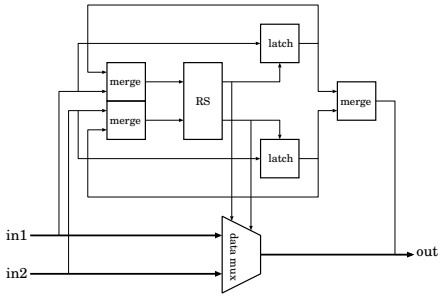

Figure 19: Implementation of the Asynchronous Multiplexor

produces the acknowledgements for storage *aw1/aw2*. The delay assumption (resulting from the chosen control input of the DW) in this implementation is that the delay of the storage (in *areg*) is less than the delay of the DW plus the environment delay between the acknowledgement and the read signal *r1/r2*. The production of *Di,ari* is simply a bundling of the data output of *areg* with the read signal *ri*.

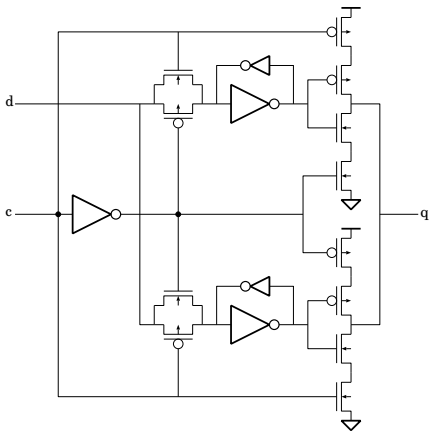

Figure 20: Implementation of the Asynchronous D-flip-flop

7 Concluding remarks

The SCPP-A design problem is to design the stages of a simplified Sproull Counterflow Pipeline Processor, through which results and instructions should flow in opposite direction as fast as possible. Each instruction–result pair must, however, meet in some stage and then interact with the corresponding processor.

This paper discusses the design of asynchronous implementations for the SCPP-A, starting from the formulation of the communication behaviors of the high-level components down to the implementation of the basic components. Throughout the design, the main objective has been to achieve as fast an implementation as possible. For this reason, and since the design involves mainly control circuitry, we have from the start chosen to use two phase signaling. We concentrated on the explanation of the asynchronous design issues and the intuition of the implementation rather than on formal techniques. During the whole design path, reflections are given on alternative approaches for the specification of the problem (the high-level communication behaviors) as well as on alternative implementations at the lower levels. This led in particular to two distinct implementations which are discussed and compared. Both of them are asynchronous, but not strictly delay-insensitive since the relative delays in the networks have been taken into account to derive faster implementations in terms of networks of basic components and to use the fastest implementations of these basic components that are allowed by the relative delays.

The high-level design given by the communication behaviors of the cop and the stage components allows as much freedom as possible in the sense that instructions and results can flow independently through the pipeline with the sole restrictions that they are not allowed to overtake each other or to swap stages. As a result, this specification allows a stage to be involved simultaneously in a communication with the stage above and in a communication with the stage below. This aspect is particularly exploited by the second design given (Fig. 12), which allows more concurrency within a stage than the first design (Fig. 9), as explained in Section 4. Due to this feature and to a smaller input-to-output distance, the second design is clearly faster than the first one (cf. Sections 4 and 5).

References

[1] Daniel W. Dobberpuhl et al. A 200-MHz 64-bit Dual-issue CMOS Microprocessor. *Digital Technical Journal*, Vol 4, No 4, 1992

[2] J.C. Ebergen. Translating circuits into delay-insensitive circuits. *CWI Tract*, Vol 56, Centre for Mathematics and Computer Science, Amsterdam, 1989

[3] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, J.V. Woods A Micropipelined ARM. *Proceedings of VLSI '93*, Grenoble, France, 1993

[4] C.G. Huang, Oxford Univ. Private communication, 1994

[5] C.R. Jesshope, I.M. Nedelchev, C.G. Huang. Compilation of process algebra expressions into delay-insensitive circuits. *IEE Proceedings-E*, Vol. 140, No. 5, pp 261–268, 1993

[6] Charles E. Molnar and Huub Schols. *The design problem SCPP-A.* Unpublished manuscript, April 1995

[7] Ad Peeters and Kees van Berkel. Single-Rail Handshaking Circuits. *Second Working Conference on Asynchronous Design Methodologies*, South Bank University, London, 1995

[8] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. *Counterflow Pipeline Processor Architecture*. Technical Report SMLI TR-94-25, April 1994

[9] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. The Counterflow Pipeline Processor Architecture. *IEEE Design & Test of Computers*, pp 48–59, Vol 11, Nr 3, Fall 1994

[10] Ivan E. Sutherland. Micropipelines. *Communications of the ACM* 32, Vol 6, pp 720–738, 1989

[11] Tom Verhoeff. Delay-insensitive codes – an overview. *Distributed Computing*, Vol 3, pp 1–8, 1988

6 Implementation of the basic components

This section gives fast implementations of the basic components that are used in our design of SCPP-A and gives an implementation of the buffer given in Figure 3 when using bundled data.

The Blocking Select element (Bsel)
The Bsel element is used in the second implementation of the stage component (Fig. 12), and in the implementation of the Sequencer (below). It is a generalisation of the common Select element (Bsel with complementary controls is a Select); its behavior is explained in Section 4. A fast implementation is given in Figure 14, containing Merge elements (which can be implemented as XORs) and Latches (which are transparent if the control is high, and locking and storing if the control is low; a fast implementation of a Latch is given in [1]).

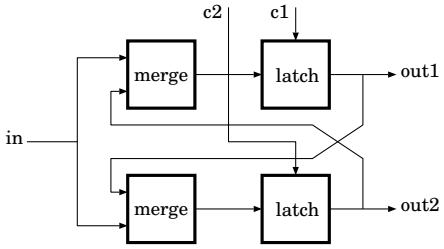

Figure 14: Implementation of Bsel

The Generalised Decision Wait (GDW)
In general, the GDW has n column inputs and m row inputs. The outputs in a column are either placed in a rectangle (denoting nondeterministic choice between the row inputs), or are encircled (denoting deterministic choice). Signals on columns are mutually exclusive, signals on rows not. If a column and a row input match, the corresponding output signal is given (deterministic or nondeterministic depending on the type of the column). In [5], a number of implementations of the GDW are discussed. For completeness, the implementation of the GDW used in Fig. 10 is given in Fig. 15 (using the fast implementation from [5]).

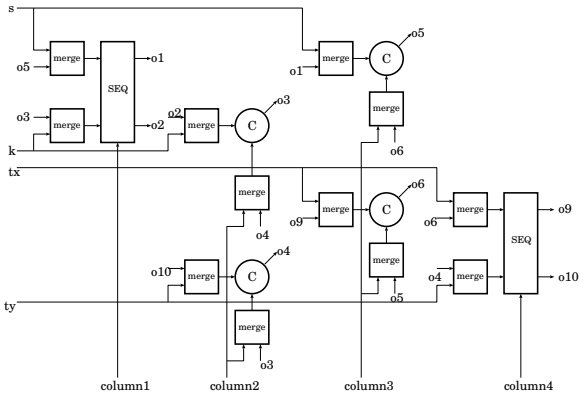

Figure 15: Implementation of the GDW cell used in Figure 10

The Call element
The specification of the Call element is given in Section 4. Its implementation is given in Figure 16.

The Sequencer (Seq)
The communication behavior of the Sequencer is given in Section 3

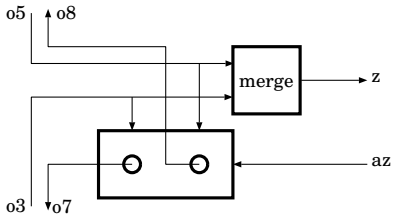

Figure 16: Implementation of Call

(in (5)). In [2, 5], several safe (no timing constraints on the environment) implementations of this important element are given. Figure 17 gives a faster implementation of Seq, which, however, does impose timing constraints on the environment. Signals *in1* and *in2* lead to an arbitration by the MutEx, which causes one of its outputs to become high. After receipt of *en*, the corresponding output signal is given. Since each input and its corresponding output alternate, this output signal will cause the output of the corresponding Merge, and possibly also the corresponding output of the MutEx, to become low. The restriction on the environment is that the next enable signal *en* is not provided until these internal symbols have become low. In our design, the Seq is used in the implementations of Cop and GDW, where this restriction is met.

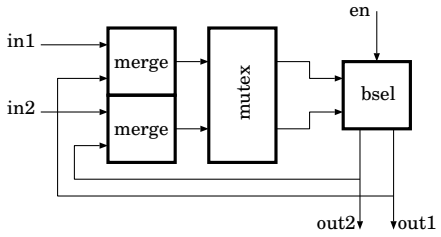

Figure 17: Implementation of Seq

The Buffer element based on bundled date encoding
In bundled data encoding each data channel is bundled with a control wire, which signals after the communicated data is complete [10]. The buffer is depicted in Figure 18; it contains two write channels (*w1/w2*) and corresponding acknowledgements (*aw1/aw2*) that indicate completion of the storage. Since we use single write-port memory elements, the incoming data and write channels need to be multiplexed. Figure 18 gives an implementation of the buffer based on bundled data usage. The asynchronous multiplexor Mux serializes the two bundled data input channels onto one. Its implementation is shown in Figure 19. The memory element *areg* consists of a set of asynchronous D-flip-flops. An implementation of the D-flip-flop (from [4]) is given in Figure 20. At each occurrence of control signal c of an asynchronous D-flip-flop, the output q becomes equal to the input d (and remains that value until the next signal c is received even if d changes). The $\mathrm{DW}_{2\times1}$ element in Fig. 18

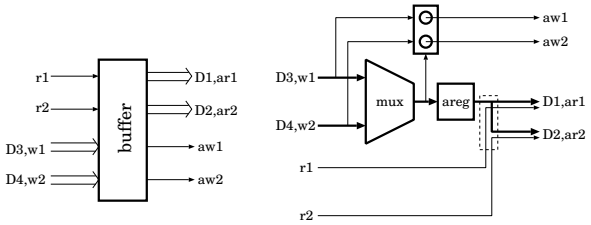

Figure 18: Buffer (left) and its bundled data implementation

An input event s? results (via Merge i and MutEx) in *c1*,*c2*=1, 0, leading to state P_{01} and via the top-most Bsel to output o1!. A subsequent input k? causes both inputs of MutEx to become high (without changing *c1*,*c2*), and results via the bottom-most Bsel to output o3!. This leads to state P_{11}. Notice that if s? and k? arrive simultaneously, the MutEx chooses either state P_{01} or P_{10}. In state P_{11}, only signals tx and/or ty can arrive, which then cause the outputs of Merges i and/or ii to become low, leading via MutEx to a state change. Notice that if tx and ty arrive simultaneously in P_{11} (or, e.g. ty in P_{01}), this leads to the initial state (P_{00}).

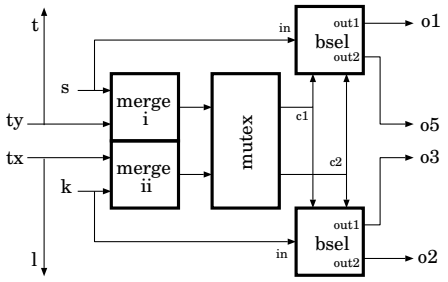

Figure 11: Implementation of (7)

The complete implementation of the stage component (using Figures 10, 11), is depicted in Fig. 12. It uses fewer basic elements and is faster than the implementation with GDW given in Fig. 9. The speed gain is due to a faster input-to-output distance and a faster state change allowing more concurrency. To illustrate the former, consider the arrival of s? in state P_{00}. In the second implementation, the delay for *aty* is 1 MutEx + 2 Merge + 1 Bsel, while the delay in the first implementation is 1 Seq + 2 Merge, which equals (cf. Section 6) 1 MutEx + 3 Merge + 1 Bsel. In both cases the delay of the Bsel equals the delay of 1 Latch (cf. Section 6; due to concurrency). To illustrate the second cause of speed gain, consider the simultaneous arrival of s? and k? in state P_{00}. In the second implementation, the MutEx chooses P_{10} or P_{01} after which both outputs (aty,z or atx,z) are produces concurrently both with a total delay of 1 MutEx + 2 Merge + 1 Bsel (cf. above and Section 6 for Call). In contrast to this, the first implementation serializes the acceptance of s? and k?; although the first output (aty or atx) is produced with delay as above, output z is produced with delay 1 Seq + 2 Merge (for the state change) + 1 Cel + 2 Merge, which is (cf. Section 6) 1 MutEx + 5 Merge + 1 Bsel + 1 Cel (considerably more than in the second implementation).

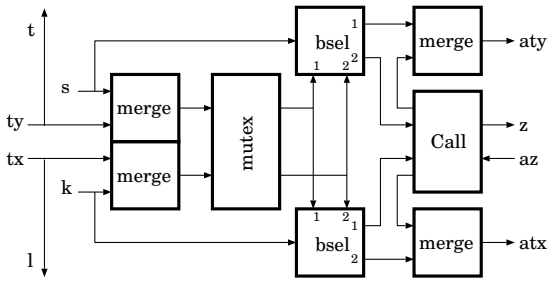

Figure 12: Implementation of the stage component

5 The complete structure

The SCPP-A design contains N stage components (numbered from 1 to N) and $N+1$ cop components (numbered from 0 to N). The middle cops ($1 \le i < N$) are placed between stages as depicted in Fig. 13, with cop and stage components as discussed in the previous sections and X/Y-buffers as discussed in Section 2 (cf. Fig. 3, 4). Components cop(0) and cop(N) are placed at the bottom and the top of the pipeline. Due to symmetry, it suffices to discuss cop(N) and the requirements on the top environment only.

The top environment $\top$ receives instructions and sends results independently. It has an input channel $K_\top$ to receive instructions, an output wire $k_\top$ to acknowledge their receipt, an input wire $ty_\top$ requesting results, and an output channel $L_\top$ to send them. Cop(N–1) can be implemented as in Fig. 8 with $k_{N+1} = k_\top$. The implementation of cop(N) depends on the way $\top$ deals with the receipt of instructions and the sending of results. If $\top$ can accept instructions at any rate, it must satisfy $(K_\top?\,; k_\top!)^*$ and the C-element in cop(N) that produces *rtx* (Fig. 8) can be replaced by a wire connecting atx_N and *rtx*. If $\top$ can*not* accept instructions at any rate, it requires an output wire $wx_\top$ indicating willingness to accept an instruction. Depending on whether $\top$ is initially willing to accept an instruction, it must satisfy either $(K_\top?\,; k_\top!\,; wx_\top!)^*$ or $(wx_\top!\,; K_\top?\,; k_\top!)^*$, in which cases the input $k_{N+2} = wx_\top$ of the C-element for *rtx* in cop(N) is inverted (in the first case, as in Fig. 8) or not.

In a similar way, the willingness to send results influences the C-element for *rty* of cop(N) (Fig. 8) and may lead to an additional output wire for $\top$ indicating readyness to send a result.

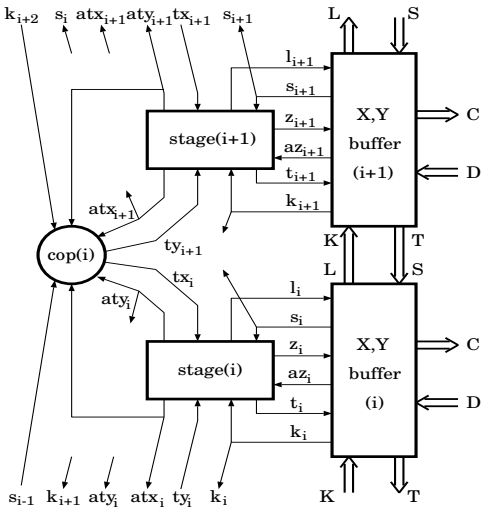

Figure 13: The global structure of SCPP-A

Remark on the performance

The time it takes an instruction to flow through an empty pipeline is per stage D(*k*,*atx*) + D(*atx*,*tx*) + D(*tx*,*l*) + D(l_i,k_{i+1}), where D(a,b) denotes the delay from a to b. This is 2 MutEx + 3 Merge + 2 Bsel + 1 Cel + D(l_i, k_{i+1}) for the second stage-implementation in Section 4 and 1 Merge more for the first one (cf. Sect. 3,4). A similar difference occurs if the result buffers are full. It should, however, be noted that the second implementation allows more internal concurrency and is therefore faster if signals arrive simultaneously (cf. Section 4). This means that in normal conditions (many full buffers) the second implementation is much faster than the first one.

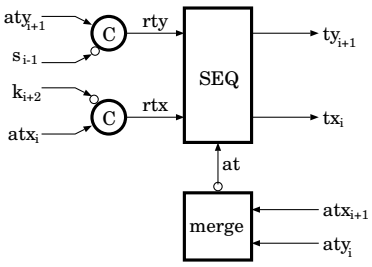

Figure 8: Implementation of Cop(i)

4 Implementation of the Stage component

This section presents two asynchronous implementations of the stage component. First, specification (2) is translated into a large Generalised Decision-Wait component (GDW, [5]) and a number of Merges. This GDW is then transformed into the second implementation, which is more tricky but also faster.

By introducing two auxiliary states, we can rewrite specification (2) such that each choice option involves only one output signal:

(6) Communication behavior of a stage component, where P_{00} is the starting state

$$
\begin{aligned}
P_{00} &= (k?; atx!, P_{10}) \,\square\, (s?; aty!, P_{01}) \\
P_{01} &= (k?; z!, Q_x) \mid (ty?; t!, P_{00}) \\
Q_x &= az?; atx!, P_{11} \\
P_{10} &= (s?; z!, Q_y) \mid (tx?; l!, P_{00}) \\
Q_y &= az?; aty!, P_{11} \\
P_{11} &= (tx?; l!, P_{01}) \,\square\, (ty?; t!, P_{10})
\end{aligned}
$$

Specification (6) can be translated directly into a large General Decision-Wait component (GDW) and a number of Merges by using the different states in (6) as the states (columns) in the GDW. The GDW is given in Figure 9, where the rectangles in the columns denote nondeterministic choice and the circles denote deterministic choice. The output signals and the state-changes can be implemented by Merges, which can be derived easily from (6). In, for instance, state P_{00}, input k must lead to output *atx* and change of state to P_{10}; this is achieved by connecting output o2 to Merge 7 (for *atx*!) and to Merge 3 (for P_{10}). This procedure also leads to Merges 8 and 9 producing signals l and t. Since the inputs of these Merges are outputs in the same row of the GDW, the Merges can be avoided by producing l and t at the input of the GDW (the dashed lines in Fig. 9). This however is only safe under the (reasonable) condition that the state change (induced by o9/o10 and Merge 2/3 and by o4/o6 and Merge 1) is faster than the loop via data transfer and cop leading to a possible next input.

The fast implementation of a GDW is discussed in [5], its speed is 1 C-element + 1 Merge for the circle outputs, and 1 Sequencer + 1 Merge for the rectangle outputs.

Second implementation of stage

To achieve a faster implementation of the stage component, we will transform the implementation given above (with the direct outputs l and t instead of Merges 8 and 9). First, notice that the fifth row of the GDW and Merge 5 (Fig. 9) can be implemented by the (faster) Call component [10] specified by:

Call = (o3!; z!; az?; o7!, Call) | (o5!; z!; az?; o8!, Call)

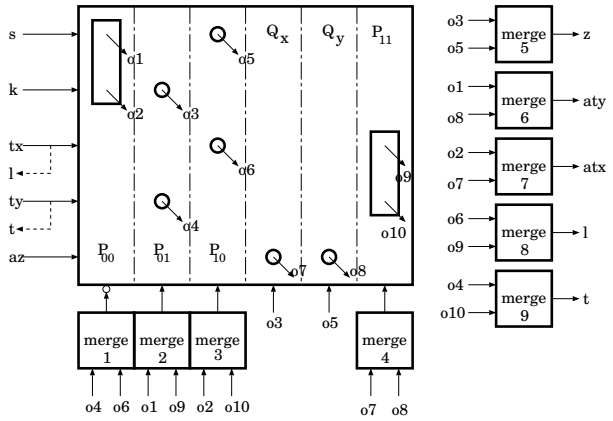

Figure 9: Implementation of Stage(i) by a GDW

Signals o3/o5 request processor interaction via z (procedure call); acknowledgement az is passed to the corresponding output (o7/o8).

This leads to the decomposition of the large GDW into a smaller GDW and a Call as depicted in Figure 10. Since signals tx/ty will not arrive before the processor interaction has been completed (and atx/aty is sent, cf. cop), Merge 4 can be replaced by a Merge with inputs o3 and o5. The communication behavior of the thus derived GDW with Merges 1–4 is given in Expression (7).

(7)
$$
\begin{aligned}
P_{00} &= (k?; o2!, P_{10}) \,\square\, (s?; o1!, P_{01}) \\
P_{01} &= (k?; o3!, P_{11}) \mid (ty?; l!, P_{00}) \\
P_{10} &= (s?; o5!, P_{11}) \mid (tx?; t!, P_{00}) \\
P_{11} &= (tx?; t!, P_{01}) \,\square\, (ty?; l!, P_{10})
\end{aligned}
$$

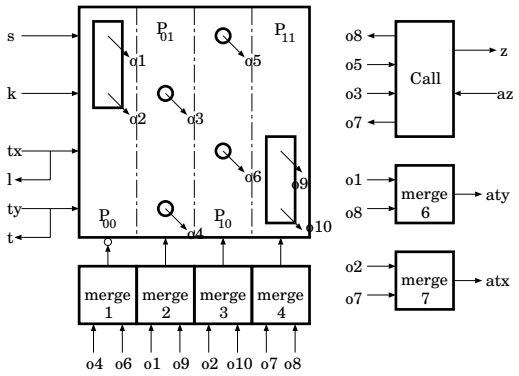

Figure 10: Implementation of Stage with GDW and Call

The significant nondeterministic choice in Expression (7) is the one in P_{00}, since it determines the output following k? or s?. In the implementation in Figure 11, these outputs are blocked by the Blocking-Select (Bsel) components until the choice for either state P_{01} or P_{10} has been made by the Merges and the MutEx.

A Bsel has two control inputs (*c1, c2*), one event input (*in*), and two event outputs (*out1, out2*). Depending on the states of *c1* and *c2*, an incoming event *in* is either passed to *out1* (if *c1,c2*$=1,0$), or passed to *out2* (if *c1,c2*$=0,1$), or it is blocked until *c1*$\neq$*c2* (if *c1,c2*=0,0). Input state *c1,c2*$=1,1$ is not allowed.

Since signals s and ty alternate (cf. (7)), the output of Merge i becomes high by s? and low by ty? (similar for k, tx, and Merge ii). In the initial state P_{00} the inputs and outputs of the MutEx are low.

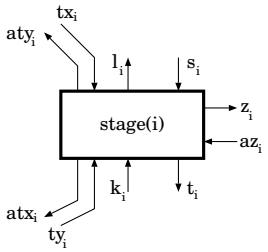

Figure 6: Schematic of Stage(i)

cation behavior of cop(i) satisfies (1). This is expressed as the first line (the first condition) in (3). The double brackets in Expression (3) mean prefix closure and repetition, that is $[\![E]\!] = \text{pref}(E^*)$. The infix operator '$\|$' denotes parallelism with synchronisation on the common signals (weaving).

Further, tx should be sent only when both stages are ready and able to transfer X. For stage($i+1$) this means that buffer X must be empty, which is the case initially and after each transfer of X to stage($i+1$) has been followed by a transfer of X to stage($i+2$). This means that cop(i) must satisfy $(tx_i!\,; k_{i+2}?)^*$, where signal k_{i+2} denotes the completion of the transfer of X to stage($i+2$). Stage(i) is ready and able to transfer X if it is not interacting with processor(i) and if buffer X is full. Notice that the completion of any processor interaction induced by a transfer of Y is already synchronised with tx by the first line of (3) and the choice of aty_i. Consequently, the resulting requirement is that buffer X is full and any processor interaction induced by the last receipt of X from below has been completed. This is concisely expressed by the condition $(atx_i?\,; tx_i!)^*$. The combination of these two conditions on tx_i is expressed in the second line of Expression (3). A symmetrical argument for ty_{i+1} leads to the third line of Expression (3).

Expression (3) gives the communication behavior of cop(i). A schematic of cop(i) is given in Figure 7.

(3) Communication behavior of the cop(i) component

$$[\![(tx_i!\,; atx_{i+1}?) \mid (ty_{i+1}!\,; aty_i?)]\!]$$
$$\| \; [\![atx_i?\,; tx_i!\,; (atx_i?\,, k_{i+2}?\,; tx_i!)^*]\!]$$
$$\| \; [\![aty_{i+1}?\,; ty_{i+1}!\,; (aty_{i+1}?\,, s_{i-1}?\,; ty_{i+1}!)^*]\!]$$

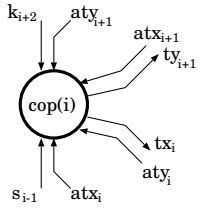

Figure 7: Schematic of Cop(i)

Remarks on the high-level specification

In the specification of SCPP-A as given by (2) and (3), only local arguments are used to decide which data transfer is granted. In view of avoiding traffic jams (strings of full buffers), using global buffer information might seem worthwhile to steer the arbitration. However, the additional complexity of the cop component would, most likely, overshadow the gain. Besides that, thought difficult to estimate, we have the impression that the alternation of transferring X and transferring Y between stages that takes place if the pipeline is relatively full (cf. (3)), helps to avoid these local traffic jams.

In the specification of SCPP-A given above, several interesting choices have been made. First, the choice of k_{i+2} in cop(i) to denote the completion of the transfer of X to stage($i+2$) is necessary: using l_{i+1} can cause clashing of communications (viz. the next X sent to stage($i+1$) can overwrite the previous X); using atx_{i+2} causes unnecessary waiting on the completion of interaction with processor($i+2$), an effect that propagates globally.

The second choice that deserves further attention is the choice to send atx/aty (in stage) only after completion of processor interaction (after $az?$ in P_{01}/P_{10}). This choice enables a small and fast implementation of the cop component (Section 3). Choosing for the atx/aty interpretation of purely 'completion of the data transfer' would lead to a more complex (and slower) cop, unless also the tx/ty interpretation is changed into purely 'full–empty (sender/receiver buffer)', in which case the stage component is more complex (viz. since tx/ty needs to be synchronised with az).

The final remark on the specification relates our cop to the one suggested in [8]. Thought the basic idea of these cops is equal, there is an essential difference in the choice of input signals. The suggested input signals for cop in [8, section 8.4] indicate 'willingness to accept an instruction', 'willingness to pass an instruction forward', and similar ones for results. However, a stage that is willing to pass an instruction forward because the result buffer is empty, will (due to the desired processor interaction) have to withdraw this willingness after receipt of a result. Consequently, the stage component (which is not implemented in [8]) cannot be implemented using this interpretation without the possible withdrawal of this signal, which is difficult since two phase signaling is aimed at and which would lead to complex delay considerations in the cop component.

3 Implementation of the Cop component

This section presents an asynchronous implementation of cop(i) in terms of basic elements. The optimal (in terms of speed) implementation of the basic elements that are used is given in Section 6.

In the specified communication behavior of cop(i) as given by Expression (3), the bottom two components are Muller C-elements with one inverted input. In order to avoid common signals in the components, that is, to make the components independent, we introduce internal signals *rtx* and *rty* as the outputs of the C-elements, as expressed in (4) below. The top-most component in Expression (4) clearly involves arbitration. Its implementation becomes clear by the (delay-insensitive) decomposition in Expression (5), where the resulting two components are a Merge with inverted output and a Sequencer respectively. Consequently, the cop(i) component can be implemented as depicted in Figure 8.

(4) Implementation of the cop(i) component

$$[\![(rtx?\,; tx_i!\,; atx_{i+1}?) \,\square\, (rty?\,; ty_{i+1}!\,; aty_i?)]\!]$$
$$, \; [\![atx_i?\,; rtx!\,; (atx_i?\,, k_{i+2}?\,; rtx!)^*]\!]$$
$$, \; [\![aty_{i+1}?\,; rty!\,; (aty_{i+1}?\,, s_{i-1}?\,; rty!)^*]\!]$$

(5) Decomposition of part of cop(i)

$$[\![(rtx?\,; tx_i!\,; atx_{i+1}?) \,\square\, (rty?\,; ty_{i+1}!\,; aty_i?)]\!]$$
$$\rightarrow \; [\![at!\,; ((atx_{i+1}?\,; at!) \mid (aty_i?\,; at!))^*]\!]$$
$$, \; ([\![rtx?\,; tx_i!]\!] \,\|\, [\![rty?\,; ty_{i+1}!]\!] \,\|\, [\![at?\,; (tx_i! \mid ty_{i+1}!)]\!])$$

2 A first step towards an implementation

The first step we take towards an implementation is to abstract from the data communication. We therefore assume that the local buffers for instructions and the local buffers for results are formed as depicted in Fig. 3. The incoming signals are mutually exclusive. Here, signal l requests to send (initializes sending) the data that is stored in buffer X via channel L (to the stage above); signal k reports receipt (storage) of a new instruction arrived via channel K (from the stage below). Similar for the other signals. The communication via channels C and D are split into an X part and a Y part. Signal *z* is forked to both buffers. The acknowledgements of receipt *azx* and *azy* are combined via a Muller C-element to signal *az*, which reports the completion of the interaction with the processor. These input and output signals of stage(i) —abstracted from the buffers— are depicted in Fig. 4. If necessary to avoid ambiguity, the signals of stage(i) are given a subscript i.

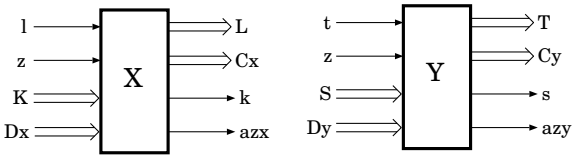

Figure 3: Schematic for buffers X (left: 3a) and Y (right: 3b)

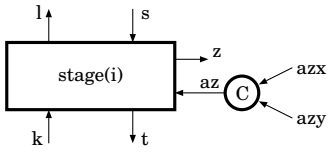

Figure 4: Stage(i) and (incomplete) its external channels

The schematics given in Fig. 3 are general. The communication can be implemented by delay-insensitive codes like double rail codes [3, 7] or DI-codes [11], or by bundled data [10]. As an example of such an implementation, the bundled data implementation of Fig. 3a is discussed in Section 6. The choice between delay-insensitive signaling and bundled data coding depends on the size of the data and on the number and sizes of the stages and processors. In general, the bundled data communication scheme involves less area and is faster [3, 7].

Basic idea

As mentioned earlier, instructions and results are not allowed to swap stages –that is, the transfer of an instruction from stage(i) to stage($i+1$) and the transfer of a result from stage($i+1$) to stage(i) should be prohibited to overlap. In order to control the traffic between stages i and $i+1$, the element cop(i) is introduced (similar to the one in [8]). The basic idea is that cop(i) sends signals tx and ty to stage(i) and stage($i+1$) respectively to initialise the data transfer of X and Y respectively; cf. Fig. 5. Of course, signals tx and ty must be sent such that the corresponding data transfers do not overlap, and only when the sender-stage is ready and able to send and the receiver-stage is ready and able to receive. In the sequel of this section we discuss which input signals are required for cop(i) to be able to do this and what the communication behaviors of the stage and cop components must be. Further, we discuss alternatives, explain the difference between our cop and the one mentioned in [8], and explain why theirs is not feasible.

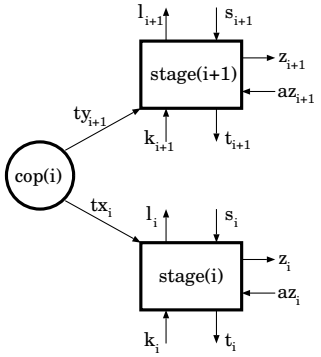

Figure 5: Initial idea of the Cop(i) element

Communication behaviors

In order to avoid overlap of the data transfers of X and Y between stages i and $i+1$, cop(i) requires an acknowledgement that the data transfer is completed. In view of the implementation of the buffers (cf. above), this acknowledgement can be sent by the receiving stage —that is, stage($i+1$) acknowledges completion of the transfer of X (by signal atx_{i+1}), and stage(i) acknowledges completion of the transfer of Y (by signal aty_i). For cop(i) this means that the next tx or ty should be sent only after the previous one is acknowledged, that is, its communication behavior must –among others– satisfy:

(1) $\quad ((tx_i!\,;atx_{i+1}?)^*;(ty_{i+1}!\,;aty_i?)^*)^*$

(recall that all subscripts refer to the stage number).

Further, a request for data transfer (tx_i or ty_{i+1}) should be sent (by cop(i)) only if both stages are ready and able for this transfer. Notice that if a data transfer induces interaction with the processor (at receiver side), the stage is not ready for the next data transfer until this interaction is completed, and hence, cop(i) must wait with sending the next request. We therefore choose, in case of such an interaction, to send the acknowledgement atx_{i+1} or aty_i after this interaction is completed. Notice that since cop(i) must wait for this completion, this choice does not induce any loss of efficiency (speed) at this abstraction level. We return to this choice later on.

The communication behavior of a stage can now be expressed as (2) below, where P_{00} is the starting state with both buffers empty; in P_{01} buffer X is empty and Y is full; in P_{10} X is full and Y is empty; and in P_{11} both buffers are full. The infix operators ‘ , ’, ‘ ; ’, ‘ | ’, and ‘□’ denote parallel execution (interleaving), sequential execution, deterministic choice, and nondeterministic choice respectively. Notice that atx acknowledges completion of data transfer of X (by $k?\,;atx!$), and, in case an interaction with the processor is induced, also acknowledges completion of this interaction (in P_{01} by ‘$az?\,;atx!$’). A schematic of stage(i) is given in Figure 6.

(2) Communication behavior of a stage component, where P_{00} is the starting state

$$
\begin{aligned}
P_{00} &= (k?\,;atx!\,,P_{10}) \;\square\; (s?\,;aty!\,,P_{01})\\
P_{01} &= (k?\,;z!\,;az?\,;atx!\,,P_{11}) \;|\; (ty?\,;t!\,,P_{00})\\
P_{10} &= (s?\,;z!\,;az?\,;aty!\,,P_{11}) \;|\; (tx?\,;l!\,,P_{00})\\
P_{11} &= (tx?\,;l!\,,P_{01}) \;\square\; (ty?\,;t!\,,P_{10})
\end{aligned}
$$

Component cop(i) must send signals tx and ty such that the transfers of data X and Y between stages i and $i+1$ do not overlap, and only when both stages are ready and able for the transfer in question. As argued above, overlap is avoided if the communi-

An asynchronous implementation of SCPP-A*

W.H.F.J. Körver and I.M. Nedelchev

Computer Systems Research Group
Department of Electronic & Electrical Engineering
University of Surrey, Guildford, Surrey GU2 5XH, UK
{W.Korver , I.Nedelchev} @ee.surrey.ac.uk

Abstract

This paper presents an asynchronous implementation of the SCPP-A design problem posed by C.E. Molnar and H. Schols. The problem is a simplification of part of the Sproull Counterflow Pipeline Processor. Although this problem can be specified concisely, it raises design issues that are intrinsically difficult to deal with. We discuss an implementation of SCPP-A where the speed of the resulting circuitry is considered to be the main design criterion. Further, we discuss a number of aspects of the problem and the design, including alternatives, which altogether, we believe, offers a good example of asynchronous design concerns.

1 SCPP-A: The problem description

This section gives an informal specification of SCPP-A. The design problem and specification are given in [6]. The SCPP-A is related to the Sproull Counterflow Pipeline Processor described in [8, 9].

The SCPP-A has N stages, $N \geq 5$, and N processors; both stages and processors are numbered from 1 to N. Each stage contains a local buffer for *instructions*, called X, and a local buffer for *results*, called Y. Instructions should flow in their original order upward in SCPP-A as fast as possible, using the K and L channels depicted in Fig. 1. Similarly, results should flow in their original order downward as fast as possible, using the S and T channels in Fig. 1. However, each instruction X and each result Y must meet and interact in some stage. This interaction means that both values, $\langle X, Y \rangle$, are send to the corresponding processor (via channel C) and are overwritten by the new instruction–result pair $\langle X', Y' \rangle$ produced by the processor and received via channel D.

The problem posed in [6] is to design all stages, preferably with identical shape. Initially all buffers are assumed empty. Since each instruction and each result must meet, they are not allowed to pass each other between stages (to swap stages), and, since their flow direction is opposite, they must interact at the stage where they meet. As a result, the specification of a particular stage can be expressed as the state graph depicted in Figure 2.

In the following sections we present an implementation for the stages. We do not bind ourselves to a particular formalism, but concentrate on the intuition of the design. Initially, we consider the stages in the middle of the pipeline; in Section 5 we deal with the top and bottom effects. The design is given first in terms of networks of basic components. In Section 6 we discuss implementations of each basic component that is used, in particular the implementation that is the fastest in this design in view of the relative delays in the network. The latter implies that our design, though it is asynchronous, is not strictly delay insensitive. Since our design concentrates on the control circuitry and since the speed of the implementation is our main concern, we use two phase signaling in all implementations, which means that each transition corresponds to a logical event.

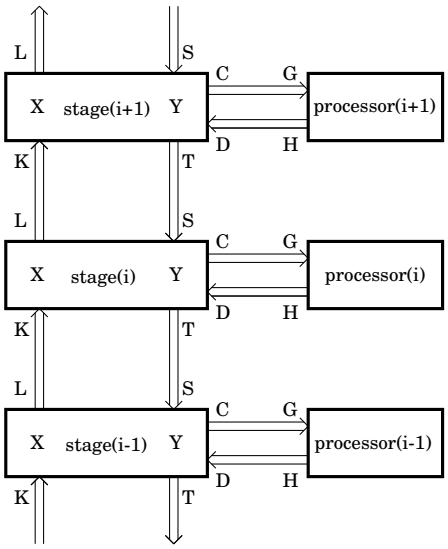

Figure 1: Part of an SCPP-A

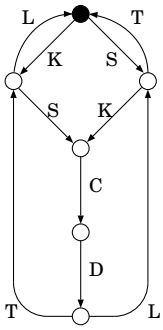

Figure 2: State graph of the communication behavior of a stage

*this research was supported by the EC under the 'Eurochip' project and by EPSRC under grant nr. GR/G 18605