UDK 004.383

A. M. SERGIYENKO
O. A. MOLCHANOV
M. K. ORLOVA

MICROCONTROLLER FOR THE LOGIC TASKS

A new SM16 microcontroller architecture is proposed which is intended for the logic-intensive applications in the field-programmable gate array (FPGA). The microcontroller has the stack architecture which provides the implementation of the most of instructions for a single clock cycle. The short but fast programs are derived due to the 16-bit instructions, which code up to three independent operations, and intensive use of the threaded code style. The framework is developed which compiles the program, simulates it, and translates to the ROM. The developed SM16 core with additional three-stack blocks, hash-table, and instructions that accelerate the execution of parsing operations is used for efficient XML-document processing and can be frequently reconfigured to the given document grammar set. The parsing speed equals to one byte per 24 clock cycles.

Keywords*:* VHDL, XML, parser, FPGA, stack processor, grammar, FSM

1. Introduction

The evolution of the central processing unit (CPU) microarchitectures during decades was intended for increasing the speed of the usual computations in different fields. For this purpose, the instruction level parallelism is exploited in the directions of pipelining, superscalar computations, data and instruction caching, branch prediction, dynamic scheduling, speculative calculations, etc. As a result, a single processor could perform averagely up to two or more instructions per clock cycle with the frequency of several gigahertz. These achievements are got at the costs of increasing the hardware volume by several decimal orders of magnitude and the power consumption up to dozens of Watts. But at present, the processor improvements stopped, in general, due to the Moore's law and the Dennard scaling law limitations [1].

The next microarchitecture evolution is expected in the form of the architecture improvements in the application specific fields. However, the RISC architecture will be likely prevalent one. One of the successful approaches is based on the complex application-specific instructions implemented in the field programmable gate array (FPGA) which stays near CPU [2].

The logic decision-intensive algorithms are implemented in the modern microprocessors ineffectively. This is due to the fact of the frequent pipeline stalls when the branches are mispredicted [1]. One of the solutions to this problem is to go back to the non-pipelined CPUs. When CPU has the application-specific instruction set, it can have the minimized hardware volume. Hence, it has the minimized clock period, and could implement a single instruction for a single clock cycle including the logic branch instructions, and doing without pipelining. Such a CPU is considered in this work.

2. XML-Document Parsing. A Case Study

The Web service is based on the queries which are specific XML-documents. The essence of the XML-filtering is to detect the XML-queries which satisfy the given set of grammars, which number can achieve more than thousands. Parsing XML-queries results in a significant slowdown in the Web service performance [3]. The experience of using XML in the databases shows that XML-parsing is a major bottleneck in the productivity gains and can increase the transaction costs up to ten times and more [4].

The grammar of a particular query type is expressed using XML-query languages such as XPath [5]. In general, such a grammar is presented as a tuple $G = (N, T, S, P)$. Here, N is a finite set

of the non-terminal symbols, T is a finite set of the terminal symbols, $S \subseteq N$ is a set of initial symbols, P is a set of rules in the form $X \rightarrow ar$, where $X \in N$, $a \in T$, and r is a regular expression over N. The rule says that X originates the sub-trees with the root a, and children that satisfy the expression r [6].

A simple XML-parsing algorithm that validates the document in terms of a given regular tree grammar is described in [6]. The algorithm is implemented in a stack finite state machine (FSM), which has three stacks: P, Y, and S. Stack P stores the symbol sets from N. Stack Y stores the sets of rules from P. Stack S stores the lists of symbol sets from N. The algorithm traverses the document tree in-depth and triggers the event phrases of a document, which include the opening and closing tags.

So, we can see that the XML-parsing algorithm is a set of FSMs, each of them represents a single grammar. Such an algorithm must contain a lot of logic and comparing operations, and is supported by such data structure like a stack.

The XML text filtering is a difficult problem because it has to support the real time processing of wide streams of various XML-requests. Different methods and accelerators have been proposed to improve this task. There are software accelerators, like an XML-filter (XFilter), which are built as FSM implemented in software [7]. According to the LazyDFA method, weakly deterministic FSM is dynamically constructed for the XML filtering [8].

FPGA is an efficient solution for the hardware filtering of XML-queries. FSM that is constructed for a specific set of grammars is implemented in FPGA in [9]. The systems based on stack FSM, which are compiled from the given grammars, are shown in [10–12]. But each exchange of the grammar set affords the redesign of the whole project which lasts a lot of time.

An FSM skeleton is proposed in [13], which is capable of being reconfigured quickly without the FPGA project redesigning. This FSM skeleton becomes a working FSM after loading the transition conditions corresponding to a specific set of XML-requests. The disadvantages of these approaches consist of the high hardware redundancy and limitations of the processed document class.

Comparing the mentioned approaches, the following conclusions are done. The hardware systems have the highest performance, but they are designed for a limited number of XML-grammars and their reconfiguration is long-lasting. The reconfigurable FSM-based hardware filtering systems have excessive hardware costs and focus on a particular class of grammars. More flexible architectures that can provide both the high throughput and the ability to quickly be reconfigured to the arbitrary XML-grammar are required. And such architecture can be based on the microcontroller adapted to the logic tasks.

3. Stack Processors for the Logic Algorithm Programming

The conclusions of the previous chapters show that the new processor architecture for the implementation of the logic decision-intensive algorithms is of demand. Such architecture has to be capable to implement effectively FSMs which perform large algorithm sets.

Usually, the microcontrollers do this task very well. In [14] the experience of FSM programming in the ARM Cortex microcontroller architecture is shown. It is proven in it that the best results are achieved in this RISC architecture when it has the minimized number of pipelined stages, which is equal to two. By this condition, the delay of the logic branch is minimized up to two clock cycles. Note that the number of stages in the RISC processors usually varies from three to five and more.

The processors which are implemented in FPGA must be adapted to its properties. These properties are the implementation of the logic functions in the look-up tables (LUTs), which input number varies from four to eight and more, a sufficient number of available pipelining registers, RAM blocks with the latent delay of two clock cycles. Besides, one has to take into account that the wire delays in FPGA achieve the value of 40% – 90% of the critical path delay.

These factors decrease the working frequency of the FPGA processors in 3 – 10 times comparing to the ASIC implementation of the same architecture. For example, the clone MIPSfpga of the popular RISC architecture has the maximum clock frequency of 60 MHz, and the microcontroller PIC32MZ with the same architecture has 200 MHz [15]. Note, that the processor with the similar architecture implemented in the advanced IC technology achieves the clock frequency up to one or more gigahertz. The RISC microprocessor cores of the same bit width which are adapted to the FPGA architecture like Xilinx Microblaze, Altera Nios, have much higher maximum clock frequency which achieves the value of 300 MHz [16].

For the implementation of application-specific systems in FPGA, it is important to get the configurable microcontrollers that have both minimized hardware volume and minimized length of its firmware because the amount of the embedded RAM blocks have significantly limited volume.

The stack processor architecture is distinguished among all microprocessor architectures. In this architecture, the registered RAM is substituted to the stack of registers, which communicates both with ALU and the return stack. The essence of this architecture consists of the implicit addressing of the working registers, direct implementation of the algorithms in Polish postfix notation, wide use of very quick procedure calls. As a result, the instructions of this architecture have a short length and can be implemented in a single clock cycle. Since these instructions support algorithms that actively use the stack addressing and subroutines, the programs that are composed for this processor occupy very small memory volume [17].

Many authors have developed several projects of stack processors, which are implemented in FPGA and are available for reproduction [18-20]. All of them have 16-bit instructions and process 16-bit data. It is shown in [20], that the stack processor has approximately 2.5 times less program length than the program for the Xilinx MicroBlaze processor in the logic branch-intensive computations. In addition, all stack processors allow the designer to increase the instruction set. In this case, the appropriate changes should be made to the description of the processor at the register transfer level.

Consequently, the architecture of the stack processor provides both firmware amount and hardware costs minimization. In addition, it is easy to develop compilers for such architecture, because, as a rule, its instruction set is a subset of the Forth language operators. It is known that this language is convenient both for grammatical parsing of lines and for the interpretation of high-level language operators. The stack processor assembly language has the same syntax as the Forth language [17]. Therefore, it is attractive to develop the stack processor architecture, which gives not only minimized hardware costs but also simplified implementation of user instructions, which are adapted to the logic computations.

4. SM16 Stack Processor Architecture

To solve the logic-intensive problems including one described in Chapter 2 the SM16 CPU architecture was developed. This 16-bit processor has a common dual-stack architecture [17], which structure is shown in Fig. 1. The eight-bit SM8 microcontroller described in [21] has been developed to implement the data communication protocols and it is a predecessor of SM16. Many instruction operations and other features were inherited from the SM8 architecture.

CPU includes a program counter (PC), Data RAM block, Program ROM block, an instruction register (IR), return address stack (Rstack), data stack (Dstack), ALU. The T, N registers are the top registers of the DStack. Register R is the top of the RStack, which also plays the role of a loop counter.

The program is loaded into the Program ROM during the FPGA configuration. It can be exchanged without reconfiguring FPGA using the memory programming tool. The dual-port Data RAM downloads the data to be processed from the external devices in the DMA mode. A and B are the index registers, and the peripheral register Ri serves for the interprocessor communications.

The HTable ROM stores the hash table for the transcoding the long tags found in the XML-documents into the numbers. The PStack, Ystack, and SStack stacks perform the same functions as the P, Y, and S stacks of the FSM described in Chapter 2.

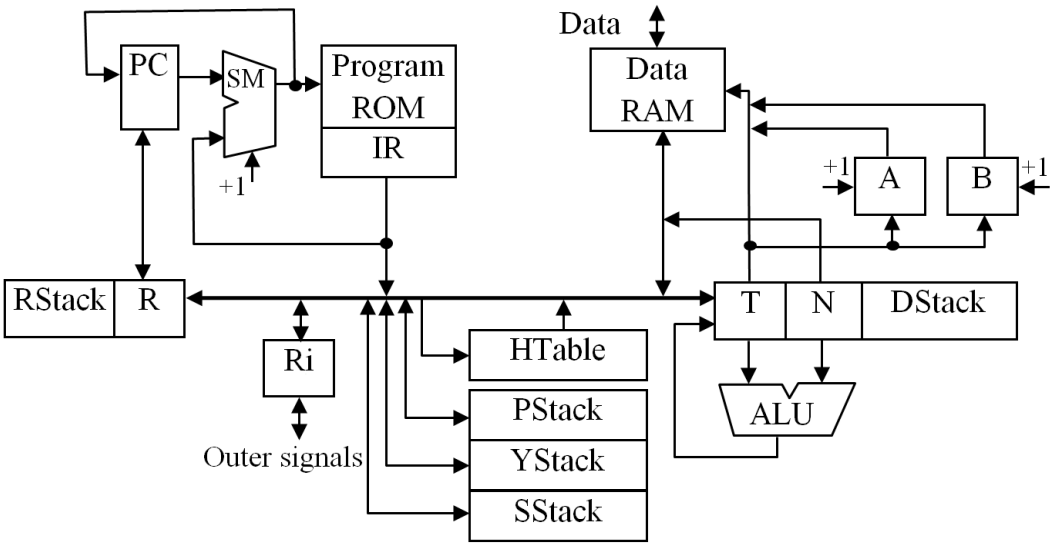

Fig. 1. SM16 processor structure

All instructions have a 16-bit width. The instruction has one to three op-code fields F1, F2, F3 (see Fig. 2). The field F1 codes the call, jump, return operations, counter decrement, and jump if it is not equal to zero (DJNZ). The field F2 codes all ALU operations, and F3 does data read and store operations in different modes including the register addressing with the address in the A or B register with the post-increment. The variable-length field D stores a constant, or a jump address.

The instructions are executed in a single cycle except for the data read and long constant loading instructions that are executed in two cycles. This feature makes the architecture friendly to the algorithms that are branch intensive. The processor can perform up to three operations F1, F2, F3 in a single clock cycle. For example, two instructions

```
: L1     @B+
        !A+ DJNZ L1
```

perform a loop, which takes only 3 clock cycles, and in which an array addressed by the B register is moved to another memory place addressed by the A register. Here, according to the Forth syntaxis, "`: L1`" means the label, `@B+`, `!A+` mean reading and writing operations, respectively, with the address post-increment.

The branch instruction lasts only a single clock cycle. This is achieved by the use of the ROM block output register as the instruction register IR and by feeding the next instruction address directly to the address input of this block bypassing the program counter PC (see Fig.1).

As a result, the logic branch-intensive algorithms are implemented without stalls. Consider an example of some FSM fragment coding which is shown in Fig.3. This example is shown in [14] as a result of the effective logic coding in the Cortex-M0+ microcontroller. Below, the code, proposed in [14] is compared with the similar code of SM16 processor.

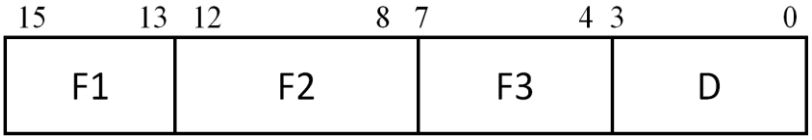

Fig. 2. SM16 instruction format

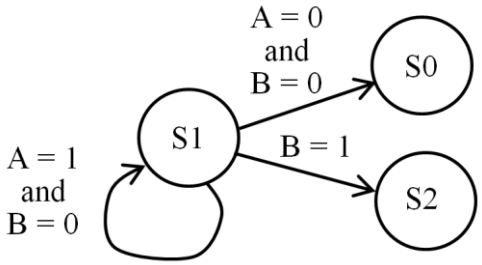

Fig. 3. Subgraph of some FSM

```
 ; Assembly code in ARM assembly syntax
S1:
    LDR     R0, [R4, #0x8]  ;  Read B
    CMP     R0, #1
    BEQ     S2              ;  Goto S2 if B = 1
    LDR     R0, [R4, #0x4]  ;  Read A
    CMP     R0, #0
    BEQ     S0              ;  Goto S0 if A = 0
    B       S1
```

Here, the code length is 14 bytes, the loop lasts 8 clock cycles taking into account that the branch operator takes 1 and 2 cycles for the undone and done branch respectively.

```
\ Assembly code in SM16 assembly syntax
   LIT 0    \ 0 to T for comparing
: S1
    INR B       \ Read B
    LIT 1 XOR   \ comparing to 1
    IF  S2      \ Goto S2 if B = 1
    INR A       \ Read A
    IF  S1 \ Goto S0 if A = 0 else Goto S1
: S0
```

Here, the code length is 10 bytes, the loop lasts 5 clock cycles. The instruction `INR` reads the respective peripheral register. We see that both the code length and the cycle duration are much less than in the effective example of the counterpart.

The stack processor architecture programming usually uses the threaded code style, i.e., the call instructions are placed very frequently. This instruction is implemented very quickly because the parameters are passed into the procedure in a natural fashion. So, the CALL and RET instructions are performed in the SR16 processor for a single clock cycle. The ability to insert a return operation in most instructions and combine it with a conditional branch reduces both the subroutine length and their duration. This helps both to speed-up the algorithm computing and to shorten the program dramatically. This makes it possible to obtain the programs of minimal length, which is important for the FPGA implementation. The next code shows an example of programming the deep if-then-else construction using these instructions.

```
: LONGIF
      TEST1
      LIT 1
      IF RET LIT 2
      TEST2
      IF RET LIT 3
;
```

This is the subroutine `LONGIF` which performs some logic testing, `TEST1, TEST2` are calls of the subroutines which check some complex conditions, the character '`;`' is the synonym of the RET instruction. As a result, each of the three testing outcomes returns figures 1, 2, or 3. This subroutine occupies only 12 bytes.

CPU has an interrupt system as well. Because the stack processor context has minimum volume, the interrupt overhead is also negligible. Due to large number of memory read instructions in the XML-parsing applications, the average run time of a single instruction is 1.2 clock cycles.

5. SM16 Processor Simulator

To develop the applications which perform the logic-intensive applications an SM16 processor simulator was designed. Its functions are: to compile the programs written in the SM16 assembly language, to load and simulate such a program, to inform about the syntax errors, to generate the VHDL files describing the Program ROM and Data RAM which contain the binary program and initial data codes, respectively.

This framework is able to read the document type definition (DTD) file which describes a set of XML-grammars and generates the SM16 program file to compute the respective parsing FSM. This program could both be modeled using the loaded XML-queries and be attached to the VHDL-model of the SM16 processor which is configured in FPGA.

The screenshot of the simulator frame is shown in Fig. 4.

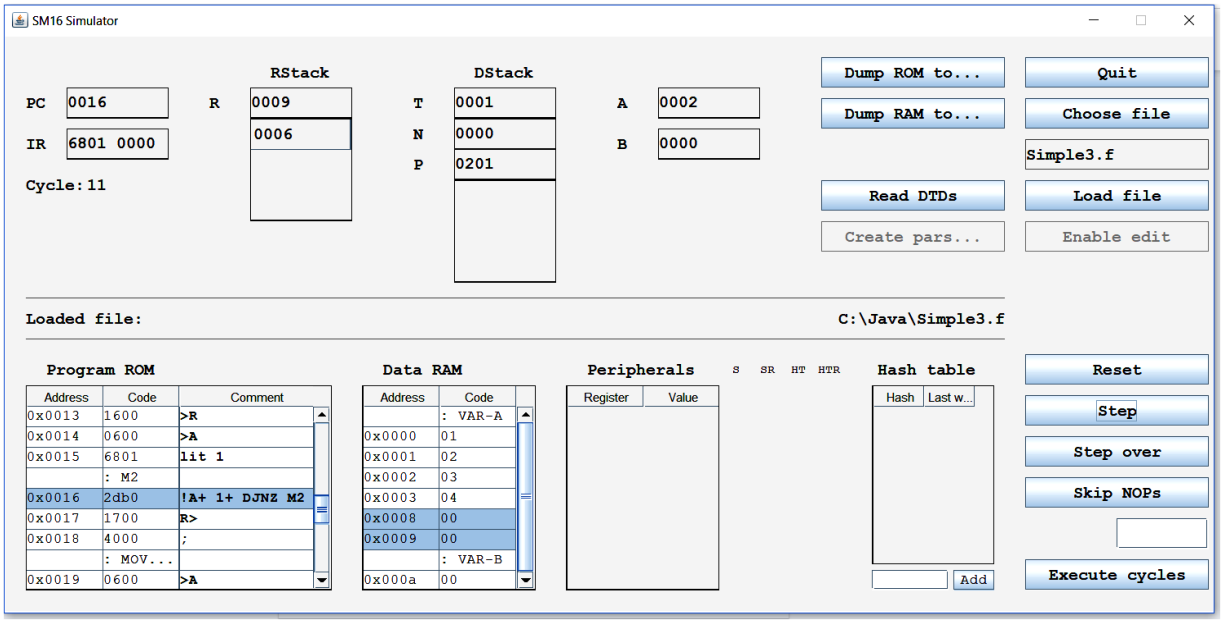

Fig.4. Screenshot of the SM16 processor simulator

6. Experimental Results

The SM16 CPU is described in VHDL and is synthesized for configuring in FPGAs of different series. The results of configuring are shown in Table 1.

Table 1. SM16 Processor Core Parameters

FPGA series	LUTs, ALMs	Registers	Maximum clock frequency, MHz
Xilinx Spartan-6	721	116	102
Xilinx Artix-7	767	119	135
Xilinx Kintex-7	773	119	190
Intel Cyclone V	1001	1080	97

When configuring in Xilinx FPGA, all the stacks are mapped into LUTs effectively preserving low hardware volume and high speed. Much worse results are achieved in the Intel Cyclone chip because these stacks are implemented in the sets of registers.

Table 2 presents the results of the SM16 CPU configured in Xilinx Spartan-6 FPGA comparing to the other processor cores. The table analysis shows that the SM16 processor has higher performance as the b16-small [19] and J1 processor [20], which are the stack processors as well at the cost of higher hardware volume. It has a much higher speed than the well-known MSP430 processor [22] and somewhat loses to the Microblaze processor [23]. Nevertheless, it should be noted that the SM16 processor has a larger instruction set which is adapted to the logic algorithms especially to handle the XML-documents. Of course, it has much higher hardware volume than the 8-bit stack-based microcontroller SM8 [24].

Table 2. Comparing Different Processor Cores

Processor core	Bit-width	Hardware costs (LUTs)	Maximum clock frequency, MHz	Speed, MIPS
b16-small	16	280	100	50 MIPS
J1	16	342	106	70 MIPS
MSP430	16	1240	65	25 MIPS
Microblaze	32	2046	130	174 DMIPS
SM8	8	181	140	94 MIPS
SM16	16	721	116	96 MIPS

There are a few hardware implementations of XML parsers comparing to the software ones. Only hardware parsers XPA [25], SCBXP [26] provide both regular expression filtering and building the XML parsing tree.

A SM16 microcontroller was developed which is programed to implement the same tasks that these hardware parsers do. For this purpose, the additional three stacks and hash table were added to the CPU core as well as the instructions which support the parsing process. Among them the instruction HASH performs the hash function calculating of the XML key words with the speed of one character per 3 clock cycles. So, the keywords are substituted to the indexes which are compared to ones stored in the precompiled hash table.

The given grammar set is loaded to the simulator framework which generates both the program ROM model and the hash table model described in VHDL. As a result, the SM16 microcontroller can compute the XML queries at the speed of approximately 7.5 megabytes per

second. Table III shows the characteristics of the mentioned hardware XML parsers and the proposed one.

Table 3. Hardware Characteristics of different XML parsers

XML-parser	Hardware costs (LUTs)	Clock frequency, MHz	Throughput, MB/s
XPA	9200	125	125
SCBXP	29200	33	200-500
SM16	880	180	7,5

Thus, the SM16 parser has only one and a half worse the performance-hardware ratio than the XRA and SCBXP parsers. However, an SM16-based solution can simultaneously support almost any number of XML grammars. This qualitatively distinguishes the developed parsing method and SM16 microcontroller from other hardware solutions.

7. Conclusion

A new SM16 microcontroller architecture is proposed which is intended for the logic-intensive applications in FPGA which is based on the stack architecture. The short but fast programs are derived due to the 16-bit instructions, which code up to three independent operations, and intensive use of the threaded code style. The framework is developed which compiles the program, simulates it, and translates to the ROM. The developed SM16 core with additional three stacks, hash-table, and instructions that accelerate the execution of parsing operations is used for efficient XML-document processing and can be frequently reconfigured to the given document grammar set. This system is not only capable of processing the XML-documents efficiently but also can be quickly reconfigured to process the documents of other grammars.

References

1. J. L. Hennessy, D. A. Patterson, “Computer Architecture. A Quantitative Approach” 6-th Ed. Morgan Kaufman Pub. 2019.
2. J. L. Hennessy, D. A. Patterson, “A New Golden Age for Computer Architecture”, Communications of the ACM. Vol. 62, 2019, pp. 48 –60.
3. M. R. Head, M. Govindaraju, R. van Engelen and W. Zhang, "Benchmarking XML Processors for Applications in Grid Web Services," SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, Tampa, FL, 2006, pp. 30–39.
4. N. Mattias, J. Jasmi, “XML Parsing: A Threat to Database Performance,” CIKM '03: Proc. of the 20th Int. Conf. on Information and Knowledge Management, 2003, pp. 175–178.
5. XML Path Language Version 1.0. Available at: http://www.w3.org/TR/xpath. Accessed 11 November 2019.
6. M. Murata, D. Lee, M. Mani, K. Kawaguchi, “Taxonomy of XML schema languages using formal language theory.” ACM Trans. on Internet Technology, Vol. 5, 2005, Iss. 4, pp. 660–704.
7. M. Altinel, M. J. Franklin, “Efficient Filtering of XML Documents for Selective Dissemination of Information,” VLDB’00: Proc. of the 26-th International Conference on Very Large Data Bases, 2000, pp. 53–64.
8. T. J. Green, A. Gupta, G. Miklau, M. Onizuka, D. Suciu, “Processing XML streams with deterministic automata and stream indexes, “TODS: ACM Trans.on Database Systems, 2004, pp. 752–788.

9. J. V. Lunteren, T. Engbersen, J. Bostian, B. Carey, C. Larsson, “XML accelerator engine.” 1st International Workshop on High Performance XML Processing, 2004, pp. 1–4.
10. R. Müller, J. Teubner, G. Alonso, “Streams on wires — a query compiler for FPGAs,” VLDB Endowment'09: Proc. of the Very Large Data Base Endowment, 2009, Vol. 1, Issue 2, pp. 229–240.
11. R. Moussalli, M. Salloum, W. Najjar, V. Tsotras, “Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs.” ICDE’11: Proc. of the IEEE 27th International Conference on Data Engineering, 2011, pp. 948-959.
12. A. Mitra, M. Vieira, P. Bakalov, W. Najjar, V. Tsotras, “Boosting XML Filtering with a Scalable FPGA-based Architecture.” CIDR–2009: Proc. of the 4th Biennal Conference on Innovative Data Systems Research, 2009, pp. 1–10.
13. J. Teubner, L. Woods, C. Nie, “XLynx — an FPGA-based XML filter for hybrid XQuery processing.” ACM TODS: ACM Transactions on Database Systems, 2013, Vol. 38, Issue 4, Article No. 23, pp 1–39.
14. J. Yiu, “Software based Finite State Machine (FSM) with general purpose processors. “ ARM: White paper, January 2013, 17 p.
15. S. L. Harris, D. M. Harris, D. Chaver, R. Owen, et al., "MIPSfpga: using a commercial MIPS soft-core in computer architecture education," in IET Circuits, Devices & Systems, Vol. 11, 2017, No. 4, pp. 283-291.
16. Processor Design. System-on-Chip Computing for ASICs and FPGAs. J. Nurmi (ed.), Springer, 2007.
17. P. Koopman, “Stack computers: the new wave”, Ellis Horwood, Mountain View Press, CA., 1989.
18. P. H. W. Leong, P. K. Tsang and T. K. Lee, "A FPGA based Forth microprocessor," Proc. IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, USA, 1998, pp. 254-255.
19. B. Paysan, “b16 — small — Less is More”, Proc. EuroForth 2004, Jul. 9, 2006.
20. J. Bowman, W. Garage, “J1: a small Forth CPU Core for FPGAs”, Proc. EuroForth’2010, pp. 1-4, January, 2010.
21. Sergiyenko A., Molchanov O., Orlova M.: Nano-Processor for the Small Tasks. 2019 IEEE 39th Int. Conf. on Electronics and Nanotechnology (ELNANO), pp. 674-677. IEEE (2019).
22. O. Girard, “OpenMSP430.” OpenCores, Rev. 1.13. 2013. http://opencores.org. Accessed 11 November 2019.
23. V. Kale, “Using the MicroBlaze Processor to Accelerate Cost-Sensitive Embedded System Development.” Xilinx, WP469, V.1.0.1, 2016. https://www.xilinx.com/products/design-tools/microblaze.html #documentation. Accessed 11 November 2019.
24. O. Molchanov, M. Orlova, A.Sergiyenko, “Software/Hardware Co-design of the Microprocessor for the Serial Port Communications,” Advances in Computer Science for Engineering and Education II, Hu Z., Petoukhov S., Dychka I., He M. -Eds., Springer, 2020, pp. 238–246.
25. Z. Dai, N. Ni, J. Zhu, “A 1 Cycle-Per-Byte XML Parsing Accelerator.” FPGA '10: Proc. of the 18th ann. ACM/SIGDA int. symp. on Field programmable gate arrays. Feb., 2010, pp. 199–208.
26. F. El-Hassan and D. Ionescu, "SCBXP: An Efficient CAM-Based XML Parsing Technique in Hardware Environments" in IEEE Transactions on Parallel & Distributed Systems. Vol. 22, 2011. No. 11, pp. 1879-1887.