PETRI NET BASED APPROACH TO TEST BENCH CONSTRUCTING FOR DATAPATH

Andrei Karatkevich
Institute of Computer Science and Electronics, University of Zielona Gora, Podgorna 50, Zielona Gora, Poland

Keywords: System design, Data path, Test bench, Simulation, Verification, Petri nets.

Abstract: Testing a data path in a digital system such as a microcontroller requires checking every possible way of sending data between the functional units. This paper considers a task of generating a test bench for a given data path, which covers every way of data sending with minimized number of simulations of microinstructions. We present a method in which a data path is modeled by a Petri net. The task of optimal test bench generation is formulated as a task of covering all transitions by a sequence with minimal length. It can be solved by finding certain T-invariant of the net and a firing sequence corresponding to it. The proposed method is illustrated by two case studies of testing data paths of simple processors.

1 INTRODUCTION

The von Neumann computer architecture defines the *control unit* as a distinct part of a design. Nowadays, a digital system is often considered as a composition of control unit (CU) and *data path* (DP) (Fig. 1). The separate synthesis methods for data path and control unit are developed (Baranov, 2008; Barkalow and Wegrzyn, 2006; Wisniewski, 2009).

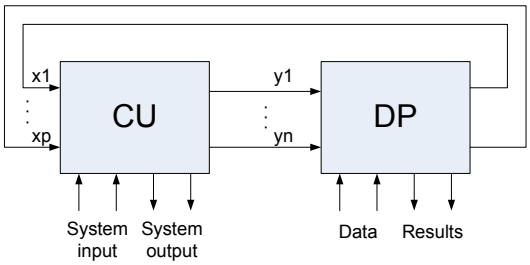

Figure 1: Digital system as a composition of CU and DP.

To verify a formal specification of a design, a test bench should be constructed. Among the necessary components of a test bench there are input vectors (stimuli) and the expected output vectors. The design is simulated with the stimuli at its input, and the output values being the result of simulation are compared to the expected values. If there is a difference between them, then the mistakes in the design are detected (Baranov, 2008).

As far as data path and control unit are designed separately, it is reasonable to test them also separately, constructing special test bench for each of them. Data path consists of such units as memory blocks, registers, arithmetic logic units, counters, multiplexors and so on. For testing a data path it is necessary to check every direct connection between its units at least once. But it is possible to write some input data directly only to the inputs of a DP. We also suppose that it is possible to read data only from the ”output” units. If a data path is represented by an oriented graph, where nodes correspond to the data path units, and arcs correspond to direct data sending between the units (a connection graph), then the task of data path testing can be presented as a task of covering all arcs of the graph by the paths from its start nodes to end nodes. Note that ”sending” data from a data path unit to itself (such as operation $i := i + 1$, where i is implemented as a counter) also should be checked; such situation can be represented in a connection graph as a self-loop. Also, *every* microinstruction should be checked, even if two or more microinstructions send data between the same pair of units (it means that a connection graph may have multi-edges).

An algorithm for automated generation of a testing sequence of microinstructions for data path by means of connection graph covering is presented in (Karatkevich and Baranov, 2010); the task is reduced to one of the variants of the route inspection problem, which can be effectively solved. But there is a problem with the graph-based approach: if a microinstruc-

Karatkevich A..
PETRI NET BASED APPROACH TO TEST BENCH CONSTRUCTING FOR DATAPATH.
DOI: 10.5220/0003363105060511
In *Proceedings of the 1st International Conference on Pervasive and Embedded Computing and Communication Systems* (PECCS-2011), pages 506-511
ISBN: 978-989-8425-48-5

tion may contain several microoperations (which is usual for control units (Baranov, 2008)), then one microinstruction may send data between more than two data path units. A test sequence consists of microinstructions, not of microoperations. Then we cannot directly obtain a test sequence from a path in a data path connection graph. So it is reasonable to model a data path not just by a graph, but by a Petri net (where a transition represents a microinstruction), and to solve the task by generating a minimized firing sequence covering all its transitions. The presented paper describes an algorithm of test sequence generation for data path, using Petri net as a data path model.

2 PETRI NETS

Petri nets are used as one of the basic models of concurrent discrete systems (Murata, 1989; Peterson, 1981). A Petri net can be considered as a bipartite oriented graph with two kinds of nodes: *places* and *transitions* (Fig. 2). Places of a net may contain *tokens*, and a configuration of the tokens is called a *marking* (state) of the net. A marking is denoted as M, with some indexes if needed. A marking is *safe* if no place contains more than one token. A marking can be changed by *firing* (execution) of the enabled transitions. An *enabled transition* is such transition that every its *input place* (a place from which an arc leads to the transition) contains a token. *Transition firing* removes a token from each input place and adds a token to each *output place* (a place to which an arc leads from the transition) of it. Note that we will need, among others, the transitions without input or output places. A transition without input places is always enabled; a transition without output places, when firing, does not add any token anywhere.

For a Petri net N with n transitions and m places, the *incidence matrix* $A = [a_{ij}]$ is an $m \times n$ matrix such that

$$a_{ij} = a_{ij}^{+} - a_{ij}^{-} \tag{1}$$

where $a_{ij}^{+} = 1$ if and only if place p_j is an output place of transition t_i, otherwise $a_{ij}^{+} = 0$; and $a_{ij}^{-} = 1$ if and only if place p_j is an input place of transition t_i, otherwise $a_{ij}^{-} = 0$. Every a_{ij} represents the number of tokens changed in place p_j by firing of transition t_i.

An integer n-vector x is a *T-invariant*, if $A^T x = 0$.

A *firing count vector* of a firing sequence (a sequence of transition firings) σ is an n-vector of nonnegative integers $\overline{\sigma}$ such that the ith entry of $\overline{\sigma}$ denotes the number of times transition t_i fires in sequence σ.

Theorem 1 (Murata, 1989): An n-vector $x \geq 0$ is a T-invariant if and only if there exists a marking M and firing sequence σ from M back to M with its firing count vector $\overline{\sigma} = x$.

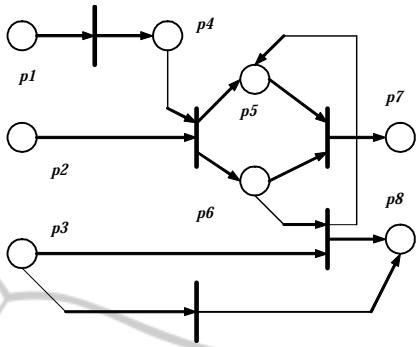

Figure 2: An example of Petri net.

3 IDEA OF THE METHOD

Let us construct for given data path a modeling Petri net in the following way. A place corresponds to every data path unit; a transition t_i corresponds to every microinstruction Y_i, and a place p_j is an input place for t_i if and only if there is a microoperation in Y_i such that it sends data *from* the unit corresponding to p_j; place p_k is an output place for t_i if and only if there is a microoperation in Y_i such that it sends data *to* the unit corresponding to p_k.

Now, let us *remove* every place corresponding to an input or output data path unit. It will cause that our net will have transitions without input places and transitions without output places. Indeed, at any moment we can write data to the input units and read from the output units. Hence any firing sequence in the obtained net will correspond to a possible way of data sending in the data path.

Suppose that initially the Petri net has no tokens. The tokens can be introduced by firing the transitions without input places. Any nonempty firing sequence leading from empty marking back to empty marking corresponds to data sending from input to output units of the data path. As follows from Theorem 1, a T-invariant corresponds to every such firing sequence. By finding a T-invariant $x > 0$ we can find a firing sequence covering all transitions, which will correspond to a sequence of microinstructions allowing to check every connection in the data path.

To find the T-invariants, it is necessary to solve the system of linear equations $A^T x = 0$. It can be solved by Gauss' method (Hefferon, 2008). Then, finding among the set of solutions one which consists of positive integers and has minimal sum of values of

the variables is a linear programming task which can be solved by the simplex method or another appropriate method (Vanderbei, 2008). When the minimal T-invariant covering all transitions is obtained, a corresponding firing sequence can be calculated from it. The test sequence of the microinstructions can be obtained from the firing sequence.

4 THE PROPOSED ALGORITHM

Below the algorithm of test sequence generation is presented.

1. Create a Petri net N, where places correspond to the *internal* data path units, and transitions correspond to possible data transfers between the units (*including* input and output ones); one transition corresponds to one microinstruction.
2. Solve the system of linear equations $A^T x = 0$ by Gauss' method. Note that the rows of A corresponding to the self-loops and to direct data sending from input to output units consist of zeros. As far as *any* values of the corresponding entries of a T-invariant are possible, use value 1 of them. If the system has no solutions, go to step 7.
3. Find the solution consisting of positive integers with minimal sum. It can be formulated as a linear programming problem with the function to be maximized $f(x) = -x_1 - x_2 - ... - x_n$ (note that not every x_i is a free variable) and problem constrains $x_1 > 0, x_2 > 0, ..., x_n > 0$. If the problem has no trivial solution, it can be solved by the simplex method.
4. Construct for the selected solution (T-invariant) x a firing sequence σ with its firing count vector equal to x such that every reached marking is safe. To do that, simulate firing of the transitions according to their number in x, starting from the empty marking. If an unsafe marking is obtained or a marking in which not all transitions have fired appropriate number of times but none of them is enabled, then the constructing should backtrack, returning to the most resent transition where another possibility of firing existed, and try another possibility. If the algorithm fails to construct σ for x, go to step 8.
5. Construct for the firing sequence σ corresponding sequence of microinstructions for the test bench in the following way. Scan the firing sequence. For every transition t_i add the corresponding microinstruction Yi, if and only if in the sequence of microinstructions there is no previous entry of Yi such that no microinstruction between those entries writes data to at least one of data path units to which Yi writes or from which it reads.
6. The test sequence is constructed successfully. The end.
7. Constructing of the test sequence failed (this possibility is discussed below). The end.
8. Constructing of the test sequence failed, an inconsistency in the design is detected (this possibility is discussed below). The end.

5 EXAMPLES

5.1 General Information

For the examples we use the designs created by means of the experimental EDA tool Abelite which implements high-level synthesis and a very fast optimizing synthesis of FSM and combinational circuits (Baranov, 2008; Baranov, 2009). The Abelite design methodology follows the common model in which any digital system is regarded as a composition of control unit and data path. One of the main concepts used in this methodology is the construction of the so-called *naked data path*, which doesn't contain any cloud circuits, only standard regular units. The data path design is described in detail in (Baranov, 2008).

5.2 A Simple Processor

As the first example we use a design of a very simple processor implementing two operations - the bubble sort and the search of the maximal element. It is an improved version of the design described in (Baranov, 2009). List of its microinstructions is presented in Table 1. The connection graph of the data path is presented in Figure 3.

The Petri net modeling the data path is shown in Figure 4. "Non-existing" places (corresponding to input and output data path units) and the incident arcs are dashed. Numbers of transitions correspond to numbers of microinstructions.

The T-invariants for the net shown in Figure 4 can be obtained by solving the following system, where numbers of variables correspond to numbers of transitions in Figure 4:

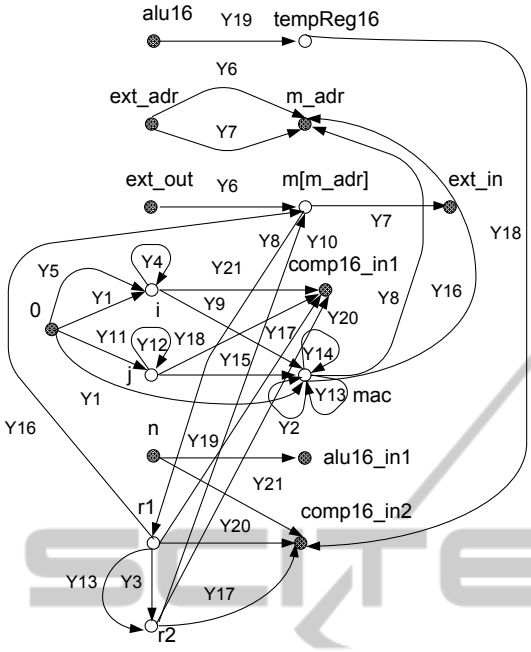

Figure 3: Connection graph for the first example.

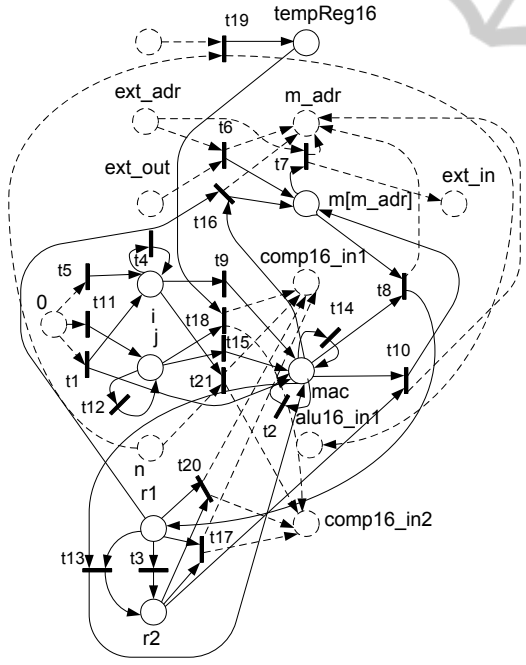

Figure 4: Petri net modeling the data path.

$$\begin{cases} -x_{18}+x_{19} &= 0 \\ x_6-x_7-x_8+x_{10}+x_{16} &= 0 \\ x_1+x_5-x_9-x_{21} &= 0 \\ x_{11}-x_{15}-x_{18} &= 0 \\ x_1-x_8+x_9-x_{10}+x_{15}-x_{16} &= 0 \\ -x_3+x_8-x_{13}-x_{16}-x_{17}-x_{20} &= 0 \\ x_3-x_{10}+x_{13}-x_{17}-x_{20} &= 0 \end{cases} \quad (2)$$

Table 1: Functional microinstructions and microoperations for the first example.

Microinstructions	Microoperations	
Y1	y16 y25	i:=0 mac:=0
Y2	y28	mac:=mac+1
Y3	y31	r2:=r1
Y4	y17	i:=i+1
Y5	y16	i:=0
Y6	y20 y23	m[m_adr]:=ext_out m_adr:=ext_adr
Y7	y15 y23	ext_in:=m[m_adr] m_adr:=ext_adr
Y8	y24 y30	m_adr:=mac r1:=m[m_adr]
Y9	y26	mac=i
Y10	y20 y23	m[m_adr]:=r2 m_adr:=mac
Y11	y18	j:=0
Y12	y19	j:=j+1
Y13	y28 y31	mac:=mac+1 r2:=r1
Y14	y29	mac:=mac-1
Y15	y27	mac:=j
Y16	y20 y23	m[m_adr]:=r1 m_adr:=mac
Y17	y8 y12	comp16_in1:=r1 comp16_in2:=r2
Y18	y7 y13	comp16_in1:=j comp16_in2 :=tempReg16
Y19	y2 y32	alu16_in1:=n tempReg16:=alu16
Y20	y9 y11	comp16_in1:=r2 comp16_in2:=r1
Y21	y6 y10	comp16_in1:=i comp16_in2:=n

The solution set can be described by the following equations:

$$\begin{cases} x_1 &= -x_9+2x_{10}-x_{15}+2x_{16}+2x_{17} \\ & \quad +2x_{20} \\ x_3 &= x_{10}-x_{13}+x_{17}+x_{20} \\ x_5 &= 2x_9-2x_{10}+x_{15}-2x_{16}-2x_{17} \\ & \quad -2x_{20}+x_{21} \\ x_6 &= x_7+2x_{17}+2x_{20} \\ x_8 &= x_{10}+x_{16}+2x_{17}+2x_{20} \\ x_{11} &= x_{15}+x_{19} \\ x_{18} &= x_{19} \end{cases} \quad (3)$$

The function which should be minimized (keeping all free and bound variables positive integers) is $\sum_{i=1}^{21} x_i = x_2+x_4+2x_7+2x_9+3x_{10}+x_{12}+x_{14}+2x_{15}+2x_{16}+6x_{17}+3x_{19}+6x_{20}+2x_{21}$. One of the minimal solutions (the T-invariant we are looking for) is $(3,1,2,1,2,5,1,6,4,1,2,1,1,1,1,1,1,1,1,1,1)$.

The following firing sequence can be obtained from this T-invariant:

$t_1t_2t_4t_6t_7t_6t_8t_3t_6t_9t_5t_8t_6t_9t_5t_{11}t_{12}t_{14}t_{17}t_8t_3t_6t_{15}t_8t_9t_{10}t_1t_{11}$
$t_{13}t_8t_9t_{16}t_1t_8t_{19}t_{18}t_{20}t_{21}$.

The resulting test sequence is:
$Y_1Y_2Y_4Y_6Y_7Y_8Y_3Y_9Y_5Y_8Y_{11}Y_{12}Y_{14}Y_{17}Y_8Y_{15}Y_{10}Y_{13}Y_{16}Y_8$
$Y_{19}Y_{18}Y_{20}Y_{21}$.

5.3 A More Complex Processor

As the second example we use more complex and realistic design - a 16-bit processor described in (Baranov, 2008). To save place we limit our consideration only to the 16-bit microoperations which transfer information from the output of one unit to the input of another unit (excluding the operations which are executed in one operational unit). List of such microoperations and corresponding microinstructions is presented in Table 2. The connection graph of the data path is presented in Figure 5.

Table 2: Microinstructions and 16-bit microoperations for the second example

Microinstructions	Microoperations	
Y1	y2	BoR[AdrW]:=RALU
Y2	y5 y6 y8	ALU1:=BoR[AdrR1] ALU2:=BoR[AdrR2] RALU:=ALU
Y3	y6 y8	ALU2:=BoR[AdrR2] RALU:=ALU
Y4	y12 y13	Adr1:=IR2 BoR[AdrW] :=M1[Adr1]
Y5	y14	BoR[AdrW]:=IR2
Y6	y15	BoR[AdrW] :=BoR[AdR2]
Y7	y12 y16	Adr1:=IR2 M1[Adr1] :=BoR[AdrR1]
Y8	y18	BoR[AdrW] :=BoR[AdrR1]
Y9	y5 y8	ALU1:=BoR[AdrR1] RALU:=ALU
Y10	y19	PC:=BoR[AdrR2]
Y11	y20	PC:=IR2
Y13	y22	BoR[AdrW]:=InpR
Y14	y24	OutR:=BoR[AdrR1]
Y17	y28	PC:=x"FFFE"
Y18	y30 y31	Adr1:=x"FFFF" M1[Adr1]:=PC
Y19	y32 y33	Adr0:=Ext_Adr M0[Adr0]:=Ext_Out
Y20	y34 y35	Adr1:=Ext_Adr M1[Adr1]:=Ext_Out
Y21	y34 y36	Adr1:=Ext_Adr Ext_in:=M1[Adr1]
Y22	y32 y37	Adr0:=Ext_Adr Ext_in:=M0[Adr0]
Y24	y39 y40	Adr0:=PC IR1:=M0[Adr0]
Y25	y39 y41	Adr0:=PC IR2:=M0[Adr0]

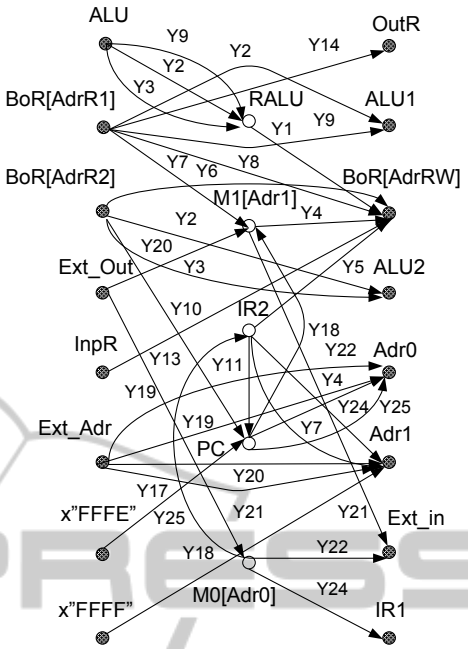

Figure 5: Connection graph for the second example.

The modeling Petri net is not shown, because in this case it is too complex to be readable. There are 5 internal blocks in this datapath: RALU, M0[Adr0], M1[Adr1], IR2 and PC. So the system of equations which should be solved to obtain the T-invariants consists of 5 equations (numbers of the variables correspond to the numbers of microinstructions):

$$\begin{cases} -x_1+x_2+x_3+x_9 &= 0 \\ x_{19}-x_{22}-x_{24}-x_{25} &= 0 \\ -x_4+x_7+x_{18}+x_{20}-x_{21} &= 0 \\ -x_4-x_5-x_7-x_{11}+x_{25} &= 0 \\ x_{10}+x_{11}+x_{17}-x_{18}-x_{24}-x_{25} &= 0 \end{cases} \quad (4)$$

Its minimal positive integer solution is: $(x_1,x_2,x_3,x_4,x_5,x_7,x_9,x_{10},x_{11},x_{17},x_{18},x_{19},x_{20},x_{21},$ $x_{22},x_{24},x_{25}) = (3,1,1,1,1,1,1,4,1,1,1,6,1,2,1,4)$.

After constructing a firing sequence corresponding to this T-invariant (item 4 of the algorithm) and test sequence corresponding to the firing sequence (item 5) we obtain: $Y_2Y_1Y_3Y_1Y_9Y_1Y_{10}Y_{18}Y_{19}Y_{21}Y_{20}Y_{21}Y_{22}Y_{24}Y_{25}Y_5Y_7Y_{17}Y_{25}$ $Y_{11}Y_{25}Y_4Y_6Y_8Y_{13}Y_{14}$. This sequence covers every connection and is more representative than the ”hand-made” sequence presented in (Baranov, 2008).

6 WHEN THE METHOD FAILS

As it can be seen from the description of the algorithm, the situations are possible in which it fails to generate a test sequence. It happens when there is no T-invariant $x > 0$ for the modeling Petri net or when for the obtained T-invariant the appropriate firing sequence does not exist. The second variant means two possibilities: there is no firing sequence for the obtained T-invariant that starts in empty marking or such sequence exists, but leads through an unsafe marking. Three mentioned situations are illustrated in Figure 6.

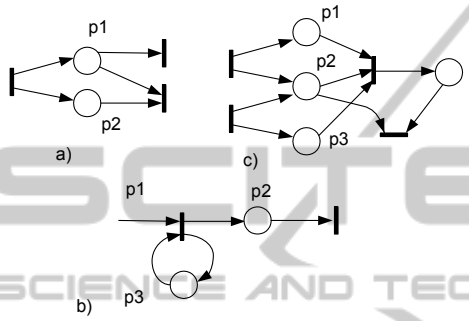

Figure 6: Examples of nets for which the method fails to generate the sequence.

The first of them (Figure 6a; there is no T-invariant without zero entries) corresponds to a case which is not impossible (however rather untypical) in a correct design. In this case any sequence of microinstructions which transfers data from input to output units of the data path either reads more than once from the same unit without writing to it between those readings (p_1), or writes to an internal unit without reading from it (p_2). Then the method described in (Karatkevich and Baranov, 2010) can be applied.

Two other situations signalize that something is wrong in the data path structure or in the structure of microinstructions. If a firing sequence leading from an empty marking back to itself and covering all transitions does not exist (Figure 6b), then there is an internal unit (p_3) from which data are read *before* writing in it. If such firing sequence exists but has to go through an unsafe marking (Figure 6c), then there is a unit (p_2) to which data are written more than once without reading from it between those writings, which means that some data are lost.

7 CONCLUSIONS

The proposed method provides possibility of automated generation of sequences of microinstructions for testing data path of a digital design constructed as a composition of a data path and a control unit. Such sequence is a necessary part of a test bench. The method we propose takes into account structure of microinstructions, which may consist of several microoperations. Using Petri net as a model of data path was found to be suitable for such cases.

However, the method described here at first generates a long sequence with multiple repetitions of some transitions and then constructs a sequence of microinstructions which may be remarkable shorter. Further research should concentrate on checking whether it is possible to build a minimized sequence directly, avoiding constructing a firing sequence corresponding to the T-invariant of the modeling Petri net (as in this method) or a postman tour in the modeling graph (as in (Karatkevich and Baranov, 2010)).

ACKNOWLEDGEMENTS

I would like to thank Samary Baranov for inspiration, fruitful discussions and consultations. The projects of the processors used for the examples are developed by him.

REFERENCES

Baranov, S. (2008). *Logic and System Design of Digital Systems*. TGU, Tallinn.

Baranov, S. (2009). Asms in high level synthesis of eda tool abelite. In *Preprints of the 4th IFAC Workshop on Discrete-Event System Design, IFAC, Gandia Beach*, pages 195–200. IFAC. (to appear online in IFAC-PapersOnLine.net).

Barkalow, A. and Wegrzyn, M. (2006). *Design of Control Units with Programmable Logic*. University of Zielona Gora, Zielona Gora.

Hefferon, J. (2008). *Linear Algebra*. electronic edition, Colchester.

Karatkevich, A. and Baranov, S. (2010). Graph based approach to test bench constructing for datapath. In *IWK'10, 55th Internationales Wissenschaftliches Kolloquium*, pages 662–667. Technische Universitaet Ilmenau.

Murata, T. (1989). Petri nets: properties, analysis and applications. *Proceedings of the IEEE*, 77:541–580.

Peterson, J. L. (1981). *Petri net theory and the modeling of systems*. Prentice-Hall.

Vanderbei, R. J. (2008). *Linear Programming: Foundations and Extensions*. Springer Verlag, 3rd edition.

Wisniewski, R. (2009). *Synthesis of compositional microprogram control units for programmable devices*. University of Zielona Gora, Zielona Gora.