Decision Diagram Synthesis from VHDL

by

Gert Jervan

A Master Thesis
Submitted to the Chair of Computer Engineering and Diagnostics of
the Department of Computer Engineering

In fulfillment of the requirements for the
Degree of Master of Science
Computer Engineering

Tallinn Technical University
Tallinn 1998

Acknowledgments

First of all, I would like to thank my supervisor Prof. Raimund Ubar, who has supported and encouraged me during the whole period, when I was at Design & Test Center. He has always been admirable generator of wonderful ideas and gave me excellent guidance. A special thank should also go to some other members of Design & Test Center: Jaan Raik, Priidu Paomets, Eero Ivask, Antti Markus, Marek Mandre, Jüri Põldre. They have through the past few years grown to be more than colleagues but good friends. Their support, encouragement and wonderful working atmosphere at D&T Center was very important.

Many thanks also to the people at Department of Computer Engineering of Tallinn Technical University and to the ESLAB group at Linköping University. Especially Prof. Zebo Peng.

Also it is necessary to mention, that this work was partially funded by Estonian Science Foundation grant 1850, German-Estonian project EST 008-96 funded by BMFT Germany and The Johnson Foundation.

Gert Jervan

Linköping

May 1998

Table of Contents

List of Figures

List of Tables

Introduction

The development of the very large scale integrated circuit (VLSI) technology has brought tremendous testing problems. Test development cost and testing cost have become an important proportion of the total product costs. Studies on the development of application specific integrated circuits (ASIC) show that testing accounts for nearly a third of the ASIC development cycle [Lev92]. Classical gate level test generation has become therefore impractical, and researchers have devoted much effort to developing an alternative field of functional testing that decreases test generation complexity [TA80, GA85, GMM89]. However, estimation of the fault coverage at the functional level is difficult due to the fact that the functional level fault model does not characterize physical faults as well as gate-level models do, and due to a big diversity of different functional models [TA80, LS85, Gho88, Sch92, CA94, MG96]. A solution of these shortcomings, hierarchical test generation, appeared in the last decade [BH90, LP92]. It incorporates the benefits of functional testing yet retains the accuracy of gate level fault models. The system is considered at different levels, and tests are created on these levels by separate tools.

Hierarchical Test Generation System

The hierarchical test generation environment [BJM+97a], [BJM+97b], [BJM+98], [JKRU96], [JMRU97a], [JMRU97b], [JMRU98a], [JMRU98b] [Rai97] consists of hierarchical test generator, high-level and low-level decision diagram (DD) model generators but also tools for high-level synthesis (HLS) and logic-level synthesis. The hierarchical Automatic Test Pattern Generator (ATPG) operates with the Structurally Synthesized Binary Decision Diagrams (SSBDD) on the lower level and with the high-level DDs on the higher abstraction level. Figure 1 presents the basic design flow and the place of the test generation there.

From behavioral level Very high speed integrated circuits Hardware Description Language (VHDL) descriptions, HLS tool generates a register-transfer level (RTL) description of the circuit. The previous versions of hierarchical test generation system (HTGS) employed example designs from SYNT HLS system as an input of the whole system. In the new version of design flow HLS tool CAMAD will be used.

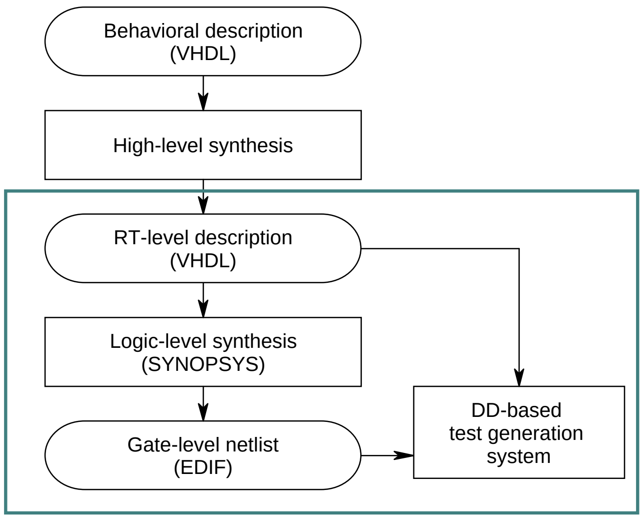

Behavioral description (VHDL)
High-level synthesis
RT-level description (VHDL)
Logic-level synthesis (SYNOPSYS)
Gate-level netlist (EDIF)
DD-based test generation system

Figure 1 The Design Cycle

In the RTL descriptions the design has been partitioned into a control part and a datapath part containing a network of interconnected functional units (FU). From RTL VHDL descriptions, high-level DD generator generates high-level DD models, which will be applied as input for the hierarchical test generator.

To generate local, structural level tests for the FUs of the design SSBDDs will be required. SSBDDs will be created from the gate-level netlist. Current system uses for logic level synthesis Design Compiler [Syn96] from Synopsys Inc. As a result of logic synthesis gate-level EDIF 2.0.0 netlist of the whole design and of each FU separately will be written out. Subsequently, the netlist is converted into SSBDDs.

As an output the ATPG generates test patterns. However, the patterns do not offer precise information about achieved fault coverage. In order to measure the actual gate-level fault coverage of the generated tests, the test patterns have to be fault simulated on the structural level description of the whole device.

In Figure 2 general structure of the hierarchical test generation environment is represented.

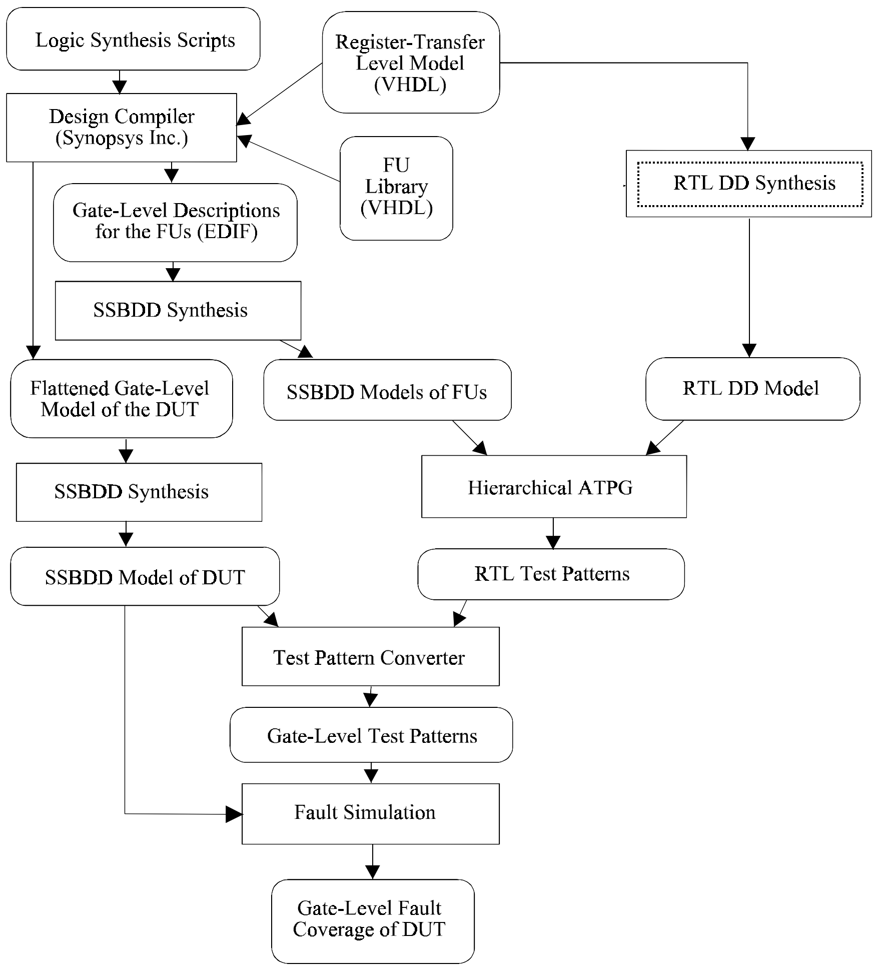


Figure 2 Hierarchical Test Generation Environment

Current thesis covers only small part of the whole hierarchical test generation environment. It focuses mainly to the high level DD generator, but also overview of theory of Decision Diagrams will be given and experimental results of the whole system will be presented. More precise information about hierarchical test generation algorithm can be found from [Rai97].

High-Level Synthesis

Due the rapid advances in technology, but also by the progress in the development of design methodologies as well as tools to support the designer in applying them, the fabrication of more and more complex electronic systems has been made possible in recent years. Those tools allow designers to move higher and higher levels of abstraction. In order to manage complexity, the design process can be decomposed according to [MLD92] into a series of subtasks, which deal with different issues.

1. System-level synthesis: The specification of the system at the highest level of abstraction is given by its functionality and a set of implementation constraints. The task of this step is to decompose the system into several subsystems (communicating processes) and to provide a behavioral description for each of them.

2. High-level synthesis starts out with an algorithmic description which specifies the computational solution of the problem, in terms of operations on inputs in order to produce the desired outputs. Elements, which appear in descriptions are similar to those of programming languages, including control structures and variables with operations applied to them. Three major subtasks are:

    - resource allocation (selection of appropriate FUs)
    - scheduling (assignment of operations to time slots)
    - resource assignment (mapping of operations to FUs).

3. RT-Level synthesis usually takes as input a description consisting of a data path and controller, presented as abstract FSM. For data path, an improvement of resource allocation and assignment can be done, while for the control path, actual synthesis is performed by generating the appropriate controller architecture from the input consisting of states and state transitions.

4. Logic-level synthesis receives as input technology independent description of the system, specified by blocks of combinational logic and storage elements. It deals with the optimization and logic minimization.

5. Technology mapping has the task of selecting appropriate library cells of a given target technology for the network of abstract gates produced as a result of logic synthesis, concluding thus the synthesis pipeline. Input of physical level

synthesis is a technology independent multi-level logic structure, a basic cell library, and a set of design constraints.

The HLS process can be divided into the three subtasks: resource allocation, operation scheduling and resource assignment. Resource allocation determines the types (for example, adder, multiplier or register) and the number of these types of resources that should be included in the design. The operation scheduling assigns each operation in the design to a time step in which it will be executed. Resource binding determines which resources should be used to implement each specific operation.

The output of a HLS system is a description at RT level, consisting of a data part which performs operations on the input data in order to produce the required output and a control part which controls the type and sequence of data manipulations. Communication and synchronization between the two parts is achieved via conditional flags and control signals. A typical data part consists of FUs, storage and interconnected hardware, while the controller is specified as a state-transition table used in later stages for controller synthesis.

The CAMAD High-Level Synthesis System

The CAMAD (Computer-Aided Modelling, Analysis and Design of digital systems) is high-level synthesis system [PKL89], developed at the Linköping University. The design methodology employed in the system is based on an extended timed Petri Net representation uniformly used throughout the design cycle. The input to CAMAD can be specified in Algorithmic Design Description Language (ADDL) [Fje92] or in S'VHDL [EKPM92]. S'VHDL was defined as a subset of the VHDL language with the purpose of using it as input for high-level hardware synthesis. The S'VHDL compiler is described in detail in [Min93].

The SYNT High-Level Synthesis System

The SYNT HLS system was originally developed by SYNTHESIA AB (Stockholm, Sweden). After merging with Cadence Design Systems, Inc. SYNT

HLS system is enhanced and available nowadays as a Cadence Alta Group Visual Architect™ system[1].

The VHDL Hardware Description Language

The IEEE Standard VHDL hardware description language has its origin in the United States Government's Very High Speed Integrated Circuits (VHSIC) program, initiated in 1980. The development of VHDL was sponsored by the US Department of Defence during the 1980s. In 1987 the language was adopted by the IEEE as a standard; this version of VHDL is known as the IEEE Std. 1076-1987 [IEEE87]. A new version of the language, VHDL'92 (IEEE Std. 1076-1993) [IEEE93], resulted after revision of the initial standard in 1993. In most cases, the language is upward compatible.

VHDL is designed to fill a number of needs in the design process. It allows a multi-level descriptions, providing support for both a behavioral and a structural view of hardware models, their mixture in description being possible.

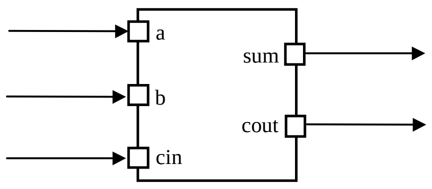
A digital electronic system can be described as a module with inputs and/or outputs. This module is called, using VHDL terminology, *design entity*. Each entity can be described as a set of *design units*. There are five different types of design units:

1. Entity declaration
2. Architecture body
3. Package declaration
4. Package body
5. Configuration declaration

The *entity declaration* provides the interface of the component to the environment, it includes a name associated with the entity and a list of ports, through which the entity communicates with its external environment. As an example, the entity

[1] http://www.cadence.com/alta/products/va_overview.html

declaration for an XOR gate with input ports X and Y, and output port Z is presented in Figure 3 (adapted from [LSU89] and [Min93])[2]



```
entity full_adder is
   port(a, b, cin: in bit; -- input ports
        sum, cout: out bit); -- output ports
end full_adder;
```

Figure 3 Full-adder: entity declaration.

The *architecture body* describes how the entity is implemented. An architecture body contains an optional declarative part and a statement part, consisting a number of *concurrent statements*, which describe the internal details of the entity using the behavioral or structural modeling style or a combination of the two. A sample behavioral description for the adder is given in Figure 4.

```
architecture sequential_behavior of full_adder is
begin
  process
    variable s: bit;
  begin
    s := a xor b;
    sum <= s xor cin after 5 ns;
    cout <= (a and b) or (s and cin) after 10 ns;
    wait on a, b, cin;
  end process;
end sequential_behavior;
```

Figure 4 Full adder: behavioral description

A package is collection of declarations such as subprograms, types, constants, components, and possibly others, which are grouped in a way that allows different design units to share them. The interface to the package, consisting of those

[2] Reserved words are written in **boldface.**

declarations which are intended to be seen from the outside, is defined in the *package declaration.* An example of package declaration is given in Figure 5. The *package body,* on the other hand, contains the hidden details, which are not visible from the outside (implementations of the functions etc.). Packages are a very efficient way for creating a standard or vendor-specific VHDL environment. The VHDL language standard includes two predefined packages: STANDARD and TEXTIO. The package STD_LOGIC_1164, which is an IEEE standard, defines a nine-value logic type with associated operators.

```
package Logic is
  type Three_level_logic is (´0´, ´1´, ´Z´);
  constant Unknown_value : Three_level_logic:=´0´;
  function Invert (Input : Three_level_logic)
    return Three_level_logic;
end Logic;
```

Figure 5 Package declaration example

Configuration declarations are an advanced facility for structural specification and will not be discussed in this thesis.

The major modelling element for behavioral specifications in VHDL is the process. A process is a sequential body of code which can be activated in response to changes in state. Processes can be executed concurrently. The statement body of a process consists of *sequential statements* which are: variable assignment, if statement, case statement, loop statement, next statement, exit statement, assertion statement, report statement, wait statement, signal assignment, procedure call, return statement and null statement.

Additional information about VHDL language can be found from [IEEE93].

VHDL Subset for Test Generation

As described earlier and depicted in Figure 6, for HTGS, HLS tool generates from behavioral level VHDL descriptions a RT level description of the circuit. This RT-level VHDL description will be used as input for high-level DD generator. In the following, mainly the VHDL subset, which is generated by HLS system CAMAD is described.

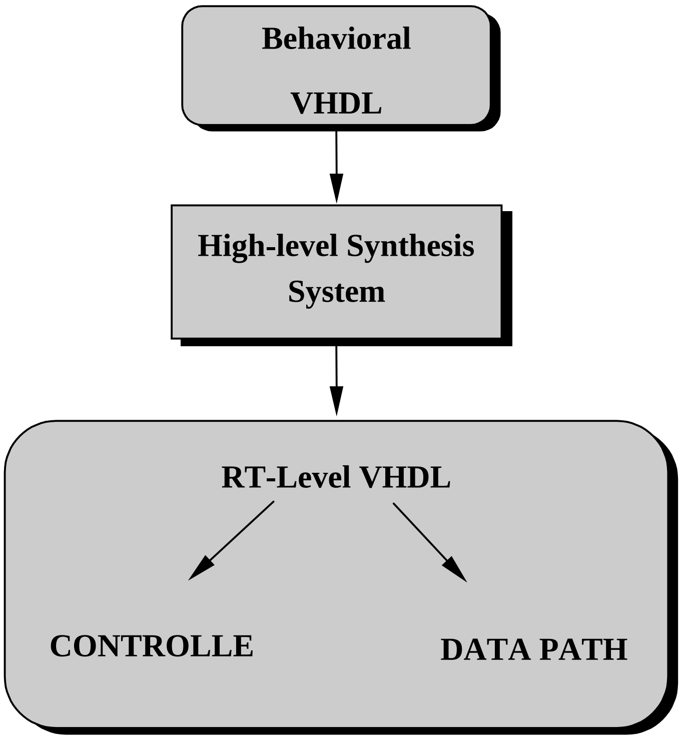


Figure 6 Design flow from behavioral to structural representation

The overall style of the design supported by the high-level DD-generator is an module described as entity and one architecture, written in structural mode. Design should be partitioned into the datapath and controller. Datapath is represented by a netlist of interconnected blocks. The building blocks of the datapath are registers, multiplexers and FUs, where functions can be arbitrary arithmetic or logic operations. A full sample datapath example is depicted in Figure 7. P_N1, P_N5 and P_N6 are primary inputs, P_N6 is primary output and signals C3, C4, C5 and C6 are conditional signals.

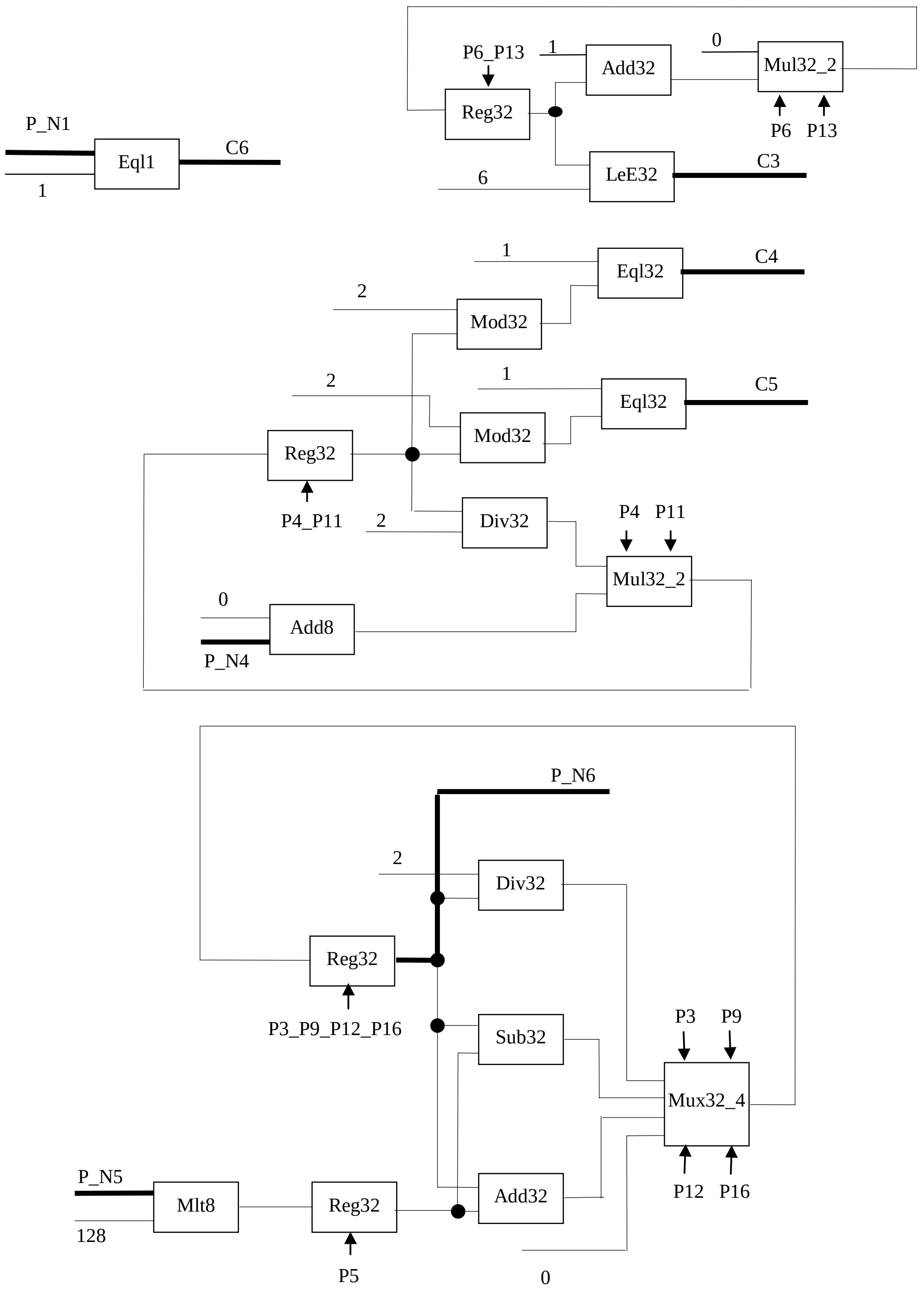


Figure 7 Data-path example of a Device

The control part of a device is described as a Finite State Machine (FSM) state table, described in behavior.

Entity declaration

The syntax[3] for declaring an entity is:

```
entity_declaration ::=
  entity identifier is
    entity_header
  end;

entity_header ::= formal_port_clause
port_clause ::= port (port_list);
port_list ::= port_interface_list
```

In port list all primary inputs and outputs should be described. Reset signal should always carry the name 'Reset' and clock signal should be named 'Clock.' At the moment only `IN` and `OUT` ports are allowed. Allowed signal types are `std_logic`, `std_logic_vector`, `u_std_logic_vector`, and types declared in ETPN_modelling package (see Appendix A: Package ETPN_modelling). Vectors are allowed in both directions (`downto` as well as `to`)

Architecture declaration

Architecture should describe structure of the entity.

An architecture body can be described using following syntax:

```
architecture_body ::=
  architecture identifier of entity_name is
    architecture_declarative_part
  begin
    architecture_statement_part
  end;

architecture_declarative_part ::=
      block_declarative_item

architecture_statement_part ::= concurrent_statement

block_declarative_item ::=
   | signal_declaration
```

[3] Throughout this thesis, the syntax of language features is presented in Backus-Naur Form (BNF) [Bac59].

```
       | component_declaration
       | configuration_specification
       | type_declaration

   concurrent_statement ::=
      block_statement
       | component_instantiation_statement
```

Signals are declared using the syntax:

```
   signal_declaration ::=
     signal identifier_list : subtype_indication ;
```

All components, which will be used must correspond to the entities described in the special library (See Appendix B: Library for describing design entities).

Separately should be declared type `State_type` and 2 special signals: `present_state` and `next_state.` Example of FSM state declaration:

```
-- FSM state declaration
TYPE State_type IS (S1, S2, S3, S4, S5, S6, S7, S8,
   S9, S10, S11, S12, S13, S14, S15, S16, S17, S18);
SIGNAL present_state : State_type;
SIGNAL next_state : State_type;
```

Note, that states should carry names S1, S2, ..., Sn

Component instantiation part describes the dataflow of the design. Component Instantiation has the following syntax:

```
   component_instantiation_statement ::=
     instantiation_label :
       component_name
         port_map_aspect ;
```

FSM of the design is described with 3 processes. All processes should carry specific names: *state_register*, *output_decode_logic* and *state_decode_logic*. *State_register* process should deal with reset handling and represent the relation between clock and state changes. *Output_decode_logic* process represents assignment of values to control signals depending of the value of present_state signal and s*tate_decode_logic* process represents the next state logic.

Output_decode_logic and *state_decode_logic* processes are represented as single `CASE` statements. Inside the `CASE` statement, only `WHEN` statements and signal assignments are allowed as shown in the following example. `WHEN OTHERS` part is optional.

```
CASE present_state IS
         WHEN S1 =>
            next_state <= S2 ;
         WHEN "001" =>
      ...
CASE ;
```

Inside `WHEN` statement currently are also allowed `IF` statements and signal assignments like in the following example:

```
WHEN S1 =>
     IF C5 = '1' THEN
        next_state <= S2 ;
     ELSE
        next_state <= S3 ;
     END IF ;
...
```

At the moment only structure `IF-THEN-ELSE` is allowed (`ELSIF` is forbidden).

In addition to the control signals, combinations of control signals are also allowed. For example, the following statement is legal:

```
P3_P9_P12_P16 <= P3 OR P9 OR P12 OR P16
```

Note, that only OR operators are allowed at the moment.

All constants should be represented in decimal format.

Full example of the design is represented in Appendix C: Design example (RT-Level VHDL).

Decision Diagrams

Decision Diagrams (DD) (previously known as Alternative Graphs) [Uba76, Uba96] may represent a set of digital (Boolean or integer) functions $y=F(x)$ of components or subcircuits in digital systems. Here, y is an output variable, and X is a vector of input variables of the component or subcircuit.

Definition 1. In the general case, a DD that represents function $y=F(X)$ is a directed, noncyclic graph $G_y = (M, \Gamma, X)$ with set of nodes M, single root node $m_0 \in M$, and relation Γ in M, where $\Gamma(m) \subset M$ denotes the set of successor nodes of m. Nonterminal nodes m for $\Gamma(m) \neq \varnothing$ have variables $x_i \in X$ as labels. Terminal nodes m for $\Gamma(m) = \varnothing$ have variables x_i, functional subexpressions of $F(X)$, or constants as labels. Let $x(m)$ be the label of node m. In graph G_y, for all nonterminal nodes m for which $\Gamma(m) \neq \varnothing$, a one-to-one correspondence exists between the values of label variable $x(m)$ and the successors, $m_k \in \Gamma(m)$ of m.

When using DDs to describe complex digital systems, we have, at the first step, to represent the system by a suitable set of interconnected components (combinational or sequential ones). At the second step, we have to describe these components by their corresponding functions which can be represented by DDs. DDs which describe digital systems at different levels may have special interpretations, properties and characteristics, however, the same formalism and the same algorithms for test and diagnosis purposes can be used, which is the main advantage of using DDs. DDs were originally introduced and proposed for diagnostic purposes by R. Ubar in 1976 [Uba76]. Binary Decision Diagrams (BDD) that were presented later by S. B. Akers [Ake78] are in fact a special case of DDs. The concept of DDs is more general than the concept of BDDs. In the following some examples of digital system types and their representation by DDs will be given.

Gate-Level Combinational Circuits

Each output of the combinational circuit is defined by some Boolean function which can be represented as a DD. The nodes of this type of DD are labelled by Boolean variables and have consequently only two output branches. The terminal nodes are labelled by logical constants 0 and l, or Boolean variables. This type of DDs is called *Structurally Synthesized Binary Decision Diagrams* (SSBDD). As an example, in Figure 8 representation of a combinational circuit by SSBDD is given. For the sake of simplicity, the values of variables on branches are omitted. By convention, the right-hand branch corresponds to 1 and the lower-hand branch to 0. In addition, terminal nodes holding constants 0,1 are omitted. Exiting from the SSBDD to the right corresponds to y =1, and exiting the SSBDD downwards corresponds to y = 0.

In SSBDDs there exists an one-to-one relationship between nodes and signal paths in the corresponding combinational circuit. This property of SSBDD is very important because it allows us to generate tests for structural faults in circuits. The original idea of SSBDDs was introduced into the test area in [Uba76].

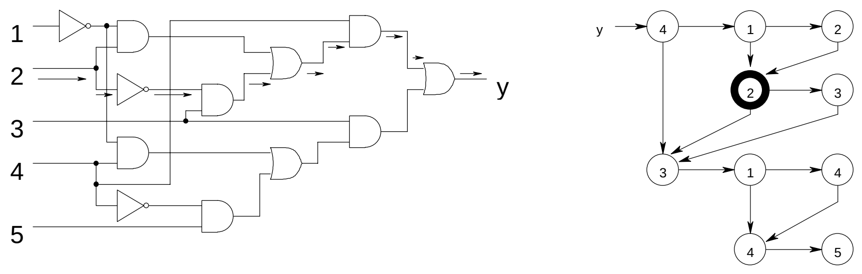


Figure 8 SSBDD for a Combinational Circuit

Similar to superposition of functions, superposition of DDs has been defined. Separate SSBDDs are to be created for maximal subcircuits that do not contain reconvergencies. In this case, each node in a SSBDD will represent a signal path in the tree-like fanout-free subcircuit. Thus, the number of the nodes in SSBDDs will be equal to the number of different paths in the tree-like subcircuits.

SSBDD models for combinational circuits can be synthesized by a simple superposition procedure. Generation of SSBDD starts from circuit output. During the generation, all gates will be recursively substituted by their respective elementary BDDs until primary inputs are reached. In order to avoid repetitive occurrences of subdiagrams in the model, the recursion can be terminated in fanout branches and SSBDDs can be synthesized for each primary output and fanout point separately. In that case the circuit will be described as a system where for each fanout-free region an SSBDD corresponds.

Digital Systems on the Register Transfer Level

In Boolean DD descriptions the DD variables were Boolean (i.e. single bit) values, whereas in register-transfer level DD descriptions, in general case, multi-bit variables are used. Traditionally, on this level a digital system is decomposed into two parts - datapath and control part. Datapath is represented by sets of interconnected blocks (functional units), each of which can be regarded as a combinational circuit, sequential circuit or a digital system on the instruction interpretation or micro-operation level. In order to describe these blocks, corresponding types of DDs can be used, as discussed above.

The datapath can be described as a set of DDs, in the form where for each register and for each primary output an DD corresponds. Here, the non-terminal nodes represent the control signals coming from the control part and terminal nodes represent signals of the datapath, i.e. primary inputs, registers, operations, constants. Figure 9 shows a datapath fragment and its corresponding DD model.

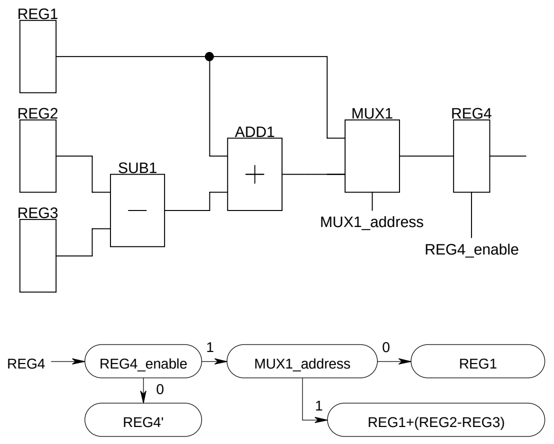


Figure 9 DD Representation of the Datapath

The control part FSM is described as a FSM state table. The state table can be represented by single DD where non-terminal nodes represent current state and inputs for the control part (i.e. logical conditions), and terminal nodes are for representing the next state logic and control signals going to the datapath. Figure 10a shows an example of a fragment of FSM state table and Figure 10b shows the corresponding DD representation. In the graph example, *q* denotes the next state and *q'* denotes current state value. The DD in Figure 10 describes the behavior of the FSM at the current state being equal to s3.

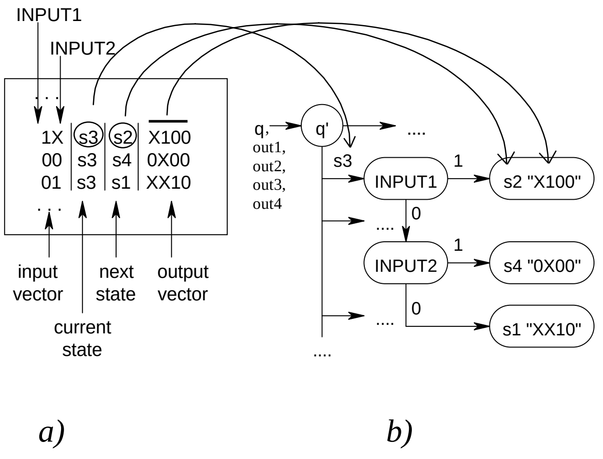


Figure 10 DD Representation of FSM

Digital Systems on the Behavioral Level

DDs describing digital systems on the behavioral level describe behavior instead of structure of the system. In DD, the variables in nonterminal nodes can be either Boolean (describing flags, logical conditions etc.) or integer (describing instruction words, control fields, etc.) The terminal nodes are labelled by constants, variables (Boolean or integer) or by expressions for calculating integer values. The number of DDs, used for describing a digital system, is equal to the number of output and internal variables used in the instruction set description. More details about using DDs to describe digital systems at the instruction interpretation level can be found in [Uba83,Uba88].

Decision Diagram synthesis from VHDL

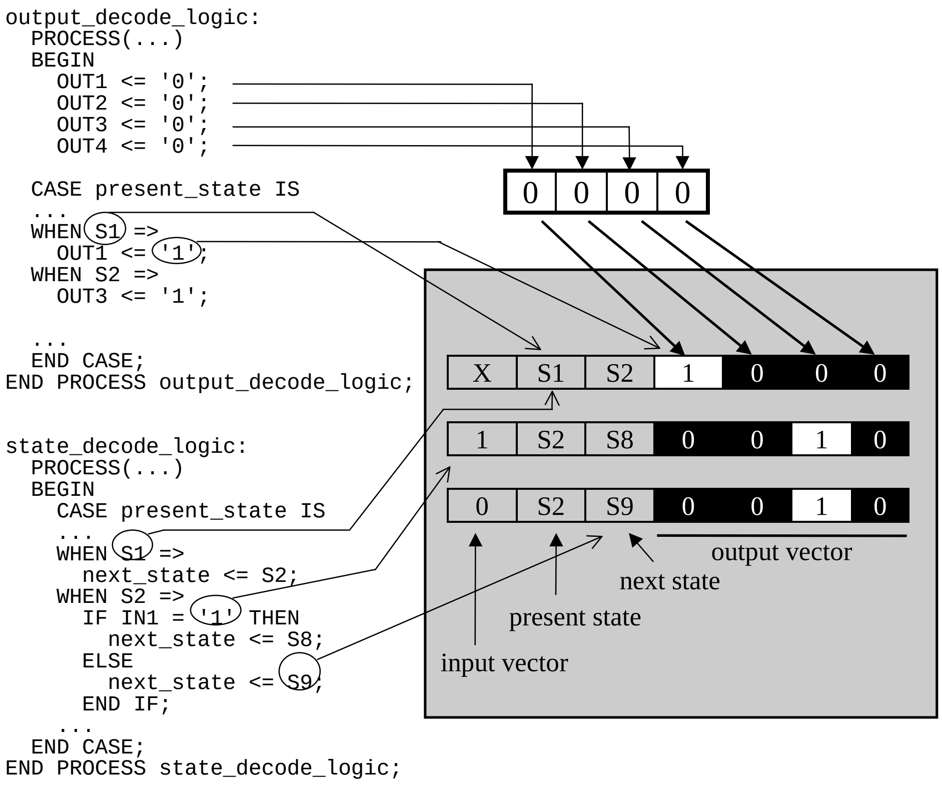
DD synthesis from VHDL consists of several steps. In RT-Level design is partitioned into datapath and control part. Both parts will be converted into DDs separately. Additionally entity declaration, signal declarations and constants should be converted into DD model format. After initial conversion datapath DDs have usually some redundancy which should be removed. As a final step, DD-model of the whole design will be written out.

To support different phases of test generation, special object-oriented library of data structures has been developed. Those classes provide exhaustive way to handle DDs and is based on special DD model file format (See Appendix D: DD Model Format). As a result, development times of different testing related software diminishes tremendously.

Control Part DD model generation

The control part is represented as a FSM state transition table behaviorally described in VHDL, as was discussed in section ”VHDL Subset for Test Generation.” DD model is generated in two steps. At the first step VHDL description is converted into intermediate memory structures, which describe FSM state table. At this step two VHDL processes (`output_decode_logic` and `state_decode_logic`) is merged together. In Figure 11 the process of creating a FSM state table from VHDL description is depicted. At the second step intermediate memory structures is converted into DD memory structures. Used VHDL constructs and description style were described in detail in section ”VHDL Subset for Test Generation.”

FSM state transition table is described in VHDL as two separate processes. One of those processes (`output_decode_logic)` describes output signal values in relation with the current state value. Second one (`state_decode_logic)` determines the next state value.



```
output_decode_logic:
  PROCESS(...)
  BEGIN
    OUT1 <= '0';
    OUT2 <= '0';
    OUT3 <= '0';
    OUT4 <= '0';

  CASE present_state IS
  ...
  WHEN S1 =>
    OUT1 <= '1';
  WHEN S2 =>
    OUT3 <= '1';

  ...
  END CASE;
END PROCESS output_decode_logic;


state_decode_logic:
  PROCESS(...)
  BEGIN
    CASE present_state IS
    ...
    WHEN S1 =>
      next_state <= S2;
    WHEN S2 =>
      IF IN1 = '1' THEN
        next_state <= S8;
      ELSE
        next_state <= S9;
      END IF;
    ...
  END CASE;
END PROCESS state_decode_logic;
```

Figure 11 FSM State Table Generation from VHDL Description

In the `output_decode_logic` process, after the process header, all output signals should be initialized. This template is used on the following steps for output vector creation. All values which are not covered by the new values inside `CASE` statement stay the same as in the template vector (black cells in output vector in Figure 11).

The `state_decode_logic` process contains only one `CASE` statement. Inside `CASE` statement there is, like in `output_decode_logic` process, multiple `WHEN` statements, which determine next state values corresponding to present state values. The input vector is extracted by Boolean expression analysis in the `IF-ELSE` statement. In the present example we can see that two state transition table lines were created from one `WHEN` construct analysis.

In Figure 12 is shown how DD-model is created from FSM state table description.

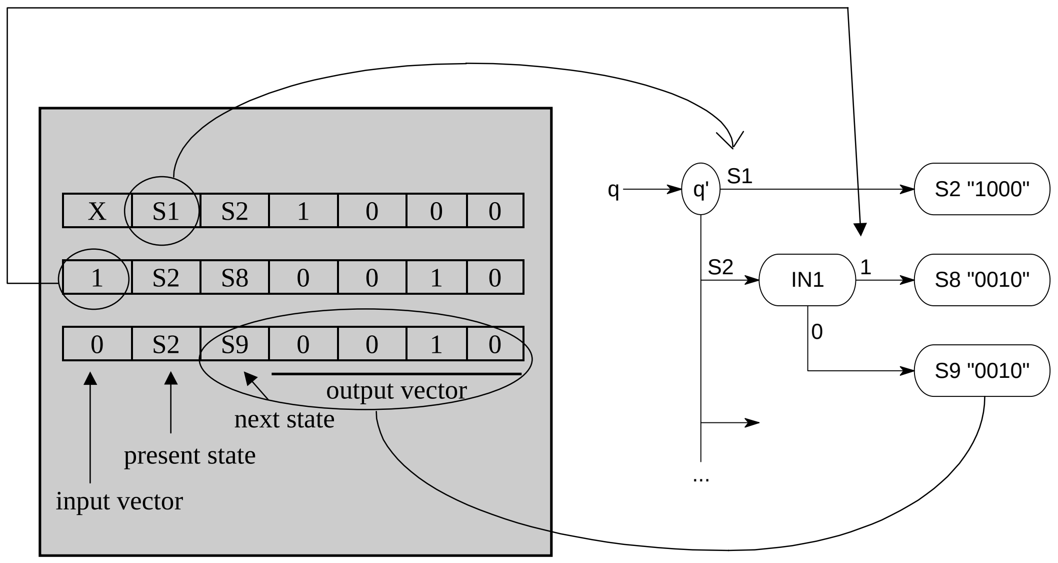


Figure 12 DD generation from FSM State Transition Table

In this example is shown the fragment of a DD for three lines of FSM state table obtained in Figure 11. As was said previously for FSM only one DD will be created. Here variable q’ represents the present state. Depending on its value, and the value in nodes weighed by input variables, next state is determined, as well as output values vector, which are given in the terminal nodes.

Datapath DD-model generation

The datapath DD-model is created from the device data-path specification in VHDL, which example was represented in Figure 7 and in Appendix C: Design example (RT-Level VHDL). As was already mentioned, data-path VHDL description is based on the library of predefined modules (registers, multiplexers, arithmetic and logic blocks, see Appendix B: Library for describing design entities). Due to this fact, the datapath DD-model creation is based on the predefined set of DD modules. For example, predefined DDs for registers and multiplexers are defined. And the general DD-model creation is based on the assembly of these small DD modules into one model.

In the following descriptions of different DD modules will be given.

Registers

The exact type, which kind of registers will be used is determined by the HLS tool. In current version of design flow, registers without reset signal is assumed. The corresponding DD is following:

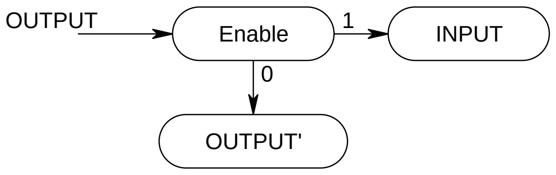


Figure 13 DD module for register

Multiplexers

As registers type so the multiplexers type is also predefined by HLS. HLS tool CAMAD defines following type of multiplexers: Input IN_i is selected when control signal C_i is ”1” and all other control signals are ”0”. In Figure 14 truth table and DD for multiplexer with 3 inputs is represented.

C1	C2	C3	OUT
1	0	0	**IN1**
0	1	0	**IN2**
0	0	1	**IN3**
All other combinations			**”0”**

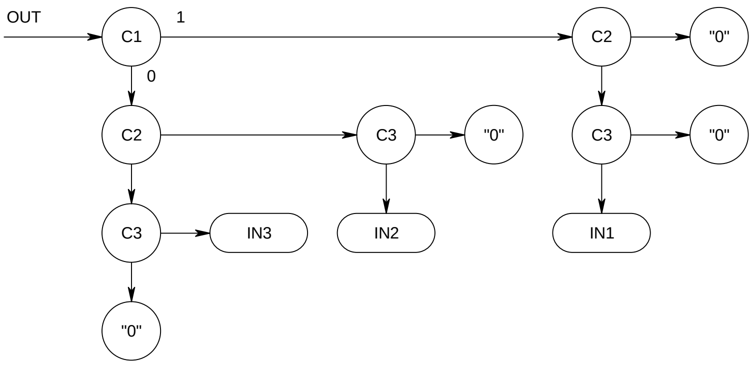
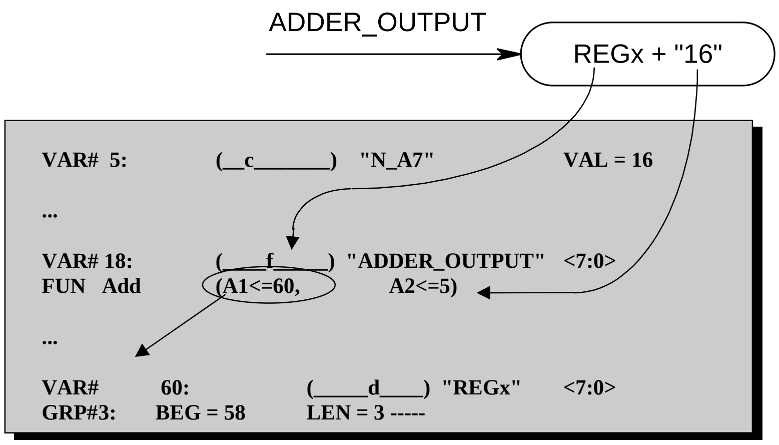


Figure 14 Truth table and DD of 3-input multiplexer

Arithmetic and Logic Blocks

All arithmetic and logic blocks are represented as a single node graphs. As an example, DD for arithmetic block adder is represented in Figure 15. All arithmetic and logic blocks should be represented with their inputs and outputs as functions in the final DD-model.



```
VAR#  5:          (__c_______)   "N_A7"              VAL = 16
...
VAR# 18:          (____f_____)  "ADDER_OUTPUT"   <7:0>
FUN   Add         (A1<=60,          A2<=5)
...
VAR#        60:              (_____d____)  "REGx"      <7:0>
GRP#3:      BEG = 58         LEN = 3 -----
```

Figure 15 Arithmetic block DD and corresponding lines in DD-model

At the first step of datapath DD creation, for every FU (multiplexer, register, arithmetic or logic block) corresponding DD will be created. At the second step minimization of the DD-model will be done.

For example, in Figure 16 is depicted sample datapath fragment, which contains one register, one multiplexer and one adder. This fragment may be initially represented by three DDs - for register, multiplexer and adder. DDs will be created as was described above and as shown in Figure 17.

When for every VHDL component instantiation statement in the datapath description corresponding DD is created, obtained model should be minimized. This process takes place as follows.

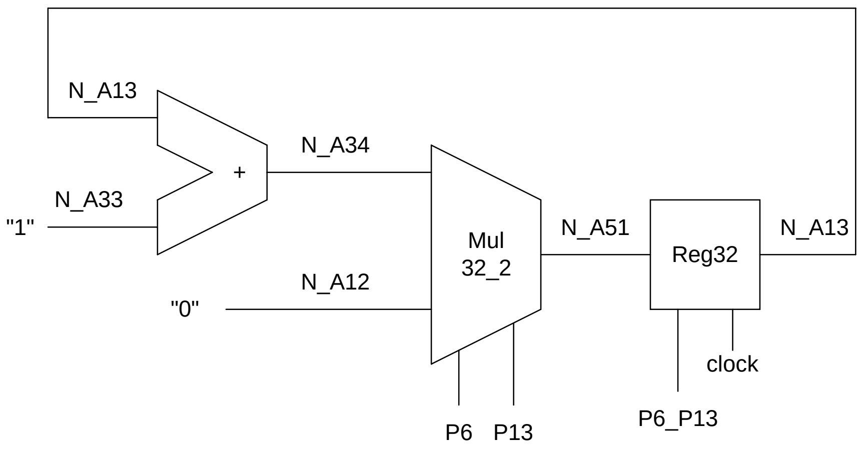
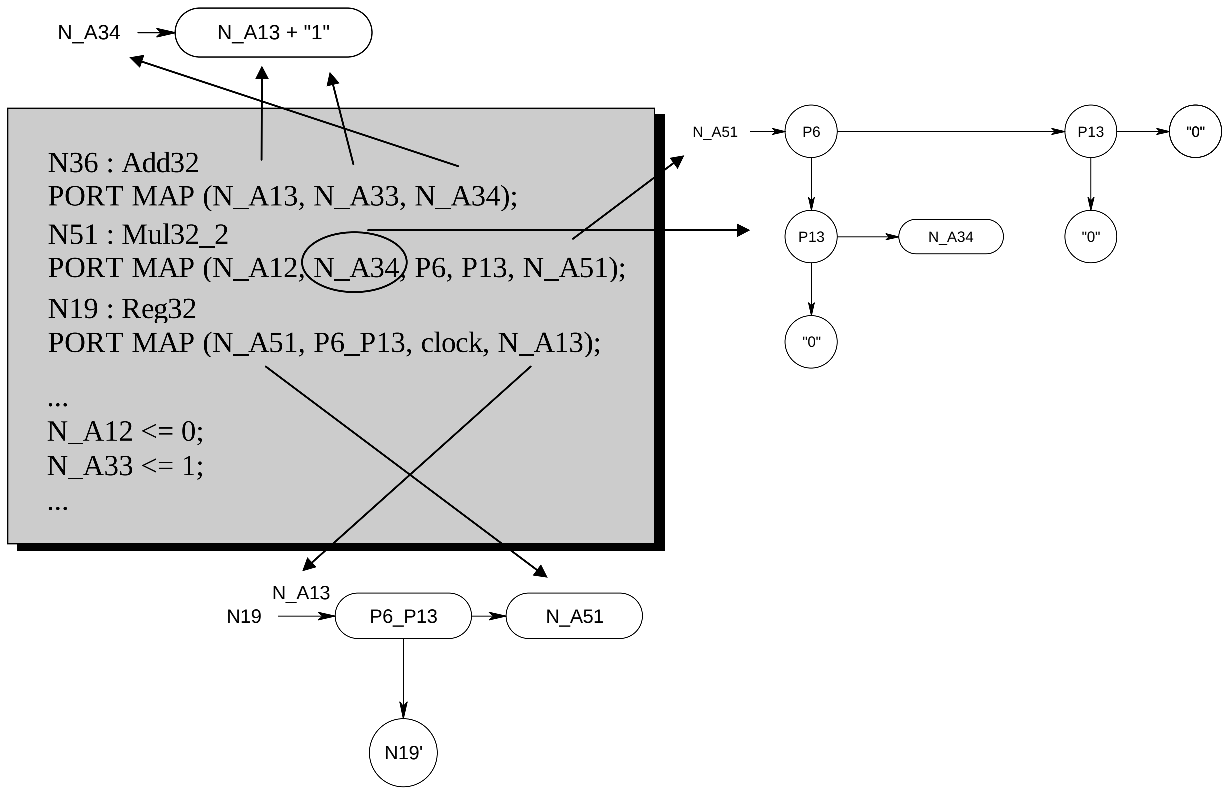


Figure 16 Datapath fragment



```
N36 : Add32
PORT MAP (N_A13, N_A33, N_A34);
N51 : Mul32_2
PORT MAP (N_A12, N_A34, P6, P13, N_A51);
N19 : Reg32
PORT MAP (N_A51, P6_P13, clock, N_A13);
...
N_A12 <= 0;
N_A33 <= 1;
...
```

Figure 17 Datapath DD-Model generation from VHDL description

Suppose that the number of variables in initial DD-model $|X|=N_X$, the number of DDs is $|G|=N_g$, and the number of nodes in DDs is $|M|=N_m$. For all G_k built for variables $x_k \in \{ X_{data\_out}, X_m \}$, where

X_{data_out} is a set of output variables from datapath (primary outputs);

X_m is a set of register variables

we analyze the terminal nodes m_i which are weighted by a variable.

If m_i is weighted by a variable, and the weight variable $x_i \notin \{X_{data\_in}, X_{ctrl\_in}, X_m\}$,

where

X_{data_in} - is a set of input data variables (primary inputs);

X_{ctrl_in} - is a set of control input variables to both control and data-path (primary inputs);

X_m - is a set of register variables

and $\neg\ [\ \exists\ m_j |\ e(m_j)=f(x_i)\ ]$, i.e. no one terminal node is not weighted by variable x_i. (e(m) is a weight function for a node m). This means that we have found such a node m_i , and a variable x_i , represented by DD G_i, which may be eliminated from the model without the lost of information. This process is called superposition. We replace the node m_i by graph G_i . And in such a way Nx:=Nx-1, Ng:=Ng-1 and Nm:=Nm-1, the numbers of variables, graphs and nodes in the model are reduced by one. Example of such a minimization is given in Figure 18.

Initial DD-Model in Figure 18 consists of three graphs: the first one is for variable N_A34, the second one for N_A51, and the third one for register N19. Two nodes may be eliminated in the present case. They are shown by gray color. As a result whole datapath fragment in Figure 16. may be represented by only one DD, as depicted in Figure 18.

The full set of DDs describing datapath represented in Figure 7 can be found in Figure 19

The full DD-model example is given in Appendix E: DD-Model example. File format of the DD-model is given in Appendix D: DD Model Format

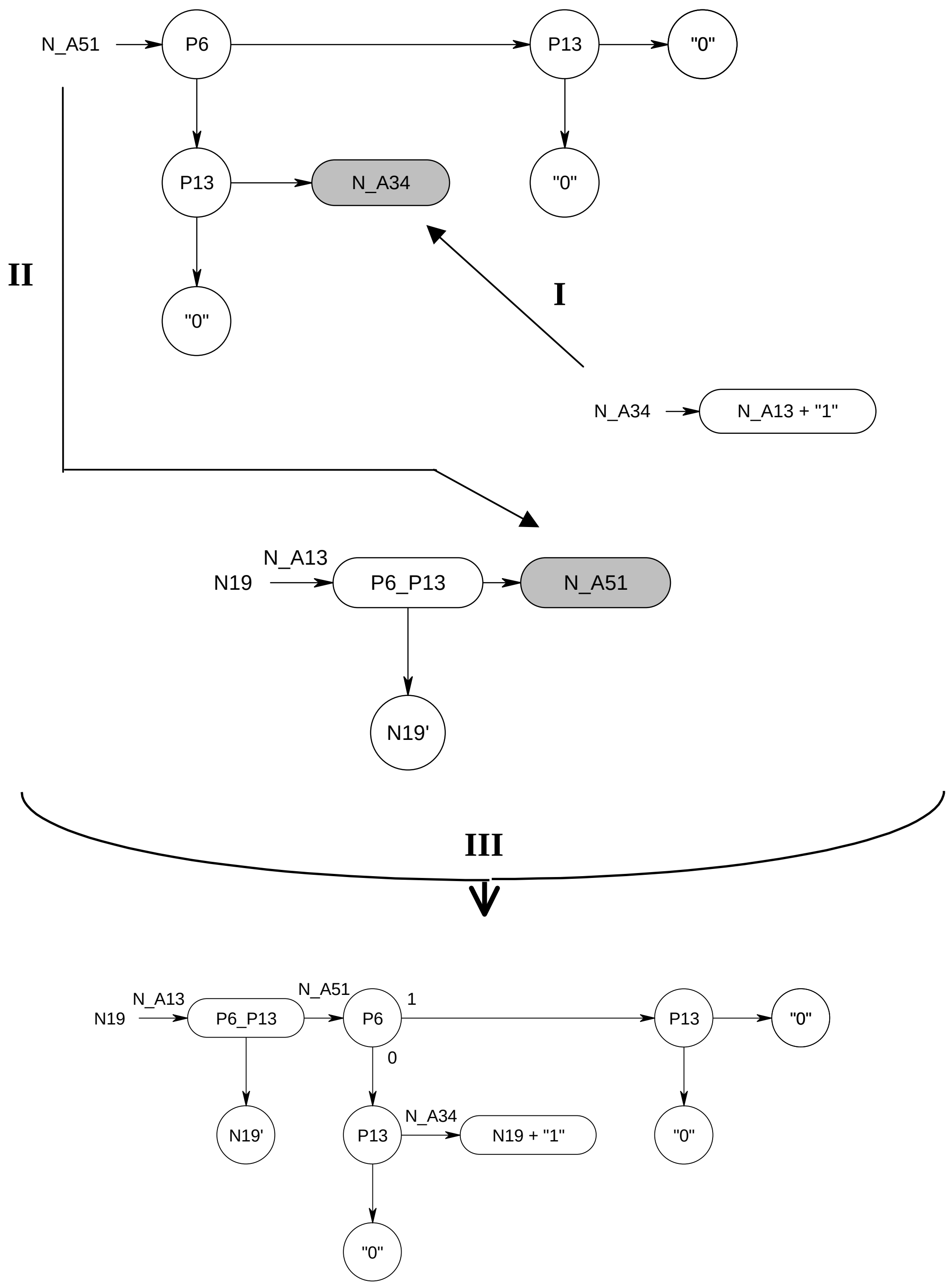


Figure 18 DD-Model minimization example

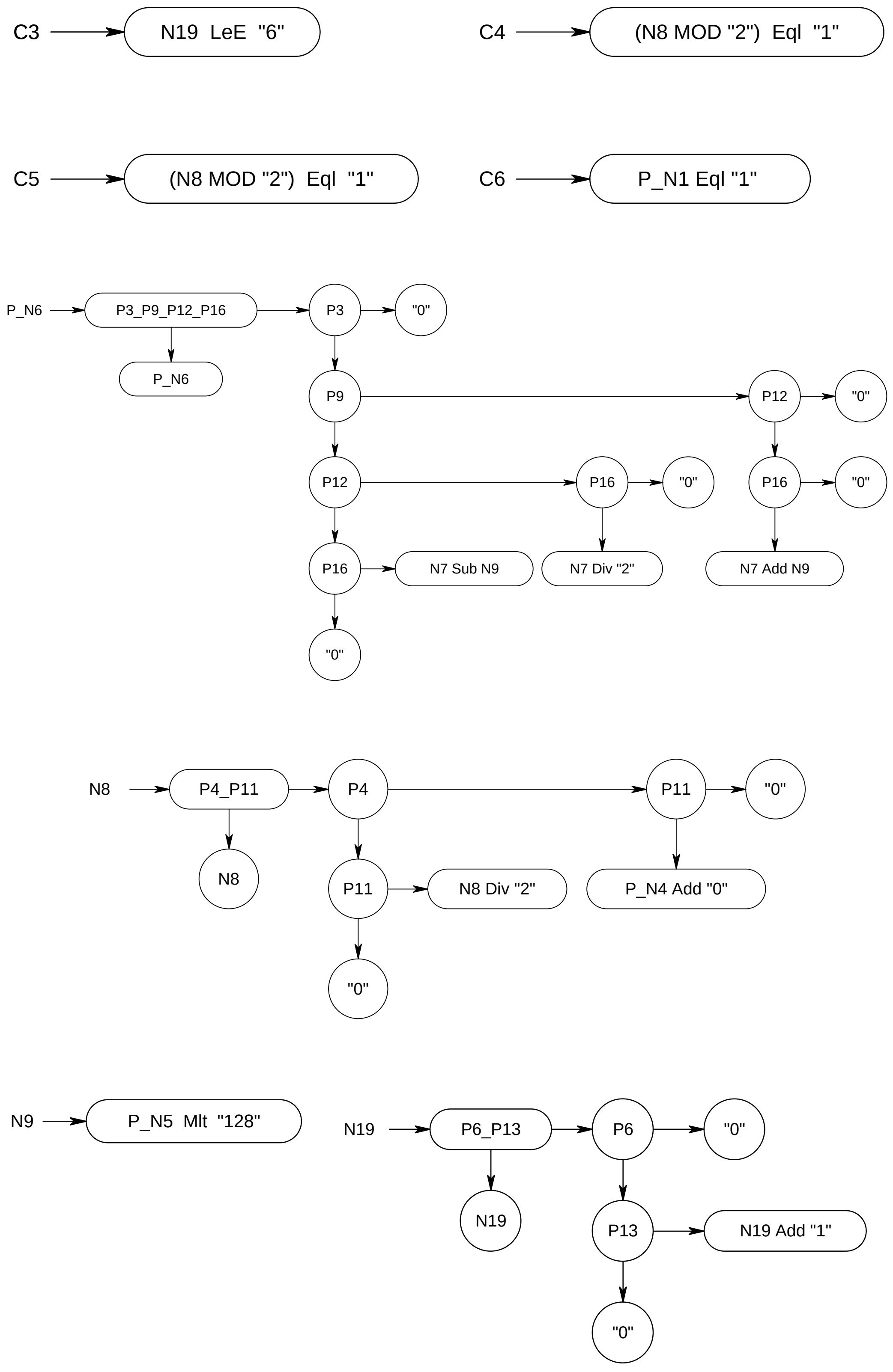


Figure 19 Representation of the datapath by a set of DDs

Experimental Results

In the following, experimental results of the hierarchical test generation system is presented. At the moment no explicit information about DD synthesis is given. The main reason is, that in all experiments, so far conducted, the DD synthesis time was less than 0,5 seconds and had no importance compare with test generation times. Please refer also to section future work for additional information about experimental part of this work.

Experiments were carried out on two highly sequential circuits, a Greatest Common Divisor (GCD), which belongs to the HLSynth92 benchmark suite, and an 8-bit multiplier example. The synthesized RTL version of the GCD circuit contains a datapath with 5 registers and an FSM with 12 states. The circuit was chosen as it reveals a number of difficult test generation problems. It contains a global data dependent loop and only one of the registers is directly observable. The multiplier mult8x8 has a complex datapath containing several feedback loops.

Actual quality of the generated test sequences was measured by applying gate-level fault simulation to the circuits and by neglecting a set of obviously untestable faults (e.g. lines tied to constants). The achieved fault coverages for datapath parts were 95.1 % for the GCD circuit and 95.9 % for mult8x8, respectively. Although control part faults were not explicitly targeted, fault coverages measured for control parts were high. Achieved test generation times were short. Both of the circuits were tested in less than 20 seconds. The number of test sequences was less than 100. Due to the fact that sequential circuits were considered, each test sequence consisted of multiple clock cycles. Table A presents the experimental results, which were run on a 233 MHz Pentium II computer with 64 MB RAM under Windows 95 operating system.

Circuit	**gcd**	**mult8x8**
Number of gate-level faults	1066	4432
Gate-level fault coverage DP (%)	95.1	95.9
Fault coverage CP (%)	89.4	92.1
Total gate-level fault coverage (%)	91.8	94.4
Test generation time (s)	14.6	17.7
Number of generated test sequences	53	93
Total test length (number of clock cycles)	627	2797

Table A Test Generation Results

In Table B, comparative results with a gate-level sequential test pattern generator HITEC [NP91], a genetic test pattern generator GATEST [RPGN94] and a novel hierarchical test pattern generation approach published in [RVE+98] are given. (Note that in [RVE+98], the high level test frames were generated manually). The comparison is carried out on the example of the GCD circuit, which is the only circuit common with the experiments in [RVE+98]. As we can see from the table, the proposed DD-based technique outperforms the other test generation tools in all categories. It achieves a higher fault coverage in a much shorter time and generates less test sequences than [RVE+98]. The number of test sequences for [NP91] and [RPGN94] is not known.

	DECIDER	Hier. [RVE+98]	GATEST [RPGN94]	HITEC [NP91]
fault coverage, %	**91.8**	90.4	62.6	74.4
time, s	**14.6**	1068	636	49320
test sequences	**53**	60	N.A.	N.A.

Table B Comparative Results

Conclusions and Future Work

Current thesis presents software for synthesizing Decision Diagrams from RT-Level VHDL descriptions. Additionally overview of hierarchical test generation environment, based on using multiple abstraction levels of DD models and theory of Decision Diagrams is given. Decision Diagrams serve as a mathematical basis for solving a wide spectrum of test tasks. The DD synthesizer generates DDs from RTL VHDL descriptions. Those descriptions are generated by HLS tool from behavioral VHDL descriptions.

Previous version of DD synthesizer used, as an input, VHDL descriptions generated by HLS system SYNT. The working prototype for the SYNT RT-Level VHDL output was finished during September 1997. From March till May 1998 I was as guest researcher at Linköping University. During this stay preparations for the next version of DD synthesizer were carried out. New version will use HLS tool CAMAD to promote further cooperation between Tallinn Technical University and Linköping University. At the moment all theoretical work is done and only actual programming work is left. According my plans, this software should be ready and deliverable during August 1998.

References

[Ake78] S.B.Akers, ”Binary Decision Diagrams,” *IEEE Trans. on Computers*, Vol. 27, pp. 509-516, 1978

[Bac59] J. Backus, ”The syntax and the semantics of the proposed international algebraic language,” *Proceedings of the ACM-GAMM Conference.* Paris, France: Information Processing, 1959.

[BH90] D. Bhattacharya and J.P. Hayes, ”A Hierarchical Test Generation Methodology for Digital Circuits,” *JETTA: Theory and Application*, Vol. 1, 1990, pp. 103-123

[BJM+97a] M.Brik, G.Jervan, A.Markus, J.Raik, R.Ubar. ”A Hierarchical Automatic Test Pattern Generator Based on Using Alternative Graphs,” *Proc. of the 4-th International Workshop on Computer Aided Design of Modern Devices and ICs*. pp. 415-420, Poznan, Poland, June 12-14, 1997.

[BJM+97b] M.Brik, G.Jervan, A.Markus, P.Paomets, J.Raik, R.Ubar. ”Mixed-Level Test Generator for Digital Systems,” *Proceedings of the Estonian Acad. of Sci. Engng*, 1997, Vol. 3 , No 4, pp. 269-280.

[BJM+98] M.Brik, G.Jervan, A.Markus, P.Paomets, J.Raik, R.Ubar. ”Hierarchical Test Generation for Digital Systems. Mixed Design of Integrated Circuits and Systems,” Kluwer Academic Publishers, pp. 131-136, 1998.

[CA94] C. Cho, J. Armstrong ”B-algorithm: A Behavioral Test Generation Algorithm.” *IEEE International Test Conference*, 1997, pp.968-979.

[EKP97] P.Eles, K.Kuchcinski, Z.Peng, ”System Synthesis with VHDL,” Kluwer Academic Publishers, 1997

[EKPM92] P.Eles, K.Kuchcinski, Z.Peng, M.Minea, ”Compiling VHDL into a High-Level Synthesis Design Representation,” *Proc. EURO-DAC/EURO-VHDL’92*, 1992, pp. 604-609

[Fje92] B.Fjelleborg, ”Pipeline extraction for VLSI data path synthesis,” *Ph.D. dissertation*, Dept. of Computer and Information Science, Linköping University, 1992.

[Fuj85] H. Fujiwara, ”Logic Testing and Design for Testability,” *MIT Press Series in Computer Systems*, MIT Press, Cambridge, Massachusetts; London, England, 1985.

[GA85] A.G. Gupta and J.R. Armstrong, ”Functional Fault Modelling and Simulation for VLSI Devices,” *Proc. ACM/IEEE 22nd Design Automation Conference*, IEEE Computer Society Press, Los

Alamitos, Calif., 1985, pp. 720-726.

[Gho88] S. Ghosh ”Behavioral Level Fault Simulation.” *IEEE Design & Test of Computers*, pp.31-42, 1988.

[GMM89] W. Geiselhardt, W. Mohrs and U. Moeller, ”FUNTEST - Functional Test Generation for VLSI-Circuit and Systems,” *Microelectronics Reliability*, Vol. 29, No. 3, Mar. 1989, pp. 357-364.

[IEEE87] ”IEEE Standard VHDL Language Reference Manual,” IEEE Std 1076-1987, March 31, 1988

[IEEE93] ”IEEE Standard VHDL Language Reference Manual,” ANSI/IEEE Std 1076-1993, (Revision of IEEE Std 1076-1987), June 6, 1994

[JKRU96] G.Jervan, H.Krupnova, J.Raik, R.Ubar. ”A Constraint-Driven Gate-Level Test Generator,” *Proc. of the 5-th Baltic Electronics Conference*. pp. 237-240, Tallinn, Estonia, Oct. 1996.

[JMRU96] G.Jervan, A.Markus, J.Raik, R.Ubar. ”Fault Model and Test Synthesis for RISC Processors,” *Proc. of the 5-th Baltic Electronics Conference*. pp. 229-232, Tallinn, Estonia, Oct. 1996.

[JMRU97a] G.Jervan, A.Markus, J.Raik, R.Ubar. ”Automatic Test Generation System for VLSI,” *Proc. of the 1-st Electronic Circuits and Systems Conference*. pp. 255-258, Bratislava, Slovakia, Sep. 4-5, 1997.

[JMRU97b] G.Jervan, A.Markus, J.Raik, R.Ubar. ”Assembling Low-Level Tests to High-Level Symbolic Test Frames,” *Proc. of the 15th NORCHIP Conference* pp. 275-281, Tallinn, Estonia, Nov. 10-11, 1997.

[JMRU98a] G.Jervan, A.Markus, J.Raik, R.Ubar, ”Hierarchical Test Generation with Multi-Level Decision Diagram Models,” *Proc. of the 7-th IEEE North Atlantic Test Workshop*, West Greenwich, RI, USA, May 28-29, 1998. (to be published)

[JMRU98b] G.Jervan, A.Markus, J.Raik, R.Ubar. ”VHDL based Test Generation System,” *Proc. of the 5th Int. Conf. on Electronic Devices and Systems*, Brno, Czech Republic, June 11-12, 1998. (to be published)

[Lev92] M.E. Levitt, ”ASIC Testing Upgraded”, *IEEE Spectrum*, Vol. 29, No. 5, 1992

[LP92] J. Lee and J.H. Patel, ”Hierarchical Test Generation Under Intensive Global Functional Constraints,” *Proc. 29th ACM/IEE Design Automation Conference*, IEEE Computer Society Press, 1992, pp. 261-266

[LS85] T. Lin, S.Y.H. Su ”VLSI functional test pattern generation - a design and implementation.” *IEEE International Test conference*, 1985, pp.922-929.

[LSU89] R.Lipsett, C.Schaefer, C.Usset, ”VHDL: Hardware Description and Design,” Kluwer Academic Publishers, 1989

[MG96] W. Mao, R. Gulati, ”Improving gate level fault coverage by RTL fault grading.” *IEEE International Test Conference*, 1996, pp.150-159.

[Min93] M.Minea, ”A VHDL compiler for a high-level synthesis system,” Research Report LiTH-IDA-R-93-23, CADLAB, June 1993.

[MLD92] P.Michel, U.Lauther, P.Duzy, ”The Synthesis Approach To Digital System Design,” Kluwer Academic Publishers, 1992

[NP91] T. M. Niermann, J. H. Patel. ”HITEC: A test generation package for sequential circuits,” *Proc. of the European Conf. Design Automation (EDAC)*, pp.214-218, 1991.

[PKL89] Z.Peng, K.Kuchcinski, B.Lyles, ”CAMAD: A Unified Data Path/Controller Synthesis Environment,” in D:A. Edwards (Editor), *Design Methodologies for VLSI and Computer Architecture*, North-Holland, 1989, pp.53-67

[Rai97] J. Raik, ”Hierarchical Test Generation Based on Alternative Graph Models,” Master Thesis, Tallinn Technical University, 1997

[RPGN94] E. M. Rudnick, J. H. Patel, G. S. Greenstein, T. M. Niermann. ”Sequential circuit test generation in a genetic algorithm framework,” *Proc. of the Design Automation Conf.*, pp. 698-704, 1994.

[RVE+98] E. M. Rudnick, R. Vietti, A. Ellis, F. Corno, P.Prinetto, M. Sonza Reorda. ”Fast sequential circuit test generation using high-level and gate-level techniques,” *Proc. of the DATE Conf.*, 1998.

[Sch92] J.M. Schoen, editor. ”Performance and Fault Modeling with VHDL.” Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[Syn96] Design Compiler Reference Manual, Version 3.5, Synopsys Inc., 1996.

[TA80] S.M. Thatte and I.A. Abraham, ”Test Generation for Micro-processors,” *IEEE Trans. on Computers*, Vol. 29, 1980, pp. 429-441.

[Uba76] R. Ubar, ”Test Generation for Digital Circuits Using Alternative Graphs,” *Proc. of Tallinn Technical University*, Estonia, No. 409, pp. 75-81 (in Russian), 1976

[Uba83] R.Ubar, “Test Pattern Generation for Digital Systems on the Vector AG-model,” *13-th International Symposium on Fault Tolerant Computing*, Milano, Italy, pp. 347-351, 1983.

[Uba88] R.Ubar, “Alternative Graphs and Technical Diagnosis of Digital Devices,” *Electronic technique*, (USSR) Vol.8, No.5 (132), pp.33-57 (in Russian), 1988.

[Uba96] R. Ubar, ”Test Synthesis with Alternative Graphs,” *IEEE Design and Test of Computers*, Vol. 13, No. 1, pp. 48-57, Spring 1996

Appendixes

Appendix A: Package ETPN_modelling

```
package ETPN_modelling is
  -- Declarations for modelling ETPN designs

  -- signal types
  subtype Unsigned1 is bit;
  subtype Unsigned8 is integer range -120  to 127;
  subtype Unsigned16 is integer; -- range 0 to 65535;
  subtype Unsigned32 is integer range -2147483647 to 2147483647;
  subtype Unsigned64 is bit_vector ( 63 downto 0 ) ;

  attribute LayoutSize : positive;
end ETPN_modelling;
```

Appendix B: Library for describing design entities

Note. Only elements with bit-width 4 are presented. Original library contains elements with different bitwidths.

```
;;
;;                                        Module Library
;;                                        ==============
;;                              (Version 3.0, 91-03-0)
;;
;; This version has the following new feature:
;; 1. Each module has a unique identifier.
;; 2. All possible operations of a module are given explicitly as a set of
;;    primitive operation identifiers.
;; 3. Different ports of a module can have different bit-widths.
;;
;; The units used for the different attributes:
;; 1. Cost : 1 000 * lamda square ( = height * bit_width(about 38) )
;; 2. Time : ns
;;
;;      Id   (Op_id Time)* Bits Cost Input_P  Output_P         ;; Comments
;;

;; 4 bit modules

(Module Add4 (OP (Add 150) (Sub 150) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1)
(Op2))) ;;+, -
(Module Sub4 (OP (Sub 150) (Add 150) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1)
(Op2))) ;;-, +
(Module Exp4 (OP (Exp 150) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1) (Op2))) ;;
exponentiation
(Module Mlt4 (OP (Mlt 150) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1 Op2))) ;;
multiplikation
(Module Div4 (OP (Div 100) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1 Op2))) ;;
divide
(Module And4 (OP (And 30) Comp) 4 0 (IP (Ip1) (IP2)) (OP (Op1))) ;; mask,
Op1:=Ip1 And Ip2
(Module Nan4 (OP (Nan 10) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1))) ;; nand
(Module Or4 (OP (Or 40) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1))) ;; or
(Module Nor4 (OP (Nor 10) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1))) ;; nor
(Module Xor4 (OP (Xor 30) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1))) ;; xor
(Module Inv4 (OP (Not 15) Comp) 4 0 (IP (Ip1)) (OP (Op1)));; not
(Module Mod4 (OP (Mod 150) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1) (Op2))) ;;
modulus
(Module Rem4 (OP (Rem 150) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1) (Op2))) ;;
remainder
(Module Eql4 (OP (Eql 30) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1 1))) ;; compare,
=
(Module Les4 (OP (Les 30) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1 1))) ;; compare,
<
(Module Grt4 (OP (Grt 30) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1 1))) ;; compare,
>
(Module LeE4 (OP (LeE 30) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1 1))) ;; compare,
<=
(Module GrE4 (OP (GrE 30) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1 1))) ;; compare,
>=
(Module NEq4 (OP (NEq 30) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1 1))) ;; compare,
<>

(Module Reg4 (OP (Reg 45)) 4 0 (IP (Ip1)) (OP (Op1)))          ;; register
(Module Alu4 (OP (Alu 120) (Add 150) (Sub 150) (Inc 160) Comp) 4 0
        (IP (Ip1) (Ip2) (Ip3 3)) (OP (Op1) (Op2 1))) ;; see notr (4)
(Module Con4 (OP (Con 0)) 4 0 (IP) (OP (Op1))) ;; constant
(Module Com4 (OP (Com 70) Comp) 4 0 (IP (Ip1) (Ip2)) (OP (Op1 1))) ;; compare,
< or =
(Module Cdt4 (OP (Cdt 0.01)) 4 0 (IP (Ip1)) (OP (Op1 1))) ;; condition signal
(Module Pad4 (OP (Pad 40)) 4 0 (IP (Ip1)) (OP (Op1))) ;; I/O pads
(Module Mul4 (OP (Mul 40) Meta) 4 0 0 (IP (Ipi)) (OP (Op1)))
  ;; multiplex i input 1 output  cost = (lambda (i) (* 1.67 (+ i 1)))
(Module Bus4 (OP (Bus 60) Meta) 4 0 0 (IP (Ipi)) (OP (Opj)))
  ;; bus i input j output cost = (lambda (i j) (* 1.67 (+ i j)))
(Module Mem4 (OP (Mem 200) Meta) 4 0 0 (IP (Ip1) (Ip2)) (OP (Op1)))
  ;; memory with i cells cost = (lambda (i) (+ 2.0 (* i 2.33)))
(Module Inc4 (OP (Inc 160) Comp) 4 0 (IP (Ip1)) (OP (Op1))) ;; inceament,
Op1:=Ip1+1
(Module Shl4 (OP (Shl 15) Comp) 4 0 (IP (Ip1)) (OP (Op1))) ;; shift left
(Module Shr4 (OP (Shr 15) Comp) 4 0 (IP (Ip1)) (OP (Op1))) ;; shift right
```

Appendix C: Design example (RT-Level VHDL)

```
-- The VHDL code generated by CAMAD:

LIBRARY MGC_PORTABLE;
USE MGC_PORTABLE.QSIM_LOGIC.ALL;

USE Work.ETPN_modelling.ALL;

ENTITY /../MAGISTER/designs/mult8x8 IS
  PORT (
    P_N1, P_N4, P_N5 : IN BIT; -- input pads
    P_N6 : OUT UNSIGNED16; -- output pads
    Reset : IN BIT := '0'; -- reset signal
    Clock : IN BIT := '0' -- clock signal
  );
END;

ARCHITECTURE Structure OF /../MAGISTER/designs/mult8x8 IS
  -- signals for component connection points
  SIGNAL N_A5 : BIT;
  SIGNAL N_A6 : UNSIGNED8;
  SIGNAL N_A7 : BIT;
  SIGNAL N_A8 : UNSIGNED8;
  SIGNAL N_A9 : UNSIGNED8;
  SIGNAL N_A10 : UNSIGNED8;
  SIGNAL N_A11 : UNSIGNED8;
  SIGNAL N_A12 : BIT;
  SIGNAL N_A13 : UNSIGNED32;
  SIGNAL N_A14 : UNSIGNED3;
  SIGNAL N_A15 : BIT;
  SIGNAL N_A17 : UNSIGNED32;
  SIGNAL N_A18 : UNSIGNED2;
  SIGNAL N_A19 : UNSIGNED32;
  SIGNAL N_A20 : BIT;
  SIGNAL N_A21 : BIT;
  SIGNAL N_A23 : UNSIGNED32;
  SIGNAL N_A24 : UNSIGNED32;
  SIGNAL N_A25 : UNSIGNED32;
  SIGNAL N_A27 : UNSIGNED2;
  SIGNAL N_A28 : UNSIGNED32;
  SIGNAL N_A30 : UNSIGNED2;
  SIGNAL N_A31 : UNSIGNED32;
  SIGNAL N_A33 : UNSIGNED32;
  SIGNAL N_A34 : UNSIGNED32;
  SIGNAL N_A36 : UNSIGNED2;
  SIGNAL N_A37 : UNSIGNED32;
  SIGNAL N_A38 : BIT;
  SIGNAL N_A39 : BIT;
  SIGNAL N_A43 : UNSIGNED32;
  SIGNAL N_A45 : BIT;
  SIGNAL N_A46 : BIT;
  SIGNAL N_A47 : BIT;
  SIGNAL N_A49 : UNSIGNED32;
  SIGNAL N_A50 : UNSIGNED32;
  SIGNAL N_A51 : UNSIGNED32;
  SIGNAL Unconnected : UNSIGNED32;

  -- control signals from the control Petri net
  SIGNAL P0, P1, P2, P3, P4, P5, P6, P7, P8, P9,
    P10, P11, P12, P13, P14, P15, P16, P17, P18, P19,
    P20, P21, P3_P9_P12_P16, P4_P11, P6_P13 : BIT;

  -- condition signals from the data path
  SIGNAL C3, C4, C5, C6 : BIT;

  -- data path component types
  COMPONENT Pad1    PORT (
      Ip1 : IN BIT; -- input
      Op1 : OUT BIT -- output
    );
  END COMPONENT;

  COMPONENT Pad8    PORT (
      Ip1 : IN UNSIGNED8; -- input
      Op1 : OUT UNSIGNED8 -- output
    );
  END COMPONENT;

  COMPONENT Pad16    PORT (
      Ip1 : IN UNSIGNED16; -- input
      Op1 : OUT UNSIGNED16 -- output
    );
  END COMPONENT;

  COMPONENT Reg32    PORT (
      Ip1 : IN UNSIGNED32; -- input
      Ic1 : IN BIT; -- load control
      clock : IN BIT; -- load control
      Op1 : OUT UNSIGNED32 -- output
    );
  END COMPONENT;

  COMPONENT Add8    PORT (
      Ip1, Ip2 : IN UNSIGNED8; -- input
      Op1 : OUT UNSIGNED8 -- output
    );
  END COMPONENT;

  COMPONENT Mlt8    PORT (
      Ip1, Ip2 : IN UNSIGNED8; -- input
      Op1 : OUT UNSIGNED8 -- output
    );
  END COMPONENT;

  COMPONENT LeE32    PORT (
      Ip1, Ip2 : IN UNSIGNED32; -- input
      Op1 : OUT BIT -- output
    );
  END COMPONENT;

  COMPONENT Eql32    PORT (
      Ip1, Ip2 : IN UNSIGNED32; -- input
      Op1 : OUT BIT -- output
    );
  END COMPONENT;

  COMPONENT Mod32    PORT (
      Ip1, Ip2 : IN UNSIGNED32; -- input
      Op1 : OUT UNSIGNED32 -- output
    );
  END COMPONENT;

  COMPONENT Add32    PORT (
      Ip1, Ip2 : IN UNSIGNED32; -- input
      Op1 : OUT UNSIGNED32 -- output
    );
  END COMPONENT;

  COMPONENT Div32    PORT (
      Ip1, Ip2 : IN UNSIGNED32; -- input
      Op1 : OUT UNSIGNED32 -- output
    );
  END COMPONENT;

  COMPONENT Sub32    PORT (
      Ip1, Ip2 : IN UNSIGNED32; -- input
      Op1 : OUT UNSIGNED32 -- output
    );
  END COMPONENT;

  COMPONENT Eql1    PORT (
      Ip1, Ip2 : IN BIT; -- input
      Op1 : OUT BIT -- output
    );
  END COMPONENT;

  COMPONENT Mul32_4    PORT (
      Ip1, Ip2, Ip3, Ip4 : IN UNSIGNED32; -- input
      Ic1, Ic2, Ic3, Ic4 : IN BIT; -- multiplex selection
      Op1 : OUT UNSIGNED32 -- output
    );
  END COMPONENT;

  COMPONENT Mul32_2    PORT (
      Ip1, Ip2 : IN UNSIGNED32; -- input
      Ic1, Ic2 : IN BIT; -- multiplex selection
      Op1 : OUT UNSIGNED32 -- output
    );
  END COMPONENT;

  -- data path component configuration
  FOR N1, N4, N5, N6 : Pad1 USE ENTITY Work.Pad1(behavior);
  FOR N1, N4, N5, N6 : Pad8 USE ENTITY Work.Pad8(behavior);
  FOR N1, N4, N5, N6 : Pad16 USE ENTITY Work.Pad16(behavior);
  FOR N7, N8, N9, N19 : Reg32 USE ENTITY Work.Reg32(behavior);
  FOR N15 : Add8 USE ENTITY Work.Add8(behavior);
  FOR N17 : Mlt8 USE ENTITY Work.Mlt8(behavior);
  FOR N21 : LeE32 USE ENTITY Work.LeE32(behavior);
  FOR N25, N38 : Eql32 USE ENTITY Work.Eql32(behavior);
  FOR N26, N39 : Mod32 USE ENTITY Work.Mod32(behavior);
  FOR N31, N36 : Add32 USE ENTITY Work.Add32(behavior);
  FOR N32, N34 : Div32 USE ENTITY Work.Div32(behavior);
  FOR N44 : Sub32 USE ENTITY Work.Sub32(behavior);
  FOR N45 : Eql1 USE ENTITY Work.Eql1(behavior);
  FOR N49 : Mul32_4 USE ENTITY Work.Mul32_4(behavior);
  FOR N50, N51 : Mul32_2 USE ENTITY Work.Mul32_2(behavior);

  -- FSM state declaration
  TYPE State_type IS (S1, S2, S3, S4, S5, S6, S7, S8, S9, S10,
    S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21);
  SIGNAL present_state : State_type;
  SIGNAL next_state : State_type;

  BEGIN
    -- component instantiation
    N1 : Pad1 PORT MAP (P_N1, N_A45);
    N4 : Pad8 PORT MAP (P_N4, N_A6);
    N5 : Pad8 PORT MAP (P_N5, N_A9);
    N6 : Pad16 PORT MAP (N_A23, P_N6);
    N7 : Reg32 PORT MAP (N_A49, P3_P9_P12_P16, clock, N_A23);
    N8 : Reg32 PORT MAP (N_A50, P4_P11, clock, N_A17);
    N9 : Reg32 PORT MAP (N_A11, P5, clock, N_A24);
    N15 : Add8 PORT MAP (N_A6, N_A7, N_A8);
    N17 : Mlt8 PORT MAP (N_A9, N_A10, N_A11);
    N19 : Reg32 PORT MAP (N_A51, P6_P13, clock, N_A13);
    N21 : LeE32 PORT MAP (N_A13, N_A14, C3);
    N25 : Eql32 PORT MAP (N_A19, N_A20, C4);
    N26 : Mod32 PORT MAP (N_A17, N_A18, N_A19);
    N31 : Add32 PORT MAP (N_A23, N_A24, N_A25);
    N32 : Div32 PORT MAP (N_A17, N_A27, N_A28);
    N34 : Div32 PORT MAP (N_A23, N_A30, N_A31);
    N36 : Add32 PORT MAP (N_A13, N_A33, N_A34);
    N38 : Eql32 PORT MAP (N_A37, N_A38, C5);
    N39 : Mod32 PORT MAP (N_A17, N_A36, N_A37);
    N44 : Sub32 PORT MAP (N_A23, N_A24, N_A43);
    N45 : Eql1 PORT MAP (N_A45, N_A46, C6);
    N49 : Mul32_4 PORT MAP (N_A5, N_A25, N_A31, N_A43, P3,
                            P9, P12, P16, N_A49);
    N50 : Mul32_2 PORT MAP (N_A8, N_A28, P4, P11, N_A50);
    N51 : Mul32_2 PORT MAP (N_A12, N_A34, P6, P13, N_A51);

    -- The FSM controller
    state_register:PROCESS( reset, clock )
    BEGIN
      IF ( reset = '0' ) THEN present_state <= S1;
      ELSIF ( clock = '1' AND clock'LAST_VALUE = '0'
              AND clock'EVENT ) THEN
          present_state <= next_state;
      END IF;
    END PROCESS state_register;

    output_decode_logic:PROCESS( clock, present_state )
    BEGIN
      P0 <= '0';
      P1 <= '0';
      P2 <= '0';
      P3 <= '0';
      P4 <= '0';
      P5 <= '0';
      P6 <= '0';
      P7 <= '0';
      P8 <= '0';
      P9 <= '0';
      P10 <= '0';
      P11 <= '0';
      P12 <= '0';
      P13 <= '0';
      P14 <= '0';
      P15 <= '0';
      P16 <= '0';
      P17 <= '0';
      P18 <= '0';
      P19 <= '0';
      P20 <= '0';
      P21 <= '0';
      CASE present_state IS
        WHEN S1 =>
          P0 <= '1';
        WHEN S2 =>
          P2 <= '1';
        WHEN S3 =>
          P3 <= '1';
        WHEN S4 =>
          P4 <= '1';
        WHEN S5 =>
          P5 <= '1';
        WHEN S6 =>
          P6 <= '1';
        WHEN S7 =>
          P7 <= '1';
        WHEN S8 =>
          P8 <= '1';
        WHEN S9 =>
          P14 <= '1';
        WHEN S10 =>
          P9 <= '1';
        WHEN S11 =>
          P10 <= '1';
        WHEN S12 =>
          P15 <= '1';
        WHEN S13 =>
          P11 <= '1';
        WHEN S14 =>
          P16 <= '1';
        WHEN S15 =>
          P17 <= '1';
        WHEN S16 =>
          P12 <= '1';
        WHEN S17 =>
          P18 <= '1';
        WHEN S18 =>
          P13 <= '1';
        WHEN S19 =>
          P19 <= '1';
        WHEN S20 =>
          P20 <= '1';
        WHEN S21 =>
          P21 <= '1';
      END CASE;
    END PROCESS output_decode_logic;

    state_decode_logic : PROCESS( C3, C4, C5, C6,
                                  present_state )
    BEGIN
      CASE present_state IS
        WHEN S1 =>
          next_state <= S2;
        WHEN S2 =>
          next_state <= S3;
        WHEN S3 =>
          next_state <= S4;
        WHEN S4 =>
          next_state <= S5;
        WHEN S5 =>
          next_state <= S6;
        WHEN S6 =>
          next_state <= S7;
        WHEN S7 =>
          IF C3 = '1' THEN
            next_state <= S8;
          ELSE
            next_state <= S9;
          END IF;
        WHEN S8 =>
          IF C4 = '1' THEN
            next_state <= S10;
          ELSE
            next_state <= S11;
          END IF;
        WHEN S9 =>
          next_state <= S12;
        WHEN S10 =>
          next_state <= S11;
        WHEN S11 =>
          next_state <= S13;
        WHEN S12 =>
          IF C5 = '1' THEN
            next_state <= S14;
```

```
          ELSE
            next_state <= S15;
          END IF;
        WHEN S13 =>
          next_state <= S16;
        WHEN S14 =>
          next_state <= S15;
        WHEN S15 =>
          next_state <= S17;
        WHEN S16 =>
          next_state <= S18;
        WHEN S17 =>
          next_state <= S19;
        WHEN S18 =>
          next_state <= S7;
        WHEN S19 =>
          next_state <= S20;
        WHEN S20 =>
          next_state <= S21;
        WHEN S21 =>
          IF C6 = '1' THEN
            next_state <= S2;
          ELSE
            next_state <= S19;
          END IF;
        WHEN OTHERS =>
          next_state <= S1;
      END CASE;
    END PROCESS state_decode_logic;

    -- Other control signals
    P3_P9_P12_P16 <= P3 OR P9 OR P12 OR P16;
    P4_P11 <= P4 OR P11;
    P6_P13 <= P6 OR P13;

    -- Constants
    N_A5 <= 0;
    N_A7 <= 0;
    N_A10 <= 128;
    N_A12 <= 0;
    N_A14 <= 6;
    N_A18 <= 2;
    N_A20 <= 1;
    N_A27 <= 2;
    N_A30 <= 2;
    N_A33 <= 1;
    N_A36 <= 2;
    N_A38 <= 1;
    N_A46 <= 1;
END;
```

Appendix D: DD Model Format

DD model format is case sensitive. It is a line-based format where maximum line length can be 256 characters. In the following the BNF syntax of DD model format is presented. The meta-syntax used obeys the following rules:

I. Syntactic categories (nonterminals) are printed in *italics*; literal words, characters and constants are enclosed to ‘quotes’.

II. If a construct is enclosed to [square brackets], it is optional.

III. If a construct is enclosed to {curly brackets}, it may be repeated zero or more times.

IV. A choice is indicated with a vertical bar |.

V. If a construct is enclosed in <chevrons>, it can occur at most once.

dd_model :=

statistics

mode

[control_signals]

dd_description

statistics :=

‘STAT#’ *natural* ‘Nods,’ *natural* ‘Vars,’ *natural* ‘Grps,’ *natural* ‘Inps,’ *natural* ‘Outs,’ *natural* ‘Cons’ *[*‘,’ *natural* ’Funs’*] [*‘,’ *natural* ’Mems’*] [*‘,’ *natural* ‘C_outs’*]*

The *natural* values reflect the number of nodes, variables, graphs, inputs, outputs, constants, functions, memory arrays and control part outputs, respectively. The number of functions and memory arrays are meaningful in the high-level descriptions. The number of control part outputs is used with the RTL descriptions divided into a control part and a datapath only.

control_signals :=

‘COUT#’ *natural {*‘,’ *natural}*

Shows the variable indexes of control signal variables. Used in RTL descriptions partitioned to datapath and control parts.

mode :=

‘MODE#’ ‘STRUCTURAL’ | ‘RTL’ | ‘BEHAVIORAL’

Indicates whether a structural gate-level model, a RTL model, or a behavioral model is being described.

dd_description :=

[{input_definition}]

[{memory_definition}]

[{constant_definition}]

[{function_definition}]

[{control_definition}]

{graph_variable_definition}

The definitions are ranged according to the order shown above. There are no memory definitions or function definitions in SSBDD models. *control_definitions* are used only in the RTL descriptions partitioned into control and datapath parts.

input_definition :=

‘VAR#’ *var_index* ‘:’ ‘(‘ *variable_flags* ’)’ *var_name var_range*

Defines a primary input of the model.

memory_definition :=

‘VAR#’ *var_index* ‘:’ ‘(‘*variable_flags*’)’ *var_name var_range [row_range] column_range*

memory_row

{memory_row}

Defines a memory array. The optional *row_range* is used with two-dimensional arrays, and it determines the range of row addresses used in memory. In one-dimensional arrays, *row_range* is omitted. In similar way, *column_range* determines the range of column addresses used in the memory variable.

memory_row :=

‘{‘ *integer {‘,’ integer}* ‘}’

Defines the contents of a memory variable. The number of integers in *memory_row* is determined by *column_range.*

row_range := mem_range

Row_range is used with two-dimensional arrays, and it determines the range of row addresses used in memory. In one-dimensional arrays, *row_range* is omitted.

column_range := mem_range

Determines the range of column addresses used in the memory variable.

mem_range := '[' *integer* '-' *integer* ']'

In *mem_range* the first integer must be less than the second one.

constant_definition :=

'VAR#' *var_index* ':' '('*variable_flags*')' *var_name* *var_range* 'VAL' '=' *integer*

Defines a constant. The integer value shows the value of the constant.

function_definition :=

'VAR#' *var_index* ':' '('*variable_flags*')' *var_name* *var_range*

'FUN#' *function_type* *arguments_definition*

Defines an operation or function.

function_type := identifier

Shows the type of the operation.

arguments_definition :=

'(' *[argument] {',' argument}* ')'

Defines the arguments (if any) of an operation.

argument :=

'A'*argument_index* '<=' *argument_variable* *range*

The *range* shows the bit-slice of the variable *argument_variable* that is used as a function argument.

argument_index := natural

Shows the index of the function argument.

argument_variable := natural

Shows the index of the variable used as the function argument.

control_definition :=

'VAR#' *var_index* ':' '(' *variable_flags* ')' *var_name var_range*

Defines a control signal. Used to define control part output signals of the RTL designs partitioned into datapath and control parts.

graph_variable_definition :=

'VAR#' *var_index* ':' '('*variable_flags*')' *var_name var_range*

graph_definition

Defines a variable for which a graph corresponds.

graph_definition :=

'GRP#' *graph_index* ':' 'BEG' '=' *natural* ',' 'LEN' '=' *natural* '-----'

node_definition | parallel_node_definition

{node_definition | parallel_node_definition}

Defines a graph in the DD model. The 'BEG=' construct shows the <u>absolute</u> index of the first node in the graph. The 'LEN=' construct in turn shows the number of nodes in the graph.

node_definition :=

nod_abs_index nod_index ': ('*nod_flags*') (' *successors* ') V =' *nod_var nod_name nod_range*

Defines an DD node. *nod_abs_index* and *nod_index* represent the <u>absolute</u> (inside the model) and <u>relative</u> (inside the graph) indexes of the node. Construct *successors* shows the successor nodes of current node which are chosen with different node values. Index of the variable labeling the node is determined with *nod_var*.

parallel_node_definition :=

nod_abs_index nod_index ‘: (v___)’ ‘(‘ ‘0’ ‘0’ ‘)’ ‘VEC =’ *nod_var_vector*

Defines a terminal node of the FSM graph of RTL description. *nod_abs_index* and *nod_index* represent the absolute (inside the model) and relative (inside the graph) indexes of the node, respectively. Indexes of the variables labeling the node are determined with *nod_var_vector.*

nod_var_vector :=

‘ ”’ state_value {signal_value} ’ ”’

state_value shows the value of the next state. The *signal_value* constructs show the values of the control signals defined in the *control_signals* construct.

state_value := natural

Shows the value of the next state.

signal_value := ‘0’ | ‘1’ | ‘X’

The *signal_value* constructs show the values of the control signals defined in the *control_signals* construct.

nod_var := natural[[‘[‘ ‘V’ ‘=’ *row_index* ’]’ *]* ‘[‘ ‘V’ ‘=’ *column_index* ’]’ *]*

Shows the index of the variable labeling the node. Optional constructs *row_index* and *column_index* are used with memory variables labeling the node. These constructs determine the indexes of the variables used for addressing rows and columns, respectively.

nod_name := string

Shows the name of the node.

nod_range := range

nod_range determines the bit-slice of the variable that labels the node. DD model format allows slices of variables to be used for labeling a node.

row_index := natural

Determine the indexes of the variables used for addressing rows of the memory variable.

column_index := natural

Determines the index of the variable used for addressing columns of the memory variable.

nod_abs_index := natural

Shows the absolute (inside the model) index of the node.

nod_index := natural

Shows the relative (inside the graph) index of the node inside the graph.

graph_index := natural

Shows the index of the graph.

variable_flags :=

< ‘i’ | ‘m’ | ‘c’ | ‘f’ | ‘o’ | ‘n’ | ’_’ | ‘F’ > *{*<‘d’> | ‘_’*}*

The variable flags have the following interpretation:

‘i’	- input variable
‘m’	- memory variable (RTL, behavioral)
‘c’	- constant variable
‘f’	- function variable (RTL, behavioral)
‘o’	- output variable
‘d’	- clock cycle delay, e.g. in registers, flipflops. (Gate-level, RTL)

The following flags are used with RTL descriptions only:

‘n’	- control part output signal
‘F’	- FSM graph variable
‘r’	- reset variable
‘s’	- state variable

nod_flags :=

< ‘i’ | ’_’ > *{* ‘n’ | ‘v’ | ‘_’*}*

The node flags have the following interpretation:

‘i’	- inverted node (in gate-level descriptions only)
‘n’	- non-terminal node (RTL, behavioral)
‘v’	- control part terminal node (RTL)

successors :=

nonterminal_successors | *terminal_successor* | *boolean_successors*

Construct *successors* shows the successor nodes of current node which are chosen with different node values.

nonterminal_successors :=

node_values ‘=>’ *successor_index {node_values* ‘=>’ *successor_index }*

This construct shows the indexes of successor nodes which will be selected with corresponding node values. (Used with RTL and behavioral models only).

terminal_successors := ‘0*’* ‘0*’*

Terminal nodes are nodes which have no successor nodes.

boolean_successors:=

natural natural

This type of construct can be used with Boolean DDs only. The first natural number indicates the relative index of the successor node when the value of current node is ‘0’, and the second number shows the relative index of the successor node when current node is ‘1’, respectively. If the index of the successor node is ‘0’, it shows that there is no successor nodes to current node with corresponding value.

node_values := natural { ‘,’ | ‘–’ *natural}*

Determines the set of node values that activate the corresponding branch. The comma ‘,’ character is used for separating the indexes; the minus sign ‘-‘ is used for index ranges, e.g. ‘3-5’, which can be alternatively written as ‘3,4,5’.

successor_index :=

natural | ‘X’

If *successor_index* is a natural number, it shows the index of the successor node. Otherwise, if *successor_index* is ‘X’, it means that the successor is undetermined.

var_index := natural

Shows the index of the variable.

var_name := string

Shows the name of the variable.

var_range := range

Shows the bitwidth of the variable.

range := [‘<’ natural ‘:’ natural ‘>’]

Range is a construct for describing bit-vectors. The first natural shows the index of the most significant bit and the latter is for the least significant bit, respectively. If range is omitted, it will default to ‘<0:0>’.

string :=

‘ ” ’ {character} ‘ ” ’

Character can be any character, except newline and double quote ‘”’.

integer :=

[‘-’]natural

Any integer number.

natural

Natural can be any non-negative number.

identifier :=

alphabetic_character{alphabetic_character | digit | ‘_’}

alphabetic_character := ‘A’|…| ’Z’ | ‘a’ |…| ‘z’

digit := ‘0’ | ‘1’ |…| ‘9’

Appendix E: DD-Model example

```
;23/05/98

STAT#           70 Nods, 62 Vars, 5 Grps, 4 Inps, 1 Outs, 14 Const, 13 Funs, 23 C_outs
COUT#          31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53

MODE#          RTL

VAR#           0:              (i_________)  "P_N1"
VAR#           1:              (i_________)  "P_N4"
VAR#           2:              (i_________)  "P_N5"
VAR#           3:              (i_________)  "Reset"

VAR#           4:              (__c_______)   "N_A5"                          VAL = 0
VAR#           5:              (__c_______)   "N_A7"                          VAL = 0
VAR#           6:              (__c_______)   "N_A10"          <7:0>          VAL = 128
VAR#           7:              (__c_______)   "N_A12"                         VAL = 0
VAR#           8:              (__c_______)   "N_A14"          <2:0>          VAL = 6
VAR#           9:              (__c_______)   "N_A18"          <1:0>          VAL = 2
VAR#           10:             (__c_______)   "N_A20"                         VAL = 1
VAR#           11:             (__c_______)   "N_A27"          <1:0>          VAL = 2
VAR#           12:             (__c_______)   "N_A30"          <1:0>          VAL = 2
VAR#           13:             (__c_______)   "N_A33"                         VAL = 1
VAR#           14:             (__c_______)   "N_A36"          <1:0>          VAL = 2
VAR#           15:             (__c_______)   "N_A38"                         VAL = 1
VAR#           16:             (__c_______)   "N_A46"                         VAL = 1
VAR#           17:             (__c_______)   "MUX_CONST"                     VAL = 0

VAR#           18:             (____f_____)  "N_A8"            <7:0>
FUN   Add      (A1<=1,         A2<=5)

VAR#           19:             (____f_____)  "N_A11"           <7:0>
FUN   Mlt      (A1<=2,         A2<=6)

VAR#           20:             (____f_____)  "C3"<31:0>
FUN   LeE      (A1<=61,        A2<=8)

VAR#           21:             (____f_____)  "C4"<31:0>
FUN   Eql      (A1<=22,        A2<=10)

VAR#           22:             (____f_____)  "N_A19"           <31:0>
FUN   Mod      (A1<=59,        A2<=9)

VAR#           23:             (____f_____)  "N_A25"           <31:0>
FUN   Add      (A1<=58,        A2<=60)

VAR#           24:             (____f_____)  "N_A28"           <31:0>
FUN   Div      (A1<=59,        A2<=11)

VAR#           25:             (____f_____)  "N_A31"           <31:0>
FUN   Div      (A1<=58,        A2<=12)

VAR#           26:             (____f_____)  "N_A34"           <31:0>
FUN   Add      (A1<=61,        A2<=13)

VAR#           27:             (____f_____)  "C5"<31:0>
FUN   Eql      (A1<=28,        A2<=15)

VAR#           28:             (____f_____)  "N_A37"           <31:0>
FUN   Mod      (A1<=59,        A2<=14)

VAR#           29:             (____f_____)  "N_A43"           <31:0>
FUN   Sub      (A1<=58,        A2<=60)

VAR#           30:             (____f_____)  "C6"<31:0>
FUN   Eql      (A1<=0,         A2<=16)
```

```
VAR#            31:             (_____d_s__)  "pres_state"          <4:0>
VAR#            32:             (___n______)  "P0"
VAR#            33:             (___n______)  "P1"
VAR#            34:             (___n______)  "P2"
VAR#            35:             (___n______)  "P3"
VAR#            36:             (___n______)  "P4"
VAR#            37:             (___n______)  "P5"
VAR#            38:             (___n______)  "P6"
VAR#            39:             (___n______)  "P7"
VAR#            40:             (___n______)  "P8"
VAR#            41:             (___n______)  "P9"
VAR#            42:             (___n______)  "P10"
VAR#            43:             (___n______)  "P11"
VAR#            44:             (___n______)  "P12"
VAR#            45:             (___n______)  "P13"
VAR#            46:             (___n______)  "P14"
VAR#            47:             (___n______)  "P15"
VAR#            48:             (___n______)  "P16"
VAR#            49:             (___n______)  "P17"
VAR#            50:             (___n______)  "P18"
VAR#            51:             (___n______)  "P19"
VAR#            52:             (___n______)  "P20"
VAR#            53:             (___n______)  "P21"
VAR#            54:             (___n______)  "P3_P9_P12_P16"
VAR#            55:             (___n______)  "P4_P11"
VAR#            56:             (___n______)  "P6_P13"

VAR#            57:             (________F_)  "CONTROL"
GRP#            0:              BEG = 0             LEN = 32 -----
 0  0:     (n___)     (0=>1     1=>2)     V = 0       "Reset"
 1  1:     (__v_)     (0        0)        VEC = "S2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
 2  2:     (n___)     (S1=>3    S2=>4     S3=>5   S4=>6   S5=>7   S6=>8   S7=>9   S8=>12   S9=>15
                      S10=>16   S11=>17   S12=>18   S13=>21   S14=>22   S15=>23   S16=>24   S17=>25
                      S18=>26   S19=>27   S20=>28   S21=>29)   V = 31                        "pres_state"
    <4:0>
 3  3:     (__v_)     (0        0)        VEC = "S2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
 4  4:     (__v_)     (0        0)        VEC = "S3 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
 5  5:     (__v_)     (0        0)        VEC = "S4 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0"
 6  6:     (__v_)     (0        0)        VEC = "S5 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0"
 7  7:     (__v_)     (0        0)        VEC = "S6 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
 8  8:     (__v_)     (0        0)        VEC = "S7 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1"
 9  9:     (n___)     (1=>10    0=>11)    V = 20      "C3"
 10 10:    (__v_)     (0        0)        VEC = "S8 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
 11 11:    (__v_)     (0        0)        VEC = "S9 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
 12 12:    (n___)     (1=>13    0=>14)    V = 21      "C4"
 13 13:    (__v_)     (0        0)        VEC = "S10 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
 14 14:    (__v_)     (0        0)        VEC = "S11 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
 15 15:    (__v_)     (0        0)        VEC = "S12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0"
 16 16:    (__v_)     (0        0)        VEC = "S11 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0"
 17 17:    (__v_)     (0        0)        VEC = "S13 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
 18 18:    (n___)     (1=>19    0=>20)    V = 27      "C5"
 19 19:    (__v_)     (0        0)        VEC = "S14 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0"
 20 20:    (__v_)     (0        0)        VEC = "S15 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0"
 21 21:    (__v_)     (0        0)        VEC = "S16 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0"
 22 22:    (__v_)     (0        0)        VEC = "S15 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0"
 23 23:    (__v_)     (0        0)        VEC = "S17 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0"
 24 24:    (__v_)     (0        0)        VEC = "S19 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0"
 25 25:    (__v_)     (0        0)        VEC = "S19 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0"
 26 26:    (__v_)     (0        0)        VEC = "S7 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1"
 27 27:    (__v_)     (0        0)        VEC = "S20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0"
 28 28:    (__v_)     (0        0)        VEC = "S21 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0"
 29 29:    (n___)     (1=>30    0=>31)    V = 30      "C6"
 30 30:    (__v_)     (0        0)        VEC = "S2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0"
 31 31:    (__v_)     (0        0)        VEC = "S19 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0"

VAR#58:    (_o___d____)  "P_N6"                 <31:0>
GRP# 1:    BEG = 32        LEN = 17 -----
 32    0:  (n___)          (0=>16   1=>1)          V = 54          "P3_P9_P12_P16"
 33    1:  (n___)          (0=>3    1=>2)          V = 35          "P3"
 34    2:  (____)          (0       0)             V = 17          "MUX_CONST"
```

```
  35   3:    (n___)       (0=>9    1=>4)         V = 41        "P9"
  36   4:    (n___)       (0=>6    1=>5)         V = 44        "P12"
  37   5:    (____)       (0       0)            V = 17        "MUX_CONST"
  38   6:    (n___)       (0=>7    1=>8)         V = 48        "P16"
  39   7:    (____)       (0       0)            V = 17        "MUX_CONST"
  40   8:    (____)       (0       0)            V = 23        "N_A25"        <31:0>
  41   9:    (n___)       (0=>13   1=>10)        V = 44        "P12"
  42   10:   (n___)       (0=>12   1=>11)        V = 48        "P16"
  43   11:   (____)       (0       0)            V = 17        "MUX_CONST"
  44   12:   (____)       (0       0)            V = 25        "N_A31"        <31:0>
  45   13:   (n___)       (0=>15   1=>14)        V = 48        "P16"
  46   14:   (____)       (0       0)            V = 29        "N_A43"        <31:0>
  47   15:   (____)       (0       0)            V = 17        "MUX_CONST"
  48   16:   (____)       (0       0)            V = 58        "P_N6"         <31:0>

VAR#59:      (_____d____)  "N8"                  <31:0>
GRP# 2:      BEG = 49     LEN = 9 -----
  49   0:    (n___)       (0=>8    1=>1)         V = 55        "P4_P11"
  50   1:    (n___)       (0=>5    1=>2)         V = 36        "P4"
  51   2:    (n___)       (0=>4    1=>3)         V = 43        "P11"
  52   3:    (____)       (0       0)            V = 17        "MUX_CONST"
  53   4:    (____)       (0       0)            V = 18        "N_A8"         <7:0>
  54   5:    (n___)       (0=>7    1=>6)         V = 43        "P11"
  55   6:    (____)       (0       0)            V = 24        "N_A28"        <31:0>
  56   7:    (____)       (0       0)            V = 17        "MUX_CONST"
  57   8:    (____)       (0       0)            V = 59        "N8"

VAR#60:      (_____d____)  "N9"                  <31:0>
GRP# 3:      BEG = 58     LEN = 3 -----
  58   0:    (n___)       (0=>2    1=>1)         V = 37        "P5"
  59   1:    (____)       (0       0)            V = 19        "N_A11"        <7:0>
  60   2:    (____)       (0       0)            V = 60        "N9"

VAR#61:      (_____d____)  "N19"                 <31:0>
GRP# 4:      BEG = 61     LEN = 9 -----
  61   0:    (n___)       (0=>8    1=>1)         V = 56        "P6_P13"
  62   1:    (n___)       (0=>5    1=>2)         V = 38        "P6"
  63   2:    (n___)       (0=>4    1=>3)         V = 45        "P13"
  64   3:    (____)       (0       0)            V = 17        "MUX_CONST"
  65   4:    (____)       (0       0)            V = 7         "N_A12"
  66   5:    (n___)       (0=>7    1=>6)         V = 45        "P13"
  67   6:    (____)       (0       0)            V = 26        "N_A34"        <31:0>
  68   7:    (____)       (0       0)            V = 17        "MUX_CONST"
  69   8:    (____)       (0       0)            V = 61        "N19"
```

Appendix F: List of Publications

1. R.Ubar, J.Raik, P.Paomets, E.Ivask, G.Jervan, A.Markus. Low-Cost CAD System for Teaching Digital Test. Proc. of the 1st European Workshop on Microelectronics Education. p. 48, Villard de Lans, France, Feb. 5-6, 1996.

2. R.Ubar, J.Raik, P.Paomets, E.Ivask, G.Jervan, A.Markus. Low-Cost CAD System for Teaching Digital Test. Microelectronics Education. World Scientific Publishing Co. Pte. Ltd. pp. 185-188, Grenoble, France, Feb.1996.

3. G.Jervan, A.Markus, P.Paomets, J.Raik, R.Ubar. Teaching Test and Design with Turbo Tester Software. Proc. of the 3rd Advanced Training Course: Mixed Design of Integrated Circuits and Systems MIXDES'96. pp. 589-594, Lodz, Poland, May 30 - June 1, 1996.

4. J.Raik, R.Ubar, G.Jervan, H.Krupnova. A Constraint-Driven Gate-Level Test Generator. Proc. of the 5-th Baltic Electronics Conference. pp. 237-240, Tallinn, Estonia, Oct. 1996.

5. R.Ubar, A.Markus, G.Jervan, J.Raik. Fault Model and Test Synthesis for RISC Processors. Proc. of the 5-th Baltic Electronics Conference. pp. 229-232, Tallinn, Estonia, Oct. 1996.

6. G.Jervan, A.Markus, P.Paomets, J.Raik, R.Ubar. CAD Software for Digital Test and Diagnostics. Proc. of the Conference on Design and Diagnostics of Electronic Circuits and Systems '97. Ostrava, Czech Republic, May 12-14, 1997.

7. M.Brik, G.Jervan, A.Markus, J.Raik, R.Ubar. A Hierarchical Automatic Test Pattern Generator Based on Using Alternative Graphs. Proc. of the 4-th International Workshop on Computer Aided Design of Modern Devices and ICs. pp. 415-420, Poznan, Poland, June 12-14, 1997.

8. G.Jervan, A.Markus, J.Raik, R.Ubar. Automatic Test Generation System for VLSI. Proc. of the 1-st Electronic Circuits and Systems Conference. pp. 255-258, Bratislava, Slovakia, Sep. 4-5, 1997.

9. G.Jervan, A.Markus, P.Paomets, J.Raik, R.Ubar. A Set of Tools for Estimating Quality of Built-In Self-Test in Digital Circuits. Proc. of the International Symposium on Signals Circuits and Systems. pp. 362-365, Iasi, Romania, Oct. 2-3, 1997.

10. G.Jervan, A.Markus, J.Raik, R.Ubar. Assembling Low-Level Tests to High-Level Symbolic Test Frames. Proc. of the 15th NORCHIP Conference pp. 275-281, Tallinn, Estonia, Nov. 10-11, 1997.

11. M.Brik, G.Jervan, A.Markus, P.Paomets, J.Raik, R.Ubar. Mixed-Level Test Generator for Digital Systems. Proceedings of the Estonian Acad. of Sci. Engng, 1997, Vol. 3 , No 4, pp. 269-280.

12. M.Brik, G.Jervan, A.Markus, P.Paomets, J.Raik, R.Ubar. Hierarchical Test Generation for Digital Systems. Mixed Design of Integrated Circuits and Systems, Kluwer Academic Publishers, pp. 131-136, 1998.

13. G.Jervan, A.Markus, P.Paomets, J.Raik, R.Ubar. A CAD System for Teaching Digital Test. Proc. of the 2nd European Workshop on Microelectronics Education, Noordwijkerhout, the Netherlands, May 14-15, 1998. (to be published)

14. G.Jervan, A.Markus, P.Paomets, J.Raik, R.Ubar. A CAD System for Teaching Digital Test. Kluwer Academic Publishers, 1998. (to be published)

15. G.Jervan, A.Markus, J.Raik, R.Ubar. Hierarchical Test Generation with Multi-Level Decision Diagram Models. Proc. of the 7-th IEEE North Atlantic Test Workshop, West Greenwich, RI, USA, pp. 26-33, May 28-29, 1998.

16. G.Jervan, A.Markus, J.Raik, R.Ubar. DECIDER: VHDL based Test Generation SystemProc. of the 5th Int. Conf. on Electronic Devices and Systems, Brno, Czech Republic, June 11-12, 1998. (to be published)
17. G.Jervan, A.Markus, R.Ubar, J.Raik. Mixed Level Deterministic - Random Test Generation for Digital Systems. Proc. of the MIXDES'98 Conf., Lodz, Poland, June 18-20, 1998. (to be published)
18. G.Jervan, A.Markus, J.Raik, R.Ubar. A Decision Diagram based Hierarchical Test Generator. Proc. of the BEC'98 Conference, Tallinn, Estonia, 1998. (to be published)