Basic Building Blocks for Asynchronous Packet Routers

I.M. Nedelchev C.R. Jesshope
Department of Electronic Engineering
University of Surrey
Guildford GU2 5XH, England

Abstract
Propagating the clock through large networks and providing correct functioning of the system is a serious engineering problem. The clock appears at different moments for two different physical points — clock skew problem. While the clock skew can be neglected for small systems, it results in major problems when building large concurrent networks. To overcome such problems we believe that the absolute solution is to eliminate the notion of clocking entirely throughout by adopting asynchronous design techniques.

Packet switches are familiar components of nowadays concurrent architectures and a good example to illustrate asynchronous design. This paper describes the asynchronous implementation of these basic building blocks for asynchronous packet routers and also demonstrates asynchronous design techniques for VLSI design.

1 Introduction
The paper presents the implementation of three asynchronous blocks with which one can build packet routing nodes for direct networks, i.e., we consider networks where there is a routing node (RN) attached to each processing element (PE). The PE injects and consumes messages on channels with which the PE is connected to the RN. In this paper we consider only mesh topologies, so each RN is connected with 4 such pairs of channels to its neighboring RNs thus conforming grid topology. It is a well known problem that deadlock can arise in such highly concurrent networks. Depending on the chosen strategy for deadlock avoidance the node architecture of these routing nodes is different, but the set of building blocks is the same — asynchronous multiplexer (MUX), address decrementor (DEC) and routing switch (RT). Each of these blocks (MUX, DEC and RT) has been specified in a CSP like notation called Delay Insensitive Algebra (DIA) [8, 7]. A compilation scheme has been developed that translates expressions in the algebra into a netlist of basic library circuits for asynchronous design [2, 3]. It has been applied to the specification of the different building blocks and thus the final circuit has been produced.

In the following section we will introduce a set of asynchronous library elements used throughout the design and then the asynchronous specification and implementation of these three blocks will be discussed.

2 Asynchronous library cells
Merge is usually implemented as a logical XOR gate. Each incoming event on a input is copied out onto its output.

Sparse Decision Wait (SDW) element is based on the definition of Decision Wait element described in [8].

\[
SDW_{n,m} = [\{r_i \rightarrow [\{s_{i,j} \& c_j \rightarrow o_{i,j}]; SDW_{n,m}\}] \]

where \(s_{i,j}\) is a vector of boolean values and if some \(s_{i,j}\) is FALSE the cell \(i,j\) is missing in the graphical symbol for the DW. The circuit symbol for the SDW element is shown in figure 1.a where \(s_{i,j}\) is FALSE for \(1 \leq i \leq 3\).

Toggle copies each odd input onto the bullet signed output and the other main events - onto the second output (figure 1.b).

\[Toggle = \text{in} \vdash \text{out}_{\text{s}} \vdash \text{in} \vdash \text{out} \vdash \text{Toggle}\]

Two-way Arbiter is a device that gives two processes mutually-exclusive access to a single shared resource (figure 1.c). The behaviour of the arbiter is defined by the following algebraic expression:

\[
A = [\{r_{in_0} \rightarrow r_{out_1}; a?; a_1; A_1\] \[
A_1 = \{r_{in_1}; r_{out_1}; a?; a_1; A\]

The request signals \(r_{in_0}\) and \(r_{in_1}\) can be concurrent. In this case one of them is blocked until the resource has been released by the other process.

Select is a device with one input \(\text{in}\) and \(n\) outputs \(\text{out}_i\) [9]. \(s\) is a binary value which specifies which output is currently selected. An incoming event on \(\text{in}\) is copied on \(\text{out}_s\).
Select = in\(^i\); out\(^i\); Select

Register is a device with \(n\) store and \(m\) pass inputs and corresponding acknowledges for both sets [9]. When there is an input store on some of the set, the register holds the values on data.in valid on data.out until there is another consequent event on pass input where the register simply copies the values from data.in on data.out. The specification for correct behaviour of the register concerning only the control signals store and pass follows:

\[
REG = [\left(\begin{array}{c}
\forall i, s_i^j \rightarrow s_{a_i}^j; \\
\forall i, p_i^j \rightarrow p_{a_i}^j; \\
\end{array}\right) \mid REG]
\]

3 Communication channels and packets format

All building blocks in this paper communicate on channels of one and the same type. Some of the channel’s wires carry the logical values of the data, the rest of the wires are control ones which provide correct clocking scheme for node-to-node communication. In this section we describe the meaning of each wire of the communication channel.

We use bundled data convention for data transmission and 2 phase signaling [9] for the control wires. By 2 phase signaling we mean that an asynchronous event is a voltage transition (0 → 5V, 5V → 0). Each channel is physically represented by several control/data wires:

- data\(_i\), \(i \leq n\): are the logical values of the current data word.
- req is an event which latches the valid data on data\(_i\).
- e is an event which indicates end of packet and it also latches the initial control word on data\(_i\). In the latter case e might be regarded as a start of the packet event.
- ack is an event which acknowledges the data from the recipient. As sometimes the existence of two acknowledge type of wires simplifies the design, we will use ack\(_e\) and ack\(_r\) for the corresponding acknowledgments for data with event e and req. In the latter case, an acknowledge ack is a result of simply merging ack\(_e\) and ack\(_r\).

The routing nodes receive and send out packets on the described above communication channels. The format of these packets can be formally expressed by the following expression: e, req → e. The first event e carries the control word which consists of control field - \(n\) bits and relative address of the destination - \(m\) bits. Consider channels with data 16 bits wide. For building 2 dimensional routing networks 2 bits are sufficient for the control field, so we can build meshes with \(2^{14} \times 2^{14}\) nodes. As we mentioned before we use relative addressing. Each address field contains the number of the hops to be travelled on the current dimension. On each hop this relative address is decremented by 1. When some relative address becomes zero, the control field in the control word indicates where the rest of the message should be copied to - reinjected on another dimension or consumed by the processing element.

After the initial control word the packet continues with data words latched by req. In fact this data words can be also control words but they will be interpreted as such by the routing nodes further on the message path. The second incoming event e indicates the end of the packet. Each event e or req is acknowledged on the corresponding channel ack\(_e\)/ack\(_r\) or if it is off-chip interface - on channel ack.

4 Asynchronous packet multiplexer - MUX

Block MUX serialises the concurrent request from its two input channels in\(_1\) and in\(_2\) on the output channel out (figure 2). The specification of MUX process is as follows:

\[MUX = [\begin{array}{c}
D_{ch} \in (in_1, e) \\
\forall e, \forall x \rightarrow e!x; ack_e!; CP_{ch} \\
\forall e, \forall x \rightarrow req!x; ack_r!; CP_{ch} \\
\forall e \rightarrow e!x; ack_e!; CP_{ch}
\end{array}]\]

Actually only the initial start of packet events e are concurrent therefore an arbitration need to be involved for resolving possible conflicts for only these signal events. All consequent requests on req lines are mutual exclusive and they can be simply merged. After the arbitration process has been resolved, the direction which input data is copied on the output channel out is determined. Therefore the result from the arbitration process controls the multiplexion for the data inputs. There is no data processing in MUX element. Its task is to transmit one incoming packet from an input channel to its output out in noninterleaving fashion on current word level regarding the other input. One possible implementation of MUX process is shown in figure 2. If a MUX block with more inputs is required it can be built out as a composition of several MUX blocks with two inputs.

![Figure 2: The implementation of MUX](image-url)
5 Address decrementor - DEC

In this section we present the address decrementor (figure 3). As we mentioned in the previous sections, we use relative addressing. On each hop the relative address of the target is decremented by 1. As the relative address and the control field are in the initial control word, DEC has to decrement only the address field from the current incoming word latched with event e. The specification of DEC follows:

\[
DEC = \begin{cases} \epsilon_{i,n}[y : z]; \text{ack}_e!; e_{\text{out}}[y - 1 : z]; C \text{P} \\ \epsilon_{i,n}[x \rightarrow \text{ack}_e!; e_{\text{out}}[x]; \text{DEC} \\ \text{req}_{\text{in}}[x \rightarrow \text{ack}_e!; \text{req}_{\text{out}}[x]; C \text{P} \end{cases}
\]

Here we specify by \([y : z]\) a variable which contains fields: \(y\) - the address field and \(z\) - the control field. With the first event \(e\), DEC accepts the control field and the relative address. It sends out the control field unchanged but the relative address is decremented by 1. With each consequent request on \(\text{req}\) and \(e\) DEC transmits the data unchanged on the output channel. One possible implementation is shown in figure 3.

![Figure 3: Implementation of DEC process](image)

The element \(\text{decel}\) is a functional block which decrements the input data by 1 if the input \(\text{op}\) is logical false. Otherwise it leaves the data unchanged. Because of these two modes of operation the delay lines (\(\text{delay}_1\) and \(\text{delay}_2\)) for \(e\) and \(\text{req}\) are different. These two delay lines are necessary to preserve the correct timing on the bundle of wires: \((e, \text{req}, \text{data})\).

6 Routing block - RT

The routing block RT (figure 4) is the most complex from the three building blocks. It has one input channel in \((\text{data}_{in}, e, \text{req}, \text{ack}_e, \text{ack}_{\text{req}})\) and as many as necessary for the specific implementation output channels \(\text{out}_{\text{ch}_i}\) \((\text{data}_{out}, \text{ch}_i, \text{req}_{\text{ch}_i}, \text{ack}_{\text{ch}_i}, \text{ack}_{\text{req}, \text{ch}_i})\). The DIA specification of RT routing block follows:

\[
RT = \begin{cases} \epsilon[x]; \text{store}!; x; \text{ack}_e!; \\ \text{if} (-zh) \text{ then } e_{ch_i}[x]; \text{ack}_{ch_i}[x]; \text{Copy}_{ch_i} \\ \text{else } \text{req}[x]; \text{ack}_e!; e_{ch_i}[x]; \text{ack}_{ch_i}[x]; \text{req}_{ch_i}[x]; \text{ack}_{\text{req}, \text{ch}_i}[x]; \text{Copy}_{\text{ch}_i} \end{cases}
\]

\[
CP_{ch} = \begin{cases} \text{req}[x] \rightarrow \text{ack}_e!; \text{req}_{\text{ch}_i}[x]; \text{ack}_{\text{ch}_i}[x]; \text{req}_{\text{ch}_i}[x]; \text{ack}_{\text{ch}_i}[x]; \text{RT} \\ \text{C} \text{P} \end{cases}
\]

It accepts the control word of the packet with the first incoming event \(e\) and if the relative address is not zero \((zh = false)\) then \(RT\) copies the control word and the rest of the message on the direct channel \(ch_i\). (We stipulate that channel \(ch_i\) is the direct one and its binary representation in the control field \(cf\) from the first control word is zero.) If the relative address is zero \((zh = true)\) then the routing block \(RT\) opens channel \(cf_{ch_i}\) with event \(e\) where \(cf\) is the control field from the control word as a binary value. This control word is saved by operation \(\text{store}!\) in an additional control register as we can see later from the implementation. One possible implementation is shown in figure 4. The boolean vector \(sc\) is defined as \((zh \land cf_1, ..., zh \land cf_n)\), where \(cf_i\) is the \(i\)th bit from the control field.

![Figure 4: Implementation of RT process](image)

7 Packet switching device using oblivious routing

Oblivious routing for mesh topology is a well known strategy for deadlock avoidance [6]. Basically the routing is initially performed on dimension \(X\) and then on dimension \(Y\) (first \(X\) then \(Y\)). Each routing node is connected to its neighbours with 4 pairs of channels - input and output one. There are another two channels for injection and consumption of the message. For each input channel there is a routing process with an assigned buffer and because the routing is performed in first \(X\) then \(Y\) way there is no possibility for channel dependency cycles [5]. The latter is a sufficient condition for deadlock freedom. One possible implementation based on the basic blocks is shown in figure 5.

On injection the address field of the first control word is zero. The control field contains the 'address' of the direction to be injected - \((X, -X, +Y, -Y)\) which is the address of an output channel for the injection \(RT\) node. Each consequent control word of the message carries the relative address of the destination for the current dimension. The con-
9 Conclusions

The paper presents an asynchronous implementation of some basic blocks for packet routing switches. It also demonstrates an asynchronous design for creating a correct clocking scheme for the different functional blocks. The design presented in the paper is an enhancement of a previous asynchronous implementation of a fully delay-insensitive bit-serial mad postman packet switch [1]. Our aim was to preserve the delay-insensitivity of the control unit but also to implement bundled data channels with data processing based on conventional logic design.

10 Acknowledgements

We gratefully acknowledge the support of the European Commission under ESPRIT III Basic Research Working Group 7225 (ACiD WG) and SERC under grant no GR/G 18605.

References


[9] Ivan Sutherland, "Micropipelines" CASM 6 1989