Design for Testability of FPGA Blocks

Stuart McCracken  
Analog Devices, Inc.  
smcra@macs.ece.mcgill.ca

and

Zeljko Zilic  
MACS Lab, McGill University  
zeljko@macs.ece.mcgill.ca

Abstract

Reconfigurable logic devices that are based on an FPGA substrate are gaining widespread acceptance. As such devices are used in many different configurations, manufacturers need to ensure that each potential configuration will not fail due to device defects. This flexibility leads to severely increased test time. We show how to use reconfigurability to speed up test and diagnosis times of individual FPGA blocks. We present a scheme to incorporate our test architecture, reducing diagnostic and test times of individual FPGA blocks. The test architecture includes added Feedback Shift Registers (FSRs) that change the circuit configuration during test. Algorithms are presented to produce test and diagnosis test sets with a minimized number of test configurations, along with the creation of an FSR that produces the test and diagnosis sets by dynamic reconfiguration of the device.

1. Introduction

Field Programmable Gate Array (FPGA) devices are playing an increasingly important role in many market segments due to their flexibility, which is beneficial both for quick design turnaround times and for adapting a system in the field. As FPGAs are becoming high volume components, maintaining their affordability is critical for continuing to increase their IC market share. Reducing the testing cost is a key to maintaining affordable FPGAs.

In the case of ASICs, the number of test vectors is predetermined as ASIC circuitry is generally static. For reconfigurable logic, however, the large number of possible configurations requires an increased number of test vectors. To reduce the test cost, the number of configurations must be reduced [1].

Equally, FPGAs have complex interrelated structures, which allow for and ensure that the device is highly flexible in terms of its applications. This flexibility is a benefit when using an FPGA, but is detrimental when testing and diagnosing the device. For effective diagnosis, each faulty element must be uniquely identified, which can be costly, but ultimately useful in further device cost reduction using fault tolerance schemes.

This paper is organized as follows. In Section 2, a background of FPGA testing and diagnosis is given, along with a discussion of some pertinent solutions that have been proposed. We present our methodology and algorithms for reducing the test times of FPGA blocks in Section 3, followed by an application to the supplementary logic and interconnect cell in Section 4. Experimental data for several interconnect blocks is presented in Section 5.

2. FPGA testing

FPGAs are based on flexible regular structures, which upon configuration can emulate both sequential and combinational logic circuits. These flexible structures contain four major types of blocks: the logic block (LB), the interconnect structure, the connection block (CB) and the input/output block (I/O).

The LBs perform the task of emulating logic, sequential or combinational, whereas the interconnect structure facilitates routing between LBs. The interconnect structure is formed of switch matrices (SM) and interconnect wires which connect SMs. The CBs connect the SMs to the LBs, whereas the I/Os connect the external world to the SM interconnect structure.

Currently, FPGAs can have over 50 000 LBs, with over 12 000 SMs, and over 800 I/Os [2]. Due to all this circuitry, there are several million configuration bits needed for it to emulate a certain desired circuit. Thus, for each configuration these bits need to be set serially, or in a low order parallel stream, which can take on the order of seconds of test time. If one needs to perform \( n \) configurations to test and diagnose a reconfigurable device, the extremely long configuration time is incurred \( n \) times. Either reducing the configuration time, or reducing the number of configurations required will improve the costs associated with testing a reconfigurable device.

2.1 FPGA testing background

Current methodologies for testing FPGAs are based on establishing sets of the necessary configurations and associated stimuli. These stimuli are either produced by the internal device’s logic or supplied externally through the I/Os [3], [4], [5], [6], [7], [8].

As mentioned, the test time is dominated by the configuration times, where the required number of configurations might be large. Nonetheless, multiple configurations ensure that a certain response pattern will indicate any faulty elements. Thus, the overall number of required configurations, the device configuration time and the test time per configuration establish the test time.
2.2 FPGA diagnosis background

For diagnosing the location of a fault, more configurations are potentially required than for simple testing. The end goal is to create test configurations, which allow for locating a fault [6], [8]. The configuration/signature patterns required for detecting a fault must, therefore, be systematically exclusive of the patterns required for detecting all other faults. This becomes an involved task for fully reconfigurable devices. Diagnosis, nonetheless, can potentially improve yield by marking faulty elements within a reconfigurable device, and applying a fault-tolerant scheme to salvage the device.

2.3 Existing testing schemes

The main drawback of reconfigurable device testing is clearly the necessity of reconfiguring the device for each test case. It is currently possible to reconfigure individual elements in certain types of reconfigurable devices [3]. However, this does not truly help in testing, as cases rarely arise where the differences between test configurations are limited to a few elements. Thus, a full reconfiguration is still necessary along with the time overhead it incurs.

We present a scheme that allows the device to reconfigure itself locally, avoiding the overhead of downloading a new bitstream. Although this requires some additional hardware, the savings in terms of testing time can be considerable.

3. FPGA testing speedup

To reduce FPGA test times, we automate the design of the self-test and self-diagnosis technique considered in [14]. This scheme takes advantage of the reconfigurability of FPGA devices, through the integration of a non-linear feedback shift register (nl-FSR). In [14], we proposed applying the scheme to one type of switch matrix layout. Practical FPGAs have historically used this type of structure, but have moved towards newer and more routable switch matrices. In this paper, we automate and extend the approach for test time reduction to establish the required hardware for various types of FPGA blocks.

The reconfigurable self-test circuit is included locally in each block (SM, LB, CB or other), to act as a feedback shift register (FSR) where the configuration bits act as the registers, and a combinational logic block provides the feedback. The basic FSR structure is shown in Fig. 1. In the case of FPGAs, the registers represent the bits contained in the SRAM memory cells, which are the configuration bits controlling the device functionality. Unlike traditional BIST schemes, the configuration bits B_0 to B_N are changed on the fly by this scheme. Incorporating this hardware into the reconfigurable device resources introduces a certain overhead, but this overhead is outweighed by the benefits in testing/diagnosis time [14].

3.1 Interconnect example

To relate the scheme to a tangible problem, after the technical description of each step, that step is applied to a given interconnect structure. The SM in question is seen in Fig. 2b.

Figure 2: a) Typical Reconfigurable Interconnect  
   b) with Switch Matrices and c) Cross-Points

3.1.1 Interconnect structure

As information is carried from LB to LB or to I/O through the interconnect matrix, this interconnect structure is one of the primary elements that needs to be certified as fault free in reconfigurable devices. This arises from the difficulty and uselessness of testing the LBs if data cannot be routed to them through potentially faulty interconnects.

The structure of a generic reconfigurable device can be seen in Fig. 2. The interconnect structure is composed of SMs which are the routing points connected on all four sides (North, East, South and West). Each SM (Fig. 2b) has a series of cross-points, seen as the grey dots in Fig. 2b, which are shown in more detail in Fig. 2c. These cross-points connect N, S, E, and W inputs through a use of pass-transistors, buffers or transmission gates.

3.1.2 Fault model for SMs

Interconnect testing must take into consideration possible open circuit faults (stuck-open) and bridged wire faults [12]. As for the SM pins, two classes of connectivity exist: those that can and those that cannot be connected through a cross-point. Thus, 3 possible faults can occur. For the non-connected case, faults can occur as permanent connections between pins (bridging faults), whereas for the connected case, they can be permanent connections and permanent disconnections (stuck-open) [12].

3.2 Procedure for test time reduction

The procedure comprises four steps. The first step involves creating a suitable model to establish the required configurations for test or diagnose. The next step is to determine the most useful FSR length. The third step involves finding the appropriate FSR sequence from the
test and diagnosis configuration sets found in step 1. Finally, in step 4, the combinational logic required to create the FSR is established.

The first step is to model the device under test (DUT). A functional model is developed to determine testing sensitivities in a specific configuration. This model is necessary for determining the test configurations and stimuli that are required to test and diagnose the circuit under test. For the case of the sample SM in Example 1, a functional model with $n$ I/Os on each side was developed to model its connectivity pattern and the possible faults. To obtain the test and diagnosis test sets, the model has to account for all pass transistors and circuit functionality.

### 3.2.1 Step 1: Establish test/diagnosis configurations

The initial step involves using the DUT model to determine which configurations are required for testing and diagnosis. Therefore, the DUT model should be reconfigurable, testable, and functional in software. Also, the DUT model needs to determine which elements in a configuration were tested and which elements were not. Equally, it should be able to determine whether or not the current configuration is even a valid test configuration. By cycling through the valid configurations/stimuli, a test table (or fault dictionary) is created, containing the circuit elements and the patterns that verify a potential fault.

Due to the relatively limited size and regularity of the reconfigurable blocks, it is possible to get a complete test table. In ASICs, getting the complete test table is too demanding and fault list simulations need to be used instead. The difference is even more pronounced for diagnosis. In the case of diagnosis, a complete diagnosis table is much too lengthy to produce for any relatively standard sized ASIC design, but is feasible with a reconfigurable device block with a regular structure.

The resulting test table can be used for Test Pattern Generation (TPG) and Diagnosis Pattern Generation (DPG). TPG is discussed first, as DPG relies on much of the TPG concepts.

#### 3.2.1.1 Test pattern generation

Once the DUT Test Table has been formed through model simulation, the minimal cover is found by solving a Unate Covering Problem (UCP), where we start with $t$ faults $F = \{f_1, f_2, \ldots, f_t\}$, and $r$ configurations $V = \{v_1, v_2, \ldots, v_r\}$. The goal of the problem is to cover all $t$ faults with a minimal set of configurations $v$, $C_{\text{test}} = \{c_1, c_2, \ldots, c_s\}$, where each configuration $c$ corresponds to a configuration $v$. The end goal is to have the case where $|C_{\text{test}}|$ is minimal. We aim to achieve the true minimal cover, since the additional configuration times resulting from a non-minimal cover is more costly than the fault list generation costs. Several methods can be applied to solving the UCP, including the Petrick Method [16], [19]. With this method, the potential minimal covers are found. Initial table reductions are achieved by finding essential configurations, row dominance and column dominance.

There are other ways to solve UCPs, including search approximation methods for a cover that is not necessarily minimal [19]. However, as we want minimal cover, we use Petrick’s method. The main argument against using Petrick’s method is the number of intermediate terms. In the case of reconfigurable devices, the number of faults is relatively small for a given block, and the test table is highly regular and includes many easy-to-find vectors due to dominance and essential vectors. Thus, the demands of this UCP problem make Petrick’s method suitable.

**Example 1:** For the sample SM in Fig. 2, the size of the test set table increased proportionally relative to the number of I/Os on each side. However, the required number of test configurations was found to be 3 using this method, regardless of the SM size.

#### 3.2.1.2 Diagnosis pattern generation

To establish the required diagnosis configurations, it is necessary to determine a set of configurations that will set a fault apart from all other faults. To do this, we can take the test set table (or fault dictionary) that has already been created, and establish a diagnosis set table by comparing (XOR) the test set for each fault to every other fault’s test set. If both faults are tested or both are not tested by a configuration/stimulus combination, this would not constitute a valid comparison point. On the other hand, if one is tested and the other is not, this would be an appropriate basis for comparison between the two faults.

Once again, solving a UCP is required to find the minimal diagnosis set. This is achieved in the same manner as for the test set table UCP. However, we can assume a starting point of accepting the minimal test set cover $C_{\text{test}}$, as initial vectors that will be used for the diagnosis set. During manufacturing test, the diagnosis vectors will only be exercised if the testing vectors fail. Thus, the test set is necessarily in the diagnosis set. Equally, by first including the test set in the final diagnosis set, the size of the UCP table is reduced considerably.

**Example 2:** Looking back to the SM of example 1, the diagnosis set was determined to be of length 11, including the 3 test configurations, from the minimal test set.

#### 3.2.2 Step 2: Determine the FSR length

At this point it is necessary to determine the FSR length required to produce the desired response. It is critical that the FSR length be limited, as the hardware overhead of including the new circuitry in the reconfigurable hardware cannot be too large. This is a case-specific problem with a result that differs when dealing with logic blocks, as opposed to interconnect blocks.

**Example 3:** In the case of the sample SM, the ideal FSR length is of size 12, as there are 2 logical sub-groupings of six pass-transistors per cross-point. This number arises from the regularities in the minimal diagnosis set.
3.2.3 Step 3: Create the FSRs

So far, the FSR construction has produced the test and diagnosis sets. To improve the traditional test methods, we now create an FSR, which will perform several tests by self-reconfiguration. This step might require the insertion of added FSR states that are not directly useful for either test or diagnosis. Nonetheless, the penalty incurred by including these states is negligible compared to a true reconfiguration. Furthermore, it is entirely possible to have breaks in the state transitions, where new configurations might be loaded to jump to another state. This is referred to as reseeding [11], [14].

```plaintext
// Main Algorithm
Best FSR Cost = -1
Best FSR = ∅
For I = 1 to Number of Test Configurations
Set 1st FSR entry to Test Config(I)
FSR Create(Test Configs – Test Config(I))
FSR = Best FSR
FSR Create(Diagnosis Configs – Test Configs)
Final FSR = Best FSR

// FSR Create Function
FSR Create(Configs) {
If Configs = ∅
FSR Cost=Number of Reconfig*Reconfig Cost +
FSR Length*FSR Length Cost
If (FSR Cost < Best FSR Cost) or (Best FSR Cost = -1)
Best FSR Cost = FSR Cost
Best FSR = FSR
Return
Else
For J = 1 to Number of Configs
Find input sequence from current FSR State to Configs(J)
Create FSR using sequence
Test New Sequence for inconsistencies
If (inconsistencies=TRUE)
Start New Seed with Configs(J)
FSR Create(Configs-Configs(J))
Else
FSR Create(Configs-Configs(J))

Algorithm 1: FSR Create – Algorithm and Application
```

Due to the nature of the problem, quick testing is important, as diagnosis only follows the testing stage if some tests fail. All the test configurations are preferably exercised with one seed download, ensuring a quick test. Hence, we design an FSR to perform all the test configurations by a single configuration download. We initialize the FSR to one of the test vectors, and find the optimal input sequence of vectors by branch-and-bound search among permutations of vectors.

The remaining diagnosis configurations are built on the solution for the testing phase. The primary objective is minimization of the number of reconfigurations. The number of useless intermediate states is to be minimized as well. In the end, a complete FSR sequence is obtained with potential reseeds. Algorithm 1 shows the pseudo code for creating the FSR in a way reminiscent to the Berlekamp-Massey algorithm [10] for LFSR creation. However, an alternative procedure is required for FSRs since they do not have the inherent linearity and mathematical properties of LFSRs.

**Example 4:** We noticed that the initial restriction of combining the test pattern into one seed proved possible, and as such, the test set went from requiring 3 reconfigurations, down to requiring only one seed (configuration). The diagnosis set equally was able to fit into the same seed. Thus, the 11 initial configurations were reduced down to one seed.

3.2.4 Step 4: Determine the combinational logic

Once the FSR sequence has been determined, it is a simple exercise to create the required combinational logic. Any logic synthesis tool, which in our case is SIS [15], can be used to create the appropriate design.

**Example 5:** The results for the combinational logic as a function of the configuration bits is given in SOP form:

\[ C_{log}ic = b_1 \cdot b_3 \cdot b_5 \cdot b_7 + b_1 \cdot b_5 \cdot b_9 \cdot b_{11} \]

Here, \( C_{log}ic \) is the circuit required for all the FSRs. As apparent, the resulting logic is fairly simple.

4. FSR scheme applied to SLICs

To illustrate the application of the scheme to other FPGA blocks, we consider a block found in some FPGA architectures known as the supplementary logic interconnect cell (SLIC) [20].

**Figure 3: General SLIC Structure [20]**

4.1 SLIC functionality

The SLIC creates a more versatile signal routing structure, while allowing for additional logic functions. The SLIC has been integrated into Lattice’s ORCA 3T and 3C FPGAs [20]. As can be seen in Fig. 3, the SLIC can act as a buffer or a decoder. When in buffer mode, the data is tri-statable and bidirectional through the BR and BL lines. Also, the decoder output can be used for further logic functionality. Equally, a mix of both modes can be used as the I/O lines are grouped into 3 subgroups.
Therefore, as there are 3 groups with 2 possible functions (buffer/decoder), there are 8 possible SLIC configurations.

4.2 SLIC testing

There are essentially 4 groupings of modes that need to be investigated to test the SLIC in all its states: when they are all buffers, when they are acting as buffers and are tri-state enabled, when they are all decoders, and when each acts partly as a buffer and partly as a decoder. The testing of all the SLICs acting as buffers is achieved by creating a chain of SLIC cells, where the even numbered rows direct the data from right to left and the odd numbered rows direct the data from left to right. This chain of buffers allows for the testing of s-a-0 and s-a-1. As for testing the SLIC in decoder mode, each SLIC can be tested for any s-a-0 and s-a-1 fault with 19 configurations. The output of each SLIC is directed towards a comparator circuit, which determines whether the output of each SLIC is the same. This test set allows for the testing of the different functions of the SLIC. After finding the individual configuration test sets, the algorithm for the construction of FSR is applied. The results of applying the proposed method will be summarized in Section 5.2.

5. Experimental results

5.1 Switch module examples

![Switch module examples](image)

Figure 4: a) Disjoint, b) Wilton and c) Universal SMs

Using the proposed scheme and algorithm for nl-FSR construction, we recorded the test time reduction and the hardware overhead for several commonly used switch matrix layouts. Three of the patterns use a one-to-one mapping between I/Os on each side. This is the case for the Disjoint, Wilton, and Universal Switch Matrices as seen in Fig 4. This figure shows the connectivity between ports, where each cross-cutting line between I/Os is a pass-transistor. Equally, certain advances in this area involve the integration of buffers as well as pass transistors [13].

![Switch module examples](image)

Figure 5: Variant a) #1, b) #2 and c) #3 SM Structures

Another variant used for benchmarking is the Buffered SM. For this type of SM, each pass-transistor is replaced by two tri-state buffers, one for each direction [13]. Finally, the last three variants used for benchmarking are more populated cross-bars. The cross-bar populations used are shown in Fig. 5. As can be seen in Fig. 5, Variant #2 is in fact the SM described in Section 3 and Fig. 2.

The algorithms were applied to these 7 different SM structures to show the benefits of this scheme. The results are summarized in the tables containing the time savings for testing, the time savings for diagnosis, and the hardware overhead incurred for these benchmarks. From the testing results in Table 1, it can be seen that regardless of the topology of the SM, as long as it is pass-transistor based, testing uses the same approach. This further proves the statements in [14], that the testing time will be reduced by 66%. For the case of the buffered SM, the savings are greater, as the larger number of test configurations can be compacted into only one seed.

Table 1 - Testing Benchmark Results for NL-FSR Insertion

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of Test Configs Without FSR</th>
<th>Number of Test Configs With FSR</th>
<th>Test Time Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disjoint</td>
<td>3</td>
<td>1</td>
<td>66%</td>
</tr>
<tr>
<td>Wilton</td>
<td>3</td>
<td>1</td>
<td>66%</td>
</tr>
<tr>
<td>Universal</td>
<td>3</td>
<td>1</td>
<td>66%</td>
</tr>
<tr>
<td>Buffered</td>
<td>6</td>
<td>1</td>
<td>83%</td>
</tr>
<tr>
<td>Variant #1</td>
<td>3</td>
<td>1</td>
<td>66%</td>
</tr>
<tr>
<td>Variant #2</td>
<td>3</td>
<td>1</td>
<td>66%</td>
</tr>
<tr>
<td>Variant #3</td>
<td>3</td>
<td>1</td>
<td>66%</td>
</tr>
</tbody>
</table>

In the case of diagnosis, in Table 2, it is apparent that the savings vary from topology to topology. Nonetheless, applying the algorithm and including the FSR circuitry reduces the diagnosis time by at least 72% in all cases.

Table 2 - Diagnosis Benchmark Results for NL-FSR Insertion

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of Diag. Configs. Without FSR</th>
<th>Number of Diag. Configs. With FSR</th>
<th>Diagnosis Time Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disjoint</td>
<td>11</td>
<td>3</td>
<td>72.7%</td>
</tr>
<tr>
<td>Wilton</td>
<td>11</td>
<td>3</td>
<td>72.7%</td>
</tr>
<tr>
<td>Universal</td>
<td>11</td>
<td>3</td>
<td>72.7%</td>
</tr>
<tr>
<td>Buffered</td>
<td>22</td>
<td>3</td>
<td>86.4%</td>
</tr>
<tr>
<td>Variant #1</td>
<td>11</td>
<td>1</td>
<td>90.9%</td>
</tr>
<tr>
<td>Variant #2</td>
<td>11</td>
<td>1</td>
<td>90.9%</td>
</tr>
<tr>
<td>Variant #3</td>
<td>15</td>
<td>1</td>
<td>93.3%</td>
</tr>
</tbody>
</table>

Looking at the results, it is noticeable from Table 3 that the testing and diagnosis savings are the same in the case when there is a one-to-one mapping between sides (as with Disjoint, Wilton and Universal). This arises from the fact that these three structures can be obtained from the same topology by permuting the pins. Furthermore, in Table 3, we recorded test and diagnosis time reductions, along with the required length of the FSR. Also, the logic complexity of the required circuitry is indicative of the hardware overhead involved in adding the FSR. We conclude that as the SM topology gets more complicated, the length of the FSR gets larger to compensate for the increased complexity of diagnosis.

To find the logic complexity, we used SIS [15] to perform technology mapping. Designs were optimized for area using the script.rugged script and the
minimal.genlib library. The metric for comparison was the final number of literals. We notice that, as the size of the FSR increased, so did the complexity of the logic. The Buffered case is the exception, due to its similarities to the disjoint structure upon which it is based.

The Buffered case is the exception, due to its similarities to the disjoint structure upon which it is based. The Buffered case is the exception, due to its similarities to the disjoint structure upon which it is based.

---

**Table 3 - Benchmark Results for NL-FSR Insertion**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Test Time Savings</th>
<th>Diagnosis Time Savings</th>
<th>FSR Length</th>
<th>Complexity (# Literals)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disjoint</td>
<td>66%</td>
<td>72.7%</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>Wilton</td>
<td>66%</td>
<td>72.7%</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>Universal</td>
<td>66%</td>
<td>72.7%</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>Buffered</td>
<td>83%</td>
<td>86.4%</td>
<td>12</td>
<td>17</td>
</tr>
<tr>
<td>Variant #1</td>
<td>66%</td>
<td>90.9%</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>Variant #2</td>
<td>66%</td>
<td>90.9%</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>Variant #3</td>
<td>66%</td>
<td>93.3%</td>
<td>18</td>
<td>73</td>
</tr>
</tbody>
</table>

5.2 SLIC example

The presented methodology of test configuration reduction can be used to test the SLIC block, described in Section 4, in buffer and decoder modes. The results can be seen in the following table.

**Table 4 – SLIC Testing Results for NL-FSR Insertion**

<table>
<thead>
<tr>
<th>SLIC Operating Mode</th>
<th># of Configs Without FSR</th>
<th># of Configs With FSR</th>
<th>Test Time Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>4</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>Decoder</td>
<td>19</td>
<td>2</td>
<td>89.5%</td>
</tr>
</tbody>
</table>

It is apparent that the described methodology for FSR creation and FSR state minimization is useful in reducing test times. In the case of the decoder, for instance, it was determined that essentially 2 FSR seeds are required for 19 test configurations. In total, 30 steps are necessary to cycle through all the test configurations. This represents a true test time saving and shows how our scheme of FSR integration can be expanded to other FPGA circuitry.

6. Conclusions

We have shown that reconfigurable logic can gain from the application of dynamic reconfigurable self-test. We applied the proposed procedures, for integrating dynamic reconfigurable self-test into a reconfigurable device, for the interconnect structure and for SLIC cells. We have shown that it is possible to reduce test time by more than 60% for a variety of interconnect structures. Furthermore, the algorithms presented in Section 3.2.1 alone are sufficient for producing the test and diagnosis vector sets as a byproduct, if the additional hardware and test time savings are not required.

7. References


