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. Such embedded systems can be found in devices ranging from simple controllers found in power plants to sophisticated multimedia set-top boxes found 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. This has resulted in the fact that much effort must be placed in the design of embedded processors. In the last decade, we have been witnessing several changes in the embedded processors design fueled by two conflicting trends. First, the industry is dealing with cut-throat competition resulting in the need for increasingly faster time-to-market times in order to cut development costs. At the same time, embedded processors are becoming more complex due to the migration of increasingly more functionality to a single embedded processor in order to cut production costs. This has led to the quest for a flexible and reusable embedded processor which must still achieve high 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 an embedded processor that comprises 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.*

1 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 the 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. The employment of embedded processors appear to grow in an exponential curve. Furthermore, it has been postulated [22] that the sales trend of embedded processors (microprocessors in this setting) will significantly outperform the sales of general-purpose PC processors.

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 faster time-to-market times and development costs reductions. We will show that two strategies have been widely used to meet such market requirements, namely programmability and reconfigurability. Finally, we show a possible future direction in the 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 2 introduces a general definition of embedded systems, discusses the characteristics of embedded systems that follow from the definition, and provides an in-depth discussion of traditional embedded processors characteristics. Section 3 discusses the need for programmability and several examples of such an approach. Section 4 discusses the use for reconfigurability and discusses how it affected the embedded processor's characteristics. Section 5 continues our discussion by describing what we think is the direction for future embedded processor that combines programmability and reconfigurability. Furthermore, we show an example of such an approach called the microcoded reconfigurable MOLEN embedded processor. Section 6 concludes this paper by stating several key observations in this paper.

2 Traditional Embedded Processor Characteristics

Embedded processors are a specific instance of embedded systems in general and therefore adhere to the characteristics of embedded systems. In this section, we provide a more traditional view on embedded processors by stating their characteristics deduced from our general definition of embedded systems:

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. However, we believe that all the issues related to the specification and design of embedded systems are very much anchored in the market reality. Consequently, in our opinion 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 low-production application-specific systems can not be considered as embedded systems, because they represent a niche market with very different set of requirements. For example, in low-production scenarios cost is usually not important while it is almost paramount for embedded systems to achieve low cost. 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.

The precise requirements of an embedded system is determined by its immediate environment. However, we still can classify the embedded system requirements in:

– **Functional requirements** are defined by the services that the embedded system has to perform for its immediate environment[1]. Such services usually include data gathering and some kind of data transformation/processing.
– **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 and discuss in more detail the implications that these characteristics have on their specification and design processes. The first and probably the most important characteristic of embedded processors is that they are **application-specific**[2]. 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. It is misleading to think that application-specific processors can not be programmed, because the signals controlling the processor can be perceived as rudimentary processor instructions, e.g., firmware or microcode [29], which could be re-arranged thus programmed. 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 software programming. The utilized software is provided by the processor integrator and/or application developer, resides on ROM memories, and is not visible to the end-user. The

[1] The immediate environment of an embedded systems can be either other embedded systems in the larger system or the world in which the larger system is placed.

[2] In accordance with our embedded systems definition, embedded processors are mass-produced application-specific processors. Therefore, we consider graphics processors in game consoles to be embedded processors. On the other hand, graphics processors intended for military simulators are not since they are not mass-produced.

end-user can not change nor reprogram the basic operations of the embedded processor, but he is usually allowed to program a (different) 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 [16]) 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 sub-processors may be present in the system;
- the hardware may include microprocessors, microcontrollers, digital signal processors (DSPs), application-specific integrated circuits (ASICs);
- the topology of the system is rather irregular;
- various software modules as well as a multitasking real-time operating system.

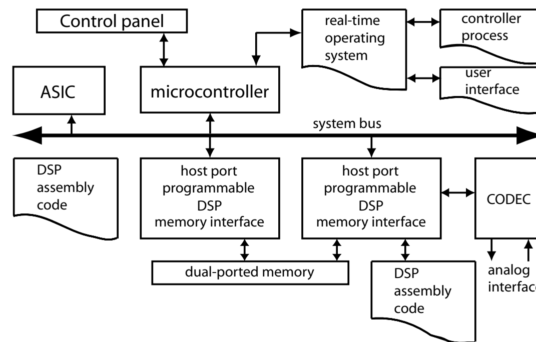

Fig. 1. Signal Processing Embedded Processor Example (from [16]).

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 also **mass-produced** elements separating them from (low-production) application-specific processors. This characteristic imposes a different set of requirements for the embedded processor design, because embedded processor vendors face fierce competition in order to gain more market capitalization. An example requirement involves the cost/performance sensitiveness of embedded processors making low cost almost always an issue. Other related design issues include: high-production volume, small time-to-market window, and fast development cycles.

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

deadlines and soft real-time deadlines. Missing hard deadline can be catastrophic while missing soft deadline only results in some non-fatal glitches. 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 systems connected to the embedded processor since they can negatively influence its behavior.

3 The Need for Programmability

In the early nineties, we were witnessing a trend in the embedded processors market that was reshaping the characteristics of traditional embedded processors as introduced in Section 2. 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. In addition, production and development costs had to be reduced in order to stay competitive. By highlighting the traditional embedded processors design, we discuss ”large scale” programmability[3] which has been used to address these two issues.

The heterogeneity of the embedded systems demanded a multitude embedded processors to be designed for a single system. This was further strengthened by the disability of the semiconductor technology at that time to produce large chips. As a result, the multitude of embedded processors requires lengthy design and verification times, especially for their interfaces. On the other hand, subsequent design cycles could be significantly reduced if only a small number of the embedded processors requires redesign. This delicate balance between long initial design cycles and possibly shortened subsequent design cycles was disturbed when advancing semiconductor technology allowed increasingly more gates to be put on a single chip. Fueled by the need to incorporate increasingly more functionality into (in order to distinguish yourself from competitors) and to decrease the cost of embedded systems, the functionalities of embedded processors were expanded. More complex and larger embedded processors did not decrease the initial design cycles. However, the subsequent redesign cycles were increased, because we are dealing with highly optimized circuits meaning 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 on a single chip[4]. The remaining time-critical function are to be implemented in high-speed circuits achieving maximum performance. The main benefit of this approach is that the large (possibly slower) chip can be reused in subsequent designs resulting in shorter design cycles. Moreover, the large chip also exhibits a more general-purpose behavior and its design becomes more like the design of general-purpose pro-

[3] 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.

[4] Possibly implemented in a slower technology in order to reduce cost.

cessors. The design of general-purpose processors can be divided into three distinct fields [14]: architecture[5], implementation, and realization.

In Section 2, 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 this light, two scenarios in the design of programmable embedded processors can be distinguished:

– **Adapt an existing general-purpose architecture** and implement such an architecture. This scenario reduces development costs albeit such architectures must be licensed. Furthermore, since such architectures were not adapted to embedded processors still some development times 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 specific application the embedded processor is intended for.

Several examples of the first scenarios can be found. A well-known example is the MIPS architecture [4]. In this case, the architecture has been adapted towards embedded processors by MIPS Technologies, Inc. which develops the architecture separately from other embedded systems vendors. Another well-known example is the ARM architecture [1]. It is a RISC architecture that was firstly intended for low-power PCs (1987), but has been quickly adapted to become an embeddable RISC core (1991). Since then the ARM architecture has been subject to numerous modifications and/or extensions in order to optimize it for targeted applications. A well-known implementation is the StrongARM core which was jointly developed by then Digital Semiconductor and ARM Ltd. This core was intended to provide high performance at extreme low-power. The most current implementation of this core developed by Intel Corp. is called the Intel PCA Application Processor [3] intended for PDA handhelds. Other example general-purpose architectures that have been adapted include: IBM PowerPC [2], Sun UltraSPARC [8], the Motorola 68000/Coldfire [5], and many more. An example of the second scenario is the Trimedia VLIW architecture [9] from Trimedia technologies, Inc. which was originally developed by Philips Electronics. Its application area was multimedia processing and can now be found in many televisions, digital receivers, and other digital video editing boards. Figure 2 shows a block diagram of the Trimedia TM-1300 processor. It contains a VLIW processor core that controls the other specialized hardware cores and performs other functions that do not need real-time performance.

Summarizing, the characteristics mentioned in Section 2 can be easily reflected in the three processor design stages (architecture, implementation, and realization). The characteristic of embedded processors being application-specific is exhibited by the

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

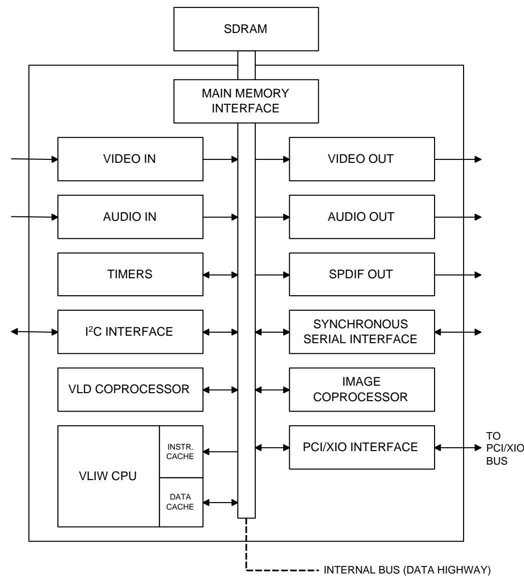

Fig. 2. The Trimedia TM-1300.

fact that the architecture only contains those operations that really need support from the applications set. 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 programmable processor core with other specialized hardware units. In addition, a multitude of different functional units exist on the programmable processor core. The mass-produced characteristic is exhibiting itself in the realization process by only utilizing proven technology that is therefore cheap and reliable. The requirement of real-time processing exhibits itself by requiring architectural support for frequently used operations, extensively parallel (if possible) implementations, and realization incorporating high-speed components.

Finally, a wide variety of design issues also have their impact on the architecture, implementation, and realization of an embedded processor. However, due to the vast variety of design issues, such as cost, performance, cost/performance ratio, high/low production volume, fast development, and small time-to-market windows, we refrain ourselves from discussing these issues in the light of architecture, implementation, realization of embedded processors. However, it must be clear that each design issue has a certain level of impact on the architecture, implementation, and realization of an embedded processor.

4 Early Time Reconfigurability

In the mid-nineties, we were witnessing a second trend in the embedded processors design that was reshaping the design methodology of embedded processors and consequently redefined some of their characteristics. Previously, in the design of embedded

processors application-specific integrated circuits (ASICs) were still commonplace and the design of ASICs required lengthy design cycles. It 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 use reconfigurable hardware, also referred to as fast prototyping. This allows embedded processor designs to be mapped early on in the design cycle to reconfigurable hardware, in particular field-programmable gate arrays (FPGAs), enabling early functionality testing and thereby reducing the number of chip roll-outs. However, such hardware initially were limited in size and therefore only small parts of embedded processor designs could be tested. Consequently, still roll-out(s) of the complete chip (implemented in ASICs) were still required in order to test the overall functionality.

In recent years, the reconfigurable technology has progressed in a fast pace and it has currently arrived at the point that embedded processor designs requiring million(s) of gates can be implemented on such structures. In addition, the performance gap that existed between FPGAs and ASICs is rapidly decreasing. This development in technology has also changed the role of reconfigurable hardware in embedded processors design. Instead of only serving fast prototyping purposes, embedded processors implemented in reconfigurable hardware are actually being shipped in final products. An additional benefit of this development is that bugs found in such embedded processors can be easily rectified resulting in much higher user satisfaction. Furthermore, design improvements can also be easily incorporated during maintenance sessions. In the following, we revisit the embedded processor characteristics mentioned in Section 2 and investigate whether they still hold in case embedded processors are build using FPGAs.

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 hard or it will not achieve the required performance levels.

static structure This characteristic has been affected the most by the utilization of reconfigurable hardware. From a pure technical perspective, the structure of a reconfigurable embedded processor is not static since its functionality can be changed, either during maintenance or during operation. However, we have to consider the frequency of this happening. In most cases, an implementation is chosen for the reconfigurable embedded processor and it is not changed anymore between maintenance intervals. Therefore, from the user's perspective the structure of the embedded processor is still static. In the next section, we will explore the possibility that the functionality of an embedded processor needs to be changed even during operation.

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 [7]) and Xilinx Inc. (Virtex II [10]) integrates on a single chip the following: memory, logic, I/O controllers, and DSP blocks.

mass-produced This characteristic is still applicable to reconfigurable hardware. Early on, reconfigurable hardware has only been used to verify the functionality of design and therefore were not implemented in actual shipped embedded processors. As

the technology progressed, it allowed reconfigurable hardware to be produced at much lower costs and therefore opening the possibility of actually shipping reconfigurable hardware in actual products. This is actually the case at this moment.

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

5 Future Embedded Processors

In Sections 3 and 4, we have shown 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. Programmability allows the utilization of high-level programming languages (like C) and thereby easing application development. Reconfigurability allows designs to be tested early on in terms of functionality and diminishes the need for expensive chip roll-outs. The merging of both strategies in the embedded processor design (if possible) will result in two main advantages. First, the design flexibility is hugely increased, because it allows easy design space exploration in both software and hardware. Second, it allows rapid application development since the software and hardware can be realized utilizing high-level programming and hardware description languages. When correctly incorporated, the combination of programmability and reconfigurability allows embedded processors to change their functionality dynamically during operation (in run-time).

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 [6] and MicroBlaze from Xilinx [11]. However, the utilization of programmable embedded processors that are augmented with reconfigurable hardware also poses several issues that must be addressed:

– **Long reconfiguration latencies:** When considering dynamic run-time reconfigurations, 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 newly introduced instructions greatly increased the complexity of the decoder hardware.

In the following, we discuss one possible approach [28] (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 and not separate instructions for each and every supported operation.
- **Reduced decoder hardware complexity:** Due to the inclusion of only a few instructions, complex instruction decoding hardware is no longer required.

In Section 5.1, we revisit microcode from its beginnings to its current implementation within a high-level microprogrammed machine. In Section 5.2, we discuss in-depth our proposed MOLEN embedded processor. Finally, in Section 5.3, we briefly highlight several other approaches in this field that are comparable in one way or another.

5.1 Revisiting Microcode

Microcode, introduced in 1951 by Wilkes [29], 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 important concept of compatibility. Simply stated, it allowed programs to be written for a specific architecture once and run at "infinitum" independent of the implementations.

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 3 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.

Over the years, the Wilkes' model has evolved into a high-level microprogrammed machine as depicted in Figure 4[6]. In this figure, the control store contains microinstructions (representing one or more micro-operations) and the sequencer determines

[6] 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.

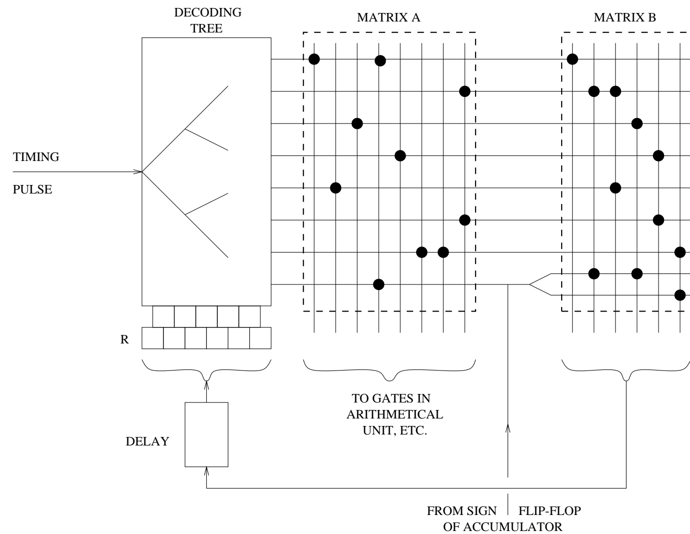

Fig. 3. Wilkes' microprogram control model [29].

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 4). That is, a microcoded engine is as a matter of fact a hybrid of the implementation having emulated instructions and hardwired instructions[7].

5.2 Microcoded Reconfigurable MOLEN Embedded Processor

In this section, only a brief description of the MOLEN embedded processor is given, We refer to [28] for a more detailed description. In its more general form, the proposed machine organization can be described as in Figure 5. In this organization, the

[7] That is, contrary to some believes, from the moment it was possible to implement instructions, microcoded engines always had a hardwired core that executed RISC instructions.

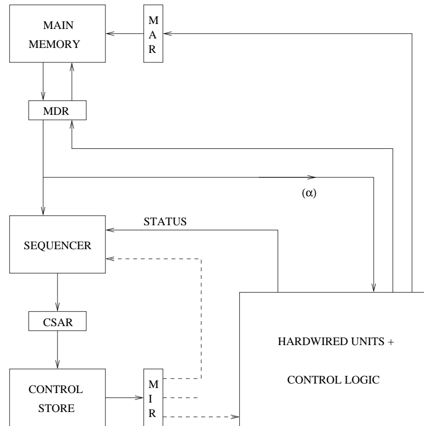

Fig. 4. A high-level microprogrammed machine.

I_BUFFER stores the instructions that are fetched from the memory. Subsequently, the ARBITER performs a partial decoding on these instructions in order to determine where they should be issued. Instructions that have been implemented in fixed hardware are issued to the core processing (CP) unit which further decodes them before sending them to their corresponding functional units. The needed data is fetched from the general-purpose registers (GPRs) and results are written back to the same GPRs. The control register (CR) stores other status information.

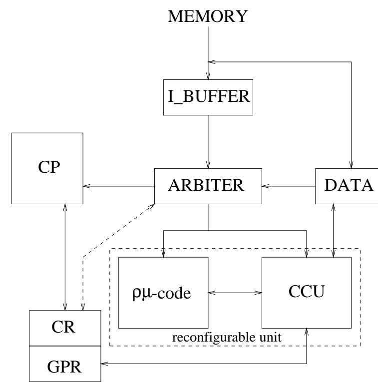

Fig. 5. The proposed machine organization.

The reconfigurable unit consists of a custom configured unit (CCU)[8] and the $\rho\mu$-code unit. An operation[9] performed by the reconfigurable unit is divided into two dis-

[8] Such a unit could be for example implemented by a Field-Programmable Gate Array (FPGA).
[9] An operation can be as simple as an instruction or as complex as a piece of code of a function.

tinct 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[10] (either loaded from memory or resident) in the $\rho\mu$-code unit. 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.

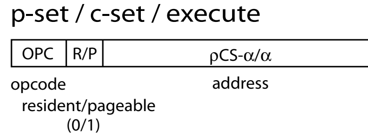

Fig. 6. The *p*-**set**, *c*-**set**, and **execute** instruction formats.

In relation to these three phases, we introduce three new instructions: *c*-**set**, *p*-**set**, and **execute**. Their instruction format is given in Figure 6. 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.

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 7. 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

[10] Reconfiguration microcode is generated by translating a reconfiguration file into microcode.

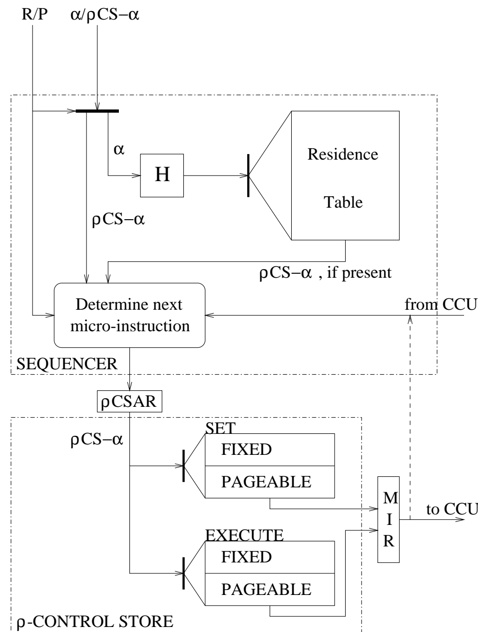

Fig. 7. $\rho\mu$-code unit internal organization.

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[11] support. 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.

[11] For simplicity of discussion, we assume that the system only allows real addressing.

The ρ-CONTROL STORE comprises two sections[12], 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 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.

5.3 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)* [25] 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 [20]: 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 [30] extends PRISC and allows PFU for implementing any combinational or sequential circuits, subject to its size and speed. The system proposed by Trimberger [27] 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

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

not obvious if an explicit SET instruction is available. The *Reconfigurable Multimedia Array Coprocessor* (REMARC) proposed by Miyamori and Olukotun [24] 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 [21] 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[23], 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 [12][13], 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 [13]. From this description, we can conclude that an explicit reconfiguration procedure is available. Gilson [17] 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 [26] 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 [15][18] 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)[19].

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.

6 Conclusions

In this positional paper, we have 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 design and thereby greatly reducing design cycles. Second, reconfigurability was initially only utilized as 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. ARM architecture. http://www.arm.com.
2. IBM PowerPC. http://www-3.ibm.com/chips/products/powerpc/.
3. Intel PCA Application Processors. http://www.intel.com/design/pca/applications processors/index.htm.
4. MIPS architecture from MIPS Technologies. http://www.mips.com.
5. Motrola 68000/Coldfire Family. http://e-www.motorola.com/webapp/sps/site/homepage.jsp?nodeId=03M0ylgrpxN.
6. Nios Embedded Processor. http://www.altera.com/products/devices/excalibur/exc-nios_index.html.
7. Stratix Family. http://www.altera.com/products/devices/stratix/stx-index.jsp.
8. Sun UltraSPARC IIe. http://www.sun.com/microelectronics/UltraSPARC-IIe/index.html.
9. Trimedia VLIW architecture. http://www.trimedia.com.

10. Virtex-II 1.5V FPGA Family: Detailed Functional Description . http://www.xilinx.com/partinfo/databook.htm.
11. Xilinx MicroBlaze. http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=microblaze.
12. P.M. Athanas. *An Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration*. PhD thesis, Brown University, Providence, Rhode Island, May 1992.
13. P.M. Athanas and H.F. Silverman. Processor Reconfiguration through Instruction-Set Metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.
14. G.A. Blaauw and F.P. Brooks. *Computer Architecture: Concepts and Evolution*. Addison-Wesley, 1997.
15. 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.
16. 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.
17. 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.
18. 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.
19. R.W. Hartenstein, R. Kress, and H. Reinig. A New FPGA Architecture for Word-Oriented Datapaths. In *Field-Programmable Logic: Architectures, Synthesis and Applications. 4th International Workshop on Field-Programmable Logic and Applications*, Lecture Notes in Computer Science, pages 144–155, Czech Republic, September 1994.
20. S.A. Hauck. Configuration Prefetch for Single Context Reconfigurable Coprocessors. In *6th International Symp. on Field Programmable Gate Arrays*, pages 65–74, California, 1998.
21. 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.
22. J. Hennessy. The Future of Systems Research. *Computer*, pages 27–33, 1999.
23. 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.
24. 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.
25. R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, Cambridge, Massachusetts, May 1994.
26. H. Schmit. Incremental Reconfiguration for Pipelined Applications. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 47–55, California, April 1997.
27. S.M. Trimberger. Reprogrammable Instruction Set Accelerator. U.S. Patent No. 5,737,631, April 1998.
28. S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN $\rho\mu$-Coded Processor. In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications (FPL2001)*, pages 275–285, 2001.
29. 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.
30. 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.