Functional test of processor-based systems

Matteo Sonza Reorda

Politecnico di Torino
Dipartimento di Automatica e Informatica
Summary

- Introduction to test
- Functional test
- Functional test of processors
- Functional test of peripheral components
- In-field test
- Conclusions
Summary

- *Introduction to test*
- Functional test
- Functional test of processors
- Functional test of peripheral components
- In-field test
- Conclusions
Hardware defects

- Electronic products are subject to hardware *defects*
- Hardware defects may arise
  - During the manufacturing process
  - During the operational life
- A defect may produce a *failure* (or misbehavior)
Test

- Test is the process of identifying faulty products, i.e., products affected by a fault or defect.

- Test is typically performed by:
  - Suitably stimulating the product
  - Observing its output behavior, looking for possible deviation with respect to the expected one.
End-of-manufacturing test

- It is typically performed by the manufacturer (of the device, board, or system)

- In this case the test
  - exploits the complete knowledge about the Unit Under Test (UUT)
  - Has full control of the UUT through the tester
Fault models

- In principle, a good test should be able to detect all possible defects
- But enumerating them is practically impossible
- Hence, *fault models* are adopted
- Example of fault model for logic blocks
  - Stuck-at
The stuck-at fault model

Circuit under test

This line is stuck-at 0 or stuck-at 1
The stuck-at fault model

Circuit under test

In order to detect the stuck-at-0 fault on this line we need to
- Force the line to 1 (creating a difference)
- Propagate the difference to an output.
Fault coverage

- Measuring the quality of a test (i.e., of an input sequence) can be done by computing the *Fault Coverage* (FC):

\[
FC = \text{percentage of faults detected by the input sequence}
\]
Importance of test

- The percentage of faulty products coming out of the production line may be quite high.
Importance of testing

- The percentage of faulty products coming out of the production line may be quite high.

\[
\text{Yield} = \text{Percentage of good products wrt manufactured products}
\]
Importance of testing

- The percentage of faulty products coming out of the production line may be quite high.

In this phase the manufacturing process is profitable.
The percentage of faulty products coming out of the production line may be quite high. In this phase it is NOT...
Importance of test

- Test is crucial to
  - Ensure the quality of the delivered products (*volume testing*)
  - Support the tuning of the manufacturing process so that yield is quickly increased

- Test may be VERY expensive
  - Mainly because of the cost of the *tester*
Importance of test

- Test is crucial to
  - Ensure the quality of the delivered products (*volume testing*)
  - Support the tuning of the manufacturing process so that yield is quickly increased
- Test may be VERY expensive
  - Mainly because of the cost of the *tester*

In some cases, the cost for test is higher than the one for manufacturing
Testers

- Testers (or Automatic Test Equipment, ATEs) are often expensive systems because
  - They must apply a high number of high speed values with high accuracy and precision
  - They must observe a high number of high speed values with high accuracy and precision
  - They must contact many signals
  - They must store a lot of values
  - ...
Design for Testability

- Sometimes it is convenient to suitably modify the design of a circuit to make the test easier (i.e., cheaper)

- Examples
  - Scan test
  - Built-In Self-Test (BIST)
Scan test

- Generating effective test sequences can be easily done for combinational circuits, while for sequential circuits it is practically unfeasible

\[ \downarrow \]

Designers modify their circuits in such a way that during the test phase they turn into pseudo-combinational circuits (scan test)
Normal Mode

Combinational logic

Inputs

Outputs
Testing memories requires stimulating them with long sequences of read/write operations; this is impossible for embedded memories.

Embedded memories are modified in such a way that during test:
- they are functionally disconnected from the rest of the circuit
- they are controlled/observed by a suitable circuitry, which generates tests and observe results
Embedded memory test
Embedded memory test

SoC

Embedded memory

SoC

Test Circuitry

Embedded memory
The adoption of DfT techniques can allow:

- Increasing the test *quality*
  - by allowing the detection of a higher percentage of defects
- Reducing test *cost*
  - by allowing the usage of cheaper testers
- Reducing test *times*
  - By allowing on-chip at-speed test
State of the art of DfT

- Practically all digital VLSI devices today adopt some DfT technique

- Limitations
  - DfT is expensive (in terms of silicon area)
  - DfT is often not documented by the silicon manufacturer ⇒ it cannot be used by others
  - DfT reconfigures the circuit for test; hence, some defects may escape DfT solutions
Test of SoCs

- Traditionally, end-of-manufacturing test is performed by the manufacturer, that owns full details about the internal structure of the system (and possible DfT structures)
  - Example: Intel and AMD test their processors fully knowing their design and implementation
- However, the SoC design paradigm heavily changed the situation
  - Sometimes the manufacturer must test a device including cores which have been designed by third parties
  - Hence, the manufacturer does not always fully know the internal structure of the device under test
In-field test

- During the operational life new defects may arise, e.g., due to *aging*
- When the product is used in a safety- or mission-critical application, it is crucial to detect these defects as early as possible, before they may create serious consequences
In-field test

- It can be performed
  - At the power-on
  - During idle periods
  - Periodically
  - Concurrently
  - ...

In-field test characteristics

- In-field test is rather different than end-of-manufacturing test
  - Up to a system company
  - No tester
  - Limited accessibility/observability
  - Duration is a strong constraint
  - Often without DfT support
Incoming Inspection

- It is typically performed by system companies wishing to assess the quality of the components they are going to use.
- This step is often required by standards (e.g., DO-254 for avionics and ISO 26262 for automotive) if the system is used for a safety-critical application.
- The system company does not have information about the internal architecture and implementation of each device (*black box*)
Board test

- It is composed of different steps
  - Bare board test
  - Test of components
  - Test of the populated board
- It uses completely different metrics
- It changes significantly from one product to another
- It is much less automated than IC test
Summary

- Introduction to test
- *Functional test*
- Functional test of processors
- Functional test of peripheral components
- In-field test
- Conclusions
Functional test

Functional test can be defined in two different ways:

Definition 1
a test performed acting on the functional inputs and observing the functional outputs, only, without resorting to any kind of Design for Testability
Functional test

- Functional test can be defined in two different ways:
  - Definition 1
    - a test performed acting on the functional inputs and observing the functional outputs, only, without resorting to any kind of Design for Testability

Functional test is thus opposed to DfT
- NO Scan test
- NO BIST
Functional test

- Functional test can be defined in two different ways:
  - Definition 1
    a test performed acting on the functional inputs and observing the functional outputs, only, without resorting to any kind of Design for Testability
  - Definition 2
    a test developed on the basis of the functional information about the module under test, only: therefore, it aims at testing the functions rather than the faults (*black box testing*)
Functional test can be defined in two different ways:

- Definition 1
  a test performed acting on the functional inputs and observing the functional outputs, only, without resorting to any kind of Design for Testability.

- Definition 2
  a test developed on the basis of the functional information about the module under test, only: therefore, it aims at testing the functions rather than the faults.

Functional test is now the opposite of structural test.
Functional test

- Def. 1 relates to *how the test is applied*
- Def. 2 relates to *how the test is generated*
- A test may also match both definitions
- Example: a device is tested using some stimuli
  - generated from functional information, only, and
  - applied without resorting to any DfT feature
Possible scenarios

- SoC end-of-manufacturing test
- SoC in-field test
- Processor Incoming Inspection
- PCB functional test
Scenario #1

- *SoC end-of-manufacturing test*
  - Functional test is used to detect defects which are not detected by other methods
  - Specially developed test programs are used
  - A tester is used (full control of inputs and outputs)
  - Fault coverage can be computed (the netlist is available)
Scenario #2

- **SoC in-field test**
- Functional test is sometimes used because it is the only feasible solution
  - No DfT
  - No tester
  - Limited resources
  - Flexible
- Specially developed test programs
- Fault coverage can be computed
Scenario #3

- **Processor (MCU) Incoming Inspection**
- Functional test is the only feasible solution
- Simple test programs are used (e.g., activating all instructions)
- Which metric to assess the test quality?
Scenario #4

- PCB functional test
- PCB test includes different steps
  - Bare board testing
  - Component test
  - Populated board test
- Functional test is often the final step
- Suitable applications are often used
- Metrics?
Summary

- Introduction to test
- Functional test
  - *Functional test of processors*
  - Functional test of peripheral components
- In-field test
- Conclusions
Functional test of processors

- It is used in different scenarios
  - End-of-manufacturing of processors
  - End-of-manufacturing of SoCs
  - End of manufacturing of PCBs
  - Incoming inspection of processors
  - In-field test of PCBs
  - ...

45
Functional test of processors

- A functional test for a processor is usually based on
  - Uploading some test code in an accessible memory
  - Forcing the processor to execute the code
  - Checking the produced results (e.g., in memory)
- This approach is sometimes denoted as *Software-Based Self-Test (SBST)*

Psarakis et al., D&T, 2010
Test Protocol

- Upload the Test Program in the code memory (or cache)
- Force the processor to execute the Test Program
- Check the results written in memory
How does it work?

- The test program and data are allocated in accessible memories.

```
LD R1, var_a
LD R2, var_b
ADD R3, R1, R2
SD var_c, R3
```

<table>
<thead>
<tr>
<th>Program Memory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1, var_a</td>
<td></td>
</tr>
<tr>
<td>LD R2, var_b</td>
<td></td>
</tr>
<tr>
<td>ADD R3, R1, R2</td>
<td></td>
</tr>
<tr>
<td>SD var_c, R3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Memory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>var_a</td>
<td>0010</td>
</tr>
<tr>
<td>var_b</td>
<td>0001</td>
</tr>
<tr>
<td>var_c</td>
<td>0000</td>
</tr>
</tbody>
</table>
How does it work?

- The test program is executed and the obtained results are collected.
How does it work?

- Test responses are externally monitored.
How does it work?

- The test program is executed and the obtained results are collected.
How does it work?

- Test responses are externally monitored
SBST application flow

1. SBST program upload
2. SBST program launch
3. SBST program execution monitoring
3a. Test results download
3b. Signature recording
3c. SBST program execution
4. [time]
SBST application procedure

- Some key aspects have to be considered:
  - Code/Data memories availability and upload time
  - Test procedure launching method
  - Monitoring capabilities
  - Test results storage and download
The test program is run by the processor core, fetching instructions from suitable/reserved memory spaces:

- **RAM cores**
  - Test programs are uploaded every time they have to be executed
  - Suitable for final tests at the end of manufacturing flow

- **Flash cores**
  - Test programs are uploaded just once
  - They are run any time it is required
  - Suitable to implement a reliability framework
Test program launch

- Test program activation may be carried out by:
  - The system reset
    - Suits for manufacturing test
  - Some interrupt/exception mechanism
    - Forced from the outside through the wrapper, or
    - Internally raised by means of a SW exception call
  - The OS by implementing a scheduling strategy that launches the test program
    - At regular intervals
    - During idle periods
    - Suits for on-line SBST program execution
Along the execution of the test program, results have to be stored in a suitable location readable from the outside.

Possible solutions:
- Some reserved memory locations
- Compression circuitries such as MISR or CRC modules to be read
  - From the outside of the chip by the tester
  - Directly by the embedded processor
Test stimuli generation

- May exploit
  - Stimuli from the designer (e.g., intended for design validation)
  - Random stimuli
  - Ad hoc stimuli, manually or automatically generated
- In any case, a *test coverage metric* should be available
Test Program Generation

- Can be done
  - Manually, or
  - Automatically
Manual Methods

- Empirical methods
- Global methods
- Unit-oriented methods
Empirical methods

- They are based on the knowledge of the Instruction Set Architecture (ISA), only

- Examples
  - All instructions
  - All instructions with all addressing modes
  - A set of code fragments to stimulate all units
  - ...

- Very approximated metrics are normally used
Global methods

- Aim at testing the whole processor starting from the ISA, only
- They are based on a systematic approach
- They do not guarantee a given fault coverage, although significant values can normally be reached
Method by Thatte and Abraham

- Was proposed in 1980 for an 8-bit processor
- Works on the RTL description of a simple processor, only, as it can be extracted from a user manual
- Defines rules to build a graph describing the instruction behavior wrt registers
- Defines a set of functional fault models
- Defines some procedures to generate sequences of instructions addressing the above fault models

Thatte and Abraham, Trans. on Comp., 1984
Functional fault models

- For the register decoding function
  - When a register is written
    - All the registers are written
    - No register is written
    - Another register is written

- For the instruction decoding and control function

- For the data storage function

- For the data transfer function

- For the data manipulation function
Procedures for test generation

- They allow detecting all the previously listed faults (one at a time)

Example

- To test the register decoding faults
  - Initialize all registers to a given value ZERO
  - For every register R
    - Write a value ONE in R
    - Read R
    - Read the other registers (they should store ZERO)
Unit-oriented methods

- They provide a sequence of operations able to exhaustively cover a given unit with respect to static and/or dynamic faults

- Examples
  - Adder
  - Multiplier
  - Cache controller
  - Branch Prediction Unit
  - ...

66
The adder and the multiplier are often large combinational blocks within the processor core.

Functional test programs for each of them may be generated in different ways:

- Resorting to a combinational ATPG
- Resorting to a specific algorithm
Resorting to an ATPG

- Extract the logic for the adder
- Run a combinational ATPG on it
- Transform each vector into an instruction (ATPG values are the input parameters)
- Add instructions to check result
Resorting to an ad hoc algorithm

- Based on a set of loops
- Stimulate the adder with a proper sequence of selected values
- Able to detect not only static faults (e.g., stuck-at faults) but also dynamic ones (e.g., delay faults)

Gizopoulos et al., ITC, 1997
Caches

- Most processors include caches.
- Their test is normally performed by silicon producers through BIST.
- The BIST facilities are seldom accessible and documented.
- Functional test is an alternative solution.

Di Carlo et al., Trans. on Comp., 2011
Functional test of caches

- Functional test addresses separately
  - The data part (storing both data and tags)
  - The control part
Testing the data part in a data cache

- It means testing a memory; therefore, the test mimics a March algorithm.
- The test is based on a proper sequence of memory accesses.
- Once the mapping policy implemented by the cache is known, transforming a March algorithm into the corresponding sequence of LOAD/STORE instructions is straightforward.
Testing a cache

- When transforming a March algorithm into a sequence of operations on cache:
  - Read with hit operations read from the cache
  - Read or write with miss operations write in the cache
Example

- If the cache is direct mapped
  - Accessing a cache line means accessing to one of the corresponding memory blocks
Testing the tag part

- This task is much more difficult, because tags can only indirectly be read and written.
- Reading a tag means accessing a cache line whose content is known, and then checking whether the expected hit or miss was performed.
- Writing a tag means forcing a miss on a given cache line.
Problems

- If the cache is set associative with LRU replacement policy, performing two write operations on the same cache line is impossible.
- In fact, the first operation marks the line as Most Recently Used, and the following access affects a different line.
- The only solution is to add further operations within the March algorithm.
- However, this may reduce the effectiveness of the March algorithm itself.
Testing an instruction cache

- This task is even more difficult
- It requires executing in a proper order a sequence of proper instructions placed in proper positions in memory
Cache controller

- Deeply embedded
  - Hard to control: suitable sequence of memory accesses
  - Hard to observe: mainly by checking whether any access produced a hit or miss
- Faults may cause either data or performance misbehaviors
- Testing the data part does not fully cover the control part (about 20% of stuck-at faults remain uncovered)
- The parts that remain uncovered are mainly those concerning the substitution policy
Observing the cache controller behavior

- If we are interested in faults affecting the static behavior (i.e., changing the produced data)
  - We can observe the data in memory at the end of the test

- If we are interested in faults affecting the dynamic behavior (i.e., changing the performance)
  - We have to evaluate the time required to execute the test program
Measuring the execution time

- This task can be performed in different ways
  - Resorting to some counter existing in the system
  - Resorting to some performance counter (if the processor offers this facility)
  - Resorting to some debug feature
  - Resorting to some ad hoc hardware
Cache controller functional test

- Testing the cache memory is not enough to test the cache controller
- Cache controller functions are relatively standard
- Developing an implementation-independent parametric test algorithm is feasible
Branch Prediction Units

- Branch Prediction Units (BPUs) are often included in modern microprocessor/microcontrollers.
- Sometimes they include relatively large memories and logic.
- Faults in BPUs:
  - may seriously affect the processor performance
  - rarely affect the data

Sanchez & Sonza, Trans. on VLSI, 2015
BPUs

- They normally take two forms
  - Branch History Table (BHT)
    - Returns prediction
  - Branch Target Buffer (BTB)
    - Returns prediction and expected target address
Automatic methods

- Generating the test program can be done in an automatic manner resorting to three approaches:
  - Randomizers
  - Macro-based methods
  - Formal methods
  - Evolutionary-based methods
Randomizers

- Data and instructions are randomly generated, following some constraints that
  - Guarantee the correctness of the test program
  - Guide the generation towards the most critical areas
- Test programs may be significantly long to execute and large to store
FRITS

- Intel heavily exploits randomizers for both testing and design validation
- FRITS is the internal tool developed for
  - Generating random programs
  - Expressing constraints

Parvathala et al., ITC, 2002
Macro-based approach

- A Macro is the basic block of the Test Program
- Each macro:
  - Focuses on a single target instruction in the µP Instruction Set
  - Includes the code for
    - Exciting the instruction functionalities
    - Making the results observable
- Some optimization technique is required for
  - Selecting the best set of macros
  - Assigning values to parameters

Corno et al., DATE, 2001
Example #1

The macro for the ADD instruction is:

**Excitation Phase**

- MOV AX, K1 ; load register AX with K1
- MOV BX, K2 ; load register BX with K2
- ADD AX, BX ; sum BX to AX

**Observation Phase**

- MOV RW, AX ; write AX to RW
- MOV RW, PSW ; write status reg to RW

Macro parameters

Observable location

RW
Example #2

The macro for the JG instruction is:

```
MOV BX, 0 ; clear BX
MOV AX, K1
CMP AX, K2 ; compare AX with K2
JG Label ; jump if AX > K2
MOV BX, 1 ; move 1 to BX
Label: MOV RW, BX; write BX to RW
```
do
{  m = select_a_macro();
    O = select_the_operands(m);
    F = compute_detected_faults(m, O);
    if( F is not empty )
        add m(O) to the test program;
}while( stopping_condition() == FALSE );
Macro Selection

- The final Test Program should have minimal length

- An adaptive approach can be adopted:
  - Macros are selected on a random basis
  - A selection probability is associated to each macro
  - At the beginning, all macros have the same selection probability
  - Each time a macro is selected, its selection probability is increased if it detects at least one fault, decreased elsewhere
Operand Selection

- An Evolutionary approach can be adopted:
  - Random values are initially chosen
  - An evaluation function is associated to each set of values
  - Current values are possibly improved using a Genetic Algorithm
Evaluation Function

- It is based on three terms:
  - A first order term, corresponding to the number of detected faults
  - A second order term, corresponding to the number of excited faults
  - A third order term, corresponding to the activity caused by the excited faults
- The value of the evaluation function is computed by fault simulation
Experimental Results

- A macro-based method was evaluated resorting to:
  - A prototype (named ATPGS) implementing the proposed approach
  - An Intel 80C51 model
ATPGS architecture

- Macro Library
- ATPG Kernel
- Evolutionary engine
- Fault Simulator
- Detected Faults
- Macros
- Macro + Operands
- Test Program
- μP netlist
- Fault List
Case Study: i8051 µC

- 8-bit microprocessor developed by Intel in the 80s
  - Quite old but still popular (USB)
  - 128 bytes directly-accessible data memory + 32 bytes *bit-addressable* data memory
  - 64KB external memory accessible using a special instruction (MOVX)
  - 4KB internal + 64KB external program memory
Case Study: i8051 μC (2)

- RT-Level description
  - 7,500 VHDL lines
  - 4 KB program + 2 KB data memory
- Gate-level description
  - 12K gates
  - 28,792 stuck-at faults
Case Study: i8051 µC (3)

- Instruction Set: 44 instructions
  - from 0-operand ones (e.g., DIV AB)
  - to 3-operand (e.g., CJNE Op1, Op2, R)
- Non-orthogonal
- Instruction Library: 81 entries
  - prologue, epilogue
  - 66 sequential operations
  - 13 conditional branches
Results

- Macro Generation
  - 2 days of an experienced programmer
- Parameter Optimization
  - 24 h (CPU time)
- Fault Coverage
  - 92%
- Test Program Length
  - 624 instructions
- #Macros
  - 213
Results (stuck-at FC)

- Random Test Program Generation: 80%
- ATPGS: 92%
- Full-scan version: 94%
Result analysis

- Most of the undetected faults are in the Divider:
  - better results could be obtained by running a combinational ATPG for computing the optimum parameters and removing untestable faults
- A nearly complete fault coverage has been obtained for the Control Unit
Formal methods

Recently, it was demonstrated that formal techniques (e.g., based on Model Checkers) can generate functional test programs even for relatively large pipelined processors.

Riefert et al., DATE, 2014
Advantages

- The method can
  - Generate the shortest test sequence for each fault
  - Identify untestable faults
  - Support further constraints (e.g., for in-field test)
Limitations

- CPU and memory requirements may prevent the scalability to large processors
The μGP approach

- Semi-automatic Test Program generation
- Based on an evolutionary algorithm (named μGP)
- Suitable internal representation, evaluation function and operators are defined
- A system architecture for automatic test program generation has been devised.

Sanchez et al., Evolutionary Optimization: the μGP toolkit, Springer, 2011
System Architecture
System Architecture

diagram showing components:
- Instruction library
- Generator
- Fault simulator
- Test program evaluator
- Simple description of μP assembly language syntax
- μGP evolutionary engine
- Test program
- Netlist
Advantages

- The method is able to always generate a test program
- Flexibility: by suitably changing the fitness function, different goals can be pursued
Limitations

- CPU requirements may limit the scalability of the method
Experimental evaluation

- Intel 80C51 model
- Sun Enterprise 250 running at 400 MHz with 2 GB RAM
- Test program generated in few days
  - Computational effort is mainly due to fault simulations
  - Evolutionary calculations required negligible CPU time
Test Programs Comparison

- General applications
  - Fibonacci, int2bin
- Exhaustive test bench
  - provided by 8051 designer
- ATPGS
  - DATE01
- Random (comparable CPU)
  - Same number of random test programs
  - Same number of random sequences of macros (DATE01)
Experimental Results (%FC)
<table>
<thead>
<tr>
<th>Processor Model</th>
<th>Generation method</th>
<th>CPU</th>
<th>ISA</th>
<th>Gate Count</th>
<th>HDL</th>
<th>FC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parwan</td>
<td>Precomputed stimuli</td>
<td>8-bit accumulator-based</td>
<td>Simple ISA 17 instructions</td>
<td>2 K</td>
<td>VHDL</td>
<td>96%</td>
</tr>
<tr>
<td>GL85</td>
<td>Functional randomizer</td>
<td>8-bit controller</td>
<td>Intel 8085 ISA 80 instructions</td>
<td>8 K</td>
<td>VHDL</td>
<td>90%</td>
</tr>
<tr>
<td>I8051</td>
<td>Feedback-based</td>
<td>8-bit CISC</td>
<td>Intel 8051 ISA 44 instructions</td>
<td>12K</td>
<td>VHDL</td>
<td>93%</td>
</tr>
<tr>
<td>Plasma</td>
<td>Deterministic algorithms</td>
<td>32-bit RISC</td>
<td>MIPS-I ISA 56 instructions</td>
<td>26 K</td>
<td>VHDL</td>
<td>95%</td>
</tr>
<tr>
<td>miniMIPS</td>
<td>Deterministic algorithms</td>
<td>32-bit RISC</td>
<td>MIPS-I ISA 51 instructions</td>
<td>33 K</td>
<td>VHDL</td>
<td>93%</td>
</tr>
<tr>
<td>Leon2</td>
<td>Feedback-based</td>
<td>32-bit RISC</td>
<td>SPARC V8</td>
<td>35 K</td>
<td>VHDL</td>
<td>Not reported</td>
</tr>
<tr>
<td>OpenRISC 1200</td>
<td>Constrained test generation</td>
<td>32-bit RISC</td>
<td>OpenRISC 1000 ISA 52 instructions</td>
<td>36 K</td>
<td>Verilog</td>
<td>93%</td>
</tr>
<tr>
<td>OpenSPARC T1</td>
<td>Deterministic algorithms</td>
<td>64-bit</td>
<td>SPARC V9 196 instructions (per core)</td>
<td>273 K</td>
<td>Verilog</td>
<td>Not reported</td>
</tr>
</tbody>
</table>
VLIW processors

- They are often used in embedded systems when digital signal processing is required
- They are characterized by
  - Regular structure
  - Customizable configuration
  - No (or limited) control circuitry
Functional test of VLIWs

- Effective test algorithms (developed for traditional processors) can be used for most modules.
- Some modules (e.g., the register file) require special test algorithms.
- Test program generation can be fully automated, once the specific VLIW architecture is known.
- The key issue is minimizing the total test time and test code size.

Sabena et al., Trans. on VLSI, 2014
Generation flow

- Processor configuration
- Automatic tool
- Test programs for each module
- Test program
Experimental data

- Case study: ρ-VEX processor, developed by Delft University of Technology
- Specific configuration
- Automatic generation of optimized test program (CPU time of about 40 hours)
## Results

<table>
<thead>
<tr>
<th>p-VEX Components</th>
<th>Faults [#]</th>
<th>Fault Coverage</th>
<th>Clock Cycles [#]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fetch</strong></td>
<td>2,156</td>
<td>99.2%</td>
<td>1,028</td>
</tr>
<tr>
<td><strong>Decode</strong></td>
<td>269,196</td>
<td>98.1%</td>
<td>760</td>
</tr>
<tr>
<td><strong>Execute</strong></td>
<td>75,554</td>
<td>98.3%</td>
<td>4,580</td>
</tr>
<tr>
<td>4 ALUs</td>
<td>75,554</td>
<td>98.3%</td>
<td>4,580</td>
</tr>
<tr>
<td>2 MULs</td>
<td>37,244</td>
<td>98.6%</td>
<td>2,253</td>
</tr>
<tr>
<td>MEM</td>
<td>1,730</td>
<td>97.2%</td>
<td>1,280</td>
</tr>
<tr>
<td><strong>Writeback</strong></td>
<td>1,420</td>
<td>98.1%</td>
<td>1,180</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>387,290</td>
<td>98.2%</td>
<td>11,081</td>
</tr>
</tbody>
</table>
Summary

- Introduction to test
- Functional test
- Functional test of processors
- Functional test of peripheral components
- In-field test
- Conclusions
Peripheral testing

- Peripheral cores can also be tested via functional test
- In this case the test requires
  - A suitable test program
    - To program the peripheral
    - To exercise the peripheral
  - Some external data (either in input or output)

Apostolakis et al., D&T, 2009
External support

To excite/observe the peripheral from the outside, two solutions are possible

Resorting to an external ATE

Adding a loopback connection
To excite/observe the peripheral from the outside, two solutions are possible:

- Resorting to an external ATE
- Adding a loopback connection

More expensive but highly flexible (any stimuli can be applied)
To excite/observe the peripheral from the outside, two solutions are possible:

- Resorting to an external ATE
  - Cheaper but less flexible (some stimuli cannot be generated)

- Adding a loopback connection
  - More flexible but more expensive
Functional test of peripherals

- Testing peripherals using functional test may follow two strategies
  - Testing the peripheral working in the configuration that is used by a given application, only
  - Testing the peripheral working in all possible configurations
Functional test of peripherals

- Testing peripherals using functional test may follow two strategies
  - Testing the peripheral working in the configuration that is used by a given application, only
  - Testing the peripheral working in all possible configurations

This is the situation when in-field test is considered
Functional test

- Testing peripherals using functional test may follow two strategies
  - Testing the peripheral working in the configuration that is used by a given application, only
  - Testing the peripheral working in all possible configurations

This is the situation when end-of-manufacturing test is considered.
Peripheral testing

- It requires
  - Configuring the peripheral (configuration code fragment)
  - Exercising the peripheral (functional fragment, including code and data)
Test architecture
Test stimuli generation

- It can be done in different ways:
  - Manually or automatically
  - Starting from a data-sheet, an RT-level description, a gate-level description
- Suitable metrics are required to guide generation (and to stop it)
Case of study

SoC

Motorola 6809 Microprocessor

RAM Memory

UART

Configuration Logic Module

Receiver Module

Transmitter Module

PIA

Configuration and functionality module

BUS
## Details

<table>
<thead>
<tr>
<th>Description</th>
<th>PIA</th>
<th>UART</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RT-level</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td># statements</td>
<td>149</td>
<td>383</td>
</tr>
<tr>
<td># branches</td>
<td>134</td>
<td>182</td>
</tr>
<tr>
<td># conditions</td>
<td>75</td>
<td>73</td>
</tr>
<tr>
<td># expressions</td>
<td>N/A</td>
<td>54</td>
</tr>
<tr>
<td><strong>Gate-level</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td># gates</td>
<td>1,016</td>
<td>2,247</td>
</tr>
<tr>
<td># faults</td>
<td>1,938</td>
<td>4,054</td>
</tr>
</tbody>
</table>
## PIA results

<table>
<thead>
<tr>
<th>Code Coverage [%]</th>
<th>Test Time</th>
<th>Test Blocks</th>
<th>Fault coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[# clock cycle]</td>
<td>[#]</td>
<td>[%]</td>
</tr>
<tr>
<td>Statement</td>
<td>99.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>98.5</td>
<td>625</td>
<td>9</td>
</tr>
<tr>
<td>Condition</td>
<td>98.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## UART results

<table>
<thead>
<tr>
<th>Code Coverage [%]</th>
<th>Test Time</th>
<th>Test Blocks</th>
<th>Fault Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[# clock cycle]</td>
<td>[#]</td>
<td>[%]</td>
</tr>
<tr>
<td>Statement</td>
<td>99.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>99.5</td>
<td>4,300</td>
<td>14</td>
</tr>
<tr>
<td>Condition</td>
<td>99.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>99.6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### UART results

<table>
<thead>
<tr>
<th>Metric</th>
<th>Coverage [%]</th>
<th>Test Time # [clock cycle]</th>
<th>Fault Coverage [%]</th>
<th>Statement #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>99.2</td>
<td>4,300</td>
<td>14</td>
<td>94.3</td>
</tr>
<tr>
<td>Branch</td>
<td>99.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition</td>
<td>99.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>99.6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Maximizing RT-level metrics can guarantee good gate-level fault coverage.
Open issues

- Minimizing the test length
  - Identifying the minimal number of configurations for test
  - Suitably generating the stimuli for each configuration
- Minimizing the external support
  - Loopback
- Testing asynchronous features (e.g., arbitration)
Testing system peripherals

- System peripheral units (e.g., interrupt controllers, DMA controllers) are even more complex to test because
  - During the usage phase, the CPU can not completely access to them, but requires passing through other peripheral modules
  - Often do manage asynchronous signals

Grosso et al., JETTA, 2012
Testing the IC requires:
- Programming the peripherals
- Programming the IC
- Setting up suitable ISRs
- Forcing peripherals to raise interrupt requests
- Checking the IC behavior
Summary

- Introduction to test
- Functional test
- Functional test of processors
- Functional test of peripheral components
- *In-field test*
- Conclusions
In-field test

- Functional test can be used for in-field test, too
- Short test procedures can be activated periodically, or during idle times
- They must be written in such a way that they don't interfere with the μP normal behavior

Paschalis et al., Trans. on CAD, 2005
Some reliability standards define strict requirements for in-field test targets.

For example, the ISO 26262 standard for automotive requires the following fault coverage (for stuck-at and transition faults), depending on the reliability level of the application.
In-field test & reliability standards

- Some reliability standards define strict requirements for in-field test targets.
- For example, the ISO 26262 standard for automotive requires the following fault coverage (for stuck-at and transition faults), depending on the reliability level of the application:

<table>
<thead>
<tr>
<th></th>
<th>ASIL B</th>
<th>ASIL C</th>
<th>ASIL D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuck + Transition fault coverage</td>
<td>&gt; 90%</td>
<td>&gt; 97%</td>
<td>&gt; 99%</td>
</tr>
</tbody>
</table>

- Mirrors
- Airbag
- Steer-by-wire
In-field test constraints

- When SBST is used for in-field test, several scenarios should be considered:
  - Test is performed when the system is not (yet) running (e.g., at startup)
  - Test is performed while the system is running (e.g., during periodical time slots)

Riefert et al., DATE, 2015
Test at startup

- In this case
  - The whole test can be performed until its end
  - The previous context has not to be preserved and then restored
  - The test can access most of the system features
  - Time duration is moderately important
  - Test code size is moderately important
Test during operation

- In this case the application is interrupted, the test launched and run for a given period (often short), and then the application is resumed.

- This means that:
  - The whole test has to be split in fragments, each short enough to be executed in a test slot; each fragment should be independent.
  - The test should minimally interfere with the application.
  - Test duration and code size are very important.
Critical units

- Some modules are particularly critical when in-field testing is considered
  - Address Calculation Unit
  - Branch Prediction Unit
  - Exception Unit
- Special techniques are required to make their test feasible and effective
Address Calculation Unit

- Used by LOAD/STORE instructions
- Embeds a large dedicated adder
- Requires computing addresses spanning the whole memory
- Its test may conflict with constraints coming from the application, such as
  - The amount of memory for test data and code
  - The position of this memory area
Proposed approach

- Test generation is based on atomic blocks, such as
  
  \[
  \begin{align*}
  \text{sd\_inst} & \quad rX, \text{ base (offset)} \\
  \text{ld\_inst} & \quad rY, \text{ base (offset)}
  \end{align*}
  \]

- A proper selection of base and offset is required to fully control the inputs of the adder.
Some figures

<table>
<thead>
<tr>
<th>Case study</th>
<th>Gates</th>
<th>SA faults (#)</th>
<th>Self-Test data size</th>
</tr>
</thead>
<tbody>
<tr>
<td>STM core</td>
<td>689</td>
<td>4,188</td>
<td>2 KB</td>
</tr>
<tr>
<td>miniMIPS</td>
<td>342</td>
<td>1,988</td>
<td>2 KB</td>
</tr>
</tbody>
</table>

Some hours of CPU time were required for generating the final test program

<table>
<thead>
<tr>
<th>Self-Test code size</th>
<th>Exec Time (cc)</th>
<th>Fault Coverage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4 KB</td>
<td>823</td>
<td>95.11</td>
</tr>
<tr>
<td>1.6 KB</td>
<td>778</td>
<td>94.27</td>
</tr>
</tbody>
</table>
Functionally untestable faults

- Some faults may be untestable during the operational phase

Examples
- Faults related to scan chains
- Faults related the debug circuitry
- Faults related to the reset circuitry

Bernardi et al., DATE, 2013
Functionally untestable faults

- In some cases these faults do not affect the normal behavior
  - Faults in the circuitry used for manufacturing test
  - Faults in the debug circuitry
  - Faults in unused circuitry
- Their number could be non-negligible
- They may significantly contribute to make difficult matching the in-field test requirements
Functionally untestable faults

- How can we identify them?
- Commercial ATPG tools do not provide this feature
- Standards and regulations require proved procedures
- Formal techniques can be used
Test program compaction

- Test duration is often an issue, especially when the test is performed during the application idle times
- Test programs are often quite long / redundant
  - Because they come from legacy code
  - Because they have (partly) be generated randomly
  - Because compaction has not been addressed during generation
Approaches

- **Static approach**
  - Compaction is addressed AFTER test program generation, e.g., checking whether some instructions can be removed without reducing the achieved fault coverage

- **Dynamic approach**
  - Compaction is addressed DURING test program generation

Gaudesi et al., ETS, 2015
Conclusions

- Functional test
  - is required for in-field test and to catch some defects
  - is a hot research topic
- Effective test algorithms can be developed for several modules
- Hybrid solutions (combining it with DfT) may be effective
- Cost-effective test generation is still an open problem
- Test program compaction is often an issue
Questions?