Technical Memo 5.5.2009 / rev. 22.8.09

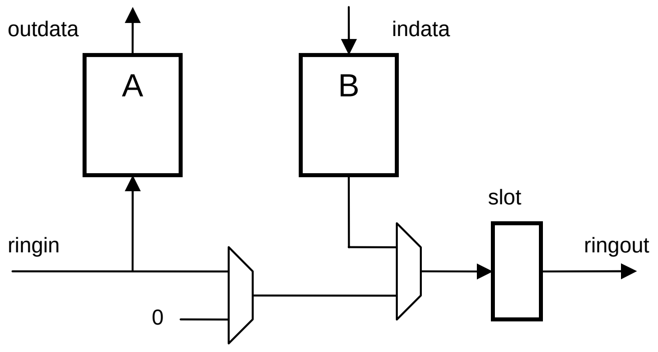
A Token-Ring for the TRM

Niklaus Wirth

With the design of the Token-Ring for the TRM (Tiny Register Machine) I pursued mainly two aims. The first is to design a network connecting several TRM cores. The second is to go for a design that is as simple as possible, considering that the TRM project is oriented towards educational hard- and software. Featuring a ring architecture, it provides a welcome alternative to the already existing bus architecture implemented by Ling Liu, allowing to compare complexity and performance.

1. The Structure of the Ring Node

The ring structure is particularly suitable, if the hardware does not feature any tri-state gates. This is the case for Xilinx FPGAs, where our system is implemented. Tri-state gates are the basis of typical, bi-directional bus architectures. A ring, in contrast, transmits in one direction only. A ring node features a ring input, a ring output, and an interface to a TRM core. Each node contains a register between the ring input and ring output. This register holds one data element and introduces a latency (delay of the data traveling through the ring) of a single clock cycle. The node also contains two buffers, one for the received data, and one for the data to be sent. Their purpose is to decouple the nodes in time and thereby to increase the efficiency of the connections. We postulate that the data are always sequences of bytes, and they are called *messages*.



Ring Node

The figure shows that if a node is sending a message over the ring, buffer B is fed to the ring output. If a message is received, the ring input is fed to buffer A. Otherwise the input is transmitted to the output, with a single cycle's delay.

We have chosen the elements of messages to be bytes. The ring and its registers, called *slots*, are 10 bits wide, 8 for data and 2 for a tag to distinguish between data and control elements.

2. The Structure of Messages, and Invariants

Messages are sequences of (tagged) bytes. The first element of a message is a *header*. It indicates the destination and the source number of the nodes engaged in the transmission. We assume a maximum of 16 nodes, resulting in 4-bit node numbers. The last element is the *trailer*. In between lie an arbitrary number of data items.

tag	data	
10	source, destination	header
0x	xxxxxxxx	data byte
01	00000000	trailer
11	00000000	token

When no messages are to be transferred, the ring is said to be *idle*. When a node is ready to send a message, it must be granted permission in order to avoid collisions. One may imagine a central agency to rotate a pointer among the nodes, and the node so designated having the permission to send its message. As we wish to avoid a central agent, we instead insert a special element into the ring which takes over the role of the pointer. It is called the *token*. In the idle state, only the token is in the ring.

Only a single message can be in the ring. When a node is ready to send a message, it waits until the token arrives, and then replaces the token by the message. The token is reinserted after the message. The header contains the number of the destination node which triggers the receiver to become active. When the message header arrives at the destination, that node feeds the message into its receiver buffer. It removes the header from the ring by replacing the slot with a zero data item.

3. The Hardware Implementation

This simple scheme was implemented in Verilog for a Xilinx FPGA. The module interface is described below. It consists of the ring input and ring output, and of the connections to the associated TRM processor.

```
module RingNode(clk, rst, wreq, rreq, ringin, ringout, indata, status, outdata);
input clk, rst wreq, rreq;
input [9:0] ringin;
input [9:0] indata;  // from processor
output [9:0] ringout;
output [7:0] status;
output [9:0] outdata;  // to processor

reg sending, receiving, rdyS;   // states
reg [5:0] inA, outA, inB, outB;   // buffer indices
reg [9:0] slot;    // element in ring
```

The buffers are implemented as LUT memories with 64 elements, 10 bits wide. Registers *inA* and *outA* are the 6-bit indices of the receiving buffer A, *inB*, *outB* those of the sending buffer.

There are 4 separate activities proceeding concurrently:

1. A byte is fed from *indata* to the sending buffer B, and the pointer *inB* is advanced (incremented modulo buffer size). This is triggered by the input strobe *wreq* with *ioadr* = 2.

2. A byte is transferred from buffer B to the ring, and pointer *outB* is advanced. This happens in the sending state, which is entered when the buffer contains a message and the token appears in the slot. The sending state is left when a trailer is transmitted.

3. A byte is transferred from the slot (ring input) to the receiver buffer A and the pointer *inA* is advanced. This is triggered by the slot containing a message header with the receiver's number. A zero is fed to the ring output.

4. A byte is transferred from buffer A to *outdata*. This happens when the TRM reads input. Thereafter the TRM must advance pointer *outA* by applying a *wreq* signal and *ioadr* = 1

The four concurrent activities are expressed in Verilog as shown below in one block clocked by the input signal *clk*. The input of buffer A is *ringin*, that of buffer B is *indata* (input from processor).

```
wire startsnd, startrec, stopfwd;
wire [9:0] A, B;   // buffer outputs

assign startsnd = ringin[9] & ringin[8] & rdyS;  //token here and ready to send
assign startrec = ringin[9] & ~ringin[8] & ((ringin[3:0] == mynum) | (ringin[3:0] == 15));
assign stopfwd = ringin[9] & ~ringin[8] & ((ringin[3:0] == mynum) | (ringin[7:4] == mynum));

assign outdata = A;
assign status = {mynum, sending, receiving, (inB == outB), (inA == outA)};
assign ringout = slot;

always @(posedge clk)
  if (~rst) begin  // reset and initialization
    sending <= 0; receiving <= 0; inA <= 0; outA <= 0; inB <= 0; outB <= 0;
    rdyS <= 0;
    if (mynum == 0) slot <= 10'b1100000000; else slot <= 0; end
  else begin
    slot <= (startsnd | sending) ? B : (stopfwd) ? 10'b0 : ringin;
    if (sending) begin // send data
      outB <= outB + 1;
      if (B[9] & B[8]) sending <= 0; end  // send token
    else if (startsnd) begin // token here, send header
      outB <= outB + 1; sending <= 1; rdyS <= 0; end

    else if (wreq) begin
      inB <= inB + 1; // msg element into sender buffer
      if (indata[9] & indata[8]) rdyS <= 1; end

    if (receiving) begin
      inA <= inA + 1;
      if (ringin[8]) receiving <= 0; end   // trailer: end of msg
    else if (startrec) begin // receive msg header
      inA <= inA + 1; receiving <= 1; end

    if (rreq) outA <= outA + 1;   // advancing the read pointer
  end
```

4. The Software Implementation

The pertinent driver software is described in Oberon. It is responsible for the maintenance of the prescribed protocoll and message format, and it is therefore presented as a module. This module alone contains references to the hardware through procedures PUT, GET, and BIT. Clients are supposed not to access the hardware interface.

The module encapsulates and exports procedures *Send* and *Rec*, a predicate *Avail* telling whether any input had been received, and a function *MyNum* yielding the node number.

```
MODULE Ring;

  PROCEDURE Avail*(): BOOLEAN;
    VAR status: SET;
  BEGIN GET(0F03H, status); RETURN ~(0 IN status)
  END Avail;

  PROCEDURE Send*(dst, typ, len: INTEGER; VAR data: ARRAY OF INTEGER);
    VAR i, k, w, header: INTEGER;
  BEGIN REPEAT UNTIL BIT(0F03H, 1);  (*buffer empty*)
    GET(0F03H, header); header := MSK(header, 0F0H) + MSK(dst, 0FH);
    PUT(0F02H, header + 200H); PUT(0F02H, MSK(typ, 0FFH)); i := 0;
    WHILE i < len DO
      w := data[i]; INC(i); k := 4;
      REPEAT PUT(0F02H, MSK(w, 0FFH)); w := ROR(w, 8); DEC(k) UNTIL k = 0
    END ;
    PUT(0F02H, 100H); PUT(0F02H, 300H)  (*trailer, token*)
  END Send;

  PROCEDURE Rec*(VAR src, typ, len: INTEGER; VAR data: ARRAY OF INTEGER);
    VAR i, k, d, w, header: INTEGER;
  BEGIN
    REPEAT UNTIL ~BIT(0F03H, 0);  (*buffer not empty*)
    GET(0F02H, header); src := MSK(ROR(header, 4), 0FH);
    GET(0F02H, typ); GET(0F02H, d);
    i := 0; k := 4; w := 0;
    WHILE MSK(d, 300H) = 0 DO
      w := ROR(MSK(d, 0FFH) + w, 8); DEC(k);
      IF k = 0 THEN data[i] := w; INC(i); k := 4; w := 0 END ;
      GET(0F02H, d)
    END ;
    len := i
  END Rec;

  PROCEDURE MyNum*(): INTEGER;
    VAR x: INTEGER;
  BEGIN GET(0F03H, x); RETURN MSK(ROR(x, 4), 0FH)
  END MyNum;

END Ring.
```

Because the TRM is a word-addressed machine, the data to be transmitted are arrays of integers, whereas the ring interface transmits bytes as elements of a packet. Each array element must therefore by sent over the ring as four bytes. Procedure *Send*, after composing and sending the header byte, unpacks each integer into 4 bytes with the aid of a rotate instruction (ROR). The *Rec* procedure packs 4 consecutive bytes into an integer (word) by rotating and masking. (The second byte of each message is the parameter *typ*, which is not used in this context).

In PUT and GET operations, the first parameter indicates the address of the interface to be accessed. Here 0F02H is the address of the data port, and 0F03H that of the status. The status consists of 8 bits. It contains the following fields:

bit 0	input buffer empty (in = out)
bit 1	output buffer full (in = out)
bits 2, 3	0
bits 4-7	ring node number

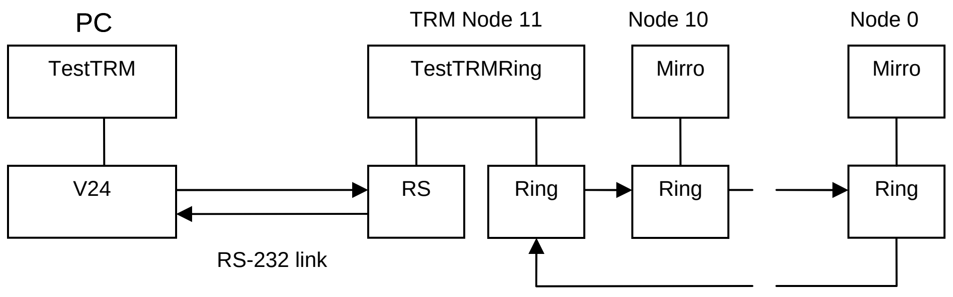
5. A Test Setup

For testing and demonstrating the Ring we use a simple test setup. It involves the program *TestTRMRing* for node 11, and the identical program *Mirror* for all nodes 0 – 10. The former is connected via an RS-232 link to a host computer running a general test program *TestTRM* for sending and receiving numbers.The main program *TestTRMRing* (running on TRM) accepts commands (via RS-232) for sending and receiving messages to any of the 12 nodes. Program *Mirror* then receives the sent message and returns it to the sender (node 11), which buffers it until requested by a read message command.

Communication over the link is performed by module RS, featuring procedures for sending and receiving integers and other items. The following are examples of commands:

```
TestTRM.SR 1 3 10 20 30 40 50 0 0~
TestTRM.SR 1 8 0 0~
TestTRM.SR 1 3 10 0 4 11 12 0 5 13 14 0 7 15 16 17 0 0~
TestTRM.SR 2~   receive message
```

The first command sends to node 3 the sequence of numbers 10, 20, 30, 40, 50. The second sends the empty message to node 8, and the third sends to node 3 the number 10, to node 4 the items 11 12, to node 5 the numbersw 13, 14, and to node 7 the numbers 15, 16, 17.



The test setup

```
MODULE TestTRMRing;
  IMPORT RS, Ring;
  VAR cmd, dst, src, x, len, typ, s, i: INTEGER;
    buf: ARRAY 16 OF INTEGER;
BEGIN
  REPEAT RS.RecInt(cmd);
    IF cmd = 0 THEN RS.SendInt(Ring.MyNum())
    ELSIF cmd = 1 THEN  (*send msg*)
      RS.RecInt(dst);
      REPEAT len := 0; RS.RecInt(x);
        WHILE x # 0 DO buf[len] := x; INC(len); RS.RecInt(x) END ;
        Ring.Send(dst, 0, len, buf); RS.RecInt(dst)
      UNTIL dst = 0;
      RS.SendInt(len)
    ELSIF cmd = 2 THEN  (*receive msg*)
      IF Ring.Avail() THEN
        Ring.Rec(src, typ, len, buf);
        RS.SendInt(src); RS.SendInt(len); i := 0;
        WHILE i < len DO RS.SendInt(buf[i]); INC(i) END
      END
    ELSIF cmd = 3 THEN RS.SendInt(ORD(Ring.Avail()))
    END ;
    RS.End
  UNTIL FALSE
END TestTRMRing.

MODULE Mirror;
  IMPORT Ring;
  VAR src, len, typ: INTEGER;
    buf: ARRAY 16 OF INTEGER;
BEGIN
  REPEAT Ring.Rec(src, typ, len, buf); Ring.Send(src, 0, len, buf) UNTIL FALSE
END Mirror.
```

6. Broadcast

The design presented here was, as already mentioned, intentionally kept simple and concentrated on the essential, the transmission of data from a source to a destination node. A single extension was made, first because it is useful in many applications, and second in order to show that it was easy to implement thanks to a sound basis. This is the facility of braodcasting a message, that is, to send it to all nodes. The ring is ideal for this purpose. If the message passes once around the ring, simply all nodes must be activated as receivers. We postulate that address 15 signals a broadcast. Then are only two small additions to the circuit are necessary, namely the addition of the term *ringin[3:0] = 15* in the expression for *startrec*, and of the term *ringin[7:4] = mynum* in that of *stopfwd*.

7. Discussion

The presented solution is remarkably simple and the Verilog code therefore brief and the circuit small. This is most essential for tutorial purposes, where the essence must not be encumbered by and hidden in a myriad of secondary concerns, although in practice they may be important too.

Attractive properties of the implementation presented here are that there is no central agency, that all nodes are perfectly identical, that no arbitration of any kind is necessary, and that the message length is not a priori bounded. No length counters are used; instead, explicit trailers are used to designate the message end. All this results in a simple and tight hardware.

The data path of the ring is widened by 2 bits, a tag for distinguishing data from control bytes, which are token, message header, and message trailer. Actually, a single bit would suffice for this purpose. Two are used here in order to retain an 8-bit data field also for headers containing 4-bit source and destination addresses.

The simplicity has also been achieved by concentrating on the basic essentials, that is, by omitting features of lesser importance, or features whose function can be performed by software, by protocols between parners. The circuit does not, for example, check for the adherence to the prescribed message format with header and trailer. We rely on the total “cooperation” of the software, which simply belongs to the design. In this case, the postulated invariants can be established and safeguarded by packing the relevant drivers into a module, granting access to the ring by exported procedures only.

A much more subtle point is that this hardware does not check for buffer overflow. Although such overflow would not cause memory beyond the buffers to be affected, it would overwrite messages, because the buffers are circular. We assume that overflow of the sending buffer would be avoided by consistent checking against pending overflow before storing each data element, for example, by waiting for the buffer not being full before executing any PUT operation:

```
REPEAT UNTIL ~BIT(adr, 1)
```

In order to avoid blocking the ring when a message has partially been stored in the sending buffer, message sending is not initiated before the message end has been put into the buffer (signal *rdyS*). This effectively limits the length of messages to the buffer size (64), although several (short) messages might be put into the buffer, ad messages being picked from the buffer one after the other.

A much more serious matter is overflow of the receiving buffer. In this case, the overflowing receiver would have to refuse accepting any further data from the ring. This can only be done by notifying the sender, which is not done by the presented hardware. For such matters, communication protocols on a higher level (of software) would be the appropriate solution rather than complicated hardware.

We consider it essential that complicated tasks, such as avoiding overflow, or of guaranteeing proper message formats, can be left to the software. Only in this way can the hardware be kept reasonably simple. A proper module structure encapsulating a driver for the ring is obviously necessary.