x , x (x), 1–27 (2004)

Future Directions of (Programmable and Reconfigurable) Embedded Processors

Stephan Wong, Stamatis Vassiliadis, Sorin Cotofana

Computer Engineering Laboratory,
Electrical Engineering Department,
Delft University of Technology,
{Stephan, Stamatis, Sorin}@CE.ET.TUDelft.NL

Abstract

The advent of microprocessors in embedded systems has significantly contributed to the wide-spread utilization of embedded systems in our daily lives. Microprocessors can be found in devices ranging from simple controllers in power plants to sophisticated multimedia set-top boxes in our homes. This is due to the fact that microprocessors, called embedded processors in this setting, are able to perform huge amounts of data processing required by embedded systems. In addition and equally important, embedded processors are able to achieve this at affordable prices. In the last decade, we have been witnessing several changes in the embedded pro-

cessors design fueled by two conflicting trends. First, the industry is dealing with cut-throat competition resulting in the need for increasingly smaller time-to-market windows. At the same time, embedded processors are becoming more complex due to the migration of increasingly more functionality to a single embedded processor resulting most likely in undesirable higher development costs. This has led to the quest for a flexible and reusable embedded processor that still must achieve the required performance levels. As a result, embedded processors have evolved from simple microcontrollers to digital signal processors to programmable processors. We believe that this quest is leading to embedded processors that comprise a programmable processor augmented with reconfigurable hardware. In this paper, we highlight several embedded processors characteristics and discuss how they have evolved over time when programmability and reconfigurability were introduced into the embedded processor design. Finally, we describe in-depth one possible approach that combines both programmability and reconfigurability in an integrated manner by utilizing microcode.

I. Introduction

A technology turning point that made embedded consumer electronics systems an everyday reality has to be the advent of microprocessors. The technological developments that allowed single-chip processors (microprocessors) made embedded systems inexpensive and flexible. Consequently, microprocessor-based embedded systems have been introduced into many new application areas. Currently, embedded programmable microprocessors in one form or another, from 8-bit micro-controllers to 32-bit digital signal processors and 64-bit RISC processors, are everywhere, in consumer electronic devices, home appliances, automobiles, network equipment, industrial control systems, etc. Interestingly, we are utilizing more than several dozens of embedded processors in our day-to-day lives

without actually realizing it. For example, in modern cars such as the Mercedes S-class or the BMW 7-series, we can find over 60 embedded processors that control a multitude of functions, e.g., the fuel injection and the anti-lock braking system (ABS), that guarantee a smooth and foremost safe drive. Furthermore, the employment of embedded (micro)processors appear to grow in an exponential curve.

In this *positional* paper, we describe several characteristics of embedded processors and investigate how these characteristics have changed over time driven by market requirements such as smaller time-to-market windows and reduced development costs. Subsequently, we discuss two widely employed strategies to meet such market requirements, namely programmability and reconfigurability. Finally, we present a possible future direction in embedded processor design that merges both strategies and thereby providing flexibility in both software and hardware design at the same time.

This paper is organized as follows. Section II introduces a definition of embedded systems, discusses the ensuing characteristics of embedded systems, and provides an in-depth discussion of traditional embedded processors characteristics. Section III discusses the need for programmability and several examples of such an approach. Section IV discusses how the use of reconfigurability affected the embedded processors characteristics. Section V describes a possible future direction in embedded processor design that combines programmability and reconfigurability. Furthermore, we show an example of such an approach called the microcoded reconfigurable embedded processor, also called the MOLEN processor. Section VI concludes this paper by stating several key observations in this paper.

II. Traditional Embedded Processor Characteristics

An embedded processor is a specific instance of embedded systems in general and therefore adhere to the characteristics of embedded systems. Since no generally accepted definition of embedded systems exists, we establish our own definition in order to facilitate the discussion on embedded system characteristics and subsequently on embedded processor design issues.

Definition: *Embedded systems are (inexpensive) mass-produced elements of a larger system providing a dedicated, possibly time-constrained, service to that system.*

Before we highlight the main characteristics of embedded systems, we would like to comment on our one sentence definition of them. In most literature, the definition of embedded systems only states that they provide a dedicated service – the nature of the service is not relevant in this context – to a larger (embedding) system. Consequently, when we refer to embedded systems as mass-produced elements we draw the separation line between application-specific systems and embedded systems. We are aware that the separation line is quite thin in the sense that embedded systems are mostly indeed application-specific systems. However, we believe that application-specific systems produced in low volumes can not be considered to be embedded systems, because they represent a niche market for which completely different requirements are valid. For example, cost is unimportant in a low-volume production scenario contrary to the paramount importance to achieve low cost for embedded systems. Finally, we include the possibility for time-constrained behavior in our definition, because even if it is not characteristic to all the embedded systems it constitutes a particularity of a very large class of them, namely the real-time embedded systems.

Clearly, the precise requirements of an embedded system is determined by its immediate environment. The immediate environment of an embedded system can be either other surrounding embedded systems in the larger embedding system or even the world in which the larger system is placed. We can classify the embedded system requirements into:

- **Functional requirements** are defined by the services that the embedded system has to perform for and when interacting with its immediate environment. Such services possibly include data gathering and exerting control to their immediate environment. This implies that some kind of data transformation must be performance within the embedded system itself.
- **Temporal requirements** are the result of the time-constrained behavior of many embedded systems thereby introducing deadlines

(explained later) for the service(s).

- **Dependability requirements** relates to the reliability, maintainability, and availability of the embedded system in question.

In the light of the previously stated embedded systems definition and requirements, we briefly point out what we think are the main characteristics of more traditional embedded processors. Furthermore, we discuss in more detail the implications that these characteristics have on the specification and design processes of embedded processors. The first and probably the most important characteristic of embedded processors is that they are **application-specific**. Given that the service (or application in processor terms) is known a priori, the embedded processor can be and should be optimized for its targeted application. In other words, embedded processors are definitely not general-purpose processors which are designed to perform reasonably for a much wider range of applications. Moreover, the fact that the application is known beforehand opens the road for *hardware/software co-design*, i.e., the cooperative and concurrent design of both hardware and software components of the processor. The hardware/software co-design style is very much particular to embedded processors and has the goal of meeting the processor level objectives by exploiting the synergism of hardware and software.

Another important characteristic of embedded processors is their **static structure**. When considering an embedded processor, the end-user has very limited access to programming. Most of the software is provided by the processor integrator and/or application developer, resides on ROM memories, and runs without being visible to the end-user. The end-user can not change nor reprogram the basic operations of the embedded processor, but he is usually allowed to program the embedded system by re-arranging the sequence of basic operations.

Embedded processors are essentially non-homogeneous processors and this characteristic is induced by the **heterogeneous** character of the process within which the processor is embedded. Designing a typical embedded processor does not only mix hardware design with software design, but it also mixes design styles within each of these categories. To put more light on the heterogeneity issue, we depicted in Figure 1 (from [7]) an example signal processing embedded processor. The heterogeneous character can be seen in many aspects of the embedded processor design as follows:

- both analog and digital circuits may be present in the system;
- the hardware may include microprocessors, microcontrollers, digital signal processors, and application-specific integrated circuits;
- the topology of the system is rather irregular;
- the software may include various software modules as well as a multitasking real-time operating system.

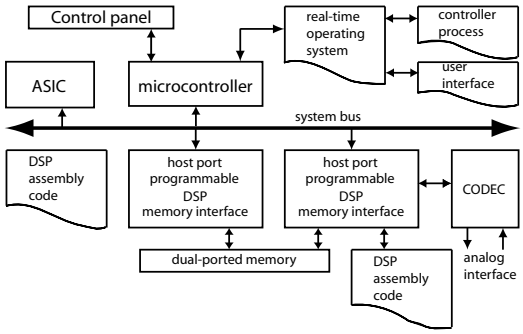

Figure 1. Signal Processing Embedded Processor Example (from [7]).

Generally speaking, the intrinsic heterogeneity of embedded processors largely contributes to the overall complexity and management difficulties of the design process. However, one can say that heterogeneity is in the case of embedded processors design a necessary evil. It provides better design flexibility by providing a wide range of design options. In addition, it allows each required function to be implemented on the most adequate platform that is deemed necessary to meet the posed requirements.

Embedded processors are **mass-produced** application-specific elements separating them from other low-volume produced application-specific processors. Embedded processors represent a much larger market segment in which embedded processor vendors face fierce competition in order to gain market capitalization. Consequently, this environment imposes a different set of requirements on the embedded processor design. For example, such requirements involve the cost/performance sensitiveness of embedded processors making low cost almost always an issue.

A large number of embedded processors performs **real-time** processing introducing the notion of *deadlines*. Roughly speaking, deadlines can be

classified into hard and soft real-time deadlines. Missing a hard deadline can be catastrophic while missing a soft deadline only results in non-fatal glitches at most. Both types of deadlines are known a priori much like that the functionality is known beforehand. Therefore, deadlines determine the minimum level of performance that must be achieved. When facing hard deadlines, special attention must also be paid to other components within the larger embedding system that are connected to the embedded processor in question since they can negatively influence its behavior.

III. The Need for Programmability

In the early nineties, we witnessed a trend in the embedded processors market that was reshaping the characteristics of traditional embedded processors as introduced in Section II. Driven by market forces, the lengthy embedded processors design cycles had to be shortened in order to keep up with or stay in front of competitors and costs had to be reduced in order to stay competitive. More specifically, the cost of an embedded processor can be largely divided into production costs (closely related to the utilized manufacturing technology) and development costs (closely related to overall design cycle). It must be clear that the production costs remain constant for each produced embedded processor due to the fact that the embedded processor design must be fixed before entering production. Since we focus on embedded processor design and not on manufacturing, the issues concerning production costs are left out of the ensuing discussion. However, we must note that the complexity of the final embedded processor design certainly has an impact on the production costs. The impact is exhibited by requiring more steps in the manufacturing process and/or a more expensive manufacturing process altogether. On the other hand, the development costs on a per embedded processor basis can be reduced by amortizing the costs over a higher production volume. Certainly, this greatly depends on the market demand and the established market capitalization. Alternatively and maybe more beneficial is to reduce the design cycle and therefore its associated costs altogether. In this section, by highlighting the traditional embedded processors design, we discuss “large scale” programmability which has been used to address the issues of lengthy design cycles and

the associated development costs. One could argue that programmability has always been part of embedded processors. However, programmability introduced in this section significantly differs from the limited (low-level) programmability of traditional embedded processors.

The heterogeneity of the embedded systems demanded a multitude of embedded processors to be designed for a single system. This was further strengthened by the fact that the semiconductor technology at the time did not allow large chips to be manufactured. Subsequently, the design of embedded processors requires lengthy design cycles and especially lengthy verification cycles for the chips and their interfaces. On the other hand, one can argue that an advantage is that subsequent system design cycles could significantly be reduced in only one or few embedded processor(s) needed to be redesigned. This delicate balance between long *initial design cycles* and possibly shortened *subsequent design cycles* was disturbed when advances in semiconductor technology allowed increasingly more gates to be put on a single chip. As a result, more functionality migrated from a multitude of embedded processors into a single one. The resulting design of such more complex and larger embedded processors did not have a large effect on the initial design cycles. However, the length of subsequent redesign cycles increased since the utilization of optimized circuits means that subsequent designs are not necessarily easier than the initial ones.

In the search for design flexibility in order to decrease design cycles and reduce subsequent design costs, functions were separated into time-critical functions and non-time-critical ones. One could say that the embedded processors design paradigm has shifted from one that is based on the functional requirements to one that is based on the temporal requirements. The collection of non-time-critical functions could then be performed by a single chip (possibly implemented in a slower technology in order to reduce cost). The remaining time-critical functions are to be implemented in high-speed circuits achieving maximum performance. The main benefit of this approach is that the larger and (possibly) slower chip can be reused in subsequent designs resulting in shorter subsequent design cycles. While this design paradigm was born out of market needs, i.e., to reduce design cycles and development costs, it is well-known in the design of general-purpose processors. In the general-purpose processor design paradigm, the

processor design can be divided into three distinct fields [5]: architecture[1], implementation, and realization.

In Section II, we stated that more traditional embedded processors are application-specific and static in nature. However, in this section we also stated that increasingly more functionality is embedded into a single embedded processor. Is such a processor still application-specific and can we still call such a processor an embedded processor? The answer to this question is affirmative since such a processor is still embedded if the other constraints (mass-produced, providing a dedicated service, etc.) are observed. Given that increasing functionality usually implies more exposure of the processor to the programmer, embedded processor have become indeed less static as they can now be reused for other applications areas due to their programmability. In the light of this all, two scenarios in the design of programmable embedded processors can be distinguished:

- **Adapt an existing general-purpose architecture** and implement it. This scenario reduces development costs albeit such architectures must usually be licensed. Furthermore, since such architectures were not adapted to embedded processors still some development time is needed to modify such architectures.
- **Build a new embedded processor architecture** from scratch. In this scenario, the embedded processor development takes longer, but the final architecture is more tuned towards the targeted application(s) and thus possibly achieving better performance than already existing general-purpose architectures. Actually, the goal is to develop an architecture for a collection of similar applications (called application domain) such that processors can be produced once and reused when placed in different environments. This reduces the overall system cost since the development costs are amortized over a higher number of embedded processors.

Several examples of the first scenario can be found. A well-known example is the MIPS architecture [13] which has been adapted resulting in several embedded processor families. In this case, the architecture has been

[1] The architecture of any computer system is defined to be the conceptual structure and functional behavior as seen by its immediate user.

increasingly more adapted towards embedded processors by MIPS Technologies, Inc. which develops the architecture independently from other embedded systems vendors. Another well-known example is the ARM architecture [21] found in many current embedded processors. It is a RISC architecture that was intended for low-power PCs (1987) at first, but it has been quickly adapted to become an embeddable RISC core (1991). Since then the architecture has been modified and extended several times in order to optimize it for its intended applications. The most well-known version is the StrongARM core which was jointly developed by Digital Semiconductor and ARM. This core was intended to provide great performance at extreme low-power. The most recent and extended implementation of this architecture is developed by Intel called the Intel PCA Application Processor [14]. Other examples of general-purpose architectures that have been adapted include: IBM PowerPC [16], Sun UltraSPARC [25], the Motorola 68k/Coldfire [18]. An example of the second scenario is the Trimedia VLIW architecture [22] from Trimedia Technologies, Inc. which was originally developed by Philips Electronics, N.V. Its application domain is multimedia processing and processors based on this architecture can be found in television sets, digital receivers, and other digital video editing boards. It contains a VLIW processor core that performs non-time-critical functions and also controls the specialized hardware units that are intended for specific real-time multimedia processing.

Summarizing, the characteristics mentioned in Section II can be easily reflected in the three design stages: architecture, implementation, and realization. The characteristic of embedded processors being application-specific processors is exhibited by the fact that the architecture only contains those instructions that are really needed to support the application domain. The static structure characteristic exhibits itself by having a fixed architecture, a fixed implementation, and a fixed realization. The heterogeneity characteristic exhibits itself by the utilization of a programmable processor core with other specialized hardware units. Such specialized hardware units can possibly be implemented on the same chip as the programmable processor core. Extending this principle further, the heterogeneity of the embedded processor also exhibits itself in the utilization of different functional units in the programmable processor core. The mass-produced characteristic is exhibiting itself in the realization process by

only utilizing proven technology that therefore should be available, cheap and reliable. The requirement of real-time processing exhibits itself by requiring architectural support for frequently used operations, extensively parallel and/or pipelined (if possible) implementations, and realizations incorporating adequately high-speed components.

IV. Early Time Reconfigurability

In the mid-nineties, we witnessed a second trend in the embedded processors design next to programmability that was likewise reshaping the design methodology of embedded processors and consequently redefined some of their characteristics. Traditionally, the utilization of application-specific integrated circuits (ASICs) was commonplace in the design of embedded processors resulting in lengthy design cycles. Such an approach requires several roll-outs of the embedded processor chips in question in order to test/verify all the functional, temporal, and dependability requirements. Therefore, design cycles of 18 months or longer were commonplace rather than exceptions. A careful step towards reducing such lengthy design cycles is to utilize reconfigurable hardware, also referred to as fast prototyping. The utilization of reconfigurable hardware allows embedded processor designs to be mapped early on in the design cycle to reconfigurable hardware, in particular field-programmable gate arrays (FPGAs), giving rise to three advantages. First, the mentioned mapping requires considerably less time than a chip roll-out and thereby shortening the development time. Second, the embedded processor functionality can be tested in an earlier stage and at the same allowing more design alternatives to be explored. Third, the number of (expensive) chip roll-outs is also reduced and thereby further reducing the development costs. However, the reconfigurable hardware was initially limited in size and speed. The limited size meant that only partial designs could be tested. Consequently, roll-out(s) of the complete embedded processor design (implemented in ASICs) were still required in order to verify the overall functionality and performance.

In recent years, the reconfigurable hardware technology has progressed in a fast pace arriving at the point where embedded processor designs requiring million(s) of gates can be implemented on such structures. More-

over, the existing performance gap between FPGAs and ASICs is rapidly decreasing. Due to these technological developments, the role of reconfigurable hardware in embedded processors design has changed considerably. In the following paragraphs, we revisit the traditional embedded processor characteristics mentioned in Section II and investigate whether they still hold in the case of FPGA-based embedded processors.

application-specific Embedded processors built utilizing reconfigurable hardware are still application-specific in the sense that the implementations are still targeting such applications. Utilizing such implementations for other purposes will prove to be very difficult or even impossible, because the required performance levels most certainly can not be achieved.

static structure From a pure technical perspective, the structure of a reconfigurable embedded processor is not static since its functionality can be changed during its lifetime. However, in most cases the design implemented in reconfigurable hardware remains fixed between maintenance intervals. Therefore, from the users perspective the structure of the embedded processor is still static. In the next section, we explore the possibility that the functionality of an embedded processor needs to be changed even during run-time. In this case, the static structure can be perceived from a higher perspective, namely the reconfigurable hardware is designed to support only a fixed (or you may call static) set of implementations.

heterogeneous This characteristics is still very much present in the case of reconfigurable embedded processors. We have added an additional technology into the mix in which embedded processors can be realized. For example, the latest FPGA offering from both Altera Inc. (Stratix [2]) and Xilinx Inc. (Virtex II [29]) integrates on a single chip the following: memory, logic, I/O controllers, and DSP blocks.

mass-produced This characteristic is still applicable to current reconfigurable embedded processors. Early on, reconfigurable hardware was expensive resulting in its sole utilization for fast prototyping purposes. As the technology progressed, reconfigurable hardware became cheaper and this opened the possibility of actually shipping reconfigurable embedded processors in final products. An important enabling trend next to reduced cost that must not be overseen is that reconfigurable hardware has also become more reliable both in production and during operation.

real-time In the beginning, we were witnessing the incorporation of re-

configurable hardware only for non-'time-critical' functions. As the technology of reconfigurable continue to progress and making reconfigurable hardware much faster, we are also witnessing their incorporation in actual products where real-time performance is required, such as multimedia decoders.

V. Future Embedded Processors

In Sections III and IV, we have argued that both programmability and reconfigurability have been introduced into the embedded processor design trajectory born out of the need to reduce design cycles and reduce development costs. In short, programmability allows the utilization of high-level programming languages (like C) making it easier to support applications on embedded processors. Reconfigurability allows designs to be tested early on in terms of functionality and diminishes the need for expensive chip roll-outs. Merging both strategies is a logical and evolutionary step in embedded processor design and has enormous potential, especially when considering that the performance of FPGAs is nearing that of ASICs. More precisely, we believe that the merging encompasses the augmentation of a programmable processor (core) with reconfigurable hardware, possibly replacing fixed (ASICs) hardware. We foresee that such an augmentation will provide several advantages:

- **improved performance** compared to a software-only implementation, because (tuned) specialized hardware implemented on the FPGA can exploit the parallelism of the supported function and allow the utilization of other performance-increasing techniques.
- **rapid application development** since the mentioned augmentation introduces the possible utilization of high-level programming and hardware description languages in the design trajectory.
- **design flexibility** is achieved by allowing design space exploration in both hardware and software due to the possible utilization of high-level programming and hardware description languages.

The mentioned advantages and enabling FPGA technologies have even resulted in that programmable processor cores are under consideration to be implemented in the same FPGA structures, e.g., Nios from Altera

[1] and MicroBlaze from Xilinx [30]. However, the utilization of programmable embedded processors that are augmented with reconfigurable hardware also poses several issues that must be addressed:

- **Long reconfiguration latencies:** In run-time reconfiguration, such latencies may greatly penalize the performance, because any computation must be halted until the reconfiguration has finished.
- **Limited opcode space:** The initiation and control of the reconfiguration and execution of various implementations on the reconfigurable hardware require the introduction of new instructions. This puts much strain on the opcode space.
- **Complicated decoder hardware:** The multitude of new instructions greatly increases the complexity of the decoder hardware.

In the following, we discuss one possible approach [24] (introduced by us) in merging programmability with reconfigurability in the design of embedded processors. The approach utilizes microcode to alleviate the mentioned problems. Microcode consists of a sequence of (simple) microinstructions that, when executed in a certain order, performs "complex" operations. This approach allows "complex" operations to be performed on much simpler hardware. In this section, we consider the reconfiguration (either off-line or run-time) and execution processes as complex operations. The main benefits of our approach can be summarized as follows:

- **Reduced reconfiguration latencies:** Microcode used to control the reconfiguration process allows itself to be cached on-chip. This results in faster access times to the reconfiguration microcode and thus in turn reduces the reconfiguration latencies.
- **Reduced opcode space requirements:** By only pointing to microcode (explained later), we only require (at most) three new instructions to support any current and future operations.
- **Reduced decoder complexity:** By introducing only a few instructions, no complex instruction decoding hardware is required.

In Section A, we revisit microcode from its beginnings to its current implementation within a high-level microprogrammed machine. In Section B, we discuss in-depth our proposed MOLEN embedded processor. Finally, in Section C, we briefly highlight several other approaches in this

field that are comparable in one way or another.

A. Revisiting Microcode

Microcode, introduced in 1951 by Wilkes [26], constitutes one of the key computer engineering innovations. Microcode de facto partitioned computer engineering into two distinct conceptual layers, namely: architecture and implementation. This is in part because emulation allowed the definition of complex instructions which might have been technologically not implementable (at the time they were defined), thus projecting an architecture to the future. That is, it allowed computer architects to determine a technology-independent functional behavior (e.g., instruction set) and conceptual structures providing the following possibilities:

- Define the computer's architecture as a programmer's interface to the hardware rather than to a specific technology dependent realization of a specific behavior.
- Allow a single architecture to be determined for a "family" of implementations giving rise to the concept of compatibility. Simply stated, it allowed programs to be written for a specific architecture once and run at "infinitum" independent of the implementations.

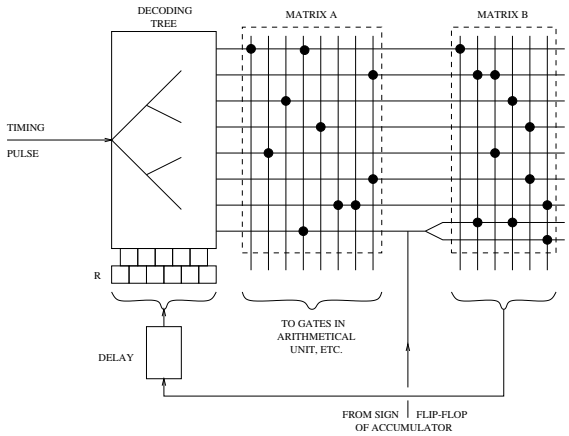

Figure 2. Wilkes' microprogram control model [26].

Since its beginnings, as introduced by Wilkes, microcode has been a sequence of micro-operations (microprogram). Such a microprogram consists of pulses for operating the gates associated with the arithmetical and control registers. Figure 2 depicts the method of generating this sequence of pulses. First, a timing pulse initiating a micro-operation enters the decoding tree and depending on the setup register R, an output is generated. This output signal passes to matrix A which in turn generates pulses to control arithmetical and control registers, thus performing the required micro-operation. The output signal also passes to matrix B, which in its turn generates pulses to control the setup register R (with a certain delay). The next timing pulse will therefore generate the next micro-operation in the required sequence due to the changed register R.

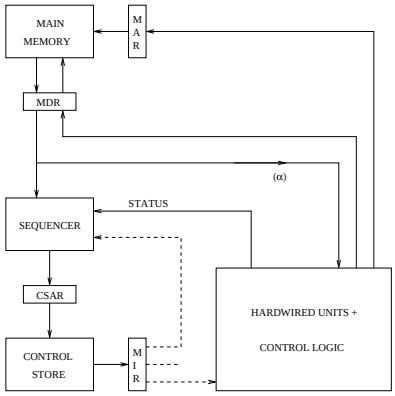

Figure 3. A high-level microprogrammed machine.

Over the years, the Wilkes' model has evolved into a high-level microprogrammed machine as depicted in Figure 3. In this figure, the memory address register (MAR) is used to store the memory address in the main memory from which data must be loaded of to which data is stored. The memory data register (MDR) stores the data that is communicated to or from the main memory. Furthermore, the control store contains microinstructions (representing one or more micro-operations) and the sequencer determines the next microinstruction to execute. The control store and the sequencer correspond to Wilkes' matrices A and B respectively. The machine's operation is as follows:

1. The control store address register (CSAR) contains the address of the next microinstruction located in the control store. The microinstruction located at this address is then forwarded to the microinstruction register (MIR).
2. The microinstruction register (MIR) decodes the microinstruction and generates smaller micro-operation(s) accordingly that need to be performed by the hardware unit(s) and/or control logic.
3. The sequencer utilizes status information from the control logic and/or results from the hardware unit(s) to determine the next microinstruction and stores its control store address in the CSAR. It is also possible that the previous microinstruction influences the sequencer's decision regarding which microinstruction to select next.

It should be noted that in microcoded engines not all instructions access the control store. As a matter of fact, only emulated instructions have to go through the microcode logic. All other instructions will be executed directly by the hardware (following path (α) in Figure 3). That is, a microcoded engine is as a matter of fact a hybrid of the implementation having emulated instructions and hardwired instructions. We have to note that contrary to some believes, from the moment it was possible to implement instructions, microcoded engines always had a hardwired core that executed RISC instructions.

B. Microcoded Reconfigurable MOLEN Embedded Processor

In this section, only a brief description of the MOLEN embedded processor is given. We refer to [24][28] for a more detailed description. In its most general form, the proposed machine organization augmented with a reconfigurable unit is depicted in Figure 4. In this organization, instructions are fetched from the main memory and are temporarily stored in the 'Instruction Fetch' unit. Subsequently, these instructions are fetched by the 'Arbiter' which decodes them before issuing them to their corresponding execution units. Instructions that have been implemented in fixed hardware are issued to the 'Core Processing Units', i.e., the regular functional units such as ALUs, multipliers, and dividers. Instructions that have been imple-

mented in reconfigurable hardware are issued to the 'Reconfigurable Unit'. Similar to other load/store architectures, the proposed machine organization executes on data that is stored in the register file and prohibits direct memory data accesses by hardware units other than the load/store unit(s). However, there is one exception to this rule, the custom configured unit (CCU), which embodies the actual reconfigurable hardware (e.g., FPGA), is also allowed direct memory data access via the 'Data Fetch/Store' unit (represented by a dashed two-ended arrow). This enables the CCU to perform much better when streaming data accesses are required, e.g., in multimedia processing. Finally, we introduced the exchange registers (XREGS) which are utilized to accommodate a more extensive argument passing mechanism (compared to registers which are restricted in number and size) between the complex implementations configured on the CCU and the application code which embeds such implementations.

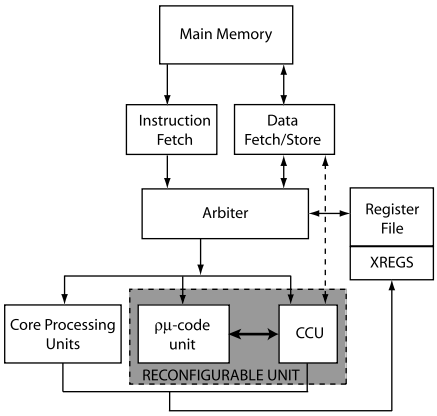

Figure 4. *The MOLEN machine organization.*

The reconfigurable unit consists of a custom configured unit (CCU), which could be for example be implemented by a Field-Programmable Gate Array (FPGA), and the $\rho\mu$-code unit. An operation, which can be as simple as an instruction or as complex as a piece of code, performed by the reconfigurable unit is divided into two distinct process phases: **set** and **execute**. The **set** phase is responsible for configuring the CCU enabling it to perform the required operation(s). Such a phase may be subdivided into

two sub-phases: partial **set** (*p*-**set**) and complete **set** (*c*-**set**). The *p*-**set** sub-phase is envisioned to cover common functions of an application or set of applications. More specifically, in the *p*-**set** sub-phase the CCU is *partially* configured to perform these common functions. While the *p*-**set** sub-phase can be possibly performed during the loading of a program or even at chip fabrication time, the *c*-**set** sub-phase is performed during program execution. In the *c*-**set** sub-phase, the remaining part of the CCU (not covered in the *p*-**set** sub-phase) is configured to perform other less common functions and thus *completing* the functionality of the CCU. The configuration of the CCU is performed by executing reconfiguration microcode (either loaded from memory or resident) in the $\rho\mu$-code unit. Reconfiguration microcode is generated by translating a reconfiguration file into microcode. In the case that partial reconfigurability is not possible or not convenient, the *c*-**set** sub-phase can perform the entire configuration. The **execute** phase is responsible for actually performing the operation(s) on the (now) configured CCU by executing (possibly resident) execution microcode stored in the $\rho\mu$-code unit.

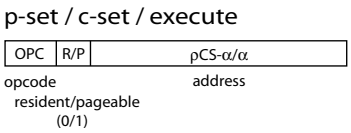

Figure 5. The *p*-**set**, *c*-**set**, and **execute** instruction formats.

In relation to these three phases, we introduce three new instructions: *c*-**set**, *p*-**set**, and **execute**. Their instruction format is given in Figure 5. We must note that these instructions do *not* specifically specify an operation and then load the corresponding reconfiguration and execution microcode. Instead, the *p*-**set**, *c*-**set**, and **execute** instructions directly point to the (memory) location where the reconfiguration or execution microcode is stored. In this way, different operations are performed by loading different reconfiguration and execution microcodes. That is, instead of specifying new instructions for the operations (requiring instruction opcode space), we simply point to (memory) addresses. The location of the microcode is indicated by the resident/pageable-bit (R/P-bit) which implicitly determines the interpretation of the address field, i.e., as a memory address α

(R/P=1) or as a ρ-CONTROL STORE address ρCS-α (R/P=0) indicating a location within the $\rho\mu$-code unit. This location contains the first instruction of the microcode which must always be terminated by an *end_op* microinstruction.

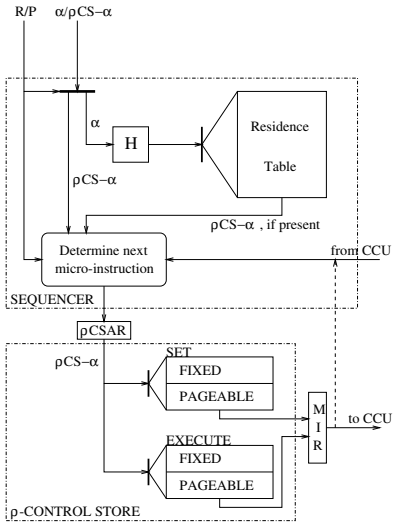

Figure 6. $\rho\mu$-code unit internal organization.

The $\rho\mu$-code unit: The $\rho\mu$-code unit can be implemented in configurable hardware. Since this is only a performance issue and not a conceptual one, it is not considered further in detail. In this presentation, for simplicity, we assume that the $\rho\mu$-code unit is hardwired. The internal organization of the $\rho\mu$-code unit is given in Figure 6. In all phases, microcode is used to perform either reconfiguration of the CCU or control the execution on the CCU. Both types of microcode are conceptually the same and no distinction is made between them in the remainder of this section. The $\rho\mu$-code unit comprises two main parts: the SEQUENCER and the ρ-CONTROL STORE. The SEQUENCER mainly determines the microinstruction execution sequence and the ρ-CONTROL STORE is mainly used as a storage facility for microcodes. The execution of microcodes starts with the SEQUENCER receiving an address from the ARBITER and interpreting it according to the R/P-bit. When receiving a memory address, it must be

determined whether the microcode is already cached in the ρ-CONTROL STORE or not. This is done by checking the RESIDENCE TABLE which stores the most frequently used translations of memory addresses into ρ-CONTROL STORE addresses and keeps track of the validity of these translations. It can also store other information: least recently used (LRU) and possibly additional information required for virtual addressing support. In the remainderm we assume that the system only allows for real addressing for simplicity of discussion. In the cases that a ρCS-α is received or a valid translation into a ρCS-α is found, it is transferred to the 'determine next microinstruction'-block. This block determines which (next) microinstruction needs to be executed:

- When receiving address of first microinstruction: Depending on the R/P-bit, the correct ρCS-α is selected, i.e., from instruction field or from RESIDENCE TABLE.
- When already executing microcode: Depending on previous microinstruction(s) and/or results from the CCU, the next microinstruction address is determined.

The resulting ρCS-α is stored in the ρ-control store address register (ρCSAR) before entering the ρ-CONTROL STORE. Using the ρCS-α, a microinstruction is fetched from the ρ-CONTROL STORE and then stored in the microinstruction register (MIR) before it controls the CCU reconfiguration or before it is executed by the CCU.

The ρ-CONTROL STORE comprises two sections[2], namely a **set** section and an **execute** section. Both sections are further divided into a **fixed** part and **pageable** part. The fixed part stores the resident reconfiguration and execution microcode of the **set** and **execute** phases, respectively. Resident microcode is commonly used by several invocations (including reconfigurations) and it is stored in the fixed part so that the performance of the **set** and **execute** phases is possibly enhanced. Which microcode resides in the fixed part of the ρ-CONTROL STORE is determined by performance analysis of various applications and by taking into consideration various software and hardware parameters. Other microcodes are stored in memory and the pageable part of the ρ-CONTROL STORE acts like a cache to

[2] Both sections can be identical, but are probably only differing in microinstruction wordsizes.

provide temporal storage. Cache mechanisms are incorporated into the design to ensure the proper substitution and access of the microcode present in the ρ-CONTROL STORE.

C. Other reconfigurability approaches

In the previous section, we have introduced a machine organization where the hardware reconfiguration and the execution on the reconfigured hardware is done in firmware via the ρ-microcode (an extension of the classical microcode to include reconfiguration and execution for resident and non-resident microcode). The microcode engine is extended with mechanisms that allow for permanent and pageable reconfiguration and execution microcode to coexist. We also provide partial reconfiguration possibilities for "off-line" configurations and prefetching of configurations. Regarding related work we have considered more than 40 machine proposals. We report here a number of them that somehow use some partial or total reconfiguration prefetching. It should be noted that our scheme is rather different in principle from all related work as we use microcode, pageable/fixed local memory, hardware assists for pageable reconfiguration, partial reconfigurations, etc.. As it will be clear from the short description of the related work, we differentiated from them in one or more mechanisms.

The *Programmable Reduced Instruction Set Computer (PRISC)* [19] attaches a Programmable Functional Unit (PFU) to the register file of a processor for application-specific instructions. Reconfiguration is performed via exceptions. In an attempt to reduce the overhead connected with FPGA reconfiguration, Hauck proposed a slight modification to the PRISC architecture in [11]: an instruction is explicitly provided to the user that behaves like a NOP if the required circuit is already configured on the array, or is in the process of being configured. By inserting the configuration instruction before it is actually required, a so-called *configuration prefetching* procedure is initiated. At this point the host processor is free to perform other computations, overlapping the reconfiguration of the PFU with other useful work. The *OneChip* introduced by Wittig and Chow [27] extends PRISC and allows PFU for implementing any combinational or sequential circuits, subject to its size and speed. The system proposed by

Trimberger [23] consists of a host processor augmented with a PFU, *Reprogrammable Instruction Set Accelerator* (RISA), much like the PRISC mentioned above. Concerning the management and control of the reprogramming procedure, Trimberger mentions that the RISA reconfiguration is under control of a hardwired execution unit. However, it is not obvious if an explicit SET instruction is available. The *Reconfigurable Multimedia Array Coprocessor* (REMARC) proposed by Miyamori and Olukotun [17] augments the instruction set of a MIPS core. As the coprocessor does not have a direct access to the main memory, the host processor has to write the input data to the coprocessor data registers, initiate the execution, and finally read the results from the coprocessor data registers. An explicit reconfiguration instruction is provided. *Garp* designed by Hauser and Wawrzynek [12] is another example of a MIPS derived Custom Computing Machine (CCM). The FPGA-based coprocessor has a direct access to the standard memory. The MIPS instruction set is augmented with several non-standard instructions dedicated to loading a new configuration, initiating the execution of the newly configured computing facilities, moving data between the array and the processor's own registers, saving/retriving the array states, branching on conditions provided by the array, etc. The coprocessor is aimed to run autonomously with the host processor. In the *OneChip-98* introduced by Jacob and Chow [15], the computing resources are loaded *on-demand* when a miss is detected. *Alternatively*, the resources are *pre-loaded* by using compiler directives. Several comments regarding these assertions are worth to be provided. If an on-demand loading strategy is employed, then the user has no control on the reconfiguration procedure. In the pre-loading strategy, an explicit reconfiguration instruction is provided to the user and the reconfiguration procedure is indeed under the control of the user. PRISM (*Processor Reconfiguration Through Instruction-Set Metamorphosis*) one of the earliest proposed CCM [3][4], was developed as a proof-of-concept system, in order to handle the loading of FPGA configurations, the compiler inserts library function calls into the program stream [4]. From this description, we can conclude that an explicit reconfiguration procedure is available. Gilson [8] CCM architecture consists of a host processor and two or more FPGA-based *computing devices*. The host controls the reconfiguration of FPGAs by loading new configuration data through a Host Interface into the FPGA Configuration Memory.

The reconfiguration process can be performed such that when one computing device is being reconfigured and, therefore, is idle, the others continue executing. The write into the configuration memory instruction can play the role of an explicit reconfiguration instruction. Therefore, a *pre-loading* strategy is employed. Schmit [20] proposes a partial run-time reconfiguration mechanism, called *pipeline reconfiguration* or *striping*, by which the FPGA is reconfigured at a granularity that corresponds to a pipeline stage of the application being implemented. An application which has been broken up into pipeline stages can be mapped to a striped FPGA. The pipeline stages are known as *stripes*; the stages of the application are called *virtual stripes*, and the hardware stages which the virtual stages are loaded into are called *physical stripes*. The PipeRench coprocessor developed by a team with Carnegie Mellon University [6][9] is focused on implementing linear (1-D) pipelines of arbitrary length. PipeRench is envisioned as a coprocessor in a general-purpose computer, and has direct access to the same memory space as the host processor. The virtual stripes of the application are stored into an on-chip configuration memory. A single physical stripe can be configured in one read cycle with data stored in such a memory. The configuration of a stripe takes place concurrently with execution of the other stripes. The *Reconfigurable Data Path Architecture* (rDPA) is also a self-steering autonomous reconfigurable architecture. It consists of a mesh of identical Data Path Units (DPU) [10]. The data-flow direction through the mesh is only from west and/or north to east and/or south and is also data-driven. A word entering rDPA contains a configuration bit which is used to distinguish the configuration information from data. Therefore, a word can specify either a SET or an EXECUTE instruction, the arguments of the instructions being the configuration information or data to be processed. A set of computing facilities can be configured on rDPA.

VI. Conclusions

In this positional paper, we described several characteristics of embedded processors that were logically deduced from embedded systems characteristics in general. Driven by market requirements, two strategies were followed in order to reduce design cycles and development costs.

First, programmability was introduced as a means to combine all non-time-critical functions to be performed by a 'general-purpose'-like embedded processor. Such an embedded processor could then be reused in subsequent designs and thereby greatly reducing design cycles. Second, reconfigurability was initially only utilized for fast prototyping. Over time, technological advances in reconfigurable hardware in terms of size and performance have led to the fact the reconfigurable embedded processors are actually incorporated in shipped embedded systems. We believe that the future of embedded processors design lies in the merging of both strategies. Programmability allows the utilization of high-level programming languages (like C) and thereby easing application development. The utilization of reconfigurable hardware combines design flexibility and fast prototyping. At the same time, the processing performance of reconfigurable hardware is nearing that of application-specific integrated circuits. Finally, in this paper we have highlighted one possible framework in which future embedded processor design can be performed. The proposed MOLEN embedded processor combines software programming (by utilizing a programmable processor core) with hardware programming (utilizing microcode to control the reconfigurable hardware). Such an approach provides possibilities in combatting several issues associated with reconfigurable hardware.

References

1. Altera Corporation. Nios Embedded Processor. http://www.altera.com/products/devices/excalibur/exc-nios_index.html.
2. Altera Corporation. Stratix Family. http://www.altera.com/products/devices/stratix/stx-index.jsp.
3. P.M. Athanas. *An Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration*. PhD thesis, Brown University, Providence, Rhode Island, May 1992.
4. P.M. Athanas and H.F. Silverman. Processor Reconfiguration through Instruction-Set Metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.
5. G.A. Blaauw and F.P. Brooks. *Computer Architecture: Concepts and Evolution*. Addison-Wesley, 1997.

6. S. Cadambi, J. Weener, S.C. Goldstein, H. Schmit, and D.E. Thomas. Managing Pipeline-Reconfigurable FPGAs. In *6th International Symposium on Field Programmable Gate Arrays*, pages 55–64, California, USA, 1998.
7. W.-T. Chang, A. Kalavade, and E.A. Lee. Effective Heterogeneous Design and Co-Simulation. In Giovanni de Michelli and Mariagiovanna Sami, editors, *Hardware/Software Co-Design*, pages 187–211. Kluwer Academic Publishers, 1995.
8. K.L. Gilson. Integrated Circuit Computing Device Comprising a Dynamically Configurable Gate Array Having a Microprocessor and Reconfigurable Instruction Execution Means and Method Therefore. U.S. Patent No. 5,361,373, November 1994.
9. S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *The 26th International Symposium on Computer Architecture*, pages 28–39, Georgia, USA, May 1999.
10. R.W. Hartenstein, R. Kress, and H. Reinig. A New FPGA Architecture for Word-Oriented Datapaths. In *4th International Workshop on Field-Programmable Logic: Architectures, Synthesis and Applications.*, Lecture Notes in Computer Science, pages 144–155, Czech Republic, September 1994.
11. S.A. Hauck. Configuration Prefetch for Single Context Reconfigurable Coprocessors. In *6th International Symp. on Field Programmable Gate Arrays*, pages 65–74, California, 1998.
12. J.R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *IEEE Symp. on FPGAs for Custom Computing Machines*, pages 12–21, California, 1997.
13. K. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.
14. Intel Corporation. Intel PCA Application Processors. http://www.intel.com/design/pca/applicationsprocessors/index.htm.
15. J.A. Jacob and P. Chow. Memory Interfacing and Instruction Specification for Reconfigurable Processors. In *ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, pages 145–154, Monterey, California, 1999.
16. C. May, E. Silha, R. Simpson, and H. Warren. *The PowerPC Architecture*. Morgan Kaufmann Publishers, Inc., 1994.

17. T. Miyamori and K. Olukotun. A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 2–11, California, 1998.
18. Motorola. Motorola 68000/Coldfire Family. http://e-www.motorola.com/webapp/sps/site/homepage.jsp?nodeId=03M0ylgrpxN.
19. R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, Cambridge, MA, USA, 1994.
20. H. Schmit. Incremental Reconfiguration for Pipelined Applications. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 47–55, California, April 1997.
21. D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley, 2000.
22. S. Rathnam and G. Slavenburg. An Architectural Overview of the Programmable Multimedia Processor, TM-1. In *Proceedings of COMPCON '96*, pages 319–326. IEEE, 1996.
23. S.M. Trimberger. Reprogrammable Instruction Set Accelerator. U.S. Patent No. 5,737,631, April 1998.
24. S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN $\rho\mu$-Coded Processor. In *Proc. of the 11th Intern. Conf. on Field-Programmable Logic and Applications (FPL2001)*, pages 275–285, 2001.
25. D.L. Weaver and T. Germond, editors. *The SPARC Architecture Manual (v9)*. Prentice Hall, 1994.
26. M. V. Wilkes. The Best Way to Design an Automatic Calculating Machine. In *Report of the Manchester University Computer Inaugural Conference*, pages 16–18, July 1951.
27. R.D. Wittig and P. Chow. OneChip: An FPGA Processor with Reconfigurable Logic. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, 1996.
28. S. Wong. Microcoded Reconfigurable Embedded Processors. PhD thesis, Delft University of Technology, Delft, The Netherlands, December 2002.
29. Xilinx Corporation. Virtex-II 1.5V FPGA Family: Detailed Functional Description. http://www.xilinx.com/partinfo/databook.htm.
30. Xilinx Corporation. Xilinx MicroBlaze. http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=microblaze.