two 512K x 8 SRAM chips. The processor uses 8-bit greyscale image data, thus leaving one half of each physical data bus free.

In the case of the left-hand (source) memory bank, this unused half was left unused, but with the right-hand bank there is an extra socket connected to one half of the bus which allows ROM devices to be fitted. This socket was used to provide the interface to the host PC, using the data lines for communication,

leaving the address lines untouched so they, together with the other 8 data lines, could be used to access the memory. The electrical side of the interface was an Altera ByteBlaster MV [3.25] JTAG interface, which would normally be used to configure the FPGA. This provides the level translation between the FPGA's 3.3V CMOS I/O pins and the 5V TTL compatible parallel port of the PC, allowing high-speed transmission. In practice, due to problems with noise and with the software on the PC, data rates were somewhat limited, requiring nearly 30 seconds to transfer an image. This gives a calculated data rate of around 17Kbits/sec. Clearly something faster is required for any practical application, but no further developments were made. This is not a problem as there are plenty of ready-made communication controller solutions [3.26] available for inclusion in future FPGA designs. These include fast RS232 serial and USB protocols.

The downloader and uploader were designed to be compatible with the image processor in terms of the control and status signals, so when the chip is reset, each block will present a 'ready' signal once it has performed any necessary initialisation, then wait for a 'go' signal. Once given this signal it will perform its function, usually outputting a 'busy' signal as it does so. When finished, it removes the 'busy' signal, and asserts a 'done' signal to indicate that it no longer needs control of the bus.

The overall operation of the system is controlled by a master state machine which sends out the triggering signals and monitors the status lines. A second state machine consisting of five states tracks the progress of the first, and is used to remove glitches from the bus control outputs. The state diagrams for these are shown in Figure 16. The bus control outputs from the control block are responsible for controlling the multiplexers and tri-state buffers which set up the data path between the downloader, processor and uploader, and the two memory banks. Since the master state machine goes through several states for each of the three stages, these control lines would need to be active for several states, and glitches may occur at the transitions from one state to the next. The second state machine removes these glitches by representing each phase with a single state, and moving from one to the next as the master state machine does.

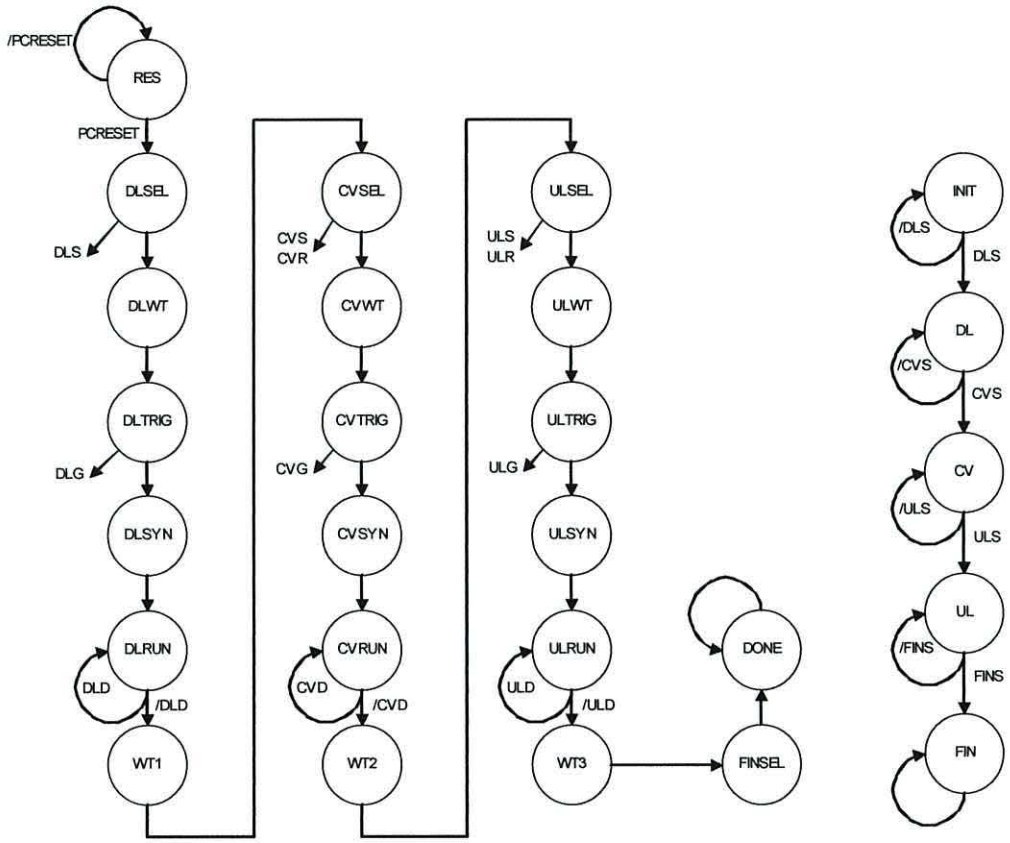

Figure 16: State machines for the test-bed controller

For each process (DownLoad, ConVolution, UpLoad) there are five main states. The SEL (Select) state is used to move the second state machine to the correct phase, and it is in this state that an additional reset pulse is given to the relevant unit to ensure that it is working correctly. Following this the WT (Wait) state allows time for the buses to settle and the selected unit to initialise, before the TRG (Trigger) state issues the command to start it. The SYN (Synchronise) state wastes some time before any of the status signals are checked, to ensure that the current phase is in action before its status is checked. The RUN state is entered, and it is here that the system will remain until the 'done' signal is issued.

3.6: Testing the Image Processor

In order to be able to perform the task for which it was designed, the system must be able to detect sharp changes in brightness from one pixel to the next, corresponding to the skin lines, while ignoring the more gradual changes corresponding to areas of skin colouration or shadow. It must also be able to detect these changes regardless of the average colour of the image, as it may be required to process images of a wide range of different skin tones. A number of test images were created which were processed by the two image processors. These were intended to reveal any flaws in the processing and to provide a view of the performance of the two systems relative to each other. Many of the test images were designed to replicate features which would be found in the real-world skin images.

Note that when images are presented, the source image is on the left, with the high-pass filtered result in the centre and the second order filtered result on the right. In some cases, though it may not be noticeable in print, there may be a 1-pixel wide border of random noise around the edges of the second-order result images, as the second-order filter completely skips the invalid pixels and doesn't write anything into the second memory bank for these, leaving the random data which was present at power-up.

3.6.1: Basic Edge Detection Tests

Since the system works by high-pass filtering, a sharp change in brightness should result in a strong output, i.e. a brighter pixel in the output image. A gentler change in brightness should result in a dimmer pixel. The aim of this basic test was therefore to determine that the system responds correctly to changes in brightness, and that it can distinguish between sudden and gentle changes. Figure 17 shows a test image consisting of a grey half and a black to white gradient half. The grey half has a brightness value of 128, half way between the black and white values. There should therefore be no difference in brightness between the grey half and the middle of the gradient and an increasing difference above and below this point. The expected outcome was that the detected 'edge' would be stronger at the top and bottom of the image where the brightness step is greater, diminishing to

nothing in the middle, and that although the gradient half of the image has a continuous change in brightness from top to bottom, it should be gentle enough that it would not be detected as an edge.

Figure 17: Gradient test image, high-pass result and 2nd order result

The processed results are shown in the centre and right images. As expected both systems found no edge in the centre of the image, while the strongest edges were found at the top and bottom, where the colour difference is greatest. It can also be seen that while the high-pass filter places its line on the lighter side of the edge, the second order filter places its line on the darker side, an effect which is more noticeable in the case of the high pass filter due to its much wider output line.
This effect can be seen more clearly in Figure 18, which shows an extract from a checkerboard pattern with 16-pixel wide squares.

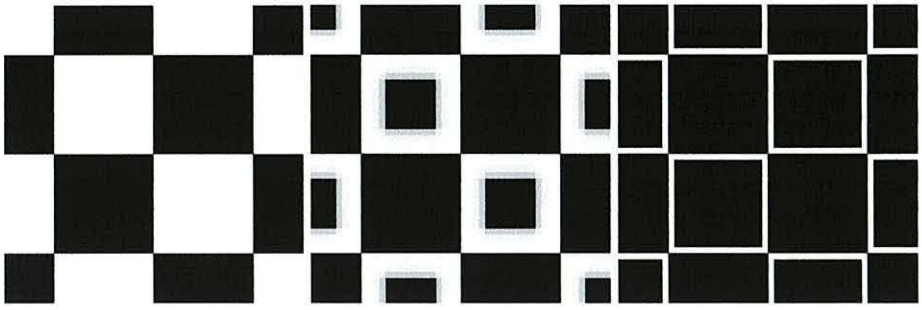

Figure 18: Enlarged section of checkerboard test pattern and result images

Both filters have found the edges of the white squares, but the second order filter has placed its output pixels outside these squares, which gives the appearance of having detected the edges of the black squares.

The reason for the second order filter's line placement can be seen if we consider a section of an image at an edge, as depicted in Figure 19.

255	255	255
255	255	255
0	0	0
0	0	0

Figure 19: Pixel values for a small image section

The two pixels to consider are shaded. When processing the upper pixel, the second order filter will read three pixels with values 255, one with value 0, and the centre pixel with weighted value -1020. The sum of these is -255, which is clipped to zero, and results in a black pixel. For the lower shaded pixel, the sum of the five pixels read by the filter is 255, resulting in a white pixel. Therefore, wherever a bright section of the image appears, the second order filter will tend to place bright pixels around but not inside its boundary.

The high-pass filter applies a low-pass filter to the image then subtracts this from the original image pixel. So, if the centre pixel under the mask is darker than the average of those around it, as would happen in the case of the lower shaded pixel in the image section, the result of the subtraction will be negative, and will be clipped to zero. The high-pass filter will therefore not place output pixels on the darker side of such an edge.

A second test of the edge detection capability was performed with an image consisting of a white half and a black half, with an increasingly blurred edge between them. The aim was to determine that the edge detector could tell the difference between a sharp edge and a softer edge. In this case, in contrast with the first test shown in Figure 17, the edges in this test are always between areas of maximum contrast, so it is the slope of the change in brightness which is varied, rather than the magnitude. The expectation is that because the magnitude of the change is the same in all cases, the system will be able to detect the edge all the way along, despite the softening of the edge from top to bottom.

Figure 20: Blurred boundary test image and result images

The result of this second test of the edge detection capabilities is shown in Figure 20. For the high-pass filter, the resulting image, in the centre, shows that the detected edge is darker and weaker in the areas with a slower brightness change than in areas with a sharp change. The edge line curves slightly because the detected line lies in the brighter half of the source image's gradients rather than on the centre line.

In the 2nd order filter's case, the detected edge is placed at the darker edge of the transition, and short horizontal lines indicate that edges have been detected between the sections of differing blur.

It is clear from this image that the 2nd order filter is more suitable for simple edge detection than for extracting small features in an image.

3.6.2: Low-frequency test

The purpose of the filtering hardware is to pick out parts of the image with a high spatial frequency, ignoring slow or gentle changes in background intensity. The low-frequency test is therefore intended to check that the system doesn't detect spurious edges within images consisting only of slow changes in brightness. The test image was created by applying a heavy blur to an image consisting of solid blocks of different grey levels. This resulted in an image in which the brightness changed smoothly between the centres of these blocks. The expected result of the test was that the image would not contain any detected edges, being completely dark.

Figure 21 shows the test image which was used, along with the results produced by the two systems. The result images are inverted and contrast-enhanced.

Figure 21: Low spatial frequency test and result images

The results of this test were almost completely black, with a very faint pattern of lines corresponding to the step changes in brightness due to the quantization, both spatially and intensity-wise, of the image. The second order filter showed a finer pattern of quantization noise, due to its smaller mask size and correspondingly higher sensitivity. Although both contrast-enhanced images appear to have similar levels of noise, the high-pass filter's output produced a noise pattern consisting of broader bands at a lower intensity than those produced by the 2nd order filter. This shows that there would be less noise in the output of the high-pass filter, making it more suitable for processing the skin lines.

3.6.3: Line Detection

The low-frequency test image was overlaid with faint lines to test the processor's ability to extract the details with a non-constant background. A test image was created which consisted of a series of lines drawn onto the image from section 3.6.2. The expectation was that the result would contain a depiction of the lines but not the varying background, the latter varying too slowly to be detected by the system. The results of these tests are shown in Figure 22.

Figure 22: Line test image and result images

The results of this test show that while both systems detected the lines, the output from the high-pass filter can be said to be the more accurate one, as it shows the lines themselves, whereas the second-order filter has placed lines around the positions of the lines in the source image. Referring back to the grey gradient test above, it was seen that the high-pass filter tends to place its output on the lighter side of the detected edge, and with the lines in this test image being generally brighter than the background, this filter will tend to produce output inside the lines, while the second order filter will tend to place output outside the lines. The larger mask size and correspondingly wider output lines of the high-pass filter can also clearly be seen.

It should be noted that the lines placed onto the background in this test were of constant brightness, which resulted in the detected lines being stronger in areas where the background was darker, and therefore the contrast was greater. A pair of new test images were created in which the difference between the lines and the background was more constant, i.e. the lines either lighten or darken the background by a constant amount.

Figure 23: Bright line test and result images

The first of these test images is shown in Figure 23. In this image the lines are narrower and more detailed, and are overlaid in such a way that they brighten the background rather than replacing it, producing a more constant difference in brightness. This is reflected in the result images, in which the variation in line intensity is smaller, as was expected. Again, the second-order filter produced outlines of the detected details, which, due to the narrow lines, gives the image an out-of-focus appearance. However, both filters have successfully extracted the lines.

The same combination of background and lines was used in a third test, but with the lines slightly darker than the background. The test image and the two results are shown in Figure 24.

Figure 24: Dark line test image

Here, the second-order filter, due to its particular result placement, produces the better result, though both processors have identified the lines, the lines are much better defined in the second-order output.

3.6.4: Skin Image Tests

The system was tested further with a range of images of skin lesions taken during the work in [3.1]. The particular set of images presented were chosen to demonstrate the system's response to a variety of different patterns and lighting conditions, covering the range of images which the system would be likely to encounter. A selection of these and their processed results are presented below.

Figure 25 : Sample skin lesion and processing results

Figure 25 shows a section of a lesion with well-defined lines, and the two processed result images. It is clear that the high-pass filtered image, shown in the centre, shows the skin lines much more clearly than the 2nd order filtered image, which shows a much finer level of detail in which the relatively coarse skin lines are lost. This was found to be the case for most of the skin images, and is a result of the difference in mask sizes between the two filters. With its 9 x 9 mask, the high-pass filter is less sensitive to the small details and tends to extract the larger skin lines only.

The higher sensitivity of the second order filter could be a problem if the image is noisy. Such noise in the image could arise from noise in the image sensor, or from quantisation or encoding noise. To test this, a sample skin lesion image obtained from the internet [3.27] was used, which had a reasonable degree of distortion due to JPEG compression artefacts. JPEG compression divides the image into 8 x 8 pixel squares before performing DCT functions, and it is at the edges of these squares that subtle discontinuities in the image can occur. It was expected, based on the quantisation noise exposed by the low-frequency test (Figure 21) that the

step changes at these boundaries would appear in the processed image as faint edges. The results of the test are shown in Figure 26.

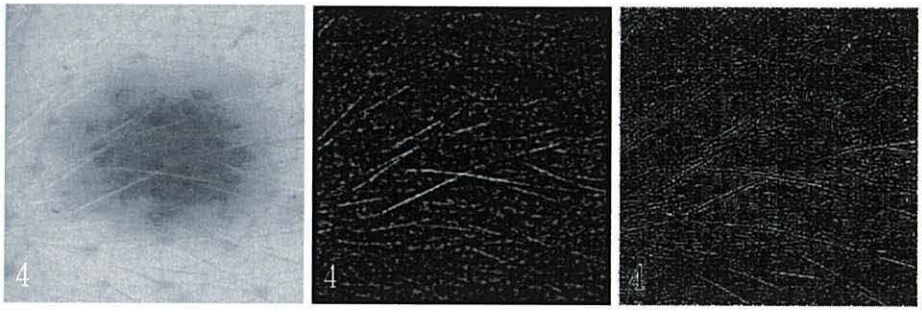

Figure 26 : Skin line test image with JPEG artefacts

The distortion is visible as a grid-like pattern of noise from the more sensitive second-order filter, while the high-pass filter has largely ignored this and extracted the larger features. In this case another problem with the filtering is revealed, as both filters have picked up the hairs lying across the lesion much more strongly than the skin texture. It is assumed that the following analysis of the processed image would include some function which could detect and ignore the longer lines produced by the hairs. [3.28]

A test was carried out with an image exhibiting poor contrast and detail, to determine how well the systems could extract the skin lines when they were not clearly visible with the naked eye. An image was used in which the skin lines were very faint, with the expected outcome being that the system would fail to detect any strong lines. Following on from the previous results, the second-order filter was expected to show more detail in its output pattern, but it was not expected that this detail would include the skin lines.

Figure 27 shows the source image and the two test results. Neither result image shows much detail, and the results are unlikely to be of much use to the subsequent processing systems.

Figure 27 : Skin line test image with low detail

The high-pass filtered image does show the skin lines very faintly, but with a lot of noise, while the second order filtered image is almost entirely noise. The expectation is that this case would be rare, as if the images are to be processed based on the skin texture, an imaging system will be employed which shows the texture properly.

If the image has clearly visible skin lines, it is expected that the result image will be much clearer. An example of such an image is shown in Figure 28.

Figure 28 : Skin line test image with good detail

Here, both filters show the skin lines clearly, but as with most cases there is less noise in the high-pass filtered output. The difference in the form of the skin lines between the inside and outside of the lesion can also be seen. It is this difference which forms the basis of the work in [3.1, 3.2], and would be the focus of the system which uses the result image.

3.6.5: Test Conclusions

Overall, the tests have shown that the two systems work, in that they are capable of extracting details from the source images, and that they perform as expected. The edge detection tests also showed that the arithmetic was being performed correctly, so that negative results were clipped to zero rather than simply being written as spurious positive values. The gamma correction is functioning correctly, as shown by the bright and clear output lines.

It was seen that the second-order filter tended to produce an output with more detail than the high-pass filter, due to its smaller mask size, though when tested with real images of skin lesions this smaller mask size tended to make it more susceptible to noise. With these lesion images the high-pass filter gave an output which was visually clearer than the second-order filter's output, Therefore it is concluded that the high-pass filter would be more suitable for the processing carried out by the subsequent systems (referenced at the start of this chapter).

The importance of a good quality source image was shown, as the image must have good contrast and an absence of noise.

3.7: Analysis of Designs

Since the convolution engine is controlled by a state machine, which performs a fixed series of operations to process each pixel, it is possible to calculate the total number of steps required to process the entire image.
Assuming a square image of N x N pixels, and a square mask of M x M pixels, we can calculate the following:

First, the image pixels can be classified into two types: Valid and Invalid. An invalid pixel is one which falls outside the area to which the centre of the mask can be moved. This invalid area is a border of INT(M/2) pixels width around the entire image, where the INT function rounds down to the nearest integer.
An invalid pixel will always take 5 states to process, since this is the number required to check a pixel for validity and advance the address counter if it is invalid. A valid pixel will require a larger number of states, dependant on the mask size.
In the integrating loop, four states are required for each pixel in the mask, except the centre pixel which requires five. Therefore each output pixel calculation requires $4M^2+1$ cycles for a mask of M x M pixels.
Adding the states required to complete the processing and write the result to memory, the total number of states required for the image is:

$$V(4M^2+8)+5N$$

Where V is the total number of valid pixels and N is the total number of invalid pixels. For an image of size P x P pixels, the number of valid pixels is

$$V=(P-(M-1))^2$$

The number of invalid pixels is then

$$N=P^2-V$$

or

$$N = 4(P - (\frac{M-1}{2}))(\frac{M-1}{2})$$

The total cycle count for a range of different image and mask sizes is shown in Table 3.

	Mask size			
Image size	3	5	7	9
32	40220	85872	139644	193472
64	170396	391280	689916	1045952
128	701084	1665648	3043836	4790720
256	2843804	6868592	12765180	20439488
512	11454620	27891312	52261884	84373952
1024	45977756	112404080	211471356	342790592

Table 3: Clock cycles required for various combinations of image and mask size

Knowing the number of cycles required to process an image allows the total processing time to be calculated, provided that the clock frequency is known. The Quartus II software provides detailed timing analysis, allowing the maximum clock frequency to be determined. The propagation delays of each part of the data path will inevitably depend on the layout of logic elements in the compiled and fitted FPGA, and this can vary slightly with each compilation as changes are made to the other hardware around the image processor, but it is possible to obtain an estimate of the maximum time required to perform each step of the processing.

The basic convolution image processor requires 1586 LEs, 48 of which are registers, and 860 of which are used in the data processor. The address processor accounts for 430 LEs. 2048 ROM bits are required to hold the gamma correction table. This large LE usage is due to the simple ways in which the divider and address processor are built; the divider is a plain combinatorial logic circuit which takes a single clock cycle to perform a division, but in doing so takes up a lot of space as it is essentially a look-up table synthesised using LEs. The address processor generates the addresses of all of the pixels under the mask simultaneously, then selects one based on the mask counter's output.

This inefficient address processor was replaced as described in section 3.2.6, with one which contains a single adder and adds an offset to the address. This reduced the logic element count for the address processor section to 77, or 71 when the convolution engine is merged with external test-bed hardware. In this case the offset table was implemented in the same way as the divider, as a look-up table generated using LEs. If the table was implemented using the FPGA's embedded memory, the estimated LE count would be around 32, consisting of the adder and validity checking logic.

It was also found that merging the convolution processor with the test-bed hardware reduced the logic element count quite considerably. When compiled on its own, there were an additional 255 logic cells used in implementing the top-level part of the design, whose function is merely to connect the state machine, data processor and address processor together. These cells were not present in the hierarchy when the processor was used in the test-bed and so the total logic element usage for the processor was 995, compared with 1233 when compiled alone (with the more efficient address processor). These extra cells represent logic which was merged with the rest of the test-bed when compiled in this way, and so were not counted as being explicitly part of the processor.

A further large reduction in logic element usage could be made by replacing the divider with one based on repeated shifting and subtraction, though this would also increase the number of clock cycles required to perform the division.

The adder and subtractor were found to require a maximum of 15ns to perform their functions, with the divider and gamma corrector taking up to 30ns. The longest delay quoted by Quartus for the address processor was around 23ns. These values will depend on the target FPGA architecture, and these particular results were obtained for an Apex 20KE device of the fastest speed grade. Experimentation with different FPGA families suggested that further speed improvements were possible, but not necessarily significant, with no more than a 2-3ns reduction in general. Hence it has been further confirmed that the results obtained for the Apex 20KE are still relevant even in the light of more recent developments in FPGA technology.

From the state diagram for the control unit, it can be seen that three states, SUBWAIT, EQWAIT and WWAIT1 are executed before the processed pixel is

written to the memory. Therefore the time taken up by these three allows the divider, subtractor and equaliser to perform their functions. The total processing time for these three units, and therefore the minimum time these three states can take, is around 75ns, or 25ns per state. This corresponds to a clock frequency of 40MHz, and for the test system, using a 256 x 256 image and a 9 x 9 mask, a total processing time of 511ms.

The second order filter, with its simplified state machine, requires 17 cycles for most pixels, or 19 if the pixel is at the end of the row, within the 254 x 254 pixel 'valid' area. Thus, there are 64262 pixels requiring 17 cycles and 254 requiring 19. This results in a total of 1,097,280 cycles required to process the image.
When compiled on its own, this system uses 173 logic elements, and Quartus estimates a maximum clock frequency of 114MHz, assuming the target device is an Apex 20KE FPGA with a –1 speed grade. This higher clock speed is due to the absence of the divider and gamma correction LUT, which add a delay to the data processor.

If operated at 114MHz, the system would require 9.6ms to process an image. For comparison with the high-pass convolution filter, if operated at 40MHz the second-order filter would require 27.4ms to process the image. This is 18.7 times the speed of the high-pass filter, due to the smaller mask size and simpler state machine.

A comparison of the two designs is shown in Table 4. Area-time products are also shown, in terms of both Fmax and the time taken to process the image. In both cases a larger number represents a better system. In the last column the numbers show the image area divided by the area-time product of the processor, to make it consistent with the results shown in Table 5.

System	LEs	Fmax	Time (ms)	Fmax / LEs	P/(Time * LEs)
HPF (first)	1586	40	511	0.03	0.08
HPF (smaller)	995	40	511	0.04	0.13
Second-order (same speed)	173	40	27.4	0.23	13.83
Second-order (max. speed)	173	114	9.6	0.66	39.46

Table 4: Performance Comparison of Image Processors

It is clear from the table that the second-order filter has a major advantage over the high-pass filter, with a significant increase in the area-time product, even when operated at the same clock frequency as the high-pass filter. Both the reduced LE count and the more efficient processing are responsible for this increase.

Comparing with a software implementation, the DSP implementation described in [3.6] requires 20 instruction cycles per pixel when using a 3x3 mask, making use of advanced features of the DSP such as parallel instructions. For a 256 x 256 image this would therefore require 1,310,720 cycles. The fastest 320C40 DSP has an instruction cycle time of 33ns, therefore this would require 43ms to process the image. Given that the DSP can perform parallel operations and is optimised for signal processing, it is certain that a general-purpose processor would require many more cycles.

The second-order filter is faster than this DSP implementation, requiring fewer cycles to complete an image, and also being capable of operating faster. This is however a specialised architecture compared with the general-purpose one in the DSP implementation. The second-order filter derives some of its speed by only reading 5 pixels per output pixel, as the corners of the mask are not used. If all nine pixels were read, using the same state machine, 29 cycles would be required and so performance would decrease, though the higher clock speed would alleviate some of this disadvantage.

We can see from the analysis of the number of cycles required by the first convolution processor that a 3x3 mask would require $4(3)^2+1 = 37$ cycles per output pixel. This is less efficient than the software implementation, and even with the slightly shorter cycle time of 25ns compared with 33ns for the DSP, this system would take longer to process the image.

Design	Size (LEs)	Image size	Mask size	Time (ms)	Score
New HPF	995	256 x 256	9 x 9	511	0.13
New 2nd order	173	256 x 256	3 x 3	9.6	39.46
Bosi [3.6]	1924	1024 x 1024	3 x 3	42	12.98
Muthukumar [3.8]	958	256 x 256	3 x 3	1.31	52.22
Benkrid [3.9]	756	720 x 576	3 x 3	37	14.83
Zhang [3.11]	2606	1024 x 1024	14 x 14	17.5	22.99
Perri [3.12]	29048	1024 x 1024	3 x 3	4.6	7.85

Table 5: Comparison of various image processor designs

The two designs presented in this chapter are compared with various examples from the literature in Table 5. The table shows the processing times quoted in the literature, and the image sizes for which these are given. The score of each is taken by dividing the number of image pixels by the product of LE count and processing time. The scores are therefore scaled to take the image size into account. The LE counts from the literature are approximate, as all of these designs were implemented with Xilinx devices.

It is clear that the design in [3.8] has the highest score, due primarily to its very fast processing time. This design takes the shortest time of any of the designs in the table to process the image, though its image size is smaller than many of the others.

The new high-pass filter design achieves the lowest score, due to its large hardware size and relatively slow processing. If the divider was replaced with a smaller one, the score could increase quite significantly, even if the new divider requires several clock cycles to perform the division. It is possible that pipelining could be used to overcome any increase in processing time of this nature, by performing the division and equalisation for each pixel while the data for the next pixel is being read, then outputting the finished pixel when the corresponding mask pixel for the next target pixel is read, allowing the same address to be used.

The second-order filter achieves the second-highest score in the table, despite the apparent inefficiency of its algorithm, due mainly to the great reduction in hardware size compared with the shift-register based designs. This is important as the smaller hardware allows parallelism to be employed, with multiple copies of the processor operating in parallel. For a simple example, four of the second-order processors operating in parallel on a 1024 x 1024 image would take the same time

to process it as a single one takes for 256 x 256 (ignoring considerations of overlap at the edges of the sub-images for this simple example). With four times the LE count and 16 times the number of pixels processed in the same time, this system would obtain a score of 157.84 by the scoring system used in the table, significantly higher than any of the other systems. In reality it may be even higher than this, as the four processors could share a common address bus, fetching four pixels at a time through a 32-bit data bus. There would therefore only need to be one address processor, controlling four data processors, reducing the LE count still further. Theoretically, even without shared address processors, over 65 copies of the second-order filter could be implemented in parallel in the FPGA on the test system, though as this would require two 520-bit data buses it is unlikely that so many processors would be used.

It is clear from these results that while a shift-register design as described in [3.6] will be capable of performing the image processing more quickly, it will also require much more hardware. The shift-register holds M complete lines plus M pixels of the next line, for a mask of M x M pixels. For a 256-pixel wide image with a 3x3 mask, this would require 4120 bits of storage, or at least 4120 LEs. For a 9x9 mask on the same image size, a total of 18,504 LEs would be required for the shift register alone. The major advantage of the second-order design presented in this chapter then is its small size.

3.8: Further Uses of the Convolution Engine - A Cellular Automaton Processor

The Cellular automaton Processor described in this section is a development of the basic convolution image processor, in which the convolution operation is replaced with a CA rule. This was intended to explore possible other uses for a general-purpose convolution engine, rather than as a highly optimised CA processor.

3.8.1: Background

A cellular automaton (CA) can be thought of as an artificial universe, divided in space into 'cells' and in time into 'generations'. At the transition from one generation to the next, each cell adopts a new state based on both its own current state and those of the cells surrounding it. The automaton can theoretically have any number of spatial dimensions, and the cells can have any finite number of possible states, but all cells share not only the same set of states but also the same rules determining the transitions between these states.

The ability of the cellular automaton to produce very complex behaviours and patterns from a set of simple rules is well known and has been widely studied. Von Neumann showed that a CA with 29 states could be used to implement a universal computer [3.29], while Zuse postulated that the universe itself could be modelled as a large CA [3.30].

More recently, with the advent of fast computers which have allowed cellular automata to be studied in detail, the complex dynamics of the cellular automata have been studied in great depth, and have been found to be useful in a wide rang of applications, from biological modelling [3.31] [3.32] and neuromorphic processing [5.32] to video compression [3.33].

One of the most widely known cellular automata is Conway's Game of Life [3.34], a two-dimensional CA in which the cells can be either 'alive' or 'dead', changing state according to two simple rules: -
If a dead cell has exactly three live neighbours, it will become alive.

If a living cell has two or three living neighbours it will remain alive, otherwise it will die.

Patterns of great complexity can arise from these two simple rules. It has been demonstrated that a certain pattern of cells acts as a 'life cell', and a grid of these cells simulates the 'universe' in which they exist, the state of each cell being shown by the presence or absence of a particular pattern at particular times. [3.35] A complete Turing Machine [3.36] has also been demonstrated. [3.37] The wide range of behaviours which are possible with this single CA rule shows that even a very simple system can behave in very complex ways, and so there is much interest in these simple 2-state CAs, though the investigation of the Game of Life is often thought of as a recreational pursuit.

The complex and often chaotic dynamics exhibited by this and similar cellular automata were not discovered fully until it became possible to use a computer to simulate the system, and the complexity of the patterns which can be investigated grows as the simulation speed increases. There is therefore a requirement to make the processing of the CA as efficient as possible, and as with the image processing systems described earlier, hardware-based processors can offer high-speed processing beyond the capabilities of software implementations.

Halbach & Hoffmann showed that an FPGA implementation of a CA could achieve a speed increase over a software implementation, if properly optimised. [3.38] This implementation uses 16 parallel processors which operate on a 4 x 4 pixel window which is moved over the image. It was also shown in this work that a benefit of a hardware implementation over a software one is that the hardware implementation will perform its calculations at the same rate for a variety of rules of different complexity, whereas a more complex rule will result in a slower software implementation. A software implementation of a 256 x 256 CA was stated as achieving 455 generations per second, with the FPGA implementation achieving at best a speed increase of 13.8 times, or 6279 generations per second, though it is not clear whether this reflects the sustained speed over the whole 'universe' or merely over the 4 x 4 window.

Shackleford et al. present an implementation of a small CA for random number generation in which there are just 64 cells. [3.39] These are connected toroidally,

and all cells are independent and implemented in parallel. This results in a very time-efficient implementation, where a single clock cycles is all that is required to compute a generation. The maximum clock rate achieved was 230MHz, or 230 million generations per second.

Kobori et al. present a method of implementing the CA [3.40] which makes heavy use of distributed RAM within the FPGA to store a subset of the overall universe, implementing a moving window inside which computations are carried out very quickly by parallel processors. This implementation also makes use of a wide memory bus to fetch and store 8 cells simultaneously. A speed increase of 250 times over a software implementation is quoted.

3.8.2: Implementation of the CA processor

During development of the convolution engine, it was realised that a version using a 3x3 mask could be modified with little difficulty to perform the processing required by a two dimensional cellular automaton. Since each outputted pixel has a value determined mathematically from the values of the equivalent pixel in the source image, and its eight immediate neighbours, a general-purpose cellular automaton processor could be made by simply replacing the mathematical function with a rule-based system. Much of the system remains unchanged, with the address processor and control state machine being re-used almost unchanged from the convolution engine. The major change is to the data path, which is modified such that the target pixel and its eight neighbours are formed into a 9-bit binary word, which is then presented to a look-up table, built in a manner similar to the gamma-correction table of the convolution engine. The table contains 512 bits, determining the value of the output pixel for each of the 2^9 possible combinations of input pixels. Thus, by changing the table, the cellular automaton rule is changed. The tables are generated from a rule description by a QBasic program, as shown in appendix A.2.

The 'next generation' value of each pixel is derived from the current value by

$$N(x,y) = F(P(x,y))$$

where F represents the look-up table and C is a bit-vector determined by

$$P(x,y) = 2^8 C(x+1,y+1) + 2^7 C(x,y+1) + 2^6 C(x-1,y+1) + 2^5 C(x+1,y)$$
$$+ 2^4 C(x,y) + 2^3 C(x-1,y) + 2^2 C(x+1,y-1) + 2^1 C(x,y-1) + 2^0 C(x-1,y-1)$$

with C being the current state of a pixel.

The particular mapping of co-ordinates to bit positions within the word is arbitrary, but the above order is the one which was used in the implementation.

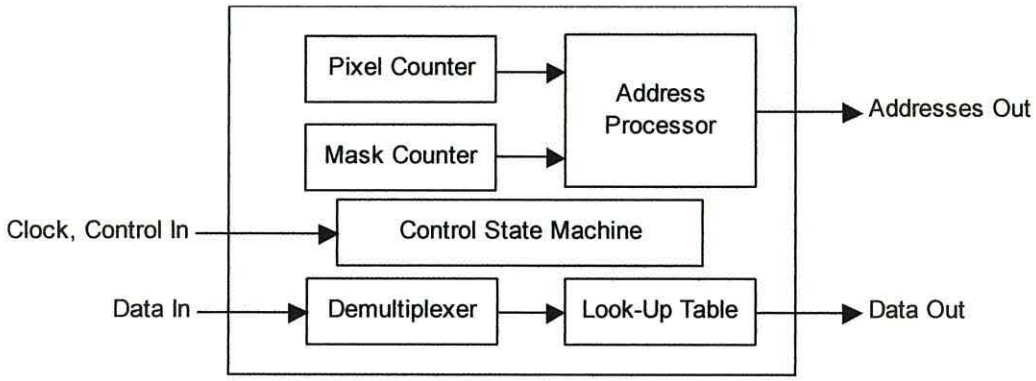

Figure 29: Cellular Automaton Processor

The FPGA implementation of this system is shown in Figure 29. The address generation logic and control state machine are nearly identical to those used in the second-order filter based image processor. The major difference is that all cells are processed rather than just the area where the mask is valid. At the edges of the image the mask addresses wrap around to the corresponding pixels on the other edge, thus the 'universe' has a finite size but no edges, and is toroidal. This requires a slight change to the address processor, as it must be ensured that when wrapping at the end of the line, the carry generated in the rollover of the X co-ordinate does not affect the Y co-ordinate. Thus, the two co-ordinates are processed separately.

Each pixel is processed in 33 clock cycles. In order to reuse much of the test-bed logic from the image processor, the image size was set at 256 x 256 pixels, though each RAM byte could only take on the values 0 or 255, and only one of the eight bits was read.

The demultiplexer is a 9-bit addressable latch, which captures the nine pixels read by the state machine and assembles them into a 9-bit word, which is fed to the look-up table to determine the cell's next state. Two memory banks are used, as in the image processor, ensuring that the assembly of the 'next' generation does not interfere with the stored 'current' generation. It would be possible to use additional multiplexers so that when a generation is completed, the data runs through the machine the other way for the next generation, i.e. from what was the

destination memory bank to what was the source bank, thus avoiding the need to copy the new generation over the old one.

3.8.3: Cellular Automaton Test results

The cellular automaton processor was tested with a few sample images, including skin line images outputted by the image processor. A few CA rules sets were used, these being Conway's Life, Simulated Annealing, Majority Rule and Hourglass. Conway's Life is detailed above, and is one of the most widely known and implemented cellular automata, and so provides a good basis for checking that the processor behaves as required.

Simulated Annealing and Majority Rule are quite similar. In the case of Majority Rule, a cell takes on whichever state the majority of its neighbourhood have. The neighbourhood includes the cell itself, so there are nine in the group. If the sum of these cells is 5 or more, the cell will be on, otherwise it will be off.

Simulated annealing works in a similar way, but the cell will be on if the sum of the active cells in the neighbourhood is exactly 4 or greater than 5. Table 6 shows the difference between the rules in terms of the activation function for the cell based on the number of active cells within its neighbourhood.

Active cells in neighbourhood	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Next state (Majority)	0	0	0	0	0	1	1	1	1	1
Next state (Sim. Annealing)	0	0	0	0	1	0	1	1	1	1
Next state (Life)	0	0	1	1	0	0	0	0	0	0

Table 6: Comparison of cellular automaton rules

One thing that links the three rules shown in Table 6 is that they are based purely on the number of active cells, not on any consideration of which particular cells are active. By contrast the hourglass rule is based on specific patterns of activity and inactivity, and considers only five cells – the target cell and its north, south, east and west neighbours. Figure 30 shows the nine combinations of these that will lead to a cell being active in the next generation. In each case, a shaded cell represents an active one, and the centre cell is the target.

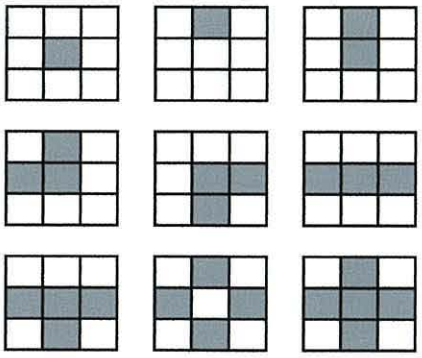

Figure 30: Combinations leading to an active cell in the Hourglass CA.

The life, majority and simulated annealing rules were tested using result images from tests run on the image processor which had been converted to black and white. These provided a good test, with bands of active and inactive cells, though any source image would have been sufficient.

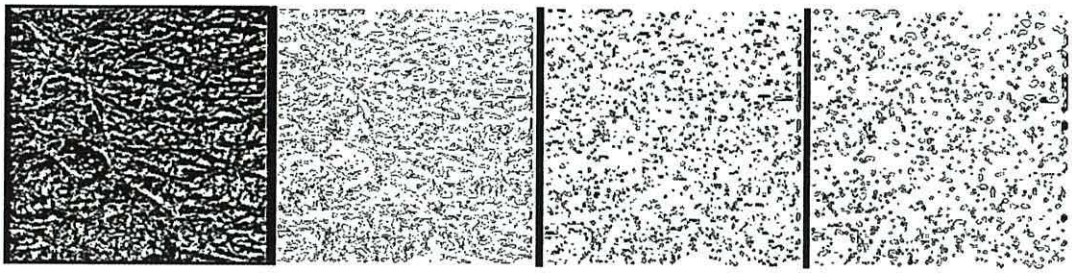

Figure 31: Generations 0-3 of a life rule test

The first four generations of the Conway's Life rule test are shown in Figure 31, where generation 0 is the source image. Although in the generated images active cells were white, in the figure they are black in order to display more clearly. The life rule was tested for 1000 generations, and the 'chaos' generated was compared with that produced by existing implementations [3.41]. The same patterns were found in both, with the well-known and common stable, oscillating and moving patterns being produced.

The majority rule was found to reach a stable pattern very quickly, as shown in Figure 32, where the right-hand image shows generation 10, beyond which there was no change in the pattern of live and dead cells. The finer features tended to either disappear or merge together to form larger areas of either active or inactive cells, and this would invariably happen within the first few generations.

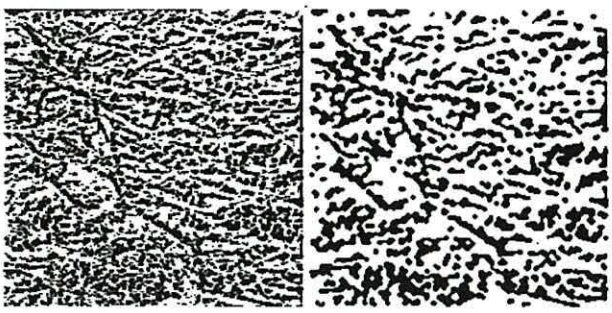

Figure 32: Generations 0 and 10 of the majority rule test

Simulated annealing was found to merge the groups of cells, but the function's changed response to the 4-active and 5-active cases tended to produce instability in the edges of these groups, which meant that the groups would generally not crystallise as with the majority rule and would instead gradually shrink and become rounder, with narrow protrusions and small groups disappearing completely.

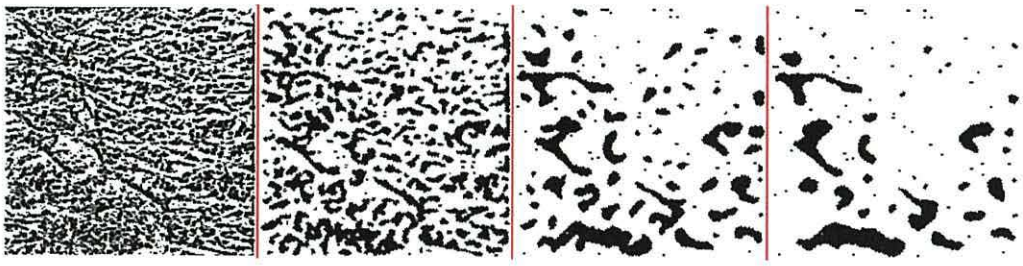

Figure 33: Generations 0, 10, 50 and 100 of the simulated annealing test

The hourglass rule was tested with a different initial pattern – a continuous sheet of active cells with a small group of inactive ones in the middle to act as a 'seed'. The directional nature of the rule was evident in the patterns produced, an example of which is shown in Figure 34. This image is inverted, as with the life results, so that active cells are black. The complex branching pattern was not static – the section which appears to be stretched vertically was in fact made up of groups of cells moving downwards, building up on top of the compacted branches below, and gradually pulling down the branches above. The wrapping of the universe at the edges of the image is also evident.

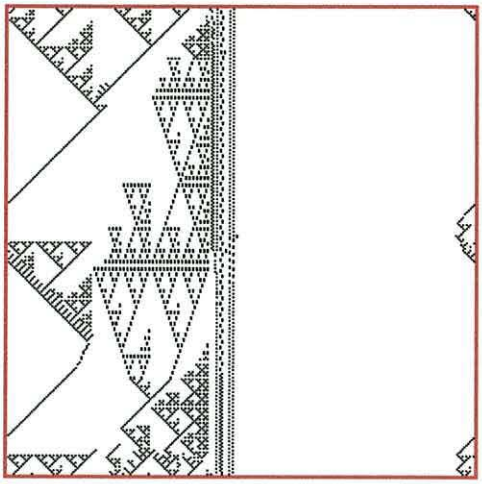

Figure 34: 500th generation of the hourglass rule

The test results showed that the CA processor is capable of correctly implementing cellular automata, producing the correct result for a set of widely known automata. The four different types of automata demonstrated showed that the look-up table based system is capable of implementing automata in which the cell state is based on either the number of surrounding 'live' cells or the positions of these. In fact, the look-up table system can implement any CA in which the cell's state is based on those of its eight neighbours, and in which there are only two states. If more than two states are required, the rule table and the input demultiplexer could be adapted easily, though for two bits per cell (4 states) the nine inputted cells would take up 18 bits, requiring a table of 262,144 x 2-bit words, which is not feasible in any of the Apex devices, though is possible with later FPGA families.

3.8.4: Performance of the CA processor

To process a cell, the system must read the nine cells under the mask, allow some time for the delay in the look-up table, then write the result. The implemented system completed these operations in 33 clock cycles using the state machine from the convolution image processor, and so required 2,162,688 clock cycles to process all 65,536 cells. At 25MHz in the test-bed, this takes 86.5ms, or a rate of 11.5 generations per second. The processor was not, however, intended to be a fast solution, it was merely intended to demonstrate the flexibility of the basic

image processor design. The maximum speed estimated by Quartus was 131.63MHz, at which speed it would take 16.4ms to process a generation, or 60.9 generations per second.

Comparing this with the implementations discussed in section [3.8.1] we see that Halbach's implementation [3.38] is capable of 6279 generations per second, though this may be only for the local window of 4x4 cells. The implementation by Shackleford et al. [3.39] is capable of 230 million generations per second, for 64 cells. This is however a fully parallel implementation and so the 65536 cells of a 256x256 universe would require an enormous amount of hardware.

By comparison, the Life32 CA engine [3.41] running on a 2.25GHz PC takes around 280μs to process each generation of a 256 x 256 toroidal universe using the Life rule. To obtain even this performance with the simplified hardware implementation presented in this chapter would require a clock rate of 7.7GHz.

It is clear then that the fastest way to implement a cellular automaton would be to implement all of the cells together in parallel, though it is also apparent that this would be the least efficient method in terms of the size of the logic required. A cellular automaton cell designed for this operation consists of a 9-input look-up table as described above, feeding the data input of a flip-flop which is clocked to yield the next generation. This is shown in Figure 35. All cells within the CA are clocked simultaneously, and the set and clear inputs on the flip-flop are used to set up an initial pattern.

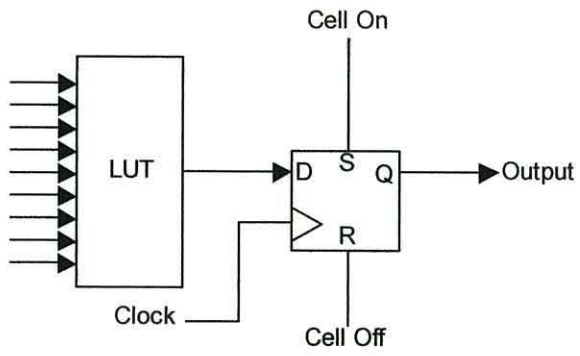

Figure 35: A single cell for a hardware cellular automaton.

The arrangement of a look-up table feeding a single-bit register is similar to the internal arrangement of a logic element within an FPGA, the difference with the CA processor being the connections between the cells. Rather than using a global switching network, the look-up table inputs would be hard-wired to the outputs of the surrounding cells. For FPGA implementation, this is relatively inefficient, as a nine input look-up table would either have to be implemented using the embedded system RAM blocks, or with a large number of logic cells. Even for the simplest case where the system is built specifically for a single cellular automaton and programmability is not required, the look-up table would require at least three logic cells to cater for its nine inputs. The implementation in [3.39] used only inputs from four neighbouring cells, and since a logic element within the FPGA is a 4-input LUT feeding a single-bit register, each CA cell maps directly onto one logic element.

The fully parallel design does have the advantage of being able to operate much faster than the serial implementation in this chapter, as even if it were clocked at the same rate, it would only require one clock cycle per generation, and so would be able to produce 2,162,688 generations in the time taken to produce a single one with the serial implementation.

In addition to this type of implementation, a line-at-once implementation such as the shift-register approach described in [3.6] could offer a performance increase without such a major increase in size. Care would have to be taken however to ensure that the edges of the grid of cells are handled correctly, as with the 'target' cell at the edge of the image, the cells within the shift register which represent the other edge of the image would represent pixels which are vertically misaligned. This is not a problem for the image processor as the pixels in places where the mask is not completely within the image are not processed, and the edges do not wrap, but in the case of a CA the wrapping of the edges to form a toroidal universe would have to be considered. If a shift register is to be used, it could take the form of a register which stores the three most recently read lines of the image, and into which a complete new line is read before processing begins. This would require more register cells than the implementations in [3.6] but this seems unavoidable if the edges are to be processed correctly.

3.9: Conclusions

3.9.1: Image Processor

The image processing system was originally designed as part of an experiment into replacing the software-based experimental processing system with hardware, to obtain a performance boost. The image processing systems presented in this chapter represent only the initial steps towards implementing a high-performance image processor, and the project was discontinued before any further progress was made.

The overall outcome of this project is that two working image processing systems have been produced and demonstrated, showing a major difference in their performance. Both have been seen to be capable of extracting the skin lines from the source images, with differing levels of detail.

Although the chosen implementations have been shown to be relatively inefficient in processing time, taking longer than alternative implementations to process the image due to the large number of memory reads which must be performed, it has been shown that they have the advantage of using much less hardware to perform the processing. It was shown that the shift-register based designs used for comparison would require a great deal more hardware if built for the same sized image as the systems presented in this chapter. Thus, these new designs trade off performance for a large reduction in hardware size.

The area-time product comparison gives a better score for the second-order filter than for similar shift-register based designs, despite the slower processing, because of the smaller LE count. It has been shown that a shift-register based design with a 9 x 9 mask, as is the case with the high-pass filter, would require many tens of thousands of LEs, and so it is likely that a system built in this way would achieve a lower area-time score than the new high-pass filter design, even if it could process the image more quickly.

Ultimately, the advantage of the implementations described in this chapter is their relatively small hardware size compared with more time-efficient

implementations. This smaller size enables the use of parallel processors to make up for the slower processing speed.

3.9.2: Cellular Automaton Processor

The cellular automaton processor demonstrates the flexibility of the basic convolution image processor, showing that a small change in the hardware can produce a quite different system. Like the image processor, the CA processor's performance suffers due to the inefficient method used for the implementation. However, like the image processor it can be improved by using one of the more efficient implementations seen in section 3.1, making use of the parallelism which is possible with a hardware implementation. It has been shown that the hardware of the CA processor has some similarities with the underlying fabric of the FPGA itself, and that an implementation based on full parallelism of the processing elements would be capable of very fast processing. Finally, it has been shown that a simple look-up-table based design such as this is a versatile system and can implement any CA in which the cells have two states with no change in architecture.

The ability of the cellular automaton, a regular grid of identical processing elements, to produce chaotic output has been shown in the case of the Life automaton, and will be seen again in a smaller form in the grid-like neural network of section 5.17.

Chapter 4: A Simple VHDL Microprocessor

The processor described in this chapter was born out of the necessity to implement functions within the FPGA which were sufficiently complex that it would have been too time-consuming or awkward to implement them in pure hardware. The original aim was to develop a simple processor which would be capable enough to perform a range of useful tasks but small enough that it could be used alongside other hardware, even in a relatively small FPGA. It was originally intended to oversee the transfer of data between the host PC and the convolution image processor, but it has since been used in a variety of applications. Its relatively small footprint and straightforward architecture make it suitable for any application where embedded software is required, and like any embedded processor, it can be augmented with external custom DSP hardware if higher performance is required.

4.1: Existing Designs

There are many existing embedded processor designs available, ranging in complexity from simple microcontrollers to full RISC processors for which operating systems are available. One obvious choice for implementation in the Altera FPGA is Altera's own Nios Embedded Processor [4.1], which can be implemented using a plug-in for the Quartus Software. Nios is a pipelined RISC implementation which can be built in 16 or 32 bit versions. Its architecture and instruction set are intended to allow efficient compilation of the control structures in high-level languages, a feature which many of the VHDL RISC implementations have in common. It can however be programmed in assembly language if necessary.

Nios is available only as an FPGA implementation built from the reconfigurable logic, and Altera do not currently produce any FPGAs with hard-wired processor cores built-in. Some types of FPGA, such as the Xilinx Virtex-II Pro and Virtex 4 types, include embedded processor cores [4.2], which are fabricated as part of the chip rather than implemented with VHDL. The Virtex 4 is available with up to two IBM power PC 405 cores, capable of 450MHz operation and delivering 700 DMIPS (Dhrystone MIPS). The Nios II high performance variant, by comparison,

is stated by Altera as being approximately equal in performance to an ARM9T core [4.3] and can deliver over 250 DMIPs. The 'economy' variant is in the same cost class as an 8051 core, delivering up to 30 DMIPs at up to 200MHz in fewer than 700 logic elements. The Nios core will generally be slower than the Virtex's PPC core, as the latter is hard-wired and is not subject to the speed limitations of the interconnections between the logic elements. Both types of processor can be extended with custom co-processors such as FPUs or specialist DSPs, or with extra instructions. In the case of the Virtex implementation, this extra hardware is connected directly to the processor's pipelines, but implemented with the FPGA's reconfigurable hardware, while in the case of Nios, it is possible to modify the core itself.

These approaches represent the top-end processors, intended for high performance in very large embedded systems, and are supported by software suites available from the FPGA manufacturers. For smaller and simpler applications, Xilinx also provide a soft-core microcontroller with a very small footprint, PicoBlaze [4.4]. This is a simple 8-bit microcontroller architecture with internal 1K program memory, 64 byte scratchpad RAM, stack and 16 registers. The core takes 2 clock cycles per instruction, and will run at 200MHz (100MIPs) in a suitable Xilinx FPGA.

The application in which the processor is to be used plays an important part in the choice of processor, as in many cases existing code must be re-used, and therefore an implementation of a standard processor core is required. Sometimes a standard operating system is required, in which case a processor architecture must be used for which there is an implementation of the particular OS.

Implementations of existing commercially available processors are provided by a variety of companies, such as CAST inc. [4.5] and HiTech Global [4.6]. These implementations, along with non-commercial implementations, cover a range of processor types, from the popular microcontrollers such as the PIC series or the 8051 to 'classic' microprocessors such as the 6500 [4.7] and 6800 series and the Z80 [4.8]. More powerful or advanced devices such as the 68000 [4.9] and the Intel Itanium [4.10] have also been developed, both commercially and as non-commercial or academic projects.

These 'real-world' processors are often implemented in an attempt to emulate legacy systems [4.8] [4.11], and if this is the case are generally built with no speed optimisations beyond a higher overall clock speed. Processors that are built for code compatibility only can be optimised with pipelining and more efficient design to provide more instructions per second for a given clock speed than the original commercial implementations.

Custom processors which are not based on 'classic' designs are also available in both commercial and free forms [4.12] with a wide range of features and performance ratings. Many of these began life as experiments or as practice projects when learning VHDL [4.13], or are intended as educational examples [4.14]. The Edulent processor described by Mezei and Malbasa [4.15] is an example of a simple educational processor, with a simple accumulator architecture and 40 instruction types, capable of running at 12.5MHz. Romero-Troncoso et al. present a more complex educational processor [4.16], with a larger register set, more complex architecture and microcode, allowing the instruction set to be customised easily.

Gustin and Bulic describe a novel architecture [4.17], again intended for educational purposes, in which every possible instruction is a MOVE instruction and the operand is the output of the ALU. It is shown that this allows the same ALU to be used for address calculation and instruction execution, saving hardware.

Paul Stoffregen's OSU8 [4.18] is a relatively simple 8-bit microprocessor with two accumulators, two pointer registers, microcode, and an instruction set similar to that found in the 6502 or its contemporaries. This design has two ALUs, one for instruction execution and a second 16-bit one for address computations. The design is a little more complex than many of the simple architectures such as Edulent, as described above, and the processor was designed to be implemented as a standalone processor in an FPGA rather than an embedded processor as part of a larger system.

An alternative class of custom processors is those which are designed to support particular compilers or high-level languages, such as JOP [4.19], a hardware implementation of the Java Virtual Machine. This is a RISC stack machine which

is essentially a direct hardware implementation of the JVM, with a few optimisations, and is intended for use in embedded and real-time systems, achieving a speed increase of 250 times over the performance of compiled Java on an embedded microcontroller. Around 1800 LEs are required and the processor is capable of running at 101MHz.

Mattos and Carro [4.20] present an alternative Java processor called Femtojava, in which the instruction set, though not extensible, can be pruned to exclude those instructions which are not used by a given program, making the hardware as small as possible for a given task.

Forth-based processor cores have also been implemented, with features that make efficient translation of Forth programs easy. Haskell and Hanna [4.21] present a Forth core which runs Forth code which has been converted to fit the processor's instruction set, which is tailored to suit the requirements of the language. This design, using 734 slices of a Xilinx Spartan II FPGA, is intended to be used where microcontrollers would traditionally be used, and is shown to be nearly 30 times faster than the equivalent compiled Forth programs running on a 68HC12 microcontroller. An alternative implementation is presented by Frank Buss [4.22] using simpler hardware and requiring 432 LEs of an Altera Cyclone device.

4.2: Architecture

The processor was originally based on an experimental reworking of the Manchester Small-Scale Experimental Machine, [4.23] (1948), a machine which was intended to be the simplest possible computer which could perform general-purpose applications. This was an extremely simple machine, as the high cost of hardware at the time, not just in monetary terms but also in terms of size and power consumption, meant that the design had to be kept as simple as possible. As a result the machine had just seven instructions, but it was shown that these were sufficient for any computation, given sufficient time and memory.

The VHDL processor was intended to be the simplest implementation of a general-purpose processor, but with a range of functions similar to modern processors. The overall aim was to produce a processor which could be used for a variety of applications but which used little enough hardware that it wouldn't have

a major impact on the space left in the chip for other devices. Although the range of functions is comparable with some of the simpler modern processors, the architecture and some of the operating methods are more similar to the SSEM. The instruction set was inspired by the simpler 8-bit microprocessors of the 70s and 80s, in particular the 6502, which has a similar set of instructions and just a single accumulator rather than a set of several general-purpose registers. As the processor was intended to be used for transferring data it was decided that an index or pointer register was essential, allowing indexed addressing modes. Rather than using two 8-bit index registers as in the 6502 it was decided to use a single 16-bit pointer register, the implementation of which was made simpler by the 16-bit architecture.

The processor uses a 16-bit data word, and requires two of these to form an instruction, the first being the instruction code itself, the second being the operand. In order to simplify the hardware, there is no instruction decoder or microcode, and the two words are read for each operation, regardless of whether an operand is required. With no instruction decoder, each bit in the op-code directly controls some part of the processor, and thus there is no explicitly hardware-defined instruction set. Only the ALU has a decoder, decoding the four least significant bits of the op-code in order to select a function. It was this instruction coding method which dictated the width of the processor's registers and data bus, as the words had to be wide enough to include all the required control signals in a single op-code.

There are two working registers which can be accessed by software, named A (accumulator) and P (pointer). All functions available in the ALU can be applied to either A or P, and in addition, the contents of P can replace the address used to fetch or store the operand, allowing for easy implementation of arrays. This means that any operation which is performed on data can also be performed on addresses. In this respect the architecture differs from many earlier processors, as in general registers which could be used for memory addressing were not also available as general-purpose data registers. Although it would be useful to have more than one accumulator and more than one pointer register, the method by which the instructions operate makes this impractical, as many more bits would be

required in the op-code, not just extra register enable bits but also extra multiplexer control bits.

The structure of the processor is shown in Figure 36, with the names of the VHDL blocks shown. The data bus is shown divided into an output bus and an input bus, though the control unit does generate the necessary bus control signals to allow these to form a single three-state bi-directional bus, as there is no point in the processor's operation where data needs to flow through both buses simultaneously.

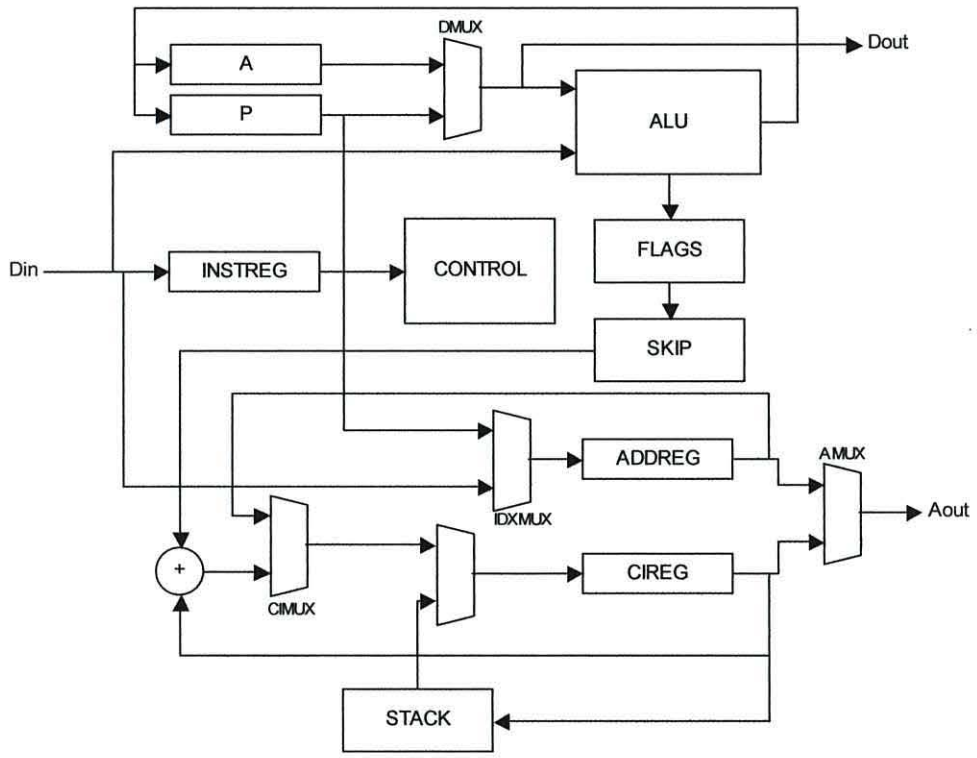

Figure 36: Simplified overview of the VHDL microprocessor

4.2.1: ALU and Registers

The ALU and registers form the data processing section. One input to the ALU always comes from memory, while the other can be taken from either A or P. The data output from the processor also comes from this multiplexer, allowing either of the registers to be outputted. This means that the result of an operation can only be written to a register and not directly to memory. If the latter is required the data must be written to the register first, then the register written to memory with a second instruction. In practice this is not a major limitation and has not been found to cause problems. This architecture was based on the architecture of the SSEM, though this earlier machine had just a single subtractor rather than a full ALU, and only one accumulator, though many more modern processors such as the 6502 have this arrangement, and the load-store architecture, where instructions either load data, store data or perform an operation on a register, is one of the characteristics of a RISC processor.

The ALU's operation is selected by the lower four bits of the op-code. The operations available are shown in Table 7. The operations included in the ALU were chosen to give the processor a useful range of abilities, though in order to keep the ALU as small as possible multiplication and division were not included. If these are required they must be implemented in software, though as the processor is intended to be included in an FPGA, it is possible to include external multiplication or division logic. This logic would have to take the form of a peripheral device, where the operands are written to registers and the result read after the required operation is performed.

The addition can be performed either with or without carry-in, if carry-in is selected the stored C flag is added to the LSB of the adder, allowing numbers wider than 16 bits to be added.

In addition to these, Boolean logic, shift and rotate operations are included, with an option to perform an extended rotate, where the C (carry) flag is included in the operation as the 17th bit. The last implemented code simply feeds the memory input through to the output, allowing the memory contents to be read into a register.

Code	ALU Output
0000	register + memory
0001	register - memory
0010	register + memory + cflag
0011	register AND memory
0100	register OR memory
0101	NOT register
0110	register XOR memory
0111	register shifted left by 1 bit
1000	register shifted right by 1 bit
1001	register rotated left by 1 bit
1010	register rotated right by 1 bit
1011	register rotated left by 1 bit with extend
1100	register rotated right by 1 bit with extend
1101	memory
1110	Reserved
1111	Reserved

Table 7: ALU control codes

The final two codes are reserved for the subroutine handling instructions JSR (jump to subroutine) and RTS (return from subroutine).

Op-code bits 5 and 4 activate the P and A registers respectively. If either bit is 1, the corresponding register will be updated with the ALU's output when the instruction completes. Both can be used simultaneously if required. If bit 8 is set to 1, the flags register will also be updated with flags derived from the ALU operation. The N flag is set when the top bit of the accumulator is 1, representing a negative number in 2's complement notation. The Z flag is set when the ALU outputs zero, and the C flag is set when a carry is generated, or a 1 is shifted out of either end of the ALU during a rotate-with-extend operation.

4.2.2: Address Processing

The CI (Current Instruction) register supplies the address of the instruction or operand currently being fetched. An address register is used to latch the address for any instructions which require access to memory. The input to this register can be taken either from the data input, if the address comes from the instruction, or from the P register if the instruction uses indexed addressing. The CI register is updated through a pair of multiplexers, and can either be incremented, loaded directly from the address register (for jumps) or controlled by the stack system. If

op-code bit 6 is set to 1 the CI will be updated during the execution phase of the instruction.

The address bus outputted by the processor can be supplied by either the CI or address registers.

4.2.3: Jump and Skip Instructions

There is only one jump instruction available, an absolute jump which is effected by loading the CI register from the address register, having first loaded the address register with the destination address. Op-code bit 6 must be set to 1 to perform a jump. Since it is possible to load the address register with the contents of P rather than the jump operand, a kind of indirect jump can be performed, but this has not been tested and there is currently no support for it in the assembler.

Conditional branching is achieved with a series of conditional skip instructions, which cause the processor to skip the following instruction when the selected condition is met. If op-code bit 7 is set, and the condition specified by bits 0 to 2 is met, the skip logic will be armed during the execution phase and the CI will be incremented by 3 rather than 1 at the start of the next instruction. If the condition is not met, the skip logic will not arm. Table 8 shows the condition codes for the different skip instructions.

Code	Skip Condition
000	N flag set
001	N flag clear
010	C flag set
011	C flag clear
100	Z flag set
101	Z flag clear
110	Never
111	Never

Table 8: Conditional skip control codes

To implement a conditional branch, skip and jump instructions are used together.

Due to the CI being incremented before a fetch rather than after, as explained below, the target specified in the jump instruction should be the address before the desired instruction. This is handled automatically by the assembler, and only needs to be taken into account when hand-assembling programs.

4.2.4: Subroutine Handling

The logic required for subroutine handling was added after the processor was completed, in order to increase its usefulness. The stack logic monitors the incoming op-codes and takes over when either JSR or RTS are detected. The JSR op-code is a jump op-code with the lower four bits set to 1110, while RTS is a jump with the lower bits set to 1111. The stack memory itself is a separate block implemented inside the stack logic, as this was simpler than adapting the processor's logic to keep the stack in system memory.

When a JSR instruction is encountered, the stack logic reads the current value of the CI and stores it in its internal memory. When JSR is encountered, the stack logic replaces the CI register's input with its own output. In both cases, the rest of the processor is allowed to perform the actual jump instruction.

4.2.5: Control

The processor has an operating cycle consisting of six states, or phases.

Phase 1:

The first phase is an idle phase, included to provide an inactive state in which the processor can safely halt.

Phase 2:

The CI is incremented by either 1 or 3, depending on whether the skip logic was successfully armed during the previous instruction.

Phase 3: The contents of the memory address pointed to by CI are loaded into the instruction register. The bits within this op-code then set up the multiplexers and registers for the selected instruction. The skip logic is reset here, so that the next CI increment will be an increment of 1.

Phase 4: The CI is incremented again, to point to the instruction's second word.

Phase 5: The address register is enabled, and the source selected by op-code bit 15 is written to it. The source can be either the data input or the P register.

Phase 6: The instruction is executed. Register enable signals are sent to the registers selected by the op-code, and if the instruction is set to be an output instruction (op-code bit 14 = 1) the memory write line is pulsed.

4.2.6: Op-Code Layout

As described above, the processor has no instruction decoder, so each bit in the op-code controls a piece of hardware directly. Table 9 shows the functions of the bits in mnemonic form.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function	IDX	I/O	AMUX	DMUX	CMUX	STP	MEM	FLAG	SKIP	CI	PTR	ACC	Op3	Op2	Op1	Op0

Table 9: Op-code bit layout

IDX	Function
0	Address register loaded from data input
1	Address register loaded from P

DMUX	Function
0	Data output supplied by A
1	Data output supplied by P

AMUX	Function
0	Address supplied by address register
1	Address supplied by CI

CMUX	Function
0	CI incremented by adder
1	CI loaded from address register

Table 10: Multiplexer functions

The FLAG, CI, PTR and ACC bits select which registers will be updated during phase 6 of the instruction. The MEM bit does the same for memory.

IDX, AMUX, DMUX and CMUX control the four multiplexers labelled in the block diagram. Table 10 shows the functions performed by these multiplexers.

I/O is used to tell the control unit whether the instruction is an input or output instruction. If set to 1, the instruction is defined as output and the control unit will generate the necessary signals to control the tri-state buffers required for a bi-directional data bus.

STP (stop) will halt the processor at the end of the instruction. Although there is currently no support in the control unit for resuming from a halt without resetting the processor completely, the changes required to accommodate this would be trivial.

SKIP activates the skip unit as described above.

Op0 – Op3 form the ALU operation code for an arithmetic or logic instruction, or the skip type for a conditional skip instruction.

As an example, the instruction with the assembler mnemonic LDA (p) causes the contents of the memory location pointed to by the P register to be loaded into the

A register. The op-code for this is $811D_{16}$, or 1000 0001 0001 1101 in binary. Referring to table n, we see that the IDX, FLAG and ACC bits are set, and the ALU code is 1101. This ALU code causes the ALU to feed the incoming data from memory directly to the register inputs. Since ACC is set, the accumulator will be updated during phase 6. FLAG is set, which will cause the flags to be updated according to the data which arrives from the memory. Only the Z (zero) flag will be affected by this, as the ALU will not generate a carry during this operation, and the N flag is generated based on the accumulator. If this is not required, the FLAG bit is cleared and the op-code becomes $810D_{16}$.
The active IDX bit causes the address register to be updated from P rather than memory during phase 5, so that during phase 6 the memory will be outputting the contents of the address stored in P. This will be written to A when the register enable signals are generated in phase 6.

4.3: The Assembler

The assembler was written in QBasic to allow software to be created quickly and easily. Since the processor was designed to perform relatively simple tasks, the early software was hand-assembled. It soon became apparent that an automated solution was required.
The instruction mnemonics file used by the assembler is shown in appendix B. This file contains the canonical list of supported instructions, with an entry for each addressing mode supported by the instructions, along with their hexadecimal op-codes.

There are a number of different addressing modes which can be used, though not all instructions support all modes. These modes are summarised below.
The simplest mode is the register-only mode. Here, a register provides the data to be operated upon, and also provides the destination for the result. The shift and rotate instructions, along with NOT, are examples of this type.
The most straightforward of the memory accessing modes is absolute addressing, where the operand field of the instruction provides the address on which the instruction operates. This can be written numerically or using an assembler label, e.g.:

`LDA 100` will load the contents of memory address 100 into A, while

`LDA mem` will load the accumulator with the contents of the address labelled by 'mem'.

It is also possible to use the operand itself as the source data, in which case a '#' is placed before the numeric operand. In the terminology of processors such as the 6502, this is immediate addressing. Only numeric operands can be accommodated in this mode, and these can be in decimal or hexadecimal, e.g.:

LDA #100 will load the value 100 (decimal) into A.

LDA #$100 will load the value 100_{16} into A.

In addition to this, there is a slight variation which is useful for creating a pointer to an array. Placing '@' in front of a label will cause the assembler to generate an instruction using the immediate addressing mode but to place the address of the label into the operand field as in the case of absolute addressing. The result of this is that the address of the label will be loaded into the register. This is most usually used with the LDP instruction.

The final addressing mode is indexed addressing, where the address of the operand is supplied by P rather than the instruction code. This is used to access arrays or look-up tables, and is represented thus:

`LDA (p)` will load the accumulator with the contents of the address pointed to by P.

It is possible to create hybrid instructions by combining the bit patterns of two ordinary instructions. For example, the normal absolute LDA instruction code is $011D_{16}$. If the PTR bit is set in this instruction, making it $013D_{16}$, the data will be loaded into both A and P simultaneously. These hybrid instructions are easily added to the assembler's op-code table whenever they become necessary.

Other assembler instructions are ORG, DC and DS. ORG is followed by a number, always in hexadecimal, which sets the assembler's program counter, telling it where to assemble the following code. DC (Define Constant) is followed by a number, series of numbers or ASCII string, or any mix of these, and is used to place values into memory. DS (Define Storage) causes the assembler's program counter to skip onward by one word, and is used to define labelled storage in

memory. The simple structure of the assembler requires that an operand is provided, but as this is ignored it can be any numeric value.

4.4: Performance

The processor requires six clock cycles to process an instruction, and the Quartus timing analyser suggests that the maximum speed of operation is 66MHz when implemented in an Apex 20K device. This corresponds to a performance of 11 million instructions per second (MIPS). In general, the tasks for which it was designed do not require much processor speed, and the fastest clock rate it was used at was 24MHz, or 4 MIPS. As it is fully static and synchronous it should run at any speed up to the maximum of 66MHz, and it should be possible to change the speed as it runs, provided that the clock gating logic does not introduce glitches. The six phase outputs are provided to allow the external circuitry to synchronise with the processor's operating cycle.

If the control state machine is suitably modified there is no reason why the processor should go through state 0 each cycle, it could go from state 5 to state 1 on all instruction cycles except where the instruction is a halt instruction, thus increasing the speed slightly to 13.2 MIPS at 66MHz. State 0 does however provide a period during which the processor is guaranteed to be not accessing memory, and this time could be used by external hardware to perform transparent DMA transfers or other memory accesses.

4.5: Analysis

The architecture of the processor is difficult to compare with other types of processor, because it is greatly simplified compared with most. One possible comparison is with the 6502, which has a similar programming model and a similar set of functions. Both devices have a single accumulator, though the 6502 has a pair of 8-bit index registers instead of a pointer register, which can be used in a base-offset addressing system. The 16-bit pointer register allows indexed addressing to be used across the whole memory map without requiring complex base-offset addressing, a useful feature when dealing with transferring data in large blocks. The connection between the P register and the ALU, which allows P

to be used as a second accumulator, makes it possible to perform pointer arithmetic much more simply than with the 6502, and also allows the use of two accumulators in sections of the program where pointers are not required. It is primarily because the processor's entire architecture is 16-bit that this is possible, if the data bus was 8-bit the P register would be 8-bit and could not hold a complete memory address.

It is difficult to compare the processor with other architectures because it does not bear close resemblance to any common processor. The relatively simple instruction set and small number of registers are somewhat similar to the 6500 series, while its reliance on conditional skip instructions instead of conditional jumps or branches, and its regular instruction timing are similar to the PIC series of microcontrollers. Although its instruction set is small and it has some features in common with RISC architectures, it is not a true RISC machine, in which it is common to see many more general purpose registers, pipelining and more complex addressing modes which can be used by compilers to make more efficient translation of high-level code. In fact, with the lack of an instruction decoder and the wide range of hybrid instructions this enables, the instruction set of this new design could potentially contain many hundreds of instructions.

It is unlikely that pipelining would be a useful addition to this design, as the addition of several pipeline stages might increase the speed slightly but would add significantly to the size of the processor. The simplicity of the execution cycle also makes pipelining more difficult, as it is hard to see how the cycle can be broken up into stages without making it less efficient.

A possible method of analysis of the design is to compare its performance and hardware size with that of other FPGA-based processors. This comparison will be based on the general-purpose processors implemented in VHDL / Verilog rather than hardware embedded units like the Virtex's PPC core or designs which are specifically intended to support a particular compiler or language, such as JOP, the Java Optimised Processor mentioned in section 4.1. The results detailed here are only a basic and approximate indication of the relative capabilities of the processors, as is discussed below. A complete and thorough analysis of the different types would require a common benchmark program to be implemented

and run on all types, which has not been done. These figures show only the performance in MIPS (Millions of instructions per second).

Type	LEs	MHz	MIPs (Est.)	MIPS / LE
New Design	440	66	13.2	0.0300
Nios 2 fast	1800	185	218	0.1211
Nios 2 std.	1400	165	127	0.0907
Nios 2 min.	700	200	31	0.0443
D68000	6332	32	3.2	0.0005
DRPIC1655X	919	59	14.75	0.0161
68HC11	1809	42	42	0.0232
DP 8051	1750	63	63	0.0360
Free6502	1064	12.5	4.3	0.0040

Table 11 : Comparison of various FPGA processor implementations

Table 11 shows the size and speeds of FPGA implementations of various processors. Apart from the Nios designs, the Free6502 design [4.7] and this new design the others are commercial IP blocks available from Hitech Global [4.6]. The Free6502 core is not a commercial design, and is quoted as using 523 CLBs in a Xilinx 4020XL device. The difference in architecture between the Xilinx and Altera devices makes a direct comparison more difficult, but the datasheets show that functionally each CLB in the Xilinx 4020 device is approximately equivalent to two of the Apex LEs, so the LE count shown in the table is based on this. It is clear that this new design has a smaller footprint than any of the others, and while it may not have the highest clock speed, its maximum speed of 66MHz is faster than several of the other designs in the table, with only Nios II running faster.

The MIPS figures are approximate, except in the case of the new design which is assumed to always require 5 cycles per instruction, and the PIC which is assumed to use 4 cycles per instruction. The figures for Nios II are obtained from the handbook for the processor [4.24]. The 68000 is known from past experience to take an average of 10 cycles per instruction, while the 68HC11 and 8051 are specified by their manufacturers as being speed-enhanced versions which offer a substantial increase in speed for a given clock speed when compared with the original devices. It is therefore assumed for the purposes of this comparison that they take a single cycle per instruction. The 6502 is known from past experience to perform around 0.7MIPS at 2MHz, thus the value in the table is calculated accordingly.

The final column in the table shows the speed in MIPS per logic element used, showing the area / performance trade-off. This shows that this new design ranks fifth by this measure, with Nios and the 8051 outperforming it. However, if the 8051 is not capable of single-cycle instructions as was assumed, its score would decrease. It is clear that the Nios 2 design is substantially more powerful than any of the others in the table, though the 8051 is not far behind the least powerful of the Nios designs, in terms of MIPS per LE. The architectural differences between the three versions of Nios account for the wide variation in MIPS per MHz for these three.

A direct comparison of speeds, either in terms of clock speed or instructions per second, can be misleading unless the two processors have similar features. In some cases certain sections of program code can be implemented more efficiently on some processors than on others, if it makes use of some processor-dependant features. Similarly a program running on a processor with more internal registers, such as the 68000 in the table above, will need fewer data transfers between registers and memory than a similar program running on a processor with just a single accumulator. Capabilities such as hardware multiply and divide will also reduce the execution time of an algorithm compared with software implementations. Even simple functions such as shift and rotate can be accelerated if the processor has a barrel shifter and can perform several shifts in one cycle.

Another point of importance is that the instructions take a fixed length of time to complete in this new design, regardless of their mode, in contrast with more complex processors such as the 68000 series, in which the more complex addressing modes add significantly to the execution time of the instructions. However, the simplicity which provides this feature also reduces the number of addressing modes available, which can result in complex operations involving pointers taking longer, as base-offset address calculations have to be done with the ALU.

The ALU design is inefficient at present, as the add, add-with-carry and subtract operations are all performed by separate adder circuits. The ALU could be rebuilt to make more efficient use of the hardware, further reducing the logic cell count, though with a possible reduction in operating speed as combining the adders into one will add extra gates to the input of the adder and thus increase the propagation

delay. Each adder currently takes at least 16 cells to implement, along with the extra cells required to implement the larger multiplexer that is required when the adders have separate outputs. If the ALU size is reduced without lowering the operating speed of the processor, its performance/area ratio will increase.

The simplicity which allows the processor to take up relatively little hardware will also pose a problem if the processor is to be expanded with a larger instruction set. There are only four bits which could be used for the ALU function code, and at present only two permutations of these are unused, though these are detected by the stack logic and used to denote subroutine jumps. However, since the subroutine logic is only activated if the instruction is specified as a jump instruction, it is possible to use these 'spare' codes for two extra ALU operations. There are also two spare codes in the conditional skip instruction set, though there is no reason why the skip code cannot be expanded to fill the four bits allowed for the ALU codes, resulting in a total of ten more conditional skip instructions. It would be difficult, however, to find enough testable conditions to make use of this feature.

The stack can also be expanded, and in fact since the RAM blocks in the Apex FPGA hold 2048 bits each, the current 16 level stack, using just 256 of these, is not using the RAM block to its full potential. In theory, the stack could be extended to 128 levels before an extra RAM block is required. Later Altera FPGAs, which have smaller RAM blocks, would enable a less wasteful implementation.

The memory usage is also less efficient than most processors, but this was necessary in order to simplify the hardware as much as possible. Two 16-bit words make up each instruction, the second word being present whether it is required or not. However, there is no reason why the operand words of instructions which do not need operands shouldn't be used for data storage, even if this complicates the design of the software. There is no support for this at present but an optimising assembler could make use of the unused operand spaces to store constants for other instructions, or variables if the assembled code is to be placed in RAM.

A feature which can work in this design's favour is the ability to create hybrid instructions, and to customise the standard set, in both cases with no change to the hardware, but simply by altering the bit pattern of the op-codes. A simple example

which was mentioned previously is the ability to load both registers simultaneously from the same source, which would clearly save an instruction cycle or 5/6 clock cycles (depending on whether state 0 is used in each cycle). This LDAP instruction can take any form available to the LDA or LDP instructions. It is also possible to specify whether flags are updated or ignored, something which is not often available on other architectures, and if it is required that an LDA instruction does not modify the flags, this can be achieved without having to back up the flags in memory, a task complicated immensely by the fact that the flags are not accessible as a register. Certain more complex instruction sequences can be combined into one instruction, for example it is possible to create an op-code which will perform a register-only operation such as a shift or rotate on A, but with the result written into P rather than A, thus preserving the contents of A for future instructions without having to write it to memory. It is also possible to perform instructions of this nature without writing any result, but instead only updating the flags. Although it may be hard to see the necessity of these instructions, they are representative of the idea that a flexible architecture such as this can, under the right conditions, make certain software operations a lot simpler.

4.6: Conclusions and Further Work

At present the processor is a capable machine which is suitable for applications requiring functions which are too complex to implement with hardware. It is ideally intended to be used in conjunction with other more specialised processing hardware, with the processor itself overseeing the movement of data and the control of the other hardware. Some degree of refinement is desirable in the case of the ALU, which is currently somewhat larger than necessary, having been implemented in an inefficient way.

It has been seen that the processor's simple hardware brings two major advantages, firstly the small footprint, which is useful when implementing systems in smaller FPGAs, and secondly its ability to operate quickly, as having relatively little hardware in the data path to add to the propagation delays allows for a higher clock speed. The two of these combine to produce a device with a high ratio of performance to area. It may not be as high as some alternative

processors, but this new design has a small footprint compared with other designs, which can be made even smaller by a more efficient implementation of the ALU. The small instruction set and limited number of registers do however mean that some of the more complex operations take more instructions to perform than with more complex processors. However, the simplicity of the control hardware also introduces the idea of hybrid instructions, which have been shown to have potential in reducing the number of discrete operations required to perform a task. At present the only additional work that needs to be done on the processor is to make the ALU more efficient, as any modifications to the instruction set or addition of new features will be dictated by the applications for which the processor is used. One particular application area could be education, as the processor's very simple hardware could make it a good platform for learning the basics of microprocessor design.

Chapter 5: Digital Neuron Models

This chapter describes the development of a set of hardware blocks which replicate the function of neurons in a greatly simplified form. The motivation behind the design of these models is to develop a simplified model of a neuron which takes up as little space in the FPGA as possible while retaining the ability to perform complex functions. Several designs are presented, showing different approaches to the problem of modelling a neuron with simple hardware. The issues associated with efficient implementation of the models in the FPGA are shown and methods of optimising the designs to make the most efficient use of the hardware are discussed.

It is shown that the neuron models are capable of performing complex functions despite their simplicity, and that a network of such models is capable of very complex dynamics under certain conditions. Finally, some simple neural circuits, a spike multiplier and a set-reset latch are demonstrated, showing that the neurons can be used to construct novel and useful building blocks for a larger and more complex system.

5.1: Background & Review

Neural networks find applications in a variety of fields, but the most usual applications are statistical analysis, pattern recognition and classification of data. In particular, a neural network can be useful when the data set has no easily described features which differentiate one class from another, or in which the differentiating features are too subtle or variable for a conventional rule-based approach to work [5.1, 5.2]. The ability of the network to determine subtle patterns in the incoming data also makes it useful when the incoming data is corrupted or obscured by noise, especially when the noise is largely periodic, as the network can be trained to ignore this and focus purely on the embedded signal. [5.3]

The term 'neural network' implies a device that operates in a manner similar to an organic brain or central nervous system, systems which operate very differently from conventional computers.

The brain is a massively-parallel computer, a device in which computation is carried out by a large number of simple processors operating in parallel. Each of these processors, a neuron, performs a relatively simple function, collecting signals from other neurons and producing signals of its own when its inputs meet certain conditions. Each neuron may be connected to fewer than ten others, or more than a thousand, while the connections themselves can have a variety of effects on the state of the neuron. It is the cumulative effect of these simple operations which gives the brain its enormous ability to process and store information, and which gives a neural network the ability to perform operations which are difficult or impossible to program a conventional computer to perform.

The neural network obtains its high performance by 'steering' the incoming data rather than by performing algorithmic processing. Once the necessary learning has taken place, and the network is trained to perform a task, any incoming data simply activates the pathways between the neurons which were set up by the training, providing an answer very quickly.

Typically, networks designed for pattern classification tend to have two or three layers. [5.4] The first layer is the input layer, to which the incoming data is presented. This data may need to be encoded in some way to suit the type of neuron model used in the network, for instance if the network uses spiking models the input to each neuron would be encoded as a spike train, where the information is carried in the frequency or timing of the network.

The last layer in any network of this type is the output layer, and in the case of a pattern-classifier network, will generally consist of one neuron for each class to which the incoming data could belong. There may also be a hidden layer of neurons between the input and output layers, as depicted in Figure 37.

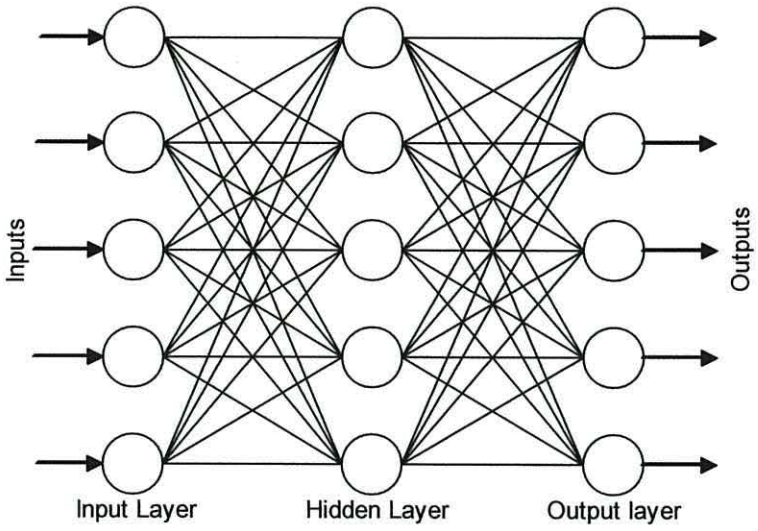

Figure 37: A three-layer neural network

5.2: Neuron Structure and Operation

The general form of a neuron [5.5] is shown in Figure 38. This is the type of neuron found in brain tissue, which differ slightly from some of the more specialised types such as sensory cells and motor neurons. The soma, or cell body, has a structure similar to most types of body cells, with the addition of a series of protrusions and a surface membrane consisting of a lipid bi-layer with unique electrochemical properties. The inputs to the cell are the dendrites, which are long branching structures, several of which may extend from a single cell. The cell's output channel is the single axon, which ends in one or more terminals which make contact with other neurons at sites called synapses.

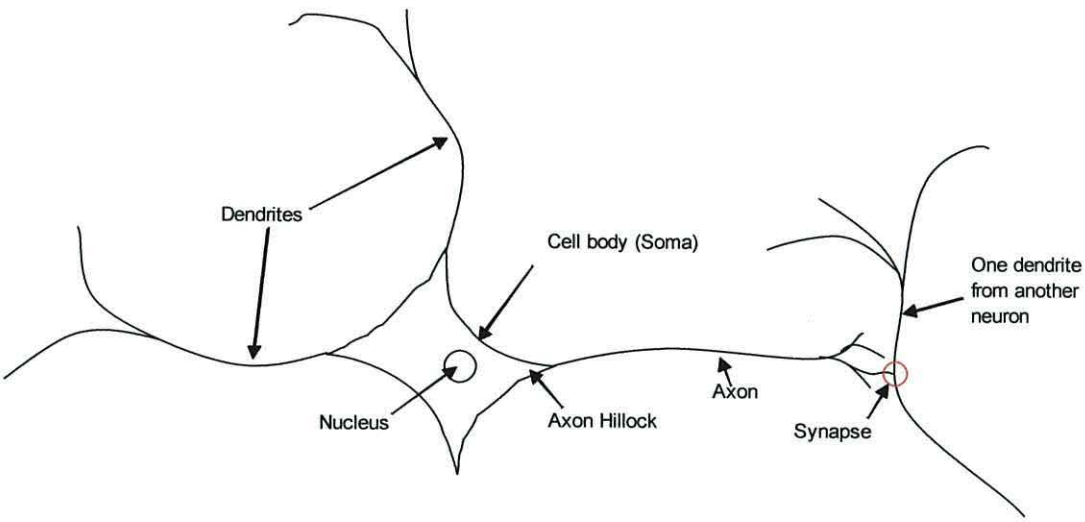

Figure 38: Form and layout of a neuron

Neurons communicate by means of action potentials, which can be thought of as short pulses of the form shown in Figure 39. The voltage and time scales in this figure are approximate, showing that the action potential usually measures around 100mV peak-to-peak and lasts for around one millisecond. These action potentials are usually approximately equal in amplitude from one to the next, but it is not the size of the pulse that determines the message to be transmitted, this information is encoded in the timing of the pulses and the effect each pulse has on the post-synaptic neuron. This effect is determined by the synapse which receives the pulse.

Action potentials occur due to the polarization of the neuron's cell membrane, which is due in turn to an imbalance of positive and negative ions when the neuron is at rest. The cell membrane has a potential difference across it, with the outside having a potential around 70mV higher than the inside when at rest. It is conventional to speak of the matter surrounding the neurons as having a potential of zero, so the neuron's internal potential is –70mV at rest. Any disturbance to this polarization will cause a change in the membrane potential at that point, which will affect the surrounding membrane, causing the disturbance to propagate across the cell.

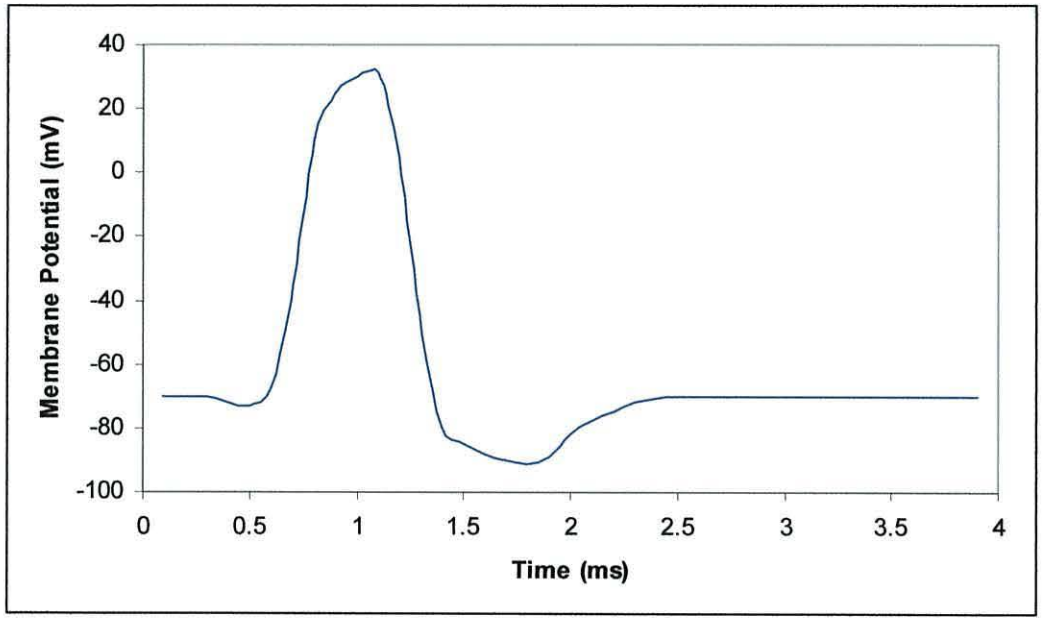

Figure 39: General form of an action potential

Embedded in the cell's membrane are a series of ion channels, which selectively allow ions to enter and leave the cell. Some of these are the ion pumps responsible for maintaining a concentration gradient of ions across the membrane, and therefore maintaining the potential difference between the inside and the outside. At the synapses, the ion channels are responsible for the change in membrane polarisation during synaptic events. Generally, sodium, potassium, calcium and chloride ions are responsible for the polarisation effect in a neuron.

Figure 40 shows a more detailed view of a single synapse. As an action potential reaches the synapse from the presynaptic neuron, the vesicles, containing neurotransmitter, move to and merge with the presynaptic membrane, releasing their neurotransmitter into the synaptic cleft. Chemical binding of the neurotransmitter with the receptor sites on the postsynaptic membrane's ion channels triggers an electrochemical process which alters the balance of positive and negative ions in the membrane, either reducing the magnitude of the polarisation (depolarisation), or increasing it (hyperpolarisation). This change in the polarisation, called a post-synaptic potential (PSP), can therefore either bring the membrane potential closer to zero, away from its normal resting point at around -70mV, or can push it further away from zero, making it more negative. A raise (toward zero) in the membrane potential brings the neuron closer to firing, so a PSP which does this is called an Excitory PSP, while a PSP which increases the polarisation is called an Inhibitory PSP.

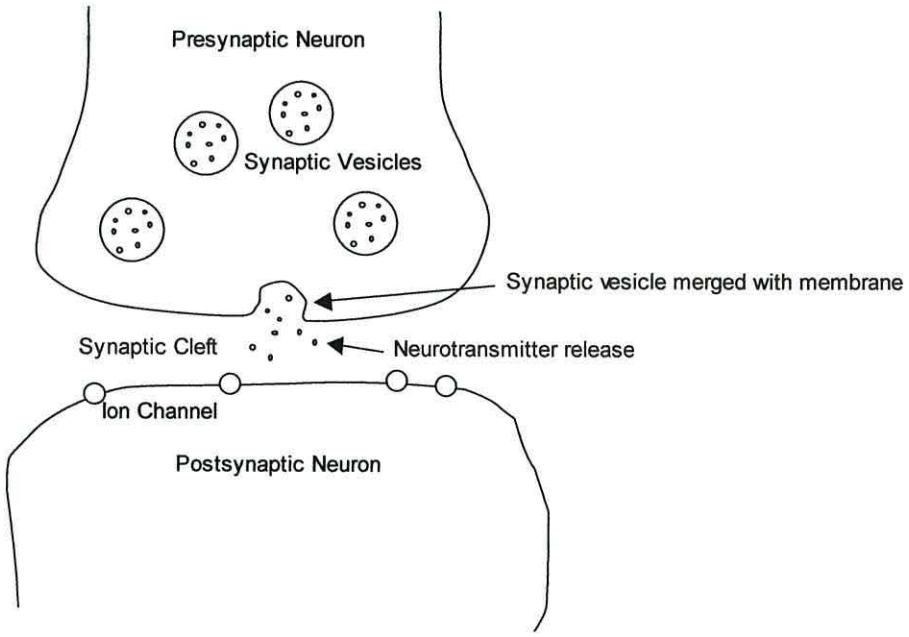

Figure 40: Detailed view of a single synapse

The magnitude of the PSP is determined by the electrochemical effect which the neurotransmitter has on the postsynaptic membrane, and as this will vary depending on the exact conditions present at each synapse, the PSPs from different synapses will all have different effects on the overall state of the neuron.

This gives rise to the concept of weighted inputs, where certain synapses have a greater effect than others, or a greater weight, while others may in fact reduce the neuron's chance of firing, having a negative weight.

Thousands of PSPs arrive from the thousands of synapses, converging on the axon hillock, where the axon joins the soma. If the cumulative effect of these raises the membrane potential at this point enough to reach a threshold, which is typically around -50mV, the neuron fires, and an action potential propagates along the axon, where it triggers the same synaptic processes in subsequent neurons.

Whenever there are no PSPs arriving from the synapses, the neuron slowly re-polarises, so that the membrane potential gradually returns to the resting value of around -70mV. Therefore, a stream of excitory PSPs may fail to trigger an action potential if the timing is such that the membrane potential is allowed to return to the resting potential between inputs. The neuron can thus be considered to be a frequency-dependant system, where a low input frequency will not produce an output. The size of the PSP generated by a particular synapse will have an effect on this frequency-dependence, as a very small PSP will have to occur more frequently than a much larger one in order to overcome the decay and cause the neuron to fire.

Figure 41 shows a simplified overview, approximated by straight lines, of the change in membrane potential with time for a neuron which is stimulated to firing point by a series of input pulses.

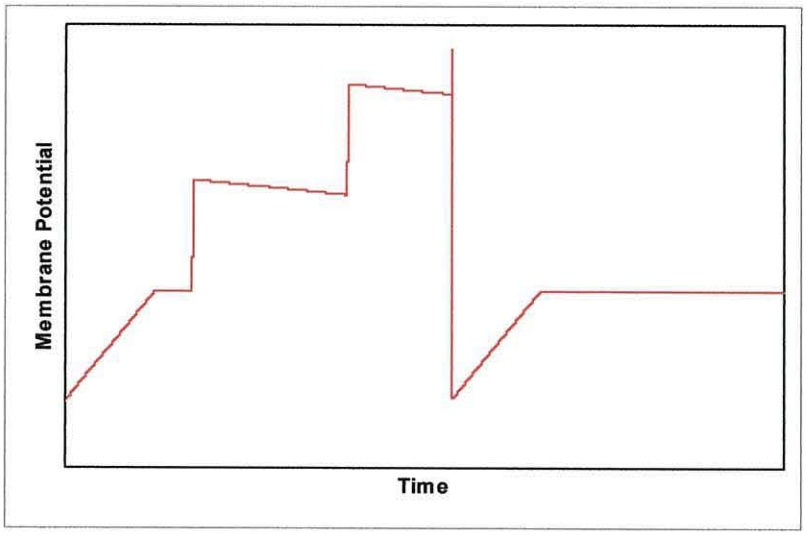

Figure 41: Simplified membrane potential response to three input spikes

As the neuron fires, the membrane potential rises rapidly before returning to a much lower value that the resting potential. Now, the ionic compounds responsible for the membrane polarisation must be replaced, and during this period, the refractory period, action potentials will not be produced, regardless of the input activity. There is therefore a minimum period of action potential generation, and a neuron which is constantly stimulated by incoming signals will have a maximum firing rate beyond which any further increase in stimulation will have no effect.

5.3: Artificial Neuron Models

The descriptions of the conventional artificial neuron models in the following sections are adapted from [5.4] and [5.5]. More complete details can be found in these texts.

5.3.1: Threshold Logic Unit

The simplest type of artificial neuron is the Threshold Logic Unit, or TLU. This provides a very simplified model of a neuron as a device which will produce an output signal if sufficiently stimulated by its input signals. The general form of such a device is shown in Figure 42.

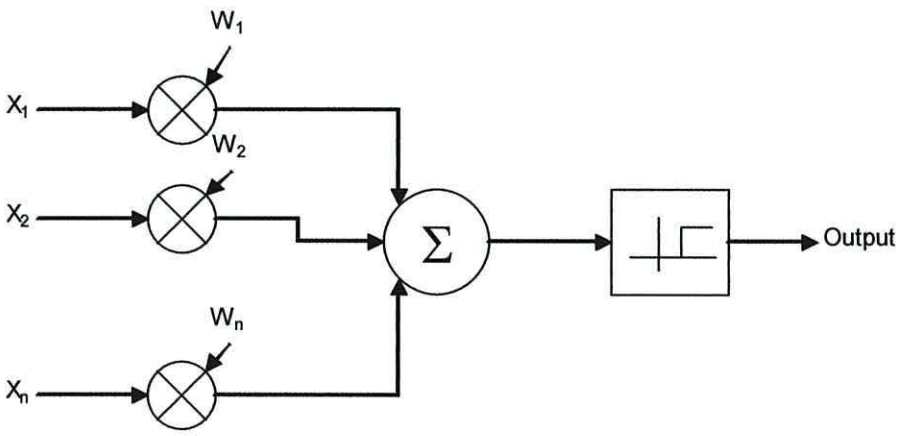

Figure 42: Block diagram of a threshold logic unit

The incoming signals are weighted and summed to determine the neuron's activation, which is then fed to a threshold unit to determine whether the neuron will fire. Weights can be positive or negative, representing excitory and inhibitory synapses.

The threshold function depicted in the figure, though lacking detail, shows the general form of the threshold response. The output, on the Y axis, is zero for any input (X axis) below a certain point, the threshold. For input above the threshold, the output is a fixed, higher value. It is conventional to consider these values to be 0 and 1, respectively.

Although this binary representation of the neuron's response limits the neuron to simple on/off outputs, it does give rise to some degree of error tolerance, as the weights can change slightly without altering the output from the neuron. Whether the activation of the neuron is just below or a long way below the threshold, the neuron will output 0, and similarly the output will be 1 whether the activation just exceeds the threshold or exceeds it by a long way.

An alternative activation function to the step function described above is the sigmoid function, which provides a continuous relationship between the activation and the neuron's output. An example is shown in Figure 43. The sigmoid curve is symmetric about the point at which the Y-axis value is 0.5, and the X axis value corresponding to this can be thought of as being equivalent to the threshold in this case.

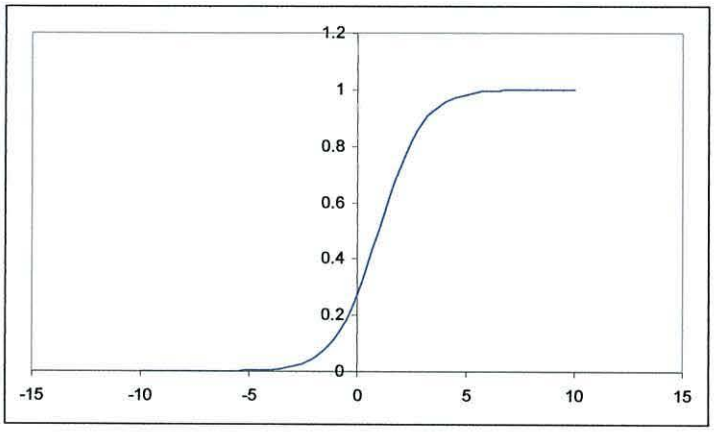

Figure 43: Example of a sigmoid function

The sigmoid function is expressed mathematically as

$$y = \frac{1}{1 + e^{-(a-\theta)/\rho}}$$

where θ is the centre point of the function and ρ determines the shape of the curve. The figure was generated with ρ = 1. Larger values make the curve flatter, while as ρ tends to zero the curve more closely approximates the shape of the step function.

The values passed between the neurons represent their level of activity, so for the case of the step function, the neurons in a network are either active or not active, represented by 0 or 1. Neurons with a sigmoid activation function can produce a variety of outputs.

5.4: Spiking Neuron Models

While the threshold logic unit can model the basic summation of inputs and threshold functions of a real neuron, many models have been developed which are based on a much more detailed analysis of the internal functions of the neuron, taking into account the spiking nature of the cells, and the details of the membrane potential. Spiking neuron models have been shown to be more computationally powerful than threshold or sigmoid-based models [5.6], demonstrating the ability to perform with a single neuron a function that requires many hidden layers in a network of TLUs. It is suggested [5.7] that the computational power of these neurons arises from the way in which information is coded, not only in the frequencies of the spikes but also in their relative timings.

5.4.1: The Integrate-and-Fire Model

The simplest of these models is the integrate-and-fire model, though the slightly more complex and realistic leaky integrate-and-fire model (LIF) is used most often. This model was first proposed by Lapicque in 1907 [5.8, 5.9], long before the mechanisms governing action potential generation were known and is based on a simplified view of the capacitance and leakage conductance of the cell membrane.

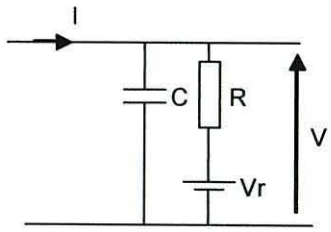

Figure 44: Overview of the Lapicque model

The structure of the model is shown in Figure 44. V_r represents the resting potential, to which the membrane potential V will gradually return in the absence of stimulation current I. C represents the membrane capacitance while R represents the leakage resistance. When I is positive, the capacitor is charged at a rate proportional to I, raising the membrane potential. It can be seen that the

model does not model the actual generation of an action potential, Lapicque postulated that when the membrane potential reached some threshold value, an action potential would be generated by some other means, and the membrane voltage would be reset to some sub-threshold value. If set to less than V_r, it would need time to return to V_r, providing a basic simulation of the refractory period.

Sample waveforms showing the response of the leaky-integrator to a varying input current are shown in Figure 45. The resting potential in the figure is -60mV, and it is apparent that the potential immediately after firing is a little lower than this. The threshold is around -45mV. The input current is large, the model fires repeatedly, until the current is lowered to a point where it cannot charge the capacitor more quickly than the resistor R is discharging it.

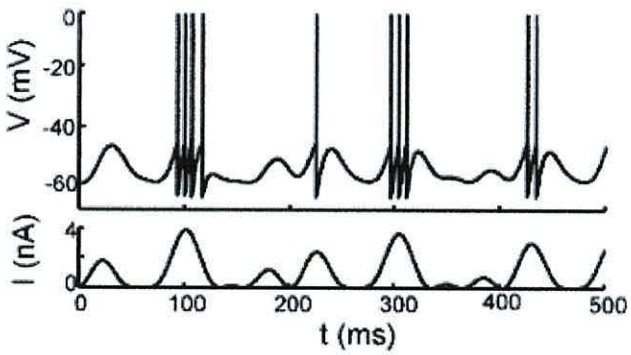

Figure 45: Response of a simple LIF model to a varying input current (adapted from [5.9])

The basic equation governing the response of the integrate-and-fire neuron is

$$\tau_m \frac{dV}{dt} = E_L - V + R_m I_e$$

where τ_m is the membrane time constant, V is the membrane voltage, E_L is the resting potential of the neuron, R_m is the total membrane resistance and I_e is the excitation current. In addition to this, the threshold rule is applied, so that when V reaches the threshold V_{th}, the neuron fires an action potential and V is reset to some value V_r.

Solving the above equation allows the subthreshold membrane potential V(t) to be calculated:

$$V(t) = E_L + R_m I_e + (V(0) - E_L - R_m I_e)e^{\frac{-t}{\tau_m}}$$

where V(0) is the membrane potential at time t = 0. This equation is valid only when $V < V_{th}$.

We can use this equation to calculate the firing rate of the neuron for a constant value of I_e. Assuming that the neuron has just fired at t=0, the membrane potential will be at the reset value, V_r. Note that this is not the same V_r as in Figure 44, as the resting potential is represented by E_L here. The next action potential will occur when V reaches V_{th}, at a time t_i,

$$V(t_i) = E_L + R_m I_e + (V(0) - E_L - R_m I_e)e^{\frac{-t_i}{\tau_m}}$$

Solving for t_i, we can obtain the firing interval for the neuron

$$t_i = \tau_m \ln(\frac{R_m I_e + E_L - V_r}{R_m I_e + E_L - V_{th}})$$

and since the firing rate $r_i = t_i^{-1}$ we obtain:

$$r_i = (\tau_m \ln(\frac{R_m I_e + E_L - V_r}{R_m I_e + E_L - V_{th}}))^{-1}$$

It is clear that in general the firing rate of this neuron model increases logarithmically as the excitation current increases, but for large I_e the increase can be considered to be approximately linear. This logarithmic response is discussed in section 5.14.2, where it is shown that the neuron can behave as an input encoder.

5.4.2: More Complex Models

The integrate-and-fire model was extended in the 1960s by the pioneering work of Hodgkin and Huxley into analysing the behaviour of squid neurons [5.10]. The Hodgkin-Huxley model [5.11], in its simplest form, provides a detailed and biologically plausible model for the membrane conductances and allows the input current I_e to be defined more realistically. The FitzHugh-Nagumo model [5.12, 5.13] was introduced later as a simplification of the Hodgkin-Huxley model. Both models are much more complex than the integrate-and-fire model, being based on modelling real-world biological and chemical processes.

5.5: Existing Implementations of Neurons and Networks

An interesting alternative to developing a system by training a neural network is the principle of evolved hardware, which was first demonstrated by Thompson [5.14], where a Xilinx XC6216 FPGA was programmed to perform a simple task – discriminating between two tones – by a process of evolution guided by a genetic algorithm rather than by explicit design of the circuit. A subset of 100 logic cells within the device was used, and a genetic algorithm starting with 50 random configuration patterns evolved a configuration pattern after 5000 generations which could distinguish flawlessly and rapidly between the tones. It was later found that most of the circuitry could be removed, leaving just 32 cells which were required, even though many of them appeared not to be connected. The circuit was shown to be making much more complete use of the dynamics of the individual transistors than a conventional digital logic design would, and its performance was very dependant on its operating environment. Later work by Fogarty et al [5.15] showed that true digital circuits could also be evolved, which were not bound to the analogue operating conditions of the underlying hardware. Simple evolved arithmetic circuits were shown which used far fewer resources than circuits generated by traditional design and optimisation methods, as the evolution process was free to explore the underlying logical structure of the logic elements themselves, rather than thinking of them as simply places where logic gates can be put. Since the first study of evolution in FPGAs there has been much work in this area, concentrating mostly on making non-hardware-dependant

circuits which function at the gate level rather than the analogue level [5.16] [5.17] [5.18] [5.19]. Gordon and Bently [5.20] provide a good overview of this field, which could be thought of as an alternative to neural networks.

There are two main methods of implementing artificial neurons in VLSI; analogue and digital. The former is based on the analogue models developed in the 50s and 60s, and is generally implemented in full-custom VLSI, making use of the analogue nature of the transistors [5.21][5.22][5.23], or sometimes in Field-Programmable Analogue Arrays (FPAAs) [5.24]. The analogue full-custom types can offer the highest neuron density, as each neuron may consists of a mere handful of transistors [5.25], equivalent to a few logic gates in a digital system, though these are expensive to develop. Full-custom VLSI implementations can also make use of mixed analogue and digital circuitry, with the spikes handled by digital gates while the integration is handled by an analogue integrator. [5.26]
The digital implementations are usually performed with FPGAs at the present time, since FPGAs allow rapid prototyping at lower cost than full-custom VLSI. Within the sphere of these digital implementations, there are many subtypes, ranging from simple implementations of threshold logic units to complete simulations based on hardware-accelerated software.

Vitabile et al. [5.27] present an efficient method for implementing a Multi-layer Perceptron in an FPGA. The MLP is a layered feed-forward network based on threshold units, and it is shown in this implementation that a replacement of the more conventional sigmoid activation function with a sinusoidal one results in a decrease in resource usage while retaining the precision and processing abilities of the network. It is also demonstrated that many of the variables and buses such as weights and pre/post synaptic connections can be reduced to small bit widths, in some cases as small as 3 bits wide, without compromising the operation of the network. A major point raised is that the main bottleneck in a system such as the one presented is the bandwidth between the parameter memory and the neurons themselves, and thus a method is presented where the network is pipelined and the neurons of the first layer are updated one at a time, writing the partial results into a series of FIFOs. Once the required number of results are present to allow the first neuron in the second layer to be updated, the second layer is processed, one

neuron at a time, and the process continues in this manner. It is shown that this method reduces the number of individual RAMs which must be used, as all parameters are held in a small number of large memories which are shared among the neurons.

The system demonstrated by Eldredge et al. [5.28] makes use of run-time reconfiguration to increase the density of a multi-layer feed-forward network using back-propagation learning by dividing the system's operation into three phases; feed-forward operation, back-propagation operation and weight updates. These three functions are implemented as separate blocks and time-multiplexed onto the FPGA, with the results of each phase being stored in external memory. The results presented show that when the system is partitioned in this way, six neurons can be fitted to each processor, compared with just one if all three phases co-exist on the same chip at the same time. It is shown that the time required to reconfigure the FPGA with its new function will slow the system down if a single chip is used, but when a large number of them are used in parallel, the advantage of being able to process 6 neurons on each chip at once outweighs the disadvantage of the extra time required for reconfiguration.

Bade and Hutchings approached the problem of reducing the hardware size from another angle, using a stochastic method [5.29] in which the neurons' activation values are represented by serial bit-streams where the magnitude is proportional to the quantity of 1's in the stream. This allows multiplication by the weights to be done with much less hardware (a single gate) than if a complete multiplier was used, and thus increases the number of neurons which will fit into a single chip.

One of the oldest FPGA implementations of a very large scale neural network is the CAM-Brain Machine (CBM), built by De Garis & Korkin [5.30][5.31] in which a total of 75 million neurons are implemented by 72 FPGA-based processors. The neurons are grouped into cellular-automaton-based 'modules' of around 1000 each, and each module is evolved by a genetic algorithm to perform a specific function. Up to 65000 of these CA modules are then loaded into a large memory to form the overall 'brain', which can be updated by the machine at a rate of 130 billion cells per second. At the time of publication, the performance gain

was estimated at around 10,000 times the performance of software running on a 400MHz Pentium II PC.

The actual cells used in the implementation of this system are quite different from either threshold logic units or spiking neurons, as they are simple CA cells of one of four types: axon, dendrite, neuron body or blank. [5.32] These exist in a 3-dimensional CA and operate together to produce the simulated neurons, of which there will clearly be fewer than there are cells in the CA. The overall effect of the set of cells which combine to produce each simulated neuron is roughly equivalent to a threshold logic unit, but the real distinction of this system is that the cellular-automaton based structure combined with the genetic algorithm used to develop the network results in the connections between the neuron bodies being 'grown' to suit the application rather than hard-wired in a regular structure. Each module has a 'chromosome' which guides the CA when growing these connections. A hundred modules are evaluated for fitness by comparing the outputs with the expected outputs for a range of input vectors, then the ten best are selected and a hundred more composed by 'mating' these ten. Eventually a module arrangement is reached which is deemed 'fit for purpose', and this can then be used in the final CBM, along with thousands of others.

While the above examples are all essentially threshold logic units, less work has been done in the area of implementing spiking neuron models in FPGAs. It is generally the case that the more complete spiking neuron designs based on a full implementation of the Hodgkin-Huxley model or similar tend to be implemented as analogue circuits, while the simple integrate-and-fire designs are often implemented digitally, but with a view to implementing large-scale networks rather than small networks or individual neurons.

Many FPGA implementations of larger networks are based on the principle of multiplexing the simulated neurons onto a small number of execution units, and simulating the neurons in a series of time-steps, with the parameters for the neurons and synapses stored in RAM, along with the status of each interconnection. Glackin, McGinnity et al [5.33] showed that while a fully parallel implementation could be expected to fit a few tens of neurons into an FPGA, a

multiplexed approach such as this could be used to implement a network of thousands of neurons with four main neuron processors.
The neuron processors used in this system consist of embedded microprocessors which have access to integrate-and-fire neuron models which they use as coprocessors. A single time-step for the entire network is calculated by operating the four processors in parallel with each processing a subset of the network. The results show an increase in speed of around 1000 times compared with a Matlab simulation of the same network, for the case of a large network of 4200 neurons and 1.9 million synapses.

Pearson et al. proposed a Biologically Plausible model [5.34], in which the simple leaky integrate-and-fire model is extended to simulate the axonal delay and noise in the weights and other parameters. Ten neuron processing elements, each consisting of a neuron model and a synapse model are implemented in parallel in a million-gate FPGA, and 120 virtual neurons and 912 virtual synapses are time-multiplexed onto each one. This implementation also makes use of 16-bit integer arithmetic in place of floating-point arithmetic, an implementation chosen for the simpler hardware involved. This implementation is not a 'neural network' as such, as it is considered to be a SIMD (Single Instruction Multiple Data) processor with custom hardware specifically geared towards simulating neural networks, a hardware-accelerated software implementation.

The SNN Emulation Engine (SEE) described by Hellmich et al. [5.35] is another, later example of this type, consisting of 3 FPGAs; a master controller, a Network Topology Computer and a Neuron State Computer. The NSC contains 3 Processing Elements, each of which has a link to fast SDRAM, containing the complete parameter set for each neuron and synapse. A total capacity of 2^{19} neurons and 803×10^6 synapses is achieved, and an acceleration factor of 30x compared with a standalone 2.4GHz PC is claimed, with the SEE operating at 50MHz. Although based on standard Xilinx Virtex II FPGAs, SEE is however a custom architecture, and its builders state that commercial and educational FPGA development systems are generally not adequate for this level of acceleration.

While many of these implementations are intended to model many hundreds or thousands of neurons, usually processing a few at a time, some work has been done in implementing smaller networks, sometimes with all the neurons of the network implemented in parallel. Roggen et al. demonstrate a simple parallel network of very simple spiking neurons [5.36] being used to control an autonomous robot. Despite the simplicity of the neurons, in which the arithmetic representation is cut down to use as few bits as possible, and the weights assume one of two fixed values, the network is shown to be capable of performing the task for which it was developed, and as a result of the simplicity 64 neurons can fit into a relatively small FPGA, along with an embedded processor for interfacing and control. It is stated that the speed of the network is two orders of magnitude greater than the software version, at the same clock speed. This is an interesting result as it demonstrates that useful neural networks can be built from neurons with greatly reduced hardware cost. Bellis et al. also showed that an autonomous robot could be controlled by a simple network [5.37], in this case a network of just four neurons, with two acting as input encoders and two as output generators. The neurons in this case were built with arithmetic hardware tailored to suit the sensors used by the robot, and were small enough to allow 40 to be implemented in parallel in a mid-range FPGA.

Upegui et al [5.38][5.39] propose a simple leaky integrate-and-fire model featuring Hebbian learning implemented in an FPGA, using a mixture of configurable logic and embedded RAM to compute the function of the neuron in a series of time-slices and a network of 30 neurons is trained to discriminate between two frequencies of input signal. It is shown that this simplified approximation uses much less hardware than more complex biologically-plausible implementations which perform a more detailed modelling of the neuron [5.40], and also that a change in the number of inputs to the neuron affects not only the hardware size but also the latency, as the need to integrate over more inputs means that more clock cycles are required.

This neuron model is a good basis for development of other simple integrate-and-fire models, as it captures the basics of neural operation.

Savich et al. discuss number representation and the design of the neuron core [5.41], concluding that fixed-point computations can be executed with much less hardware than their floating-point counterparts, generally achieving a halving in the area required and an increase in clock rate when comparing implementations in which the range and precision of the two formats are similar. It is also shown that the area required can be reduced by as much as 80% if the weights are processed serially with a multiplier-accumulator, rather than being processed simultaneously with a series of separate multipliers and an adder tree.

Schrauwen and Van Campenhout address the issue of hardware size in an alternative way, realising that while software implementations tend to follow the architecture of the host computer, a hardware implementation can be based around a completely different architecture, which can result in significant improvements in operating speed or resource usage in an FPGA. The method proposed is that of serial arithmetic [5.42], where 1-bit adders replace the wider parallel adders and registers are replaced by shift registers. The result of this is that the neuron requires far fewer logic cells than an equivalent implementation with parallel arithmetic, and due to the simpler hardware it also can operate at a much higher clock speed, though as it uses serial arithmetic more clock cycles are required to perform each operation. This approach has been previously taken by others such as Torres et al. [5.43] with similar findings.

5.6: FPGA Spiking Neuron Model

It was decided to investigate the implementation of a spiking neuron model rather than a simpler model such as a threshold logic unit because the spiking neuron model is capable of more complex and interesting dynamics. A TLU can be thought of as a static system, it responds to steady inputs, producing a steady output signal. When the inputs change the output may change, or it may not, but it will quickly reach a new steady state, with any changes in output following changes in input in terms of timing. A spiking neuron model responds not to the steady-state magnitude of the input signal but to the timing of the pulses, producing pulses at a rate proportional to its excitation, and can therefore both produce and respond to signals with more complex time-dependant characteristics. A spiking neuron model also more closely mimics the dynamics of a real neuron, and it was for these reasons that this type of neuron model was chosen for implementation.

The Spiking Neuron Model presented in this chapter is intended to model the timing-dependant operation of a real neuron without modelling the internal functions in detail. As described above, the model does not represent the full, complex dynamics of a real neuron, but replicates the operation of a simplified leaky integrate-and-fire model with integration, decay and the threshold function modelled as approximations.

It was decided that the neuron model design should have two major features; firstly, it should operate independent of any others to which it is connected, and therefore should not require to be synchronised with the others in the network. This should bring the neuron a little closer to an analogue implementation, where processing is carried out in real-time. Secondly, the hardware should be as simple as possible, to make its LE count as low as possible. This latter requirement necessitates a trade-off between the precision and realism of operation and the size of the hardware. In addition, if the model is kept very simple, it should be possible for it to behave as if it is performing its functions in real-time, using a high enough clock speed to keep latency small.

5.6.1: First Implementation of a Leaky Integrator Model

The neuron model was built with four inputs, a simplification which makes it suitable for simple neural networks with reduced connectivity. This was chosen arbitrarily, and the model can be extended to have more inputs, or reduced to fewer inputs. It was also built to operate at the relatively low speeds at which real neurons operate, with firing rates of a few kilohertz at most. This low activity rate compared with the tens or hundreds of megahertz clock rates possible with the FPGA logic makes it possible to perform a large number of internal operations between input spikes.

5.6.2: Overview

The basic neuron model has four inputs and one output, along with a series of control signals which allow parameters to be updated in real-time. The internal structure of the model is summarised in Figure 46.

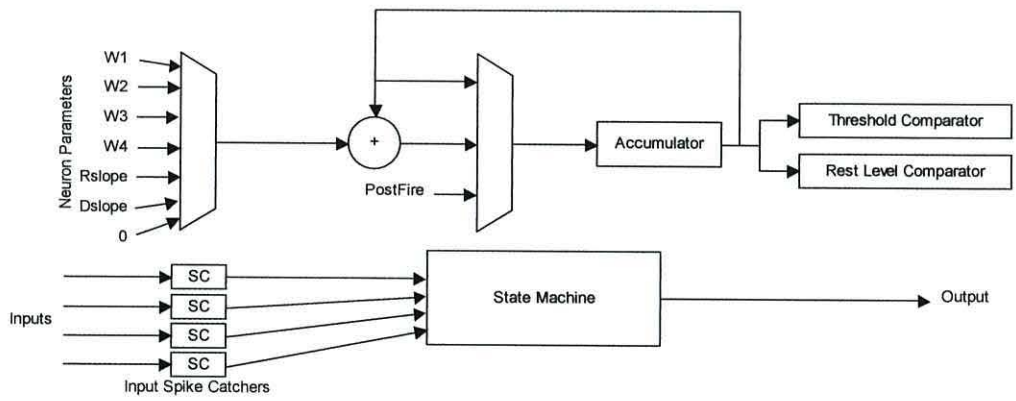

Figure 46: Block diagram of the neuron model structure

Since the neurons communicate through short spikes, it will generally not be possible for the receiving neuron to detect a spike if it has to do so by repeatedly checking its inputs under the control of a state machine while also performing other tasks necessary for the model. An output spike is one clock cycle in length, so there is a chance that the receiving neuron will not be checking its input during that particular cycle, and will miss the spike. To correct this the inputs are fed into Spike Catchers, in which the input spike sets a flip-flop which can be read at the

next convenient time, and is reset by the state machine once the spike has been received.

Output spikes are generated directly by the controlling state machine as it passes through its firing state.

5.6.3: Neural Processing Core

The modelling of the simplified neuron function is performed by a simple data processing system consisting of an accumulator, adder and two multiplexers. The value in the accumulator represents the membrane potential in arbitrary units. While the membrane potential in a real neuron rests at -70mV with no activity, and can rise to around +90mV during firing, the simulation uses only positive integers, resulting in a representation of the membrane potential which has the same form, simplified, as that of a real neuron, but with a DC offset and expressed in arbitrary units in order to simplify the design of the arithmetic logic. This is simpler even than fixed-point, though as fixed-point representation involves using the top n bits as the integer and the remaining bits as the binary fraction, an integer representation using 16 bits can be thought of as being equivalent to a fixed point representation but with all the numbers multiplied by some power of 2. The multiplexer feeding the accumulator allows fully synchronous operation without using logic in the clock generation. The accumulator is clocked on the falling edge of every clock cycle, regardless of the current process operation. Whenever the data in the accumulator is required to remain unchanged, the multiplexer feeds the accumulator's output back to its input. This method ensures that short glitches will not affect the accumulator. These glitches are inevitable in FPGA logic when decoding wide bit vectors due to the fine-grained nature of the device, which often requires that the state decoding logic is formed from several cascaded layers of logic cells.

When the membrane potential is changed, either by an input spike, or by the gradual re-polarisation between spikes or after firing, this multiplexer feeds the output of the adder back to the accumulator. The adder produces the sum of the current accumulator value and a selected parameter, chosen by the state machine and control logic and fed from the parameter registers by a second multiplexer

under the control of the state machine. Any negative numbers must be entered in two's complement representation.

5.6.4: Neuron Control

The basic operating cycle consists of checking the four inputs for spikes, adding the weights to the accumulator if any spikes have been received, then adding the decay constant to the accumulator to represent re-polarisation. The chain of states in the state machine which perform this checking function is balanced so that if an input doesn't have a spike waiting to be integrated, extra 'blank' states are executed in place of the integration states. This ensures that the processing loop takes a constant number of clock cycles to execute each time (15 in the case of 4 inputs), regardless of the input signals, and therefore the decay function works at a constant rate. Each spike catcher is cleared immediately after the integration state, and since this is only done if a spike has been captured, there is no chance of the spike catcher being cleared at the instant a spike arrives, provided that the spike rate is significantly lower than the clock rate, as was assumed when the neuron was designed. If a spike has been captured, it must have occurred within the last loop cycle, and therefore the next spike will not occur for some time.

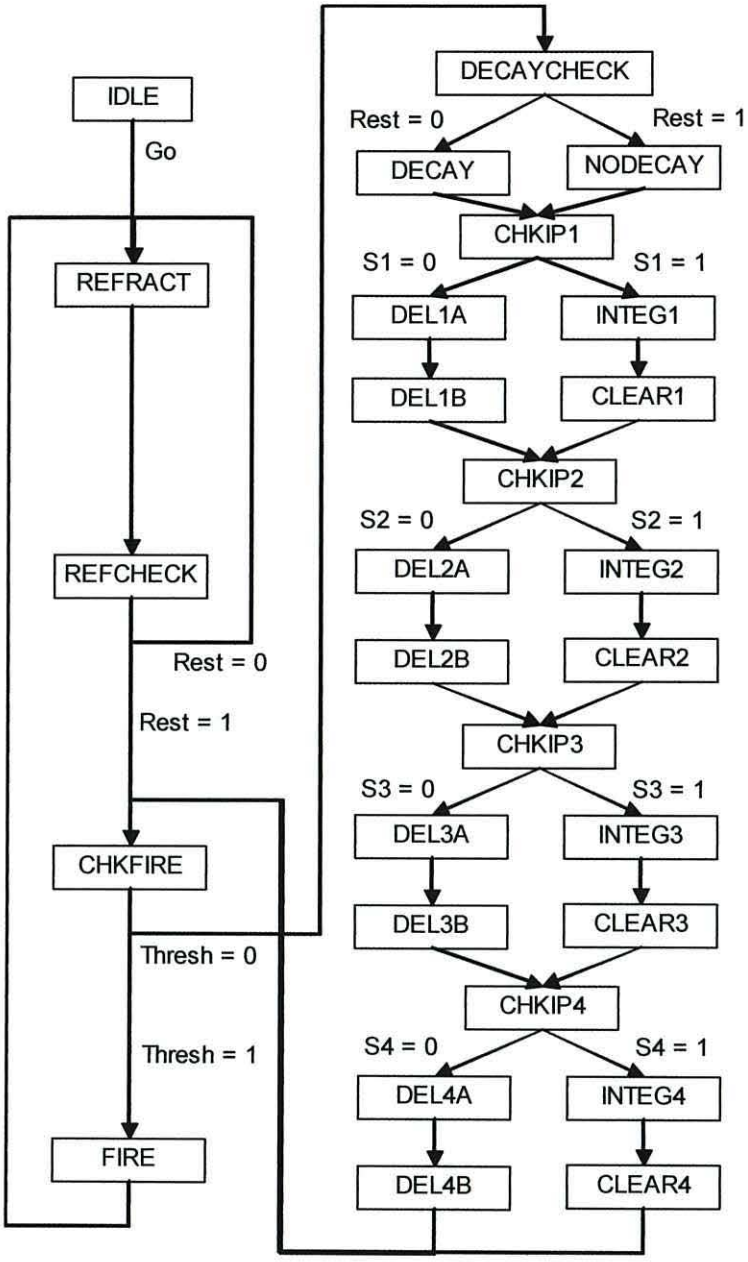

Figure 47: State transition diagram for the four input neuron model

The state diagram for this neuron model is shown in Figure 47. The neuron starts in the idle state, waiting for a 'GO' signal, a facility which allows all neurons in a network to be reset and started simultaneously. The period between resetting the neuron and starting the model with the go signal is used to load the model parameters into the registers without any spurious responses from the neuron's output due to incorrectly set parameters.

Once the neuron model is started by the GO signal, it enters the refractory period, during which time it will not respond to input stimuli. To ensure this, the spike catchers are constantly reset during this loop. If this were not the case, any incoming spikes during this period would be held until the start of the integration loop and would be integrated once the refractory period was finished. The refractory period ends when the resting level comparator signals that the membrane potential is within the limits set for the resting potential. At this point the system enters its main processing loop.

At the start of each pass through the loop, the output from the threshold detector is checked, and if it is signalling that the threshold has been exceeded, the state machine passes through the FIRE state and back to the refractory loop. During the FIRE state the accumulator is loaded with the post-firing potential specified in one of the parameter registers, and an output spike is generated.

If the threshold has not been exceeded, the processing continues with the decay of the membrane potential and the integration of the spikes. Firstly, if the accumulator is not at the resting value, the decay constant, a negative value, is added to the accumulator, then the four inputs are checked in succession and if a spike has been captured at any input, the weight value for that input is added to the accumulator. The use of a negative constant for the decay slope allows the same adder to be used for the decay, integration and refractory period, saving hardware.

The neuron's parameters are held in a set of registers which appear from outside the unit to be ten words of write-only memory. The addresses for the parameters are shown in Table 12. The registers allow the parameters to be updated while the neuron is operating, and in particular allow the weights to be adjusted without stopping the neuron.

Address	Parameter
0	Weight 1
1	Weight 2
2	Weight 3
3	Weight 4
4	Upper limit of resting potential
5	Lower limit of resting potential
6	Threshold
7	Decay slope
8	Refractory period slope
9	Post-firing membrane potential

Table 12: Parameter addresses for the neuron model

5.7: Testing the First Neuron Model

The aim of the tests presented in this section is to demonstrate that the neuron performs the integrate-and-fire function correctly, and to show that the decay function and refractory period are correctly handled. The timing-dependant response of the neuron to the input spikes and its response to inhibitory inputs are also shown.

The basic test of the excitory response consists of feeding a train of spikes into an excitory input, with each spike increasing the membrane potential, until the threshold is exceeded and the neuron fires.
Referring back to the assumptions made when designing the neuron, it was assumed that the spike frequency on any input would be low compared with the neuron model's clock frequency, and so a number of decay cycles would have elapsed between input spikes. If the spike frequency is low enough the decay will cause the potential to drop sufficiently far between spike inputs that it will never be able to reach the threshold. This is tested by varying the decay slope and the frequency of the input spikes.

The neuron was set up with the resting potential boundaries set to 1000 and 1050, the threshold set to 3000, and the post-firing potential set to 200. The refractory and decay slopes were set to 17 and -6 respectively. The choice of these values was not determined by any specific requirements, but to provide a useful range of membrane potentials over which the neuron could be tested and to provide a large enough response to be displayed graphically.

5.7.1: Simple Spike-Train Test

The neuron was tested with a train of three spikes. Figure 48 shows the input and output spike trains and the simulated membrane potential as the spikes are entered. The input weight was set to 1000, so assuming that the resting potential is probably just above 1000, two spikes should be enough to fire the neuron with no decay present. With moderate decay, three spikes will be required. The resting potential will not be at exactly 1000 (its lower bound) because the refractory slope

is +17, so the initial build-up of potential will result in the potential resting at the closest multiple of 17 to 1000, which in this case is 1003.

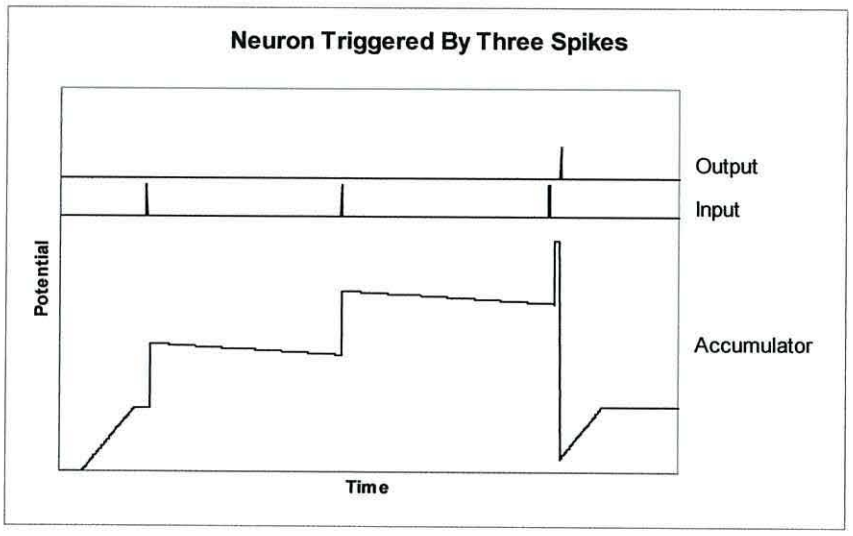

Figure 48: Neuron model response to a train of three input spikes

The decay can be seen between the input spikes as a gradual drop in the membrane potential. Each spike causes a sharp rise in the potential, with the third spike causing the potential to exceed the threshold. Once this happens, the neuron fires, outputting a spike and entering the refractory period. Due to the refractory slope value as described above, this second resting potential is actually 1016.

In order to test the timing dependant response of the neuron, the decay function was made more aggressive, taking the membrane potential back down more quickly, to show that the three spikes entered in the first test could no longer fire the neuron. These three spikes were then brought closer together to show that a faster train of spikes could overcome this more aggressive decay and cause the neuron to fire normally.

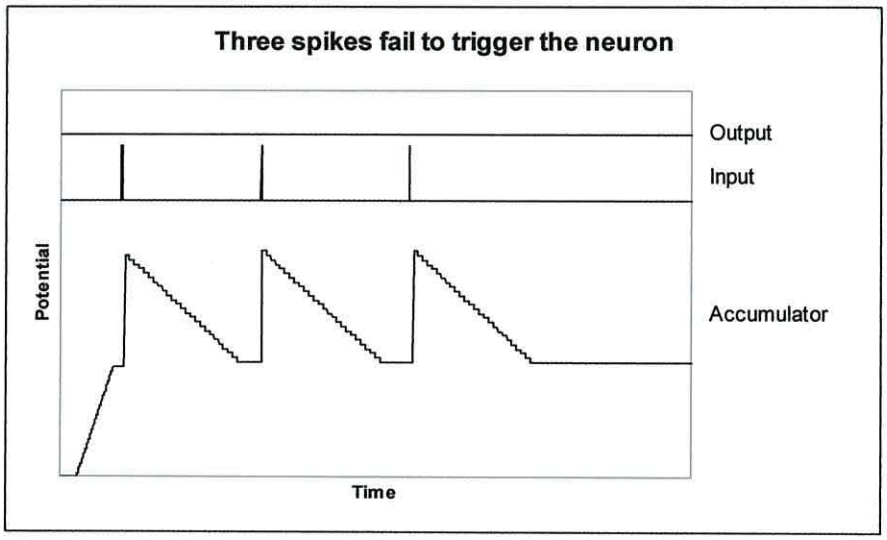

Figure 49: Neuron fails to respond to three input spikes

Figure 49 shows a second test using the same input spike train as the first test. Here, the neuron was set with a more aggressive decay, using a decay slope value of -40. This extra decay slope was enough to ensure that the spikes failed to trigger the neuron, because the time period between spikes was long enough for the membrane potential to reach the resting potential, and therefore cancel the increase brought about by each spike. Note that the time scale used in this figure differs from that used in the first result graph, in order to show the entire waveform.

Figure 50 shows a faster spike train fed to the neuron with the same parameters as the previous test. Here, the neuron fires, because the shorter delay period between spikes does not allow time for the membrane potential to fully decay.

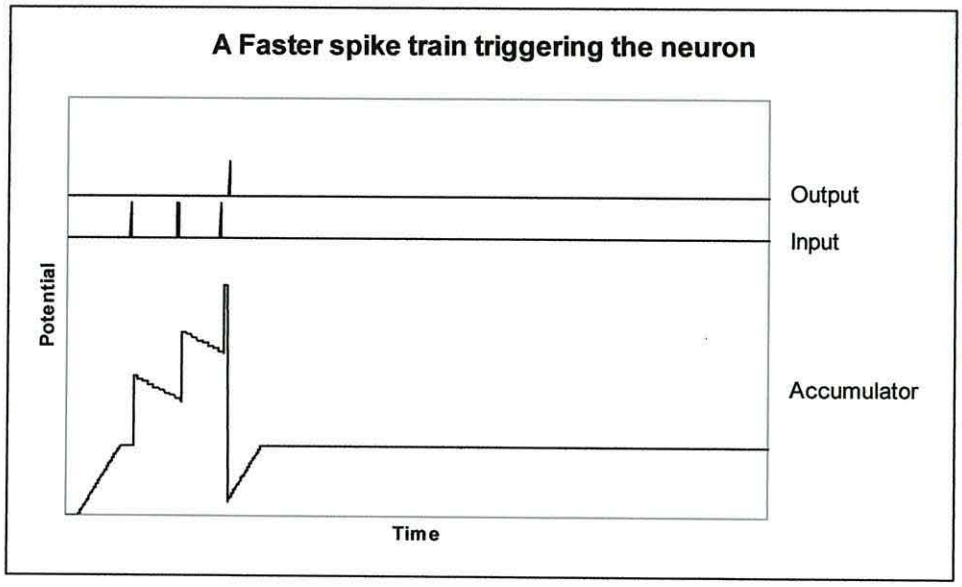

Figure 50: A faster spike train triggering the neuron

This graph uses the same timescale as Figure 49, showing the shorter delay between spikes.

This dependence on the timing of the input spikes shows that for any configuration of the neuron's parameters, there will be a minimum rate of input spikes below which the neuron will not be triggered.

5.7.2: Refractory Period Test

During the refractory period, the neuron should ignore any incoming signals, and should clear the spike catchers so that any spikes caught during this period are not read once the refractory period is over. Figure 51 shows the two spikes arriving too close together, so that the second arrives during the refractory period. For the purposes of this test the weight of the input was set to exceed the threshold value with a single spike.

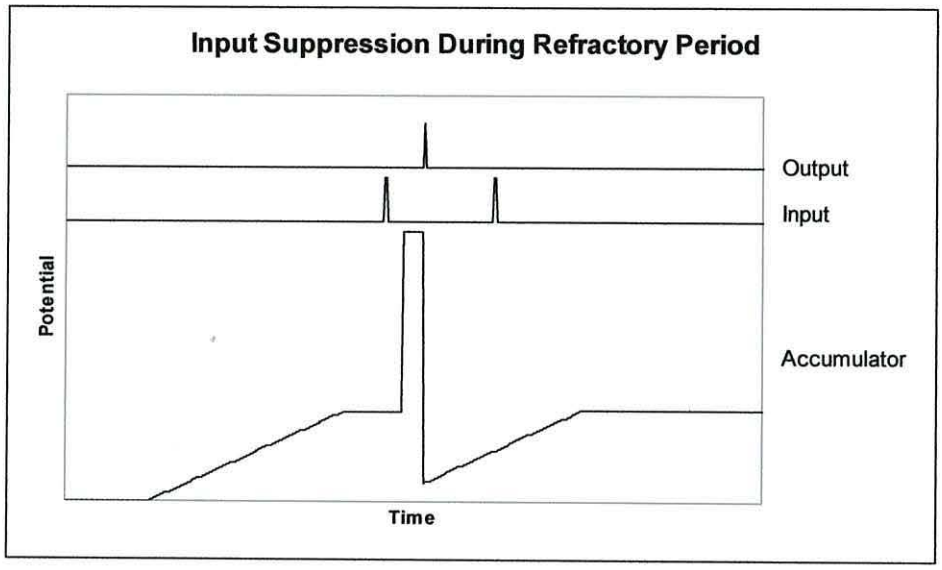

Figure 51: Input suppression during refractory period

It can be seen from Figure 51 that the second spike is completely ignored. There is also an apparent delay between the input spike causing the membrane potential to exceed the threshold, and the neuron firing, which is due to the short period used for testing. In order to complete the simulation quickly using the Quartus simulator, the parameters and clock speed were set so that the refractory period was very fast, and the spikes were fed in at rates far higher than a real neuron would encounter. The delay caused by the few states performed by the state machine between integrating the spike and firing would be a much smaller percentage of the spike-to-spike time if the neuron was used with biologically realistic input spike rates.

5.7.3: Inhibitory Input Test

An important aspect of the neuron's functionality is the ability to assign a negative weight to an input to simulate an inhibitory synaptic response. To test this the second input was given a weight of -990 and spikes were inputted to both inputs simultaneously. The result of this test is shown in Figure 52.

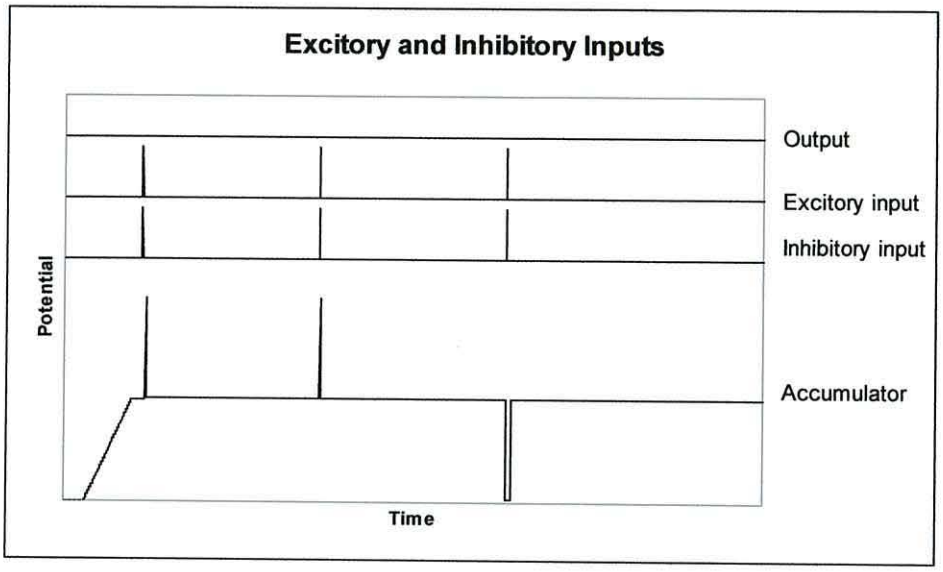

Figure 52: Neuron response to excitory and inhibitory inputs

Figure 52 shows that the first two spike pairs arrived at a time when the neuron was either checking inputs 3 or 4, or performing the decay function. This can be seen from the potential waveform, which shows that the excitory input was read first, raising the potential, followed shortly by the inhibitory input, which lowered the potential back to the resting level. In the third case the spikes appear to have arrived after the neuron checked input 1 but before it checked input 2. Referring to the state transition diagram in Figure 47, there are two states, DEL1A and DEL1B, in this interval. The waveform shows that the inhibitory input was read first, followed by the excitory input after a slightly longer delay than in the first cases, after the neuron had read inputs 3 and 4.

This response shows a potential problem with this particular neuron design, as an inhibitory input that does not coincide with an excitory input will push the

simulated membrane potential below the resting level. The resting level detector simply detects whether the potential is within the correct range, and if it is not, the state machine makes assumptions about whether the potential is higher or lower than the resting level based on the operational phase (resting, firing, refractory period) it is currently simulating. If an inhibitory input causes the membrane potential to drop below the resting level during normal neural operation, the state machine will assume that the potential is above the resting level and will apply the decay function accordingly. If an excitory spike arrives in time, the lowered membrane potential will suppress its effect, but if no other spikes arrive within a suitable time period, the potential will decay all the way down to zero, and then continue past zero. Figure 53 shows the response of the neuron to a single inhibitory input with a weight of -500.

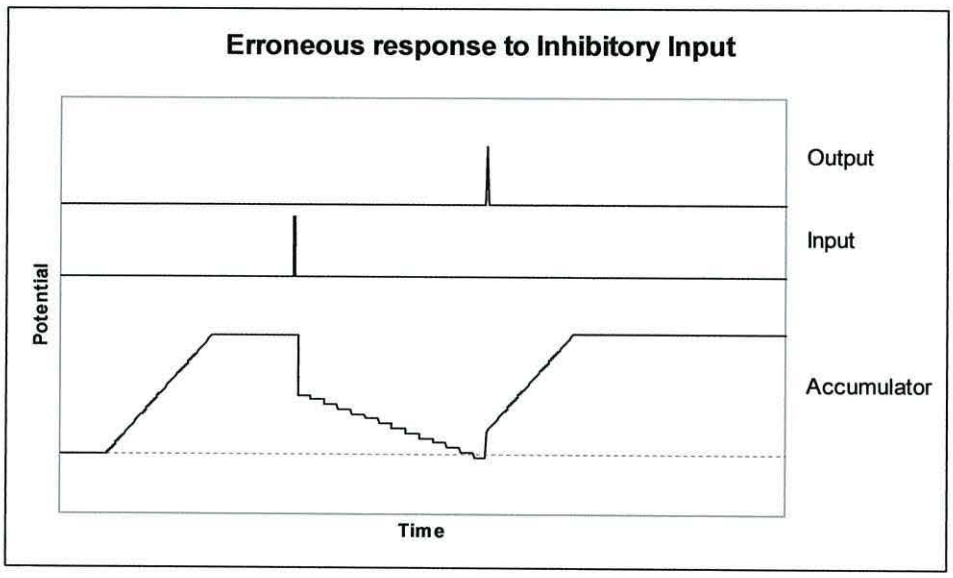

Figure 53: Neuron responding incorrectly to inhibitory input

The dashed line in Figure 53 represents zero potential, and it can be seen that the membrane potential crosses this line after decaying away from the resting level. The threshold comparator assumes that the membrane potential is an unsigned integer, but the repeated subtraction, even in an adder which is not explicitly built to use signed numbers, will eventually take the value past zero, making it negative, if expressed in 2's complement notation. This negative number is also, if

interpreted as being unsigned, a very large positive number, which exceeds the threshold, and causes the neuron to fire incorrectly.

There are various ways to correct this problem. If the resting level comparator could specify to the state machine whether the potential is above or below the resting level, the state machine could choose to either increase or decrease the potential to bring it back in range. This would allow an inhibitory input to have the desired effect without the problem described above, but would require extra logic and possibly an extra parameter defining the positive 'decay' slope. An alternative method is to prevent the potential being pushed below the resting potential at any time, though this would also prevent an inhibitory input from affecting any future excitory inputs. The effect of this on the operation of the neuron are shown in section 5.9, where a more complex neuron model is developed.

5.7.4: Hardware Test

The previous tests were performed entirely in simulation, using the simulator module of the Quartus software. The neuron model was also tested in real hardware, using an FPGA development board. Figure 54 shows a schematic view of the additional hardware connected to the FPGA to provide input pulses and to display the output.

For this test, the setup parameters were loaded into the neuron from ROM when the system powered up. A pulse generator with variable frequency provided the input spikes, through a transistor buffer which matched the generator's output to the FPGA's 3.3V CMOS logic levels. The upper byte of the 16-bit membrane potential was connected through a simple 8-bit resistor-tree DAC to an oscilloscope, to provide a trace similar to those shown in the simulated tests.

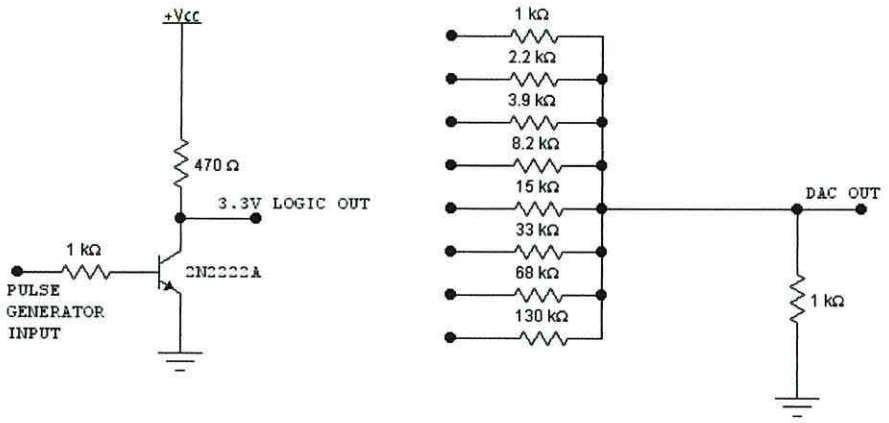

Figure 54: RTL level translator and resistor-tree DAC

In order for this to provide an accurate view, the resting level bounds were set to 10000 and 10050, and the threshold was set to 30000, thus ensuring that the upper 8 bits of the membrane potential would change significantly enough to show a detailed trace during the test. The weight for the input was 2000.

The oscilloscope traces are shown in Figure 55. The upper trace is the output from the DAC and the lower trace is the input signal.

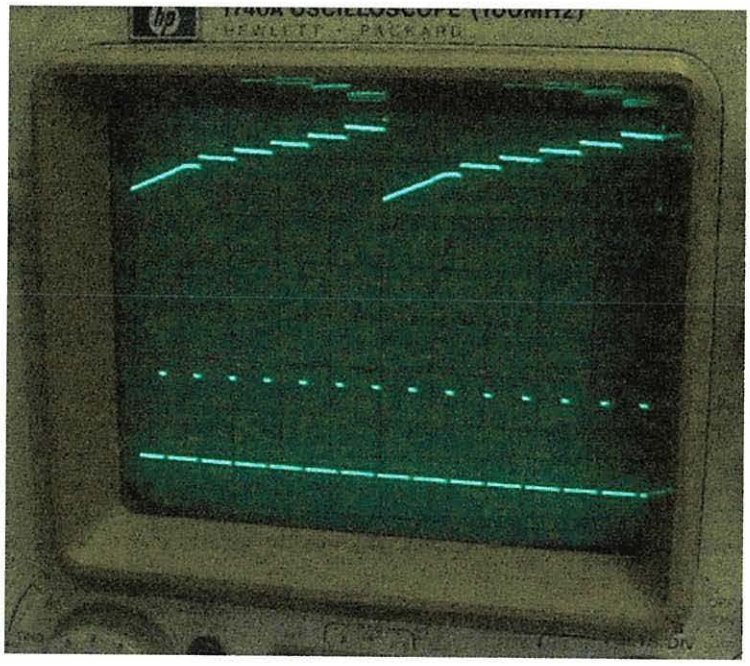

Figure 55: Photograph of oscilloscope traces during neuron test

The slight decay of the potential is visible between the jumps caused by the input spikes, as is the refractory period during which the incoming spikes are ignored.

5.7.5: Conclusion

The neuron model has been shown to be capable of the functions for which it was designed, namely the leaky integration, threshold-based firing and refractory period simulation. The response matches the expected response, and the response of similar simple spiking neuron models [5.38].

The inhibitory input test demonstrated that the inhibitory function works but is flawed, the flaw arising from the simplified processing performed by the neuron model. Methods of correcting this flaw were proposed, but not implemented as this correction is addressed by the more complex neuron design of section 5.9.

Further analysis of this design and its performance can be found in section 5.12.

5.8: An Experimental Neural Network

In order to check the resource usage of this first neuron model, a simple network was built with sixteen neurons and a set of control hardware. The aim of this stage of the development was not to produce a working neural network with practical applications, but instead to investigate the basic infrastructure required to set up and control such a network.

The neurons were implemented in parallel, which limits the number which can be fitted into the device. The neurons were found to use between 350 and 360 logic elements each, varying slightly across the set of 16, depending on the optimisations which the compiler could apply. Assuming 360 LEs per neuron, a total of 32 neurons could fit in the chip, so with just 16 neurons in the network, half of the chip's logic resources were left available for control logic.

The network connections were chosen arbitrarily and a variety of different arrangements were implemented, showing little change in the logic usage of the neurons from one to another. A few of these arrangements are shown in Figure 56.

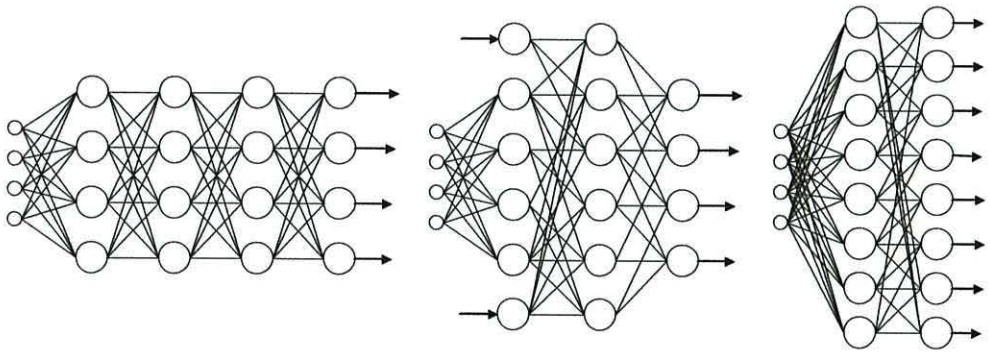

Figure 56: Three examples of networks of 16 neurons

The three networks shown represent stages in the conversion of the network from a four layer to a two layer network. The smaller circles represent the four inputs to the network, though the two layer network was also tried with 8 inputs, showing no significant change in resource usage other than the four extra pins. Note that only the four layer network can have full connectivity between the layers, as each neuron has only four inputs. The other networks were either wired randomly or so that neuron N in a layer takes input from the nearest four neurons in the previous

layer. The inputs to the top and bottom neurons in the three layer network represent feedback, where each neuron was fed by the four main network outputs.

5.8.1: User Interface Hardware

Since these neuron models do not use any of the FPGA's internal memory blocks, it was decided to make use of these to hold firmware which would provide a user interface, allowing the neuron parameters to be altered while the network is operating.

The simple microprocessor described in chapter 4 was used to provide this interface. VGA display circuitry was used to output the results, initially providing a complete screen display, but later restricted to 32 x 32 character cells to save video memory. All RAM, ROM and video memory required by the system was implemented inside the FPGA.

Figure 57 shows the overall layout of the system. The complete system used 63% of the 300K-gate FPGA's logic resources (7,259 LEs, or 189,000 gates) and 66% of its internal memory (98,560 bits total).

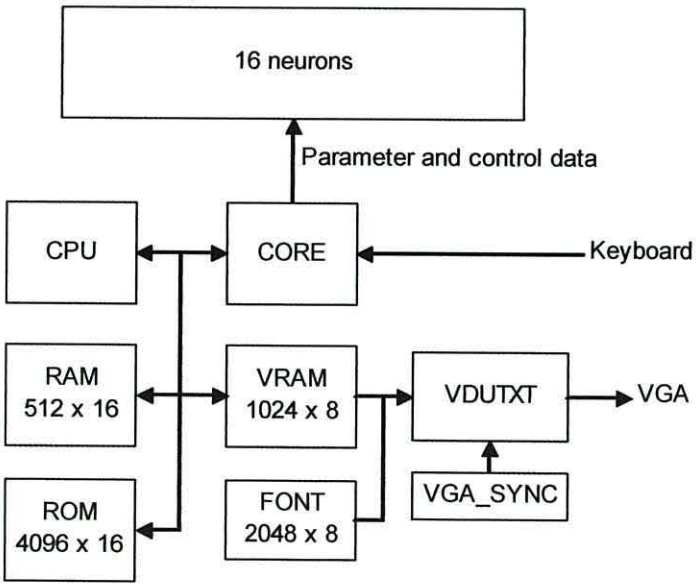

Figure 57: Block diagram of neural network control system

The CPU is assisted by the block named CORE, which provides address decoding and input/output interfaces for the system. The original design used the Digilab

board's four onboard switches for input, these were later replaced by a keyboard providing a more straightforward user interface.

The memory map for the processor is shown in Table 13.

Address (Hex)	Function
0000-1FFF	ROM (4096 words)
2000-3FFF	RAM (512 words)
4000-5FFF	Neuron parameter write
6000-7FFF	Neuron readback
8000-9FFF	I/O ports
A000-BFFF	Video RAM (512 bytes)
C000-DFFF	Keyboard register
E000-FFFF	Unused

Table 13: Memory map for network controller

The I/O ports allow the processor to control the neurons, control the on-board LEDs, and read the switches. The keyboard register is read/write, and when a key is pressed this will contain the scan code of the key. When written to, the register is cleared, allowing the software to determine whether a new key code has arrived. This function is assisted by a state machine built into CORE which provides a bridge between the CPU and the keyboard receiver. The particular keyboard used was chosen because its default scan code set produces a single code for each key, whereas a standard PC keyboard will produce longer code sequences for some of the keys, complicating the interface.

The neurons are accessed as write-only memory, with each one taking up 16 words of the memory map. Of these, the first ten are implemented as described earlier. The neuron read-back port allows the membrane potential of any neuron in the network to be read, the original intention being that these values would be displayed on the screen to allow the network to be monitored. The read-back port was implemented as part of the network hardware but no use was made of it by the software during these initial tests.

The video output provides 32 lines of 32 characters with a fixed character set. Each byte in the video memory corresponds to a single screen character cell. The video synchronisation block VGA_SYNC, along with the receiver for the keyboard, were adapted from the code provided with 'Rapid Prototyping of

Digital Systems' [5.44], the text accompanying Altera's University Program UP1 and UP2 development boards.

5.8.2: External Hardware

Figure 58 shows the development board with the keyboard and VGA interface fitted. The VGA interface, visible at the top right hand corner of the picture, simply provides clamping diodes and pull-up resistors for the red, green and blue signals, and was based on the interfaces provided on other development boards in the Digilab series. The early versions of the firmware used the four buttons visible below the four digit display for input.

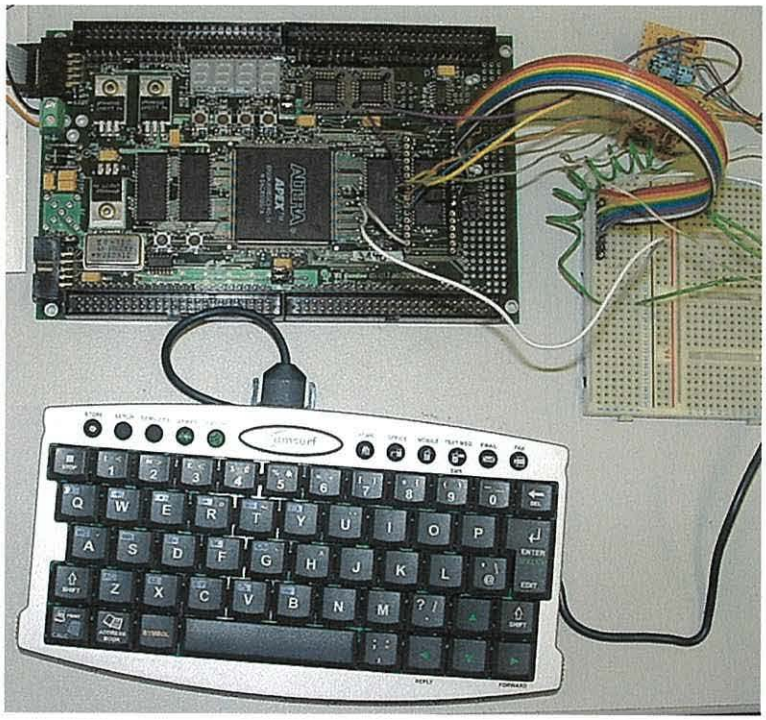

Figure 58: Photograph of the Digilab system with keyboard and VGA interface

5.8.3: User Interface Software

The software allows the neurons' parameters to be changed in real-time. Initially, due to the limited input capability of the four switches, this was limited to simply selecting a parameter and incrementing or decrementing it. Various methods were

tried to allow direct number entry with the buttons, but eventually the keyboard was fitted to simplify matters.

Using the arrow keys, a neuron (left/right) and a parameter (up/down) are selected and enter is pressed to update it. The parameter displayed the screen is blanked, and a new four digit number can be entered. Once the last digit is entered, the software returns to parameter selection mode, and the updated parameter is written to the correct register in the selected neuron.

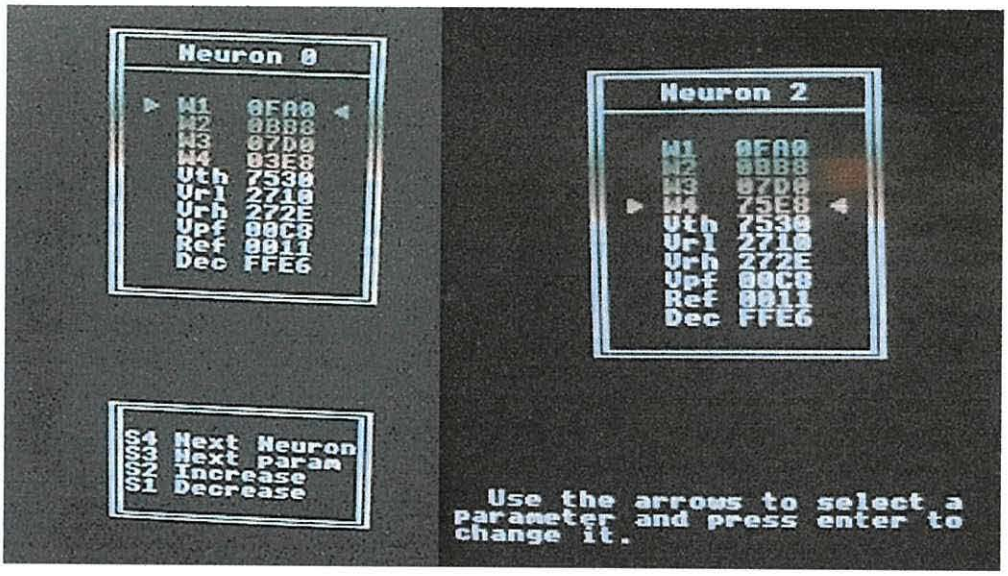

Figure 59: Photographs of the user interface displays

5.8.4: Conclusion

This experimental network, though not functionally useful, demonstrates the basic methods of building such a network using these neurons, and has allowed some degree of investigation into the hardware requirements for a control system capable of allowing the parameters of the neurons to be adjusted in real time.

The network overall required 7259 LEs, of which 5641 were used by the 16 neurons, averaging 352.56 LEs per neuron. 14 of the neurons used 352 LEs each, one used 354 and one used 359, though it is not clear why this difference should occur. The difference was in the number of LUT-only LEs, with all neurons having the same number of registers, and it is likely that the variation occurred as parts of the neuron logic were merged with other blocks of circuitry.

The processor and its supporting circuitry made up the remaining 1618 LEs in the device, and it is likely that this figure would increase slightly as the number of neurons increases, as the readback port is implemented by a multiplexer which would clearly require more logic cells if more inputs were required. This increase shouldn't be significant compared with the number of LEs required by each new neuron.

The target device, with 11,520 logic elements, could hold a network of around 28 neurons with controlling hardware of this size. It is likely however that a network in which the neurons' parameters can be controlled in this way would be used with a controlling PC, and thus the controlling hardware would consist of some kind of PC interface whose job is simply to receive data from the PC and pass it on to the neurons' control buses. The display and keyboard interface, though useful in a standalone system like this, would not be required if the controlling logic was simply an interface to a host PC.

5.9: A More Flexible Neuron Model

While the simple four-input neuron model covers the basics of neural functionality, it does suffer from a few limitations. The major limitation is the fixed set of four inputs, if fewer than four are required the neuron is overcomplicated, while if more than four are required the system must be rebuilt accordingly.

If one or more input is not required when the neural network is built, it can be tied to ground and the compiler should optimise away much of the logic associated with it, but the state machine will still perform the extra integration states. This may not be a bad thing, as the rate of decay (in membrane potential units per clock cycle) programmed by a particular decay slope value will change if the number of states in the main loop is changed. A change in the main loop is unavoidable if more inputs are added, as these will each have to have their own integration states, in addition to the extra spike catchers and multiplexer inputs.

A second neuron model was built to address some of these issues, the intention being to produce a more accurate model capable of more complex neural dynamics, while retaining some degree of compatibility with the first design. In order to build more flexibility into the neuron, a two-part design was used, with the synapses being separate from the neuron body. The outputs from the synapses are summed and fed to a single input on the neuron, which behaves as a leaky integrator with a threshold triggered firing function. The general layout of the design is shown in Figure 60.

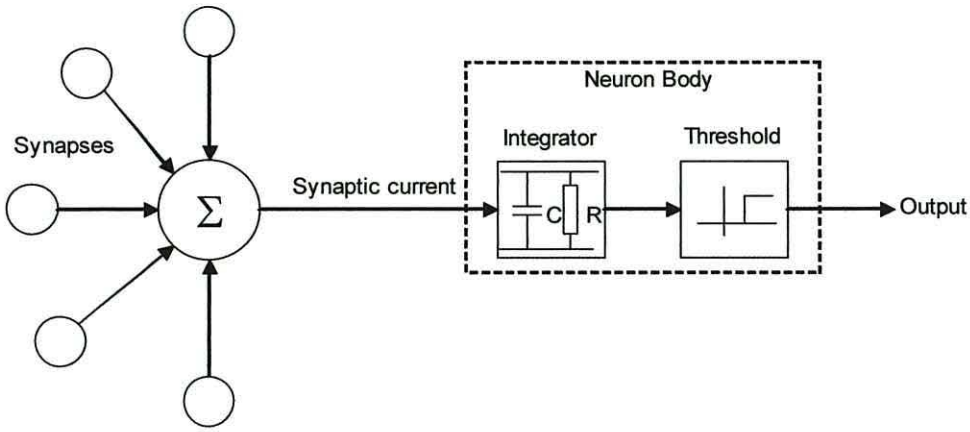

Figure 60: Overview of second neuron model

5.9.1: Neuron Body

The core of the neuron provides a simplified simulation of the processes which take place at the axon hillock. The synaptic current, summed over all the synapses, arrives here and is integrated over time by a method similar to that used in the first neuron design. The adder used in the integration has a form of bounds-checking, so that in the event that its output is less than the resting potential, it outputs the resting potential instead, addressing one of the problems found earlier with the first design. With this modification to the adder a mostly inhibitory input from the synapses will not push the membrane potential below the resting potential, but an inhibitory input coinciding with an excitory input will reduce or suppress the effect on the membrane potential.

A major change to the operation of this neuron is that the decay and refractory period slopes are no longer programmed in terms of the step change per cycle, but instead as the number of cycles required for a step change of one unit. This allows a much slower decay to be programmed, and also removes the requirement for the resting level comparator to have a wide detection window, as with a change of one unit each time it is guaranteed that the membrane potential will reach exactly the resting level without going past it.

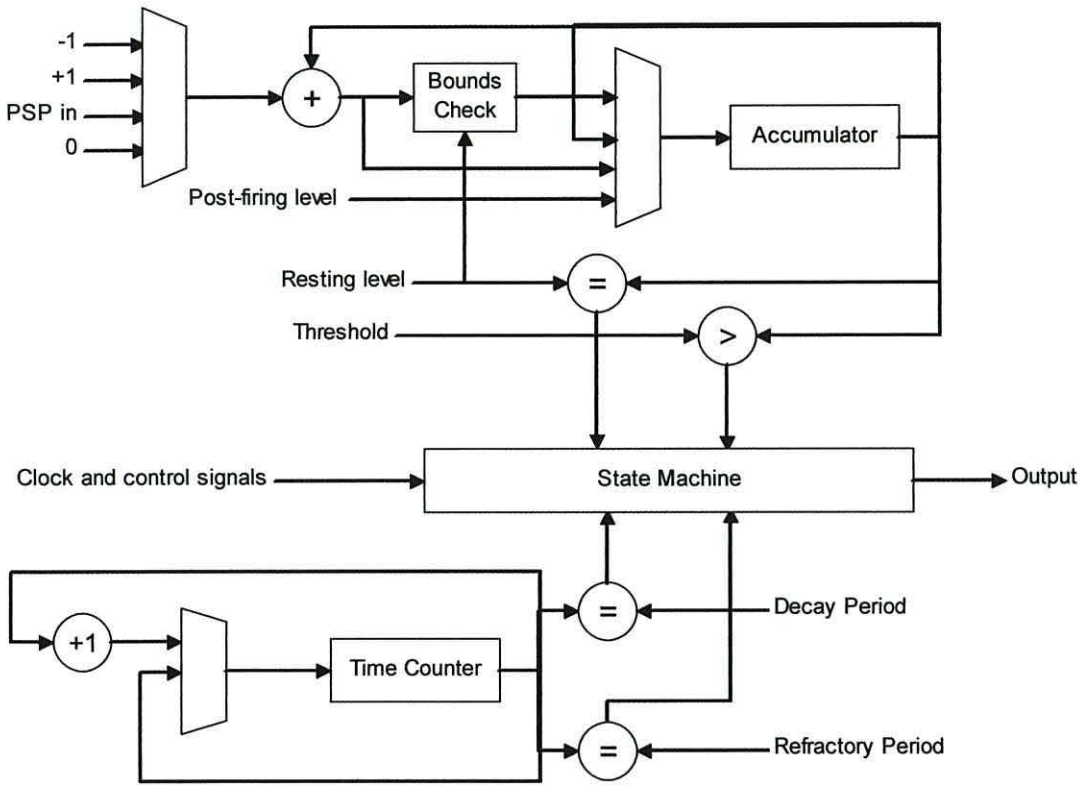

Figure 61: Block diagram of second neuron model

Figure 61 shows a block diagram of the neuron model. This model is simpler than the first version, since there is only one input and the spike handling is done by the synapses. A state machine, clocked on the rising edge of the input clock, provides the control signals for three multiplexers, and receives input from a number of comparators in order to control the flow of states. The two registers, the accumulator and the time counter, are clocked on the negative edge of the clock and are clocked on every cycle regardless of the operation being performed.

The accumulator is used, as before, to hold the simulated membrane potential, and is accompanied by an adder and a series of multiplexers. The input multiplexer selects a value to be added to the accumulator, either the incoming PSP, or the slope values for the decay and refractory period (-1, +1 respectively).

The output from the adder feeds a bounds-checking block which ensures that the new input to the multiplexer cannot be lower than the resting potential. This can be bypassed by the accumulator's input multiplexer to handle the refractory period correctly, where the membrane potential is expected to be lower than the

resting potential. This second multiplexer also allows the accumulator to be updated with the post-firing potential, or to be left unchanged.

Two comparators are used to signal to the state machine that the membrane potential is at the resting potential or has exceeded the threshold.

The time counter is a feature not present in the first neuron design, which counts state machine loop cycles, incrementing during any clock cycle in which its input multiplexer is set accordingly. During the refractory period, this is used to provide a delay between each increment of the accumulator, while during normal neuron operation it is used to ensure that the decay is performed once per N cycles of the main state machine loop, where N can be set along with the normal neuron parameters. A larger value of N results in a slower decay.

The same counter is used for both functions because the two functions can never overlap, as normal neural functions are suspended during the refractory period. This results in a saving in hardware.

With just a single input to this neuron, the state machine required to control it is simpler than for the earlier design. The state transition diagram is shown in Figure 62.

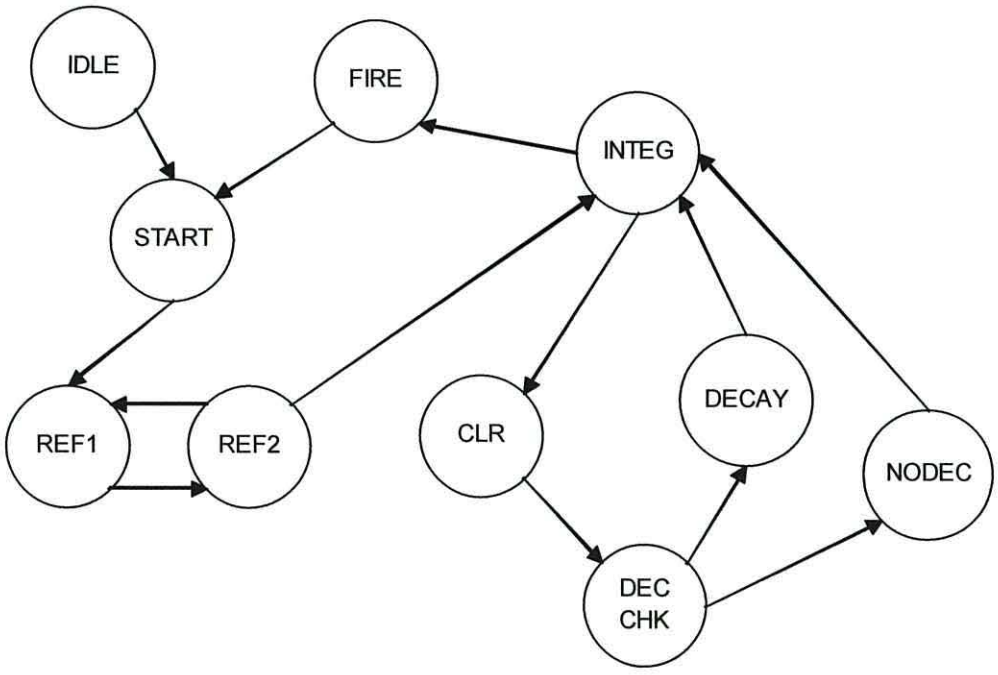

Figure 62: State transition diagram for the second neuron model

As with earlier designs, after a reset signal is applied the neuron waits in an idle state until triggered by the GO signal. Waiting in this state allows the neuron's parameter data to be loaded without causing spurious neural outputs, and allows all neurons in a network to be started simultaneously with correct configurations.

Once activated, the system passes through the START state, which loads the post-firing potential into the accumulator and clears the time counter, preparing the neuron for the refractory period.

State REF1 activates the time counter, so that it advances one count per clock cycle. However, the state machine will remain in this state until the refractory period comparator indicates that the programmed delay has elapsed, at which point the state machine proceeds to state REF2. Here, the accumulator is incremented, and the synapse control outputs SYNC and SYN_CLR are pulsed. If the accumulator matches the resting potential, the refractory period is over and the neuron can proceed with processing, otherwise the state machine returns to state REF1.

Under normal neural processing, the state machine runs a four state loop consisting of states INTEG, CLR, DEC_CHK, and either DECAY or NODEC, depending on the time counter's output. When in the INTEG (integrate) state, the incoming data is added to the accumulator, using the bounds-checked output of the adder, and a decision is made to proceed to either CLR if the neuron is not to fire, or FIRE if the threshold has been exceeded.

The CLR state is used to generate a pulse on the SYNC output, which signals to the synapses that the PSP has been read. Once this has been done, the DEC_CHK state is used to increment and check the time counter. If the decay period comparator indicates that enough cycles have elapsed, the next state will be the DECAY state, which decrements the accumulator and clears the time counter. Otherwise, the NODEC state is executed in place of DECAY, to ensure that the main loop always consists of four cycles.

When the neuron fires, the state machine passes through the FIRE state, which generates an output pulse, then back to START to begin the refractory period. Only a reset signal will cause the neuron to return to the idle state.

5.9.2: Synapse Control

Two control signals are produced by the neuron, SYNC and SYN_CLR. The SYNC signal outputs a pulse for each integration cycle, telling the synapse that its output has been read. The SYN_CLR signal pulses continuously during the refractory period, and can be used to clear the synapses so that incoming spikes are ignored during this time. For maximum flexibility, the SYNC output also pulses during the refractory period, so it is possible to continue with normal synapse operation during this period if it is required.

5.10: Synapse Design

The synapses were designed to replicate the synaptic functionality described in [5.5], and inspired by previous work in spiking neuron modelling. [5.45, 5.46]

5.10.1: Simple Synapse

The simple synapse replicates the function of the inputs on the earlier neuron design. Incoming spikes are captured by a rising-edge triggered flip-flop, and the synapse outputs either zero or its weight value, until cleared by a reset input. Referring back to the neuron's synapse control outputs, this reset input would be connected to the SYNC signal rather than SYN_CLR, as the latter is only active during the refractory period. The SYNC signal pulses once for each integration cycle, ensuring that a synapse will provide an output for a single cycle only.

The structure of the simple synapse is shown in Figure 63.

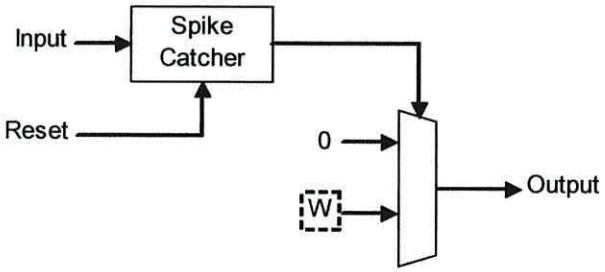

Figure 63: Logic structure for the simple synapse design

A problem encountered with this synapse design was that the synapses attached to a neuron will be reset every time the neuron passes through the CLR state, and so if a spike from another neuron arrives during this cycle it will be ignored. Assuming that the neurons in a network will have different weight and threshold parameters, it is inevitable that even though all neurons will start at the same time, triggered by the GO signal, they will not remain synchronised for long and occasionally a signal will be received by a synapse at the same time as the clear signal arrives. Initially the neuron was modified to issue the synapse reset signal only if the incoming PSP was not zero, so the synapses would only be cleared when necessary. However, this does not achieve the desired result, because all

synapses are cleared by the same signal and therefore if a single synapse is active, the others will also be cleared. This also causes a problem if an inhibitory synapse is triggered at the same time as an excitory one, and their weights sum to zero. In this case the neuron will not detect the incoming synaptic currents, and subsequently not issue the clear signal. The synapses will fail to respond to any subsequent input until another synapse is triggered and the sum of the PSPs becomes non-zero.

In addition to this problem, this simplified 'one-shot' synapse design provides only a very basic simulation of the function of a synapse. Since the synapse is an electrochemical device which operates by controlling the flow of ions through the cell membrane, it will generally not deliver its output in the form of a single short pulse causing a step change in membrane potential, but instead as a longer pulse, which may, if conditions are correct, cause the neuron to produce an action potential after a short delay.

In order to address these issues, a second, more complex synapse was designed.

5.10.2: More Complex Synapse

The second synapse design allows full control of both the amplitude and duration of the post-synaptic response. While the first synapse design produces a one-cycle output pulse of amplitude A, this new design produces a stretched pulse of amplitude W over a period of L cycles. Ignoring the decay function in the neuron body, provided that $A = WL$, the effect on the neuron should be the same, but may be delayed depending on the value of L. However, it will be necessary to include some consideration of the decay function whenever $L > 1$, as an elongated output pulse increases the chance of a decay cycle being executed during the PSP. If L=1 and W=A the synapse produces the same output as the simpler version.

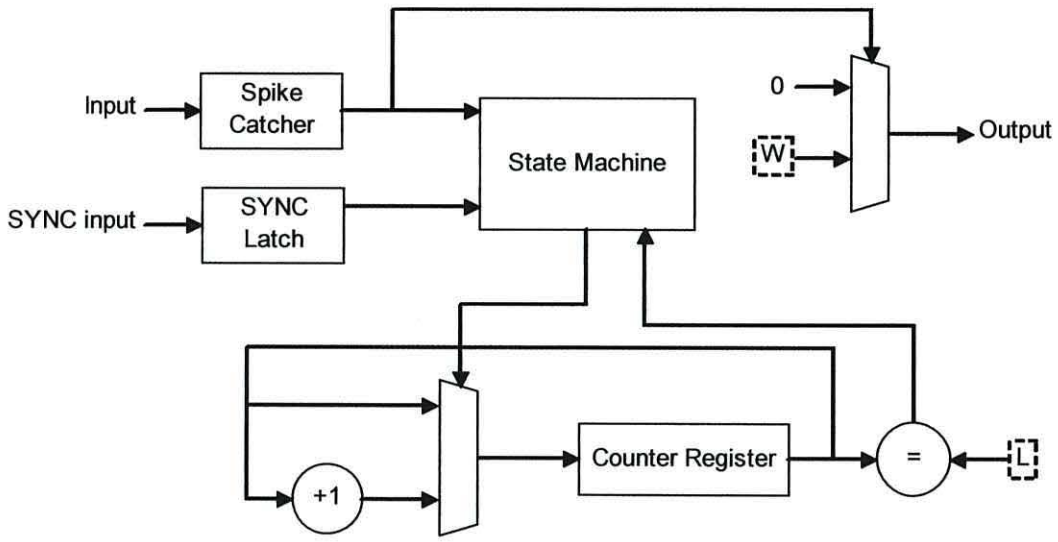

Figure 64: Logic diagram of a more complex synapse

The logic structure of this new synapse design is shown in Figure 64. The spike catcher and output multiplexer from the original synapse are still present, with the addition of a synchronous counter to count the neuron's integration cycles and a simple state machine to control it. A duplicate of the spike catcher is used to catch the SYNC pulses, and hold them until the state machine has read them. Once read, the pulse is cleared, thus ensuring that the state machine does not inadvertently perform two counter increment cycles due to a slow sync pulse remaining active for too long.

It was discovered during testing of the neuron and synapse together that under some rare circumstances an incoming spike could coincide with the SYNC signal from the neuron in such a way that the synapse would clear its output just before the neuron performed an integration cycle. To correct this problem, the spike catcher was modified to consist of two stages, the first being a standard D-type flip-flop clocked by the rising edge of the spike, and the second being an extra flip-flop clocked by the falling edge of the SYNC input, through which the first flip-flop's output is fed. The spike catcher's output is therefore sampled on the falling edge of the SYNC pulses, delaying the synapse's response slightly but hence ensuring that the state machine does not clear the output signal too soon.

The counter is built with synchronous logic, with a multiplexer to ensure that it is only updated when required.

The two parameters, W and L, are supplied by a pair of registers which allow these parameters to be updated at run-time by external hardware.

The state machine controlling the synapse is very simple, consisting of four states as shown in Figure 65. When a spike is received, the state machine enters a loop which increments the counter register once for each synchronising pulse received from the neuron, and then returns to a waiting state, resetting the spike catcher in the process. Since the output of the spike catcher controls the output multiplexer, the synapse will output its weight until the counter value reaches the L parameter value.

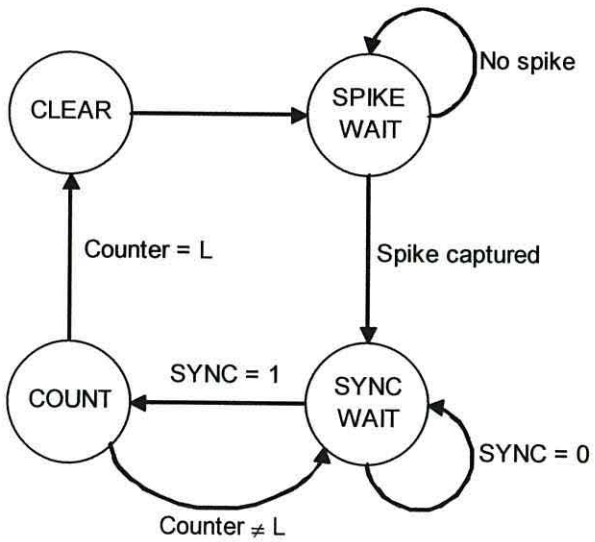

Figure 65: State diagram for the synapse control state machine

With the synapses separated from the neuron, it is possible to adapt them individually to make the most efficient use of the hardware. The initial designs for the synapses and the neuron itself use 16-bit words for all parameters, but it is possible to use larger or smaller words if the application requires them. If a particular synapse in an evolved network is found to use values for W or L which can be represented in fewer than 16 bits, it can be replaced with a synapse design with narrower buses and registers, saving a handful of logic cells. Table 14 shows the logic cell requirements for the synapse, compiled for Altera's Apex 20KE FPGA series, with L and W ranging from 16 bits down to 10 bits. Both parameters were the same width in all cases.

Width of L and W	LEs
16	84
15	80
14	75
13	70
12	66
11	61
10	56

Table 14: Logic element usage for different parameter widths

5.11: A RAM-Based Neuron Design

While some of the more advanced FPGAs include embedded hardware blocks designed to enable fast DSPs to be built efficiently, these blocks usually consist of multipliers or multiplier-accumulator units, intended for rapid implementation of digital filters. No multipliers are required by the current neuron designs, as the only functions used are addition, subtraction and Boolean logic. The output of the synapse can be thought of as a kind of multiplication, where the output is the weight multiplied by the synapse activity. However, as the synapse activity is either 1 or 0 at present, the multiplication is implemented with a multiplexer.
The other type of embedded hardware block present in most modern FPGAs, whether or not embedded DSP blocks are present, is the embedded memory block. These are useful when large amounts of memory are needed, and it is not practical or desirable to produce these using the logic elements. In this section a neuron design is presented which makes use of these embedded memory blocks to reduce its logic cell count compared with the previous model.

Using flip-flop registers to store the neuron's parameters has both advantages and disadvantages. The advantages of such a design include the ability to read all parameters simultaneously and without interruption, and also an improvement in portability, since all VHDL compilers can handle inferred latches regardless of the target technology. The major disadvantage is the high logic cell count of such a design when used in an FPGA.
This disadvantage can be reduced by replacing the registers with a single block of RAM, taking advantage of the FPGA's internal RAM cells. The Apex 20K series FPGAs contain a number of Embedded System Blocks (ESBs) which contain memory cells and can be configured as simple RAM or ROM, or more complex functions such as FIFO buffers. The 20K300E device used for the tests contains 72 ESBs, each of which has 2048 bits of memory, which can be configured as 2048 x 1, 1024 x 2, 512 x 4, 256 x 8 or 128 x 16 bit memory arrays. The neuron was redesigned to use one of these blocks instead of logic-cell registers for parameter storage. Figure 66 shows a block diagram of this new design.

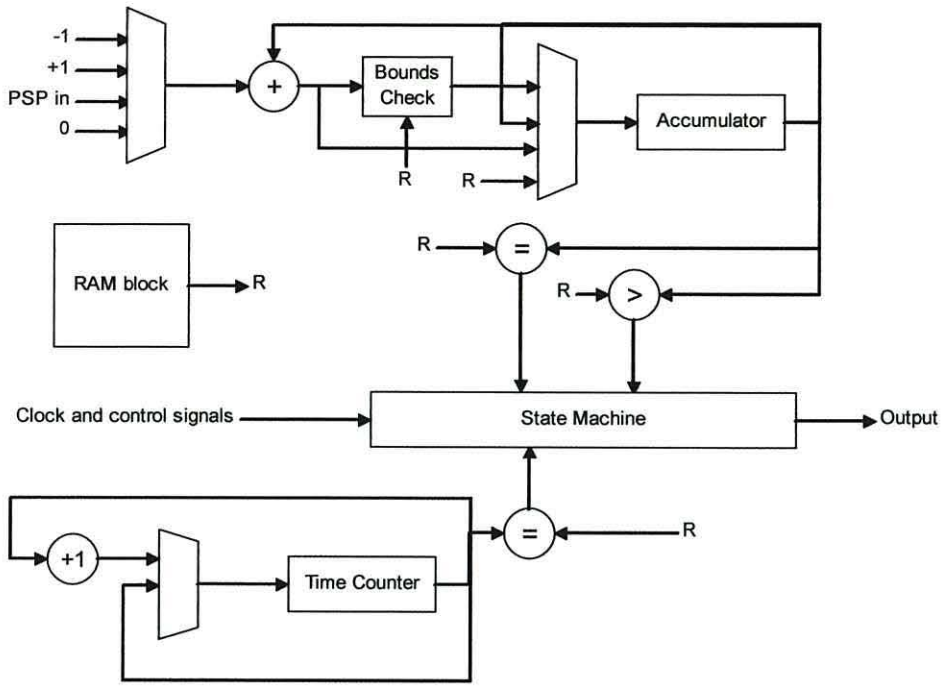

Figure 66: Block diagram of the neuron model using RAM instead of registers

It can be seen from the figure that the major change is that the parameters are replaced by a single signal, represented by R. This is the RAM's output data bus, and a series of additions to the state machine are used to produce an address which selects the parameter required at each state. There is also now only one comparator circuit in the timing block, as with the parameters being fed from a single source, the two comparators in the earlier design became identical.

This design change also necessitated a small change in the operation of the neuron, as there is one state in the state machine for the previous neuron where two parameters are used simultaneously. The state machine was modified to check that the membrane potential hadn't exceeded the threshold during the CLR state, rather than the INTEG state as was the case with the previous design. This was necessary because with RAM, as described above, only a single parameter can be accessed at any point in time, and the resting potential is required during the INTEG state to ensure that the bounds checking in the input adder works correctly.

The RAM block implemented for the neuron is oversized, by necessity, as five parameters are required. This leads to a requirement for the parameter address but to be 3 bits wide, resulting in an 8-word memory. There are therefore three unused

words of memory, and these are simply ignored by the hardware. In fact, the memory usage is less efficient still, as the ESB can implement only a single memory at any time. With 16-bit parameters, 80 bits are required in total, and 128 bits are implemented, leaving 1920 of the 2048 bits unusable. It is possible that a future neuron design in which the synapses are incorporated back into the neuron would be able to use this extra memory for synapse weights, though the problems of multiplexing the various weights and other parameters as they are required may overcome the saving in logic cells from the use of the memory block rather than registers. However, some later FPGA types, notably the Stratix II type from Altera, have smaller memory blocks of 512 bits each, in larger quantities, and these could be used to make a more efficient use of the resources available.

In order to retain compatibility with the previous, register-based neuron design, the RAM was implemented as dual-ported RAM, which in the Apex 20K series is not true dual-ported RAM with two completely independent bi-directional ports, but is a type of dual port RAM where one port is read-only and the other write-only. The write-only port resembles the register inputs as far as any controlling system is concerned, and the dual-ported operation allows the parameters to be updated in real-time without interrupting the operation of the neuron.

Replacing the registers with RAM in this way reduced the neuron's logic element usage from 267 LEs to 157 LEs, a saving of 110 elements, or some 40%. It is clear that 80 of these LEs were saved by removing the 80 bits of storage, and a further 16 were saved by removing the redundant comparator in the timing circuit. The remainder are likely to have been those elements responsible for providing the input logic for the registers, a complex demultiplexer which allowed asynchronous updating of the parameters.

We can see, therefore, that making use of embedded features such as these can be useful in reducing the footprint of the neuron model. However, in the case of the synapses, it would not necessarily be a good idea to replace the registers with RAM, as there are a limited number of ESBs available, 72 in the EP20K300E device used for the development of the neurons. Assigning one of these per neuron is feasible, but in most cases there will be many times more synapses than neurons, so this would place a major restriction on the size of the network. Taking the example of the network cell discussed in the next sections, nine synapses are

used per neuron, which means that with ten memory blocks required in total for the cell, just seven cells could be implemented in the chip.

5.12: Analysis of Designs & Conclusion

The neuron model, like many logic designs, relies heavily on registers for storage of parameters and working values. These registers are produced in the VHDL code by inferred latches or flip-flops, and are implemented in the FPGA using the flip-flops present in the logic elements. As can be seen from Table 14, the logic element usage for the synapse design decreases as the bit width of the parameters decreases. The same is true for any design, as an N-bit wide register will require N logic elements, since there is only one flip-flop available in each logic element. In cases such as the accumulator in the neuron, the circuitry preceding the register can often be partially built into the same LEs, since most FPGA logic elements consist of some form of combinatorial logic feeding the flip-flop, and this explains the variable logic element usage found when testing different arrangements of neuron and synapse. Used alone, there will be no logic preceding each unit's parameter registers, but when used as part of a larger system, there may be such logic present, which will result in a merging of units, and a slight variation in logic element usage. The element usage will tend to increase in steps as the arithmetic pathway width increases, due to the limited number of inputs to each LE. As an example, VHDL 'IF' statements such as `IF X="00000000"` are widely used in the neuron designs. The above statement will clearly generate a block of logic with eight inputs, which, assuming there is no merging of this function with surrounding circuitry, will require at least two logic elements. Using the Cascade Chains built in to the LEs allows the construction of this block without using extra cells to join the input cells together, and although there is some reduction in speed associated with these chains, in the same way as the ripple carry adder suffers from a speed reduction over more advanced types, the extra delay will generally be smaller than if several layers of logic elements were used, as the cascade signals do not pass through the signal routing channels which connect the logic cells. In the Altera Apex series FPGAs, up to ten logic elements can be chained efficiently in this way.

We can therefore see that the minimum logic element usage of a functional element such as a zero-detector or an adder will equal the bit-width of the element rounded up to the next multiple of four. The maximum usage of such a block will be one cell per bit, if its outputs feed a register, due to the previously discussed distribution of flip-flops in the FPGA. In practice, due to compiler optimisations which may have a cascading effect leading back from a seemingly unrelated register, the logic cell usage of any particular section of a large, complex logic circuit is often hard to determine before it is compiled. For example it was found in various tests involving not only the first neuron design but also the processor of chapter 4 that logic cell usage changed, sometimes significantly, when debugging outputs were fitted to the block to allow the internal registers to be read by external hardware. It appears that the compiler was merging the registers with other hardware when they were truly embedded, but had to implement them as register-only LEs with the debugging outputs present.
Ultimately, it is the logic cell usage which determines the number of neurons which can fit into an FPGA. The FPGA used for most of the tests was an Altera Apex 20KE device, with 11,520 logic elements (LEs), so the original neuron design, using 400 cells, could be duplicated 28 times in one device.

Table 15 shows the logic cell requirements and maximum operating frequency (as given by the Quartus compiler/fitter) for the neuron and synapse designs, along with parameters by which their performance can be compared. From this table it can be seen that a neuron with four inputs, similar to the early design, would require at least 509 LEs, excluding the circuitry required to sum the synapse outputs and to load the parameters. This reduces the number of neurons which can be implemented to 22 or fewer.

A test network cell, consisting of a neuron body, nine synapses and the required control logic, was built to test the logic cell requirements. 1150 LEs were required, roughly a tenth of the capacity of the chip.

Device	LEs	Fmax (MHz)	Clk/cycle	Int (us)	Fmax / LE	Int^{-1} / LE
First neuron design	400	39.55	15	0.38	0.10	0.01
Second neuron design	267	36.53	4	0.11	0.14	0.03
RAM-based design	173	29.58	4	0.14	0.17	0.04
Synapse	84	123.82				
RAM-based 4-i/p	509	29.58				
RAM-based 9-i/p	1150	29.58				

Table 15: Logic Element usage and operating speed of the neural elements

The extra logic required for the network cell consists of an adder tree to connect the nine synapses to the neuron, and an address decoder to allow the synapses and the neuron to share the same data and address buses, allowing the cell to appear as a single block of memory to the host processor. The adder tree for 9 inputs was defined in the simplest possible way, allowing the compiler to determine the architecture. This required 128 logic cells when compiled alone, but in the completed test network the logic evidently merged with the surrounding circuitry, as the typical LE count was 112. The address decoder logic took up two of the network cell's LEs.

We can see that a network cell requiring a tenth of the host device's logic capacity will restrict the size of a network to just ten cells. In fact, if additional logic is required for control or I/O purposes the network size will be restricted further. Methods of using the FPGA's resources more efficiently are therefore desirable if large networks are to be implemented in parallel.

The reduction in maximum operating frequency from the first design to the second is interesting, and hard to explain. It is likely that the slight reduction from the original design to the second design is due to the addition of the bounds-checking in the accumulator update loop, though the addition of the time counter will play a part as it adds more logic with which the state machine must interact during the operation of the neuron. The subsequent larger reduction when the neuron was rebuilt with embedded RAM may be due to the additional interconnects necessary to connect the RAM block to the logic, as this is the only part that has changed significantly.

In any case, the operating frequency is well beyond that which is required for biologically plausible firing rates, as is shown in section 5.17.1, where it is found that firing rates of around 2KHz can be obtained with a clock rate of 500KHz. It would be likely though that a higher clock frequency would be used, as this would

decrease latency, making the neuron respond more quickly to its inputs. Figure 51, in section 5.7.2 shows a reason for wanting to do this, as it can be seen that the membrane potential lingered above the threshold for a short time before the neuron fired, as it took several clock cycles for the state machine to reach the threshold checking state. The second neuron design would have less of a problem here, as it performs fewer operations per integration cycle.

The number of clock cycles required to perform a single integration cycle, and the resultant time required to do this if operating at the maximum operating frequency, are also given in Table 15, along with the area-time products, both in terms of clock speed against LE usage and number of integrations per second against LE usage. It can be seen that the RAM-based design, despite having the lowest operating frequency, scores highest on both counts, due to its small hardware size. The first neuron design scores poorly because of its large LE requirement, and also because of its relatively long integration cycle time.

Comparing these neuron designs with other published designs yields mixed results. There are few other published designs with which a detailed comparison can be made, as most of the designs are either substantially more complex or are designed for time-multiplexed operation with the neuron acting as a custom processor. Comparing with the design in [5.38][5.39], where a 30 input neuron without learning uses 23 slices of a Xilinx XC2S200 device (equivalent to around 46 LEs in an Apex device), we can see that the neuron designs presented in this chapter use much more hardware than this design, though this alternative design uses 9-bit data rather than 16-bit, holds the synapse weights in a RAM block, and uses hard-coded threshold, resting potential and post-firing potential values. It also requires 31 clock cycles to perform a time-step, and would require 5 cycles in the case of a 4-input version. The first neuron design presented in this chapter performs an integration cycle over four inputs in 15 clock cycles, while the second design requires 4 cycles regardless of the number of inputs. This means that the second neuron design presented in this thesis will have the advantage of a faster execution cycle when larger numbers of inputs are used. Furthermore the second neuron design presented in this thesis will operate at the same speed regardless of the number of inputs, and thus all neurons in a network will operate at the same

speed even if some have more synapses attached than others. It is also capable of performing more complex post-synaptic functions (such as the 'slow' PSP functions and the synapse-less operation) than the neuron described in [5.38], though clearly at the expense of a significantly higher resource usage. It is clear however that while some of this increase in resource usage stems from the wider data bus and accumulator, a good deal of the extra resources are used by the registers holding the parameters and the general purpose comparators used to perform bounds checking and threshold checking, as no compiler optimisations can be applied to these based on the number against which they are checking, since this is not known at compile-time.

The neural network presented by Roggen et al. [5.36] uses neuron models which are simplified in the same manner, but much more so, with fewer bits to represent the membrane potential, hard-coded weights and threshold, and a simple refractory period of one cycle. The neuron is quoted as using 109 LEs of a similar FPGA when compiled standalone, or 90 LEs when compiled as part of the network, showing a variation due to compiler optimisations as described earlier in this section. The quoted f_{max} is 42MHz, with n+1 clock cycles required for an integration cycle over n inputs. Thus the area/time products can be calculated as for the new designs presented in this thesis. In terms of Fmax / LE usage, the score is 0.39, while the score in terms of integration cycles is 0.08 for 4 inputs, or 0.01 for the 26 input version described.

For 4 inputs, both of the scores are higher than the highest score achieved by either the first or second neuron designs presented here, due to the higher clock speed and much smaller size of Roggen's design. However, with 26 inputs the neuron scores lower when integration cycle speed is considered, as the number of clock cycles per integration cycle increases. By contrast the second of the designs presented in this chapter would achieve the same score regardless of the number of inputs.

This neuron is much simpler than the designs presented in this chapter, mainly due to its smaller bus width, but also due to its hard-coded parameters, which remove many of the logic cells used as parameter memory. The result of these simplifications is clearly a loss of flexibility, as without adjustable parameters the response of the neurons cannot be altered at run-time. In the implementation described only the connections between the neurons can be reconfigured at run-

time. As with the implementation in [5.38], the number of clock cycles required to perform one time step depends on the number of inputs to each neuron, as the input summation and the decay function are time-multiplexed. Both of these previous implementations demonstrate that this method saves hardware, as both are significantly smaller than the implementations presented in this chapter, though this method does also require that all neurons in the network are synchronised and processing is carried out in steps, whereas the new designs presented here are capable of working in real-time and completely unsynchronised, integrating all their inputs together without any extra clock cycles required as the number of inputs increases. The constant number of cycles required to perform one integration in the second of these new designs, 4 compared with 27 or 31 for the designs described above means that although it has a lower maximum operating frequency, it can perform more integrations per second with a similar clock rate. The capability for unsynchronised operation brings the neuron model a step closer to an analogue implementation, which in turn is more biologically-realistic.

It is clear, therefore, that the neuron models presented in this chapter compare favourably with other published designs in some ways, such as their flexibility and efficiency. The more complete second design and its RAM-based follow-up are capable of more than simple spiking operation, depending on how the synapses are programmed. They can also be used without synapses for additional, potentially useful modes of operation. It is also apparent that although the maximum operating speeds may be lower in these designs, they take fewer clock cycles to perform an integration cycle than designs which integrate their inputs in turn, and could therefore operate faster when used at the same clock speed.
This efficiency and flexibility comes at the cost of greatly increased hardware usage, however. Much of this can be attributed to the extra hardware required to implement the extra functionality, and some of it can be attributed to the increased width of the accumulator and arithmetic hardware.

5.13: Learning Considerations

The process by which a network can be trained was not investigated during this project, but would be a major future development. It is presently possible to adjust the parameters of the neural models in real-time through the control signals and data buses, so it will theoretically be possible for a learning system to be attached to the network without requiring any changes to be made to the neurons. The learning system would need access to the signals passing between the neurons, in order to determine which connections need to be strengthened or weakened by looking at the neurons' responses to their inputs.

A common form of learning in spiking neural networks is Spike-Timing Dependant Plasticity [5.47][5.5], a correlation-based (Hebbian) learning system in which a synapse's response is adjusted if there is a strong correlation between the activity of the presynaptic and postsynaptic neurons. The training patterns, input and output, are applied to the network so that the output neurons are forced to fire in the desired output pattern, and the synaptic weights are strengthened where both the pre- and post-synaptic neurons fire. This type of learning is the most common type used with implementations of spiking neurons, as its time-dependence suits the time-dependant response of this type of neuron.

There is usually a time window function, so that if the two neurons fire within a certain time of each other they are said to be firing together. If the presynaptic event occurs just before the postsynaptic event it is considered to have contributed to the firing and that particular synaptic connection is strengthened, and if the postsynaptic event leads the input it is considered to have had no effect and so the strength is reduced. [5.48][5.25] It is possible that a processor system attached to the network could read the activity of the neurons and make these adjustments to the weights in real-time. This is presently the only method by which learning can be implemented with these new neuron models, and any system which requires more low-level control of the synapses will require that the synapse model be rebuilt to incorporate the learning hardware. Such learning systems have been implemented in previous designs [5.38] but not in cases where the synapses are modelled separately from the neuron body. This area shows one of the advantages

of having all synapses built-in to the neuron and handled in turn, as in addition to enabling the same hardware to be used for all synapses, the same learning hardware can be used for all synapses. With the synapses modelled separately, as with the second of the designs presented in this chapter, there would have to be one 'learning unit' per synapse, resulting in a considerable increase in hardware even if the learning unit is not too complex.

The learning unit itself will need the ability to modify the weight, and would therefore need a simple ALU, with just add and subtract functions. It would also need some kind of phase detector to detect the relative timing of the spikes, and a window generator – probably based on a counter – which enables the weight change only if the spikes are close enough in time. It would also have to have some method of disabling it, so that learning can be performed only when required.

From these requirements only a few speculative values regarding the logic cell usage can be derived. It is clear that the ALU will require at least as many logic cells as there are bits in the weight register, though as the ALU is purely combinatorial and the weight register is purely a register, it is possible that the compiler might merge some or all of the logic elements. The window counter's LE requirement will depend on how many bits are required, and the rest of the logic can't be estimated until it is designed.

A potential issue with this type of learning system is that the second of the designs in this chapter is capable of implementing both fast and slow PSPs, and as such has two parameters stored in the synapse's registers, both of which contribute to the overall effect on the membrane potential. The STDP-based learning would work for adjustments to the magnitude of the PSP, but not for changes to its length, as lengthening the PSP can make the neuron fire just as increasing the magnitude will, but will add a delay which would result in the neuron firing outside the time-window used in the learning process. The published designs which implement STDP with spiking neurons in FPGAs generally do not have the capability to modify the length of the PSP, so it is difficult to say how much more complex this would make the learning system.

This method would potentially have the advantage of allowing the neurons to learn by themselves, adjusting their own synaptic weights without requiring an external controller. It would also increase the size of each neuron and thus decrease the number of neurons which could fit into an FPGA. Assuming that the software for a controller in an embedded processor performs the same operations on each neuron and can therefore cope with any number of them without a significant increase in code size, the embedded processor option would be much more efficient in hardware usage for larger numbers of neurons. It would however have the disadvantage of slowing down as the number of neurons grows, whereas fitting the learning systems to each neuron individually would allow them to operate at the same speed regardless of the number of neurons present.

Alternatively, it is possible to train the network off-line, in software simulation, and then load the parameters into the control registers when training is complete. In this case, however, the neuron models do not actually need to be built with RAM-like registers, as the parameters can be hard-coded into the VHDL and the network recompiled. This would almost certainly result in a reduction in the logic cell usage of each neuron, and would also result in a wide variation of logic cell usage from one to the next, as the compiler may be able to optimise the hardware differently depending on the bit patterns of the parameters. In addition any synapses which are not required could be removed from the VHDL rather than simply programmed into an inactive state.

Implementation of learning algorithms was not a part of this project, but would be a major part of any follow-up project work.

5.14: Usage of the New Neuron Model

In this section a series of examples are presented to demonstrate the set-up and operation of the second neuron model.

5.14.1: Loading Parameter Data

Since the neuron's parameters are held in registers or RAM, the first step when the network is initialised is to load these values through the data bus. A short '0' pulse on the reset line causes the neuron to enter a resting state, in which no processing occurs. The parameters are loaded as depicted in Figure 67, using the data and address buses and the CE and WR signals. These replicate the functions of the Chip Enable and Write pins on a static RAM device.

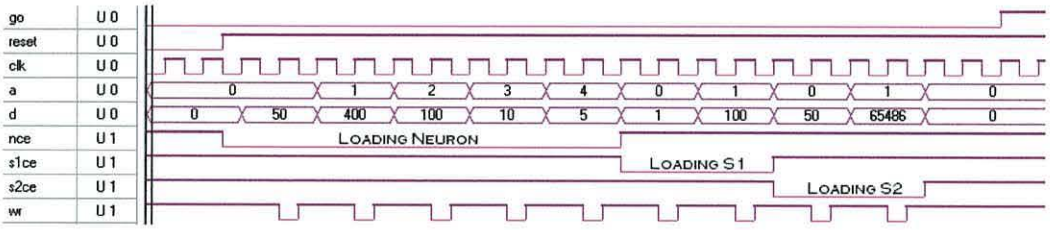

Figure 67: Quartus simulator view showing parameters being written to the registers

The waveform view shows the process of loading the data for a simple test system with one neuron and two synapses. The active-low reset pulse can be seen at the left of the traces, with the leading edge of the GO pulse at the far right.

The CE signal is set low to enable writing to the registers. This allows the use of a decoder to select one of the neurons in a network based on the upper bits of the address bus, though for a simple test system as depicted above, the three CE signals are brought out to individual pins. When WR goes low, the data will be written into the selected register or memory cell.

The values shown in the waveforms for the data bus ('d' in the figure) are as follows: The neuron is set with a post-firing potential of 50, a threshold of 400 and a resting level of 100. The timer settings for the decay and refractory period timers are 10 and 5 respectively.

The first synapse is set to have a weight of 100, delivered in a single cycle. The second synapse remains active for 50 cycles, and has a weight of -50. When a

negative weight W is required, the value written to the register should be $2^N + W$, where N is the bit-width of the neuron's accumulator and PSP input. For W = -50 and a 16-bit neuron the required value is 65,486.

5.14.2: Using the Neuron without Synapses

Here the use of a neuron without synapses will be examined, where it will be seen to act as a basic input encoder.

For normal neuron operation the neuron body will be accompanied by one or more synapses, but because these are not built in to the neuron it is possible to operate it without them and to feed an externally generated stimulation signal into the PSP input. If this is held constant, it will be added to the accumulator on every cycle of the main loop (4 clock cycles) and, if it is large enough to overcome the decay, will cause the neuron to fire after a certain number of cycles. Since the neuron integrates its input until the threshold is exceeded, it is clear that a larger number fed in to this input will trigger firing after fewer clock cycles than a smaller number, thus increasing the firing rate of the neuron. The neuron is therefore acting as a basic magnitude to frequency converter, and can be used to encode an input stimulus, perhaps from a sensor, as a spike train of varying frequency.

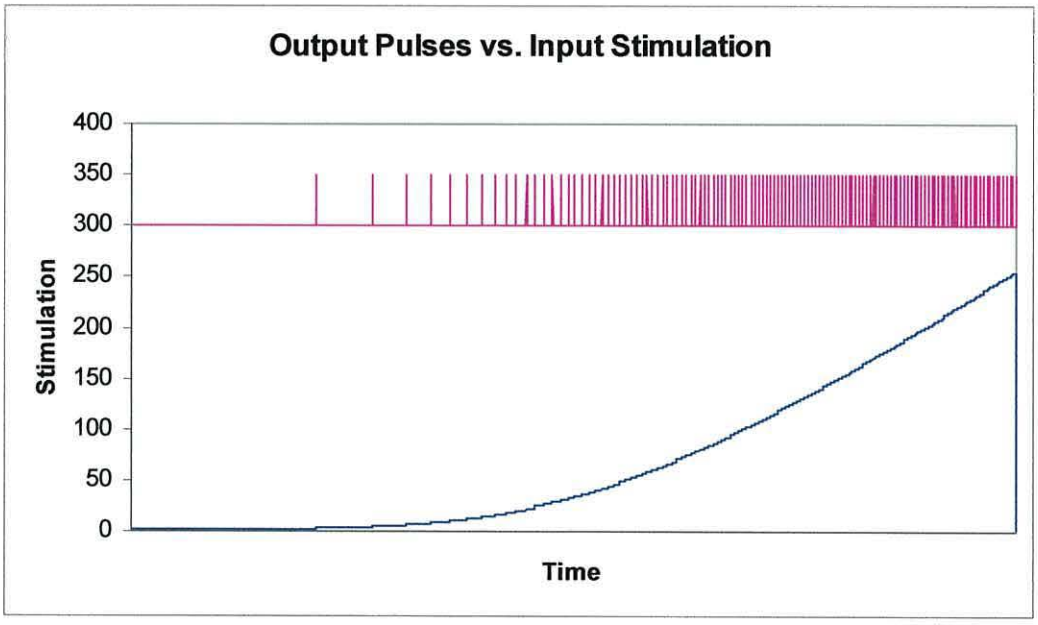

Figure 68: Output pulse train and stimulation for an input neuron

Figure 68 shows the neuron's response to a gradually increasing input value. The input word started at 1 and increased by 2 each time the neuron fired, allowing the number of clock cycles between firings to be measured and the firing frequency to be calculated. The curvature of the input signal is caused by its increments being synchronised with the gradually increasing frequency of the output pulses so that the input is incremented each time the neuron fires.

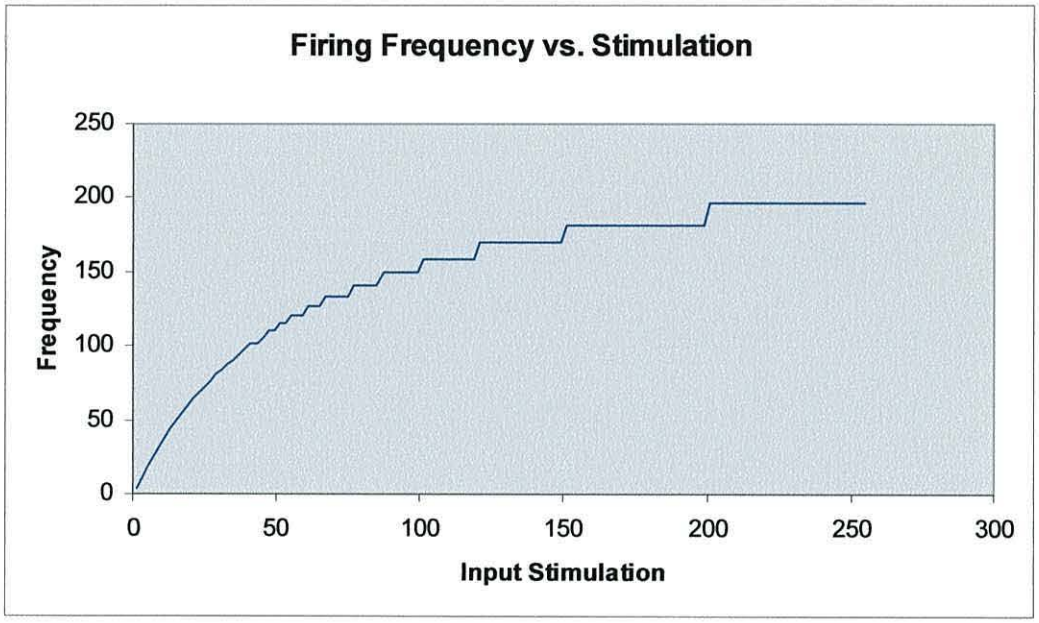

Figure 69: Neuron firing frequency against stimulation input

Figure 69 shows the form of the firing frequency change as the stimulation current (represented by the binary word supplied to the PSP input) changes. Since the magnitude of the input affects the firing period, the curve has a logarithmic form, as found from the analysis of the integrate-and-fire model in section 5.4.1. The form of this response is more important than the actual values, which are expressed in kilohertz in the figure. These values will depend on the settings of the threshold and resting potential, the decay timing, and the clock frequency supplied to the neuron, but the shape of the curve will be consistent for any configuration. This curve shape has been previously demonstrated in existing analogue neuron models [5.45].

An interesting point to note is the apparent quantisation of the frequency, and also the reduction in accuracy as the input word increases. This is due to the constant time interval between integration cycles in the neuron, and the fact that the neuron will fire in a particular cycle if the threshold has been exceeded at all, regardless of how far over the threshold the membrane potential has gone.

As an example, it can be seen from the graph that an increase in the frequency occurs at a stimulation value of around 150. As the input changes from 149 to 151, the frequency increases, but the change to 153 does not produce a change. This is because at an input of 149 the threshold was exceeded in a certain number, N, of integration cycles, where in cycle N-1 the membrane potential just failed to reach the threshold. However, when the input word increased to 151, the increase in potential on each cycle was enough to exceed the threshold in cycle N-1, reducing the firing period by one integration cycle. When the input increased to 153, however, the increase in the membrane potential on each cycle was enough to exceed the threshold by a slightly larger amount in N-1 cycles, but not quite enough to exceed it in N-2 cycles.

The resting level will usually be greater than zero, to allow for the refractory function, and the threshold will be lower than the maximum value which can be represented with the chosen accumulator width, to allow the membrane potential to exceed the threshold without exceeding the limits of the accumulator. Therefore, the difference between the two will be smaller than the maximum possible value of the input word. This implies that there will be a value of input word which, when added to the accumulator during at the start of the integration cycle, will cause it to exceed the threshold immediately. Once this input value is reached, the neuron will fire at a rate determined by the length of its refractory period, and any further increase in input value will not increase the firing rate.

The parameters given to the neuron during the tests typically put the threshold at 3000 and the resting potential at 1000, so an input word of 2001 will cause the potential to exceed the threshold on the first cycle.

If the input word increases further, it will eventually reach a point where the sum of the resting potential and the input word exceeds the capability of the accumulator, with the result being truncated. With a 16-bit accumulator and a resting potential of 1000 as described above, when the input word reaches 64,536

the limit of the accumulator will be exceeded and the resulting potential in the accumulator will be zero. The second integration cycle will then result in a potential value of 64,536, and the neuron will fire. Since the neuron now takes two integration cycles to fire, the firing rate has decreased.

While in practical networks it is likely that the erratic behaviour for large input numbers could be a problem, this section has demonstrated that the neuron without synapses can be used as an input encoding device.

5.14.3: Single Synapse Operation

When using a single synapse, the neuron and the synapse can be connected directly together. The schematic view of this connection, as shown by the Quartus II schematic editor, is shown in Figure 70. This schematic shows two types of parameter entry, with the synapse taking its two parameters through two 16-bit input buses, L and W, and the neuron having a single data input through which all parameters are loaded, and an address input to control this. If both units are of the type with address and data inputs, as shown in Figure 71, an additional address decoder is required which can map the units' individual address spaces into a single continuous address space.

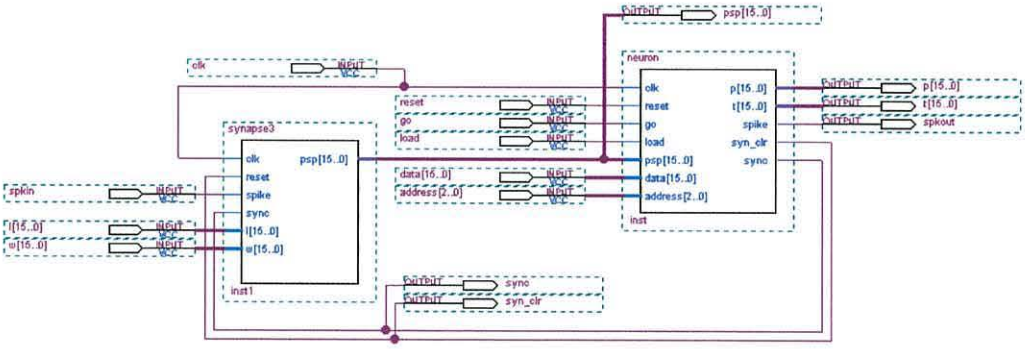

Figure 70: Quartus symbol layout for a single synapse test system

Another item to note about Figure 70 is that most of the signals passed between the neurons have pins attached for testing, allowing the internal operation of the

neuron and the circuit as a whole to be measured. A practical network would not have taps on the synapse control lines, and would use a version of the neuron without the T output. This is connected to the time counter and is only used for debugging. It is almost certain, however, that the P (membrane potential) output would be present in such a system, along with the PSP input to each neuron. These would be connected through a series of multiplexers to a single output channel, which would allow the simulated potentials present in any neuron to be displayed graphically. It was found during the course of the designs that adding these extra readouts would tend to increase the logic size of the neuron, as registers which were otherwise optimised away by the compiler were forced to exist as coded, to allow their contents to be read out easily.

When using the more complex synapse, the neuron's SYN_CLR (Synapse Clear) and SYNC outputs connect to the RESET and SYNC inputs on the synapse.

5.14.4: Operation with more than one synapse

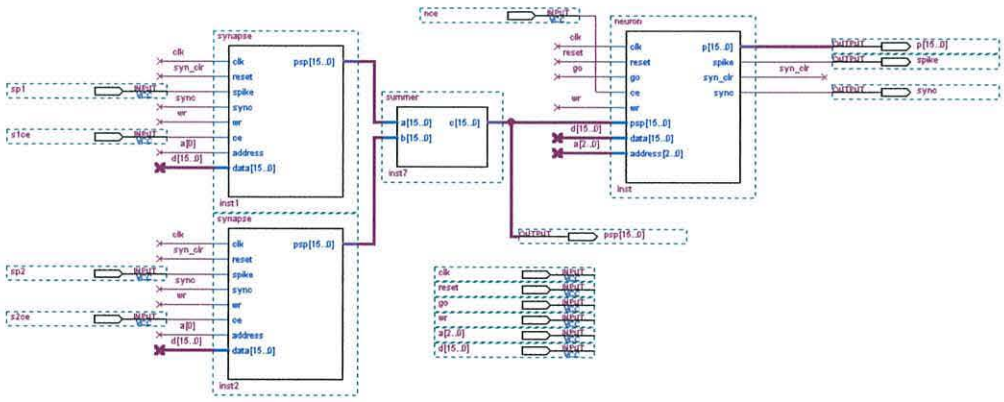

Figure 71: Extract from schematic showing a neuron with two synapses

If the neuron is used with more than one synapse, an extra element is required between the synapses and the neuron in order to add the synapse outputs. There is no limit to the number of synapses which can be attached to a neuron, although care must be taken to ensure that the sum of all the synapse weights, which could be produced if all excitory synapses are triggered simultaneously, can never exceed the maximum number which can be transmitted through the PSP bus to the neuron.

5.15: Testing the Second Neuron Model

A series of tests were carried out using the system with two synapses shown in Figure 71. These, as in the case of the first neuron model, were carried out using the Quartus simulator.

5.15.1: Simple Excitory Tests

To test the basic functionality of the neuron, a number of input spikes were fed to an excitory input in order to make the neuron fire. The weight of the input was set to 250, and the resting and threshold potentials to 250 and 750 respectively. These are a quarter of the values applied to the first neuron when it was tested in section 5.7, the reason for this being that the maximum decay slope available with this design is a decrease of 1 per integration cycle, whereas the first neuron design could apply a much sharper decay. This new neuron will therefore require many more clock cycles to allow the membrane potential to decay between any two values than were required by the previous design with similar values, so to reduce the amount of data which had to be handled during testing the potentials were scaled to restrict them to a smaller range.

The tests performed here were similar to those performed with the first design, with the resting potential, threshold and synaptic weight set up to ensure that three spikes entered in a rapid enough sequence would result in the neuron firing.

The simulated result of this first test is shown in Figure 72, where three spikes inputted 200us apart cause the membrane potential to exceed the threshold. For this test the decay system was set to produce a decay of 1 unit per 10 integration cycles.

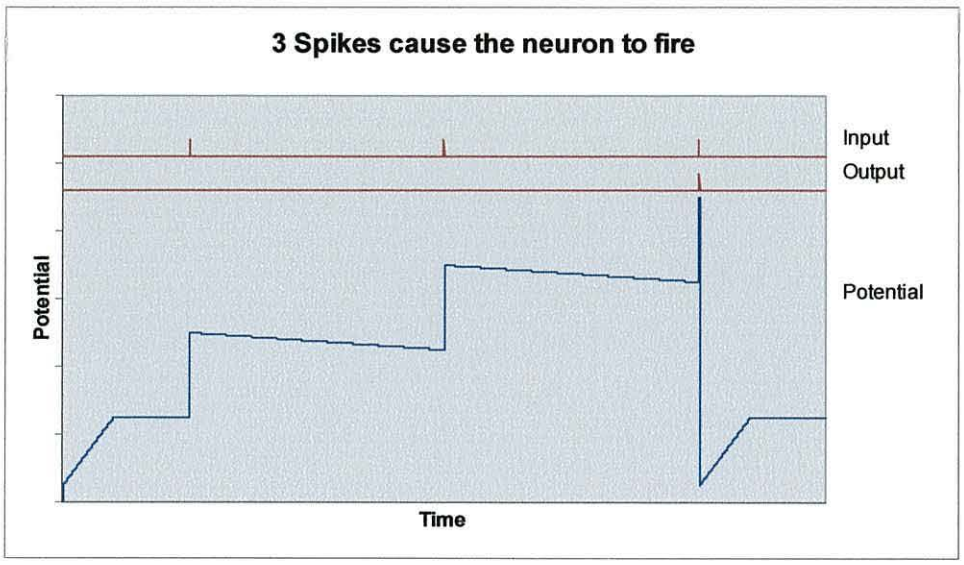

Figure 72: 3 spikes inputted to the second neuron design causing it to fire

This response shows that the neuron functions as the first version, with the integration, decay, threshold firing and refractory period parts of the response working correctly.

In the second stage of the test, the decay value is set to produce a decay of 1 unit on each integration cycle, thus making the decay ten times as fast as in the first stage. The three spikes are fed in at 200us intervals again, with the expectation being that the more aggressive decay would reduce the membrane potential sufficiently quickly after each spike that the three spikes would not be sufficient to exceed the threshold. Figure 73 shows the result of this test.

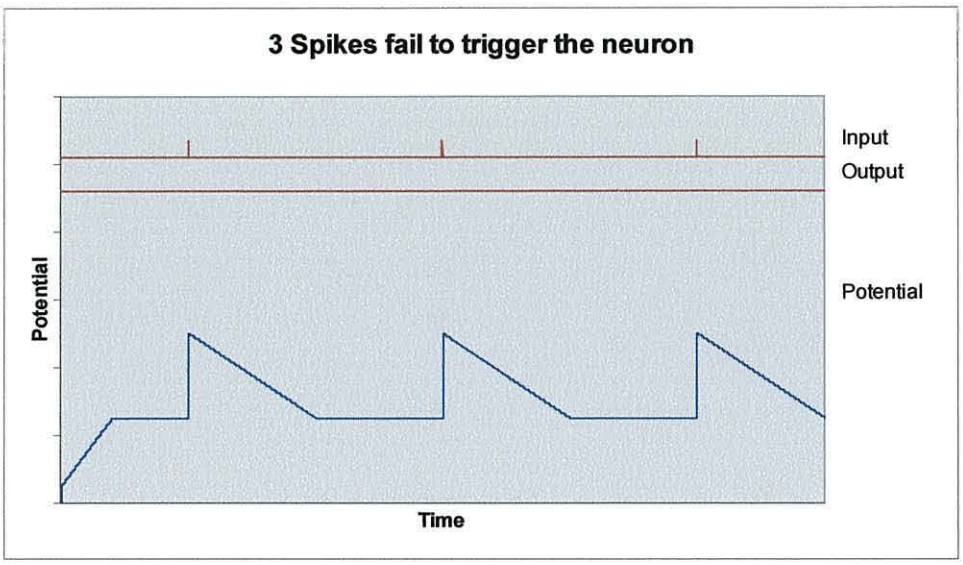

Figure 73: A faster decay prevents the spikes from causing the neuron to fire.

It can be clearly seen that the faster decay returned the membrane potential to the resting potential after each spike, long before the subsequent spike arrived. As with the first neuron design this prevented the neuron from firing.

Keeping the same decay setting, the period of the spike train was reduced, with the spikes arriving at 50us intervals in the third test. Figure 74 shows the result of this test, with the same timescale as the previous two results to show the faster spike train.

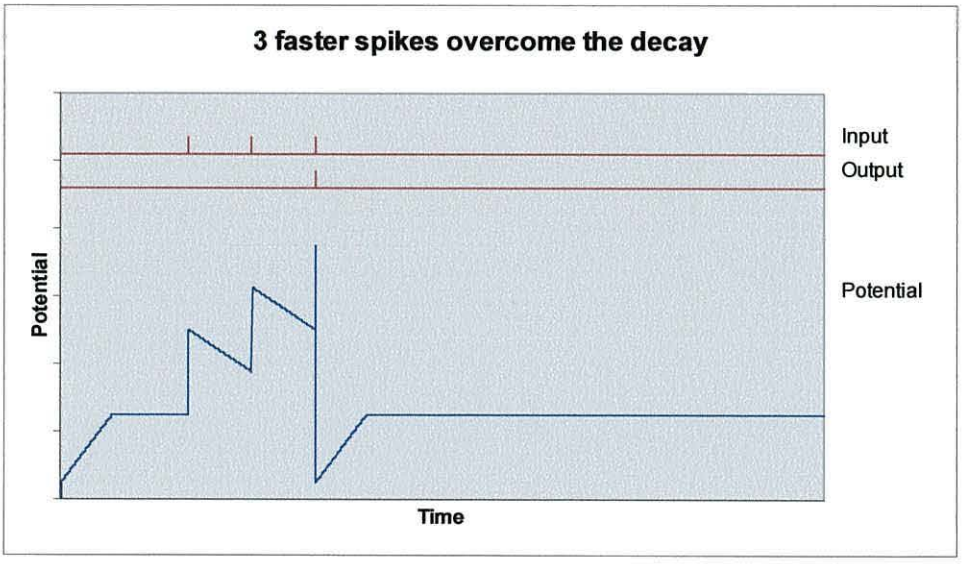

Figure 74: A faster spike train overcomes the faster decay and causes the neuron to fire.

It is clear that the faster spike train results in the membrane potential increasing in steps before it has been able to reach the resting potential, eventually reaching the threshold as was the case with the less aggressive decay in the first test.

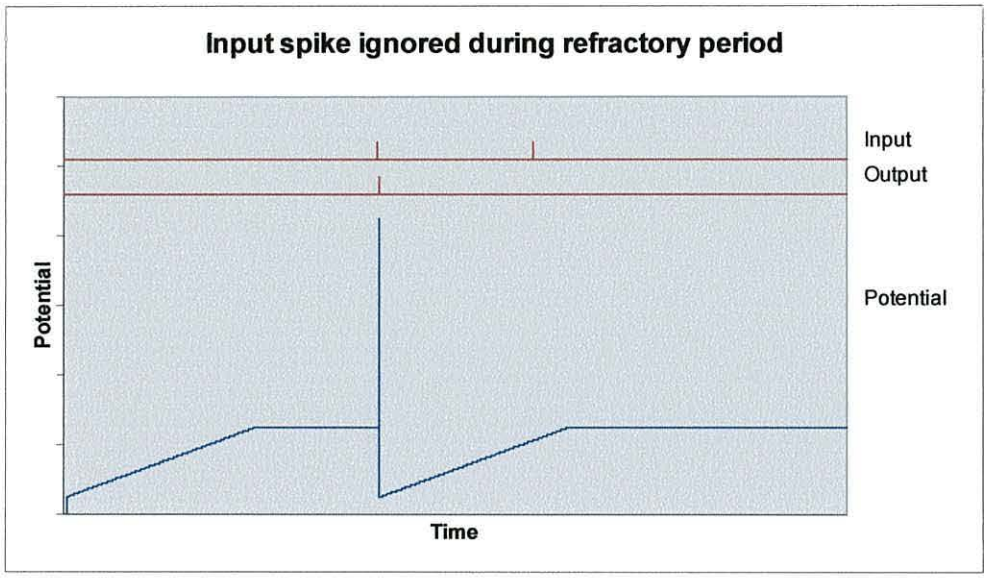

Figure 75: Input spike ignored during refractory period

As an additional test, the weight on the synapse was increased to 600, allowing a single spike to trigger the neuron to fire. Two spikes were then sent in rapid succession to verify that the neuron ignored the second spike, which arrived during the refractory period. Figure 75 shows the result of this test, demonstrating that the second spike had no effect on the membrane potential.

5.15.2: Inhibitory Response

Inhibitory inputs were also tested, with one synapse set to have a negative weight. Figure 76 shows the result of applying an inhibitory input some time after an excitory input. The membrane potential, increased by the excitory input, is reduced by the inhibitory input as expected. Although both inputs have a weight of 100, the inhibitory input delivers this as 10 cycles with a weight of -10 rather than a single cycle, so the decrease in the membrane potential due to this is more gradual than the increase due to the first spike.

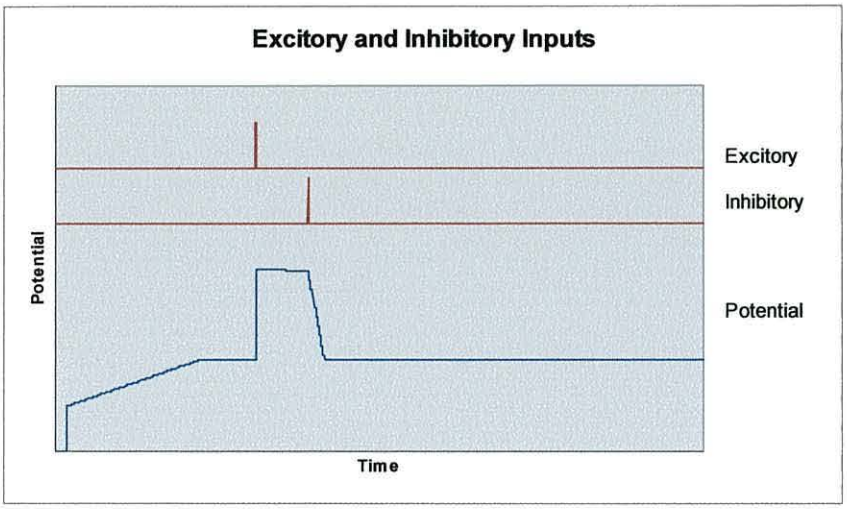

Figure 76: Membrane potential response to excitory and inhibitory inputs

This response shows that in inhibitory input can be used to make the neuron less likely to fire, as it can cancel all or part of the membrane potential increase from previous excitory inputs. It also demonstrates that the bounds-checking logic is functioning correctly, as the membrane potential did not fall below the resting potential. The second PSP exerted a change of –100 on the potential, but the effect of the decay function meant that the potential was less than 100 above the resting potential when this PSP began.

5.15.3: Slow PSP Response

The simple response of the previous tests, where the input spikes each cause a sharp increase in the membrane potential, was the only type of response which could be modelled with the first neuron design. However, this second implementation with its more complex synapses can also model some more complex behaviour, simply by changing the form of the post-synaptic response. In the above test, the synapse was set to deliver its weight in the form of a single-cycle pulse but as it is possible to extend this output pulse, the synapse could deliver its weight as a longer pulse of lower intensity, delivering the same effect but over a longer period. Figure 77 shows the result of applying a single input spike to an input programmed with this type of response.

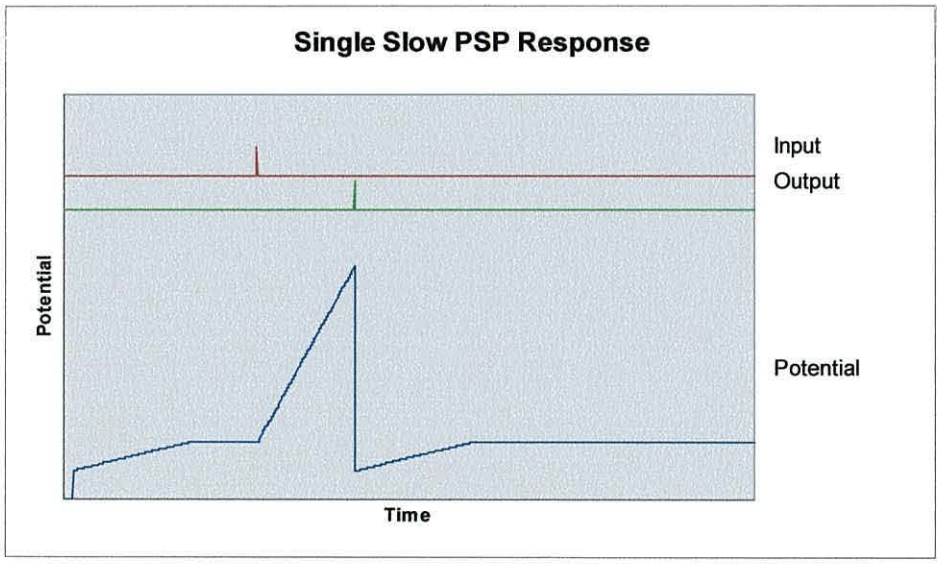

Figure 77: Delayed firing with a slow PSP

The overall weight, the product of the length and amplitude of the pulse, is sufficient that if delivered in a single cycle, it would exceed the threshold immediately and the neuron would fire as soon as it had integrated the input. With a longer pulse of lower intensity the membrane potential has to build up gradually until the threshold is exceeded. The firing is therefore delayed.

A second similar test was performed with both inputs set to provide a slow response, triggered so that their outputs would overlap. The resulting waveforms are shown in Figure 78.

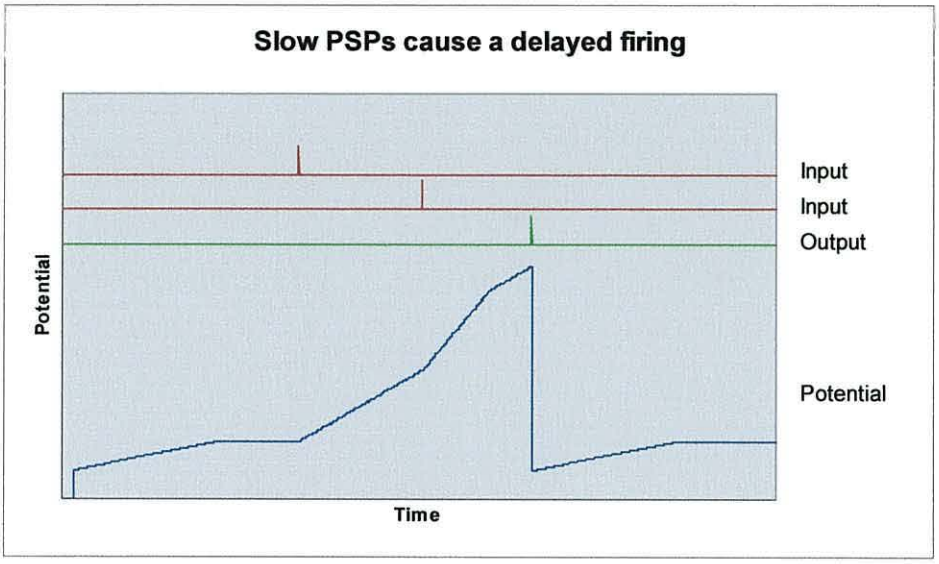

Figure 78: Two overlapping slow PSPs and their effect on the membrane potential

As expected, the rate of change of the membrane potential increased after the second input pulse, then decreased again once the first pulse ended. Each synapse had a weight of 2 and a pulse length of 200, which results in a total delivered effect of +200 on the membrane potential, although at the end of the pulse the actual increase in the potential will be less than 200, since several decay cycles are executed while the synapse's output is active.

For comparison with earlier tests, the neuron was set up in the same way as the first of the excitory tests, as shown in Figure 72, but with the W and L parameters reversed so that instead of a single-cycle PSP of +250 it delivers a PSP of magnitude +1 over 250 cycles. The result of this was that the neuron fired after 3 spikes as before, but with an additional delay, as shown in Figure 79.

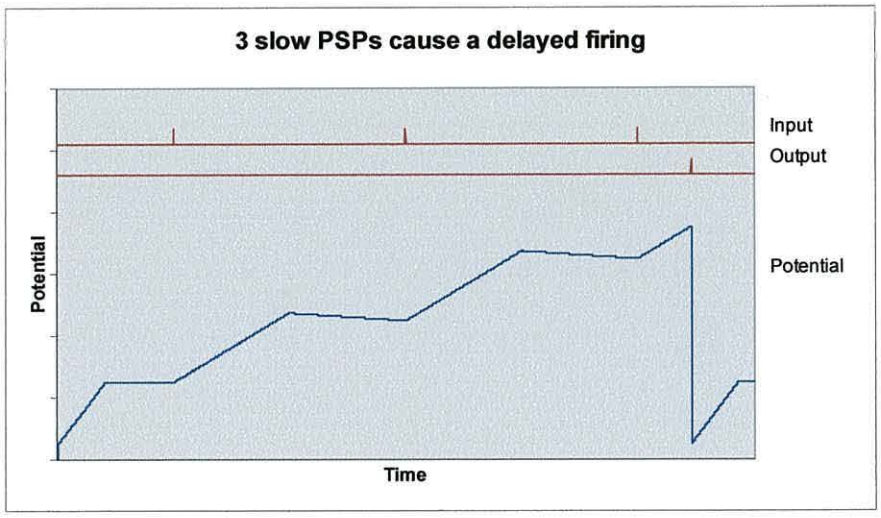

Figure 79: Repeat of the simple excitory test with slow PSPs

Figure 80 shows a comparison of the effect on the membrane potential of two forms of post-synaptic potential (PSP) which can be delivered by the synapses. A 'fast' PSP is one which delivers its weight as a short pulse of high amplitude, while a 'slow' PSP delivers the same overall effect but over a longer period. With sufficient precision in the parameters, a wide range of different types of PSP is available, ranging from one extreme to the other. For this test, the fast synapse had a weight of 200 and a pulse length of 1, while the slow synapse had a weight of 2 and a pulse length of 100. It can be seen from the diagram that the membrane potential after the length of time required for the slow PSP to finish had elapsed was the same in both cases, as in both cases the decay cycles were executed at the same times. The slope of the membrane potential driven by the slow PSP is therefore not constant, as when a decay cycle is executed the potential decreases slightly. Although the two responses eventually reach the same point, the fast PSP causes a response which peaks at a higher value. This means that if the threshold was set low enough, a fast PSP could cause the neuron to fire, while a slow PSP delivering the same overall effect would not. The slow PSP would therefore have to have a slightly greater weight.

This effect only applies, however, if the decay period is relatively short compared with the length of the PSPs. In all the tests which were carried out on this test system, the decay period was set to 10, so the membrane potential will decrease by 1 every 10 integration cycles.

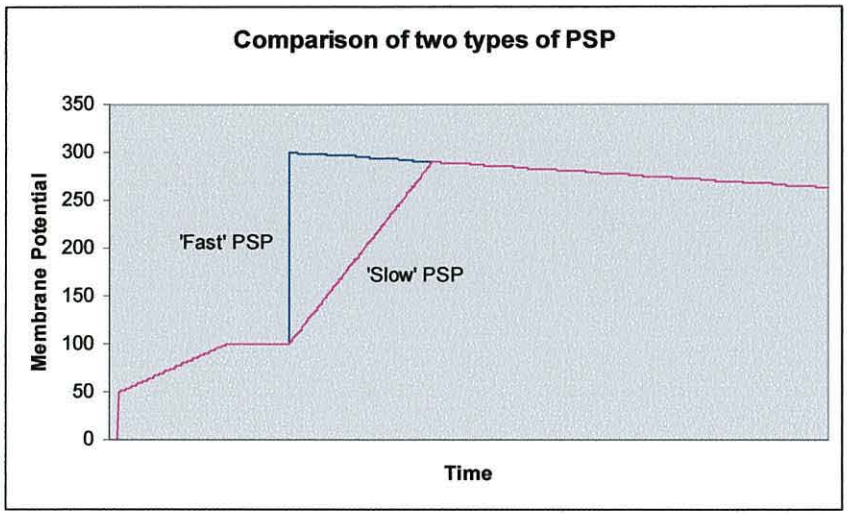

Figure 80: Comparison of the effects of fast and slow PSPs

5.15.4: Conclusion

The tests presented in this section have verified that the second neuron model can perform the same functions as the earlier model, leaky integration, threshold-based firing and refractoriness. It has also been shown that this more complex neuron model is capable of exhibiting dynamics which were not possible with the earlier simpler model, by extending the length of the PSP from the synapse. It has also been shown that this model does not suffer from the problems associated with the inhibitory synapse response which caused the earlier model to fire after a single inhibitory input.

5.16: Operation as part of a network

The neurons in a neural network can be arranged and connected in a variety of different ways. The number of synapses required for each neuron may depend on both the size and the form of the network. In a fully-connected network, each neuron will take input from all other neurons, and may also require an input channel from outside the network. This will require a large number of synapses per neuron, and for large networks there will be a large quantity of interconnections between neurons to take into account. Such a network can be programmed to behave as any type of less fully connected network, such as those depicted in Figure 81, since any connection present in these simpler networks will be present in a fully-connected one.

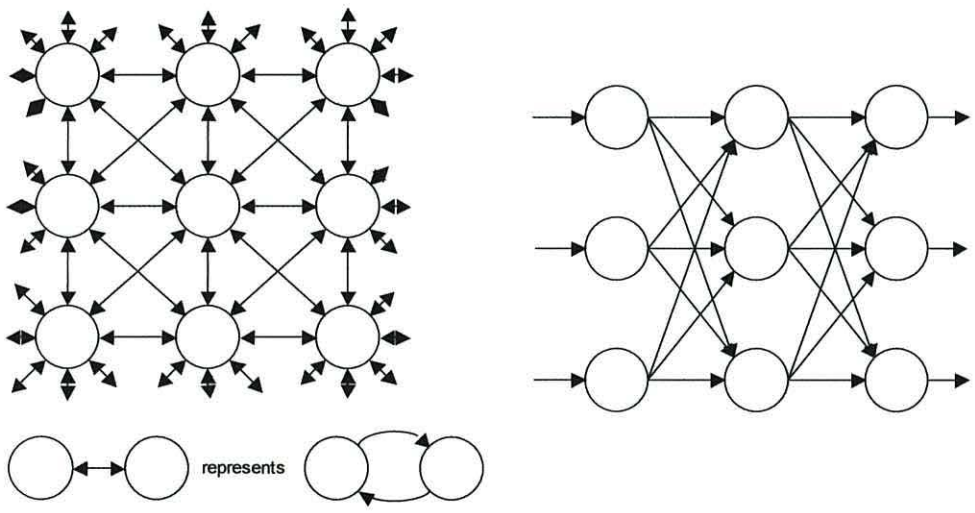

Figure 81: Nearest-neighbour and three-layer feedforward networks

A biological brain will generally not be fully-connected, as the neurons can only take input from the immediately adjacent neurons with which they make physical contact. There may be small clusters of neurons which are fully connected within the cluster, but in a large brain such as that of a human, there will not be any neurons which take synaptic input from all others in the brain. However, as the biological neuron is an electrochemical device, the levels of hormones and stray neurotransmitters can modify the neural response over a large section of the brain by interfering with the operation of the synapses [5.49].

The layered network as shown in Figure 81 (right) is of a topology commonly used with simpler networks of steady-state neurons, such as threshold logic units , where an input signal is presented to the first layer, and the result read from the output layer once the network has stabilised. These networks are often used for pattern recognition or classification, or for more complex functions such as data transformation [5.33].

Networks with neurons connected in a regular lattice arrangement as shown in Figure 81 (left) are more often used for experimentation, though they have been demonstrated in control applications [5.36]. These networks, employing large amounts of connectivity and feedback between neurons, are used more often with spiking neurons than TLUs.

A particular class of network, the small-world network [5.50], is one in which there is less than full connectivity between nodes, but a signal can travel from any node to any other node in a relatively small number of steps. For a neural network, this is of course dependant on the willingness of the intermediate nodes to pass the signal.

5.17: Small Network Testing

It is well known that groups of neurons can produce oscillatory behaviour, on both large and small scales. [5.51] On the large scale, the 'brain waves' associated with the different levels or states of consciousness have been widely studied, and large-scale oscillatory networks are thought to be responsible for tremor-based disorders of the nervous systems [5.52]. On the small scale, groups of neurons known as Central Pattern Generators (CPGs) have been identified in animal nervous tissue, and it is these CPGs that are responsible for many types of repetitive motor functions such as peristalsis in the digestive system or the beating of the heart. [5.53] The latter is a function which is performed constantly, while the former is performed on demand, controlled by inputs from the rest of the nervous system but sequenced by the oscillations within the CPG. On a more complex scale it has been determined that some aquatic animals' swim patterns are controlled by these networks, with the oscillation arising from synaptic interactions between the neurons.

The feedback provided by the connections between the neurons has been shown to be one of the mechanisms by which real CPGs oscillate [5.54], and in [5.55] it is shown that the network will settle into a repeating pattern, which can be perturbed by external inputs, though many CPGs are quite stable and will return to their natural firing pattern quickly once external stimulation is removed.

CPGs in many animals can be quite small, often with a few tens of neurons and the behaviour of these has been studied and even replicated with analogue artificial neurons. [5.55]

An experimental system was constructed using the second neuron model to determine whether this oscillatory behaviour could be replicated using such a simplified model. The first of the neuron designs is not suitable for this experiment, as it lacks the capacity to produce a time-delayed response, having only a simple instant-action synaptic model. This would result in all the connected neurons firing almost simultaneously and subsequently entering their refractory periods, and thus ignoring feedback from each other.

The overall aim of the tests is to replicate the oscillatory behaviour of the CPGs discussed above in a network of simplified artificial neurons. The first aim is to

determine whether simple oscillatory behaviour can be obtained, then subsequently to determine whether perturbing the network with extra input signals would result in a change to the pattern of oscillation, and whether the network would return to a stable state.

5.17.1: Network Layout

The lobster's stomatogastric CPGs studied in [5.55] typically contain 25 neurons, but due to the limited space in the target FPGA, a smaller network of 9 neurons was used. These were connected as a 3 x 3 grid, with each node taking input from its neighbours, and the edges connecting to their opposites, e.g. a toroidal network. It can be seen that in the case of a 9 neuron network such as this there is full connectivity, as each neuron takes input from 8 others, but if the network were larger the nearest-neighbour connections would not be sufficient for full connectivity. Thus the 3 x 3 network can, by disabling particular synapses, be made to resemble any smaller network. A schematic view of this network is shown in Figure 81 (left). The number of neurons was chosen due to the limitations on the number which would fit into the chip, while the 3 x 3 layout was chosen as the most logical way of arranging 9 neurons.

Each neuron was provided with 9 synapses, and while in most cases the extra synaptic inputs were unused, two of the neurons were connected to push buttons so that extra stimulation could be provided.

The nine spike outputs from the network were connected through pulse extenders to a PC for data logging. The pulse extenders work in a manner similar to the synapse models, but with a single-bit output and a hard-coded pulse length of 16 cycles. These are provided purely to prevent the PC missing any spikes due to their short duration, and also to allow the network activity to be displayed visually. For basic visual testing, the network's outputs were displayed on a VGA monitor, as a grid of green blocks which flashed red when the corresponding neuron fired. This, however, required that the network was made to run much more slowly than normal, and in fact firing rates of around 1Hz were obtained, by running the neuron models with a clock of 250Hz. Since for any given set of parameters, the firing rate increases in direct proportion with the clock rate, this implies that with the same parameter settings, biologically plausible firing rates of

around 2KHz could be obtained with a clock rate of just 500KHz, though it is likely that a practical network would run with a higher clock rate and parameters adjusted to obtain a much larger range of membrane potential values during normal operation, allowing more precision in the sub-threshold dynamics.

Initially, all neurons were set up with the same parameters, with the horizontal and vertical links being the only active ones, the others having weights of zero. To ensure feedback, the weights were set up so that a single spike inputted to any neuron would cause it to fire after a short delay, with a low intensity but temporally stretched PSP from each synapse.

5.17.2: Single Stimulus Response

When the centre neuron was stimulated, the network settled into a steady firing pattern, with each neuron being re-triggered once its neighbours fired. This was made possible because of the delayed response from the slow PSP, which meant that the first neuron would have finished its refractory period before the others fired. When another stimulus was applied the network entered a brief period of instability, with no discernable repeating pattern, and then settled into a new pattern.

Figure 82 shows a sample firing chart for a short duration run of this network. Each neuron is represented by a row in the chart, with the black bars representing the output spikes. It can be clearly seen that the delay after the first spike is much longer than the subsequent firing periods, this is due to the number of inputs contributing to the firing in each case.

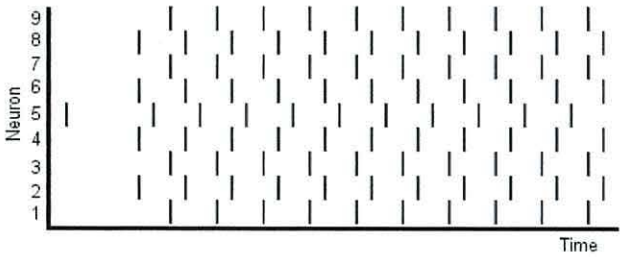

Figure 82: Firing chart for a simple network

The first output spike from the centre neuron (5) triggers the four orthogonally-adjacent neurons (2, 4, 6, 8), but through a single synapse on each one. When these fire, they retrigger the centre neuron through four of its inputs, resulting in the membrane potential exceeding the threshold in approximately a quarter of the time. These four neurons also stimulate the four corner neurons (1, 3, 7, 9), but only through two synapses each, resulting in a longer delay before they fire. The network then settles into a steady periodic behaviour.

5.17.3: Multiple Stimulus Response

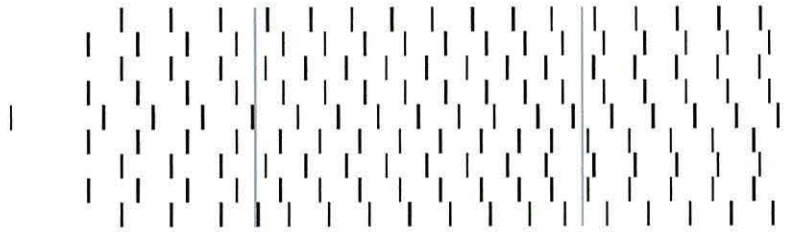

Figure 83: Firing pattern with additional stimulation indicated by the grey bars

Figure 83 shows the result of applying additional stimulation to the network once it was in a stable firing pattern. The left-hand grey line shows the point at which neuron 1 (lowest trace) was stimulated by an external input, causing it to fire early, upsetting the pattern. The period between the two grey lines shows a small degree of variation in the firing pattern, but after the right-hand line the network has settled into a regular but different sequence, and the pattern is periodic again. It was observed that when the threshold values were the same for all neurons, the pattern resulting from the initial stimulation was always the one shown in Figure 82. If a single threshold was increased or decreased by a small amount, the pattern was found to be generally similar, but with the changed neuron firing slightly out of step with the others.

The system can be seen to have a number of different periodic orbits, characteristic of the dynamics of a nonlinear system [5.56], with the stimulation causing it to switch between orbits.

5.17.4: Complex Dynamics

When the thresholds for the neurons were all different, but spaced regularly, the network would fall into a repeating pattern of somewhat greater complexity, and would take a longer time after the initial stimulation to do so. Until the network reached the stable point, it would fire in a chaotic and unpredictable way, though since the neurons themselves are predictable units, the pattern will be the same on each run, if starting from a resting state in each case.

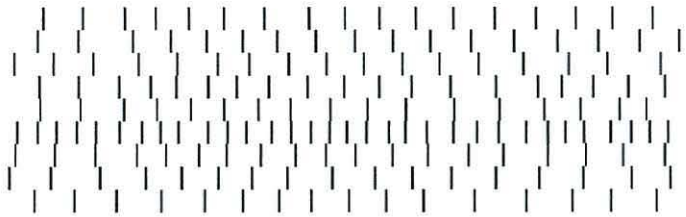

Figure 84: Part of an unstable network's firing pattern

An example of an unstable network's firing sequence is shown in Figure 84. In this case, all the synapses were set up to produce 'slow' PSPs which delivered a change in the membrane potential greater than the threshold, so that the neurons were guaranteed to fire within a relatively short number of integration cycles. A lower threshold, or stimulation from more than one input, would produce a shorter firing delay. All the neurons were set up with randomly chosen different threshold values, and it can be seen from the firing sequence that no neuron was firing at a constant rate throughout the whole period.

It was found to be hard to determine whether a network firing in this way was actually firing chaotically, or whether it was simply producing a repeating pattern with a very long period of repetition. The visual display made it easy to spot short-period repetition, and some medium-period repetition could be seen in the firing plots obtained from the data logger, but to find long period repetition would require more complex techniques, which were not investigated. Chaotic dynamics have been shown to exist in neural systems, both in real neurons [5.57] and in complex neuron models [5.58] and it appears from the response of this network that chaotic dynamics can be achieved with greatly simplified models too.

5.17.5: Analysis

It is clear that if CPGs are the controlling element in cyclic processes such as locomotion and digestion, the output pattern produced by the CPG can be perturbed by incoming neural signals and made to change. Examples of this are found in animals with simpler neural structures, such as the tadpole [5.54], in which it is clear that the neural circuits responsible for the swimming motion must be able to change their output or the tadpole would not be able to change its direction of travel. We have seen that the experimental network, when set up to produce a stable oscillation, can be made to change its pattern, both temporarily

and permanently, by external stimulation. This stimulation could come in the form of additional spikes fed into the network from outside, as was seen in section 5.15.3, or in the form of modifications to the weights or thresholds, as was shown in section 5.15.4. This latter case would be a result of chemical changes to the neurons' operating environment. [5.49]

Although the real neural CPGs are more complex in their interconnections, and more complex in terms of individual neurons' dynamics, this experiment has shown that the neuron model is capable, when working as part of a network, of performing functions similar to those observed in real neural circuits.[5.55] It has not yet been determined whether this network of neuron models can replicate exactly the measured behaviour of a real CPG, or whether the output is actually chaotic, but the network was shown to produce complex dynamics with a number of periodic orbits, making the transition between these orbits when stimulated by external inputs, in a manner very similar to that of the simple biological CPGs of animals such as the tadpole or lobster. It has also been seen that the dynamics produced would not be possible using the simpler first neuron model of section 5.6, as the more sophisticated time-dependant response of the second neuron model is required for these dynamics to occur.

5.18: Some Functional Elements Built With Neurons

It has been demonstrated that logic functions ranging from simple AND, OR or XOR gates to adders and multipliers can be realised with threshold logic units by careful choice of the weights on the inputs. [5.59], [5.60] These threshold units, as was discussed in section 5.3.1, respond to steady-state inputs, with the output potentially only settling when the inputs are stable. In this respect they are functionally similar to the combinatorial logic functions which have been implemented in them. Most of the published work on using neurons or neuromorphic hardware to design logic hardware has focused on threshold units, though some work has been done in using spiking neurons for logic circuits [5.61], though this was done with analogue VLSI neurons rather than digital ones.
A driving factor behind this use of neurons rather than traditional hardware is the discovery that some functions require less hardware when implemented in this way In the case of adders and multipliers, it was found that the hardware requirements grow less quickly than with traditional methods as the size of the input word increases. [5.59]
The following neural circuits were intended to be simple test circuits to demonstrate the flexibility of the neuron model. The experiments presented in the previous section have shown that it is possible to obtain complex and interesting behaviours from a network by simply altering the parameters. The aim of the tests in this section is to demonstrate that certain behaviours can be tailor-made by setting up the network specifically according to a design, rather than by any process of evolution.
The circuits presented in this section are purely speculative, but as they are very simple it is not unlikely that a large evolved neural network could contain circuits similar to these.

5.18.1: Simple Logic Gates

The easiest type of logic gate to implement with these spiking neuron models is the OR gate, as it is possible to set up a neuron with sensitive inputs so that a single spike inputted to either input will cause the neuron to fire. An AND function is much more difficult, as although it is possible to reduce the weights so

that the neuron will only fire when both inputs receive a spike simultaneously, the same effect can be produced by simply increasing the frequency of the spike train fed to a single input (see section 5.7.1). One possible solution to this is to 'buffer' the inputs with other neurons whose refractory period will limit the frequency of the spike train, while an alternative is to impose a limit on the frequency of any spike trains applied as inputs to the system overall.

A NOT function can be achieved by combining a single inhibitory synapse with a constant excitory bias input. [5.61] This is only possible with the second, more complex neuron model, where a stimulation current can be applied to the input (section 5.14.2). The bias will result in the neuron firing continuously with a period determined by the total time required for the integration of the bias to exceed the threshold and the time spent in the refractory period. When a spike arrives at the inhibitory synapse, the output of this synapse cancels the bias and the neuron is not stimulated, and stops firing. The length of the PSP produced by this synapse determines how long the neuron remains in this state, after which it will begin firing again.

5.18.2: Spike multiplier

A spike multiplier accepts an incoming spike and provides a train of output spikes, the number of output spikes being determined by the number of neurons in the circuit. It is possible to replicate this function to a certain degree if the synapses are configured so that they are not cleared during the refractory period, it is possible to extend the output pulse from a synapse so that it lingers through the refractory period after the first spike and retriggers the neuron. However, this can only produce output pulse trains at a fixed frequency. Chaining neurons can theoretically provide a wide variety of different output signals, as the intervals between the pulses are set by the firing delays of the intermediate neurons.

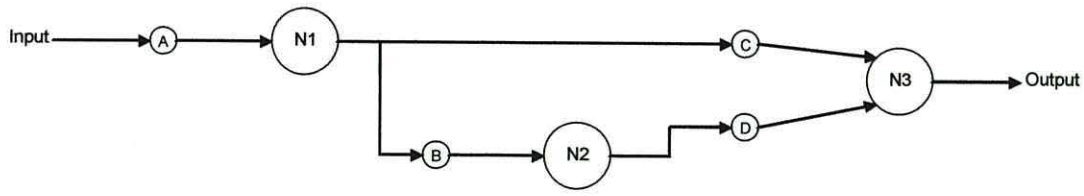

Figure 85: Diagram of a spike multiplier

The basic form of such a circuit is a chain of neurons, as shown in Figure 85, each of which causes the next in the chain to fire, after a delay. An extra neuron at the end of the chain acts as an OR gate, so that all pulses appear at a single output. This is the simplest form of the circuit, forming a spike doubler.

If the chain is a loop, as in the set-reset latch to be discussed in the next section, then the output will fire at twice the frequency of either of the neurons in the loop.

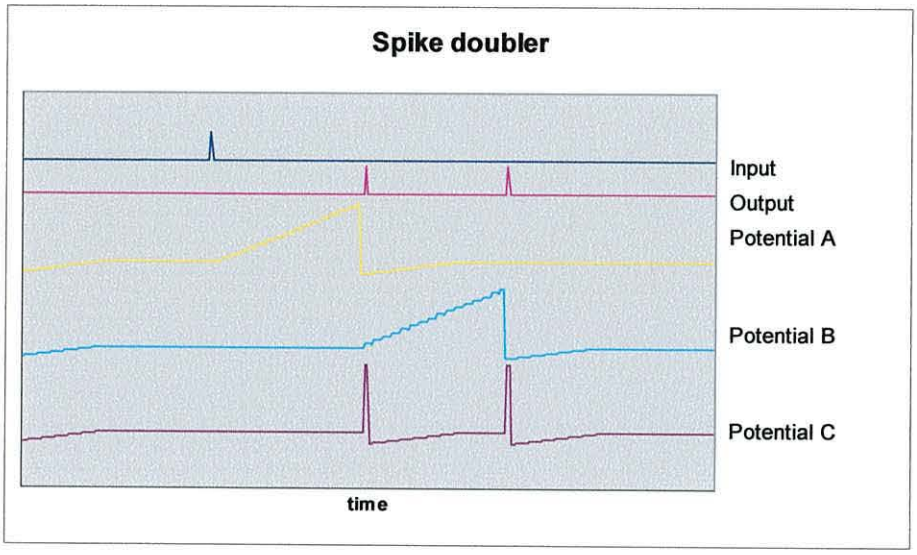

Figure 86: Spike doubler waveforms

As can be seen in Figure 86, the spike doubler relies on the membrane potential building slowly in N1 and N2, so that the firing is delayed. This requires that the PSP emitted by the synapses on these neurons is of a low intensity, but lingers for some time after the incoming spike is received. It is also necessary for the output neuron, N3, to have a short enough refractory period that it can recover from the firing induced by N1 before N2 fires. If this is not the case, then a spike doubler would require a chain of more than two neurons, tapped at suitable intervals to allow the cumulative delay of the chain to exceed the refractory time of the output neuron.

While this circuit may be purely speculative, its simplicity suggests that it is not unlikely that a large evolved neural network would contain at least one similar chain of neurons.

5.18.3: Neuron Set-Reset Latch

The wide range of post-synaptic responses possible with a synapse which can provide a stretched output pulse allows for a range of interesting neural circuits to be built. One such circuit is the Set-Reset latch, which has an approximate functional relationship with its electronic counterpart. In this circuit, two neurons are coupled into a feedback loop as shown in Figure 87.

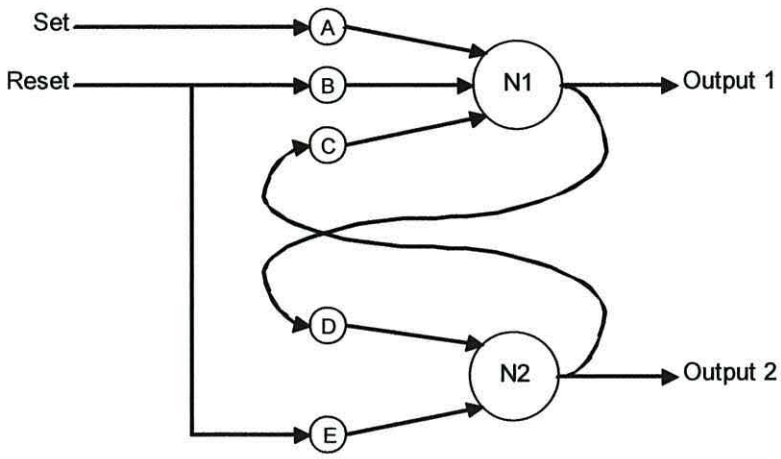

Figure 87: Cross-coupled neurons acting as a set-reset latch

It is important to note that while the electronic equivalent of such a circuit is based on steady-states, so that when triggered its output maintains a constant logic level until the alternative input is triggered, this steady-state output is not possible with spiking neurons because by definition they do not hold steady outputs. They can, however, be made to fire at a steady rate once triggered by an input spike. If the information carried by the spike trains is assumed to be encoded in their frequency, this could be thought of as a steady state in terms of the information outputted.

In the circuit of Figure 87, neuron 1 has three synapses while neuron 2 has two. If it can be assured that the resetting spike will never arrive during neuron 1's refractory period, then synapse E can be removed to further simplify the system.

Synapse A has a large excitory weight, delivered as a relatively short pulse, enough to cause N1 to fire immediately. Synapses C and D, the feedback

synapses, have similarly large weights but produce longer PSPs of lower intensity, each allowing the originating neuron to complete its refractory period before the other is triggered. Synapses B and E have strong inhibitory effects, stretched over a long enough period to stifle the feedback pulses, at least to the point where they just fail to trigger an action potential. The two neurons each have very short refractory periods, although provided that the PSPs from the feedback synapses are long enough, this need not necessarily be the case. In addition, there is no requirement for there to be just two neurons involved in the feedback loop, if a longer chain of neurons was used, a ring oscillator with start/stop control would be created.

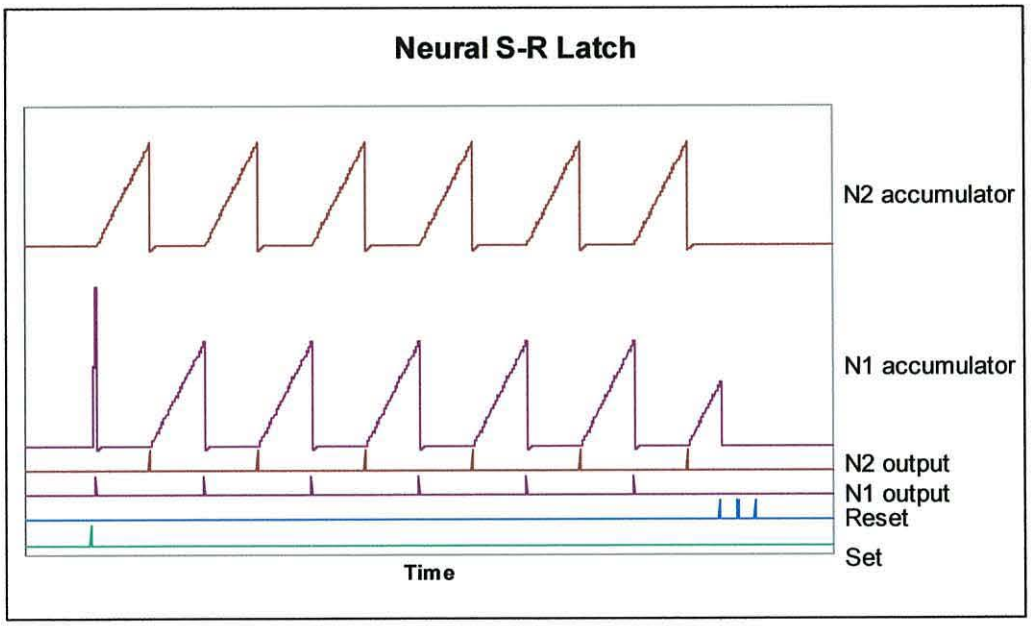

Figure 88: Test waveforms for the set-reset latch

Test waveforms from this circuit are shown in Figure 88. The lowest trace shows the spike input to the 'set' input, which initiates a large change in the membrane potential of N1. This causes N1 to fire, which in turn initiates a longer, slower build-up of membrane potential in N2. The cycle repeats, with each neuron's firing causing the other to fire after a short delay. Finally, a series of spikes on the reset input causes the rise in membrane potential in N1 to be stopped before the threshold was reached, stopping the circuit. Three spikes were fed into the reset input to ensure that it would stop the feedback correctly, in case the first arrived

during the refractory period but due to the very short refractory period of both neurons the feedback was stopped by the first spike.

The S-R latch circuit was tested using the 3 x 3 network described earlier. Due to the simplicity of the circuit, the nearest-neighbour connections in the network were adequate for reproducing the circuit. The unused synapses were set to weight 0, pulse length 1 so that they had no effect on the neurons.

In order to make the full set of signals visible, two additional neurons were used to buffer the set and reset inputs, their input synapses being set up to trigger firing immediately. The outputs of these, and the outputs of the two cross-coupled neurons, can be seen in the firing sequence plot of Figure 89. The particular choice of neurons was made based on the fact that neurons 5 (centre) and 1 (one corner) were connected to two of the Digilab's buttons for testing.

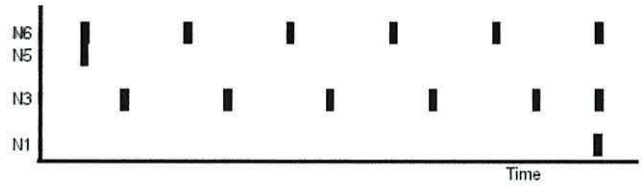

Figure 89: Firing sequence for initial test of S-R latch, showing erroneous response

The initial test was performed with the thresholds of the two cross-coupled neurons set differently, so the firing pattern in Figure 89 shows a slight asymmetry in the oscillation. An anomaly can be seen at the end of the sequence when the reset neuron (N1) is triggered, causing both N3 and N6 to fire simultaneously. This does have the effect of stopping the oscillation, as both neurons then enter their refractory periods and so clear their synapses. However, it is not the expected behaviour, as the inhibitory synapses feeding N1's output to N3 and N6 should not be able to fire the neurons.

The reason for this anomalous behaviour is the way in which negative numbers are represented by the system. The inhibitory weights used were -500, represented as a 16 bit number as 2^{16} - 500, or 65036. The resting potential was set to 100, the thresholds to 180 or 200, and the post-firing potential to 80. If we assume that the membrane potential would be somewhere between 100 and 200 for each of the oscillator neurons, adding 65036 to this would produce a number between 65136

and 65236, which, being less than 2^{16} would not be truncated in the addition so the bounds-checking performed in the integration step would not detect a problem. The new membrane potential then becomes very large, exceeding the threshold and causing the neuron to fire.

To remedy this, the normal operating range of potentials was raised so that the resting potential was 1000, the threshold was 1200 and the post-firing potential was 980. Thus, adding 65036 to the subthreshold range now yields a number between 66036 and 66236, which would be truncated by the 16-bit adder to produce values between 500 and 700, which is correct as it represents a subtraction of 500 from the original potential.

This behaviour demonstrates one of the pitfalls of a simple model such as this, something which must be taken into account when choosing the neuron's parameters. The anomaly occurred because with the membrane potential at 100, there isn't 'room' below this to accommodate a change of -500 without an error. This means that the resting potential and threshold must be set adequately high above zero that the largest inhibitory weight can be accommodated.

In theory, there is no reason why the model should behave any differently as the base value for the membrane potential changes. Provided that the threshold and post-firing potential are the same distance from the changed resting potential, there should be no change in the overall response of the neuron. However, if the network is trained by a learning algorithm, it is possible the these anomalies, being part of the neuron's operating repertoire, could be made use of by the algorithm, resulting in a network which performs its function in an unusual way.

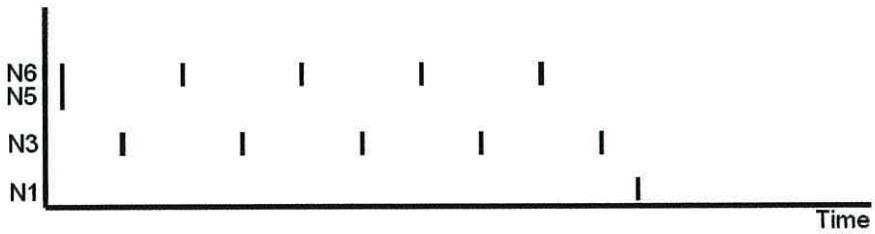

Figure 90: S-R latch running correctly with new parameters

The output of the corrected system is shown in Figure 90. Again, N5 starts the sequence by causing N6 to fire immediately, then N3 and N6 fire alternately until the inhibitory output from N1 stops the process.

This circuit is a simpler version of the pattern generators studied in section 5.17, but in this case the function of the circuit was designed rather than simply arising from the neurons' dynamics and coupling.

5.18.4: Conclusion

The circuits presented here demonstrate that while the neurons may be capable of complex neuromorphic responses when used in a network, their simplicity means that circuits can be explicitly designed without needing to evolve them or train the network. The problem of implementing logic functions in spiking neurons was discussed, and while it has been shown in other publications that Boolean logic functions can be implemented in neurons such as threshold logic units, the lack of a steady state in a spiking system complicates the design of some logic functions, such as AND or NOT, but not others such as the OR gate.

The spike multiplier showed that a simple circuit could be designed and would work as expected, demonstrating the spike-timing based operation of the neurons and also the Boolean OR function, while the S-R latch circuit, which is really more of a controllable ring oscillator, showed that oscillatory dynamics can be designed into a network.

5.19: Overall Conclusion

The design and implementation of these neuron models have shown that even a simplified representation of an already simple neuron model can replicate the basic functions of a biological neuron, and can exhibit relatively complex behaviour. We have seen that a network of such models can also exhibit very complex dynamics, which further work may establish as being chaotic. The sequences produced certainly do not exhibit short-period repetitions.

A comparison of the two neuron designs shows that the simpler neuron requires a smaller area in the chip, but also has a more limited range of functionality. The more complex design provides a better approximation of the real neuron, but with the disadvantage of using a greater area of the chip, resulting in a reduction in the number of neurons which can be used in a network. This more complex model has the capability to produce delayed effects due to the longer output pulses which can be supplied by the synapses. This delay was what permitted the periodic behaviour demonstrated with the test network, as if there was no delay the neurons would be in their refractory periods when re-stimulated by their neighbours. Modelling the synapses as separate units also allows a greater range of different setups to be produced, as the number of synapses is not restricted and can be changed without affecting the operation of the neuron body. For large numbers of inputs this system is also more efficient than designs where the inputs are integrated in turn, as it integrates the net effect of all post-synaptic potentials simultaneously and therefore requires only one integration cycle to cover all inputs.

In comparing the neuron models presented in this chapter with other published works it is evident that some other models have been developed which are smaller in terms of area occupied. However, in those cases a large proportion of the saving in area is attributable to the reduction in flexibility of those other models, which have hard-coded thresholds and decay rates. If future work is to incorporate learning mechanisms as discussed briefly in section 5.13, then flexibility in such parameters may well turn out to be essential, and hence the reduced-area models presented elsewhere may not have the necessary flexibility.

When comparing performance in terms of integration time per logic element, the models presented here compare favourably with others, and crucially the scaling behaviour with increasing numbers of inputs is far better for the models presented here, with constant integration time regardless of the number of inputs. This constant integration time also means that the neuron models presented here have a clear advantage over the majority of the other models considered, namely that they can be combined into networks without explicit synchronization, making the models presented closer to an analogue implementation, which in turn is more biologically-realistic.

As was discussed earlier, other works have shown that an alternative, though slower method of producing a large neural network is to time-multiplex the neurons onto a single model. The results presented in this section certainly support this, as it has been shown that only a few neurons will fit into a moderate-sized FPGA, and while the current generation of FPGAs may be many times larger, even the largest could only allow a hundred or so neurons to exist in parallel.

However, the performance of the simplified neuron model suggests that a mixture of time-multiplexing and parallel operation could be employed, as it has been shown that a model simple enough to fit several times into a medium-capacity FPGA can perform complex operations, so several of these could be employed in parallel to speed up the execution of the time-multiplexed network. This parallelism idea fits with what was found during the image processing tests, where each individual processor may not be able to operate very quickly, but if several are used in parallel the performance could potentially rise significantly.

An important consideration when building a neural network in this way is that it will clearly be less tolerant of faults in the neurons than one built fully-parallel, as if all the neurons in the network are time-multiplexed onto a single physical neuron model, a failure in this circuit would stop the whole network. If two instances of the neuron model were used, a failure could potentially stop half of the network. A fully-parallel network would only lose a single neuron if a single neuron model was to fail, and it may be possible for the network to continue

operation, perhaps producing a few errors. In the same way that the human brain can cope with individual cells dying without failing completely, such a 'graceful deterioration' property would certainly be desirable in systems based on such neural networks

The experiments with the neuron models have also revealed the ways in which the simpler arithmetic logic can produce incorrect results. We have seen that placing the operating range of the model too close to the limits of the representable numbers can cause erratic operation, and can sometimes cause the models to fail in ways which do not immediately reveal the true cause. The first model's response in the cases where inhibitory synapses are used showed the importance of bounds checking. However one of the outcomes of this work is a clear understanding of the scope and impact of these anomalies.
The resting potential detection in this first model could also give rise to unpredictable operation, as it is possible for the membrane potential to rest at any value between the two bounds, and so if it is at the top of the band, a certain input may cause the threshold to be exceeded, whereas the same input might not cause this if the potential was resting at the bottom of the band. These effects arise from the simplified arithmetic in use in the models, and are another indication of the trade-off between simplicity and operating capability.

The results of the tests conducted with the small network in section 5.17 show that the second neuron model, despite its simplicity, is capable of a biologically-plausible response when used in a network, showing periodic firing with stable patterns similar to that found in the CPGs such as the lobster stomatogastric ganglion described in [5.55]. It was found that only the second of the neuron designs, with its greater flexibility in terms of PSP response, was suitable for this application, as the biologically-plausible functionality is dependant on the ability of this model to delay firing due to a slow build-up of the membrane potential. This type of PSP response is common in analogue implementations of neurons, as it is easy to do, by restricting the current into the cell, but it is a novel approach in digital spiking neurons, and as has been seen, shows promising results.

The spike multiplier and S-R latch circuits in section 5.18 show an interesting way of building more complex systems from the neuron models. It is easy to see that these two examples represent just a small part of the possible range of functions which could be built in this way. It was stated in the text that the spike multiplier can be extended to produce any number of spikes when triggered by an input, and this is made possible by the flexible nature of the input to the neuron model, which can allow input from as many synapses as are required. These two simple examples show the neurons behaving in ways which might not be obvious from the descriptions of the operation of the neuron presented at the start of this chapter. Extensions to these models can be imagined, such as the extension of the S-R latch to produce a ring oscillator as described in section 5.18.3. With just a simple modification to allow the neuron to pass through its refractory period without clearing the synapses, it should be possible to make a single neuron re-trigger itself, and thus it may be possible to make an S-R latch with a single neuron. It is also easy to envisage neural circuits built in this way which can perform Boolean logic functions. The output neuron in the spike multiplier is acting as an OR gate, replicating the input spikes regardless of which input they occur through. An AND gate could simply be a neuron with two inputs, each of which can provide half the required increase in membrane potential required to exceed the threshold. Coincident inputs on both inputs would cause it to fire, but a single input would not. Clearly there are issues with this idea which would need to be addressed if Boolean logic is to be used, for instance there is no easy way to make an equivalent of a NOT gate, and also the AND gate could be triggered to fire by consecutive pulses on a single input, if the decay is not aggressive enough. However, there is no reason why pure Boolean logic should be necessary, as the range of functions possible even with these simple neuron models allows for more interesting effects than are possible with digital logic gates.

5.20: Further Work

There is much left to be explored with these neuron models. For a start, there are numerous ways in which the model can be extended to allow a greater range of capabilities, and a few areas have been identified where error-checking or bounds-checking logic could be added to make the operation more reliable.

The decay function in the later neuron models is one area which could be made a little more flexible. At present the decay works by subtracting 1 from the accumulator every n cycles, which means that the maximum decay slope is a decay of 1 every cycle. Adding an extra parameter and altering the logic could allow this to be increased, so that the extra decay parameter, d, is subtracted every n cycles rather than the constant decay of 1. Similarly the refractory period slope could be modified to allow a sharper increase. The decay rate does not necessarily have to be constant, but could instead be made proportional to the membrane potential, giving an exponential decay rather than a linear decay. It would be necessary to determine whether there is any improvement in the neural model arising from such a change, or whether the neuron is just as capable with either method of decay. It was shown that biologically-plausible network responses are possible with a linear decay, so it seems that an exponential decay is unnecessary.

Another potentially useful modification would be to allow the inputs to push the membrane potential below the resting potential, perhaps as far as some lower bound set as an additional parameter, so that the decay gradually brings the potential back up to the resting potential, and allowing inhibitory inputs to make the neuron harder to fire. An alternative to this would be to simply raise the threshold each time the potential tries to go below the resting potential, then gradually bring it back down to normal with the same hardware as used for the decay. This may be a little more efficient in terms of hardware than the previous method, as it avoids the need for a dual-polarity decay function, and should produce the same result.

It is desirable to experiment with the bit-widths of the registers and arithmetic circuits, to investigate the trade-off between logic element usage and operating ability. It is expected that a neuron built with narrower buses and registers will not be able to perform as great a range of functions as one built with wider buses, as the range of possible values for the parameters will decrease, and it will be more difficult to obtain fine variations in weights and thresholds with a coarser quantisation of parameters. In this case having the synapses outside the main neuron model is desirable, as it is possible to vary the parameter width in these

without affecting the parameters in the neuron body. Although the synapses, like the neurons, were built with 16-bit registers and data paths, the parameters used in the tests rarely exceeded the capabilities of 10-bit registers, or 8-bit in the case of the L parameters. It was shown earlier that a reduction in parameter width leads to a reduction in LE usage, and this may be useful when implementing large networks.

When developing networks of neuron models there will need to be some means of control, to allow the parameters to be adjusted, either manually or by learning algorithms. A simple embedded controller was presented with the first neuron model, but this would be more useful implemented on a PC as a graphical interface. Some means of encoding inputs as spike trains and decoding spike trains to produce output signals would also be useful.

The neural circuits presented in section 5.18 could be developed further, to refine the existing circuits and to develop more simple building blocks which could be combined to create more complex systems. It would then be useful to compare these systems with similar systems produced by some manner of learning or evolution of the network, to determine whether the hand-built blocks would actually exist in an evolved system. This would require that systems are built to allow the network to learn, either by adding these to the neuron and synapse models, or by implementing a learning controller which can alter the weights through the neurons' data buses.

Chapter 6: Overall Conclusions

The experiments presented in this work have demonstrated various approaches to implementing systems which would traditionally be software-based. The image processing hardware presented in chapter 3 was implemented as a direct hardware translation of a software algorithm, while the neuron models were implemented as hardware systems exhibiting some degree of parallelism, in the case of the neuron with separate synapses. It was found that there are implementation issues common to both approaches, as well as some issues specific to a particular type of implementation.

The image processing hardware showed that the direct translation of the software algorithm to hardware resulted in a system with relatively low performance but with the advantage of small hardware size. Two implementations were demonstrated, and it was shown that both could perform the required function, extracting the skin lines from the image. The results showed that the two systems, high-pass filter and second-order filter, performed differently and produced different outputs. The second-order filter was seen to be more susceptible to noise and distortion in the source images, but produced a result image with more detail, due to its smaller mask size.

Methods of increasing the efficiency of the processor were shown, such as the use of space-inefficient but relatively fast divider logic to perform a division by a fixed divisor. Clearly this approach is not suitable for a general purpose divider where the divisor is variable, but in this particular instance, and in other similar cases, it is a good solution. The use of a software program to generate the VHDL code was shown, a useful feature of a hardware description language which can save a lot of time.

The second-order filter was seen to operate much more quickly than the high-pass filter, also requiring considerably less hardware. The increase in processing speed was due mainly to the smaller mask size, though a slightly more efficient control state machine assisted in this increase, as did the overall clock frequency increase enabled by the simpler hardware.

While the large hardware requirement of the divider in the high-pass filter reduced its area-time product score, the second-order filter performed well by this measure, due mainly to its very small hardware size. It was seen that although the lack of 'local' storage of pixels or partial sums meant more accesses to memory were required, thus increasing the processing time, the small number of registers required led to the hardware being considerably smaller than many other implementations of convolution filters. This reduction in hardware size is the principal advantage of the systems developed in this chapter.

The flexibility of the hardware developed for the image processor was demonstrated in the conversion of the system to process cellular automata. It was shown that the CA processor is a special case of image processor, which can operate in the same was as the basic image processors, with just a few modifications to the way memory is accessed. The system that was demonstrated was shown to be a versatile general-purpose CA processor which could be adapted to a wide range of different rules with no change to the architecture. Methods of accelerating the system were discussed, and it was shown that methods similar to those that can be applied to accelerate the image processing could also be used to accelerate the CA processing.

The development and use of a simple microprocessor in chapters 4 and 5 showed the ease with which software can be combined with custom hardware to create a more adaptable system. It was shown that a very simple structure lacking many of the parts of a conventional modern microprocessor, such as an instruction decoder or microcode, can still be useful as a general-purpose processor. Restricting the processor to an accumulator-based architecture did not prevent it from being useful, and the lack of instruction decoding allows the creation of often useful hybrid instructions which are not possible on a machine with a rigidly defined instruction set. The simplicity of the design resulted in a compact, useful processor with a high performance-to-area ratio when compared with many other embedded processors.

In chapter 5 a simplified spiking neuron model was developed, and it has been demonstrated that even though it is a good deal simpler than most artificial neuron models it can still behave in a manner similar to a real neuron, with some simplification. It has been shown that a biologically accurate model of the internal working is not required, and an approximation based on simple integer arithmetic will suffice.

Two models for the neuron were demonstrated, the first design being relatively simple and as a result somewhat limited. It was shown that a neuron model in which the synapses are multiplexed would have to have its parameters changed if extra inputs are added, as the extra inputs will lengthen the integration cycle. A second neuron model was presented in which the synapses are modelled separately, and perform their operations as true parallel processes, more closely resembling the real biological neuron.

It was also demonstrated that even this simple neuron model takes up enough space within the FPGA as to make a parallel implementation of a large network difficult. It is possible to build small networks with the model, and networks of over a hundred neurons, though not possible with the FPGA which was used for the experimental work, are possible with current FPGAs. The possible reduction in logic element usage with a reduction in the precision of the parameters used by the neuron models was discussed, and this was demonstrated in the case of the synapse model and also in the case of the RAM-based neuron model, the latter making use of the otherwise unused memory blocks in the FPGA in order to reduce its logic usage. The FPGA architecture's limitations pertaining to the neuron development were also discussed, as it was shown that while the embedded RAM blocks can be used by the neuron model itself, if the synapses are modelled separately the relatively coarse-grained nature of the Apex series FPGA's RAM blocks makes them unsuitable for use in the synapse model. Later FPGA types such as the Stratix series could be used to make a more efficient implementation using the RAM blocks for parameters.

The importance of error checking in the arithmetic used in a system such as the neuron was demonstrated by the incorrect operation of the neuron models under certain circumstances. The first model, lacking bounds checking, was shown to

behave incorrectly in response to inhibitory inputs which place the simulated membrane potential at a level the control logic is not equipped to cope with. The use of a variable decay slope also necessitated a degree of vagueness in the definition of the resting potential, this being defined by upper and lower bounds. It was suggested that this vagueness could lead to unpredictable operation, though this was not observed. The second, improved model addressed these issues, adding bounds checking and handling the decay of the potential in an alternative and more reliable way.

The neuron models which were developed were shown to be simple yet capable, and although they may be larger in terms of hardware size than some models from the literature the increased size is due to greater flexibility in the parameters, greater precision in the simulation of the membrane potential and a greater range of synaptic responses than simpler implementations such as [5.36]. The second model is also capable of supporting any number of inputs with no change in the integration cycle time, which is a side-effect of many of the simpler implementations [5.38]. They are also able to operate in a network without any explicit synchronisation, and thus the network is not performing a series of time-steps but is running in real-time, which is more biologically-realistic.

The simple neuron model was shown to be capable when used as part of a network of producing complex periodic and non-periodic dynamics. A small network was found to behave in a complex and apparently non-periodic way following simple stimulation, given sufficient feedback. It was demonstrated that the network appears to have stable states, which in this case are dynamically stable, repeating the same firing sequence with a short period. The network has more than one such stable state, and can be made to transition from one to another in an unpredictable manner by disrupting the pattern with external stimulation. This was seen to be consistent with behaviour seen in real-world neural circuits, notably the Central Pattern Generators found in many animals.

Construction of simple but novel neural circuits was demonstrated, showing that the neuron models can be used together to perform functions which are a good deal more complex than the simplicity of the model would suggest. These circuits

could be used as building blocks for larger systems, using the neurons instead of conventional logic circuits to take advantage of their complex time-dependant responses. Methods for implementing Boolean logic functions were also suggested.

In summary, we have seen methods of implementing systems in hardware which would traditionally have been software-based. The trade-offs between performance and hardware size were revealed, and methods for making efficient use of the FPGA's hardware were discussed. It was shown that making a direct translation from software to hardware produced a system with relatively small hardware size but also relatively low processing speed. It was shown however that a high ratio of performance to area can be achieved with such small hardware, even when the actual performance is not particularly high.

Exploiting parallelism to increase performance was discussed in the cases of the images processors and the neurons. The parallel nature of the neuron models shown in the demonstrated networks allows the system to perform in real-time with relatively low clock rates or to perform very fast with higher clock rates. The simplified neurons show promising and interesting results, as complex dynamics were obtained from both small and large neural systems. It was demonstrated that a network of these simplified high-performance neurons can demonstrate complex periodic and non-periodic dynamics. Simple neural circuits built with the developed models showed interesting results and abilities, and the potential to be developed further and used to create more complex systems.

Bibliography

[2.1] Altera, '**Apex 20K Programmable Logic Device Family datasheet**', Feb 2002
Available at www.altera.com

[2.2] Virtex documentation and datasheets available from www.xilinx.com

[2.3] **Altera Quartus II software** available from www.altera.com.

[2.4] VHDL Analysis and Standardization Group, **IEEE std. 1076-2002**, ,
http://www.eda.org/vhdl-200x/

[2.5] Haldar, M, Nayak, A, Shenoy, N, et al., **FPGA hardware synthesis from MATLAB**
VLSI DESIGN 2001: FOURTEENTH INTERNATIONAL CONFERENCE ON VLSI DESIGN : 299-304 2001

[2.6] Roy, S, Banerjee, P, **An algorithm for trading off quantization error with hardware resources for MATLAB-based FPGA design**
IEEE T COMPUT 54 (7): 886-896 JUL 2005

[2.7] **Handel-C**, part of the DK Design Suite from Celoxica Inc.,
http://www.celoxica.com/technology/c_design/handel-c.asp

[2.8] Lewis, E, **Reconfigurable Computer Hardware Design**, MEng dissertation, 2002

[2.9] Kramberger, I, Solar, M, **DSP Acceleration using A Reconfigurable FPGA**

[2.10] O. Albaharna, P. Cheung, and T. Clarke, "**Virtual hardware and the limits of computational speed-up**," in *Proc. IEEE Int. Symp. Circuits Syst.*, 1994, pp. 159–162.

[2.11] Shoa, A, Shirani, S, **Run-time reconfigurable systems for digital signal processing applications: A survey,** VLSI SIG PROC SYST 39 (3): 213-235 MAR 2005

[2.12] Galanis, MD, Dimitroulakos, G, Goutis, CE, **Performance improvements from partitioning applications to FPGA hardware in embedded SoCs,** J SUPERCOMPUT 35 (2): 185-199 FEB 2006

[2.13] Singh, S, Slous, R, **Accelerating Adobe Photoshop with reconfigurable logic**
IEEE SYMPOSIUM ON FPGAS FOR CUSTOM COMPUTING MACHINES, PROCEEDINGS : 236-244 1998

[2.14] Altera, '**Stratix II Device Handbook**', available from www.altera.com

[2.15] El Camino GmbH, **Digilab 20Kx240**, www.elcamino.de

[3.1] Round, AJ, Duller, AWG, Fish, PJ, **Lesion classification using skin patterning,** SKIN RES TECHNOL 6 (4): 183-192 NOV 2000

[3.2] She, ZS, Fish, PJ. **Analysis of skin line pattern for lesion classification,** SKIN RES TECHNOL 9 (1): 73-80 FEB 2003

[3.3] Vega-Rodriguez, MA, Sanchez-Perez, JM, Gomez-Pulido, JA, **An optimized architecture for implementing image convolution with reconfigurable hardware**
TSI PRESS S 16: 131-136 2004

[3.4] Hsiao, PY, , Chen, CH, et al., **Real-time realisation of noise-immune gradient-based edge detector,** IEE P-COMPUT DIG T 153 (4): 261-269 JUL 2006

[3.5] Torres-Huitzil, C, Arias-Estrada, M: **FPGA-based configurable systolic architecture for window-based image processing,** EURASIP J APPL SIG P 2005 (7): 1024-1034 MAY 11 2005

[3.6] Bosi, B, Bois, G, Savaria, Y: **Reconfigurable pipelined 2-D convolvers for fast digital signal processing,** IEEE T VLSI SYST 7 (3): 299-308 SEP 1999

[3.7] Gribbon, KT, Bailey, DG, Johnston, CT, **Using design patterns to overcome image processing constraints on FPGAs,** DELTA 2006: THIRD IEEE INTERNATIONAL WORKSHOP ON ELECTRONIC DESIGN, TEST AND APPLICATIONS : 47-53 2006

[3.8] Muthukumar, V, Rao, DV, **Image processing algorithms on reconfigurable architecture using HandelC,** PROCEEDDINGS OF THE EUROMICRO SYSTEMS ON DIGITAL SYSTEM DESIGN : 218-226 2004

[3.9] Benkrid, K, Belkacemi, SD, **Design and implementation of a 2D convolution core for video applications on FPGAs,** THIRD INTERNATIONAL WORKSHOP ON DIGITAL AND COMPUTATIONAL VIEDO, PROCEEDINGS : 85-92 2002

[3.10] Cardells-Tormo, F, Molinet, PL, **Area-efficient 2-D shift-variant convolvers for FPGA-based digital image processing,** 2005 IEEE WORKSHOP ON SIGNAL PROCESSING SYSTEMS - DESIGN AND IMPLEMENTATION (SIPS) : 209-213 2005

[3.11] Zhang, MZ, Ngo, HT, Livingston, AR, et al. **An efficient VLSI architecture for 2-D convolution with quadrant symmetric kernels,** IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, PROCEEDINGS : 303-304 2005

[3.12] Perri, S, Lanuzza, M, Corsonello, P, et al. **A high-performance fully reconfigurable FPGA-based 2D convolution processor,** MICROPROCESS MICROSY 29 (8-9): 381-391 NOV 1 2005

[3.13] Rosas, RL, de Luca, A, Santillan, FB, **SIMD architecture for image segmentation using Sobel operators implemented in FPGA technology**, 2005 2ND INTERNATIONAL CONFERENCE ON ELECTRICAL & ELECTRONICS ENGINEERING (ICEEE) : 77-80 2005

[3.14] Saldana, G, Arias-Estrada, M, **FPGA-based customizable systolic architecture for image processing applications**, 2005 INTERNATIONAL CONFERENCE ON RECONFIGURABLE COMPUTING AND FPGAS (RECONFIG 2005) : 17-24 2005

[3.15] F.G.Lorca, L Kessal and D.Demigny. "**Efficent ASIC and FPGA implementation of IIR filters for Real time edge detection**". In the International Conference on image processing (ICIP-97) Volume 2. Cct 1997

[3.16] Hsiao, PY, Li, LT, Chen, CH, et al., **An FPGA architecture design of parameter-adaptive real-time image processing system for edge detection**, 2005 EMERGING INFORMATION TECHNOLOGY CONFERENCE (EITC) : 38-40 2005

[3.17] Kraut, J, Author, Reprint Author Kraut Jay Kraut, Jay, **Hardware edge detection using an Altera Stratix nios2 development kit,** 2006 CANADIAN CONFERENCE ON ELECTRICAL AND COMPUTER ENGINEERING, VOLS 1-5 : 1172-1175 2006

[3.18] Altera, Application Note 364, "**Edge Detection Reference Design**", Oct 2004, available from www.altera.com

[3.19] Dick, C, "**Image processing on an FPGA based custom computing platform**" ISSPA 96 - FOURTH INTERNATIONAL SYMPOSIUM ON SIGNAL PROCESSING AND ITS APPLICATIONS, PROCEEDINGS, VOLS 1 AND 2 : 361-364 1996

[3.20] Amira, A, Bouridane, A, **An FPGA implementation of Discrete Hartley Transforms** SEVENTH INTERNATIONAL SYMPOSIUM ON SIGNAL PROCESSING AND ITS APPLICATIONS, VOL 1, PROCEEDINGS : 625-628 2003

[3.21] Uzun, IS, Amira, A, Bouridane, A, **FPGA implementations of fast Fourier transforms for real-time signal and image processing,** IEE P-VIS IMAGE SIGN 152 (3): 283-296 MAY 2005

[3.22] JASC Software Inc. Paint Shop Pro

[3.23] Weisstein, Eric W. "**Binomial Distribution.**" From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/BinomialDistribution.html

[3.24t] K.E. Batcher: **Sorting Networks and their Applications**. Proc. AFIPS Spring Joint Comput. Conf., Vol. 32, 307-314 1968

[3.25] Altera Inc. '**ByteBlaster MV data sheet**', www.altera.com

[3.26] http://www.altera.com/products/ip/communications/ipm-index.jsp

[3.27] Google Search: '**Skin Lesion**'

[3.28] She, ZS, Duller, AWG, Fish, PJ, **Enhancement of lesion classification using divergence, curl and curvature of skin pattern,** SKIN RES TECHNOL 10 (4): 222-230 NOV 2004

[3.29] John von Neumann, **Theory of Self-Reproducing Automata**, 1969

[3.30] Konrad Zuse, **"Rechnender Raum" (Calculating Space)**, 1969

[3.31] Alber, M, Kiskowskiy, M, Glazier, J, Jiang, Y, **ON CELLULAR AUTOMATON APPROACHES TO MODELING BIOLOGICAL CELLS**

[3.32] Ermentrout, G, Edelstein-Keshet, L: **Cellular Automata Approaches to Biological Modelling**

[3.33] Tomohiro Miura, Tai Tanaka, Yoshikazu Suemitsu and Shigetoshi Nara, "**Complete and compressive description of motion pictures by means of two-dimensional cellular automata**", Physics Letters A 346 (2005): 296--304

[3.34] Gardner, M, '**Wheels of Life and other Amusements**', Freeman, New York (1983)

[3.35] David Bell, **"Unit Life Cell"**, Jan 1996
http://www.radicaleye.com/lifepage/patterns/unitcell/ucdesc.html

[3.36] Berlekamp E.R, Conway J.H and Guy R.K., '**Winning Ways for your Mathematical Plays**', Academic Press, London

[3.37] Rendell, P. **"This is a Turing Machine Implemented in Conway's Game of Life."**
Available on the Internet at: `http://www.rendell.uk.co/gol/tm.htm`.
Paper available at: http://www.cs.ualberta.ca/~bulitko/F02/papers/tm_words.pdf

[3.38] Halbach, M, **Hoffmann, R. Implementing Cellular Automata in FPGA logic**
Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)

[3.39] Shackleford, B, Tanaka, M, Carter, R.J, Snider, G, **FPGA Implementation of Neighborhood-of-Four Cellular Automata Random Number Generators**

[3.40] Kobori, T, Maruyama, T, Hoshino, T, **"A Cellular Automata System with FPGA"**
Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)

[3.41] Life32: a Life engine for Windows, Johan C Bontes, 2002.
Available at life32.lifepatterns.net

[4.1] Altera, **NIOS Embedded Processor**
http://www.altera.com/products/ip/processors/nios/nio-index.html

[4.2] Xilinx, **Virtex 4 capabilities / PowerPC Processor**
http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/capabilities/powerpc.htm

[4.3] ARM, '**ARM9 Family**'
http://www.arm.com/products/CPUs/families/ARM9Family.html

[4.4] **PicoBlaze™ 8-bit Microcontroller Reference Design for FPGAs and CPLDs**:
www.xilinx.com/picoblaze

[4.5] www.cast-inc.com

[4.6] www.hitechglobal.com

[4.7] **Free6502 core**
http://www.sprow.co.uk/fpgas/free6502.htm

[4.8] http://zxgate.sourceforge.net/

[4.9] David Lynch, "**A Motorola MC68008 Op-code compatible VHDL Microprocessor**"
https://www.cs.tcd.ie/Michael.Manzke/fyp2003-2004/DavidLynch.pdf

[4.10] Wunderlich, RE, Hoe, JC, **In-system FPGA prototyping of an Itanium microarchitecture,** PR IEEE COMP DESIGN : 288-294 2004

[4.11] Neil Franklin, "**PDP-10 Clone Microprocessor in an FPGA**"
http://neil.franklin.ch/Projects/PDP-10/

[4.12] www.opencores.org

[4.13] John Kent : http://members.optushome.com.au/jekent/FPGA.htm

[4.14] Augusto, N, Cortes, M, Centoducatte, P, "**A CPU for Educational Applications Designed With VHDL and FPGA**"

[4.15] Mezei, I, Malbasa, V, **Using VHDL to improve an FPGA based educational microcomputer,** EUROCON 2005: THE INTERNATIONAL CONFERENCE ON COMPUTER AS A TOOL, VOL 1 AND 2 , PROCEEDINGS : 799-802 2005

[4.16] Romero-Troncoso, R. de J. , Ordaz-Moreno, A, et al., **8-bit CISC microprocessor core for teaching applications in the digital systems laboratory,** RECONFIG 2006: PROCEEDINGS OF THE 2006 IEEE INTERNATIONAL CONFERENCE ON RECONFIGURABLE COMPUTING AND FPGAS : 300-304 2006

[4.17] Gustin, V, Author, Reprint Author Gustin Veselko Gustin, Veselko , Bulic, P, et al. **Learning computer architecture concepts with the FPGA-based "move" microprocessor** COMPUT APPL ENG EDUC 14 (2): 135-141 AUG 2006

[4.18] Paul Stoffregen, "**OSU8 microprocessor**"
http://pjrc.com/tech/osu8/index.html

[4.19] M. Schoeberl, JOP: **A Java Optimized Processor for Embedded Real-Time Systems**, PhD thesis, Vienna University of Technology, 2005
From www.jopdesign.com

[4.20] Mattos, JCB, Carro, L, **Efficient architecture for FPGA-based microcontrollers** 2002 IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, VOL V, PROCEEDINGS : 805-808 2002

[4.21] Haskell, RE, Hanna, DM, **A VHDL-Forth core for FPGAs,** MICROPROCESS MICROSY 28 (3): 115-125 APR 23 2004

[4.22] Buss, F: **A Forth-like CPU,** http://www.frank-buss.de/forth/cpu1/

[4.23] F.C. Williams T. Kilburn & G.C. Tootill, **Universal High-Speed Digital Computers: A Small-Scale Experimental Machine,** Proc. of the I.E.E., Vol 98, Part II, No. 61, Feb. 1951

[4.24] **Nios II Processor Reference Handbook**, Altera http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

[5.1] Liu, Y., Guo, H. C., Zou, R., et al., **Neural network modeling for regional hazard assessment of debris flow in Lake Qionghai Watershed, China**, Environmental Geology (Berlin) 49 (7): 968-976 APR 2006

[5.2] Beltran, N. H., Duarte-Mermoud, M. A., Bustos, M. A., et al., **Feature extraction and classification of Chilean wines**, Journal of Food Engineering 75 (1): 1-10 JUL 2006

[5.3] "**Learning to Drive**", IET Engineering & Technology, May 2006

[5.4] Gurney, K, "**An Introduction to Neural Networks**", UCL Press 1997

[5.5] Dayan, P, Abbott, L.F., "**Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems**", MIT Press

[5.6] Maass, W. (1997) "**Networks of Spiking Neurons: The Third Generation of Neural Network Models**" Neural Networks, 10(9):1659–1671.

[5.7] Kunkle, D, Merrigan's, C, "**Pulsed Neural Networks and their application**" http://www.redfish.com/dkunkle/mypapers/pnn.pdf

[5.8] Lapicque, L. **Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarization**. J. Physiol. Pathol. Gen. 9:620–635; 1907.

[5.9] Abbott, LF, **Lapicque's introduction of the integrate-and-fire model neuron (1907)** BRAIN RES BULL 50 (5-6): 303-304 NOV-DEC 1999

[5.10] Hodgkin, A. L. and Huxley, A. F. (1952) "**Measurement of Current-Voltage Relations in the membrane of the Giant Axon of Loligo**", Journal of Physiology 116: 424-448

[5.11] Hodgkin, A. L. and Huxley, A. F. (1952) "**A Quantitative Description of Membrane Current and its Application to Conduction and Excitation in Nerve**" Journal of Physiology 117: 500-544

[5.12] FitzHugh R. (1961) **Impulses and physiological states in theoretical models of nerve membrane**. Biophysical J. 1:445-466

[5.13] Nagumo J., Arimoto S., and Yoshizawa S. (1962) **An active pulse transmission line simulating nerve axon**. Proc IRE. 50:2061–2070.

[5.14] A. Thompson. **An evolved circuit, intrinsic in silicon, entwined with physics.** In T. Higuchi, M. Iwata, and L. Weixin, editors, Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware (ICES 96), pages 390–405, Berlin, 1997. Springer.

[5.15] T. Fogarty, J. Miller, P. Thomson. **Evolving Digital Logic Circuits on Xilinx 6000 Family FPGAs**. In P.K. Chawdrhy, R. Roy, R.K. Pant (Eds.), Soft Computing in Engineering Design and Manufacturing, pages 299-305, Springer Verlag, London, 1998

[5.16] M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, and T. Higuchi. **Hardware evolution at functional level**. In International conference on Evolutionary Computation: The 4th Conference on Parallel Problem Solving from Nature, pages 62{71, 1996.

[5.17] D. Levi and S. Guccione. **Geneticfpga: Evolving stable circuits on mainstream fpga devices**. In The First NASA/DoD Workshop on Evolvable Hardware. IEEE Computer Society, 1999.

[5.18] J. Torresen. **Evolvable Hardware as a New Computer Architecture**. Proc. of the International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet, 2002

[5.19] G. Hollingworth, S. Smith, A. Tyrell. **Safe Intrinsic Evolution of Virtex Devices**. proceedings of 2nd NASA/DoD Workshop on Evolvable Hardware, pages 195-204, 2000

[5.20] T.G.W. Gordon, P.J. Bentley. **On Evolvable Hardware**. In S. Ovaska, L. Sytandera (Eds.), Soft Computing in Industrial Electronics, pages 279-323, Physica-Verlag, Heidelberg, Germany, 2002

[5.21] Lee, YJ, Lee, J, Kim, YB, et al. **Low power real time electronic neuron VLSI design using subthreshold technique**, 2004 IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, VOL 4, PROCEEDINGS : 744-747 2004

[5.22] Alice C. Parker, Aaron K. Friesz, and Afshaneh Pakdaman, **Towards a Nanoscale Artificial Cortex**: ww1.ucmss.com/books/LFS/CSREA2006/CDE8278.pdf

[5.23] Chicca, E. and Badoni, D. and Dante, V. and D'Andreagiovanni, M. and Salina, G. and Carota, L. and Fusi, S. and Del Giudice, P. **A VLSI recurrent network of integrate-and-fire neurons connected by plastic synapses with long term memory** , IEEE Transactions on Neural Networks, 14:(5) 1297-1307, Sep, 2003

[5.24] Sekerli, M, Butera, RJ, **An implementation of a simple neuron model in field programmable analog arrays**, P ANN INT IEEE EMBS 26: 4564-4567 Part 1-7 2004

[5.25] Indiveri, G, Chicca, E, Douglas, R, **A VLSI array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity**, IEEE T NEURAL NETWOR 17 (1): 211-221 JAN 2006

[5.26] A.F. Murray, A.V.W. Smith. **Asynchronous VLSI Neural Networks Using Pulse-Stream Arithmetic**. IEEE Journal of Solid-State Circuits, pages 688-697, 23: 3, 1988

[5.27] Vitabile, S, Conti, V, Gennaro, F, et al., **Efficient MLP digital implementation on FPGA**, DSD 2005: 8TH EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN, PROCEEDINGS : 218-222 2005

[5.28] J. G. Eldredge and B. L. Hutchings, "**Density enhancement of a neural network using FPGA's and run-time reconfiguration**," in Proc. IEEEWorkshop FPGA's Custom Computing Machines, D. A. Buell and K. L. Pocek, Eds., Napa, CA, Apr. 1994, pp. 180–188.

[5.29] S. Bade and B.L.Hutchings, "**FPGA based stochastic neural network implementation**," proc IEEE workshop on FPGAs for custom computing machines, pp.189-198, 1994.

[5.30] H. de Garis. **Growing an Artificial Brain with a Million Neural Net Modules Inside a Trillion Cell Cellular Automaton Machine**. Proc. of the Fourth International Symposium on Micro Machine and Computer Science, pages 211-214, 1993

[5.31] JVSPS, **"The CAM-Brain Machine (CBM) : Real Time Evolution and Update of a 75 Million Neuron FPGA-Based Artificial Brain"**, Hugo de Garis, Michael Korkin. (Accepted, to appear in Journal of VLSI Signal Processing Systems (JVSPS), Special Issue on Custom Computing Technology)

[5.32] Felix Gers, Hugo De Garis and Michael Korkin. **'Codi-1 Bit: A simplified cellular automata based neuron model.'** In Proceedings of AE97, Artificial Evolution Conference, October 1997

[5.33] Glackin, B, McGinnity, TM, Maguire, LP, et al., **A novel approach for the implementation of large scale spiking neural networks on FPGA hardware**, LECT NOTES COMPUT SC 3512: 552-563 2005

[5.34] Pearson, M, Gilhespy, I, Gurney, K, et al., **A real-time, FPGA based, biologically plausible neural network processor**, LECT NOTES COMPUT SC 3697: 1021-1026 2005

[5.35] Hellmich, HH, Geike, M, Griep, P, et al., **Emulation engine for spiking neurons and adaptive synaptic weights**, PROCEEDINGS OF THE INTERNATIONAL JOINT CONFERENCE ON NEURAL NETWORKS (IJCNN), VOLS 1-5 : 3261-3266 2005

[5.36] D. Roggen, S. Hofmann, Y. Thoma, and D. Floreano. **Hardware spiking neural network with run-time reconfigurable connectivity in an autonomous robot**. In NASA/DoD Conf. on Evolvable Hardware, pages 189–198, 2003.

[5.37] Bellis, S, Razeeb, KM, Saha, C, et al., **FPGA implementation of spiking neural networks - an initial step towards building tangible collaborative autonomous agents** 2004 IEEE INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE TECHNOLOGY, PROCEEDINGS : 449-452 2004

[5.38] Upegui, A, Pena-Reyes, CA, Sanchez, E, **A hardware implementation of a network of functional spiking neurons with hebbian learning**, LECT NOTES COMPUT SC 3141: 233-243 2004

[5.39] Upegui, A, Pena-Reyes, CA, Sanchez, E, **An FPGA platform for on-line topology exploration of spiking neural networks**, MICROPROCESS MICROSY 29 (5): 211-223 JUN 1 2005

[5.40] E. Ros, R. Agis, R. R. Carrillo E. M. Ortigosa. **Post-synaptic Time-Dependent Conductances in Spiking Neurons: FPGA Implementation of a Flexible Cell Model**. Proceedings of IWANN'03: LNCS 2687, pp 145-152, Springer, Berlin, 2003.

[5.41] Savich, AW, Author, Reprint Author Savich, Antony W. , Moussa, M, et al., **The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study**, IEEE T NEURAL NETWOR 18 (1): 240-252 JAN 2007

[5.42]: Schrauwen, B, Van Campenhout, J, **Parallel hardware implementation of a broad class of spiking neurons using serial arithmetic**, ESANN'2006 proceedings - European Symposium on Artificial Neural Networks, Bruges (Belgium), 26-28 April 2006, d-side publi., ISBN 2-930307-06-4.

[5.43] O. Torres, J. Eriksson, J. M. Moreno, and A. Villa. **Hardware optimization and serial implementation of a novel spiking neuron model for the POEtic tissue**. Submitted to IPCAT'03.

[5.44] J.O. Hamblen & M.D. Furman, **'Rapid Prototyping of Digital Systems'**, Kluwer, Boston, 2001

[5.45] Indiveri, G. **VLSI reconfigurable networks of integrate-and-fire neurons with spike-timing dependent plasticity** , The Neuromorphic Engineer Newsletter, 2:(1) 4,7, 2005, The Neuromorphic Engineer Newsletter

[5.46] Christodoulou, C, Bugmann, G, Clarkson, TG, **A spiking neuron model: applications and learning**, NEURAL NETWORKS 15 (7): 891-908 SEP 2002

[5.47] Hebb, D. O. (1949). **The organization of behavior: A neurophychological study**. New York: Wiley-Interscience.

[5.48] Saudargiene, A, Porr, B, Worgotter, F, **How the shape of pre- and postsynaptic signals can influence STDP: A biophysical model**, NEURAL COMPUT 16 (3): 595-625 MAR 2004

[5.49] Birzniece, V, Backstrom, T, Johansson, IM, et al. **Neuroactive steroid effects on cognitive functions with a focus on the serotonin and GABA systems**, BRAIN RES REV 51 (2): 212-239 AUG 2006

[5.50] Watts, D. J. Strogatz, S. H. "**Collective Dynamics of Small-World Networks**." Nature 393, 440-442, 1998.

[5.51] B. J. Norris, A. L. Weaver, L. G. Morris, A. Wenning, P. A. Garcia, and R. L. Calabrese, **A Central Pattern Generator Producing Alternative Outputs: Temporal Pattern of Premotor Activity**, J Neurophysiol, July 1, 2006; 96(1): 309 - 326.

[5.52] Alfons Schnitzler, Lars Timmermann, Joachim Gross, **Physiological and pathological oscillatory networks in the human motor system**, Journal of Physiology - Paris 99 (2006) 3–7

[5.53] A. A.V. Hill, M. A. Masino, and R. L. Calabrese, **Intersegmental Coordination of Rhythmic Motor Patterns**, J Neurophysiol, August 1, 2003; 90(2): 531 - 538.

[5.54] Roberts A (2001) **Early functional organization of spinal neurons in developing lower vertebrates**. Brain Res Bull 53: 585–590

[5.55] Selverston, AI, Author, Reprint Author Selverston Allen I. Selverston, Allen I. , Ayers, J, et al. **Oscillations and oscillatory behavior in small neural circuits**. BIOL CYBERN 95 (6): 537-554 DEC 2006

[5.56] J.M.T. Thompson and H.B. Stewart "**Nonlinear dynamics and chaos: geometrical methods for engineers and scientists**", Wiley, 1986.

[5.57] K. Aihara, G. Matsumoto, **Chaotic oscillations and bifurcations in squid giant axons**, in: A.V. Holden (Ed.), Chaos, Manchester University Press, Princeton University Press, Manchester, Princeton, 1986, pp. 257–269

[5.58] AIHARA K, MATSUMOTO G, IKEGAYA Y, **PERIODIC AND NON-PERIODIC RESPONSES OF A PERIODICALLY FORCED HODGKIN-HUXLEY OSCILLATOR**, Journal of Theoretical Biology 109 (2): 249-270 1984

[5.59] Bohossian, V, Hasler, P, Bruck, J, **Programmable neural logic**, IEEE T COMPON PACK B 21 (4): 346-351 NOV 1998

[5.60] E. Allender, **A note on the power of threshold circuits**, in Proc. 30th IEEE Symp. Foundations Comput. Sci., 1989.

[5.61] Joye N, Schmid, A, Leblebici, Y, Asai, T, Amemiya, Y, **Fault-Tolerant Logic Gates using Neuromorphic CMOS circuits**, Presented at: IEEE 3rd Conference on Ph.D. Research in Microelectronics and Electronics, Bordeaux, France, July 2-5, 2007

Appendix A: Extracts from QBasic Software

A.1: Extract from the divider generator code

This program produces the VHDL divider code. The full 32,768 line LUT is generated, which was then trimmed manually to remove the lines which were not required.

```
OPEN "div81.vhd" FOR OUTPUT AS 1
quot$ = CHR$(34)
ta$ = CHR$(9)
PRINT #1, "library ieee;"
PRINT #1, "use ieee.std_logic_1164.all;"
PRINT #1, ""
PRINT #1, "entity div81 is"
PRINT #1, "port ("
PRINT #1, ta$ + "i: in std_logic_vector(14 downto 0);"
PRINT #1, ta$ + "o: out std_logic_vector(7 downto 0));"
PRINT #1, "end div81;"
PRINT #1, ""
PRINT #1, "architecture stuff of div81 is"
PRINT #1, "begin"
PRINT #1, ta$ + "process(i)"
PRINT #1, ta$ + ta$ + "begin"
PRINT #1, ta$ + ta$ + ta$ + "case i is"
FOR m1 = 0 TO 32767
      z = m1
      f = INT(m1 / 81)
      dec = z
      GOSUB convrt
      ad$ = decbin$
      dec = f
      GOSUB convrt
      da$ = decbin$
      ad$ = "000000000000000000" + ad$
      ad$ = RIGHT$(ad$, 15)
      da$ = "0000000000000000" + da$
      da$ = RIGHT$(da$, 8)
      a$ = "when " + CHR$(34) + ad$ + CHR$(34) + " => o <= " +
CHR$(34) + da$ + CHR$(34) + ";"
      PRINT a$
      PRINT #1, ta$ + ta$ + ta$ + a$
      NEXT m1
PRINT #1, ta$ + ta$ + ta$ + "when others => o <= " + CHR$(34) +
STRING$(8, "0") + CHR$(34) + ";"
PRINT #1, ta$ + ta$; "end case;"
PRINT #1, ta$ + "end process;"
PRINT #1, "end stuff;"
CLOSE 1
END
```

A.2: Examples of code for building the CA rule table from a rule definition

This is an example of the program that generated the Life rule table. The lines marked in bold are changed to suit the particular cellular automaton being implemented.

```
OPEN "life.mif" FOR OUTPUT AS 1

PRINT #1, "WIDTH = 8;"
PRINT #1, "DEPTH = 512;"
PRINT #1, "ADDRESS_RADIX=UNS;"
PRINT #1, "DATA_RADIX=HEX;"
PRINT #1, "CONTENT BEGIN"

FOR q = 0 TO 511
      dec = q
      GOSUB convrt
      b$ = "0000000000" + bin$
      b$ = RIGHT$(b$, 9)
      c$ = RIGHT$(b$, 8)
      sum = 0
      FOR f = 1 TO 8
            n$ = MID$(c$, f, 1)
            IF n$ = "1" THEN sum = sum + 1
      NEXT f
      IF q > 255 THEN centre = 1 ELSE centre = 0

      npix$ = "00"
      IF centre = 0 AND sum = 3 THEN npix$ = "FF"
      IF centre = 1 AND (sum = 2 OR sum = 3) THEN npix$ = "FF"
      v$ = STR$(q) + " : " + npix$ + ";"
      PRINT #1, v$
      NEXT q
      PRINT #1, "END;"
      CLOSE 1
END

convrt:
        bin$ = ""
        h$ = HEX$(dec)
        FOR i% = 1 TO LEN(h$)
                digit% = INSTR("0123456789ABCDEF", MID$(h$, i%,
1))-1
                IF digit% < 0 THEN bin$ = "": EXIT FOR
                j% = 8: k% = 4
                DO
                bin$ = bin$ + RIGHT$(STR$((digit% \ j%) MOD 2), 1)
                        j% = j% - (j% \ 2): k% = k% - 1
                        IF k% = 0 THEN EXIT DO
                LOOP WHILE j%
        NEXT i%
        decbin$ = bin$
RETURN
```

For the majority rule, the lines in bold above are replaced by:

```
sum = 0
FOR f = 1 TO 9
n$ = MID$(b$, f, 1)
IF n$ = "1" THEN sum = sum + 1
NEXT f
npix$ = "00"
IF sum > 4 THEN npix$ = "FF"
```

For the simulated annealing the last line of this is changed to:

```
IF sum > 5 OR sum = 4 THEN npix$ = "FF"
```

For the hourglass rule the section becomes:

```
c$ = MID$(b$, 1, 1)
se$ = MID$(b$, 2, 1)
s$ = MID$(b$, 3, 1)
sw$ = MID$(b$, 4, 1)
e$ = MID$(b$, 5, 1)
w$ = MID$(b$, 6, 1)
ne$ = MID$(b$, 7, 1)
n$ = MID$(b$, 8, 1)
nw$ = MID$(b$, 9, 1)

npix$ = "00"
nb$ = e$ + w$ + s$ + n$ + c$
PRINT nb$
IF nb$ = "00001" OR nb$ = "00010" OR nb$ = "00011" OR nb$ =
"01011" OR nb$ = "10101" OR nb$ = "11001" OR nb$ = "11101" OR nb$
= "11110" OR nb$ = "11111" THEN
npix$ = "FF"
END IF
```

These examples show the various ways in which a rule can be defined. The rest of the program builds the look-up table in Altera's Memory Initialisation File (MIF) format.

Appendix B: Assembler Mnemonics file for the VHDL Microprocessor

```
LDA,abs,011D
LDA,idx,811D
LDA,imm,211D
STA,abs,4200
STA,idx,C200
ADD,abs,0110
ADD,idx,8110
ADD,imm,2110
SUB,abs,0111
SUB,idx,8111
SUB,imm,2111
ADC,abs,0112
ADC,idx,8112
ADC,imm,2112
AND,abs,0113
AND,idx,8113
AND,imm,2113
OR,abs,0114
OR,idx,8114
OR,imm,2114
XOR,abs,0116
XOR,idx,8116
XOR,imm,2116
NOT,imp,0115
SHL,imp,0017
SHR,imp,0018
ROL,imp,0019
ROR,imp,001A
RXL,imp,011B
RXR,imp,011C
LDP,abs,002D
LDP,idx,802D
LDP,imm,202D
STP,abs,5200
STP,idx,D200
ADDP,abs,1020
ADDP,idx,9020
ADDP,imm,3020
SUBP,abs,1021
SUBP,idx,9021
SUBP,imm,3021
SLP,imp,1027
SRP,imp,1028
SKN,imp,0080
SKP,imp,0081
SKCS,imp,0082
SKCC,imp,0083
SKZ,imp,0084
SKNZ,imp,0085
JMP,imp,0840
CMP,abs,0101
CMP,idx,8101
CMP,imm,2010
LDPF,abs,012D
LDPF,imp,812D
LDPF,imp,212D
CMPP,abs,1101
CMPP,idx,9101
CMPP,imm,3101
HLT,imp,0400
JSR,abs,084E
RTS,imp,004F
```

End of file