Fifty Shades of Synchrony

Andreas Steininger

TU Wien, Embedded Computing Systems Group E182-2,
Treitlstr. 3, A-1040 Vienna,
steininger@ecs.tuwien.ac.at

Abstract. This paper reflects on how, from the view of a VLSI designer, we coordinate activities in our daily lives. Example cases from transportation and recreational activities illustrate that both, synchronous and asynchronous approaches are in use, and there are good reasons for that. Interestingly, there is rarely a "black or white" – combined approaches can be found in many places.

It is the author's hope that this paper can serve two purposes, namely (a) to illustrate the differences in the fundamental principles of the two timing paradigms and their implications for a doctor's consultation, e.g., and (b) to inspire the reader to leverage the proven "grey" approaches from everyday life for solving problems in the VLSI domain.

1 Introduction

Since several decades more or less all reasonably complex logic devices are clocked, which means their timing is based on the synchronous paradigm. This approach is very efficient in both, design and implementation. In parallel to that, a relatively small research community of "asynchronous revolters" has kept on elaborating conceptually elegant methods for designing circuits that operate in a "self-timed" fashion, i.e. without a global clock. The continuous debate on which approach, synchronous or asynchronous, is preferable has been fueled by the increasing visibility of shortcomings of the dominant synchronous approach caused by the proceeding miniaturization of device structures. Recent examples for such deficiencies are power consumption/heat dissipation, and parameter or voltage variations. More often than not, this debate unveils substantial misunderstandings on the differences between asynchronous and synchronous paradigm. It is the aim of this paper to contribute to a clarification here that is comprehensible for non-experts. To this end we will go away from the domain of VLSI systems and look for comparable problems we encounter in everyday life. There we will investigate which solutions – synchronous or asynchronous – we employ there. Beyond (hopefully) providing a good illustration for the key concepts of the approaches, such a comparison may inspire new solutions for VLSI that leverage the wealth of (intuitive) knowledge and experiences of generations that shaped these everyday solutions.

2 The VLSI designer's world

In the VLSI domain the problem is to coordinate some kind of cooperative execution, like shared/distributed processing of a task (like in pipelined or parallel execution units), access to a shared resource, or simply a data exchange. In the latter, e.g., the receiver must capture the data at a point in time when they are actually provided by the sender. Synchronization is required and becomes non-trivial, as soon as the partners operate concurrently, i.e. each partner operates at its own speed, determined by the complexity of its service (which may itself vary with input data or operating mode), communication delays, tolerances and jitter, and, most notably, coordination with other partners. Especially the latter can lead to extremely complex transitional dependences in the speed of operation.

There are two fundamental ways to establish a synchronization: In (the most pronounced form of) *the synchronous paradigm*, all partners are provided with the same notion of time, and a global schedule determines which partner has to perform which activity at which point in time. In fact, all partners receive the same clock signal whose edges define a discrete time base. A partner's schedule can be as simple as "capture your input data with every rising clock edge", or it may be a very involved behavior, determined by a protocol state machine. The important concept here is that each partner performs its activities *based on its local view only*, without caring about the others. The global schedule (circuit design) takes care that those local activities, in their combination, provide the desired overall service. The clock period is usually constant, and each partner has to finish its local task by a given deadline (usually the next active clock edge), where most often being too early does not pose a problem but does not yield a benefit either.

The asynchronous approach is based on an explicit communication between the partners, which is, unlike the actual application data flow that may occur between them as well, just dedicated to the coordination of their activities. This so called handshake is composed of some partner initiating an activity (request) and another one (or more) confirming its completion (acknowledge). For a data exchange, the sender will, e.g., provide the data to the receiver's input, then request the exchange; and the receiver will capture the data and acknowledge their reception. The important point here is that the partners *communicate* to establish the coordination, so there must be suitable channels available for that (the handshake signals)[1]. This communication allows taking care of the partner's state. In fact, the handshake establishes a closed-loop timing control between the partners that adapts the timing of the local activities to the abilities of the respective partner(s). In order to consider the partners' state, one needs to know all partners involved in a cooperative execution. In a multicast communication, e.g., the sender can remove its data only after having received the acknowledges of all intended receivers.

[1] The handshake may be intertwined with the data.

It is often non-trivial to directly identify the completion of a local event which is needed to trigger a request or acknowledge signal. For the simple example of a data exchange, e.g., it is not obvious when exactly the data have arrived at the receiver's input, neither when exactly their capturing by the receiver is finished. The lack of such exact knowledge compromises the otherwise very beneficial closed-loop behavior of the "delay-insensitive" asynchronous approach. In contrast, the asynchronous "bounded delay approach" simply accepts this imperfection and uses a time delay as an indirect measure for completion, thus often saving significant overheads.

3 Synchrony and asynchrony in daily life

As a first real-life example let us look at transportation. Here the service is to have a group of persons moved from location A to B jointly, and synchronization is required to have these persons on the transportation medium upon departure.

The synchronous incarnation of this problem is a train: There is a global schedule known to everyone, which gives a deadline for boarding. The train will depart as scheduled, irrespective of who has boarded, and, as the author knows well from his daily ride to work, without mercy for latecomers. No handshake, no communication. The passengers are acting concurrently in the sense that each of them has his own history of getting up in the morning, not finding his skirt, meeting a friend on the way[2], getting stuck in a traffic jam[3], etc. Therefore they arrive at the train at different times, and they are "synchronized" by the joint departure.

The asynchronous counterpart would be a family returning with the car from a hike. Dad will start driving home with the car, as soon as mom, Tom and Nelly are comfortably sitting in the car. As long as Nelly is still flirting with the incredibly nice boy she just met on the trail, there is no way for dad to start going – even if since days he had insisted on being back home again before 7pm, in order to make it for a beer with his friend Gordon.

Some key differences in the concepts become clearly visible here: For the train it is neither known nor important who will join – the alignment of all passengers' largely different concurrent activities to one closed control loop would cause unmanageable complexity. Here the synchronous approach that abandons all history and just sticks to the global schedule is an essential means of decomposing this enormously complicated system; essentially moving the responsibility to each single passenger.

Even if he wished, this does not work for dad's problem, since he cares to have the whole family on board.

An interesting hybrid approach is the airplane, where the luggage check-in is ruled by a strict static time schedule (synchronous), while then the plane will not depart without any of the checked-in passengers (asynchronous) – at that point the airline knows each single passenger and, "for security reasons",

[2] This is an example of the transitional dependence mentioned above.
[3] This is an example of a shared resource requiring/enforcing synchronization.

cares to have everyone on board. While it was easy for dad to see that the family is complete, the airline companies check completion by means of barcode scanners, accompanied by manual counting through the flight attendants. This fussy procedure nicely illustrates that completion detection can be quite tricky. Interestingly, even in real life "bounded delay approaches" are sometimes being pursued to circumvent completion detection. One (for the author particularly annoying) example is the microwave oven: The user has to enter (and know!) a cooking time instead of the desired temperature, simply because time is cheaper to measure.

Another interesting aspect we can study with these transportation examples is fault tolerance: If Nelly drove away in her new boyfriend's car without saying a word (handshake!), then dad would search the whole surroundings for her instead of driving home. So asynchronous systems tend to deadlock in case of faults. This property is deeply buried in their "wait for all" philosophy imposed by the closed loop. Obviously such a concept will never work for a train. The synchronous approach can easily handle the absence of a partner – it simply does not care. Another way of viewing this is that the asynchronous paradigm has no flexibility with the number of completion messages but is arbitrarily flexible about their arrival times, while in the synchronous case there is no flexibility in time but arbitrary flexibility in the number of completing partners.

However, if the train is late for whatever reason, some of its next track segments will be used by other trains already[4], and passengers will miss their connections. Here the coordinated mission fails, as on that level of hierarchy the underlying global schedule essentially relies on the correct behavior of all partners. So the don't care philosophy fails if there are partners that are mission critical (like the professor in a lecture).

In practice one often finds solutions between these extremes, like an asynchronous approach with time-out (waiting for a colleague to drop by in the office during the afternoon, but at some point then going home when he still does not show up), or a combined synchronous/asynchronous approach. An example for the latter will be presented in the next section.

Before that, let us briefly turn to a different example, namely meeting friends to go for a bike tour. Some 30 years ago (like when the author did such tours) all partners simply agreed on a time and a street corner to meet, and everyone tried to be 5 minutes earlier because he knew the others will be unnecessarily waiting at the street corner if he is late – or even leave without him. Today (like when the author's son is doing such tours), for the same simple purpose of going on a bike tour, there is an incredible amount of communication on cell phone, skype or whatsapp like "Shall we meet at 4 or better at 5?", "Will Andrey also join?", "I will be 20 minutes late, wait for me" or "Decided not to come, don't wait for me", to negotiate for the actual meeting time. Why this change?

[4] To avoid crashes in these cases, sensors and signals are installed to provide the communication infrastructure required for a switch to the asynchronous paradigm at that point.

In earlier times, in the absence of mobile communication, there was no choice but resorting to the common notion of time which was the only (mobile) available relevant infrastructure then[5]. This forced us to use the synchronous paradigm. Nowadays the mobile phones provide the cheap mobile peer-to-peer communication channels required for the asynchronous approach. And for the reasons mentioned earlier this indeed seems to be the preferable approach for reasonably small groups.

4 The king of timing optimization

An extremely enlightening construction is the doctor's appointment. It starts with an "asynchronous" call at the doctor's office[6], upon which a date and time (synchronous!) are assigned for a consultation. However, when arriving synchronously, e.g. before the deadline, at the doctor's office, the patient waits until being called (handshake), and this can take a substantial amount of time. This can be regarded as a locally asynchronous globally synchronous approach, and although the abbreviation "LAGS" does not sound as nice as GALS[7], this principle is surprisingly often encountered in our everyday life. It can be interpreted as follows: We use a synchronous "appointment" approach to roughly schedule our daily agendas – that would simply be too complicated to be done asynchronously, "on demand". In a perfect world that alone would be sufficient, but in reality we cannot precisely meet all deadlines, as we have, e.g., to accommodate asynchronous requests as well (like accidentally meeting an old friend on the way), so we use an asynchronous approach for fine-tuning. This also relieves us from being too pessimistic about our deadlines, thus saving waiting time and increasing performance.

This brings up the issue of *waiting times* in general. Fundamentally, coordination of activities among concurrently operating partners implies some form of waiting for each other. In a synchronous setting, the rendezvous time must be chosen such that each partner (we care about) can be in time even under the worst circumstances. That is why we usually plan for some safety margin when going to an important date like our wedding – the idea is to make sure we make the deadline even if we get caught in a traffic jam or the car breaks[8]. Clearly, this causes unnecessary waiting in all but the worst case situations, for all partners. It is the responsibility of each partner to decide which margin he wants to have. Each partner's waiting time is then the difference between the considered worst case and the actual case. Under good conditions all partners will thus be early and wait. The train will normally not leave the station before schedule even if all passengers have already boarded.

[5] Note how immensely important global time as a common infrastructure is for our society, exactly for this type of synchronous coordination

[6] Synchronous doesn't make sense here unless one regularly gets ill

[7] "Globally Asynchronous Locally Synchronous", an approach recently used in VLSI

[8] That's probably the only reasonable argument why airlines urge passengers to be at the airport 2 hours before departure

In an asynchronous setting the completion/arrival of the slowest/latest partner determines the start of the joint activity. The slowest partner therefore does not have to wait, while all others wait. Recall that in the synchronous setting the rendezvous time was determined such that even the slowest partner can reach it under worst case conditions. Therefore, in the average case, even the slowest partner will arrive earlier than that, and all partners save waiting time. That is why asynchronous systems are said to have higher performance, namely average case performance, than synchronous ones with their worst-case performance.

However, why then do we notoriously have to wait for hours in the doctor's waiting room? This is because the doctor abuses this scheme: In the synchronous domain, he gives you a date for which he knows that he will never make it. This forces you to arrive at a time when he is not ready for sure, and at that time he switches to the asynchronous mode, which means you will always be the waiting partner and he will never wait (or have you ever seen the doctor welcome you with the words "I have been waiting for you already!"). Through that unfair trick the doctor increases your waiting time beyond the one you already had in the synchronous domain – so this system is not beneficial for the patient[9]. The doctor is using it because he considers himself a precious resource whose waiting time has thus to be minimized. Admittedly, there are no good alternatives: A purely asynchronous approach (visit the doctor on demand, scheduling by priority), is necessarily practiced in emergency stations, but has proven to cause even higher waiting times in practice for the lower priority cases. And a purely synchronous system with conservative estimation of the time per patient would severely decrease the doctor's throughput, and also disallow him to take care of sporadic emergency cases.

Actually, the doctor uses another trick to avoid waiting times on his side: buffering. Most often there is more than one patient waiting patiently, so should the doctor be faster with processing one patient, there is already a next one waiting (who of course also got a ridiculously optimistic rendezvouz time). This nicely illustrates the function of a buffer (waiting room) for compensating fluctuations in the processing time. Such a FIFO buffer – sometimes even signified by the fact that you literally get a number – is to the benefit of all patients, as it increases the doctor's throughput. However, it is yet another unfair trick of doctors to introduce some dubious kind of priority (for emergency cases like his brother-in-law's cousin) sometimes.

Obviously there is no point in using the LAGS scheme for starting a rock concert, where the sheer number of participants clearly calls for a synchronous solution. And on the family hike dad won't make friends by always looking at the watch and telling "Hurry up, we must be back at the car at 5pm!" or not starting the ride back at 4:45pm even if everyone is sitting in the car – this is clearly an asynchronous case. However, even the friends going on a bike tour from the example above (at least the classic version) employ a LAGS solution by agreeing on a time to meet and then wait until everyone (they care about) is here. In fact, we use this scheme many times a day. We use the synchronous schedule

[9] Interestingly this word's interpretation as an adjective becomes symptomatic here

for planning our day (you simply cannot plan if you give all partners arbitrary freedom), i.e. scheduling ourselves as a kind of shared resource, but then add flexibility by switching to the asynchronous mode, accounting for the fact that the interactions between the myriads of parallel and interacting processes on our planet are so complicated in detail that we call them unpredictable.

Note that in LAGS we often need to assign an ID to each of the partners. As an example, imagine the doctor using a schedule of the patients, and the patients establishing a FIFO order by pulling numbers as they arrive. The patients will (even in the absence of dubious priority) not know in which order to enter the doctor's room, and the doctor will clearly call them by name. In technical terms, when the jitter of arrival gets larger than the synchronous period (planned processing time per patient), out-of-order arrival is possible and identifiers need to be introduced. This is neither necessary in a purely synchronous approach ("Take the 8:30 train") nor in a purely asynchronous approach with strict FIFO ordering (pulling a number at a government office).

5 Conclusion

We have discussed real-life scenarios where either pure synchronous or pure asynchronous timing is beneficially used, and these have essentially confirmed the experiences from the VLSI domain. However, it turned out that in our daily lives we often use a locally asynchronous globally synchronous (LAGS) approach, which is quite in contrast to the GALS approach found in the VLSI domain. What VLSI designers might take away from this observation is that (a) the large complexity found on system level might be much easier to handle in a (coarse-grained) synchronous approach than with interacting asynchronous control loops, while (b) using continuous time for fine-tuning the local timing might be more appropriate than using ridiculously high clock frequencies to avoid losing performance when performing that task in the synchronous domain.

Admittedly, these are just relatively vague ideas, and some of the parallels drawn may even turn out wrong upon closer investigation. It is not the author's claim that this is a scientific paper (after all it is lacking related work and references), but it is hoped that this different view at the problem of VLSI timing may inspire new solutions. At minimum, however, the paper shall provide something to contemplate about during the next hours the reader spends in the doctor's waiting room.