

High Level Programming for Real Time

FPGA Based Image Processing

D Crookes, K Benkrid, A Bouridane, K Alotaibi and A Benkrid

School of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, UK

ABSTRACT

Reconfigurable hardware in the form of Field Programmable Gate Arrays (FPGAs) has been proposed as a way of obtaining high performance for computationally intensive DSP applications such as Image Processing (IP), even under real time requirements. The inherent reprogrammability of FPGAs gives them some of the flexibility of software while keeping the performance advantages of an application specific solution.

However, a major disadvantage of FPGAs is their low level programming model. To bridge the gap between these two levels, we present a high level software environment for FPGA-based image processing, which aims to hide hardware details as much as possible from the user. Our approach is to provide a very high level Image Processing Coprocessor (IPC) with a core instruction set based on the operations of Image Algebra. The environment includes a generator which generates optimised architectures for specific user-defined operations.

1. INTRODUCTION

Image Processing application developers require high performance systems for computationally intensive Image Processing (IP) applications, often under real time requirements. In addition, developing an IP application tends to be experimental and interactive. This means the developer must be able to modify, tune or replace algorithms rapidly and conveniently.

Because of the local nature of many low level IP operations (e.g. neighbourhood operations), one way of obtaining high performance in image processing has been to use parallel computing [1]. However, multiprocessor IP systems have generally speaking not yet fulfilled their promise. This is partly a matter of cost, lack of stability and software support for parallel machines; it is also a matter of communications overheads particularly if sequences of images are being captured and distributed across the processors in real time.

A second way of obtaining high performance in IP applications is to use Digital Signal Processing (DSP) processors [2,3]. DSP processors provide a performance improvement over standard microprocessors while still maintaining a high level programming model. However, because of the software based control, DSP processors have still difficulty in coping with real time video processing.

At the opposite end of the spectrum lie the dedicated hardware solutions. Application Specific Integrated Circuits (ASICs) offer a fully customised solution to a particular algorithm [4]. However, this solution suffers from a lack of flexibility, plus the high manufacturing cost and the relatively lengthy development cycle.

Reconfigurable hardware solutions in the form of FPGAs [5] offer high performance, with the ability to be electrically reprogrammed dynamically to perform other algorithms.

Though the first FPGAs were only capable of modest integration levels and were thus used mainly for glue logic and system control, the latest devices [6] have crossed the Million gate barrier hence making it possible to implement an entire System On a Chip. Moreover, the introduction of the latest IC fabrication techniques has increased the maximum speed at which FPGAs can run. Design's performance exceeding 150MHz are no longer outside the realm of possibilities in the new FPGA parts, hence allowing FPGAs to address high bandwidth applications such as video processing.

A range of commercial FPGA based custom computing systems includes: the Splash-2 system [7]; the G-800 system [8] and VCC's HOTWorks HOTI & HOTII development [9]. Though this solution seems to enjoy the advantages of both the dedicated solution and the software based one, many people are still reluctant to move toward this new technology because of the low level programming model offered by FPGAs. Although behavioural synthesis tools have made enormous progress [10, 11], structural design techniques (including careful floorplanning) often still result in circuits that are substantially smaller and faster than those developed using only behavioural synthesis tools [12].

In order to bridge the gap between these two levels, this paper presents a high level software environment for an FPGA-based Image Processing machine, which aims to hide the hardware details from the user. The environment generates optimised architectures for specific user-defined operations, in the form of a low level netlist. Our system uses Prolog as the basic notation for describing and composing the basic building blocks. Our current implementation of the IPC is based on the Xilinx 4000 FPGA series [13].

The paper first outlines the programming environment at the user level (the programming model). This includes facilities for defining low level Image Processing algorithms based on the operators of Image Algebra [14], without any reference to hardware

details. Next, the design of the basic building blocks necessary for implementing the IPC instruction set are presented. Then, we describe the runtime execution environment.

2. THE USER'S PROGRAMMING MODEL

At its most basic level, the programming model for our image processing machine is a host processor (typically a PC programmed in C++) and an FPGA-based Image Processing Coprocessor (IPC) which carries out complete image operations (such as convolution, erosion etc.) as a single coprocessor instruction. The instruction set of the IPC provides a core of instructions based on the operators of Image Algebra. The instruction set is also *extensible* in the sense that new compound instructions can be defined by the user, in terms of the primitive operations in the core instruction set. (Adding a new primitive instruction is a task for an architecture designer).

The coprocessor core instruction set

Many IP neighbourhood operations can be described by a template (a static window with user defined weights) and one of a set of Image Algebra operators. Indeed, simple neighbourhood operations can be split in two stages:

- A 'local' operator applied between an image pixel and the corresponding window coefficient.
- A 'global' operator applied to the set of intermediate results of the local operation, to reduce this set to a single result pixel.

The set of local operators contains 'Add' ('+') and 'multiplication' ('*'), whereas the global operator contains 'Accumulation' (' Σ '), 'Maximum' ('Max') and 'Minimum' ('Min'). With these local and global operators, the following neighbourhood operations can be built:

<i>Neighbourhood Operation</i>	<i>Local operation</i>	<i>Global operation</i>
Convolution	*	Σ
Additive maximum	+	Max
Additive minimum	+	Min
Multiplicative maximum	*	Max
Multiplicative minimum	*	Min

For instance, a simple Laplace operation would be performed by doing convolution (i.e.

Local Operation = ' Σ ' and *Global operation* = '*') with the following template:

~	-1	~
-1	4	-1
~	-1	~

The programmer interface to this instruction set is via a C++ class. First, the programmer *creates* the required instruction object (and its FPGA configuration), and subsequently *applies* it to an actual image. Creating an instruction object is generally in two phases: firstly build an object describing the operation, and then generate the configuration, in a file. For neighbourhood operations, these are carried out by two C++ object constructors:

image_operator (template & operator details)

image_instruction (operator object, filename)

For instructions with a single template operator, these can be conveniently combined in a single constructor:

Neighbourhood_instruction (template, operators, filename)

The details required when building a new image operator object include:

- The dimension of the image (e.g. 256×256)
- The pixels size (e.g. 16 bits).
- The size of the window (e.g. 3×3).
- The weights of the neighbourhood window.
- The target position within the window, for aligning it with the image pixels (e.g. 1,1).
- The ‘local’ and ‘global’ operations.

Later, to apply an instruction to an actual image, the *apply* method of the instruction object is used:

Result = instruction_object.*apply* (input image)

This will reconfigure the FPGA (if necessary), download the input pixel data and store the result pixels in the RAM of the IPC as they are generated.

The following example shows how a programmer would create and perform a 3 by 3 Laplace operation. The image is 256 by 256; the pixel size is 16 bits.

```
// First, generate the FPGA configuration in a file

laplace = new Neighbourhood_instruction (Input_image(256,256,16), local(mult),global (accum),
                                         window(3,3,1,1, { {~, -1, ~},
                                                             {-1, 4, -1},
                                                             {~, -1, ~} })), "MyLaplace");

// .....
// Subsequently, execute the algorithm on the FPGA-based IPC

Output_image = laplace.apply(Input_image);
```

2.1 Extending the Model for Compound Operations

In practical image processing applications, many algorithms comprise more than a single operation. Such compound operations can be broken into a number of primitive core instructions.

Instruction Pipelining: A number of basic image operations can be put together in series. A typical example of two neighbourhood operations in series is the ‘Open’ operation. To do an ‘Open’ operation, an ‘Erode’ neighbourhood operation is first performed, and the resulting image is fed into a ‘Dilate’ neighbourhood operation as shown in Figure 1.

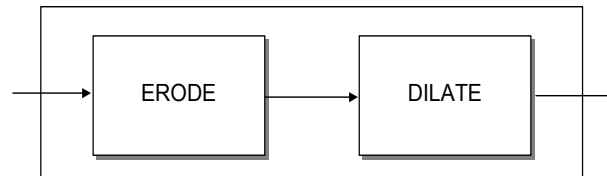


Figure 1 ‘Open’ complex operation

This operation is described as follows in our high level environment:

```

Erode = new image_operator ( Input_image(256,256, 16), local(add), global(minimum),
                             window(3,3, 1, 1, { { ~, 0, ~ },
                                                    { 0, 0, 0 },
                                                    { ~, 0, ~ } }));

/* ‘Open’ Complex operation */
/* Note that ‘Erode’ stands for the input image of ‘Dilate’ operation. */

Open = new Neighbourhood_instruction (Erode, local(add), global(maximum),
                                       window(3,3, 1, 1, { { ~, 0, ~ },
                                                            { 0, 0, 0 },
                                                            { ~, 0, ~ } }}, “My_Open”);
  
```

Task parallel: A number of basic image operations can be put together in *parallel*. For example, the Sobel edge detection algorithm can be performed (approximately) by adding the absolute results of two separate convolutions. Assuming that the FPGA has enough computing resources available, the best solution is to implement the operations in parallel using separate regions of the FPGA chip.

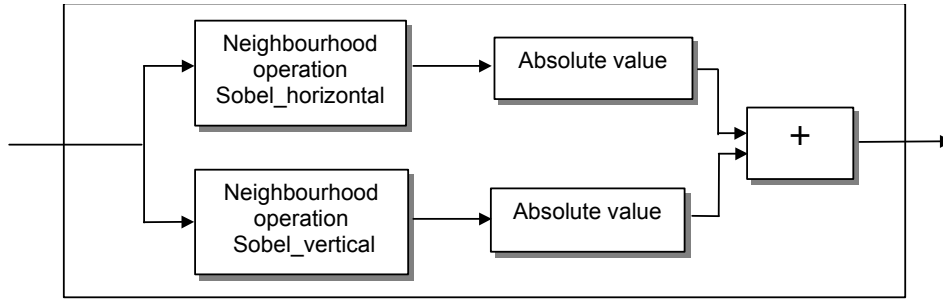


Figure 2 *Sobel complex operation*

The following is an example of the code, based on our high level instruction set, to define and use a Sobel edge detection instruction. The user defines two neighbourhood operators (horizontal and vertical Sobel), and builds the image instruction by summing the absolute results from the two neighbourhood operations.

```

Sobel_horizontal = new image_operator ( Input_image(256,256,16), local(mult), global(accum),
                                         window(3,3 , 1,1 {{ 1, 2, 1 },
                                                             { 0, 0, 0 },
                                                             {-1, -2, -1 }} ));

Sobel_vertical   = new image_operator ( Input_image(256,256,16), local(mult), global(accum),
                                         window(3,3 , 1, 1, {{ 1, 0, -1 },
                                                             { 2, 0, -2 },
                                                             { 1, 0, -1 }} ));

/* Sobel Complex operation */

Sobel = new image_instruction(plus( abs(Sobel_horizontal), abs(Sobel_vertical) ), "MySobel");

```

The generation phase will automatically insert the appropriate delays to synchronise the two parallel operations.

3. ARCHITECTURES FROM OPERATIONS

When a new *Image_instruction* object (e.g. *Neighbourhood_instruction*) is created (by **new**), the corresponding FPGA configuration will be generated dynamically. In this section, we will present the structure of the FPGA configurations necessary to implement the high level

instruction set for the neighbourhood operations described above. As a key example, the structure of a general 2-D convolver will be presented. Other neighbourhood operations are essentially variations of this, with different local and global operators sub-blocks.

A general 2D convolver

As mentioned earlier, any neighbourhood image operation involves passing a 2-D window over an image, and carrying out a calculation at each window position.

To allow each pixel to be supplied only once to the FPGA, internal line delays are required. These synchronise the supply of input values to the processing elements, ensuring that all the pixel values involved in a particular neighbourhood operation are processed at the same instant[15, 16]. Assuming a vertical scan of the image, Figure 3 shows the architecture of a generic 2-D convolver with a P by Q template. Each Processing Element (PE) performs the necessary Multiply/Accumulate operation.

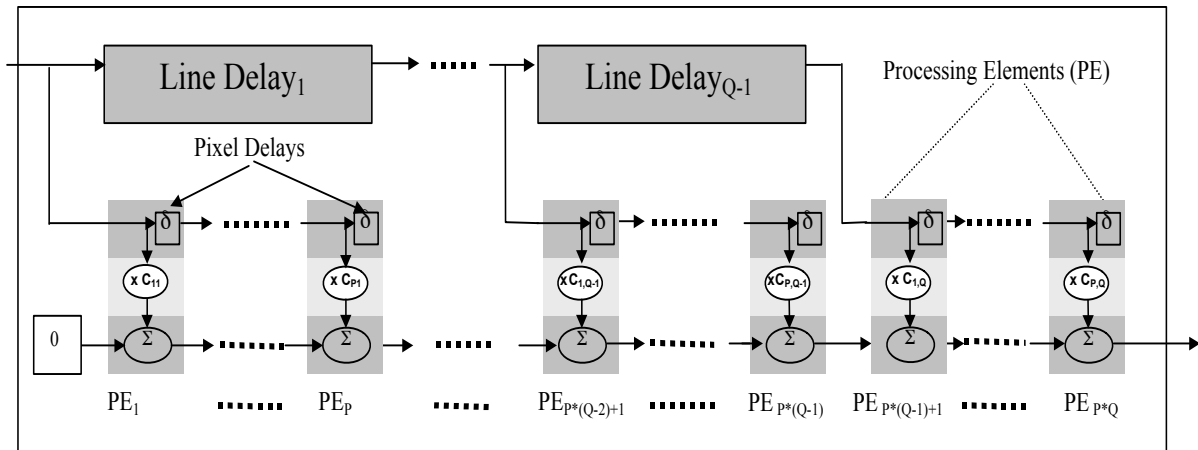


Figure 3 *Architecture of a generic 2-D, P by Q convolution operation*

Architecture of a Processing Element

Before deriving the architecture of a Processing Element, we first have to decide which type of arithmetic to be used- either bit parallel or bit serial processing.

While parallel designs process all data bits simultaneously, bit serial ones process input data one bit at a time. The required hardware for a parallel implementation is typically 'n' times the equivalent serial implementation (for an n-bit word). On the other hand, the bit serial approach requires 'n' clock cycles to process an n-bit word while the equivalent parallel one needs only one clock cycle. However, bit serial architectures operate at a higher clock frequency due to their smaller combinatorial delays. Also, the resulting layout in a serial implementation is more regular than a parallel one, because of the reduced number of interconnections needed between PEs (i.e. less routing stress). This regularity feature means that FPGA architectures generated from a high level specification can have more predictable layout and performance. Moreover, a serial architecture is not tied to a particular processing word length. It is relatively straightforward to move from one word length to another with very little extra hardware (if any). For these reasons, we decided to implement the IPC hardware architectures using serial arithmetic.

Note, secondly, that the need to pipeline the bit serial Maximum and Minimum operations common in Image Algebra suggests we should process data Most Significant Bit first (MSBF). Following on from this choice, because of problems in doing addition MSBF in 2's complement, there are certain advantages in using an alternative number representation to 2's complement. For the purposes of the work described in this paper, we have chosen to use a redundant number representation in the form of a radix-2 Signed Digit Number system (SDNR) [17]. Because of the inherent carry-free property of SDNR add/subtract operations, the corresponding architectures can be clocked at high speed. There are of course several alternative representations which could have been chosen, each with their own advantages. However, the work presented in this paper is based on the following design choices:

- Bit serial arithmetic

- Most Significant Bit First processing
- Radix-2 Signed Digit Number Representation (SDNR) rather than 2's complement.

Because image data may have to be occasionally processed on the host processor, the basic storage format for image data is still, however, 2's complement. Therefore, processing elements first convert their incoming image data to SDNR. This also reduces the chip area required for the line buffers (in which data is held in 2's complement). A final unit to convert a SDNR result into 2's complement will be needed before any results can be returned to the host system. With these considerations, a more detailed design of a general Processing Element (in terms of a local and a global operation) is given in Figure 4.

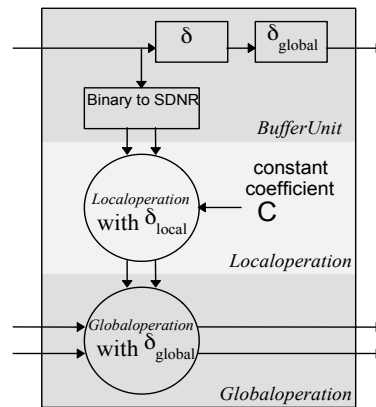


Figure 4 *Architecture of a standard Processing Element*

Design of the Basic Building Blocks

In what follows, we will present the physical implementation of the five basic building blocks stated in section 2 (the adder, multiplier, accumulator and maximum/ minimum units). These basic components were carefully designed in order to fit together with as little wastage as possible.

The ‘multiplier’ unit

The multiplier unit used is based on a hybrid serial-parallel multiplier outlined in [18]. It multiplies a serial SDNR input with a two's complement parallel coefficient $B = b_N b_{N-1} \dots b_1$ as shown in Figure 5. The multiplier has a modular, scaleable design, and comprises four distinct basic building components [19]: *Type A*, *Type B*, *Type C* and *Type D*. An N bit coefficient multiplier is constructed by:

$$\textit{Type A} \rightarrow \textit{Type B} \rightarrow (N-3) * \textit{Type C} \rightarrow \textit{Type D}$$

The coefficient word length may be varied by varying the number of type C units. On the Xilinx 4000 FPGA, *Type A*, *B* and *C* units occupy one CLB, and a *Type D* unit occupies 2 CLBs. Thus an N bit coefficient multiplier is 1 CLB wide and $N+1$ CLBs high. The online delay of the multiplier is 3.

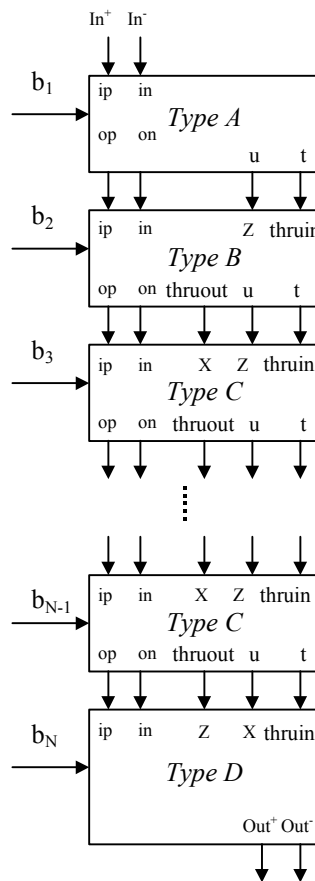


Figure 5 Design of an N bit hybrid serial-parallel multiplier

The ‘accumulation’ global operation unit

The accumulation unit is the global operation used in the case of a convolution. It adds two SDNR operands serially and outputs the result in SDNR format as shown in Figure 6. The accumulation unit is based on a serial online adder presented in [20]. It occupies 3 CLBs laid out vertically in order to fit with the multiplier unit in a convolver design.

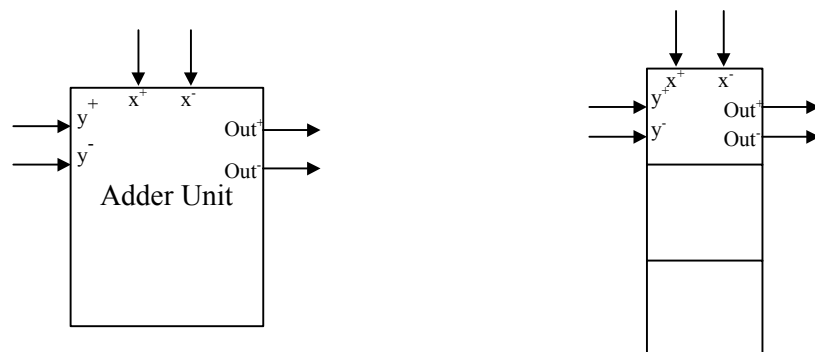


Figure 6 Block diagram and floorplan of an accumulation unit

The ‘Addition’ local operation unit

This unit is used in additive/maximum and additive/minimum operations. It takes a single SDNR input value and adds it to the corresponding window template coefficient. The coefficient is stored in 2’s complement format into a RAM addressed by a counter whose period is the pixel word length. To keep the design compact, we have implemented the counter using Linear Feedback Shift Registers (LFSRs). The coefficient bits are preloaded into the appropriate RAM cells according to the counter output sequence. The input SDNR operand is added to the coefficient in bit serial MSBF.

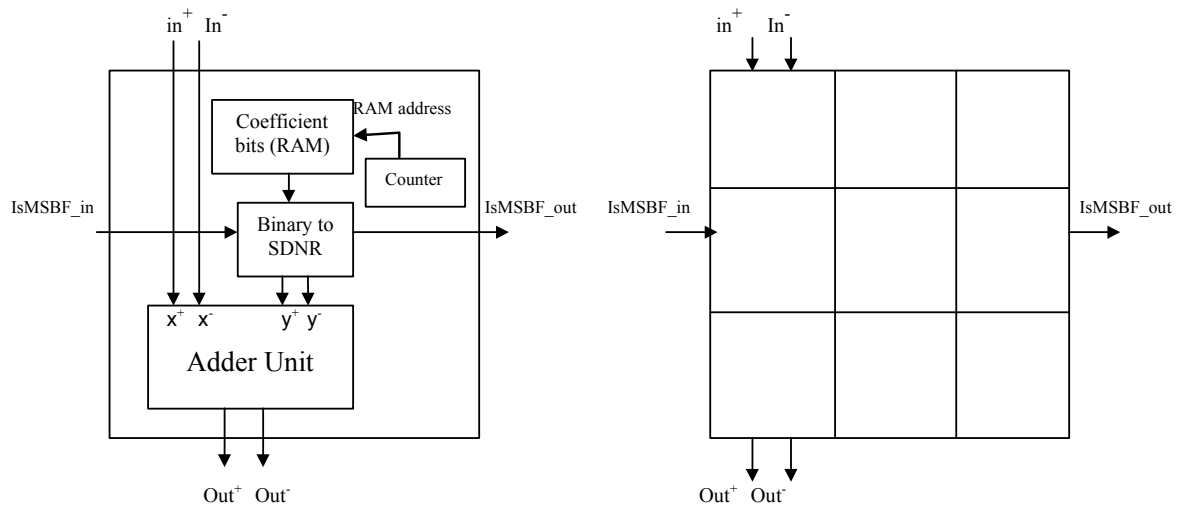


Figure 7. Block diagram and floorplan of an ‘Addition’ local operation unit

The adder unit occupies 3 CLBs. The whole addition unit occupies 9 CLBs laid out in a 3x3 array. The online delay of this unit is 3 clock cycles.

The Maximum/Minimum unit

The Maximum unit selects the maximum of two SDNR inputs presented to its input serially, most significant bit first. Figure 10 shows the transition diagram of the finite state machine performing the maximum ‘O’ of two SDNRs ‘X’ and ‘Y’. The physical implementation of this machine occupies an area of 13 CLBs laid out in 3 CLBs wide by 5 high. Note that this will allow this unit to fit the addition local operation in an Additive/Maximum neighbourhood operation. The online delay of this unit is 3, compatible with the online delay of the accumulation global operation.

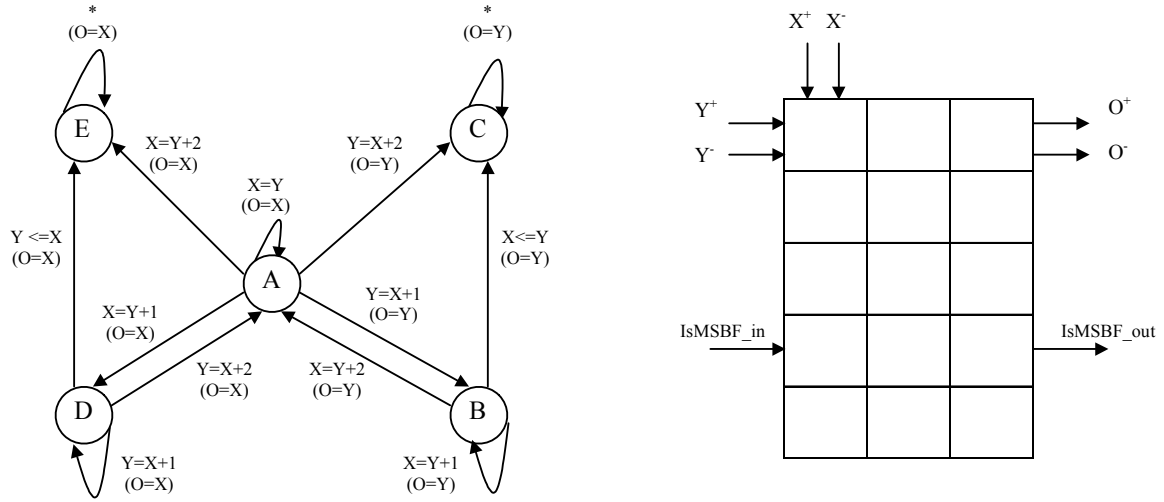


Figure 8. State diagram and floorplan of a Maximum unit

The minimum of two SDNRs can be determined in a similar manner knowing that $\text{Min}(X,Y) = -\text{Max}(-X,-Y)$.

5. THE COMPLETE ENVIRONMENT

The complete system is given in Figure 11. For internal working purposes, we have developed our own intermediate high level hardware description notation called HIDE4k [21]. This is Prolog-based [22], and enables highly scaleable and parameterised component descriptions to be written.

In the front end, the user programs in a high level software environment (typically C++ or can interact with a Dialog-based graphical interface, specifying the IP operation to be carried out on the FPGA in terms of *Local* and *Global* operators, window template coefficients etc. The user can also specify:

- The desired operating speed of the circuit.
- The input pixel bit-length.

- Whether he or she wants to use our floorplanner to place the circuit or leave this task to the FPGA vendor's Placement and Routing tools.

The system provides the option of two output circuit description formats: EDIF netlist (the normal), and VHDL at RTL level.

Behind the scenes, when the user gives all the parameters needed for the specific IP operation, the intermediate HIDE code is generated. Depending on the choice of the output netlist format, the HIDE code will go through either the EDIF generator tool to generate an EDIF netlist, or the VHDL generator tool to generate a VHDL netlist. In the latter case, the resulting VHDL netlist needs to be synthesised into an EDIF netlist by a VHDL synthesiser tool. Finally, the resulting EDIF netlist will go through the FPGA vendor's specific tools to generate the configuration bitstream file. The whole process is invisible to the user, thus making the FPGA completely hidden from the user's point of view. Note that the resulting configuration is stored in a library, so it will not be regenerated if exactly the same operation happens to be defined again.

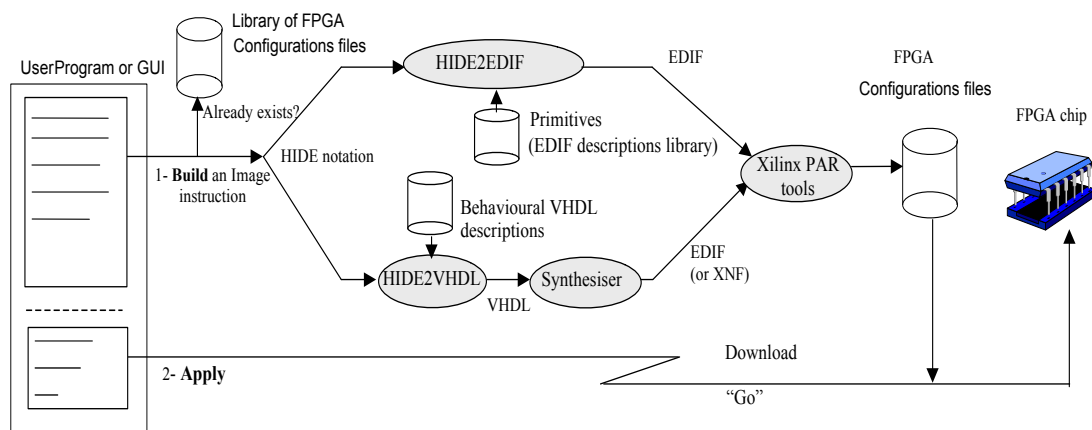


Figure 9 *Outline structure of the complete environment*

Complete and efficient configurations have been produced from our high level instruction set for all the Image Algebra operations and for a variety of complex operations including

‘Sobel’, ‘Open’ and ‘Close’. They have been successfully simulated using the Xilinx Foundation Project Manager CAD tools.

Figure 10 presents the resulting layout for a Sobel edge detection operation on XC4036EX-2 for 256x256 input image of 8-bits pixels. An EDIF configuration file, with all the placement information, has been generated automatically by our tools from the high level description in 2.1. Note that the generator optimises the design, and uses just a single shared line buffer area for the two (task parallel) neighbourhood operations. The resulting EDIF file is fed to Xilinx PAR tools to generate the FPGA configuration bitstream. The circuit occupies 475 CLBs. Timing simulation shows that the circuit can run at a speed of 75MHz which leads to a theoretical frame rate of 143 frames per second.

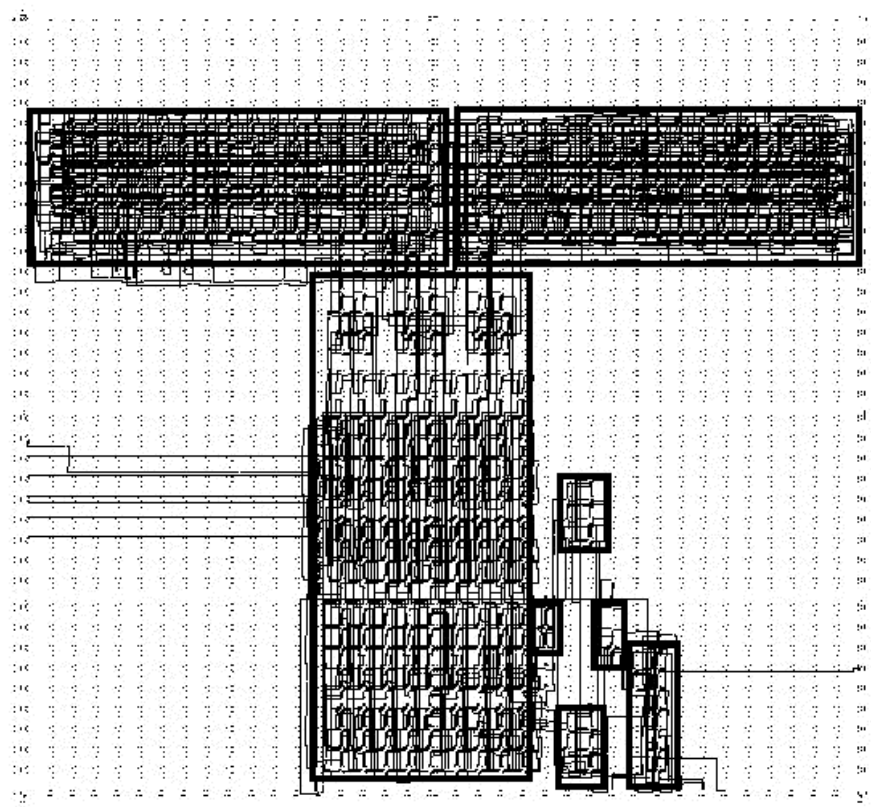


Figure 10 *Physical configuration of Sobel operation on XC4036EX-2*

Figure 11 presents the resulting layout for an 'Open' operation on XC4036EX-2 for 256x256 input image of 8-bits pixels. As previously, EDIF configuration file with all the

placement information has been generated, automatically by our tools from the corresponding high level description presented in section 2.1. The resulting EDIF file is then fed to Xilinx PAR tools to generate the FPGA configuration bitstream. The circuit occupies 962 CLBs. Timing simulation shows that the circuit can run at a speed of 75MHz which leads to a theoretical frame rate of 133 frames per second.

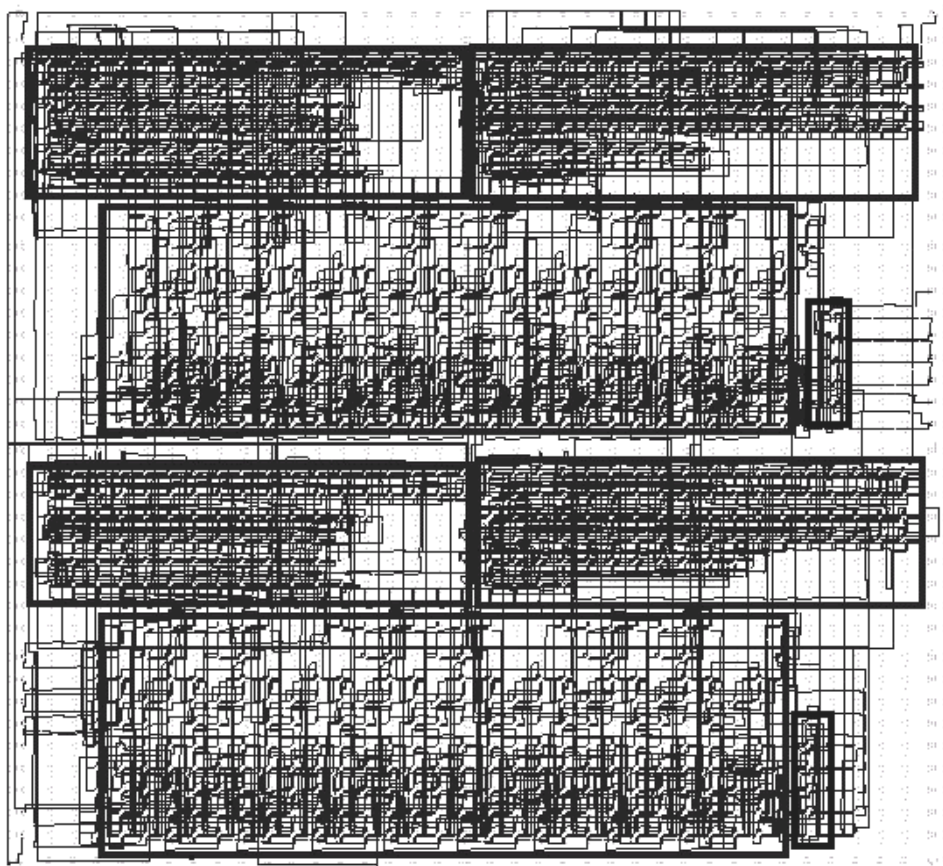


Figure 11 *Physical configuration of Open operation on XC4036EX-2*

6. CONCLUSIONS

In this paper, we have presented the design of an FPGA-based Image Processing Coprocessor (IPC) along with its high level programming environment. The coprocessor instruction set is based on a core level containing the operations of Image Algebra. Architectures for user-

defined compound operations can be added to the system. Possibly the most significant aspect of this work is that it opens the way to image processing application developers to exploit the high performance capability of a direct hardware solution, while programming in an application-oriented model. Figures presented for actual architectures show that real time video processing rates can be achieved when starting from a high level design.

The work presented in this paper is based specifically on Radix-2 SDNR, bit serial MSBF processing. In other situations, alternative number representations may be more appropriate. Sets of alternative representations are being added to the environment, including a full bit parallel implementation of the IPC [23]. This will give the user a choice when trying to satisfy competing constraints.

Although our basic approach is not tied to a particular FPGA, we have implemented our system on XC4000 FPGA series. However, the special facilities provided by the new Xilinx VIRTEX family (e.g. large on-chip synchronous memory, built in Delay Locked Loops etc.) make it a very suitable target architecture for this type of application. Upgrading our system to operate on this new series of FPGA chips is underway.

REFERENCES

- [1] Webber, H C (ed.), 'Image processing and transputers', IOS Press, 1992.
- [2] Rajan, K, Sangunni, K S and Ramakrishna, J, 'Dual-DSP systems for signal and image-processing', Microprocessing & Microsystems, Vol 17, No 9, pp 556-560, 1993.
- [3] Akiyama, T, Aono, H, Aoki, K, et al, 'MPEG2 video codec using Image compression DSP', IEEE Transactions on Consumer Electronics, Vol 40, No 3, pp 466-472, 1994.

- [4] L.A. Christopher, W.T. Mayweather and S.S. Perlman, 'VLSI median filter for impulse noise elimination in composite or component TV signals', IEEE Transactions on Consumer Electronics, Vol 34, no. 1, pp. 263-267, 1988.
- [5] J. Rose and A. Sangiovanni-Vincentelli, 'Architecture of Field Programmable Gate Arrays', Proceedings of the IEEE Volume 81, No7, pp 1013-1029, 1993.
- [6] http://www.xilinx.com/products/virtex/ss_vir.htm
- [6] Arnold, J M, Buell, D A and Davis, E G, 'Splash-2', Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, pp 316-324, June 1992.
- [7] Gigaops Ltd., The G-800 System, 2374 Eunice St. Berkeley, CA 94708.
- [8] Chan, S C, Ngai, H O and Ho, K L, 'A programmable image processing system using FPGAs', International Journal of Electronics, Vol 75, No 4, pp 725-730, 1993.
- [9] <http://www.vcc.com/>
- [10] http://www.synopsys.com/news/pubs/snug/snug99_papers/Jaffer_Final.pdf
- [11] FPL99.
- [12] Hutchings.
- [13] Xilinx 4000.
- [14] Ritter G X, Wilson J N and Davidson J L, 'Image Algebra: an overview', Computer Vision, Graphics and Image Processing, No 49, pp 297-331, 1990.
- [15] Shoup, R G, 'Parameterised Convolution Filtering in an FPGA', More FPGAs, W Moore and W Luk (editors), Abington, EE&CS Books, pp 274, 1994.
- [16] Kamp, W, Kunemund, H, Soldner and Hofer, H, 'Programmable 2D linear filter for video applications', IEEE Journal of Solid State Circuits, pp 735-740, 1990.

- [17] Avizienis A, 'Signed Digit Number Representation for Fast Parallel Arithmetic", IRE Transactions on Electronic Computer, Vol. 10, pp 389-400, 1961.
- [18] Moran, J, Rios, I and Meneses, J, 'Signed Digit Arithmetic on FPGAs', More FPGAs, W Moore and W Luk (editors), Abington, EE&CS Books, pp 250, 1994.
- [19] Donachy, P, 'Design and implementation of a high level image processing machine using reconfigurable hardware', PhD Thesis, Department of Computer Science, The Queen's University of Belfast, 1996.
- [20] Duprat, J, Herreros, Y and Muller, J, 'Some results about on-line computation of function', 9th Symposium on Computer Arithmetic, Santa Monica, September 1989.
- [21] D Crookes, K Alotaibi, A Bouridane, P Donachy and A Benkrid, 1998, 'An Environment for Generating FPGA Architectures for Image Algebra-based Algorithms', ICIP98, Vol.3, pp. 990-994.
- [22] Clocksin W F and Melish C S, 1994, 'Programming in Prolog', Springer-Verlag.