Asynchronous Interfaces for IOPT-Flow to Support GALS Systems

Joao Almeida*, Filipe Moutinho ', and Rogerio C'ampos-Rebelo * NOVA University Lisbon, NOVA School of Science and Technology, Monte de Caparica, Portugal
tUNINOVA, Center of Technology and Systems, Monte de Caparica, Portugal ^Polytechnic Institute of Beja, School of Technology and Management, Beja, Portugal jg.almeida@campus.fct.unl.pt, fcm@fct.unl.pt, rcr@uninova.pt

Abstract

Throughout the course of time, distributing a global clock signal over a synchronous circuit has become a demanding task as a result of the broadening size and complexity of modern circuits. Globally Asynchronous Locally Synchronous (GALS) systems emerge as a solution to the laborious task of distributing a global clock over a large circuit, through the partitioning of said circuit into smaller, and therefore, more manageable blocks. The DS-Pnet (Dataflow, Signals and Petri nets) modelling language and its associated framework IOPT-Flow focus on supporting the development of cyber-physical systems, however, they may be a strong push to the development of GALS systems, through their multiple available tools that comprise a graphical editor, a simulator and automatic code generation tools, namely a VHDL (VHSIC Hardware Description Language) code generator. In order to facilitate the implementation of said GALS system in the IOPT-Flow framework, some components were created, these work together to form asynchronous interfaces that are a crucial element to any GALS system, thus providing options to designers that intent to develop a GALS system utilizing the IOPT-Flow framework.

I. Introduction

Technology, and by extension electronics, has suffered tremendous growth in the preceding years, as a result synchronous circuits are abundant, introducing themselves in our day-to-day lives. However, these synchronous circuits have become much larger and keep growing in size, in order to meet increasing demands. For a synchronous system, the larger the circuit the harder it is to distribute a global clock signal with a low clock skew. There are, however, two alternatives that circumvent this issue. First, asynchronous circuits that do not possess a global clock signal, yet are not designer friendly due to a lack of tools, and thus, are mainly considered for peripherals. The other alternative is Globally Asynchronous Locally Synchronous (GALS) systems, these systems are composed of smaller synchronous blocks that are capable of communicating with one another asynchronously. Each block has its own clock generator, meaning, that the task of distributing the clock signal may be simplified.

The DS-Pnet (Dataflow, Signals and Petri nets) modelling formalism and its associated framework IOPT-Flow [1] [2] emerge to support and accelerate the development of cyberphysical systems, by offering a set of tools that include a graphical editor, a simulator and code generation tools, that are capable of generating C, VHDL (VHSIC Hardware Description Language) and JavaScript code. In order to facilitate the development of GALS systems in the IOPT-Flow framework, some components were created. These components collaborate to form asynchronous interfaces, which are a crucial element to any GALS system, since they enable communication between blocks. In this paper, these components will be explained, discussed and validated through simulation. These components may be found at http://gres.uninova.pt/iopt-flow/.

Further down in this paper, three main sections will be presented, related work, contribution and validation. In section II, related work, two subsections may be distinguished, of which, the first addresses GALS systems, with emphasis on the communication protocols they employ, while the second focuses on the DS-Pnet modelling formalism and the IOPT- Flow framework. In section III, contribution, the components created in IOPT-Flow, as well as their respective behaviours and mechanisms, are explained in detail. Lastly, in section IV, validation, a comparison is made between the simulation performed in the IOPT-Flow simulator and the simulation performed in the Xilinx ISE.

II. Related Work

A. GALS Systems

A Globally Asynchronous Locally Synchronous (GALS) System is characterized by having multiple synchronous blocks that are capable of interacting with one another through an asynchronous environment. Each block is locally synchronous only to its own clock generator, these clock generators can be completely independent and may have nominally different frequencies. Having smaller blocks grants a GALS system the possibility to fine-tune its clock signal in order to achieve smaller clock skews and significant power savings [3]. However, the main advantage of GALS is its modular design, that is to say, each block may easily be replaced or reused according to the options available to the designer [4] [5].

In literature, two methods for communication in a GALS system may be commonly found, the first is the use of an asynchronous wrapper [6] [7] that envelops the block and employs handshake protocols to exchange data with other blocks, and the second is the use of a FIFO buffer for each communication channel [8] [9], each block communicates with the buffer as opposed to directly with one another, thus bypassing any synchronization issues. The latter method has become dominant over the former throughout the years.

In order to implement the asynchronous wrapper method, each data channel requires an output port to be added to the transmitter block and an input port to be added to the receiver block. As many ports as necessary may be added, however, communication introduces latency and for that reason should be kept to a minimum. As mentioned before, these ports communicate with each other through handshake protocols, these utilize pairs of Request/Acknowledge signals that accompany the data signal in order to accomplish safe data transfer. Furthermore, these protocols are categorized according to the amount of events required to transmit data, they may be either 2-phase or 4-phase. Additionally, signals may communicate through either push channels or pull channels [10].

In push channels, the transmitter block asserts both the request signal and the data signal in order to signal its intent to transmit data, then, upon receiving this signal, the receiver will retrieve said data and respond by asserting the acknowledge signal. In pull channels the opposite happens, the receiver block is the one asserting the request signal in order to request data, when possible the transmitter will respond by asserting both the data signal and the acknowledge signal. Pull channels are less common in literature and will not be further addressed in this paper.

For a 2-phase handshake protocol only two events are exchanged. The ports that employ this protocol are capable of reading transitions in the signals. First, the transmitter port will assert the data signal and then switch the value of the request signal, either from low to high or from high to low, the initial state of the signals is irrelevant for this scheme. The receiver port is then capable of detecting a transition in the request signal, it will retrieve data and invert the acknowledge signal when ready for a new data transfer.
For a 4-phase handshake protocol four events are exchanged. The initial values of the request and acknowledge signals are now relevant and initially set to low, although other initial values could be implemented. The transmitter will first assert its request signal, the receiver will respond by asserting its acknowledge signal, then, the request signal will lower, and finally, the acknowledge signal will lower concluding the data transfer cycle. Additionally, multiple schemes may be distinguished accordingly to when data is valid, namely, early, late or broad schemes. For the early scheme, data is valid from the moment the request signal being is until the moment the acknowledge signal being asserted. For the late scheme, data is valid between the request signal being lowered until the acknowledge signal is lowered. Lastly, the broad scheme encompasses both of the previous schemes, in which, data is valid from the request signal being asserted until the acknowledge signal being lowered.
As for the FIFO-based GALS systems, different architectures exist and differentiate themselves by the approaches they take in order to interact with the FIFO buffer. One solution is controlling the clock generator in order to enable safe communication between the blocks, three different clocking schemes are commonly found in literature, pausible clocking scheme, stretchable clocking scheme and data driven clocking scheme. The second solution is to add synchronizers, these synchronizers are usually composed of a series of flip-flops that reduce metastability as data passes through. The third and last solution is to take advantage of known bounds of each blocks frequency to determine in which cycles it is safe to transmit data. While these architectures may also be implemented with the asynchronous wrapper method, the most common implementation is the clock control method.

B. DS-Pnet and IOPT-Flow

Cyber-physical systems have a strong presence in todays world. In order to support the development of these systems some Petri net based tools have emerged. However, they do not offer good support to solve data-driven problems. And so, the DS-Pnet (Dataflow, signals and Petri nets) modelling formalism introduces data-flow nodes to address such issues.
Due to their roots in Petri nets, DS-Pnet models are also graphical. The two main components are the Petri net nodes that handle the reactive portion of the system and the data-flow nodes that handle mathematical expressions to calculate new values. Additional elements are present, such as read-arcs that link components and input/output signals/events that interact with the physical world and other blocks. An example of a DS-Pnet model is illustrated in Figure 1, in it the following components may be seen:

- Three Petri net places, represented by yellow circles, they depict the systems state by the marks they hold;
- Three Petri net transitions, represented by cyan squares, they consume and create marks on connected Petri net places, this is, they are responsible for advancing the systems state;
- One input signal, represented by the green circle, it depicts a signal entering the system;
- Three input events, represented by green diamonds, they depict events entering the system;
- Two output signals, represented by red circles, they depict signals exiting the system;
- One data-flow operation, represented by a grey trapezoid, operations are responsible for handling complex data manipulation;
- Petri net arcs, represented by solid black arrows, they connect Petri net elements;
- Read-arcs, represented by dotted blue arrows, they connect all other elements.

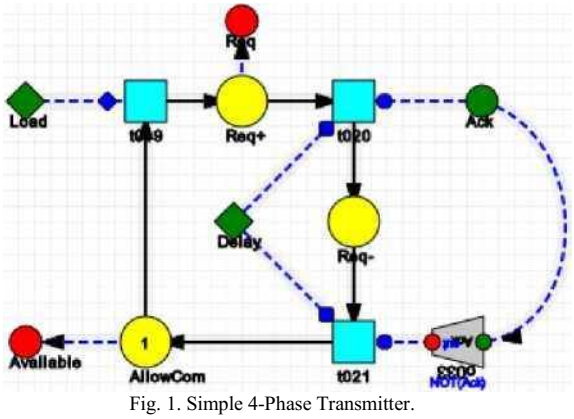

Fig. 1. Simple 4-Phase Transmitter.

While not present in Figure 1, output events exist, they are represented by red diamonds and depict events exiting the system. To avoid confusion, events and signals are similar but not the same, signals have a lasting state where events are active during for a single moment.
To support the DS-Pnet modelling language, surfaces the IOPT-Flow modelling framework. It is web-based and encompasses multiple tools, namely, a graphical editor, a simulator and automatic code generation tools. Upon reaching the website the editor is readily present, to the left a menu displays all of the editors functionalities, the components and access to other tools. Furthermore, whenever a component is selected, a window will appear on the right that enables editing the properties of said component. The simulator is used to debug and simulate the model. It holds some functionalities such as the ability to control simulation speed and to display wave graphs. Lastly, the automatic code generation tools are capable of exporting C, VHDL and JavaScript code.

III. Contribution

In order to establish communication through an asynchronous environment an interface is required. To this end, some components were created, utilizing the IOPT-Flow framework. These asynchronous interfaces are composed of a transmitterreceiver pair, that correspond to the ports of an asynchronous wrapper. Information is carried out from the transmitter to the receiver and always in this direction, in case information needs to be sent in the other direction, another transmitter-receiver pair should be added. As previously mentioned, these ports communicate through a handshake protocol.

Some of the created components transmit data while others merely transmit events. While the components capable of transmitting data will have more use, some systems may not require for data to be transmitted and so simpler interfaces were created, a block may simply intend to notify the other that an event has occurred.

The components may be distinguished in five different categories. Each category has four components, a transmitter and a receiver for both 2-phase and 4-phase handshake protocol. A brief summary of each will now be presented:

- Simple: This is the simpler interface, it is only capable of transmitting events;
- SimpleBuffer: This interface is similar to the previous one with the addition of being capable of holding multiple events at a time, a transmitter may hold a set amount of events to be transmitted and a receiver may hold a set amount of transmitted events yet to be read by the receivers block.
- BurstBuffer: This interface continues to build upon its predecessor. It is still capable of holding and transmitting events, however, it transmits all the events currently held at once. Events are loaded one at a time onto the transmitter, which will then transmit all the events in its buffer through a data channel, the receiver receives all the events that are then read by the block one at a time.
- Data: This is a simple interface that is capable of transmitting data;
- DataBuffer: This interface is capable of transmitting data and holding up to 3 data values, either in the transmitter or receiver.

Now, a more in depth description of each component will ensue, note that not all components will be explained in detail since they hold many similarities. Furthermore, a delay event will be present in every component, which is used to artificially simulate delay in communication. This event should be removed before implementation and will not be further talked about below.

A. Simple

Figure 1 depicts the simple 4-phase transmitter, in its initial state. The initial state is held until a load event occurs. Once it does, the transition it is connected to will fire, consuming the mark in AllowCom and creating a mark in Req+. The models output signals are connected to Petri net places and will reflect their values. For that reason, the available signal will go from high to low since AllowCom has become empty, likewise, Req will go from low to high. Once Ack is high the corresponding transition will fire consuming Req+ and creating a mark on Req- , again, the signals will reflect their Petri net places and Req will go from high to low. The operation connected to Ack is a simple inverter, and so, once Ack has lowered, the last transition will fire consuming Req- and creating a mark on AllowCom, the system has then reached its initial state.

Some components possess complementary Petri net places, these Petri net places sole purpose is to counter balance other Petri net places, meaning that the Petri net place they are complementary to will have their transitions gated as to not allow more than one mark on them at any given time. They will not be further addressed below. Figure 2 displays the simple 4-phase receiver, in its initial state. As explained before, the receiver has been designed

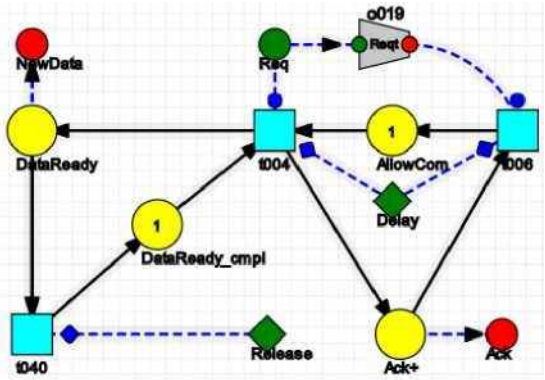

Fig. 2. Simple 4-Phase Receiver.

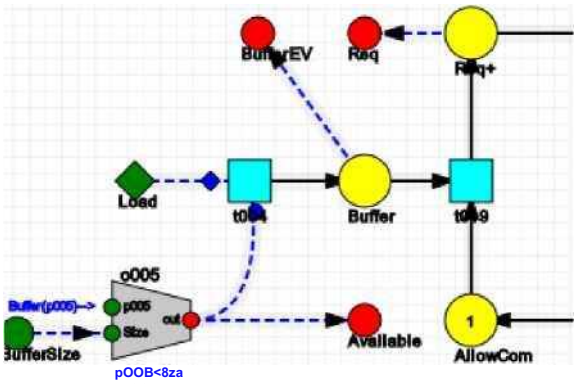

Fig. 3. Buffer addition.

to interact with the transmitter and for this reason some of its outputs will be the other's inputs and vice versa, namely, the Req and Ack signals that correspond to the request and acknowledge signals from the handshake protocol and in later components signals corresponding to the data channel, more on that later. This component may be viewed as having two sections, one that employs the handshake protocol to the right of t004 and one that informs the synchronous block that new information has arrived to the left. The state of the system will progress once Req is high, a mark will be consumed on AllowCom and two marks will be created, one on DataReady and another on Ack+, and so, NewData and Ack will rise to match their associated Petri net places . To the left, NewData signals the synchronous block that an event has been received from the transmitter, once the synchronous block has retrieved this information it should fire the Release event, then the corresponding transition will fire and this section will await new information. To the right, in a similar fashion to the transmitter, the operation connected to Req is a simple inverter, and so, once Req is lowered the transition it is connected to will fire, this section will then await new information. Only after both sections have been concluded is the system again in its initial state and ready for new information.

B. SimpleBuffer

As mentioned before, these components will be nearly identical to the simple components with the addition of a buffer. Figure 3 depicts how the buffer was implemented, while the figure pertains to the 4-phase transmitter, the addition is very similar for all components. First, it should be noted that a new input signal is present, BufferSize, this input is the maximum value held by the buffer not counting information already in transmission. A new operation is also present, it merely verifies if the buffer is not yet full, furthermore, the Available output signal is instead connected to this operation. The Petri net place named Buffer holds a mark for every event yet to transmit, and its value can be seen from outside the component through the output signal BufferEv. Lastly, as long as there are marks on the buffer, once a communication cycle comes to a conclusion a new one may immediately begin.

C. BurstBuffer

The BurstBuffer components are capable of transmitting multiple events at once through a data channel in order to save on communication cycles. For the transmitter, three operations have been added, as well as an output signal, OutData. The first operation will hold its output until either a Load event occurs, increasing the output value by one, or a new communication cycle begins, resetting the output to zero, furthermore, the output from this operation will be connected to both the BufferEv signal and to the second operation. The third operation, connected to t019, merely checks if the buffer is not empty before initiating a new communication cycle. The second operation, will be transparent while there is a mark on AllowCom and otherwise hold its value, in other words, whenever the system is not exchanging handshake signals, the operation will hold the value on the buffer, and when otherwise exchanging handshake signals the operation will hold its output. The system was modelled this way because the data signal must be valid during communication in order to avoid metastability. As an additional note, once again, the value being currently transmitted does not count towards the buffers capacity.
As for the BurstBuffer receiver, a couple of operations have been added as well as a new input signal, InData. This InData input signal will correspond to the OutData output signal from the previous component. Its important to note that even though a data signal is being transmitted it represents a number of events and, for this reason, events are retrieved from the receiver by the receiver's synchronous block one at a time. The most prominent adjustments from the previous category are the following, the operation that verifies if the buffer is not full now adds the current amount in the buffer to the amount of events in InData, a new operation has been added to handle the buffer, whenever data is received the operation adds InData to its buffer, if an event has been read by the correspondent synchronous block then the buffer will be decreased by one, otherwise, the output value is held. As long as there are events in the buffer, the section of the receiver responsible for communication with the synchronous block will be active.

D. Data

The Data interface, unlike the previous interfaces, will transmit data between the two blocks. It most closely resembles the Simple interface. With the addition of the elements in figure 4: a few new signals and an operation. The InData input signal will be the data the synchronous block intends to transmit, the OutData output signal will be the data transmitted through the interface, and lastly, the operation is transparent while there is a mark in AllowCom, much alike the BurstBuffer, and will otherwise hold its output.
As for the receiver, a similar addition is made, but the InData input signal will correspond to the OutData output signal from the transmitter and the OutData output signal from the receiver will be the value read by the synchronous block, the operation will function in the same matter, however, instead of reading the AllowCom Petri net place, the condition to turn the operation transparent is a high acknowledge

signal accompanied by a low request signal that was high during the previous clock. Furthermore, the left section from the Simple interface, responsible for informing the synchronous block that new information is available, will now also be connected to this transition, this is done because a late data valid scheme has been employed.

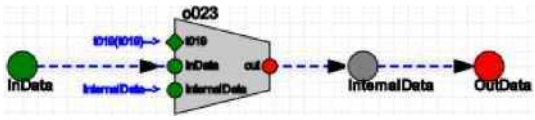

Fig. 4. Additional segment to the data component.

E. DataBuffer

The last interface, much like the previous one, transmits data, the difference is in the integration of a buffer. Unlike a buffer for events, the data values must be stored instead of counted, furthermore, the buffer size is fixed to three. A late data valid scheme is employed for the 4-phase protocols.

In figure 5, the 4-phase DataBuffer transmitter is depicted, in its initial state. The initial model from the simple 4-phase transmitter is still present, however, many additions have been made. Just below to the bottom right a section composed of two Petri net places and two Petri net transitions keep track of how many data values are stored inside the system, whenever a new value is added the Buffer place is incremented and the Available place is decremented, the opposite happens whenever a communication cycle reaches its conclusion. To the left, the load event is present as well as the InData signal and the OutData signal. Once an event is loaded, the value in InData is stored inside one of the Register operations it is connected to, this value is kept until a new value is provided. The WritePointer operation determines in which Register the new data is kept. Whenever the system is ready for a new communication cycle, the Multiplexer operation will hold the value from one of the Register operations, once again, which

Register is read is determined by the ReadPointer operation. The Multiplexer operation is granted to hold the value from the start of the communication cycle until a new cycle begins.

The mechanisms used for the receiver component will be very similar to those of the transmitter, apart from the portion kept from the Simple receiver.

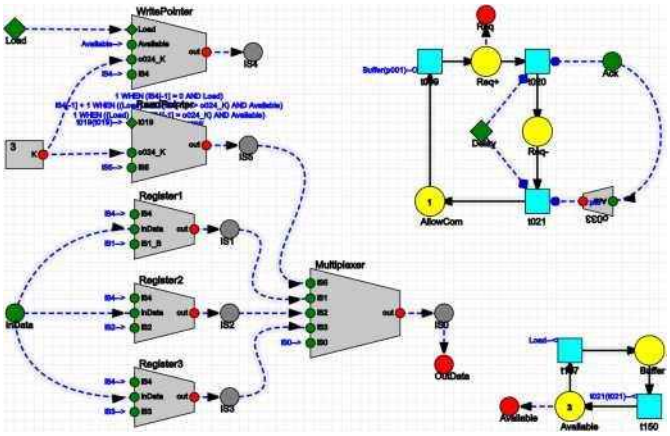

Fig. 5. DataBuffer 4-phase transmitter.

IV. VALIDATION

In figure 6, one can see the communication between the two Simple 4-phase components, on the left the transmitter and on the right the receiver. Connected to the Load input event is a Petri net transition, this is the recommended approach to using the transmitter block, additionally, this transition should be connected to the Available output signal in order to block any untimely Load events. Likewise, the Release input event should be accessed through a Petri net transition connected to the NewData output signal. These transitions may then be fired whenever the synchronous block intents to having only those small restrictions. Another notable detail is that the Req output signal from the transmitter is the receivers Req input signal, and in a similar fashion, the Ack output signal from the receiver is the transmitters Ack input signal.

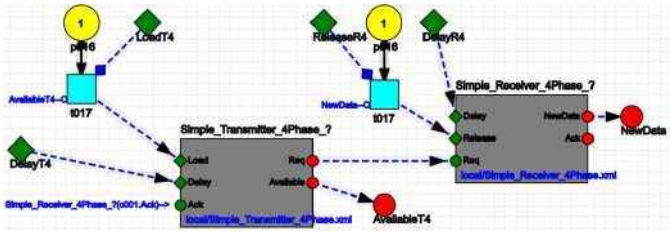

Fig. 6. Communication between the two Simple 4-phase components.

Figure 7 depicts the wave graph from the simulation of communication of a single event between the two Simple 4-phase components in the IOPT-Flow framework. Figure 8 depicts the wave graph from the same simulation in Xilinx ISE. From the model, VHDL code was generated on the IOPT- Flow framework and directly uploaded onto Xilinx ISE, the only modification was the removal of delay in the input and output signals. Then a test bench was created with the desired inputs and promptly simulated. The two simulations present similar behaviour which will now be further explored. Since the IOPT-Flow framework is not yet ready for multiple clock domains, a single clock is used and communication delays are emulated through the delay events. This way, it was possible to validate the developed asynchronous interfaces, but it is not possible to completely and accurately simulate GALS designs.

First, a Load event occurs, and as a result the Available signal will lower, since the transmitter is now busy, and the request signal will be asserted, in order to transmit this new event. Then, DelayR4 occurs, this is the aforementioned delay event to simulate communication delay, as a result the request signal will reach the receiver component and as a consequence the acknowledge signal is asserted, additionally, NewData is also asserted to inform the receivers synchronous block that a new event has been transmitted from the other synchronous block. Next, the events DelayT4, DelayR4 and, again, DelayT4 occur, the request and acknowledge signals will rise and lower accordingly. As the lowered acknowledge signal is received by the transmitter, the Available signal is asserted to signify to the connected synchronous block that it may transmit a new signal. Lastly, once the receivers synchronous block has taken notice of the transmitted event it should fire the Petri net transition connected to the release input, as a result NewData will lower and the receiver is ready for another communication cycle. The other asynchronous interfaces will have similar responses, according to their behaviour. These components can be visualized at http://gres.uninova.pt/iopt-flow/.

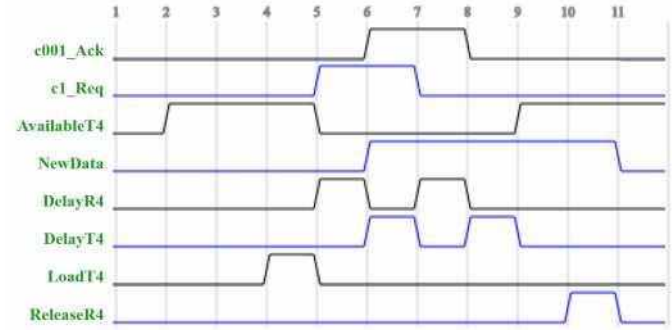

Fig. 7. Wave graph from the simulation performed on the IOPT-Flow framework.

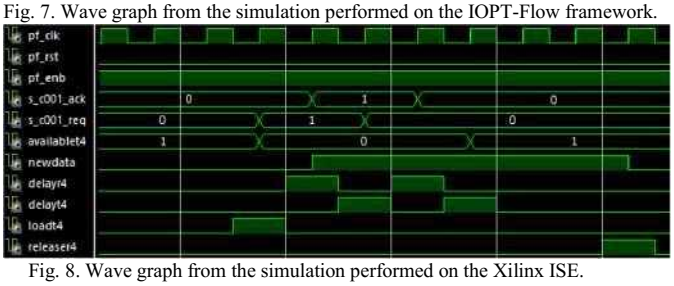

Fig. 8. Wave graph from the simulation performed on the Xilinx ISE.

V. Conclusion

With the purpose of supporting the development of GALS systems in the IOPT-Flow framework, a handful of components were created. These components are distinguished in five different categories each with their own behaviours and mechanisms in the interest of providing more options to the designer. This paper brings to light these components and seeks to explain how they function. Lastly, a comparison is made between the IOPT-Flow simulation and the Xilinx simulation of the aforementioned components. The models of the components are available online at http://gres.uninova.pt/iopt- flow/, where their VHDL code can be automatically generated.

Acknowledgment

This work was partially financed by Portuguese Agency ”Fundacao para a Ciência e a Tecnologia” (FCT), in the framework of project UID/EEA/00066/2019.

References

[1] F. Pereira and L. Gomes, “Combining Data-Flows and Petri Nets for Cyber-Physical Systems Specification,” in *Technological Innovation for Cyber-Physical Systems,* L. M. Camarinha-Matos, A. J. Falcão, N. Vafaei, and S. Najdi, Eds. Cham: Springer International Publishing, 2016, pp. 65-76.

[2] F. Pereira and L. Gomes, “The IOPT-Flow framework pairing Petri nets and data-flows for embedded controller development,” in *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, Oct 2016, pp. 4832-4837.

[3] A. Iyer and D. Marculescu, “Power and performance evaluation of globally asynchronous locally synchronous processors,” in *Proceedings 29th Annual International Symposium on Computer Architecture*, May 2002, pp. 158-168.

[4] R. Jain, D. Padole, M. M. B. Kulkarni, Abhijeet, and A. Singhal, “Design of Globally Asynchronous Locally Synchronous (GALS) System using FPGA,” in *International Journal of Enhanced Research in Science Technology Engineering*, 2014.

[5] F. K. Gurkaynak, S. Oetiker, N. Felber, H. Kaeslin, and W. Fichtner, “Is there hope for GALS in the future?” in *4th ACiD Workshop of the European commissions fifth framework programme*, 2004.

[6] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber, and W. Fichtner, “Globally-asynchronous locally-synchronous architectures to simplify the design of on-chip systems,” in *Twelfth Annual IEEE International ASIC/SOC Conference (Cat. No.99TH8454)*, Sep. 1999, pp. 317-321.

[7] E. Amini, M. Najibi, and H. Pedram, “Globally asynchronous locally synchronous wrapper circuit based on clock gating,” in *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, March 2006, pp. 6 -.

[8] P. Teehan, M. Greenstreet, and G. Lemieux, “A Survey and Taxonomy of GALS Design Styles,” *IEEE Design Test of Computers*, vol. 24, no. 5, pp. 418-428, Sep. 2007.

[9] J. Ax, N. Kucza, M. Vohrmann, T. Jungeblut, M. Porrmann, and U. Rck- ert, “Comparing Synchronous, Mesochronous and Asynchronous NoCs for GALS Based MPSoCs,” in *2017 IEEE 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, Sep. 2017, pp. 45-51.

[10] A. Peeters and K. van Berkel, “Single-rail handshake circuits,” in *Proceedings of the 2Nd Working Conference on Asynchronous Design Methodologies*, 06 1995, pp. 53 - 62.