Software-based Self-Test Generation for Microprocessors with High-Level Decision Diagrams

Raimund Ubar, Anton Tsertov, Artjom Jasnetski, Marina Brik

Tallinn University of Technology
Department of Computer Engineering
Outline

✓ Software-based Self-Test
✓ High-level modeling of microprocessors
✓ Test generation on behavioral level
✓ Case study
✓ Experimental results
Main focus of the SBST research is on the quality of test

Problem:
- Scalability of the Gate-Level and RT Level based SBST methods is low

Possible solution:
- Moving towards behavioral level
What’s new?

- We are proposing the methodology for processor modeling on behavioral level
- The processor can be modeled using it’s instruction set only
- Automatic generation of SBST programs from high level models
- High-Level Decision Diagrams
- Testing targets are functional variables instead of instructions
High-Level modeling of microprocessors

- Microprocessor is provided at behavioral level (instruction set)

- based on the instruction set, the High-Level Decision Diagrams can be built

- **Control part** is represented by **non-terminal** nodes

- **Data path** is represented by **terminal** nodes

### Instruction set:

- **I1**: MOV A,M, A ↵ IN
- **I2**: MOV R,A, R ↵ A
- **I3**: MOV M,R, OUT ↵ R
- **I4**: MOV M,A, OUT ↵ A
- **I5**: MOV R,M, R ↵ IN
- **I6**: ADD R, A ↵ A + R
- **I7**: ORA R, A ↵ A ∨ R
- **I8**: ANA R, A ↵ A ∧ R
- **I9**: SUB R, R ↵ R - 1
- **I10**: MOV C,R, C ↵ R
- **I11**: CMA R,C, R ↵ C
- **I12**: JMP PC, C, IF C=0 THEN \( PC = IN \)
Test generation at behavioral level

- The test program is generated from HLDD with symbolic test patterns

- Two types of test:
  - **Conformity** tests – for **control part**
  - **Scanning** tests – for **data path**

- Test patterns are generated separately
Generation of conformity tests

✓ **Conformity** tests are testing **control part**

✓ Test patterns are generated by solving constraints like:
  - $f_1 \neq f_2 \neq \ldots \neq f_n$, where $f$ is a function of terminal node

✓ Activation of **non-terminal nodes**, representing the instructions
Generation of conformity tests

Benefit:

- Testing all the functions of A with the same LOAD and STORE conditions will **reduce the probability of fault masking**

For $D=1,n$

- $I_5$: Load $R = \text{IN}(1)$
- $I_1$: Load $A = \text{IN}(2)$
- $I_D$: $D$
- $I_4$: Store $A$

End For

Test data

Signature

\[ \begin{align*}
I & \quad \text{IN} \\
6 & \quad A + R \\
7 & \quad A \lor R \\
8 & \quad A \land R
\end{align*} \]
Generation of scanning test

✓ **Scanning** tests are testing **data path**

✓ Test patterns are generated using gate-level ATPG

✓ Activation of **terminal nodes**, representing the data
Generation of scanning tests

Benefit:

- Signature **improves the fault diagnosis** by holding the trace of each test step

For \( j=1,n \):

1. \( l_5: \) Load \( R = \text{IN1}(j) \)
2. \( l_1: \) Load \( A = \text{IN2}(j) \)
3. \( l_6: \) ADD \( A = A + R \)
4. \( l_4: \) Store \( A \)

End For
Test program generation

- Automatically composed from the generated HLDD graphs
- Test program has a **compact cyclic representation**

```
FOR all instructions
  FOR all test patterns
    INIT data
    Execute test
    Store signature
  END FOR
END FOR
```
Reducing the Probability of Fault Masking

- The functions are tested with the same LOAD and STORE conditions
- „Small portions“ of the functionality (variables) are the test targets, instead of instructions

**Instruction set:**
- I₁: Load R₁ and R₂ with D
- I₂: Read R₁
- I₃: Read R₂

**Instruction driven testing:**

Testing of the instruction I₁:
- I₁: R₁ = D, R₂ = D*  *** D* is the faulty value
- I₂: Read R₁
- I₃: Read R₂

Testing of R₁:
- I₁: R₁ = D, R₂ = D*  *** Initialization
- I₂: Read R₁ = D  *** Correct reading

Testing of R₂:
- I₁: R₁ = D, R₂ = D*  *** Initialization
- I₂: Read R₂ = D*  *** The fault F2 is detected

**Functionality driven testing:**
From MP Instruction Set to RTL Structure

<table>
<thead>
<tr>
<th>OP</th>
<th>B</th>
<th>Semantic</th>
<th>RT level operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>READ memory</td>
<td>$R(A1) = M(A)$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>WRITE memory</td>
<td>$M(A) = R(A2)$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Transfer</td>
<td>$R(A1) = R(A2)$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Complement</td>
<td>$R(A1) = \overline{R(A2)}$</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Addition</td>
<td>$R(A1) = R(A1) + R(A2)$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Subtraction</td>
<td>$R(A1) = R(A1) - R(A2)$</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>Jump</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Conditional jump</td>
<td>IF C=1, THEN $PC = A$, ELSE $PC = PC + 2$</td>
</tr>
</tbody>
</table>

Instruction code:
ADD A1 A2
OP=2. B=0. A1=3. A2=2
$R_3 = R_3 + R_2$
$PC = PC+1$
Case study

Microprocessor under test – PARWAN

- 8-bit data and address bus
- 16 instructions
- No pipeline

Instruction Set
LDA, ADD, SUB, AND, CLA, SMA, BRA, NOP

Program
label start
nopp
lda a
...

Memory file
0. 10101010
2. 01010101
...

Simulation

Gate-level fault simulation

Test vectors

Gate level description

Statistics

Fault coverage report

Behavioral Level

Register Transfer Level

RTL description

Gate Level

Model

HLDDs

0

1
Parwan: HLDD Model

**ALU Data Path:**

- **AC** → **I** → **P_1** → **OP_1** → **M'** → **OP_2** → **P_3** → **#0**
- **OP_3** → **M''**

- **Indirect addressing:**
  - **M''** → **P**
  - **0-15** → **M'** → **LOC(M')**

- **Direct addressing:**
  - **M'** → **P**
  - **0-15** → **A**
  - **0-255** → **LOC(A)**

- **Output behaviour:**
  - **M' (A)** → **OP** → **5** → **AC**

**Instruction format:**

<table>
<thead>
<tr>
<th>OP</th>
<th>I</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

**ALU Flags:**

- **N** → **OP** → **I** → **P** → **0 - 3** → **F_N(AC,M')**
- **Z** → **OP** → **I** → **P** → **0 - 3** → **F_Z(AC,M')**
- **C** → **OP** → **I** → **P** → **2,3** → **F_C(AC,M')**
- **V** → **OP** → **I** → **P** → **2,3** → **F_V(AC,M')**

**Instruction addressing:**

- **OP, I, P** → **PC_P** → **0-15** → **PC_A** → **0-255** → **LOC(PC_A)**

**Next memory page calculation:**

- **PC_P** → **OP_1** → **4** → **P_1**

**Next PC Offset calculation:**

- **PC_A** → **OP_3** → **7** → **P_2**
- **0-3, 5** → **PC_A + 2**
- **0-15** → **PC_A** → **4, 6** → **A_1**
- **2** → **Z** → **0** → **1**
- **4** → **C** → **0** → **1**
- **8** → **V** → **0** → **1**
- **PC_A + 2**
Test Program for Parwan Microprocessor

FOR VAR1 = 0, 1, 2, 4, 8, 9  (For all single byte ALU instructions)
FOR VAR2 = 0, 2, 4,… N    (For all data operands)
FOR VAR3 = 0, 2, 4, 8      (For all branch operations)

k) LDA, VAR2              (Data initialization)
k+2) l=0, P=0, OP = VAR1  (ALU Test, Flag initialization)
k+3) l=1, P=VAR3, OP=7    (Branch Test)
k+4) m                   (Jump address for fixing AC1 = AC)
k+5) ADD, CONST           (Fixing AC2 ≠ AC1)
k+7) ADD, LOC (REF)       (Signature calculated: REF = REF + AC1)
k+9) STA, LOC (REF)       (Finish: Signature updated)

END VAR3
END VAR2
END VAR1

m) ADD, LOC (REF)         (Signature calculated: REF = REF + AC2)
m+2) JMP, k+9
# Experimental results

<table>
<thead>
<tr>
<th>Module</th>
<th>Gate Level Stuck-at Faults</th>
<th>Fault Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Tested</td>
</tr>
<tr>
<td>AC</td>
<td>156</td>
<td>137</td>
</tr>
<tr>
<td>IR</td>
<td>228</td>
<td>161</td>
</tr>
<tr>
<td>PC</td>
<td>590</td>
<td>560</td>
</tr>
<tr>
<td>MAR</td>
<td>342</td>
<td>242</td>
</tr>
<tr>
<td>SR</td>
<td>130</td>
<td>99</td>
</tr>
<tr>
<td>ALU</td>
<td>962</td>
<td>939</td>
</tr>
<tr>
<td>SHU</td>
<td>310</td>
<td>310</td>
</tr>
<tr>
<td>Total</td>
<td>2718</td>
<td>2446</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>569</td>
<td>189</td>
<td>779</td>
</tr>
<tr>
<td>Data (Bytes)</td>
<td>1214</td>
<td>517</td>
<td>132</td>
</tr>
</tbody>
</table>

Conclusion

- Methodology for processor modeling on behavioral level
- Automatic generation of SBST programs from HLDDs
  - Reduced probability of fault masking
  - Better diagnostic opportunities
  - Compactness of the test representation
- Test length overhead
Acknowledgement

- EU FP7 STREP project BASTION
- Testonica Lab
- Estonian ICT project FUSETEST
- Estonian SF grants 8478, 9429
- Supported by the Tiger University Program of the Information Technology Foundation for Education

www.testonica.com

19