Published in *Systolic Array Processors*, J. McCanny, J. McWhirter and E. Swartzlander Jr. (eds.), Prentice Hall International, 1989, pp. 589–598.

COMPUTER-BASED TOOLS FOR REGULAR ARRAY DESIGN

Wayne Luk and Geraint Jones
Programming Research Group, Oxford University Computing Laboratory, 11 Keble Road, Oxford, England OX1 3QD

Mary Sheeran
Computing Science Department, Glasgow University, Glasgow, Scotland G12 8QQ

Abstract. We present an overview of a prototype system based on a functional language for developing regular array circuits. The features of a simulator, floorplanner and expression transformer are discussed and illustrated.

INTRODUCTION

Implementing algorithms on a regular array of processors has many advantages. Besides offering an efficient realisation of parallel structures, regular patterns of interconnections also provide an opportunity for simplifying their description and their development. Various approaches for regular array design have been proposed; examples include methods based on dependence graphs [5], recurrence equations [14], and algebraic techniques [16].

This paper presents an overview of a prototype system for regular array development. The system is based on μFP [15], a functional language with mechanisms for abstracting spatial and temporal iteration. These abstractions result in a succinct and precise notation for specifying designs. Moreover, the explicit representation of various forms of spatial iteration simplifies the production of layouts, and the declarative nature of the language allows designs to be refined by simple equational reasoning. Our aim is to exploit these features of μFP to provide an integrated set of tools for capturing, exploring, refining and evaluating regular array designs.

Various versions of the tools have been written in the functional language Orwell [18], whose simplicity and conciseness make it easy to investigate alternative implementations. We have prototyped facilities for simulation, floorplanning and expression transformation; these will be reviewed and demonstrated in the following sections.

THE LANGUAGE

Designs are represented as expressions in μFP, a descendant of the functional language FP [1]. Objects in μFP are either atoms (such as numbers) or tuples of objects: for instance the object $\langle 0, \langle 1, 2\rangle\rangle$ is a 2-tuple containing the number 0 and the tuple $\langle 1, 2\rangle$. A stream is an infinite tuple of objects which one can regard as the values of a signal at successive clock cycles. The behaviour of a design is captured by a function that maps an input stream to an output stream; so $\mathit{Add}\ \langle\langle 1,2\rangle,\langle 3,4\rangle,\langle 5,6\rangle,\ldots\rangle = \langle 3,7,11,\ldots\rangle$. Since one can describe a stream by a function that delivers the stream irrespective of its argument, the use of streams abstracts time indices from circuit descriptions.

A higher-order function takes one or more functions as argument and returns a function as result. Combinators are higher-order functions that capture common patterns of computation as parametrised expressions. These patterns can be used to describe behaviour, in which case the behaviour of a composite device is expressed in terms of the behaviour of its components; or they can be used to encapsulate the spatial organisation of a circuit, in which case they describe the wiring together of components to form the composite device.

Consider first of all the combinator *sequential composition*, which corresponds to connecting the input of one component to the output of another:

$$(F \; ; \; G)\, x \quad = \quad G\,(F\, x)$$

(we use reverse composition to conform with the convention that signals flow from left to right and also to preserve compatibility with relational description of circuits [17]). *Construction*, on the other hand, corresponds to broadcasting the input to each component of a composite circuit. The output will be a stream of n-tuples where n is the number of components in the construction. Figure 1 shows possible geometric interpretation of $(F \; ; \; G)$ and $[F, G]$.

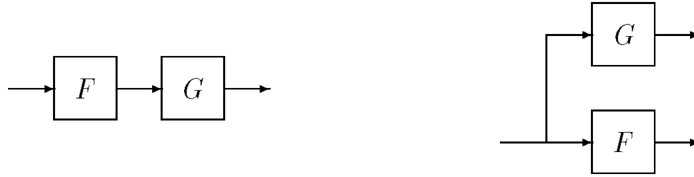


Figure 1 Sequential composition and construction.

It is not difficult to see that $(F \; ; \; [G, H]) = [(F \; ; \; G), (F \; ; \; H)]$. μFP contains a set of such algebraic theorems, which equate distinct expressions with identical behaviour. They can be verified by the semantics of μFP [15], but we do not go into the details here.

A stream of tuples can be manufactured by left- and right-handed append functions

$$\begin{aligned} [x, [y_0, y_1, \ldots, y_{n-1}]] \; ; \; App_L &= [x, y_0, y_1, \ldots, y_{n-1}], \\ [[y_0, y_1, \ldots, y_{n-1}], x] \; ; \; App_R &= [y_0, y_1, \ldots, y_{n-1}, x], \end{aligned}$$

and one can also select the first or the second of every element of a stream of pairs,

$$\begin{aligned} [x, y] \; ; \; El_1 &= x, \\ [x, y] \; ; \; El_2 &= y. \end{aligned}$$

Their respective inverses, ${El_1}^{-1}$ and ${El_2}^{-1}$, pair every item of a stream with an undefined object, so that $({El_1}^{-1} \; ; \; El_1) = ({El_2}^{-1} \; ; \; El_2) = Id$ where Id is the identity function: $(Id \; ; \; x) = (x \; ; \; Id) = x$.

Since pairs of circuits frequently arise, we have

$$\begin{aligned} [x, y] \; ; \; \mathsf{fst}\, F &= [(x; F), y] \quad \text{(apply to first)}, \\ [x, y] \; ; \; \mathsf{snd}\, F &= [x, (y; F)] \quad \text{(apply to second)}. \end{aligned}$$

A pair of circuits with orthogonal interconnections can be placed beside or above each other (Figure 2),

$$\begin{aligned} [a,[b,c]]\ ;\ F{\rightarrow}G &= [[p,q],r] \text{ where } [a,b];F=[p,s] \text{ and } [s,c];G=[q,r], \\ [[a,b],c]\ ;\ F{\downarrow}G &= [p,[q,r]] \text{ where } [b,c];F=[s,r] \text{ and } [a,s];G=[p,q]. \end{aligned}$$

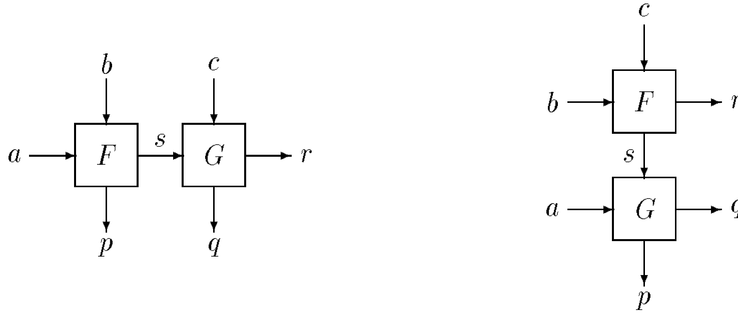


Figure 2 Beside and above.

Next, examples of combinators that capture common cases of spatial iteration will be given (Figure 3). Repeated sequential composition is given by

$$\begin{aligned} F^0 &= Id, \\ F^{n+1} &= F\ ;\ F^n, \end{aligned}$$

while repeated parallel composition is given by

$$[x_0, x_1, \ldots, x_{n-1}]\ ;\ \propto F = [(x_0;F),(x_1;F),\ldots,(x_{n-1};F)]$$

($\propto F$ is often pronounced as "map F").

Triangular arrays of latches often arise at the boundaries of pipelined circuits, so we have the combinators *right* and *left triangle*

$$\begin{aligned} [x_0, x_1, \ldots, x_{n-1}]\ ;\ \triangle_R F &= [x_0,(x_1;F),\ldots,(x_{n-1};F^{n-1})], \\ [x_0, x_1, \ldots, x_{n-1}]\ ;\ \triangle_L F &= [(x_0;F^{n-1}),(x_1;F^{n-2}),\ldots,x_{n-1}]. \end{aligned}$$

Horizontal and vertical forms of *reduction* can be used to implement continued sums and similar kinds of loops. For example, $[0,[x_0,x_1,x_2,\ldots]];/_H Add = \sum x_i$, where 0 is the function that produces a constant zero output. Horizontal and vertical *arrays* are generalisation of reduces that provide an immediate output after each iteration.

These circuit forms also obey various theorems which can be used for circuit optimisation. For instance, provided that $(\mathcal{D};\mathcal{A}) = Id$, one can prove by induction that

$$\backslash_H F = \mathsf{snd}(\triangle_R \mathcal{D})\ ;\ \backslash_H(F;\mathsf{snd}\mathcal{D})\ ;\ App_R\ ;\ \triangle_R \mathcal{A}. \qquad (1)$$

Such μFP theorems are especially useful for optimising regular array circuits by pipelining [16].

THE TOOLS

We now provide an overview of the design tools for the notation introduced in the preceding section. A user interacts with the tools through a command interpreter which acts as an interface to the main modules and the support facilities. The support facilities perform a variety of housekeeping functions and convert between external and internal data representation; the main modules include a simulator, a floorplanner, and an expression transformer.

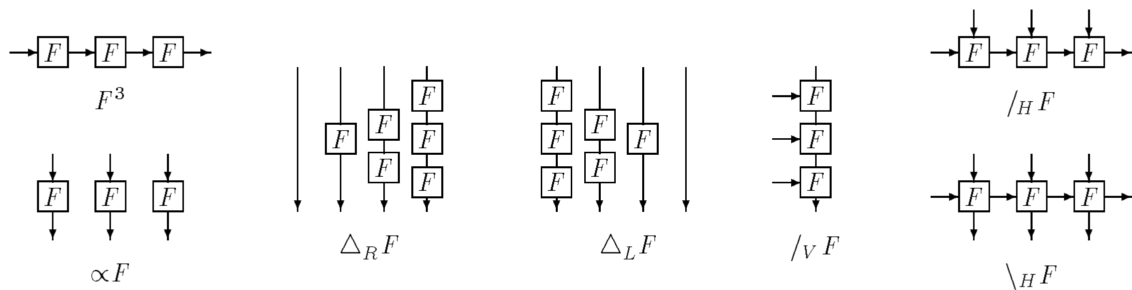


Figure 3 Geometric interpretation of some combinators.

Housekeeping functions

The housekeeping functions include support for maintaining a menu-driven system and on-line help facilities, for summarising, examining and editing contents of the databases, and for interfacing the databases to the external file system.

Currently the system contains two databases: one for storing expressions in abstract syntax, and the other containing geometric information for floorplanning.

Converting between representations

The external representation of μFP expressions is in concrete syntax; it is the form that humans prefer to use. Alternative representations of concrete syntax are used for symbols not normally found on a typewriter keyboard – so for example $\propto F$ is represented by `@F`, $F \downarrow G$ by `F \|/ G`, and $\triangle_R F$ by `/\R F`.

Tree structures are used in the abstract syntax of μFP. They form the internal representation which is interpreted and manipulated by the main modules.

There is a parser for converting expressions in concrete syntax to the corresponding representation in abstract syntax. The two main operations involved are identifying the type of an expression – for example, whether it is a primitive or a combinator – and converting infix operators to prefix form. The "unparser" does the opposite: it converts the internal representation to the external syntax for pretty printing.

Simulator

The simulator is basically a behaviour-interpreter for μFP expressions; it also contains various functions for generating constant or varying streams. Multi-phase clock systems and n-slow circuits [7] can be simulated by functions interleaving n streams or selecting every n^{th} element of a stream; the latter can also be used for selecting specific cycles of simulation output. Stream functions are implemented by translating them to object-level functions using the rules in [15] – so for example stream addition is implemented by mapping integer addition over an infinite tuple of pairs of integers. The implementation of the simulator has been simplified since Orwell itself allows higher-order functions and infinite data structures.

Most simulators only accept numeric inputs and produce numeric outputs. For many problems it is often more instructive to obtain a symbolic description of the outputs in terms of the inputs. A novel feature of our simulator is its ability to handle both numeric and symbolic data: this enables the use of the same circuit description

for producing both numeric and symbolic results. Symbolic simulation is especially useful for locating errors in the circuit description.

Other features of the simulator include the simulation of designs at various levels of abstraction simultaneously (for instance part of the circuit can be described at word-level while other parts are represented at bit-level), and output can be produced as rectangular waveforms for bit-level signals.

Floorplanner

The floorplanner sketches a picture of a design. It takes the same circuit and input descriptions as those for the simulator. The circuit can be drawn with the input and output connections aligned to the horizontal, or to the vertical, or to both horizontal and vertical if the data is in the form of a two-tuple. The size and connection positions can be supplied by the user and are stored in the relevant database; if not, default values are used.

There are four components in the floorplanner: a placer, a sizer, a router and an output generator. The placer produces from the internal representation of the design a hierarchical description of the layout. The subcircuits are placed according to a set of rules currently embedded in the code of the placer. Next, the sizer adds the dimensions and the connection positions to the description of the primitive cells, and the result is then passed to the router which ensures that the connections between adjacent cells are joined together properly. The output of the router is fed to an output generator. Currently, output can be produced in one of the following three formats:

1. a format that can be displayed on an ordinary text terminal and is suitable for a line printer,
2. a format for a bit-mapped screen using high-resolution graphics,
3. in LaTeX [6] format.

The floorplanner also includes facilities for drawing particular parts of a circuit and for producing layouts to a specified level of detail.

Expression transformer

μFP has many theorems for refining designs. The application of these theorems constitutes a sequence of transformations which can be used for documenting and justifying design decisions. The expression transformer is an experiment to discover the rudiments of providing mechanical support for this style of development. It interprets definitions and algebraic theorems as symmetric rewrite rules. These rules are currently embedded in the code of the transformer. At present the transformation of expressions is completely user-guided: the user selects a subexpression by a pattern-matching mechanism, and then specifies the rule to be applied. The system performs the transformation and reports the substitutions and the assumptions used.

AN EXAMPLE: CONVOLVER DESIGNS

We shall provide a flavour of using the tools by developing some simple designs for convolution. Given the data stream x, and the coefficient stream $[w_0 \ldots w_{N-1}]$, a

convolver computes the result stream y such that

$$\forall\, t.\; y_t = \sum_{0 \le n < N} x_{t-n} \times w_{n,t}.$$

We shall need adders, multipliers and latches, which are described respectively by the predefined primitives *Add*, *Mult* and $\mathcal{D}$.

Let us first show how the input signals, $w_{n,t}$ and x_t, can be generated. Of course one can only simulate or draw a circuit of a particular size; we choose $N = 4$. In response to the prompt "`>`" for a user command, we type

```
> sims [x,[w0,w1,w2,w3]]
```

The simulator treats a symbol as a constant function and produces a stream of time-stamped symbols laid out vertically:

```
   0: <x_0, <w0_0, w1_0, w2_0, w3_0>>
   1: <x_1, <w0_1, w1_1, w2_1, w3_1>>
   2: <x_2, <w0_2, w1_2, w2_2, w3_2>>

Output interrupted!
```

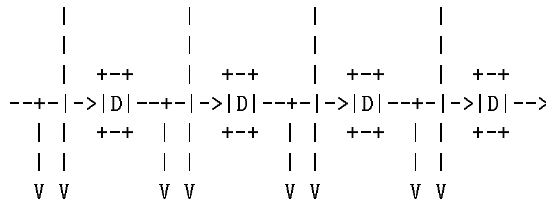
The infinite output stream can be interrupted at any time. Alternatively one can specify the number of cycles for the simulation. A constant stream of numbers can be produced in a similar manner. To save typing we define `ws = [w0,w1,w2,w3]`.

Now a straightforward way to implement convolution is to broadcast delayed versions of x to each w_n, forming N products which are then added together. Arbitrarily we choose to use a horizontal array of broadcast cells. Each cell, described by $[Id, El_1; \mathcal{D}]$, passes the pair $\langle x_{t'}, w_{i,t}\rangle$ downward unaltered and outputs $x_{t'-1}$ to the right. The rightmost x output is disposed of by sequentially composing $\backslash_H[Id, El_1; \mathcal{D}]$ with El_1.

```
> sims  [x,ws] ; \H [Id,El1;D] ; El1

   0: <<x_0, w0_0>, <?, w1_0>, <?, w2_0>, <?, w3_0>>
   1: <<x_1, w0_1>, <x_0, w1_1>, <?, w2_1>, <?, w3_1>>
   2: <<x_2, w0_2>, <x_1, w1_2>, <x_0, w2_2>, <?, w3_2>>
   3: <<x_3, w0_3>, <x_2, w1_3>, <x_1, w2_3>, <x_0, w3_3>>
   4: <<x_4, w0_4>, <x_3, w1_4>, <x_2, w2_4>, <x_1, w3_4>>

Output interrupted!
```

One can sketch this circuit using the floorplanner. The same input signals for the simulator, which define the size of the array, can be used. The expression placed to the left of "`>>`" will be excluded from the picture.

```
> draw [x,ws] >> \H [Id,El1;D]
```



A horizontal reduce of multiply-adders, $MAdds = /_H Mac$ where $Mac = \mathsf{snd} Mult; Add$, can then be used to form the result.

```
> sims [0,[[a,u],[b,v],[c,x],[d,y]]] ; MAdds

   0: ((((a_0 * u_0) + (b_0 * v_0)) + (c_0 * x_0)) + (d_0 * y_0))
   1: ((((a_1 * u_1) + (b_1 * v_1)) + (c_1 * x_1)) + (d_1 * y_1))

Output interrupted!

> draw [0,[[a,u],[b,v],[c,x],[d,y]]] >> MAdds

    ||       ||       ||       ||
    VV       VV       VV       VV
  +----+   +----+   +----+   +----+
  |Mult|   |Mult|   |Mult|   |Mult|
  +----+   +----+   +----+   +----+
    |        |        |        |
    V        V        V        V
  +---+    +---+    +---+    +---+
->|Add|--->|Add|--->|Add|--->|Add|--->
  +---+    +---+    +---+    +---+
```

To put the broadcast circuit and the multiply-adders together, we rewite $MAdds$ in the form of a horizontal array by applying the theorem called "`/H.\H`" in the expression transformer,

```
> select MAdds

MAdds = {/H Mac}

MAdds> apply /H.\H

MAdds = {\H (Mac ; El2^-1) ; El2}
using /H f -> \H (f;El2^-1);El2 with f = Mac.
```

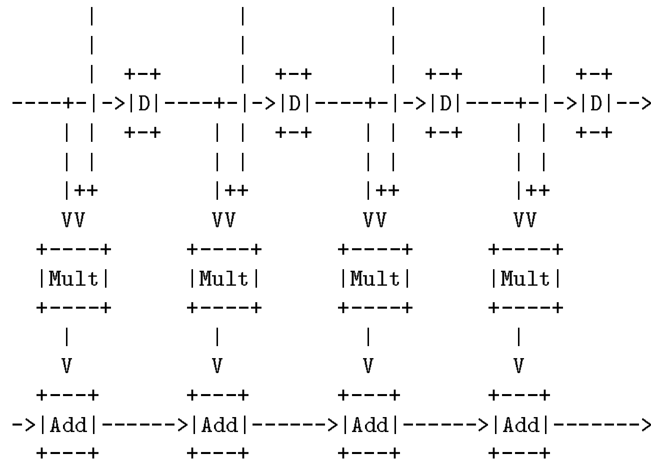
(the curly brackets indicate the expression transformed by the theorem). Our first convolver, $Cv1$, is formed by placing the broadcast circuit above $MAdds$. $Cv1$ can be simplified by combining the two horizontal arrays into one:

```
> select Cv1

Cv1 = {(\H [Id, El1 ; D]) \|/ (\H (Mac ; El2^-1))}

Cv1> apply \H.\|/

Cv1 = {\H ([Id, El1 ; D] \|/ (Mac ; El2^-1))}
using (\H f) \|/ (\H g) -> \H (f \|/ g)
with f = [Id, El1 ; D] and g = Mac ; El2^-1.
```

Hence one can define $Cv1 = \backslash_H CvCell1$ where $CvCell1 = [Id, El_1; \mathcal{D}] \downarrow (Mac; El_2{}^{-1})$.

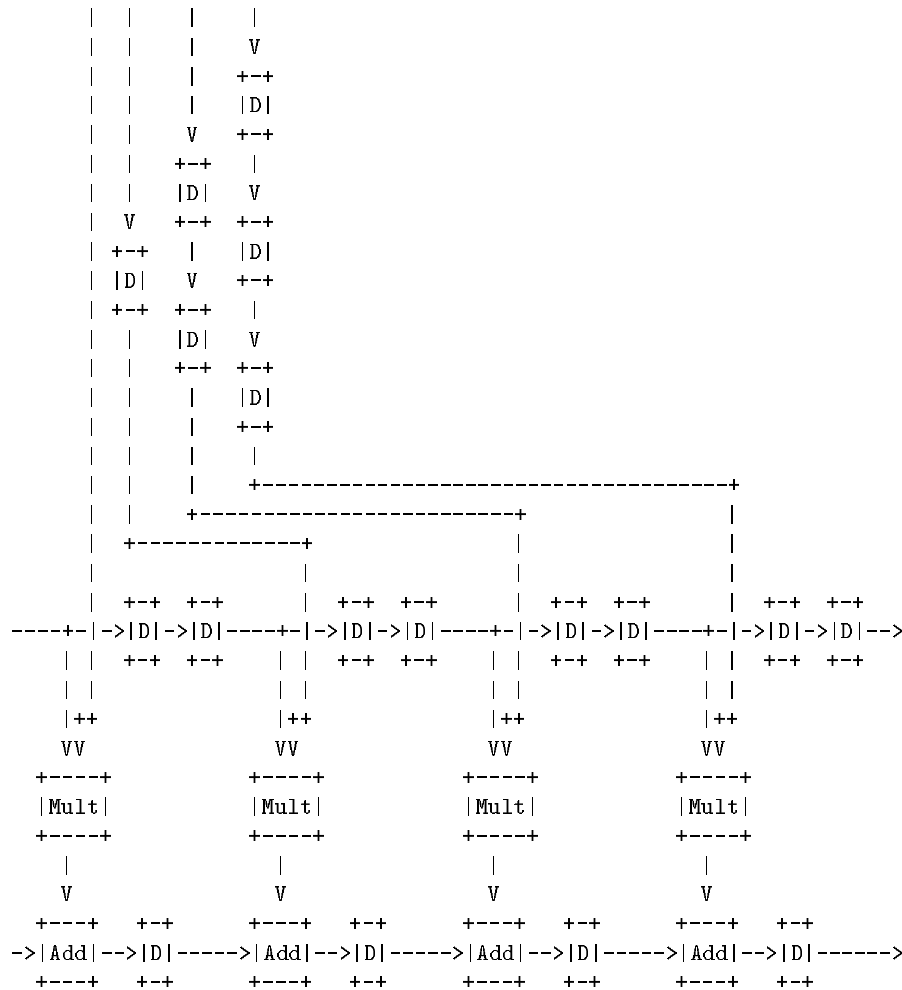
```
> sims [[0,x],ws] ; Cv1 ; El2 ; El1

   0: ((((x_0 * w0_0) + (? * w1_0)) + (? * w2_0)) + (? * w3_0))
   1: ((((x_1 * w0_1) + (x_0 * w1_1)) + (? * w2_1)) + (? * w3_1))
   2: ((((x_2 * w0_2) + (x_1 * w1_2)) + (x_0 * w2_2)) + (? * w3_2))
   3: ((((x_3 * w0_3) + (x_2 * w1_3)) + (x_1 * w2_3)) + (x_0 * w3_3))
   4: ((((x_4 * w0_4) + (x_3 * w1_4)) + (x_2 * w2_4)) + (x_1 * w3_4))

Output interrupted!
```

```
> draw [[0,x],ws] >> Cv1
```



To make the clock period independent of the array size, one can pipeline $Cv1$ to get $Cv2 = \backslash_H(CvCell1; \mathsf{snd} \propto \mathcal{D})$. The floorplanner can be used to sketch $Cv2$ and the associated skewing circuitry given by $\mathsf{snd}(\triangle_R \mathcal{D})$.

```
> draw [[0,x],ws] >> snd /\R D ; Cv2
```



```
> sim [[0,x],ws] ; snd /\R D ; Cv2 ; El2 ; El1

   0: ?
   1: ?
   2: ?
   3: ?
   4: ?
   5: ?
   6: ?
   7: ((((x_3 * w0_3) + (x_2 * w1_3)) + (x_1 * w2_3)) + (x_0 * w3_3))
   8: ((((x_4 * w0_4) + (x_3 * w1_4)) + (x_2 * w2_4)) + (x_1 * w3_4))
   9: ((((x_5 * w0_5) + (x_4 * w1_5)) + (x_3 * w2_5)) + (x_2 * w3_5))

Output interrupted!
```

Notice the use of `sim` rather than `sims` suppresses displaying the structure of outputs containing undefined objects. The simulation suggests that $Cv2$ is a delayed version of $Cv1$, that $\mathsf{snd}\triangle_R\mathcal{D}; Cv2; El_2; El_1 = Cv1; El_2; El_1; \mathcal{D}^N$. Equation (1) can be used to verify this observation.

The design can be carried down to bit-level by replacing the components and the signals by their bit-level counterparts. Furthermore, data-conversion functions can be used to map one form of data representation to another: for instance the function $[0, Id]; /_H(\mathsf{fst}([Id, 2]; Mult); Add)$ converts a stream of tuples of bits (most significant bit first) into a stream of the corresponding unsigned integers. This allows the specification and simulation of designs containing components described at different levels of abstraction.

EXPERIENCE AND EXTENSIONS

The tools have been exercised on various word-level and bit-level regular array designs, including convolvers, rank evaluators [9], multipliers [10], recursive filters [11], sorters [17], and motion estimators [4, 3]. One of the motion estimator circuits [3] is shortly due for fabrication. The tools have also been used for undergraduate and graduate teaching.

Although the current set of tools is intended for experimenting with different ways of interpreting and manipulating expressions in our notation rather than for use in a production environment, early designer experience [2] has been encouraging. It is confirmed that the simplicity and succinctness of our notation enable circuits to be described very elegantly and quickly with practice, and that the system facilitates the rapid exploration of choices in designing a circuit to perform a particular function. Similar benefits have been reported for other hardware development tools [13, 12] based on functional languages.

Our work has provided a basis for a design environment which enables the cost-effective production of regular array circuits. Current research is directed towards enhancing the completeness, effectiveness and robustness of our tools, and extending them to encompass relational [17] and heterogeneous [8] array descriptions. Other possible extensions include: improving the interface to designers and to other tools (such as conventional computer-aided design tools and theorem provers), and the development of a comprehensive library of cells and transformation strategies.

Acknowledgement. We are grateful to A. S. Bhandal, K. Page and K. Thapar for providing valuable feedbacks about our tools and documentations. This work had been undertaken as part of the UK Alvey Programme (Project ARCH 013) whose support is gratefully acknowledged. The first author also expresses his gratitude to the Croucher Foundation for its support.

References

[1] J. Backus, 'Can programming be liberated from the von Neumann style? A functional style and its algebra of programs'. *Commun. ACM*, vol. 21, no. 8, p. 613, 1978.

[2] A. S. Bhandal, 'Early feelings on muFP and the Orwell system'. Plessey Research Caswell Limited, 1987 (unpublished).

[3] A. S. Bhandal, V. Considine, and G. E. Dixon, 'An array processor for video picture motion estimation'. In *Proceedings of International conference on systolic arrays* (this volume), Killarney, 1989.

[4] A. S. Bhandal, D. J. Griffin, and V. Considine, 'Array architectures'. In *Proceedings of Alvey Conference*, Swansea, 1988.

[5] S. Y. Kung, *VLSI array processors.* Prentice-Hall, New Jersey, 1988.

[6] L. Lamport, LaTeX*: a document preparation system.* Addison-Wesley, 1986.

[7] C. E. Leiserson and J. B. Saxe, 'Optimizing synchronous systems'. *Journal of VLSI and Computer Systems*, vol. 1, no. 1, p. 41, 1983.

[8] W. Luk, 'Specifying and developing regular heterogeneous designs'. In preparation.

[9] W. Luk and G. Jones, 'The derivation of regular synchronous circuits'. In K. Bromley, S. Y. Kung, and E. Swartzlander, editors, *Proceedings of International conference on systolic arrays*, p. 305, San Diego, 1988.

[10] W. Luk and G. Jones, 'From specification to parametrised architectures'. In G. Milne, editor, *Proceedings of International conference on the fusion of hardware design and verification*, p. 263, Glasgow, 1988.

[11] W. Luk and G. Jones, 'Systolic recursive filters'. *IEEE Transactions on Circuits & Systems*, vol. 35, no. 8, p. 1067, 1988.

[12] J. T. O'Donnell, 'Hydra: hardware description in a functional language using recursion equations and higher order combining forms'. In G. Milne, editor, *Proceedings of International conference on the fusion of hardware design and verification*, p. 305, Glasgow, 1988.

[13] D. Patel, M. Schlag, and M. Ercegovac, '$\nu\mathcal{FP}$: an environment for the multi-level specification, analysis, and synthesis of hardware algorithms'. In J-P. Jouannaud, editor, *Functional programming languages and its applications*, p. 238. Springer-Verlag, 1985.

[14] P. Quinton, 'Automatic synthesis of systolic arrays from uniform recurrent equations'. In *Proceedings of the 11th Annual Symposium on computer architecture*, p. 208, 1984.

[15] M. Sheeran, *μFP – a language for VLSI design.* D.Phil. Thesis, Programming Research Group, Oxford University, November 1983.

[16] M. Sheeran, 'Designing regular array architectures using higher order functions'. In J-P. Jouannaud, editor, *Functional programming languages and its applications*, p. 220. Springer-Verlag, 1985.

[17] M. Sheeran and G. Jones, 'Relations + higher-order functions = hardware descriptions'. In *Proceedings of CompEuro*, p. 303, Hamburg, 1987.

[18] P. L. Wadler, 'An introduction to Orwell'. Programming Research Group, Oxford University, 1985.