Intro. to Computer Architecture

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨csdsp@bristol.ac.uk⟩

January 9, 2018

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

- Agenda:
  1. comments, questions, recap, then
  2. a short overview of (hardware-oriented) strategies for **testing** and **debuging**.

Notes:

COMS12200 lecture: week #5

Quote

Beware of bugs in the above code; I have only proved it correct, not tried it.

– Knuth (http://en.wikiquote.org/wiki/Donald_Knuth)

Quote

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

– Kernighan (http://en.wikiquote.org/wiki/Brian_Kernighan)

Notes:

- Example:
  - The Intel Pentium (P5) model processor has a 273-pin package; in the worse case there are

$$2^{273} = 151771007205135083665582961470587414581438\backslash 034300948400097797844510851897281656913 92$$

    possible inputs (ignoring outputs, and maintained state).
  - In 1994, a researcher discovered the processor had a bug:
    - The processor should have been able to compute

$$(4195835 \cdot 3145727)/3145727 = 4195835.$$

    - The so-called FDIV bug [5] resulted in

$$(4195835 \cdot 3145727)/3145727 = 4195579.$$

  - Intel recalled the defective processors, at an estimated cost of $475 million.

Notes:

COMS12200 lecture: week #5

Definition

Given

$$\text{specification} \rightsquigarrow \text{design} \rightsquigarrow \text{implementation} \rightsquigarrow \textbf{Design Under Test (DUT)}$$

we should carefully distinguish between

validation	$\Rightarrow$	check whether specification matches requirements (i.e., is fit for purpose)
	$\simeq$	"is this the correct specification?"
verification	$\Rightarrow$	check (or *prove*) whether implementation matches specification
	$\simeq$	"is the implementation correct?"
testing	$\Rightarrow$	check (or *prove*) whether DUT matches specification
	$\simeq$	"is the DUT correct?"

Notes:

Definition

A **fault** describes a (static) bug in the DUT:

- **design faults** (or **functional faults**) i.e., design is incorrect in some way,
- **implementation faults**, e.g., design is correct but synthesis produced an incorrect implementation,
- **manufacture faults**, i.e., implementation is correct but manufacturing defects produce an incorrect DUT,
- **operational faults**, e.g., due to wear (or "old age") or the impact of operational limits.

Definition

An **error** is an instance of the fault being triggered during (dynamic) use of the DUT.

Definition

A **failure** is the *effect* of an error(s); perhaps the DUT malfunctions, for example, or perhaps it explodes.

Notes:

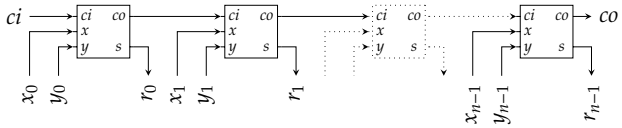
- Another important classification wrt. faults is whether they are
  - **permanent**, or
  - **transient**,

  in either case noting a fault may only occur after a **burn-in** period: this motivates explicit burn-in tests [2], which could be viewed as a type of stress test.
- Example faults include the following:
  - timing (or delay) faults, where a signal may settle to the correct value but more slowly or quickly than expected,
  - stuck-at faults, where a signal is permanently 0 or 1,
  - closed connection (or bridged connection) faults, where two wires are "fused" when they should be separate,
  - open connection faults, where a wire is "broken" so that one end is disconnected from the other.

A case-study: testing a ripple-carry adder (1)

- Recap: we produced the following design for (ripple-carry) addition

Circuit



- Problem: if we implement this design, how do we
  - test whether the implementation is correct, *and*
  - diagnose (or debug) the implementation (and/or design) if not.
- Solution: we need (at least) two components ...

Notes:

Notes:

Definition

The term **test fixture** is typically used to capture the *physical* infrastructure for testing, e.g., the equipment used to interact with a DUT and thereby provide input (or stimuli) and inspect output.

Definition

The term **test bench** is typically used to capture the *logical* infrastructure for testing, e.g.,

1. a set of inputs,
2. a mechanism to produce an expected output given an input,
3. a mechanism to check the actual vs. expected output.

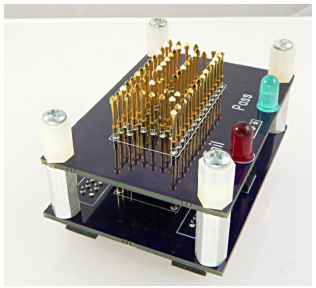
Note that where the DUT is simulated, the test bench can be entirely virtual; otherwise, a text fixture is required to interact with the physical DUT.

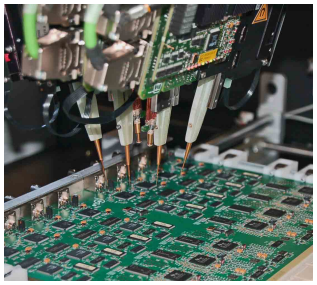
Notes:

- Problem: how do form connectivity with the DUT?
- Solution: given a packaged IC, or PCB, we actually have (at least) two options

1. **in-circuit test**, *with* test fixture ⇒ bed of nails
2. **in-circuit test**, *without* test fixture ⇒ flying lead

e.g.,





both of which are easier where **test points** are available.

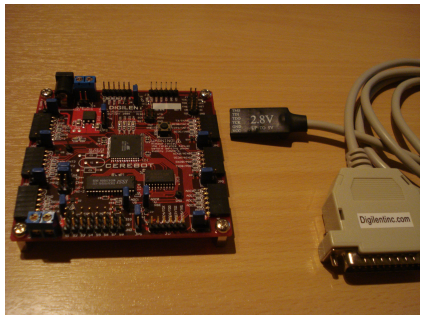
http://cdn-blog.adafruit.com/uploads/2012/09/testfixture_nails5.jpg
http://electromechpk.com/uploads/products/ElectricalandFlyProbeTesting.jpg

A case-study: testing a ripple-carry adder (4)

Component #1: a test fixture

- Problem: how do we interact with (e.g., control) the DUT?
- Solution: **boundary scan** [1].
  - **Joint Test Action Group (JTAG)** [3] is a common approach.
  - The basic idea is to include two on-chip components
    1. a **scan-chain** that links together all the latches (resp. flip-flops), and
    2. a **JTAG interface** that allows external access to the scan-chain,

    i.e.,



    1. Test Data In (TDI),
    2. Test Data Out (TDO),
    3. Test Clock (TCK),
    4. Test Mode Select (TMS), and
    5. Test Reset (TRST) which is optional.

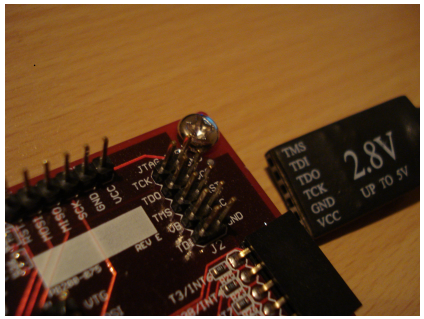
  - We then use these components to interact with the DUT ...

Notes:

A case-study: testing a ripple-carry adder (4)
Component #1: a test fixture

- Problem: how do we interact with (e.g., control) the DUT?
- Solution: **boundary scan** [1].
  - **Joint Test Action Group (JTAG)** [3] is a common approach.
  - The basic idea is to include two on-chip components
    1. a **scan-chain** that links together all the latches (resp. flip-flops), and
    2. a **JTAG interface** that allows external access to the scan-chain,

    i.e.,



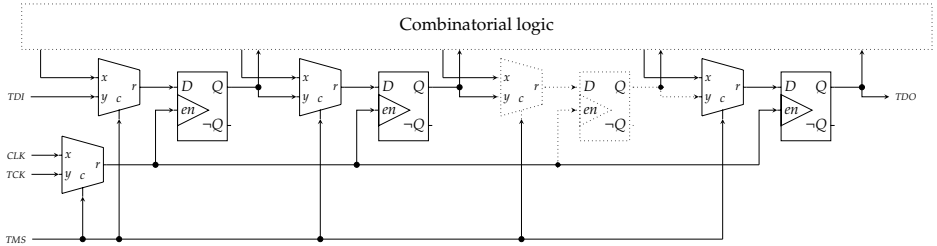
    1. Test Data In (TDI),
    2. Test Data Out (TDO),
    3. Test Clock (TCK),
    4. Test Mode Select (TMS), and
    5. Test Reset (TRST) which is optional.

  - We then use these components to interact with the DUT ...

© Daniel Page ⟨cs@bristol.ac.uk⟩
Intro. to Computer Architecture git # 3b6f641 @ 2018-01-09 University of BRISTOL

Notes:

Example

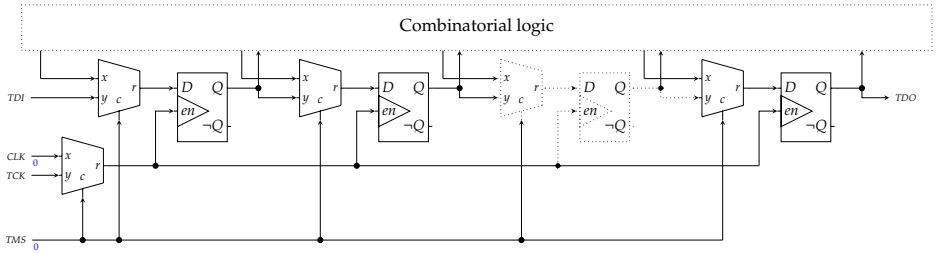


1. Initially, the circuit computes results as normal.

Notes:

A case-study: testing a ripple-carry adder (5)
Component #1: a test fixture

Example



1. Initially, the circuit computes results as normal.

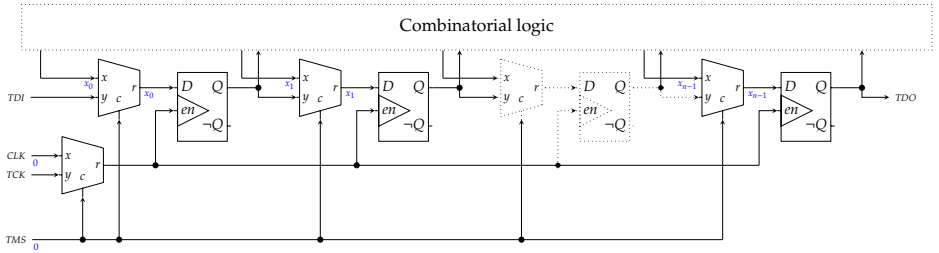
Notes:

© Daniel Page ⟨cs@bristol.ac.uk⟩
Intro. to Computer Architecture
git # 3b6f641 @ 2018-01-09
University of BRISTOL

A case-study: testing a ripple-carry adder (5)

Component #1: a test fixture

Example



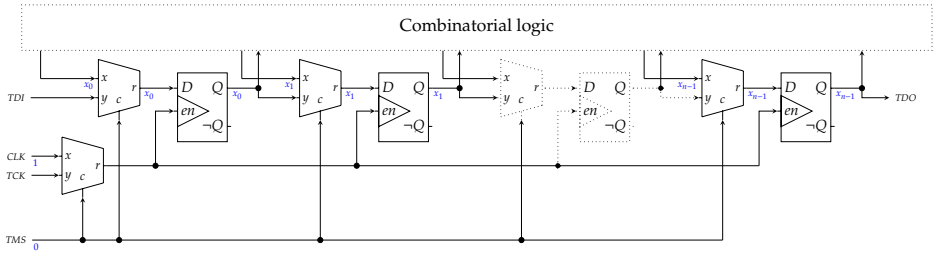
1. Initially, the circuit computes results as normal.

Notes:

A case-study: testing a ripple-carry adder (5)

Component #1: a test fixture

Example

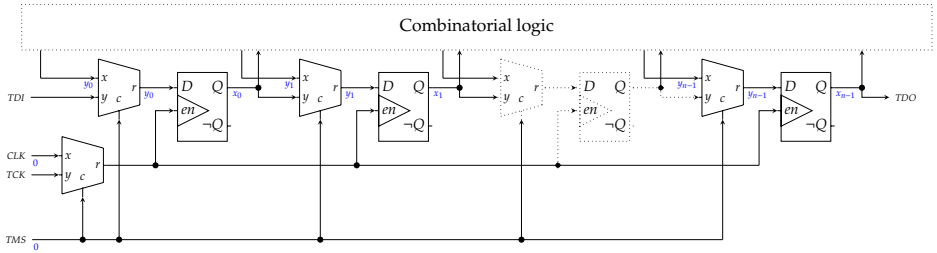


1. Initially, the circuit computes results as normal.

Notes:

© Daniel Page ⟨cs@bristol.ac.uk⟩
Intro. to Computer Architecture
git # 3b6f641 @ 2018-01-09
University of BRISTOL

Example

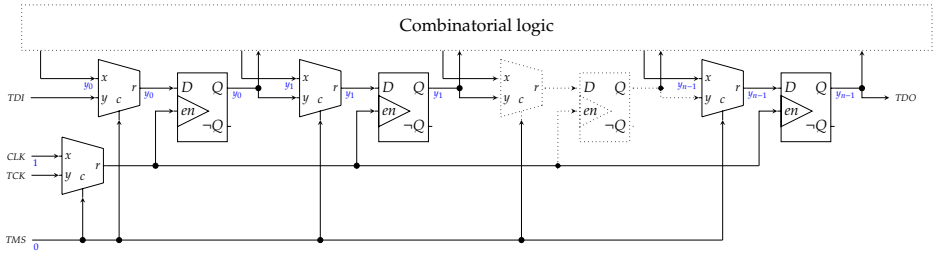


1. Initially, the circuit computes results as normal.

Notes:

A case-study: testing a ripple-carry adder (5)
Component #1: a test fixture

Example

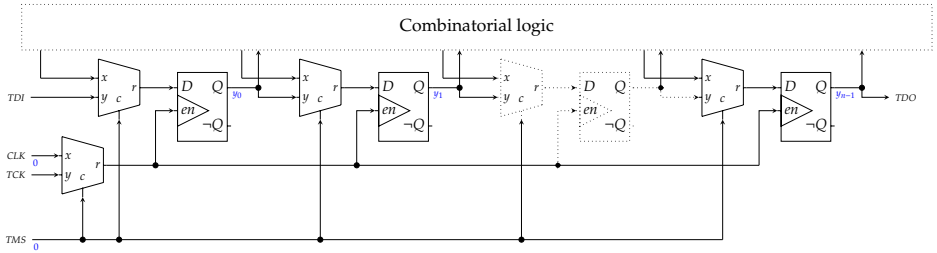


1. Initially, the circuit computes results as normal.

Notes:

© Daniel Page ⟨cs@bristol.ac.uk⟩
Intro. to Computer Architecture
git # 3b6f641 @ 2018-01-09
University of BRISTOL

Example

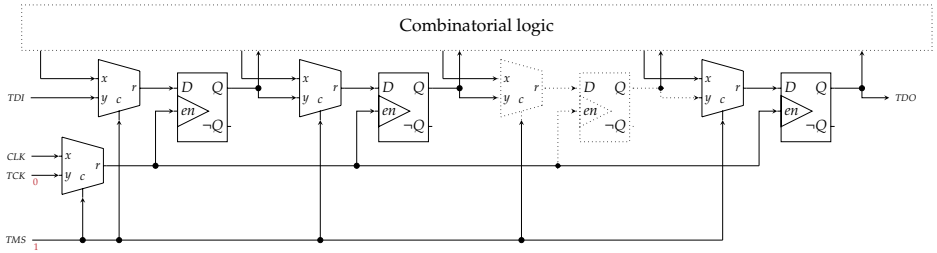


1. Initially, the circuit computes results as normal.

Notes:

A case-study: testing a ripple-carry adder (5)
Component #1: a test fixture

Example



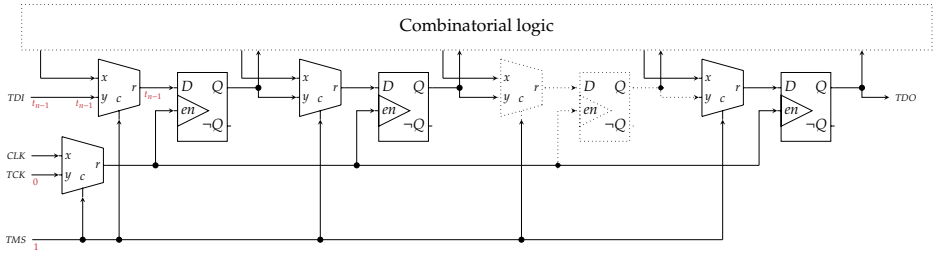
1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.

Notes:

A case-study: testing a ripple-carry adder (5)

Component #1: a test fixture

Example



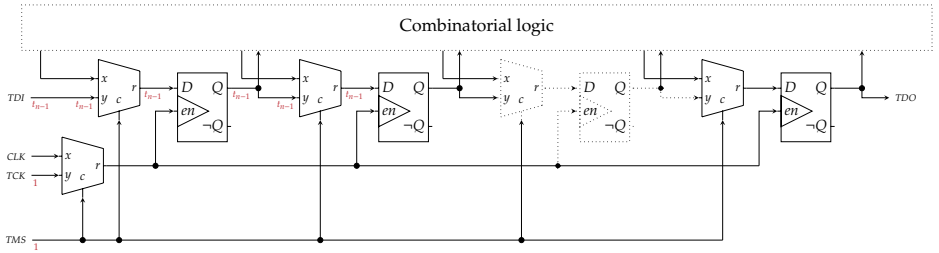
1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.

Notes:

A case-study: testing a ripple-carry adder (5)

Component #1: a test fixture

Example



1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.

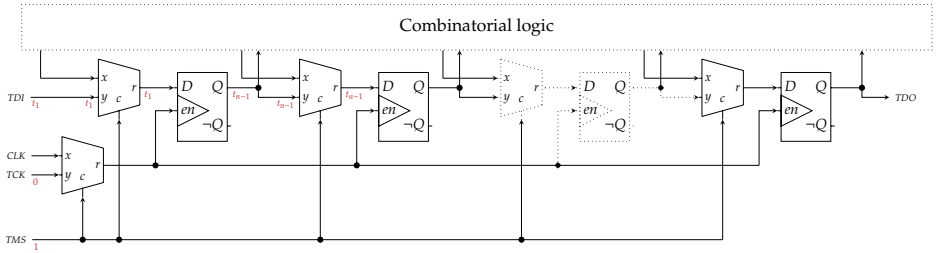
Notes:

© Daniel Page ⟨cs@bristol.ac.uk⟩
Intro. to Computer Architecture
git # 3b6f641 @ 2018-01-09
University of BRISTOL

A case-study: testing a ripple-carry adder (5)

Component #1: a test fixture

Example



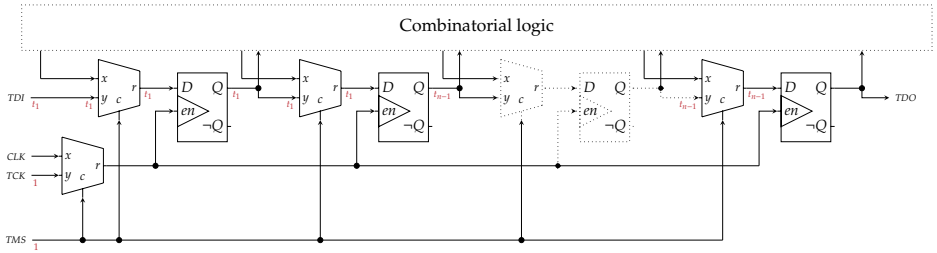
1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.

Notes:

A case-study: testing a ripple-carry adder (5)

Component #1: a test fixture

Example



1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.

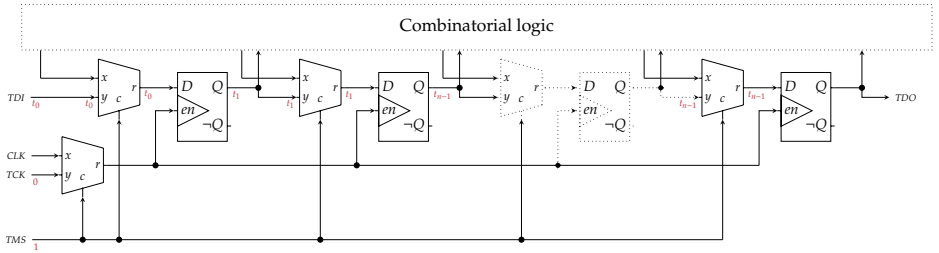
Notes:

© Daniel Page ⟨cs@bristol.ac.uk⟩
Intro. to Computer Architecture
git # 3b6f641 @ 2018-01-09
University of BRISTOL

A case-study: testing a ripple-carry adder (5)

Component #1: a test fixture

Example



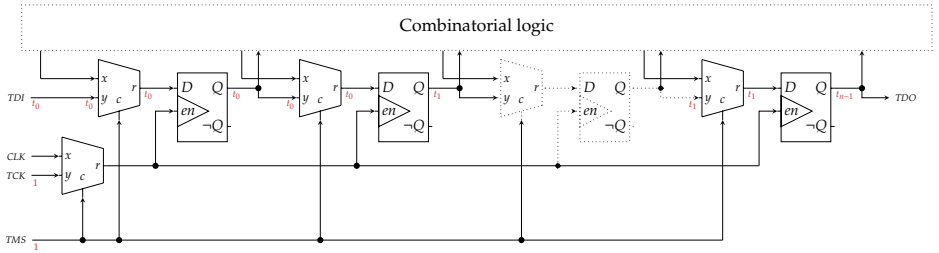
1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.

Notes:

A case-study: testing a ripple-carry adder (5)

Component #1: a test fixture

Example

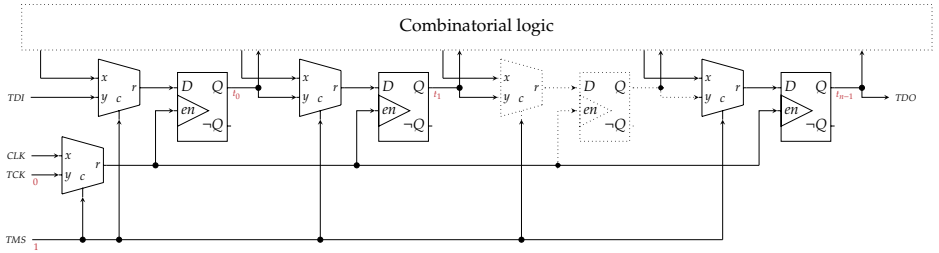


1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.

Notes:

© Daniel Page ⟨cs@bristol.ac.uk⟩
Intro. to Computer Architecture
git # 3b6f641 @ 2018-01-09
University of BRISTOL

Example

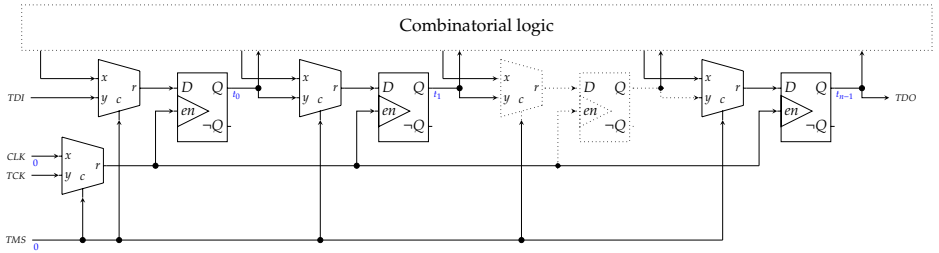


1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.

Notes:

A case-study: testing a ripple-carry adder (5)
Component #1: a test fixture

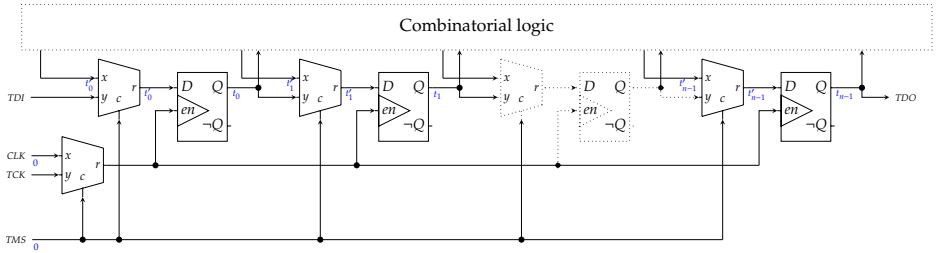
Example



1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.
4. The circuit is placed in normal mode using TMS and clocked until the required result is computed.

Notes:

Example

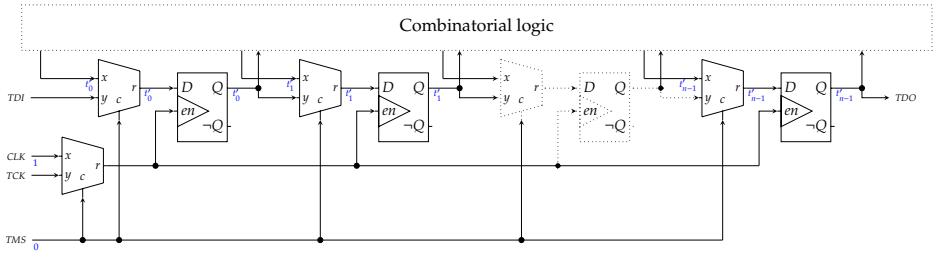


1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.
4. The circuit is placed in normal mode using TMS and clocked until the required result is computed.

Notes:

A case-study: testing a ripple-carry adder (5)
Component #1: a test fixture

Example

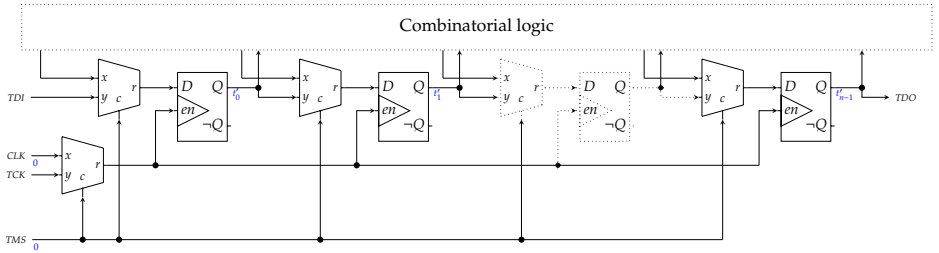


1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.
4. The circuit is placed in normal mode using TMS and clocked until the required result is computed.

© Daniel Page ⟨cs@bristol.ac.uk⟩
Intro. to Computer Architecture
git # 3b6f641 @ 2018-01-09
University of BRISTOL

Notes:

Example

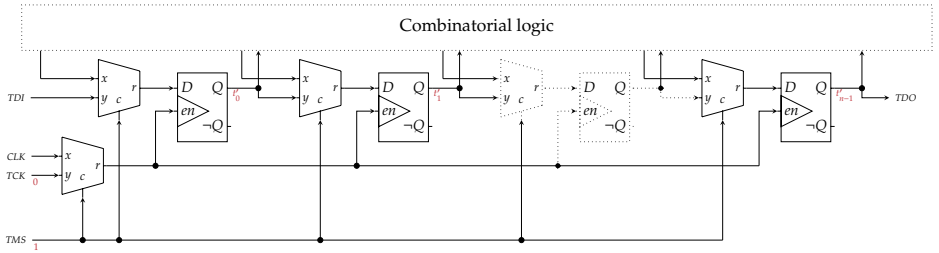


1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.
4. The circuit is placed in normal mode using TMS and clocked until the required result is computed.

Notes:

A case-study: testing a ripple-carry adder (5)
Component #1: a test fixture

Example



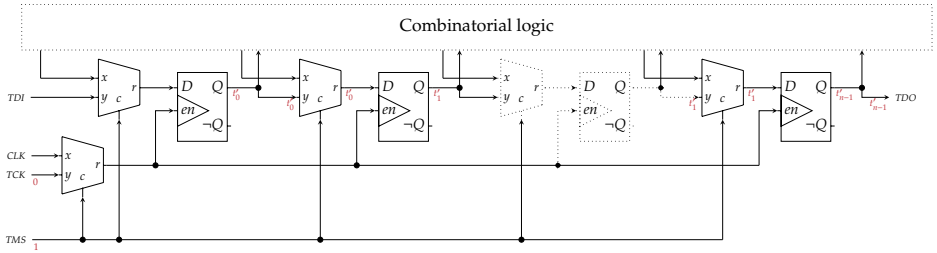
1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.
4. The circuit is placed in normal mode using TMS and clocked until the required result is computed.
5. The circuit is placed in debug mode using TMS.

Notes:

A case-study: testing a ripple-carry adder (5)

Component #1: a test fixture

Example



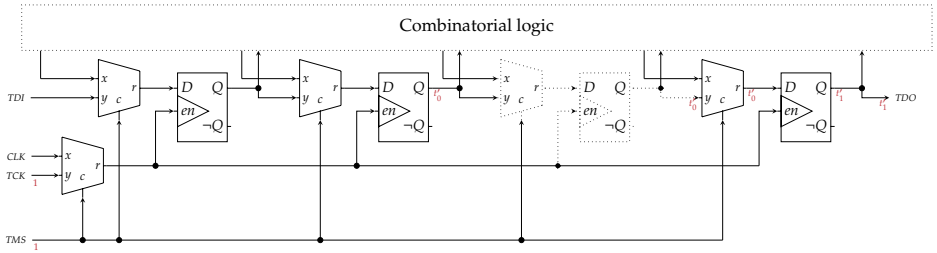
1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.
4. The circuit is placed in normal mode using TMS and clocked until the required result is computed.
5. The circuit is placed in debug mode using TMS.
6. The result vector is scanned out through TDI; each edge on TCK writes to TDO and shifts the scan-chain along.

Notes:

A case-study: testing a ripple-carry adder (5)

Component #1: a test fixture

Example

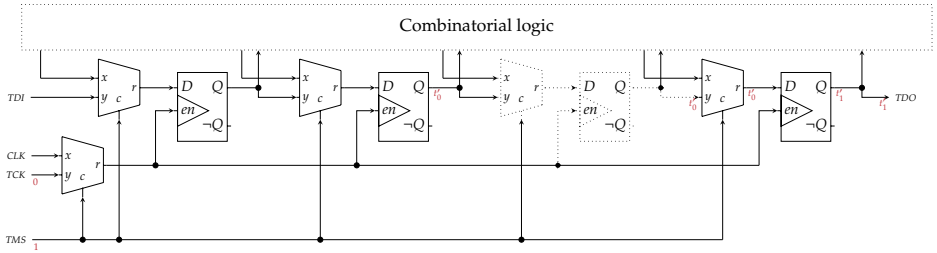


1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.
4. The circuit is placed in normal mode using TMS and clocked until the required result is computed.
5. The circuit is placed in debug mode using TMS.
6. The result vector is scanned out through TDI; each edge on TCK writes to TDO and shifts the scan-chain along.

© Daniel Page ⟨cdp@cs.bris.ac.uk⟩
Intro. to Computer Architecture
git # 3b6f641 @ 2018-01-09
University of BRISTOL

Notes:

Example

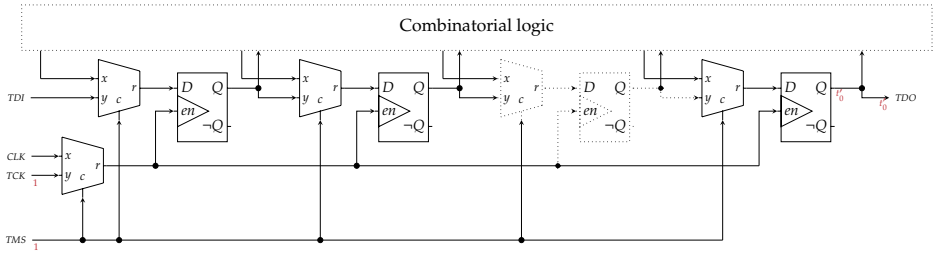


1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.
4. The circuit is placed in normal mode using TMS and clocked until the required result is computed.
5. The circuit is placed in debug mode using TMS.
6. The result vector is scanned out through TDI; each edge on TCK writes to TDO and shifts the scan-chain along.

Notes:

A case-study: testing a ripple-carry adder (5)
Component #1: a test fixture

Example

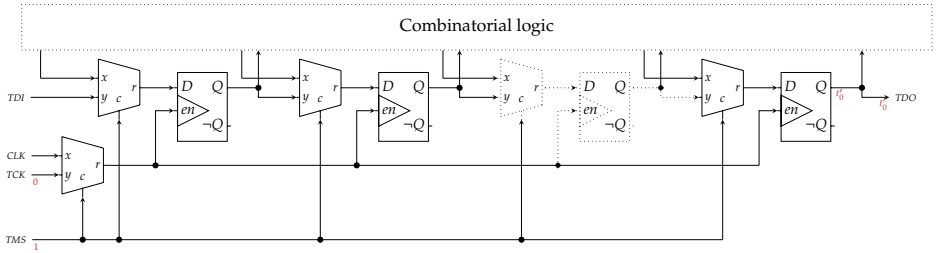


1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.
4. The circuit is placed in normal mode using TMS and clocked until the required result is computed.
5. The circuit is placed in debug mode using TMS.
6. The result vector is scanned out through TDI; each edge on TCK writes to TDO and shifts the scan-chain along.

© Daniel Page ⟨cdp@cs.bris.ac.uk⟩
Intro. to Computer Architecture
git # 3b6f641 @ 2018-01-09
University of BRISTOL

Notes:

Example

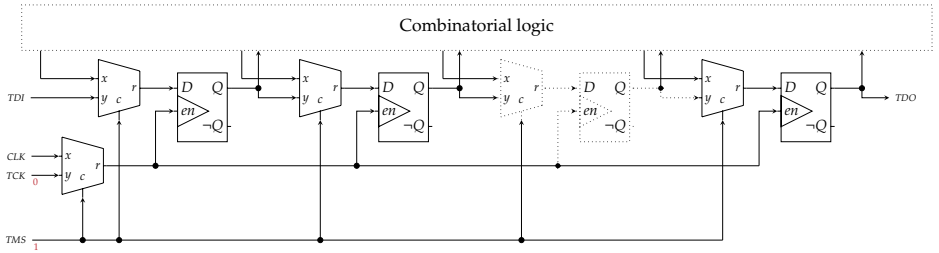


1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.
4. The circuit is placed in normal mode using TMS and clocked until the required result is computed.
5. The circuit is placed in debug mode using TMS.
6. The result vector is scanned out through TDI; each edge on TCK writes to TDO and shifts the scan-chain along.

Notes:

A case-study: testing a ripple-carry adder (5)
Component #1: a test fixture

Example



1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.
4. The circuit is placed in normal mode using TMS and clocked until the required result is computed.
5. The circuit is placed in debug mode using TMS.
6. The result vector is scanned out through TDI; each edge on TCK writes to TDO and shifts the scan-chain along.

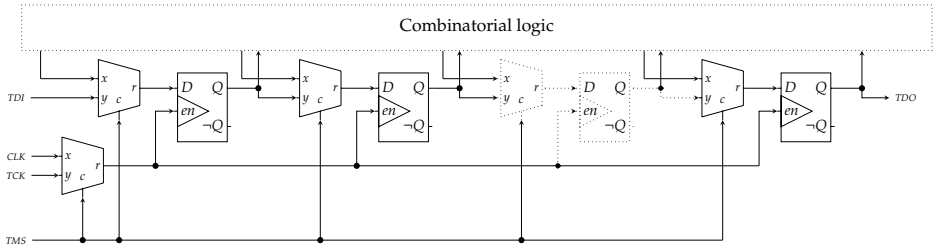
Notes:

© Daniel Page ⟨cdp@cs.bris.ac.uk⟩
Intro. to Computer Architecture
git # 3b6f641 @ 2018-01-09
University of BRISTOL

A case-study: testing a ripple-carry adder (5)

Component #1: a test fixture

Example



1. Initially, the circuit computes results as normal.
2. Then, at some point, it is placed in debug mode using TMS.
3. The test vector is scanned in through TDI; each edge on TCK reads from TDI and shifts the scan-chain along.
4. The circuit is placed in normal mode using TMS and clocked until the required result is computed.
5. The circuit is placed in debug mode using TMS.
6. The result vector is scanned out through TDI; each edge on TCK writes to TDO and shifts the scan-chain along.

Notes:

A case-study: testing a ripple-carry adder (6)

Component #2: a test bench

Definition

A **test vector** (or **test pattern**) is a single input and associated output; using a *set* of n test vectors

1. increases the level of coverage (i.e., proportion of DUT exercised), and
2. increases our *confidence* the DUT is correct.

- Item: given an oracle [4] O for functionality f as realised by the DUT,
  - construct a set

$$\{(x_0, r_0 = O(x_0)), (x_1, r_1 = O(x_1)), \ldots, (x_{n-1}, r_{n-1} = O(x_{n-1}))\}$$

    of test vectors, then
  - interact with the DUT to check

$$f(x_i) \stackrel{?}{=} O(x_i)$$

    for all $0 < i < n$.

Notes:

A case-study: testing a ripple-carry adder (7)
Component #2: a test bench

Definition

A (set of) test vector(s) could be classified at least per

1. **exhaustive** :
2. **heuristic** :
3. **diagnostic** :

Definition

The generation and use of a (set of) test vector(s) forms a wider taxonomy still: they could be deemed

1. **online** or **offline**,
2. **static** or **adaptive**, and
3. **random** or **deterministic**.

Notes:

A case-study: testing a ripple-carry adder (7)
Component #2: a test bench

Definition

A (set of) test vector(s) could be classified at least per

1. **exhaustive** : $x \in \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$
2. **heuristic** : $\begin{cases} \text{small} & \text{cases st. } x \in \{\pm 0, \pm 1, \ldots, \pm 2^m - 1\} \text{ for } m \ll n \\ \text{boundary} & \text{cases st. } x \in \{-2^{n-1}, -1, 0, +1, 2^{n-1} - 1\} \end{cases}$
3. **diagnostic** : an x selected st. if $f(x)$ fails, we infer that a specific fault exists

Definition

The generation and use of a (set of) test vector(s) forms a wider taxonomy still: they could be deemed

1. **online** or **offline**,
2. **static** or **adaptive**, and
3. **random** or **deterministic**.

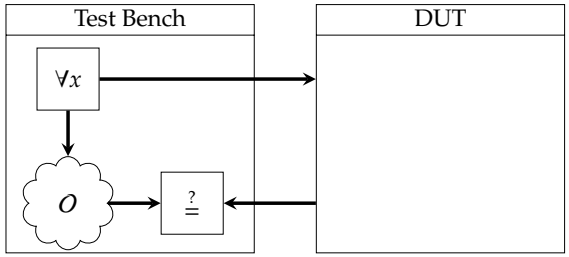
Notes:

© Daniel Page ⟨cdp@bristol.ac.uk⟩
Intro. to Computer Architecture git # 3b6f641 @ 2018-01-09 University of BRISTOL

A case-study: testing a ripple-carry adder (8)

Component #2: a test bench

- Problem: how could we organise the test bench?
- Solution(s):



1. stand-alone test bench $\simeq$ *external* test bench

Notes:

- It is obviously possible to enumerate a list of positive and negative features for each approach; one or other may be the better option for a given context. Although non-exhaustive, the following attempts to capture some of the main points:

  1. The stand-alone test bench:

     +ve: can pre-compute test vectors, using an oracle that might be represented by a database, or an alternative implementation, for example,
     +ve: test plan can be adaptive,
     −ve: tests are typically applied after manufacture only, so not close to time of use,
     −ve: is less resource efficient in the sense it does require a dedicated test fixture,
     +ve: is more resource efficient in the sense there is 1 test fixture for n DUTs.

  2. The BIST approach:

     −ve: cannot (as easily) pre-compute test vectors, since the storage requirement is typically too great,
     −ve: cannot cope (as easily) with faults that occur in the test mechanism itself,
     +ve: tests can be applied close to time of use,
     −ve: test plan cannot (as easily) be adaptive, in the sense it must be fixed when the DUT is manufactured,
     +ve: is more resource efficient in the sense it does not require a dedicated test fixture,
     −ve: is less resource efficient in the sense there are n test benches and test fixtures for n DUTs (because they are effectively built into *each* DUT).

- The concept of *arithmetic* BIST, for example relies on testing arithmetic identities such as

$$(x + y) - (x + y) \stackrel{?}{=} 0$$

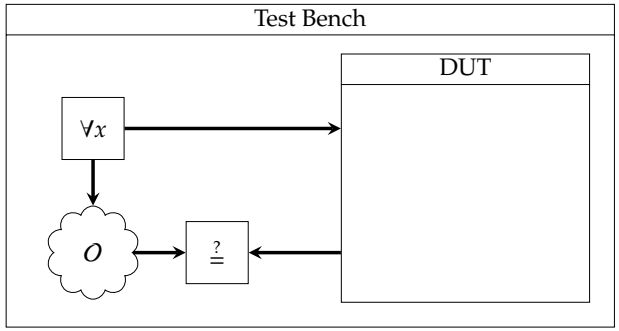
or

$$(x + y) + z \stackrel{?}{=} x + (y + z).$$

where the advantage of doing this is we can just select random x, y and z and check whether the comparison holds (vs. having to construct an explicit set of test vectors). Clearly this concept translates to non-arithmetic cases *if* similar identities are possible.

A case-study: testing a ripple-carry adder (8)
Component #2: a test bench

- Problem: how could we organise the test bench?
- Solution(s):



1. stand-alone test bench $\simeq$ *external* test bench

© Daniel Page ⟨cdp@bristol.ac.uk⟩
Intro. to Computer Architecture git # 3b6f641 @ 2018-01-09 University of BRISTOL

Notes:

- It is obviously possible to enumerate a list of positive and negative features for each approach; one or other may be the better option for a given context. Although non-exhaustive, the following attempts to capture some of the main points:

  1. The stand-alone test bench:

     +ve: can pre-compute test vectors, using an oracle that might be represented by a database, or an alternative implementation, for example,
     +ve: test plan can be adaptive,
     −ve: tests are typically applied after manufacture only, so not close to time of use,
     −ve: is less resource efficient in the sense it does require a dedicated test fixture,
     +ve: is more resource efficient in the sense there is 1 test fixture for n DUTs.

  2. The BIST approach:

     −ve: cannot (as easily) pre-compute test vectors, since the storage requirement is typically too great,
     −ve: cannot cope (as easily) with faults that occur in the test mechanism itself,
     +ve: tests can be applied close to time of use,
     −ve: test plan cannot (as easily) be adaptive, in the sense it must be fixed when the DUT is manufactured,
     +ve: is more resource efficient in the sense it does not require a dedicated test fixture,
     −ve: is less resource efficient in the sense there are n test benches and test fixtures for n DUTs (because they are effectively built into *each* DUT).

- The concept of *arithmetic* BIST, for example relies on testing arithmetic identities such as

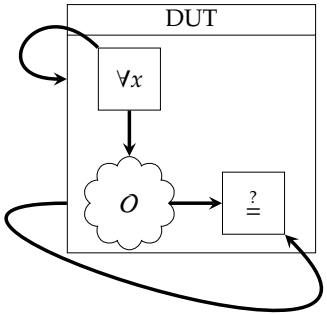
$$(x + y) - (x + y) \stackrel{?}{=} 0$$

or

$$(x + y) + z \stackrel{?}{=} x + (y + z).$$

where the advantage of doing this is we can just select random x, y and z and check whether the comparison holds (vs. having to construct an explicit set of test vectors). Clearly this concept translates to non-arithmetic cases *if* similar identities are possible.

- Problem: how could we organise the test bench?
- Solution(s):



2. **Built-In Self-Test (BIST)** ≃ *internal* test bench

Notes:

- It is obviously possible to enumerate a list of positive and negative features for each approach; one or other may be the better option for a given context. Although non-exhaustive, the following attempts to capture some of the main points:

  1. The stand-alone test bench:

     +ve: can pre-compute test vectors, using an oracle that might be represented by a database, or an alternative implementation, for example,
     +ve: test plan can be adaptive,
     −ve: tests are typically applied after manufacture only, so not close to time of use,
     −ve: is less resource efficient in the sense it does require a dedicated test fixture,
     +ve: is more resource efficient in the sense there is 1 test fixture for n DUTs.

  2. The BIST approach:

     −ve: cannot (as easily) pre-compute test vectors, since the storage requirement is typically too great,
     −ve: cannot cope (as easily) with faults that occur in the test mechanism itself,
     +ve: tests can be applied close to time of use,
     −ve: test plan cannot (as easily) be adaptive, in the sense it must be fixed when the DUT is manufactured,
     +ve: is more resource efficient in the sense it does not require a dedicated test fixture,
     −ve: is less resource efficient in the sense there are n test benches and test fixtures for n DUTs (because they are effectively built into *each* DUT).

- The concept of *arithmetic* BIST, for example relies on testing arithmetic identities such as

$$(x + y) - (x + y) \stackrel{?}{=} 0$$

or

$$(x + y) + z \stackrel{?}{=} x + (y + z).$$

where the advantage of doing this is we can just select random x, y and z and check whether the comparison holds (vs. having to construct an explicit set of test vectors). Clearly this concept translates to non-arithmetic cases *if* similar identities are possible.

Conclusions

- Take away points:

1. On the whole, similar best-practices can be leveraged in software and hardware testing ...
2. ... but a bug in software means re-compilation whereas a bug in hardware means re-manufacturing, so, in a sense

$$\text{verification} \gg \text{testing}.$$

3. Robust methodology and a **design-for-test** ethos ideally form first class requirements:
   - formulate a **test plan**, and
   - consider infrastructure for said plan,

   at the *design* stage, to then make it easier and/or more useful when applied to a DUT.
4. This general area is often viewed as secondary to "getting the job done" (i.e., developing a design); the *costs* involved mean it's a larger element of "the job" than you might expect.

Notes:

References

[1] *Wikipedia: Boundary scan*. URL: http://en.wikipedia.org/wiki/Boundary_scan (see pp. 21, 23).

[2] *Wikipedia: Burn-in*. URL: http://en.wikipedia.org/wiki/Burn-in (see p. 14).

[3] *Wikipedia: Joint Test Action Group (JTAG)*. URL: http://en.wikipedia.org/wiki/Joint_Test_Action_Group (see pp. 21, 23).

[4] *Wikipedia: Oracle*. URL: http://en.wikipedia.org/wiki/Oracle_(software_testing) (see p. 79).

[5] *Wikipedia: Pentium FDIV bug*. URL: http://en.wikipedia.org/wiki/Pentium_FDIV_bug (see p. 9).

Notes: