2019 IEEE 39th International Conference on Electronics and Nanotechnology (ELNANO). 2019/4/16. P.674-677.

Nano-Processor for the Small Tasks

Anatoliy Sergiyenko
Dept. of Computer Engineering
Igor Sikorsky Kyiv Polytechnic Institute
Kyiv, Ukraine
aser@comsys.kpi.ua

Oleksii Molchanov
Dept. of System Programming and Specialized Computer Systems
Igor Sikorsky Kyiv Polytechnic Institute
Kyiv, Ukraine
oleksii.molchanov@gmail.com

Maria Orlova
Dept. of System Programming and Specialized Computer Systems
Igor Sikorsky Kyiv Polytechnic Institute
Kyiv, Ukraine

***Abstract*—The eight-bit stack processor architecture is proposed, which is designed for the FPGA implementation. The microprocessor with this architecture has small hardware costs, reduced software amount, and ability to add up to hundred new user instructions to its instruction set. The microprocessor architecture is adapted for programming the serial port communications and is able to perform the data stream parsing. It is effectively used for the Internet of Things applications.**

Keywords—stack processor, Forth, FPGA, VHDL, IoT

I. INTRODUCTION

In recent years, the Internet of Things (IoT) technology is expanded dramatically. The main feature of it is the transfer of the data flows from far-end applied devices through the communication channels in real time. These devices usually have to provide both high speed and small energy consumption, as well as low cost. They contain some processor core which implements the proper communication protocol, too. Many IoT solutions are worth being implemented in the field programmable gate array (FPGA) because of its high flexibility, minimized energy consumption, and far distance upgrading possibility [1].

Many new FPGA series are intended for the IoT implementation. The examples are Intel Max-10, Cyclone-5, Xilinx Spartan-7, Lattice CrossLink, iCE40, MachXO3. These FPGAs have small footprint and minimized energy consumption. Some of them have built-in configuration ROM. But the FPGA expanding at t he field of IoT market is partially limited by the absence of the effective solutions which provide the interface fulfilment. The usual solution is to configure the 32-bit microprocessor soft core, arranging the device by the proper DRAM, and EEPROM. These DRAM and EEPROM provide storing and implementation the linux-like OS, whith the usual IP protocol stack. But the mentioned above FPGA series have the limited hardware volume to implement this solution except of some high-end chips. Besides, such a solution does not provide the minimum cost and energy consumption, and needs some period of time for the OS loading from EEPROM to DRAM, which expands dramatically its ready-to-use time.

In the work, a microprocessor soft core is proposed, which fits all the FPGA series mentioned above, and provides the effective IoT device implementation. It has well-known stack architecture, which is effectively programmed by the Forth language [2]. It is distinguished in small hardware volume, effective instruction set, possibility of the custom instruction addition.

II. MICROPROCESSOR ARCHITECTURE FOR IOT IMPLEMENTATION

The IoT device usually implements a set of serial interfaces such as I2C, SPI, Ethernet and others. At the same time, it is more rational to use the microprocessor core, which has both small hardware costs, and simple programming and debugging procedures. In addition, such a core can replace the finite state machines, which are needed for control of a designed system. Usually, the RISC processors can be considered as those, which match the described characteristics.

The FPGA manufacturers propose one or a set of the RISC processor architectures which are configured in the proper FPGAs. So, Xilinx proposes the Picoblaze and Microblaze architectures. The first of them is the 8-bit architecture with very small hardware volume but it has limited possibilities, and it is not effective for the IoT protocol implementation. The second of them is 32-bit architecture, which is able to implement the linux OS, but has the increased hardware volume and needs the outer DRAM attachment. The Altera Nios-II core has the same properties. Some clones of the common microcontrollers, such as i8051, are effectively used for the IoT functions, but they have the increased hardware volume because their architecture properties does not fit effectively the FPGA features [3-5].

The architecture is proposed in [5], which is named as Nanoblaze. Due to its characteristics of speed and hardware volume, it occupies the place between Picoblaze and Microblaze architectures. Probably, it is fit well to the IoT purposes. But it had not achieved the proper expanding due to the fact that its name falls in the infringement of the Xilinx copyright. Nevertheless, the prefix Nano effectively characterizes the processor architecture name standing for the small footprint but high-speed architecture.

For the implementation of the IoT system in FPGA, it is important to have a configurable microcontroller with both minimized hardware and software. This is dictated by the fact that the memory blocks, which are embedded in FPGA, have significantly limited volume. It is desirable to have such a microcontroller which instruction set can be manually adjusted by the programmer to the needs of the project, to simplify programming, allow the program subroutines to be reused and, as a result, to minimize the program length. Its instruction set has to be adapted for scheduling the data transfer through the interfaces. The architecture of such a microprocessor is proposed in this work.

The stack processor architecture is distinguished among all microprocessor architectures. Its instruction set differs in

that the operands have implicit addressing because they are usually placed in a few fixed stack registers. Such instructions have a short length because they have the implicit register addressing. Since these instructions support the algorithms that actively use the stack addressing, the programs that are composed for this processor occupy very small memory volume [6].

Various authors have developed several projects of stack processors, which are implemented in the FPGA and which are available for the reproduction [7-9]. All of them have the 16-bit instructions and process the 16-bit data. It is shown in [9], that the stack processor has approximately 2.5 times less program length than the program for the Xilinx MicroBlaze processor when the IoT project is designed. Here, the stack processor implements the data exchange protocol through the Ethernet interface. Additionally, all stack processors allow the designer to extend the instruction set. To do so, 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 the firmware amount and the hardware costs minimization. In addition, it is easy to develop the compiler for such an architecture, because, as a rule, its instruction set is a subset of the Forth language commands [10]. It is known that this language is convenient for both 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 as well [6]. Therefore, it is attractive to develop the stack processor architecture, which gives not only the minimized hardware costs but also the simplified implementation of the user instructions, which are adapted to the serial port communications.

III. NANO-PROCESSOR ARCHITECTURE

The nano-processor architecture SM8 was designed, which satisfies the stack processor properties. The structure of the developed SM8 microprocessor is shown in Fig. 1. This 8-bit processor has the well-known two-stack architecture [6]. It consists of a program counter PC, Data RAM, Program ROM, instruction register IR, user instruction encoder UIE, return address stack RS, data stack DS, ALU and peripheral registers R0,..., Rn, n < 32. The registers T, N are the top registers of the D-stack and are designed to store the operands and the ALU results. The Program ROM has the volume up to 7936 bytes, and the Data RAM has up to 256 bytes, and both of them have a common address space.

The main SM8 microprocessor instructions are sampled in Table 1. All instructions, except CALL, LIT, and IF, have the length of 8 bits. The branch and input-output instructions are executed in two cycles, and the rest of the instructions are single cycle instructions. Due to the frequent use of the CALL, LIT, and IF instructions, the average duration of one instruction execution is equal to 1.5 clock cycles.

The user-defined instructions are implemented as follows. First, the instruction code is associated with the specified address in the user subroutine library, where the user-defined subroutine is located. Second, when the control-flow approaches this instruction, the instruction code is written to the IR register. Then, the code is encoded by UIE to the address of the subroutine, associated with it. The return address (address of the next instruction) is saved in the R stack.

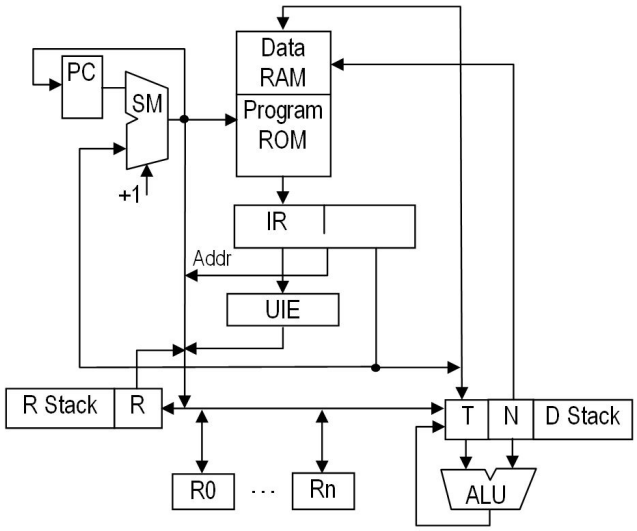

Fig. 1. Structure of the SM8 microprocessor

After that, the control-flow is passed to the first instruction of the subroutine (i.e., the subroutine is 'called') and all instructions in it are executed.

TABLE I. INSTRUCTIONS SET OF THE SM8 MICROPROCESSOR

Name	Instruction	Description
CALL	001 Addr	PC+1 -> R, PC = Addr, subroutine call
INR	010 n	Rn -> T, data receiving
OUTR	011 n	Rn = T, data sending
NOP	0000 0000	No operation
LIT	0000 0001 B	B -> T, constant input
IF	0000 0010 D	PC = PC + D by T = 0, else PC = PC + 1
DUP	0000 0110	N = T
SWAP	0000 1001	X = T, T = N, N = X
@	0000 1010	T = RAM[T], memory reading
!	0000 1011	RAM[T] = N, memory writing, T ->
R>	0000 1100	R -> T
>R	0000 1110	T -> R
RET	0000 1101	R -> PC, return from the subroutine
DROP	0000 1111	T ->, stack purge
NOT	0001 0000	T = not T
OR	0001 0001	T = T or N
AND	0001 0010	T = T and N
XOR	0001 0011	T = T xor N
ADD	0001 1000	T = T + N
INC	0001 1001	T = T + 1
SUB	0001 1010	T = T – N
DEC	0001 1011	T = T - 1
	1xxx xxxx	User instruction

The return of the control flow from the subroutine to the next instruction is performed by the RET instruction. This subroutine can also read and process the operand fields that follow the byte of the opcode. But the return address in the R register must be properly corrected.

The user-defined instruction is coded by a single byte comparing to the two-byte CALL instruction. Therefore, these instructions can save the software memory volume comparing to the equivalent instruction implementation using the CALL instruction.

The user-defined instructions can be stored in both Program ROM and Data RAM. Thus, a microprocessor can store a certain dynamic data processing script, which is formed by the user instructions and respective data bytes for them. It can perform a line parsing as well. For example, this line can be a string of decimal calculator operations and digits.

A common problem for many FPGA-based architectures is the reconfiguration process. Usually, the need for the reconfiguration leads to the recompilation of the hardware netlist, which is a very CPU-intensive and time-consuming task (it can last from minutes to hours). The solution to this problem is presented in several works [11], [12] as an implementation of task-specific architectures that can be reconfigured 'on-the-fly'. For example, in [12] the authors propose the segment-based architecture for the XML filtering. The sequence of configured segments implements the XPath pattern of the interesting part of the whole XML. This pattern can be changed 'on-the-fly', and the hardware reconfiguration takes from nano- to microseconds.

Another approach is implemented in SM8. The processing script, which is saved in RAM, can be rewritten or loaded from the other memory, for example, from ROM. Such a simple rewriting allows for the user the system exchanging in terms of microseconds. As far as such action is performed in a synchronization point, neither data loss nor wrong behavior happens. In such a way, the segments in [12] are reconfigured without stop of the input data processing with preserving all the currently processed XML node tree parts.

An assembler was developed for programming the SM8 microprocessor. The assembler is written in Java and is called from the command line. Below, an example of a program in the SM8 assembly language is shown, which performs a single-byte transfer to the I2C port.

```
DEFINE nap 9          \ memory address width
DEFINE WAITRDY 82h    \ user instruction – wait for
                      \ port is ready
DEFINE DELAYN  83h    \ user instruction – delay for
                      \ N cycles
EQU START  2
EQU A_SEND 4
EQU D_SEND 5
EQU STOP        12
EQU PAUSE  15
ORG 256               \ program segment begin
\Write byte to I2C
: WR2I2C (r1 - I2C addr, r2 - inner addr,
          r3 - byte, r8 - I2C data,
          r9 - I2C control)
       lit START outr r9
       inr r1 outr r8 lit A_SEND outr r9 WAITRDY
       inr r2 outr r8 lit D_SEND outr r9 WAITRDY
       inr r3 outr r8 lit D_SEND outr r9 WAITRDY
       lit STOP   outr r9
       lit PAUSE  outr r9
       lit 100    DELAYN  xor if END
;
: DELAYN           (N --  - N cycles)
       dec dup ifn  DELAYN
;
: WAITRDY           (do while rdy=1)
       inr r10           \ 0-th bit = rdy
       lit 1 and if WAITRDY
;
: END
```

The assembly language of the SM8 core uses the syntax of the Forth language. Therefore, the comments here are enclosed in parentheses or followed after a backslash.

The label follows a colon. Operators and literals are separated by spaces. A semicolon indicates the subroutine return instruction. Some special operators, called the pragmas, are used in the script for the special purpose. They define the association between identifiers and literals, macros, initial addresses of the memory segments.

As it is seen from the example above, none of the subroutines contain the RET instruction. It is explained by the fact that the semicolon sign represents the RET instruction in the Forth language. The user also can specify its own RETs in his subroutine if needed.

The assembler generates two VHDL files, which contain the data and programs in the memory and the user instruction encoder content. When the processor is configured in the Intel FPGA, the memory loading file is generated. As a result, this assembly language by its user properties occupies an intermediate position between the usual assembly language and the high-level language. Thanks to this, the writing and debugging of programs are significantly accelerated. Besides, the VHDL model of the SM8 microprocessor core is equipped by a disassembler. Such a feature significantly simplifies the program debugging in the VHDL simulator.

After two VHDL files are generated, they become part of VHDL project of developed system with SM8 microprocessor core. Peripheral devices and connections between them and registers in microprocessor are described in that project as well.

IV. Experimental Results

The SM8 microprocessor core is described in VHDL and has synthesized for different FPGA circuits. Table II presents the results of the SM8 microcontroller synthesis in the Xilinx Spartan-6 FPGA while setting the optimization parameters for hardware costs. Also, the parameters of the microprocessors, which were synthesized in the same conditions, are presented in this table for a comparison.

The analysis of the table shows that the SM8 microprocessor has the lowest hardware costs in the look-up tables (LUTs), and the highest speed in millions of instructions per second (MIPS) among the stack processors. This is explained by the fact that the reduction of the data bit width up to eight digits reduces both hardware costs and delay in ALU.

Table III shows the results of the configuration of this core in the Intel-Altera FPGAs. They are compared to the results of the Nios II processor configurations, which are available from the open resources, not taking into account its peripheral and memory management units. This comparison shows that the SM8 core provides much less hardware

volume by the approximately same clock frequency. The estimated power consumption of this core is equal to 50 mW, which dynamic ratio by the highest clock frequency is equal to only 13 mW when configuring in the MAX10 device. Such nano-processor architecture helps to implement the IoT system in the smallest FPGAs providing both the small cost and minimized power consumption.

At present, this core is utilized in the FM radio station exciter as a communication processor, which provides the control and data transfers through the serial interfaces like I2C, SPI, RS232, Ethernet. Its use showed its high effectiveness especially in the firmware development and debugging.

TABLE II. MICROPROCESSOR CORE PARAMETERS IN XILINX FPGA

Micro-processor	Instruction bit width	Hardware, LUTs	Max. clock frequency, MHz	Speed, MIPS
FS8051 [13]	8, 16, 24	1293	89	30
KCPSM6 [4]	18	87	140	70
MSL16 [6]	16	235	100	67
b16-small [7]	16	280	100	50
J1 [8]	16	342	106	70
SM8	8, 16	181	140	94

TABLE III. MICROPROCESSOR CORE PARAMETERS IN INTEL FPGA

Micro-processor	FPGA	Hardware volume	Max. clock frequency, MHz	Speed, MIPS
Nios II/f [14]	MAX10	2268 LE	150	135
Nios II/f [14]	Cyclone 5	867 ALM	170	150
SM8	MAX10	1164 LE	150	100
SM8	Cyclone 5	210 ALM	205	140

V. CONCLUSION

The proposed SM8 microprocessor core has small hardware costs at high performance and reduced hardware volume. It is designed to implement the communication functions of the IoT devices. The core is described in VHDL and can be implemented in an FPGA of any series. The programmer has the ability to add his own instructions to the instruction set without changing the core description. The developed assembler provides to develop and compile the programs written in the Forth language style. This simplifies the design of devices that implement the protocols for the serial port communications through interfaces such as RS232, I2C, SPI, Ethernet.

REFERENCES

[1] A. Engel and A. Koch, "Heterogeneous Wireless Sensor Nodes that Target the Internet of Things," *IEEE Micro*, vol. 36, no. 6, pp. 8-15, 2016.

[2] P. Koopman, "Stack Computers: The New Wave", Elis Horwood, 1989.

[3] J. Kylliainen, T. Ahonen, and J. Nurmi, "General-Purpose Embedded Processor Cores - The COFFEE RISC Example" in *Processor Design. System-on-Chip Computing for ASICs and FPGAs*, J. Nurmi, Ed., Kluwer Academic Publishers / Springer Publishers, 2007, pp. 83-100.

[4] K. Chapman, "PicoBlaze for Spartan-6, Virtex-6, and 7-Series (KCPSM6)", 2013, Accessed: Dec. 14, 2018. [Online]. Available: http://bcf.usc.edu/~franzke/courses/ee209/2017-03/tools/KCPSM6_User_Guide_30Sept13.pdf

[5] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, 4th ed. Berlin, Germany: Springer-Verlag, 2014.

[6] P. Koopman, *Stack computers: the new wave*. Hemel Hempstead, UK: Ellis Horwood, 1989.

[7] P. H. W. Leong and P. K. Tsang, "A FPGA Based Forth Microprocessor," in *Proc. of the IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 1998, pp. 254-255.

[8] B. Paysan. (Jul. 9, 2016). b16-small — Less is More. Presented at EuroForth 2004. [Online]. Available: https://bernd-paysan.de/b16-small.pdf

[9] J. Bowman and W. Garage, "J1: a small Forth CPU Core for FPGAs," in *Proc. of the EuroForth'2010*, Jan. 2010, pp. 43-46.

[10] M. Kelly and N. Spies, *Forth: A Text and Reference*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1986.

[11] M. Najafi, M. Sadoghi, and H.-A. Jacobsen, "Configurable Hardware-based Streaming Architecture using Online Programmable-Block," in *2015 IEEE 31st International Conference on Data Engineering (ICDE 2015)*, Seoul, South Korea, Apr. 13-17, 2005, pp. 819-830.

[12] J. Teubner, L. Woods, and C. Nie, "XLynx — An FPGA-based XML Filter for Hybrid XQuery Processing", *ACM Transactions on Database Systems (TODS)*, vol. 38, no. 4, Nov. 2013, Art. no. 23.

[13] O. Maslennikov, J. Shevtshenko, and A. Segyienko, "Configurable microcontroller array," in *Proc. of the Parallel Computing in Electrical Engineering (PARELEC'02)*, Warsaw, Poland, Sep. 25, 2002, pp. 47-49.

[14] Intel, Santa Clara, CA, USA. *Nios II Performance Benchmarks. DS-N28162004*. (2018). Accessed: Dec. 15, 2018. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ds/ds_nios2_perf.pdf