A Reconfigurable Compute Engine for Real-Time Vision Automata Prototyping

G.M. Quénot*, I.C. Kraljić, J. Sérot, B. Zavidovique

Laboratoire Système de Perception
DGA/Etablissement Technique Central de l'Armement
16 bis avenue Prieur de la Côte d'Or, 94114 ARCUEIL Cedex FRANCE
Email: ik@etca.fr

Abstract

This paper describes a Reconfigurable Compute Engine (the Data-Flow Functional Computer, or DFFC) developed at ETCA and dedicated to rapid prototyping of real-time vision automata. The Computer consists of a regular 3D array of very coarse grain application-specific FPGA called the Field-Programmable Operator Array (FPOA); each FPOA includes 2 Configurable Data-Paths (CDP) and 10 Input/Output Ports (IOP). Specific development tools allow an easy and efficient use of the Computer: a high-level description (in a functional language) is compiled into a DFFC configuration using an Operator Library. Several significant applications (connected component labeling, non-linear filtering, colored object tracking) have been implemented using our tools. An environment for automatic derivation of vision automata from a DFFC configuration is currently under development.

1 Introduction

Field-Programmable Gate Arrays have seen widespread use as reconfigurable computing elements in the past few years [2, 3, 4, 9]. The advantages are obvious: low cost, highly reconfigurable, short design time. Indeed, today's FPGAs can implement an extremely wide range of applications (glue logic functions, processors, co-processors...) due to their generic architecture: current FPGAs contain a large number of very simple basic cells. For example, Xilinx's FPGAs are composed of a large matrix (8×8 up to 24×24) of identical Configurable Logic Blocks (CLBs), each containing 1 or 2 flip-flops and a combinatorial function generator [10]. But this universality can be a drawback when a more specific range of applications is aimed at. Considering data-path-intensive image processing applications, "useful" modules (16-bit ALU, multiplier...) are rather inefficiently implemented (low operating frequency and integration). Another problem with FPGAs is their lack of simplicity if one wants to go deep into the actual FPGA implementation: the basic elements are simple gates and flip-flops.

A better approach is to have a few basic cells of a much higher complexity and adapted to the target applications. There has been attempts to develop FPGAs with more complex cells [5] but their complexity is still far too low. This paper presents a device that is suitably considered a very coarse grain application-specific FPGA for image processing. Let us call it the Field-Programmable Operator Array (FPOA). Its granularity is at the operator level instead of at the gate level. Where the FPGAs' basic blocks include a few gates, the FPOAs' basic blocks include a few 8-/16-bit data-flow operators; and where the FPGA's I/O blocks and switch matrices handle 1-bit signals, the FPOA's basic I/O ports and switch matrices handle 10-bit busses (9 bits data, 1 bit backward acknowledge).

The Field-Programmable Operator Array was designed to process digital video streams on the fly [1]: at every time step all the operations corresponding to one iteration of the algorithm for one pixel are executed in parallel (Multiple Instruction streams Single Data stream) instead of one operation executed for all the pixels (Single Instruction stream Multiple Data streams) [6]. The FPOA-based computer operates with the digital video pixel clock and each processor (FPOA basic block) executes exactly one in-

*G.M. Quénot is now with Hyperparallel Technologies, Ecole polytechnique, X-Pole, 91128 Palaiseau Cedex, FRANCE. Email: quenot@hyperparallel.polytechnique.fr

struction per cycle (one instruction can involve several operations). The coarse grain approach, although it remains limited so far to the target applications, has several advantages in terms of speed, density and easy programming.

Section 2 presents the Field-Programmable Operator Array's architecture and its programming, while Section 3 describes the Data-Flow Functional Computer compute engine (DFFC) built around the FPOA. The DFFC's development tools for rapid vision automata prototyping and a few applications are presented in Section 4. Section 5 concludes the paper and discusses future work related to the DFFC compute engine.

2 The Field-Programmable Operator Array

The FPOA's basic cells are the following:

1. an operating cell called the Configurable Data-Path (CDP).
2. an I/O cell called the Input/Output Port (IOP).
3. a routing cell called Switch Matrix.

An Input/Output Port is a bidirectional 10-bit bus interface and can be configured either as a sending, a receiving or a feedback port. Two Switch Matrices allow a Configurable Data-Path to access 6 IOPs corresponding to the 6 directions North, South, West, East, Up, Down, allowing CDPs to be mesh-connected into 3D arrays (but other 6-connected topologies are possible). Switch Matrices route full 9-bit data busses with a 1-bit backward acknowledge. Thus a complete working Field-Programmable Operator Array basic block includes 1 Configurable Data-Path, 6 Input/Output Ports, 1 Input Switch Matrix and 1 Output Switch Matrix (Figure 1).

The Field-Programmable Operator Array developed at the Système de Perception laboratory contains 2 such basic blocks (Figure 2); however in this FPOA, the 2 Configurable Data-Paths are coupled 'vertically': the upper-one's Down port is internally connected to the lower-one's Up port (thus an $n \times n$ 2D matrix of FPOAs actually implements an $n \times n \times 2$ 3D array of CDPs). This means that contrary to fine grain FPGAs, the internal and external FPOA connections are identical: there is no "break" between the inside of the FPOA and its outside. The Field-Programmable Operator Array's main features are summarized in table 1. Using a 0.6 μm technology, one can expect to include as much as 8 similar basic blocks on a single chip, and achieve operating frequencies in the 50-100 MHz range.

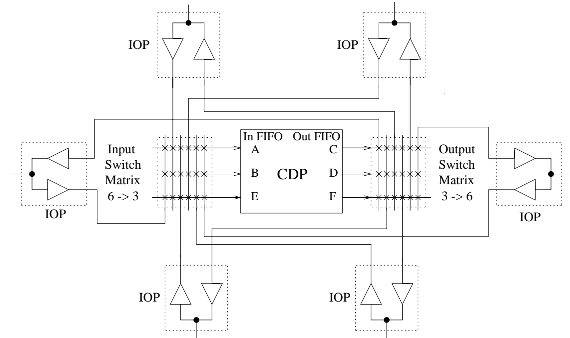


Figure 1: The Field-Programmable Operator Array basic block

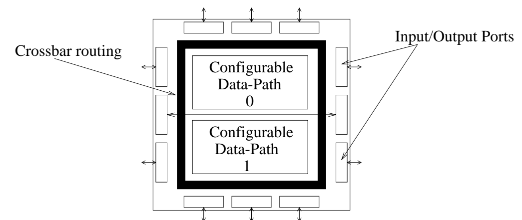


Figure 2: The Field-Programmable Operator Array

Package	144 pins PGA
Technology	1 μm CMOS
Die size	8.9 mm $\times$ 9.6 mm
Operating frequency	25 MHz
Configurable Data-Paths	2
External I/O Ports	10
Internal I/O Ports	2
Switch matrices	4
Programming technology	memory cells
Configuration bits	10,294
Logic gates	33,000
Internal memory	8.5 Kbits

Table 1: Characteristics of the Field-Programmable Operator Array

2.1 The Configurable Data-Path

The Configurable Data-Path (CDP) is the basic operating cell of the Field-Programmable Operator Array, its architecture has been designed to implement

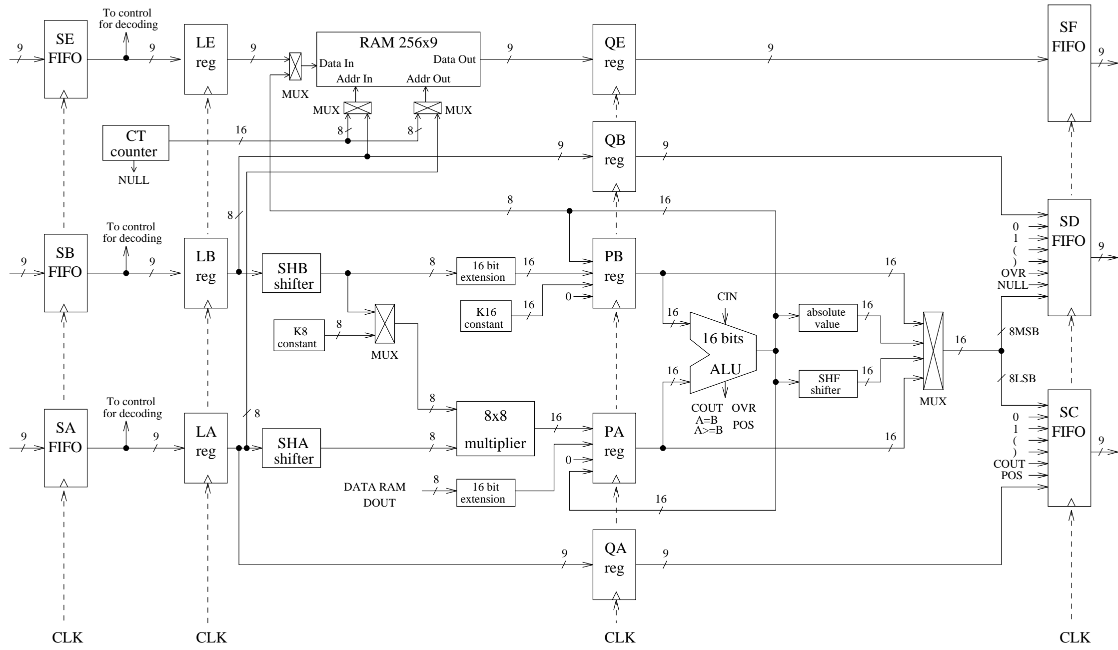


Figure 3: The Configurable Data-Path (controller not included)

efficiently a wide range of low-level image processing basic operators. Indeed, apart from arithmetic/logic operations (16-bit ALU, 8×8-bit multiplier), a single Configurable Data-Path can implement: an 8/16-bit histogrammer, an n-line/pixel delay, a line/column sum. Input to the CDP is 9-bit wide, allowing processing of 8-bit pixel numerical values (arithmetic/logic operations) as well as control values (beginning/end of frame/line; allowing line/pixel delays...).

The Configurable Data-Path is a programmable 3-stages pipelined data-path, it has 3 input stacks and 3 output stacks (Figure 3). These stacks are synchronous FIFOs with 8 9-bit words, their bandwidth is 25 MBytes/second. The FIFOs are necessary to avoid any dead-lock between CDPs. The data-path structure is the following: the first pipeline stage decodes the inputs and generates commands for the next stages, the second stage performs 8-bit operations (8×8-bit multiplication and 8-bit shifts), and the third stage performs 16-bit operations (2901-type ALU, absolute value, min/max and output shift). Other features are a 256×9 bits RAM which can be used as a FIFO, as a dual port RAM operator or as local RAM for specific purposes, and two 8-bit presetable counters cascadable into a 16-bit counter. The data-path can perform 50 millions 8- or 16-bit operations/second.

The data-path is configured through a programmable state-machine (a 64×32 bits program is provided for its description). Each 32-bit word describes the type of data required on the data-path inputs, the operations to be performed, the data to be sent to the outputs and the next state. The execution of each state is validated only if the data required on the input FIFOs is present and if there is enough room on the output FIFOs. Each of the 6 FIFOs can be connected to one I/O Port through 2 Switch Matrices (Figure 1): an input FIFO is connected to the receiving part of the port, an output FIFO is connected to the sending part of the port. It must be underlined that the receiving part of one I/O Port can be connected to 1, 2 or all 3 input FIFOs and that one output FIFO can be connected to 1, 2..., or even all 6 sending parts of the IOPs. This feature allows the replication of data flows inside the CDP. Apart from operating functions, a CDP can simultaneously be used for routing purposes, i.e. the FIFOs/IOPs not used for the operating part can be configured as a routing channel.

2.2 Programming an FPOA

The function of a Configurable Data-Path is defined by:

1. A static programming register containing static definitions for the data-path (ALU operation, constants...).

2. A state-machine definition describing dynamic definitions for the data-path.

Programming its IOPs and Switch Matrices is done through a static configuration register independent from the CDP function, which specifies connections between FIFOs and IOPs. Thus both the functionality and the connections of a Field-Programmable Operator Array are defined by (re)programmable elements (actually memory cells) scattered through the Configurable Data-Paths, the I/O Ports and the Switch Matrices.

Practically, the FPOA is programmed (and tested) through a 1-bit scanpath, which internally splits into 6 different scanpaths. The whole programming (including scanpath selection) is done via 6 input lines: a clock, a reset, and a 4-bit command bus. All the FPOA's flip-flops (static programming and configuration registers, input/output FIFOs, counters, pipeline registers, data-path constants) are daisy-chained into scanpaths. Programming the program 64×32 RAM and the data 256×9 RAM is done by first programming the input FIFOs and then transfering their contents to the RAMs. A total of 5147 bits is needed to configure a Configurable Data-Path entirely (including data and state machine RAMs and FIFOs), and it takes about 15 ms per CDP to achieve the configuration.

3 The Data-Flow Functional Computer

The Data-Flow Functional Computer (DFFC) is a 3D $8 \times 8 \times 8$ array of FPOAs physically made of 8 boards containing each 8×8 FPOAs (see figures 4,5), thus implementing a 3D $8 \times 8 \times 16$ array of CDPs. The 1024 available data-paths allow large data-flow graphs to be implemented. Several different graphs can be simultaneously implemented and independently executed as long as they fit in the DFFC's 1024 Configurable Data-Paths. The highly regular structure of the Data-Flow Functional Computer allows only local connections (there are no "long lines" as in FPGAs). However long distance connections are possible through basic blocks configured as routing elements. An optimized place and route can yield a 30 to 50% CDP operating rate on a 3D array (this rate would be significantly lower on a 2D array). Among other possible topologies is a 2D array with 4-connectivity and longer connections using the 2 remaining I/O Ports.

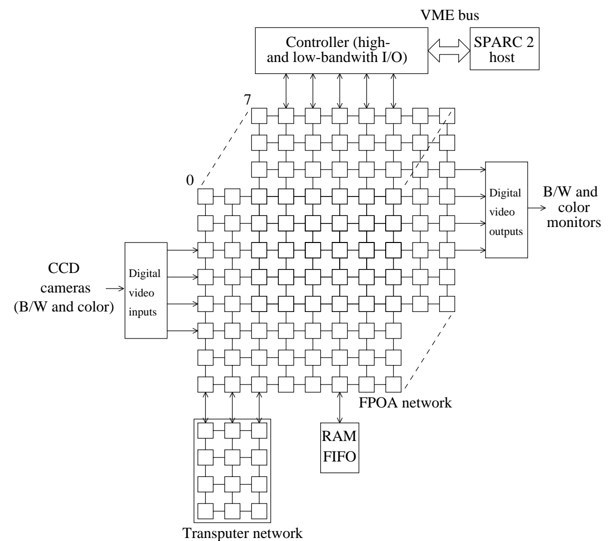
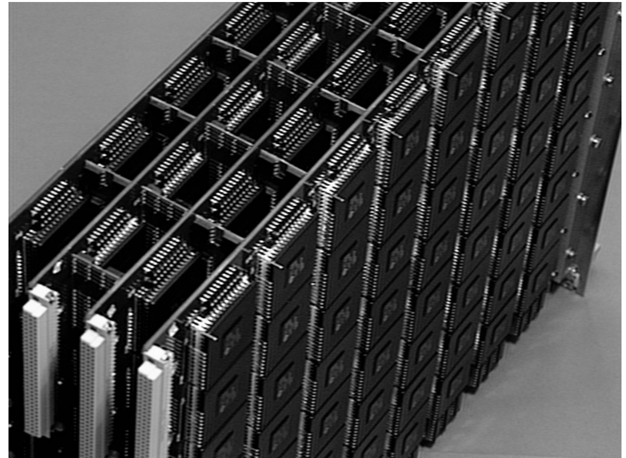


Figure 4: The Data-Flow Functional Computer's architecture

The DFFC's inputs/outputs are B/W and color cameras/monitors and low-bandwidth data-flow I/O ports. A SPARC 2 workstation is the host for the DFFC's programming environment. Current extensions include RAM boards configurable as Look-Up Tables or FIFOs; but any extension can be connected to the DFFC as long as it meets the IOP protocol (synchronous FIFO-like, 9 bits data, 1 bit acknowledge). The FPOAs' scanpaths are daisy-chained, thus the Computer is configured as easily as a single FPOA, through a scanpath and a few control signals. The presence of a T800 Transputer board allows to emulate also high-level image processing algorithms. The SPARC 2 host workstation runs the user-friendly graphical Development Tools implementing all the design flow layers: from high-level (design specification and input) to low-level (configuration and running of the DFFC) including the debugging phase. The Computer is synchronized with the digital video pixel clock (up to 25 MHz pixel; actually, the digital video I/Os are slave to the computer clock). A data flow requires 4 clock cycles to pass through a basic block: 3 internal CDP pipeline stages plus a cycle needed for the data to go from the output FIFO(s) to the neighbour's input FIFO(s). The 25 MHz maximum operating frequency is not a flip-flop toggle limit but the actual speed at which every part of the FPOA (data-path, state-machine, memory, FIFOs) can operate whatever the complexity of the operator assigned to a basic block. Unlike fine grain FPGA-based hardware emulators, the DFFC can operate at the actual speed

(a)



(b)

Figure 5: The Data-Flow Functional Computer (a) and a 4×8×8 array of FPOAs (b)

of the emulated hardware since it is optimized at the operator level.

4 The DFFC Development Tools

The Functional Computer operates within a concept called **emulation-derivation** (Figure 6):

1. Emulation: an image processing algorithm is emulated in real-time on the Functional Computer. This phase **proves the existence of an architecture implementing the algorithm and validates its real-time behaviour on real-world scenes.**
2. Derivation: the validated Functional Computer configuration implementing the architecture is then automatically analyzed and **the actually used resources are optimized**. Two analysis levels are under investigation: the coarse one would produce a moderately optimized version using FPOAs and standard chips (FIFOs, RAMs), and the finer one would produce a fully optimized version using custom VLSI chips.

4.1 Emulation

The basic idea behind the implementation of an algorithm is to express it as a data-flow graph. The image processing algorithm to be emulated is expressed in a Functional-Programming language source file [8]. A compiler converts this source file into an operator Data-Flow Graph which is then translated into a Configurable Data-Path Graph. The compilation/translation phases use an Operator Library containing Data-Flow operators implementing the language primitives. Currently 2 libraries are available:

- A low-level library containing currently 150 single CDP operators (arithmetic and logic, comparators, constants, counters, 8-bit histogrammers, multiplexers...).
- A medium-level library containing 50 multiple-CDP macro-operators (convolvers, LUTs, 16- and 24-bit histogrammers...).

Whereas the low-level library is complete (it contains all the language basic primitives), the medium-level library can be extended easily with no need for intimate knowledge of the CDP's architecture.

The CDP Graph is mapped into an actual DFFC Network Configuration ready to be loaded. A complete 1024 FPOAs configuration is loaded in about 15 seconds. The algorithm is executed in real-time using the CCD cameras as input flows and sending the output flows to video monitors. Debugging is done by executing the algorithm step-by-step using the low-bandwidth interfaces and the possibility to access any

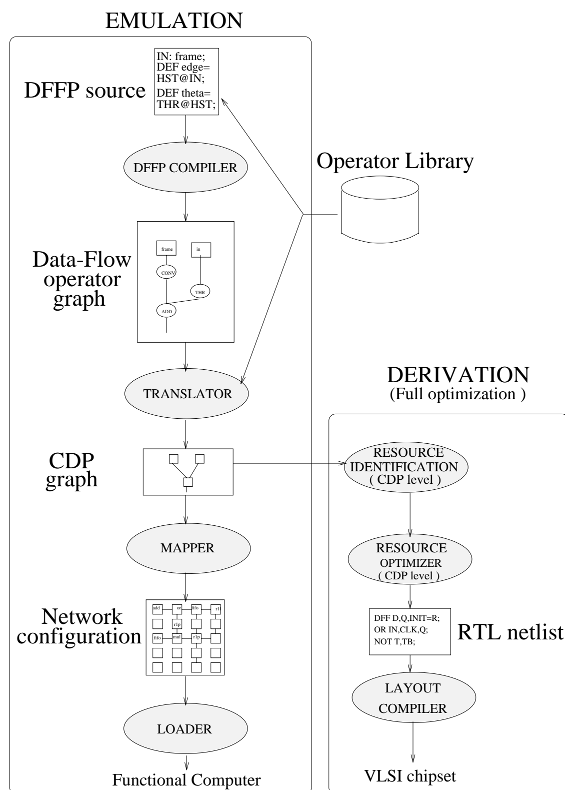


Figure 6: Design flow

FPOA register as well as the contents of the RAMs and FIFOs from the host. A Graph Compiler session is shown in Figure 7, while Figure 8 shows a typical Interactive Configuration Editor session. Both programs run on the SPARC 2 host.

We have developed some significant applications such as a non-linear Nagao-like filter, a connected component labeling and a colored object tracking (fully described in [7]). Table 2 shows a few examples of actual algorithms and their global resource utilization, i.e. the percentage of used ALUs, multipliers, Data RAMs... in each Configurable Data-Path. The CDP quotation in the table does not include CDPs used only for routing.

A Nagao-like filter is a non-linear "edge-preserving smoothing" filter, i.e. it removes high frequency spatial noise while preserving the edges of the original image. The functional programming language description of a Nagao-like filter emulated on the Functional Computer is shown in appendix A. Figure 9 shows results of our Nagao-like filter.

Connected component labeling consists in assigning a unique label to each connected region of a binary image. Appendix B describes the functional programming language source of the connected component labeling implemented on the DFFC computer. Results of a labeling are shown in figure 10.

4.2 Derivation

Our current research is the derivation of vision automata from emulation results. The actual DFFC configuration implementing the application is used as an input to the derivation software. A first approach is to replace the Configurable Data-Paths used only for routing by direct connections and the Configurable Data-Paths used as FIFOs by commercial FIFOs, and to keep other FPOAs (exactly as an algorithm emulated on a "classical" fine grain compute engine can be turned into a mix of FPGAs and standard LSI/MSI devices). This could result in an automaton implementable as a board or Multi-Chip Module (MCM). A second approach is based on a CDP-level 2-step analysis: identification of the effectively used CDP resources and optimization of these resources. The resulting automaton can be synthesized as a VLSI chipset (similar to the turn of an FPGA implementation into an ASIC). As indicated in table 2, a significant resource reduction can be expected at CDP level (local resource optimization). A global resource optimization phase will further reduce the chipset area; it will include resource sharing, controller collapsing.

5 Conclusion

We have presented a Data-Flow Functional Computer aimed at the rapid prototyping of vision automata. Algorithms are first emulated in real-time on the Computer, and our goal is to automatically derive the corresponding computer configurations into boards/MCM/VLSI chipsets, same behaviour and performance guaranteed but reduced resources. The Computer consists of a network of 512 Field-Programmable Operator Arrays. The FPOA is a custom-built very coarse grain FPGA whose architecture is adapted to implement efficiently low-level image-processing algorithms. A user-friendly graphical programming environment has been developed; it compiles a Functional Programming source file into a DFFC network configuration using an extensible Operator Library currently containing more than 200 low- to mid-level operators. Several significant applications have been successfully implemented on the Computer using our development tools. An environment for automatic derivation of vision automata from

DFFP Grapher

File: fpg.fp (Load) Save (Compile) Macro Expansion Level: 7 (Layout..) Zoom (Quit)

DFFP to Graph Tutorial Compiler - Vers 1.2

DFFP Source

```
// DFFC ALGORITHM for EXTRACTING SEGMENTS ALONG PREDOMINANT DIRECTIONS

CONSTANT Nl = 572;                      // Video frame dimensions
CONSTANT Np = 800;
CONSTANT Nd = 2;                        // Number of directions
CONSTANT Lpk = [1,2,1,2,4,2,1,2,1];     // Low-pass convolution kernel

BEGIN dirSegs (Depth=10) [      // Extracts main theta-oriented segments
  INPUT  edges:TRAME(PIXEL);    // Binary image of theta oriented edges
  INPUT  theta:PIXEL;           // Orientation (0..255)
  OUTPUT segs:TRAME(PIXEL);]    // Binary image of selected segments

// Compute the projection along theta using a oriented gray level ramp
DEF rampTheta = RAMP(Nl,Np,Depth).theta;
DEF projTheta = HISTO(Depth).GATE.[ rampTheta, edges ];

// Extract main theta-oriented lines
DEF lines = FILT.[ rampTheta, PEAKS(0).projTheta ];

// Select segments within edges
DEF segs = MSK.[ edges, lines ];
END

MAIN [
  VIDEO INPUT    videoIn:TRAME(PIXEL);  // Original gray-level frame(s)
  VIDEO OUTPUT  videoOut:TRAME(PIXEL);  // Output binary images
  ASYNC INPUT      thrIn:PIXEL;]        // Gradient magnitude threshold

// Smooth image to reduce quantization effects
DEF lowPassed = CONV(Lpk[]).videoIn ;

// Computes direction and magnitude of gradient
DEF grad = GRAD.lowPassed ;
DEF modGrad = 1.grad;
DEF dirGrad = 2.grad;

// Detect edge points by thresholding the gradient magnitude
DEF edges = THR.[ modGrad, MU(1).thrIn ] ;

// Compute histogram of gradient orientation at significant points
DEF hisDir = HISTO.GATE.[ dirGrad, edges ] ;

// Compute 2 main edge orientations from this histogram
DEF mainDirs = PEAKS(Nd).hisDir ;
DEF theta1 = NTH(1).mainDirs;
DEF theta2 = NTH(2).mainDirs;

// Select edges along this directions
DEF edges1 = MSK.[ MSK.[ dirGrad, theta1 ], edges ];
DEF edges2 = MSK.[ MSK.[ dirGrad, theta2 ], edges ];

// Extract 1st and 2nd main direction lines and output result
DEF segs1 = dirSegs:[ edges1, theta1 ];
DEF segs2 = dirSegs:[ edges2, theta2 ];
DEF videoOut = OR.[ segs1, segs2 ] ;
END
```

Log Display

```
fpc: 0 error(s) detected, 0 warning(s) flagged
fpg: layout: initial cost=28... final cost=2
```

Figure 7: The Graph Compiler

Figure 8: The DFFC's Interactive Configuration Editor

name	Edge detector	3×3 convolution	2048 word LUT	Nagao-like filter	Erosion
CDP number	13	22	25	86	110
I/O ports %	41	43	51	50	47
input FIFOs %	46	56	70	63	62
output FIFOs %	35	31	32	36	33
multiplier %	0	41	0	0	0
true ALU op %	23	36	0	37	43
Data RAM %	15	18	32	13	10
counters %	30	54	0	41	54
Prog RAM size%	3	8	0	9	10

Table 2: Global resource utilization for different algorithms

emulation results is under development.

The DFFC Compute Engine is dedicated to real-time image processing, and in this specific field of applications the advantages of a coarse grain Field-Programmable Operator Array over a standard fine grain FPGA are obvious. The operating frequency is application-independent, i.e. the real-time execution is guaranteed for any application. This is a major improvement over the unpredictable operating frequency of current FPGAs. The second important advantage is the easy programming and debugging of the Data-Flow Functional Computer. Indeed, the FPOA architecture allows the implementation of the functional data-flow paradigm to its fullest, thus permitting the programming of the Functional Computer from a high-level language with no involvement by the user in the translation process. But the application-specific architecture of the Field-Programmable Operator Array implies its unability to implement any kind of circuitry. This is actually the main drawback of the FPOA over a standard FPGA. We believe nonetheless that this lack of flexibility is largely compensated by the FPOA advantages cited above.

This Field-Programmable Operator Array is adapted for the on the fly computing of 8/16-bit integer values, and its field of applications is thus not restricted to real-time image processing. However applications outside of the image processing range have not been studied as yet. More generally, the Field-Programmable Operator Array concept (very coarse grain application-specific FPGA) can be adapted to other applications. New FPOAs should be developed to suit the application: for example, a 64-bit floating point basic operating cell would be adapted to the resolution of partial derivative equations by finite difference methods.

References

[1] E. Allart and B. Zavidovique. Functional Image Processing through Implementation of Regular Data-Flow Graphs. In *21st Annual Asilomar Conference on Signals, Systems on Computers, Pacific Grove, CA*, Nov 1987.

[2] J.M. Arnold, D.A. Buell, and E.G. Davis. Splash 2. In *Proc. of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–322, 1992.

[3] P. Bertin, D. Roncin, and J. Vuillemin. *Programmable Active Memories: a Performance Assessment (PRL Research Report 24).* Digital Equipment Corporation, Paris Research Laboratory, March 1993.

[4] S. Casselman. Virtual Computing and The Virtual Computer. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA*, pages 43–48, April 1993.

[5] D. Hill, B. Britton, B. Oswald, N-S. Woo, S. Singh, C-T. Chen, and B. Brambeck. ORCA: A New Architecture for High-Performance FPGAs. In *Proceedings of the Second International Workshop on Field-Programmable Logic and Applications, Vienna, Austria*, pages 52–60, September 1992.

[6] S.Y. Kung, S.C. Lo, S.N. Jean, and J.N. Hwang. Wavefront Array Processors: Concept to Implementation. *IEEE Computer 20(11)*, pages 18–33, July 1987.

[7] G. Quénot, C. Coutelle, J. Sérot, and B. Zavidovique. Implementing Image Processing Applications on a Real-time Architecture. In *Proceed-*

ings of Computer Architectures for Machine Perception, New Orleans, USA, pages 34–42, December 1993.

[8] J. Sérot, G.M. Quénot, and B. Zavidovique. Functional Programming on a Data-flow Architecture: Applications in Real-Time Image Processing. *International Journal of Machine Vision and Applications*, May 1993.

[9] D. Van den Bout, J. Morris, D. Thomae, S. Labrozzi, S. Wingo, and D. Hallman. Anyboard: An FPGA-based, reconfigurable system. *IEEE Design & Test of Computers*, pages 21–30, September 1992.

[10] Xilinx. *The Programmable Logic Data Book*. 1993.

A Nagao-like filter functional programming source

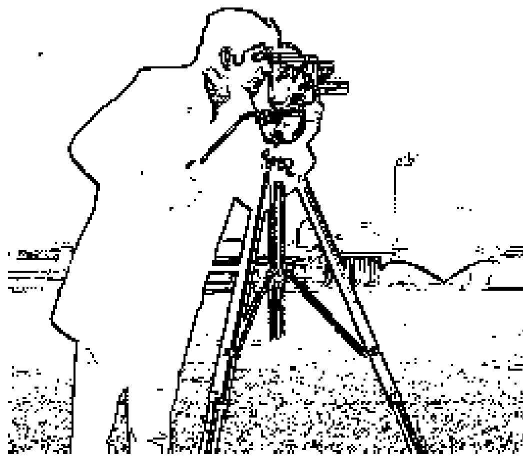
All the undefined functions in the source files are standard library functions. These functions implement either a single Configurable Data-Path operator (e.g. `SUB`) or a multiple-CDP macro-operator (e.g. `CONV3`).

```
// Functional source implementing a Nagao-like non linear filter

// Line pixel number
CONSTANT Np=768;
// Normalized low-pass filter coef.
CONSTANT Lp=1, 2, 1, 2, 4, 2, 1, 2, 1, 16

// MAIN ——-
BEGIN MAIN [
VIDEO INPUT In : FRAME(PIXEL);
VIDEO OUTPUT Out : FRAME(PIXEL); ]

// 3x3 neighborhood extraction (input: original frame)
DEF Vi = NEIG3 @ In;

// Disparity computation
DEF Ds = SUB @ [MAX Vi, MIN Vi];

// Low pass filtering (3x3 convolution using the Lp mask)
DEF Mn = CONV3(Lp) @ In;

// 3x3 neighborhood extraction (input: disparity frame)
DEF Vd = NEIG3 @ Ds;

// 3x3 neighborhood extraction (input: low pass filtered frame)
DEF Vm = NEIG3 @ Mn;

// Partial associative sorting

DEF Min1 = 1 @ Vd;
DEF Tmp1 = 1 @ Vm;
DEF Min2 = MIN @ [Min1, 2 @ Vd];
DEF Tmp2 = MUX @ [Tmp1, 2 @ Vm, EQU @ [Min2, 2 @ Vd]];
DEF Min3 = MIN @ [Min2, 3 @ Vd];
DEF Tmp3 = MUX @ [Tmp2, 3 @ Vm, EQU @ [Min3, 3 @ Vd]];
DEF Min4 = MIN @ [Min3, 4 @ Vd];
DEF Tmp4 = MUX @ [Tmp3, 4 @ Vm, EQU @ [Min4, 4 @ Vd]];
DEF Min5 = MIN @ [Min4, 5 @ Vd];
DEF Tmp5 = MUX @ [Tmp4, 5 @ Vm, EQU @ [Min5, 5 @ Vd]];
DEF Min6 = MIN @ [Min5, 6 @ Vd];
DEF Tmp6 = MUX @ [Tmp5, 6 @ Vm, EQU @ [Min6, 6 @ Vd]];
DEF Min7 = MIN @ [Min6, 7 @ Vd];
DEF Tmp7 = MUX @ [Tmp6, 7 @ Vm, EQU @ [Min7, 7 @ Vd]];
DEF Min8 = MIN @ [Min7, 8 @ Vd];
DEF Tmp8 = MUX @ [Tmp7, 8 @ Vm, EQU @ [Min8, 8 @ Vd]];
DEF Min9 = MIN @ [Min8, 9 @ Vd];
DEF Tmp9 = MUX @ [Tmp8, 9 @ Vm, EQU @ [Min9, 9 @ Vd]];

LET Out = Tmp9;

END
```

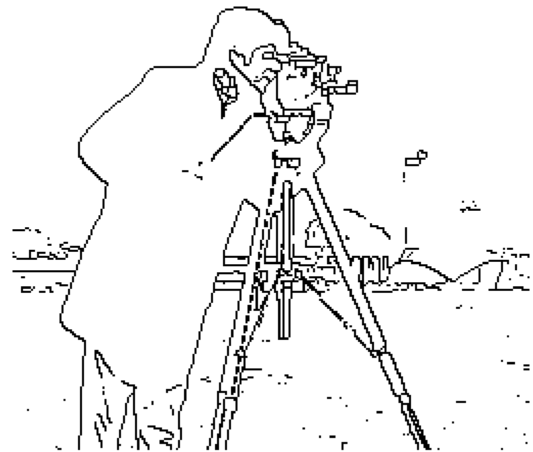
B Connected component labeling functional programming source

Note: one Transputer is involved in the equivalence-table compression (computing of a unique label for all the connected regions).

```
// Connected component labeling functional source

// Line pixel number
CONSTANT Np=768;

// MAIN ——-
BEGIN MAIN [
VIDEO INPUT In: FRAME(PIXEL);
VIDEO OUTPUT Out : FRAME(PIXEL);
ASYNC INPUT Thr : PIXEL; ]

// Input 8-bit coded grey level image thresholding
DEF Bin= THRLD @ [In, $ @ Thr];

// Binary image horizontal labeling
DEF Lbh = HLABG @ Bin;

// Label propagation
DEF Lbv = VLABG @ [Lbh, Bin];

// Equivalent label detection
DEF Edl = EQDET @ [Bin, Lbv];

// Label equivalence resolution
DEF Ers = EQRES @ Lbv;

// Dynamic label look-up table loading and label image
transcoding
DEF Lbl = LABLUT @ [Lbv, Ers];

// Label image outputting
LET Out = Lbl;

END
```



(a) Edge detection without preliminary Nagao-like filtering



(b) Edge detection with preliminary Nagao-like filtering

Figure 9: Nagao-like filtering

(a) Original binarized picture

(b) Resulting picture

Figure 10: Connected component labeling