High-level Architectural Simulation of the Torus Routing Chip

Lasse Natvig*

Department of Computer and Information Systems (IDI)
Norwegian University of Science and Technology (NTNU)
N-7034, Trondheim, Norway, E-mail: Lasse.Natvig@idi.ntnu.no

Abstract

This paper presents a simulation model of the Torus Routing Chip (TRC) written in Verilog. The model represents the functional behaviour of the routing chip down to the flit (byte) level. The TRCs are self-timed and interconnected in a 4 by 4 torus (mesh with wrap-around) having unidirectional channels along the x and y-dimension. To avoid deadlock situations, the TRC implements two virtual channels on every physical channel. The model is presented in a top down manner with emphasis on the modelling of the packet routing algorithm, asynchronous channels, controlled access to shared resources and the increased complexity caused by virtual channels. The testing of the model as well as experience from using Verilog to develop a high-level architectural simulation is discussed.

Keywords*: Torus Routing Chip (TRC), verilog, architectural simulation*

.

1. Introduction

It is becoming increasingly difficult to *master the complexity* involved in the design of increasingly more complex computer systems. On the hardware side, *Hardware Description Languages* (HDLs) have become important tools to cope with this challenge. The use of HDLs has to a large extent replaced schematics. The transition is in many ways similar to the replacement of assembly code by high-level languages. Computer simulation of simplified models and prototypes of these complex systems are established techniques to unveil design errors at an early stage. It may also lead to an improved co-operation with end-users during system development. HDLs, such as VHDL and Verilog, provide simulation as an integrated part of the language.

The main motivation for the work reported in this paper, is to learn about the usefulness of Verilog as a tool for developing *high-level simulations of computer systems*. This type of simulation is often written in standard programming languages or specialised general purpose simulation languages. A benefit of using HDLs in this context, is that subsequent integration with or transition to a model at a more detailed level should become easier if both models are expressed in the same language.

Verilog is lacking several "high-level" features which are said to make VHDL better suited for implementing (large) high-level models [5]. The simulation model described herein is at the behavioural, also referred to as algorithmic level. The aim is to achieve correct behavioural representation and provide a testbed to try out algorithms or strategies that are crucial for correct behaviour. The possibility of mapping the model into hardware at a later stage should be given a balanced amount of thought. Too much thought on HW-realisation may make it too difficult to master the complexity at the current phase of the design process. Too little thought may result in a model not realisable in HW or with only modest performance.

With the term *architectural simulation*, as used in the title of this paper, we mean a behavioural level approach with an added goal of representing the most significant parts of the system architecture that the TRC component is used in. As a case for our study we have chosen the Torus routing chip (TRC) described by William Dally and Charles Seitz in [1]. This self-timed VLSI chip provides efficient message passing in a multiprocessor. To provide testing in a realistic environment, we simulate 16 processors interacting through 16 TRCs.

Section 2 of the article introduces the TRC routing chip. In Section 3 we present central parts of the Verilog model in a top-down manner. Section 4 presents some of our experience and the planned continuation of the project.

* This work was partly supported by the Norwegian Research Council, and was done while the author was at Nordic VLSI, Trondheim.

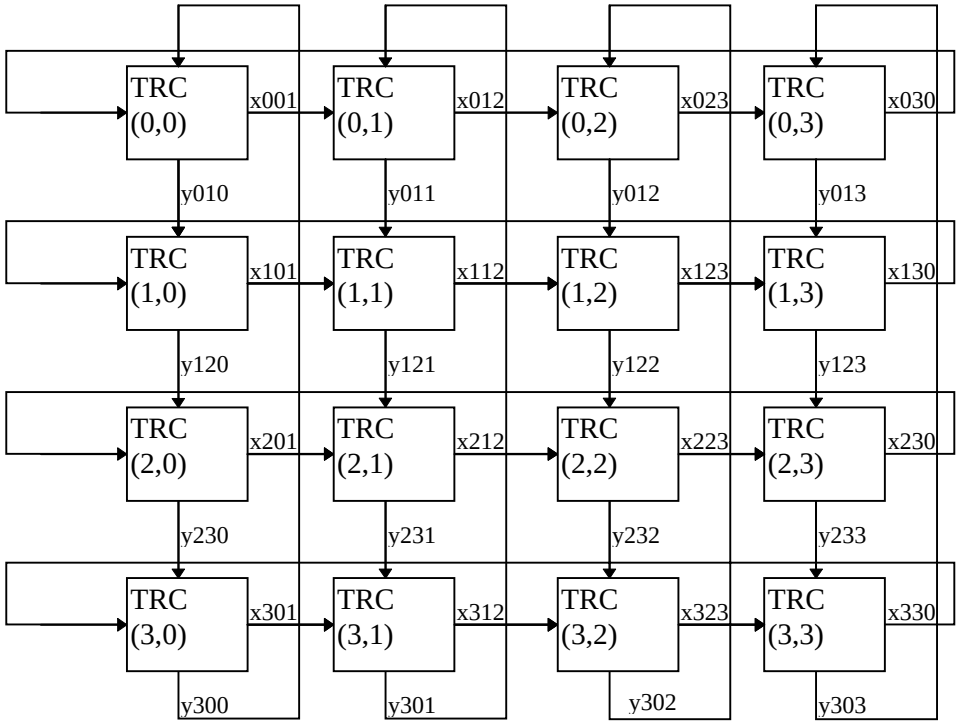

Figure 1: 4 x 4 mesh of TRC modules with (y,x) numbering

2. The Torus Routing Chip (TRC)

We have traditionally had two main categories of multiprocessors, those based on message passing, and those based on shared memory. Nowadays, the shared memory machines are more and more often based on message passing, which becomes a crucial component regarding performance. It is quite common to use a dedicated processor or to have a dedicated routing chip that increases the message passing performance and reduces the load on the main processor of each node. The TRC is such a chip.

The TRC provides routing of packets along two dimensions called x and y. Every TRC is connected to a processor. The processors are connected through the TRCs in an arbitrary k-ary n-cube interconnection network [3]. Structures with more than two dimensions can be constructed using cascaded TRCs as shown in [1]. In this paper we concentrate on a 2-ary 4-cube i.e. a two dimensional mesh structure with 4 by 4 nodes and wrap-around connections. A single TRC receives outgoing packets from its connected processor, and pass incoming packets to the same processor. In addition it will transfer packets that are passing on its way to another node.

A *packet* is a sequence of bytes containing the *relative address* followed by a sequence of non-zero data bytes and is terminated by a tail-byte (zero). In this paper, we often use the term *flit (flow control digit)* to denote a byte in a packet. In general, a flit need not to be the same as one byte [3].

The relative address is adjusted in each TRC a packet goes through on its way to the destination. In a two-dimensional mesh, the TRCs are given a unique (y, x) address. A TRC sends a packet along the x-dimension to the neighbour TRC with higher x-address. The node with the highest x-address is connected to the TRC with the same y-address and with x-address zero, i.e. wrap around. Similar for the y-dimension. The structure is shown in Figure 1.

The TRC uses the *wormhole routing* principle where, as long as the transmission route is free, flits are forwarded towards their destination in a pipelined or "flit-train" fashion. The main benefit compared to store and forward routing is reduced latency since we do not have to wait for the end of the packet before transmitting flits to the next node.

The TRC uses a deterministic routing scheme based on the relative x and y address. As long as the relative x-address is non-zero it is decremented and the packet is forwarded to the next TRC-chip along the x-dimension. When the relative-x-address is zero the packet is routed similarly along the y-connections (if it has not reached its final destination). *Typically* many packets will coexist in the mesh-structure. Therefore, if a TRC finds that a required

channel is being used by another packet it blocks the incoming packet until the channel is free.

The blocking of packets combined with the simple deterministic x-y-routing scheme could give rise to deadlocks caused by channel dependency cycles (circular wait). The TRC uses virtual channels to avoid this problem. Each physical channel provides two virtual channels, referred to as VC0 and VC1 in this paper. For a single physical channel, both the virtual channels may transmit data concurrently in a time-multiplexed manner. The VCs prevent deadlock by converting possible channel dependency cycles into spirals [1].

Figure 1 shows a 4 by 4 mesh of TRC modules connected along the x and y dimensions as described above. The connection between a TRC and its connected processor is not shown. Note that the connections, also called channels, are unidirectional. The x and y connections are 12 bits wide. Each consists of an eight bit channel and two pairs of request (REQ) / acknowledge (ACK) lines used to control the two virtual channels. To simplify the figure, only the data-transport direction is shown, and the ACK lines which must go in the opposite direction are not shown. The connection to the processor is 10 bits wide, 8 bits of data and one pair of REQ/ACK control lines. It is modelled as two uni-directional lines. The structure is regular and may easily be changed to other *2*-ary *n*-cubes. It is also possible to have a number of TRCs along the x dimension which differs from that in the y-direction. (The maximum size of the structure is limited by the fact that the relative x and y-addresses are stored in a byte. Thus we may have up to 256 x 256 processors connected by the TRCs).

3. Modelling the TRC in Verilog

A Verilog model of the TRC has been implemented and tested. This section presents those parts we believe are of most interest. To simplify, the first four subsections describe the system without virtual channels. In subsection 3.5 virtual channels are introduced. The modifications caused by the virtual channels on the parts of the model that already have been presented are outlined. Section 3 contains some design solutions that were rejected. These are included to enlighten limitations of Verilog and how you may walk around some of them.

3.1 The system module

The top level module, `system`, is declared as shown in the following extract. "..." denotes that code has been removed to save space. Non-verilog statements (pseudo code) are shown in *italics*.

```
// system.v
// 4x4 mesh of TRCs, numbered (y,x): (0,0)...(3,3)
parameter MAX_x = 3, MAX_y = 3;
...
wire [`TRC_DP_WIDTH-1:0] x001, x012, x023, x030,
                         x101, x112, x123, x130,
                         x201, x212, x223, x230,
                         x301, x312, x323, x330;
wire [1:0] x001_ack, x012_ack, ...
wire [`TRC_DP_WIDTH-1:0] y010, y011, ...
wire [1:0] y010_ack, y011_ack, ...
wire [`PROC_DP_WIDTH-1:0] toP0, toP1, ... toP15;
wire [`PROC_DP_WIDTH-1:0] fromP0, ... fromP15;

PROC  #(0)  p0 ( toP0,  fromP0);
...
PROC #(15) p15 (toP15, fromP15);

TRC #(0,0)   t0(x030,x030_ack,y300,y300_ack, fromP0,
               x001,x001_ack,y010,y010_ack, toP0);
...
TRC #(3, 3) t15(x323,x323_ack, ... y303_ack, toP15);
```

The size of the mesh structure is specified by the parameters MAX_x and MAX_y. The declarations of the wires and module instantiations follow the naming scheme illustrated in Figure 1. The numbering of the x-channels follows the pattern <*from-y*><*from-x*><*to-x*>, and y-channels <*from-y*><*to-y*><*to-x*>. An automatic way of generating these declarations for a given pair of values of MAX_x and MAX_y is currently not available. We will make a simple script in perl to generate this automatically. The script will be very useful when testing different mesh-configurations of TRCs. To simplify the declarations, we have included the 8 data-lines and the two REQ lines of a channel as one declaration. The ACK lines are, however, declared separately due to their opposing direction.

To test the model of the TRC chip properly we have modelled a structure of 16 processors, one connected to each of the TRCs. `PROC` is a simple Verilog behavioural model that contains tasks for sending and receiving packets. Every processor is given its own processor number as a parameter on instantiation. The processor model is a natural and practical place to include the various system-test programs. Every instance of the TRC chip knows its unique pair of x and y-address. This is useful during debugging and testing, see Section 4.

3.2 TRC main structure and the routing algorithm

The main structure of the Verilog code modelling the TRC chip is shown below. Since toggling of the REQ lines is used to tell a TRC that new data has arrived on an input channel, it is necessary to reset these to a fixed value or safe state, before transmission of data begins. In the initial block

labelled boot_trc every TRC resets its own outgoing control lines. When all its incoming control lines have been reset it knows that all its control lines are set to a proper state, and it is safe to start processing of packets. (We assume that control lines are undefined (x) initially, and they are reset by setting them to zero.) The variable ready is used to signal this condition from boot_trc to the tasks that process packets as shown.

```
module TRC(xin,  xin_ack,   yin,  yin_ack,  proc_in,
           xout, xout_ack,  yout, yout_ack, proc_out);
  parameter y_addr = 0; // (0,0) as default for (y,x)
  parameter x_addr = 0; // these are overridden when
     // making instances of the module
  input  ['TRC_DP_WIDTH-1:0]  xin;
  output               [1:0]  xin_ack;
  ...
  reg ready; //True if TRC is ready to receive packets
  task sendx ... // send one flit on x-output
  task sendy ...
  task sendp ... // to own processor
  initial begin : boot_trc
    ready = 'FALSE;
    initialization of other registers
    reset control-lines (REQ and ACK) going out
       from this TRC
    wait until all incoming control-lines
      (REQ and ACK) have been reset
    ready = 'TRUE;
  end // boot_trc

  task process_xin; // x-dimension input
    declarations
    if (ready) then
      read and process flits
  endtask / process_xin   ...
  task process_yin ... // y-dimension input
  task process_pin ... // input from own processor

  initial forever process_xin;
  initial forever process_yin;
  initial forever process_pin;
endmodule // TRC
```

This boot-process is not fault-tolerant, i.e. a failing TRC will cause neighbour TRCs to hang. However, it has not been a goal in this work to model the possibility of failing TRCs.

The central part of the TRC chip is three concurrent processes, each listening on one of the three input channels x-in, y-in and proc-in. This is modelled using three infinite loops (initial forever). The following pseudocode shows the processing of flits arriving at the x-input channel (task process_xin). Note that the value of the relative x-address denotes the number of hops *remaining* to be taken in the x-dimension and similarly for the relative y-address. Both the relative x and y addresses are included in the packet even if some of them are zero.

The processing of incoming flits on the y-input is similar, except that the relative x-address has been stripped away so that the packet starts with the relative y-address. The relative x and y-addresses of a packet are calculated by the sending processor. Consequently, a packet arriving at the processor input of a TRC can be handled in the same way as a packet arriving on the x-input.

```
// Pseudocode for task process_xin

wait for new flit on x_input
read relative_x_address from x_input

if relative_x_address is zero then
  strip it // i.e. do not pass it
  wait for relative_y_address on x_input

  if relative_y_address is zero then
    strip it // i.e. do not pass it

    wait for and read flit-data from x-input
    send flit_data to own processor

    while (flit_data <> zero) do
      wait for and read flit_data from x-input
      send flit_data to own processor

  else // relative_y_address is not zero
    decrement relative_y_address
    send relative_y_address along y_dimension

    pass all flit_data in the packet arriving
      on x-input along the y-dimension until
      the tail byte has been sent

else // relative_x_address is not zero
  decrement relative _x_address
  send relative_x_address along x-dimension

  wait for relative y-address (may be zero) on
    the x-input and pass it along the x-dimension

  pass all flit_data in the packet arriving at
    the x-input along the x-dimension until
     the tail byte has been sent
```

3.3 Self-timed TRC, asynchronous data channels

The addressing range of the TRC enables multiprocessors with as many as 64*k* processors interconnected in a mesh. To avoid the problem of distributing a global clock in such a large system, the TRC is self-timed. Each data channel uses a two-cycle signalling convention based on a pair of REQ and ACK-lines. When REQ = ACK the receiver is ready for the next flit. The sender waits for this condition before attempting to send data. It then writes the flit and toggles the REQ-line. (As we will see in Section 3.6, the sender must reserve the data lines before they are written to, and release them after having toggled the REQ-line.) The receiver waits for the REQ-line to be toggled, it reads the data and toggles the ACK-line when it is ready to receive more data. The protocol is easily modelled in Verilog.

3.4 Suspending and resuming blocked packets

Since we have three parallel main processes in a TRC, one for each of the input channels, we may easily get conflicts in the use of the outgoing channels. As an example, consider a packet being transmitted from x-input to x-output concurrently with a packet arriving on the processor-input that should be forwarded on the x-output.

When one of the three possible users (input channels) has reserved an outgoing channel, it holds the channel until the whole packet has been transmitted on it. For each byte transmitted we must ensure that the byte is part of the packet which currently has access. If not, the packet must be blocked. Consider the output channel along the x-dimension. Control of access to the output channel was implemented in the task `sendx`. This contains a parameter (`requester`) which tells whether it is asked to send a byte (`data_flit`) on the x-output on behalf of a packet arriving on either x-, y- or processor-input. Our first attempt was to use a 2-bit register called `xout_used_by` declared in every TRC-instance. Waiting for safe access was implemented as:

```
wait( (xout_used_by == requester) ||
      (xout_used_by == 'NONE));
// has got access, or free
if (xout_used_by == 'NONE)
  xout_used_by = requester; // reserve for access
```

Freeing of the channel is simply `requester = 'NONE` placed in `process_xin` when it knows that the tail-byte has been sent and received. This will *not* work correctly. It demonstrates an important side of the task concept in Verilog, which may be surprising for people with experience from stack-oriented programming languages: ***Each invocation of a task uses the same storage*****!** [4]. Assume that x-input currently has reserved (allocated) the channel and `sendx` with p-input as requester has to wait since the condition in the first line evaluates to false. When a following call to `sendx` with x-input as requester is made to send the next flit in the package, the call will change the one *shared* value of `requester` and also the call to `sendx` on behalf of p-input will proceed. Consequently, the `sendx` call on behalf of p-input will not wait long enough. This will most likely spoil both packets.

This problem was attempted to solve by declaring an event associated with the freeing of the `xout` channel. If `sendx` finds that the output channel is occupied by another packet, it starts to wait on that event. When the TRC knows that the last byte in a packet has successfully been received, it triggers the event signalling that the channel has been freed. Waiting calls to `sendx` must then do a retry for requesting the channel. However, the fact that the input argument `requester` is shared by all calls to `sendx` done by the same TRC still caused problems. When a call to `sendx` resumes from waiting it is possible that the value of `requester` has been overwritten by another call during its waiting time. Consequently, we may get situations where the retry loop allocates the channel with wrong value of `requester`. An attempt to solve this was to use two separate variables (registers) called `xWaitingReq1` and `xWaitingReq2` to store a maximum of two waiting requests. As the mix of Verilog and pseudo code below shows, these are used for storing the value of the parameter requester before starting to wait on the `xout_freed` event, and to restore the correct value of the parameter when the call to `sendx` resumes after this waiting.

```
reg [1:0] xout_used_by, xWaitingReq1, xWaitingReq2;
event xout_freed;
...
task sendx;
  input [1:0] requester;
  input 'BYTE in;
  begin
    while (!(xout_used_by == requester))
    begin // request loop
      if (xout_used_by == 'NONE)
        xout_used_by = requester;
        // you won the channel
      else
      begin
        store value of requester in either of
          xWaitingReq1 or xWaitingReq2 (one of them
          will have the value `NONE  (Note-a))
        @(xout_freed);
        restore the value of requester from one of
          xWaitingReq1 or xWaitingReq2 having value
          different from `NONE, and reset the
          chosen xWaitingReq-variable to `NONE
      end
    end // of while
    // OK to proceed now, i.e send a flit
    ...
endtask // sendx
```

Note that the correct operation of this method is based on the observation that code-sequences such as those before and after the `@(xout_freed)` statement in the while loop are executed without preemption for one TRC instance at the time. This will be the case if the sequences are without calls to wait and other constructs that imply that the simulation time is advanced. The access to the y-output channel and the processor-output to the processor is being controlled similarly.

3.5 Virtual channels

As described in Section 2 the TRC uses virtual channels to avoid the deadlock problem. Two virtual channels, called VC0 and VC1, share each physical channel. A packet is sent on VC1 until it eventually reaches the node with address zero along the current routing dimension. At this node it starts using VC0 for the remaining hops along

this dimension. This strategy is used for both the x and y-dimension.

When a packet that has been shifted over to VC0 goes from routing along the x-dimension to the y-dimension it starts by using VC1, but may shift to VC0 by the same criteria that is used along the x-dimension. One might intuitively think that it is correct to continue on VC0 as long as a packet has shifted from VC1 over to VC0. However, it is possible to get a circular wait situation solely along the y-channels in one column of TRCs, just as it is possible along the x-channels in one row. Therefore, we can not allow some packets to start along the VC0 channel, but must have the possibility to switch from VC1 to VC0 for all packets being routed along the y-dimension.

The main motivation for the virtual channels is that a blocked packet being transmitted on one channel should not block the other virtual channel. Thus, the control signals REQ and ACK used to control the transmission of data on the channel must be duplicated to give one pair of signals for each of the virtual channels. This makes it possible to transmit data concurrently on both VC0 and VC1 in an interleaved fashion [1].

A simple way to extend the TRC Verilog model to process packets that arrive on two different virtual channels was chosen. This is to split the TRC task `process_xin` into two very similar tasks, one for each of the virtual channels.

Consider VC0 and VC1 on the x-input-channel. How far the TRC has reached in the processing of a packet on one of the VCs will most often be different from the state of progress regarding an eventual packet on the other VC. Having one task `process_VC0_xin` for the VC0 channel and another (`process_VC1_xin`) for the VC1 channel solves the problem of maintaining the state of progress at each channel. The need is similar along the y-dimension, but we do not have more than one (logical) channel for data from the processor. Consequently we get 5 main tasks of this kind in the TRC model.

There are some small differences between the two tasks handling input on the same physical channel. E.g., the VC1 task must decide whether to forward the data on VC0 or VC1, whether the VC0 task knows that further hops must be on VC0.

Similarly, the task `sendx` was split into two tasks (`sendxVC0` and `sendxVC1`), one for each of the two VCs. As an example, consider a TRC chip with x-address = 0 (at the left border of the torus). Its task for processing incoming flits on VC1 should forward these along the VC0 channel on x-output if they have not reached their destination x-address. The task that is processing flits arriving from the processor connected to the same TRC will send flits along the VC1 of the x-output.

Packets that should be forwarded along one virtual channel should not be blocked by traffic on the other virtual channel. If we reconsider the way of controlling access to the channel presented in section 3.4 we see that `sendxVC0` should use one set of `xWaitingReq` variables while `sendxVC1` uses another set.

The introduction of Virtual Channels violates the assumption made in section 3.4. that a maximum of two requesters may wait on the same (now virtual) channel. Consider the following example situation where a TRC-chip is handling a long package from its own processor by sending flits on its y-output VC1. Simultaneously the same TRC may receive flits on both VC0 and VC1 of the x-input, and both these packets may have reached their column along the x-dimension so that they should start travelling along the VC1 of y-output. In addition, it may arrive a packet on VC1 of y-input that should continue on VC1 on y-output.

It was then considered to extend the Verilog code for this access control even further by allowing up to four waiting requests. While unhappily thinking of extending the complexity of the model in this way, another problem in the model was found. Again it was caused by Verilog tasks being very different from procedures in stack oriented languages.

When the waiting for access is implemented in the way described in section 3.4, different requesters may call the same task for sending a byte on a virtual channel, e.g. `sendxVC0`. When the virtual channel is freed, these calls are resumed from their waiting state. There are two problems that may arise when such suspended tasks are "back in business".

The first problem is (as we have seen before) that the input parameter (in this case `in`, which is the flit to send) may have been overwritten by another call to the same task. This may be solved by storing the input parameter in an array indexed by the requester. When the task is resumed after waiting, it checks the `xWaitingReq` variables, picks one of the waiting requests, and sends the value of `in` stored by the requester that gets the virtual channel. This technique complicates further, but seems to work at first sight.

The second problem is caused by the fact that we cannot (and perhaps should not) control which call to the `sendxVC0` task that is resumed first. An arbitrary of the waiting calls will according to our algorithm pick a requester and send a byte on behalf of it. As an example, a call to `sendxVC0` done by `process_pin` may be resumed and may send a byte on behalf of a packet arriving on the x-VC1-input (`process_VC1_xin`). After having sent the byte, the send-call returns and task `process_pin` believes that it may proceed to send the next byte, which is wrong. Also, the task `process_VC1_xin` does not proceed even though one of its byte have just been sent. This bug was

hard to find! If a task could get access to information about the identity of its caller it would again be possible to proceed with this solution at the price of increased complexity. As far as I know, this is not possible in Verilog. The introduction of the `xWaitingReq` variables turned out to be a "dead end".

The need for a call to `sendxVC0` to know the identity of its caller can be circumvented by letting each caller-task (`process_VCO_xin`, `process_VC1_xin`, etc.) have its own variant of `sendxVC0` (called `X0sendxVC0`, `X1sendxVC1`, etc.). When the sendx-task is resumed after waiting it knows the identity of its caller through its own identity!. Moreover, this solution removes the need for the `xWaitingReq` variables and the storing of input parameters since there will be only one call to each of the sendx-tasks at the same time. This simplifies a lot, but the drawback of the solution is that we need a lot of tasks like `X0sendxVCO` that are very similar but not completely equal. (There are many combinations, however not all are needed.).

3.6 Controlling access to the datalines

When the TRC starts operation we do not generally know which of the two VCs that first will be used on a given x- or y-output line. Both must be initialised with REQ = ACK, i.e. ready for transmission. If for instance VC1 uses the data-lines first and the packet is blocked we will have `REQ1 != ACK1` until the packet is unblocked. In the meantime it must be possible to use VC0. Therefore, the data lines must be freed for other use even if the receiver is not ready for receiving more data on the other virtual channel. But, we must also assure that the two virtual channels do not use the data-lines at the same time.

We assume that the receiver is always able to read one byte on the currently used virtual channel during the data transmission time. This is possible since the receiver will only toggle the ACK line when it is ready for more data. The access to the data lines during the transmission time is controlled in `sendx_VC0` as shown below. The register `xdu` (**x d**ata lines **u**sed) is used to ensure mutual exclusion of the two users. (`sendxVC1` is similar).

```
wait(REQ0 == ACK0);
// ready to send data, must get data-lines alone
while (xdu) // while datalines are in use
  #1; // wait a small amount of time
xdu = 'TRUE; // reserve it for use
write to datalines
toggle REQ
wait long enough to be sure that the receiving
  process has read the data
xdu = 'FALSE; // release for use by others
```

The variable `xdu` is local to each TRC. The correct operation of this method is based on the same observation about non-preemption of code-sequences as was mentioned at end of Section 3.4.

To summarise, the solutions presented in this and the two preceding subsections implement controlled resource sharing at two different levels. Sections 3.4 and 3.5 explain how virtual channels are reserved at the packet level using Verilog events. This subsection explained how mutual exclusion concerning the physical data lines was implemented at the flit level by a lock-variable and busy waiting.

4. Testing, experience and future work

4.1 Testing

The structure of 16 TRCs serving 16 independent processors in a 2D mesh has been tested by several small Verilog test programs. They are typically written as a separate task that is included in the Verilog model of the processors. The same task is thus executed by all processors. However, the test programs typically use conditional statements and the unique processor number known by every processor instance to control which processor(s) should send the given packet(s) at the given time. This corresponds to the *SPMD* (Single Program Multiple Data) *programming style* that currently is the most popular parallel programming style.

Each TRC has three input channels and three output channels being handled concurrently, with two virtual channels on each of the x- and y-channels. The processors may send and receive packets concurrently. With 16 processors and TRCs we have close to 200 parallel activities in the system being debugged. In such a system it is important to start the debugging with simple tests to avoid being drowned in debugging information. The first set of tests typically transmitted a single packet between a given pair of TRCs. The next step was to send a single packet to a random destination, and let the receiver forward the message to another random destination. This "random walk test" of a *single* packet for a large number of iterations gave a good test of the routing algorithm. (Correct routing was checked automatically by the destination by including the destination-address in the packet being sent.)

To test how the virtual channels solve the deadlock problem in a proper way it was necessary to let all the processors send and receive packets concurrently in a random manner. Before the virtual channels were properly implemented, such tests relatively fast gave a deadlock situation. The current version of the model has been tested by letting each of the 16 processors send 1000 packets of

random length concurrently at random times, and all arrive correctly.

4.2 Experience

Every instance of the TRC and the PROC modules in the Verilog model has a local one bit register `debug` initialised to false. The code is instrumented with debug statements of the form `if (debug) $display("%m ....")`. The use of *hierarchical naming* makes it very easy to switch on and off the debugging information for a given set of TRC or PROC instances at given values of the simulation time. This was particularly useful in late phases of the debugging, to avoid being drowned in debug information. Often, we were only interested in what was happening on one or two of the links in the 4 by 4 torus. In a system with many modules, the "%m" giving the hierarchical name of the current instance saves a lot of typing in such debug statements.

When tracking down errors in the code for virtual channels that gave deadlock situations it was very useful to have a simple *diagnostic module* outside the main parts of the system. This module was invoked at the end of the simulation and displayed the x/y/p_used_by variables in a convenient format. Again hierarchical naming showed to be very practical, since it allows the designer to probe exactly those values that are of interest.

The difference between a programming language and a hardware description language is visualised by the task concept in Verilog. In the context of the TRC hardware we have been modelling, we cannot assume a stack for pushing and popping of parameters and local variables. But, even with shared storage, the task concept is very useful for making structured and more readable code. However, with "to much" experience with procedural abstractions in high level programming languages it is easy to make errors as those described in section 3.

Our experience from the development of the TRC simulation model in verilog can be summarized in one sentence; *"Verilog is a practical language for developing architectural simulations, but remember that tasks are not procedures!"*.

4.3 Future work

The current model has only a very crude representation of the time consumed in the various parts of the TRC behaviour. We believe this has been sufficient for testing the functionality of the model. A more detailed time modelling will be developed to make the model more accurate regarding performance.

We also want to make one or several models of the same TRC chip but at a more detailed level. A model at the RTL level following the HW structure described by Dally and Seitz [1] is being developed by Pauline Haddow [2] and will give more precise performance figures that will be used to improve the existing model. The next step would be to modify it into a RTL model than can be synthesised. A comparison of readability, size and performance accuracy of such TRC models at different levels would be very interesting.

Further tests will be developed to do performance studies at the architectural level. We will use message traces to represent more realistic load on the system than what is done by the synthetic load from our test programs.

Acknowledgements

The author wishes to thank Nordic VLSI for a stimulating working environment, and Pauline Haddow for discussions about the TRC routing chip and for comments on a draft of the paper.

References

[1] Dally, William J. and Seitz, Charles L., *The torus routing chip*, Distributed Computing (1986) 1:187-196.

[2] Haddow, Pauline, *A Generalisation of Router Chip Design, In proceedings of* 5th Euromicro Workshop on Parallel and Distributed Processing, London, January 1997.

[3] Hwang, Kai, *Advanced Computer Architecture, Parallelism, Scalability, Programmability.* McGraw Hill, 1993.

[4] Palnitkar, Samir, *Verilog HDL: A Guide to Digital Design and Synthesis*, SunSoft Press, Sun Microsystems Inc. 1996.

[5] Smith, Douglas J, *VHDL & Verilog Compared & Contrasted - Plus Modelled Example Written in VHDL, Verilog and C*, from 33rd Design Automation Conference.