Loading $\rho\mu-$code: Design Considerations

G.Kuzmanov, G.N. Gaydadjiev, S. Vassiliadis

Computer Engineering Laboratory, Electrical Engineering Dept., EEMCS, TU Delft, The Netherlands
E-mail : {G.Kuzmanov,G.N.Gaydadjiev,S.Vassiliadis}@ET.TUDelft.NL
http://ce.et.tudelft.nl/

***Abstract*— This article investigates microcode generation, finalization and loading in MOLEN $\rho\mu$ processors. In addition, general solutions for these issues are presented and implementation for Xilinx Virtex-II Pro platform FPGA is introduced.**

Keywords: Reconfigurable architectures, MOLEN, implementation, loading microcode.

I. Introduction

Reconfigurable hardware extensions of general purpose processors (GPP) have indicated considerable potentials for speed-ups of computationally demanding algorithms. Numerous design concepts and organizations have been proposed to support the Custom Computing Machine (CCM) paradigm from different prospectives [1]–[4]. An example of a detailed classification of CCMs can be found in [5]. Recently, the MOLEN $\rho\mu-$ processors for CCM organizations have been proposed [6]. The MOLEN concept provides a flexible and easily extendable framework for hardware/software co-design of complex computing systems by extending the traditional microcode. The presented paper addresses some specific issues related to the microcode design and maintenance within the MOLEN processors. More specifically, we investigate the problems related to the generation, memory alignment and loading of configuration microcodes.

Hereafter, the discussion is organized as follows. Section II gives a brief background on the MOLEN organization. Section III introduces the FPGA configuration format for the targeted Xilinx technology. In Section IV, problems related to generation, alignment and loading of reconfigurable microcodes are discussed. Section V proposes solutions to different problems with respect to efficient hardware implementations. Finally, the discussion is concluded in Section VI.

II. The MOLEN organization

This section presents the MOLEN $\rho\mu$-coded Custom Computing Machine organization, introduced in [6] and illustrated in Figure 1. The ARBITER performs a partial decoding on the input instructions flow in order to determine where they should be issued. The arbiter controls the proper co-processing of the GPP and the reconfigurable units. Figure 2 depicts a general design of a MOLEN arbiter. It is closely connected to three major components of the CCM: the GPP, the memory and the $\rho\mu$-unit. Instructions implemented in fixed hardware are issued to the core processor (GPP). Instructions for custom execution are redirected to the *reconfigurable unit*, referred to as *$\rho\mu$-unit*. The reconfigurable unit consists of a custom computing unit

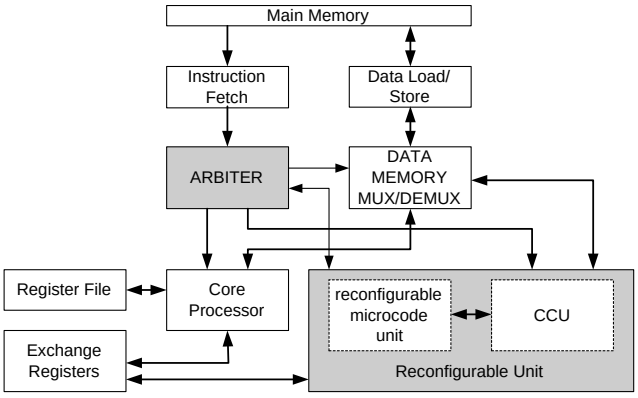

Fig. 1. The MOLEN machine organization

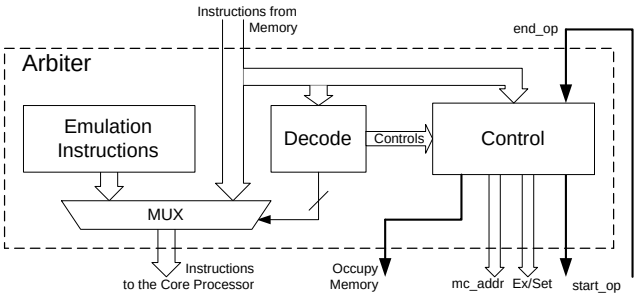

Fig. 2. General view of the Arbiter

(CCU) and the $\rho\mu$-code unit. An operation, executed by the reconfigurable unit, is divided into two distinct phases: **set** and **execute**. The **set** phase is responsible for reconfiguring the CCU hardware enabling the execution of the operation. This phase may be divided into two subphases - partial set (**pset**) and complete set (**cset**). In the **pset** phase the CCU is partially configured to perform common functions of an application (or group of applications). Later, the **cset** subphase only reconfigures those blocks in the CCU, which are not covered in the **pset** sub-phase in order to *complete* the functionality of the CCU.

To perform the actual reconfiguration of the CCU, reconfiguration microcode is loaded into the $\rho\mu$-code unit and then executed. The **execute** phase is responsible for the actual operation execution on the CCU, performed by running the *execution microcode*. It is important to emphasize that both the **set** and **execute** phases do not specify a certain operation to be performed. Instead, the **pset**, **cset** and **execute** instructions

(*reconfigurable instructions*) directly point to the memory location where the reconfiguration or execution microcode is stored. The microcode engine is extended with mechanisms that allow permanent and pageable reconfiguration and execution code to coexist.

III. FPGA CONFIGURATION MICROCODE

The reconfiguration files generated after synthesis, contain random bit patterns and will highly depend on the targeted FPGA technology. They contain the configuration commands and configuration data needed to configure the different FPGA resources, e.g., switch boxes, interconnect resources, look-up tables and any additional technology dependent information. Usually a configuration file can be considered as a stream of bits and is often referred to as *configuration bitstream* in the literature. Such bit streams are produced by the FPGA synthesis tool, e.g. Synopsis, Xilinx, Altera or Lattice. As it can be assumed, a pre-defined and widely accepted standard for such binary streams does not exists and different vendors use the most convenient format for their technology. It is very important to understand that the same high-level hardware description file will result in complete different configuration bitstreams when different technologies are targeted.

The first target for MOLEN implementation is the Virtex-II Pro [7] FPGA from Xilinx. Virtex II Pro devices incorporate one up to four PowerPC 405 GPP cores, FPGA reconfigurable hardware, dedicated RAM blocks and dedicated high-speed I/O blocks. The FPGA fabric is similar to Virtex II. Although the Virtex II and Virtex II Pro devices are not bitstream compatible, the same considerations hold true for both types and will be referred to as *V2* from now on. V2 devices are organized in columns corresponding to the column organization of the FPGA's logic resources [8]. In other words the V2 configuration memory can be visualized as a rectangular array of bits, grouped in vertical *frames* that are one-bit wide and go from the top of the array to the bottom. The frame is the atomic unit of configuration, this is the smallest piece of the configuration memory that can be written (or read). Such organization allows partial reconfiguration that can be performed with or without shutting down the device. The partial reconfiguration is a very important option, since configuration stream size, and hence the loading time, strongly depends on the targeted device, e.g XC2V1000 incorporates 1104 configuration frames, 3392 bits per frame or 3,744,768 configuration bits in total, while XC2V10000 has 3212 configuration frames, 10432 bits per frame (33,507,584 bits). It is obvious that full reconfiguration of XC2V10000 will be ten times longer than XC2V1000 using identical programming conditions (interface type and clock). Vitrex II Pro sizes and download times for the parallel slave mode assuming a programming clock frequency of 50MHz are depicted in Table I. A full reconfiguration that takes roughly 48 milliseconds (as for XC2VP50) may be prohibitive in many real-time applications. A reconfiguration of a single frame (a very likely scenario), however, takes about 18 microseconds that would be acceptable.

An example configuration bitstream looks as follows:

TABLE I
VIRTEX II PRO SIZES AND PROGRAMMING TIMES (50MHZ PAR. MODE)

Device	No. of Frames	No. of bits	Config. time
XC2VP2	884	1,305,440	3.26 ms
XC2VP7	1,320	4.484,472	11.21 ms
XC2VP20	1,756	8,214,624	20.54 ms
XC2VP50	2,628	19,005,696	47.55 ms

```
Dummy word                                FFFF FFFFh
Synchronization word                      AA99 5566h
Packet Header: Write to CMD register      3000 8001h
Packet Data: RCRC                         0000 0007h
Packet Header: Write to FLR register      3001 6001h
Packet Data: Frame Length                 0000 00--h
Packet Header: Write to COR               3001 2001h
Packet Data: Configuration options        ---- ----h
Packet Header: Write to MASK              3000 C001h
Packet Data: CTL mask                     0000 0000h
Packet Header: Write to CMD register      3000 8001h
Packet Data: SWITCH                       0000 0009h
Packet Header: Write to FAR register      3000 2001h
Packet Data: Frame address                0000 0000h
Packet Header: Write to CMD register      3000 8001h
Packet Data: WCFG                         0000 0001h
```

In the above example the first set of commands will prepare the configuration logic for rewriting the memory frames. All commands are described as 32-bit words, since configuration data is internally processed from a common 32-bit bus. From this data sequence, the first dummy word pads the front of the bitstream to provide the clock cycles necessary for initialization of the configuration logic. No actual processing takes place until the synchronization word is loaded. Since the V2 configuration logic processes data as 32-bit words, but can be configured from arbitrary data sources, e.g. a serial or 8-bit source, the synchronization word is used to define the 32-bit word boundaries. That is, the first bit after the synchronization word is the first bit of the next 32-bit word. The frame length indicates how many 32-bit words of configuration data, depicted as - - - - - - - -h, will be sent from the configuration controller and will contain "random" data.

IV. LOADING MICROCODE: PROBLEMS AND SOLUTIONS

In the original MOLEN architectural description, the end of reconfiguration microcode is marked by an end_op microinstruction. Conceptually, this is correct, however it creates some implementation drawbacks with respect to whether the reconfigurable operation is set or $execute$. That is, whether the microcode, stored into memory is a sequence of microinstructions ($execute$), or a configuration bitstream (set).

Microcode termination. In case of $execute$ microcode, end_op instruction at the end of the microcode segment is sufficient for the proper termination of the reconfigurable operation, provided the microcode is properly aligned into memory. This technique, however, would not work in cases of the set microcode, because the reconfiguration bitstreams are an arbitrary bit sequence as discussed in Section III. It is almost impossible to find a unique bit pattern, which can not be extracted from the reconfiguration bitstreams, thus used as end_op microinstruction. Therefore, it is possible, that a reconfiguration microcode loading is terminated earlier. Obviously, other techniques should be utilized for proper microcode segment termination. Figure 3 depicts three possible solutions

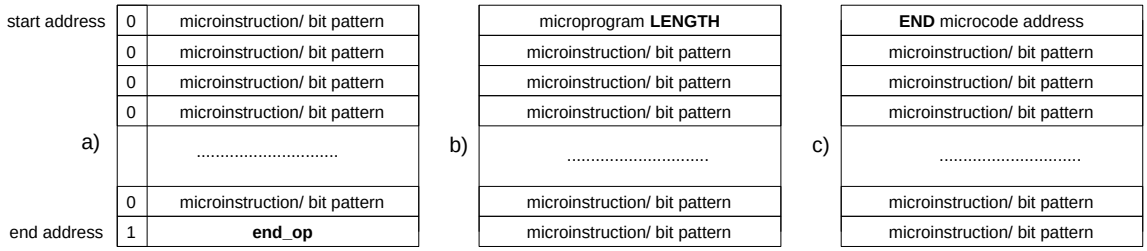

Fig. 3. Microcode termination techniques

that can be utilized to solve the pointed problem. On Figure 3a) a flag bit is utilized, to indicate whether the memory word is an end_op (1), or any other microinstruction/reconfiguration bit pattern (0). This approach is applicable for both *set* and *execute* microcodes, but it is costly in terms of memory space. Microprogram (resp. reconfigurable bitstream) alignment into the main memory is also severe, since the end_op microcode should be strictly aligned in the end of the microprogram segment/block. The examples in Figure 3 b) and c) are functionally equivalent to each other in terms of memory space and differ only in the potential hardware implementations. In both cases, an additional microcode word is aligned at the starting address of the microprogram segment. This word may contain either the length of the microprogram (Figure 3 b) or its final address (Figure 3 c). The latter two techniques are more efficient in terms of memory space since a single extra microinstruction word is required.

Microcode finalization. *The process of preparing the microcode for its final alignment into the targeted main memory is called microcode finalization*. In all three cases of microcode termination, extra termination information should be explicitly added to the microprogramable configuration code. In the case of end_op attached in the end, additional flag bit fields should be inserted into the microcode. The expanded set microcode bit patterns should be properly aligned to fit in the targeted memory. There is a variety of different design tools that can potentially be used for set microcode generation, e.g., Xilinx or Altera. There is also a number of GPPs, which can be used in the MOLEN organization framework. Therefore, an automated process that will perform the transformation from "raw" configuration stream to set microcode is required.

The automated process of microcode finalization for MOLEN is depicted in Figure 4. This figure shows the place of the finalization tool in the MOLEN CCU design process. The CCU algorithm described in any hardware description language, can be targeted to different FPGA technologies. This allows technology independent description that can be synthesized to any particular technology utilized by MOLEN. The result of the synthesis tool is the binary configuration file augmented with technology specific commands as discussed earlier. This file is ready to be loaded into the FPGA via any of the configuration paths supported, e.g. JTAG or dedicated configuration controller. The MOLEN paradigm requires the configuration stream, referred as the set microcode, to be positioned in the system main memory (similar to the software

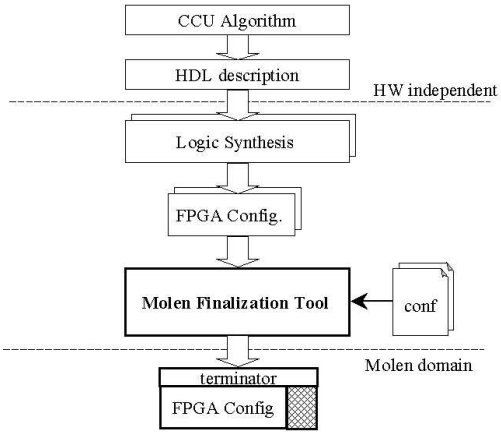

Fig. 4. MOLEN Finalization

modules) and be loaded via the $\rho\mu$ unit. The later $\rho\mu$ unit should know where is the end of the set microcode as discussed earlier. In addition the set microcode should "fit" nicely in the targeted memory architecture. For example if the targeted MOLEN organization consists of ARM7 processor with 16-bit wide external memory and Xilinx Virtex II FPGA utilizes the CCM part, every configuration word will use two subsequent address places.

It should be stated that such issues as the set code endianess are transparent to the proposed approach and do not require special consideration. All of the above is performed by the Finalization tool automatically. The configuration file (indicated as $conf$) contains information about the MOLEN organization needed for the set microcode finalization. The product of the Finalization tool is a binary file ready to be used inside the MOLEN paradigm, and can be a linkable object, or a high-level data structure, incorporating the binary information, that can be included directly in a C project before compilation.

V. Loading microcode: $\rho\mu$-unit implementation

In addition to the finalization process it is required that the $\rho\mu$-coded unit manages the format of the set microcode and transforms it, to the used hardware configuration channel. The loading hardware of the set microcodes as described in this section is the major part of the complete data path between the main memory and the particular configuration controller.

FPGA implementation. We have designed an $\rho\mu$-coded unit utilizing the microcode termination mechanism with the end microprogram address value stored at the starting microcode location. A general view of the design is depicted

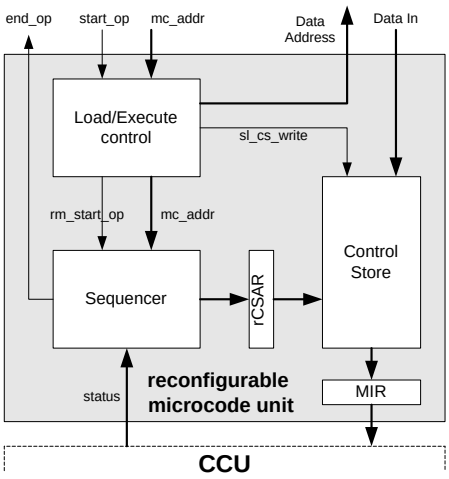

Fig. 5. General view of the $\rho\mu$-code unit

in Figure 5. The load/execute control block is responsible for loading microprograms from the external memory. It generates the starting signal *rm_start_op* to the sequencer once the desired microprogram (at address *mc_addr*) is already transferred or available in the control store. The *start_op* signal is generated by the Arbiter and initiates a reconfigurable operation. The load/execute block sequentially generates the addresses of the microprogram in the main (external) memory with starting address *mc_addr*. During this address generation, the *sl_cs_write* signal is active thus, the desired microprogram is loaded into the control store via the write-only port *Data_In*. Once the desired microprogram is available in the control store, i.e., the end address of the microprogram in the external memory is reached, signal *rm_start_op* is activated and the sequencer starts generating the microcode addresses towards the rCSAR (reconfigurable Control Store Address Register). The microinstruction to be executed is transferred to the CCU via the Microinsruction Register (MIR). *Status* signals from the CCU are directed to the sequencer to determine next microcode address. Once the CCU completes its task, the sequencer generates signal *end_op* to the arbiter. The arbiter initiates the execution of the next instruction from the application program, which can be either a reconfigurable or a fixed one from the core processor ISA.

We assumed a microcode word length of 32 bits and external (off-chip) memory segment of 4Mx32-bit (22-bit address) for microprograms. Virtex II Pro, has been used as a target reconfigurable technology. The (on-chip) control store has been designed to handle up-to 8K 32-bit microcode words. As primary microcode storage units (the control store), we have used the BRAM blocks of the FPGA fabrics, configured as a single dual port memory. Each of the ports is unidirectional - one read-only and one write only. The read-only port is used to feed the MIR, while the write-only one loads microcodes from the external memory into the pageable section of the control store. The VHDL code of the $\rho\mu-$code unit has been synthesized with Project Navigator ISE 5.1 S3 of Xilinx. The target FPGA chip was XC2VP20, speed grade 5. Hardware

TABLE II
SYNTHESIS RESULTS FOR XC2VP20, SPEED -5

Number of Slices	173 out of 10304	1%
Number of Slice Flip Flops	96 out of 20608	$< 1\%$
Number of 4 input LUTs	315 out of 20608	1%
Number of BRAMs:	15 out of 112	13%
Minimum clock period	8.160ns	
Maximum Frequency	122.549MHz	

costs reported by the synthesis tools are presented in Table II. In addition, a hardware link to the configuration device pins (e.g., the byte-parallel controller, of Virtex II Pro) is required. The configuration bitstream provided by our implementation in a byte wide fashion is to be routed and transformed if necessary to the device configuration pins. Since this is a straightforward implementation issue with a minimal hardware overhead, it is not covered by this paper. Moreover, different configuration paths may be utilized, e.g., SelectMAP, serial (master or slave) or boundary-scan, instead of the parallel path.

VI. CONCLUSIONS

In this paper, we addressed several specific problems related to the microcodes management in the MOLEN reconfigurable computing paradigm. More precisely, the generation, memory alignment and loading of configuration microcodes were investigated. The microcode termination issue was discussed in more details and alternative design considerations were proposed. Some specific features of a new microcode finalization tool were outlined after a detailed description of the set microcode finalization process. An analysis of several possible solutions of the discussed problems with respect to optimal hardware complexity and memory usage were presented. The described design considerations had been taken into account for an FPGA implementation of the $\rho\mu$-unit, presented in the end. Synthesis results indicate a very low hardware resources utilization of the targeted reconfigurable technology, selected to be Virtex II Pro of Xilinx (XC2VP20 device).

REFERENCES

[1] M.Wazlowski, L.Agarwal, T.Lee, A.Smith, E.Lam, H.Silverman, and S.Ghosh, "PRISM-II Compiler and Architecture," in *Proc.IEEE Workshop on FPGAs for Custom Computing Machines*, Napa Valley,CA, April 5-7, 1993, pp. 9–16.

[2] R.W.Hartenstein, R.Kress, and H.Reining, "A new FPGA Architecture for Word-Oriented Datapaths," in *4th International Workshop on Field Programmable Logic and Applications:Architectures, Synthesis and Applications*, September 1994, pp. 144–155.

[3] S.M.Trimberger, *Reprogramable Instruction Set Accelerator*.

[4] S. C. Goldstein, H. Schmit, M. Moe, M. Bidiu, S. Cadambi, R. Taylor, and R. Laufer, "PipeRench: A coprocessor for Streaming Multimedia Acceleration," *The 26-th International Symposium on Computer Architecture*, pp. 28–39, May 1999.

[5] M. Sima, S. Vassiliadis, S.Cotofana, J. van Eijndhoven, and K. Vissers, "Field-Programmable Custom Computing Machines - A Taxonomy," in *12th International Conference on Field Programmable Logic and Applications (FPL)*, Montpellier, France, Sep 2002, pp. 79–88.

[6] S. Vassiliadis, S. Wong, and S. Cotofana, "The MOLEN $\rho\mu$-Coded Processor," in *11th International Conference on Field Programmable Logic and Applications (FPL)*, vol. 2147. Belfast, UK: Springer-Verlag Lecture Notes in Computer Science (LNCS), Aug 2001, pp. 275–285.

[7] *Virtex-II Pro Platform FPGA handbook v1.0*, Xilinx Corporation, 2002.

[8] "Virtex Series Configuration Architecture User Guide," no. XAPP151, Sept. 2000.