Asynchronous Arbitration Primitives for New Generation of Circuits and Systems

Andrey Mokhov*, Danil Sokolov, Victor Khomenko, Alex Yakovlev
Newcastle University, Newcastle upon Tyne, United Kingdom
*Corresponding author: andrey.mokhov@ncl.ac.uk

***Abstract*—This paper presents an overview of a family of asynchronous arbitration primitives designed to increase the resilience and efficiency of the new generation of circuits and systems. We cover primitives for synchronisation and decision-making with an emphasis on interfacing analogue and digital worlds, sampling of non-persistent signals, and efficient handling of correlated sensor events.**

I. Introduction

The new generation of circuits and systems is breaking conventional walls between different timing, power and technology domains. Modern 'synchronous' and 'digital' systems are emphatically asynchronous-at-large and analogue-at-heart: they contain tens to hundreds of timing and voltage domains, some even thousands [1]. Asynchronous arbitration primitives are now used both to orchestrate the communication between different clock domains [2] and to control the analogue-digital interfaces in on-chip power regulators [3].

This paper presents an overview of the recently developed asynchronous arbitration primitives that address the problem of interfacing between hazardous and/or analogue worlds and hazard-free asynchronous circuits in a safe way. The primitives are also useful in purely digital settings, as they increase the robustness of event-driven asynchronous circuits by shortening their *event sensitivity interval*, where an unexpected event, e.g. caused by a particle strike, can corrupt the state of the system.

We cover synchronisation and decision-making primitives in Sections II and III, respectively. All of the presented circuit specifications and implementations have beed developed using the open-source asynchronous design tool Workcraft [4] and are publicly available under the MIT license [5].

Formal specification of asynchronous circuits

Signal Transition Graphs (STGs) are commonly used for the specification, verification and synthesis of asynchronous circuits. See a comprehensive background on STGs in [6].

As a brief introduction, let us examine the STG specification shown in Fig. 1(left-middle). The STG specifies the behaviour of a circuit with the inputs $\{\mathsf{sig}, \mathsf{ctrl}\}$ and the output san. Rising and falling *signal transitions* are depicted by text nodes, e.g. sig+ corresponds to the signal sig switching from 0 to 1. There are also *dummy transitions*, such as e, which do not correspond to changes of signal values (think of them as abstract events). Input, output and dummy transitions are conventionally shown in red, blue and black colour, respectively.

Directed arcs express the *precedence* relation between transitions, e.g. the arc san+ $\longrightarrow$ ctrl- indicates that the input

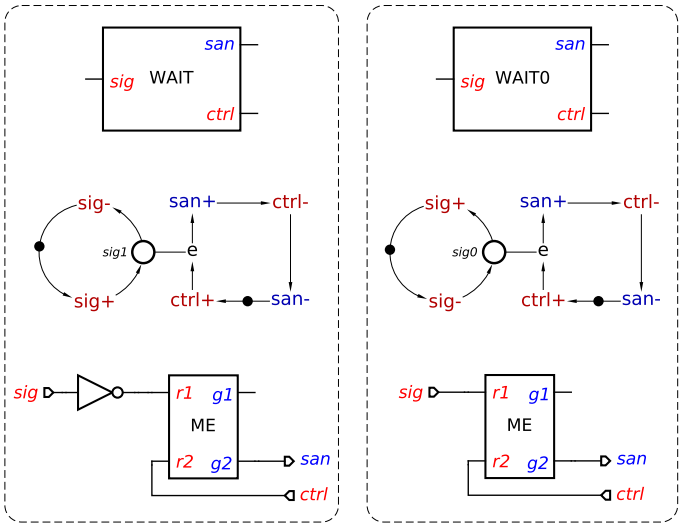

Fig. 1. WAIT and WAIT0: block diagram, specification and implementation.

ctrl goes low only after the output san goes high. An arc can be *marked* with a *token*, shown as a black dot, to specify the initial state of the circuit: the tokens before the transitions sig+ and ctrl+ imply the initial state sig=ctrl=san=0.

A transition is *enabled* to occur if it has tokens on each of its incoming arcs, otherwise it is *disabled*, e.g. ctrl+ is enabled and san- is disabled. When a transition occurs, it *consumes* the tokens on its incoming arcs are *produces* them on its outgoing arcs, e.g. ctrl+ moves the token to the arc ctrl+ $\longrightarrow$ e. Undirected arcs, called *read arcs*, are used to *test* the existence of a token in a *place*, shown as a circle (think of it as just a placeholder for a token), e.g. the read arc sig1 —— e allows the dummy e to occur only if the place is marked, i.e. if sig=1.

A transition is *persistent* if, once enabled, it will occur without first becoming disabled. The dummy e is the only *non-persistent* transition in the example: sig- can disable it by consuming the token from sig1. Non-persistence can manifest itself as a *hazard* in the circuit, whereby a gate starts switching but is stopped midway resulting in a short analogue pulse.

II. Synchronisation primitives

This section covers the primitives for the synchronisation of hazard-free signal transitions in asynchronous circuits with potentially hazardous signals coming from the environment.

WAIT and WAIT0

The WAIT element [7], shown in Fig. 1(left), synchronises the asynchronous handshake ctrl/san with the non-persistent

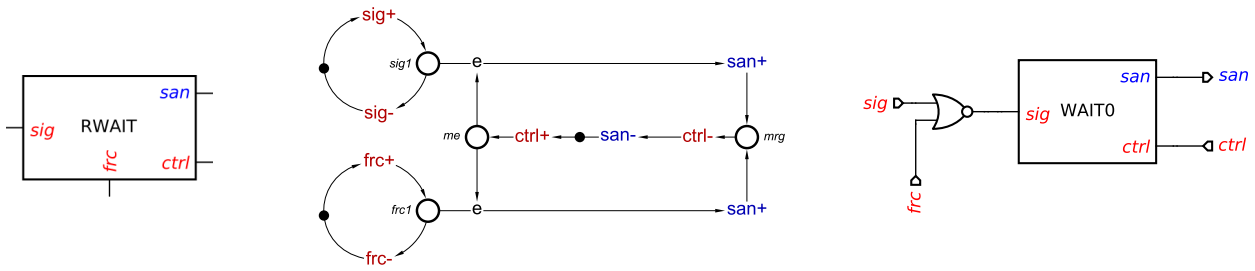

Fig. 2. RWAIT: block diagram, specification and implementation.

input sig. According to the STG specification, sig is unconstrained and is allowed to switch between 0 and 1 values with no regard to the output handshake – this is captured by the loop with signal transitions sig+ and sig-. Initially ctrl=0 and the input is isolated from the output san. By raising the ctrl input, the element can be switched into the *waiting mode*, where it stays until sig becomes high and enables the dummy e. The output san=1 is then produced and persistently held[1] until WAIT is reset by releasing the ctrl input (the transition ctrl-).

Note that an input spike (sig+ followed by sig-) in the waiting mode can be too short fot the WAIT element to register it, in which case the spike is ignored. The non-persistent behaviour and the associated metastability is fully contained within the element, guaranteeing a clean hazard-free output. This is achieved using the *mutual-exclusion* ME element [8].

WAIT is a fundamental synchronisation primitive that is used for implementing other, more sophisticated components presented in this paper. The symmetric version of the element that waits for the input to become low is called WAIT0; its top-level block diagram, the STG specification and the implementation are shown in Fig. 1(right).

RWAIT and RWAIT0

RWAIT and RWAIT0 are modifications of the WAIT and WAIT0 elements, respectively, with a possibility to persistently cancel the waiting request. This is useful when the input is no longer expected to change or the change is no longer relevant for the asynchronous controller, and hence the output handshake needs to be released.

See the block diagram of RWAIT in Fig. 2. The additional input frc can be used to force the reset of the output handshake in the waiting mode. The STG specifies that the output transition san+ can be caused either by sig+ (the top branch) or by frc+ (the bottom branch). The implementation reflects the resulting OR-causality [9]: the inputs sig and frc are simply combined via a NOR gate, whose output is synchronised with the handshake ctrl/san using the WAIT0 element.

WAIT01 and WAIT10

WAIT01 and WAIT10 elements wait for a rising or falling edge of the input signal, respectively. Note that this is subtly different from waiting for a high or low input value, e.g. a signal can be initially low, and to generate a falling edge event it must first go high. The WAIT01 specification in Fig. 3(left-middle) tracks the input changes via two dummy transitions

[1]Here and further on, the output san is the 'sanitised' (persistent) version of the 'dirty' (non-persistent) input sig.

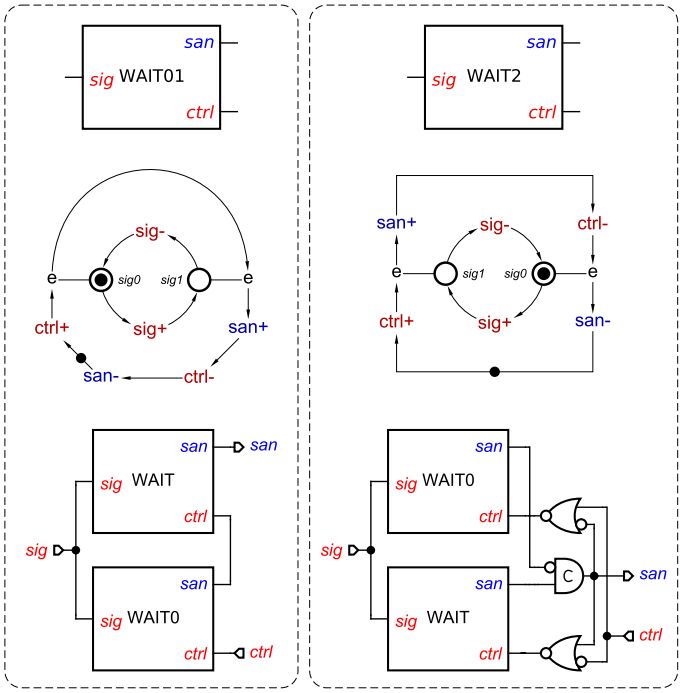

Fig. 3. WAIT01 and WAIT2: block diagram, specification, implementation.

that are enabled in sequence when sig=0 and sig=1 hold. This can be implemented by connecting WAIT0 and WAIT in sequence, as shown in Fig. 3(left-bottom).

An element waiting for an arbitrary fixed pattern of alternating 0s and 1s, e.g. the symmetric WAIT10 element, can be implemented analogously.

WAIT2

WAIT2 is another combination of WAIT and WAIT0: it uses a 2-phase output handshake, waiting for high and low input values, one after the other, see Fig. 3(right). One can think of WAIT2 as a 2-phase version of the WAIT element, or as a C-element whose input sig is hardened against hazards.

The STG contains two loops: the inner sig loop, which is unconstrained, and the outer handshake loop that synchronises the rising (ctrl+ $\longrightarrow$ san+) and the falling (ctrl- $\longrightarrow$ san-) phases with conditions sig=1 and sig=0, respectively.

The implementation uses a toggle-like controller to steer the rising and falling edges of ctrl to the inputs of the WAIT and WAIT0 elements, in sequence, and take care of their appropriate reset. The controller was developed in WORKCRAFT [4] using conventional asynchronous design flow (we omit further details due to the lack of space).

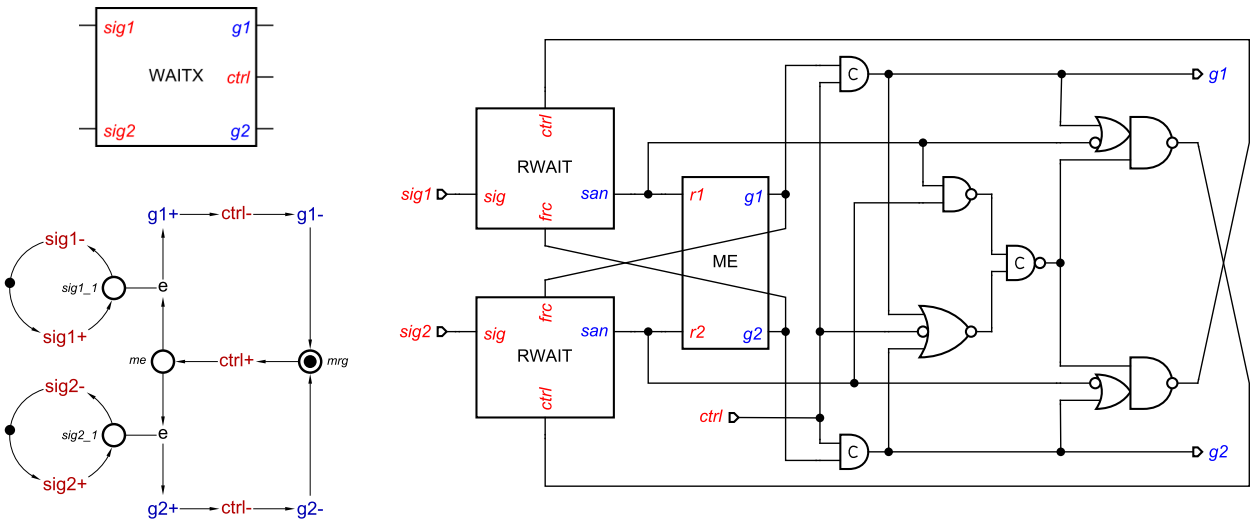

Fig. 4. WAITX: block diagram, specification and implementation.

III. Decision-making primitives

This section presents a family of *decision-making* components that perform non-trivial event coordination and rely on the previously introduced synchronisation primitives.

WAITX

The WAITX element [10] arbitrates between two non-persistent inputs $\{\mathsf{sig1}, \mathsf{sig2}\}$, producing a clean asynchronous dual-rail handshake: depending on which of the two signals arrives first, exactly one of the grant signals $\{\mathsf{g1}, \mathsf{g2}\}$ is issued, see Fig. 4. The place me with two consuming arcs represents the arbitration decision that needs to be made: if the inputs arrive very close to each other, both of the two dummy transitions can become enabled but only one of them can occur, since me can have at most one token. In the reset phase both branches are merged in the place mrg.

WAITX isolates the outputs both from the metastability associated with non-persistent inputs, as well as from the metastability associated with making the decision of which input signal arrives first. The implementation relies on RWAIT elements for synchronisation with non-persistent signals, and uses an ME element to make the decision on their arrival order. See [10] for further implementation details and for a generalisation to more than 2 input signals.

WAITX2

WAITX2 behaves as WAITX in the rising phase and as WAIT0 in the falling phase, i.e. it does not release the output asynchronous handshake until the winning input signal goes low. It uses a 2-phase output handshake similarly to WAIT2, and the specification, shown in Fig. 5, is therefore a combination of the STGs for WAITX and WAIT2.

The implementation comprises WAITX and two WAIT0 elements controlled by toggle-like asynchronous logic, which activates the right WAIT0 element in the reset phase. The synthesis and technology mapping of the control was performed using conventional asynchronous design approaches automated in WORKCRAFT [4].

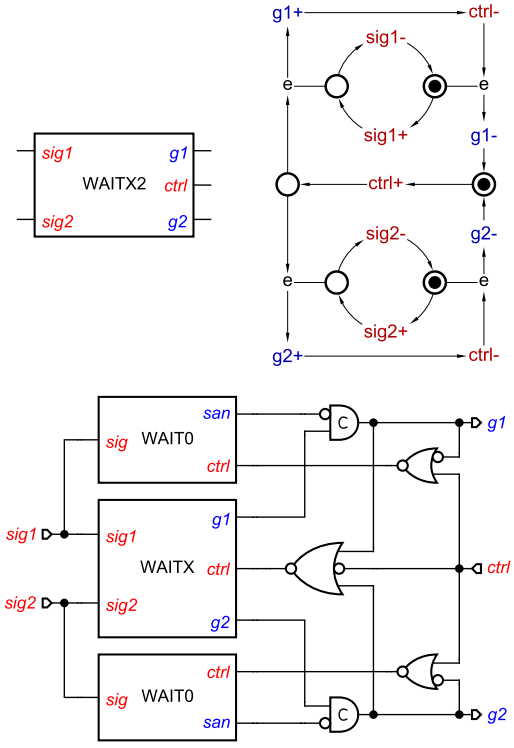

Fig. 5. WAITX2: block diagram, specification and implementation.

SAMPLE

The purpose of the SAMPLE element is to check whether the voltage on the input sig is above the threshold, see Fig. 6. The specification is similar to that of WAITX but the two dummy transitions are controlled by conditions corresponding to the state of the same input sig.

The implementation is based on WAITX that decides which of the two inputs, sig or its inverted version, becomes high first. Note that both inputs of WAITX may be high at the same time, e.g. during a transition of sig, in which case SAMPLE is allowed to make an arbitrary decision. However, if the input signal is stable, no metastability resolution is required and SAMPLE can therefore quickly produce the correct output.

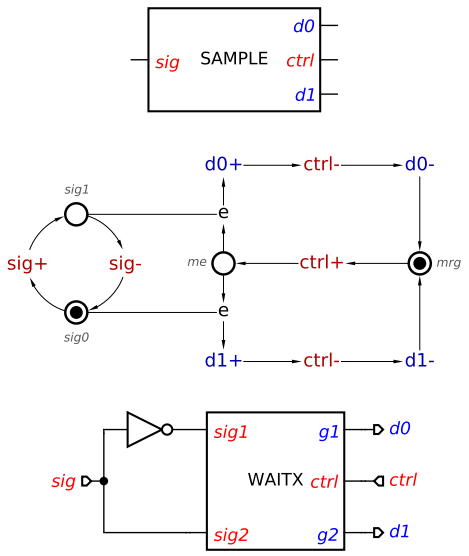

Fig. 6. SAMPLE: block diagram, specification and implementation.

Opportunistic Merge

The *opportunistic merge* OM element [11] merges two request-acknowledgement channels $\{\mathsf{r1}/\mathsf{a1}, \mathsf{r2}/\mathsf{a2}\}$ into one r/a and can opportunistically bundle requests from different input channels if they arrive sufficiently close to each other.

Fig. 7(middle) clarifies the difference between the standard *merge* element [12] and OM. The conceptual state graph of the merge element is shown on the left. Note that the bottom state of the graph is not persistent: outputs a1 and a2 disable each other, hence this is a decision-making element. The state graph for OM, shown on the right, has an additional 'opportunistic bundle' transition labelled by $\{\mathsf{a1}, \mathsf{a2}\}$ that sends acknowledgements to both input channels.

The intended application of OM is to handle concurrent (and potentially correlated) requests from several clients to a kind of service that benefits all the clients simultaneously. Examples include triggering an alarm by (any of) several sensors, recharging of a shared DRAM, and various kinds of power management [3]. All these use cases benefit from serving multiple requests in bundles. The STG specification of OM, as well as further implementation details can be found in [11].

The input channels of OM are assumed to be hazard-free, but one can use the synchronisation primitives presented in Section II to generate clean hazard-free handshakes from hazardous input signals, e.g. produced by analogue sensors.

IV. Conclusions

The paper presented an overview of asynchronous arbitration primitives that can be used as basic building blocks for the new generation of circuits and systems, where resilient and efficient synchronisation between multiple clock and voltage domains is an important challenge. The primitives were developed using formal methods and are publicly available [5]. Our current research is focused on providing support for their automated insertion into asynchronous controllers operating on the boundary with hazardous environment.

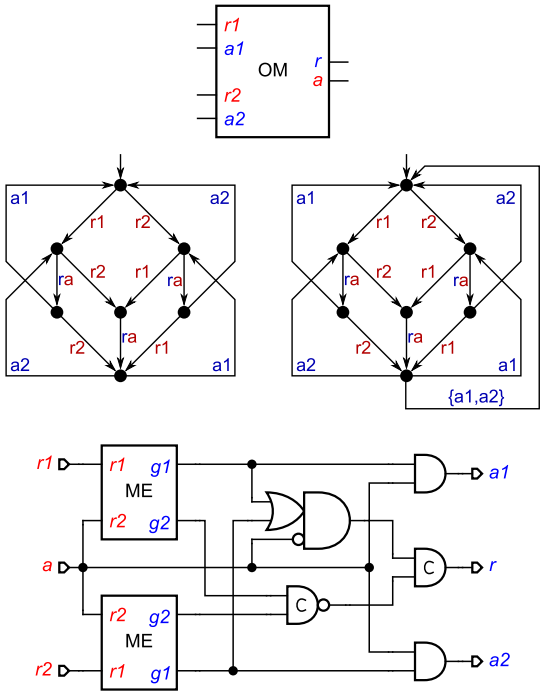

Fig. 7. OM: block diagram, conceptual state graphs, implementation.

Acknowledgements

This research was supported by EPSRC grant EP/L025507/1 "A4A: Asynchronous design for Analogue electronics".

References

[1] B. Bohnenstiehl, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu, A. T. Tran, E. Adeagbo, and B. M Baas. KiloCore: A 32-nm 1000-Processor Computational Array. *IEEE Journal of Solid-State Circuits*, 2017.

[2] W. Jiang, D. Bertozzi, G. Miorandi, S. Nowick, W. Burleson, and G. Sadowski. An asynchronous NoC router in a 14nm FinFET library: Comparison to an industrial synchronous counterpart. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[3] D. Sokolov, V. Dubikhin, V. Khomenko, D. Lloyd, A. Mokhov, and A. Yakovlev. Benefits of asynchronous control for analog electronics: Multiphase buck case study. In *Design, Automation & Test in Europe Conference (DATE)*, 2017.

[4] WORKCRAFT website. www.workcraft.org.

[5] Open-source library of asynchronous arbitration primitivies. GitHub repository: https://github.com/workcraft/arbitration-primitives.

[6] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis for Asynchronous Controllers and Interfaces*. Springer, 2002.

[7] D. Sokolov, V. Khomenko, A. Mokhov, A. Yakovlev, and D. Lloyd. Design and verification of speed-independent multiphase buck controller. In *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2015.

[8] D. J. Kinniment. *Synchronization and Arbitration in Digital Systems*. John Wiley and Sons, 2008.

[9] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny. On the models for asynchronous circuit behaviour with OR causality. *Formal Methods in System Design*, 9(3):189–233, 1996.

[10] V. Khomenko, D. Sokolov, A. Mokhov, and A. Yakovlev. WAITX: An Arbiter for Non-Persistent Signals. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), 2017*. IEEE, 2017.

[11] A. Mokhov, V. Khomenko, D. Sokolov, and A. Yakovlev. Opportunistic merge element. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), 2015*. IEEE, 2015.

[12] M. Greenstreet. Real-time merging. In *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 186–198. IEEE, 1999.