Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 200 (2008) 33–50
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Dataflow Architectures for GALS

Syed Suhaib, Deepak Mathaikutty, and Sandeep Shukla

FERMAT LAB
Bradley Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, USA.

Abstract

In Kahn process network (KPN), the processes (nodes) communicate by unbounded unidirectional FIFO channels (arcs), with the property of non-blocking writes and blocking reads on the channels. KPN provides a semantic model of computation, where a computation can be expressed as a set of asynchronously communicating processes. However, the unbounded FIFO based asynchrony is not realizable in practice and hence requires refinement in real hardware. In this work, we start with KPN as the model of computation for GALS, and discuss how different GALS architectures can be realized. We borrow some ideas from existing dataflow architectures for our GALS designs.

Keywords: Kahn process networks, globally asynchronous locally synchronous, unbounded FIFO channels.

1 Introduction

Globally asynchronous locally synchronous (GALS) designs are gaining importance due to the fact that the synchrony assumption is failing in large synchronous designs. This is because of the ever increasing clock frequencies, which causes the time taken for a signal to propagate between different components to be longer than the clock period [1]. In a GALS design, the communication between the synchronous components occur asynchronously. The synchrony assumption holds within each synchronous component. However, there are other challenges facing the design of GALS systems. There is a lack of tools and design methodologies to facilitate GALS designs. In most cases, GALS designs are constructed using ad hoc methods, where synchronous components are encapsulated with some wrapper logic and communication is handshake driven. Furthermore, these ad hoc approaches are not easily subject to formal reasoning about the correctness of a design. Hence, we need to identify the basic ingredients for a successful GALS design methodology.

The objective of this paper is to facilitate a methodological approach to GALS design borrowing models of computation (MoCs) [2], and architectural concepts

doi:10.1016/j.entcs.2008.02.005

from dataflow computing [3,4], and principles employed in latency insensitive systems [5,6,7,8]. In this paper, we (i) use Kahn Process Network (KPN) as the MoC for specification of a computation, (ii) show refinements of KPN into various GALS architectures, (iii) discuss tradeoffs for various architectures.

Design Methodology for GALS: The design methodology we propose for GALS design can be summarized as follows: Given the description of a system, the first step is to identify the behaviors of the system as a collection of concurrent processes communicating asynchronously with unbounded FIFOs according to a KPN MoC. To ensure the correctness of the KPN model, the specification can be validated for its decidable properties related to composition, determinacy, etc. Architectural exploration can then be performed to identify the appropriate GALS architecture based on the KPN model. Some of the constraints for choosing an appropriate architecture that we identify include area (for example, number of extra interconnects, storage elements, etc), and latency for enabling communication. Based on the result of architectural exploration, appropriate refinements are applied to the KPN model.

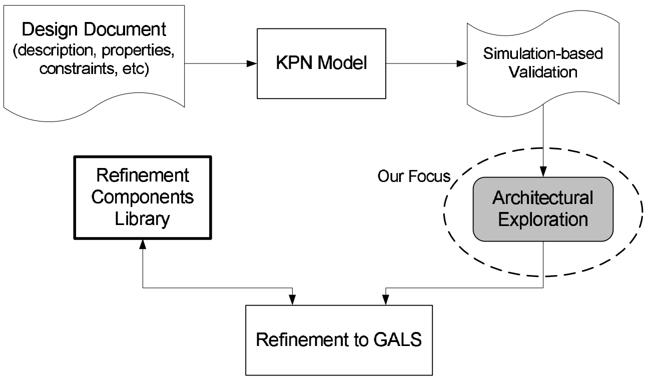
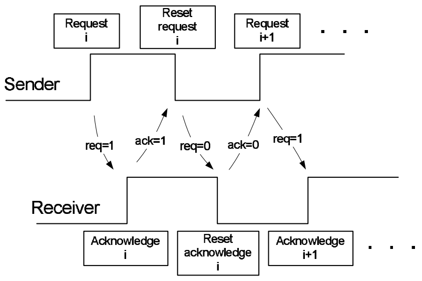

Design Document
(description, properties, constraints, etc)
KPN Model
Simulation-based Validation
Refinement Components Library
Our Focus
Architectural Exploration
Refinement to GALS

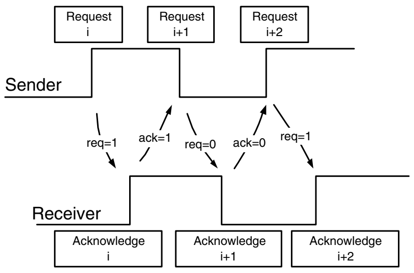
Fig. 1. Design Methodology for GALS

For this paper, our focus is a number of refinements of KPN into realizable GALS architectures. We discuss four different possible architectures along with their tradeoffs. Figure 1 illustrates this design methodology.

Model of Computation for GALS: Most of the design languages used in the industry such as SystemC [9], SpecC [10], support synchronous specifications, and use discrete event as their MoC. A GALS design can be modeled in these languages using discrete event MoC, however, it is not very natural. KPN is a well known simplistic model used for expressing behaviors involving dataflow such as streaming audio, video, and other 3D multimedia applications as well as DSP applications. The behaviors of the applications are modeled as a collection of concurrent processes communicating data via first-in-first-out (FIFO) channels with unbounded capacity. The processes in a KPN are sequential programs that consume data (referred as *tokens*) from their input channels and produce data on their output channels. These processes execute based on blocking read and non-blocking write schemes. A KPN model closely relates to a GALS description, where the processes can be seen as



(a) Four-phase handshake



(b) Two-phase handshake

Fig. 2. Handshake Protocols

synchronous components, and their interaction as asynchronous.

We illustrate four different architectures in this paper: handshake-based, FIFO-based, controller-based, and lookup-based architectures. For the FIFO-based architecture, we show two variants: (i) architecture based on handshake protocol with asynchronous FIFO, and (ii) architecture based on principles of LIP.

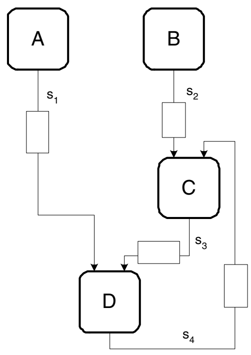
During discussions of these architectures, we dwell upon their tradeoffs, and when these architectures should be selected.

2 Background

In this section, we discuss some background material for understanding of the architectures illustrated in this paper.

The handshake in most asynchronous circuits use signaling involving requests and acknowledgements. This computational model is used for dataflow computing [11,12], where the arrival of data triggers an operation. Of the many known handshaking protocols are the four-phase handshake (Figure 2(a)) and two-phase handshake (Figure 2(b)).

The latency insensitive protocols (LIPs) [5,6,7,8] have been applied on synchronous systems, where all components are assumed to receive the same clock but some interconnects are too long for signals to propagate within a single clock cycle. The protocol involves encapsulation of all components of the design with a wrapper logic that communicates with addition signals: *valid* and *stall*. Extra storage elements are added along the long interconnects for segmenting longer signal delay paths into shorter signal delay paths with propagation delay less than a clock period. We focus our discussion on the role of valid and stall signals. A sender sends valid data (validity of data denoted by valid signals) to its receiver on every clock, and whenever the stall signal is not set. Once the stall is asserted, the sender does not send valid data. In the case of request-acknowledge signals, the request signal will always be followed by an acknowledge signal for passing new data.



a) KPN Model

ticks	t_1	t_2	t_3	t_4	t_5	t_6
$\mathcal{A}$	✓	✓	✓	✓	-	✓
$\mathcal{B}$	✓	-	✓	✓	✓	✓
$\mathcal{C}$	-	✓	-	✓	-	-
$\mathcal{D}$	-	✓	✓	-	✓	✓

b) Corresponding Clock Table

Fig. 3. Running Example

3 A Running Example

For this paper, we discuss different architecture with a running example to compare their pros and cons. We use the KPN diagram shown in Figure 3 that consists of four processes: $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$ which connect by channels s_1, s_2, s_3, and s_4 with unbounded FIFOs. The processes $\mathcal{A}$ and $\mathcal{B}$ are source processes that produce tokens on channels s_1 and s_2. The process $\mathcal{C}$ has an initial token on channel s_4. A possible behavior of the network is as follows: Processes $\mathcal{A}$ and $\mathcal{B}$ execute producing tokens on their respective outputs. Process $\mathcal{D}$ cannot execute as there are no tokens from $\mathcal{C}$, so $\mathcal{C}$ executes first, followed by $\mathcal{D}$.

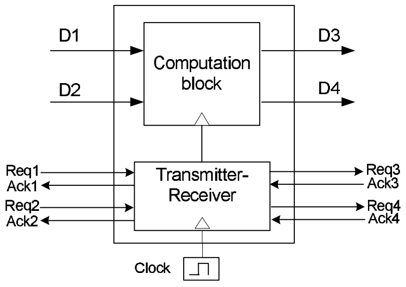
The components in GALS are associated with clocks, which are unknown to the designer. The clocks for these components are assumed to be independent. These clocks can either be generated locally by using gates such as inverters in a locked loop fashion, or can be from an external source.

For our KPN example of Figure 3a, we consider sample clock ticks shown in Table 3b. These ticks can be seen by an observer that is observing the design synchronously and analyzing the clock realization for different components. A clock tick represents a time stamp based on when the components are fired. The clock ticks (✓) signify when the clocks of the respective components are triggered. For example, the component $\mathcal{A}$ is observed to trigger at t_1, t_2, t_3, t_4, and t_6, and component $\mathcal{B}$ triggers at t_1, t_3, t_4, t_5, and t_6. From the clock table, it can be said that components $\mathcal{A}$ and $\mathcal{B}$ execute in parallel at t_1. Please note that the information presented in Table 3b is a sample observation from an observer when the design executes. We are using this table to illustrate our point. These clock relationships are not known to the designers at design time. So, as far as the designer is concerned, the components are completely asynchronous with respect to each other.

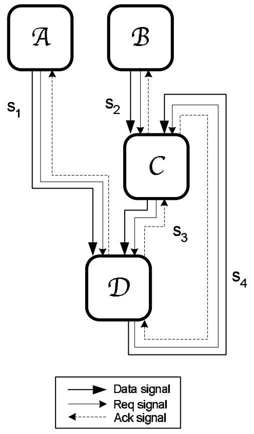
4 Handshake based GALS Architecture

In the handshake-based GALS architecture, the synchronous components communicate directly via handshaking schemes. A receiver-transmitter unit (RTU) is added to each component to ensure proper execution of the request-acknowledge based handshake protocol. Each signal (carrying valid data) is augmented with two extra signals for control purposes: request and acknowledge. The components follow the signaling protocol discussed earlier.

Consider a source component and a destination component, where the source component sends data to the destination component. The data can be sent or received when it is triggered by its clock. Figure 4(a) shows a component with two input and two output signals. D1, D2, D3, and D4 are the input/output data signals, and req1, ack1, req2, ack2, req3, ack3, req4, and ack4 are its corresponding request and acknowledgement signals.



(a) Component with RTU



(b) GALS example

Fig. 4. Handshake-based GALS Architecture

In a network, a synchronous component executes when the following conditions hold: (i) all its input request signals are requesting (req=1) , (ii) all its input acknowledge signals are waiting for new request (ack=0). Once, both these conditions hold, the component executes based on its clock. Until these conditions are true, the synchronous component is disabled.

Figure 4(b) can be refined from Figure 3 where the RTU are added to each process, and the communication between nodes handled by: data signal, request signal (req) and acknowledge signal (ack).

Given handshake based GALS in Figure 4(b) and its corresponding clocks in Table 3b, we analyze its simulation trace based on four-phase handshaking protocol. Figure 5 illustrates what signals are updated at different clock ticks. Now, based on the clock table, components $\mathcal{A}$ and $\mathcal{B}$ trigger at t_1, i.e. $\mathcal{A}$'s $s_1.req$=1 and $\mathcal{B}$'s $s_2.req$=1. In t_2, component $\mathcal{C}$'s $s_2.ack$=1 and $\mathcal{D}$'s $s_1.ack$=1, however, component $\mathcal{A}$ keeps waiting as no acknowledgement has been received from component $\mathcal{D}$. Note that the data is transmitted when the request signal is set to ‘1’, and the sender knows that the receiver is ready to receive a new value when acknowledge signal is set to ‘0’.

Pros and Cons: The signaling protocol has been used for static dataflow architectures [4]. At most one valid data value can be present on a communication

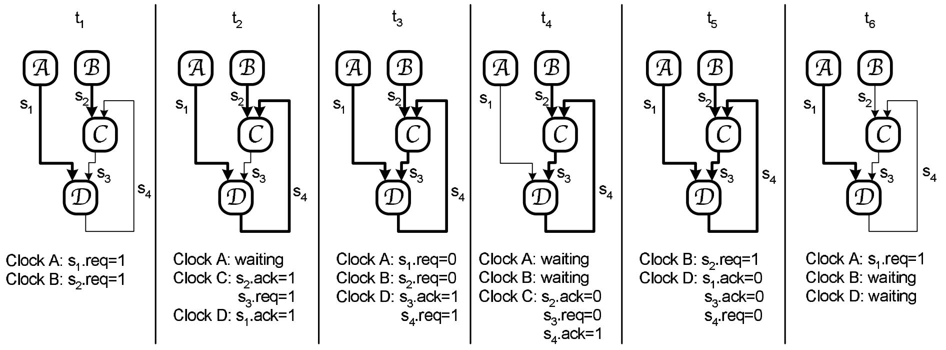


Fig. 5. Simulation Trace for Handshake-based Architecture

signals. This is one of the disadvantages of the architecture since the components would only execute if new data can be stored on the outputs. This also restricts parallelism in the design. Multiple handshakes are required for transferring data from one component to another which consumes more power and limits the performance. Secondly, if there are n inter-component signals, then $2 * n$ additional signals are required for request and acknowledgements. For the example shown, a total of 12 (4+2*(4)) signals are required.

5 FIFO-based GALS Architecture

We discuss two variants for implementing a FIFO-based GALS architecture. These are based on (i) handshaking scheme, and (ii) principles of LIP.

In the handshaking scheme for FIFO-based GALS architecture, the components are refined with the protocol discussed in Section 4, where RTUs are added to all components. An asynchronous FIFO with a bounded size is placed between the components. The component now handshakes with this bounded FIFO. We explain this with an example shown in Figure 6 where two components A and B communicate with an N-size FIFO in between.

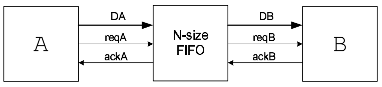


Fig. 6. Asynchronous FIFO with Handshake

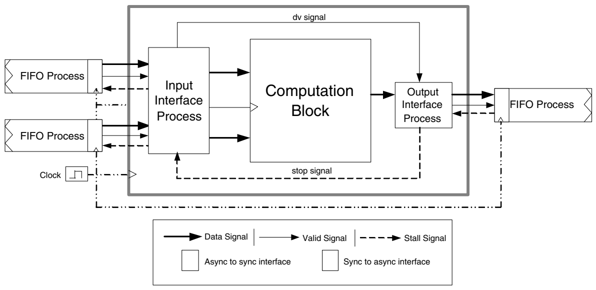
The asynchronous FIFO placed in between two components (A and B) will require RTUs on both its ends. Component A's RTU will communicate with RTU of the FIFO facing towards A. The RTU of the FIFO facing B will communicate with the RTU of B. For the four-phase handshake protocol, four handshakes will be required to communicate a single data from component A to FIFO, and the same from the FIFO to component B. In other words, a total of 8 handshakes will be needed to communicate a data from component A to component B. In the case of two-phase handshake, the total handshakes for exchanging one data will 4. Such an architecture will be very expensive with respect to the performance of the design.

We now propose a new GALS architecture based on the principles of LIPs. Recall that the communication is handled by valid-stall signals which are generated on the clocks of the components. Valid and stall signals are added for each inter-component signal.

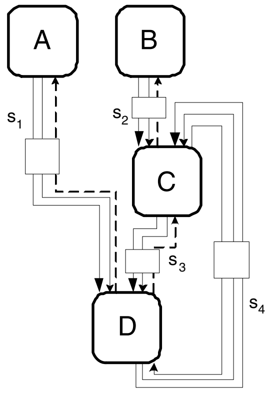
The protocol involves refinement of each component with: (1) Input interface process (IIP), and (2) output interface process (OIP). Asynchronous FIFOs are placed between two components for communication. These FIFOs are equipped with interfaces that ensure correct communication between components with independent clocks. Detail information about these FIFOs can be found in [13]. Figure 7(a) illustrates the block diagram of a component in this architecture.

Input interface process with barrier synchronization (IIP): The IIP is placed at the input of the synchronous component. The main idea of this process is to barrier synchronize (align) all the valid inputs for the computational block. The block can only execute once all the inputs have been realized. Each IIP contains buffers for each input signal to store input data values. There are exactly two storage elements for each input. This is because when the computation block is stopped by the IIP, the incoming inputs need to be stored, and the stall signals for the appropriate source components have to be enabled. The need for the stall signal is realized as soon as the first storage element is filled. By the time the stall signal is enabled, the source component could have placed another valid value on the signal. Therefore, the second storage is needed to store this value.

IIP takes input data signals with their corresponding valid signals from its source components, and a *stop* signal from OIP to indicate that OIP is not ready to accept new values. IIP provides data to the computation block, a dv signal (stands for data-valid signals) to the OIP indicating it is sending a valid value, and stall signals to its source components. The IIP works in two phases: In the first phase it reads all inputs and stores the data values in its buffer. In the second phase it provides the data values to the component based on its input valid signals which are written to its output.



(a) Block Diagram of Component



(b) FIFO-based GALS

Fig. 7. FIFO-based GALS Architecture

There are three possible scenarios that can occur: (i) *All input valid signals are*

1. During the first phase, the IIP stores the data, and at the start of the second phase, the values are provided to the computation block. The IIP outputs a 1 on its *dv* signal, and 0 is placed on all output stall signals. (ii) *All input valid signals are not 1.* In the first phase, the IIP reads all the valid values from its inputs, and stores them in their respective storage elements. During the second phase, the IIP sends 0 on its *dv* signal. For the inputs where valid value was not realized, the IIP places 0 on their corresponding stall signals. (iii) *The stop signal from OIP is enabled.* In the first phase, the IIP will read and store the inputs. In the second phase, the IIP outputs a 0 on *dv* signal, and places 0 on all its output stall signals.

Output interface process (OIP): The OIP is placed at the output of the synchronous component, and contains one buffer to store the result of the synchronous block. The inputs of OIP are *dv* signal from IIP, data from the computation block, and stall signals from its destination FIFOs. OIP reads and stores the value from the computation block whenever a 1 is received on the *dv* signal. The OIP places the valid value received from the computation block to its output when the stall signal from the FIFO is disabled. In the case when the stall signal from the FIFO is enabled, the data from the computation block is stored in its buffer, and 1 is placed on the *stop* signal to the IIP.

FIFO process: The $FIFO$ process provides the communication between two synchronous components. At each end of the FIFO, there are interfaces that communicate with the synchronous component. Note that this FIFO stands as an interface between the components running on different clocks. So, this FIFO has an synchronous to asynchronous interface on its input end, and an asynchronous to synchronous interface on its output end. Details about such interfaces can be found in [13]. The FIFO enables a stall to its source component when the buffer becomes full. Valid data is written on its output based on the clock of the destination component when valid value is present. Figure 7(b) illustrates a diagram of a FIFO-based GALS architecture.

Now, consider the example of FIFO-based GALS in Figure 7(b) and the clocks of its corresponding components in Table 3b, we analyze its simulation trace which depends on the size of the FIFOs on the communication channels. Table 1 shows the size on the channels along with the number of valid values present on the channels. Note that the components execute on its clock when the data is present in the FIFOs of its input channels and its output channel FIFOs are not full. The components $\mathcal{A}$ and $\mathcal{B}$ trigger on t_1, i.e. $\mathcal{A}$ and $\mathcal{B}$ will produce a valid value and store it in its output FIFO channels s_1 and s_2. At t_2, the clocks of components $\mathcal{A}$, $\mathcal{C}$, and $\mathcal{D}$ arrive. Component $\mathcal{C}$ executes as it has valid values on both its input signals s_2 and s_3 which are realized at t_1, and the token is removed from their respective FIFOs (recall that the input from $\mathcal{D}$ to $\mathcal{C}$ has an initial valid value as shown in t_0). Component $\mathcal{D}$ will not execute since no input is received from $\mathcal{C}$ at t_1. The count of valid value on s_3 is '0'. Component $\mathcal{A}$ produces another value which is stored in s_1. Now, the maximum FIFO size of s_1 is '2', so at this point, the channel reaches its maximum capacity. Therefore, a stall signal to component $\mathcal{A}$ is enabled to stop it from producing newer values (denoted by a *). The stall signal is disabled when

-	FIFO size	t_0	t_1	t_2	t_3	t_4	t_5	t_6
s_1	2	0	1	2*	1	2*	1	2
s_2	3	0	1	0	1	1	2	3*
s_3	3	0	0	1	0	1	0	0
s_4	2	1	1	0	1	0	1	1

* denotes stall signal is enabled

Table 1
Count of Valid Values on Channels.

the FIFO on s_1 is ready to accept more values. Note that the stall signals to the components are enabled/disabled by the FIFO.

Pros and Cons: The number of valid and stall signals added would increase from the handshake based GALS architecture because of the FIFOs placed in-between the components. However, the components in this architecture may not necessarily stop after every execution. A component will only get stalled when no data is seen on any of its inputs, or if the FIFO buffers at its output channels become full. The stall signals form a back pressure mechanism that ensure that the data is not lost during communication [14]. This type of architecture is closely related to the static dataflow architecture with the difference that more number of tokens can be stored on the channels. The FIFO-based architecture will have a better performance with respect to the handshake-based architecture, and increases parallelism.

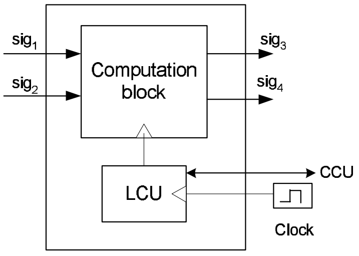
6 Controller-based GALS Architecture

The controller-based GALS architecture is realized by refining each process in a KPN network into a synchronous component with a local control unit (LCU). The LCUs of the components communicate asynchronously with a central control unit (CCU) to request for a permission to execute. Figure 8(a) shows the block diagram of a component with an LCU unit, and Figure 8(b) illustrates a controller-based architecture.

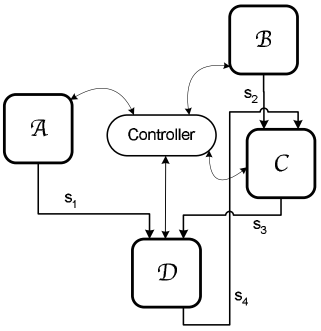
The execution of the computation block is controlled by its LCU. The LCU sends a request message to the CCU. The format of the request message is as follows:

RequestMsg = { Component_id: String; Component_Status: boolean; Execution_Status: boolean; Input_Signal_list: String list; Output_Signal_list: String list;}

The *Component_id* contains a unique name of the component. The *Component_Status* can be *true* or *false*. A *Component_Status* = *true* means that the component is requesting for a grant status, whereas a *Component_Status* = *false* means that the component is requesting for the up-



(a) Component with LCU



(b) Controller-based GALS

Fig. 8. Controller-based GALS Architecture

date. The *Execution_Status* contains information about the previous grant request. This information is used by the CCU for updating its local structure[1]. The *Input_Signal_list* and *Output_Signal_list* contain the inputs and outputs of the component.

Algorithm 1. LCU execution steps on clock arrival

```
Step 1: Initialize the request message structure.
Step 2: Send request message to CCU with $Component\_Status = true$
Step 3: **If** grant=$true$
            Enable computation block for execution.
            $Execution\_Status = true$
**else** $Execution\_Status = false$
Step 4: Send request message to CCU with $Component\_Status = false$.
```

The request signal passes the address of the *RequestMsg* structure to the CCU with *Component_Status* = *true*. The CCU upon receiving the address of the request message, retrieves the information and responds by giving a grant as *true* or *false*. An enabled grant request has grant=*true*, otherwise vice versa. When the LCU receives a grant=*true* from CCU, it enables the computation block for execution. After execution, *Execution_Status* is set to *true* and *Component_Status* is set to *false*, and the request signal is sent back to the CCU. If grant=*false* is received from CCU, *Execution_Status* as well as *Component_Status* are set to *false*, and the request signal is sent back to the CCU. The algorithm 1 defines the steps of the LCU that occur on each clock of the component. This is because the components in GALS only fire on the arrival of their clocks.

[1] We will discuss this later

Next, we discuss the functionality of the CCU. The CCU is an asynchronous component that receives the request messages from the LCUs of different components, and based on the presence of values on the signals, grant the requests accordingly. The CCU consists of a simple structure that stores the presence and absence of values of different signals of the network. The storage structure for CCU is as follows:

SignalStatus = { Signal_Name: String; Value_Status: Boolean; }

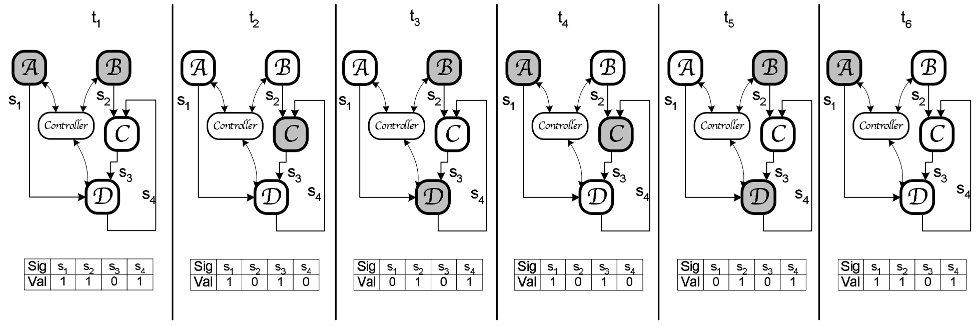
The $SignalStatus$ structure is stored as a list of structures. An alternate implementation can be organizing the same data as a hash table. The $Signal\_Name$ is associated with the signal connecting two synchronous components, and its $Value\_Status$ corresponds to a boolean value, which if high means that the signal has a valid value, and low means that the signal does not have a valid value. Algorithm 2 shows the steps taken by the CCU when it receives a request.

Algorithm 2. CCU execution steps on receiving request

```
If $Component\_Status = true$
        Fetch the appropriate status values from the $SignalStatus$ structure.
        If $SignalStatus$ values of all inputs are high, and all outputs are low
                grant=$true$
        else grant=$false$.
else if $Component\_Status = false$
        If $Execution\_Status = true$ (atomic step)
                Set all inputs to low in Signal_Status table.
                Set all outputs to high in Signal_Status table.
```

If more than one request is received by the CCU, the grant status is computed for all the requesting components. The update to the $SignalStatus$ structure only occurs if the message received from an LCU contains $Component\_Status = false$ and $Execution\_Status = true$. This update is done in an atomic step. Furthermore, when many requests are received by CCU, a case where two requests require updating the same signal value will never exists. This is due to the fact that the grant signals are always generated before the update is done, and the update occurs only on those signals that are either inputs or outputs to the components receiving true grant signals. So, if there are two components connecting each other, then they both will never be provided the grant request at the same time.

Now, consider the example of Controller-based GALS in Figure 8(b) and the clocks of its corresponding components in Table 3b, we analyze its simulation trace of the signal status table. Figure 9 shows the presence of values at each clock tick. The signals between the components and the controller handles the exchange of messages. Recall that we have initially assumed that signal s_4 has an initial value. The components that execute during the clock are shaded. For instance, at clock



Sig	s_1	s_2	s_3	s_4
Val	1	1	0	1

Sig	s_1	s_2	s_3	s_4
Val	1	0	1	0

Sig	s_1	s_2	s_3	s_4
Val	0	1	0	1

Sig	s_1	s_2	s_3	s_4
Val	1	0	1	0

Sig	s_1	s_2	s_3	s_4
Val	0	1	0	1

Sig	s_1	s_2	s_3	s_4
Val	1	1	0	1

Fig. 9. Simulation Trace of Signal Status Table in Controller

tick t_2, clocks of component $\mathcal{A}$, $\mathcal{C}$, and $\mathcal{D}$ arrive, however only $\mathcal{C}$ executes since s_1 and s_2 have valid values (realized at t_1). Components $\mathcal{A}$ and $\mathcal{B}$ do not receive an enabled grant signal from the CCU, as the values s_1 and s_2 are high in the signal status table at t_1. Secondly, if $\mathcal{A}$ and $\mathcal{B}$ were to be executed, then their previous values would have been overridden.

Pros and Cons: In the controller-based GALS architecture, there is no back-pressure [14] mechanism which is seen in the FIFO-based GALS architecture and other existing GALS designs [15]. The synchronous components execute based on the grant requests received by the CCU. Also, each component has a simple communication model between the LCU and CCU for grant request. However, the CCU can be a major bottleneck for the design. This is because the request for all the components of the network are handled by this one single unit. Secondly, each component has back and forth (req/grant) signals to the CCU. The number of additional signals depend on the number of components in the design. The throughput of this architecture will be similar to the handshake-based architecture because at most only one token (valid value) can exist on a single arc (i.e. inter-component signal[2]). Furthermore, some of the ideas such as the use of a centralized controller have been borrowed from the tagged-token dataflow architectures [4].

7 Lookup-based GALS Architecture

A storage mapping unit (SMU) is added to each component in the lookup-based GALS architecture. The communication between the components is based on reading and writing from a lookup storage which is placed on the chip for fast access to data. This lookup table acts as a shared storage between components which removes the need for explicit signal exchanges. Such an architecture can be considered specifically for GALS, since accessing to main memory would be very expensive and time consuming. Figure 10a illustrates a diagram of a component where s_1 and s_2 are inputs and, s_3 and s_4 are outputs.

[2] A signal connecting two components

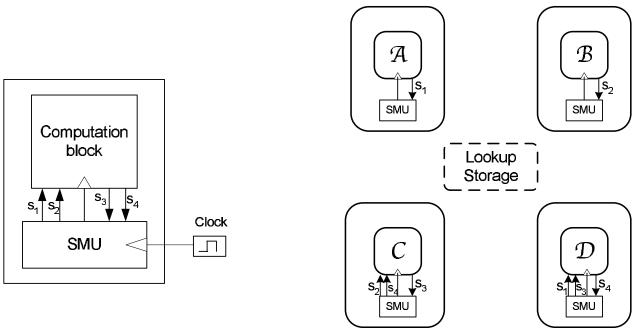


Fig. 10. Block Diagram of a Component in Lookup-based GALS Architecture

Data Structure: The addresses of the inputs and outputs for a component are stored internally within a component's SMU. The storage structure in the SMU is shown in Figure 11.

Address In1	Bound In1	Address In2	Bound In2	▬ ▬ ▬	Address Out1	Bound Out1	▬ ▬ ▬ ▬ ▬

Fig. 11. Storage Structure in SMU

The storage structure contains fields for the inputs and outputs. Each input and output field is divided into two parts: address and bound. The address part points to the location of the inputs/outputs in the lookup storage. Initially, the address part for each input and output field contains its initial (starting) address. The bound part represents the maximum number of valid data locations that can be stored starting from the initial address location. In other words, the bound represents the maximum valid values that can be saved at a given time.

Purpose of the structure: The organization of the fields clearly helps in identifying the addresses of the inputs and outputs simultaneously. Once the address is accessed, to either read or store another value, the address can be incremented by 1 until the bound is reached. In other words, to access the next location, the address part is incremented as follows: $(address + 1) \% \; bound$. Here, *address* and *bound* represent the corresponding address and bound locations (of the input/output).

On-chip Lookup Storage: We consider that the lookup storage is placed on the chip for fast access of data. We assume that the time required for accessing data is faster than the clock of any component in the architecture. For an on-chip storage, this can be a feasible assumption. The lookup storage size can be computed based on the number of elements that can be stored. For the lookup storage, we assume that it is split into different segments. We assume that a storage location is 32-bits in length. The number of segments in the lookup storage depends on the number of inter-component signals. Consider that there are n signals in the GALS design, and each signal i has a bound sz_i associated with it. The *datasize* corresponds to the size of data stored in the storage location. Therefore, the total number of segments in the lookup storage is i, and the total size of the storage is computed as follows:

Algorithm 3. Functionality of SMU

Step 1: Retrieve inputs whose *present* bits are '0', and outputs whose *present* bits are '1' in the local storage.
Step 2: If the *present* bits of all the inputs are '1' and that of all the outputs are '0', then the computational block is enabled for execution with data at input location. Otherwise, jump to Step 8.
Step 3: The *present* bits of all inputs are set to 0 in the local storage locations.
Step 4: The outputs from computation block are stored in the local storage and the corresponding *present* bits are set to '1'.
Step 5: The data for inputs and outputs is written back in an atomic step to the same addresses from where these were read.
Step 6: The local address of all inputs are incremented by '1' % *bound* to point to the next read location.
Step 7: The local address of all outputs are incremented by '1' % *bound* to point to the next write location.
Step 8: For all inputs retrieved in the current cycle with *present* bits as '1', and all outputs retrieved in the current cycle with *present* bits as '0', increment their corresponding address fields.

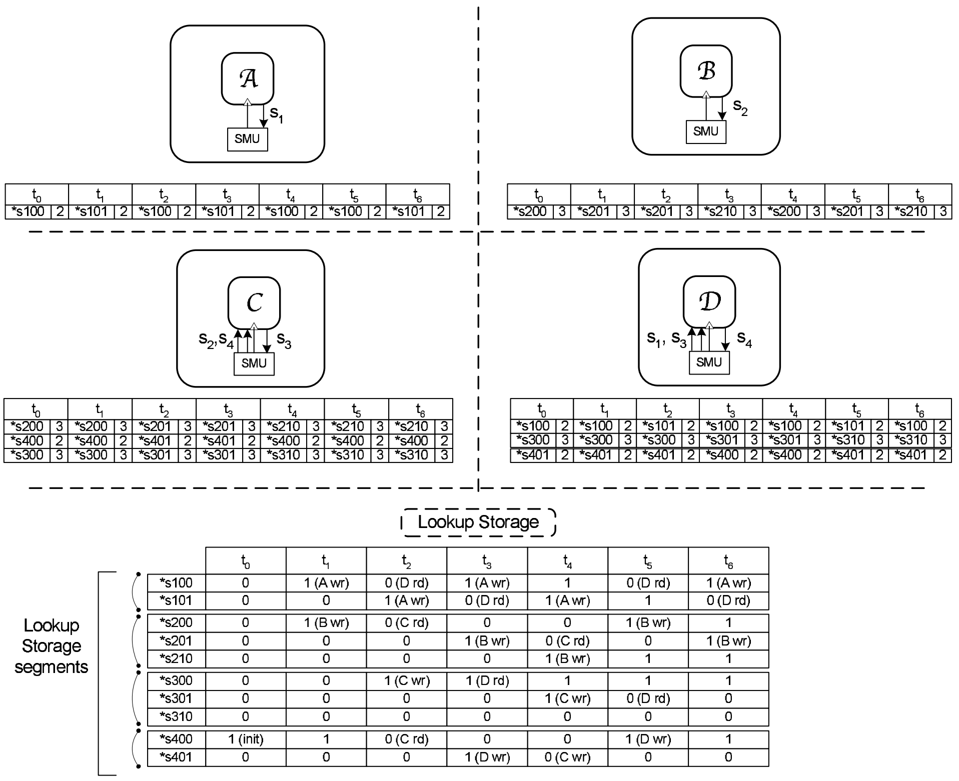
$$\sum_{i=1}^{n} sz_i * datasize = (sz_1 + sz_2 + \ldots + sz_n) * datasize$$

We now look at how the data is organized in the lookup storage. We assume that the data that is retrieved is 32 bits. From these 32 bits, the most significant bit (MSB) represents the *present* bit. The *present* bit if set to 1 implies that the data is valid, otherwise it is invalid. The actual data is 31 bits.

Functionality of a Storage Mapping Unit (SMU): The SMU maps the addresses of each input/output to the correct lookup storage locations. The SMU contains local storage elements to store the inputs/outputs that were retrieved earlier. This is based on the number of input and output fields. We assume initially that all storage is empty. The SMU also has the capability to extract the MSBs of the data retrieved. This can be implemented as a simple function.

On each clock of the component, the functionality of SMU is defined in Algorithm 3:

Now, consider the example of the lookup-based GALS (Figure 12) and the clocks of its corresponding components (Table 3b). For comparison purposes, we consider the bounds for each address to be the same as the corresponding size of the FIFOs considered in the FIFO-based architecture. The lookup-storage size can be computed by



t_0		t_1		t_2		t_3		t_4		t_5		t_6	
*s100	2	*s101	2	*s100	2	*s101	2	*s100	2	*s100	2	*s101	2

t_0		t_1		t_2		t_3		t_4		t_5		t_6	
*s200	3	*s201	3	*s201	3	*s210	3	*s200	3	*s201	3	*s210	3

t_0		t_1		t_2		t_3		t_4		t_5		t_6	
*s200	3	*s200	3	*s201	3	*s201	3	*s210	3	*s210	3	*s210	3
*s400	2	*s400	2	*s401	2	*s401	2	*s400	2	*s400	2	*s400	2
*s300	3	*s300	3	*s301	3	*s301	3	*s310	3	*s310	3	*s310	3

t_0		t_1		t_2		t_3		t_4		t_5		t_6	
*s100	2	*s100	2	*s101	2	*s100	2	*s100	2	*s101	2	*s100	2
*s300	3	*s300	3	*s300	3	*s301	3	*s301	3	*s310	3	*s310	3
*s401	2	*s401	2	*s401	2	*s400	2	*s400	2	*s401	2	*s401	2

	t_0	t_1	t_2	t_3	t_4	t_5	t_6
*s100	0	1 (A wr)	0 (D rd)	1 (A wr)	1	0 (D rd)	1 (A wr)
*s101	0	0	1 (A wr)	0 (D rd)	1 (A wr)	1	0 (D rd)
*s200	0	1 (B wr)	0 (C rd)	0	0	1 (B wr)	1
*s201	0	0	0	1 (B wr)	0 (C rd)	0	1 (B wr)
*s210	0	0	0	0	1 (B wr)	1	1
*s300	0	0	1 (C wr)	1 (D rd)	1	1	1
*s301	0	0	0	0	1 (C wr)	0 (D rd)	0
*s310	0	0	0	0	0	0	0
*s400	1 (init)	1	0 (C rd)	0	0	1 (D wr)	1
*s401	0	0	0	1 (D wr)	0 (C wr)	0	0

Fig. 12. Simulation of Lookup-based GALS Architecture

$$\sum_{i=1}^{4} 32 * sz_i = 32 * (2 + 3 + 3 + 2) = 3200 bits = 400 bytes.$$

Figure 12 shows how the addresses are maintained in each component's SMU and how they change based on the arrival of clocks. As discussed earlier, the data is accessed (read for inputs and written for outputs) based on the local addresses. The SMU knows the appropriate segment where the addressees reside. In the example shown in Figure 12, we represent each address as *[signal name][location] for ease of readability. For instance, *s301 points to the appropriate location where the data of signal s3 is stored along with its offset 01. After every read/write by a component, the offset is incremented by 1 modulo bound of the signal. Also, for simplicity, the lookup storage structure shown only illustrates the presence (1) and absence (0) bits for the data. In actual storage, the data is read and written to these locations, along with the appropriate assignment of presence and absence bit to the MSB.

Pros and Cons: One of the main advantages of this approach is that more than one data values can be stored in lookup storage as compared to the previous architecture where only one valid value can be placed on the components output. The second advantage of this approach is that the throughput of this architecture would be high it ensure higher parallelism that the handshake based GALS architecture and the controller based GALS architecture. Another advantage of this approach is that there are no inter-component signals, hence keeping the design simple. Most of the overhead is involved in accessing the lookup storage. There are various areas where this overhead can be reduced. One such example is that when the data is retrieved from the storage and it has the presence bit, but other conditions are not satisfied for its computational block to execute, then this data can be stored in the SMU's local storage. Accessing the same storage location twice for the same data is unnecessary. The same can be applied when accessing the location for reading data from the output address. If the *present* is 0, then we know that no other component will write to the same address, and hence this bit will not be set to 1 by any other component. However, many other components can read from the same location. This type of architecture is similar to the dynamic dataflow architecture where a token matching scheme is implemented and data is retrieved from memory.

8 Comparison of the four architectures

Table 2 illustrates the execution of the components of our example for the four different architectures based on the clocks considered. Performance of the entire system can be analyzed based on how the components execute, and how many times the components execute. It can be realized that the handshake-based (using four-phase handshake) architecture had the worst performance, as each component executed twice in order to communicate one value across. Using a two-phase handshake for the handshake-based architecture would have improved this performance, but complicated the architecture. The drawback with the handshake based architecture is that there are twice more signals for each signal in the network.

Architecture	t_1	t_2	t_3	t_4	t_5	t_6
Handshake (4-phase)	A,B	C	D	-	B	A
FIFO-based	A,B	A,C	B,D	A,B,C	B,D	A,B
Controller-based	A,B	C	B,D	A,C	B,D	A
Lookup-based	A,B	A,C	A,B,D	A,B,C	B,D	A,B

Table 2
Execution of Components in Different Architectures

The performance of the controller based architecture was better than that of the handshake based architecture but worse than the FIFO-based and lookup-based architectures. Each component has signals going back and forth to the CCU, therefore for such an architecture, two signals are added to communicate with the CCU.

Next, we compare the FIFO-based architecture and the lookup-based architecture. In terms of performance, the lookup-based architecture is better, as each component does a fetch on its clock and stores the corresponding data in its local storage. As a result, the data in the storage has already been read. However, the main overhead for this approach is that there are many reads and writes to the storage for each component. In the case of FIFO-based architecture, the FIFOs are placed in-between the components. Similar to the handshake-based architecture, each signal is associated with two additional signals (valid and stall). However, the encapsulation of the computation block includes a barrier synchronizer, which functions the same as a join. Table 3 illustrates the overhead associated with the four architectures.

	Computation Complexity	Signal Overhead (n signals & m components)	Communication Media
Handshake-based	RTU	2*n	-
FIFO-based	IIP and OIP	4*n	FIFOs
Controller-based	LCU	2*m	CCU
Lookup-based	SMU	0	Lookup Storage

Table 3
Overhead Associated with GALS Architectures

The handshake-based GALS architecture should be chosen as the target architecture when there is a constraint on adding additional elements such as communication media (Table 3). Here, the cost associated with additional signals such as placement and routing is not an issue. FIFO-based architecture is a good choice as the target architecture for GALS, if additional signals can be added easily with FIFOs. Such an architecture would be best for performance driven applications. The controller-based GALS architecture is better if there is a constraint on number of signals can be added, and the ratio of the components in the design over the number of inter-component signals is higher. Hence, less number of signals will be added in this architecture than the handshake-based architecture. If addition of extra storage elements on the chip is not an issue, and storage accessing time is assumed to be little, then the lookup-based GALS architecture is best. It was realized by the example that the Lookup-based GALS architecture had the best performance if there are no constraints for additional elements/signals on the chip, and the accessing time was assumed to be negligible.

9 Conclusion & Future Work

In this work, we promote the idea of using KPN as the model of computation for designing GALS. We provide a design methodology for GALS with the focus of this paper on architectural exploration. We borrow ideas from existing dataflow architectures, and use them in our GALS architectures. We illustrate four different architectures for implementing GALS with a running example. We show the overhead complexity involved in these architectures. We do not discuss on issues involving meta-stability, and cross domain synchronization, as the focus of this paper is on exploration of different GALS architectures. The underlying formalism of our framework and identifying the formal properties associated with the proposed refinements are part of our on-going work. Furthermore, the proof obligations for the proposed refinement schemes in terms of their correctness will be established in our future work.

References

[1] M. Bohr. Interconnect scaling - The real limiter to high performance VLSI. *In IEEE Int. Electron Devices Meeting*, pages 241–244, 1995.

[2] E. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[3] J. Dennis. First version of a data flow procedure language. In G. Goos and J. Hartmanis, editors, *Proceedings of the Programming Symposium*. Springer-Verlag, 1974.

[4] Arvind and R. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[5] L. Carloni and A. Sangiovanni-Vincentelli. Coping with latency in SoC design. *In IEEE Micro, Special Issue on Systems on Chip*, 22(5):12, October 2002.

[6] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, 2001.

[7] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd annual conference on Design automation*, pages 657–662. ACM Press New York, NY, USA, 2006.

[8] S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla. Validating families of latency insensitive protocols. *IEEE Transactions on Computers*, 55(11):1391–1401, 2006.

[9] T. Grotker. *System Design with SystemC.* Kluwer Academic Publishers Norwell, MA, USA, 2002.

[10] A. Gerstlauer. *System Design: A Practical Guide with SpecC.* Kluwer Academic Publishers, 2001.

[11] W. Acherman and J. Dennis. VAL-ORIENTED ALGORITHMIC LANGUAGE, PRELIMINARY REFERENCE MANUAL. Technical Report MIT-LCS-TR-218, MIT, 1979.

[12] A. Davis. The architecture and system method of DDM1: A recursively structured Data Driven Machine. In *Proceedings of the 5th annual symposium on Computer architecture*, pages 210–215. ACM Press New York, NY, USA, 1978.

[13] T. Chelcea and S. Nowick. Robust interfaces for mixed-timing systems. *IEEE Transactions on VLSI*, 12(8):857–873, Aug 2004.

[14] L. Carloni. The Role of Back-Pressure in Implementing Latency-Insensitive Systems. *Electronic Notes in Theoretical Computer Science*, 146:61–80, 2006.

[15] D. Kim, M. Kim, and G. Sobelman. Asynchronous FIFO Interfaces for GALS On-Chip Switched Networks. In *International SoC Design Conference*, 2005.