Field-Programmable Custom Computing Machines – A Taxonomy –

Mihai SIMA[1,2], Stamatis VASSILIADIS[1], Sorin COTOFANA[1],
Jos T.J. van EIJNDHOVEN[2], and Kees VISSERS[3]

[1] Delft University of Technology, Department of Electrical Engineering,
PO Box 5031, Mekelweg 4, 2600 GA Delft, The Netherlands,
{M.Sima,S.Vassiliadis,S.Cotofana}@et.tudelft.nl
[2] Philips Research Laboratories, Department of Information and Software Technology,
Box WDCp-045, Professor Holstlaan 4, 5656 AA Eindhoven, The Netherlands,
jos.van.eijndhoven@philips.com
[3] TriMedia Technologies, Inc., 1840 McCarthy Boulevard, Sunnyvale, CA 95035, U.S.A.,
kees.vissers@trimedia.com

Abstract. The ability for providing a hardware platform which can be customized on a per-application basis under software control has established *Reconfigurable Computing* ($\mathcal{RC}$) as a new computing paradigm. A machine employing the $\mathcal{RC}$ paradigm is referred to as a *Field-Programmable Custom Computing Machine* (FCCM). So far, the FCCMs have been classified according to implementation criteria. For the previous classifications do not reveal the entire meaning of the $\mathcal{RC}$ paradigm, we propose to classify the FCCMs according to architectural criteria. To analyze the phenomena inside FCCMs, we introduce a formalism based on microcode, in which any custom operation performed by a field-programmed computing facility is executed as a microprogram with two basic stages: `SET CONFIGURATION` and `EXECUTE CUSTOM OPERATION`. Based on the `SET/EXECUTE` formalism, we then propose an architectural-based taxonomy of FCCMs.

1 Introduction

The ability of providing a hardware platform which can be transformed under software control has established *Reconfigurable Computing* ($\mathcal{RC}$) [28], [42], [29] as a new computing paradigm in the last ten years. According to this paradigm, the main idea in improving the performance of a computing machine is to define custom computing resources on a per-application basis, and to dynamically configure them onto a *Field-Programmable Gate Array* (FPGA) [11]. As a general view, a computing machine working under the new $\mathcal{RC}$ paradigm typically includes a *General-Purpose Processor* (GPP) which is augmented with an FPGA. The basic idea is to exploit both the GPP flexibility to achieve medium performance for a large class of applications, and FPGA capability to implement application-specific computations. Such a hybrid is referred to as a *Field-Programmable Custom Computing Machine* (FCCM) [7], [16].

Various FCCMs have been proposed in the last decade. Former attempts in classifying FCCMs used implementation criteria [15], [31], [26], [47], [23], [38]. As the user observes only the architecture of a computing machine [4], the previous classifications do not seize well the implications of the new $\mathcal{RC}$ paradigm as perceived by the user. For this reason, we propose to classify the FCCMs according to architectural criteria. In order to analyze the phenomena inside FCCMs, yet without reference to a particular instruction set, we introduce a formalism based on microcode, in which any task (operation) to be performed by a field-programmable computing facility models its execution pattern on that of a microprogrammed sequence with two basic stages: `SET CONFIGURATION`, and `EXECUTE CUSTOM OPERATION`. The net effect of this approach is to allow a view on an FCCM at the level defined by the reference of the user, i.e., the architectural level, decoupled from lower implementation and realization hierarchical levels. The reader may note the similarity between the preceding formalism and the *requestor/server* formalism of Flynn [13]. Based on the `SET/EXECUTE` formalism, we propose an architectural-based taxonomy of FCCMs.

The paper is organized as follows. For background purpose, we present the most important issues related to microcode in Section 2, and the basic concepts concerning SRAM-based FPGAs in Section 3. Section 4 introduces a formalism by which the FCCM architectures can be analyzed from the microcode point of view, and Section 5 presents the architectural-based taxonomy of FCCMs. Section 6 concludes the paper.

2 The Microcode Concept

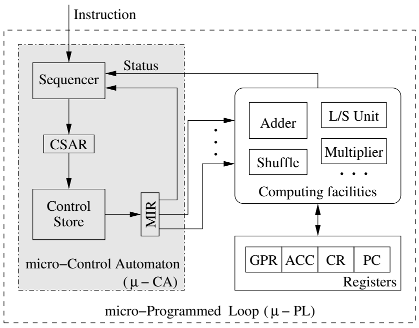

Fig. 1. The basic microprogrammed computer

Figure 1 depicts the organization of a microprogrammed computer as it is described in [32]. In the figure, the following acronyms were used: GPR – General Purpose Registers, ACC – Accumulator, CR – Control Registers, and PC – Program Counter. For such a computer, a microprogram in *Control Store* (CS) is associated with each incoming instruction. This microprogram is to be executed under the control of the *Sequencer*, as follows:

1. The sequencer maps the incoming instruction code into a control store address, and stores this address into the *Control Store Address Register* (CSAR).
2. The microinstruction addressed by CSAR is read from CS into the *MicroInstruction Register* (MIR).
3. The microoperations specified by the microinstruction in MIR are decoded, and the control signals are subsequently generated.
4. The computing resources perform the computation according to control signals.
5. The sequencer uses status information generated by the computing facilities and some information originating from MIR to prepare the address of the next microinstruction. This address is then stored into CSAR.
6. If an *end-of-operation* microinstruction is detected, a jump is executed to a instruction fetch microsubroutine. At the end of this microsubroutine, the new incoming instruction initiates a new cycle of the microprogrammed loop.

The microinstructions may be classified by the number of controlled resources. Given a hardware implementation which provides a number of computing resources (facilities), the amount of explicitly controlled resources during the same time unit (cycle) determines the verticality or horizontality of the microcode as follows:

- **A microinstruction which controls multiple resources in one cycle is *horizontal*.**
- **A microinstruction which controls a single resource is *vertical*.**

Let us assume we have a *Computing Machine* (CM) and its instruction set. An *implementation* of the CM can be formalized by means of the doublet:

$$\mathrm{CM} = \{\mu P \,,\, \mathcal{R}\} \tag{1}$$

where μP is the microprogram which includes all the microroutines for implementing the instruction set, and $\mathcal{R}$ is the set of N *computing (micro-)resources* or *facilities* which are controlled by the microinstructions in the microprogram:

$$\mathcal{R} = \{r_1 \,,\, r_2 \,,\, \ldots \,,\, r_N\} \tag{2}$$

Let us assume the computing resources are hardwired. If the microcode[4] is exposed to the *user*, i.e., the instruction set is composed of microinstructions, there is no way to adapt the architecture to application but by custom-redesigning the computing facilities set, $\mathcal{R}$. When the microcode is not exposed to the *user*, i.e., a microroutine is associated with each instruction, then the architecture can be adapted by rewriting the microprogram μP.

Since the architecture of the vertical microinstructions associated with hardwired computing facilities is fixed, the adaptation procedure by rewriting the microprogram has a limited efficiency: a new instruction is created by threading the operations of fixed (i.e., inflexible) computing facilities rather than generating a full-custom one.

If the resources themselves are microcoded, the formalism recursively propagates to lower levels. Therefore, the implementation of each resource can be viewed as a doublet composed of a *nanoprogram* (nP) and a *nano-resource set* ($n\mathcal{R}$):

$$r_i = \{nP \,,\, n\mathcal{R}\}\,, \quad i = 1, 2, \ldots, N \tag{3}$$

Now it is the rewriting of the nanocode which is limited by the fixed set of nano-resources.

The presence of the reconfigurable hardware opens up new ways to adapt the architecture. Assuming the resources are implemented on a programmable array, adapting the resources to the application is entire flexible and can be performed on-line. In this situation, the resource set $\mathcal{R}$ metamorphoses into a new one, $\mathcal{R}^*$:

$$\mathcal{R} \longrightarrow \mathcal{R}^* = \{r_1^* \,,\, r_2^* \,,\, \ldots \,,\, r_M^*\}, \tag{4}$$

and so does the set of associated vertical microinstructions. It is obvious that writing new microprograms with application-tuned microinstructions is more effective than with fixed microinstructions.

[4] In this presentation, by *microcode* we will refer to both microinstructions and microprogram. The meaning of the microcode will become obvious from the context.

At this point, we want to stress out that the microcode is a *recursive formalism*. The micro and nano prefixes should be used against an *implementation reference level*[5] (IRL). Once such a level is set, the operations performed at this level are specified by *instructions*, and are under the explicit control of the *user*. Therefore, the operations below this level are specified by *microinstructions*, those on the subsequent level are specified by *nanoinstructions*, and so on.

3 FPGA Terminology and Concept

A device which can be configured *in the field* by the end user is usually referred to as a *Field-Programmable Device* (FPD) [11], [19], [5]. Generally speaking, the constituents of an FPD are *Raw Hardware* and *Configuration Memory*. The function performed by the raw hardware is defined by the information stored into the configuration memory.

The FPD architectures can be classified in two major classes: *Programmable Logic Devices* (PLD) and *Field-Programmable Gate Arrays* (FPGA). Details on each class can be found for example in [6]. Although both PLD and FPGA devices can be used to implement digital logic circuits, we will pre-eminently above all use the term of *FPGA* hereafter to refer to a programmable device. The higher logic capacity of FPGAs and the attempts to augment FPGAs with PLD-like programmable logic in order to make use of both FPGA and PLD characteristics, support our choice for this terminology.

Some FPGAs can be configured only once, e.g., by burning fuses. Other FPGAs can be reconfigured any number of times, since their configuration is stored in SRAM. Initially considered as a weakness due to the volatility of configuration data, the re-programming capabilities of SRAM-based FPGAs led to the new $\mathcal{RC}$ paradigm. By reconfiguring the FPGA under software control, application-specific computing facilities can be implemented on-the-fly.

A discussion on choosing the appropriate FPGA architecture is beyond the goal of this paper. More information concerning this problem can be found for example in [19].

4 FPGA to Microcode Mapping

In this section, we will introduce a formalism by which an FCCM architecture can be analyzed from the microcode point of view. This formalism originates in the observation that every custom instruction of an FCCM can be mapped into a microprogram.

As we already mentioned, by making use of the FPGA capability to change its functionality in pursuance of a reconfiguring process, adapting both the functionality of *computing facilities* and *microprogram in the control store* to the application characteristics becomes possible with the new $\mathcal{RC}$ paradigm. For the information stored in FPGA's configuration memory determines the functionality of the raw hardware, the dynamic implementation of an instruction on FPGA can be formalized by means of a microcoded structure. Assuming the FPGA configuration memory is written under the

[5] If it will not be specified explicitly, the IRL will be considered as being the level defined by the instruction set. For example, although the microcode is exposed to the *user* in the RISC machines, the RISC operations are specified by *instructions*, rather than by microinstructions.

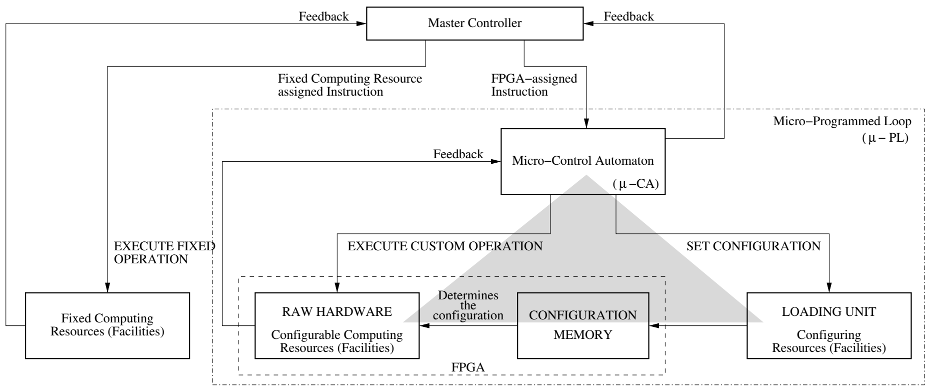

Fig. 2. The microcode concept applied to FCCMs. The Δ arrangement.

control of a *Loading Unit*, the control automaton, the FPGA, and the loading unit may have a Δ arrangement, as depicted in Figure 2. The circuits configured on the raw hardware and the loading unit(s) are all regarded as controlled resources in the proposed formalism. Each of the previously mentioned resources is given a special class of microinstructions: `SET` for the loading unit, which initiates the reconfiguration of the raw hardware, and `EXECUTE` for the circuits configured on raw hardware, which launches the custom operations.

In this way, any custom operation of an FCCM can be executed in a reconfigurable manner, in which the execution pattern models on that of a microprogrammed sequence with two basic stages: `SET CONFIGURATION`, and `EXECUTE CUSTOM OPERATION`. It is the `SET/EXECUTE` formalism we will use in building the taxonomy of FCCMs.

It is worth to specify that only `EXECUTE FIXED OPERATION` microinstructions can be associated with fixed computing facilities, because such facilities cannot be reconfigured. Also, assuming that a multiple-context FPGA [11] is used, activating an idle context is performed by an `ACTIVATE CONFIGURATION` microinstruction, which is actually a flavor of the `SET CONFIGURATION` microinstruction. In the sequel, we will refer to all loading unit(s) and resource(s) for activating the idle context as *Configuring Resources (Facilities)*.

Since an FCCM includes both computing and configuring facilities, the statement regarding the verticality or horizontality of the microcode as defined in Section 2 needs to be adjusted, as follows:

Definition 1. *For an FCCM hardware implementation which provides a number of* computing and configuring facilities*, the amount of explicitly controlled* computing and/or configuring facilities *during the same time unit (cycle) determines the verticality or horizontality of the microcode.*

Therefore, any of the `SET CONFIGURATION`, `EXECUTE CUSTOM OPERATION`, and `EXECUTE FIXED OPERATION` microinstructions can be either vertical or horizontal, and may participate in a horizontal microinstruction.

Let us set the *implementation reference level* as being the level of instructions in Figure 2. In the particular case when the microcode is not exposed to the upper level, an explicit `SET` instruction is not available to the *user*. Consequently, the system performs by itself the management of the active configuration, i.e., without an explicit control provided by *user*. In this case, the *user* "sees" only the FPGA-assigned instruction which can be regarded as an `EXECUTE CUSTOM OPERATION` microinstruction visible to the instruction level. Here, we would like to note that the `EXECUTE FIXED OPERATION` microinstruction is always visible to the *user*. Conversely, when the microcode is exposed to the upper level, an explicit `SET` instruction is available, and the management of the active configuration becomes the responsibility of the *user*.

5 A Proposed Taxonomy of FCCMs

Before introducing our taxonomy, we would like to overview the previous work in FCCM classification.

In [15] two parameters for classifying FCCMs are used: *Reconfigurable Processing Unit (RPU) size* (*small* or *large*) and *availability of RPU-dedicated local memory*. Consequently, FCCMs are divided into four classes. Since what exactly means *small* and what exactly means *large* is subject to the complexity of the algorithms being implemented, the differences between classes are rather fuzzy. Also, providing dedicated RPU memory is an issue which belongs to *implementation level* of a machine; consequently, the implications to the *architectural level*, if any, are not clear.

The *Processing Element (PE) granularity*, *RPU integration level* with a host processor, and the *reconfigurability of the external interconnection network* are used as classification criteria in [31]. According to the first criterion, the FCCMs are classified as *fine-*, *medium-*, and *coarse-grain* systems. The second criterion divides the machines into *dynamic* systems that are not controlled by external devices, *closely-coupled static* systems in which the RPUs are coupled on the processor's datapath, and *loosely-coupled static* systems that have RPUs attached to the host as coprocessors. According to the last criterion, the FCCMs have a *reconfigurable* or *fixed* interconnection network.

In order to classify the FCCMs, the *loosely coupling* versus *tightly coupling* criterion is used by other members of the FCCM community, e.g., [26], [47], [23], [38]. In the loosely coupling embodiment, the RPU is connected via a bus to, and operates asynchronously with the host processor. In the tightly coupling embodiment, the RPU is used as a *functional unit*.

We emphasize that all these taxonomies are build using *implementation* criteria. As the user observes only the architecture of a computing machine, classifying the FCCMs according to architectural criteria is more appropriate. Since FCCMs are microcoded machines, we propose to classify the FCCMs according to the following criteria:

- The verticality/horizontality of the microcode.
- The explicit availability of a `SET` instruction.

While the first criterion is a direct consequence of the proposed formalism, several comments regarding the second criterion are worth to be provided. An user-exposed `SET` instruction allows the reconfiguration management to be done explicitly in software, thus being subject to deep optimization. The drawback is that a more complex compiler is needed for scheduling the SET instruction at a proper location in time. Conversely, if SET is not exposed to the user, such management will be done in hardware. This time, the compiler is simpler, but at the expense of a higher reconfiguration penalty. With a hardware-based management, the code compatibility between FCCMs with different FPGA size and reconfiguration pattern can be preserved. Since the user has no concern about the reconfiguration, the configuration management is an implementation issue, much like the cache management in a conventional processor is.

In order to describe the classification process, several classification examples will be provided subsequently. We have to mentioned that, for each system, the IRL has been chosen such that as much FCCM-specific information as possible is revealed.

PRISC [33] is a RISC processor augmented with Programmable Functional Unit (PFU). Custom instructions can be implemented on the PFU. The specification of such instruction is done by means of a preamble to the RISC instruction format. When a custom instruction is called, the hardware is responsible for updating the PFU configuration: if a reconfiguration is needed, an exception which stalls the processor is raised, and a long latency reconfiguration process is initiated. Since the reconfiguration is not under the direct control of the user, a dedicated instruction for reconfiguration, i.e., SET CONFIGURATION, is not exposed to the user. Only one fixed or programmable functional unit is explicitly controlled per cycle; therefore, the microcode is *vertical*.

The *PipeRench* coprocessor [8] consists of a set of identical physical *Stripes* which can be configured under the supervision of a *Configuration Controller* at run-time. PipeRench also includes a *Configuration Memory* which stores virtual stripe configurations. A single physical stripe can be configured per cycle; therefore, the reconfiguration of a stripe takes place concurrently with execution of the other stripes. Pipelines of arbitrary length can be implemented on PipeRench. A program for this device is a chained list of configuration words, each of which includes three fields: configuration bits for each virtual pipeline stage of the application, a *next-address* field which points to the next virtual stripe, and a set of flags for the configuration controller and four *Data Controllers*. Therefore, the configuration word is a horizontal instruction. Since the configuration controller handles the multiplexing of the application's stripes onto the physical fabric, the scheduling of the stripes, and the management of the on-chip configuration memory, while the user has only to provide the chained list of the configuration words, we can conclude that there is no user-exposed SET instruction.

The (re)configuration of the Nano-Processor [46] or RaPiD [10] is initiated by a master unit at application load-time. Each system may be used, at least theoretically, in a multi-tasking environment, in which the applications are switched on or idle. Since no further details whether the user can or cannot manage the reconfiguration are given, we classify such systems as *not obvious information about an explicit* SET *instruction*.

The Colt/Wormhole FPGA [3] is an array of Reconfigurable Processing Units interconnected through a mesh network. Multiple independent streams can be injected into the fabric. Each stream contains information needed to route the stream through the fabric and to configure all RFUs along the path, as well as data to be processed. In this way, the streams are self-steering, and can simultaneously configure the fabric and initiate the computation. Therefore, SET is explicit and the microcode is horizontal.

Following the above mentioned methodology, the most well known FCCMs can be classified as follows:

1. Vertical microcoded FCCMs
   (a) With explicit `SET` instruction: PRISM [2], PRISM-II/RASC [43], [44], RISA′ [39], RISA″ [39], MIPS + REMARC [30], MIPS + Garp [21], OneChip-98″ [23], URISC [12], Gilson's FCCM [14], Xputer/rALU [17], Molen vertically-coded processor [41], MorphoSys system [38].
   (b) Without explicit `SET` instruction: PRISC [33], OneChip [47], ConCISe [25], OneChip-98′ [23], DISC [45], Multiple-RISA [40], Chimaera [20].
   (c) Not obvious information about an explicit `SET` instruction: Virtual Computer [9], [46], Functional Memory [27], CCSimP (load-time reconfiguration) [35], NAPA [34].
2. Horizontal microcoded FCCMs
   (a) With explicit `SET` instruction: CoMPARE [36], Alippi's VLIW [1], RISA‴ [39], VEGA [24], Colt/Wormhole FPGA [3], rDPA [18], FPGA-augmented TriMedia/CPU64 [37], Molen horizontally-coded processor [41].
   (b) Without explicit `SET` instruction: PipeRench [8].
   (c) Not obvious information about an explicit `SET` instruction: Spyder [22], RaPiD (load-time reconfiguration) [10].

We would like to mention that applying the classification criteria on OneChip-98 machine introduced in [23], we determined that an explicit `SET` instruction was not provided to the user in one embodiment of OneChip-98, while such an instruction was provided to the user in another embodiment. It seems that two architectures were claimed in the same paper. We referred to them as OneChip-98′ and OneChip-98″. The same ambiguous way to propose multiple architectures under the same name is employed in [39]. For the Reconfigurable Instruction Set Accelerator (RISA), our taxonomy provides three entries (RISA′, RISA″, RISA‴).

6 Conclusions

We proposed a classification of the FCCMs according to architectural criteria. Two classification criteria were extracted from a formalism based on microcode. In terms of the first criterion, the FCCMs were classified in vertical or horizontal microcoded machines. In terms of the second criterion, the FCCMs were classified in machines with or without an explicit `SET` instruction. The taxonomy we proposed is architectural consistent, and can be easily extended to embed other criteria.

References

1. C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami. A DAG-Based Design Approach for Reconfigurable VLIW Processors. In *IEEE Design and Test Conference in Europe*, Munich, Germany, 1999.
2. P. Athanas and H. Silverman. Processor Reconfiguration through Instruction-Set Metamorphosis. *IEEE Computer*, 26(3):11-18, 1993.

3. R. Bittner, Jr. and P. Athanas. Wormhole Run-time Reconfiguration. In *Proc. 5th ACM/SIGDA Intl. Symp. on FPGAs*, pp. 79-85, Monterey, California, 1997.
4. G.A. Blaauw and F.P. Brooks, Jr. *Computer Architecture. Concepts and Evolution*. Addison-Wesley, Reading, Massachusetts, 1997.
5. G. Brebner. Field-Programmable Logic: Catalyst for New Computing Paradigms. In *Proc. 8th Intl. Workshop FPL'98*, pp. 49-58, Tallin, Estonia, 1998.
6. S. Brown and J. Rose. Architecture of FPGAs and CPLDs: A Tutorial. *IEEE Transactions on Design and Test of Computers*, 13(2):42-57, 1996.
7. D.A. Buell and K.L. Pocek. Custom Computing Machines: An Introduction. *Journal of Supercomputing*, 9(3):219-230, 1995.
8. S. Cadambi et al. Managing Pipeline-Reconfigurable FPGAs. In *Proc. 6th ACM/SIGDA Intl. Symp. on FPGAs*, pp. 55-64, Monterey, California, 1998.
9. S. Casselman. Virtual Computing and the Virtual Computer. In *Proc. IEEE Workshop on FCCMs*, pp. 43-48, Napa Valley, California, 1993.
10. D. Cronquist et al. Architecture Design of Reconfigurable Pipelined Datapaths. *Advanced Research in VLSI*, pp. 23-40, 1999.
11. A. DeHon. Reconfigurable Architectures for General-Purpose Computing. A. I. 1586, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1996.
12. A. Donlin. Self Modifying Circuitry - A Platform for Tractable Virtual Circuitry. In *Proc. 8th Intl. Workshop FPL'98*, pp. 199-208, Tallin, Estonia, 1998.
13. M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948-960, September 1972.
14. K.L. Gilson. Integrated Circuit Computing Device Comprising a Dynamically Configurable Gate Array Having a Microprocessor and Reconfigurable Instruction Execution Means and Method Therefor. U.S. Patent No. 5,361,373, 1994.
15. S. Guccione and M. Gonzales. Classification and Performance of Reconfigurable Architectures. In *Proc. 5th Intl. Workshop FPL'95*, pp. 439-448, Oxford, United Kingdom, 1995.
16. R. Hartenstein, Becker, and R. Kress. Custom Computing Machines versus Hardware/Software Co-Design: From a Globalized Point of View. In *Proc. 6th Intl. Workshop FPL'96*, pp. 65-76, Darmstadt, Germany, 1996.
17. R. Hartenstein et al. A Novel Paradigm of Parallel Computation and its Use to Implement Simple High-Performance Hardware. *Future Generation Computer Systems*, (7):181-198, 1991/1992.
18. R. Hartenstein, R. Kress, and H. Reinig. A New FPGA Architecture for Word-Oriented Datapaths. In *Proc. 4th Intl. Workshop FPL'94*, pp. 144-155, Prague, Czech Republic, 1994.
19. S. Hauck. The Roles of FPGA's in Reprogrammable Systems. *Proc. of the IEEE*, 86(4):615-638, 1998.
20. S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao. The Chimaera Reconfigurable Functional Unit. In *Proc. 5th IEEE Symp. on FCCMs*, pp. 87-96, Napa Valley, California, 1997.
21. J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proc. 5th IEEE Symp. on FCCMs*, pp. 12-21, Napa Valley, California, 1997.
22. C. Iseli and E. Sanchez. A Superscalar and Reconfigurable Processor. In *Proc. 4th Intl. Workshop FPL'94*, pp. 168-174, Prague, Czech Republic, 1994.
23. J. Jacob and P. Chow. Memory Interfacing and Instruction Specification for Reconfigurable Processors. In *Proc. 7th ACM/SIGDA Intl. Symp. on FPGAs*, pp. 145-154, Monterey, California, 1999.
24. D. Jones and D. Lewis. A Time-Multiplexed FPGA Architecture for Logic Emulation. In *Proc. IEEE CICC'95*, pp. 487-494, Santa Clara, California, 1995.
25. B. Kastrup, A. Bink, and J. Hoogerbrugge. ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator. In *Proc. 7th IEEE Symp. on FCCMs*, pp. 92-100, Napa Valley, California, 1999.

26. B. Kastrup, J. van Meerbergen, and K. Nowak. Seeking (the right) Problems for the Solutions of Reconfigurable Computing. In *Proc. 9th Intl. Workshop FPL'99*, pp. 520-525, Glasgow, Scotland, 1999.
27. A. Lew and R. Halverson, Jr. A FCCM for Dataflow (Spreadsheet) Programs. In *Proc. 3rd IEEE Symp. on FCCMs*, pp. 2-10, Napa Valley, California, 1995.
28. W. Mangione-Smith and B. Hutchings. Reconfigurable Architectures: The Road Ahead. In *Reconfigurable Architectures Workshop*, pp. 81-96, Geneva, Switzerland, 1997.
29. W. Mangione-Smith et al. Seeking Solutions in Configurable Computing. *IEEE Computer*, 30(12):38-43, 1997.
30. T. Miyamori and K. Olukotun. A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications. In *Proc. 6th IEEE Symp. on FCCMs*, pp. 2-11, Napa Valley, California, 1998.
31. B. Radunović and V. Milutinović. A Survey of Reconfigurable Computing Architectures. In *Proc. 8th Intl. Workshop FPL'98*, pp. 376-385, Tallin, Estonia, 1998.
32. T. Rauscher and P. Adams. Microprogramming: A Tutorial and Survey of Recent Developments. *IEEE Transactions on Computers*, C-29(1):2-20, 1980.
33. R. Razdan and M. Smith. A High Performance Microarchitecture with Hardware-Programmable Functional Units. In *Proc. 27th Annual Intl. Symp. on Microarchitecture*, pp. 172-180, San Jose, California, 1994.
34. C. Rupp et al. The NAPA Adaptive Processing Architecture. In *Proc. 6th IEEE Symp. on FCCMs*, pp. 28-37, Napa Valley, California, 1998.
35. Z. Salcic and B. Maunder. CCSimP - An Instruction-Level Custom-Configurable Processor for FPLDs. In *Proc. 6th Intl. Workshop FPL'96*, pp. 280-289, Darmstadt, Germany, 1996.
36. S. Sawitzki, A. Gratz, and R.G. Spallek. Increasing Microprocessor Performance with Tightly-Coupled Reconfigurable Logic Arrays. In *Proc. 8th Intl. Workshop FPL'98*, pp. 411-415, Tallin, Estonia, 1998.
37. M. Sima, S. Cotofana, J.T. van Eijndhoven, S. Vassiliadis, and K. Vissers. 8×8 IDCT Implementation on an FPGA-augmented TriMedia. In *Proc. 9th IEEE Symp. on FCCMs*, Rohnert Park, California, 2001.
38. H. Singh et al. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Application. *IEEE Tran. on Computers*, 49(5):465-481, 2000.
39. S. Trimberger. Reprogrammable Instruction Set Accelerator. U.S.Patent No. 5,737,631, 1998.
40. S. Trimberger. Reprogrammable Instruction Set Accelerator Using a Plurality of Programmable Execution Units and an Instruction Page Table. U.S. Patent No. 5,748,979, 1998.
41. S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN $\rho\mu$-coded Processor. In *Proc. 11th Intl. Conference FPL-2001*, pp. 275-285, Belfast, N. Ireland, U.K., 2001.
42. J. Villasenor and W. Mangione-Smith. Configurable Computing. *Scientific American*, pp. 55-59, 1997.
43. M. Wazlowski et al. PRISM-II Compiler and Architecture. In *Proc. IEEE Workshop on FCCMs*, pp. 9-16, Napa Valley, California, 1993.
44. M. Wazlowski. *A Reconfigurable Architecture Superscalar Coprocesor*. PhD thesis, Brown University, Providence, Rhode Island, 1996.
45. M. Wirthlin and B. Hutchings. A Dynamic Instruction Set Computer. In *Proc. 3rd IEEE Symp. on FCCMs*, pp. 99-109, Napa Valley, California, 1995.
46. M. Wirthlin, B.L. Hutchings, and K.L. Gilson. The Nano Processor: A Low Resource Reconfigurable Processor. In *Proc. 2nd IEEE Workshop on FCCMs*, pp. 23-30, Napa Valley, California, 1994.
47. R. Wittig and P. Chow. OneChip: An FPGA Processor With Reconfigurable Logic. In *Proc. 4th IEEE Symp. on FCCMs*, pp. 126-135, Napa Valley, California, 1996.