# Field-Programmable Logic: Catalyst for New Computing Paradigms

Gordon Brebner

Department of Computer Science
University of Edinburgh
Mayfield Road
Edinburgh EH9 3JZ
Scotland

**Abstract.** This paper discusses the changes in computational viewpoint that have been, or are being, facilitated by the advent of Field-Programmable Logic. To quote the FPL'98 workshop aim, the paper is concerned with exploring the evolution 'from tinkertoy to parallel computing paradigm'. The central point is to discourage thinking in terms of just 'hardware' and 'software', with FPL being considered within the former category. This may have been appropriate for initial FPL applications, but is inappropriate when seeking to exploit its full potential. A summary of the revised viewpoint is that both control flow and data flow methods should be equally accessible to the algorithm designer, and also that flexibility in architectures should be made available as an aid to designing algorithms.

## 1 Introduction

The main catalytic effect of Field-Programmable Logic (FPL) is indicated in its very name: logic circuitry can be programmed, that is, it can be regarded as 'soft'. This fact is one driving force for a review of the traditional roles of things like hardware and software, and circuits and programs. There are other, related, driving forces, including the emergent fields of:

- hardware/software co-design;
- custom computing machines; and
- algorithms for configurable parallel computers.

So far, research into hardware/software co-design *per se* has mainly been a separate area from research into field-programmable logic, although many papers on the use of FPL include hardware-software partitioning as a feature of system design. In contrast, custom computing machines have been at the heart of FPL research activity, evidenced in particular by the highly-successful series of IEEE Symposia on FPGAs for Custom Computing Machines. Hartenstein *et al* present a very good discussion of the relationship between the fields of hardware/software co-design and custom computing machines in [7]. In tandem with these two very practically oriented areas, there has been much interest in configurable computers within the parallel algorithms community, in particular through the reconfigurable mesh model [1]. It is natural to ask what links there are between this area and the more practical areas, given that all are concerned with reconfigurability.

This paper presents a rather more general discussion of the new digital computing paradigms[1] that lie ahead as a result of these driving forces. Thus, the concern is with how computing systems implement required functions, rather than the equally-important question of how a user specifies functional requirements in the first place. However, consideration of the concerns addressed here is essential to thoughful consideration of the latter question.

The new paradigms offer an exciting vision to those who currently only see FPL as a mechanism for easy prototyping and cheap glue logic, or for enabling occasional field upgrades of computer system 'hardware'. The author believes that the key to this vision is to stop regarding FPL as being merely a functionally-equivalent alternative to ASICs (one helpful step in this direction is to stop using the suggestive terms 'hardware' and 'software').

The central discussion points of the paper follow in the next five sections. Section 2 discusses a fundamental paradigm change in system design: from a world where there is:

- higher-level algorithm/program/software design on one side; and
- lower-level architecture/circuit/hardware design on the other side

to a new world with:

- programs/circuits;
- algorithms/architectures; and
- softness/hardness

as three distinct design trade-offs. Section 3 examines this new world more closely, identifying that these three trade-offs can recur at different levels of abstraction within designs.

After the review of the new world, Section 4 focuses on programmability itself, examining how it can manifest itself within system components. In Section 5, there is a discussion of the different characteristics of computational circuits and interconnected processing elements — although both have a basic graph-style abstract structure, they have differing computation properties, and so it seems reasonable to distinguish between them. After this, Section 6 briefly examines the role of memories in the new computational world. Finally, Section 7 draws some conclusions, and points to the major research problems that lie ahead.

[1] There is no attempt here to address analogue computing paradigms, although most of the ideas related to digital circuitry will carry over to analogue circuitry.

## 2 The fundamental paradigm change

### Control flow v. data flow

A central feature of the traditional engineering of computer systems is that, at some point, there is a stage of partitioning of the system into hardware components and software components; even in the enlightened world of hardware/software co-design, there is, by definition, the notion of separate hardware and software components.

The fundamental paradigm shift suggested by field-programmable logic, and related developments in reconfigurable systems, is that there is a more natural basic choice, which is between:

- control flow approaches; and
- data flow approaches.

To date, consideration of these two approaches, and their relative merits and demerits, has largely been the domain of computer architecture and programming language specialists. However, now seems the time to place them in front of a larger audience. Here, control flow is typically exemplified by sequential program execution, where instructions cause data manipulation. At a low-level, this includes microprocessor and DSP instruction sequences, and at higher levels, programs in imperative and functional languages. Data flow is exemplified by interconnected processing elements with data moving between elements, not under some central programmed control. At a low level, this includes logic circuitry, and at higher levels, interconnected ALUs or even interconnected processors. Another interpretation of these approaches is control flow as 'computing in time' and data flow as 'computing in space'.

To a very large extent in the past, control flow approaches have been associated with the software designer, and data flow approaches with the hardware designer. This has been forced by the underpinning technology respectively available to these classes of designer. A main impact of FPL should be that a data flow approach can be made available to the software designer, as an alternative form of programming. One noteworthy existing example is the field of systolic arrays, which uses a data flow approach with constrained interconnection and timing constraints. Control flow approaches are already made available to hardware designers in many cases, principally through the use of state machines.

This general world of control flow v. data flow matches a world that is well-familiar to those specialising in complexity theory and the analysis of algorithms. There, there are control flow-style models, such as the Turing Machine and the Random Access Machine, and data flow-style models, such as boolean circuits and algebraic circuits. In this more abstract world, algorithm designers are often comfortable with working in terms of circuits, rather than programs, not feeling encumbered by extra baggage associated with the circuits having a hardware nature.

One further development of the control flow and data flow theme is to envisage algorithms that make use of both approaches in combination. This might be termed 'control flow/data flow co-design'. It is an effect already achieved at a high system level when hardware/software co-design is employed. Here, however, there is no general presumption about what is hard and what is soft.

### Algorithms v. architectures

As a consequence of focusing on the choice between control flow and data flow styles, a new interpretation can be placed on the traditional notions of 'algorithm' and 'architecture'. Currently, the prevalent situation is that algorithms are seen as a software (and control flow) matter and architectures are seen as a hardware (and data flow) matter, and so it is necessary to fit algorithms to fixed architectures.

In the new view of computing paradigms, a better interpretation of these notions is that an algorithm specifies a special-purpose computation, described in terms of an architecture which supplies a more general-purpose computational mechanism. This is a particular instance of a layered implementation, considered in more general in the next section. A key point however, is that the architecture need not be considered as a fixed and given entity. That is, an implementor can use algorithm/architecture co-design to develop an algorithm supported by an apt architecture. The architecture may supply a control flow model, a data flow model, or a combination of both if this is convenient for algorithms to use. FPL is a technology that, together with conventional processors, enables such co-design, in its most general form, to be used as a routine implementation method.

### Hardness v. softness

Given the above paradigm shift, the hardware/software issue now occupies a separate dimension to the control flow/data flow issue, and also a separate dimension to the algorithm/architecture issue. This dimension is concerned with the benefits of hardness:

- first, at some point, use of hardware is necessary to give a physical basis for computation;
- second, hardware may give better brute-force performance than software.

That is, the business of hardness v. softness is decoupled from the nature of the computational model being implemented. A processor core is an example of a control flow model implemented in hardware; an FPL device is an example of a data flow model implemented in hardware — in both cases, the hardware provides a physical basis that can run an effectively-infinite range of different algorithms. Processors are sufficiently mature and well-understood artifacts that it is possible to have instruction set architectures that are relatively stable over time. This is not the case for FPL devices at the moment, but the author is confident that more stable 'instruction set architectures' will emerge over time. Some initial thoughts on what a less device-dependent ISA might be like are contained in [4].

## 3 Layered implementation

The key feature of selecting between control flow and data flow approaches to computation is that choice need not occur at only one level of a system design. That is, a conventional approach of partitioning a system at the topmost level only is just one special case of a more general model. At any level of abstraction of the detail of a system, a specification of required system or sub-system behaviour might be implemented using either model or, indeed, a co-design using both.

This general idea offers a possible reconcilation for the theological debates in the configurable community that concern the best granularity for interconnected processing elements. At one level, a data flow model with chunky processing elements may be apt; however, either in implementing these elements or for other computation at the same level, simpler processing elements may be used. This flexibility essentially arises from convincing the designer that there is not an unbreakable commitment to hardware at any particular stage.

In order to implement a model at a higher level of abstraction in terms of a lower level, the notions of simulation and emulation are relevant. These two terms are used with different meanings by different people. To paraphrase the Oxford English Dictionary, simulation is the use of a computer model to imitate that conditions of a process; and, to quote the OED, emulation is 'the technique by which a computer or software system is enabled to execute programs written for a different type of computer, by means of special hardware or software'.

In the conventional FPL world, the most usual case of simulation is that a software program is being used to imitate the behaviour of a hardware circuit. Simulation methods also occur in the complexity theory and analysis of algorithms community — for example, results on how Turing Machines can simulate boolean circuits, and *vice versa*, are well-known and well-understood.

In this new setting, concerned with layered implementation, the notion of emulation is the more relevant. A control flow model might emulate a data flow model, and a data flow model might emulate a control flow model. For example, the first direction might be when a program for a processor is used as a simulator of programmed logic circuitry, and the second direction might be when programmed logic circuitry is used to implement a simple processor core (e.g., [5]).

Thus, the mapping between a higher-level model and a lower-level model may be orchestrated from either the higher level or the lower level. The first case corresponds to traditional compilation, and the second case corresponds to emulation. From the higher level, a control flow or data flow description of a component is converted into a form executable by a programmable lower-level implementation (for example, program compilation, automatic place and route for FPL). From the lower level, a programmable implementation is programmed so that it can perform emulation by interpreting a description of a higher-level component. The key point in both cases is that the mapping may involve crossovers between control flow and data flow in either direction. Further, in both cases, the higher level component might be considered as being the 'algorithm', and the lower level component the 'architecture'. However, note that the algorithm might, in turn, be implementing an architecture for a higher level.

## 4 Three-level programmability

Notice that the discussion in the previous section leant heavily on the notion of programmability of a lower-level component. Considering an overall system, then it is perfectly natural for its behaviour to be programmable, either directly by a human user or more subtly as an adaptive system reacting to its environment. It is only at the lowest level of committing to hardware (for example, for a block on a chip) that there is no underlying programmable component. With current developments, such as the Virtual Socket Interface (VSI) [8] for system-on-chip design, one might even argue that there is an element of programmability at the chip level.

In trying to understand the novelty that arises from the programmability of FPL, it is useful to identify three different levels of programmability that apply to control flow models (i.e., to programming). In order of increasing frequency over time, these are:

– different programs can be executed;
– executing programs can be modified;
– dynamic behaviour through choice in control flows.

Of these, the first and third are perfectly normal, but the second is generally viewed as very bad practice nowadays — mostly because understandability (and hence more obvious correctness) is deemed more important than any possible performance gains.

For data flow models (i.e., circuits), a similar three-level classification of programmability can be derived, and this reflects types of FPL research carried out and also continuing. In the same order as above, these are:

– different circuits can be executed;
– executing circuits can be modified;
– dynamic behaviour through choice in data flows.

Many examples of FPL applications use only the first level: a device can be used to execute different circuits, in sequence over (perhaps lengthy) time. Indeed, if this level of programability is not used, then it is questionable why FPL is being used in the first place.

It appears that there is a much stronger case for indulging in the second level of programmability in data flow models than there is in control flow models. This is due to the weaker nature of the third level compared to that in the control flow model, where there is considerable run-time flexibility in the order that data items are processed and the operations that are performed on them. Incorporating equivalent flexibility directly into a particular circuit will result in an extremely large circuit, with many components redundant at any particular instant in time. Dynamic modification of circuitry is an attractive alternative that allows efficiency in circuit size. The extreme end of the scale is when different circuits time share an execution medium, that is, there is dynamic replacement of circuitry, rather than mere modification.

Note that there is an alternative approach when dealing with circuits that may have an excessive size, but include temporal redundancy. This is to employ virtual circuitry (a.k.a. virtual hardware), where circuits may have large sizes, and an operating system is used to manage the problems transparently (e.g., [2,3]). However, such methods of disguising circuit modification during execution do not preclude other, finer-grain, applications. As an example, parameter passing is one way of incorporating dynamic behaviour into control flow programs. Modification of executing circuits can be used to achieve a similar effect in a data flow model, by 'folding' parameter values into the functionality of the circuitry. It is also possible to modify data flows by changing interconnections between components.

What is currently lacking is any systematic way for controlling the extent of in-execution circuit modification — one practical reason for this has been the fact that partially-reconfigurable FPL technologies have only recently emerged to motivate consideration of these problems in detail.

## 5 Distinguishing circuits and networks

The preceding discussion does not deal with the class of regular parallel architectures very well. At first sight, a parallel architecture with interconnected processing elements might seem a natural candidate that fits the data flow model. This is acceptable for something like a systolic array; however, it is not entirely apt for less-constrained cases where the processing elements are conducting more random-style programmed communication. This can be expressed loosely as a distinction between MIMD and SIMD styles, to use the archaic terminology of Flynn [6], but the author prefers to avoid this famous taxonomy, since it rather constrains thinking to the models that seemed feasible many years ago.

To deal with this tension, it seems tempting at first to try to abstract away any differences, either using the fact that both are essentially graph-based, or using a more abstract model of communicating processes. However, the author believes that it is worth making a distinction, when discussing architectures at least. The following are criteria for differentiating circuits and networks of interconnected elements:

- tighter coupling v. looser coupling;
- communication each time step v. more asynchronous communication;
- randomish (by design) topology v. more structured topology.

In time, a more thorough definition will be necessary. For now, note that the first points in each item capture the essence of data flow, whereas the second points in each item capture the essence of interconnected control flow elements. In terms of using these different classes, the first involves continous attention to communication, wheras the second involves communication as a backdrop to computation.

This distinction points to interconnection as an important third technique, in addition to the control flow and data flow techniques (not forgetting the interconnection also occurs implicitly within the data flow model). In regular parallel architectures, the interconnection is between control flow entities. More interestingly, interconnection is important in order to co-design systems from control flow and data flow components. For example, at a low level of abstraction, the interconnection of a processor core with an FPL array is a very important issue.

Like the other two components, interconnection can often be programmable itself, either by programming of the interconnection fabric or by packet routing switches. This makes additional fluidity available to the algorithm/architecture designer. One further extension, that fits within the overall general model, is the use of active interconnections, where data flow, or perhaps control flow, function is inserted within the interconnection.

## 6 The role of memory

As a final piece in the overall picture of the new paradigm for computing, the role of memory must be considered. This is a significant adjunct to control flow, data flow and interconnection.

First, memory is necessary as the means of recording state information in both the control flow and data flow models, a fact which is very familiar. In the control flow model, memory is used for things like processor registers, cache memory and main memory, which are essential as repositories for the data being manipulated during the control flow. In the data flow model, memory is used for things like input and output registers and, in sequential circuitry, for internal latches, flip-flops and registers. Pipeline registers and buffers may be used to affect the timing characteristics of synchronous data flow models.

Second, memory is necessary as the means of recording the programmed features in both control flow and data flow models. In principle, this use is a special-case of the first use, in that the memory contains data for a lower-level implementation of the programmable model. This is fairly self-evident for conventional stored-program computers, where the same memory is used to hold both programs and data. Paradoxically, however, the contents of the memory are being used at two different levels of implementation. The observation is less evident in most present-day FPL technologies, since these incorporate special-case configuration memory distributed across the FPL array, which is disjoint from memory used to store data processed by the array. Furthermore, the interfacing of these two types of memory is often significantly different. However, this does stress that two implementation levels are involved.

Third, memory may act as a subsititute for direct interconnection, effectively simulating an all-to-all, but sequential rather than parallel, interconnect. In the context of parallel computation, the relative merits of shared memory versus explicit inter-processor communication are much debated. Rather than rehearse these arguments here, there is one observation worth making. The use of memory for interconnection might be seen as offering a further reference point using the criteria for differentiating interconnected processing elements and the data flow model. Its properties are:

- loose coupling;
- very asynchronous communication;
- random topology

In other words, it is a variant on physical interconnection that enables random interconnections, with arbitrarily-large latencies, between processing elements.

Finally, memory may be a substitute for use of a control flow or data flow component, by facilitating computation through lookup tables if an algorithm designer deems this efficient. For a limited number of inputs, with limited ranges of values, this can be considered as the ultimate in programmed components. This use establishes memory as not only a necessary adjunct to the control flow and data flow models, and interconnection, but also as a first-order component in its own right.

## 7 Conclusions and major problems

This paper has introduced a revised view of the system design process, to reflect an age of configurability and programmability in components once regarded as fixed. The author firmly believes that such a global view is needed before tackling the huge problems associated with the practicalities of designing in such a framework.

These problems include the notations used for describing designs, the tools for processing these notations and the compilation/emulation used for implementing higher-level views in terms of lower-level views. The hope is that there can be a softening of barriers between the tool sets used for hardware design and for software design. Ultimately, the problems also include the design of future novel hardware, to act as the physical basis for the revised design process.

It is also the hope that algorithm designers might have a higher profile in the world of configurable computing, with the incentive of being able to contemplate both control flow/data flow co-design, and algorithm/architecture co-design to obtain solutions that have very good performance, but perhaps without having to delve into the mysteries of real hardware to achieve these benefits.

## Acknowledgement

The author thanks the participants in the February 1998 Dagstuhl-Seminar on Dynamically Reconfigurable Architectures, for the wide-ranging technical discussion that helped him to form a more general view of the subject.

## References


1. Ben-Asher, Peleg, Ramaswami and Schuster, "The Power of Reconfiguration", Journal of Parallel and Distributed Computing, **13**, 1991, pp.139–153.
2. Brebner, "The Swappable Logic Unit: a Paradigm for Virtual Hardware", Proc. 5th Annual IEEE Symposium on Custom Computing Machines, IEEE Computer Society Press 1997, pp.77–86.
3. Brebner, "Automatic Identification of Swappable Logic Units in XC6200 Circuitry", Proc. 7th International Workshop on Field-Programmable Logic and Applications, Springer LNCS 1304, 1997, pp.173–182.
4. Brebner, "Circlets: Circuits as Applets", Proc. 6th Annual IEEE Symposium on Custom Computing Machines, IEEE Computer Society Press, 1998.
5. Donlin, "Self Modifying Circuitry — A Platform for Tractible Virtual Circuitry", Proc. 8th International Workshop on Field-Programmable Logic and Applications, Springer LNCS, 1998.
6. Flynn, Some Computer Organisations and their Effectiveness", IEEE Trans. on Computers, **21**, 1972, pp.948–960.
7. Hartenstein, Becker and Kress, "Custom Computing Machines vs. Hardware/Software Co-Design: From a Globalized Point of View", Proc. 6th International Workshop on Field-Programmable Logic and Applications, Springer LNCS 1142, pp.65–76.
8. VSI Alliance Architecture Document, VSI Alliance, 1998.