Globally Asynchronous Locally Synchronous FPGA Architectures

Andrew Royal and Peter Y. K. Cheung

Department of Electrical & Electronic Engineering,
Imperial College, London, UK
{a.royal, p.cheung}@imperial.ac.uk

Abstract. Globally Asynchronous Locally Synchronous (GALS) Systems have provoked renewed interest over recent years as they have the potential to combine the benefits of asynchronous and synchronous design paradigms. It has been applied to ASICs, but not yet applied to FPGAs. In this paper we propose applying GALS techniques to FPGAs in order to overcome the limitation on timing imposed by slow routing.

1 Introduction

Most Field Programmable Gate Arrays (FPGAs) are designed with one or more global clocks. FPGAs with multiple clock domains must provide some mechanism for synchronising data passing between them, which will increase latency and be prone to metastability.

In addition a signal routed over a great distance on an FPGA must pass through many long wires, transistor switches and buffers. Consequently these long connections usually prove to exhibit the longest delays in the whole device. If transmission is to occur over one clock cycle or even a fraction of a clock cycle, this routing delay will limit the clock frequency.

One solution is to pipeline the routing, as used in [1]. A potential problem of this is that the number of clock cycles allocated to the routing must be determined at the routing stage and may impact upon any cycle allocation assumed at the circuit design stage. Also, concurrent data travelling along different routing paths need to contain exactly the same number of pipeline stages or data will arrive on different cycles.

We could also use asynchronous circuits. Rather than assume that data takes a clock period to pass through a pipeline stage, asynchronous circuits use handshake protocols to indicate when each stage has data to pass to the next stage. This allows each stage to be independently timed. Were we to use asynchronous routing for FPGAs, the speed of the rest of the circuit would no longer be limited the speed of the routing. As they have no clocks, there is no need for multiple clock domains and no problem with synchronisation.

Asynchronous FPGAs have already been proposed for prototyping asynchronous circuits. However, the drawbacks of asynchronous circuits deter designers from using them in preference to synchronous arrays. A compromise is

to use a Globally Asynchronous Locally Synchronous (GALS) system. In such a system synchronous modules with locally generated clocks are used, with asynchronous connections between them. Hence we retain the advantages of synchronous circuits, but can also exploit the advantages of asynchronous routing. This technique has not previously been applied to FPGAs.

In this paper we propose adding asynchronous routing to a synchronous FPGA. In section 2 a brief overview of asynchronous communication, asynchronous FPGAs and Globally Asynchronous Locally Synchronous Systems is given. We go on to describe an architecture for applying this work to synchronous FPGAs in section 3 and assess its viability in section 4. Finally, in section 5 we describe our future work into improving this architecture.

2 Background and Related Work

2.1 Asynchronous Systems

In this paper, we use the term "Asynchronous" to refer to circuits designed without clocks, also known as Self-Timed circuits, where the clock is replaced by handshaking signals. In a synchronous system, all blocks are assumed to have finished computation when a clock edge arrives. The blocks in Self-timed systems independently indicate completion by sending out a request and only proceed when that request has been acknowledged. Blocks only operate as needed, there is little redundant processing. Also, a self-timed system is very composable as blocks can be individually optimised and timing of one block does not affect another. We wish to exploit this feature for our FPGA architecture.

Data is transmitted using a bundled data protocol [2]. This means that data signals are grouped into a bundle and the validity of the whole bundle is indicated by a single request/acknowledge handshake pair. The bundling constraint states that a request must arrive after the corresponding data, so data can be latched safely, so the delay of request lines often needs tuning.

Asynchronous designs require some circuit components which are rarely used in synchronous design. Ebergen [3] showed many of the circuit elements required to build delay insensitive circuits. We require some of these components for building our architecture. A C-element is a component which fires a transition on its output when each of its inputs have made transitions in the same direction, it is effectively an AND for events. An isochronic fork is simply a signal which branches out to two destinations, where the delay on the wires is negligible compared to the delays of the gates they are driving and so can be assumed to arrive at the same instant. A mutual exclusion element (often ME element or MUTEX) is used to arbitrate between requests. ME elements may go metastable if the requests arrive close to each other, but the Seitz arbiter [4] is a mutual exclusion element designed such that any metastability is internal and not propagated to its outputs. Finally, Micropipelines [2] are an asynchronous equivalent of synchronous pipelines, where the registers are controlled by request/acknowledge signals rather than a clock.

2.2 Asynchronous FPGAs

There have been several attempts to implement asynchronous circuits on FPGAs designed for synchronous circuits [5], [6]. The main problems with this are that synchronous FPGAs are not designed to be hazard free and do not provide many of the components commonly used in asynchronous design.

There have also been FPGAs designed specifically for implementing asynchronous circuits. MONTAGE [7] is an extension of the synchronous TRIPTYCH architecture [8]. Fast feedback in functional units allows asynchronous-specific components to be built and specific arbiter blocks are also provided. Routing is organised to allow isochronic forks. PGA-STC [6] is similar to MONTAGE, but also includes a reconfigurable delay line which can be used in the implementation of a bundled data protocol. The delay uses a ring coupled oscillator which is unfortunately very large and power consuming. STACC [9], [10] is loosely based on Sutherland's micropipeline design [2]. The data array can be like that of any synchronous FPGA, but the clock is replaced by control signals from a timing array which consists of a micropipeline-like structure.

Asynchronous FPGAs are not widely used. They are fraught with problems with hazards, critical races and metastability. Asynchronous circuits are hard to design and tools have only recently begun to reach maturity. Asynchronous buses are difficult to construct [11]. We also find that the additional completion detection circuitry required takes considerable area and power and slows the circuit down. Hence we propose a Globally Asynchronous Locally Synchronous FPGA as a compromise between synchronous and asynchronous styles.

2.3 Globally Asynchronous Locally Synchronous (GALS) Systems

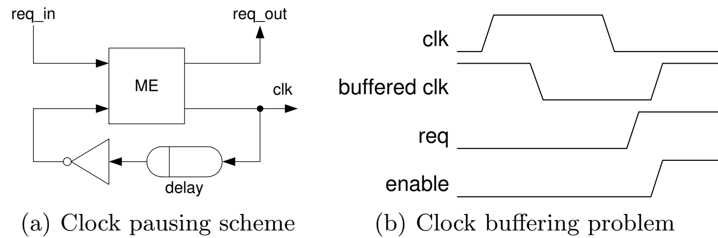
Globally Asynchronous Locally Synchronous (GALS) Systems combine the benefits of synchronous and asynchronous systems. Modules can be designed like modules in a globally synchronous design, using the same tools and methodologies. Each block is independently clocked, which helps to alleviate clock skew. Connections between the synchronous blocks are asynchronous.

Early work on GALS systems ([12] and [13]) introduced clock stretching or pausing. When data enters a synchronous system from an asynchronous environment, registers at the input are prone to metastability. To avoid this, the arrival of data is indicated by an asynchronous handshaking protocol. When data arrives, the locally generated clock is paused: in practice the rising edge of the clock is delayed. Once data has safely arrived, the clock can be released so data is latched with zero probability of metastability on the datapath. [14] used ME elements to arbitrate between the clock and incoming requests, which helped to eliminate metastability. [15] introduced asynchronous wrappers, standard components which can be placed around synchronous modules to provide the handshake signals and make them GALS modules.

The local clock generator is constructed from and inverter and a delay line, similar to an inverter ring oscillator. The problem with using inverters alone as a delay line is that it is difficult to accurately tune the clock period as process

variations and temperature affect the delay. Hence accurate delay lines have been developed which are capable of maintaining a stable clock frequency [16], [17]. These use a global reference clock for calibration. The former can use either standard cells or full custom blocks for the tunable delay and was shown to maintain a frequency within 1% of the chosen value.

To make the clock pausable, an ME element is added to the ring as shown in figure 1(a). This arbitrates between the rising edge of the clock and an incoming request. Hence the clock is prevented from rising as the input registers are being enabled by the request and metastability is prevented. For each bundle of data a port controller, request and ME element is required. Only when all of the ME elements have been locked out by the clock is the rising clock edge permitted to occur.



(a) Clock pausing scheme (b) Clock buffering problem

Fig. 1. Pausable clock

Port controllers are required to generate and accept handshaking signals at the inputs and outputs of modules. These port controllers are asynchronous state machines, which are similar to on inputs rather than a clock.

A problem with clock pausing is that the clock is delayed as it is distributed across the clock domain, but the clock must be paused at its source. When the clock releases the ME elements there may still be a clock edge in the buffer tree. Hence it is possible that registers will be enabled as the clock rises, as shown in figure 1(b). But while the source clock is high, ME elements will remain locked so for this phase of the cycle no requests are permitted. For this reason, we must ensure that the delay of the clock buffer is shorter than the duration of the high phase of the clock. Limiting this delay limits the size of the clock tree, hence defining the size of GALS blocks.

3 System architecture

We propose converting a conventional, synchronous FPGA into a GALS system. To do this we partition the FPGA into smaller blocks of FPGA cells. Within one of these blocks, the local connections are synchronous to a local clock for that block and hence the block resembles the original FPGA. However, longer communication channels between blocks become asynchronous.

Figure 2 shows the proposed architecture in place around a block of FPGA cells. Note in particular the dividing line between the synchronous and asynchronous domains. All of the FPGA cells are in an isolated block above the line in the synchronous domain. Internally, the FPGA block could resemble any synchronous FPGA as it is hidden from the rest of the system. Below the line there is an asynchronous wrapper: this interfaces between the synchronous and asynchronous domains. Outside the asynchronous wrapper blocks are connected together using asynchronous routing. All of these blocks are explained in detail in the following section.

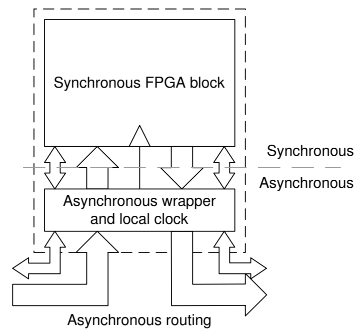


Fig. 2. FPGA block with asynchronous routing

3.1 Additional Synchronous FPGA Fabric

Figure 3 shows the boundary of a synchronous FPGA block. The FPGA block could contain any type of FPGA cell and its associated routing. There could be a number of different levels of routing within the block, for example nearest neighbour routing or fast interconnect spanning the block, as there are within a conventional FPGA. At the boundary, we can use the same routing schemes to connect to the asynchronous interface.

In this instance only one input port and one output port is shown for clarity, though it would be possible to design a system in which each FPGA block has several input and output ports to allow communication with a number of different FPGA blocks. As well as the data itself, the asynchronous ports needs to communicate with the synchronous to indicate when data is valid. To accomplish this, a wire for each port spans the routing leaving the block. For nearest neighbour or other unidirectional connections, one of the inputs to each block is allowed to connect to the data valid wire for each input port. Similarly, one of the output wires from each block may connect to the data valid wire of the output port. All other inputs and outputs from the FPGA cells can be used for data transfer.

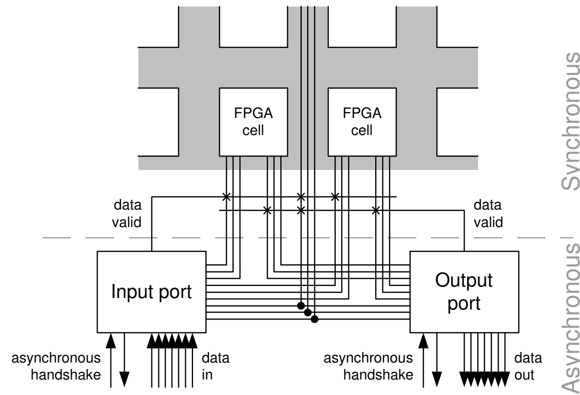


Fig. 3. Interconnect between FPGA cells and GALS ports

We work under the assumption that should we use part of an FPGA cell to create a data valid signal, that signal could be mapped to any of the inputs or any output as appropriate. In the case of a look-up table (LUT) based FPGA this can easily be done. Hence by only allowing one wire per cell to map to an input data valid signal and one for the corresponding output signal, we save on switching with little impact on flexibility.

Similarly, we also allow longer routing channels to be connected to the data valid signals. Again, in order to cut down on the number of switches used we only allow a fraction of the wires in the channel to be connected to the data valid signals. However, as long routing wires can usually be configured with data flow in either direction, we allow the chosen wires to be connected to both the input and output data valid signals. Here, we can expect some loss in flexibility as each routing wire could be connected to a completely different area of the FPGA block. But as each port can only have a single data valid signal, we should not need to allow many connections to the routing. In many cases it will be necessary to route to an FPGA cell and perform some logic function to merge several data valid signals into a single signal, which could be done in a boundary FPGA cell with a nearest-neighbour connection to the port's signals.

In figure 3, data is shown to enter the ports. This is merely a grouping of input wires and output wires, there is no need for any processing or even latching as that is handled in the synchronous part of the FPGA block. However, handshake signals accompany each "bundle" of data and so inside the synchronous FPGA block the data valid signal corresponding to the handshake must control the same data.

To complete this part of the system, we need to place a few requirements on any circuit mapped to the FPGA block. As discussed above, data valid signals need to be generated or processed. Data must leave the module with corresponding data valid signal and be latched when incoming data is valid. Any data signals

also need to be routed to the boundary where they will leave the FPGA block accompanied by the handshake. For our implementation of the output port, we require that the data valid signal be "differential", i.e. it indicates valid data by changing its value and that no valid data is present by remaining at the same value.

3.2 Asynchronous Wrapper

The asynchronous wrapper shown in figure 2 is formed of 2 components: a local clock generator and port controllers. These components were described in more detail in section 2.3. For our implementation we use a four phase bundled data protocol. The interface between the synchronous and asynchronous domains is facilitated by making the synchronous signals differential, so an event is created whenever a signal changes. Our port controllers have been designed under the assumption that the synchronous block produces these differential signals and so they are a requirement of the circuit mapped to the FPGA block.

Note that we require a separate clock tree for each locally synchronous block. Clearly using the global trees featured in current FPGAs would be wasteful, hence it is preferable to use a dedicated local clock buffer. As mentioned in section 2.3, to prevent the size and delay of the clock buffers from becoming too large a limit of the size of the FPGA blocks within each wrapper is imposed.

3.3 Asynchronous routing

The four phase bundled data transmission protocol continues into our routing. Not only must the data be routed, but also the corresponding handshakes. Furthermore, the bundling constraint must by maintained by delaying the request lines sufficiently that they always arrive after the corresponding data. When data arrives at a register, that register must be disabled so it will not go metastable if the rising edge of the clock arrives at the same instant. Once the register has won control of the ME element ahead of the clock, the register can be enabled and the ME element released.

Transfer of data is facilitated by inserting micropipelines into the routing. We exclusively use unidirectional wires to make point-to-point connections rather than using buses. The configuration in the routing is greatly simplified and in particular if micropipeline stages are used they need only operate in a single direction. The overall routing scheme is shown in figure 4. We have no long lines as long, slow lines are what we are trying to avoid. Instead, all long connections are made through a series of block-to-block connections. Each wire entering the FPGA block can either be routed to an input port or bypass the block completely. A connection between any two modules must pass through at least one micropipeline, which helps reduce the time each module remains paused and eliminate deadlock. Connections between rows must pass through at least two micropipeline stages, which adds a little latency.

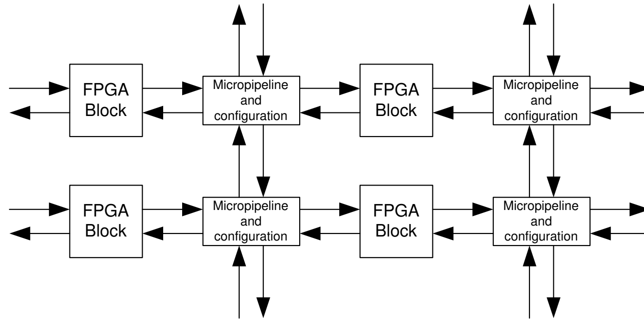


Fig. 4. Routing Scheme

4 Analysis of architecture

This system confers several advantages. Our aims of metastability free independent clock domains and synchronous blocks whose timings are independent of the routing are met. The asynchrony of the routing allows data tokens to take an arbitrary number of clock cycles to reach their destinations. Indeed, as the source and destination are not in the same clock domain the number of clock cycles which pass in each domain as the data is transmitted can be expected to be different. Additionally, the FPGA system becomes more composable, i.e. the asynchrony of blocks makes it easy to design the component blocks independently without too much concern on how they will operate together as a system. It is no longer timing but the handshaking protocol which ensures data integrity. Independent design also allows the blocks to be independently optimised. As well as preventing metastability, clock pausing can be used to force a module to wait for data to arrive before proceeding, without wasting energy on redundant clocking.

However, the scheme does have several disadvantages. Firstly, we have now forced a separation of local routing signals, which are contained within the synchronous FPGA block, and global routing signals. This reduces the flexibility of the FPGA as a router is no longer able to use long routing tracks for local routing or join shorter routing tracks to make longer connections.

Secondly, we force additional constraints on placement. Any circuit which is too big to fit into a single FPGA block will necessarily need to be partitioned across several different blocks which will be in different clock domains. In some cases this may be inappropriate. For example feedback loops should ideally be contained on a single block due to the latency incurred. Due to the differing data rates, partitioning which results in data being transferred between blocks every cycle should be avoided.

To interface with the port controllers the synchronous block may need some additional circuitry. Some form of synchronous handshake is required to indicate

to the output controller when data is valid and to accept data from the input port when incoming valid data is present. In our implementation we require differential data valid signals. If the design is contained within a single block, the external ports may already include these signals. However, this need not necessarily be the case and if a design needs to be partitioned the required signals will almost certainly not be present at the block boundary and therefore need to be inserted. This may prove to be problematic as the timing of such signals requires not only the circuit information but also some knowledge of the pattern of the data.

Finally, as the size and complexity of the system implemented grows, it may become necessary to partition across many blocks. In this case, data being sent from a number of blocks to a single destination may be required to arrive at the same time. To make allowances for this, rendezvous elements are needed at each input port. These elements must be fully configurable to allow several possible combinations of data channels or to allow a bypass when rendezvous is not required. This will however move the system away from one in which the only configurable components are within the synchronous blocks.

5 Conclusion and Future work

We have presented an extension to existing FPGA architecture with the potential to prevent long routing delays from dominating FPGA performance. However, the solution is not without its drawbacks.

To address some of these problems, some alternative architecture may be required. It has already been mentioned that configurable rendezvous elements are required at the input ports which adds configuration required in the routing. It may also be possible to join local clock trees to effectively combine two smaller blocks into a larger block under a single clock domain, though great care must be taken to maintain the balance of the tree. If this is possible, it may also be possible to join all clock trees and allow a globally synchronous mode to be retained alongside the GALS mode. Request lines need some tuning to force requests to arrive after the data and meet the bundling constraint, but alternatively we can use a dual rail protocol to provide delay insensitive routing. Though we currently use a four phase protocol, a two phase protocol may have some advantages.

We have yet to fully verify the scheme and prove its effectiveness. The difficulty lies in simulation as we need to concurrently simulate synchronous, asynchronous and FPGA components. Furthermore we have yet to extensively investigate how circuits may map to the system.

References

1. Mirsky, E., DeHon, A.: MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In: Proceedings of the IEEE Symposium in Field-Programmable Custom Computing Machines. (1996) 157–166

2. Sutherland, I.E.: Micropipelines. Communications of the ACM (1987) 720–738
3. Ebergen, C.: A Formal Approach to Designing Delay-Insensitive Circuits. Distributed Computing **5** (1988) 107–119
4. Seitz, C.L.: System timing. In Mead, C.A., Conway, L.A., eds.: Introduction to VLSI Systems. Addison-Wesley (1980)
5. Brunvand, E.: Using FPGAs to Implement Self-Timed Systems. J. VLSI Signal Process. **6** (1993) 173–190
6. Maheswaran, K.: Implementing Self-Timed Circuits in Field Programmable Gate Arrays. Master's thesis, University Of California Davis (1995)
7. Hauck, S., Burns, S., Borriello, G., Ebeling, C.: An (fpga) for Implementing Asynchronous Circuits. In: IEEE Design & Test of Computers. Volume 11. (1994) 60–69
8. Borriello, G., Ebeling, C., Hauck, S., Burns, S.: The Triptych FPGA Architecture. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **3** (1995) 491–501
9. Payne, R.E.: Self-Timed FPGA Systems. In: Proceedings of the 5th International Workshop on Field Programmable Logic and Applications. (1995)
10. Payne, R.: Self-Timed Field Programmable Gate Array Architectures. PhD thesis, University of Edinburgh (1997)
11. Molina, P.: The Design of a Delay-Insensitive Bus Architecture using Handshake Circuits. PhD thesis, Imperial College (1997)
12. Chapiro, D.M.: Globally Asynchronous Locally Synchronous Systems. PhD thesis, Stanford University (1984)
13. Pěchouček, M.: Anomalous response times of input sychronisers. IEEE Transactions on Computers **C-25** (1976) 133–139
14. Yun, K.Y., Donohue, R.P.: Pausible clocking: A first step toward heterogeneous systems. In: Proceedings of the International Conference on VLSI in Computers and Processors. (1996) 118–123
15. Bormann, D.S., Cheung, P.Y.K.: Asynchronous wrapper for heterogeneous systems. In: Proceedings of the International Conference on Computer Design (ICCD). (1997) 307–314
16. Moore, S.W., Taylor, G.S., Cunningham, P.A., Mullins, R.D., Robinson, P.: Self-calibrating clocks for globally asynchronous locally synchronous systems. In: Proceedings of the International Conference on Computer Design (ICCD). (2000) 37–78
17. Olsson, T., Nilsson, P., Meincke, T., Hemam, A., Tokelson, M.: A digitally controlled low-power clock multiplier for globally asynchronous locally synchronous designs. In: The IEEE International Symposium on Circuits and Systems (ISCAS). Volume 3. (2000) 13–16