Microcoded Reconfigurable Embedded Processors: Current Developments

Stephan Wong, Stamatis Vassiliadis, Sorin Cotofana

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

Abstract. It is well-known that the main disadvantages associated with reconfigurable hardware are long reconfiguration latencies, high opcode space requirements, and complex decoder hardware. To overcome these disadvantages, we use microcode since it allows emulation of "complex" operations which are performed using a sequence of smaller and simpler operations. Microcode is used to control the reconfiguration of the reconfigurable hardware, either online or off-line, and the execution on the reconfigurable hardware. Due to the multitude of microcodes and their sizes, it is not feasible to provide on-chip storage for all microcodes. Consequently, the loading of microcode into a limited on-chip storage facility is becoming increasingly more important. In this paper, we present two methods of loading microcodes into such an on-chip storage facility.

1 Introduction

Traditionally, embedded processor design was very much similar to microcontroller design. This meant that for each targeted set of applications, an embedded processor was designed in specialized hardware (commonly referred to as Application Specific Integrated Circuits (ASICs)). In the early nineties, we were witnessing a shift in the embedded processor design approach fueled by the need for faster time-to-market. This resulted in the utilization of programmable processor cores that were augmented with specialized hardware units, which were still implemented in ASICs. This meant that time-critical tasks were implemented in specialized hardware units while other less time-critical tasks were implemented in software to be run on the programmable processor core [1]. This approach allows a programmable processor core to be re-used for different sets of applications without redesigning it. This leaves only the specialized hardware units to be (re-)designed for the targeted set of applications.

Currently, we are witnessing a new trend in embedded processor design that is again quickly reshaping the embedded processor design approach. Time-critical tasks are now being implemented in field-programmable gate array (FPGA) structures or comparative technologies [2–5] instead of ASICs. The reasons for and the benefits of such an approach include the following:

– **Increased flexibility:** The functionality of the embedded processor can be quickly changed and allows early implementation of future functionalities not specified at design time.

- **Good-enough performance:** The performance of FPGAs has increased tremendously and is quickly approaching that of ASICs [6]. Furthermore, due to the mostly high-level specification of embedded processor designs (usually in VHDL), existing designs can be mapped on new FPGA structures much faster and fully exploit the performance benefits they provide.
- **Faster design times:** Faster design times are achieved by re-using intellectual property (IP) cores or by slightly modifying them. Design faults can be quickly identified and rectified without the need for additonal embedded processor roll-outs.

The above mentioned benefits have even resulted in that programmable processor cores are under consideration to be implemented in the same FPGA structures, e.g., [7]. However, the utilization of such reconfigurable hardware also poses several issues that must be addressed:

- **Long reconfiguration latencies:** This is the time it takes for the reconfigurable hardware to change its functionality. 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 utilization of a programmable processor core that also controls the reconfigurable hardware has resulted in the fact that many instructions (usually associated with a single operation) must be introduced that initiate and control the reconfiguration and execution processes. Extending the functionality of the FPGA structure now 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 this paper, we discuss a new approach using microcode that alleviates 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 paper, 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:** The usage of microcode to control the reconfiguration process allows such code to be cached inside the embedded processor. This allows faster access times to the reconfiguration microcode and thus in turn reduces the reconfiguration latencies.
- **Reduced opcode space requirements:** As will be explained later, the usage of microcode requires only the inclusion of several instructions that point to such microcode. By executing microcode, we perform the required operation(s). Therefore, there is no need for separate instructions for each and every supported operation.
- **Reduced decoder hardware complexity:** Due to the inclusion of only a few instructions (that point to the microcode), we do not require complex (instruction) decoding hardware.

An important aspect when utilizing microcodes is how to load them into an on-chip storage facility. This paper proposes two simple microcode loading methods. The first method loads microcode in a straightforward manner by overwriting already loaded microcodes. The second method does this by taking into account which microcodes are frequently used and thus only overwriting less frequently used microcodes.

This paper is organized as follows. Section 2 discusses the concept of microcode and shows an example of how microcode can be utilized to support reconfiguration of reconfigurable hardware and the execution on such hardware. Section 3 discusses in short the MOLEN reconfigurable microcoded processor [8] which utilizes microcode to also support partial reconfigurability. In Section 4, we discuss two methods of how microcodes can be loaded into an on-chip storage facility. Finally, Section 5 concludes this paper with some closing remarks.

2 What is microcode?

Microcode, introduced in 1951 by Wilkes [9], constitutes one of the key computer engineering innovations. Microcode allowed the emulation of “complex” instructions by means of “simple” hardware (operations) and thus provided a prime force to the development of computer systems as we know them today. Microcode de facto partitioned computer engineering into two distinct conceptual layers, namely: architecture[1] 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 1 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.

[1] Architecture here and in the rest of the presentation denotes the attribute of a system as seen by the programmer, i.e., the conceptual structure and functional behavior of the processor, and it is distinct from the organization of the dataflow and physical implementation of the processor [10].

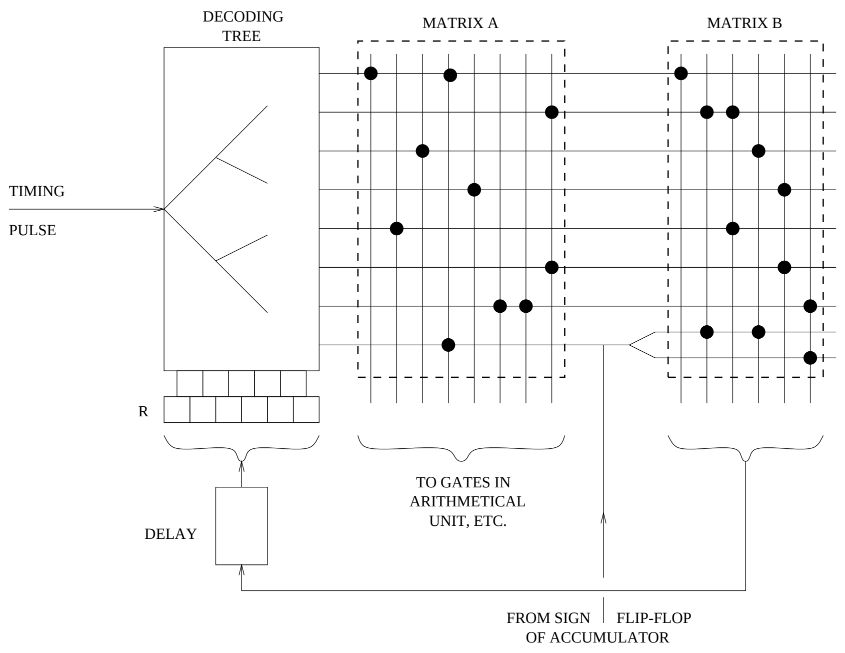
DECODING TREE
MATRIX A
MATRIX B
TIMING
PULSE
R
DELAY
TO GATES IN ARITHMETICAL UNIT, ETC.
FROM SIGN FLIP-FLOP OF ACCUMULATOR

Fig. 1. Wilkes' microprogram control model [9].

Microcode has become a major component requiring a large development effort from mainframes to PC processors. To illustrate the magnitude and the importance that microcode has played, we present some key facts of "real" machine microcode in Table 2 [10]. In this table, it can be observed that several languages have been developed/used (3, 6 assembler languages and 1, 2 higher level languages for the 4381, 9370 respectively) with a substantial amount of microcode being developed for the implementations. This is indicated by the number of modules used, the number of lines of actual microcode (LOC), and the number of lines of actual microcode together with comments (DLOC).

system	# of assembler languages	# of higher languages	modules	LOC	DLOC
IBM ES/4381	3	1	1,505	480,692	791,696
IBM ES/9370	6	2	3,130	796,136	1,512,750

Table 1. Some facts from two IBM Enterprise Servers [11].

Over the years, the Wilkes' model has evolved into a high-level microprogrammed machine as depicted in Figure 2. In this figure, the control store contains the microinstructions (that represent one or more micro-operations) and the sequencer determines

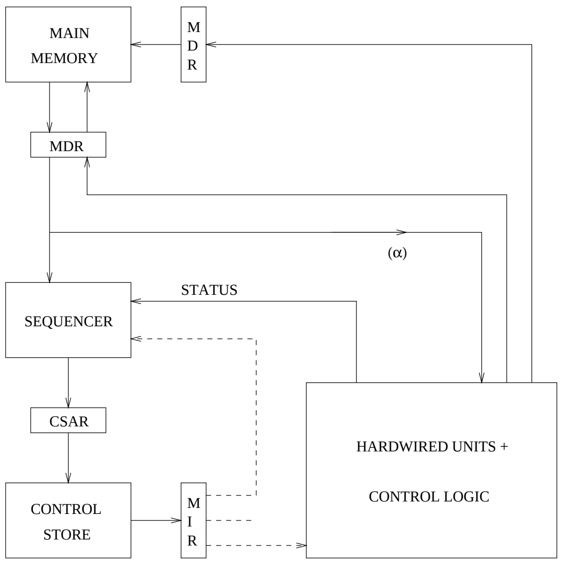

Fig. 2. A high-level microprogrammed machine.

the next microinstruction to execute. The control store and the sequencer correspond to the matrices A and B respectively in the Wilkes' model. 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.

As mentioned before, the MIR generates micro-operation(s) depending on the microinstruction. In the case that only one micro-operation is generated controlling a single hardwired resource, the microinstruction is called vertical. In all other cases, the microinstruction is called horizontal. The execution of the microinstructions stops whenever the *end_op* microinstruction is encountered.

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 that have been implemented will be executed directly by the hardware (following path (α) in Figure 2). That is, a microcoded engine is as a matter of fact a hybrid of the implementation having emulated instructions and hardwired instructions[2].

2.1 A microcode example

In this section, we discuss how to perform a 64-bit signed magnitude multiplication. Obviously, a single 64-bit signed multiplier can be implemented in hardware to perform the required operation. However, in this example we have elected to perform the operation using four 32-bit unsigned multipliers. This example shows that operations exhibiting common functionalities will greatly benefit from our approach. We must note that this example only serves illustration purposes and our approach is in no way limited to such "small" structures. In this example, we assume that the scheme has to be implemented in an FPGA and discuss how the reconfiguration process of the FPGA can be performed using microcode. Finally, we discuss how microcode can also be used to control the execution process on the FPGA.

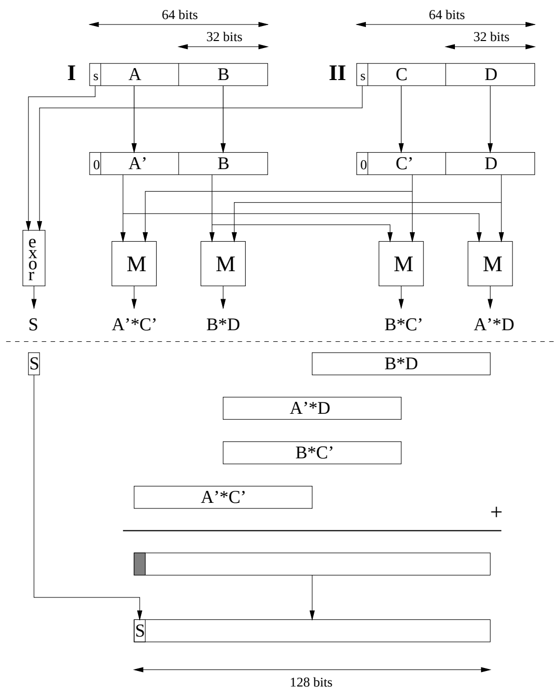

Fig. 3. Performing 64-bit multiplication using four 32-bit unsigned multipliers.

A signed magnitude multiplication of two 64-bit numbers I and II can be performed using four 32-bit unsigned multipliers as follows (see Figure 3):

[2] 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.

1. Substitute the sign bits of both numbers I and II with zeroes.
2. Perform an exor over the two sign bits. This determines the sign of the final result.
3. Perform four 32-bit unsigned multiplications on the 32-bit parts of both remaining numbers.
4. Add the intermediate results of the four multiplications.
5. Substitute the most significant bit of the result after addition with the new sign bit.

The aforementioned multiplication can be placed inside an FPGA by writing a VHDL model, synthesizing it to obtain a reconfiguration file, and then loading the resulting reconfiguration file into the FPGA. In most of the traditional approaches, each reconfiguration file was associated with a reconfiguration instruction and by issuing such an instruction, a reconfiguration of the FPGA was performed. These approaches require continuous architectural extensions, i.e., adding new reconfiguration instructions, for new functions implemented on the FPGA. Consequently, it puts a lot of strain on the opcode space of any instruction set. We propose to use microcode to control the reconfiguration process and show later on that only one new instruction is needed for this purpose.

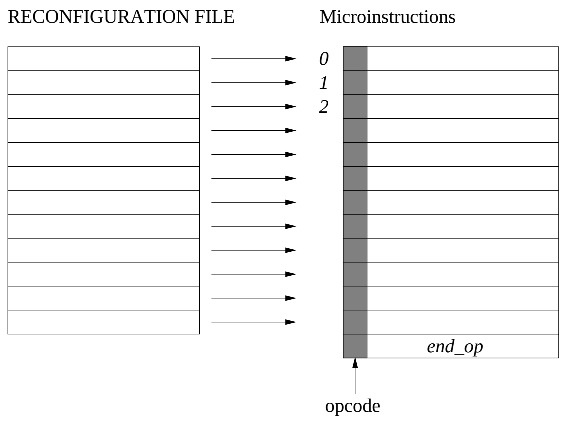

Fig. 4. A straightforward translation from reconfiguration file to reconfiguration microprogram.

A straightforward method that translates a reconfiguration file into a reconfiguration microprogram is depicted in Figure 4. A reconfiguration file is sliced into smaller pieces of equal size that will constitute the latter part of a microinstruction. The (microinstruction) opcode identifies the microinstruction as containing reconfiguration file slices. The *end_op* microinstruction signifies the end of the reconfiguration microprogram. The reconfiguration of the FPGA can still be performed using existing hardware simply by ignoring the microinstruction opcodes. We are aware that many techniques, e.g., encoding or compression of microinstruction fields, can be applied to reduce the reconfiguration microprogram size, but they do not conceptually add anything to this discussion and are therefore left out of this discussion.

Having obtained a reconfiguration microprogram, we can mark the sections responsible for reconfiguring certain parts of the FPGA. We show a higher level representation of the reconfiguration microprogram in Table 2 since showing it at microinstruction level does not provide any insight.

setup datapaths from register file towards input register
setup the input registers
setup the substitution datapaths
setup the intermediate registers
setup datapaths towards multipliers
setup multiplier 1
setup multiplier 2
setup multiplier 3
setup multiplier 4
setup the exor
setup datapaths towards addition structure
setup the addition structure
setup the pre-result register
setup datapath from exor towards end result register
setup end result register
setup datapath from result register towards register file
end_op

Table 2. A high-level reconfiguration microprogram

Instead of introducing new instructions for each and every reconfiguration microprogram, we introduce a *single* instruction (called SET) that points to the location of the first microinstruction of the reconfiguration microprogram. By executing the microprogram starting from this location till the end_op microinstruction, the reconfiguration is performed. The SET instruction format is as follows:

SET, resident/pageable, location address; *execute the reconfiguration microcode starting from* 'location address'.*Continue execution until the* end_op *microinstruction is encountered.*

The meaning of 'resident/pageable' and how it effects the interpretation of the 'location address' field is discussed later. Similar to using microcode to control the reconfiguration process, we can also use microcode to control the execution process of the implementation that is currently loaded into the FPGA. For example, an execution microprogram of the multiplication depicted in Figure 3 is given in Table 3. Again, we introduce a single instruction (called EXECUTE) that points to the location of the first microinstruction of the (now) execution microcode. The instruction format is as follows:

EXECUTE, resident/pageable, location address; *execute the execution microcode from* 'location address'. *Continue execution until the* end_op *microinstruction is encountered.*

load	the input values from register file
substitute	the sign bits with zeroes
exor	the two sign bits
multiply	sub-fields
add	the remaining intermediate results
substitute	the most significant bit with new sign bit
end_op	

Table 3. A high-level execution microprogram

The most obvious storage facility for all the microprograms (either reconfiguration or execution) is the main memory. However, the loading of microprograms can have a tremendous effect on the overall performance, because no computations can start before the FPGA is configured. Assuming a straightforward translation of a reconfiguration file into a reconfiguration microprogram, the following example gives a good estimate of the load latency of an average-sized reconfiguration file:

– Assume that it takes approximately 1.5 reconfiguration byte to reconfigure one gate in an FPGA, it would require 1.5 Mbytes to reconfigure 1 million gates of an average-sized Xilinx Virtex-II FPGA.
– Assume that we use a high-speed RAMBUS memory to store such a reconfiguration file and that the memory bandwidth is 3.2 GByte/s, it would require about 0.5 milliseconds to load the reconfiguration file.
– Assuming that the FPGA is tied to a 1 GHz processor, the 0.5 milliseconds of load latency translates into 500.000 processor cycles in which the processor has to wait before the reconfiguration can even start.

A load latency that corresponds to 500.000 lost processor cycles is huge and it is obvious that it must be reduced. There are two ways to reduce the load latencies. First, frequently used microprograms can be stored on-chip (*resident*) close to the FPGA with the control store (see Figure 2 and assume that the hardwired units have been replaced with FPGA units) being the most obvious location. Second, other less frequently used microprograms are stored in the main memory and can be cached (*pageable*) on-chip. Due to the close proximity of the microprograms to the FPGA, load latencies can be greatly reduced. Both methods are supported by the SET and EXECUTE instructions in which the 'resident/pageable'-field determine the interpretation of the 'location address'-field, i.e., as a control store address or as a memory address. The second method is slightly more complicated than the first one, because it encompasses the translation of memory addresses into control store addresses (CS-α). Section 4 discusses in more detail the loading (paging) of microprograms into the control store and several related issues. Before we do that, we shortly present the MOLEN reconfigurable microcoded processor that utilize microcode to support reconfigurable hardware that can also be partailly reconfigured.

3 The MOLEN reconfigurable microcoded processor

In the previous section, we discussed the concept of microcode and microcoded processor organizations. Furthermore, we presented an example of performing 64-bit signed magnitude multiplication using four 32-bit unsigned multipliers. In this example, we introduced the utilization of microcode in both the reconfiguration and execution processes. Recently, we have introduced the MOLEN reconfigurable microcoded processor that is utilizing microcode to incorporate architectural support for reconfigurable hardware structures such as FPGAs [8]. Its internal organization is depicted in Figure 5.

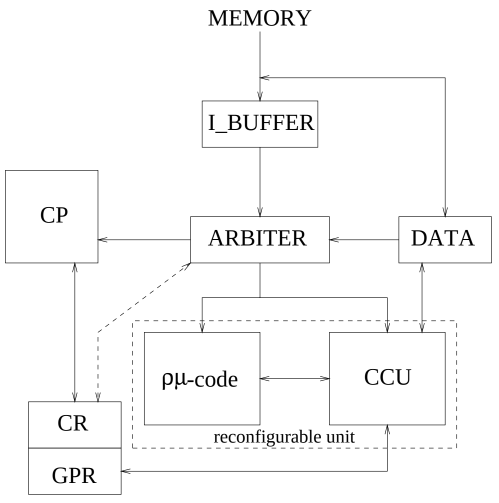

Fig. 5. The internal organization of the MOLEN processor.

In this organization, instructions are fetched from the memory and stored in the instruction buffer (I_BUFFER). The ARBITER fetches instructions from the I_BUFFER and performs a partial decoding on the 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. The instructions entering the CP unit are further decoded and then issued to their corresponding functional units. The source data are fetched from the general-purpose registers (GPRs) and the results are written back to the same GPRs. Other status information are stored in the control registers (CRs).

The reconfigurable unit consists of a custom configured unit (CCU)[3] and the $\rho\mu$-code unit (shown in Figure 6 as the combination of the sequencer and the ρ-CONTROL STORE). An operation[4] executed by the reconfigurable unit is divided into two distinct process phases: **set** and **execute**. The **set** phase is responsible for reconfiguring the CCU hardware enabling the execution of the operation. Such a phase may be subdivided into two subphases, namely partial **set** (*p*-**set**) and complete **set** (*c*-**set**). The *p*-**set**

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

phase is envisioned to cover 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. 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. Furthermore, the *c*-**set** sub-phase only partially reconfigures remaining blocks in the CCU (not covered in the *p*-**set** sub-phase) in order to *complete* the functionality of the CCU by enabling it to perform other less frequent functions.

For the reconfiguration of the CCU, reconfiguration microcode[5] within the $\rho\mu$-code unit (either already resident or loaded from main memory) is executed to perform the actual reconfiguration. It can be noted that in case the reconfigurable logic cannot be partially set or in case the partial setting is not convenient then the *c*-**set** can be used by itself to perform the entire configuration. The **execute** phase is responsible for the actual execution of the operation on the CCU.

In the **execute** phase, the operation is performed by executing execution microcode within the $\rho\mu$-code unit (either already resident or loaded from main memory). By executing the execution microcode on the CCU (already configured to the needed implementation), the desired operation is performed. We must note that both the **set** and **execute** phases do not specify a certain operation that needs to be performed and then execute the corresponding reconfiguration or execution microcode. Instead, the *p*-**set**, *c*-**set**, and **execute** instructions directly point to the location where the reconfiguration or execution microcode is stored. In this way, different operations are performed by executing different reconfiguration microcodes and different execution microcodes, respectively. That is, instead of specifying new instructions for the operations (requiring instruction opcode space), we simply point to location addresses. We have shown that the location of the microcode can be either on-chip (called *resident*) or in the main memory (called *pageable*).

Summarizing, the MOLEN reconfigurable microcoded processor utilizes three new instructions to support reconfiguration of and execution on reconfigurable hardware. The combination of p-**set** and c-**set** allows even the support for partial (run-time) reconfigurations. Furthermore, by separating support for any operation into two distinct phases, the **set** and **execute** phases, allows reconfiguration latencies to be hidden when they are scheduled far apart from each other. Finally, by keeping microcode close to the CCU inside the $\rho\mu$-code unit, we can greatly reduce load latencies of microcodes. In the next section, we present two methods of how to load microcode that is stored in the main memory into the $\rho\mu$-code unit.

4 Loading of microprograms

As mentioned before, our approach provides permanent on-chip storage for frequently used microprograms[6] and temporary on-chip storage for less frequently used microprograms that must be loaded from the main memory. In the traditional microcoded

[5] The reconfiguration microcode is generated by translating the needed reconfiguration file into reconfiguration microcode (see Section 2.1).

[6] In this section, we do not distinguish between reconfiguration microcode and execution microcode. They are conceptually the same and are referred to in this section as microprograms.

machine (see Figure 2), resident microprograms were stored inside the control store. In our approach, we utilize a modified control store, called the reconfigurable control store (ρ-CONTROL STORE), to store both frequently used and less frequently used microprograms. The ρ-CONTROL STORE is discussed in Section 4.1, which also introduces the problem of loading microprogram into the ρ-CONTROL STORE. This is followed by a discussion of two microprogram loading methods that utilize the ρ-CONTROL STORE in Sections 4.2 and 4.3.

4.1 The reconfigurable control store

The reconfigurable control store (ρ-CONTROL STORE) together with an extended sequencer is depicted in Figure 6.

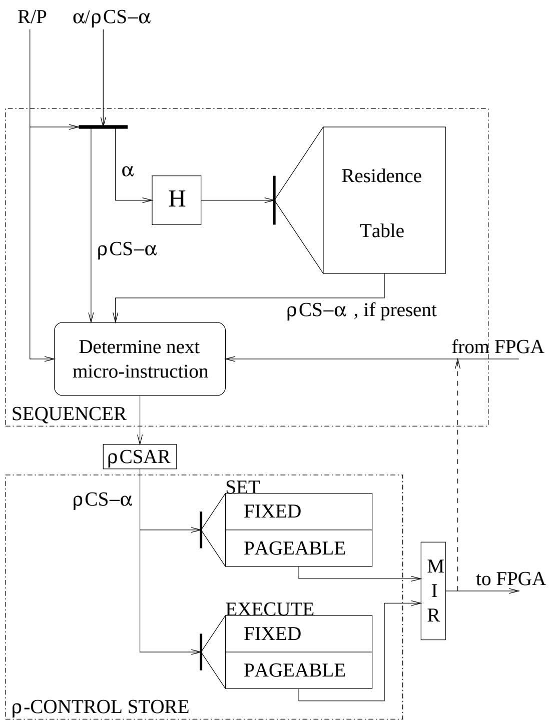

Fig. 6. Internal organization of the ρ-CONTROL STORE and the sequencer.

Both the sequencer and the ρ-CONTROL STORE operate as described in Section 2 with the following differences:

– The residence table contains the most recent translations of memory addresses (α) into ρ-CONTROL STORE addresses (ρCS-α). The actual translation is performed by a hashing function (H) since the memory address space is much larger than the one of the ρ-CONTROL STORE.

– The residence/pageable (R/P) bit determines the interpretation of the 'location address'-field of both the SET and EXECUTE instructions.
– The basic addressable unit inside the ρ-CONTROL STORE is a module which consists of at least one microinstruction up to several hundreds of microinstructions. Since a microprogram may consist of several modules (possibly of varying size), we assume for simplicity's sake that every block within the ρ-CONTROL STORE can store the largest known module.

Two problems might occur when loading microprograms. The first problem occurs when a microprogram exceeds the available storage space. The second problem occurs when several microprograms must be loaded, but their combined size exceeds the available storage space. The first problem can be solved by loading a smaller set of modules that constitute the larger microprogram. When considering separate modules as distinct microprograms, we can translate the first problem into the second one. The second problem can be solved by swapping out currently loaded microprograms in order to create storage space for the to be loaded microprogram. In this paper, we propose two methods to solve the two mentioned problems. The first method (called *straight*) loads the microprogram starting from a certain point in the ρ-CONTROL STORE and continues until the end is reached. All remaining (non-loaded) modules must be accessed from the main memory. The second method (called *evict*) tries to create enough storage space inside the ρ-CONTROL STORE when remaining storage space is too small for the to be loaded microprogram.

4.2 The *straight* loading method

In this section, we discuss the *straight* microprogram loading method. Before discussing the method, we assume the following

– Each block in the ρ-CONTROL STORE has a $first$ bit which tells whether the block contains the first microinstruction (and thus the first module) of a microprogram or not.
– Each block in the ρ-CONTROL STORE is indexed by the memory address of the first microinstruction of the module that is loaded into the block in question.
– Jumps (conditional or direct) inside the modules only jump to the first instruction of modules.

Considering the case that we want to load microprogram A for the first time into the ρ-CONTROL STORE and the first instruction of microprogram has the memory address α, the *straight* method works as follows:

1. Determine using address α which entry in the residence table to use. Each residence table entry is associated with a ρ-CONTROL STORE block.
2. Start loading the microprogram into this block and mark the block with the $first$ bit set to one. The remaining modules of the microprogram are written into the consecutive blocks and their $first$ bits are all set to zero. This is done until the end of the storage space is reached or the *end_op* microinstruction is encountered.

The execution of the microprogram starts by determining the correct ρ-CONTROL STORE block using the residence table. The actual execution can only start if the corresponding $first$ bit of the block is set and continues until:

- a jump microinstruction is encountered;
- the *end_op* microinstruction is encountered;
- another block with the $first$ bit set is encountered;
- the end of the ρ-CONTROL STORE is encountered.

In the first scenario, the jump to memory address must be compared to the indices of the ρ-CONTROL STORE in order to find the correct next block to execute. In the latter two situations, the remaining modules of the microprogram must be executed from the main memory since no mechanisms are present to remedy these situations. In both situations the performance is greatly penalized due to increased loading latencies. In addition, there are two scenarios during the execution of a microprogram that can increase the occurence of the latter two situations:

- Microprograms are loaded into the ρ-CONTROL STORE that overwrites frequently used microprograms. This might result in many reloadings of the frequently used microprograms (when blocks with the $first$ bit set to one are overwritten) or the execution of modules from the main memory (when latter parts are overwritten).
- The hashing function favors blocks at the end of the ρ-CONTROL STORE. This results in that many microprograms will not be loaded in their entirety and thus requiring the execution of non-loaded modules from the main memory.

The first scenario is difficult to avoid since the starting location of a microprogram within the ρ-CONTROL STORE is very much determined by the hashing function. In order to avoid the first scenario, further research must be performed on how to define the hashing function given a certain set of microprograms. The second scenario can be more easily avoided using the following two enhancements:

- An additional $fail$ counter (initialized at zero) is added to all the residence table entries. Every time that a microprogram has to be loaded starting from a certain block location and it fails, the $fail$ counter of the corresponding residence table entry is incremented with a constant that is greater than one. The next time the same microprogram must be loaded, the microprogram is loaded into $fail$ block locations higher than indicated by the residence table entry. In this way, the chance of a microprogram not being loaded in its entirety will decrease.
- Allow microprograms that reach the end of the ρ-CONTROL STORE to wrap around and continue loading from the first block location.

4.3 The *evict* loading method

In the previous section, we have discussed the *straight* method of loading microprograms that does not take into account that some microprograms might be used more frequently than others. In this section, we propose the *evict* loading method which evicts less frequently used microprograms in order to create more storage space when loading a new microprogram. Again, the same assumptions as for the *straight* method apply here. The *evict* microprogram loading method is based on the following concept:

1. Determine the next possible empty block to start loading the microprogram by:
   - determining the first unused block in the ρ-CONTROL STORE.
   - if this is not possible, determine the least recently used block.
2. Load the first module into the indicated block until:
   - an *end_op* instruction is encountered in the microprogram. In this case, stop loading.
3. Consider the remainder of the microprogram as a new microprogram with the module following the lastly loaded module as the first module of the new microprogram. Continue with step 1.

While the concept remains simple, the actual implementation of the *evict* method is much more complex. This is due to the fact that the location of microprogram modules in the main memory is addressed by the memory address of the first microinstruction of such modules. Therefore, when a microprogram is split into several pieces inside the ρ-CONTROL STORE, the memory addresses of the first microinstruction within the modules must be kept. Furthermore, when executing jumps within a microprogram, additional accesses to the residence table must be performed to determine the correct ρ-CONTROL STORE block to jump to. In addition, another table next to the residence table must be introduced, which is called the occupance table. The following describes what information must be kept inside both tables:

- OCCUPANCE TABLE: This table is addressed by the ρ-CONTROL STORE address and for each address one bit is kept. This bit indicates whether the associated ρ-CONTROL STORE location is used by a microprogram or not.
- RESIDENCE TABLE: This table is addressed by hashed memory addresses. Each entry contains *at least* the following information:
  - LRU: this contains the least recently used information.
  - α: memory address. This address indicates from which memory location the microprogram was fetched.
  - ρCS-α:ρ-CONTROL STORE address. This address indicates to which ρ-CONTROL STORE position the microprogram is storing the first module of the microprogram.
  - α_{next}: next memory address. This address indicates where to jump to in case that the loaded microprogram does not contain the *end_op* instruction.

In order to further explain the concept, we discuss an example which loads a microprogram and also executes this microprogram. The following assumptions are made:

- The microprogram has a length of 10 modules (starting with module 1). The memory addresses of the modules i are α_i.
- Module 4 contains a conditional branch to module 9.
- Module 10 contains a conditional branch to module 5.

Loading the microprogram

The loading of the microprogram is performed as follows:

1. Determine the next possible block into which we can start loading. This is assumed to be ρCS-α_i.
2. Start loading the microprogram into the ρ-CONTROL STORE in block ρCS-α_i.
3. The occupance table must be updated to reflect the used/occupied block. Furthermore, in the residence table an entry is made at location HASH(α_i) which contains at least the following information:
   – α_i: the memory address of the first microinstruction of the microprogram.
   – ρCS-α:the ρ-CONTROL STORE block address.
   – $\alpha_{next} = \alpha_{i+1}$: The memory address of the microinstruction which starts the remainder of the microprogram, if exist.
4. If the *end_op* microinstruction was not encountered, determine the next possible block to continue loading of current microprogram now starting with memory address α_{i+1}. Go to step 1.

(WRITE SOMETHING TO SUMMARIZE)

Executing the microprogram

The execution starts by performing a hash function on the memory address in order to determine the residence table entry containing the ρ-CONTROL STORE address. The execution of the microprogram is not fully described as no specifics on the microprogram were given. However, some key points during execution (especially considering the conditional branches) are discussed:

1. After executing module 5, module 6 must be executed. Since module 6 is greater than m_{last}, it must be stored in one of the following sections (in this case, there are only two sections). Using the α_{next} address, the entry in the residence table is accessed to find that the starting block of the next section is (ρCS($\alpha + 10$)). By calculating the difference between the module number and m_{last} (i.e., $6 - 5 = 1$), the correct module can be found by adding 'difference-1' to ρCS($\alpha + 10$). The '-1' takes into account that jumping from section to another implicitly means jumping at least one module.
2. If the conditional branch in module 4 decides to jump forward, the next module to be executed is 9. Using the residence table, we can lookup α_9 and find ρCS-α_i.
3. If the conditional branch in module 10 decides to jump backwards, the next module to be executed is module 5. Using the residence table, we can lookup α_9 and find ρCS-α_5.

There is a performance limiting scenarios. There is a small penalty associated with jumping from one section to another since the residence table must be accessed.

5 Conclusion

In this paper, we discuss the usage of microcode to control both the reconfiguration process of reconfigurable hardware and the execution process on such hardware. Furthermore, we suggest that only two new instructions are needed in order to support reconfigurable hardware at the architectural level. We have also presented the MOLEN reconfigurable microcoded processor that utilize three new instructions in order to include support for partial reconfigurations. Finally, this paper also discussed the loading of microprogram from the main memory into an on-chip storage facility (the ρ-CONTROL STORE). This is, because it is not practical to store all microprograms on-chip. Two microprogram loading methods, *straight* and *evict*, were proposed and discussed in more detail.

References

1. S. Rathnam and G. Slavenburg, "An Architectural Overview of the Programmable Multimedia Processor, TM-1," in *Proceedings of the COMPCON '96*, pp. 319–326, 1996.
2. D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths," in *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, pp. 23–40, March 1999.
3. R. Razdan and M. Smith, "A High-Performance Microarchitecture with hardware-programmable Functional Units," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pp. 172–180, November 1994.
4. R. Wittig and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 126–135, April 1996.
5. J. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proceedings of the IEEE Symposium of Field-Programmable Custom Computing Machines*, pp. 24–33, April 1997.
6. "Virtex-II 1.5V FPGA Family: Detailed Functional Description ." http://www.xilinx.com/partinfo/databook.htm.
7. "Nios Embedded Processor." http://www.altera.com/products/devices/excalibur/exc-nios_index.html.
8. S. Vassiliadis, S. Wong, and S. Cotofana, "The MOLEN $\rho\mu$-Coded Processor," in *Proceedings of the 11th Internal Conference on Field-Programmable Logic and Applications (FPL2001)*, pp. 275–285, August 2001.
9. M. V. Wilkes, "The Best Way to Design an Automatic Calculating Machine," in *Report of the Manchester University Computer Inaugural Conference*, pp. 16–18, July 1951.
10. A. Padegs, B. Moore, R. Smith, and W. Buchholz, "The IBM System/370 Vector Architecture: Design Considerations," *IEEE Transactions on Computers*, vol. 37, pp. 509–520, May 1988.
11. G. Triantafyllos, S. Vassiliadis, and J. Delgado-Frias, "Software Metrics and Microcode Development: A Case Study," *Journal of Software Maintenance: Research and Practice*, vol. 8, pp. 199–224, May-June 1996.