

Asynchronous Communication without Waiting: from the Concept to the Hardware

Anthony C. Davies and Ian G. Clark

Department of Electronic Engineering, King's College London,
Strand, London, WC2R 2LS, England,

E-mail: tonydavies@ieee.org, IGClark@iee.org

Abstract - A mechanism for wait-free communications between asynchronous processes is described and an implementation in hardware is shown. The description goes from basic concepts through to the development of a traditional gate and flip-flop level design.

1 Introduction

Technology is enabling implementation of single chips incorporating huge numbers of independent high performance high-speed digital processors. Clock distribution difficulties may lead to less use of conventional fully-synchronous designs and systems known as 'GALS' (globally asynchronous locally synchronous) communicating asynchronously may become popular. In this context, wait-free communications between writing and reading processes is attractive.

If at all times the writer is to be free to write and the reader free to read, with no mutual interference or use of arbiters to control access to shared memory, then at least four 'slots' are needed for holding the data items to be communicated. Several four-slot solutions have been devised and at least one implemented for use in real-time systems [1, 2].

2 Required behaviour of a wait-free ACM

Waiting is inherent in some forms of interprocess communications such as serial channels, but when accessing 'reference data' [3] there is no requirement for the reader to read every item written provided that the 'newest' data is available to the reader. Many writes are allowed between two reads or many reads between two writes. The writer does not have to wait for the reader and therefore may write at any time.

In any fully asynchronous multi-processor system a read and a write can partially or completely overlap in time, which must never result in a simultaneous read from and write to the same location. The ACM has to be able to avoid this by providing sufficient 'slots'.

Three behavioural properties must be satisfied:

(a) *Coherence*

Each data item is typically a multi-bit object (for example an array or record) so the reader must always obtain a coherent item, as opposed to one which is partly 'old' and partly 'new'.

(b) *Sequentiality*

Once a data item has been read, all earlier items are effectively 'out of date'. A '..new, old, new..' sequence is incorrect. If an item has been read, subsequent reads must never access any previously written item.

(c) *Freshness*

A mechanism could comply with the above two properties by repeatedly providing the reader with the first item written. That is obviously unacceptable, so data coherence and sequencing are necessary but not sufficient. The data items must be fresh (e.g. not 'out of date').

Simpson has provided a thorough discussion of all three properties [7, 8]

3 Typical structure of an ACM

The writer and reader may be assumed embedded in endless loops, in which the writer prepares or obtains a new data item and writes to the ACM, and the reader independently reads from the ACM and uses the new data item.

Typical ACM designs comprise slots for holding the data items in memory shared by writer and reader, and control variables to ensure that writing and reading is directed to the correct slot.

4 Algorithmic description of a typical ACM

To illustrate the development of an implementation, a simple ACM design was chosen from a number of alternatives [12], derived from a software-based scheme proposed by Anderson and Gouda [4] for creating an ‘atomic’ [5] data-area. The core has been extracted and simplified to form the basis for a wait-free ACM which is easily implemented in hardware. This ‘Anderson-Gouda’ ACM is similar to one proposed long before by Simpson [6].

There are four slots, arranged conceptually in two rows and two columns, addressed by using array notation:

$$\text{slot}[\text{row}, \text{column}]$$

where *row*, *column* are each binary (0 or 1)

Shared (binary) control variables ensure that the writer can always write to an available slot and the reader always reads from the slot with the most recently available data.

Successive writes with no intervening read use the same row and alternate column. Successive reads with no intervening write use the same column and row as the previous read (and of course get the same data every time). For a strictly alternating write, read sequence, the alternate row and alternate column is used for each successive write and read.

Informally, the process may be described as follows:

Writer Process:

- find which row the reader is/has been looking at and write to the opposite row and write to the opposite column of the previous write
- update shared variables so that the reader can discover where the writing took place

Reader Process:

- find the slot of the last completed write and read from it

The Appendix shows a high-level ‘re-phrasing’ and simplification of the essentials of the algorithm. Upper case denotes a shared variable and lower case denotes a variable local to reader or writer. *Z* is a shared two-element array of type bit, *SLOT* is a two-dimensional four-element array of type ‘data item’ and other variables are all single bits.

A further simplification is shown below, requiring 7 shared variables, so having a storage size (space complexity [4]) of $4b+7$ bits (where *b* is the number of bits required per data item).

Minimal algorithm description:

Writer Process:

- w1:** *wp*, *walt* := **not** *RP*, **not** *walt*
- w2:** *SLOT*[*wp*, *walt*] := input
- w3:** *Z*[*wp*] := *walt*
- w4:** *WP* := *wp*

Reader Process:

- r1:** *RP* := *WP*
- r2:** *ralt* := *Z*[*RP*]
- r3:** output := *SLOT*[*RP*, *ralt*]

Step **w1** of the write cycle is a concurrent operation; the two assignments may overlap or be in either sequential order, and of course in the hardware implementation they can be truly simultaneous.

At each write cycle, *wp* is made opposite to *RP* (to avoid the *row* of the latest reader activity) and *walt* is made opposite to the previous *walt* (to alternate the writing *column*). After the write, *WP* and *Z[wp]* are updated so that the reader can find the location of the most recent completed write. At each read cycle, *RP* is set to *WP* and *ralt* is set to *Z[RP]* (respectively the *row* and *column* of the last completed write).

Thus, *RP* indicates to the writer the *row* of the latest read activity (enabling the writer to avoid a collision and hence to maintain coherence) while *WP*, *Z[•]* inform the reader of the location of the latest completed write (so ensuring freshness). Fig. 1 illustrates the concept.

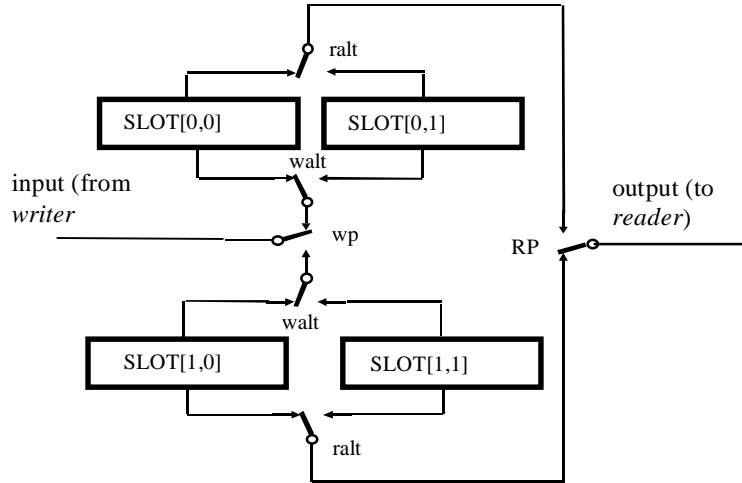


Figure 1: Concept diagram, showing the data flows

5 Hardware implementation

From the algorithmic description a static-logic hardware implementation using D-type flip-flops and NAND gates may be developed directly, as shown in Fig. 2. The two ‘switch’ blocks A and B can be implemented with NAND gates as shown in Fig. 3. Finally, Fig. 4 shows how the control part is used to address the memory forming the slots.

The timing sequence of the instructions corresponds to the waveforms used to clock the flip-flops. Of course, actual instruction time durations depend on the processor and technology used. A substantial delay can be expected between successive writes to a slot (**w2**) and between successive reads from a slot (**r3**). However, because of the asynchronism, nothing at all may be assumed about the relative timing of the reader and writer, so any **r** may overlap to any extent with any **w**. For large data-objects, the duration of **w2** and **r3** may be substantial, increasing the probability that they will sometimes overlap in time.

For simplicity, the initialisation scheme is omitted: it just involves setting all flip-flops to a desired initial state.

Using static logic would be unlikely in a modern high-speed CMOS chip; some form of skew-tolerant multi-phase domino logic would be more likely. However, the implementation principles would be similar.

6 Metastability Risks

In a data transfer between asynchronous processes, it is inevitable that occasionally the set-up or hold times of latches will not be complied with, which may result in a metastable transient [9,10,11]. The small probability of metastability cannot be ruled out in an asynchronous transfer. It is most unlikely in the data transfer path because of the control-variable steering but must be

allowed for in some control variables and not lead to failure of any of the three behavioural properties.

Fig. 2 shows where metastability could arise in the control variables. Three flip-flops (*wp*, *RP*, *ralt*) are vulnerable because they are clocked from one process while their data-input comes from the other process. Since writer and reader timing is independent, no guarantee can be given that the set-up and hold times of these flip-flops will always be complied with. Our preliminary analysis indicates that the required behavioural properties are maintained with metastability of any one variable in the control algorithm provided that the metastable transient has settled to its final value (old or new value) before the value is used. This implies an upper limit to the execution rate of the writer and reader instructions.

7 Conclusions

Many wait-free ACM proposals are difficult to fully understand from the original descriptions. Few consider implementation at the hardware level. The example shows how a high-level description may be transformed into a straightforward implementation. An increased need for such ACMs in real-time systems seems likely because of a trend towards 'GALS' integrated circuits [13].

Acknowledgement

The U.K. EPSRC is thanked for financial support (Grant No. GR/L92471)

References

- [1] Simpson H.R. 'Four-slot fully asynchronous communication mechanism', *IEE Proceedings, Part E, Computers and Digital Techniques*, 1990, 37, 17-30
- [2] Campbell E. 'DIA temporal characteristics and their experimental verification', British Aerospace Defence Dynamics Ltd., Univ. of York, Admiral Management Services, 1992
- [3] Simpson H.R. 'The Mascot Method' *IEE Software Engineering Journal*, 1986, 1, 103-120
- [4] Anderson J.H. and Gouda M.G. 'A Criterion for Atomicity', *Formal Aspects of Computing*, 1992, 4, 273-298
- [5] Lamport L. 'On Interprocess Communication. I. Basic Formalism and II. Algorithms', *Distributed Computing*, 1986, 1, 77-101
- [6] Simpson H. 'Fully Asynchronous Communication' *IEE Colloquium on MASCOT in Real-time Systems*, London, 12 May 1987, 2/1-2/6
- [7] Simpson H.R. 'New algorithms for asynchronous communication', *IEE Proceedings, Part E, Computers and Digital Techniques*, 1997, 144, 227-231
- [8] Simpson H.R. 'Correctness analysis for class of asynchronous communication mechanisms', *IEE Proceedings, Part E, Computers and Digital Techniques*, 1992, 139, 35-49
- [9] Chaney T.J., Molnar C.E. 'Anomalous Behaviour of Synchroniser and Arbiter Circuits', *IEEE Trans*, 1973, C-22, 421-422
- [10] Seitz C.L. 'System Timing' in Mead C. and Conway, L. *Introduction to VLSI Systems*, Addison Wesley, 1980, 218-262
- [11] Kinniment D.J., Yakovlev A. and Gao B. 'Metastable behaviour and system performance' *Proc. 2nd UK Forum on Asynchronous Systems*, Department of Computing Science, University of Newcastle upon Tyne, July 1997
- [12] Clark I.G., Davies A.C. 'A Comparison of some wait-free Communications Mechanisms', *Proc. Wksp on Asynchronous Interfaces: Tools, Techniques and Implementations (AINT'2000)*, 19th-20th July 2000, Delft, Netherlands, 23-29.
- [13] Davies A.C. 'Reasons, Protocols and Mechanisms for Communicating Asynchronously between Digital Processes – A Tutorial Review', to be presented at SCS'01, Iași, Romania, 10-12 July 2001

Appendix Essentials of a (4b+9) ACM algorithm derived from Anderson and Gouda

```

mechanism Anderson_Gouda_four_slot is
  SLOT:array[bit,bit] of data := ((null,null),(null,null));
  Z : array[bit] of bit := (0, 0);
  WP, RP : bit := 0, 0;
  procedure write(item : in data) is
    d, wp, walt : bit := 0, 0, 0;
  begin
    d := RP;
    wp, walt := not d, not walt;
    SLOT[wp, walt] := item;
    Z[wp] := walt;
    WP := wp;
  end write;
  function read return data is
    ralt, rp : bit := 0, 0;
  begin
    rp := WP;
    RP := rp;
    ralt := Z[rp];
    return SLOT[rp, ralt];
  end read;
end Anderson_Gouda_four_slot;

```

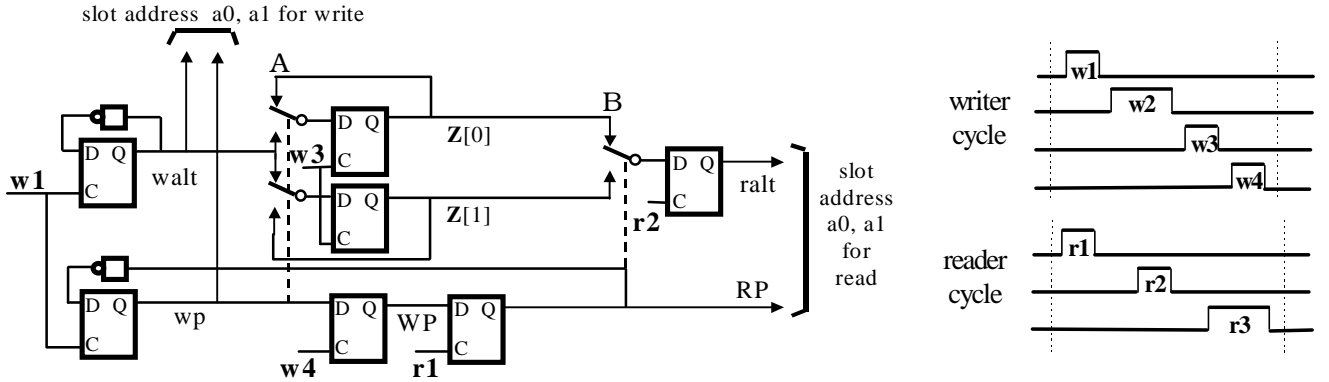


Fig. 2: Hardware implementation

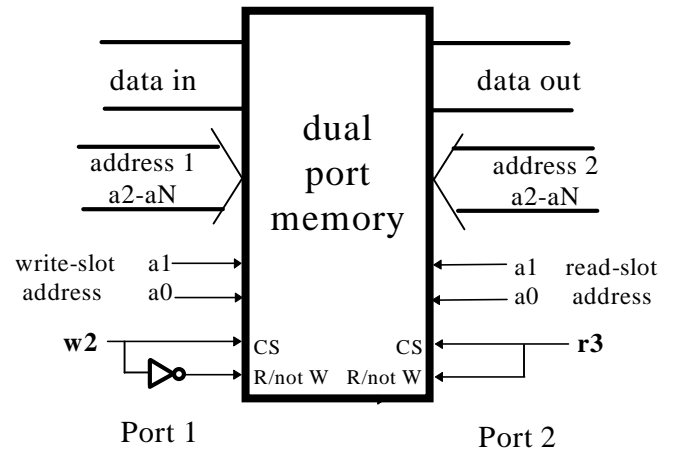
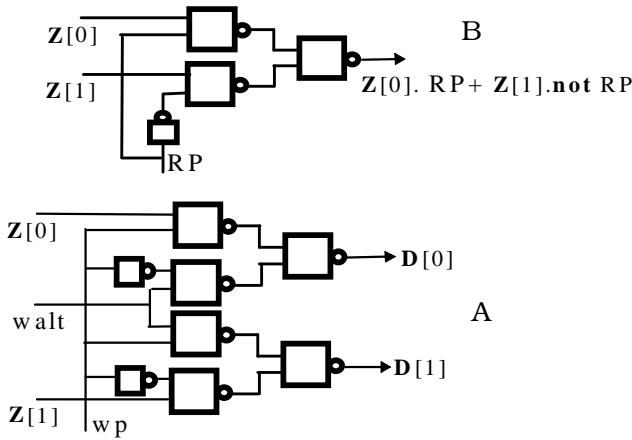


Fig. 3: Detail of 'switches' A, B in Fig. 2 Fig. 4: Memory addressing scheme to access the slots