TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Computer Engineering

IAF40LT

Harri Bendi 113130

LABORATORIAL OPERATING INSTRUCTIONS FOR TEST PROGRAMME DESIGN USING PARWAN MICROPROCESSOR

Bachelor thesis

Raimund-Johannes Ubar

PhD

Professor

Artjom Jasnetski

MSc

PhD student

Tallinn 2015

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Harri Bendi

08.06.2015

Abstract

The ultimate goal of this thesis is to implement Parwan Test KIT within a wider range of students to to get a grasp on benefits regarding This thesis presents a theoretical backround about high-level decision diagrams and gives an overview of how these could be used in software-based self-tests. The goal Components under test form Parwan microprocessor. By combining Parwan instruction set with high-level decision diagrams 2 laboratory works are created.

brings out the good qualities

Abstract is an essential and compulsory part of the thesis. It provides an overview about the aims-, the most important issues, results and conclusions of the thesis. Abstract is a short overview of the thesis which does not explain or justify anything but presents the content of the work. Abstract in Estonian is called *Annotatsioon* and in Russian *Аннотация*.

Depending on the thesis language the abstracts must be presented as following:

- If the thesis is written in English, the abstract is ½ A4 long and the abstract in Estonian (*Annotatsioon*) is of length 1 A4.
- If the thesis is written in Russian, the abstract is ½ A4 long, which is followed by abstract in Estonian (*Annotatsioon*) of length 1 A4, and abstract in English of length 1 A4.

For abstracts not in the main thesis language, thesis title in foreign language is added in between the heading Abstract and the abstract content.

The last paragraph of abstract is obligatory and must be written accordingly:

This thesis is written in English and is 50 pages long, including 6 chapters, 19 figures and 6 tables.

Annotatsioon

Laboratoorse töö juhendi koostamine mikroprotsessori Parwan testprogrammide projekteerimiseks

The requirements of foreign abstract are presented under Abstract.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 50 leheküljel, 6 peatükki, 19 joonist, 6 tabelit.

Table of abbreviations and terms

Table of abbreviations is divided into 2 sections. The first part consists of Parwan modules. The second part originates from theoretical part.

ALU	Arithmetical logic unit
AC	Accumulator
IR	Instruction register
PC	Program counter
MAR	Memory address register
SR	Status register
SHU	Shifter unit
CTL	Control unit
HLDD	High-level decision diagram
SBST	Software-based self-test
ATE	Automated test equipment
ATPG	Automated test pattern generator
MUT	Modules under test
ISA	Instruction set architecture
MP	Microprocessor
SHMP	Simple hypothetical microprocessor
FSM	Final state machine

VHDL	Virtual hardware design language
POI	The page operand of the instruction
OOI	The offset operand of the instruction
RTL	Register-transfer level
TCL	Tool command language

Table of contents

List of figures

List of tables

1. Introduction

The goal of this work is to ease the process of reassessment and manipulation of testing of digital microprocessors. To reach that goal, this thesis gives an overview about the usage of high-level decision diagrams in testing of digital systems, an insight how Parwan microprocessor works and presents 2 laboratory works meant for an individual usage by students. This document is formatted according to the requirements and following the template for thesis.

2. Theoretical basis

This topic offers an insight into high-level decision diagrams based self-test generator for microprocessors. The following material is based on sources [1] and [2].

2.1. Abstract

Microprocessor modeling using the theory of high-level decision diagrams (HLDD) for representing instruction sets creates a new convolution of software-based self-test (SBST) methodology. A set of HLDD graphs is formally derived from the instruction set of the microprocessor. The proposed abstraction allows representing a broad class of faults in microprocessors by a uniform high-level fault model in terms of conditional node faults in HLDDs. Based on this model, formal procedures are developed for test program generation. Tests are created on the basis of pre-constructed subroutine templates for initialization and observation of results, supported by HLDD-guided content generation. These templates can be derived in a formal way from HLDDs. The experimental results show the superiority of the new method by achieving a higher quality of tests compared to the previous results.

2.2. Software-based self-test in general

General idea behind SBST is to execute the test program on embedded processor for the purpose of testing the processor itself and its surrounding resources. SBST is a non-intrusive test methodology that is based on the available processor resources and instruction set. In SBST, the role of the ATE is loading the test program into memory and reading the final test results back after the execution of test program is finished.

In general, the development of an SBST program consists of four steps:

1. Creation and optimization of test pattern delivery templates in assembly language;
2. Module-level instruction-imposed (functional) constraint extraction;
3. Test generation process for each module of the processor under test;
4. Translation of test patterns to self-test programs.

The last step is basically a process of joining the test pattern with the test pattern delivery template.

The quality of the SBST test is primarily affected by the test patterns. One of the ways to obtain the test patterns is executing an automated test pattern generator (ATPG). To ease the task of ATPG, processor can be divided into modules under test (MUT). An alternative way is to use random test patterns for MUTs. Although the gate level fault coverage for MUT is acceptable in deterministic and random test pattern generation, some of the generated patterns are typically functionally infeasible when considering the processor as a whole. Thus, ATPG has to be guided with functional constraints to produce functionally feasible test patterns.

2.3. The usage of HLDD during test program generation

In the following it is considered that SBST program generation for microprocessors at the behavioural-level uses processor instruction set architecture (ISA). The following describes how to automate test program generation on the basis of high-level decision diagram which in turn are created from the instruction set.

2.3.1. Advantages of HLDD

Compared to traditional cases, instead of having the instructions as test targets, the HLDDs are considered as test target in this approach. To each such function a well-defined location of the processor structure corresponds (in the general case, a register with its input logic). In the traditional case when considering instructions as test targets, larger portion of the structure should be reasoned, which makes the diagnosis more difficult and increases the possibility of fault masking.

The main impacts of the proposed approach are: higher fault coverage, reduced probability of fault masking, better diagnostic opportunities, and the overall compactness of the test program due to its cyclical organization.

2.3.2. Causes for advantages of HLDD

The higher fault coverage is achieved by targeting a novel hard-to-test fault class called "*unintended actions*" which in traditional methods is neglected. The reduced probability of fault masking is achieved by targeting one-by-one the proofs of correct behaviour of

only “small portions” of the microprocessor functionality (called „*angel’s approach*“[1]) instead of targeting the detection of faulty instructions (called „*devil’s approach*“[2]) as in the traditional case. By targeting one by one only “small portions” of functionality during testing, it is possible to improve diagnostic resolution.

The proposed abstraction in the form of HLDD model allows developing a uniform high-level fault model in terms of *conditional node faults* in HLDDs, which represents a broad class of faults traditionally used in testing microprocessors. Based on this model, formal procedures are developed for test program generation. Tests are created on the basis of pre-constructed subroutine templates for initialization and observation of results, supported by HLDD-guided content generation. These templates can be derived in a formal way from HLDDs.

2.3.3. Test parts and strategies using HLDD

A test is composed of three parts:

1. Initialization of the Microprocessor (MP) - bringing the MP into a state which is needed for the following test action;
2. Test action;
3. Registration of the response to the test action.

Two test strategies are used:

1. Scanning testing – for testing the data path of the MP, represented by terminal nodes of HLDD graphs;
2. Conformity testing – for testing the control part of the MP, represented by non-terminal (internal) nodes.

To achieve high fault coverage, a hierarchical approach is used: the control functions are tested on the basis of behavioural level information, whereas the data for testing the data-path is generated by traditional gate-level ATPG. Detailed properties of conformity and scanning tests can be found in Appendix 6.

---

[1] To prove that part functions correctly; to prove that path is OK.

[2] To prove that part is erroneous; to prove the existence of the fault.

2.4. Example of representing microprocessor with HLDD

Considering as an example a simple hypothetical microprocessor (SHMP) with its instruction set in Table 1 and a general behavioural level structure in Figure 1.

Table 1. Instruction set of SHMP.

OP	**B**	**Mnemonic**	**Semantics and RT level operations**
0	0	LDA A1, A	READ memory $R(A1) = M(A)$, $PC = PC + 2$
	1	STA A2, A	WRITE memory $M(A) = R(A2)$, $PC = PC + 2$
1	0	MOV A1,A2	Transfer $R(A1) = R(A2)$, $PC = PC + 1$
	1	CMA A1,A2	Complement $R(A1) = \neg R(A2)$, $PC = PC + 1$
2	0	ADD A1,A2	Addition $R(A1) = R(A1) + R(A2)$, $PC = PC + 1$
	1	SUB A1,A2	Subtraction $R(A1) = R(A1) - R(A2)$, $PC = PC + 1$
3	0	JMP A	Jump $PC = A$
	1	BRA A	Conditional jump (Branch inst.) IF C=1, THEN $PC = A$, ELSE $PC = PC + 2$

Denote the instructions of the microprocessor (MP) as the values of a complex variable I represented as concatenation of 5 instruction sub-variables $I = OP.B.A1.A2.A$. The variables OP and B denote two fields of the operation code, $A1$ and $A2$ are register addresses, and A is the memory address. Let $V(OP) = V(A1) = V(A2) = \{0,1,2,3\}$ and $V(B) = \{0,1\}$.

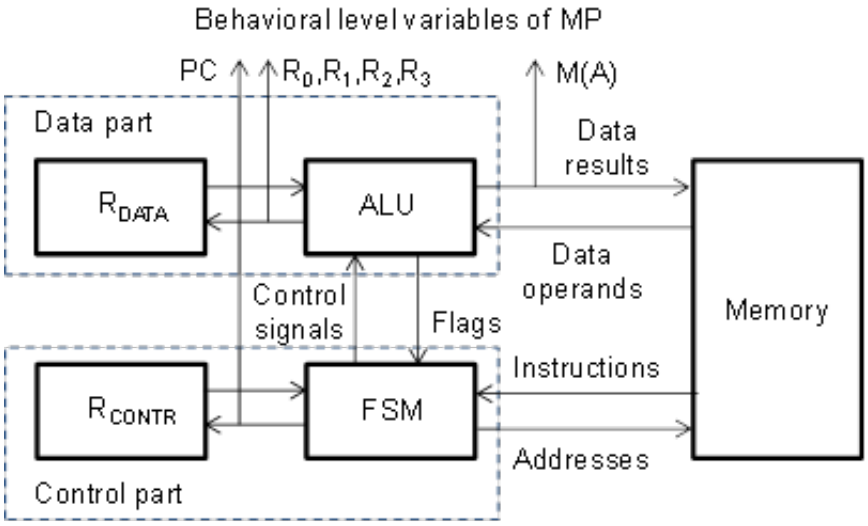

Figure 1. Behavioural level structure of SHMP.

MP can be partitioned into three parts: control part, datapath and memory. There are two register blocks R_{DATA} and R_{CONTR} in the MP: the register block in the datapath consists of 4 general data registers R_{DATA} = {R0, R1, R2, R3} and the control part includes 2 control registers R_{CONTR} = {PC, A} where PC is the program counter, and A is the address register for addressing the data. ALU is a combinational part of the MP which covers all data manipulation circuits, decoders, multiplexers, demultiplexers etc. Control part includes finite state machine (FSM) with state register and control logic.

Consider MP functionally as a set of the following behavioural level functions:

$R_i = f_i\,(I,\ S(R_i)) = f_i\,(OP,\ B,\ S(R_i))$ where $R_i \in R_{DATA}$, i = 0,1,2,3, and $S(R_i) = \{R_{DATA}, M(A)\}$ is the set of data arguments for the functions f_i (a set of source registers);

$PC = f_{PC}\,(I,\ C,\ PC) = f_i\,(OP,\ B,\ PC)$ where C is the flag variable serving as the condition for the branch operation;

$M(A) = f_M\,(I,\ S(M(A))) = f_i\,(OP,\ B,\ S(M(A)))$ where $S(M(A)) = \{R_{DATA}, M(A)\}$.

The functionality of MP can now be represented by a set of behavioural level variables $Z = R_{DATA} \cup R_{CONTR} \cup M(A)$ and by a set of functions $F = \{f_0, f_1, f_2, f_3, f_{PC}, f_M\}$. The behaviour of MP can be modeled by the functional basis F and monitored through the variables in Z. For modeling of F the behavioural level HLDD model is used.

2.4.1. Matching HLDD nodes with instruction set

The HLDD model of SHMP given by the instruction set in Table 1 is depicted in Figure 2. It represents the set of 7 functions in F in the form of 7 HLDDs, respectively: G_{Ri}, i = 0,1,2,3; $G_{R(A2)}$, G_{PC}, and $G_{M(A)}$. The 4 graphs G_{Ri} are merged and share a similar sub-graph which represents the logic of ALU. The graphs $G_{R(A1)}$ and $G_{R(A2)}$ are accessed when modeling the nodes $R(A1)$ and $R(A2)$, respectively, in the graphs G_{Ri} or $G_{M(A)}$.

For simplicity, the nodes are called by the names of node variables or by the expressions in the nodes. To distinguish the nodes which are labeled by the same variable in the given HLDD, subscripts are used at that node variable. For example, in the graphs G_{Ri}, there are three different nodes labeled by the same variable B, and the subscript at B distinguishes the nodes.

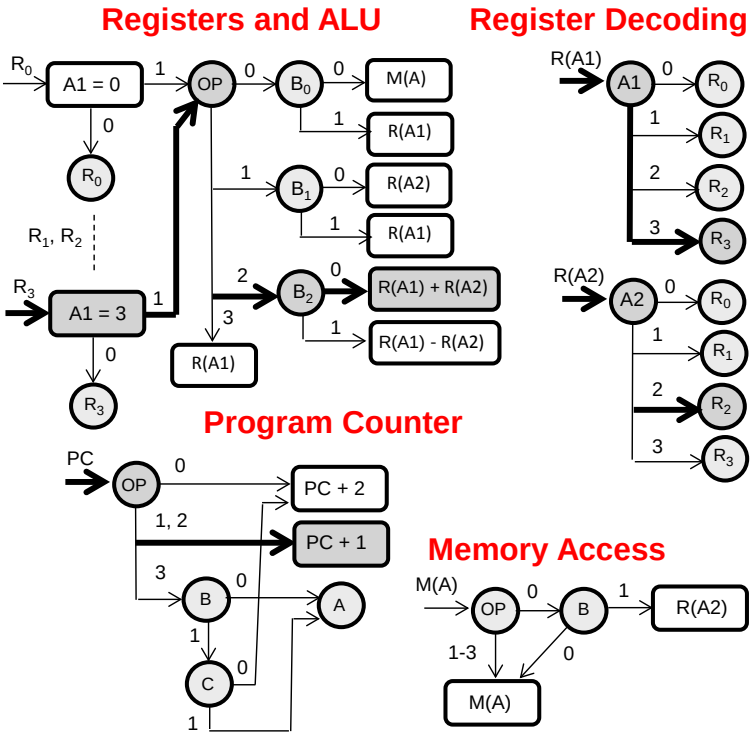

Figure 2. HLDD model for the SHMP.

Each instruction in Table 1 can be modeled by corresponding paths in the HLDD model. When simulating an instruction, its related path in the HLDD is activated. For example, when simulating the instruction I = (OP=2.B=0.A1=3. A2=2), the following paths in Figure 2 are activated: G_{R3}: $l(A1=3, OP, B_2, R(A1)+R(A2)$, $G_{R(A1)}$: $l(A1,R3)$,

$G_{R(A2)}$: $l(A2,R2)$, G_{PC}: $l(OP, PC+1)$, respectively, in the graphs G_{R3}, $G_{R(A1)}$, $G_{R(A2)}$ and G_{PC}. The activated paths are highlighted by bold edges and gray coloured nodes in Figure 2.

Each HLDD node can be regarded as a functional unit of the microprocessor exercised by a corresponding instruction. For example, the terminal nodes which are labeled by variables may represent registers or buses, whereas other terminal nodes which are labeled by arithmetic or logic expressions represent the data manipulation units within the ALU. The non-terminal (internal) nodes of HLDDs are representing the units for interpretation of control information (*OP*, *B*, *C*, etc.) which may be decoders, multiplexers or de-multiplexers. For example, the node $A1 = 0$ in G_{R0} represents de-multiplexer, the node *A*2 in $G_{R(A2)}$ represents a multiplexer, the nodes *OP* and *B* in the graphs represent decoders.

Because of this one-to-one mapping between the nodes in HLDDs and the corresponding high-level functional units, the HLDD nodes can be used as a checklist for high-level test planning and organization of test programs for microprocessors. For formalized test program generation, however, a suitable high-level (behavioural) fault model is needed (more about the subject ‘uniform fault model for microprocessors’ can be read in Appendix 3and Appendix 4).

2.5. Conclusion for node testing using HLDD

To conclude, for testing a node in the HLDD model, three actions are needed:

1. Local fault activating by satisfying the activation constraints (see Appendix 5);
2. Topological propagation of the fault through the HLDD by satisfying the propagation constraints (see Appendix 5);
3. System-level fault propagation through the high-level components of the system, represented by HLDDs.

The first two tasks are formulated by the constraints of the fault model, whereas the third task is the test generation problem which is covered in

3. Basics of Parwan microprocessor

This chapter provides an overview of how Parwan microprocessor works. The work is combined with the materials from [3] and [4].

3.1. Parwan in general

Parwan is an eight-bit microprocessor which has an 8-bit Data Bus and a 12-bit Address Bus for external accesses. It has a limited number of arithmetic and logic instructions, several jump and branch instructions, subroutine call instructions. Some instructions have an addressing mode, which provides for direct and indirect addressing. Parwan has an accumulator, a reduced ALU, a shifter, program counter, and a total of five flags: overflow, carry, zero, sign, interrupt enable(not used by in this thesis). The behavioural description and dataflow description of Parwan is implemented using VHDL.

3.2. Memory organization

Parwan has a 12-bit address bus, which is partitioned into sixteen pages of 256 bytes each, as depicted in Figure 3. The four most significant bits of the address are for the page address and the remaining eight bits of the address are for the offset within the page. For simplicity, the addresses are usually written in hexadecimal. The page address and offset are separated by colon [**:**]. These page areas are not continuous.

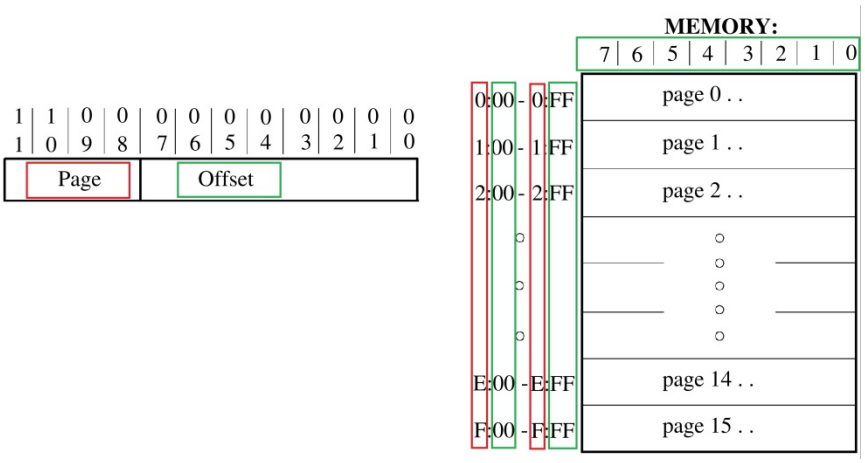

Figure 3. Page and offset parts of Parwan addresses.

3.3. Parwan components

In Figure 4, overview of different Parwan components with their inputs and outputs is presented. Connections between those components also indicate to a number of bits that are carried.

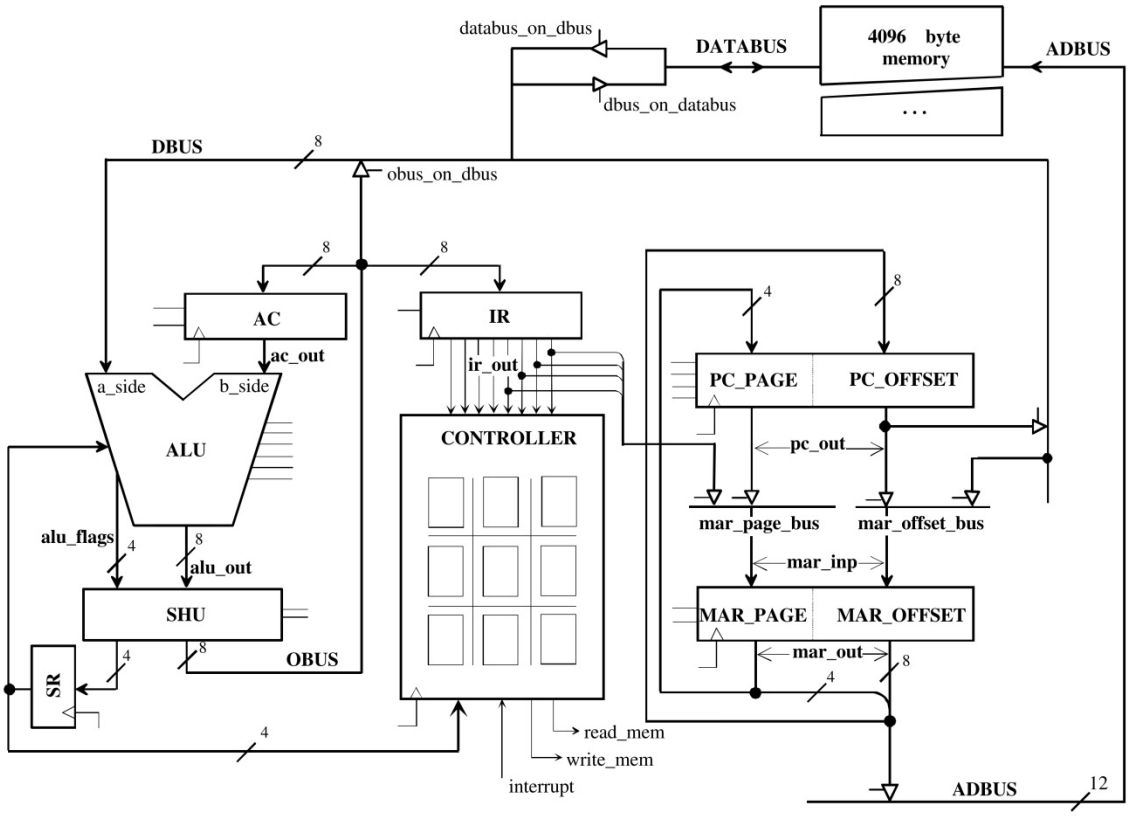

Figure 4. Bussing structure of Parwan.

A simplified verison of Parwan components is given in Figure 5.

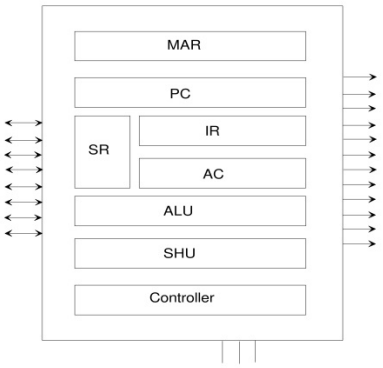

Figure 5. General layout of Parwan.

3.4. Addressing mode

If an instruction has a page operand and an offset operand for the location of its data, it is in full addressing mode, meaning this mode can have both direct and indirect addressing. In total, full address instructions use 2 bytes. The 4th bit of the 1st byte - seen as B4 in Figure 6- indicates whether addressing is in direct or indirect mode.

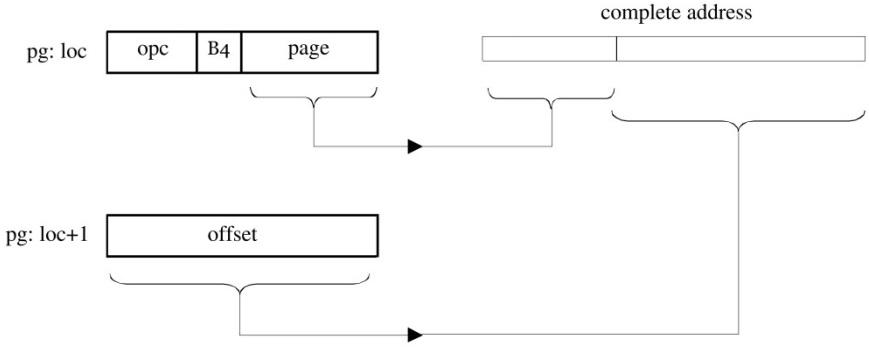

Figure 6. Addressing in full-address instructions.

If an instruction has only an offset operand for the location of its data (as for branch and jsr operations), it is in page addressing mode, where the effective address is obtained by concatenating the page address of the instruction with the offset operand, as seen in Figure 7 In the page addressing mode, an instruction can only reference memory locations within the page where the instruction appears.

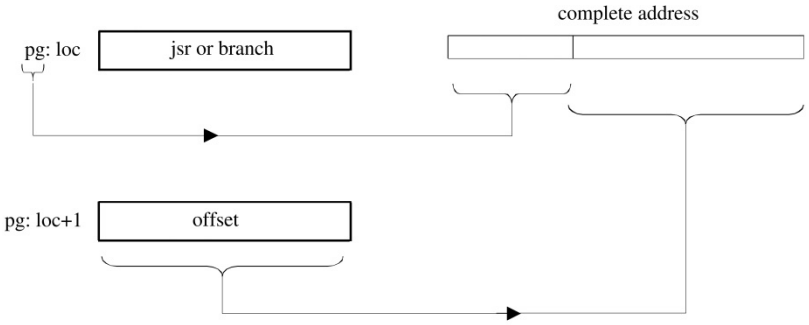

Figure 7. Addressing in page-address instructions.

In the direct addressing mode, the actual address is obtained by concatenating the page operand with the offset operand of the instruction.

As presented in Figure 8, in the indirect addressing mode, the offset for the actual address (blue) is fetched from the memory location of the address obtained by concatenating the page operand (POI) and the offset operand (OOI) of the instruction (red). Because of the page separation, the page part of the actual address (blue) is the same as the page operand of the instruction (POI).

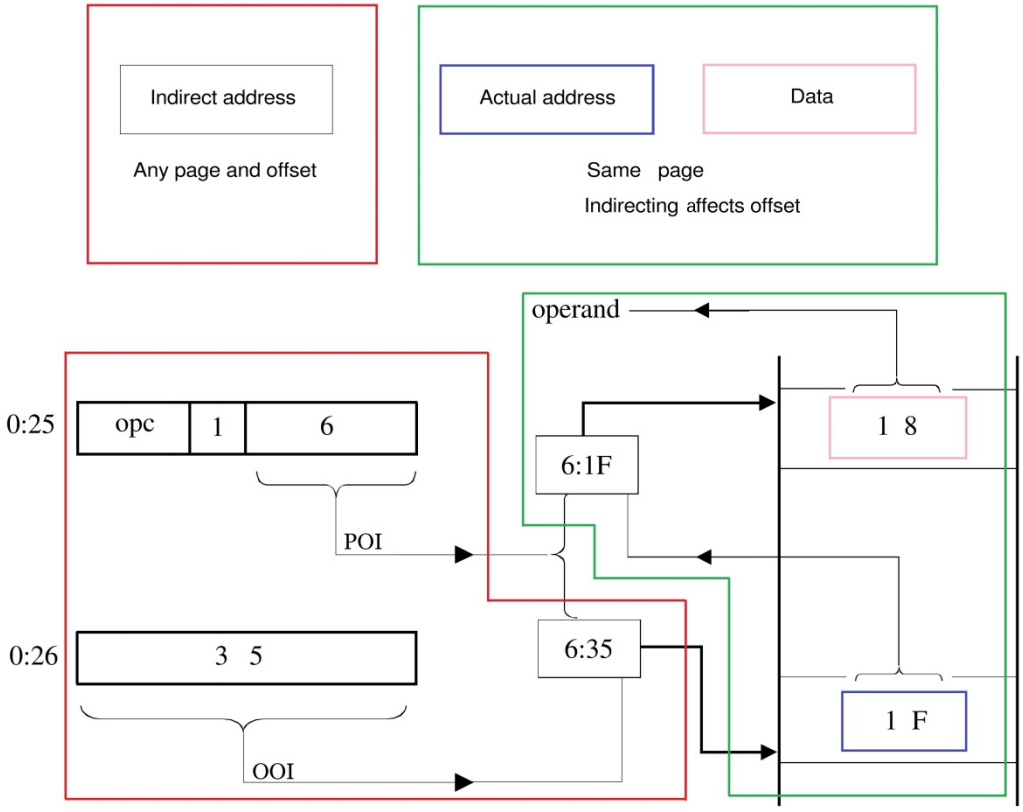

Figure 8. An example of indirect addressing in Parwan.

3.5. Instructions

Parwan microprocessor has 17 instructions in total:

- Load and store operations;
- Arithmetic and logical operations ;
- Jump and branch instructions.

All instructions have different address schemes, flags they use and flags they influence during operations. Details considering the usage of different instructions are shown in

Table 2 Instructions with a full address scheme have 12 address bits to use and enable indirect addressing. Instructions with 8 address bits have only direct addressing capabilities. Instructions with non-existent address scheme do not need to point anywhere in memory to have the instruction executed without issues. As a consequence, they have no address bits.

Table 2. Summary of Parwan instructions.

Instruction mnemonic	**Brief description**	**Address bits**	**Address scheme**	**Indirect address**	**Flags use**	**Flags set**
LDA loc[3]	Loads data from loc to AC	12	FULL	YES	----	--zn
AND loc	AC data AND loc	12	FULL	YES	----	--zn
ADD loc	Add loc to AC	12	FULL	YES	-c--	vczn
SUB loc	Subtracts loc from AC	12	FULL	YES	-c--	vczn
JMP adr	Allows long jumps between the memory pages	12	FULL	YES	----	----
STA loc	Stores data from AC to loc	12	FULL	YES	----	----
JSR tos	Jump with return; short range jumps only	8	PAGE	NO	----	----
BRA_V adr	Branch to adr if overflow	8	PAGE	NO	v---	----
BRA_C adr	Branch to adr if carry	8	PAGE	NO	-c--	----
BRA_Z adr	Branch to adr if zero	8	PAGE	NO	--z-	----
BRA_N adr	Branch to adr if negative	8	PAGE	NO	---n	----
NOP	No operation; used with JSR to guarantee return	-	NONE	NO	----	----
CLA	Clear AC	-	NONE	NO	----	----
CMA	Complement AC	-	NONE	NO	----	--zn

[3] Loc as memory data

Instruction mnemonic	Brief description	Address bits	Address scheme	Indirect address	Flags use	Flags set
CMC	Complement carry	-	NONE	NO	-c--	-c--
ASL	Arith shift left	-	NONE	NO	----	vczn
ASR	Arith shift right	-	NONE	NO	----	--zn

3.6. Operation Codes

Parwan microprocessor instructions operate using 8 bits. The combination of values of those 8 bits dictates which instruction is to be carried out, how addressing is done and which flags are used (Table 3). Understanding operation codes is crucial in finding tested paths and nodes in Parwan HLDD. It should be noted that instructions with opcode bits with a value other than $111_2 = 7_{10}$ and $110_2 = 6_{10}$ use full addressing mode, hence the youngest bits indicate to a page address. Since there are 10 instructions with opcode bits $111_2 = 7_{10}$, the 4 youngest bits are required to distinguish different instructions. Therefore the youngest bits can not be used to indicate to a certain address.

Table 3. Parwan instruction opcodes.

Instruction Mnemonic	Opcode Bits 7 6 5	Direct/Indirect Bit[4] 4	Youngest Bits 3 2 1 0
LDA loc	0 0 0	0/1	Page adr
AND loc	0 0 1	0/1	Page adr
ADD loc	0 1 0	0/1	Page adr
SUB loc	0 1 1	0/1	Page adr
JMP adr	1 0 0	0/1	Page adr
STA loc	1 0 1	0/1	Page adr
JSR tos	1 1 0	-	- - - -
BRA_V adr	1 1 1	1	1 0 0 0
BRA_C adr	1 1 1	1	0 1 0 0
BRA_Z adr	1 1 1	1	0 0 1 0

[4] Also as B4 (see Figure 6)

Instruction Mnemonic	**Opcode Bits 7 6 5**	**Direct/Indirect Bit5 4**	**Youngest Bits 3 2 1 0**
BRA_N adr	1 1 1	1	0 0 0 1
NOP	1 1 1	0	0 0 0 0
CLA	1 1 1	0	0 0 0 1
CMA	1 1 1	0	0 0 1 0
CMC	1 1 1	0	0 1 0 0
ASL	1 1 1	0	1 0 0 0
ASR	1 1 1	0	1 0 0 1

3.7. Parwan Test Environment

The Parwan test environment (PTE) is composed to automate and simplify fault simulation for Parwan microprocessor. PTE joins multiple tools and digital design in a single automated workflow. The goal is to write SBST program which generates test vectors and gain sufficient fault coverage for processor under test.

Using framework depicted in Figure 9, different SBST strategies and algorithms can be implemented as test programs and evaluated. Also, different fault simulators can be involved to analyse different fault classes. According to fault coverage report, different improvements can be made for design or test program. All this enables an improvement in the testability of design or in the test program itself.

[5] Also as B4 (see Figure 2)

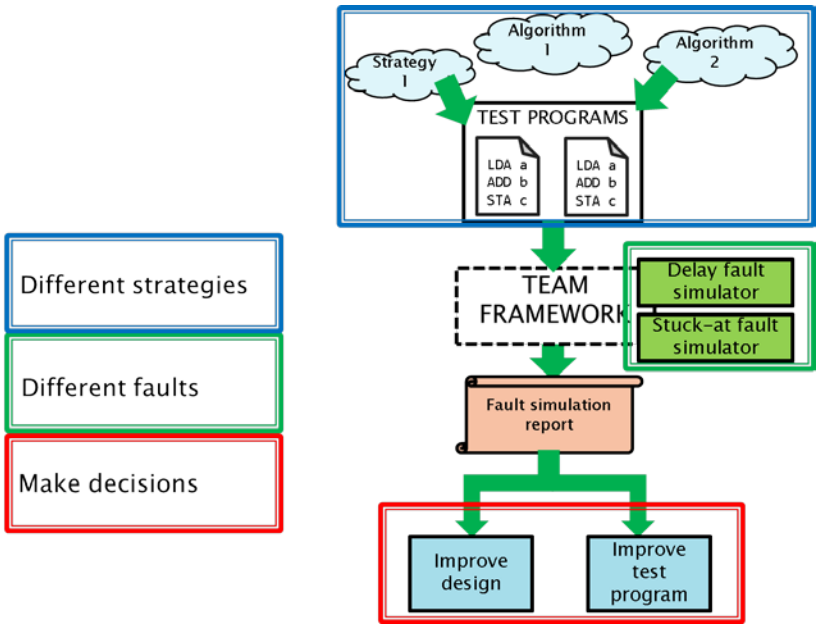

Figure 9. Advantages of laboratory framework TEAM.

Target of this environment is to estimate the fault coverage of the processor under test, achieved by composing test program. The fault coverage can be estimated for each module of Parwan processor.

Overview of the simulation process in PTE can be seen in Figure 10 and is divided into following steps:

1. The first phase is to compose test program in assembly language, using the knowledge base about the processor under test (mainly instruction set). When the test program is written, Test Environment can be started. Since ModelSim does not recognize assembly, the assembly program is translated into a memory file (Self-test program).
2. The next phase is processor simulation in ModelSim with composed memory file. During the simulation process, the input signals of the module under test are being extracted and composed into test vectors. The extracted test vectors are then composed to a specific format (done by Python scripts), used by fault simulators from Turbo Tester tool set.
3. The final phase is gate level fault simulation using Turbo Tester tools. After the fault simulation is completed, the results can be analysed to improve the test program code.

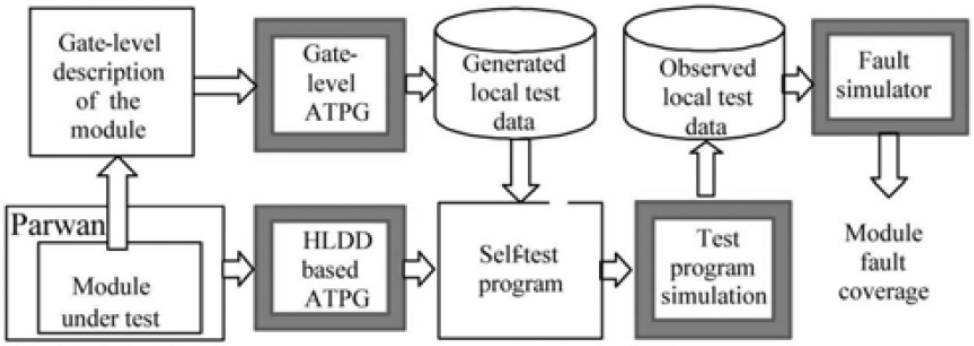

Figure 10. Experimental set-up.

4. Laboratory work nr. 1:

The goal of this laboratory work is to provide an overview of how to operate with PTE, to introduce the usage of script files and to give an insight into how to improve testing results by modifying test programs written in assembly.

4.1. Task 1: Setting up a testing environment

In order to effectively measure the quality of our testing programme written for Parwan microprocessor in assembly, it is necessary to install some software first:

1. ModelSim software[6]:
   http://www.mentor.com/company/higher_ed/modelsim-student-edition
2. Python compiler[7]:
   http://www.python.org/download/

Named software is required to have Parwan Test KIT (PTK) functional and running. The Test KIT itself contains following elements:

- Parwan processor VHDL files, RTL description;
- Parwan processor AGM files, Gate-level description;
- Python scripts;
- TCL scripts;
- Windows Batch script.

4.2. Task 2: Running the scripts

In order to run the fault simulation, test program must be written. Now, in the root folder of the KIT, the assembly file **program.asm** exists. In order to complete the current laboratory work, You need to overwrite file program.asm with Your assembly

[6] Requires a student license for activation. Information about activation will be provided via e-mail after the installation.
[7] Works only with version 2.7.*

language. When the program file is composed, it can be used to estimate the fault coverage for each Parwan module (see Figure 4).

There are 2 options to review steps that execution of RUN.bat has processed:

1. Running the script under the command prompt shell:
    a. Press **[Win**] **+ [R**]. In text field, type **cmd** and press **Enter**;
    b. Copy the path where RUN.bat file is located. Path can be found on the address bar;
    c. Open cmd-window, type **cd**, paste correct path (use **right click** -> **paste**, not **[ctrl] + [v]**) and press **Enter.**
2. Running the script directly:
    a. Prior to executing RUN.bat, open it with Notepad;
    b. Scroll to the end of the file and before the command **exit /b %errorlevel%** (line 178 in the original RUN.bat file), type the following:

    pause
    c. Save modified file -new name is irrelevant since no other program is using this file- in the folder where the original RUN.bat file is located;
    d. Execute the new file directly.

During execution of RUN.bat, 2 entries are required:

1. Parwan module to be tested. Requires an entry of a respective number;
2. File name that stores the initial memory dump. Requires to type **memory.mem**.

4.3. Task 3: Analysis

After the execution, text file **report.txt** is created in the root folder. Open it with Notepad to view fault coverage percentage for a chosen Parwan module using Your program. If the value of fault coverage percentage for a given module is less than the one given in Table 4, repeat modifying Your assembly program until the value exceeds figure in the table.

4.4. Example code snippet

On page 30, the example code snippet with comments is given. The snippet demonstrates some basic logic and arithmetic operations, jumps between subroutines and addressing. The code snippet is fully commented to ease the understanding of what the program is doing during each instruction. Since memory.bat file is programmed to compile only pure assembly, it is unable to identify comments. Therefore it is necessary to remove comments before running RUN.bat file during the experiment.

```
jmp example   // jump to subroutine labeled ’example’
label example // subroutine ’example’
nopp          // no operation
lda a         // load a to AC. AC holds value 10
add b         // add b to AC. AC becomes 30
sta a         // store sum 30 from AC to memory address a
jmp next      // jump to subroutine labeled ’next’

label clear   // subroutine ’clear’
nopp          // no operation. Use always, when JSR is used
cla           // clear AC
jmpi clear    // jump back to subroutine ’clear’

label next    // subroutine ’next’
ldai addr     // load value 20 to AC indirectly
jsr clear     // jump with return to subroutine “clear”

label end     // end of program
jmp end

int addr 3001 // points at address 3001

at 3000       // at memory address 3000
int a 10      // variable a; at memory address 3000
int b 20      // variable b; at memory address 3001
```

The fault coverage for ALU using this example is 56.47% which is a bad result. This indicates to a necessity to modify our initial assembly file. It should be noted that these laboratory works are for educational purposes which eliminates the necessity to optimize Your code. For a better result, it is also suggested to increase the number of operands or use some instructions repetitively to imitate instruction cycles.

4.5. Required information for LAB1

In order to write Your program, a variant must be chosen, depending on Your student code. 2 last digits in the student code determine which file has to be written and which additional Parwan module has to tested (besides ALU). Detailed information is seen in the Table 4. It is recommended to create a flowchart for any given algorithm before writing assembly files. Additional Parwan module is to be tested with Your file. Report paper can be found in Appendix 1.

Table 4. Variant table for LAB1.

Student code	**Base file in assembly for testing ALU**	**Required fault coverage for ALU**	**Additional Parwan module to be tested**
xxxxx0 - xxxxx1	Addition	70	
xxxxx2 - xxxxx3	Multiplication	80	
xxxxx4 - xxxxx5	Subtraction	70	
xxxxx6 - xxxxx7	Fibonacci	80	
xxxxx8 - xxxxx9	Exponentiation	80	
xxxx0x - xxxx1x			AC
xxxx2x - xxxx3x			IR
xxxx4x - xxxx5x			PC
xxxx6x - xxxx7x			MAR, SR
xxxx8x - xxxx9x			SHU

5. Laboratory work nr. 2:

The goal of this laboratory work is to provide an overview of Parwan HLDD model and method of how to write an assembly by combining HLDD model with opcodes.

5.1. Using Parwan HLDD model

General layout of Parwan HLDD model (Figure 11 and Figure 12) is based on processes that affect different Parwan components and consists of the following parts:

1. ALU data path;
2. ALU flags;
3. Indirect addressing;
4. Direct addressing;
5. Instruction addressing;
6. Output behaviour;
7. Instruction addressing;
8. Next memory page calculation;
9. Next PC offset calculation.

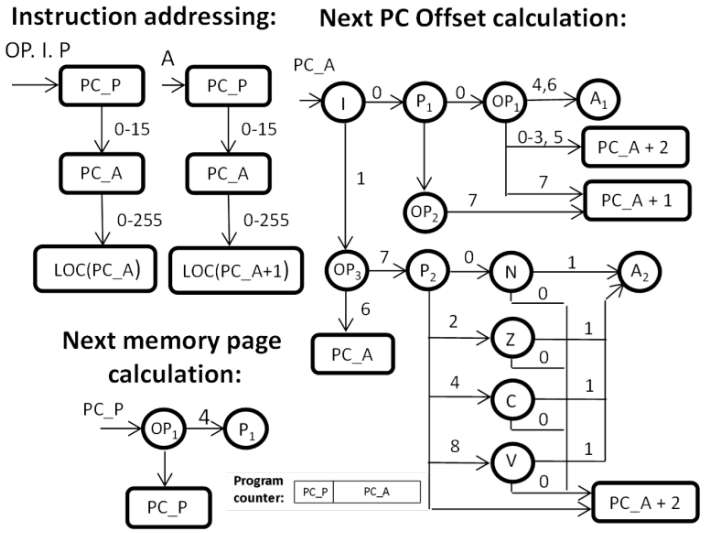

Figure 11. Parwan HLDD model (1).

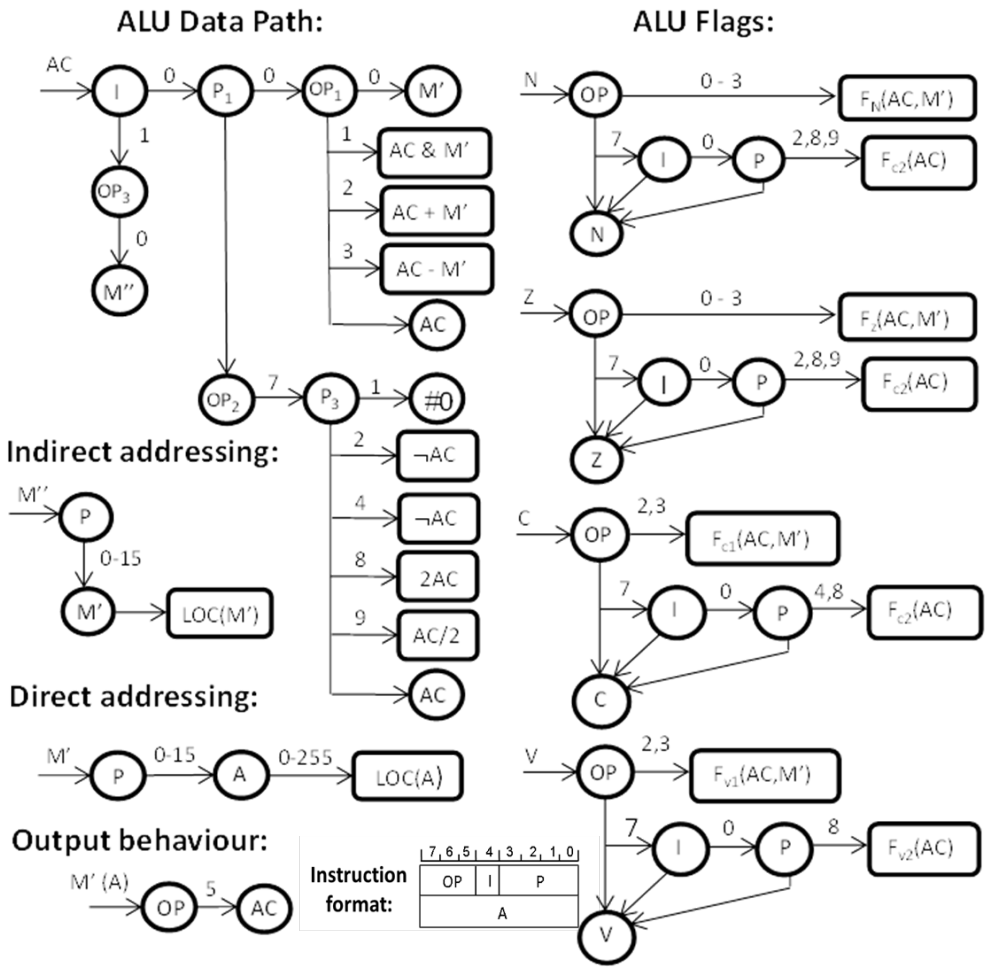

Figure 12. Parwan HLDD model (2).

In addition, formats for instructions and program counter are provided to ease mapping between Parwan HLDD model and Parwan instruction opcodes (see Table 3. 'Parwan instruction opcodes.'). Subscripts for internal nodes are used to indicate to a different path how this node is reached. Hence, on a logical level, $[Node]_x = [Node]_y$ (f.e $P_1 = P_3$). The numbers above the edges connecting nodes indicate to bits used in file 'memory.mem' and are written in decimal (not binary). For better understanding how paths along given HLDD-s are related to Parwan instructions, the following is advised:

1. Convert decimal numbers to binary;
2. Match binary numbers with respective Parwan instructions (Table 3 on page 24).

Each part has an input (the leftmost indicator), at least 1 internal node and 1 terminal node. To differentiate which opcode bits have logical high and which ones logical low

value, follow a path from activated terminal node to input of the respective graph. Knowing the sequence of when each path was activated, it is possible to derive the sequence of instructions used. Therefore it is logically possible to 'remap' the original test program.

It should be noted that the goal is to test Parwan components, not just mere instructions. Hence, before writing an assembly program, it is necessary to use HLDD to eliminate nodes that are only partially tested or not tested at all.

5.2. Example of path propagation on Parwan HLDD

Based on a simple example, certain paths on Parwan microprocessor HLDD are activated during execution of RUN.bat file. Step-by-step illustration of how nodes in HLDD are activated with some instructions is given in Figure 13, Figure 14, and Figure 15. Nodes that are fully tested are marked green, whereas nodes which are partially tested are coloured yellow.

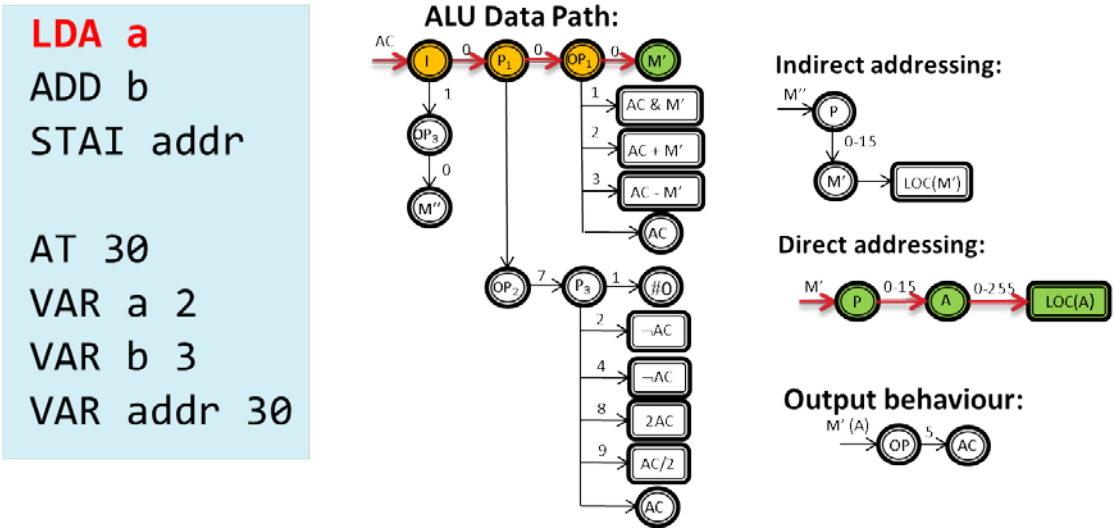

Figure 13. Path propagation on Parwan HLDD (1).

The first row of the program is a load instruction. ALU is loaded with value of a. This operation covers some nodes of the graph. Opcodes are as it follows:

- opcode = $0000\underline{0}000_2 = 0_{10}$;
- address = $00011110_2 = 30_{10}$.

For simplicity, the direct/indirect bit (B4) is underlined.

```
LDA a
ADD b
STAI addr

AT 30
VAR a 2
VAR b 3
VAR addr 30
```

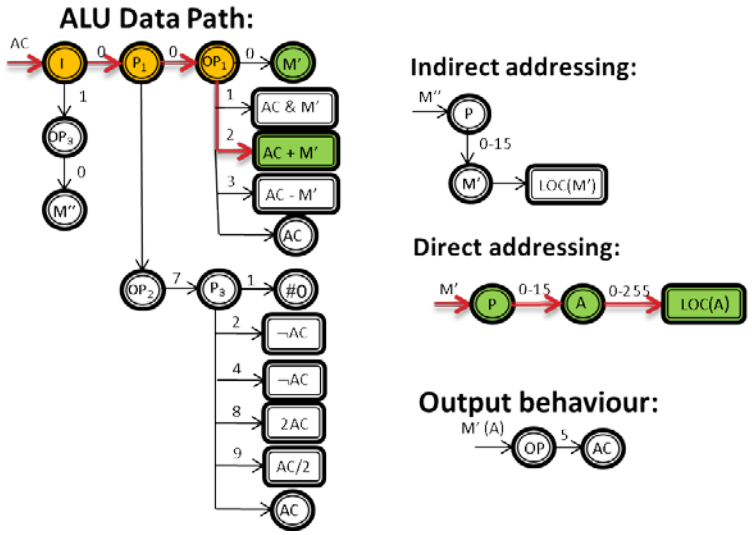

Figure 14. Path propagation on Parwan HLDD (2).

The next instruction is an adding operation. Although 1 more terminal node is activated, no internal node is yet fully tested. Opcodes:

- opcode = $010\underline{0}0000_2 = 64_{10}$;
- address = $00011111_2 = 31_{10}$.

```
LDA a
ADD b
STAI addr

AT 30
VAR a 2
VAR b 3
VAR addr 30
```

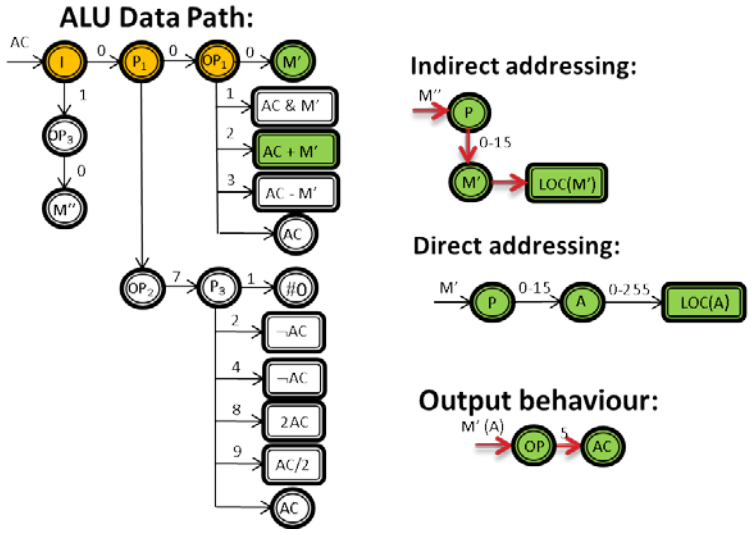

Figure 15. Path propagation on Parwan HLDD (3).

The last instruction is a store operation. It uses indirect addressing, which results in more nodes being covered. Opcodes:

- opcode: $10110000_2 = 176_{10}$;
- address: $00011110_2 = 30_{10}$.

It should be noted that during indirect mode, the address point to another address within the same page, which in turn points to and operand, which is also in the same page.

As it is seen, with a simple peace of program like that, the target to cover all nodes of the graphs remains unreachable. The only nodes that are fully tested in HLDD are terminals in ALU graph and nodes related to addressing.

5.3. Task1: Compliance with Parwan HLDD

Based on the Parwan HLDD, compose a corresponding assembler programme. Compile that assembler programme into memory.mem test pattern file and find a fault coverage for Parwan ALU-module. Your variant can be found in Table 5.

Depending on Your variant, follow these instructions:

1. Find terminal nodes on Parwan HLDD model to be tested;
2. Follow paths from terminals to root nodes;
3. Separate all nodes with a clear marking:
    - Fully tested;
    - Partially tested;
    - Not tested;
4. Write an assembly test program that goes through all nodes;
5. Find fault coverage for Your assembly program.

In addition: Instructions that use node P_i , must be in the same order as presented in the Table 5.

5.4. Task 2: Deriving instructions

In the following, it is necessary to derive an assembly program with limited conditions. The task is based on an arbitrary scenario.

Scenario: After viewing the result report, it turned out that we can only observe flags during our test, but luckily with the aid of the property of at-speed testing we were able to determine the sequence of when each flag was up. We know that step 1 happened earlier compared to step 2 and jumping was used only after load operation. We are also aware that no command with a full address scheme was used during even steps (2,4,6...) and shift operations were used only after store operations. It was previously agreed to avoid using add instruction, to use load operation only once and never repeat the same instruction in a row.

The sequence goes as it follows (only flags that altered their value are shown):

Step	1	2	3	4	5	6	7	8	9	10	11	12
Flags set	zn	N	zn	-	-	vczn	-	c	zn	zn	-	vczn

Derive an assembly program that would result in that sequence. Find a fault coverage for ALU and for the same additional Parwan module as in LAB1, using derived testprogram written in assembly.

5.5. Required information for LAB2

Before starting with Your tasks, a variant must be chosen from Table 5. As in task 1, it is dependent on Your student code.

Student code base: xxxWYZ

Table 5. Variant table for LAB2.

The value of Y	**The value of Z**	**The value of W**	**Value of node I**	**Value of node P**	**Value of node OP**
0 - 1		0	0	0,2,4	0,2,7
2 - 3		1	1	1,3,5	0,3,7
4 - 5		2	0	0,3,6	1,5,7
6 - 7		3	1	3,4,7	1,2,7

The value of Y	The value of Z	The value of W	Value of node I	Value of node P	Value of node OP
8 - 9		4	0	1,6,8	3,4,7
	0 - 1	5		2,5,6	2,6,7
	2 - 3	6		0,7,9	1,6,7
	4 - 5	7		3,8,9	5,6,7
	6 - 7	8		1,4,5	0,1,7
	8 - 9	9		3,5,9	3,5,7

Each node value is derived differently: the value of node P is dependent on only the value of W, the value of node I is dependent on only the value of Y and the value of node OP is dependent on either the value of Y or the value of Z.

Since 1 variant has 3 values for both OP and P nodes , then there are a total of 9 combinations. Considering that instructions, which use node P_i during the execution of RUN.bat must form the same order (from left to right) as presented in the Table 5, then the value P_i remains unchanged until all 3 OP nodes have been used once.

Report template for LAB can be found in Appendix 2

6. Conclusions

This work gives a description about the methods of using high-level decision diagrams with the instruction set of Parwan microprocessor . With the materials provided in this thesis, students have a chance to apply their knowledge about testing of digital systems and be a potential pioneers in developing easy-to-use and fairly accurate testing tools and to improve test programs.

The thesis is divided into three sections. The first part gives an insight into the theory about the usage of high-level decision diagrams as a basis for self-test generation for microprocessors. The second part covers the main aspects of Parwan microprocessor components and instructions that are relevant to testing approaches. The third topic is aimed at students to make them acquainted with the latest approaches regarding the usage of HLDDs. It also contributes to combining software and hardware for the purpose of effectively testing hardware systems.

References

[1] Jasnestki, A., Raik, J., Tšertov, A., Ubar, R. High-Level Decision Diagram based Self-Test Generation for Microprocessors, Tallinn, 2014

[2] Jasnetski, A., Ubar, R., Tšertov, A., Brik, M. Software-based self-test generation for microprocessors with high-level decision diagrams. - Proceedings of the Estonian Academy of Sciences, 2014, 63, 1, 48–61

[3] Navabi, Z, McGraw Hill Inc. Chapter 10 - CPU modeling and design, 1999 (http://www.ohio.edu/people/starzykj/network/Class/ee514/parwan.pdf) (22.04.2014)

[4] Jasnetski, A. Software-Based Selt-Test, Tallinn, 2014

[5] Kruus, H., Jasnetski, A., Tšertov, A., Ubar, R. Mixed-Level Fault Coverage Analysis of Tests for Microprocessors, Tallinn, 2014

[6] Jasnetski, A., Ubar, R., Tšertov, A., Kruus, H. Laboratory Framework TEAM for Investigating the Dependability Issues of Microprocessor Systems. - The 10th European Workshop on Microelectronics Education - EWME, 2014, 1-4

Appendix 1 - Report paper for LAB1

Variant (write 2 last numbers only): .....

Results:

1. Using base file, the fault coverage for ALU is: .....
2. Using base file, the fault coverage for additional Parwan ..... module is ..... .

Questions:

1. If the fault coverage in Your results would be the test result of a microprocessor in mass production, would that be high enough? Why?

2. Is it enough to test just 1 terminal node to determine whether all the non-terminal (internal) nodes on a respective path are tested? Why?

3. Name 3 fault classes.

4. In the given illustration Figure 16, 3 faults could be activated. What kind of faults F_i are activated? In which fault classes these belong to(refer to Appendix 3 for assistance)? Which fault is the root cause that propagates fault to other HLDD graphs?

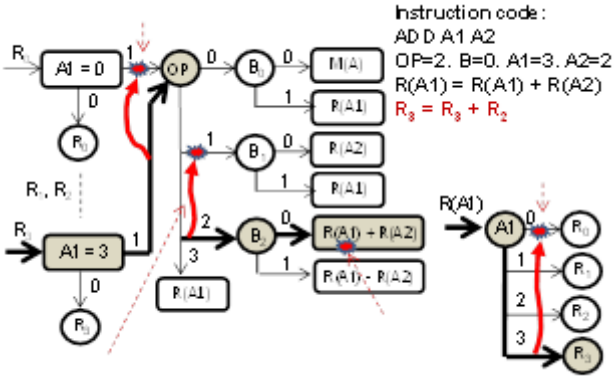

Figure 16. Example of HLDD faults .

Appendix 2 - Report paper for LAB2

Variant (write 2 last numbers only): .....

Results:

Fault coverage for ALU in task 1: .....

Fault coverage for ALU in task 2: .....

Questions:

1. Does changing the order of instructions change the fault simulation process? Does the initial fault coverage remain the same?
2. Is it wise to derive an assembly program by observing only 1 system component, given that arbitrary assumptions are correct? Why?
3. Is it wise to derive an assembly program by just knowing which terminals were activated on the HLDD graphs?
4. How many instructions is it possible to derive, if there are 5 terminal nodes that are coloured?

Appendix 3 - State of the art in MP modeling

Faults affecting the operation of microprocessor can be divided into the following classes:

- addressing faults affecting register decoding;
- addressing faults of instruction decoding and -sequencing functions;
- faults in the data-storage function (registers);
- faults in the data-transfer function (Bus);
- faults in the data-manipulation function (ALU blocks).

For multiplexers under a fault, for a given source address, any of the following may happen:

F1: no source is selected;
F2: a wrong source is selected;
F3: more than one source is selected and the multiplexer output is either a wired-AND or a wired-OR function of the sources, depending on the technology.

For demultiplexers under a fault, for a given destination address:

F4: no destination is selected;
F5: instead of, or in addition to the selected correct destination, one or more other destinations are selected.

An instruction *I* can be viewed as a sequence of *microinstructions*, where every microinstruction consists of a set of *micro-orders* which are executed in parallel. Micro-orders represent elementary data-transfer and data manipulation operations.

Addressing faults affecting the execution of an instruction may cause one or more of the following fault effects:

F6: one or more microorders not activated by the microinstructions of *I*;
F7: microorders are erroneously activated by the microinstructions of *I*;
F8: a different set of microinstructions is activated instead of, or in addition to.

The data storage facility is usually implemented as a memory. Under a fault any of the following may happen to the memory cell array:

F9: one or more cells are stuck at 0 or 1;

F10: one or more cells fail to make a $0 \rightarrow 1$ or $1 \rightarrow 0$ transitions;

F11: two or more pairs of cells are coupled; by this we mean a transition from *x* to *y* in one cell of the pair, say cell *i*, changes the state of the other cell, say *j*, from *x* to *y* or from *y* to *x*, where $x \{0,1\}$, and $y = \bar{x}$.

The data-transfer function implements all the data transfers along the buses between the registers and functional units of a microprocessor. For buses under a fault:

F12: one or more lines can be stuck at 0 or 1;

F13: one or more lines may form a wired-OR or wired-AND function due to shorts or spurious coupling;

F14: data manipulation faults.

In case of functional units for data processing, no specific F14 model has been proposed for microprocessors. It has been usually assumed that a complete test set for data manipulation faults can be derived for the functional units by some other techniques.

The disadvantage of the described approach lies within the big set of fault models too densely tailored to the hardware details, and as well the fact that only the special class of digital systems, microprocessors, are handled here. The fault classes defined are not suitable to be extended to cover a broader class of digital systems.

Appendix 4 - A uniform fault model based on HLDD

In the following, a uniform fault model based on HLDDs is modeled which targets the functional testing of nodes in the model. Each path in an HLDD describes the behaviour of the system in a specific mode of operation (working mode). The faults which may have effect on the behaviour of this working mode can be associated with nodes along the path. A fault in each node may cause incorrect exit from the path activated by a test which would mean activation of another (wrong) path in the HLDD terminating at a wrong terminal node.

From this point of view, the following abstract fault model for non-terminal nodes $m \in M_N$ with node variables $z(m)$ in HLDDs can be defined: the HLDD based fault model for micro-processors includes three fault classes:

D1:The output edge for $z(m) = v$ is broken; notation: $z(m)/\varnothing$; (similar to SAF $z/0$ for the line z);

D2:The output edge of a node m for $z(m) = v$, $v \in V(z(m))$ is always activated; notation: $z(m)/v$; (it is similar to the logic level stuck-at fault (SAF) $z/1$ for the line z);

D3:Instead of the given edge for $z(m) = v_i$, another edge v_j or a set of edges V_j is simultaneously activated; notation: $z(m)/(v_i \rightarrow V_j)$.

The Table 5 depicts the correspondence of the HLDD-based fault model to 14 microprocessor fault classes described above.

Table 6. HLDD microprocessor faults.

Microprocessor faults	**HLDD faults**	
F1, F4, F6	D1	Internal nodes
F3, F5, F8	D2	Internal nodes
F2, F3, F5, F7, F8	D3	Internal nodes
F9-F14	D3	Terminal nodes

The faults F1, F4 and F6 can be explained on DDs as the fault class D2 which describes missing activities as faulty behaviour (broken edge on the HLDD). F2 and F7 refer to an

erroneously activated operation compared to the expected one, which corresponds to the fault class D3. F3, F5 and F8 correspond both to D2 and D3. The faults F9 - F14 can be formally covered by the fault class D3 for terminal nodes, since they are strictly related only to the node, however the data needed for detecting these faults should be generated using the implementation details at the lower level. In this sense, the terminal nodes represent the interface between high- and low-abstraction level of hierarchy.

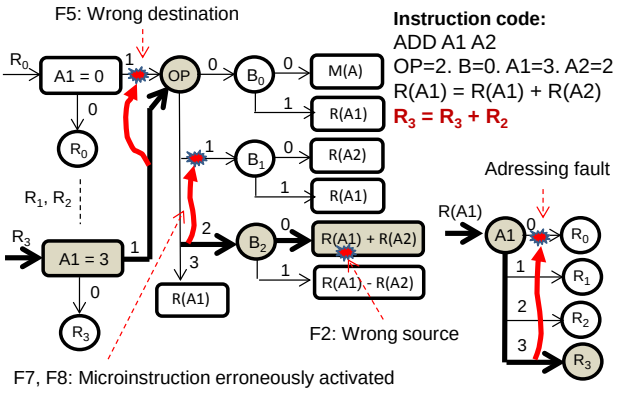

Figure 17. Illustration of different faults in HLDDs.

Consider in Figure 12 how different fault models described in Section 4.1 can be represented in a uniform way as the node faults on the HLDD model. Addressing fault is illustrated in the graph $G_{R(A1)}$. Instead of the edge 3 of the node *A*1, another edge 0 (or both edges simultaneously) are activated. This fault can propagate to other HLDDs of the model. For example, this fault can cause in the ALU graph either fault F2 (wrong source), or the fault F5 (wrong destination). The fault types F7 or F8 are illustrated by the fault of the node OP called „instead of the edge 2 the edge 1 is activated“. All these faults belong to the class D3 of the HLDD fault model.

Appendix 5 - HLDD-based conditional node fault model

The HLDD based conditional node fault model consists of two types of conditions (constraints):

1. activation of the fault of the node;
2. propagation of the fault signal through HLDD.

For non-terminal nodes $m \in M_N$, with $V(z(m))$:

1. propagation constraints: for all values $v \in V(z(m))$ of the node m, non-overlapping paths must be activated from the root node through m to non-coinciding terminal nodes $m_v \in M_{T,m}$, respectively, where $M_{T,m} \subseteq M_T$ is the subset of all terminal nodes reached from m;
2. activation constraints:

$$\forall v,j \in V(z(m)), i \neq j: [z(mv) \neq (z(mj) \neq \Psi T(MT,m) \neq 0 \neq 1], \quad (1)$$

where $\Psi_T(M_{T,m})$ is calculated by bit-wise OR/AND (depending on the technology) over the functions $z(m_v)$ in terminal nodes $m_v \in M_{T,m}$.

For terminal nodes $m \in M_T$:

1. propagation constraint lies in activating a single path from the root node to the tested node m;
2. activation constraints consist in the test patterns generated for testing the function $z(m)$.

Considering solution of the previously stated propagation constraints as a local test set $T(m)$ for testing the node m, the test set $T(m)$ consists of the dynamic and static parts: $T(m) = T_{ST}(m).T_{VAR}(m)$ where the dynamic part $T_{VAR}(m)$ means the set of values $V(z(m))$. Denoting by $R(m)$ the set of data registers to be initialized by the values which satisfy the activation constraints (1).

Appendix 6 - Properties of conformity and scanning tests

The main conception of the test generation using HLDDs can be characterized by the following targets:

1. Improved fault coverage and diagnostic resolution;

D4:Reduced probability of fault masking;

D5:The compactness of the whole test program due to its cycles-based organization.

The higher fault coverage is achieved by special targeting of the novel *hard-to-test* fault class called "*unintended actions*" which in traditional methods is neglected.

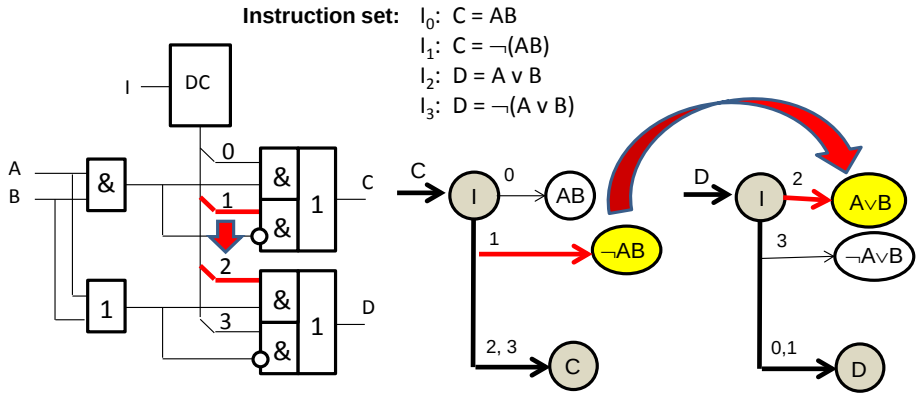

Figure 18. Illustration of a hard-to-test fault.

Consider in Figure 12 the four instructions I_0 - I_3, the *n*-bit gate-level implementation (registers are not shown), and the HLDD model for the registers C and D with their input logic. Assume, there is a gate level OR-type of short between the outputs 1 and 2 of the decoder, i.e. the instruction I_1 implies additional unintended action I_2 for the register D and changes its content. Normally, when testing I_1, we read only the register C, but we will not read the register D, because it is not involved in execution of I_1. In this way the fault will escape. Such a fault can be considered as an erroneous „unintended action" added to I_1. It would be difficult to catch such erroneous „supplements" when testing the instructions because the number of such cases may be huge.

However, the proposed approach does not target the testing of instructions (with all of their possible faulty modifications). Instead of that, targeting by testing the behaviour of

functional (control) variables, and exercising them separately for all of their values. For instance, in this example, we will not discover this fault when testing the variable C, however, we will detect the fault when testing D by the instruction I_2 to check if the register D holds its content.

The reduced probability of fault masking is achieved by targeting the proofs of correct behaviour of „small portions“ of the functionality of the microprocessor instead of having the goal to test the behaviour of instructions.

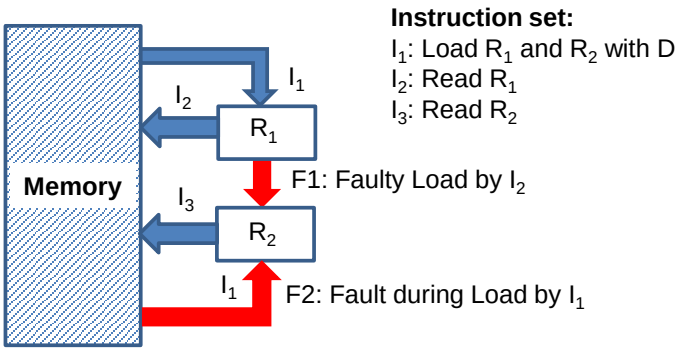

Figure 19. Illustration of fault masking.

Consider the case of high-level testing of the circuit in Figure 13 using three instructions I_1, I_2 and I_3. Assume there are two faults in the circuit, of which F1 means faulty overwriting of R_2 with the content of R_1 during the instruction I_2

Traditional test procedures which target instruction testing will not detect the fault:

Testing the instruction I_1:

- I_1: R_1= D, R_2=D* where D* is a faulty value;
- I_2: Read R_1 (correct reading), but R_2 = D (the faulty value D* is overwritten with the correct value of D);
- I_3: Read R_2 (correct reading, the fault F2 has escaped).

In the proposed approach, instead of testing the instruction, the functional variables R_1 and R_2 are to be tested separately:

Testing R_1:

- I_1: R_1=D, R_2=D* where D* is the faulty value;
- I_2: Read R_1 = D (correct reading).

Testing R_2:

- I_1: R_1=D, R_2=D* where D* is the faulty value;
- I_3: Read R_2 = D* (the fault is detected).

To reduce the probability of fault masking, the initialization and observation sequences are kept constant for the whole test of the variable under test. When recording the test results, always a single variable under test is targeted. In other case, when trying to observe more than one variables, each observation action may cause changes in the the state of the processor, which in its turn may activate other possible faults and cause fault masking.

Improved diagnosing opportunities result as well from the same strategy of targeting by test as „little portions“ of functionalities as possible.