Clock Synchronization through Handshake Signalling

Joep Kessels,* Ad Peeters,* Paul Wielage,* and Suk-Jin Kim§

Abstract

We present a method for synchronizing pausible clocks in GALS (Globally Asynchronous, Locally Synchronous) systems. In contrast to most conventional GALS schemes the method is not based on including in each ring oscillator a synchronizing element (such as for instance an arbiter) which on one side can pause the clock and on the other side offers a handshake interface. Instead, we propose a scheme in which each synchronous module has both an incoming and an outgoing clock signal, which have been obtained by opening the module's ring oscillator. Since these clock signals also behave as handshake signals, handshake circuits can be used to synchronize the clocks.

We demonstrate the technique in the context of processors and memories. All the designs have been simulated and showed functionally correct.

Keywords: *GALS systems, pausible clocks, asynchronous crossbar/bus, processor/memory architectures.*

1. Introduction

Two developments undermine the role of globally clocked VLSI circuits. In the first place, the trend towards system-on-chip designs leads to chips containing several memories and IP modules which all have different cycle times. Secondly, in future technologies it will become increasingly difficult to distribute high-speed low-skew clock signals. Therefore, future chips will contain several locally clocked submodules, which communicate through dedicated glue logic. These heterogeneous systems are called GALS (Globally Asynchronous, Locally Synchronous) systems [3]. Two kinds of GALS systems can be distinguished depending on the way the synchronous submodules communicate.

- In a *data synchronization* system, the submodules have free-running clocks and the data being communicated is synchronized from one clock domain to the other. Examples of data synchronizers are the well-known two-register or double-latch synchronizer [12, 4] or more elaborate synchronizing schemes such as pipeline synchronization [16]. Since these synchronizers have to deal with metastable states, the designer has to trade off between safety and low latency. Moreover, special precautions are needed to keep multi-bit words consistent.
- In a *clock synchronization* system, the submodules have so-called pausible clocks (also known as stretchable or stoppable clocks), which are ring oscillators that can be halted. Several approaches have been proposed to stop and restart the clock safely. One approach first samples the inputs in latches and then waits until all (potential) metastability has been resolved before releasing the clock [12, 14]. Recent approaches incorporate a mutual-exclusion element (or arbiter) in the ring oscillator [19, 2, 20, 10, 7, 9] in order to pause the clock when the asynchronous environment wants to communicate with the module. In each cycle, the arbiter decides whether either the internal clock or the environment may proceed. The fact that the designs in both approaches have to resolve metastability, makes their worst-case cycle time unpredictable. Designs for clock synchronization avoiding arbiters are addressed in [12, 15, 3]. The synchronous modules in these designs indicate when they want to communicate and additional circuitry then converts these indications into handshakes.

The clock synchronization approaches mentioned above have all one property in common: they include in the ring oscillator a synchronizing element that on one side can pause the clock and on the other side offers a handshake interface. Each approach has its specific synchronizing element (such as for instance an arbiter). The fact that either the clock or the environment may proceed implies that input and output operations are mutually exclusive (no overlap in the data validity intervals). A communication between two clock domains then implies two conversions: first from one clock domain to handshakes and then from handshakes to the other clock domain. Therefore, during such a commu-

*Philips Research Laboratories, NL-5656 AA Eindhoven, The Netherlands, {joep.kessels,ad.peeters,paul.wielage}@philips.com

§Kwang-Ju Institute of Science and Technology, Kwang-ju, 500-712 South-Korea, sukjinkim@kjist.ac.kr

nication first the one clock and then the other is paused.

We propose a scheme in which the handshake signals are obtained directly by opening the module's ring oscillator. The simplest form of clock synchronization is then by means of a C-element which synchronizes the two clock domains directly with a small and predictable timing overhead (and without the need for any conversion). The common clock signal generated by the C-element can be used to support bidirectional communication. Arbiters are only needed when they are unavoidable, for example for sharing resources between independent clock domains.

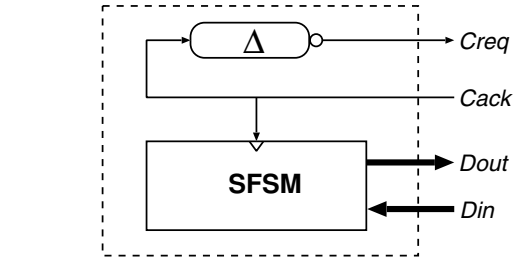
We demonstrate the scheme in the context of processors and memories. We aim at robust modular designs that allow plug-and-play compositionality. Therefore the designs given below may not be the fastest ones, but they are rather easy to reason about. The designs have been implemented in a 0.18 μm CMOS technology. They have been simulated using back annotation with estimated wire loads (with a real model of the memory and a worst-case behavioural model of the processor) and showed functionally correct.

2. A locally clocked module

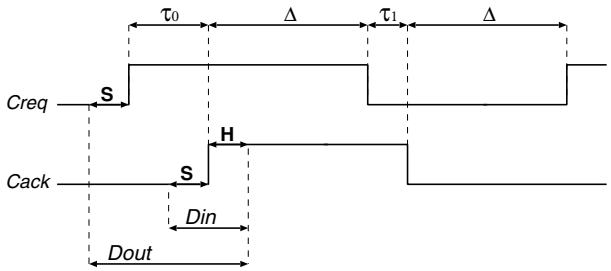
Fig. 1(a) shows the design of a locally clocked module, which contains a synchronous finite state machine SFSM and the provisions for a pausible ring oscillator consisting of a delay line Δ with an inverter (open circle at the end of the delay line). We obtain a ring oscillator by directly connecting the two clock signals *Creq* and *Cack*. We define the *cycle time* of a component as the minimum time it takes for that component to execute one cycle (an up- and a downgoing transition). Therefore the cycle time of this locally clocked module is $2 * \Delta$.

Fig. 1(b) shows a timing diagram of the two clock signals of the locally clocked module. The module delays incoming signal *Cack* over period Δ and then inverts it before sending the result out as signal *Creq*, whereas the environment delays edges of *Creq* for certain (possibly variable) times ($\tau 0$ and $\tau 1$) before feeding them back via *Cack*. Note that the two clock signals execute a handshake protocol in which every clock cycle corresponds with a four phase handshake. Therefore, handshake circuits can be connected to such a handshake port. *Handshake circuits* [1, 11] are constructed from a small set of basic components that use handshake signalling for communication.

The figure also shows the intervals during which the data signals in the bidirectional data path are valid. We assume SFSM to be designed as a conventional synchronous module that can safely operate with a cycle time of $2 * \Delta$. Therefore we may assume that outgoing data *Dout* is valid in the interval starting just (setup time *S*) before the rising edge of *Creq* and, since SFSM is driven by clock signal *Cack*, these signals remain valid until the hold time *H* after *Cack*.



(a) Design



(b) Timing behaviour

Figure 1. Synchronous module with pausible clock

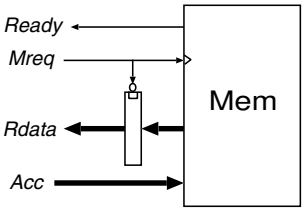
One can easily extend this interval by latching *Dout* when *Creq* is high (latches transparent when *Creq* is low). The incoming data signals (*Din*) must meet the setup and hold requirement with respect to *Cack*.

When comparing the proposed synchronization scheme (direct scheme) with the schemes based on inserting an arbiter in the oscillator ring (arbiter schemes), the following differences can be observed:
- the direct scheme offers an active handshake port (the clock generator starts each handshake by making *Creq* high), whereas an arbiter scheme offers a passive handshake port (the environment has to take the initiative);
- in the direct scheme the clock can be paused in both phases, whereas in an arbitration scheme the clock can only be paused in the low phase;
- the direct scheme allows bidirectional communication, whereas in an arbiter scheme the input and output operations exclude each other in time.

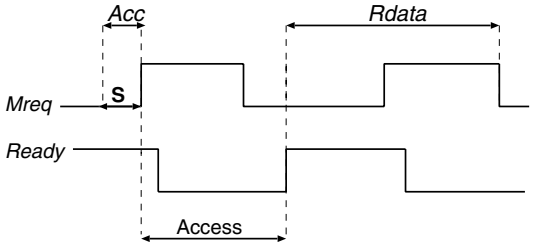
The last difference is very important, since bidirectional communication is the normal way of communication in synchronous systems. It also makes the handshake circuits different from the conventional ones which offer one-way communication.

3. The memory

Fig. 2(a) shows a conventional memory, which has incoming data signals *Acc* (access information) and outgoing data signals *Rdata* (in the case of a read access, the data being read). The access information consists of an address, a read/write indication *wr* (when true indicating a write access), and –in the case of a write access– the data being written. In addition it has two control signals: output signal *Ready* and input signal *Mreq*.



(a) Design



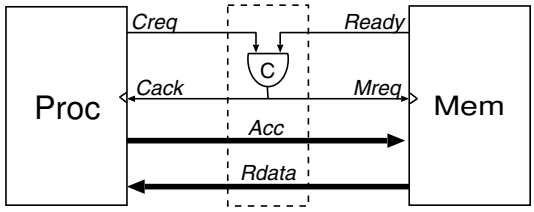
(b) Timing behaviour

Figure 2. Memory

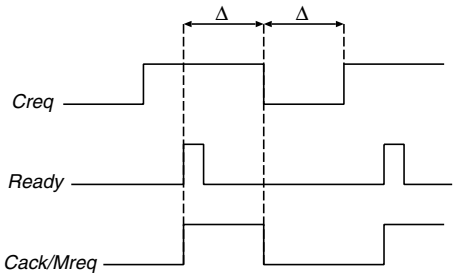
Fig. 2(b) shows the timing diagram of the control signals and the corresponding data validity scheme. Note that these control signals also execute a handshake protocol. When the memory is ready for an access, output signal *Ready* is high. In this state the memory waits for signal *Mreq* to go high and when this happens, it simultaneously makes signal *Ready* low and starts an access corresponding to the access information. As soon as the memory access is completed and signal *Mreq* is low, signal *Ready* is made high again.

Since the memory we used stores the access information, it requires no hold time. Instead it has a rather long setup time of 0.37 ns. These requirements are appropriate for synchronous circuits, but they may lead to additional delay matching in an asynchronous design (as we will see in some of the circuits that we present below).

In the case of a read access, the *Rdata* signals are valid when *Ready* is high. Since we aim at robust solutions that allow the processor clock to be delayed with respect to



(a) Design



(b) Fast processor and slow memory

Figure 3. One processor with one memory

memory signal *Mreq*, we extend the validity period of the read data by latching the memory output when *Mreq* is high (the delay in the processor clock mentioned above can be due to the clock synchronization circuit as well as the clock tree in the processor module).

Since the access time is defined with respect to the rising edge of *Mreq*, the moment of time of the falling edge is not important as long as it occurs before the completion of the access. Therefore, the handshake protocol of the memory is more flexible in time than the protocol of the processor with its fixed delays for both the low and the high phase. Note that a memory access does not correspond with a conventional handshake since it starts and ends with signal *Ready* high.

4. One processor with one memory

Fig. 3 shows a simple system with synchronized clocks consisting of a processor and a memory. The dotted box in the middle, which connects the two active ports, is a so-called bidirectional *passivator* (one of the basic handshake components). Both the processor and the memory operate with a common clock signal, which is obtained by synchronizing both the up- and the downgoing transitions in the signals *Creq* and *Ready* (by means of the C-element in the passivator), and at the rising edges of this common clock signal there is bidirectional communication between the two mod-

ules. This two-phase synchronization could lead to a loss of performance in the case that both modules have fixed asymmetric delays. However, such a performance loss is avoided due to the flexibility in timing offered by the handshaking mechanism of the memory. Therefore, the cycle time of the system is the maximum of the cycle times of the two modules plus the overhead introduced by the C-element. If the memory is faster than the processor, the common clock signal is a symmetric signal based on the delay line of the processor and the cycle overhead is the cycle time of the C-element, which in our implementation is 0.4 ns. If, however, the memory is slower than the processor, we find the more irregular timing behaviour shown in Fig. 3(b) in which we assume the delay of the C-element to be negligible. Since signal *Ready* goes low directly after the start of a new access, the common clock signal is asymmetric in that only the low phase is stretched to match the cycle time of the memory. In this case, the cycle overhead is 0.2 ns, since only the rising edge delay matters.

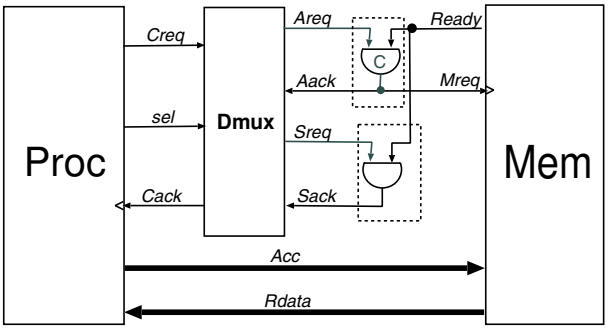
In principle only the upgoing transitions need to be synchronized, since the downgoing ones are not important for the communication. Therefore, one could be tempted to replace the C-element by an AND-gate. In such a design, however, the interface circuitry does not enforce handshake signalling, which implies, for example, that a minimum cycle time of $2 * \Delta$ it is not guaranteed anymore.

This design offers the advantage that the system automatically runs at the minimum cycle time. More convincing advantages of the proposed synchronization scheme show up in more complex designs, of which we present the following cases:
- the memory is not accessed every clock cycle (section 4.1);
- the processor does not wait for the completion of a write access (section 4.2);
- the memory consists of several modules (section 5);
- several processors share a memory (section 6);
- several processors share several memories (section 7).

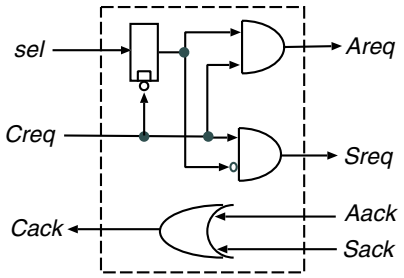
4.1. Conditional memory accesses

In many cases the processor does not need to access the memory every clock cycle. If in that case, memory cycles take longer than processor cycles, the design shown in the previous section does not give the maximum performance. In this section we present a solution in which the processor clock is conditionally synchronized with the memory. For this purpose the processor has an additional output signal *sel* indicating whether the memory is accessed or not.

Fig. 4(a) shows the global design of the processor/memory interface supporting conditional memory accesses. The control circuitry consists of three handshake components: a handshake demultiplexer (Dmux), a passivator and a wait component. Depending on control signal



(a) Global design



(b) Handshake demultiplexer (Dmux)

Figure 4. Conditional memory accesses

sel, the handshake signals *Creq* and *Cack* are demultiplexed over two active ports: *access* port *A* (with signals *Areq* and *Aack*) and *skip* port *S* (with signals *Sreq* and *Sack*). The handshakes through access port *A* are again synchronized with the memory by means of a passivator, whereas the handshakes through skip port *S* are delayed by a wait component until the memory is ready. Therefore, the first skip cycle following an access cycle can be delayed. However, since a skip cycle does not start a new memory cycle, all subsequent skip cycles are not delayed but performed at full processor speed. In the general case, the wait component is constructed from an asymmetric C-element with arbitration. In this case a simple AND-gate will do, since the wait condition (*Ready*) is stable during a skip cycle (a condition is defined as *stable* if it can only make a transition from *false* to *true*).

Fig. 4(b) shows the design of the handshake demultiplexer. Signal *sel* is fed into a latch which is transparent when *Creq* is low. Simple AND-gates can then be used to demultiplex *Creq* over *Areq* and *Sreq* (one may be tempted to use asymmetric C-elements for the demultiplexing, but signal *sel* may become high when *Creq* is high and in that case *Areq* should remain low). The two acknowledge signals are combined into one outgoing acknowledge signal

Cack by means of an OR-gate, which introduces skew between the processor and the memory clock signals (*Cack* and *Mreq*).

One can apply three peephole optimizations. First, the two AND-gates in the skip path can be combined into one three input AND-gate. Secondly, the AND-gate and C-element in the access path can be combined into an asymmetric C-element, which has *Creq* and *Ready* as both-way inputs, and the latched selection signal as conditional input for rising edges (here we use the fact that when *sel* is low, *Areq* remains low). Thirdly, by shuffling inverters around we get faster gates. After these optimizations, the overhead of a skip cycle is 0.41 ns, and for a slow memory, the overhead of an access cycle is 0.39 ns.

This design only makes sense in a single processor architecture if the cycle time of the memory is larger than the cycle time of the processor. In a later section we demonstrate that it also can be used advantageously in architectures in which processors share a memory.

4.2. Posted write accesses

In the design presented above, the first skip cycle after a memory access is delayed until the memory is ready. However, such a delay is only necessary after a read access, in which case the processor has to wait for the read data to become available. For write accesses this synchronization is superfluous. Therefore we can refine the design by supporting so-called *posted write* accesses, which means that the processor does not wait for the completion of a write access, but instead continues concurrently with the memory. In such a design, the interface circuit needs to store information about whether the previous memory access was a read access or not. For this purpose a conventional flipflop *read* is introduced, which at every rising edge of clock signal *Cack* stores the value $sel * \overline{wr}$. The output of this flipflop can then be used to generate signal *ReadRdy*, which is high only if the memory is not busy with a read access ($ReadRdy = Ready + \overline{read}$). Signal *ReadRdy* should then be used as the wait condition for the skip port (instead of signal *Ready* in the previous design).

Fig. 5 shows the simulation results of a design with posted write accesses. The signal ordering has been partly determined by the causal relations in the interface circuitry. Since the processor is driven by signal *Cack*, this signal is used to define the cycles. Initially, the memory is ready and the processor starts with a read access. The next cycle is a skip cycle, which is issued by the processor by making *Creq* high. The demultiplexer transfers the rising edge from *Creq* to *Sreq* and then the wait component delays the edge until the memory has completed the read access. As soon as *Ready* becomes high, *Cack* goes high implying that the processor starts executing the skip cycle. The second skip cycle is not delayed because the memory remains ready. The fourth cycle is a posted write access, which implies that the subsequent skip cycle is not delayed although the memory is not ready.

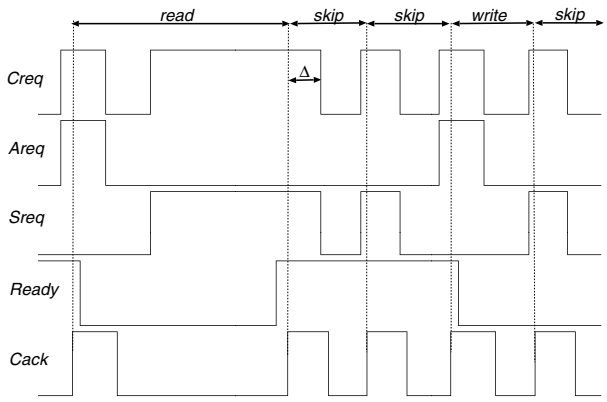


Figure 5. Posted write accesses

5. One processor with several memories

It is straightforward to generalize the architecture proposed in the previous section to one in which the processor can access several memories. First the demultiplexer has to be generalized to one having several access ports and one skip port with the latter one being activated when all select signals are low. During a read access, the outputs of the memories have to be multiplexed depending on the memory being accessed. Control of that multiplexer can based on the *read* flipflops that have been introduced for the posted writes (this multiplexing may lead to delay matching in the *Cack* signal).

For each active port of the demultiplexer a *ReadRdy* signal is generated (the *ReadRdy* signal of the skip port holds by definition) and the handshake of each port is delayed (by an AND-gate) until the *ReadRdy* signals of all *other* ports hold. By delaying the acknowledge instead of the request signals, we obtain maximum concurrency. This design allows several memories to be active with a write access and even two memories with a read access (must be the last two accesses).

Fig. 6 shows the result of a simulation in which the processor is copying information from a fast memory into a slow one. For this purpose the processor starts by first reading one data word and then executes a sequence of alternating read and write accesses. In each write access, it writes the data read in the one but last read acces. The read memory is fast enough to handle a read access every processor cycle (*Rready* is the ready signal of the fast read memory), whereas the cycle time of the slow write memory is about

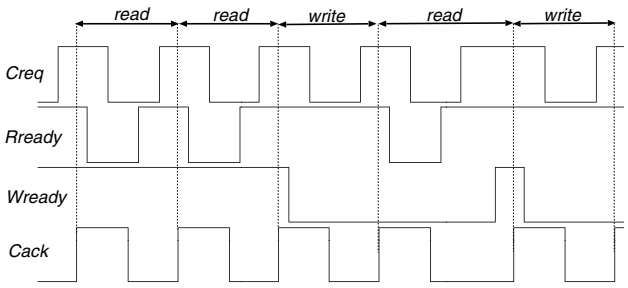


Figure 6. Processor accessing two memories

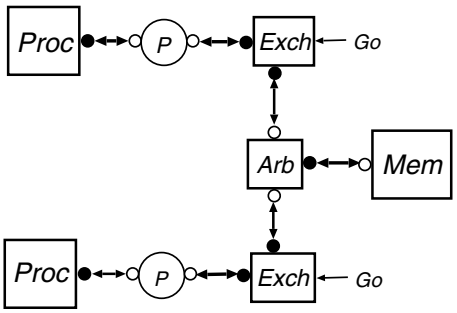


Figure 7. Two processors sharing a memory

three times the cycle time of the processor (see *Wready* signal). Since the design supports posted write accesses, the processor is not held up during the first three accesses (read, read and write). From signal *Wready* we see that during the third read access, both memories are active concurrently. From this moment on, both a read and a write access take about the time of a write cycle only. Note that the processor could even perform an additional skip cycle without much time penalty.

6. Several processors sharing one memory

In this section we design a system in which two processors share a memory. In order to allow this shared memory to be easily incorporated in one of the previous designs, we make the interface between the processor and the access circuitry equal to the interface between the processor and the memory proper. But first we have to solve a problem in the memory access protocol. At the end of a read access, the memory – in one communication – delivers read data while receiving access information for the next access. This fact complicates sharing, since these two accesses may come from different processors. We solve this problem by making the memory passive, which means (by inversion) converting the ready signal into an acknowledge signal *Mack*. In this way the partitioning of the sequence of handshake events into handshakes is shifted by one event. Sending access information (*Mreq* high) and receiving read data (*Mack* low) then become events in the same handshake, which means that accesses and handshakes now correspond. Note that in this scheme (in [11] called the late data valid scheme for pull channels) the read data is valid when *Mack* goes low.

Fig. 7 shows the design of a system in which two processors share a memory. Since this design is more complex than the previous ones, we present it at the level of handshake modules. Basic components are represented by circles and compound modules by boxes. We use the well-known convention that bullets indicate active ports and open

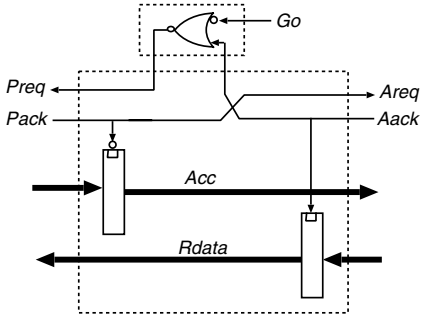


Figure 8. Simple exchange module

circles passive ports. In the design we have two processors (*Proc*) each communicating with a so-called exchange module (*Exch*) via a passivator (P). Each exchange module receives the access information from the processor, then accesses the passive memory via the arbiter module (*Arb*) and in the case of a read access transfers the read data to the processor.

Fig. 8 shows the design of the exchange module, which has two active bidirectional handshake ports: port P for the processor and port A for the arbiter. The exchange module consists of two handshake components: a repeater and a two-way transferrer. On top we have the repeater, which generates an infinite sequence of handshakes. In fact, the repeater provides only the oscillation inverter with an additional input (*Go*) for initialization. The handshakes of the repeater are sent to the transferrer, which for each handshake from the repeater executes two overlapping handshakes: first an upgoing phase via the P-port followed by an upgoing phase via the A-port, then a downgoing phase via the P-port followed by a downgoing phase via the A-port. One would obtain a correctly functioning system by directly connecting the passive memory to the A-port of the exchange module. In that case, the latching registers are not needed and compared to the design in section 4 we only moved the activity-driving inverter from the memory to the

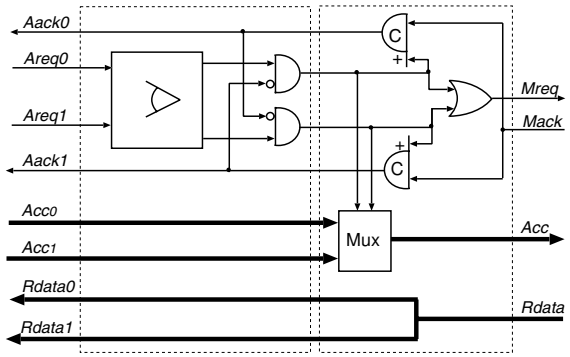
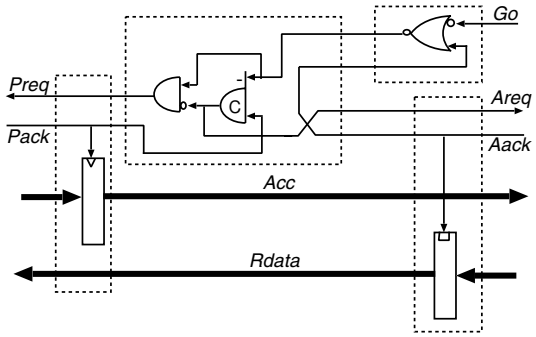


Figure 9. Handshake memory arbiter

exchange module. Since the memory is shared, however, the A-port of the exchange module is connected to one of the two passive ports of the handshake arbiter (*Arb*) and for that reason the latching registers are needed. The access information has to be latched, since the upgoing phase at the P-port may now take the time of a memory access and during that time the processor may change the access information. The read data has to be latched, since if the memory is fast, it can be active serving the other processor before the read data has been received. The read latches in the exchange module make the latches at the output of the memory in Fig. 2(a) superfluous (functionally they replace these latches).

Fig. 9 shows the design of the memory arbiter, which consists of two handshake components: a mutual exclusion element and a multiplexer. In the multiplexer, delay matching is needed in signal *Mreq* to deal with the setup time of the memory as well as to compensate for the delay introduced by the multiplexer in the *Acc* data path. The delay matching requirements can be reduced by using the ungated (before the AND-gates) grant signals in the mutual exclusion element. The timing overhead of the memory arbiter is 0.88 ns.

The bullet/open-circle convention allows us to easily identify the activity cycle paths. A module is called an *activity source* if it has only active ports, an *activity sink* if it has only passive ports, and otherwise it is called an *activity pipe*. The activity sources contain the inverters making a cycle path oscillate. Since the design contains four activity sources, it also contains four cycle paths. Each *Proc* cycle path is closed by a passivator (activity sink), whereas each *Exch* cycle path is closed by two activity sinks: on one side by a passivator and on the other side via the arbiter (activity pipe) by the shared memory.



(a) Design

$$
\begin{array}{l}
(\ \ P_{req}\uparrow;\ \ [P_{ack}] \\
;\ \ (\ (P_{req}\downarrow;\ [\neg P_{ack}])\ ||\ (A_{req}\uparrow;\ [A_{ack}])\ ) \\
;\ \ A_{req}\downarrow;\ \ [\neg A_{ack}] \\
)^*
\end{array}
$$

(b) Behaviour

Figure 10. Exchange module as pipeline stage

6.1. Posted write accesses

Since the exchange module offers at its P-port a memory interface, we can insert in between the processor and the passivator any of the conditional access modules of the previous two sections. However, if we insert the posted write circuitry of section 4.2, it makes sense to also modify the exchange module, since the exchange module in Fig. 8 will always delay the processor when the memory is serving the other processor. Such an unnecessary delay would be avoided if the exchange module would behave as a pipeline stage.

Fig. 10(a) shows the design of such an exchange module, which consists of four handshake components: a repeater, a sequencer (design based on a so-called T-element) and two one-way transferrers. The handshakes of the repeater are sent to the sequencer, which for each handshake from the repeater executes two handshakes. Fig. 10(b) shows the behaviour of the exchange module (using the convention introduced in [6]). The exchange module first executes the upgoing phase of a handshake via the P-port. Subsequently the module completes the handshake via the P-port while simultaneously performing the upgoing phase of a handshake via the A-port. Finally it executes the downgoing phase of the handshake via the A-port. Since the downgoing phase of the handshake via the P-port is performed concurrently with the upgoing phase of the handshake via the A-port,

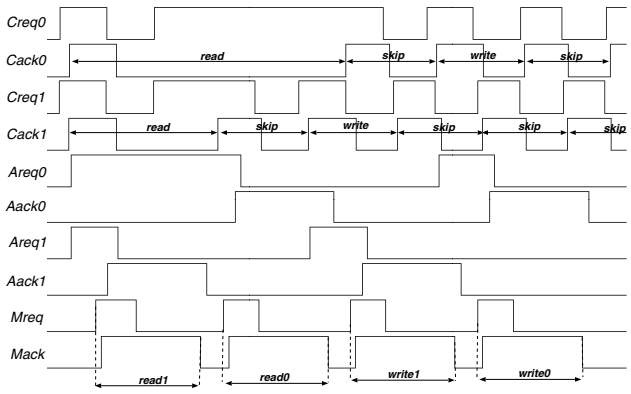


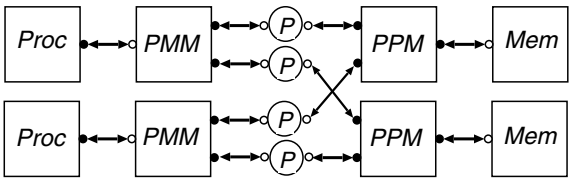
Figure 11. Shared memory with posted write accesses

the processor can complete its handshake independently of any delay introduced by the arbiter. Consequently, the access information has to to remain valid after completion of the handshake at the P-port, which makes the use of latches that are transparent when *Pack* is low not safe. Therefore the access information is stored in flipflops at the rising edges in *Pack* (which complies with the processor interface). The cycle time of this exchange module is 0.49 ns.

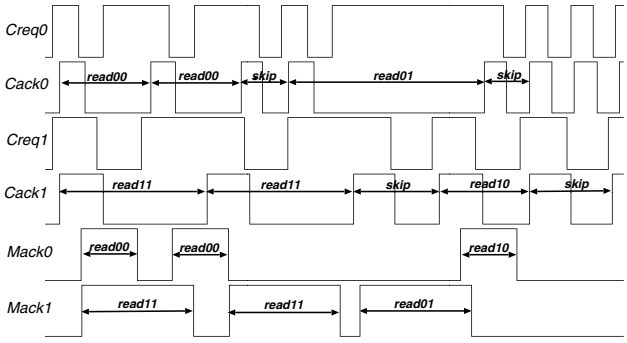
Fig. 11 shows the simulation results of a design combining memory sharing with posted writes. In this simulation both processors start at the same time and they both execute a read, skip, and write cycle followed by a series of skip cycles. From the simulation results we see that the arbiter decides to give $Pr\ o_{\mathbb{C}}$ priority. Therefore $Pr\ o_{\mathbb{C}}$ is already executing its skip cycle when the memory is performing the read access of $Pr\ o_{\mathbb{6}}$.

7. Several processors sharing several memories

The handshake modules presented so far can be used as plug-and-play building bricks to construct interconnect networks connecting several processors with several memories. In this section we present both a crossbar and a bus network connecting two processors with two memories. Fig. 12(a) shows the crossbar network in which the two processors only compete if they access the same memory. Module *PMM* is the module connecting one processor to two memories (section 5) and module *PPM* is the module connecting two processors to one memory with posted writes (section 6.1). Fig. 12(b) shows a timing simulation in which the processors and memories all have different cycle times (from a timing perspective a truly heterogeneous system). In this simulation, the processors only perform read accesses with $read_{ij}$ standing for a read access from processor i to memory j. There is only one collision: the third access of processor 0 and the second access of processor 1 are both to memory 1. Since the access of processor 0 starts later, it is delayed until the access of processor 1 has been completed.



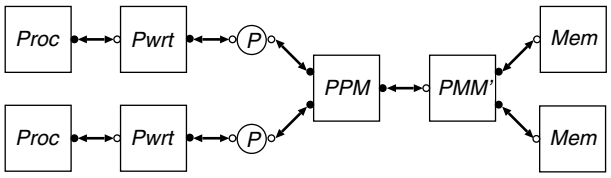
(a) Design



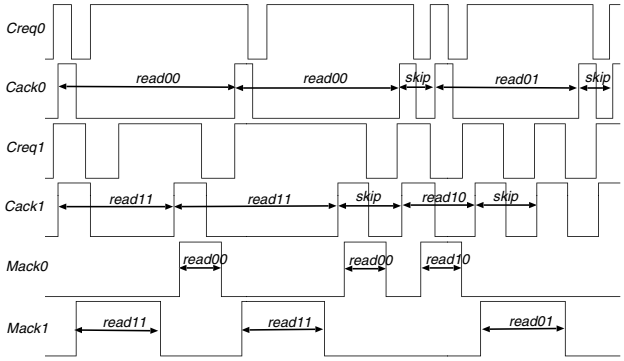
(b) Simulation trace

Figure 12. Crossbar network

In the crossbar network, multiplexing is done before sharing. If we do it the other way around, a bus network is obtained. Fig. 13(a) shows the bus network, which at the processor side of the common communication channel, is the same as the design in which two processors share a memory with posted write accesses (section 6.1). Module *Pwrt* is the module supporting posted write access (section 4.2). Module *PMM'* is a reduced version of *PMM* in that it does not contain the circuitry supporting posted writes. Moreover the fact that the module interfaces directly with passive memories, can be used to simplify the control circuitry. Fig. 13(b) shows a timing simulation of the bus network, in which everything –apart from the network– is the same as in the simulation of Fig. 12(b). Note that all memory accesses are now mutually exclusive. Compared to a crossbar a bus network needs less area, but it also offers less performance, since the processors have to compete for every memory cycle.



(a) Design



(b) Simulation trace

Figure 13. Bus network

8. Concluding remarks

We have presented a clock synchronization scheme that supports safe communication between independently clocked submodules. The scheme is based on the well-known technique of pausible clocks. However, instead of including in each ring oscillator a standard element for pausing the clock (such as for instance an arbiter), we propose a scheme in which each synchronous module has both an incoming and an outgoing clock signal, which have been obtained by opening the module's ring oscillator. Since these clock signals behave as handshake signals, handshake circuits can be used to synchronize the clocks. The simplest way of obtaining clock synchronization is then by means of a C-element, which introduces only a small and predictable timing overhead.

We have demonstrated the scheme for several processors/memory architectures. We constructed the interface circuits from rather small handshake components. Local transformations (based on global invariants) were later used to optimize the designs. The circuits were designed in a 0.18 μm CMOS technology. Simulations –with back annotation based on estimated wire loads– showed the circuits to be correct. The measured timing overheads have been presented. In many cases one can speed up the design by subtracting the minimum timing overhead of the synchronization circuitry from the delay line in the processor.

We used a conventional memory, which means a memory with a long setup and a short hold time, This is exactly the behaviour that is needed by a synchronous environment. However, in several of our designs (the memory sharing ones) the long setup times led to a larger cycle overhead, whereas on the other hand it was often required to extend the data validity period of the read data. Therefore, for asynchronous circuits, memories that store the read data instead of the access information (no setup and long hold times) are more appropriate.

Arbitration-free synchronization has a fundamental restriction: the clock of the synchronous module is paused until the communication via the active port is completed (in [9] called *demand communication*). A design, in which the module's clock is never held up, because a communication (via a passive port) is only performed when the other side has indicated its willingness to communicate (*probed/poll communication*), would require arbitration. A simple example may help to explain the difference between the two types of communication. Assume the synchronous module has to put a data item in a full buffer. In a demand communication the module's clock is paused until the buffer has a vacancy. This type of communication can easily be implemented without an arbiter (in fact the exchange module in fig. 10 can be simplified to a one-place buffer). However, if the clock should not be paused when the buffer is full, a conditional communication via a probed passive port is needed. In that case the synchronous module receives each clock tick a boolean indicating whether the communication took place or not. Such conditional communications require arbitration, but that would also have been the case in a fully asynchronous implementation (choice on a probe). Our conjecture is that the proposed synchronization scheme only requires arbitration, if this would also have been the case in a fully asynchronous design.

There are many processor/memory architectures that we did not discuss, but for which the synchronization technique is also useful. For instance memories with varying response times, such as memories with caches. The scheme can also be applied for the communication between processors and peripheral units. We proposed the scheme for clock synchronization of communicating synchronous modules. The scheme can, of course, also directly be applied to combine synchronous and asynchronous (handshake) modules. In the asynchronous bus interface presented in [5] the processor and the peripheral units communicate through variables using a nonput broadcast channel for synchronization. With the proposed clock synchronization scheme, this design can easily be adapted to support the communication between a synchronous/asynchronous processor and synchronous/asynchronous peripheral units.

We have not discussed the problem of initialization. In [1, 11] a structural solution is presented for the initialization of handshake circuits, which is based on the so-called *initialization property* of each handshake component. It prescribes that if all handshake input signals are low, the component is forced into its initial state, and in that state all outgoing handshake signals are also low. We can apply this solution at system level, if the processors and memories also have this initialization property, which means that both modules also get an additional request signal *Go* indicating that the module must perform its function (when low the module should be reset and its handshake output signals should become low as well). We can replace the inverter in the ring oscillator by the repeater shown in the previous section and in that way obtain a ring oscillator having the initialization property. In addition the synchronous finite state machine should be reset when *Go* is low. For the memory we have to force the outgoing control signal to low when signal *Go* is low.

One may argue that systems based on ring oscillators are not well suited to applications having to meet hard timing requirements. There are, however, techniques for calibrating the delay lines on-the-fly using (low frequency) oscillators [8, 18, 17]. These techniques have even been applied in products [13]. A second observation is that since both the ring oscillator and the processor are CMOS circuits on the same chip, timing variations due to external causes, such as fluctuations in temperature, supply voltage etc, will have the same effect on both circuits. Therefore, the processor will automatically adapt its speed to these external conditions. If the clock signal is derived from an external crystal oscillator, its frequency is fixed and must therefore be low enough to be able cope with the worst-case conditions. Therefore, if synchronization with an external timing reference is required, it should be done at the external interfaces with the coarsest possible time grain. Note that the proposed synchronization technique can also be applied for this purpose.

References

[1] K. v. Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[2] D. S. Bormann and P. Y. Cheung. Asynchronous wrapper for heterogeneous systems. In *Proc. International Conf. Computer Design (ICCD)*, Oct. 1997.

[3] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, Oct. 1984.

[4] W. J. Dally and J. W. Poulton. *Digital Systems Engineering*. Cambridge University Press, 1998.

[5] J. Kessels, A. Peeters, T. Kramer, M. Feuser, and K. Ully. Designing an asynchronous bus interface. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 108–117. IEEE Computer Society Press, Mar. 2001.

[6] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[7] S. W. Moore, G. S. Taylor, P. A. Cunningham, R. D. Mullins, and P. Robinson. Interfacing asynchronous and synchronous subsystems. In A. Yakovlev and R. Nouta, editors, *Asynchronous Interfaces: Tools, Techniques, and Implementations*, pages 129–132, July 2000.

[8] S. W. Moore, G. S. Taylor, P. A. Cunningham, R. D. Mullins, and P. Robinson. Self-calibrating clocks for globally asynchronous locally synchronous systems. In *Proc. International Conf. Computer Design (ICCD)*, Sept. 2000.

[9] J. Muttersbach. *Globally-Asynchronous Locally-Synchronous Architectures for VLSI Systems*. PhD thesis, ETH, Zürich, 2001.

[10] J. Muttersbach, T. Villiger, and W. Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 52–59, Apr. 2000.

[11] A. M. G. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Eindhoven University of Technology, June 1996.

[12] M. Pečhouček. Anomalous response times of input synchronizers. *IEEE Transactions on Computers*, 25(2):133–139, Feb. 1976.

[13] Philips Semiconductors. PCA5010 Pager baseband controller . `http://www.semiconductors.philips.com/pip/PCA5010`, 1998.

[14] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T.-P. Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, C-37(9):1005–1018, Sept. 1988.

[15] C. L. Seitz. System timing. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[16] J. N. Seizovic. Pipeline synchronization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 87–96, Nov. 1994.

[17] G. Taylor, S. Moore, S. Wilcox, and P. Robinson. An on-chip dynamically recalibrated delay line for embedded self-timed systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 45–51, Apr. 2000.

[18] S. Temple and S. B. Furber. On-chip timing reference for self-timed microprocessor. *Electronics Letters*, 36(11):942–943, 2000.

[19] K. Y. Yun and R. P. Donohue. Pausible clocking: A first step toward heterogeneous systems. In *Proc. International Conf. Computer Design (ICCD)*, Oct. 1996.

[20] K. Y. Yun and A. E. Dooply. Pausible clocking-based heterogeneous systems. *IEEE Transactions on VLSI Systems*, 7(4):482–488, Dec. 1999.