Design and Implementation of an Asynchronous TDMA Network-on-Chip

Rasmus Bo Sørensen Mark Ruvald Pedersen Madava Dilshan Vithanage

June 1, 2011

Abstract

We have designed and implemented an asynchronous TDMA Network-on-Chip on the basis of the Æthereal Network-on-Chip. The Æthereal Network-on-Chip was initially designed by Philips. We have analyzed the elasticity of our implementation and seen that the bandwidth of the network recovers after an increased delay on a link. The network is elastic in the sense that it can absorb phase differences in the network, without lowering the bandwidth of the network.

1 Introduction

When facing hundreds of cores on a single chip in the near future, the interconnection network becomes an even more important part of any design. The cost of such a network should be low in comparison to the cores, scale linearly with the number of cores, and allow as much concurrent (contention-free) communication as possible. Making the traffic in the interconnect packet-based enables many packets to be routed in the interconnect simultaneously. A Network-on-Chip (NoC) is a packet-based interconnect that scales very well with the number of resources in the system.

As chips become larger the difficulties of routing one global clock to the whole chip (with an acceptable skew) makes it infeasible to have a global clock. Avoiding the global clock is a tremendous advantage, especially compared to power usage. Designing an Asynchronous NoC would solve the problem of bandwidth and the problem of routing one global clock. Our main design goal has been to minimize the switch complexity, at a cost of NI complexity.

2 Design and Implementation

Everything produced in this project is available at Google code:
http://asyncnoc-2011.googlecode.com/svn/trunk

2.1 Asynchronous TDMA Network-on-Chip

We will design and implement an asynchronous TDMA Network-on-Chip, on the basis of the Æthereal NoC, designed by Philips. TDMA stands for Time Division Multiple Access, which means the communication resources are divided in time and split into sections allowing multiple data units to be routed simultaneously.

The Æthereal NoC is a statically scheduled NoC, where a compile-time schedule has been made that ensures no two data units are routed on the same link at once. With this static scheduling, bandwidth between resources is known at compile-time and this network is therefore well suited for hard real-time systems.

Dividing communication resources into time slots require synchronization of these resources, to make sure that no time slots are dropped. In a synchronous NoC this synchronization is easily derived from the global clock. In our asynchronous implementation, the synchronization is done by waiting for data units on all inputs of a node to be present before routing them through the switch. This ensures that no data units are dropped and that the time slots are transmitted in order. Asynchronous circuits are elastic with respect to timing. In our NoC the elasticity means that constant phase differences between the transmission of flits in two neighboring switches as well as minor fluctuations in the propagation delays of links, can be absorbed by the network.

2.2 Protocol

Our protocol is simplistic: Packets are statically source-routed using cut-through routing, with 1 flit being equal to 1 phit. Packets can however consist of an arbitrary number of flits. Delimitation of packets is indicated by the *Start-Of-Packet* (SOP) and *End-Of-Packet* (EOP) bit fields, as shown in figure 1.

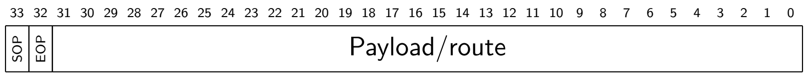


Figure 1: General format of all flits.

SOP and EOP are used to encode four types of flits (SOP,EOP):

(0,0) Empty space: Not a real flit, since empty spaces are not part of a packet. An empty-space is a valid token, which is sent when no other data is to be transmitted on a token-channel. The existence of empty-spaces is required since all input channels of the switch must provide tokens before any of them can be processed.

(1,0) Header: Holds the route. The route is a sequence of directions that the entire packet should follow.

(1,1) Body: Holds the actual payload data that is to be delivered to destination resource.

(0,1) End body: Holds payload data - like a body-flit - but also signifies the last flit of packet.

Hence packets always begin with a header, followed by zero or more bodies and end with an end-body. There may be zero or more empty-spaces in between packets. Figure 2 shows how SOP and EOP are grey encoded as a packet is sent.

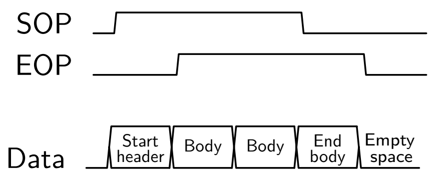


Figure 2: Example of timing diagram of a packet with 4 flits, 3 of which carry payload.

Since we are using cut-through source-routing, it is only possible to store the route in the first few flits of a packet – the header. In our case the header is only 1 flit long, and contains the full remaining route: It contains only what *remains* of the route, since previous routing-directions are shifted out for each hop made. This is further explained in section 2.3.1. As we will see later, this allows for simple routing-decision logic since the destination-port is always specified by the two least significant bits.

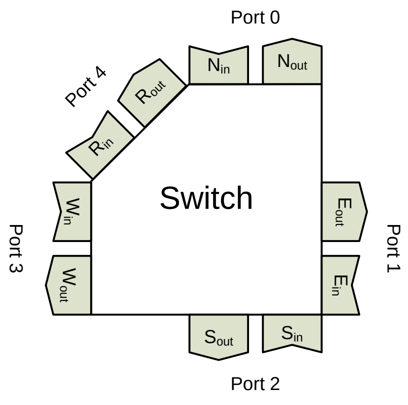
Since a flit is 34 bits, 2 of which is always spent on SOP and EOP, this leaves 32 bits to the route. With each route entry being 2 bits, a maximum of 16 hops can be encoded in the header. However, as will be explained in section 2.3.1, the last entry has to be of opposite direction than the second-last entry. Hence only 15 hops can effectively be encoded in the header. If we consider a rectangular $M \times N$ mesh, then the diameter of this graph in terms of hops is $M + N - 2$. Thus if any resource-node should be able to communicate all other nodes, we must obey $M + N - 2 = 15$. This allows for the different configurations: $\{M, N\} \in \{\{1, 16\}, \{2, 15\}, \ldots, \{7, 10\}, \{8, 9\}\}$, where $\{M, N\} = \{8, 9\}$

have highest theoretical bandwidth due to having the largest number of links[1]. Note that the largest *square* mesh supported is 8×8.

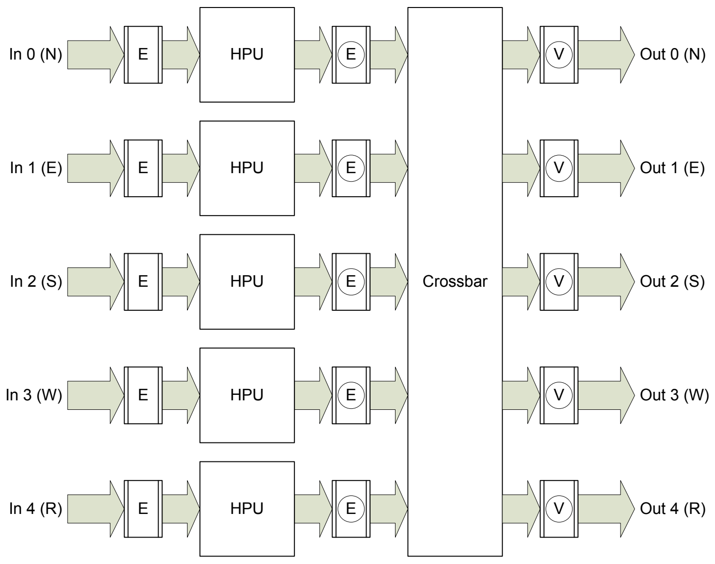
2.3 Switch

Besides network-interfaces, the main component of any NoC is the switch. Anticipating use in large meshes, it is wise to make the switch friendly towards 2D chip layout. As such, a switch has five ports numbered 0 to 4, representing the north, east, south, west, and the resource ports respectively. Network-interfaces would mainly be connected to the resource port, however this is not required.

The switch consists of a routing unit (HPU) followed by a crossbar and it is pipelined at two stages with channel latches. The initialization of the channel latches are shown in figure 3b.



(a) Switch



(b) Top-level Block Diagram of the Switch

Figure 3

2.3.1 Header Parsing Unit

As mentioned in the protocol description, the entire route is contained in the header. The route is a sequence of north, south, east or west directions that the packet should be routed towards. Hence the route is relative to the starting point; ie. if all resources wanted to send a packet to the same destination – in different time slots – all routes would be unique. This implies that routing tables of NIs will be different.

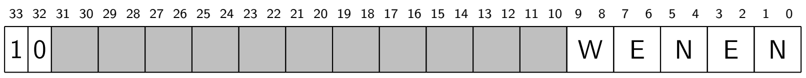
Directions are encoded as shown in figure 3, in binary. Hence north is 00_2, east is 01_2, etc. An exception is the resource whose port index (4) is not representable with only 2 bits. For a packet to be routed to make its final hop into the resource, this is encoded as if the packet makes a $180°$ U-turn: If an incoming packet on the west-port, specifies that it should be routed back to where it came from (the west), it is actually routed to the resource. Such exceptions do not incur more logic, but it requires each HPU to know on which port it is located.

It is possible to simplify logic by always looking in the same place for the current routing direction. We always look at the two least significant bits, one-hot decoding these to drive the control signal sel into the crossbar-stage. When we have set the control signal, the header is allowed to move onward, towards the next switch – this is done by a matched delay. Since all HPUs should look at the two least significant bits, we must shift the header[2] upon exit of the HPU. What we shift-in is treated as *don't*

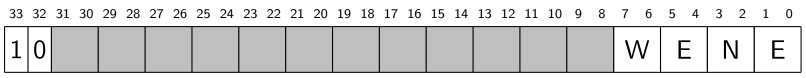
[1]The number of links in a $N \times M$ mesh can be derived to $(M-1)(N+N-1)+N-1 \iff 2MN-M-N$.
[2]Meaning only the lower 32 bits should be shifted by 2 bits. SOP and EOP are passed through unchanged.

care, since any well-formed packet will arrive at its destination resource before any HPU will decode these shifted-in bits[3].

Figure 4 and 5 shows an example of how the route in the header, guides a packet through the mesh. In this case the packet is sent from resource (2,0) to resource (0,2).



(a) Incoming header to switch (2,0) from resource (2,0).



(b) Outgoing header from switch (2,0).

Figure 4: Processing of header. This header specifies that the packet is to be routed towards the east output port of the switch. Header is then shifted. The final direction would route to the resource, not west.

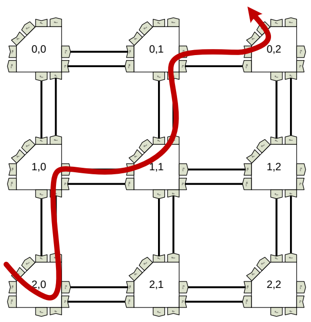


Figure 5: Packet travelling in mesh, having the initial header as shown in figure 4a.

Hence the header-parsing unit has to:

1. Identify the header.
2. If a header is found:
   (a) Parse the header – setting the crossbar according to next-hop field,
   (b) Update the route such that the next-next-hop will become the next-hop at the next switch.
3. Keep crossbar-settings for all flits of the packet.
   (a) Reset settings upon receiving an empty-space.
4. Leave non-header flits unaltered.

As crossbar-settings has to be preserved for all flits of the packet, the HPU becomes stateful. The HPU speculatively computes the one-hot decoded sel signal for use in the crossbar-stage. When it has been determined that the flit is a header, sel is latched forward to the crossbar-stage. Since the select latch is a simple latch, not a token-latch, we should be careful to not accidentally opening it. Hence we must only open the latch when we are sure data is valid – as can be seen from figure 6, an early data validity scheme is assumed[4]. The sel signal driving the demux in the crossbar-stage, should be stable before we allow the token to propagate; hence we need a matched delay as shown in figure 6.

Finally, empty-spaces should not interfere with other incoming flits of the switch. Empty spaces can be ignored in two ways:

[3]These don't care bits (greyed out in figure 4) could be of use in the NI: Instead of shifting in some arbitrary constant (presently 00), HPUs could shift-in their port index. This would allow the NI to do a reverse lookup in its routing tables to find out the sender of packet.

[4]This is in accordance with the validity scheme used by all our token-latches, which use the *simple latch controller*. Note that validity schemes must be compatible.

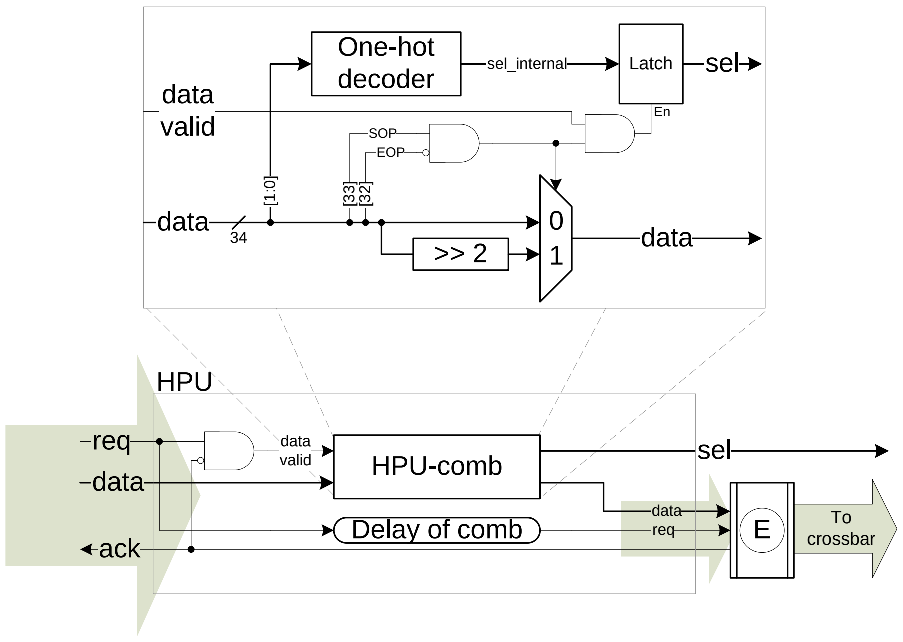


Figure 6: Simplified Header-parsing unit. Resetting by empty-spaces not shown. Only switches are shown.

1. Being encoded as all-zero's, so that when they are OR'ed in the final stage of the crossbar, they have no effect on real data.
2. Adding an exception to the one-hot encoding so that $sel\_internal$ is all-zero's when encountering an empty-space. This suppresses the empty-space, after the AND array in the crossbar-stage. This allows for non-zero data in the lower 32 bits of empty-spaces.

In fact both approaches are used in our design. Likewise, empty-spaces are silently created at output of the crossbar-stage if nothing is routed towards an output port.

2.3.2 Crossbar

The crossbar we have implemented is fully generic, and can be instantiated with a given number of channels (A channels is one input port and one output port), it should be noted that the crossbar is a fully connected network, thus it does not scale linearly. To simplify the logic, the select signal to the de-multiplexers is one-hot encoded. This results in a de-multiplexer that can be made from a number of parallel AND gates, the logic depth is only one AND gate, making the de-multiplexer very fast.

Because of the property of the static routing protocol, only one phit is routed to one output port of a switch in each time slot. The multiplexer can simply be implemented as a number of parallel OR gates, also a logic depth of one OR gate.

Synchronization of time slots Synchronization of the time slots are done by waiting for packets on all the input channels, when all input packets are ready they are transferred to the output latches of the crossbar stage of the switch, C-elements are used to synchronize the request and acknowledge signals. Two C-elements are used to synchronize the request and acknowledge as seen in figure 7.

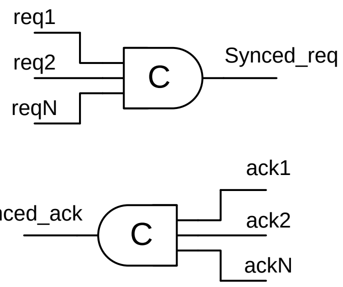


Figure 7: Synchronizing C-elements in the crossbar.

2.3.3 Token latches

As described in [2], a token-latch is made up of a latch controller and a normal latch. We refer to [2] for a more in-depth treatment of asynchronous circuits; here, we just detail how a synthesizable token-latch can be made without a custom hardware implementation of the C-element.

We have used the so called *simple latch controller* in all token-latches; cf. figure 8.

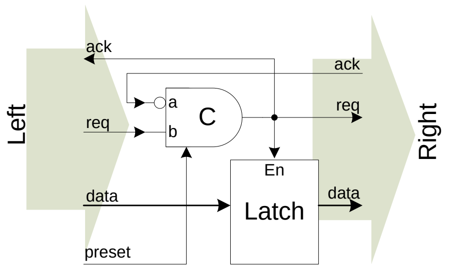


Figure 8: High-level view of our token-latch. The C-element constitutes the simple latch-controller.

The C-element has hysteresis and therefore state. Hence all C-elements need proper initialization. The initialization value is specified by a VHDL `generic`, and the C-element is initialized by asserting the $preset$ signal. While a C-element can be implemented elegantly at the transistor-level in ASICs, we are not so fortunate on FPGAs. Our proposal – that we have tested successfully on actual hardware – is shown in figure 9. In fact, figure 9 will synthesize to exactly one LUT4 on a Xilinx Spartan 3E FPGA.

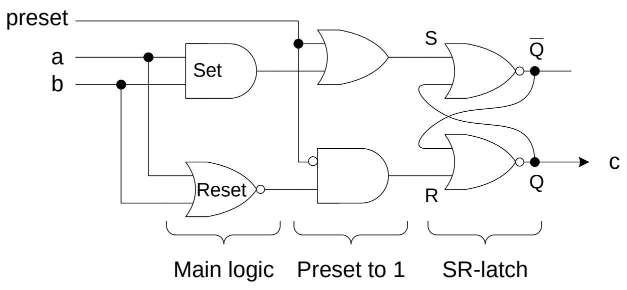


Figure 9: C-element suitable for FPGA implementation.

2.4 Testbench

The testbench consists of a 3x3 mesh network, where a producer and a consumer together plays the role of a resource. The producer generates flits that is read from a corresponding test vector file containing the raw flit data, while the consumer asserts the flits that are received by comparing it with the flit data also stored in a corresponding file. The static routing of packets that is used in the testbench is shown in figure 10.The outer rim of the mesh requires dummy resources so that all input channels can be synchronized. These dummy resources use an insignificant amount of hardware because a producer consists of a single NOT gate while a consumer consists of a short wire. It should be noted that they are also required for implementation.

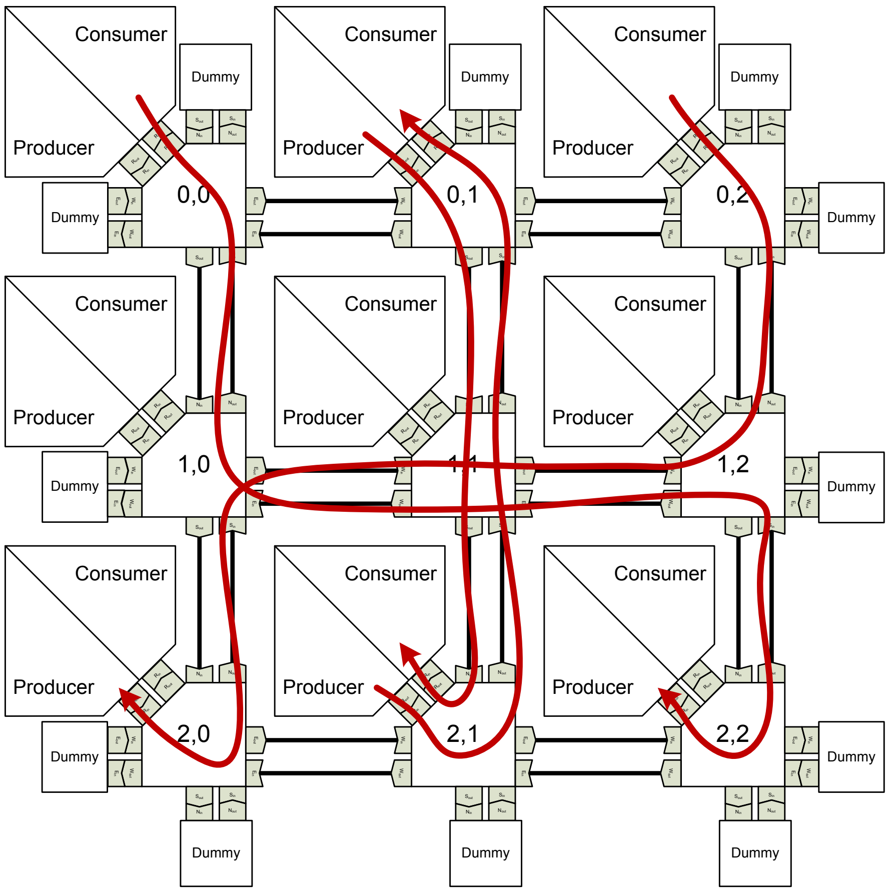


Figure 10: tb_NoC Setup

Another testbench with a mesh of 9×9 switches was used for performance analysis. This testbench routes no packets, i.e. only empty spaces are sent by the producers, but this is not important for performance analysis. Furthermore a global timer is used to log a global wall-clock time when the synchronized request signal is generated in the crossbar of all the switches in corresponding files. These files are then used to generate a "heatmap" of the NoC where the dynamic behavior of the mesh network can be easily visualized. The heatmap is used to represent the cycle time between corresponding request high events where the colour red represents the slowest and the colour blue represents the fastest. It can also be used to observe the phase changes of the request signal in the entire mesh network. The heatmapping software, `noc_viz`, is a small and portable C++ program using the PNGwriter library. A step function was used to inject a delay into a link on switch $(0, 0)$ to simulate a slowing link and to visualize the impact on performance. On-chip temperature differences and crosstalk may cause non-uniform link latencies.

3 Performance characteristics

In this project we will concentrate on simulation, even though we have implemented the switch in synthesizable VHDL. The performance characteristics of our switch nodes will only be investigated through behavioral simulations, with the propagation delays we have specified. Therefore we can not give any real values for the characteristics.

What we can investigate is how the network is able to cope with fluctuating propagation delays on links and how it handles phase changes between nodes in the network.

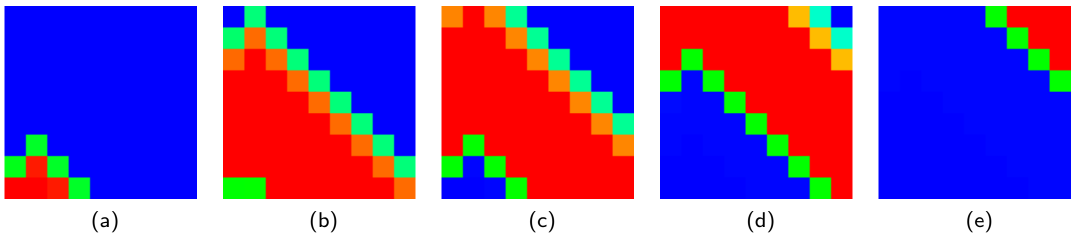


Figure 11: Cycle times for synchronized request signals

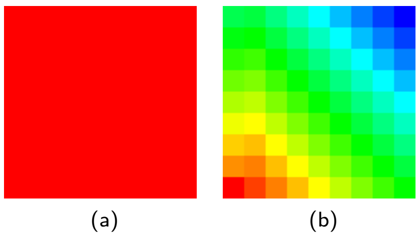


Figure 12: The phase of the handshake cycle at (a)the beginning of the simulation and at (b)the end of the simulation.

Due to synchronization, in the steady state the network will only be as fast as the slowest node. When we introduce an increase in propagation delay on a wire, we can visualize the change in bandwidth of the network by plotting the cycle time for the synchronized request signal for each switch node in the network. The plotted cycle time of the synchronized request signal can be seen in figure 11.

In figure 11a the wave front of larger cycle times has started to propagate into the network from the point where the extra wire delay was introduced. Note that in figure 11c the back of the wave appears. In figure 11e the wave has almost propagated out of the network.

At the end of the simulation, the cycle time is the same as the initial cycle time, but the state of the network is not the same as the initial state. The phase of the handshake cycle at the beginning and the end of the simulation can be seen in figure 12. Figure 12a is the phase of the handshake cycle at the beginning of the simulation and figure 12b is the phase of the handshake cycle at the end of the simulation. It should be noted that the colour in the plot of the phase does not carry any significant meaning, rather the difference of the colours is of importance for analysis. It is seen that the phase of the handshake cycle is uniform at the beginning of the simulation. At the end of the simulation, after the cycle time has recovered, the phase is no longer uniform. We have calculated the phase difference between the blue area and the red area to be approximately $820°$. Meaning that the red area is more than two time slots behind the blue area, showing the true elasticity of the asynchronous NoC. Comparing these results to a synchronous case, the synchronous case would have to be able to tolerate a skew of more than twice the clock period.

The reason why the network can tolerate such a large phase difference is because of bufferspace in the link and HPU between two neighboring switch crossbars. This phase difference in the network will result in changing latencies in the network, depending on the taken route. The phase difference will never be larger than the tolerable phase difference in the network, because it normalizes itself by momentarily lowering the bandwidth in parts of the network.

4 Discussion

The asynchronous Fibonacci circuit can run in an FPGA, this shows that the basic primitives, we have implemented, works (C-elements, Latch-controllers, Latches). One thing missing for the NoC to run in an FPGA is the matched delays on the request signals. Matched delays could be avoided by using dual-rail instead of bundled data.

While we have focused purely on the interconnect, for a full NoC implementation it is also required that each resource has its own Network Interface (NI). The NI should contain routing tables and time slot information, and do the translation of an address of a resource to a time slot and a route. The information on which time slot should be sent now, from the NI, is taken care of by a counter that increments its value after each handshake. The counters of all the NIs can be reset, by a global reset signal. This could suitably be done by the preset signal which is already in place.

In our simulation we do not take the possibility of clumping of tokens into account. Clumping of tokens in self-timed rings is a phenomenon that will not show up in a simulation with our simple model of the C-elements. In [1] we see that for tokens to clump together the C-element model should change faster if it has been changes recently, the propagation delay is smaller right after the C-element just changed. We recommend that the phenomenon of token clumping should be investigated further, possibly in simulation by making a more precise model of the C-elements taking the effects described in [1], into account.

5 Conclusion

We have designed and implemented an interconnect, which does not use a global clock and scales well with many resources. The interconnect is based on the asynchronous Æthereal TDMA Network-on-Chip designed by Philips. Our implementation works correctly in simulation, and all packets are delivered successfully as the testbench reports. We have shown that even when the phase varies in the network, the network is still able to operate at full speed. Nonetheless the effects of dynamic link delays are not clear and should be investigated further.

References

[1] High-Level Time-Accurate Model for the Design of Self-Timed Ring Oscillators, Jérémie Hamon, 2008

[2] Asynchronous Circuit Design – A Tutorial, Jens Sparsø, 2006