Arbitrating Instructions in an $\rho\mu$-coded CCM

Georgi Kuzmanov and Stamatis Vassiliadis

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

Abstract. In this paper, the design aspects of instruction arbitration in an $\rho\mu$-coded CCM are discussed. Software considerations, architectural solutions, implementation issues and functional testing of an $\rho\mu$-code arbiter are presented. A complete design of such an arbiter is proposed and its VHDL code is synthesized for the VirtexII Pro platform FPGA of Xilinx. The functionality of the unit is verified by simulations. A very low utilization of available reconfigurable resources is achieved after the design is synthesized. Simulations of an MPEG-4 case study suggest considerable performance speed-up in the range of 2,4-8,8 versus a pure software PowerPC implementation.

1 Introduction

Numerous design concepts and organizations have been proposed to support the Custom Computing Machine (CCM) paradigm from different prospectives [6, 7]. An example of a detailed classification of CCMs can be found in [8]. In this paper we propose a design of a potentially performance limiting unit of the MOLEN $\rho\mu$-coded CCM: the arbiter [11]. We discuss all design aspects of the arbiter, including software considerations, architectural solutions, implementation issues and functional testing. A synthesizable VHDL code of the arbiter has been developed and simulated. Performance has been evaluated theoretically and by experimentation using Virtex II Pro technology of Xilinx. Synthesis results and performance evaluation for an MPEG-4 case study suggest:

- Less than 1% of the reconfigurable hardware resources available on the selected FPGA (xc2vp20) chip are spent for the implementation of the arbiter.
- Considerable speed-ups in the range of 2,4-8,8 of the MPEG-4 encoder are feasible when the SAD function is implemented in the proposed framework.

The remainder of this paper is organized as follows. The section to follow contains brief background on the MOLEN $\rho\mu$-coded processor and describes the requirements to the design of a general arbiter. In Section 3, software-hardware considerations for a particular arbiter design for PowerPC and Virtex II Pro FPGA are presented. Section 4 discusses functional testing, performance analysis and an MPEG-4 case study with experimental results obtained from a real chip implementation of the arbiter. Finally, we conclude the paper with Section 5.

2 Background

This section presents the MOLEN $\rho\mu$-coded Custom Computing Machine organization, introduced in [11] and described in Fig. 1. The ARBITER performs a partial decoding on the instructions in order to determine where they should be issued. Instructions implemented in fixed hardware are issued to the core processor. Instructions for custom execution are redirected to the *reconfigurable unit*. The reconfigurable unit consists of a custom computing unit (CCU) and

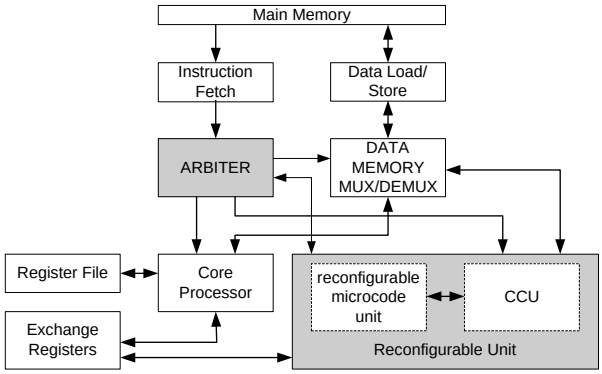


Fig. 1. The MOLEN machine organization

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** sub-phase only reconfigures that blocks in the CCU, which are not covered in the **pset** sub-phase in order to *complete* the functionality of the CCU.

General Requirements to the Arbiter. The arbiter controls the proper co-processing of the GPP and the reconfigurable units. It is closely connected to three major units of the CCM, namely the GPP, the memory and the $\rho\mu$-unit. Each of these parts of the organization has its own requirements, which should be considered when an arbiter is designed. Regarding the core processor, the arbiter should: 1) Preserve the original behavior of the core processor when no reconfigurable instruction is executed. Create the shortest possible critical path penalties. 2) Emulate reconfigurable instruction execution behavior on the core processor using its original instruction set and/or other architectural features.

Regarding the $\rho\mu$-unit the arbiter should: 1) Distribute control signals and the starting microcode address to the $\rho\mu$-unit. 2) Consume minimal hardware resources if implemented in the same FPGA with the $\rho\mu$-unit. Thus more resources will be available for the CCU.

For proper memory management the arbiter should be designed to: 1) Arbitrate the data access between the $\rho\mu$-unit and the core processor. 2) Allow speeds within the capabilities of the utilized memory technology, i.e., not creating performance bottlenecks in memory transfers.

Reconfigurable instructions should be encoded in consistence with the instruction encoding of the targeted general purpose architecture. The arbiter should also provide proper timing for reconfigurable instruction execution to all units referred above. In Fig.2, a general view of an Arbiter organization is pre-

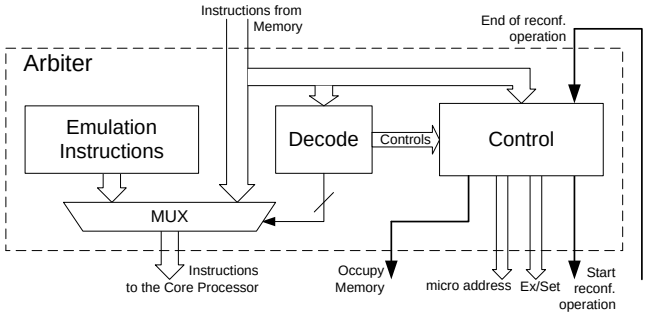


Fig. 2. General Arbiter organization

sented. The operation of such an arbiter is entirely based on decoding the input instruction flow. The unit either redirects instructions, or generates an instruction sequence to control the state of the core processor during reconfigurable operations. In such an organization, the critical path penalty to the original instruction flow can be reduced to just one 2-1 multiplexer. Once either of the three reconfigurable instructions has been decoded, the following actions are initiated:

1. Emulation instructions are multiplexed to the processor instruction bus. These instructions drive the processor into wait or halt state.
2. Control signals from the decoder are generated to the control block in Fig.2. Based on these controls, the control block performs the following: a) Redirects the microcode location address of the corresponding reconfigurable instruction to the $\rho\mu$-unit. b) Generates an internal code signal (Ex/Set) for the decoded reconfigurable instruction and delivers it to the $\rho\mu$-unit. c) Initiates reconfigurable operation by generating *'start reconf. operation'* signal to the $\rho\mu$-unit. d) Reserves the data memory control for the $\rho\mu$-unit by generating *memory occupy* signal to the (data) memory controller. e) Enters a wait state until signal *'end of reconf. operation'* arrives.

An active *'end of reconf. operation'* signal initiates the following actions: 1) Data memory control is released back to the core processor. 2) An instruction sequence is generated to ensure proper exiting of the core processor from the wait state. 3) After exiting the wait state, the program flow control is transferred to the instruction immediately after the reconfigurable instruction, executed last.

3 CCM Implementation

The general organization presented in the previous section has been implemented on Virtex II Pro, a platform FPGA chip of Xilinx.

Software considerations: Because of performance reasons, we decided not to use PowerPC special operating modes instructions (exiting special operating modes is usually performed by interrupt). We employed the *'branch to link register'* (**blr**) to emulate a wait state and *'branch to link register and link'* (**blrl**) to get the processor out of this state. The difference between these instructions is that **blrl** modifies the link register (LR), while **blr** does not. The next instruction address is the effective address of the branch target, stored in LR. When **blrl** is executed, the new value loaded into LR is the address of the instruction following the branch instruction. Thus the emulation instructions, stored into the corresponding block in Fig.2 are reduced to only one instruction for wait and one for 'wake-up' emulation.

Let us assume the following mnemonics for the three reconfigurable instructions: *'**pset** rm_addr'*, *'**cset** rm_addr'* and *'**exec** rm_addr'*. To implement the proposed mechanism, we only need to initialize LR with a proper value, i.e. the address of the reconfigurable instruction. This should be done by the compiler with the *'branch and link'* (**bl**) instruction of PowerPC. In the assembly code of the application program the *'complete set'* instruction should look like this:

bl *label1* → ***bl*** — branch to *label1* ; LR = *label1*
label1: **cset** *rm_addr* → ***blr*** — branch to *label1* ; LR = *label1*

Obviously, the processor will execute branch instruction to the same address, because LR remains unchanged and points to an address containing **blr** instruction. Thus we drive the processor into an eternal loop. It is the responsibility of the arbiter to get the processor out of this state. When the reconfigurable instruction is complete, an *'end_op'* signal is generated by the $\rho\mu$-unit to the arbiter, which initiates the execution of **blrl** exactly twice. Thus, the effective address of the next instruction is loaded into the LR, which points to the address of the instruction immediately following the reconfigurable one and the processor exits the eternal loop. Below, the instructions generated by the arbiter to finalize a reconfigurable operation are displayed (instruction alignment is at 4 bytes):

label1: **cset** *rm_addr* → ***blrl*** — branch to *label1* ; LR = *label1+4*
→ ***blrl*** — branch to *label1+4* ; LR = *label1+4*
label1+4: **next instr** → ..... — next instruction ; LR = *label1+4*

This approach allows executions of *blocks of reconfigurable instructions* (BRI): *We define **BRI** as any sequence of reconfigurable instructions starting with the instruction **'bl'** and containing arbitrary number of consecutive reconfigurable instructions. No other instructions can be utilized within a BRI.* Utilizing BRI saves the necessity to initialize LR every time a reconfigurable instruction is invoked, thus saving a couple of **bl** instructions. In this scheme only one **bl** instruction is used to initialize LR in the beginning of the BRI. The time spent for executing a single reconfigurable operation (T_ρ) is estimated to be the time for the *reconfigurable execution* ($T_{\rho E}$), consumed by the $\rho\mu$-unit, plus the time for

three *unconditional taken branch instructions* (T_{UTB}) : $T_\rho = 3 \times T_{UTB} + T_{\rho E}$. Assuming the number of reconfigurable instructions in the BRI to be N_{BRI}, the execution time of a reconfigurable instruction within a BRI costs: $T_\rho = 2 \times T_{UTB} + T_{\rho E} + \frac{T_{UTB}}{N_{BRI}}$ In other words, *the time penalty for single reconfigurable instruction execution is* $3 \times T_{UTB}$ *and within a BRI execution - between* $2 \times T_{UTB}$ *and* $3 \times T_{UTB}$. Optionally, the *'instruction synchronization'* instruction (**isync**) can be added before a BRI to avoid out-of-order executions of previous instructions during reconfigurable operation.

Instruction encoding. To perform the MOLEN processor reconfigurations, the PowerPC Instruction Set Architecture (ISA) is extended with three instructions. To encode these three instructions, we have considered the following: 1.) The encoding scheme should be consistent with the PowerPC instruction format with opcodes (OPC) encoded in the six most-significant bits of the instruction word (see Fig.3). 2.) All three instructions have the same OPC field and same instruction form, which is similar to the I-form. Let us call the new form of the reconfigurable instructions *R-form*. 3.) The OPCodes of the instructions are as close as possible to the OPC of the emulation instructions (shortest Hamming distance), i.e. **blr** and **blrl**. From the free opcodes of the PowerPC architecture, such is opcode '6' ("000110_b"). 4.) Instruction modifiers are implemented in the two least-significant fields of the instruction word, to distinguish the three reconfigurable instructions. 5.) A 24-bit address, embedded into the instruction word, specifies the location of the microcode in memory. A modifier bit R/P (Resident/Pageable), assumed to be a part of the address field, specifies where the microcode is located and how to interpret the address field. If R/P=1 a memory address is specified, otherwise an address of the on-chip storage in the $\rho\mu$-code unit is referred. The address always points to the location of the first microcode instruction. This first address should contain the length or the final address of the microcode. A microprogram is terminated by an *end_op* microinstruction.

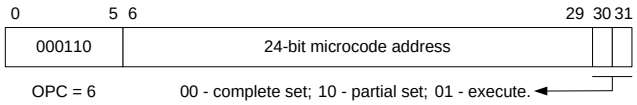


Fig. 3. Reconfigurable instruction encoding: R-form

Hardware requirements. To implement the instruction bus arbitration, we have considered the following: 1) Information, related to instruction decoding, arbitration and timing is obtained only through the instruction bus. 2) PowerPC instruction bus is 64-bit wide and instructions are fetched in couples. 3) Speculative dummy prefetches should not disturb the correct timing of a reconfigurable instruction execution. 4) Both the arbiter and the $\rho\mu$-unit strobe input signals on rising clock edges and generate output controls on falling clock edges.

The $\rho\mu$-code arbiter for PowerPC has been described in synthesizable VHDL and mapped on the Virtex II Pro FPGA. Fig. 4 depicts the timing of this im-

plementation. The unit uses the same clock (*'sl_mem_clk'*) as the instruction

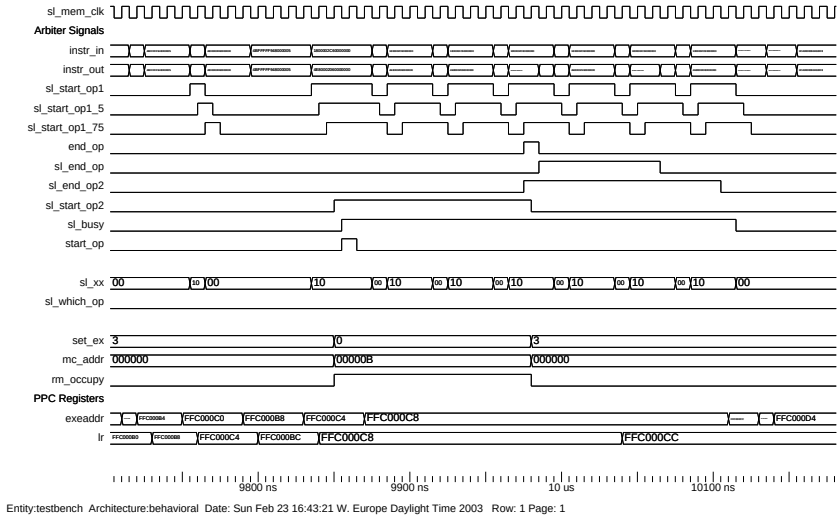


Fig. 4. Reconfigurable instruction execution timing

memory, in this case a fast ZBT RAM. The only inputs of the arbiter are the *input instruction* bus (*'instr_in'*) and *end of (reconfigurable) operation* (*'end_op'*).

The *decode unit* of the arbiter (Fig. 2) decodes both OPCodes of the fetched instructions. Non-reconfigurable instructions are redirected (via MUX) to output *'instr_out'*, directly driving the instruction bus of PowerPC. Alternatively, when either of the decoded two instructions is reconfigurable, the instruction code of **blr** is multiplexed via *'instr_out'* from the *'emulation instructions'* block. Obviously, the critical path penalty to the original instruction flow is just one 2-1 multiplexer and the decoding logic for a 6-bit value. The decode block generates two internal signals to the control block - *sl_start_op1* (explained later) and *sl_xx*. The latter signal indicates the alignment of the fetched instructions with respect to the reconfigurable ones. A one represents a reconfigurable instruction, a zero - any other instruction. For example, assuming big endian alignment: *"sl_xx=10"* means a reconfigurable instruction at the least-significant and a non-reconfigurable instruction at the most-significant address.

The control block generates signal *start (reconfigurable) operation* (*'start_op'*) for one clock cycle delayed with two cycles after the moment a reconfigurable operation is prefetched and decoded, thus filtering short (dummy) prefetches. In Fig. 4 the rising edge of the internal signal *sl_start_op1* indicates the moment a reconfigurable operation is decoded. One can see that signal (*'start_op'*) is generated only when the reconfigurable instruction is really fetched, i.e. when *sl_start_op1* takes longer than one clock cycle. Dummy prefetch filtration has been implemented by two flip-flops, connected in series and clocked by com-

plementary clock edges. The outputs of these flip-flops are denoted by signals *sl_start_op1_5* and *sl_start_op1_75*. The output control to the $\rho\mu$-unit, *sl_start_op* is generated between two falling clock edges.

Synchronously with the decoding of a reconfigurable instruction, the two instruction modifier fields (output signal *set_ex*) and *microcode address* (24-bit output *mc_addr*) are registered on rising clock edge (recall Fig.3). The internal flip-flop *sl_which_op* is used only when both of the fetched instructions are reconfigurable (*sl_xx="11"*) to ensure the proper timely distribution of *set_ex*, *mc_addr* and controls. In addition, two internal signals (flip-flops) are set when reconfigurable instruction is decoded. These two signals denote that the $\rho\mu$-unit is performing an operation (*sl_start_op2*) and that the arbiter is busy (*sl_busy*) with such an operation, therefore another reconfigurable execution can not be executed. To multiplex the data memory ports to the $\rho\mu$-unit during reconfigurable operations, signal *rm_occupy* is driven to the data memory controller.

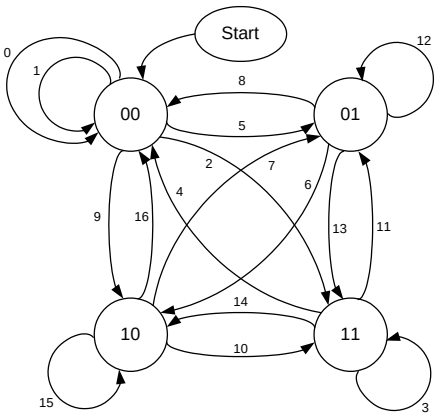
When a reconfigurable instruction is over, *'end_op'* is generated by the $\rho\mu$-unit and the *sl_start_op2* flip-flop is reset, thus releasing the data memory (via *rm_occupy*) for access by other units. Now, the control logic should guarantee that the **blrl** instruction is decoded exactly twice. This is done by a counter issuing active *sl_end_op* for precisely two **blrl** cycles, i.e., eight clocks. Instruction codes of **blr** and **blrl** differ only in one bit position. Therefore, redirecting *sl_end_op* via the MUX to this exact position of *'instr_out'* while **blr** is issued, drives **blrl** to the PowerPC. When *'end_op'* is strobed by the arbiter, another counter generates the *sl_end_op2* signal to prevent other reconfigurable operations from starting executions before the current reconfigurable operation has finished properly. The falling edge of signal *sl_end_op2* synchronously resets signal *busy*, thus enabling the execution of reconfigurable operations coming next.

4 Arbiter Testing, Analysis, and Case Study

Testing. To test the operation of the arbiter, we need a program, strictly aligned into memory, which tests all possible sequences of instruction couple alignments. Fig. 5(a) depicts the transition graph of such a test sequence, where a bubble represents an instruction couple alignment (1=reconfigurable instruction, 0 = other instruction). Arrows (transitions) fetch the next aligned instruction couple. Minimum 16 transitions cover all possible situations. The number next to each arrow indicates its position in the program sequence. An extra transition (arrow 0) tests the dummy prefetch filtration. Fig. 5(b) depicts its assembly code.

Performance analysis. Let us assume T to be the execution time of the original program (say measured in cycles) and T_{SEi} - time to execute kernel i in software, which we would like to speed-up in reconfigurable hardware. With respect to T_ρ from Section 3, $T_{\rho i}$ is the execution time for the reconfigurable implementation of kernel i. Assuming $a_i = \frac{T_{SEi}}{T}$ and $s_i = \frac{T_{SEi}}{T_{\rho i}}$, the speed-up of the program with respect to the reconfigurable implementation of kernel i is:

$$S_i = \frac{T}{T - T_{SEi} + T_{\rho i}} = \frac{1}{1 - (a_i - \frac{a_i}{s_i})} \quad (1)$$



(a) Transition graph

```
        ...
        bl    labelA

labelB  bl    labelC
        nop

//dummy prefetch here
labelA  bl    labelB
labelC  bl    label1

label1: cset  rm_addr
        .......
```

(b) Dummy prefetch test

Fig. 5. Test program

Identically, assuming $a = \sum_i a_i$, all kernels potential candidates for reconfigurable implementation would speed-up the program with:

$$S = \frac{T}{T - \sum_i T_{SEi} + \sum_i T_{\rho i}} = \frac{1}{1 - (a - \sum_i \frac{a_i}{s_i})},\ S_{max} = \lim_{\forall s_i \to \infty} S = \frac{1}{1-a} \quad (2)$$

Where S_{max} is the theoretical maximum speed-up. Parameters a_i may be obtained by profiling the targeted program, or along with s_i, by running an application on the real MOLEN CCM. Further in this Section, we will use (1) and (2) to compare theoretical to actual experimental speed-up.

Experimental testbench. To prove the benefits of the proposed design we followed an experimental scenario. First, we use *profiling data* for the application to *extract computationally demanding kernels*. Second, we *design hardware engines*, which implement these kernels in performance efficient hardware. Further, we go trough the following steps, to get experimental data for analysis:

1. Describe the MOLEN organization and the hardware kernel designs in VHDL and synthesize them for the selected target FPGA technology.
2. Simulate the pure software implementations of the kernels on a VHDL model of the core processor to obtain performance figures.
3. Simulate the hardware implementations of the same kernels, embedded in the MOLEN organization, and mapped on the target FPGA (i.e.,VirtexIIPro).
4. Estimate the speed-ups of each kernel (s_i) and for the entire application (S_i, S), based on data from the previous steps and the initial profiling.
5. Download the FPGA programming stream into a real chip and run the application, to validate the figures from simulations.

Performance speed-up: an MPEG-4 case study The application domain of interest in our experimentations is the visual data compression and

in particular the MPEG-4 standard. For parameter a_i, we use profiling data reported in literature [1–4, 10]. Values of some "global" parameters (a_i) regarding overall MPEG-4 performance may be within a standard deviation of 20% [3], with respect to the particular data. On the other hand, "local" parameters regarding implemented kernels ($T_{SEi}, T_{\rho i}, s_i$) are less data dependent, thus more predictable (accuracy within 5%). Table 1 contains experimental results for the implementation of the most demanding function in MPEG-4 encoder, the Sum-of-Absolute Differences (SAD), utilizing a design, described in [9] and assuming memory addressing schemes, discussed in [5]. The SAD kernel takes 3404 PowerPC cycles to execute in pure software. For its reconfigurable execution in MOLEN, we run two scenarios: a)worst case, when SAD execution microcode address is pagable and not residing in the $\rho\mu$-unit; and best case, when the microcode is fixed into the $\rho\mu$-unit. Experimental results in Table 1 strongly suggest that considerable speed-up of MPEG-4 encoders in the range of 2,41-8,82 is achievable only by implementing the SAD function as CCU in the MOLEN CCM organization. Both experimental and theoretical results indicate that for great kernel speed-ups ($s_i >> 1$), the difference in overall performance (S_i) between worst and best case (pageable and fixed μ-code) is diminishing.

Table 1. Speed-up for pageable $\rho\mu$-code, fixed $\rho\mu$-code, and theoretical maximum.

MPEG-4 SAD	$T_{SEi} = 3404[cyc]$		
	Pag.	Fixed	Theor.
$T_{\rho i}$,[cyc]	87	51	-
s_i	39	67	∞
$S_i, (a_i = 0, 6)$	2,41	2,45	2,50
$S_i, (a_i = 0, 66)$[3],[4]	2,80	2,86	2,94
$S_i, (a_i = 0, 88)$[2],[10]	7,01	7,51	8,33
$S_i, (a_i = 0, 90)$[1]	8,13	8,82	10

Table 2. Synthesis Results for xc2vp20, Speed Grade -5

Slices	84 of 10304	$< 1\%$
Flip Flops	69 of 20608	$< 1\%$
4 input LUTs	150 of 20608	$< 1\%$
Clock period	7.004ns	
Frequency	142.776MHz	

FPGA synthesis results. The VHDL code of the arbiter has been simulated with Modeltech's ModelSim and synthesized with Project Navigator ISE 5.1 S3 of Xilnx. The target FPGA chip was XC2VP20. Hardware costs obtained by the synthesis tools are reported in Table 2. Post-place-and-route results indicate the total number of slices to be 80 and memory clock frequency of 100 MHz to be feasible. These results strongly suggest that at trivial hardware costs the $\rho\mu$-arbiter design can arbitrate the PowerPC instruction bus without causing severe critical path penalties and frequency decreases. Moreover, virtually all reconfigurable resources of the FPGA remain available for CCUs. Regarding the total number of flip-flops in the arbiter design (69), most of them (52) are used for registering *mc_addr* and *set_ex* outputs. Thus only 17 flip-flops are spent for the control block, including the two embedded counters (2×4 flip-flops).

5 Conclusions

In this paper, we proposed an efficient design of a potentially performance limiting unit of an $\rho\mu$-coded CCM: the arbiter. The general $\rho\mu$-coded machine organization MOLEN was implemented on the platform FPGA Virtex II Pro and the arbitration between reconfigurable and fixed PowerPC instructions investigated. All design aspects of the arbiter have been described, including software considerations, architectural solutions, implementation issues and functional testing. Performance has been evaluated analytically and by experimentation. Synthesis results indicate trivial hardware costs for an FPGA implementation. Simulations suggest that considerable speed-ups (in the range of 2,4-8,8) of an MPEG-4 case study are feasible when the SAD function is implemented in the proposed framework. The presented design will be implemented on an FPGA prototyping board.

Acknowledgements: This research is supported by PROGRESS, the embedded systems research program of the Dutch scientific organization NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW.

References

1. H.-C. Chang, L.-G. Chen, M.-Y. Hsu, and Y.-C. Chang. Performance analysis and architecture evaluation of MPEG-4 video codec system. In *IEEE International Symposium on Circuits and Systems*, vol. II, pp. 449–452, 28-31 May 2000.
2. H.-C. Chang, Y.-C. Wang, M.-Y. Hsu, and L.-G. Chen. Efficient algorithms and architectures for MPEG-4 object-based video coding. In *IEEE Workshop on Signal Processing Systems*, pp. 13–22, 11-13 Oct 2000.
3. J. Kneip, S. Bauer, J. Vollmer, B. Schmale, P. Kuhn, and M. Reissmann. The MPEG-4 video coding standard - a VLSI point of view. In *IEEE Workshop on Signal Processing Systems,(SIPS98)*, pp. 43–52, 8-10 Oct. 1998.
4. P. Kuhn and W. Stechele. Complexity analysis of the emerging MPEG-4 standard as a basis for VLSI implementation. In *SPIE Visual Comunications and Image Processing (VCIP)*, vol. 3309, pp. 498–509, Jan. 1998.
5. G. Kuzmanov, S. Vassiliadis, and J. van Eijndhoven. A 2D Addressing Mode for Multimedia Applications. In *SAMOS 2001*, vol. 2268 of *Lecture Notes in Computer Science*, pp. 291–306, July 2001. Springer-Verlag.
6. 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*, pp. 9–16, 5-7 April, 1993.
7. R.W.Hartenstein, R.Kress, and H.Reining. A new FPGA Architecture for Word-Oriented Datapaths. In *FPL 1994*, pp. 144–155, 1994.
8. M. Sima, S. Vassiliadis, S. Cotofana, J. T. van Eijndhoven, and K. Vissers. Field-Programmable Custom Computing Machines. A Taxonomy. In *FPL 2002.*, vol. 2438 of *Lecture Notes in Computer Science*, pp. 79–88, Sept. 2002. Springer-Verlag.
9. S. Vassiliadis, E. Hakkennes, J. Wong, and G. Pechaneck. The Sum Absolute Difference Motion Estimation Accelerator. In *EUROMICRO 98*, vol. 2, Aug. 1998.
10. S. Vassiliadis, G. Kuzmanov, and S. Wong. MPEG-4 and the New Multimedia Architectural Challenges. In *15th SAER'2001*, 21-23 Sept. 2001.
11. S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN $\rho\mu$-coded processor. In *FPL 2001*, pp. 275–285, Aug. 2001.