STAR-DUST: Hierarchical Test of Embedded Processors by Self-Test Programs

Ulrich Bieker[1], Martin Kaibel[2], Peter Marwedel[1], Walter Geisselhardt[3]

[1] Dept. of Computer Science XII, University of Dortmund, D-44221 Dortmund, Germany
[2] Siemens, HL DC E EC, D-81541 München, Germany
[3] Dept. of Electr. Eng., University of Duisburg, D-47048 Duisburg, Germany

Abstract: This paper describes the hierarchical test-generation method STAR-DUST, using self-test program generator RESTART, test pattern generator DUST, fault simulator FAUST and SYNOPSYS logic synthesis tools. RESTART aims at supporting self-test of embedded processors. Its integration into the STAR-DUST environment allows test program generation for realistic fault assumptions and provides, for the first time, experimental data on the fault coverage that can be obtained for full processor models. Experimental data shows that fault masking is not a problem even though the considered processor has to perform result comparison and arithmetic operations in the same ALU.

1 Introduction

Test generation methods have traditionally focussed very much on gate level descriptions. Advantages of this approach include the availability of well-defined fault models and a good fault coverage. Disadvantages include the growing complexity of gate-level descriptions, resulting in large tool execution times and the lack of applicability of these techniques during early design phases.

As another extreme case, test generation methods based on purely behavioural circuit descriptions (for example, on instruction set descriptions) have a rather weak relation to possible physical defects.

Consequently, in our earlier work we have focussed on test generation methods based on circuit descriptions at an intermediate level. More precisely, we have chosen the RT structural level since physical defects can easily be represented at this level. In particular, we have focussed on exploiting programmability of processors such as DSP processors [Lee88], core processors [Adv95, LSI96], and ASIPs [Wou94, HD95]. Such processors are amenable to self-testing by running self-test programs on the processors themselves. It has been shown, that such self-test programs can be generated automatically from test specifications for each of their components [Krü91, BM95].

*contact author: P. Marwedel
address: University of Dortmund, Informatik 12, 44221 Dortmund, Germany
phone: +49 231 755 6111; home: +49 231 718772; fax: +49 231 755 6116
e-mail: marwedel@icd.de

So far, there has been neither a full integration of self-test program generation into standard test environments nor has there been any proof of the good fault coverage expected for this approach. Introducing such a full integration and demonstrating the achievable fault coverage are the main goals of this paper. We will show that self-test programs provide a good fault coverage not only for the data path (which was expected) but also for the controller. For the testing the ALU of processor having just a single ALU, the same ALU has to perform also the result comparison. This could potentially lead to fault masking effects. Our experimental data shows that fault masking can be avoided, even for the simple processor which we will consider.

In the appendix, we will also include an example demonstrating the limitations of the approach and draw conclusions for the type of modelling required for obtaining good fault coverages. In addition, we will mention how STAR-DUST can be used in a design procedure leading to a high fault coverage.

This paper is structured as follows: section 2 describes related work, section 3 explains the procedure used in STAR-DUST and section 4 contains an example. In section 5 we will present our results, and section 6 provides final remarks. In the appendix, we demonstrate why a purely behavioural modelling approach would not work.

2 Related Work

Test generation based on RT-level descriptions has been tried in a number of cases. For example, WHISTLE [RSS89] focusses on library-based designs. Application of such approaches still seems to be limited.

Some authors have tried to exploit the presence of complex resources in modern data paths for test generation and compression. For example, Kunzmann [Kun94] and Rajski [RT96] have published papers in this area.

Some authors have proposed to exploit programmability of digital processors. A very well-known approach for testing processors is that of Abraham et al. Abraham's [BA84] approach is based on instruction set models. This modelling paradigm is both a strength and a limitation of the approach. The strength is that is does not require information other than the information available in instruction set manuals. However, this information is not sufficient for obtaining a strong relation between the model and physical faults.

Consequently, Lee and Patel have proposed the ARTEST system [LP92a, LP92b], which is based on RT-level descriptions and is able of generating instructions resulting in the justification and propagation of test vectors. However, the generation of full self-test programs seems to be beyond the scope of ARTEST.

Being able to generate full self-test programs performing all the essential operations is the main strength of RESTART [BM95] and its predecessor MSST [Krü91]. RESTART (retargetable compilation of self-test programs using constraint logic programming) is the tool used for this paper. RESTART is the very first tool which has incorporated ideas employed in retargetable compilers (see [MG95] for survey).

3 Structure of STAR-DUST test generation process

The work described in this paper is directed at using RESTART for the automatic generation of a comprehensive processor test program based on realistic fault models. This requires the availability of both RT-level and gate-level models for the same processor. These models are used by the tools implementing the essential tasks of our current approach:

1. synthesis of gate-level descriptions for each of the RT-level components,
2. test pattern generation for each of the RT-level components from their gate-level descriptions,
3. generation of machine instructions implementing justification and response propagation for each of the test patterns,
4. fault simulation computing the fault coverage achieved by running self-test programs.

The tools and data formats currently used for these four tasks are shown in fig. 1.

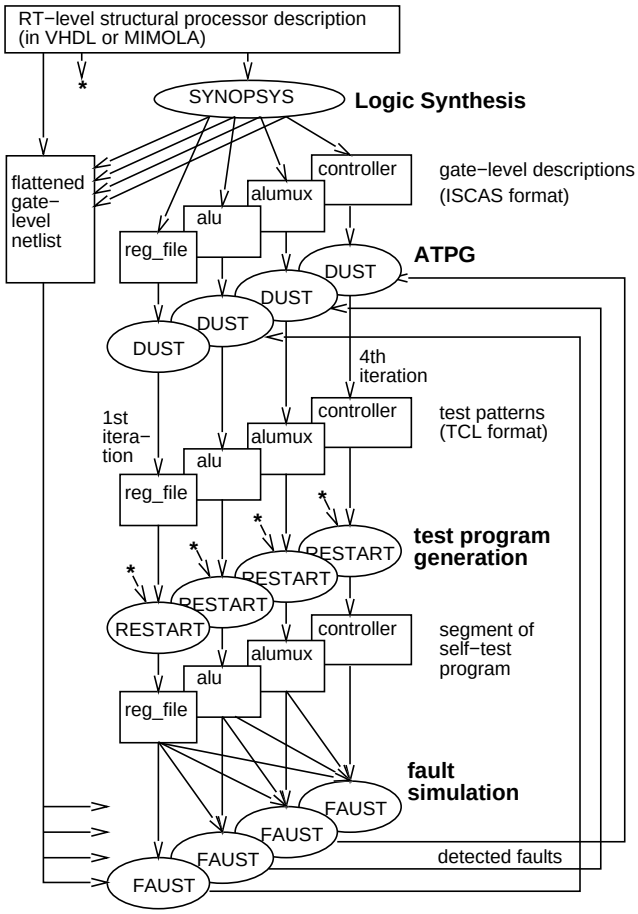


Figure 1: STAR-DUST test generation process

We start with an RT-level structural description of the processor under test. Input formats currently available for this purpose include VHDL [IEE92] or MIMOLA [BBH+94].

From this specification, we have to obtain equivalent gate level descriptions. For this purpose, we run the SYNOPSYS logic synthesis tools [Syn95] for each of the RT-level components. In fig. 1 it is assumed that our processor consists of just four components: register file `reg_file`, an ALU, an ALU multiplexer, and a controller.

Gate-level descriptions for each of the components are required for the tool implementing task 2. An overall flattened netlist for the entire processor is required for fault simulation (task 4). Gate-level descriptions generated by SYNOPSYS tools have to be translated into the ISCAS benchmark format required for ATPG and fault simulation tools.

Next, we select a heuristic order for processing RT-level components. (in figure 1, components drawn in front are processed first). Respecting that order, we do the following for each RT-level component:

- We use the DUisburg Sequential Test generator DUST [GK91a] for obtaining test pattern sets for the current component (e.g. the register file). DUST is an enhanced implementation of the BACK algorithm [Che88].

  Concerning a certain RT component, there may be some restrictions on the input vectors X(t) and state vectors Z(t) of the current component which can be justified. These restrictions may be imposed e.g., by encoding restrictions or the connections of the processor in which the RT component is embedded. For all but the first iteration of the loop, faults covered by already generated test program segments are also taken into account. If all faults are covered by such segments, the component is effectively skipped.

  Test pattern produced by DUST have to be translated into the test code language (TCL) accepted by the next tool. This translation is straightforward.

- Self-test programs are synthesized using RESTART. RESTART produces programs justifying test patterns and evaluating test responses. Evaluation of test responses is done by conditional jumps.

  Example: Consider a test pattern for an ALU. Suppose control code "10" activates the add operation of the alu, and assume that binary values "0111" and "0001" have been selected as test patterns by DUST. Then, a test for the add operation is specified in TCL by the following statement:

  TEST alu(0111,0001,10);

  The result should be 1000. RESTART generates code, which activates (i.e. justifies) the + operation and checks the result by a conditional jump:

```
IF 0111 + 0001 = 1000
THEN increment program counter
ELSE jump to error label;
```

  Every TEST statement is compiled to a conditional jump. If no error occurs, the program continues with the execution of the next instruction of the self-test program, otherwise a jump to an error procedure is performed. TCL allows the specification of all kinds of tests including memory test loops.

  The following requirements must be met by processors to be tested with code generated by RESTART:

1. The processor to be tested must be able to perform a comparison operation and to perform a conditional jump.
2. The processor must be programmable.
3. The program counter must be observable (on the other hand, a scan path is not necessary).
4. A single instruction cycle is required.

Details on the code generation technique in RESTART can be found in recent papers and books [BM95, Bie95b, Bie95a].

- Programs generated by RESTART are then used as initial stimuli for fault simulation at the gate-level. This way, the coverage of the programs produced by RESTART can be computed. Fault simulation is based on the gate-level stuck-at fault model.

  In our first approach we use FAUST (FAult Simulation Tool), a very efficient single pattern, single fault propagation method. FAUST has been extended to handle a ROM, assuming faults only on the data and address line of the ROM. The instruction memory itself is assumed to be fault free. To speed up the fault simulation process we will replace FAUST by PARIS (PARallel Iterative Simulator), a parallel pattern single fault propagation simulator [GM93, GK91b].

  Information about covered faults is exploited in the next cycle of the loop in order to reduce the size of the required test pattern set.

The loop terminates if all RT-level components have been considered.

RESTART, DUST and FAUST are the essential ingredients of this test generation process. Hence, we call this process STAR-DUST.

4 An Example

In order to demonstrate the test generation flow, let us use the CPU `SIMPLECPU` depicted in fig. 2. `SIMPLECPU` is a small programmable microprocessor consisting of eight modules. The `SIMPLECPU` controller (shaded area) consists of a program counter, an instruction memory, an incrementer, and a multiplexer.

A 16 x 4 register file, a 4-bit ALU and a second multiplexer make up the datapath. The register file and the program counter are synchronized by a clock. Control signals are denoted by "c" followed by an index range. The 8-bit program counter addresses the 256 x 22 bit instruction memory. The ALU computes a condition output that enables the controller multiplexer to perform conditional jumps.

The gate-level description of `SIMPLECPU` can be synthesized. The resulting circuit consists of 467 gates and 72 D-flip-flops.

Let us now explain the experimental procedure for generating a test program for `SIMPLECPU`. We start by developing the self-test program for testing the register file (1st iteration) and catenate self-test program segments for the other RT-components.

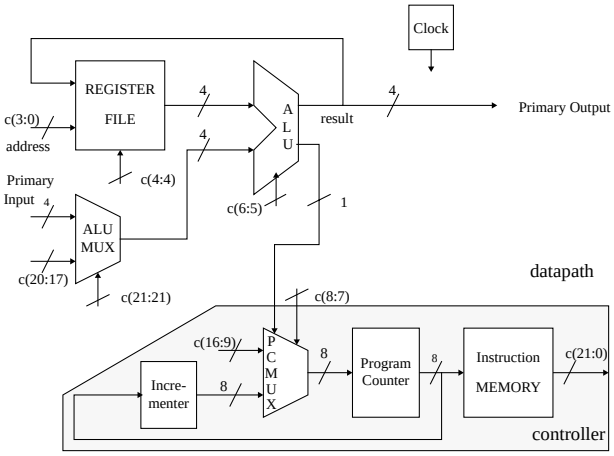


Figure 2: SIMPLECPU

1st iteration: We assume D-flip-flops have neither a set nor a reset input. Therefore, we can not detect faults which prevent the initialization of the D-flip-flops. DUST marks such faults as **untestable**. In total, `SIMPLECPU` has 6 untestable faults.

Using test frames [GK91a], we prune the search space of the test generator. Faults which can not be detected because of a test frame are called **functionally untestable**. We found 4 faults in the register file and 2 faults in the controller to be functionally untestable.

As example for a test frame we consider the register file, which is embedded in the complete circuit. If a certain register cell should be tested, the signal `wr_enable` must be '0' (read mode) and the ALU must perform a comparison operation in order to compare the output value of the register file with the expected value. The comparison is done by a subtract operation and a comparison with zero, i.e. (`outdata` - expected value) = '0'. Therefore, in the non-faulty case the output of the ALU which feeds the data input of the register file is restricted to the binary value 0000. To summarize, in the context of self-test programs generated by RESTART, `wr_enable` = '0' implies `indata` = 0000. This restriction concerning test patterns of the register file is specified as test frame.

2nd iteration: By applying the abovementioned 193 test patterns to the register file we achieve a fault coverage of 49.08% for `alu`. To detect the remaining faults of `alu`, DUST generates 17 test patterns. The resulting self-test code consists of 51 instructions, i.e., 3 instructions are necessary to apply an ALU test pattern and to check the test response. We illustrate such a test cycle by an example.

One of the 17 generated test patterns activates the + operation of the ALU with data "0000" and "1111". The according TCL statement is:

```
TEST alu(0000,1111,10);
```

The expected test response of the ALU output signal `result` is "1111". Table 1 shows the resulting code for above TCL statement.

Additionally, we expect the test pattern "1111" at every instruction cycle at the primary input. The first instruction loads register 0 with the binary value "0000" which has to be applied to the first input port of the ALU. Value "0000" directly originates from the instruction memory (signal `c`(21:17)). The second instruction applies the generated test pattern and activates the `+` operation of the ALU ("1111" originates from the primary input). Instruction 3 finally checks the result of

21	20:17	16:9	8:7	6:5	4	3:0	Comment
1	0000	X···X	00	01	1	0000	REG[0] := 0000
0	XXXX	X···X	00	10	1	0000	REG[0] := REG[0] + 1111
0	XXXX	11111111	10	11	0	0000	IF (REG[0] - 1111) = 0 THEN PC := PC+1 ELSE PC := 255

Table 1: Code for an `alu` test cycle

the ALU by activating the comparison operation of the ALU and performing a conditional jump. In case of a fault, the program jumps to the user defined error addres 255 (the program counter must be observable). Otherwise the self-test program continues with the execution of the next test statement. A fault in the comparison unit of the ALU is detected by different conditional jumps and therefore can be observed at the program counter.

An additional reset pin exists to initialize the program counter with 0. Therefore, the first instruction of the generated self-test program is located at address 0.

3rd iteration: Test generation for `alumux` is effectively skipped, because the test program segments generated for `reg_file` and `alu` already provide a 100 % fault coverage for this component.

4th iteration: Testing the controller is mainly implicitly done by the self-test program developed so far. Adding 6 (conditional and unconditional) jump instructions results in a fault coverage of 98.36% for the controller. Currently, these 6 TCL statements have to be added manually, but automation of this step can be added to RESTART.

5 Results

5.1 Results for `SIMPLECPU`

The entire self-test program for `SIMPLECPU` consists of 250 instructions and the achieved fault coverage for `SIMPLECPU` (including data path and controller) is 99.63%.

Table 2 gives information and results concerning the example processor `SIMPLECPU`.

RT-component	gates	DFF's	flt's	det	untest	f_untest	aborted	fault coverage	efficiency
`reg_file`	316	64	2256	2252	0	4	0	99.82%	100.00%
`alu`	70	0	436	436	0	0	0	100.00%	100.00%
`alumux`	13	0	76	76	0	0	0	100.00%	100.00%
`controller`	68	8	488	480	6	2	0	98.36%	100.00%
`SIMPLECPU`	467	72	3256	3244	6	6	0	99.63%	100.00%

Table 2: `SIMPLECPU` results

For every RT component and the entire processor, table 2 shows the number of gates, the number of D-flip-flops, the number of stuck-at faults, the number of detected faults, the number of untestable

faults, the number of functional untestable faults, the number of aborted faults, the fault coverage for the stuck-at fault model, and the efficiency.

To illustrate the abovementioned iterations consider table 3. It shows the number of generated test patterns and the resulting fault coverage for the different components iteration by iteration. Initially, we generated 193 test patterns for the register file. The resulting test code for these 193 patterns already detects e.g., 81.35% of the controller faults. The final self-test program consists of $193 + 3 * 17 + 6 = 250$ microinstructions. With the help of the 6 jump instructions of step 4, we detected 82 remaining faults of the controller.

	additional patterns	`reg_file`	`alu`	`alumux`	`controller`
1st iteration	193	99,82 %	49,08 %	100 %	81,35 %
2nd iteration	17	99,82 %	100 %	100 %	81,55 %
3rd iteration	0 (skipped)	99,82 %	100 %	100 %	81,55 %
4th iteration	6 branches	99,82 %	100 %	100 %	98,36 %

Table 3: Course of development of self-test program and fault coverage

The results show, that the self-test program approach can achieve high fault coverages for the data path as well as for the controller. Interestingly enough, fault masking can be avoided even for this simple processor.

	DUST	RESTART	FAUST
CPU seconds	4.12	93.56	5.77

Table 4: CPU times measured on a Sparc 20

Table 4 shows the CPU times for the different tools for SIMPLECPU. For DUST the sum of CPU times required to generate test patterns for all RT components is given. The hierachical test approach divides the complex test problem for the entire processor in several less complex subproblems. Therefore, we are able to generate a test within minutes.

For this example, RESTART is the tool consuming most the computing time. To some extend, this is caused by focussing on the functionality of RESTART, rather than on its speed. RESTART is implemented in a non-standard programming paradigm, called *constraint logic programming*, which might have caused some runtime penalty.

5.2 Results for Tanenbaum Example

As a slightly more complex example, we have studied an architecture described by Tanenbaum [Tan90]. For this example, the gate-level description contains 1936 gates and 296 D-type flip-flops. There are 6803 possible stuck-at faults, of which 38 are untestable. Using the STAR-DUST procedure, a fault coverage of 95.88 % is obtained, corresponding to an efficiency of 96.44 %. The resulting self-test program requires 246 instructions. The architecture allows up to 255 instructions. A higher fault coverage could have been achieved without the limit on the number of instructions.

For this example, automatic code generation for loops was not possible due to the limited support of loops in the particular architecture. Hence, a loop for testing the register file was manually specified leading to a total of 54 static instructions. All other instructions were synthesized from automatically generated TCL statements.

6 Final Remarks

6.1 Characteristics of the STAR-DUST approach

STAR-DUST meets a large number of different objectives:

- STAR-DUST is the first test generation process computing the fault coverage which can be achieved by self-test programs.

  Computed coverage is implicitly based on the assumption that the synthesized gate-level description is the one actually implemented.

- STAR-DUST is a hierarchical test generation process reducing the complexity of generating test patterns for the entire processor to that of generating test patterns for each of the components.

  Test generation proceeds at a high level of abstraction (using RESTART) while at the same time preserving the high fault coverage through gate level fault modelling.

- STAR-DUST reduces the length of the test program by considering faults covered by tests generated for one component during subsequent generation of test patterns.

- Errors during logic synthesis usually result in error reports during fault simulation. Hence, fault simulation is effectively used for validating the consistency of RT-level and gate-level models. STAR-DUST can be employed for model validation! In particular, logic synthesis results can be validated this way.

- Processor testability can be improved with the help of additional redesign cycles.

6.2 Future Work

Ongoing research is extending the work of the current paper into the following directions:

- Experimentation with larger processor models.
- Optimization of the sequence in which RT-components are considered.
- More sophisticated testing of the branching logic.
- More sophisticated fault models for the controller.
- More detailed control over hierarchy expansion/hierarchy preservation.
- Code size reduction through more efficient usage of loops.

- Tighter integration of the tools.
- Attempts to avoid the flat netlist currently required by FAUST.
- Analysis of the benefit of additional redesign cycles.
- Use of the approach for validating HDL descriptions.

Several of these directions are expected to be promising.

6.3 Conclusion

In this paper, we have described a hierarchical approach to testing, combining a high level of design abstraction with precise fault modelling and we have demonstrated that self-test programs can achieve a high coverage of gate-level stuck-at faults. This is the very first time, detailed information about the characteristics of this approach is available. Experimental data shows that fault masking can be avoided even for the very simple architecture that we considered.

Also, the current approach opens opportunities for testability improvements and model validation.

References

[Adv95] Advanced RISC Machines Ltd. ARM. web pages. *http://www.arm.com/*, 1995.

[BA84] D. Brahme and J. A. Abraham. Functional testing of microprocessors. *IEEE Trans. on Computers*, pages 475–485, 1984.

[BBH+94] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA language 4.1. *Universität Dortmund, Informatik 12, http://ls12-www.informatik.uni-dortmund.de*, 1994.

[Bie95a] U. Bieker. Retargetable compilation of self-test programs using constraint logic programming. *in: P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors, Kluwer Academic Publishers*, 1995.

[Bie95b] U. Bieker. *Retargierbare Compilierung von Selbsttestprogrammen digitaler Prozessoren mittels Constraint-logischer Programmierung (in German).* Shaker Verlag, Aachen, ISBN 3-8265-10925, 1995.

[BM95] U. Bieker and P. Marwedel. Retargetable self-test program generation using constraint logic programming. *32nd Design Automation Conference*, 1995.

[Che88] W. Cheng. The back algorithm for sequential test generation. *Proceedings International Conference on Computer Design*, pages 66–69, 1988.

[GK91a] N. Gouders and R. Kaibel. Advanced techniques for sequential test generation. *Proc. ETC*, pages 293–300, 1991.

[GK91b] N. Gouders and R. Kaibel. PARIS: A parallel pattern fault simulator for synchronous sequential circuits. *Proc. ICCD*, pages 542–545, 1991.

[GM93] W. Geisselhardt and M. Mojtahedi. New methods for parallel pattern fast fault simulation for synchronous sequential circuits. *ICCAD*, 1993.

[HD95] I.-J. Huang and A. Despain. Synthesis of application specific instruction sets. *IEEE Trans. on CAD*, pages 663–675, 1995.

[IEE92] Design Automation Standards Subcommittee of the IEEE. Draft standard VHDL language reference manual. *IEEE Standards Department, 1992*, 1992.

[Krü91] G. Krüger. A tool for hierarchical test generation. *IEEE Trans. on CAD, Vol. 10*, pages 519–524, 1991.

[Kun94] A. Kunzmann. Test pattern generation hardware motivated by pseudo-exhaustive test techniques. *EURO-DAC*, pages 240–245, 1994.

[Lee88] E. Lee. Programmable DSP architectures, parts i and ii. *IEEE ASSP Magazine*, Oct. 1988 & Jan. 1989, 1988.

[LP92a] J. Lee and J.H. Patel. Hierarchical test generation under intensive global functional constraints. *Proc. 29th Design Automation Conf.*, pages 261–266, 1992.

[LP92b] J. Lee and J.H. Patel. An instruction sequence assembling methodology for testing microprocessors. *Proc. of the Intern. Test Conference*, pages 49–58, 1992.

[LSI96] LSI Logic Inc. web pages. *http://www.lsil.com/products/unit5_5.html*, 1996.

[MG95] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[RSS89] R. Renous, G. Silberman, and I. Spillinger. WHISTLE: a workbench for test program development of library-based designs. *IEEE Computers*, pages 27–41, 1989.

[RT96] J. Rajski and J. Tyszer. Multiplicative window generators of pseudo-random test vectors. *European Design & Test Conference (ED&TC)*, 1996.

[Syn95] Inc Synopsys. Synopsys version v3.3a. *700 East Middlefield Road. Mountain View, CA 94043-4033, USA*, 1995.

[Tan90] A.S. Tanenbaum. Structured computer organization. *Prentice Hall, (3rd edition)*, pages 161–186, 1990.

[Wou94] R. Woudsma. EPICS, a flexible approach to embedded DSP cores. *Int. Conf. on Signal Processing and Applications and Technology*, 1994.

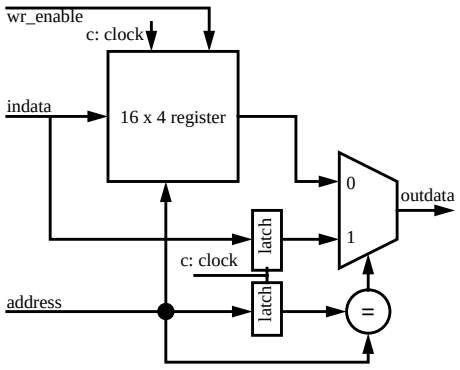
Appendix: Why a behavioural test would not work

As an example for the insufficiency of a behavioural test, we consider the register file of our example. The behavior of the register file is described as follows:

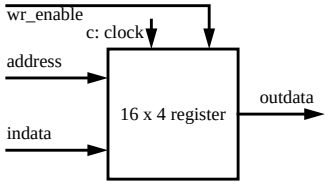
```
ENTITY RegFile IS
 PORT (address  : IN  UNSIGNED (3 Downto 0);
       indata   : IN  UNSIGNED (3 Downto 0);
       wr_enable: IN  UNSIGNED (0 Downto 0);
```

```
       c        : IN  bit;          -- clock
       outdata  : OUT UNSIGNED (3 Downto 0));
END RegFile;

ARCHITECTURE behavior OF RegFile IS

TYPE rfile IS ARRAY (15 Downto 0)
        OF UNSIGNED (3 Downto 0);
SIGNAL register_file: rfile;

BEGIN  --  16 x 4 register file
 p: PROCESS (address, indata, wr_enable, c)

 BEGIN  --  PROCESS p

  IF ( (c = '1') and (not c'stable) ) THEN
   IF (wr_enable = CONV_UNSIGNED(1,1) ) THEN
    register_file(CONV_INTEGER(address)) <= indata;
   END IF;
  END IF;

 END PROCESS p;
 outdata <= register_file(CONV_INTEGER(address));
END behavior;
```

This behavior can be implemented by RT structures as shown in fig. 3a) and fig. 3b).



a) RT structure with bypass register for the register file



b) simpleRT structure for the register file

Figure 3: Register File

Both structures have the same functionality. However, because of the bypass register the register

file in fig. 3a) is much faster than the simple structure of fig. 3b). Additional buffer registers store `indata` and `address` in fig. 3a). The multiplexer is controlled by a comparator and selects from the latch for `indata` and the output of the 16 x 4 register file. Every latch comes with 4 additional D-flip-flops. Of course, testability of the structure shown in fig. 3b) is much better than the testability of the structure shown in fig. 3a).

In our first approach a behavioural standard test loads, reads and checks every register of fig. 3a) with two test patterns: "1010" and "0101". This is specified by the following TCL statement:

```
FOR address := 0 TO 15 DO TEST register_file := 1010;
```

The generated code has 64 instructions and requires 64 test patterns. The resulting fault coverage is disappointing: Only 13.25%.

The reason for the low fault coverage is quite simple: A pattern which is written into a register cell and read immediately in the next instruction cycle is buffered in the bypass register. A behavioural test writes and checks only the buffer register. Possible solutions to this problem include:

1. a redesign of the register file, possibly leading to a larger cycle time,
2. expanding `reg_file` down to the next level of the hierarchy before attempting test generation,
3. manually specifying a TCL program interleaving the tests of the different locations within `reg_file`. applying a set of test patterns generated by an ATPG tool at the gate level.

To achieve a high fault coverage, DUST has been used in the actual experiments to generate test patterns for the gate-level model synthesized by SYNOPSYS. This corresponds to implicitly using solution 2. DUST generates a sequence of 193 test patterns for the embedded register file. This compares with 755 test patterns generated by SYNOPSYS for the same hardware. The 193 test patterns are compiled by RESTART and the resulting fault coverage of the code is 99.82% for the register file.