Third International Conference
Modelling and Development of Intelligent Systems
October 10 - 12, 2013
"Lucian Blaga" University
Sibiu - Romania

Algebraic model for the CPU arithmetic unit behaviour

Anca Vasilescu

Abstract

Modern computer systems are regarded as a sum of interconnected and communicating resources. Both the design and the operation of each of these resources, and the global behaviour and performance of the entire computer system are equally important. This approach points to a component-based analysis and development of such systems, each component being able to be specified and verified as a specific agent. Formal methods represent a reliable solution for systematically and exhaustively studying the specific agents involved in describing computer components behaviour, providing the appropriate tools for both the agents' environment modeling and the target agents' properties formal verification.

An algebraic formal framework for modelling the interconnecting processes involved in the agents' description is advanced here using the SCCS process algebra and its corresponding automatic verification benchmark, CWB-NC. In this paper we add a new component model to our formal framework by considering the CPU arithmetic unit. The original approach followed in the present paper consists in developing an SCCS based algebraic model for the arithmetic unit behaviour. The authors' contributions are both the definitions of the SCCS agents for modelling the target behaviour and the proofs for the bisimulation equivalence between those agents. Adding these results to other similar results obtained in our framework, we have important prerequisites in the future work for modelling the behaviour of the entire ALU consisting of arithmetic unit, logic unit and specific control circuits.

1 Introduction

Computer architecture provides a structured and organized view upon the computer system hardware components. With respect to the final users' demands, better solutions for designing and assembling hardware components are investigated. These solutions usually target the increasing system scalability, the components' accurate operation or reducing components' assembling costs.

Modern computer systems are regarded as a sum of interconnected and communicating resources. Both the design and the operation of each of these resources, and the global behaviour and performance of the entire computer system are equally important. This approach points to a component-based analysis and development of such systems, each component being able to be specified and verified as a specific agent.

Formal methods represent a reliable solution for systematically and exhaustively studying the specific agents involved in describing computer components behaviour, providing the appropriate tools for both the agents' environment modeling and the target agents' properties formal verification.

Considering the computer architecture description at the digital logic level, the agent-based approach is applied in this paper to cover both the digital logic circuits design and verification. An algebraic formal framework for modelling the interconnecting processes involved in the agents' description is advanced here using the SCCS process algebra [3] and its corresponding automatic verification benchmark, CWB-NC [22]. Using the operational semantics of the given SCCS algebra, we may evaluate and formal verify how the proposed implementation-based model relates to the intended specification-based definitions of the given components behaviour. As an extra mark for our model correctness, an automatic verification of the target agents' equivalence is applied using the CWB-NC tool.

This formal framework represents our research interest for obtaining an algebraic model for the entire computer operation based on the interconnected hardware components. Our main results have already aimed to a set of hardware components, as follows: counter registers [11], memory component [14], [15], logic part of the processor arithmetic logic unit [19].

In this paper we add a new component model to our formal framework by considering the other main part of the processor ALU, namely the arithmetic unit. The computer's Arithmetic-Logic Unit (ALU) is a Combinational Logic Circuit (CLC), a part of the execution unit as a core component of all Central Processing Units (CPUs) of modern computers. A concrete structure of the ALU is considered in order to achieve the most addressed arithmetic operations. The original approach followed in the present paper consists in developing an SCCS based algebraic model for the arithmetic unit (AU) behaviour. The authors' contributions are both the definitions of the SCCS agents for modelling the AU behaviour and the proofs for the bisimulation equivalence between those agents. Jointly these results and the results from [19] will be important prerequisites in our future work for modelling the behaviour of the entire ALU consisting of arithmetic unit, logic unit and specific control circuits.

2 Preliminaries

This section considerations are following our presentations of the same subjects made in [19].

2.1 Arithmetic Logic Unit

The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU for central processing unit [4], [9]. The CPU is made up of three major parts, as follows: control, register set and arithmetic logic unit (ALU). The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit performs the required microoperations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

Instead of heaving individual registers performing the microoperations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviated ALU [4], [9]. To perform a microoperation, the content of specified registers are placed in the inputs of the common ALU. The ALU performs an operation and the result of the operation is then transferred to a destination register. The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period. The shift microoperations are often performed in a separate unit, but sometimes the shift unit is made part of the overall ALU.

For the target of this paper we consider a specific structure of the computer's ALU represented in Figure 1, adapted from [6].

This diagram is divided into three sections: Logic unit, Arithmetic unit and Decoder. The inputs are a, b, S_0 and S_1. The a and b inputs are used as the regular, 1-bit inputs for all operations. The S inputs operate as enable lines since for each of the four possible combinations of S values, only one of

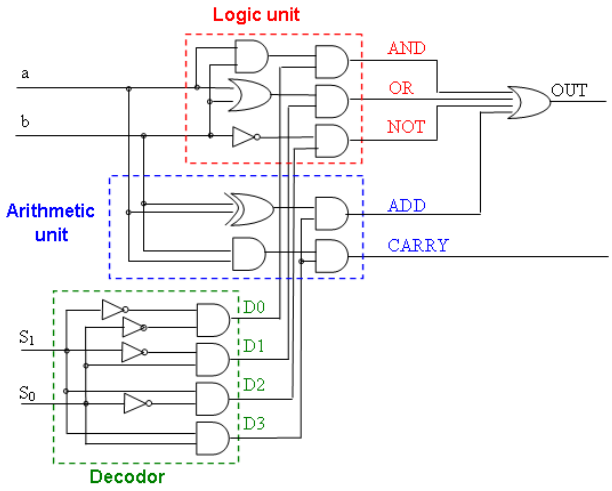

Figure 1: 1-bit UAL

the decoder outputs $D0$, $D1$, $D2$, $D3$ will be "turned on". Thus, the function of the Decoder subpart is to figure out which of the four operations will be done: AND, OR, NOT or ADD. On the right side of the circuit, all of the outputs are ORed together. However, only one of the four inputs of the OR gate could potentially be an 1 due to the enable lines.

For practical operations an 8-bit ALU is more convenient. For this, the previous diagram needs to be repeated 8 times, eventually considering also the specific lines for managing the carry bit.

In the next sections we will consider in details the arithmetic part of this structure as a collection of two boolean operations, $a\,XOR\,b$ and $a\,AND\,b$, we will define an algebraic model for this unit behaviour and we will prove its correctness.

2.2 Process algebra SCCS

The process algebra SCCS, namely *Synchronous Calculus of Communicating Systems* [1] is derived from CCS, especially for achieving the synchronous interaction in the framework of modelling the concurrent communicating processes. Both in CCS and in SCCS, processes are built from a set of atomic actions A. Denoting the set of labels for these actions by Λ, a CCS action is either (1) a *name* or an input on $a \in \Lambda$ denoted by a, (2) a *coname* or an output on $a \in \Lambda$ denoted by $\overline{a}$ or $\tilde{}a$ or (3) an internal on $a \in \Lambda$ denoted by τ. In SCCS the *names* together with the *conames* are called the *particulate actions*, while an *action* $\alpha \in \Lambda^*$ can be expressed uniquely (up to order) as a finite product $a_1^{z_1} a_2^{z_2} ...$ (with $z_i \neq 0$) of powers of names. Note the usual convention that $a^{-n} = \overline{a}^n$ and that the action **1** in SCCS is the action τ from CCS and it is identified in SCCS with the empty product. An SCCS *process* P is defined with the syntax:

P	::= nil	termination
	| α:P	prefixing
	| P+P	external choice
	| P $\times$ P	product, synchronous composition
	| P$\backslash L$	restriction, $L \subseteq A \cup \overline{A}$
	| P$[f]$	relabelling with the morphism $f : A \cup \overline{A} \to A \cup \overline{A}$

In this grammar, the restriction is inherited from CCS. There is also an SCCS specific restriction denoted by the $\restriction$ operator and structural related with the CCS operator by P$\backslash L$ = P $\restriction$ E where E = (A-L)* is the submonoid of A generated by the set difference A-L. By definition, the P $\restriction$ E agent is

forced to execute only the actions from the set E as the external actions and the agent P$\backslash L$ is forced to not execute the actions from the set L, except as the internal actions.

The operational semantics for SCCS is given via inference rules that define the transition available to SCSS processes. Combining the product and the restriction, SCCS calculus defines the synchronous interaction as a multi-way synchronization among processes.

3 The model for the arithmetic unit behaviour

As we have already mentioned in Preliminaries, we consider the arithmetic part of the arithmetic logic unit represented in the previous Figure 1. For the diagrammatic representation of this part we use in the next Figure 2 our own software LCD [13] developed for representing the digital-logic circuits and simulating their behaviour.

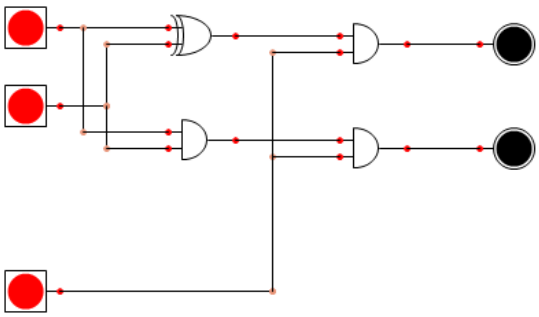

Figure 2: 1-bit UAL - Arithmetic part

3.1 The algebraic model

As the main results of this paper, we define in this section the algebraic model for the AU behaviour based on three kinds of agents: (1) the basic agents - corresponding to the main logic gates AND and XOR, (2) the enabling agents - corresponding to the connection of the AU with the decoder and (3) the arithmetic agents - corresponding to the two AND gates level.

(1) The basic agents are: ANDab and XORab. Their definitions are:

$$\text{ANDab} = \text{AND}[\Phi\text{AND}_{ab}]$$

based on the agent

$$\text{AND} = \sum_{x,y\in\{0,1\}} (in1_x in2_y \overline{out}_z : \text{nil})$$

with the Boolean evaluation $z = x$ AND y and the morphism ΦAND_{ab} defined by the relabelling pairs $in1 \mapsto upa$, $in2 \mapsto upb$ and $out \mapsto ANDabout$;

$$\text{XORab} = \text{OR}[\Phi\text{XOR}_{ab}]$$

based on the agent

$$\text{XOR} = \sum_{x,y\in\{0,1\}} (in1_x in2_y \overline{out}_z : \text{nil})$$

with z = x XOR y and the morphism ΦXOR$_{ab}$ defined by the relabelling pairs $in1 \mapsto downa$, $in2 \mapsto downb$ and $out \mapsto XORabout$.

(2) The enabling agents are: EADD and ECARRY. Their definitions are:

$$\text{EADD} = \text{AND}[\Phi\text{AND}_{EADD}]$$

with ΦAND$_{EADD}$ defined by $in1 \mapsto XORabout$, $in2 \mapsto downD3$ and $out \mapsto ADDout$;

$$\text{ECARRY} = \text{AND}[\Phi\text{AND}_{ECARRY}]$$

with ΦAND$_{ECARRY}$ defined by $in1 \mapsto ANDabout$, $in2 \mapsto upD3$ and $out \mapsto CARRYout$.

(3) The arithmetic agents are:

$$\text{ArithmADD} = (\text{XORab} \times \text{EADD})\backslash\{XORabout\}$$

and

$$\text{ArithmCARRY} = (\text{ANDab} \times \text{ECARRY})\backslash\{ANDabout\}$$

We also need some agents for modelling the distribution of the electric signal on the circuit wires. These agents depend on the number of forked lines in a circuit node. Hence, for the fork of the signal into two lines the agent is

$$\text{NODE2} = \sum_{x\in\{0,1\}} (in_x\overline{up}_x\overline{down}_x : \text{nil}).$$

We need three appropriate rellabeled agents based on the agent NODE2, as follows:

$$\text{NODE2\_a} = \text{NODE2}[\Phi 2_a] \tag{1}$$

with $\Phi 2_a$ defined by $in \mapsto a$, $up \mapsto upa$ and $down \mapsto downa$;

$$\text{NODE2\_b} = \text{NODE2}[\Phi 2_b] \tag{2}$$

with $\Phi 2_b$ defined by $in \mapsto b$, $up \mapsto upb$ and $down \mapsto downb$;

$$\text{NODE2\_D3} = \text{NODE2}[\Phi 2_{D3}] \tag{3}$$

with $\Phi 2_{D3}$ defined by $in \mapsto D3$, $up \mapsto upD3$ and $down \mapsto downD3$.

Using the above agents, we are now ready to define two agents for modelling the AU behaviour: a low-level specification agent EArithm based on the behaviour of the arithmetic unit and a high-level specification agent SpecEArithm based on the definition structure of the arithmetic unit circuit.

Hence, the implementation of the arithmetic part of the ALU based on the behaviour of the circuit is given by the agent:

$$\begin{aligned}\text{EArithm} &= \\ &= (\text{ArithmADD} \times \text{ArithmCARRY} \times \text{NODE2\_a} \times \text{NODE2\_b} \times \text{NODE2\_D3})\backslash \\ &\backslash\text{Comm\_EArithm}\end{aligned} \tag{4}$$

where the set of communicating actions is

$$\text{Comm\_EArithm} = \{upa, downa, upb, downb, upD3, downD3\}.$$

The specification of the arithmetic part of the ALU based on the definition of the circuit represented in Figure 2 is given by the agent:

$$\text{SpecEArithm} = \sum_{x,y,m\in\{0,1\}} (a_x b_y D3_m \overline{ADDout}_s \overline{CARRYout}_t : \text{nil}) \tag{5}$$

where the Boolean evaluations are:

$$s = \begin{cases} 0, \text{ if } m = 0 \\ x \text{ XOR } y, \text{ if } m = 1 \end{cases} \quad \text{and } t = \begin{cases} 0, \text{ if } m = 0 \\ x \text{ AND } y, \text{ if } m = 1 \end{cases}$$

Note that the binary number $\overline{ts}_{(2)}$ is exactly the binary sum $x +_2 y$.

3.2 The formal proof of the agents bisimilarity

In this section we will prove that the two previous specification agents for the AU are bisimulation equivalent, the appropriate equivalence in the theory of concurrent communicating processes. This result is very important for the target of this paper since it means that the behaviour of the AU modeled by the implementation agent EArithm is correct with respect to the AU definition modeled by the specification agent SpecEArithm.

Proposition 1 *The previous agents SpecEArithm and EArithm are bisimulation equivalent.*

Proof: The bisimulation relation '$\sim$' is a congruence over the class $\mathcal{P}$ of agents [3].

We consider the low-level specification for the arithmetic part of the ALU given by the previous agent EArithm (4):

$$\begin{aligned} &\text{EArithm} = \\ &\quad = (\text{ArithmADD} \times \text{ArithmCARRY} \times \text{NODE2\_a} \times \text{NODE2\_b} \times \text{NODE2\_D3}) \backslash \\ &\quad \backslash \text{Comm\_EArithm} \end{aligned}$$

where the set of communicating actions is

$$\text{Comm\_EArithm} = \{upa, downa, upb, downb, upD3, downD3\}.$$

We evaluate this agent in few steps corresponding to the inside agents.

$$\begin{aligned} &\text{ArithmADD} = (\text{XORab} \times \text{EADD}) \backslash \{XORabout\} = \\ = \ &(\sum_{x,y\in\{0,1\}} (downa_x downb_y \overline{XORabout}_z : \text{nil}) \times \sum_{z,m\in\{0,1\}} (XORabout_z downD3_m \overline{ADDout}_s : \text{nil})) \backslash \\ \backslash \ &\{XORabout\} \end{aligned}$$

where z = x XOR y and s = z AND m.

After we apply the product (SCCS synchronous composition) and the restriction on the internal communicating action $XORabout$, the agent expression is:

$$\text{ArithmADD} = \sum_{x,y,m\in\{0,1\}} (downa_x downb_y downD3_m \overline{ADDout}_s : \text{nil})$$

Following the logic expressions s = z AND m and z = x XOR y we have s = z AND m = (x XOR y) AND m, meaning $s = \begin{cases} 0, \text{ if } m = 0 \\ x \text{ XOR } y, \text{ if } m = 1 \end{cases}$.

Analogously, the expression for the ArithmCARRY agent is:

$$\begin{aligned}&\text{ArithmCARRY} = (\text{ANDab} \times \text{ECARRY})\backslash\{ANDabout\} = \\ = \;& (\sum_{x,y\in\{0,1\}} (upa_x upb_y \overline{ANDabout}_z : \text{nil}) \times \sum_{z,m\in\{0,1\}} (ANDabout_z upD3_m \overline{CARRYout}_t : \text{nil}))\backslash \\ \backslash \;& \{ANDabout\}\end{aligned}$$

where z = x AND y and t = z AND m.

After we apply the product (SCCS synchronous composition) and the restriction on the internal communicating action $ANDabout$, the agent expression is:

$$\text{ArithmCARRY} = \sum_{x,y,m\in\{0,1\}} (upa_x upb_y upD3_m \overline{CARRYout}_t : \text{nil})$$

Following the logic expressions t = z AND m and z = x AND y we have t = z AND m = (x AND y) AND m, meaning $t = \begin{cases} 0, \text{ if } m = 0 \\ x \text{ AND } y, \text{ if } m = 1 \end{cases}$.

Following the previous definitions (1), (2) and (3) of the corresponding agents NODE2_a, NODE2_b and NODE2_D3, we have

$$\text{NODE2\_a} = \sum_{x\in\{0,1\}} (a_x \overline{upa}_x \overline{downa}_x : \text{nil})$$

$$\text{NODE2\_b} = \sum_{y\in\{0,1\}} (b_y \overline{upb}_y \overline{downb}_y : \text{nil})$$

$$\text{NODE2\_D3} = \sum_{m\in\{0,1\}} (D3_m \overline{upD3}_m \overline{downD3}_m : \text{nil})$$

Considering all the previous agents expressions and the set of the internal, communicating actions Comm_EArithm = $\{upa, downa, upb, downb, upD3, downD3\}$, we conclude that:

$$\begin{aligned}&\text{EArithm} = \\ &= (\text{ArithmADD} \times \text{ArithmCARRY} \times \text{NODE2\_a} \times \text{NODE2\_b} \times \text{NODE2\_D3})\backslash\text{Comm\_EArithm} = \\ &= (\sum_{x,y,m\in\{0,1\}} (downa_x downb_y downD3_m \overline{ADDout}_s : \text{nil}) \times \\ &\times \sum_{x,y,m\in\{0,1\}} (upa_x upb_y upD3_m \overline{CARRYout}_t : \text{nil}) \times \\ &\times \sum_{x\in\{0,1\}} (a_x \overline{upa}_x \overline{downa}_x : \text{nil}) \times \sum_{y\in\{0,1\}} (b_y \overline{upb}_y \overline{downb}_y : \text{nil}) \times \\ &\times \sum_{m\in\{0,1\}} (D3_m \overline{upD3}_m \overline{downD3}_m : \text{nil}))\backslash\{upa, downa, upb, downb, upD3, downD3\} = \\ &= \sum_{x,y,m\in\{0,1\}} (a_x b_y D3_m \overline{ADDout}_s \overline{CARRYout}_t : \text{nil})\end{aligned}$$

with the logic evaluations: $s = \begin{cases} 0, \text{ if } m = 0 \\ x \text{ XOR } y, \text{ if } m = 1 \end{cases}$ and $t = \begin{cases} 0, \text{ if } m = 0 \\ x \text{ AND } y, \text{ if } m = 1 \end{cases}$.

If you compare this final expression for the low-level specification agent EArithm with the definition of the high-level specification agent SpecEArithm given in (5) it is obvious that these two agents represent the same circuit behaviour, meaning they are bisimulation equivalent, as required. □

For $m = 1$, the final expressions for s and t are validating the name of the part we are discussing about, namely arithmetic unit. This is because the final logic expressions for s and t are modelling the two specific outputs of an half adder, respectively: s represents the sum of the two input bits and t represents the carry bit, as follows:

x	y	$x +_2 y$	t	s
0	0	$\overline{00}_{(2)}$	0	0
0	1	$\overline{01}_{(2)}$	0	1
1	0	$\overline{01}_{(2)}$	0	1
1	1	$\overline{10}_{(2)}$	1	0

This result of bisimilarity shows that the behaviour of the AU follows the definition of the corresponding arithmetic circuit and, on the other hand, it is a guarantee of using these agents in other complex models.

3.3 The automatic verification of the agents bisimilarity

For the implementation-specification pair of agents EArithm-SpecEArithm, we have used the CWB-NC platform [22] for verifying the appropriate agents bisimilarity. The corresponding CWB-NC answer for this test is TRUE and the specific result is pointed in Figure 3:

```
cwb-nc> es LoadArithmUAL.cws
Executing CWB-NC script file LoadArithmUAL.cws, directing output to std_out.
September 20, 2013   17:17
Execution time (user,system,gc,real):(0.001,0.000,0.000,0.001)
cwb-nc> Execution time (user,system,gc,real):(0.036,0.000,0.000,0.036)
cwb-nc> Execution time (user,system,gc,real):(0.003,0.000,0.000,0.003)
cwb-nc> ( The output has been put into std_out )
cwb-nc> eq EArithm SpecEArithm
Building automaton...
States: 4
Transitions: 16
Done building automaton.
Transforming automaton...
Done transforming automaton.
TRUE
Execution time (user,system,gc,real):(29.276,0.000,0.027,29.276)
cwb-nc> _
```

Figure 3: Automatic verification with CWB-NC

This CWB-NC answer authenticates the theoretical result proved above using the SCCS operational semantics. It is an important benefit of our work to have the implementation-specification pair of bisimilar agents, but, unfortunately, the execution time achieved here is not convenient. It is one of our future work targets to improve this time.

Using the CWB-NC is still a reliable approach, following the research interest revealed by the consistent publications like [7] or [5] relating to the CWB-NC, even in connection with CCS, SCCS and other modelling and verification tools.

4 Conclusions

It is our general target to obtain an algebraic-based formal framework for modelling and verification the computer system behaviour. This is following a multi-agent approach, each agent individually representing a specific computer hardware component. Out of our overall interests, both the specification

and implementation modelling levels, and verification of the CPU arithmetic unit behaviour have been considered in this paper.

For the given AU structure, we have defined appropriate SCCS agents based on the definition and on the behaviour of the AU and we have proved the bisimulation equivalence between the defined agents, authenticating the correctness of the behaviour with respect to the AU definition. Based on these results, it follows that we may use these agents in the next steps for modelling other hardware components having the AU structure as internal part, for example the more complex processing units.

We also consider as future work directions the possibility of moving on from this combination based on SCCS - CWBNC to another modern opportunities based on functional programming. At this moment, an interesting and modern solution could follow the Alvis project results for modelling and/or encoding the embedded, especially rule-based systems. Following [18], [21] and [17], Alvis is developing in Krakow, Poland starting with 2009. It is based on CCS and XCCS process algebras, it is defined for the design of concurrent especially real-time systems and it also provides a possibility of a formal model verification. One of the main Alvis advantages consists in combining a flexible graphical modelling approach for interconnections among agents with a high-level programming language used for the description of agents' behaviour. Even if Alvis is based on CCS and XCCS, its internal high-level programming language is based on the Haskell syntax instead of algebraic equations. In [21], the functional programming language Haskell [8] is appreciated as the most natural way of encoding a rule-based system into an Alvis model. Moreover, Haskell features like lazy evaluation, pattern matching or high level functions make it a very attractive proposition for the Alvis interests.

From our point of view, the Alvis project means an opportunity for future work consisting of replacing the equation-based algebraic modelling approach by a Haskell-based functional approach. From the educational point of view, the Haskell opportunities for our students are already a topic of our interests [20]. From the scientific point of view, passing to the functional approach is expecting to substantially improve the CWB-NC execution time obtained here for automatic verification of the agents' bisimilarity equivalences.

If Alvis is adding the Haskell facilities over the (X)CCS process algebra characteristics, we also have the alternative of the CHP library - as a set of Haskell packages for implementing the concurrency ideas from Hoare's CSP [2]. The beginning of Communicating Haskell Processes, namely CHP research framework is in [10]. Both Alvis and CHP have gathered the research and practical results in corresponding PhD thesis [12], [16].

References

[1] R. Milner, Calculi for synchrony and asynchrony. *Theoretical Computer Science* 25, 1983, pp. 267–310.

[2] C.A.R. Hoare, *Communicating sequential processes*, Prentice-Hall, 1985.

[3] R. Milner, *Communication and concurrency*, Prentice Hall, 1989.

[4] M.M. Mano, *Computer System Architecture*, Prentice Hall Intl., 1993.

[5] D. Zhang, R. Cleaveland, E.W. Stark, The Integrated CWB-NC/PIOATool for Functional Verification and Performance Analysis of Concurrent Systems. *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science Volume 2619, 2003, pp 431-436.

[6] A.S. Tanenbaum, *Structured Computer Organization*, Pearson Prentice Hall, 2006.

[7] L. Aceto, A. Ingolfsdottir, K. Larsen, J. Srba, *Reactive Systems: Modelling, Specification and Verification*, Cambridge University Press, 2007.

[8] G. Hutton, *Programming in Haskell*, Cambridge University Press, 2007.

[9] M.M. Mano, M. Celetti, *Digital Design*, Pearson Prentice Hall, 2007.

[10] N.C.C. Brown, Communicating Haskell Processes: Composable explicit concurrency using monads. *Communicating Process Architectures* 2008, pp.67-83.

[11] A. Vasilescu, Counter register. Algebraic model and applications. *WSEAS Transactions on Computers*, Issue 10, Vol. 7, pp. 1618-1627, Oct. 2008.

[12] P. Matyasik, *Design and analysis of embedded systems with XCCS process algebra*, PhD Thesis, AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, Krakow, Poland, 2009.

[13] O. Rădoi, A. Ivan, A. Vasilescu, Logic Circuit Designer. *ACTA UNIVERSITATIS APULENSIS Mathematics - Informatics*, Special Issue as Proc of ICTAMI 2009, Alba-Iulia, Romania, September 3-6, 2009, pp.979-990.

[14] A. Vasilescu, Algebraic model for the synchronous D-flip-flop behaviour. *Proc. of Intl. Conf. Modelling and Development of Intelligent Systems MDIS 2009*, Sibiu, Romania, October 22-25, pp.308-315.

[15] A. Vasilescu, Algebraic model for the behaviour of a D-flip-flops-based memory component. *book Mathematical Methods, Computational Techniques, Intelligent Systems, Proc. of the 12^{th} WSEAS Intl Conf MAMECTIS'10*, El Kantaoui, Sousse, Tunisia, May 3-6 2010, pp.42-47.

[16] N.C.C. Brown, *Communicating Haskell Processes*, PhD Thesis, The University of Kent, Computer Science subject, UK, May 2011.

[17] M. Szpyrka et al., Introduction to modelling embedded systems with Alvis. *Automatyka*, Tom 15, part 2, 2011, pp.435-442.

[18] M. Szpyrka, P. Matyasik, R. Mrowka, Alvis - modelling language for concurrent systems. *Intelligent Decision Systems in Large-Scale Distributed Environments*, ser. Studies in Computational Intelligence, Springer-Verlag, Volume 362, 2011, ch. 15, pp 315-341.

[19] A. Vasilescu, A. Băicoianu, Algebraic model for the CPU logic unit behaviour. *book Recent Researches in Computer Science, Proc of 15th WSEAS Intl Conf on Computers (Part of the 15th WSEAS CSCC Multiconference)*, Corfu Island, Greece, July 15-17, 2011, pp. 521-526.

[20] A. Vasilescu, F.-R. Drobotă, Reasons for studying Haskell in University. *Proc. of The 7^{th} Intl Conf on Virtual Learning, Virtual Learning - Virtual Reality*, Braşov, Romania, November 2-3 2012, pp. 394-400.

[21] M. Szpyrka, T. Szmuc, Design and Verification of Rule-Based Systems for Alvis Models. *Intelligent Systems Reference Library*, Volume 43, 2013, pp 539-558.

[22] CWB *** The CWB-NC homepage on http://www.cs.sunysb.edu/~cwb.

Anca Vasilescu
Transilvania University of Braşov
Department of Mathematics and Computer Science
Iuliu Maniu Street 50, 500091 Braşov
ROMANIA
E-mail: *vasilex@unitbv.ro*