A Plan for Router Testing in Network-On-Chip

Zachariah Thomas Link

University of Nebraska-Lincoln

Follow this and additional works at: http://digitalcommons.unl.edu/opentheses

Part of the Engineering Commons


http://digitalcommons.unl.edu/opentheses/45
A Plan for Router Testing in Network-On-Chip

By

Zachariah Thomas Link

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Telecommunications Engineering

Under the Supervision of Professor Chunsheng Liu

Lincoln, Nebraska

December 2004
A Plan for Router Testing in Network-On-Chip

Zachariah Thomas Link, M.S.
University of Nebraska, 2004

Adviser: Chunsheng Liu

Network-On-Chip (NoC) is becoming a promising paradigm for core-based VLSI systems. NoCs rely on an on-chip network to provide high performance communication among cores. Testing of NoC-based VLSI systems poses considerable challenges, especially for the testing of the network components, e.g. routers and interconnections.

This thesis presents a new method for testing the routers in a packet switch on-chip network. The method relies on a progressive reuse of the network resources for test data transportation, which can significantly reduce overall test cost.

This thesis first presents a scheme for router testing by progressively reusing the network. Possible test wrapper architecture is then depicted to show necessary requirements on wrapper implementation. Next, a new test-scheduling algorithm is presented. This algorithm integrates router testing with core testing such that the two processes can be carried out simultaneously. This resulting test time is reduced. Finally, experimental results on ITC ’02 SoC benchmarks are presented to show the effectiveness of the proposed method. Compared to the previous scheme based on the use of boundary scan, the new method in this thesis leads to substantial reduction on test application time of the system. It also helps reduce hardware overhead and ATE pins.
# Table of Contents

Chapter 1 .............................................................. 1
Introduction .......................................................... 1
  1.1 VLSI and System-On-Chip ................................. 1
  1.2 Network-On-Chip ............................................. 2
  1.3 NoC System Testing .......................................... 5
Chapter 2 .............................................................. 7
Router Testing ........................................................ 7
  2.1 Boundary Scan ................................................ 7
  2.2 Router and Input Reuse ...................................... 9
  2.3 Test Wrapper Architecture ................................. 10
  2.4 Router Testing Algorithm ................................. 11
Chapter 3 .............................................................. 14
Combining Router Testing and Core Testing .................... 14
  3.1 Core Test Scheduling ....................................... 14
  3.2 Integrated Router and Core Scheduling ................. 15
Chapter 4 .............................................................. 20
Experimental Results .............................................. 20
  4.1 ITC '02 Benchmarks ......................................... 20
  4.2 Simulation Results ........................................... 23
Chapter 5 .............................................................. 27
Future Work ........................................................ 27
  5.1 Possible Improvements ..................................... 27
  5.2 Uncontrolled Obstacles .................................... 28
  Conclusion ......................................................... 29
References .......................................................... 30

Appendix A .......................................................... 33
Sample Simulation Results ....................................... 33

Appendix B .......................................................... 34
Simulation Source Code (Example) ............................. 34
List of Figures

Figure 1: NoC Example................................................................. 2
Figure 2: X-Y Routing Example.................................................. 4
Figure 3: Boundary Scan Approaches a) 1x1 b) 2x2 .................. 8
Figure 4: Router Testing with Wrapper ..................................... 9
Figure 5: Wrapper Example for Core and Router ...................... 11
Figure 6: Router Testing Groups a) Multicast b) Unicast .......... 12
Figure 7: 4 Input Router Testing Groups ................................. 13
Figure 8: Core Scheduling Pseudo Code................................. 15
Figure 9: Integrated Router and Core Scheduling Pseudo Code..... 16
Figure 10: Integrated Core and Router Scheduling Example ....... 17
Figure 11: Pictorial View of Integrated Router and Core Scheduling 18
Figure 12: Possible NoC Architecture for Benchmark d695 ....... 21
Figure 13: Possible NoC Architecture for Benchmark g1023 ...... 21
Figure 14: Possible NoC Architecture for Benchmark p22810 .... 22
Figure 15: Possible NoC Architecture for Benchmark p93791 ...... 22

List of Tables

Table 1: Boundary Scan Comparison ........................................ 23
Table 2: Simulation Results Based on Average of Core Test Time 25
Table 3: Average of Core Test Time/10 Simulation Results ............. 26
Chapter 1

Introduction

This thesis presents a plan for testing routers and cores on Network-On-Chip (NoC) systems. It is important to understand the foundation of NoC before detailing the specifics on router and core testing.

1.1 VLSI and System-On-Chip

Very Large Scale Integrated (VLSI) is the primary design paradigm for today’s integrated circuits (ICs). As design and processing techniques improve, design complexity continues to increase and device size has decreased significantly. It is now possible to integrate ten or hundreds of millions of transistors on a single silicon chip.

VLSI design has allowed for the introduction of System-On-Chip (SoC). An SoC integrates various functional blocks, such as microcontroller and memory units, input and output interfaces, and digital signal processing (DSP) units, all known as cores, on a single silicon chip. The use of SoC design diagram substantially reduced design cost increased performance and led to short time to market of VLSI systems. It also helps to protect Intellectual Property (IP). However, SoC has also brought about new problems. Among them are increased clock skew, long wiring delays, high power/energy dissipation, and complicated designs. For example, interconnection wires from one core to another could be very long, which creates problems with heat and power dissipations, as well as signal delays from the source to the destination. The more complex the SoC is,
the more wires and connections and are needed and system performance is significantly limited.

As the number of cores in an SoC increases, interconnection has become the new bottleneck of system performance. It has been shown that traditional point-to-point and shared-bus architectures can no longer provide the future SoCs with scalable performance [1, 2, 7].

1.2 Network-On-Chip

Network-On-Chip (NoC) has been introduced as a promising interconnection architecture of the next generation of SoCs. The basic idea of NoC is similar to the idea behind a computer network. An NoC consists of a set of cores and routing mechanisms, or routers in the case of a packet-switched network infrastructure. In a packet-switched implementation, each router in an NoC is connected to other routers in a matrix formation. Figure 1 gives an example of a conceptual NoC.

![Figure 1: NoC Example](image_url)
As presented in [7], NoCs typically use the message-passing communication model. Again, similar to a computer network, messages are sent as packets called flits. Each flit contains a header, payload (data) and a trailer. The header contains information to form a path between the source and destination, the payload contains the data for the destination to receive, and the trailer contains information for parity or for a cyclic redundancy check (CRC). Packet-based networks provide better utilization of network resources in an NoC [8].

In addition to topology, NoC is also described by its approaches to packet-based network in terms of routing, switching, and buffering [9]. Buffering is the mechanism to store incoming or outgoing messages when the core is busy or output channel is busy, respectively. The switching mechanism defines what a router's basic duty is, to receive an incoming packet from the input and transmit this packet on the appropriate output. The routing mechanism determines how messages from the source travel to the destination [8].

In this thesis, all work is based on a packet-switched network introduced in [10] implemented in a 2-D mesh topology. XY routing [9] is the routing mechanism. XY routing takes advantage of the NoC layout, which is often visualized in a 2-dimensional mesh topology. This routing algorithm gets its name from the way a path is determined from the source router to destination router, using Cartesian coordinates to reference the locations of routers and cores within a NoC. The path is determined by finding the horizontal path to the destination (X direction), followed by the vertical path to the destination router (Y direction). Figure 2 shows XY routing in a conceptual 3x3 NoC.
In Figure 2, the lower left hand corner is the source router (0,2) and the upper right hand corner is the destination router (2,0). The X direction is found first by subtracting the X coordinate value of the destination location from the X coordinate value of the source location. In this example, the value of the X direction is 2. If the value is positive, the direction of the path is to the right. If the value is negative, the direction is to the left. However, if the value is zero, only the vertical or Y direction is needed. Once the X direction is found, the Y direction needs to be found in the same manner by subtracting the Y value of the destination address from the Y value of the source address. In this case the Y value is a negative 2. This means there are two hops in the up direction. The resulting X-Y Routing path from source to destination is: (0,2), (1,2), (2,2), (2,1), (2,0).
1.3 NoC System Testing

Although NoC Design has been extensively studied, the testing of NoC-based systems has not received much attention. Testing of an NoC-based system can be classified as testing the cores and the network infrastructure, specifically the routers.

Algorithms for testing core-based NoC systems have been studied recently in [12, 13, 14, 15]. These methods presented for core testing involved assume a bus-based test access mechanism (TAM) and were mostly limited to time minimization. A cost effective bus was proposed in [16, 17]. This approach considered variables such as power, chip area for the TAM, and other system constraints in addition to test time minimization.

NoC router testing has received only limited attention. Prior work [18] treats routers as an individual circuit and applies a traditional boundary scan approach for testing. This methodology requires additional hardware and input pins for automated test equipment (ATE). This results in increased cost and test time. As NoC matures, the design complexity will increase and require more complex testing. This trend demands a testing methodology involving easier test access and minimized test time.

Testing the routers within a NoC is challenging. In traditional testing, a dedicated test access mechanism is used to send and receive test data to and from cores. This TAM requires additional on-chip hardware that leads to increased hardware overhead and cost. For an NoC system, however, new approaches have been proposed to reuse the existing network for testing both the routers and cores. This reuse methodology has been addressed in [1, 2]. It has been shown that reusing the network for testing not only
reduces the dedicated hardware used for TAMs, but also reduces the overall test time.

This thesis proposes a new approach to test both cores and routers by reusing the network already present in an NoC. It shows that progressively reusing the network and scheduling core and router testing simultaneously, testing time can be significantly reduced. It also helps to reduce the need for dedicated testing hardware.
Chapter 2

Router Testing

Before the cores in an NoC can be tested, the network has to be tested fault-free. Note that this thesis neglects the testing of interconnections, which is relatively easy, but focuses on the testing of routers.

2.1 Boundary Scan

Boundary scan is formalized as IEEE standard 1149.1. This standard was needed as a result of newer technologies developing cores with larger pin counts. The standard specifies protocols for easy test access to individual cores in the system through the use of dedicated shifted in/out and internal shift registers.

In [5], a router testing scheme for NoC using boundary scan is presented. The main components include the Test Access Port (TAP) controllers, shift registers and comparators. The TAP controller provides the interface from the electronic circuit being tested (cell) to the outside world. The TAP controller sends appropriate test data to shift registers which are surrounding the core, in serial. These shift registers send the appropriate test data from the TAP controller to the cell. Output signals from the core also feed into shift registers which send the output results to a comparator. This comparator compares the outputs from several cores to determine if the test was successful [5].
Figure 3: Boundary Scan Approaches a) 1x1 b) 2x2

Two approaches, cell-based and cluster-based, are proposed as shown in Figure 3. Figure 3a uses boundary scan to test a single router or core by itself. This is known as the Cell-based approach. An advanced approach to boundary scan involves grouping similar cores to reduce testing time. Figure 3b shows a 2x2 group of routers being tested by boundary scan. This is known as the Cluster-based approach. The procedure to test the routers (or cells) in a group is similar to the method described above. However, in a group of routers the testing is done in parallel. Note the comparators required for analyzing test results on-chip are not shown in Figure 3, but would be connected to the TAP controllers. The exact size of the groups is determined by several factors such as the number and size of the cells, the cost of the circuitry to test the group, and the ability to determine the location of a fault, or failed test [5].

There are some disadvantages to the boundary scan approach. First, additional hardware is required for ATE pins and internal shift registers. Boundary scan results in a large test time since test data is sent to the shift registers in serial. As NoC design matures, routers may require more complex testing for future additional capabilities such as improved routing and switching schemes [6].
2.2 Router and Input Reuse

This thesis proposes reusing the routers in a NoC for testing the router infrastructure. This approach is similar to the approaches discussed in [1, 2]. Automatic test equipment (ATE) sends test data in packets to the router through the input pins. These packets are routed through the internal routing infrastructure and channels, until the test packet reaches the destination router. All routers are assumed to be surrounded by a small amount of additional circuitry so called test wrapper. This wrapper accepts the test data from the ATE and applies the test data on the router, configured in test mode. The wrapper receives test data in packets called flits (flow control units). Each flit is a section of a complete packet containing a header, payload and a tail. The header contains control information for the wrapper to be properly configured in a specific test mode, the payload contains test data for the router, and the tail defines the end of the packet. The response of the test can be sent back to the ATE or processed on the chip. Figure 4 illustrates wrappers accepting test data and transmitting the test response through two input ports.

![Figure 4: Router Testing with Wrapper](image)
Reusing the input ports and routers to test the routing infrastructure, provides significant advantage over the method using boundary scan. Reuse provides parallelism for the testing procedure. Since the routers and channels are being used to send the test data, the available bandwidth is equal to the size of the channels. In this example, the channel size is 32 bits, which is wider compared to boundary scan, test data is delivered from the TAP to the shift registers in serial. It is easily seen that testing the routers in parallel with large bandwidth allows the total test time to be significantly reduced. In addition to reduced test time, the amount of additional hardware is also limited only to the size of the wrapper [6]. There is no requirement for dedicated ATE pins or on-chip hardware as used in boundary scan.

2.3 Test Wrapper Architecture

The wrapper used in router testing needs to integrate additional functions in addition to the IEEE P1500 compliant wrapper architecture [3]. It is possible for the wrapper to be integrated to include the router and the core, rather than only the router as assumed above. This implementation could simplify hardware and reduce cost. An example of this integrated wrapper is shown in Figure 5.

The main additional functions needed by the test wrapper are packing and unpacking. These are not specified by the P1500 standard, but are required for both testing and functional mode. This wrapper accepts the flits from the ATE and unpacks the control and test data to be used by routers and cores. With this information determined, the testing would then be carried out according to the P1500 standard.
When a test response is ready to be transmitted, the response data must be assembled in the form of a packet before being sent to its destination through the network infrastructure. This may not be required for router testing if the response is handled on chip. However, the packing function is needed for core testing as the response is sent to an output port [6].

2.4 Router Testing Algorithm

The algorithm for testing routers is based on a progressive reuse of the routers for testing. First, the routers connected directly to the input cores are tested. It is assumed that the test responses can be processed on-chip. Therefore the output ports are not used. Next, neighboring routers are identified and tested as a group. Note that these routers are now accessible from the input ports. After this group is tested, another group of routers that are accessible are selected. Therefore every time a group of untested routers that can be directly accessed from the input ports, or through the tested routers, are being tested.
This process continues until all routers are tested. Figure 6 shows a generic test process using this progressive grouping. Different numbers are used to indicate routers in a different group.

Figure 6: Router Testing Groups a) Multicast b) Unicast

It is important to note that the scheme shown in Figure 6a assumes a multicast network, where test data can be routed from one source router to multiple destination routers. In case of a unicast network, where one source can access only one target, more groups are needed. This is shown in Figure 6b.

Figure 7 shows an example of a NoC with four inputs. The matrix is based on the d695 benchmark from the ITC'02 benchmarks, which will be discussed in the future chapters. The numbers shown on this figure are not the router groupings as shown in Figure 6. The numbers on Figure 7 show the groupings of the routers that are actually tested by an input.
Figure 7: 4 Input Router Testing Groups

Figure 7 shows group 1 containing 4 routers directly accessible from the input ports. This group will be tested first. The four routers in group 2 are then directly accessible through the tested routers in group 1. Note there are only 4 routers in group 2. This is due to the assumption of a unicast network hence each group can only have a maximum of 4 routers. In group 3, the three shaded routers could be tested. However, the XY routing causes a conflict on the use of the channels. Therefore, only two routers can be tested in this group while the third will be tested in group 4. For the same reason, only one router can be tested in groups 4 and 5. These routers are the group of routers directly connected to the input cores. These routers are always tested first. There are also 4 routers shown in group 2.
Chapter 3

Combining Router Testing and Core Testing

It can be seen that core testing and router testing are based on the reuse of the network. Moreover, the two processes can be integrated such that routers and cores can be tested simultaneously, and test time can be further reduced.

3.1 Core Test Scheduling

Core test scheduling algorithms in NoC are presented in [1, 2]. A straightforward method of testing the routers and cores is to test all routers before scheduling any cores. The scheduling algorithm presented in [1, 2] assumes the network infrastructure is fully functional or has already been tested in this manner. The pseudo code in Figure 8 demonstrates how core testing is scheduled once all routers have been tested.

The first step is sorting all cores in the NoC in decreasing order of test time, hoping that scheduling cores with a longer test time first can lead to less test time. Step 2 permutes the orders of the I/O pairs and the best result is selected from all permutations. For each permutation of I/O pairs, all cores are attempted to be scheduled. For each core, the algorithm examines all I/O pairs to find a free pair. If no free I/O pair is available, the current time is updated and the algorithm repeats the search for a free I/O pair. Once a free I/O pair is found, the path is checked for availability. Even if a free I/O pair is available, the path from the input to the core and from the core to the output can still be blocked if a channel along the path is currently being used for another test (channel
conflict). If there is a free I/O pair but the path is blocked, the algorithm will select the next core for scheduling. If all cores have been attempted at the current time but the path is always blocked, the current time is updated and the algorithm restarts at line 6.

1. Sort cores in decreasing order of test time;
2. Permute all possible orders of I/O pairs;
3. For every permutation
4. While there are unscheduled cores
5. For each unscheduled core
6. Find a free I/O pair;
7. If no free I/O pair
8. Update current time, repeat from 6;
9. Else
10. Check the corresponding routing path;
11. If path is blocked
12. If all cores have been attempted
13. Update current time, repeat from line 6;
14. Else
15. Try next core on the list;
16. Else
17. assign core to the path, update time labels;
18. Update time labels.

Figure 8: Core Scheduling Pseudo Code

A successful scheduling results from a free I/O pair being found for a core with an available path. Each successful scheduling is completed by updating time labels on the appropriate I/O pair, core and channels used for this test. This insures the I/O pair, core and channels will not be used again until this particular core has been tested. This process continues until all cores have been scheduled.

3.2 Integrated Router and Core Scheduling

The algorithm in the last section can be improved to integrate router and core
testing, allowing core scheduling to take place concurrently with router testing. In Figure
6, one can observe that after group 3 a path is free from Input 1 to Output1 and from
Input 2 to Output 2, with all routers on these paths tested. Therefore, cores on these paths
can be tested from this point of time, along with the remaining untested routers. An
improved algorithm is shown in Figure 9. Note this schedule algorithm is referenced as
an algorithm, but does not guarantee the most efficient schedule in all cases. However, in
all cases a schedule will result.

1. Sort cores in decreasing order of test time;
2. Permute all possible orders of I/O pairs;
3. For every permutation
4. Do router testing while no free routing path for core testing
5. While router testing unfinished and at least at least one I/O pair
   used for testing routers
6. While there are unscheduled cores
7. For each unscheduled core
8. Find a free I/O pair;
9. If no free I/O pair
10. Update current time, repeat from 6;
11. Else
12. Check the corresponding routing path;
13. If path is blocked
14. If all cores have been attempted
15. Update current time, repeat from line 6;
16. Else
17. Try next core on the list;
18. Else
19. assign core to the path, update time labels:
20. Update router status, update time labels.

Figure 9: Integrated Router and Core Scheduling Pseudo Code

Since routers need to be tested before cores can be scheduled, there is a period of
time before core scheduling where only routers are scheduled, shown by line 4 in Figure
9. This is extremely similar to the router testing algorithm previously presented in
Chapter 2. Here the routers directly connected to the input ports are tested first. Routers continue to be grouped and tested as described earlier. The difference is seen when an output router is tested. This signals an I/O pair may be available to test cores and the algorithm starts a combined phase, where routers and cores are scheduled simultaneously.

This combined phase, beginning on line 5 of Figure 9, first attempts to schedule a core on any possibly free I/O pair. The core scheduling is performed similarly as described in Figure 8, but some exceptions are made. First, this algorithm always reserves at least one I/O pair for testing routers. Therefore in a NoC with 4 I/O pairs, a maximum of 3 I/O pairs at any time can be used to schedule cores until all routers have been tested. Secondly, after an attempt to schedule each cores is complete, the previous algorithm would simply go back to searching for free I/O pairs. However, in this integrated approach, the free I/O pairs are used to schedule untested routers. This allows both routers and cores to be scheduled simultaneously. This concurrent testing continues until all routers have been scheduled. After that, all I/O pairs are used for core scheduling and the core scheduling algorithm described in section 3.1 is used until all cores are scheduled.

Figure 10: Integrated Core and Router Scheduling Example
Figure 10 illustrates this integrated algorithm. Groups 1, 2, and 3 are groups of only routers being scheduled, corresponding to line 4 in Figure 9. Once group 3 has been scheduled, there are 2 I/O pairs available since the routers on Input 1, Input 2, Output 1, and Output 2 have all been tested. This begins the combined phase of the algorithm. Only one I/O pair can be used to schedule cores while the other I/O pair is used to schedule the cores marked with numbers 4 through 9. Assuming Input 2 and Output 2 are used as a pair, they will be used for core testing. This I/O pair could be used to schedule any core as long as the required routers are tested and there is no resource conflict. An example of available cores would be the cores attached to the routers along the bottom of Figure 10.

While this I/O pair is used for scheduling cores, Input 1 and Output 1 are continually schedule the routers in the top two rows marked 4 through 9. Once these routers have been scheduled, router scheduling is complete and only cores are being scheduled. Both I/O pairs are used for scheduling the remaining cores until all cores are scheduled.

Figure 11: Integrated Router and Core Scheduling on d695
Figure 11 illustrates the integrated scheduling through an example. The data was gathered from simulation results based on the d695 benchmark, using 4 I/O pairs. This benchmark and others are discussed in the next chapter. It can be seen that the entire scheduling can be viewed as a three stage process: router only testing, router and core concurrent testing and core only testing. Idle time is represented by the shaded areas.
Chapter 4

Experimental Results

In this section, experiments are performed to examine the improvement of the proposed reuse method over the boundary scan approach as well as the integrated scheduling over the separate scheduling schemes.

4.1 ITC '02 Benchmarks

The benchmarks used for these simulations were first presented at the IEEE International Test Conference (ITC'02) in October 2002. These benchmarks are provided and updated by companies such as Hewlett-Packard, Faraday Technologies, Texas Instruments and Philips Semiconductors and Universities such as Duke University and University of Stuttgart. The benchmarks include the number of cores, routers, inputs, outputs and length of chains and tests for each core.

There were 4 main benchmarks used in the experiments: d695, g1023, p22810, and p93791. The d695 benchmark is a 4x3 network containing 12 routers and 10 cores, contributed by Duke University. An example of this benchmark using NoC architecture with 2 I/O pairs is shown in Figure 12. Note that the groups of routers are numbered in the order of testing.
Figure 12: Possible NoC Architecture for Benchmark d695

The g1023 benchmark is a 4x4 network containing 16 routers and 14 cores, contributed by the University of Stuttgart. An example with 3 I/O pairs is shown in Figure 13.
The p22810 benchmark is a 4x6 network containing 24 routers and 23 cores, contributed by Philips Semiconductors. An example with 4 I/O pairs is shown in Figure 14.

The p93791 benchmark is a 3x5 network containing 15 routers and 14 cores, contributed by Philips Research. An example with 2 I/O pairs is shown in Figure 15.
4.2 Simulation Results

Two Simulations were run to compare testing times. In the first group of simulations, the router grouping approach presented previously in this thesis is compared to the boundary scan approach. These methodologies were used to simulate testing all routers in each of the four benchmarks.

The results of the simulations are shown in Table 1. Each benchmark was simulated for 2, 3, and 4 I/O pairs. A percentage is given showing the increase in test time using the new algorithm compared to the boundary scan approach.

<table>
<thead>
<tr>
<th></th>
<th>Boundary Scan</th>
<th>Network Reuse</th>
<th>Percentage of Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>d695</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2/2</td>
<td>65970</td>
<td>26376</td>
<td>60.02%</td>
</tr>
<tr>
<td>3/3</td>
<td>65970</td>
<td>22608</td>
<td>65.73%</td>
</tr>
<tr>
<td>4/4</td>
<td>65970</td>
<td>18840</td>
<td>71.44%</td>
</tr>
<tr>
<td>g1023</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2/2</td>
<td>34763</td>
<td>28616</td>
<td>17.68%</td>
</tr>
<tr>
<td>3/3</td>
<td>34763</td>
<td>21462</td>
<td>38.26%</td>
</tr>
<tr>
<td>4/4</td>
<td>34763</td>
<td>14308</td>
<td>58.84%</td>
</tr>
<tr>
<td>p22810</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2/2</td>
<td>306525</td>
<td>295992</td>
<td>3.44%</td>
</tr>
<tr>
<td>3/3</td>
<td>306525</td>
<td>197328</td>
<td>35.62%</td>
</tr>
<tr>
<td>4/4</td>
<td>306525</td>
<td>147996</td>
<td>51.72%</td>
</tr>
<tr>
<td>p93791</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2/2</td>
<td>1999300</td>
<td>1311119</td>
<td>34.42%</td>
</tr>
<tr>
<td>3/3</td>
<td>1999300</td>
<td>1006698</td>
<td>49.65%</td>
</tr>
<tr>
<td>4/4</td>
<td>1999300</td>
<td>770046</td>
<td>61.48%</td>
</tr>
</tbody>
</table>

Table 1: Boundary Scan Comparison

The results in Table 1 show network reuse reducing the total test time when compared to the boundary scan approach. These simulations show the reduction in test time can be as high as 61%!
In the second simulation, router testing is scheduled separately from core testing, i.e., the system test time is a sum of the two. The third simulation was based on the integrated algorithm. This tests routers with all inputs until an output was available for an I/O pair to test cores. Once such an I/O pair was found free, the concurrent testing of routers and cores can begin. Both simulations were run for 2, 3, and 4 I/O pairs resulting in 6 total simulations for each benchmark. Tables 2 and 3 list the simulation results by benchmark and number of I/O pairs for the second simulation and the third, respectively.

In Table 2, we set the router testing time required for a router to the average of the testing times of all the cores. While in Table 3, smaller routers are assumed, whose testing time is 1/10 the average of the testing times of all the cores. For each benchmark, we show the separate testing time for routers and cores and then the combined testing time when the integrated scheduling is used.

As shown in the results, the integrated algorithm does show an improvement over the straightforward approach. In almost every case, the testing time for the integrated algorithm is either comparable to router testing separate from core testing, or much less. This improvement of the integrated algorithm is shown as a test time reduction percentage.

Table 3 shows slightly different results. This time there is still improvement using the new algorithm compared to the straightforward algorithm for benchmarks d695 and g1023. However, for the other two benchmarks, p22810 and p93791, almost all simulations were worse for the new algorithm compared to the straightforward algorithm. This can be seen by viewing the test time reduction percentage. The p22810
and p93791 benchmarks use cores with longer core test times. These longer core test times cause problems when scheduling cores and routers simultaneously.

<table>
<thead>
<tr>
<th>Router Test time</th>
<th>2 I/O</th>
<th>3 I/O</th>
<th>4 I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>Separate Combined</td>
<td>Separate Combined</td>
<td>Separate Combined</td>
</tr>
<tr>
<td>Routers only</td>
<td>25376</td>
<td>22608</td>
<td>18840</td>
</tr>
<tr>
<td>Cores only</td>
<td>18869</td>
<td>13412</td>
<td>10705</td>
</tr>
<tr>
<td>Combined</td>
<td>41509</td>
<td>29651</td>
<td>22703</td>
</tr>
<tr>
<td>Total</td>
<td>45245</td>
<td>29651</td>
<td>22703</td>
</tr>
<tr>
<td>Test Time Reduction</td>
<td>8.26%</td>
<td>17.68%</td>
<td>23.16%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Router Test time</th>
<th>2 I/O</th>
<th>3 I/O</th>
<th>4 I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>Separate Combined</td>
<td>Separate Combined</td>
<td>Separate Combined</td>
</tr>
<tr>
<td>Routers only</td>
<td>28616</td>
<td>21462</td>
<td>14308</td>
</tr>
<tr>
<td>Cores only</td>
<td>25062</td>
<td>17925</td>
<td>16489</td>
</tr>
<tr>
<td>Combined</td>
<td>53914</td>
<td>37525</td>
<td>30797</td>
</tr>
<tr>
<td>Total</td>
<td>53678</td>
<td>37525</td>
<td>30797</td>
</tr>
<tr>
<td>Test Time Reduction</td>
<td>-0.44%</td>
<td>4.73%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Router Test time</th>
<th>2 I/O</th>
<th>3 I/O</th>
<th>4 I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>Separate Combined</td>
<td>Separate Combined</td>
<td>Separate Combined</td>
</tr>
<tr>
<td>Routers only</td>
<td>295992</td>
<td>197328</td>
<td>147996</td>
</tr>
<tr>
<td>Cores only</td>
<td>271384</td>
<td>180905</td>
<td>150921</td>
</tr>
<tr>
<td>Combined</td>
<td>575532</td>
<td>361392</td>
<td>284396</td>
</tr>
<tr>
<td>Total</td>
<td>575532</td>
<td>361392</td>
<td>284396</td>
</tr>
<tr>
<td>Test Time Reduction</td>
<td>-1.44%</td>
<td>4.45%</td>
<td>4.86%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Router Test time</th>
<th>2 I/O</th>
<th>3 I/O</th>
<th>4 I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>Separate Combined</td>
<td>Separate Combined</td>
<td>Separate Combined</td>
</tr>
<tr>
<td>Routers only</td>
<td>699128</td>
<td>524346</td>
<td>436955</td>
</tr>
<tr>
<td>Cores only</td>
<td>611991</td>
<td>482352</td>
<td>333091</td>
</tr>
<tr>
<td>Combined</td>
<td>1311119</td>
<td>1086053</td>
<td>739499</td>
</tr>
<tr>
<td>Total</td>
<td>1311119</td>
<td>1086053</td>
<td>739499</td>
</tr>
<tr>
<td>Test Time Reduction</td>
<td>0.00%</td>
<td>-7.88%</td>
<td>3.97%</td>
</tr>
</tbody>
</table>

Table 2: Simulation Results Using Complex Router Architecture
Among other reasons, anytime there is a conflict due to a core being tested, the conflict is present for a longer time due to the longer length of the core test. For the benchmarks containing shorter core test times, the conflicts would not be present for long, allowing for more optimal simultaneous testing.
Chapter 5

Future Work

The integrated algorithm presented in this thesis does show improvement when scheduling routers and cores simultaneously. However, the scheduling algorithm for router and core testing is only part of the whole that makes NoC. In the area of testing, there are still technologies which must be developed and possibly standardized to minimize test time and cost. Future developments must take into account or overcome certain obstacles which cannot be controlled.

5.1 Possible Improvements

The integrated algorithm presented in this thesis shows an improvement over the straightforward algorithm, but there are some areas that could be improved upon to maximize efficiency and minimize testing time and cost. The pictorial view of the integrated algorithm shown in Chapter 3, Figure 11, shows some idle time in the middle of the time line. If possible, the overall test time will be reduced if this idle time is minimized or, if possible, eliminated. This could possibly be achieved by using a method which prioritizes routers from the inputs to the outputs, forcing these routers to be scheduled first. This would provide a free I/O pair as quickly as possible, allowing core scheduling to begin sooner. The current core scheduling algorithm could be used since this algorithm attempts to schedule cores with longer test times first.

In addition, this new method could be further improved by allowing I/O pairs to
change. For example, if Input 1 and Output 1 form an I/O pair, it may be more efficient to use Input 1 and Output 2 to schedule a specific core. One possible reason is that Input 1 and Output 2 provide a shorter path, resulting in fewer conflicts for future core and router scheduling as well as overall test time reduction.

5.2 Uncontrolled Obstacles

There are some conditions which often cannot be improved upon. One example of this is the location of the inputs and outputs. Often the location of the inputs can prevent one input from testing a router or core simply because of its location within the NoC. For example, if a certain NoC contains multiple inputs in one area on the network infrastructure, channel conflicts could often occur. One input currently scheduling a router or a core could have a high possibility of preventing another input close to it from scheduling a router or core, due to channel conflict. Unfortunately, this situation is at the mercy of the chip designers who design the chip layout. It is easy to see that the number of inputs also greatly improves the overall test time. The test time improves when there are more inputs and paths to schedule routers and cores for testing.

The most important area of future research is the test wrapper architecture. The method this architecture is implemented and designed will be a key to maximizing test time and keeping hardware overhead to a minimum. Hardware requirements for the wrapper can be significantly reduced if a single wrapper is designed for both the core and its router. This suggestion was discussed in Chapter 2.
Conclusion

This thesis has proposed a new algorithm for router testing in a NoC system. This algorithm relies heavily on the reuse of the network infrastructure within an NoC. Possible wrapper architecture for the router was also presented. Combining the router and core wrapper into one wrapper can reduce the wrapper hardware and is important in the reuse of the network for router testing. Finally, an integrated algorithm for scheduling tests for routers and cores was presented. Simulations were run comparing results using this integrated algorithm to previous work based on boundary scan. A paper based on this thesis was also submitted to the Design Automation Conference (DAC) in Anaheim, CA.
References


Appendix A

Sample Simulation Results

These results are the simulation results after running the integrated algorithm using the d695 benchmark with 2 I/O Pairs.

testtime is 6206

testtime is 2507

testtime is 836

testtime is 3863

testtime is 9869

testtime is 4605

testtime is 25

testtime is 588

testtime is 5829

testtime is 3359

router test time is 3768

Total core permutation 5040
Core permutation 1000: besttime=41509
Core permutation 2000: besttime=41509
Core permutation 3000: besttime=41509
Core permutation 4000: besttime=41509
Core permutation 5000: besttime=41509
Overall Best Testtime=41509
Core 1: start 41369, end 41394, input (1, 2), output (2, 3)
Core 2: start 40781, end 41369, input (1, 2), output (2, 3)
Core 3: start 38274, end 40781, input (1, 2), output (2, 3)
Core 4: start 26376, end 32205, input (1, 3), output (2, 2)
Core 5: start 28709, end 34915, input (1, 2), output (2, 3)
Core 6: start 18840, end 28709, input (1, 2), output (2, 3)
Core 7: start 34915, end 38274, input (1, 2), output (2, 3)
Core 8: start 32205, end 36810, input (1, 3), output (2, 2)
Core 9: start 40673, end 41509, input (1, 3), output (2, 2)
Core 10: start 36810, end 40673, input (1, 3), output (2, 2)
Appendix B

Simulation Source Code (Example)

This is the subroutine used to for the router scheduling only algorithm. The code shown is the entire subroutine without the initialization of variables. This subroutine was also used for the integrated algorithm, with some minor changes.

```c
while (num_tested!=ROW*COLUMN)
{
    //INPUT TESTING
    //check if inputs have been tested
    if (main_matrix[io[0]->in->x][io[0]->in->y]==0)
    {
        cout<<"inputs tested first"<<endl;
        for (int cnt1=0;cnt1<NUM_IO_HEU;cnt1++)
        {
            main_matrix[io[cnt1]->in->x][io[cnt1]->in->y]=1;
            num_tested++;
            chn_timcbound[io[cnt1]->in->x][io[cnt1]->in->y][0]=cur_time+ROUTER_TEST_TIME;
            chn_timcbound[io[cnt1]->in->x][io[cnt1]->in->y][1]=cur_time+ROUTER_TEST_TIME;
            chn_timcbound[io[cnt1]->in->x][io[cnt1]->in->y][2]=cur_time+ROUTER_TEST_TIME;
            cur_time=cur_time+ROUTER_TEST_TIME;
        }
    }
    //NEIGHBOR TESTING
    else //need to test all neighbors
    {
        EOG=0;
        //find all neighbors that can be tested
        for (int qq=0;qq<ROW;qq++)
            for (int ww=0;ww<COLUMN;ww++)
                if (main_matrix[ww][qq]==1)
                {
                    //find neighbors and try to test them
                    if ((main_matrix[ww][qq-1]==0) && (qq-1>=0))
                    {
                        EOG++;
                        rows[EOG]=qq-1;
                        cols[EOG]=ww;
                        main_matrix[ww][qq-1]=2;
                    }
                }
        for (int qq'=0;qq'<ROW;qq'++)
            for (int ww'=0;ww'<COLUMN;ww'+++)
                if (main_matrix[ww'][qq']]==1)
                {
                    //find neighbors and try to test them
                    if ((main_matrix[ww'][qq'-1]==0) && (qq'-1>=0))
                    {
                        EOG++;
                        rows[EOG]=qq'-1;
                        cols[EOG]=ww';
                        main_matrix[ww'][qq'-1]=2;
                    }
                }
            cur_time=cur_time+ROUTER_TEST_TIME;
        }
    }
```

if ((main_matrix[ww][qq+1] == 0) && ((qq+1) <= (ROW-1)))
    {
        EOG++;
        rows[EOG] = qq + 1;
        cols[EOG] = ww;
        main_matrix[ww][qq+1] = 2;
    }
if ((main_matrix[ww-1][qq] == 0) && ((ww-1) >= 0))
    {
        EOG++;
        rows[EOG] = qq;
        cols[EOG] = ww-1;
        main_matrix[ww-1][qq] = 2;
    }
if ((main_matrix[ww+1][qq] == 0) && ((ww+1) <= (COLUMN-1)))
    {
        EOG++;
        rows[EOG] = qq;
        cols[EOG] = ww+1;
        main_matrix[ww+1][qq] = 2;
    }

// ALGORITHM for deciding which input has shortest path
for (int i=1; i<= EOG; i++) // for each TESTABLE ROUTER LOOP, decide which input should test it
    {
        for (int inputx=0; inputx<NUM_IO_HEU; inputx++)
            {
                if (io[inputx]->used == 1)
                    {
                        cout << "cant use this input" << inputx << "=" << io[inputx]->in->x << ""," << io[inputx]->in->y << "endl;
                        path_length[inputx] = 100;
                    }
                else
                    {
                        cout << "trying to test " << cols[i] << ""," << rows[i] << " with input " << io[inputx]->in-
                        >> x << ""," << io[inputx]->in->y << "endl;
                        if (routers_tested[inputx] > 0)
                            path_length[inputx] = 100;
                        else
                            path_length[inputx] = 0;
                        while (channel_conflict == 0)
                            {
                                // X DIRECTION
                                if (cols[i] == io[inputx]->in->x)
                                   {" // cout << "need to test x only" << endl;
                                else // need to test y direction of path
                                    {
                                        // first determine if y is < or > than the input
                                        if (cols[i] < io[inputx]->in->x) // must go LEFT
                                            {
                                                cout << "must go LEFT" << endl;
                                            }
                                        else // must go RIGHT
                                            {
                                                cout << "must go RIGHT" << endl;
                                            }
                                }
                            }
                    }
                }
            }
for all cols between the input and dest. col, test the paths
for (int k=io[inputx]->in->x;k>cols[i];k--)
{
//test the channels between these nodes
if (cur_time>=chn_timebound[k][io[inputx]->in->y][2])
{
    if (main_matrix[k-1][io[inputx]->in->y]==1)
        path_length[inputx]+;
    else if (main_matrix[k-1][io[inputx]->in->y]==2)
    {
        if (((k-1)==cols[i])&&(io[inputx]->in->y==rows[i]))
            path_length[inputx]+;
        else
            {path_length[inputx]=100;
             channel_conflict=1;}
    }
    else
    {
    }
}
else //must go RIGHT
{
    for (int k=io[inputx]->in->x;k<cols[i];k++)
    {
//test the channels between these nodes
if (cur_time>=chn_timebound[k][io[inputx]->in->y][0])
{
    if (main_matrix[k+1][io[inputx]->in->y]==1)
    {
        path_length[inputx]+;
    }
    else if (main_matrix[k+1][io[inputx]->in->y]==2)
    {
        if (((k+1)==cols[i])&&(io[inputx]->in->y==rows[i]))
            path_length[inputx]+;
        else
            {path_length[inputx]=100;
             channel_conflict=1;}
    }
    else
    {
    }
}
else
{
    path_length[inputx]=100;
    channel_conflict=1;
}
}
cout << "path length [inputx] = " << path_length[inputx] << " testeda2 = " <<testeda2 << endl;

// Y DIRECTION
if (rows[i] == io[inputx]->in->y) {
    //cout << "should not need Y direction" << endl;
    else //need to test Y direction of path
    {
        //first determine if Y is < or > than the input
        if (rows[i] < io[inputx]->in->y) //must go UP
            {
                for (int k = io[inputx]->in->y; k > rows[i]; k--)
                    {
                        //test the channels between these nodes
                        if (cur_time >= chn_timebound[cols[i]][k][1])
                            {
                                if (main_matrix[cols[i]][k-1] == 1)
                                    {
                                        path_length[inputx]++;
                                    }
                                else if (main_matrix[cols[i]][k-1] == 2)
                                    {
                                        //if ((io[inputx]->in->x == cols[i]) && ((k-1) == rows[i]))
                                            if ((k-1) == rows[i])
                                                {
                                                    path_length[inputx]++;
                                                }
                                            else
                                                {
                                                    path_length[inputx] = 100;
                                                    channel_conflict = 1;
                                                }
                                    }
                            }
                        else
                            {
                            }
                    }
            }
        else //must go DOWN
            {
                cout << "main_matrix[cols[i]][k-1] is 
                " << main_matrix[cols[i]][k-1] << " x,y " << cols[i] << "," << k-1 << endl;
                cout << "io[inputx]->in->y is " << io[inputx]->in->y << endl;
                path_length[inputx] = 100;
                channel_conflict = 1;
                if (chn_timebound[cols[i]][k][1] > (cur_time + ROUTER_TEST_TIME))
                    {
                        cur_time = chn_timebound[cols[i]][k][1];
                    }
                else //must go DOWN
                    {
                    }
for (int k=io[inputx]->in->y;k<rows[i];k++)
{//test the channels between these nodes
if (cur_time>=chn_timebound[cols[i]][k][3])
{
    if (main_matrix[cols[i]][k+1]==1)
    {
        path_length[inputx]++;
    }
    else if (main_matrix[cols[i]][k+1]==2)
    {//if ((io[inputx]->in->x==cols[i])&&(k+1)==rows[i])
        if ((k+1)==rows[i])
        {
            path_length[inputx]++;
        }
        else
        {
            path_length[inputx]=100;
            channel_conflict=1;
        }
    }
    else
    {
    }
}
else
{
    path_length[inputx]=100;
    channel_conflict=1;
    if (chn_timebound[cols[i]][k][3]>(cur_time+ROUTER_TEST_TIME))
    {
        cur_time=chn_timebound[cols[i]][k][3];
    }
}
}
testeda2=0;
//
cout<<"input "<<io[inputx]->in->x<<","<<io[inputx]->in->y<<" has path length of "<<path_length[inputx]<<" to "<<cols[i]<<","<<rows[i]<<endl;
//
cout<<"output is"<<endl;
//*
for (int q=0;q<ROW;q++)
{
for (int w=0;w<COLUMN;w++)
{
cout<<main_matrix[w][q]<<" ";
}
cout<<endl;
}*
//force us out of while statement if no errors found
} //end of while loop for channel_conflict
39

// OR PUT IT HERE
 testeda2=0;
channel_conflict=0;
// end of input x IF/ELSE statements
// end of INPUT FOR LOOP, still in the testable router input

// Here we should know the # of channels for each router to get to a certain router to test
// determine which input should test this router
// copy the array to a different array to be sorted
for (int t=0; t<NUM_IO_HEU; t++)
{
    path_length_sorted[t]=path_length[t];
}

// BUBBLE SORT!!
for (int i7=0; i7<NUM_IO_HEU-1; i7++)
{
    for (int j=0; j<NUM_IO_HEU-i7; j++)
    {
        if (path_length_sorted[j+1] < path_length_sorted[j])
        {
            tmp = path_length_sorted[j];
            path_length_sorted[j] = path_length_sorted[j+1];
            path_length_sorted[j+1] = tmp;
        }
    }
}

// check to see who has the shortest path available
// when found one, compare to see if they are testing the least amount of routers
// if there is a tie, the first input testing the lowest number of routers gets to test another
shortest_path_input=100;
for (int i8=0; i8<NUM_IO_HEU; i8++)
{
    if (path_length[i8] == path_length_sorted[0])
    {
        if (routers_tested[i8] == 0)
        {
            if (shortest_path_input == 100)
                shortest_path_input = i8;
            else if (i8 > shortest_path_input)
                shortest_path_input = i8;
            else
                {} //tie
        }
    }
}
}

// if ((cols[i]==2)&&(rows[i]==0))
// {
//    cout<<"shortest path input is "<<path_length[shortest_path_input]<<endl;
//    cout<<"routers tested are: "<<routers_tested[shortest_path_input]<<endl;
//    cout<<"router "<<cols[i]<<"","<<rows[i]<<" TESTED BY "<<io[shortest_path_input]->in->y<<","<<io[shortest_path_input]->in->x" at time "<<cur_time<<endl;
// at this point we know which input is testing the router
// set lowest_routers_tested variable high again
lowest_routers_tested=100;
//if path_length_sorted[0]==100, then that means all paths had conflicts
if (path_length_sorted[0]<100) & &(shortest_path_input!=100))
{
//here we must "claim" the channels for the amount of time needed to test the router

//Y DIRECTION
if (cols[i]==io[shortest_path_input]->in->x)
   {} //cout<<"need to test y only"<<endl;
else //need to test x direction of path
{
   if(cols[i]<io[shortest_path_input]->in->x) //must go LEFT
   {
      for (int k=io[shortest_path_input]->in->x;k>cols[i];k--)
         chn_timebound[k][io[shortest_path_input]->in->y]=cur_time+ROUTER_TEST_TIME;
   }
   else //must go RIGHT
   {
      for (int k=io[shortest_path_input]->in->x;k<cols[i];k++)
         chn_timebound[k][io[shortest_path_input]->in->y]=cur_time+ROUTER_TEST_TIME;
   }
}

//X DIRECTION
if (rows[i]==io[shortest_path_input]->in->y)
   {} //cout<<"no need to claim Y direction"<<endl;
else //need to test x direction of path
{
   if(rows[i]<io[shortest_path_input]->in->y) //must go LEFT
   {
      for (int k=io[shortest_path_input]->in->y;k>rows[i];k--)
         chn_timebound[cols[i]][k][1]=cur_time+ROUTER_TEST_TIME;
   }
   else //must go RIGHT
   {
      for (int k=io[shortest_path_input]->in->y;k<rows[i];k++)
         chn_timebound[cols[i]][k][3]=cur_time+ROUTER_TEST_TIME;
   }
}

//mark this router as tested!
main_matrix[cols[i]][rows[i]]~3; //was =1
//mark io as used until after testing this router
io[shortest_path_input]->timebound=cur_time+ROUTER_TEST_TIME;
//mark all its channels as used until this time
chn_timebound[cols[i]][rows[i]][0]=cur_time+ROUTER_TEST_TIME;
chn_timebound[cols[i]][rows[i]][1]=cur_time+ROUTER_TEST_TIME;
chn_timebound[cols[i]][rows[i]][2]=cur_time+ROUTER_TEST_TIME;
chn_timebound[cols[i]][rows[i]][3]=cur_time+ROUTER_TEST_TIME;
//also increase count of total routers tested
num_tested++;
//increase the routers_tested[] of the winner
routers_tested[shortest_path_input]++;

//also output if it is an output

//clear path_length f and routers tested or next round
for (int q1=0;q1<NUM_IO_HEU;q1++)
    path_length[q1]=0;
shortest_path_input=100;
}//end of IF statement
else  //this means a router that could be tested has no valid path
    //therefore we set the matrix for that router back to 0
{
    main_matrix[cols[i]][rows[i]]=0;    //flag the router as untested
}
}//end of TESTABLE ROUTER loop

//number of routers tested per input is cleared for next set of inputs
for (int q1=0;q1<NUM_IO_HEU;q1++)
routers_tested[q1]=0;
cur_time=cur_time+ROUTER_TEST_TIME;  //increase cur_time not sure how to really do this
}//end of else, this is testing neighbors
//    cout<<"output is"<<endl;
for (int q=0;q<ROW;q++)
{
    for (int w=0;w<COLUMN;w++)
    {
        if (main_matrix[w][q]==3)
        {
            main_matrix[w][q]=1;
            //cout<<main_matrix[w][q]<<" ";
        }
        cout<<endl;
    }
    //cout<<"num tested "<<num_tested<<endl;
    //make sure none are marked used
    for (int q1=0;q1<NUM_IO_HEU;q1++)
        io[q1]->used=0;
}//end of router testing WHILE statement

cout<<"curtime is "<<cur_time<<endl;
exit(0);