ABSTRACT
The traditional approach of using a bus for communication between different hardware components is not suitable anymore for modern multicore systems. In this paper this problem is solved with a Network-on-Chip. A protocol based on existing protocols is developed and a router that uses this protocol is built. To build this router Clash is used, a new hardware description language based on Haskell. This is also a test for the language, as no routers have yet been described using Clash.

Keywords
Clash, NoC, Network-on-Chip, mesh network, router.

1. INTRODUCTION
Modern hardware architectures try to use more and more processors to support parallelism in software. As more processors are added the traditional approach of a bus for the communication between components is not suitable any more. The bus will be busy most of the time and other resources will have to wait a lot to use the bus. Peer to peer communication fixes this problem. By adding links between every two resources, it is not possible that a resource has to wait to send data. But peer to peer is not very scalable, as the number of processors grows the number of lines grows exponentially, resulting in chips on which most space is used by wires, and not by components. A more scalable approach is needed. The NoC (Network-on-Chip)\(^3\) is a promising paradigm. This gives better scalability than a bus and peer to peer.

In an NoC all the processor cores are placed in a network. Each core is individually linked to a router. All the routers are connected in a grid to make a network. This shares many characteristics with the internet network. In that case not cores, but computers are connected to a router in the network. It shares some characteristics with internet but it also has a lot of differences. Therefore new protocols should be made to run on these networks.

It is possible to use the OSI layered model and fill it with protocols already used in the internet. Then a logical choice would be to use the IP/TCP protocol. These protocols were made for large, uncontrollable networks. Because the topography of the network is static the environment is much more controllable. Therefore less protocol overhead is needed to get successful communication. Also the layered stack of the OSI model is not needed on these networks. The model was built to make networks were different services can use different protocols at lower levels, but still use the same network. An NoC only needs to support one service that always uses the same protocol. Therefore the network only needs to service that one single protocol. Also the layering in the protocol is not mandatory as no parts of the protocol will be replaced by other parts. Using some layering in the protocol may still be useful to keep the overview over the entire system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

19th Twente Student Conference on IT, June 24th, 2013, Enschede, The Netherlands.
Copyright 2013, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.
The NoC has a static network. No routers can be removed or added once it is running. So routing does not have to take into account that new routers may be added. Routers can still disappear from the network as they can break. Also the links between the routers are static. No links will be added or removed. Taking this into account a simpler routing algorithm and routing table can be used. Where a big dynamic network needs to be able to cope with changes, this network is static and will not have those changes.

1.1 Cλash
Cλash\(^1\) is a hardware describing language in development. One of the goals of this language is to make a clearer one-to-one translation from the code to the hardware. It offers a higher level of abstraction for describing hardware than classical VHDL. As the function of a router should be describable in a high level language, it also should be possible to translate this without too much problems to Cλash.

In this paper we will compare several existing routing protocols for NoC's. For the best protocol, possibly with some adaptations, a hardware router will be built using Cλash. This language is still under development so we might run into problems using it. It can lack some useful functionality when building routers. After the research we can make recommendations for improvements of Cλash.

2. RESEARCH QUESTIONS
The research questions coming from this are:

- Is it possible to build NoC routers using Cλash?
- Does Cλash lack functionality?
- Does Cλash have benefits over traditional hardware description languages?

3. RESEARCH METHODS
To choose a protocol, a literature study is needed. This will compare a few protocols and choose a favorite for the rest of this paper. Maybe some minor changes will have to be made to the protocol. If we want to add some more options a few control possibilities will be added. If some options are not needed they can be removed. So the basic structure of the protocol will not be changed, only extra options might be added or removed.

From the first step a good protocol for NoC emerges. The second step is to build a router for this protocol. The router will be described using Cλash\(^1\). To do so first a functional description of the router will be made using standard Haskell. Then the Haskell program is tested to evaluate if the program is actually working to the description of the protocol. For the testing a big amount of data has to be created. As the routers will be used in an environment where a lot of sources can create data a lot of data can pass through the router in all directions. All possible paths and lengths of data have to be tested.

When the router is functional in Haskell a translation has to be made to a Cλash program. This maintains the structure of the program, the functionality does not change. The main changes will be the use of different data types and functions that use those types. After the translation is done a new round of software testing is done.

If during this process any problems using Cλash are found recommendations will be made for the improvement of the language. As the language is still under development some features are not yet available. Also some features may not have been considered yet, as they have not been needed in projects so far.

4. RELATED WORK
Already some processors have been built using an NoC structure to evaluate the possibilities of NoC's with many cores. Examples are the MIT RAW\(^2\), TRPS\(^3\), Intel 80core\(^4\) and Tilera TILPro64\(^5\). All of these use routers with a simple protocol to communicate between different processors. These processors show that the concept of a NoC is possible. They also show that a larger number of cores on a single chip is possible with NoC's. The TRPS has 40, the Tilera TILPro64 has 64 and the Intel 80core has 80 cores. All of these processors use a 2D mesh network to connect the cores. Not all of those meshes are square, some are more stretched rectangular. Most of the processors use wormhole switching to connect the routers. These two properties, a mesh shape and a wormhole switched network make a good starting point for a protocol.

A fellow student in this course has made a framework for NoC's \(^3\). The framework supports different topologies and protocols for a network. This framework is also built in Cλash. The framework can help testing the router built in this paper.

5. PROTOCOL
5.1 OVERVIEW
The protocol for the network needs to keep the router simple and it needs to reduce traffic on the network. The entire network is built on a chip or an FPGA, therefore it has limited space. A protocol where the routers need a lot of computational force would also need a lot of space on the chip, therefore it would not be suitable.

Also in a network with possibly many cores that all can produce data the amount of overhead is an issue. The network can already get congested only by the data produced by the cores. When all cores send data to a certain area in the network, that part of the network can get congested easily. A lot of overhead data and resends would only congest it more. Because in this paper the entire network is known, and under control, overhead can be kept to a minimum. All routers will know the protocol and implement it correctly, so no options for situations where routers do not stick to the protocol have to be made.

A third point to consider for this protocol is in order arrival of a packet. In traditional networks in order arrival is often not guaranteed. This means that at an end point parts of packets have to be stored until the rest of the packet arrives. When packets get larger this also needs larger buffers. As space is limited, this is not desirable.

We made the choice for a wormhole routing protocol. It needs little overhead and no resends exist. So the traffic can be kept to a minimum. Also the computational power needed in a router is small. Another advantage is the small buffers that are needed for wormhole routing in the routers. Large buffers would need more space on the chip. A final argument to use wormhole switching is that in order arrival is guaranteed. No large buffers at end points are needed.

The protocol is based on a combination of two protocols that are already in use. The basic design of flits is based on the Intel 80core\(^8\). From this the design of a flit with a number of control bits with each flit is derived. Also the principle of sending the addressing data in the first flit(s) and the data in the rest is used. In the final design two of the control bits have been removed. Only one control bit, the valid, header and tail bit have been implemented. The basic design of a flit is showed in Figure 2.
The packet addressing of this protocol is not used, because of the poor scalability. Not an address is sent with the packet, but the direction the packet has to take at each router. In a network with routers with 5 ports, this takes 3 bits per hop. Only 10 of these 3 bits fit in a 32 bit flit. For every 10 hops a packet has to take, an extra flit will be added. For larger networks this could mean a large number of extra flits. Therefore another addressing option has been chosen.

<table>
<thead>
<tr>
<th>Fc</th>
<th>V</th>
<th>H</th>
<th>T</th>
<th>32bit address or data</th>
</tr>
</thead>
</table>

**Fc: Flow Control**

**V: Valid**

**H: Header**

**T: Tail**

**Figure 2. Low level design of a flit**

For the addressing an x-y option has been chosen. This is based on [10]. The network this router is designed for has a mesh layout. On such a grid an x-y address gives a good representation of the position of the router. On a regular mesh routing can be done simply by comparing the x- or y-coordinate with the coordinates of the router. No routing tables or large scale computations are needed. This again reduces the space needed by the router. In a mesh this eventually means that routers are addressed with an x,y coordinate where the 0,0 is in the lower left corner.

### 5.2 CONTROL BITS

#### 5.2.1 Flow Control

The flow control bit is used for flit level flow control. When at the receiving side the buffers threaten to become overloaded, the flow control bit can be sent to the sending side. When receiving this bit, a sender is not allowed to send data in the next clock cycle.

#### 5.2.2 Valid

The valid bit states that a flit contains actual data. When this bit is not received, the receiving side does not have to evaluate the data, only the rest of the control bits.

#### 5.2.3 Header

The header bit is only sent with the first flit of a packet. This bit tells that the data should be evaluated as an address. This bit is never sent without a valid bit.

#### 5.2.4 Tail

The tail bit is only sent with the last flit of a packet. When receiving this bit, the virtual circuit corresponding to the packet can be torn down. This bit is never sent without a valid bit.

### 5.3 ADDRESSING

As stated earlier, addresses are built up from an x and y coordinate. Each of this coordinate is 16 bits, so a maximum mesh of 65536 * 65536 can be addressed. The address of the router and it’s corresponding core are the same. A packet with an address will always be sent to the core, addressing the router is not possible. As no separate protocol runs on the routers, sending packets to those routers is not needed.

### 5.4 ROUTING

The routing strategy has been chosen as simple as possible. No advanced options to prevent starvation of certain links have been implemented. More complex routing is beyond the scope of this paper.

If the address of a flit is the address of the core, next to the router, it is of course forwarded to the core, when possible.

Next only the x coordinate is evaluated. When the x coordinate of the packet is higher than the x coordinate of the router it is forwarded to the eastern port, if it is lower, it is forwarded to the western port. Forwarding will only be performed if the port is free. That is it is not blocked by the router on the other side (not(blokX)) and the port is not sending yet (not(sending)) && isNullPacket sendX) where X is the direction. If the packet could not be sent in the former steps, the y coordinate will be evaluated. If that coordinate is higher than the y of the router, the flit will be forwarded north, if it is lower it will be forwarded south. This process will be performed for every incoming buffer, each clock period. Code for this routing is showed in figure 3.

The setVirtualCircuits function is called from the router with the current state. That functions folds over the setVirtualCircuit function with the buffer list as input, so the function is used for each incoming link. After each iteration of the function, it returns an updated RouterState. The isNullPacket function returns a boolean value that is true if the packet is an empty packet (Control Continue). The newVCSState function sets packets in the outgoing buffers and updates the RouterState for the new virtual circuit. The complete functionality of that function is showed in section 7.3.

### 6. ROUTER

A router has five incoming and five outgoing ports. As it is in a mesh, 4 ports are linked to other routers, the fifth port is linked to a core. Each port is 36 bits so entire flits can be received and send in a single period. Each incoming port has a buffer for a maximum of two flits. Internally the router needs a number of variables to store the information about virtual circuits that are already set up. Incoming packets will always first be written to the buffer. From the buffer the router will evaluate the packet and send it to its destination. The router can send flits on in two different stages.

The first stage is the continuation of virtual circuits that already have been set up. No routing has to be done in this stage; the router only has to check if it is possible to forward the flit. When the flit is a tail, some additional bookkeeping has to be done to tear down the circuit. The virtual circuit has to be closed, so in the next period a new one can be set up.

The second stage is the creation of new virtual circuits through a router. This stage has to take into account that some outgoing links are already in use by a circuit. If all possible outgoing links are in use the flit has to wait in the buffer until a circuit is torn down. When creating a circuit again some bookkeeping has to be done to remember the virtual circuit.
After the protocol was specified, it had to be translated to Cλash. The biggest step for this was building the router in standard Haskell. As Cλash does not support recursion yet, this entire description was built without recursive definitions. There were no further constraints during the process.

7.1 TYPES
Haskell supports a higher level of abstraction for types. Flits have not been described in bits, but with descriptive names. The type definition of the flits is shown in figure 4.

```
data Packet = ControlState | Content State Body | Tail State Body
data State = Blocking | Continue
data Body = Address (Word16,Word16) | Data Int32
```

Figure 4 Haskell Types

A Packet can have one of three constructors. A Control packet is a flit without the valid bit. So this packet only contains control bits, more specific only the flow control bit. A Content packet is a packet with the valid bit. It always has a valid payload. This packet can have the header bit, but never the tail bit. A Tail packet is a packet with the tail bit. It is always the last packet in a virtual circuit. It always has the valid bit and therefore has valid payload.

State is the direct translation of the flow control bit. Blocking is the active flow control bit and Continue is the non-active bit. This State is used in all types of packets.

The Body is only defined for Content and Tail packets. Control packets also have a body. That body has no meaning however, therefore it is not showed in this description. The Address body is used with the header bit. This implicitly means that the body is an address with an x and y coordinate. Both x and y are 16 bits. The Data body is the body of a packet without the header bit. It contains 32 bits of payload. It is represented here by an Int32. For the router, the payload has no specific meaning, so any 32 bit data type can be inserted here.

7.2 ROUTERSTATE
The internal state of the router is represented by a record as shown in figure 5. The record contains lists of variables. In each list is a variable for each link, a router has five links, so each list has length five. Lists instead of tuples have been used, even though the lists have a fixed length, because Haskell has a number of functions that work with lists that can save a lot of work.

The state has ingoing buffers (buffer) of length two, outgoing buffers (send) where packets can be stored before they can be sent in the next clock period. Also information about the virtual circuits is stored. For each incoming link the current outgoing link (to) is stored, and for each outgoing link, the connected incoming link (from) is stored. Both of these lists have an extra Boolean value (sending, toBool) that indicate that the value in the lists is not outdated. These variables are stored to send on packets that are already part of a virtual circuit. One of the two variables would be enough, as the other value can be derived from the first. To keep the rest of the design more simple, both values are stored. The last value in this routerstate are the blocking Booleans (block). These variables store the value of the flow control bit that has been received on each link. By these values the router can decide if it is allowed to send on packets.

```
type Buffer = (Packet,Packet)
data routerstate = RouterState{address :: Word16,send :: [Integer],to :: [Integer],sending :: [Bool],toBool :: [Bool],buffer :: [Buffer],block :: [Bool]}
```

Figure 5 Routerstate

7.3 THE ROUTER
The router is defined as a single function. As input it has its current state and a list of five incoming packets, one for each link. The result of the function is a tuple of the new routerstate and a list of five outgoing packets, again one for each link. All the different stages that the router goes through are all separate functions. Most of these functions work on lists. As input they get the current state. From the state one of the lists is used, with a zipWith, map, or foldl another function is used on that list. The output of each of the functions is the changed list. The changes are saved in the state and passed on to the next stage. The function of setting up new virtual circuits is more complex. It
The entire router has been tested to understand the meaning of packets, without the translation step has been omitted. The entire router has been built without recursion, so the final translation will be possible without too many problems. The Haskell source that is available has been used for further testing. If the translation to Cλash is performed well, the test results using that source will be the same.

Figure 6 Setting up a new virtual circuit

7.4 HASKELL TO CλASH
The final stage of building hardware in Cλash is translating the Haskell source to Cλash. Due to lack of time, this last translation step has been omitted. The entire router has been built without recursion, so the final translation will be possible without too many problems. The Haskell source that is available has been used for further testing. If the translation to Cλash is performed well, the test results using that source will be the same.

Figure 7 The testing network

8. TESTING
The first round of testing was performed on a single router with manual input. Also the output and the internal state of the router were evaluated manually. All possible situations have been tried, all combinations of input and internal state. From this test the router showed all expected output.

For a second round of testing the router has been used in a small network. For this testing a framework built by another student in the same course was used. This framework was specifically built for NOC networks. Any type of router and core can be used with this framework. It also offers several layouts for the network. As this router only supports mesh layouts, that option has been used. The cores used during the testing only sent back the packets that they were received. No real cores like CPUs or GPUs were used. This ensures that the environment is more predictable. The test was performed on a small 3x3 network as showed in figure 7. Because the router has always 5 ports, the network was built in a torus topology. The results of the second tests were as expected. All packets arrived in order at their destination, and no packets were lost.

9. EVALUATION
One of the main questions of this paper is how Cλash performed for the purpose of building a router, compared to standard hardware description languages like VHDL.

One of the first advantages of Cλash is the possibility to build new types. These types have been used to define packets. All four control bits were translated to descriptive names. This is an advantage for the programmer over a number of bits, the programmer would normally see. It makes the code more readable, as words describe directly the meaning of the control bits, where bits need an extra description on their meaning. Now a reader can understand the meaning of packets, without knowing the order of the different control bits.

Another use of these types is in the body of the packets. The body exists of 32 bits, but in a header it is two 16 bits coordinates. In Cλash it is not needed to split the body in two when it is an address, in the typing it is already defined that in a header packet the body is always two 16 bit coordinates. Using this technique there is no need for conversion to interpret incoming data differently.

When running the test, the high level types have an extra advantage. The output of tests can be written in type names. Hardware tools mostly show only the values of bits and bytes. A tester really needs to take a good look at those values to know what is going on. Text based output gives a much better view of what is going on. No special tricks are needed to get this output, it is already part of the language.

Higher order functions for lists are very useful for building routers. Most functionality of the router does the same for each link. This functionality can be implemented in one function which can be called for each link using these higher order functions. With iterative languages the same can be done with loops. A disadvantage of loops is that they don’t automatically return lists. The programmer has to put results in an desired data structure. Higher order functions put the results in the same lists as the input. This results in much shorter code with less possible bugs.

Cλash does not have ready to use test environments which are available for other languages. A tester has to build entire tests himself. Where other languages have tools that already implement a clock and default values, in Cλash all these basic settings have to be set manually. When the further development of the language is a success, also tools for these purposes have to be developed.

10. CONCLUSION & FUTURE WORK
This paper shows that it is possible to build NOC routers using Cλash. A simple router has been developed and tested. It also has some advantages over other, iterative programming based, languages. The higher level of type definitions shows to be the biggest advantage. But Cλash is still in development and needs to be developed further to become a mature language. In this development also additional tooling for the language will have to be developed to really make it a good choice for defining hardware.

A next step in the process will be the translation of the Haskell definition to real Cλash and eventually to hardware. Then an extra test phase can be run on an FPGA. A fist extension of the router will be the routing algorithm. The current algorithm does not prevent starvation from certain links. A more extensive algorithm will prove a next test for Cλash, as most algorithms...
are described iteratively and not functionally. Another extension would be to make this router suitable for networks with other topologies. More topologies, than only simple mesh structures are possible in a NOC, such as ring structures, or irregular mesh structures. These structures should be possible with only some changes to the router, because the defined protocol can work on more topologies.

11. REFERENCES