Development of a Parallel Image Processing Architecture in VHDL

Stephen Rose
Supervisor: Prof Thomas Bräunl

1st June 2012

Abstract

Image processing tasks such as filtering, stereo correspondence and feature detection are inherently highly parallelisable. The use of FPGAs (Field Programmable Gate Arrays), which can be operated in highly parallel configurations, can thus be a useful approach in imaging applications. Many image processing tasks fall under the category of SIMD (Single Instruction Multiple Data) operations, which are a weakness of conventional CPUs, leading to the application of technologies such as FPGAs, Graphics Processing Units (GPUs), and Digital Signal Processors (DSPs) [1, 2]. The programmable blocks of an FPGA, and their interconnections, can be configured to optimise the device's performance at specific tasks. This means that commodity FPGAs can approach the speed of ASICs (Application Specific Integrated Circuits) in these applications, while retaining far greater flexibility. For embedded systems in particular, this performance can enable rapid vision processing that would be impossible on standard embedded microprocessors [3, 4].

This project has focussed on the development of a synthesisable VHDL design suitable for use as a general purpose image co-processor. In this approach, the FPGA's Logic Blocks (CLBs) are allocated to discrete Processing Elements (PEs) that are connected in a grid configuration. Each PE corresponds to a single pixel of the target image, simplifying the adaptation of common image processing algorithms. The results of this implementation suggest that while theoretically, such a design has significant advantages, current FPGA hardware is insufficient for the real-world application of this general-purpose system at high resolutions.

Acknowledgements

I would like to extend my thanks to Professor Dr. Thomas Bräunl for his ongoing assistance and supervision, as well as for providing the opportunity to work on such an interesting topic.

I would also like to thank FAHSS IT for allowing me such flexibility to deal with my study and research obligations.

Finally, I would like to thank my family and friends for their patience and support not only through this year, but through my entire time at UWA.

Contents

Abbreviations Used

ASIC Application Specific Integrated Circuit

CLB Configurable Logic Block

CPU Central Processing Unit

DDR Double Data Rate SDRAM

DSP Digital Signal Processor

FILO First In Last Out

FPGA Field Programmable Gate Array

HDL Hardware Design Language

IC Integrated Circuit

ISE Integrated Software Environment

LUT Lookup Table

PA Processing Array

PE Processing Element

RTL Register-Transfer Level

SIMD Single Instruction Multiple Data

VHDL VSIC Hardware Design Language

VSIC Very-high Speed Integrated Circuit

1 Background

The aim of this project is to design and implement a highly-parallel image processing architecture in VHDL, suitable for implementation on a Xilinx Field-Programmable Gate Array (FPGA). The eventual implementation is for real-time image processing to take place on the FPGA device, then for processed information to be communicated to a central processor for display and actioning. This project covers the initial VHDL implementation of a general purpose image co-processor. The architecture itself, as well as the implemented instruction set is discussed, and simulated examples are used to demonstrate the capabilities of such a system at accomplishing common image processing tasks. The results of the design synthesis are then explored, along with the ramifications of these results.

1.1 Image Processing

Vision processing is an application than can benefit significantly from parallelisation, with similar or identical steps having to be performed on each pixel or block of pixels that make up each image. This is especially true for what is known as low-level image processing operations [5, 6]. For these low-level operations that do only operate on neighbouring pixels, the use of a mesh connected processing array can achieve a very high level of performance[7]. There are also dedicated SIMD processors which offer significant image-processing parallelism, at high speeds, but with significantly less flexibility than an FPGA-based approach [8].

1.2 FPGAs

Field Programmable Gate Arrays are hardware units that can be reconfigured to perform a variety of digital logic functions. While Field Programmable Gate Arrays (FPGAs) may not achieve the performance of dedicated hardware, this technology avoid the onerous costs of fabrication for small-run integrated circuits [9]. FPGAs also have other benefits, offering a significant degree of flexibility through

the design process, such as a choice of Hardware Description Languages (HDLs), and a tight design loop, as repeated design iterations do not require physical fabrication or reconfiguration[10]. There is also the potential for field reprogramming, so that updated designs can be easily integrated into pre-existing systems, or dynamic reconfiguration, to adapt the FPGA to particular tasks [11, 12]. More specialised image processing implementation can take advantage of these DSP slices, which are automatically used when appropriate. There is widespread industry integration of FPGAs and cameras to create "Smart Camera" systems that can allow pre-defined image processing and filtering tasks to take place in real time [13, 14].

1.2.1 Logic Area

As discussed above, FPGAs are available with different numbers and configurations of CLBs. The FPGA also requires the use of these blocks to enable additional routing between units, as well as connections to other blocks such as RAM and IO [11]. The availability of logic blocks forms a key part of the design challenges faced when developing designs for FPGA synthesis. As the node-size of FPGA logic blocks is reduced, the affordability and logic density of FPGAs improves, and complex designs become more feasible to implement [15]. there have also been advances in FPGA packaging, which allow multiple dies to be treated as a single large FPGA device [16]. This sort of approach reduces costs, by maintaining small die sizes, while still increasing FPGA logic capacity[16].

Spartan-6 XC6SLX45 The XC6SLX45 is a mid-range entry in the Xilinx Spartan-6 range, sharing similar features to the other devices in the family. These include a large number of logic units, as well as specialised units such as Digital Signal Processor (DSP) cores, serial communications units, and memory controllers. The XC6SLX45 offers a total of 6822 logic slices, distributed throughout the device [17]. Pairs of these slices create a Configurable Logic Block (CLB), of varying degrees of complexity (for example, with or without shift registers, wide multi-

plexers or a carry path) [15]. Of most importance for this project is the four LUTs (Lookup Tables) contained within each slice; these LUTs are the primary method through which FPGAs implement digital logic.

1.2.2 Operating Clock Speed

FPGAs operate at lower clock speeds than conventional CPUs. For a synchronously clocked design, the rate at which the FPGA can operate is limited by the number of nested logic stages that must be implemented per clock cycle. This is to ensure that the last logic stage has settled before the output changes again [11]. Through the synthesis of FPGA designs, the critical path (the longest nested chain of logic) can be measured and constrained, to ensure that the design can operate at an acceptable rate. Timing analysis of a synthesised FPGA design includes the delay due to logic units, as well as the delays introduced by the routing of signals throughout the FPGA. The worst case total delay for a single-clocked signal is the limiting factor on the design's operational speed.

1.3 HDLs

Hardware Description Languages (HDLs) offer one way for the required behaviour and structure of an FPGA to be entered. Synthesis is the process of converting behavioural and structural HDL descriptions into a description of physical configuration [18]. In the case of FPGA development, the synthesised code can be transmitted as a bitstream that is sent to the FPGA, reconfiguring the logic blocks (CLBs), connections between them, and other components such as I/O blocks [15]. During the synthesis stage, the utilisation of various FPGA elements is also calculated, including the routing between units. The structure and behaviour of designs will generally be altered to better "fit" the target device. This optimisation may require changes to be made to the structure of algorithms, or the layout of the FPGA, with small changes having the potential to improve performance by orders of magnitude [19].

1.3.1 VHDL

VHDL was originally developed to describe the behaviour of standard Integrated Circuits (ICs), and was later extended by the IEEE to enable a subset of this language to be used in the synthesis of FPGA designs [18]. VHDL is a high-level, strongly typed language, with its general syntax inherited from ADA. In addition to its use in describing synthesisable behaviours and architectures, VHDL can also be used to generate tests. VHDL code used for testing can generally not be implemented onto hardware (synthesised), but can be extremely useful for investigating the behaviour of systems under development. The VHDL test code that was used to simulate various behaviours of the system is included in the relevant sections.

1.4 Higher Level Approaches

There has been significant development in the area of higher-level languages that can be synthesised for FPGA usage. Generally such languages simplify the design process, and can then be converted into a low-level description for final optimisation and synthesis. Two such languages, both variants of C, are SA-C (Single Assignment C) and Handel-C [20, 21]. The intention of such languages is to provide an "algorithm-level" interface for implementing image processing tasks [7]. Tools such as Matlab also provide paths for FPGA synthesis, simplifying many image processing implementations.

2 The Architecture

A suitable architecture for extremely parallel image processing will be recognisant of the structure of image data, as well as the properties of common image processing tasks. The chosen architecture can best be described as processing array (PA) made up of a large number of simple processing elements (PEs). Each unit corresponds to a single pixel of the image frame (or subset of the frame). Each PE is connected to its immediate neighbours via one of four ports, these connections enable the values of the neighbouring pixels to be directly used in operations being applied to the central pixel. This sort of structure is effective for SIMD processes, as the same operation can be performed on all units simultaneously. The concept of every PE corresponding directly to an image pixel simplifies the understanding of the system, and allows the straightforward implementation of many common image processing tasks. The development of this kind of massively parallel image processing architectures are an area of significant current research, with a wide variation in design choices such as PE capabilities, interconnectivity and overall system scale [22, 23]. Such digital systems must balance limited logical and physical resources to achieve the most useful combination of the utility of individual PEs, and the number of PEs that can be affordably included within the system. There is also some application of mixed digital and analog approaches, aimed at reducing the resource consumption of each PE, as well as overall power consumption [24].

2.1 Processing Elements

Each processing element is capable of performing basic arithmetic functions, and has a small amount of local storage. The initial implementation has two 8-bit registers, labelled A and B. Register A acts as an accumulator, storing the result of arithmetic operations. The current value stored in A is accessible to the neighbouring processing units following each clock cycle. The B register supplied additional operands, and can also be loaded with the values of an adjacent

PE's register A, so that these values can be used in future calculations. There is also a stack that is available for local storage, shared between A and B. The synthesis of one PE is shown in Figure 1, this schematic shows the large number of logic elements that are required for every PE in the system.

2.1.1 Inputs and Outputs

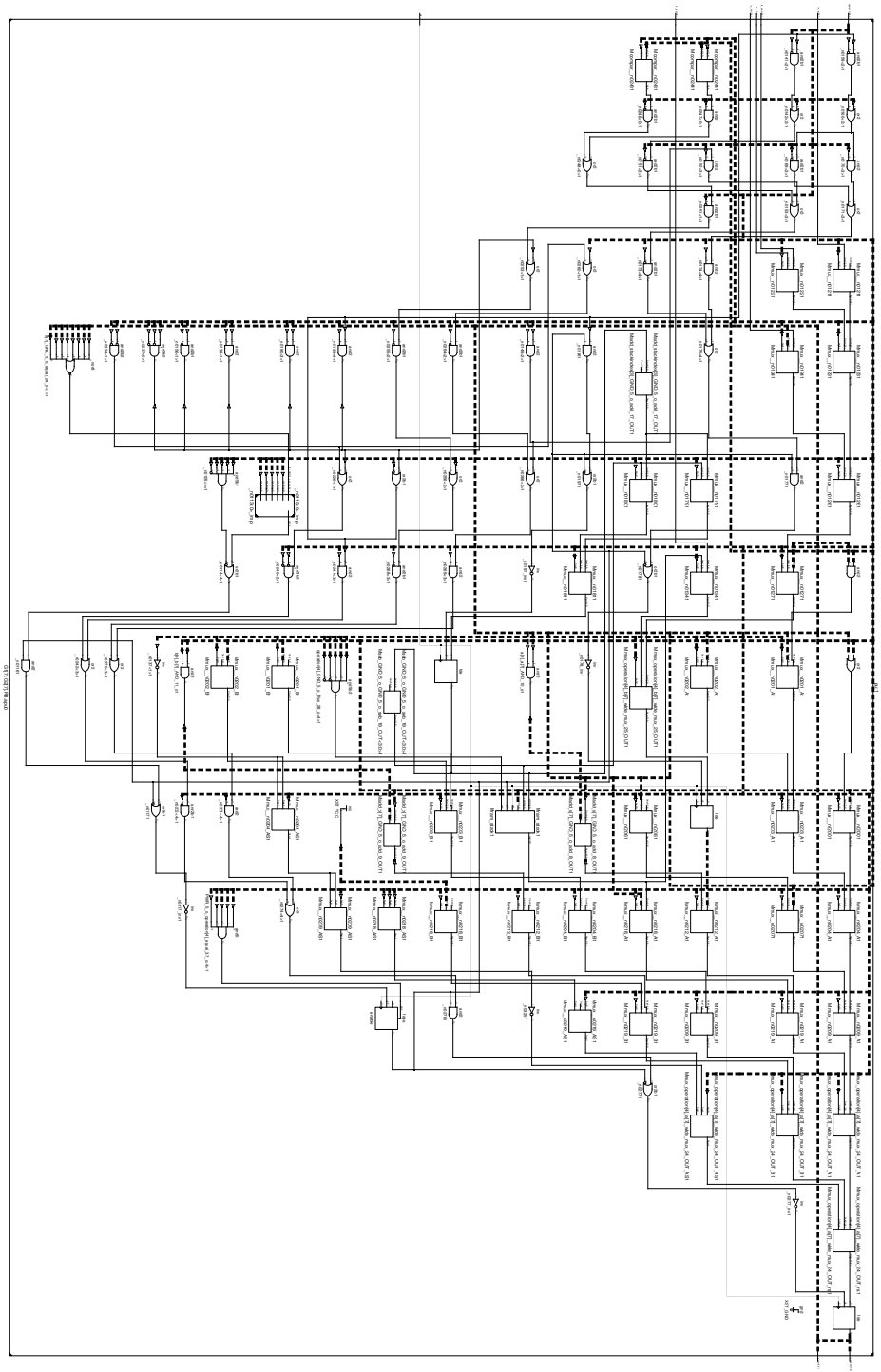
Multiple Directional Inputs (up to 4x8 bits): All processing elements have at least two directional inputs that allow simple access to the A value of cardinally adjacent processing elements. For the common case of a non-edge processing unit, there are four of these inputs, one from each of the PE above, below, to the left and to the right.

Single Chained Input (8 bits): To allow a general image loading instruction, this port was created. This port is connected to the "next" PE, which is generally the unit directly to the right, except in the case of the PEs along the right edge of the array, in which case it is connected to the leftmost PE on the line below. The use of this port to populate the processing array is shown in section 3.1.

Single Directional Output (8 bits): As the processing element will be outputting an identical signal (the current value of A) to each of the neighbouring units, a single output suffices in the general case. This output signal is connected to the directional inputs of adjacent units.

Operation Input (5 bits): Each PE is wired to receive the same instruction. However, it is likely that the result of certain processing units will be eventually discarded. Under a serially computing image processing scenario, these pixels would not be operated on, however, due to the highly parallel nature of this architecture, there is no performance penalty for processing excess pixels, as long as there are sufficient processing units available. The number and complexity of the operations implemented directly affect the logic resources that each PE will

Figure 1: Expanded RTL Schematic of a Processing Element



consume once synthesised, creating a key design trade-off that is explored in later sections. The implemented operations are shown in Table 1 on page 10.

2.1.2 Internal Signals

Accumulator A (8 bits): The accumulator stores the result of arithmetical and logical operations.

Register B (8 bits): Register B is used to provide an additional operand for arithmetic and logical operations. This is also where values from neighbouring processing elements are initially loaded.

Stack (16x8bits): The stack gives FILO storage that can be expanded to occupy the total distributed RAM allocated to each CLB. The precise allocation of this storage differs for each FPGA model, with the XC6SLX45 having 16x8-bit, dual-port distributed RAM blocks. It is also possible to force this storage to the FPGA's shared block RAM, or even to external DDR, if additional storage is required. It is possible to push values to the shared stack from A or B, and pop values from the stack to A or B.

Enable Bit: For a significant number of image processing tasks, every processing element is concurrently performing an identical operation. For the cases where this is not appropriate, an enable bit is included. In the current implementation, this signal is internal to the PE itself, and is set according to the value of accumulator A, via the Zero Flag. There is a single instruction to re-enable all PEs, ensuring a known configuration, once other tasks have been previously performed.

Zero Flag The zero flag is set when the value of accumulator A is zero. It is then possible to use this flag to selectively disable PEs.

2.1.3 Choice of Instruction Set

Due to the nature of FPGAs, there is a fundamental design compromise required between the complexity of each processing element, and the number of elements that are used in a given grid (which determines the resolution). The complexity (FPGA area) of each element corresponds to the number and complexity of operations it is required to perform [25]. FPGAs of a given series will come in a range of models, with a differing number and distribution of logic blocks. The instruction that requires the greatest number of cascaded logic changes is the factor determining the speed at which the system as a whole can operate. For example, an instruction to find the median of the current PE's value, and that of the the PEs directly adjacent to it would require a greater number of logic stages than any other instruction described. While such an instruction would be a significant advantage when median filtering (section 3.3.2) is required, its inclusion would be of significant detriment to the system's general purpose performance. For certain scenarios this trade-off is acceptable, particularly if the final application of the system is known in detail. It is also possible to avoid this trade-off, with some systems taking advantage of the reconfigurable nature of FPGAs by dynamically reconfiguring the deployed system to accomplish a specific task [11, 12, 26]. For the purposes of this image processing system, the performance of general instructions has been prioritised over performance at specific tasks.

2.2 The Processing Array

Many common image processing tasks are particularly localised, operating on adjacent groups of pixels, while generally having minimal input from pixels that are further away. Under this massively parallel architecture, a Processing Array is created, consisting of a large number of linked Processing Elements (PEs). Each PE is connected to its immediate cardinal neighbours (that is, to the PEs above, below, and to the left and right). These connections enable the simple implementation of many common image processing processes. There are also additional

Table 1: Processing Element Instruction List

OpCode	Instruction	Description	Dataflow
0	NOP	No operation	-
1	CopyBA	Copy from B to A	a $\Leftarrow$ b
2	CopyAB	Copy from A to B	b $\Leftarrow$ a
3	OrBA	Bitwise OR	a $\Leftarrow$ a or b
4	AndBA	Bitwise AND	a $\Leftarrow$ a and b
5	XorBA	Bitwise XOR	a$\Leftarrow$ a $\oplus$ b
6	SumBA	Add B to A	a $\Leftarrow$ a+b
7	SubBA	Subtract B from A	a $\Leftarrow$ a $-$ b
8	AbsA	Absolute Value of A	a $\Leftarrow |a|$
9	ClrA	Clear A	a $\Leftarrow$ 0
10	ClrB	Clear B	b $\Leftarrow$ 0
11	LdUp	Load from PE above into B	b $\Leftarrow$ above.a
12	LdDown	Load from PE below into B	b $\Leftarrow$ below.a
13	LdLeft	Load from PE to the left into B	b $\Leftarrow$ left.a
14	LdRight	Load from PE to the right into B	b $\Leftarrow$ right.a
15	Load	Load from Previous PE into B	b$\Leftarrow$ prev.a
16	Swap	Swap values of A and B	b $\Leftrightarrow$ a
17	DblA	Double the value of A	a $\Leftarrow$ a $\times$ 2
18	HalveA	Halve the value of A	a $\Leftarrow$ a $\div$ 2
19	MaxAB	Return maximum of A and B	a$\Leftarrow$ max (a, b)
20	MinAB	Return minimum of A and B	a$\Leftarrow$ min (a, b)
21	PushA	Push value of A to stack	Stack $\Leftarrow$ a
22	PopA	Pop value of A from stack	a $\Leftarrow$ Stack
23	PushB	Push value of B to stack	Stack $\Leftarrow$ b
24	PopA	Pop value of B from stack	b $\Leftarrow$ Stack
25	IncA	Increment A	a $\Leftarrow$ a+1
26	DecA	Decrement A	a $\Leftarrow$ a$-$1
27	DisableZ	Disable PEs with a=0	-
28	EnableAll	Enable all PEs (default)	-

connections throughout the grid to enable the loading of image information, the distribution of the clock and instructions, as well as the output of results. Due to the finite dimensions of the modelled grid, there are nine different configurations of processing element (Figure 2). For each unique edge or corner of the grid structure, a slightly different configuration of processing element is generated. N is the dimension of the processing array, i.e. an N-size array contains N^2 processing elements. This prevents a processing element from attempting to read inputs from elements that do not exist. For example the "RT" (Right-Top) processing element does not have an in_right or in_above port instantiated, as there are not appropriate neighbouring ports to connect to. A three PE by three PE array has be used to simply demonstrate the operation and validity of this architecture for basic operations, while larger systems have been synthesised as required. In addition to the required changes to the directional input port configuration for each of these nine cases, the Left Top (LT) element, and the Right Bottom (RB) element are used to output and input image data for the entire grid, and are connected directly to the outer structure of the processing array.

Figure 2: The Nine PE Configurations (with example coordinates)

LT (0, 0)	CT (X, 0)	RT (N-1, 0)
LC (0, X)	CC (X, X)	RC (N-1, X)
LB (0, N-1)	CB (X, N-1)	RB (N-1, N-1)

2.2.1 VHDL Generation of the Array

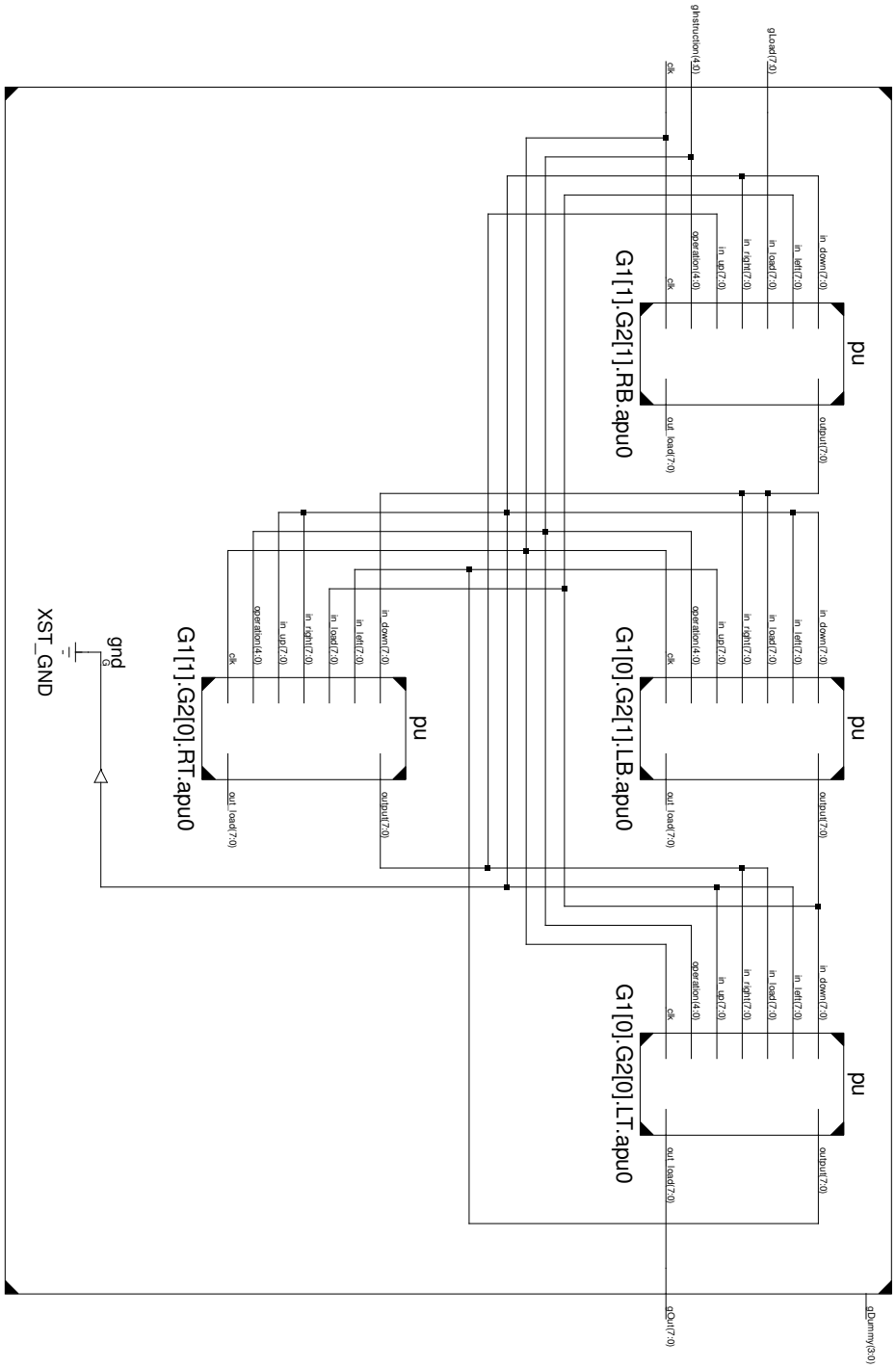
The required processing elements are generated through a pair of nested generate statements. As the synthesiser iterates through the nested loops, PEs of the required configuration are generated and linked via an array of output signals. Every processing element will be of one of the configurations shown in Figure 2, with the exact configuration determined by the values of the generate loop variables, as shown in Algorithm 1.

Algorithm 1 Generation of Processing Elements

```
G1 : for i in N-1 downto 0 generate--In X direction
G2 : for j in N-1 downto 0 generate--In Y direction

--This example shows one of the nine configurations.

        LT: if ((i=0) and (j=0)) generate
        --Generate Left Top unit.
        apu0: pu port map(
        operation =>gInstruction, --Same for all PEs
        clk=>clk,
        dummy=>gDummy(N*j+i),
        output=>dout(i,j)          --Read by adjacent units.

        in_down=>dout(i,j+1),      --Only two directional connections.
        in_right=>dout(i+1,j),

        in_load=>dout(i+1,j), --For loading data
        out_load=>gOut,            --Unique to LT
        );

        end generate LT;

--Similar for other eight cases

end generate G2;
end generate G1;
```

The result of this synthesis can be seen in Figure 3. The processing array level ports of gLoad, gOut, gInstruction and clk can be seen along the outer region of the schematic, connecting to the internal ports, as described in the relevant port map (Algorithm 1) . In this case the processing array has been implemented as a square grid of n processing elements in each direction. This means that the total array contains n^2 elements. While this initial implementation is a square, this is not a particular requirement, and only minimal changes need to be made to the generate loops (the addition of a new variable for the range checking in the y-direction).

Figure 3: Unexpanded Schematic of a 2x2 Processing Array



3 Exploration of Image Processing Tasks

The implementation of a number of general image processing tasks demonstrates the use of the massively parallel processing array architecture. These are intended to demonstrate a broad range of general imaging requirements. In some cases, the inclusion of more specialised instructions has the potential to substantially simplify or accelerate the performance of specific tasks. In these scenarios, the potential changes are explored. However, the priority remains on general purpose performance. The simulation of these tasks was performed by Xilinx ISim (ISE Simulator).

3.1 Loading the Processing Array

The current configuration of the general purpose image processing architecture is based upon receiving a single serial stream of pixel information, which is then distributed through the processing array. The input value stream is connected to the final processing element, at the bottom right of the processing array. Each successive value is loaded into register B via the in_load port, then copied to register A. This value will then be read by the next processing element, on the following cycle.

Algorithm 2 Loading the Processing Array

```
--Loading the Array
for i in 0 to N*N-1 loop
        gLoad <= InputSignal(i);
        gInstruction <= Load;
                wait for clk_period;
        gInstruction <= CopyBA;
                wait for clk_period;
end loop;
```

When this procedure is applied using test values from 1 through 9, the opera-

tion of the array is clear. The first input value must travel through the entire array before reaching the top-left position. The simulation output (Figure 7) shows a behavioural simulation of this same example. Once the value of the top-left processing element has been set, the entire processing array has been propagated with values. At this point, the required image processing can be performed.

Figure 4: After First Step

-	-	-
-	-	-
-	-	$\leftarrow 1$

Figure 5: After Fifth Step

-	-	-
-	$\leftarrow 1$	2
3	4	5

Figure 6: The Final Result

1	2	3
4	5	6
7	8	9

This loading procedure requires two clock cycles per individual processing element, due to the completely serial nature of this method of image loading, the time taken linearly increases with the number of pixels. To give approximate performance requirements, Table 2 shows the portion of total clock cycles that are required for the loading operation. The transfer of the processed image data

Figure 7: Simulation of Load Operation

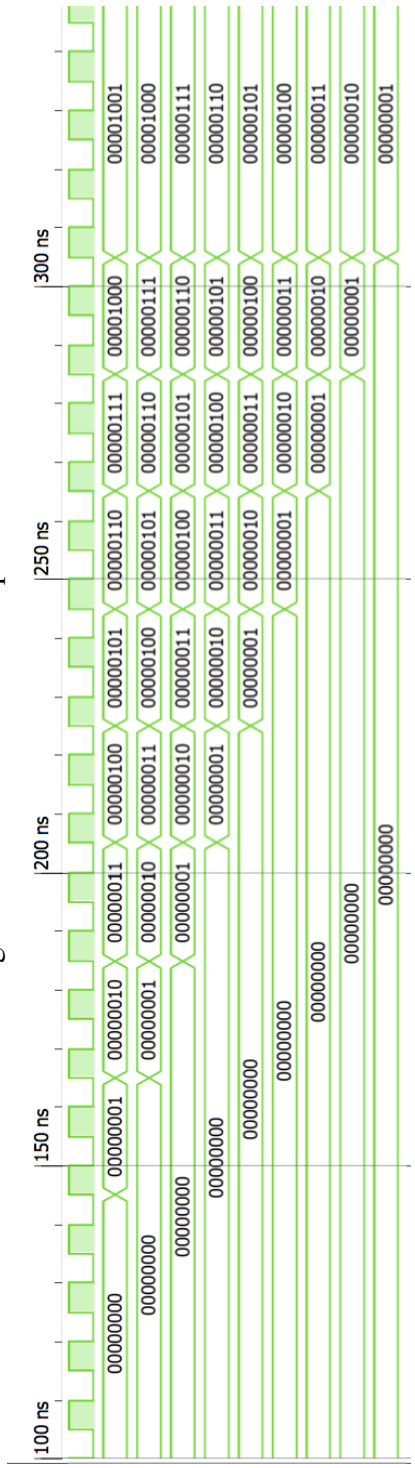


Table 2: Array Loading Performance Calculations

X	Y	Pixels/Frame	FPS	Pixels/second	Pixel Loading Frequency
128	128	16 384	30	491 520	~1 MHz
256	256	65 536	30	1 966 080	~4 MHz
512	512	262 144	30	7 864 320	~16 MHz

from the grid is automatically performed as the data for the next frame is being loaded. The dependence of performance on processing array size differs from the performance of the array on the parallelisable tasks which will be explored later.

3.2 Reduction Operations

A key aspect of this array-based architecture is the potential for quickly reducing information in the array structure into vector or scalar data [27]. Common reduction operations include finding the maximum or minimum values within the image, as well as arithmetic operations such as summation and multiplication of active processing elements. The processing array can be structured in such a way as to minimise the operations required for the reduction. The example below shows the use of row and column reduction, but the theoretical performance of the system with the addition structures such as a binary tree would be significantly greater, at the cost of additional FPGA area [28].

3.2.1 Finding the Array Maximum

The example explored here is the reduction image data to find the pixel of greatest intensity (the maximum).

Algorithm 3 Reduction to find Maximum

```
--To find the maximum value for each Column
for i in 0 to (N-1) loop
        gInstruction <= LdDown
                wait for clk_period;
        gInstruction <= MaxAB;
                wait for clk_period;
end loop;

--To find maximum for each Row
for i in 0 to (N-1) loop
        gInstruction <= LdRight
                wait for clk_period;
        gInstruction <= MaxAB;
                wait for clk_period;
end loop;
```

There are two stages involved in performing a complete minimum or maximum reduction. In the first stage the maximum value in each column is found, and this becomes the value for every PE in each column (Figure 9). This is performed by each PE loading the value of the PE below it, then on the next clock cycle, keeping the higher of its own value, or the value from below. This procedure is then repeated N-1 times, sufficient for correct operation in the worst case, when the maximum value is in the bottom row. The maximum value in each row can be found similarly, again with the operation performed N-1 times. If the column reduction operation has already been performed, every PE now contains the maximum value that was found in the array (Figure 10), which can then easily be read from the output port on the first element (LT).

Figure 8: Initial Values for Reduction (Finding the Maximum)

0	1	2
3	4	5
6	7	8

Figure 9: All Column Maximums Found

6	7	8
6↑	7↑	8↑
6↑	7↑	8↑

Figure 10: Then Row Maximums Found

8	←8	←8
8	←8	←8
8	←8	←8

The separation of row and column reduction thus enables the completion of these operations in $O(n)$ time.

3.3 Simple Image Filtering

3.3.1 Mean Filtering

When performing image processing tasks such as edge detection or disparity mapping, pre-filtering is used to increase accuracy. Due to the nature of this image processing architecture, it is simple to perform a four-way "fast" mean filter, replacing each pixel with the mean of its cardinal neighbours (Figure 11). The operation of such a filter on a small array is shown in Figure 13. Note that the

values of the edge elements are not useful, as their means include values that are beyond the edge of the processing array. This loss of edge pixels is common in image processing algorithms.

Figure 11: Mask for Four-way Mean Operation

	1/4	
1/4		1/4
	1/4	

Figure 12: Initial Array Values

5	5	4	6
6	6	4	5
7	30	6	4
5	7	3	6

Figure 13: Following Four-way Mean Operation

-	-	-	-
-	11	5	-
-	7	10	-
-	-	-	-

It can be seen that the outlier of "30" has been removed. However, while the outlier itself has been filtered away, it has still had a substantial impact on two of the remaining image pixels, illustrating a key downside of mean-based filtering, and other linear filtering schemes. The choice of which mean window to use is a compromise between retaining image sharpness, and the noise reduction requirements, with too large a window creating a "blur" effect. For the processing array

implementation, the mean filtering is performed by loading the four neighbouring pixels into the stack, then popping them back in turn. To avoid overflowing during the interim state, each operand is divided by four before it is summed. The central pixel now contains the four-way mean of the adjacent pixels.

3.3.2 Median Filtering

The non-linear technique of median filtering is commonly used as a pre-filtering stage, as it does not suffer from the aforementioned disadvantages of mean-based approaches[10]. The structure of FPGAs lends itself to the application of "fast-median" algorithms, which generally give reasonable approximations of the true median, while being much simpler to implement. The standard 3x3 fast median is executed by first finding the median of each column, and then finding the median of these results. This approach is used in specialised median filtering stages [29], but proved to be too computationally expensive for this massively parallel implementation. For this reason, the median filtering operations shown in algorithm 5 have not been implemented on the PEs. This is because the synthesis of median-specific instructions to allow either vertical and horizontal medians to be calculated in a single instruction requires the use additional logic levels. This is significantly greater than the logic required for any other single instruction, setting a maximum clock speed that can be reached by the system. This means that the addition of these instructions significantly degrades performance for other image processing tasks.

Algorithm 4 Fast Median Filtering with a 3x3 window (using specialised instructions)

```
--Finding Vertical Medians

gInstruction<=MedianUD; --Load Vertical Median into B
        wait for clk_period;
gInstruction<=CopyBA; --Copy B into A
        wait for clk_period;


--Finding Horizontal Medians
gInstruction<=MedianLR; --Load Horizontal Median into B
        wait for clk_period;
gInstruction<=CopyBA; --Copy B into A
        wait for clk_period;
```

Algorithm 5 Specialised Instructions for 3x3 Fast Median Filtering

```
when MedianUD => --Vertical Median
if ((in_up >= a and a >= in_down) or
        (in_down >= a and a >= in_up))
        then b<=a;

elsif ((a >= in_up and in_up >= in_down) or
        (in_down >= in_up and in_up >= a))
        then b<=in_up;

elsif ((a >= in_down and in_down >= in_up) or
        (in_up >= in_down and in_down >= in_up))
        then b<=in_down;
         end if;

when MedianLR => --Horizontal Median
if ((in_left >= a and a >= in_right) or
        (in_right >= a and a >= in_left))
        then b<=a;

elsif ((a >= in_left and in_left >= in_right) or
        (in_right >= in_left and in_left >= a))
        then b<=in_left;

elsif ((a >= in_right and in_right >= in_left) or
        (in_left >= in_right and in_right >= in_left))
        then b<=in_right;
        end if;
```

Figure 14: Initial Values for use with Fast Median Filtering

0	1	2
3	200	5
6	7	8

Figure 15: All Column Medians Found

↓	↓	↓
3	7	5
↑	↑	↑

Figure 16: Then Row Medians Found

-	-	-
→	5	←
-	-	-

3.3.3 Sobel Operator

The Sobel operator is used as an edge detection algorithm, finding gradients in the x and y directions. Using the separability of the Sobel operator (Figure 18), this operation can be easily performed in a highly parallel way. The separated matrices are known as the smoothing and difference operators [30]. Sobel-based edge detection is an area that FPGAs can perform very well in, due to the large number of simple operations that are required [31].

Figure 17: Mask for Sobel Operator (x-direction)

-1		1
-2		2
-1		1

This separability allows the component gradients to be determined by separate row and column operations, a good match for the massively parallel processing array. The application of the Sobel operator in the x-direction will be used to

Figure 18: Separability of Sobel Operator (x-direction) [27]

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} . \begin{pmatrix} -1 & 0 & 1 \end{pmatrix}$$

Figure 19: Initial Array Values for Sobel

0	1	2	3
8	10	12	7
8	9	10	11
12	13	14	15

demonstrate this process. After a set of image data has been loaded into the array (Figure 19), every processing element loads the value of the PE directly above it, and pushes this value to the stack, it then loads the value of the element below it into register B. Both of these values are loaded before any arithmetic operations take place, so that other PEs are able to read the initial values correctly. Once both values are loaded, the initial pixel value in A is doubled, and the pixels from the two vertically adjacent elements subtracted. The result of these operations is shown in Figure 20. The row operations must then be performed. Again the stack is used to ensure that the value of A remains untouched until all loading is complete. For this portion of the algorithm, the outcome is that the final value is the previous value of the left PE, minus the previous value of the right PE

Figure 20: Following Column Operations

-	-	-	-
8	10	12	0
-4	-5	-6	0
-	-	-	-

Figure 21: Sobel Process Complete in the x-direction

-	-	-	-
-	4	-10	-
-	-2	5	-
-	-	-	-

(Figure 21). The instructions required to perform these operations is shown in algorithm 6.

Algorithm 6 Sobel Operator in the x-direction

```
--Sobel Operator in X direction

--        First Part of Sobel in x (Columns)
gInstruction <= LdUp;
        wait for clk_period;
gInstruction <= PushB;
        wait for clk_period;
gInstruction <= LdDown;
        wait for clk_period;
gInstruction <=DblA;
        wait for clk_period;
gInstruction <=AddBA;
        wait for clk_period;
gInstruction <=PopB;
        wait for clk_period;
gInstruction <=AddBA;
        wait for clk_period;
--Every PE = above + 2*itself + below

--        Second Part of Sobel in X (Rows)
gInstruction <= LdLeft;
        wait for clk_period;
gInstruction <= PushB;
        wait for clk_period;
gInstruction <= LdRight;
        wait for clk_period;
gInstruction <=CopyBA;
        wait for clk_period;
gInstruction <=PopB;
        wait for clk_period;
gInstruction <=SubBA;
        wait for clk_period;
--Every PE = left - right
--Sobel in X direction completed.
```

Figure 22: Separability of Sobel Operator (y-direction)

$$\begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} . \begin{pmatrix} 1 & 2 & 1 \end{pmatrix}$$

A similar process is used to approximate image gradients in the y-direction Figure 22 . Once the gradient magnitudes are found, the edge intensity can be calculated using a polar conversion. Alternatively, an approximation of the overall edge intensity can be found by summing the absolute values of the gradients of the x and y direction.

3.4 Stereo Vision

The hardware required to capture stereo images can be implemented inexpensively. However, the computational performance required to process these images in a reasonable time remains prohibitive for many applications, particularly for embedded devices. For this reason the potential for FPGAs to accelerate this processing has been an area of keen development [32, 33, 9, 34]. The algorithms discussed below are area-based techniques for correlating two stereo images. In stereo correspondence calculations, the algorithm compares a window (for example a 5x5 square) in one image, with every possible window in the other image. The relative pixel offset between a window area and its best match (greatest correlation) gives a value of stereo disparity. This is repeated for every window of the initial image, with a greater disparity indicating that the object is closer to the cameras. On a serial processor, smaller window sizes dramatically increase the total number of operations required, and hence time taken [32]. In conventional stereo imaging, each image is the current output frame of calibrated left and right cameras. For the purposes of stereo vision, calibrated cameras are those aligned along a single horizontal plane. This calibration means that stereo correspondence comparisons only need to be made in the x axis, significantly reducing the com-

putational intensity of the processes. The Sum of Absolute Differences (SAD) algorithm is a simple method for determining a correlation between two images. Under the SAD method, smaller window sizes generally give a greater level of accuracy. The performance of stereo vision systems can often be improved by finding edges (via the Sobel operator or similar), and then running a correspondence algorithm such as SAD on the processed edges [35].

3.4.1 Sum of Absolute Differences

The instructions required to calculate the SAD are shown in 7. The pixel values of the left and right stereo image are subtracted (Figure 23), and the absolute value of these differences is taken (Figure 24). This absolute value is then summed along each three pixel column (Figure 25) and row (Figure 26). When these operations are completed, each PE contains the 3x3 SAD value for that disparity. The disparity is then increased (shift one of the images across by one pixel), and the operations repeated.

Figure 23: Example Left and Right Image Differences

1	4	6
-1	8	9
2	-3	-2

Figure 24: Absolute Value of Left and Right Image Differences

1	4	6
1	8	9
2	3	2

Algorithm 7 Summing Absolute Differences

```
--        3x3 SAD Implementation
--  A has left image, B has right image
gInstruction <= SubBA;
        wait for clk_period;
gInstruction <= AbsA;
        wait for clk_period;
--Now have AD for each pixel pair.


gInstruction <= LdUp;
        wait for clk_period;
gInstruction <= PushB;
        wait for clk_period;
gInstruction <= LdDown;
        wait for clk_period;
gInstruction <= AddBA;
        wait for clk_period;
gInstruction <=PopB;
        wait for clk_period;
gInstruction <= AddBA;
        wait for clk_period;
--Now Column SAD done.

gInstruction <= LdLeft;
        wait for clk_period;
gInstruction <= PushB;
        wait for clk_period;
gInstruction <= LdRight;
        wait for clk_period;
gInstruction <= AddBA;
        wait for clk_period;
gInstruction <=PopB;
        wait for clk_period;
gInstruction <= AddBA;
        wait for clk_period;
--Sum of AD done.
--Each PU now has 3x3 SAD for this disparity.
```

Figure 25: Column SAD

↓	↓	↓
4	15	17
↑	↑	↑

Figure 26: Sum of Absolute Differences for a Single Disparity Level

-	-	-
→	36	←
-	-	-

The inclusion of additional higher-level logic will be required to maintain the value of the lowest SAD, and the disparity for which this occurred. The disparity that gave the lowest SAD will become that pixel's value in the final disparity map. The generated disparity map then gives an indication of the relative distance of each image pixel from the cameras. The potential for more specialised FPGA co-processors to dramatically enhance SAD systems is already known [36, 20].

3.5 Demonstration of Conditional Activation

The current implementation of this general purpose image co-processor supports only one method of conditional activation. This is provided by the DisableZ operation, which disables all PEs that have a zero value in register A. When a PE is disabled, it does not execute any further operations until after the issue of an EnableAll command. These operations demonstrate basic conditional execution within the array.

Figure 27: Initial Values for Conditional Execution

0	1	2
3	4	5
6	7	8

Figure 28: After DisableZ is issued

Off	1	2
3	4	5
6	7	8

Figure 29: All Active PEs Incremented by One

Off	2	3
4	5	6
7	8	9

Figure 30: Final Values

0	2	3
4	5	6
7	8	9

4 Results

The synthesis results in Table 3 show that the FPGA resources required to implement the PA structure grows significantly as the required array size increases. The quadratic increase seen in Figure 31 is to be expected, as the number of PUs required rises with the square of array dimension.

Table 3: Synthesis Results

N	PEs	Slice LUTs	Max Clock	LUT Utilisation of XC6SLX45
3	9	1150	136 MHz	4%
6	36	5102	115 MHz	18%
9	81	11 516	52 MHz	42%
12	144	20 494	65 MHz	75%

While there are FPGAs with significantly larger numbers of logic slices than the XC6SLX45, results show that due to the quadratic scaling of LUT utilisation, that any full-resolution implementation of this complete set of general purpose instructions will become difficult to synthesise on commodity hardware. The highest-end model of Xilinx's new Virtex-7 range (the XC7V2000T) offers 1 955k logic cells, of similar structure to the 43k logic cells found on the XC6SLX45 (both have four LUTS, and eight flip-flops per block). Assuming the scaling of LUT utilisation continues quadratically as expected, this would give a potential n of around 95, for a total of 9 025 processing elements [37]. Thus while the synthesis of such a massively parallel design on affordable hardware in the near future may require a tighter focus on which tasks will be performed (reducing the logic resources required for each PE), future hardware will be significantly more capable of implementing such a design, while retaining general utility.

A reduction in maximum clock speed can also be seen as the internal routing of the FPGA becomes more complex, and there is a greater level of resource contention. The slower than expected operating frequency of the n=9 processing array (Table 3) was reproducible, and appeared to be caused by an optimisation that favoured heavily sharing of arithmetic resources between a large number of

Figure 31: LUT Utilisation following Synthesis

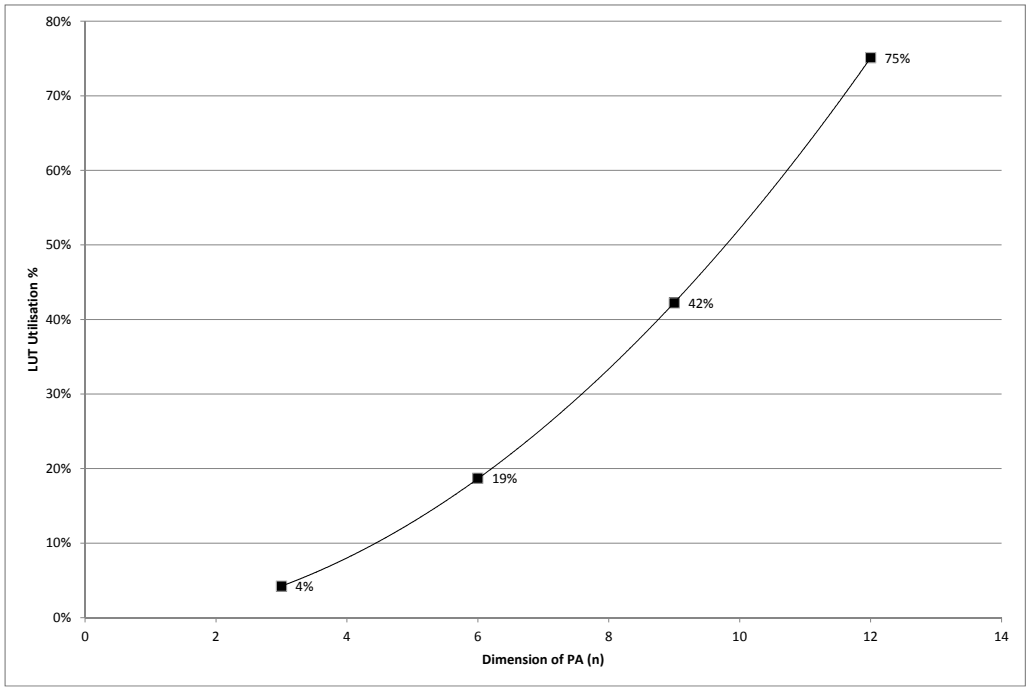
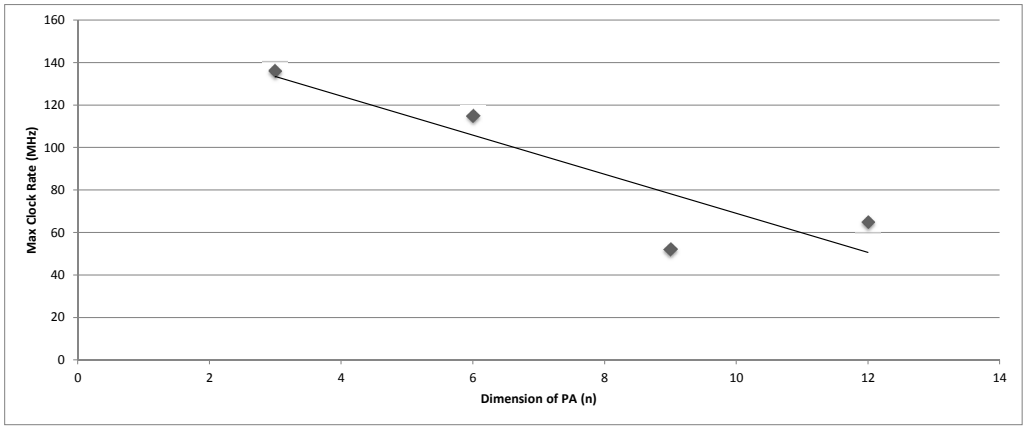


Figure 32: Maximum Clock Speed



processing elements. It is possible to adjust synthesiser performance to prioritise for logic area or clock speed, through configuring its approach to logic block sharing, however, this did not have a significant impact on the synthesis results. The relatively low clock speeds that can be seen for the larger PAs would significantly increase the time required to load in image frames, but in many cases would not substantially slow the image processing itself.

5 Conclusion

This implemented system shows that a system consisting of a parallel processing array, made up of a large number of interconnected PUs can achieve high levels of theoretical image processing performance, performing many processing in a small number of clock cycles. This sort of architecture would be especially effective when operating on larger image sizes, as many common image processing tasks can be performed at a reduced order compared to serial processing.

With the current instruction set, commercially available FPGA hardware does not offer sufficient logic area to implement this generalised a parallel architecture on a realistic scale, as even relatively small processing arrays will consume the entirety of available logic resources of current FPGA hardware. Unfortunately, with current technology, there are still significant compromises that must be made to the generalised capabilities of individual processing elements, to ensure it is possible to implement this kind of massively parallel architecture at useful image sizes. The chosen instruction set was relatively complex compared to other research approaches [24, 23], and it is possible that further simplification of the PEs abilities in favour of a greater level centralised logic would allow for improved scaling.

6 Future Work

This project has demonstrated the possibility of developing a general purpose image processing architecture through the application of a massively-parallel grid architecture. In this case the architecture's utility was benchmarked through the use of the Xilinx iSim simulator. It is expected that future work would include the synthesis of this design onto physical FPGA hardware, allowing the connection of cameras and output screens. Unfortunately, current FPGA hardware does not allow for the synthesis of full-frame size processing arrays.

6.1 Hardware Implementation

Throughout its development, this architecture was synthesised for the Xilinx Spartan-6 XC6SLX45 FPGA. The configuration onto a physical device would only requires the addition of for input/output pin allocations [15]. It is likely that to achieve useful results on physical hardware, an FPGA model with a greater number of CLBs would be required. It is possible to explore techniques such as handling small sections of the image at one time. The viability of such an approach depends on the specifics of the imaging tasks that must be performed. For example, as covered earlier, for calibrated cameras the SAD method only examines windows along a single horizontal line, so a system than can break-down the image into horizontal strips could be appropriate, while other imaging tasks would be most performant with square image sections. This kind of image partitioning of images thus creates its own set of design compromises. With current hardware it is possible that a dynamically reconfigurable approach would also be useful, by creating a versatile system that can perform a variety of tasks, but requires reconfiguration when moving between them. This may allow a better balance to be struck between PE capability and PA size.

6.2 Camera Interfacing

The architecture in its current state has been developed based around a purely serial image signal as the input to the processing grid. This requires the image data for each pixel to traverse the grid from the final PE to the appropriate element. It is likely that the specific output and timing requirements of the cameras used would necessitate the development of a buffer stage, and increased logic dealing with missed pixels, resynchronisation and timing.

6.3 Conditional Logic

For this kind of architecture to be more readily used, especially via a higher level programming interface, there should be a greater investigation of the systems requirements in terms of conditional logic functions. A greater number of conditional flags would allow a broader range of conditional behaviours, at a relatively low cost in terms of FPGA logic area. This would include the development of a nested conditional structures, and operations such as while loops. It would also be useful to pass each PU information about its coordinates during generation (this could be done with a generic map), so that operations can be performed conditionally of the PE's specific array location.

6.4 Connection Changes

Another set of potential interconnections that may yield useful results is the creation of diagonal connections between processing elements. This would directly increase the accuracy of some operations, for example, mean filtering. Improvements in filtering processes are particularly useful in computationally constrained scenarios, as often filtering improvements can give greater additionally accuracy than equivalent improvements to more high-level operations. The addition of diagonal interconnections would also enable the simplification of currently possible image processing tasks. Processing element interconnections that create a binary-tree configuration would also provide improved performance for reduction tasks

[28]. The changes to array performance that can caused by changes to inter-array connectivity can be significant, with one potential path being the simplification of each PE, while creating a more complex structure between neighbouring elements As one example, allowing local chaining between PEs can allow them to act as a single, more capable PE when appropriate [25].

References

[1] D. M. Harvey, S. P. Kshirsagar, and C. Hobson, "Low Cost Scaleable Parallel Image Processing System," *Microprocessors and Microsystems*, vol. 25, pp. 143–157, May 2001.

[2] A. Nieto, V. Brea, D. L. Vilariño, and R. R. Osorio, "Performance Analysis of Massively Parallel Embedded Hardware Architectures for Retinal Image Processing," *EURASIP Journal on Image and Video Processing*, vol. 2011, p. 10, Sept. 2011.

[3] W. S. Fife and J. K. Archibald, "Reconfigurable On-Board Vision Processing for Small Autonomous Vehicles," *EURASIP Journal on Embedded Systems*, vol. 2007, p. 080141, Dec. 2006.

[4] B. Tippetts, D. Lee, and J. Archibald, "An On-board Vision Sensor System for Small Unmanned Vehicle Applications," *Machine Vision and Applications*, vol. 23, no. 3, pp. 403–415, 2012.

[5] J. Batlle, J. Marti, P. Ridao, and J. Amat, "A New FPGA/DSP-Based Parallel Architecture for Real-Time Image Processing," *Real-Time Imaging*, vol. 8, no. 5, pp. 345 – 356, 2002.

[6] J. Xiong and Q. Wu, "An Investigation of FPGA Implementation for Image Processing," in *Communications, Circuits and Systems (ICCCAS), 2010 International Conference on*, pp. 331 –334, july 2010.

[7] D. Crookes, "Architectures for High Performance Image Processing: The future," *Journal of Systems Architecture*, vol. 45, no. 10, pp. 739 – 748, 1999.

[8] A. Fijany and F. Hosseini, "Image Processing Applications on a Low Power Highly Parallel SIMD architecture," in *Aerospace Conference, 2011 IEEE*, pp. 1 –12, march 2011.

[9] C. Murphy, D. Lindquist, A. M. Rynning, T. Cecil, S. Leavitt, and M. L. Chang, "Low-Cost Stereo Vision on an FPGA," pp. 333–334, IEEE, Apr. 2007.

[10] Y. Hu and H. Ji, "Research on Image Median Filtering Algorithm and Its FPGA Implementation," in *Intelligent Systems, 2009. GCIS '09. WRI Global Congress on*, vol. 3, pp. 226 –230, may 2009.

[11] R. Mueller, J. Teubner, and G. Alonso, "Sorting Networks on FPGAs," *The VLDB Journal*, vol. 21, pp. 1–23, June 2011.

[12] A. Ruta, R. Brzoza-Woch, and K. Zielinski, "On Fast Development of FPGA-based SOA Services," *Design Automation for Embedded Systems*, vol. 16, pp. 45–69, Mar. 2012.

[13] N. Roudel, F. Berry, J. Serot, and L. Eck, "A New High-Level Methodology for Programming FPGA-Based Smart Cameras," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pp. 573 –578, Sept. 2010.

[14] J. Dubois, D. Ginhac, M. Paindavoine, and B. Heyrman, "A 10 000 fps CMOS Sensor with Massively Parallel Image Processing," *Solid-State Circuits, IEEE Journal of*, vol. 43, no. 3, pp. 706–717, 2008.

[15] Xilinx, *Spartan-6 FPGA Configuration*, 2010.

[16] Xilinx, *Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency*, 2010.

[17] Xilinx, *Spartan-6 Family Overview*, 2011.

[18] P. J. Ashenden, *The Designer's Guide to VHDL*. Burlington: Morgan Kaufmann, 3 ed., 2008.

[19] J. Woodfill and B. Von Herzen, "Real-time Stereo Vision on the PARTS Reconfigurable Computer," pp. 201–210, IEEE Comput. Soc, 1997.

[20] B. Draper, W. Najjar, W. Bohm, J. Hammes, B. Rinker, C. Ross, M. Chawathe, and J. Bins, "Compiling and Optimizing Image Processing Algorithms for FPGAs," in *Computer Architectures for Machine Perception, 2000. Proceedings. Fifth IEEE International Workshop on*, pp. 222 –231, 2000.

[21] M. Prieto and A. Allen, "A Hybrid System for Embedded Machine Vision using FPGAs and Neural Networks," *Machine Vision and Applications*, vol. 20, no. 6, pp. 379–394, 2009.

[22] T. Kurafuji, M. Haraguchi, M. Nakajima, T. Gyoten, T. Nishijima, H. Yamasaki, Y. Imai, M. Ishizaki, T. Kumaki, Y. Okuno, *et al.*, "A Scalable Massively Parallel Processor for Real-time Image Processing," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 334–335, IEEE, 2010.

[23] W. Miao, Q. Lin, W. Zhang, and N. Wu, "A Programmable SIMD Vision Chip for Real-Time Vision Applications," *Solid-State Circuits, IEEE Journal of*, vol. 43, pp. 1470 –1479, June 2008.

[24] P. Dudek and P. Hicks, "A General-Purpose Processor-per-Pixel Analog SIMD Vision Chip," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 52, pp. 13 – 20, Jan. 2005.

[25] T. Komuro, S. Kagami, and M. Ishikawa, "A Dynamically Reconfigurable SIMD Processor for a Vision Chip," *Solid-State Circuits, IEEE Journal of*, vol. 39, pp. 265 – 268, Jan. 2004.

[26] B. Krill, A. Ahmad, A. Amira, and H. Rabah, "An Efficient FPGA-based Dynamic Partial Reconfiguration Design flow and Environment for Image and Signal Processing IP cores," *Signal Processing: Image Communication*, vol. 25, pp. 377–387, June 2010.

[27] T. Bräunl, *Parallel Image Processing*. Springer, Jan. 2001.

[28] L. Zhuo, G. Morris, and V. Prasanna, "Designing Scalable FPGA-Based Reduction Circuits Using Pipelined Floating-Point Cores," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, p. 147a, april 2005.

[29] P. Wei, L. Zhang, C. Ma, and T. S. Yeo, "Fast Median Filtering Algorithm based on FPGA," in *Signal Processing (ICSP), 2010 IEEE 10th International Conference on*, pp. 426 –429, oct. 2010.

[30] T. A. Abbasi and M. U. Abbasi, "A Proposed FPGA Based Architecture for Sobel Edge Detection Operator," *Journal of Active and Passive Electronic Devices*, vol. 2, no. 4, pp. 271 – 277, 2007.

[31] R. Rosas, A. de Luca, and F. Santillan, "SIMD Architecture for Image Segmentation using Sobel operators implemented in FPGA technology," in *Electrical and Electronics Engineering, 2005 2nd International Conference on*, pp. 77 – 80, sept. 2005.

[32] A. Darabiha, J. Rose, and W. J. MacLean, "Video-Rate Stereo Depth Measurement on Programmable Hardware," in *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, vol. 1, (Los Alamitos, CA, USA), p. 203, IEEE Computer Society, 2003.

[33] M. Brown, D. Burschka, and G. Hager, "Advances in Computational Stereo," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, pp. 993–1008, Aug. 2003.

[34] S. Wong, S. Vassiliadis, and S. Cotofana, "A Sum of Absolute Differences Implementation in FPGA hardware," in *Euromicro Conference, 2002. Proceedings. 28th*, pp. 183 – 188, 2002.

[35] S. Hadjitheophanous, C. Ttofis, A. Georghiades, and T. Theocharides, "Towards Hardware Stereoscopic 3D Reconstruction," in *Automation Test in Europe Conference Exhibition, 2010*, pp. 1743 –1748, Mar. 2010.

[36] P. Greisen, S. Heinzle, M. Gross, and A. P. Burg, "An FPGA-based Processing Pipeline for High-Definition Stereo Video," *EURASIP Journal on Image and Video Processing*, vol. 2011, p. 18, Nov. 2011.

[37] Xilinx, *Virtex-7 Family Overview*, 2012.

A PEs.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity pe is
6
7  port (
8          in_up:       in  signed  (7 downto 0):=
               to_signed(0,8);
9          in_down:     in  signed  (7 downto 0):=
               to_signed(0,8);
10         in_left:     in  signed  (7 downto 0):=
               to_signed(0,8);
11         in_right:    in  signed  (7 downto 0):=
               to_signed(0,8);
12
13         in_load:     in  signed  (7 downto 0);
14         out_load:    out signed  (7 downto 0);
15
16         output:      out  signed (7 downto 0);
17
18         operation:   in  STD_LOGIC_VECTOR (4 downto 0);
19         clk :        in  STD_LOGIC
20         );
21
22 end pe;
23
24
```

```
25 architecture behavior of pe is
26         --Registers
27         signal a : signed (7 downto 0) := to_signed
              (0,8);
28         signal b : signed (7 downto 0) := to_signed
              (0,8);
29
30         --Enable Signals
31         signal zeroflag : bit :='0';
32         signal enable   : bit :='1';
33
34         --16x1byte Stack
35         type stacktype is array (0 to 15) of signed (7
              downto 0);
36         signal stack : stacktype := (others => (others
              => '0'));
37         signal stackindex : integer range 0 to 15 :=0;
38
39
40
41
42 begin
43
44 process (clk,a)
45     begin
46
47         if(rising_edge(clk) and enable='1') then
48
49         case operation is
50         when "00000" => --NOP
```

```
51
52          when "00001" => --Copy B to A
53              a <= b;
54
55          when "00010" => --Copy A to B
56              b <= a;
57
58          when "00011" => --BITWISE OR
59              a <= a or b;
60
61          when "00100" => --BITWISE AND
62              a <= a and b;
63
64          when "00101" => --BITWISE XOR
65              a <= a xor b;
66
67          when "00110" => --AddBA
68              a <= a + b;
69
70          when "00111" => --SubBA
71              a <= a-b;
72
73          when "01000" => --AbsA
74              a <= abs(a);
75
76          when "01001" => --Clr A
77              a <= "00000000";
78
79          when "01010" => --Clr B
80              b <= "00000000";
```

```
81
82          when "01011" => --Load from UP into B
83              b <= in_up;
84
85          when "01100" => --Load from DOWN into B
86              b <= in_down;
87
88          when "01101" => --Load from LEFT into B
89              b <= in_left;
90
91          when "01110" => --Load from RIGHT into B
92              b <= in_right;
93
94          when "01111" => --Load In
95              b <= in_load;
96
97          when "10000" => --SwapAB
98              a<=b;
99              b<=a;
100
101         when "10001" => --Double A
102             a <= (SHIFT_LEFT(a,1));
103
104         when "10010" => --Halve A
105             a <= (SHIFT_RIGHT(a,1));
106
107         when "10011" => --Maximum of A and B
108                 if a < b
109                 then a <= b;
110                 end if;
```

```
111
112             when "10100" => --Minimum of A and B
113                     if a > b
114                     then a <= b;
115                     end if;
116
117
118             when "10101" => --PushA
119                 stack(stackindex) <= a;
120                 stackindex <= stackindex + 1;
121
122             when "10110" => --PopA
123                 a <= stack(stackindex-1);
124                 stackindex <= stackindex - 1;
125
126             when "10111" => --PushB
127                 stack(stackindex) <= b;
128                 stackindex <= stackindex + 1;
129
130             when "11000" => --PopB
131                 b <= stack(stackindex-1);
132                 stackindex <= stackindex - 1;
133
134             when "11001" => --IncA
135                 a<=a+1;
136
137             when "11010" => --DecA
138                 a<=a-1;
139
140             when "11011" => --DisablePU if A=0.
```

```
141             if( a=0 ) then enable<='0'; end if;
142
143             when others =>
144
145     end case;
146
147
148
149 end if;
150
151             if(operation="11100") then enable<='1';
                    end if;
152
153             --output A to each.
154             out_load    <= a;
155             output      <= a;
156
157     end process;
158
159
160 end architecture;
```

B PA.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity PA is
6      --generic (N : Integer := 15); Default to 15x15
7  port (
8          gInstruction:    in       STD_LOGIC_VECTOR     (4
               downto 0);
9          gLoad:           in       signed   (7 downto 0);
10         gOut:            out      signed   (7 downto 0);
11         gDummy:          out STD_LOGIC_VECTOR(N*N-1
              downto 0);
12         clk :            in       STD_LOGIC
13         );
14
15 end entity PA;
16
17
18 architecture structure of PA is
19
20 component pe
21 PORT
22
23         (
24
25         in_up:  in  signed (7 downto 0):= to_signed
              (0,8);
```

```
26          in_down:    in  signed (7 downto 0):=
                to_signed(0,8);
27          in_left:    in  signed (7 downto 0):=
                to_signed(0,8);
28          in_right:   in  signed (7 downto 0):=
                to_signed(0,8);
29
30          in_load:    in  signed (7 downto 0):=
                to_signed(0,8);
31          out_load:   out signed (7 downto 0):=
                to_signed(0,8);
32
33          output   :  out signed (7 downto 0):=
                to_signed(0,8);
34
35          operation:  in  STD_LOGIC_VECTOR;
36          clk :       in  STD_LOGIC
37          );
38 end component;
39
40 type testtype is array (N-1 downto 0, N-1 downto 0) of
       signed (7 downto 0);
41 signal dout : testtype;
42
43 begin
44
45 G1 : for i in N-1 downto 0 generate
46 G2 : for j in N-1 downto 0 generate
47
48 LT: if ((i=0) and (j=0)) generate
```

```
49      apu0: pe port map
50      (
51          operation =>gInstruction,
52          clk=>clk,
53
54          in_down=>dout(i,j+1),
55          in_right=>dout(i+1,j),
56          in_load=>dout(i+1,j),
57
58          out_load=>gOut,
59
60          output=>dout(i,j)
61
62      );
63  end generate LT;
64
65  CT: if ((i/=N-1 and i/=0 ) and j=0) generate
66      apu0: pe port map
67      (
68          operation =>gInstruction,
69          clk=>clk,
70
71          in_down=>dout(i,j+1),
72          in_right=>dout(i+1,j),
73          in_load=>dout(i+1,j),
74          in_left=>dout(i-1,j),
75
76          output=>dout(i,j)
77
78      );
```

```
79 end generate CT;
80
81 RT: if ((i=N-1) and (j=0)) generate
82     apu0: pe port map
83     (
84         operation =>gInstruction,
85         clk=>clk,
86
87         in_down=>dout(i,j+1),
88         in_load=>dout(0,j+1),
89         in_left=>dout(i-1,j),
90
91         output=>dout(i,j)
92
93     );
94 end generate RT;
95
96 RC: if ((i=N-1) and (j/=0 and j/=N-1)) generate
97     apu0: pe port map
98     (
99         operation =>gInstruction,
100        clk=>clk,
101
102        in_up=>dout(i,j-1),
103        in_down=>dout(i,j+1),
104        in_load=>dout(0,j+1),
105        in_left=>dout(i-1,j),
106
107        output=>dout(i,j)
108
```

```
109       );
110  end generate RC;
111
112  RB: if ((i=N-1) and (j=N-1)) generate
113       apu0: pe port map
114       (
115            operation =>gInstruction,
116            clk=>clk,
117
118            in_up=>dout(i,j-1),
119            in_left=>dout(i-1,j),
120            in_load=>gLoad,
121
122            output=>dout(i,j)
123
124       );
125  end generate RB;
126
127  CB: if ((i/=N-1 and i/=0 ) and j=N-1) generate
128       apu0: pe port map
129       (
130            operation =>gInstruction,
131            clk=>clk,
132
133            in_up=>dout(i,j-1),
134            in_right=>dout(i+1,j),
135            in_load=>dout(i+1,j),
136            in_left=>dout(i-1,j),
137
138            output=>dout(i,j)
```

```
139
140       );
141   end generate CB;
142
143   LB: if ((i=0) and (j=N-1)) generate
144       apu0: pe port map
145       (
146           operation =>gInstruction,
147           clk=>clk,
148
149           in_up=>dout(i,j-1),
150           in_right=>dout(i+1,j),
151           in_load=>dout(i+1,j),
152
153           output=>dout(i,j)
154
155       );
156   end generate LB;
157
158   LC: if ((i=0) and (j/=0 and j/=N-1)) generate
159       apu0: pe port map
160       (
161           operation =>gInstruction,
162           clk=>clk,
163
164           in_up=>dout(i,j-1),
165           in_down=>dout(i,j+1),
166           in_right=>dout(i+1,j),
167           in_load=>dout(i+1,j),
168
```

```
169             output=>dout(i,j)
170
171         );
172     end generate LC;
173
174     CC: if ((i/=0 and i/=N-1) and (j/=0 and j/=N-1))
        generate
175         apu0: pe port map
176         (
177             operation =>gInstruction,
178             clk=>clk,
179
180             in_up=>dout(i,j-1),
181             in_down=>dout(i,j+1),
182             in_right=>dout(i+1,j),
183             in_load=>dout(i+1,j),
184             in_left=>dout(i-1,j),
185             output=>dout(i,j)
186
187         );
188     end generate CC;
189 end generate G2;
190 end generate G1;
191
192 end structure;
```