# Instruction Set Extensions for Multi-Threading in LEON3

M. Danek, L. Kafka, L. Kohout, J. Sykora
Department of Signal Processing, UTIA AV CR
Pod vodarenskou vezi 4
Praha 8, 182 08, Czech Republic
Email: danek@utia.cas.cz

*Abstract*— **This paper describes instruction set extensions for a variant of multi-threading called micro-threading for the LEON3 SPARCv8 processor. We show an architecture of the developed processor and its key blocks - cache controller, register file, thread scheduler. The processor has been implemented in a Xilinx Virtex2Pro FPGA. The extensions are evaluated in terms of extra resources needed, and the overall performance of the developed processor is evaluated on a simple DSP computation typical for embedded systems.**

## I. Introduction

Current processors have reached their maximum operating frequency, and performance improvements must be sought in better organization of the computation. One area for improvements is the tolerance of latency of data caused e.g. by a memory or I/O access, which is usually handled by context switching and executing computation threads that have data available in processors that support multithreading.

As the silicon area becomes cheaper as a consequence of the Moore's law, it has become viable to extend processors to support in hardware execution of multiple threads on one processor or in a multiprocessor cluster. Two significant examples are the SUN Microsystems OpenSPARC T1/T2 and the MIPS MT processors. OpenSPARC T1/T2 is an open-source version of the UltraSPARC T1/T2 [1], [2]; T1 has been ported to the Xilinx FPGAs, while MIPS MT [3] is a commercial processor available as an ASIC. The architecture complexity of the open-source OpenSPARC T1/T2 is too high for embedded applications, which is due to their primary domain in server and desktop computing. Also the context switch time for T1/T2 is high, about 1000 clock cycles. We do not know of any other multithreaded processor available in source code to the design community.

To overcome this we have designed and implemented instruction set extensions for the simpler LEON3 SPARCv8 processor suitable for embedded applications. This paper describes the architecture of the modified LEON3 [4] processor (which we call UTLEON3) and the impact of the architectural improvements on the processor performance. The goal has been twofold: First, we wanted to implement in silicon machine-level instructions that were identified as necessary for efficient micro-threading (a variant of multi-threading with fast context switching) [5] to have a true picture how these extensions are expensive in terms of silicon. Second, we wanted to achieve an efficient implementation of these extensions, that is outperform the original LEON3 processor by better handling memory and execution latencies.

The paper is structured as follows: Section II describes the extra machine instructions that implement micro-threading in SPARC. Section III describes the architecture of the key blocks that implement the micro-threaded extensions. Section IV compares FPGA implementations of the classical LEON3 and the new UTLEON3 in terms of resource requirements. Section V evaluates the speedup of a simple assembly program in a legacy version executed on the classical LEON3 as well as the UTLEON3 processor, and a micro-threaded version executed on the UTLEON3 processor. Section VI compares UTLEON3 with the OpenSPARC T1/T2 processor. Section VII concludes the paper.

## II. Micro-Threading

Micro-threading is a multi-threading variant that decreases the complexity of context management. The goal of micro-threading is to tolerate long-latency operations (LD/ST and multi-cycle operations such as floating-point) and to synchronize computation on register access. For an overview of multi-threading see [6].

In a simple case the context can be represented by the program counter and by window pointers to the register file. Micro-threading has been developed both on the assembly and C levels. The basic conceptual unit is a family of threads that share data and implement one piece of a computation. In a simple view one family corresponds to one for-loop in the classical C; in micro-threading each iteration (each thread) of a hypothetical for-loop (represented by a family of threads) is executed independently according to data dependencies. A family is synchronized on termination of all its threads. For more details on micro-threading see [5], [7], [8].

A possible speedup generated by micro-threading comes from the assumption that while one thread is waiting for its input data, another thread has its input data ready and can be scheduled in a few clock cycles and executed. Another assumption is that load and store operations themselves need not be blocking since the real problem arises just when an operation accesses a register that does not contain a valid data value. Finally, the thread management logic is considered simple enough to fit in the processor hardware reasonably well in the current technologies.

The hardware requirements of microthreading are: use of a self-synchronizing register file (i-structures, [9]), register states to be managed autonomously in the register file, pipeline stalls prevented by context switch in hardware, and thread status and context switch managed autonomously in a hardware thread scheduler.

The micro-threading support on the machine level is represented by the following instructions:

- **launch** - switches the processor from the legacy mode (user or protected) to the microthreaded mode.
- **allocate** - allocates a family table entry, needed to create a family of threads.
- **setxxxxx** - fills in the allocated family table entry with parameters required by the *create* instruction.
- **create** - creates (a family of) threads based on a family table entry.
- **.registers** - a pseudoinstruction that specifies the number of registers needed by a thread.

Furthermore, each 32-bit instruction word is extended by another two bits that act as an instruction for thread scheduling. Valid combinations are:

- **cont** - continue thread execution,
- **swch** - switch the context to another thread, e.g. on memory load to prevent possible pipeline stall,
- **end** - end thread execution, i.e. the thread ends at this instruction.

The format of assembly instructions has been extended by a field delimited by a semicolon that may contain an explicit instruction for the scheduler. If the field is missing, *cont* is assumed by default.

```
clr %r2
ld [%r1 + %g0], %r3 ; swch
add %r3, %g0, %r4 ; end
```

To keep the 32-bit organization of the memory system in SPARCv8 2-bit extensions for groups of 15 instructions are grouped in one 32-bit instruction word that is located at the beginning of each cache line. One cache line is formed by 16 words. The first word of each cacheline is skipped in the micro-threaded mode (explained later in the text). The organization of one instruction cache line is shown in Figure 1.

Micro-threading relies on the use of a self-synchronizing register file based on *i-structures* [9]. To implement the i-structures each register has to be extended to contain the state of its value. A register can be

- **empty** - on power-on reset,
- **pending** - a memory load operation has been requested and no thread has accessed the register since,
- **waiting** - a memory load operation has been requested and a thread has accessed the register since,
- **full** - the register contains valid data.

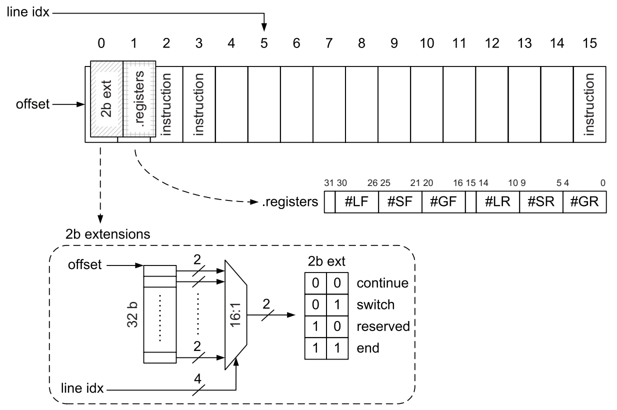


Fig. 1. Organization of the instruction cache. 16 words = 1 cache line

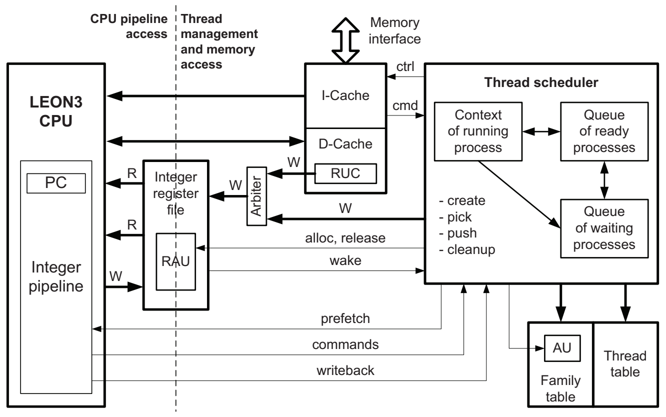


Fig. 2. Architecture of UTLEON3. RUC - register update controller, RAU - register allocation unit, AU - allocation unit.

In the micro-threading model a pending register can be accessed by at most one thread - either by the thread that initiated the pending data update, or by its direct sibling (only unidirectional data dependencies between direct sibling threads are allowed in micro-threading).

A sample program execution is shown in Figure 3. The processor starts in the legacy mode on power-on reset, then it switches to the microthreaded mode. The parent thread gets synchronized with the children threads by reading the register $\%l2$. On completion of all microthreads the processor switches back to the legacy mode.

## III. UTLEON3 Architecture

Figure 2 shows the architecture of UTLEON3, an extended LEON3 with ISE for micro-threading. We have maintained full backward compatibility with LEON3. The core is a 32-bit integer pipeline that executes all legacy instructions. Thread management is implemented in a thread scheduler, which can be seen as a simple 2-bit processor. The instruction word of UTLEON3 is 34 bits wide. All registers have been extended by 2 bits that capture register states, each register is 34 bits long.

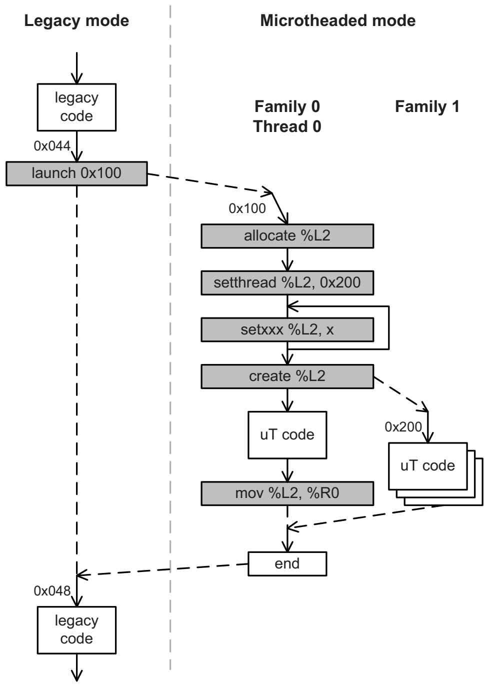


Fig. 3. Program flow.

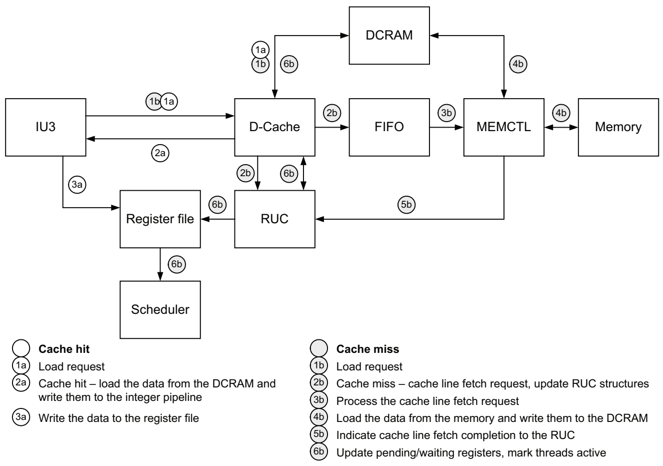

DCRAM
IU3
D-Cache
FIFO
MEMCTL
Memory
Register file
RUC
Scheduler
Cache hit
1a Load request
2a Cache hit – load the data from the DCRAM and write them to the integer pipeline
3a Write the data to the register file
Cache miss
1b Load request
2b Cache miss – cache line fetch request, update RUC structures
3b Process the cache line fetch request
4b Load the data from the memory and write them to the DCRAM
5b Indicate cache line fetch completion to the RUC
6b Update pending/waiting registers, mark threads active

Fig. 4. Data cache hit/miss.

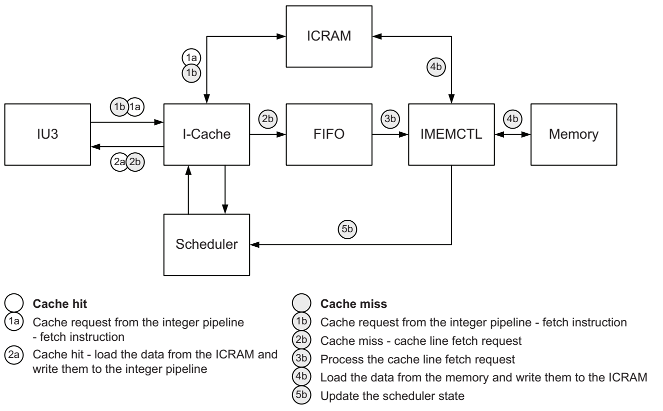

ICRAM
IU3
I-Cache
FIFO
IMEMCTL
Memory
Scheduler
Cache hit
1a Cache request from the integer pipeline - fetch instruction
2a Cache hit - load the data from the ICRAM and write them to the integer pipeline
Cache miss
1b Cache request from the integer pipeline - fetch instruction
2b Cache miss - cache line fetch request
3b Process the cache line fetch request
4b Load the data from the memory and write them to the ICRAM
5b Update the scheduler state

Fig. 5. Instruction cache hit/miss. Requests originated in the fetch stage.

### A. Cache Controllers

Load and store requests do not block the integer pipeline. Requests are queued and executed when the corresponding cache line fetch completes.

Memory accesses are decoupled from the integer pipeline. The cache controllers are divided in two parts connected through cache line fetch request FIFOs. The pipeline side cache controllers store fetch requests in the FIFOs. The memory side cache controllers process the queued requests. On completion of an instruction cache line fetch all threads waiting for the cache line are marked as ready for execution in the scheduler (put in the *active* queue). Cache lines that are used by threads are locked to prevent their eviction and guarantee forward progress.

On completion of a data cache line fetch all registers that have been waiting for the data in the cache line are updated by the register update controller (RUC). Cache line fetch scenarios are shown in Figures 4, 5, 6, 7.

### B. Thread Scheduler

The thread scheduler manages the family and thread tables, creates threads, switches context and cleans up the tables on thread completion (see Figure 2). Dynamic register allocation is performed on thread creation by the register allocation unit (RAU). Family table and thread table store information on threads being processed in the processor. Context switch can be the result of an explicit *swch* or *end* instruction, an instruction cache miss or it can occur on reading a register not marked *full*. Threads can be in one of six states; the state transition diagram is shown in Figure 8. Thread creation and context switch is shown in Figures 9 and 10.

## IV. Implementation Results

We have implemented and tested the designed architecture in the Xilinx XUP-V2Pro development board with the XC2VP30 FPGA. The initial unoptimized design operates at 25MHz (the register file runs at 75MHz to implement 6 access ports while using dual-port BRAMs). Implementation results are shown in Table I. The columns *LEON3* and *UTLEON3* compare complete systems with a processor, 1kB ROM, 4kB RAM and UART. The remaining columns show resource requirements of both legacy blocks (e.g. *CACHE*) and the micro-threaded blocks (e.g. *UTCACHE*). IU3 - integer pipeline, RF - register file, FTT - family thread table, TT - thread table, SCHED - thread scheduler.

LEON3 was configured with 8 register windows, cache associativity 1, cache set size 1kB, cache line size 8W (maximal allowable value for LEON3).

UTLEON3 was configured with 8 register windows, cache associativity 1, cache set size 1kB, cache line size 16W, family table size 8 items, thread table size 32 items.

It can be seen that UTLEON requires about 50% more slices and 200% more BRAMs than LEON3; the BRAM consumption is mainly driven by the extra information stored

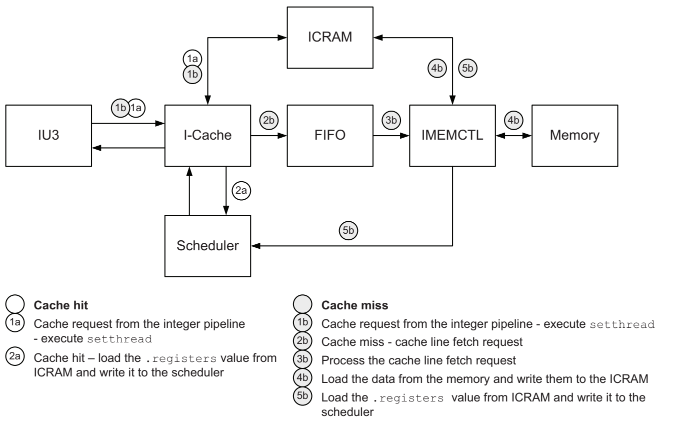

ICRAM
1a
1b
4b
5b
IU3
1b 1a
I-Cache
2b
FIFO
3b
IMEMCTL
4b
Memory
2a
Scheduler
5b
Cache hit
1a Cache request from the integer pipeline - execute setthread
2a Cache hit – load the .registers value from ICRAM and write it to the scheduler
Cache miss
1b Cache request from the integer pipeline - execute setthread
2b Cache miss - cache line fetch request
3b Process the cache line fetch request
4b Load the data from the memory and write them to the ICRAM
5b Load the .registers value from ICRAM and write it to the scheduler

Fig. 6. Instruction cache hit/miss. Requests originated in the execute stage.

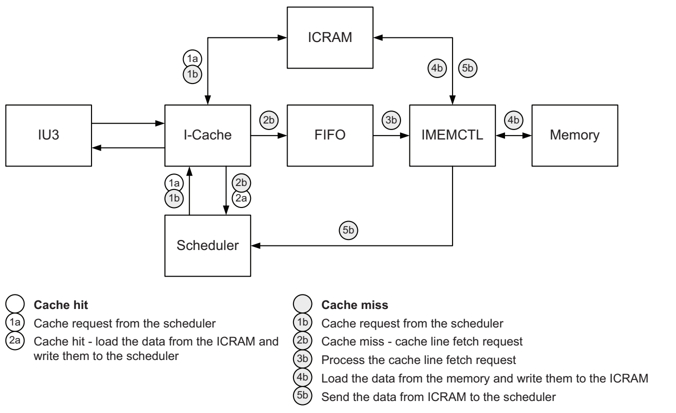

ICRAM
1a
1b
4b
5b
IU3
I-Cache
2b
FIFO
3b
IMEMCTL
4b
Memory
1a
1b
2b
2a
Scheduler
5b
Cache hit
1a Cache request from the scheduler
2a Cache hit - load the data from the ICRAM and write them to the scheduler
Cache miss
1b Cache request from the scheduler
2b Cache miss - cache line fetch request
3b Process the cache line fetch request
4b Load the data from the memory and write them to the ICRAM
5b Send the data from ICRAM to the scheduler

Fig. 7. Instruction cache hit/miss. Requests originated in the scheduler.

in cache tags and requirements on simultaneous cache tag access.

### A. Running Programs in Hardware

The design is downloaded in the board using IMPACT and operated using the Aeroflex-Gaisler GRMON tool. Classical or microthreaded programs are compiled with an extended version of the GNU binutils tools, and either put in the ahbrom.vhd file and synthesized in a ROM, or downloaded as ELF files with GRMON.

As the current version of GRMON does not support UT-LEON3 debugging, programs cannot be stepped or stopped once their execution starts, but the instruction and bus trace history shows a (limited) execution history of the user microthreaded program. Program results can be inspected in the memory. Performance data can be read from performance counters that measure specific events in the system (e.g. overall clock cycles, cache miss count, pipeline idle time).

## V. Simple Benchmark Program

The envisioned target application area of the processor is embedded digital signal processing systems. To evaluate this we implemented a simple vector scaling operation:

$$z_i = Ax_i + Y; i \in \{0..N-1\} \quad (1)$$

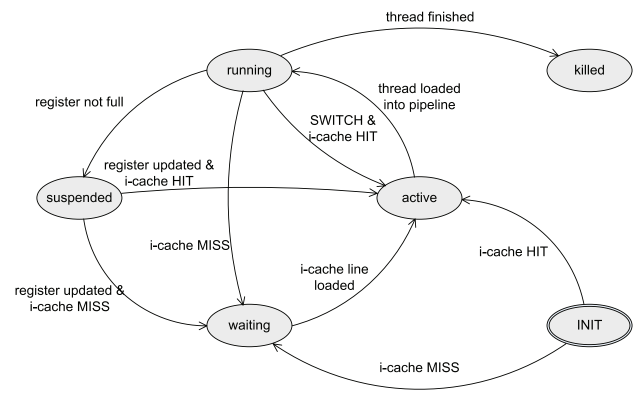


Fig. 8. Transitions between thread states.

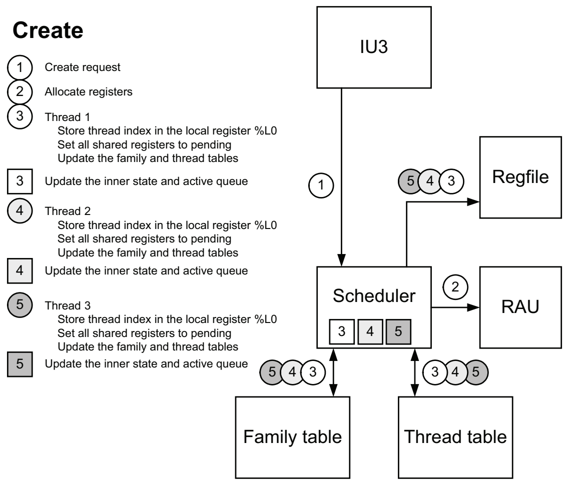

Create
1 Create request
2 Allocate registers
3 Thread 1
Store thread index in the local register %L0
Set all shared registers to pending
Update the family and thread tables
3 Update the inner state and active queue
4 Thread 2
Store thread index in the local register %L0
Set all shared registers to pending
Update the family and thread tables
4 Update the inner state and active queue
5 Thread 3
Store thread index in the local register %L0
Set all shared registers to pending
Update the family and thread tables
5 Update the inner state and active queue
IU3
1
5 4 3
Regfile
Scheduler
3 4 5
2
RAU
5 4 3
3 4 5
Family table
Thread table

Fig. 9. Thread creation in scheduler.

where $x$ and $z$ represent the input and output arrays, respectively. $A$ and $Y$ are integer scalar variables unknown during the compilation, and the parameter $N$ specifies the length of the vector operation. We have tested $N \in \{64, 96, 128\}$, the tables show values for $N = 128$.

Figure 11 depicts an actual implementation of the program; the left part shows the legacy version using a *for*-loop, while the right side shows the microthreaded version. The microthreaded version creates one family of threads (marked F1 in the picture) that correspons to the classical *for*-loop.

The *create-family* pseudo-command from the code example is further decomposed into a sequence of assembly-level instructions: **allocate**, **setstart**, **setlimit**, **setstep**, **setthread**, and **create**. These instructions will be assigned their parameters from the arguments of the *create-family* pseudo-command. Besides the obvious parameters (*index*, *start*, *limit*, *step*), which directly correspond to a classical *for*-loop construct, there are some other that need an explanation: *global*, *shared*, and *blocksize*.

The *global* and *shared* parameters specify lists of variables (registers) that will be made visible to the family being created. The difference between these two is that the *global* ones stay

TABLE I

DEVICE UTILIZATION SUMMARY

| Resource type | LEON3 | UTLEON3 | CACHE | UTCACHE | IU3 | UTIU3 | RF | UTRF | FTT | TT | SCHED |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Slices | 5844 | 9686 | 1402 | 3277 | 2108 | 2723 | 0 | 250 | 53 | 6 | 1134 |
| Slice Flip Flops | 2730 | 5307 | 243 | 1402 | 895 | 1314 | 0 | 282 | 30 | 0 | 540 |
| Total 4 input LUTs | 10333 | 16957 | 2505 | 5845 | 3763 | 4694 | 0 | 434 | 75 | 10 | 1884 |
| used as logic | 10296 | 16563 | 2505 | 5843 | 3717 | 4618 | 0 | 344 | 57 | 10 | 1776 |
| used as shift registers | 37 | 64 | 0 | 2 | 46 | 76 | 0 | 0 | 0 | 0 | 0 |
| used as RAMs | 0 | 330 | 0 | 0 | 0 | 0 | 0 | 90 | 18 | 0 | 108 |
| BRAMs | 16 | 44 | 0 | 7 | 0 | 0 | 2 | 1 | 5 | 10 | 0 |

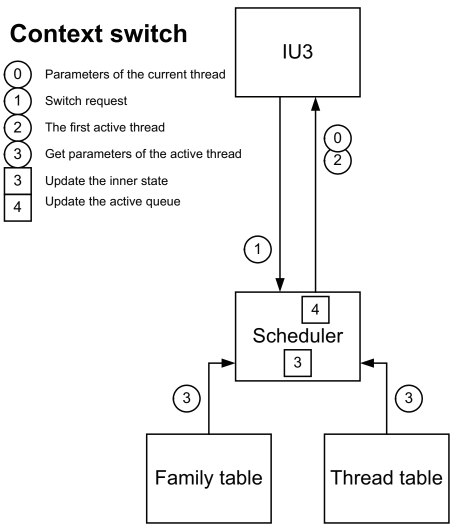
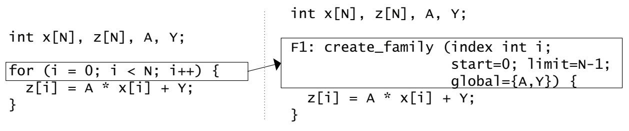

Context switch
0 Parameters of the current thread
1 Switch request
2 The first active thread
3 Get parameters of the active thread
3 Update the inner state
4 Update the active queue
IU3
1
0
2
4
Scheduler
3
3
3
Family table
Thread table

Fig. 10. Context switch in scheduler.



```
int x[N], z[N], A, Y;

for (i = 0; i < N; i++) {
  z[i] = A * x[i] + Y;
}
```

```
int x[N], z[N], A, Y;

F1: create_family (index int i;
                   start=0; limit=N-1;
                   global={A,Y}) {
  z[i] = A * x[i] + Y;
}
```

Fig. 11. The benchmark program; legacy code and microthreaded code.

fixed during the course of execution of the family, while the *shared* ones are assumed to be passed–and possibly modified–from one thread to another. This sharing of data is strictly unidirectional and always only between two adjacent threads in the family, i.e. from a thread indexed $i$ to a thread indexed $i + 1$. Global and shared variables are directly supported by the machine architecture by the means of thread global and shared registers. The quantity of these registers can be individually customized for each family using the **.registers** assembly directive.

In the benchmark example the *global* parameter is used to specify that variables $A$ and $Y$ will be passed to the thread family.

The final family parameter to be described is the *blocksize*. This parameter is optional for it does not affect the *semantics* of the computation, but it affects its pace. The *blocksize* specifies the maximal number of threads of a given family that are allowed to co-exist at any moment. This enables a compiler or assembly-level programmer to artificially throtle the rate of thread creation so as not to congest the memory subsystem or the underlaying large register file from which the registers are dynamically allocated. Also, as the processor does not implement virtualization of some internal data structures yet (notably the *Family Table* and *Thread Table*), we are bound to use the *blocksize* parameter to prevent certain families from consuming all available internal resources.

TABLE III

PERFORMANCE RESULTS - *force-nop* COUNT

| setblock | UTLEON3 - micro-threading | | | | | |
|---|---|---|---|---|---|---|
| | unroll=1 | | unroll=2 | | unroll=4 | |
| 1 | 8375 | 81% | 4079 | 75% | 1991 | 66% |
| 4 | 856 | 30% | 335 | 20% | 88 | 8% |
| 8 | 171 | 8% | 84 | 6% | 88 | 8% |
| 16 | 147 | 7% | 84 | 6% | 88 | 8% |

The benchmark algorithm from Figure 11 was manually compiled and optimized. We have performed basic instruction scheduling to prevent pipeline interlock stalls, and unrolled the loops 2X and 4X. The results of program execution are shown in Table II. Each line shows cycle counts and performance change in % for a specific *setblock* value (i.e. the maximum number of co-existing threads from one family = the maximum number of threads from one family that can coexist in the thread table at once). The performance improvement between the original LEON3 and UTLEON3 in the legacy mode is given by different cache fetch policy - UTLEON3 fetches whole cachelines of 16 words on a cache miss whereas LEON3 fetches just one word.

The performance values for UTLEON3 in the micro-threaded mode reflect the efficiency of context switch mechanism. For *setblock=1* the cycle counts reflect the overhead due to filling in the thread table; this situation does not make use of fast context switching. The remaining lines show more realistic situations. The column *unroll=1* indicates that the executed threads were too short (in relation to the context switch time) to mask the memory access latencies. In the remaining cases the UTLEON3 in the micro-threaded mode solved the benchmark program faster than the original LEON3 processor.

TABLE II
PERFORMANCE RESULTS - EXECUTION CYCLE COUNT

| setblock | LEON3 | | UTLEON3 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | legacy | | micro-threading | | | | | |
| | | | | | unroll=1 | | unroll=2 | | unroll=4 | |
| 1 | 1831 | 100% | 1698 | 93% | 10329 | 564% | 5457 | 298% | 3019 | 165% |
| 4 | 1831 | 100% | 1698 | 93% | 2810 | 153% | 1705 | 93% | 1078 | 59% |
| 8 | 1831 | 100% | 1698 | 93% | 2017 | 110% | 1399 | 76% | 1078 | 59% |
| 16 | 1831 | 100% | 1698 | 93% | 1985 | 108% | 1399 | 76% | 1078 | 59% |

Table III shows the number of *force-nop* operations in the pipeline; this operation is issued when the scheduler cannot issue immediately any thread to the integer pipeline (the *active* queue is empty). The values in the table indicate the performance results can be still improved. For example, at present one *force-nop* is issued during each context switch, which can be avoided. More such situations can be discovered after a further analysis.

## VI. UTLEON3 VS. OPENSPARC T1/T2

The OpenSPARC T1 and T2 processors are open-source implementations of the UltraSPARC 2005 and 2007 architectures [1]. OpenSPARC T1 has been ported to Xilinx FPGAs; in the Xilinx XUPV5-LX110T Evaluation Platform T1 has been reported to operate at 62.5MHz in a 125MHz MicroBlaze system [10]. As the UTLEON3 processor is based on the SPARC v8 ISA it is only natural to compare it against the OpenSPARC processors.

UltraSPARC 2005, 2007 [1] architectures include support for Chip-Level Multithreading (CMT). The purpose of the CMT is to define multi-processing interface between the software and hardware. In CMT a processor (physical module which plugs into a system interconnect fabric) is a collection of physical cores. Physical core is a pipeline with caches and other associated hardware. One or more software threads - called strands - may be scheduled on one physical core. A strand is the software-visible state (PC, NPC, GP and FP registers, ASRs etc.) that the hardware must maintain in order to execute a software thread. From the ISA point of view the strand behaves like a virtual processor including its own MMU context. Therefore, the recommended programming model for CMT processors is either Posix Threads (pthreads) or OpenMP, which are both well-known industry standards. The incentive is to offer coarse-grained parallelism for task-level multitasking.

Contrary to that, the micro-threaded concurrency model employed in UTLEON3 is fine-grained. The goal of micro-threading is to extract instruction-level parallelism from existing sequential algorithms [11].

## VII. CONCLUSIONS

This paper has described an initial implementation of instruction set extensions for micro-threading in SPARC. The architecture of key functional blocks of the UTLEON3 processor have been presented together with implementation data for Xilinx XC2VP30 FPGA. The speedup of micro-threading in UTLEON3 over identical programs in LEON3 has been shown and discussed.

This is an on-going work; the final development of UTLEON3 will be made available to the research community at [12].

## ACKNOWLEDGMENT

This work was supported and funded by the European Commission under Project Apple-CORE No. FP7-ICT-215215, and by the Czech Ministry of Education under Project No. 7E08013. The paper reflects only the authors' view; neither the European Commission nor the Czech Ministry of Education are liable for any use that may be made of the information contained herein. For information about the Apple-CORE project see [12].

## REFERENCES

[1] T. Takayanagi, J. L. Shin, B. Petrick, J. Su, and A. S. Leon, "A dual-core 64b ultrasparc microprocessor for dense server applications," in *DAC*, S. Malik, L. Fix, and A. B. Kahng, Eds. ACM, 2004, pp. 673–677.

[2] P. Kongentira, K. Aingaran, and K. Olukotum, "Niagara: a 32-way multithreaded SPARC processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[3] K. D. Kissell, "MIPS MT: A multithreaded RISC architecture for embedded real-time processing," in *HiPEAC*, ser. Lecture Notes in Computer Science, P. Stenström, M. Dubois, M. Katevenis, R. Gupta, and T. Ungerer, Eds., vol. 4917. Springer, 2008, pp. 9–21.

[4] J. Gaisler, E. Catovic, and S. Habinc, *GRLIB IP Library User's Manual.* Gaisler Research, 2007.

[5] C. R. Jesshope and B. Luo, "Micro-threading: A new approach to future RISC." in *Proceedings of the 5th Australasian Computer Architecture Conference*. IEEE Computer Society press, 2000, pp. 34–41.

[6] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *ACM Comput. Surv.*, vol. 35, no. 1, pp. 29–63, 2003.

[7] C. Jesshope, "Scalable instruction-level parallelism," in *Computer Systems: Architectures, Modeling, and Simulation*. Springer Berlin / Heidelberg, 2004, pp. 383–392.

[8] C. R. Jesshope, "muTC - an intermediate language for programming chip multiprocessors," in *Asia-Pacific Computer Systems Architecture Conference*, 2006, pp. 147–160.

[9] Arvind and R. S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transaction on Computers*, vol. 39, no. 6, pp. 300–318, 1990.

[10] Sun Microsystems. RAMP retreat August, 2008 update. http://www.opensparc.net/publications/presentations/ramp-retreat-august-2008-update.html.

[11] C. R. Jesshope, J.-M. Philippe, and M. van Tol, "An architecture and protocol for the management of resources in ubiquitous and heterogeneous systems based on the svp model of concurrency," in *SAMOS*, ser. Lecture Notes in Computer Science, M. Berekovic, N. J. Dimopoulos, and S. Wong, Eds., vol. 5114. Springer, 2008, pp. 218–228.

[12] The Apple-CORE Consortium. Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs. http://www.apple-core.info.