Microcode Processing: Positioning and Directions

MICROCODE IS AN IMPORTANT INNOVATION IN COMPUTER ENGINEERING. THE AUTHORS DISCUSS THE EVOLUTION OF MICROCODE FROM ITS INTRODUCTION TO ITS DECLINE AND TO ITS LIKELY RESURGENCE IN CUSTOM COMPUTING MACHINES. FURTHERMORE, THEY PRESENT A MICROCODED MACHINE AUGMENTED WITH FIELD-PROGRAMMABLE GATE ARRAYS (FPGAS) AND PROVIDE EXPERIMENTAL EVIDENCE THAT IT CAN SUBSTANTIALLY INCREASE THE PERFORMANCE OF SOME MEDIA BENCHMARKS.

Stamatis Vassiliadis
Stephan Wong
Sorin Cotofana
Delft University of Technology

Microcode, which M.V. Wilkes introduced in 1951, constitutes an important computer engineering innovation.[1] Microcode allowed the emulation of complex instructions through simple hardware (operations) and thus greatly helped drive the development of computer systems. Microcode actually partitioned computer engineering into two distinct conceptual layers: architecture and implementation. Architecture in this article denotes the programmer's view of a system's attributes—such as the processor's conceptual structure and functional behavior. It is distinct from the dataflow's organization and the processor's physical implementation.[2] The partitioning is partly because emulation allowed the definition of complex instructions that might have been technologically unimplementable (at the time they were defined), thus projecting an architecture to the future. That is, microcode let computer architects determine a technology-independent functional behavior (such as the instruction set) and select conceptual structures providing the following possibilities:

- Computer architects could define the computer architecture as a programmer's interface to the hardware rather than to a specific technology-dependent realization of a specific behavior.
- Computer architects could determine a single architecture for a family of implementations, giving rise to the important concept of compatibility. Simply stated, you could write programs for a specific architecture once and run them "ad infinitum" independent of the implementations.

Since its beginnings, microcode, as introduced by Wilkes, has been a sequence of micro-operations, called a microprogram. Such a microprogram comprises pulses for operating the gates associated with arithmetical and control registers. Figure 1 depicts the method of generating this sequence of pulses.[1] First, a timing pulse initiating a micro-operation enters the decoding tree that in turn generates an output signal depending on setup register R. This output signal passes to matrix

0272-1732/03/$17.00 © 2003 IEEE Published by the IEEE computer Society

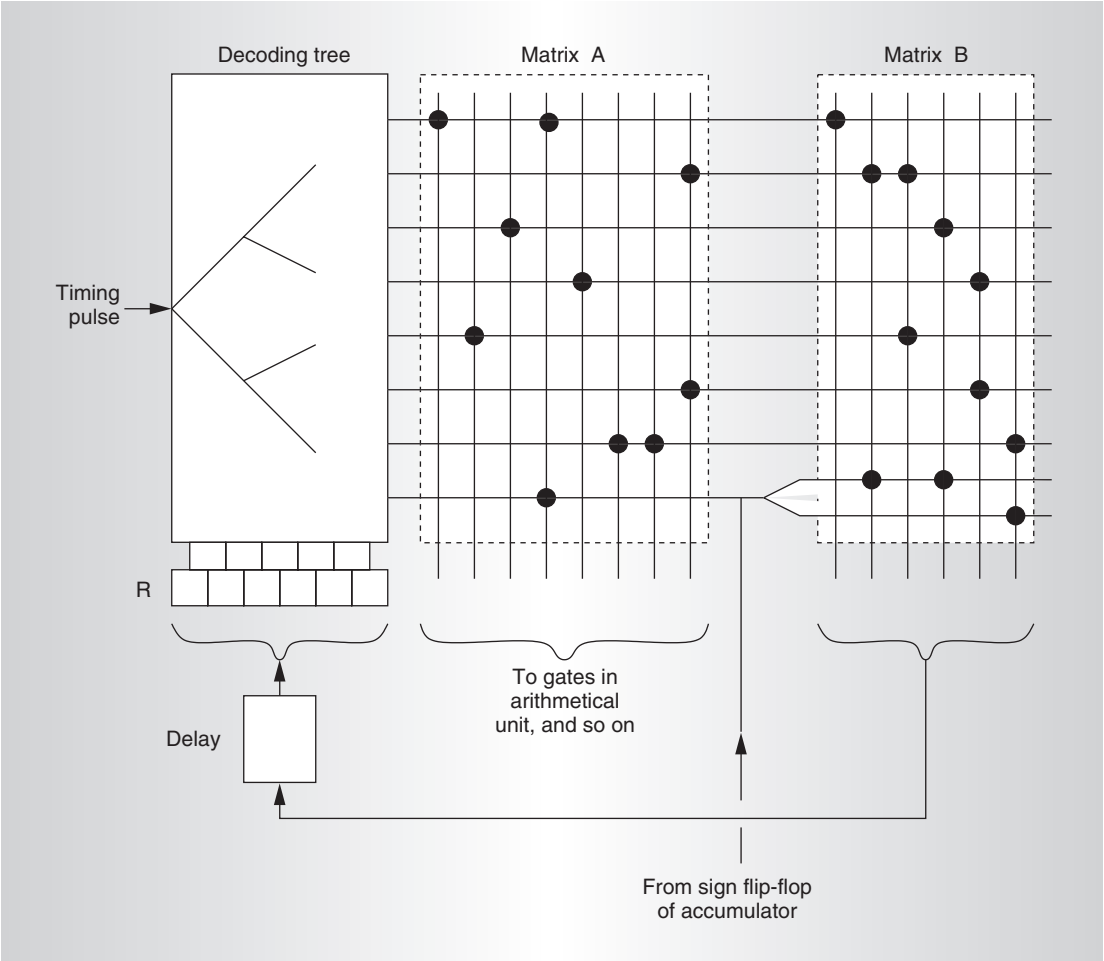

Figure 1. Wilkes' microprogram control model.

Table 1. Some facts from two IBM enterprise servers.

System	No. of assembly languages	No. of higher languages	Modules	Lines of code	Lines of code plus comments
IBM ES/4381	3	1	1,505	480,692	791,696
IBM ES/9370	6	2	3,130	796,136	1,512,750
Total	9	3	4,635	1,276,828	2,304,446

A, which in turn generates pulses to control arithmetical and control units and thus perform the required micro-operation. The output signal also passes to matrix B, which in its turn generates pulses to control and update setup register R (with a certain delay). The next timing pulse, therefore, generates the next micro-operation in the required sequence due to the updated setup register R.

Microcode has become a major component, requiring a large development effort, from mainframes to PC processors. To understand the magnitude and the importance that microcode has played, consider some key facts of "real" machine microcode depicted in Table 1.[2] As you can observe, several assembly and higher-level languages have been developed and used, with a substantial amount of code developed for the implementations.[3] This is indicated by the number of modules used, the number of lines of code, and the number of lines of code plus comments.

Microcode has evolved considerably from its initial structure. In this article, we look at

current architectures and briefly discuss some technological advances that somehow diminished the microcode appeal and other technological advances that could reintroduce its usefulness.

Modern microcode concepts

Here we discuss concepts associated with the microcode structure and some organizational techniques that microcode employs. Figure 2 depicts a high-level microprogrammed computer.[4]

The control store in Figure 2 contains the microinstructions (which represent one or more micro-operations) and corresponds to matrices A and B in Figure 1. The machine's operation is as follows:

1. The control store address register (CSAR) contains the address of the next microinstruction in the control store. The microinstruction there is then forwarded to the microinstruction register (MIR).
2. The MIR decodes the microinstruction and generates smaller micro-operations that the hardwired units or control logic must perform.
3. The sequencer uses status information from the control logic or results from the hardwired units to determine the next microinstruction and stores its control store address in the CSAR. The previous microinstruction could also influence the sequencer's decision about which microinstruction to select next.

As mentioned earlier, the MIR generates micro-operations depending on the microinstruction. If only one micro-operation is generated that controls a single hardwired resource, the microinstruction is called *vertical*. In all other cases, the microinstruction is called *horizontal*. Finally, the control store could provide permanent (fixed) storage to frequently used microprograms (a sequence of microinstructions) and temporary (pageable) storage for less-frequently-used microprograms. In microcoded machines, not all instructions have or will have access to the control store. In fact, only emulated instructions have to go through the microcode logic; the hardware can directly execute all other instructions that the system has implemented, following path α in Figure 2.

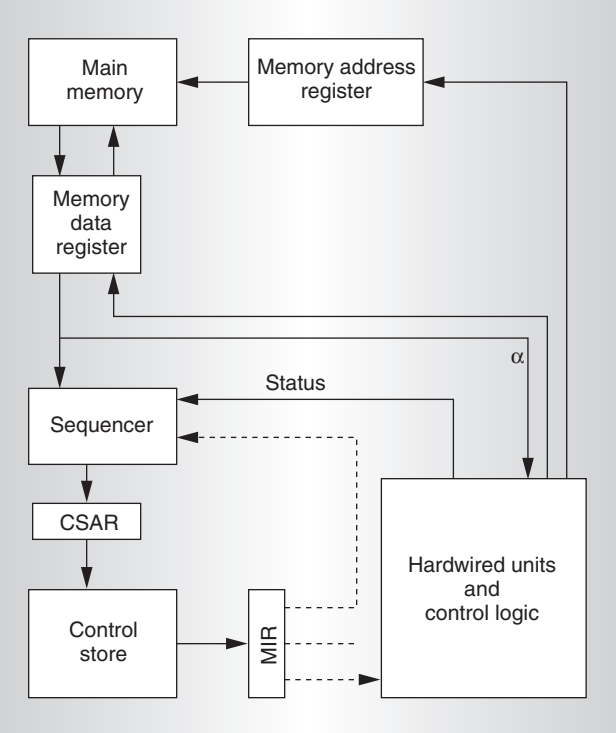

Figure 2. High-level microprogrammed computer.

That is, a microcoded machine is a hybrid with emulated and hardwired instructions. Contrary to some beliefs, from the moment it was possible to implement instructions, microcoded machines always had a hardwired core that executed reduced-instruction-set computer (RISC) instructions.

Microcode advantages?

Why does an extremely successful mechanism appear to be in decline, and what is happening to the advantages of microcoded machines? To examine this question, we explore a simple example. We consider the requirements imposed by moving one or more characters (represented by bytes) from one memory location to another. Because the generic case is too complex and will not serve any purpose to our discussion, we further assume that at most 256 bytes are to be moved and that the source and destination addresses are always aligned at word boundaries. The generic case requires many more test and boundary conditions that will only increase the code sizes of the examples.

This example is by no means casual, because it relates to the move character fami-

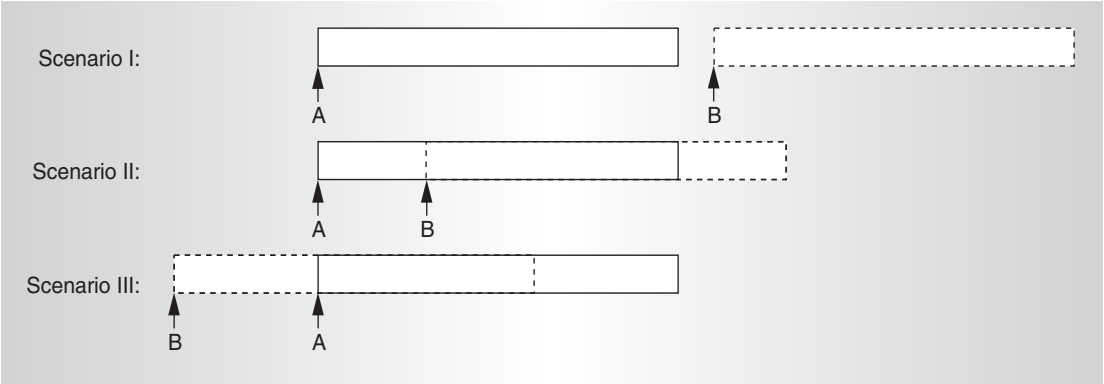

Figure 3. Three scenarios of moving characters.

Table 2. Sample program for case 2.

Label	Instruction	Label	Instruction
start:	lw r8, A	overlap:	add r14, r8, r10
	lw r9, B		add r15, r9, r10
	lw r10, d	word_aligned?:	sll r13, r10, 30
	beq r8, r9, stop		beq r13, r0, yes_aligned
	sub r11, r9, r8		sub r14, r14, 1
	bltz r11, no_overlap		sub r15, r15, 1
	sub r12, r11, r10		lb r12, 0(r14)
	bltz r12, overlap		sb 0(r15), r12
no_overlap:	sub r11, r10, 4		sub r10, r10, 1
	bltz r11, last_bytes		beq r10, r0, stop
	lw r12, 0(r8)		jmp word_aligned?
	sw r12, 0(r9)	yes_aligned:	sub r14, r14, 4
	add r8, r8, 4		sub r15, r15, 4
	add r9, r9, 4		lw r12, 0(r14)
	sub r10, r10, 4		sw r12, 0(r15)
	jmp no_overlap		sub r10, r10, 4
last_bytes:	beq r10, r0, stop		beq r10, r0, stop
	lb r12, 0(r8)		jmp yes_aligned
	sb 0(r9), r12	stop:	
	add r8, r8, 1		
	add r9, r9, 1		
	sub r10, r0, 1		
	jmp last_bytes		

ly of instructions, which includes the controversial complex-instruction-set computer (CISC) operation, move character long.

Figure 3 depicts three potential scenarios. The first shows no overlap between the string to be moved and the moved string. In this scenario, moving characters does not pose any problem.

In the second scenario, however, moving the first, (then second, and so on) characters in a straightforward manner will overwrite the string to be moved. The third scenario also has an overlap, but because the starting point B is in front of A, it does not pose any problems when moving the characters in a straightforward manner.

The described move character operation is an application requirement and is what the programmer intended to do. Thus, it has no relation to any architectural paradigm. The intention of the move instruction is to actually perform a "true" move. That is, the programmer does not intend to replicate bytes using move instructions and overlap. There are several ways of performing such an operation with varying involvement of software and hardware.

Case 1. The first approach is to introduce a complex instruction and assume that the hardware somehow will execute it. The instruction has the format: MOVE R1, R2, d, where R1 is the source address, R2 is the destination address, and d is the number of bytes to be moved.

Case 2. A second approach is to assume simple instructions, and produce a program like the one shown in Table 2.[5]

Table 3. Sample program for case 3.*

Label	Load/store slot	Integer slot 1	Integer slot 2	Branch slot
start:	lw r8, A			
	lw r9, B			
	lw r10, d	sub r11, r9, r8		beq r8, r9, stop
		sub r11, r10, 4	sub r12, r 11, r10	bltz r11, no_overlap
		add r14, r8, r10	add r15, r9, r10	bltz r12, overlap
no_overlap:	lw r12, 0 (r8)			bltz r11, last_bytes
	sw r12, 0(r9)	add r8, r8, 4	add r9, r9, 4	
		sub r10, r10, 4	sub r11, r10, 8	jmp no_overlap
last_bytes:	lb r12, 0(r8)			beq r10, r0, stop
	sb 0(r9), r12	add r8, r8, 1	add r9, r9, 1	
		sub r10, r10, 1		jmp last_bytes
overlap:		sll r13, r10, 30		
word_aligned?:				beq r13, r0, yes_aligned
		sub r14, r14, 1	sub r15, r15, 1	
	lb r12, 0(r14)	sub r10, r10, 1		
	sb 0(r15), r12			beq r10, r0, stop
		s11 r13, r10, 30		jmp word_aligned?
yes_aligned:		sub r14, r14, 4	sub r15, r15, 4	
	lw r12, 0(r14)	sub r10, r10, 4		
	sw r12, 0(r15)			beq r10, r0, stop
				jmp yes_aligned
stop:				

* This program does not execute floating-point instructions; therefore, this table does not show the floating-point instruction slot.

Case 3. A third approach is to decide on an architecture that will allow combining multiple instructions and executing them in parallel.[6,7,8] You could then rewrite the move character program as in Table 3. The program assumes five instruction slots for different instructions: one slot for load/store instructions, two slots for integer instructions, one slot for floating-point instructions, and one slot for branch instructions. Furthermore, we assume that the arithmetic units produce their results in one cycle.

Case 4. If we find (using simulations) that in most of the cases (say 80 percent) a programmer is moving 8 bytes without overlap, then we can rewrite the move character program as in Table 4.

Case 5. With the same assumptions of the architecture in case 3, the program from case 4 can be rewritten as in Table 5 (on p. 27).

The differences among all the cases are the hardware involvement, the software involvement, their complexity, performance, and cost (in terms of area). The trained eye will recognize that case 1 is what has been called CISC architecture. Cases 2 and 4 are RISC architectures with programming tricks to improve performance, and cases 3 and 5 are very long instruction word (VLIW) architectures with the same considerations as for the RISC architectures.

Microcode design principles

What might not be so obvious is that cases 2 through 5 are microcoded sequences for the complex instruction that case 1 describes. To comprehend this, you need to understand how a microcoded machine is actually designed. The design principles of such a machine are as follows:

- *Principle 1.* We design (and always have designed) hardware that is simple, meaning it is implementable using current technologies and meeting certain con-

Table 4. Sample program for case 4.

Label	Instruction	Label	Instruction
start:	lw r8, A	overlap:	add r14, r8, r10
	lw r9, B		add r15, r9, r10
	lw r10, d	word_aligned?:	sll r13, r10, 30
	beq r8, r9, stop		beq r13, r0, yes_aligned
	sub r11, r9, r8		sub r14, r14, 1
	bltz r11, no_overlap		sub r15, r15, 1
	sub r12, r11, r10		lb r12, 0(r14)
	bltz r12, overlap		sub 0(r15), r12
no_overlap:	sub r11, r10, 8		sub r10, r10, 1
	beq r11, r0, 8bytes		beq r10, r0, stop
no_overlap2:	sub r11, r10, 4		jmp word_aligned?
	bltz r11, last_bytes	yes_aligned:	sub r14, r14, 4
	lw r12, 0(r8)		sub r15, r15, 4
	sw r12, 0(r9)		lw r12, 0(r14)
	add r8, r8, 4		sw r12, 0(r15)
	add r9, r9, 4		sub r10, r10, 4
	sub r10, r10, 4		beq r10, r0, stop
	jmp no_overlap2		jmp yes_aligned
last_bytes:	beq r10, r0, stop	8bytes:	lw r12, 0(r8)
	lb r12, 0(r8)		sw r12, 0(r9)
	sb 0(r9), r12		lw r12, 4(r8)
	add r8, r8, 1		sw r12, 4(r9)
	add r9, r9, 1	stop:	
	sub r10, r0, 1		
	jmp last_bytes		

straints (such as cycle, cost, or area).

- *Principle 2.* We emulate and thus microcode the instructions that are nonimplementable or are not frequently used.
- *Principle 3.* We permanently cache the microcode (emulation code) of certain, frequently used, nonimplementable instructions (or instances of) in the control store to improve performance. We cache all other microcode corresponding to non-frequently-used nonimplementable instructions (or instances of) on demand with appropriate replacement policies.
- *Principle 4.* We write microcode to either control a single facility (vertical microcode) or control multiple facilities (horizontal microcode).

Going back to our cases, if we assume that we can implement all instructions in case 2 (principle 1), then the program in case 2 emulates case 1 and we can store it in the control store; thus, it is its microcode (principle 2). We can store it permanently—or not, depending on simulation evaluations (principle 3)—and because we assume that the program only executes one instruction at a time, we have a vertical microcode (principle 4). We can apply the same principles to case 3 with the addition that this case is a horizontal microcoded machine, because multiple facilities are controlled in a single cycle. Cases 4 and 5 are just different ways of performing the same emulation. In essence, they do move characters but are geared toward improving the performance of the most frequent cases. So they possibly split the microcode into permanently and nonpermanently stored emulation code to compact the control store memory. To put things in perspective, by eliminating instructions that require emulation and by exposing vertical microcode to the programmer, we have a RISC architecture. When the exposure involves horizontal microcode, we have a VLIW architecture.

We are ready to consider the developments

Table 5. Sample program for case 5.

Label	Load/store slot	Integer slot 1	Integer slot 2	Branch slot
start:	lw r8, A			
	lw r9, B			
	lw r10, d	sub r11, r9, r8		beq r8, r9, stop
		sub r11, r10, 4	sub r12, r11, r10	bltz r11, no_overlap
		add r14, r8, r10	add r15, r9, r10	bltz r12, overlap
no_overlap:		sub r16, r10, 8		
	lw r12, 0(r8)			beq r16, r0, 8bytes
no_overlap2:	lw r12,0(r8)			bltz r11, last_bytes
	sw r12, 0(r9)	add r8, r8, 4	add r9, r9, 4	
		sub r10, r10, 4	sub r11, r10, 8	jmp no_overlap2
last_bytes:	lb r12, 0(r8)			beq r10, r0, stop
	sb 0(r9), r12	add r8, r8, 1	add r9, r9, 1	
		sub r10, r10, 1		jmp last_bytes
overlap:		sll r13, r10, 30		
word_aligned?:				beq r13, r0, yes_aligned
		sub r14, r14, 1	sub r15, r15, 1	
	lb r12, 0(r14)	sub r10, r10, 1		
	sb 0(r15),r12			beq r10, r0, stop
		sll r13, r10, 30		jmp word_aligned?
yes_aligned:		sub r14, r14, 4	sub r15, r15, 4	
	lw r12, 0(r14)	sub r10, r10, 4		
	sw r12, 0(r15)			beq r10, r0, stop
				jmp yes_aligned
8bytes:	sw r12, 0(r9)			
	lw r12, 4(r8)			
	sw r12, r(r9)			
stop:				

* This program does not execute floating-point instructions; therefore, this table does not show the floating-point instruction slot.

leading to the proposal that eliminates instructions requiring emulations (at least from the processor's point of view). First of all, technological improvements allow more complex instructions to be implemented in hardware. Examples are multiply, divide, and square root. All these operations were microcoded in one time period or another. Consequently, any instruction that we can implement in hardware does not need emulation; thus, we have no need for microcode. The implication here is that, to a certain extent, improvements in technology have diminished the microcode's effectiveness. Furthermore, the programmers did not frequently use instructions that were complex, had side effects, or were difficult to comprehend, making their elimination from the architecture desirable and thus further diminishing the microcode's appeal.

The event of caching was the second strike against microcode's effectiveness for general-purpose computers. We have added caches to the processor to improve performance of memory accesses. The bottom line is, a sequence of instructions accomplishing a complex task (captured by a CISC instruction as in case 1) that resides frequently (almost permanently) in caches requiring few access cycles does not need to reside in the control store. Thus, the CISC instruction needs no emulation by a microcoded machine. Instead, a sequence of simple instructions (no microcode) can emulate the complex task.

Is microcode a thing of the past?

At first glance, it appears that microcode is on its way out. Maybe so. However, some circumstances may indicate otherwise. First, while

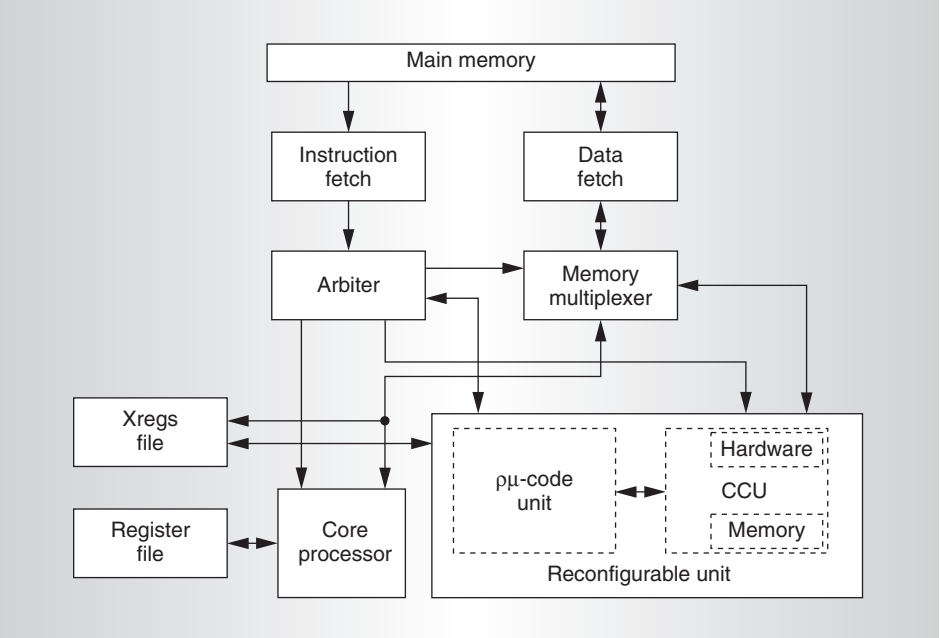

Figure 4. Reconfigurable microcoded Molen processor.

technologies allow increasingly more instructions to be implementable, several instructions (such as start I/O instructions) cannot possibly be implemented in a hardwired mode. Designers have implemented I/O instructions, for example, using multiple levels of emulation (that is, using multiple levels of microcode, frequently called picocode). As far as we can see, microcode for machines that require I/O processors or processing mostly keep or change the microcode name, but not the substance. The second reason relates to the rise of custom computing machines. Such machines compute complex tasks and have the hardware changing via reconfiguration. Given that reconfiguration is a complex function that executes by using a "long" program, special caching is necessary to reduce the reconfiguration times, reintroducing the microcode concept. Because reconfigurations are complex behaviors, covering all of them with one hardware implementation is not possible. Therefore, emulation (and thus microcode) is needed (principle 2). Furthermore, caching as we know it today may not be capable of handling large and frequent reconfigurations until reused. The implication here is the reintroduction of special storage and techniques to hold reconfigurations resembling control store, possibly with modifications, to accommodate special reconfiguration needs. We could achieve this by pointing to emulation code (microcode) that either resides on chip (frequently used) or in main memory, and is loaded in caches like control store when needed (less frequently used)—principle 3. Because of the configurable nature of the hardware, we can configure it to contain many subunits (facilities) to optimally perform the functions by exploiting parallelism. In this case, depending on the available parallelism, we can use either vertical or horizontal microcode (principle 4). Such microcode constructions have begun to appear. In this article, we consider an augmented general-purpose paradigm with reconfigurable microcode.[9] Figure 4 depicts the organization of the Molen reconfigurable processor.

The Molen processor fetches instructions from the main memory and stores them in the instruction fetch unit. The arbiter fetches instructions from the instruction fetch unit and performs a partial decoding on the instructions to determine where they should be issued. Instructions implemented in fixed hardware are issued to the core processor. The core processor further decodes the instructions and issues them to their corresponding functional units. The (core) processor fetches the source data from the general-purpose registers (GPRs) and writes the results back to the same GPRs.

Table 6. Media results of the Molen processor from simulations (ML = execution cycles).

Input data	ijpeg encoder		mpeg2enc
	Picture testimg (No. of cycles)	Picture vigo (No. of cycles)	Frames testframes (No. of cycles)
Default cycles	6,512,947	149,363,055	93,245,923
DCT ML = 282	4,680,431 (–28.14%)	109,815,770 (–26.48%)	81,560,420 (–12.53%)
VLC ML = 200	6,094,054 (–6.43%)	140,486,670 (–5.94%)	
SAD ML = 234			74,608,673 (–19.99%)
DCT(282) + VLC(200)	4,505,811 (–30.82%)	106,157,895 (–28.93%)	
DCT(282) + SAD(234)			62,928,429 (–32.51%)

The reconfigurable unit consists of a custom configured unit (CCU)—possibly implemented by using a field-programmable gate array (FPGA)—and the ρμ-code unit that encompasses the functionality of the sequencer and control store as depicted in Figure 2. The reconfigurable unit is able to perform operations that can be as simple as a single instruction or as complex as a piece of application code that describes a certain function. Therefore, the processor divides an operation that the reconfigurable unit executes into two distinct process phases: set and execute. The set phase is responsible for reconfiguring the CCU hardware, enabling the execution of the operation. We may subdivide the set phase into two subphases: partial set (p-set) and complete set (c-set). We envision the p-set phase covering common functions of an application or set of applications. More specifically, in the p-set phase, the CCU is partially configured to perform these common functions. Although the system can possibly perform the p-set subphase during program loading or even at chip fabrication time, it performs the c-set subphase during program execution. Furthermore, the c-set subphase only partially reconfigures remaining blocks in the CCU (not covered in the p-set subphase) to complete the CCU's functionality by enabling it to perform other less-frequent functions. We use the exchange registers (XRegs) file to provide a general methodology to pass arguments to and results from the reconfigurable unit.

Finally, we performed simulations of the Molen processor using two well-known multimedia benchmarks (ijpeg and mpeg2enc) and the SimpleScalar toolset (v2.0).[10] We could configure the CCU to implementations that perform the discrete cosine transform (DCT), the Huffman coding (a variable length coding technique denoted by VLC), or the calculation of the sum of absolute differences (SAD).[11] In the simulations, we assume that an FPGA structure (an Altera APEX20K) operating at a considerably lower core frequency (indicated by synthesis software FPGA Express from Synopsys) augments a 1-GHz general-purpose processor. Due to this, we normalized the cycles to perform DCT, VLC, and SAD to reflect this discrepancy.[9] As Table 6 shows, the simulation results have shown an average overall decrease in cycles by about 30 percent.

We have shown two scenarios that indicate microcode's current usefulness. We believe that microcode use will continue in the future, especially in the advent of the increasing use and acceptance of reconfigurable computing in many new application areas, such as network processing. In this light, the challenge lies in semiautomatically determining the most adequate design parameters, such as the size of the fixed and pageable storage within the control store and the microinstruction width, to meet the requirements posed by new application areas. Another challenge lies in defining a common reconfigurable microcode that we can use to configure and to execute on various reconfigurable hardware implementations. In conclusion, the introduction of microcode use in custom computing machines is opening up new research and engineering challenges. MICRO

References

1. M.V. Wilkes, "The Best Way to Design an Automatic Calculating Machine," *Proc. Manchester Univ. Computer Inaugural Conf.*, Ferranti Ltd., 1951, pp. 16-18.

2. A. Padegs et al., "The IBM System/370 Vector Architecture: Design Considerations," *IEEE Trans. Computers*, vol. 37, no. 5, May 1988, pp. 509-520.
3. G. Triantafyllos, S. Vassiliadis, and J. Delgado-Frias, "Software Metrics and Microcode Development: A Case Study," *J. Software Maintenance: Research and Practice*, vol. 8, no. 3, May-June 1996, pp. 199-224.
4. G. Tomlinson and P. Adams, "Microprogramming: A Tutorial and Survey of Recent Developments," *IEEE Trans. Computers*, vol. 29, no. 1, Jan. 1980, pp. 2-20.
5. G. Kane and J. Heinrich, *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, 1992.
6. S. Vassiliadis, B. Blaner, and R. Eickemeyer, "SCISM: A Scalable Compound Instruction Set Machine," *IBM J. Research and Development*, vol. 38, no. 1, Jan. 1994, pp. 59-78.
7. *TriMedia32 Architecture* (preliminary specification), TriMedia Technologies, 2000.
8. *Intel IA64 Architecture: Software Developer's Manual*, Intel Corp., 2000.
9. S. Vassiliadis, S. Wong, and S. Cotofana, "The MOLEN $\rho\mu$-coded Processor," *Proc. 11th Int'l Conf. Field-Programmable Logic and Applications* (FPL 2001), Lecture Notes in Computer Science, vol. 2147, Springer-Verlag, 2001, pp. 275-285.
10. D.C. Burger and T.M. Austin, *The SimpleScalar Tool Set, Version 2.0*, tech. report CS-TR-1997-1342, Univ. of Wisconsin-Madison, 1997.
11. S. Vassiliadis et al., "The Sum-Absolute-Difference Motion Estimation Accelerator," *Proc. 24th Euromicro Conf.*, 1998, pp. 158-163.

Stamatis Vassiliadis is the chair professor of the Computer Engineering Laboratory in the Electrical Engineering Department at Delft University of Technology, Delft, the Netherlands. His research interests include embedded systems, multimedia processors, computer architecture, hardware design of computer systems, custom computing machines, hardware/software co-design, computer arithmetic, low-power design, nanocomputing, and network processing. Vassiliadis has a PhD in electrical engineering from Politecnico di Milano, Italy. He is an IEEE Fellow and a member of the IEEE Computer Society.

Stephan Wong is an assistant professor with the Computer Engineering Laboratory in the Electrical Engineering Department at Delft University of Technology, Delft, the Netherlands. His research interests include embedded systems, multimedia processors, complex instruction set architectures, reconfigurable and parallel processing, microcoded machines, and network processors. Wong has a PhD in computer engineering from Delft University of Technology, Delft, the Netherlands.

Sorin Cotofana is an associate professor with the Computer Engineering Laboratory in the Electrical Engineering Department at Delft University of Technology, Delft, the Netherlands. His research interests include computer arithmetic, parallel architectures, embedded systems, neural networks, fuzzy logic, computational geometry, and computer-aided design. Cotofana has a PhD in electrical engineering from Delft University of Technology, the Netherlands. He is a senior member of the IEEE Computer Society.

Direct questions and comments about this article to Stephan Wong, Computer Engineering Laboratory, Electrical Engineering Department, Delft University of Technology, Delft, the Netherlands; j.s.s.m.wong@ewi.tudelft.nl.