IMPLEMENTATION OF A THREE-SLOT SIGNAL ACM

F. Hao, A. Yakovlev, E. G. Chester, F. Xia, I. G. Clark*, D. Shang
School of Electrical, Electronic and Computing Engineering, Univ. of Newcastle upon Tyne, UK
*Also with School of Computing and Information Systems, Kinston University, UK

ABSTRACT

As important communication components of asynchronous systems, the ACMs have been studied for many years. A well known Pool using 4 data slots was proposed by H. R. Simpson. However, under certain assumptions, the number of slots in shared memory can be reduced to 3. Mutex, David Cells and SYNCs are used here to implement the 3-slot Signal. The design performed well, maintaining all the required asynchronous properties. It is also a potential building block for the design of low-power heterogeneous systems.

Key words: ACM, Signal, Petri net, David Cell

1. INTRODUCTION

An Asynchronous data Communication Mechanism (ACM) is a scheme, which manages the transfer of data between two processes not necessarily synchronised for the purpose of data transfer. The provider of data is called the "writer", and the user of data is referred to as the "reader". The general scheme of these kinds of data communication mechanisms is shown as follows:

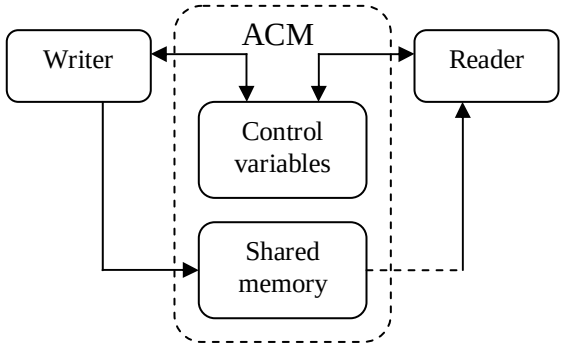

Figure 1 ACM Using Shared Memory and Possibly Control Variables

1.1 Classification of ACMs

ACM protocols can be classified in different ways. According to the number of slots in shared memory, they could be 1-slot, 2-slot, 3-slot and 4-slot mechanisms [1]. Based on whether overwriting and re-reading are permitted, ACMs can also be classified into 4 types [2], as shown below.

TABLE 1 -Classification of ACMs

	NRR	RR
NOW	Channel	Message
OW	Signal	Pool

In table 1, NRR stands for Non Re-Reading and NOW means Non OverWriting. The names of "Channel", "Signal" and "Pool" are retained from Simpson's classification system [3], which is based on the number of items of data in the ACM and its modification by the reader and writer accessing the ACM. "Message" is introduced as the dual of "Signal".

4-slot ACMs have been well studied and designed by many researchers. However, under certain assumptions of atomicity of statements, the slot number in shared memory could be reduced to 3 while still maintaining the main asynchronous properties [4]. This implementation of the 3-slot Signal was based on those studies.

2. ALGORITHM OF THE 3-SLOT SIGNAL

The algorithm of the 3-slot Signal can be found in [5]:

Writer:
wr: write slot w;
w0: l:=w;
w1: w=differ (l, r);

Reader:
r0: wait until (r!=l) r:=l;
rd: read slot r;

Figure 2 Algorithm of 3-Slot Signal

In Figure 2, the statements of *wr* and *rd* are the data accesses, and the others are control variable statements used to determine which slot is to be accessed by reader and writer.

From the definition in table 1, Signal does not allow re-reading, that is, when there are no new data items in the shared memory, the reader will keep waiting until a new one is available. This is the function of the *r0* statement.

For the writer side, the new data is stored in the slot which is neither being read (*r*) nor just written (*l*). This is performed by the 'differ' function in statement *w1*. In the implementation, the differ function is defined according to the following table:

TABLE 2 -Differ Function

l=1	
r=1, 2 (*r*!=3)	*w*=3
r=3	*w*=2
l=2	
r=1	*w*=3
r=2, 3 (*r*!=1)	*w*=1
l=3	
r=1, 3 (*r*!=2)	*w*=2
r=2	*w*=1

The differ function can then be constructed using 3 SYNC arbiters (see 4.2).

3. PETRI NET MODEL

To investigate the algorithm, Petri net models were built and analysed.

Figure 3 shows the Petri net for the writer. The initial state in the figure is *w*=1 and *r*!=1. When a write access signal (write_start) comes, the token moves to the next place from the initial one, which enables the transition *wr1*. After the transition is fired, *l* is set according to *w* (not shown in the Petri net), and *w* is determined by the current value of *r*. The write cycle is finished with a write_done signal sent back to the environment.

The process for the reader is similar to that for the writer. The initial state in Figure 4 is *r*=2. When there is a read_start signal, the reader decides the next step according to the current value of *l*. If *l* is the same as *r*, the reader keeps waiting. Otherwise, *l* is assigned to *r*, such as the transition *r21*. Consequently, the data item in slot *r* is read. With a read_done signal sent out, the read cycle is finished.

4. IMPLEMENTATION

From the Petri net models mentioned, circuits were built. The places in each Petri net were implemented by David Cells, a kind of memory element [6]. The transitions were certain processes between two David Cells.

Because the control variable *l* is shared by both reader and writer, if the statements are non-atomic, it will cause problems in some cases. When a consecutive pair of statements *w0* and *w1* is executed between the start and finish of *r0*, the 3-slot ACM fails both data coherence and data freshness requirements [4]. To solve this problem, a mutual exclusion element (mutex) is added between the reader and writer, which avoid the conflict at *l*.

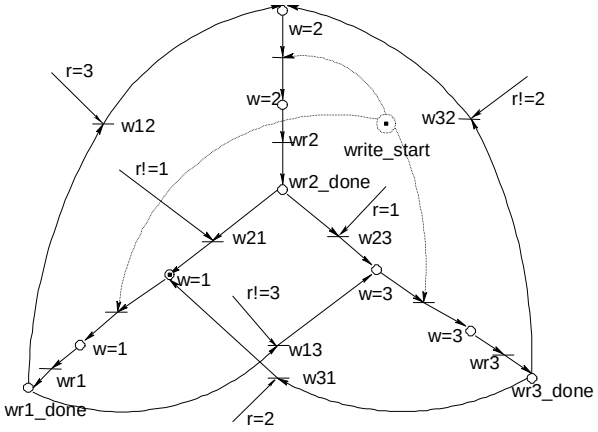

Figure 3 Petri Net for Writer

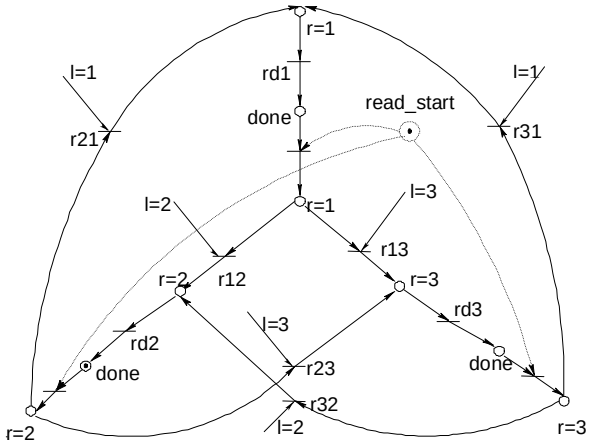

Figure 4 Petri Net for Reader

4.1 David Cell (DC)

A David Cell is built essentially around SR flip-flops, as shown in Figure 5. It represents the marking of the corresponding places in the Petri net. A place holding a token is represented by a DC with state 10 in the flip-flop. State 01 associates with the absence of a token in place.

The control logic of David Cell is shown in Figure 5, [7] gives a detailed explanation of this.

4.2 SYNC Arbiter

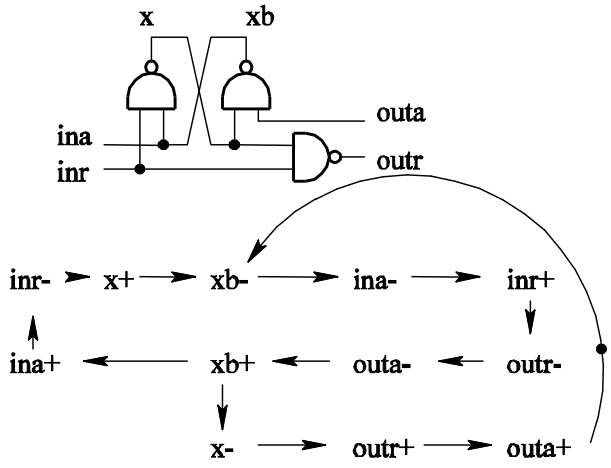

Figure 5 David Cell

A SYNC arbiter, shown in Figure 6, is built from a mutex and an AND gate. When ck0 is low, both of the outputs are low. When ck0 becomes high, the outputs are determined by rbar. If rbar is high, rbar_1 gives a high signal; otherwise, rbar_0 becomes high.

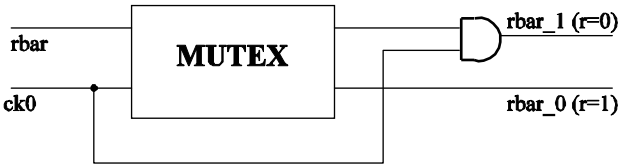

Figure 6 Implementation of SYNC Arbiter

4.3 Implementation of Control Circuits

The control circuits are split into a read side and a write side. Each of them is connected to the Mutex.

The signal flow of the write side is illustrated in Figure 7. When a new data item is available, a write start signal comes to the first DC. The DC sends a permission to shared memory to write the data item into a certain slot. After writing, the Mutex receives a request signal from the second DC. The grant signal from the Mutex passes to the latch, which is used to store the current *l*. The completion signal from the latch is sent to SYNC abiters as ck0 in Figure 6. According to the current value of *r*, the SYNC arbiters determine which slot will be written next. With the third DC sending out a completion signal, the write cycle is completed.

Figure 8 illustrates the signal flow of the read side. When a read access signal comes to the read side, the first DC passes it to the C elements. The circuit keeps waiting until *l*, from the write side, is not equal to *r*. A request is then sent to the Mutex. The grant signal from the Mutex is sent to a latch, which is used to store the current *r*. According to *r*, the data item in a particular slot is read. At the same time, a completion signal 'read done' comes out. The read cycle is thus completed.

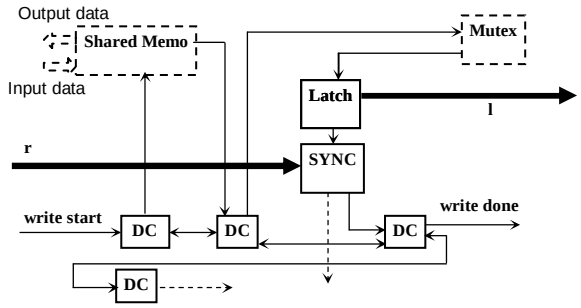

Figure 7 Signal Flow of Write Side

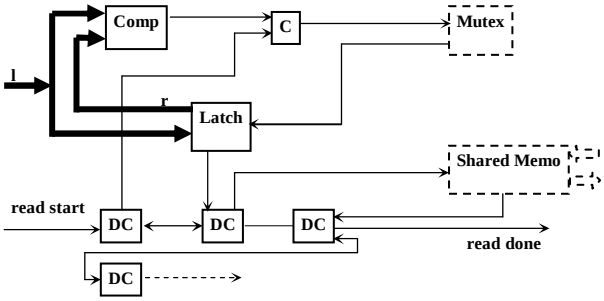

Figure 8 Signal Flow of Read Side

The circuits in these two figures implement only one branch of Figure 3 and 4. The rest two branches can be implemented in the same way. In Figure 7, the first two DCs represent the two places before and after the transition "wr1(2, 3)" The transition of wr1(2, 3) is implemented in the shared memory. The transition following wr1(2, 3)_done place is implemented by the circuits between second and third DCs. Figure 8 can be related to Figure 4 similarly.

4.4 Implementation of the Data Path

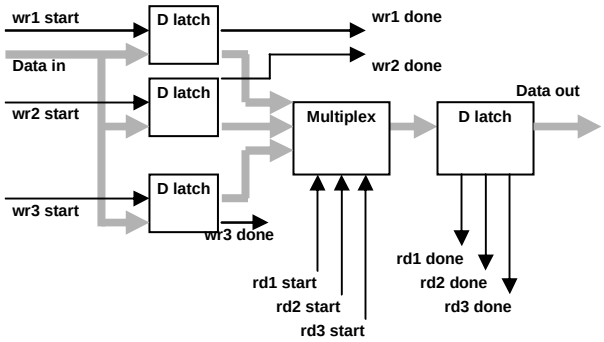

Figure 9 Structure of the Data Path

The data path contains 4 D-latch sets, and one multiplexer, as shown in Figure 9. Each D-latch set is

made from 8 D-latches, which can pass 8 bits in parallel. When a wr_start, from the write side, comes to the data path, the D-latch set stores the data item and passes it to the output, which is connected to one of the inputs of the multiplexer set. Subsequently, a wr_done is delivered to the write side.

The rd_starts are connected to the multiplexer set to control the link from its data inputs to outputs. The D-latch set stores data from the multiplexer set and transfers it to the outputs.

5. SIMULATION RESULTS

Digital simulations of the circuit were carried out with the Cadence™ tool. Figure 10 shows the resulting waveform of one digital simulation.

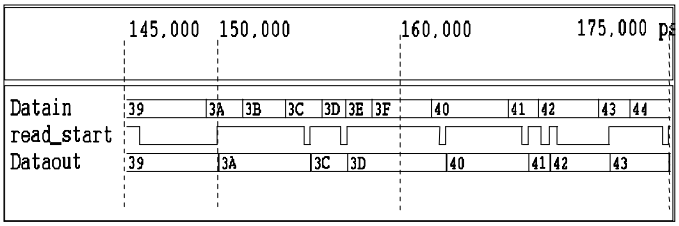

Figure 10 Result Waveforms

In this sequence, after data items 39 and 42 were read by the reader, another read request arrived. The reader did not respond to the requests until new data items were available. During this period, the read_start stayed low. On the other hand, when the writer delivered the data items quickly, such as 3A to 3F, overwriting occurred (3B, 3E and 3F were overwritten). The time taken by the reader and writer outside the ACM was controlled by two independent random number generators written in Verilog. An exponential distribution was assumed and the same mean value was set for both w0 to wr and rd to r0. This gave enough variation for reader waiting and writer overwriting to appear.

6. CONCLUSIONS AND FUTURE WORK

Models and implementations of the 3-slot Signal have been presented introduced. The design maintains the properties of asynchronous systems. Apart from the conventional emphasis on ACMs, this design has the potential of providing freedom for the power-aware designer of heterogeneous systems, by allowing temporal and power decoupling of various kinds between active elements of a system. Since Message is the dual of Signal, the hardware design of a 3-slot Message can be developed in a similar way.

More work needs to be done on the analogue simulation to obtain detailed information of the design, such as circuit sizes, timing properties and effects of metastability. Furthermore, applications of ACMs are to be investigated.

7. ACKNOWLEDGEMENT

This work is part of the Coherent project (http://async.org.uk/coherent) at the Newcastle University supported by the EPSRC grant (GR/R32666).

8. REFERENCES

[1] Simpson, H.R., “Four-slot fully asynchronous communication mechanism”, IEE Proc. Vol. 137, Pt.E, No.1, PP.17-30, January 1990.

[2] Fei Xia, Alex V. Yakovlev, Ian G. Clark, Delong Shang. "Asynchronous communication mechanisms: classification and hardware implementations", MPCS'02, Fourth International Conference on Massively Parallel Computer Systems, sponsored by Euromicro, 10-12 April 2002, Ischia, Italy.

[3] Simpson, H.R., Campbell, E., “Real-time network architecture: principles and practice”, Proc. AINT’2000, Asynchronous Interfaces: Tools, Techniques and Implementations, p.5 and handouts, TU Delft, The Netherlands, July 19-20, 2000.

[4] F. Xia, I. G. Clark. "Studying the three-slot asynchronous communication mechanism", 7th UK Asynchronous Forum, 20-21st December 1999, University of Newcastle upon Tyne.

[5] Fei Xia, Ian Clark. "Algorithms for Signal and Message Asynchronous Communication Mechanisms and their Analysis", Volume 50, Number 2 (2002), pp.205-222, Fundamenta Informaticae, IOS Press

[6] David, R. "Modular Design of Asynchronous Circuits Defined by Graphs", IEEE Trans. on Comp., Vol. 26, No. 8, pages 727-737. August 1977.

[7] A. Yakovlev, F. Xia, D. Shang. "Synthesis and Implementation of A Signal-Type Asynchronous Data Communication Mechanism", Seventh Int. Symp. on Asynchronous Circuits and Systems (Async'2001), March 2001, Salt Lake City, Utah.