Abstract -- Composing synchronous intellectual property (IP) blocks over asynchronous communication links for an System-On-Chip (SoC) design is a challenging task, especially for ensuring the functional correctness of the overall design. In this paper, we propose a trace based framework to assist in validation of globally asynchronous locally synchronous (GALS) designs. We provide a specific characterization of synchronous IPs in our framework such that no specific protocol is required for asynchronous communication between them. We theoretically show that IPs with single activation property, composed asynchronously, are behaviorally equivalent to those composed synchronously.
A Trace Based Framework for Validation of
SoC Designs with GALS Systems

Syed Suhaib       Deepak Abraham       Sandeep Shukla
Contents

1 Introduction and Related Work 2
2 Formal Theory 3
3 Validation of SoCs with GALS 10
4 Case Study: FIR Filter 14
5 Conclusion 16

List of Figures

1 FIR filter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
Abstract

Composing synchronous intellectual property (IP) blocks over asynchronous communication links for an System-On-Chip (SoC) design is a challenging task, especially for ensuring the functional correctness of the overall design. In this paper, we propose a trace based framework to assist in validation of globally asynchronous locally synchronous (GALS) designs. We provide a specific characterization of synchronous IPs in our framework such that no specific protocol is required for asynchronous communication between them. We theoretically show that IPs with single activation property, composed asynchronously, are behaviorally equivalent to those composed synchronously.
1 Introduction and Related Work

Intellectual property (IP) reuse is gaining increasing importance in the current and upcoming System-on-a-Chip (SoC) designs. Reuse of pre-existing components such as memories, processor cores, and dedicated hardware blocks chosen from an IP library seems to be the only way to mitigate the productivity crisis and shortening time-to-market cycles [1]. Therefore, the correct composition of these existing IP blocks has been a challenging task for an SoC design. However, the ever increasing clock frequencies invalidate the synchrony assumption between IP blocks due to long interconnects [2, 3, 4]. As a result, globally asynchronous locally synchronous (GALS) designs have emerged to face such challenges for an SoC design [5, 6]. A GALS design has the advantage of incorporating synchronous designs with asynchronous communication. However, there is a lack of good theoretical models and methodologies for designing GALS systems due to the fact that they involve both synchrony as well as asynchrony together.

To clearly analyze the asynchrony aspects of GALS, a theory of desynchronization, which involves removing the notion of clocks for a design, has been proposed in the past [5]. Also, various characterization of synchronous systems have surfaced to achieve a GALS design involving the notions of endochrony and isochrony [7]. These notions are tied to the synchronous programming language, SIGNAL [8], which mainly targets embedded software systems. Also, using SIGNAL, Mousavi et. al proposed a formal framework for correctly transforming a GALS design into a synchronous design for modeling and validation purposes [9].

In this paper, we provide a trace based framework for design and validation of SoCs with a GALS design. Our goal is to provide designers with a framework, where synchronous IP designs can be clearly understood based on their traces,
and the communication between them can be modeled asynchronously. Furthermore, in our framework, we provide a characterization of synchronous IPs, where IPs composed either synchronously or asynchronously are behaviorally equivalent. We theoretically show the behavioral equivalence for both the compositions. For our theoretical analysis, we propose the notions of declocking and reclocking which are closely related to the theory of desynchronization. Also, our notion of single activation, closely relates to endochrony and isochrony, and is applicable to SoC designs. Our single activation means that a system is associated with a single clock that dictates its execution. Our characterization guarantees that a process is latency equivalent to its declocked version. By latency equivalent, we mean that the sequence of events carrying valid data on the output signals is same [3]. Furthermore, reclocking of such a declocked process will yield the exact same behavior as of the original process. Since, asynchronous composition of synchronous IPs is behaviorally equivalent to their synchronous composition, no specific protocol is needed for facilitation of asynchronous communication. We use the example of an FIR filter to show how trace based framework can be used for validation and designing of a GALS system.

2 Formal Theory

Let \( \mathcal{V} \) denote the universal set of all variables. Each variable \( v \in \mathcal{V} \) can be assigned a data value depending on its type, on an occurrence of an event \( (e) \). Let \( D \) be the set of all data values and, \( T \) be a partially ordered set (set of tags). An event is defined as \( e \in (D \times T) \cup \{\tau\} \), where \( e \) is informative when \( e \in D \times T \),

\[ 1 \]We formally define latency equivalence in the following section
otherwise \( e = \tau \) is an absent event. For a variable \( v \), an occurrence of an informative event \( e \), with a data value \( d \in D \) and a tag \( t_j \in T \) such that \( e = (d,t_j) \), will assign the data value \( d \) to \( v \) at time \( t_j \). For each variable, a series of data values are assigned which forms a trace. The trace of a variable shows the order in which values are assigned to a variable. This order of values assigned depends on events caused due to an input or result of some functionality. A trace, in other words, is an observation of the values of variables. A trace can be finite or infinite.

**Definition 2.1** An unclocked trace \( (\mathcal{E}) \) is a sequence of informative events whose occurrence is associated with a variable \( v \). Consider \( \mathcal{E}(v) = e_1 e_2 \ldots \) where \( e_1 e_2 \ldots \) are informative events.

We now analyze a trace in a clocked domain, where events occur based on a clock\(^2\). In the clocked domain, the sequence number of the events correspond to a clock interval, which represents an observation point of an event. If we consider an observer who can look at all the individual clocks and their ticks, and put them in a global time scale, then each individual clocks can be analyzed w.r.t that time scale. The clock of the variable is inferred by its trace. The clock of a variable \( v \) is denoted by \( \hat{v} \). In each clock interval of a variable, an event may occur. Let \( C \) be the set of all the clocks that can be associated to variables. For any variable \( v \), its clock \( \hat{v} \) contains the set of time instants at which its events occur. For example, \( \hat{v} = \{t_1,t_5,t_6\} \) denotes the time instance at which events of variable \( v \) occur.

We define a clocked trace as follows:

**Definition 2.2** A clocked trace \( (\mathcal{E}) \) is a sequence of either informative or absent events whose occurrence is associated with a variable \( v \). Consider \( \mathcal{E}(v) = \ldots \)

\(^2\)Given a timeline, a clock is the division of timeline into slots using boundaries
where events are either informative or absent.

Example 2.3 A clocked trace of variable $v_1$ with its clock \{\(t_1, t_3, t_5, t_7\)\} is represented as:

\[
\begin{array}{cccccc}
t_1 & t_2 & t_3 & t_4 & t_5 & t_7 \\
\end{array}
\]

\[
\sigma(v_1): e_1 \tau e_2 \tau e_3 \tau e_4
\]

For a trace $\sigma$ of variable $v$, $\sigma(v)[i]$ denotes the $i^{th}$ event of the trace of $v$. For the above example, $\sigma(v_1)[5] = e_3$. An empty trace is a trace of a variable which is not associated with any informative event, and is represented by $\varepsilon$. An append operator ($\oplus$) is used to illustrate a trace creation. For example: $e_1 e_2 \oplus e_3 = e_1 e_2 e_3$. The same applies to unclocked traces as well.

A process is a collection of traces, and can be classified as synchronous or asynchronous. A synchronous process has all its events indexed by a single clock. New events are computed for all its variables at every clock. A synchronous process can be realized by a synchronous observation defined as follows.

Definition 2.4 Synchronous Observation: An observation of a synchronous process, $P$, is defined as a 3-tuple $\langle V, \Sigma, \hat{c} \rangle$, where $V$ is the variable set of $P$, $\Sigma$ contains traces of all variables in $V$ and $\hat{c}$ is the clock associated all the variables in $V$, i.e $\forall v \in V$, $\hat{v} = \hat{c}$.

Example 2.5 Consider a synchronous increment process $\langle V, \Sigma, \hat{c} \rangle$, where $V = \{v_1, v_2\}$ and $\Sigma = \{\sigma(v_1), \sigma(v_2)\}$. The increment process is defined as follows: $v_1 = x$ and $v_2 = f(x) = x + 1$. If the trace associated with $\sigma(v_1) = e_1 \tau e_2 \tau e_3$ with events $e_1 = (5, t_1), e_2 = (7, t_3), e_3 = (8, t_5)$ then $\sigma(v_2) = e_1 \tau e_2 \tau e_3$ with $e_1 = (6, t_1), e_2 = (8, t_3), e_3 = (9, t_5)$. 

5
Next, we look at *asynchronous* observations. An asynchronous observation can either be of a synchronous process or an asynchronous component such as a process that models memories, buffers, or asynchronous FIFOs with handshakes. An *asynchronous process* is not associated with any clock. Such a process reacts whenever an input is received. In the asynchronous observation, the information of clock is absent, and only informative events are present in the trace. Therefore, in an asynchronous observation, unclocked traces are considered for each variable.

**Definition 2.6** Asynchronous Observation: *An asynchronous observation for a process written as (\( \mathcal{P} \)) is a two-tuple \( \langle V, \Sigma \rangle \), where \( V \) is the set of all variables of \( \mathcal{P} \) and, \( \Sigma \) contains all unclocked traces \( (\mathcal{S}) \) of variables in \( V \).*

We define two forms of composition: asynchronous composition and synchronous composition. We first look at asynchronous composition of asynchronous observations.

**Definition 2.7** The asynchronous compositions of asynchronous observations \( (\||^a) \) for processes, \( \mathcal{P}_1 \) and \( \mathcal{P}_2 \) is defined as follows:

\[
\mathcal{P} = \mathcal{P}_1 \||^a \mathcal{P}_2 = (V, \Sigma), \text{where }
\begin{align*}
V &= V_1 \cup V_2 \\
\Sigma &= \Sigma_1 \land^a \Sigma_2
\end{align*}
\]

where, \( \land^a \) denotes the conjunction of unclocked traces of a variable defined as follows:

\[
\forall v \in (V_1 \cap V_2) : \mathcal{S}(v) = \mathcal{S}_1(v) = \mathcal{S}_2(v)
\]

\[
\forall v \in (V_1 \setminus V_2) : \mathcal{S}(v) = \mathcal{S}_1(v)
\]
The asynchronous composition (∥a) operator is associative and commutative. For transforming a synchronous observation to an asynchronous observation, we define *declocking* as follows:

**Definition 2.8** Declocking: ⟨V, Σ, ĝ⟩ → ⟨V, Σ⟩. *It involves using the asyntrace function, where ∀v ∈ V, asyntrace : Σ → Σ, which transforms σ(v) ∈ Σ to σ2(v) ∈ Σ. The clock associated with the variables is removed during this transformation. The asyntrace function is defined as follows: asyntrace(σ(v)) = un(σ(v), 1), where*

\[
un(σ(v), i) = \begin{cases} 
un(σ(v), i + 1), & \text{if } (σ(v)[i] = τ) \\
σ(v)[i] @ un(σ(v), i + 1), & \text{otherwise}
\end{cases}
\]

Declocking removes all the absent events from the clocked traces of a synchronous observation. As a result, we get an unclocked trace for each clocked trace. Next, we define *reclocking*, such that given a clock, it transforms an asynchronous observation to a synchronous observation. *Reclocking* is defined as follows:

**Definition 2.9** Reclocking: ⟨V, Σ⟩ × C → ⟨V, Σ, ĝ⟩ For any c ∈ C, ∀v ∈ V, we define syntrace : Σ × c → Σ where

\[
syntrace(σ, c) = f(σ, c, 1) and
\]

\[
f(σ, c, i) = \begin{cases} 
σ[i] @ f(σ, c, i + 1), & \text{if } (t_i ∈ c) \\
t @ f(σ, c, i + 1), & \text{otherwise}
\end{cases}
\]
The function *synctrace* is applied to all unclock traces to transform them to clocked traces. The function places events of an unclock trace with respect to a clock provided, to form a clock trace. An absent event is placed when no event is present for a clock interval.

Until now, we have defined declocking to transform a synchronous observation to an asynchronous observation, and reclocking to transform an asynchronous observation back to a synchronous observation when a clock is provided. Now, we define synchronous composition of synchronous observation, where each observation may have a different clock. For a synchronous composition of two processes, one clock is selected for the resultant process and enforced for both the processes. Hence, declocking and reclocking has to be performed as shown below with the selected clock to synchronize them and obtain a synchronous observation.

**Definition 2.10** Synchronous Composition of Synchronous Observations (∥) The synchronous composition of synchronous observations for processes, *P*₁ and *P*₂ is defined as follows:

\[
P = P₁ ∥ P₂ = \langle V, Σ, ˆc \rangle, \text{where } \begin{cases} 
V = V₁ \cup V₂ \\
Σ = Σ₁ \land ˆc Σ₂ \\
ˆc = ˆc₁ | ˆc₂
\end{cases}
\]

and \( \land ˆc \) denotes the conjunction of clocked traces of each variable. Since, all variables in a synchronous process have the same clock, we need to transform the traces of all variables from one process to another depending on whose clock is chosen. Considering *P*₁’s clock is chosen (\( ˆc = ˆc₁ \)). Then,

\[
\forall v \in V₁ : \overline{σ}(v) = \overline{σ₁}(v)
\]
∀v ∈ (V_2) : \sigma(v) = syntrace(asyntrace(\sigma_2(v)), \hat{c}_1)

The synchronous composition operator (||) is commutative and associative. The communication between two observations happens on shared variables. Now, when these traces are composed together, the clock for the traces of the shared variable in one of the observations will be updated along with rest of the variables. Consider the following example:

**Example 2.11** We compose the increment process (defined earlier) with an isEven process, \(\langle V, \Sigma, \hat{c} \rangle\), and \(V = \{v_2, v_3\}\) and \(\Sigma = \{\sigma(v_2), \sigma(v_3)\}\). The isEvent process is defined as follows: \(v_2 = y\) and \(v_3 = g(y) = \text{if } (y \mod 2 = 0) \text{ then } true \text{ else } false\).

The composed process will have three variables with \(v_2\) as the common variable. The trace of \(v_2\) is \(\sigma(v_2) = e_1 \tau \tau e_2 \tau \tau e_3\) with \(e_1, e_2, e_3\) being \((6, t_1), (8, t_4), (9, t_7)\) and \(v_3\) is \(\sigma(v_3) = e_1 \tau \tau e_2 \tau \tau e_3\) with its events \(e_1, e_2, e_3\) as \((true, t_1), (true, t_4), (false, t_7)\).

Now, assuming the clock of the increment process is chosen to be the clock of the composed process, we declock and reclock the traces of isEvent process giving \(\sigma(v_2) = e_1 \tau e_2 \tau e_3\) with \(e_1, e_2, e_3\) as \((6, t_1), (8, t_3), (9, t_5)\) and \(\sigma(v_3) = e_1 \tau e_2 \tau e_3\) with its events \(e_1, e_2, e_3\) as \((true, t_1), (true, t_3), (false, t_5)\). The clock of the composed process is same as the clock of increment process.

We have now presented our trace based framework, where both asynchronous as well as synchronous aspects of an SoC can be modeled. In the next section, we define various notions and characterizations needed for correctness validation of SoCs with GALS, where IPs are composed asynchronously.
3 Validation of SoCs with GALS

In this section, we show that IPs composed asynchronously are functionally equivalent to IPs composed synchronously given our characterization. Our correctness criterion is based on the notion of latency equivalence, which is defined as follows:

**Definition 3.1** Latency Equivalent ($\equiv_e$): Two traces, $\sigma_1$ and $\sigma_2$, are said to be latency equivalent iff $\text{asyntrace}(\sigma_1) = \text{asyntrace}(\sigma_2)$.

This notion can be extended to processes. Two processes are said to be latency equivalent iff their output is latency equivalent when both are provided the same inputs. In our framework, we describe a process in terms of its observation, hence for all common variables for each trace in one observation, if there exists a unique trace in the other observation, then the two observations are said to be latency equivalent.

**Lemma 3.2** A synchronous observation of a synchronous process is latency equivalent to an asynchronous observation of the same process.

**Proof sketch:** For a synchronous observation, the order of informative events for each clocked trace remain the same after the $\text{asyntrace}$ function is applied to it. We know that application of function $\text{asyntrace}$ to a clocked trace gives a unique unclocked trace. Hence, we get all unique unclocked traces for a synchronous process. Also, for each trace in the synchronous observation of a synchronous process, there exists a unique trace in its asynchronous observation. Therefore, the unclocked traces from the application of the function $\text{asyntrace}$ are the same as the unclocked traces of the asynchronous observation. Thus, this implies that two
observations are latency equivalent. In other words, a synchronous observation and its declocking (i.e. its asynchronous observation) are latency equivalent.

We now define the notion of correctly declocking a synchronous process. We say that a synchronous process can be correctly declocked if the following condition holds:

Consider two clocked processes $P_1, P_2$ and their corresponding unclocked processes $\overline{P}_1, \overline{P}_2$, then

\begin{align*}
\text{Condition 1: } P_1 = P_2 &\iff \overline{P}_1 = \overline{P}_2
\end{align*}

Condition 1 states that given two clocked processes, if their traces in the clocked domain are equal then they can be uniquely declocked and their traces in the unclocked domain will be equal. Also, given two processes in the unclocked domain, they can be transformed to clocked processes uniquely given a clock, and therefore will have the same traces. Before we check the condition for declocking, we first define the single activation property:

**Property 3.3 Single Activation:** For a clocked process, $\forall v \in V, \hat{v}_i = \hat{v}_j$.

The single activation property means all the variables in a synchronous process are updated simultaneously. For the associated clock, the absence of events occurs at the same timestamps for all variables.

We now prove that if property 1 holds, then condition 1 is true. We consider two cases:

**Case 1 Declocking** $P_1 = P_2 \Rightarrow \overline{P}_1 = \overline{P}_2$: If L.H.S (synchronous) is equal then R.H.S. (asynchronous) is also equal. The proof of this condition is trivial due to the single activation property. The removal of clock from both processes, $P_1$ and $P_2$, results in removal of absent events from the clocked traces of $P_1$ and $P_2$, makes
them equal in the unclocked domain.

Case 2 Reclocking $P_1 = P_2 \iff \overline{P}_1 = \overline{P}_2$: Here, we consider the reverse that if two processes, $\overline{P}_1$ and $\overline{P}_2$, are equal in the unclocked domain, then their synchronous processes are also equal. We use our reclocking function to recreate the clocked trace. For any $e \in C, \forall v \in V$, syntrace is applied. Application of syntrace to $\overline{P}_1$ and $\overline{P}_2$ results in insertion of events for a variable at the same timestamp when the clock is present. Hence, for both processes created $P_1$ and $P_2$, the number of events inserted and the timestamp of insertion are the same. Therefore, $P_1 = P_2$ if $\overline{P}_1 = \overline{P}_2$. Hence, condition 1 satisfies for synchronous processes.

Therefore, we can conclude that for any synchronous process, its corresponding asynchronous observation can be correctly formed. Implication: For a $\sigma \in \Sigma$ there exists a unique $\overline{\sigma} \in \overline{\Sigma}$. We conclude that given this transformation, an observation of any synchronous process can be correctly seen in an unclocked domain. By correctness here, we imply that the traces in the clocked and the unclocked domain are latency equivalent.

We have shown that given two processes with single activation property, the asynchronous observations and synchronous observations are latency equivalent. Therefore, asynchronous composition of such processes will also be latency equivalent to their synchronous composition. Therefore, no specific protocol is required for their asynchronous composition.

We now define a system, which is a composition of processes.

**Definition 3.4** System: A system is a composition of processes defined as $\Phi = P_1^* \parallel^* P_2^* \parallel^* \ldots \parallel^* P_n^*$, where $P^* = P | \overline{P}$ and $\parallel^* = \parallel \parallel^a$.

A system is a composition of observations of processes, where the composition is commutative and associative. Next, we look at declocking a system.
To correctly declock a synchronous system, it must have the property of single activation. Consider a synchronous system $\Phi = P_1 \parallel P_2$ and its declocking $\overline{\Phi} = \overline{P}_1 \parallel^a \overline{P}_2$. The following conditions must satisfy:

1. **Declocking a synchronous system:** For a system $\Phi$, its $\overline{\Phi}$ can be uniquely formed. To prove this, we will consider $P_1 = \langle V_1, \Sigma_1, \hat{c} \rangle$ and $P_2 = \langle V_2, \Sigma_2, \hat{c} \rangle$. (1) $\forall v_i \in V_1$, a unique $\overline{\sigma}(v_i)$ can be formed from $\sigma(v_i)$. This is true because of single activation property. (2) Similarly, $\forall v_j \in V_2$, a unique $\overline{\sigma}(v_j)$ can be formed from $\sigma(v_j)$. (3) $\forall v \in V_1 \cap V_2$, $\overline{\sigma}(v)$ can be formed uniquely. To prove this, we know that for a synchronous composition, $\forall v_i, v_j \in V$ if $\sigma(v_i) = \sigma^*$ and $\sigma(v_j) = \sigma^*$, then $v_i = v_j$. Now, we consider $v_x, v_y \in V_1 \cap V_2$ and $v_x \neq v_y$. Therefore, $\sigma(v_x) \neq \sigma(v_y)$.

Since, the position of absent events in the two traces is the same, removal of absent events from the trace would yield $\overline{\sigma}(v_x) \neq \overline{\sigma}(v_y)$ as the events on both traces will be different. Therefore, $\forall v \in V_1 \cap V_2$, $\overline{\sigma}(v)$ will be unique. Hence, $\Phi$ can be uniquely declocked to $\overline{\Phi}$.

2. **Reclocking a synchronous system:** For a system $\overline{\Phi}$ given a clock $c$, its $\Phi$ can be uniquely formed. The proof of this is similar to the previous one for declocking. Considering the three cases: (1) $\forall v_i \in V_1$, a unique $\sigma(v_i)$ can be formed given $c$ from $\overline{\sigma}(v_i)$. Give the clock $c$, a unique $\sigma(v_i)$ can be formed using the function $\text{syntrace}$. (2) Similarly, $\forall v_j \in V_2$, a unique $\sigma(v_j)$ can be formed from $\sigma(v_j)$ and clock $c$. (3) $\forall v \in V_1 \cap V_2$, $\sigma(v)$ can be formed uniquely. For $v_x, v_y \in V_1 \cap V_2$ and $v_x \neq v_y$, we get $\overline{\sigma}(v_x) \neq \overline{\sigma}(v_y)$. The timestamps at which the absent events are inserted is same for both traces. As the events for both traces are different, we get $\sigma(v_x) \neq \sigma(v_y)$ and hence are unique. Therefore, the asynchronous system can be reclocked uniquely given a clock. Hence, we formulate the following theorem.

**Theorem 3.5** If all the processes of a system have a single activation property,
then the processes can be asynchronously composed for GALS without any protocol.

4 Case Study: FIR Filter

![FIR filter diagram](image)

Figure 1: FIR filter

The FIR filter system consists of three different modules: Stimuli, FIR and Display (Figure 1). We analyze the three IPs independently in our trace based framework. For all the three modules, we show the abstract behavior represented by variables. The Stimuli module when given a trigger as input generates a valid data. This data generation takes four cycles. For Stimuli module, we represent its behavior by three variables: \( \text{trigger} \), \( \text{Sticycle} \) and \( \text{datastart} \). The variable \( \text{trigger} \) is a boolean (T or F) type, and is used to trigger the Stimuli module. The variable \( \text{Sticycle} \) is used to illustrate the reset cycles, and finally the variable \( \text{datastart} \) is the data output. When a trigger is received, the \( \text{Sticycle} \) starts, and at the fourth cycle, the valid data, denoted by \( \text{vdata} \) is seen on the variable \( \text{datastart} \). The trace of the variables of the Stimuli IP is shown below:

\[
\begin{align*}
\sigma(\text{trigger}) : & \quad T \quad \tau \quad \tau \quad \tau \\
\sigma(\text{Sticycle}) : & \quad 1 \quad 2 \quad 3 \quad 4 \\
\sigma(\text{datastart}) : & \quad \tau \quad \tau \quad \tau \quad \text{vdata}
\end{align*}
\]
It can be realized that the module does not have the single activation property, since the clocks of variables trigger, Sticycle and datastart are not the same. As this module does not have single activation, we cannot declock this module in our framework. We now assume that all the modules possess single activation. Therefore, to make this module single activation, we consider all variables of an IP have a valid value at time stamp whenever any variable has a valid value. Therefore, for the variable trigger, we consider false (F) instead of τs to ensure that it has a valid value which is false. Also for the variable datastart, we consider 0 to be an invalid value, but the value is not absent. The invalid values are discarded in implementation. The new trace is shown below:

<table>
<thead>
<tr>
<th></th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
</tr>
</thead>
<tbody>
<tr>
<td>σ(trigger) :</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>σ(Sticycle) :</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>σ(datastart) :</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>vdata</td>
</tr>
</tbody>
</table>

Next, we consider the FIR module where the computation occurs in ten cycles. We consider three variables for the FIR module: datastart, FIRcycle and compdata. The datastart variable either has valid or invalid data (which is 0), or the data is absent. The FIRcycle variable represents the computation cycles. We abstract out the computation of the FIR, and represent only its computational cycle to make the example easy to understand. The compdata variable has a valid value when the computation is complete, denoted by cdata, otherwise it is 0 or absent. The valid cdata can be seen on variable compdata ten cycles after vdata is seen on variable datastart. The trace for the FIR module is shown below:
Similarly, we can denote the traces of the variable of the display module. Due to lack of space, we do not show its traces.

Now, we consider their compositions. For the Stimuli and FIR module, we realize that the common variable $datastart$ has the same trace. Therefore, the two modules are composed and communication happens on this shared variable. Since, both these modules have the property of single activation, their synchronous composition also has single activation\(^3\). Similarly, the Display module is composed with Stimuli and FIR module. We get a design of these three modules interaction, which also has the property of single activation. Given this property, from theorem 3.5, this system can be correctly implemented as a GALS system. The traces of the variables and their composition aids in validating the design.

Also, due to the availability of various verification tools for synchronous designs, we can formally verify the synchronous version of this SoC, and its correct GALS implementation can be implemented without the need for verifying this GALS design.

5 Conclusion

In this work, we provide a trace based approach for design and validation of an SoC. Our framework allows modeling of synchrony as well as asynchrony. We

\(^3\)Follows from the definition of synchronous composition
formally show that if a synchronous design has a single activation property, then it can be correctly implemented as a GALS design. Our characterization is theoretically shown to be correct. As a result of our characterization, no specific protocol will be needed for asynchronous communication. We show the example of an FIR filter in our framework.

References


and validating globally asynchronous design in synchronous frameworks. In 