Design and Verification of On-Chip Communication Protocols

Peter Böhm
Oxford University Computing Laboratory,
Wolfson Building, Oxford, OX1 3QD, England
Email: peter.boehm@comlab.ox.ac.uk

Tom Melham
Oxford University Computing Laboratory,
Wolfson Building, Oxford, OX1 3QD, England
Email: tom.melham@comlab.ox.ac.uk

***Abstract*—Modern computer systems rely more and more on on-chip communication protocols to exchange data. To tackle performance requirements these protocols have become highly complex, which makes their formal verification usually infeasible with reasonable time and effort.**

We present an initial case study for a new approach towards the design and verification of on-chip communication protocols. This new methodology combines the design and verification processes together, interleaving them in a hand-in-hand fashion.

In our initial case study we present the design and verification of a simple arbiter-based master-slave communication system inspired by the AMBA High-performance Bus architecture. Starting with a rudimentary, sequential protocol, the design is extended by adding pipelining and burst transfers. Both extensions are realized as transformations of a previous version such that the correctness of the former leverages the verification of latter. Thus, we also argue about the correctness of both extended designs.

I. Introduction

Modern computer systems rely more and more on highly complex on-chip communication protocols to exchange data. The enormous complexity of these protocols results from tackling high-performance requirements. Protocol control can be distributed, and there may be non-atomicity or speculation. Moreover, different system components may have separate clocks or adjustable clock frequencies, requiring asynchronous communications. These complexities arise in many important circuit design areas, such as multicore architectures, system-on-chip, or network-on-chip designs.

Whereas the efforts of chip manufacturers to validate or even formally verify their designs has increased over the last years, the complexity of communication protocols makes their formal verification usually infeasible within reasonable time and effort.

We present a new approach towards the design and formal verification of on-chip communication protocols. The approach can be summarized as follows. We start with a design model for a basic protocol that can be formally verified with reasonable effort; this is then extended with advanced features step-by-step to meet performance demands, handle asynchronous communication, or improve fault-tolerance. These extensions are realized by mathematical transformations of a previous design, rather than constructing a new design. The correctness of an extended design is obtained from the correctness of the previous version and either the transformation itself (*correctness-by-design*) or a refinement or simulation relation between the two versions.

Using this approach, the verification is interleaved with the protocol design process in a hand-in-hand fashion. The task of obtaining the next, more advanced design then splits into three main challenges: (i) is there an algebraic model to derive the extended design from the previous one? (ii) how does the refinement or simulation relation between them look? (iii) how does the correctness statement need to be modified during the design steps?

Verification by stepwise refinement is, of course, not new (cf. Section I-A). The novelty of our approach lies in the application of this methodology to protocol verification, and in the technical details of the specific optimising transformations we propose to investigate. Given a clean algebraic model for the three parts mentioned above, this leads to a new methodology combining design and verification into a single process. This approach is new, to the best of our knowledge, with respect to high-performance on-chip communication protocol design and verification.

This paper presents an initial case study demonstrating the extension of a simple, rudimentary communication protocol with pipelining and burst transfer features. The basic protocol is a sequential arbiter-bases master-slave communication protocol inspired by the AMBA High-performance Bus (AHB) architecture [1] but reduced to its basics. It supports sequential master-slave communication, including possible slave-side wait-states for memory system operations.

We formally specify a design model realizing the protocol using the Isabelle/HOL [2] theorem prover. Masters and slaves are specified at gate-level as state machines using transition functions. The arbiter is abstracted to a function providing bus grant signals to the masters and obeying a simple fairness property.

The verification is performed using a combination of interactive proofs with Isabelle/HOL and the open-source model checker NuSMV [3]. The NuSMV model checker is used via the oracle-based, domain-reducing interface IHaVeIt [4]. To argue about the behavior over time within the theorem prover, we had to transfer the model-checked temporal logic properties to cycle-accurate statements. This was done using a formalization of an execution trace semantics which relates a hardware cycle to a current state.

Finally, we describe transformations to the design model realizing pipelining and burst transfers of variable but known length. We formulate local and global correctness properties for the new designs and argue about their validity. The correctness is obtained from the correctness of the previous design and reasoning about the applied transformation. Hence, we show transformations that conserve correctness properties from its input design and provide *correctness-by-design* properties.

Even though our technical approach in this case study is modelling in higher order logic, and a combination of theorem proving with Isabelle and model checking with NuSMV, this is not necessarily our plan for future development of this work. We are primarily interested in capturing the right collection of primitive definitions and proof structures to support our planned refinement approach. Therefore, we are deliberately not starting with a pre-conceived notion of what language this future work will happen in, or what tool support we need to provide. Within the overall project, we are focusing on the right basic structuring principles for protocol descriptions with the specific features we are looking at. Choice of language and tools will come after, once we know what we need.

Organization of the paper: Next we discuss related work. In Section II we present the basic overall structure of our communication system and introduce notation used. The design and verification of the basic, sequential design is detailed in Section III. Afterwards, we present the transformation and correctness for pipelining in Section IV and for burst transfers in Section V. Finally, we conclude and outline future work in Section VI.

A. Related Work

Most existing work n this area of formal verification addresses the verification of specific protocols. For example, Roychoudhury *et al.* [5] present a formal specification of the AMBA protocol. They use an academic protocol version and verify design invariants using the SMV model checker. In [6] Amjad verifies latency, arbitration, coherence, and deadlock freedom properties on a simplified AMBA model. Schmaltz *et al.* present initial work [7] of a generic network on chip model as a framework for correct on-chip communication. They identify key constraints on architectures and show protocol correctness provided these constraints are satisfied.

All these approaches rely on a post-hoc protocol verification, which is a key difference to the methodology presented here. Even the framework presented in [7] relies on a post-hoc verification of protocol properties against their constraints. This verification approach becomes more and more infeasible, due to the complexity of modern communication protocols.

In [8] Müffke presents a framework for the design of communication protocols. It provides a dataflow-based language for protocol specification and decomposition rules for interface generation. These rules relate dataflow algebra and process algebra. Aside from noting that correct and verified protocol design is still an unsolved problem, Müffke does not address the verification aspect in general. He claims that the generated interfaces are correct by construction in the sense that if the generated interface is implemented correctly than the behavior of the components complies with the protocol specification. But he neither addresses the protocol correctness itself nor the verification of the implementation against the specification.

The basic idea behind our approach is similar to Intel's integrated design and verification (IDV) system [9]. The IDV system justifies its transformations by a *local proof* using simple equivalence checking. We expect that focusing on transformations tailored specifically for high-performance on-chip communication protocols result in more intricate refinement steps than can be handled by equivalence checking.

Verification by refinement or simulation relations is also used in many related areas. In [10] Aagaard *et al.* present a framework for microprocessor correctness statements based on simulation relations. Chatterjee *et al.* [11] verify a memory consistency protocol against weak memory models using refinement via model-checking. In [12] Datta *et al.* present a framework to derive verified security protocols using abstraction and refinement steps. They introduce a similar approach to ours but nevertheless neither of them deals with on-chip communication or even complexities arising from high-performance demands.

II. BASICS

In this Section we present the overall structure of our communication system and introduce basic notation used throughout the paper.

The presented design is an arbiter-based master-slave system inspired by the AHB architecture. A number of masters (specified by NS) are interconnected with a number of slaves (specified by NS) via a communication bus. Bus access on the master-side is granted by an arbiter. The bus itself consists of a ready signal plus separated control and data parts. Additionally, the masters are interconnected with the arbiter via signals to request and grant the bus.

The masters are connected to the bus using multiplexers controlled by the arbiter. The bus signals generated by the slaves are obtained via large or-trees. The bus is defined in Definition 1.

Definition 1 (Master-Slave Communication Bus) *The master-slave communication bus is defined as the tuple of signals between the master and slaves:*

$$\begin{aligned} bus^t &= (rdy^t, trans^t, wr^t, addr^t, wdata^t, rdata^t) \\ &\in (\mathbb{B}, \mathbb{B}, \mathbb{B}, \mathbb{B}^{ad}, \mathbb{B}^d, \mathbb{B}^d) \end{aligned}$$

where the components are:

- rdy *is the afore mentioned bus ready signal.*
- $trans$ *is the signal indicating either a idle transfer* (0) *or a data transfer* (1)*.*
- wr *denotes if a data transfer is a read* (0) *or write* (1) *transfer.*
- $addr$ *denotes the address bus of width* ad*. The address consists of a local address part (the lower* sl *bits) specifying the memory address within a slave and a device address (the upper* $ad - sl$ *bits) specifying the currently addressed slave.*
- $wdata$ *denotes the write data bus of width* d*.*
- $rdata$ *denotes the read data bus of width* d*.*

We refer to the last two components as data bus *and to the second to fourth components as* control bus.

Since the generation of the different bus signals has to be slightly modified during the presented transformations, they are defined in detail within the corresponding sections.

The basic, pivotal protocol characteristics can be summarized as: (i) every transfer consists of an address and a data phase, (ii) the end of each phase is defined by the bus signal rdy, and (iii) the bus is granted to some master at all times.

The first two properties are illustrated in Fig. 1. Every transfer starts with an address phase during which the signals on the control bus need to be generated. The

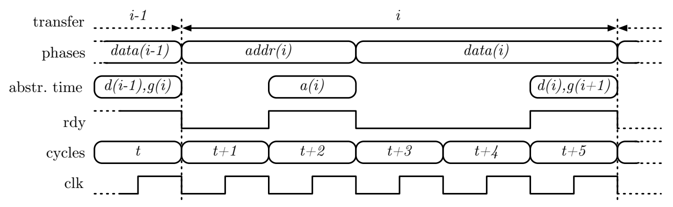

Fig. 1. Sequence of Sequential Transfers

address phase ends with an active rdy signal. At that time, the control bus signals have to be valid.

After the address phase, the data phase starts. It is again completed by an active rdy signal. In case of a write transfer, the master has to provide valid data on the $wdata$ bus during this phase. In case of a read transfer, the slave has to provide valid data on the $rdata$ bus at the end of this phase when the rdy signal is active.

The third characteristic is detailed as follows: the arbiter always grants the bus to some master. In case no master has requested bus access, the bus is granted to a default master $defM$. If a master is granted the bus but there is no data to transmit, the master has to initiate an $idle$ transfer. This transfer is distinguishable from a data transfer and a slave has to acknowledge with a zero wait-state response.

A. Notation

Throughout the next sections we use natural numbers $u \in [0 : NM - 1]$ and $v \in [0 : NS - 1]$ to refer to the u-th master and v-th slave and denote them with $m[u]$ and $s[v]$, respectively. When the number of the master or slave is irrelevant, we omit the u or v and use simply m or s to refer to any of them. We assume that $NM = 2^k$ for $k \in \mathbb{N}$ and $NS = 2^{ad-sl}$.

In order to specify the design they are related to, we index them with either seq for sequential, $pipe$ for pipelined, or $burst$ for burst design. When we refer to an arbitrary design, we omit the index.

We denote a bit vector $x \in \mathbb{B}^n$ with $x[n-1:0]$ and refer to its a-th bit with $x[a]$ where $a \in \{0, \dots, n-1\}$. For a bit vector $x \in \mathbb{B}^n$ we use the predicate $unary(x)$ to specify that x is a unary coded bit vector, i. e. $x[i] = 1 \iff x[j] = 0$ for all $j \neq i$. Moreover, we denote the value of x interpreted as a unary number by $\langle x \rangle_u$ and define it as $\langle x \rangle_u = a \iff x[a] = 1$. Analogously, we denote the value of a bit vector $x \in \mathbb{B}^n$ interpreted as a binary number by $\langle x \rangle$. It is defined by $\langle x \rangle = \sum_{i=0}^{n-1} x[i] \cdot 2^i$. We denote the binary representation of a unary coded bit vector $x \in \mathbb{B}^n$ with $n = 2^k$ by $bin_u(x) \in \mathbb{B}^k$ and define it as $y = bin_u(x) \iff \langle y \rangle = \langle x \rangle_u$.

Finally, we define a signal sig as a function from clock cycles to Boolean values. Thus, $sig : \mathbb{N} \to \mathbb{B}^n$ for some $n \in \mathbb{N}$. We refer to the value of a signal with sig^t. In order to refer to the value of a signal during a time interval, we use the notation $sig^{[a:b]} = x$ with $x \in \mathbb{B}^n$ as a shorthand for $sig^a = \ldots = sig^b = x$. Similarly, we use $sig^{[a:b]} = sig'^{[a:b]}$ for a signal sig' as a shorthand for $sig^a = sig'^a \wedge \ldots \wedge sig^b = sig'^b$.

III. Sequential Design

In this Section we present a simple, sequential realisation of the protocol without any support of advanced functionality such as pipelining or burst transfers. We start by introducing the concept of an *abstract transfer*. Afterwards, we specify the main parts of the communication system, i. e. arbiter, salve, and master, by an implementation-close description and reason about local correctness. Finally, we define the bus components in the sequential design and argue about global correctness in Section III-D.

The following definition of an abstract transfer relies on the basic protocol property that there is always on master who is granted the bus (cf. Section II). This definition is crucial to the reasoning throughout this paper. The definition is illustrated in Fig. 1.

Definition 2 (Abstract Sequential Transfer) *The i-th abstract sequential transfer $tr_{seq}(i)$ is defined in terms of three cycle-accurate time points, a corresponding grant signal vector $gnt(i) \in \mathbb{B}^{NM}$, and a single bit $id(i) \in \mathbb{B}$ indicating if the transfer is a idle transfer or not. The first time point $g(i) \in \mathbb{N}$ is the point in time when the bus is granted to the master $\langle gnt(i) \rangle_u$. The second time point $a(i) \in \mathbb{N}$ is the point in time when the address phase ends, i. e. when the rdy signal is active the first time after $g(i)$. The third time point $d(i) \in \mathbb{N}$ denotes the time when the data phase of transfer i ends, i. e. when the rdy signal is active for the second time after $g(i)$.*

$$\begin{aligned} tr_{seq}(i) &= (gnt(i), id(i), g(i), a(i), d(i)) \\ &\in \mathbb{B}^{NM} \times \mathbb{B} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \end{aligned}$$

The components are defined as

$$\begin{aligned} gnt(i) &= arb.grant^{g(i)} \\ id(i) &= trans^{a(i)} \\ g(i) &= \begin{cases} 0 & : i = 0 \\ d(i-1) & : \textit{otherwise} \end{cases} \\ a(i) &= \min\{t > g(i) \mid rdy^t\} \\ d(i) &= \min\{t > a(i) \mid rdy^t\} \end{aligned}$$

where $arb.grant$ denotes the arbiter configuration component specifying the current grant vector. It is specified in detail in the next section.

When talking about transfers, we often need to refer to the time point when the host of the granted master requested this transfer. This is done via the *transfer request time* $trt(i)$ of transfer i.

Definition 3 (Transfer Request Time) *Given a transfer $tr(i)$ we define the transfer request time $trt(i)$ as the time point where the master $\langle gnt(i)\rangle_u$ received the corresponding startreq signal. Let $x = \langle gnt(i)\rangle_u$, then $trt(i)$ is defined by:*

$$trt(i) = \begin{cases} \max\{j \leq g(i) \mid m[x].startreq^j\} & : id(i) \\ \infty & : \neg id(i) \end{cases}$$

Note that the second case specifies $trt(i)$ for an idle transfer. This is only done for reasons of completeness and is not used within the proofs as we only refer to $trt(i)$ in case of a data transfer.

A. Arbiter

The arbiter grants bus access to a master $m[u]$ by activating the corresponding bit $grant[u]$ in the grant bit vector. In case no master requests the bus, the arbiter grants the bus to a default master $defM$ as defined in Section II. In the scope of this project, we abstract the arbiter to a combinatorial circuit relying on an abstract function af generating a new grant vector given the current one and the current request vector. Thus, $af(grant, req)$ returns a new grant vector.

The inputs of the arbiter are a request signal from every master, thus a bit vector $req_{in} \in \mathbb{B}^{NM}$, and the rdy signal of the bus. The arbiter updates the $grant$ bit vector at the end of an address phase, i.e. at $a(i)$. In order to store the information whether an active rdy signal represents the end of an address phase or the end of a data phase, we introduce a singe bit register $aphase$.

However, if the $grant$ vector is updated at the end of the address phase, the old vector is still required during the data phase to select the correct $wdata$ output from a transmitting master. Therefore, the old grant vector is stored as a delayed grant vector in the register $dgrant$.

Moreover, a request vector req has to be maintained in order to store which masters requested the bus.

Thus, the configuration of the sequential arbiter is:

$$\begin{aligned} arb^t_{seq} &= (grant^t, dgrant^t, req^t, aphase^t) \\ &\in \mathbb{B}^{NM} \times \mathbb{B}^{NM} \times \mathbb{B}^{NM} \times \mathbb{B} \end{aligned}$$

The arbiter is specified by the following update rules: Let $upd^t = aphase^t \wedge rdy^t$ and $u \in [0 : NM-1]$, then:

$$\begin{aligned} grant^{t+1} &= \begin{cases} af(req^t, grant^t) & : upd^t \\ grant^t & : \text{otherwise} \end{cases} \\ dgrant^{t+1} &= \begin{cases} grant^t & : upd^t \\ dgrant^t & : \text{otherwise} \end{cases} \\ req^{t+1}[u] &= \begin{cases} 1 & : req_{in}[u] \\ 0 & : af(req^t, grant^t)[u] \\ & \quad \wedge\ upd^t \\ req^t[u] & : \text{otherwise} \end{cases} \\ aphase^{t+1} &= \begin{cases} \neg aphase^t & : rdy^t \\ aphase^t & : \text{otherwise} \end{cases} \end{aligned}$$

Note that $aphase$ is initialized with 0.

Lemma 1 (Sequential Arbiter Correctness) *Let $u \in [0 : NM-1]$ be a master index. Then, the local arbiter correctness is described by the following:*

$$\begin{aligned} & unary(grant^t) \wedge (grant^t \neq grant^{t+1} \implies rdy^t) \\ & \wedge\ (aphase^t \iff t \in [g(i)+1 : a(i)]\ \textit{for some } i) \\ & \wedge\ (req_{in}[m]^t \implies grant^{t'}[m]\ \textit{for some } t' > t) \end{aligned}$$

Finally, the arbiter provides the outputs $grant^t$ to the masters, i. e. $grant[u]$ for $u \in [0 : NM-1]$ to master m_u, as well as to the control bus multiplexers. Additionally, it provides $dgrant^t$ to the write data bus multiplexer.

B. Slave

The task of a slave is to perform read or write accesses to an attached memory system mem. Regarding the bus, the slave has the inputs $sel \in \mathbb{B}$ indicating that the slave is currently addressed, the rdy signal, the $wdata$ component, and the control bus. The $addr$ signal is reduced to the local address $addr[sl-1:0] \in \mathbb{B}^{sl}$. the upper part of the address bus, namely $addr[ad-1:sl]$, is used to generate the select signal sel by an address decoder (cf. Section II).

In case a slave is currently addressed, indicated by an active sel input at the end of the address phase, it has to sample the control bus data in case of a data transfer. Afterwards, the actual memory system access is performed during the data phase. Within that access the memory system can activate a memory busy signal $mem.busy$. The requested data is delivered by the memory system in the cycle when $mem.busy$ is inactive for the first time after the start of the request. We assume that the memory is busy for at most k cycles.

At the end of the memory request, the slave activates the rdy_{out} output and provides the read data on the $rdata_{out}$ output in case of a read access.

As we will see during pipelining in Section IV, the sequential slave is a little bit more complex than the pipelined one. This results from the fact that the sequential slave has to generate the rdy signal indicating the end of the address phase, $a(i)$, additionally to the rdy signal indicating $d(i)$. As the address data can be sampled during one cycle, a unit delay register rdy' is used to delay an active rdy single by one cycle. Then, if rdy' and sel is active, the rdy_{out} output is enabled to generate the $bus.rdy$ signal at $a(i)$.

Moreover, in case of an $idle$ transmission, the slave just produces a rdy_{out} signal in the next cycle (at $d(i)$).

The slave configuration is defined as the tuple

$$s^t_{seq} = (state^t, wr^t, addr^t, wdata^t, mem^t)$$

where

- $state \in \{idle, req\}$ denotes the automaton state.
- $wr \in \mathbb{B}, addr \in \mathbb{B}^{sl}, wdata \in \mathbb{B}^d$ denote the registers to sample the corresponding bus signals.
- $mem : \mathbb{B}^{sl} \to \mathbb{B}^d$ denotes the local memory.

The slave is realized in a straight forward way according to the above description. We omit details here.

The local correctness statement reads as follows.

Lemma 2 (Sequential Slave Correctness) *Given that $sel \wedge rdy'$ holds at time t on sequential slave s_{seq}. Let $t' = \min(k > t \mid \neg mem.busy^k)$, then:*

$$\begin{aligned}
&rdy^t_{out} \wedge (\neg trans^t \implies rdy^{t+1}_{out}) \\
&\wedge\ (trans^t \implies \\
&\qquad \neg rdy^{[i+1:t'-1]}_{out} \wedge rdy^{t'}_{out} \wedge \\
&\qquad (\neg wr^t \implies rdata^{t'}_{out} = mem^{t'}[addr^t]) \wedge \\
&\qquad (wr^t \implies mem^{t'}[addr^t] = wdata^{t'}))
\end{aligned}$$

C. Master

The master provides the interface between the communication system and an attached host system. Within the scope of this case study, the master is our main interest as we present transformations to add advanced functionality to it.

The task of the master is to handle host requests to transfer data. Thus the master has inputs from the host denoted $startreq \in \mathbb{B}$, indicating a transfer request, and host data signals denoted $hwr \in \mathbb{B}$, $haddr \in \mathbb{B}^{ad}$, and $hwdata \in \mathbb{B}^d$ for the respective transfer data. The master has to perform a bus request to the arbiter in case it is not granted the bus. It has to transfer the data according to the sequential schedule, hence the next time it is granted the bus.

Additionally, in case there is no data to transmit but the master is granted the bus, it has to initiate an idle transfer in order to meet the protocol requirements.

The inputs to master $m[u]$ for $u \in [0 : NM - 1]$ regarding the bus are the signals $rdy \in \mathbb{B}$, $grant[u] \in \mathbb{B}$, and $rdata \in \mathbb{B}^d$ as defined in Section II.

As outputs the master provides the signals $trans_{out} \in \mathbb{B}$, $wr_{out} \in \mathbb{B}$, $addr_{out} \in \mathbb{B}^{ad}$, and $wdata_{out} \in \mathbb{B}^d$ needed for the corresponding bus signals. It provides a request signal $req \in \mathbb{B}$ to the arbiter and a busy signal $busy \in \mathbb{B}$ as well as a signal $rdata_{out} \in \mathbb{B}^d$ for the read data to the host. The purpose of the $busy$ signal is the following: the correct transmission of a host request is shown if the master is not busy while the transfer is initiated ($startreq^t \implies \neg busy^t$).

The configuration of the master consists of a $state$ component, a set of registers to sample a host request, and a register to sample data from the $rdata$ bus at the end of a read request. In detail, the configuration of a sequential master m_{seq} is defined as the tuple

$$m^t_{seq} = (state^t, vreq^t, lwr^t, laddr^t, lwdata^t, lrdata^t)$$

where the components are:

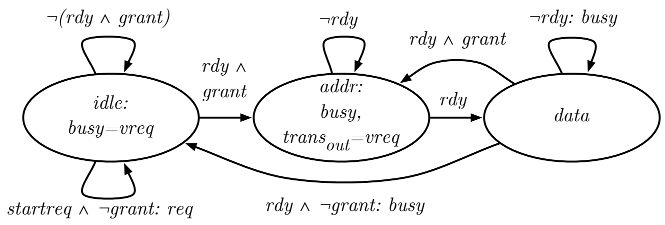

Fig. 2. Sequential Master Control Automaton

- $state \in \{idle, aphase, dphase\}$ denotes the automaton state: $idle$ is the idle state, $aphase$ the state denoting the address phase, and $dphase$ the state denoting the data phase, respectively.
- $vreq \in \mathbb{B}$ denotes that a valid request is currently processed.
- $lwr \in \mathbb{B}$ denotes the local copy of the hwr input. It is written when a host request is sampled.
- $laddr \in \mathbb{B}^{16}$ denotes the local copy of the address analogously to wr.
- $lwdata \in \mathbb{B}^{32}$ denotes the local $hwdata$ copy.
- $lrdata \in \mathbb{B}^{32}$ denotes the register used to sample the $rdata$ bus.

The control automaton is shown in Fig. 2 illustrating the update of the $state$ component and the output signal generation. The bus outputs different from $trans_{out}$ are straight forward the respective local components.

The other configuration components are updated according to the following specification.

$$vreq^{t+1} = \begin{cases} 1 & : (idle^t \wedge startreq^t \wedge \neg busy^t) \\ & \quad \vee (data^t \wedge \; grant^t \wedge rdy^t) \\ 0 & : data^t \wedge rdy^t \\ & \quad \wedge \neg(startreq^t \wedge grant^t) \\ vreq^t & : \text{otherwise} \end{cases}$$

$$lrdata^{t+1} = \begin{cases} rdata^t & : data^t \wedge rdy^t \wedge \neg lwr^t \\ lrdata^t & : \text{otherwise} \end{cases}$$

$$x^{t+1} = \begin{cases} hx^t & : startreq^t \wedge \neg busy^t \\ x^t & : \text{otherwise} \end{cases}$$

where $x \in \{lwr, laddr, lwdata\}$ and $idle^t$, $data^t$ denote that the automaton is in the corresponding state.

Next we argue about the local correctness of the master. We split this argumentation into two parts: (i) local correctness with respect to the host interface, and (ii) local correctness with respect to the bus.

We call a master *locally correct with respect to the host interface* iff it requests the bus upon a valid host request, i. e. a request initiated when the master has not been busy. Therefore, the master shall activate the $busy$ signal during a transfer. Moreover, every time the master is granted the bus, the busy signal has to be inactive for at least on cycle to enable the host to initiate a transfer.

Lemma 3 (Master Correctness wrt. Host Interface) *Given a master m_{seq}, a transfer $tr_{seq}(i)$, and the corresponding transfer request time $trt(i)$. If $startreq^{trt(i)} \wedge \neg busy^{trt(i)}$ holds, then:*

$$\begin{aligned} &(req^{trt(i)} \vee grant^{trt(i)}) \wedge busy^{[trt(i)+1:d(i)-1]} \\ &\wedge (grant^{d(i)} \implies \neg busy^{d(i)}) \\ &\wedge (\neg grant^{d(i)} \implies busy^{d(i)} \wedge \neg busy^{d(i)+1}) \end{aligned}$$

Proof: The proof is obtained by simple reasoning on the control automaton of the master and can be checked automatically. ■

The second correctness part is the more interesting one. We say that the master is *locally correct with respect to the bus* iff the following properties hold: (i) the master reacts to an active rdy and $grant$ signal by starting a transmission; either a $idle$ or a $data$ transmission, (ii) it keeps the bus control signals stable during the address phase, (iii) in case of a write transfer it keeps the write data stable during the data phase, and (iv) in case of a read transfer it samples the $rdata$ bus correctly at the end of the data phase.

Next we define a predicate called $lcorr$ formulating the above properties. Afterwards, we state and prove the correctness lemma for the sequential master.

Definition 4 (Local Master Correctness wrt. Bus) *Given a master m, a transfer $tr(i)$, and a host request $hreq^{trt(i)} = (hwr, haddr, hwdata)$. Then, the local master correctness predicate $lcorr(m, tr(i), hreq^{trt(i)})$ is defined by: $lcorr(m_{seq}, tr(i), hreq^{trt(i)})$ holds iff either $m \neq \langle gnt(i) \rangle_u$ or $m = \langle gnt(i) \rangle_u$ and*

$$\begin{aligned} &trans_{out}^{[g(i)+1:a(i)]} = d(i) \\ &\wedge id(i) \implies (wr_{out}^{[g(i)+1:a(i)]} = hwr^{trt(i)} \\ &\qquad\qquad \wedge addr_{out}^{[g(i)+1:a(i)]} = haddr^{trt(i)})) \\ &\wedge (id(i) \wedge hwr^{trt(i)}) \implies \\ &\qquad\qquad wdata_{out}^{[a(i)+1:d(i)]} = hwdata^{trt(i)} \\ &\wedge (id(i) \wedge \neg hwr^{trt(i)}) \implies lrdata^{d(i)+1} = rdata^{d(i)} \end{aligned}$$

Lemma 4 (Local Master Correctness wrt. Bus) *Given a sequential master m_{seq}, a transfer $tr_{seq}(i)$, and a corresponding host request $hreq_{seq}^{trt(i)}$. Then*

$lcorr(m_{seq}, tr_{seq(i)}, hreaq_{seq}^{trt(i)})$ holds.

Proof: The proof is obtained in two steps: (i) by reasoning about the control automaton of the master, a similar statement is proven which argues about the register contents at time $g(i)$ instead of the host inputs at time $trt(i)$, (ii) afterwards, the claim is obtained with Lemma 3. ■

D. Global Correctness

In this Section we argue about the global correctness of the sequential communication system. First, we define the values of the bus components in detail. Afterwards, we argue about the bus signal generation and introduce some global invariants which ease the argumentation about global correctness. Finally, we state the global correctness theorem of the sequential design and argue about its validity.

Definition 5 (Sequential Bus Components) *Let $u = \langle arb.grant^t \rangle_u$ and $du = \langle arb.dgrant \rangle_u$. Then the components of the bus from Definition 1 in the sequential design are defined by:*

$$\begin{aligned} rdy^t &= \bigvee_{v=0}^{NS-1} s[v].rdy_{out}^t \\ trans^t &= m[u]_{seq}.trans_{out}^t \\ wr^t &= m[u]_{seq}.wr_{out}^t \\ addr^t &= m[u]_{seq}.addr_{out}^t \\ wdata^t &= m[du].wdata_{out}^t \\ rdata^t &= \bigvee_{v=0}^{NS-1} s[v].rdata_{out}^t \end{aligned}$$

As mentioned in Section II, the control bus signals are obtained from the master currently granted the bus and selected by $bin_u(grant)$ using multiplexers. Analogously, the $wdata$ bus is obtained from the master outputs by a multiplexer controlled by $bin_u(dgrant)$.

In the following, we introduce global properties and invariants for the sequential communication system. The first invariant argues about the set of masters. It states that there is exactly one master that is not idle during a transfer. It is exactly the master who is granted the bus.

Invariant 1 (Master Uniqueness) *Given a transfer $tr_{seq}(i)$. Then:*

$$\exists! u \in [0 : NM-1].\quad m[u].state^{[g(i)+1:d(i)]} \neq idle \wedge m = \langle gnt(i) \rangle_u$$

Proof: The validity of this invariant follows from the following three properties: (i) the $unary(grant)$ property of the arbiter, (ii) the fact that the $grant$ signal only changes after the active rdy signal at the end of the address phase, and (iii) the fact that a master only leaves the $idle$ state if the $grant$ and rdy signals are active. ■

Given the master uniqueness we show that the bus signals generated by the masters are obtained correctly.

Lemma 5 (Master to Bus Correctness) *Given a sequential transfer $tr_{seq}(i)$ and let $u = \langle gnt(i) \rangle_u$. Then:*

$$\begin{aligned} trans^{[g(i)+1:a(i)]} &= m[u].trans_{out}^{a(i)} \wedge \\ wr^{[g(i)+1:a(i)]} &= m[u].wr_{out}^{a(i)} \wedge \\ addr^{[g(i)+1:a(i)]} &= m[u].addr_{out}^{a(i)} \wedge \\ wdata^{[a(i)+1:d(i)]} &= m[u].wdata_{out}^{d(i)} \end{aligned}$$

Proof: The lemma is obtained from Invariant 1, the local master correctness from Lemma 4, and the arbiter correctness from Lemma 1. ■

Next, we formulate an invariant similar to Invariant 1 but for the set of slaves. It states that all slaves are in $idle$ state during the address phase and there is exactly one slave that is not in the $idle$ state during the data phase. This is exactly the slave addressed by the master.

Invariant 2 (Slave Uniqueness) *Given a sequential transfer $tr_{seq}(i)$. Let $u = \langle gnt(i) \rangle_u$ be the granted master. Then:*

$$\begin{aligned} &\forall v \in [0 : NS-1].\quad s[v].state^{[g(i)+1:a(i)]} = idle \\ \wedge\ &\exists! v \in [0 : NS-1].\quad s[v].state^{[a(i)+1:d(i)]} \neq idle \wedge \\ &\qquad v = \langle m[u].addr[ad-1 : sl] \rangle \end{aligned}$$

Proof: The invariant is obtained in three steps: (i) the address decoder correctness ensures that $unary(sel)$ holds and $\langle sel \rangle_u = addr[ad-1 : sl]$, (ii) the master correctness (Lemma 4) and Lemma 5 entail that the address bus is stable during the whole address phase, and (iii) reasoning on the slave control automaton shows that the only time interval the idle state is left is between the rdy signal at time $a(i)$ and the next one ($d(i)$). ■

A property similar to Lemma 5 regarding the rdy and $rdata$ signals of the bus can be formulated. It is obtained from Invariant 2 and the slave correctness in Lemma 2.

Lemma 6 (Slaves to Bus Correctness) *Given a transfer $tr_{seq}(i)$ and let $v = \langle addr^{a(i)}[ad-1 : sl] \rangle$ be the addressed slave. Then:*

$$\begin{aligned} rdy^{[a(i):d(i)]} &= s[v].rdy_{out}^{[a(i):d(i)]} \wedge \\ rdata^{d(i)} &= s[v].rdata_{out}^{d(i)} \end{aligned}$$

Finally, the overall correctness theorem reads as follows.

Lemma 7 (Overall Correctness Sequential System) *Let $hreq_m^k$ denote a host request at time k with $startreq^k$ on master m_{seq}. Moreover let*

$$w = s[\langle haddr^k[ad-1:sl]\rangle].mem(haddr^k[sl-1:0])$$

denote the addressed memory word. Then:

$$\begin{aligned} \neg busy_m^k \implies\ & \exists i.\ (k = trt(i) \\ & \wedge\ tr(i) = (gnt(i), id(i), g(i), a(i), d(i)) \\ & \wedge\ gnt(i) = m_{seq} \wedge id(i) \\ & \wedge\ (hwr^k \implies w^{d(i)} = hwdata^k) \\ & \wedge\ (\neg hwr^k \implies lrdata^{d(i)+1} = w^{d(i)})) \end{aligned}$$

Proof: The first two clauses follow from the arbiter correctness in Lemma 1 and the correctness of the master with respect to the host interface (Lemma 3). The third follows from the transfer definition in Definition 2 and again the arbiter correctness. The last three clauses are given by the master to bus correctness in Lemma 5, the slave to bus correctness in Lemma 6, and the local correctness of both (Lemmas 4 and 2). ■

Note that one would need a much more evaluated arbitration system in order to provide the read data from time k instead of time $d(i)$.

IV. Pipelined Protocol

In this section, we derive a communication system supporting pipelined data transfer from the previous sequential design based on a correctness-preserving transformation. The idea behind the pipelining of this protocol is to execute the address phase of a transfer i in parallel with the data phase of address $i-1$. This is possible because of the separated control and data buses. A sequence of pipelined transfers is shown in Fig. 3.

The bus for the pipelined system stays exactly the same as for the sequential system.

$$bus^i = (rdy^i, trans^i, wr^i, addr^i, wdata^i, rdata^i)$$

As we focus on the transformation applied to the master, we do not go into details regarding arbiter and slaves here. The arbiter is obtained by ignoring the $aphase$ bit register from the sequential arbiter and updating the $grant$ vector each time the rdy signal is active. Since the $grant$ vector is updated every time rdy is active, we have to introduce a third grant register denoted $ddgrant$ which is updated with the data from $dgrant$ in exactly the same way as $dgrant$ with $grant$.

This results from the following: The master for the next transfer is specified by the $grant$ vector at time $g(i)$. In contrast to the sequential arbiter, the grant vector is also updated at time $g(i)$. Thus the control parts of the bus during the address phase are not specified by the $grant$ but by the $dgrant$ vector. But then, we need even one more copy of the former grant vector to control the data multiplexer during the data phase.

The slave is obtained by removing the unit delay rdy'. The pipelined slave only generates a rdy signal after the memory access, thus at the end of the data phase.

Finally, we adopt the abstract transfer definition.

Definition 6 (Abstract Pipelined Transfer) *The i-th abstract pipelined transfer is defined analogous to the sequential transfer from Definition 2 as the tuple $tr_{pipe}(i) = (gnt(i), id(i), g(i), a(i), d(i))$ where the components are defined as:*

$$\begin{aligned} gnt(i) &= grant^{g(i)} \\ id(i) &= \langle gnt(i)\rangle_u.trans_{out}^{a(i)} \\ g(i) &= \begin{cases} 0 & : i = 0 \\ a(i-1) & : otherwise \end{cases} \\ a(i) &= \min\{t > g(i) \mid rdy^t\} \\ d(i) &= \min\{t > a(i) \mid rdy^t\} \end{aligned}$$

The new definition is illustrated in Fig. 3. Note that the only difference to the sequential definition is that $g(i)$ is now recursively defined over $a(i-1)$ instead if $d(i-1)$. This represents the pipelining in the abstract transfer and results in the additional properties $a(i) = d(i-1)$ for $i > 0$ and $g(i) = d(i-2)$ for $i > 1$.

Next we specify the transformation for the master followed by the local correctness argumentation. Finally, we reason again about the global correctness of the pipelined communication system.

A. Master

Our gaol is to obtain a master supporting pipelined transfers from the master only supporting sequential transfers. We derive the new master in an algebraic way such that we can use the correctness of the former

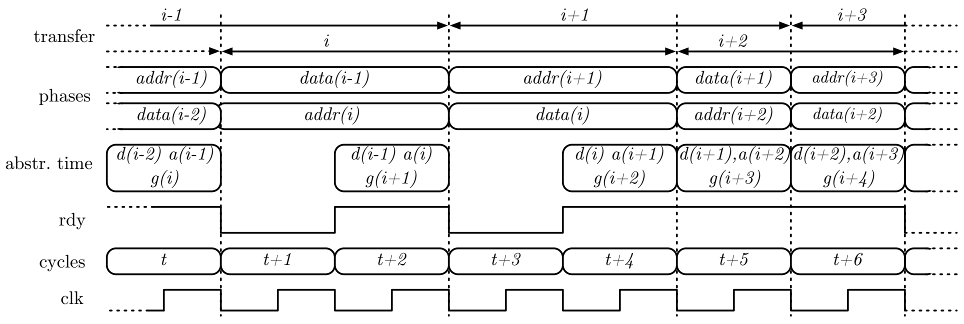

Fig. 3. Sequence of Pipelined Transfers

master to argue about the new one. The basic idea behind the transformation is to execute two sequential masters in parallel. Then we restrict the behavior of the parallel system by excluding some *bad* executions. In the following, we denote the two sequential masters with $m1$ and $m2$, hence $m_{pipe} = (m1, m2)$.

These behavioral constraints are obtained by modifying the inputs to the *internal* sequential masters. This is realized by a logic denoted $InpTrans$ splitting the inputs into two sets of inputs for $m1$ and $m2$. Additionally, we need to combine the outputs of the two internal masters to generate the outputs of a single master. This combinatorial function is called $TransOut$ in the following.

If we execute two sequential masters in parallel, the state space of $m_{pipe}.state$ is equal to the cartesian product of the state components of the two sequential masters.

$$m_{pipe}.state \in m_{seq}.state \times m_{seq}.state$$

The key question is now which behavior has to be excluded from the purely interleaved execution of both control automatons. We aim to obtain a behavior where $m1$ dominates $m2$ in the sense that $m2$ only becomes active in situations where the sequential master could not executed the required behaviour. We have the following two key properties which have to be maintained: (i) $m2$ never requests the bus and (ii) $m2$ only leaves the idle state in case $m1$ is at the end of the address phase and an additional, parallel transfer from m_{pipe} is required. The first property is required to prohibit the situation that the bus is granted to m_{pipe} and the master would have to 'decide' which of the sequential masters requested the bus (and has stored the data to transmit). Moreover, the arbiter cannot handle two requests from one master in the sense to be able to grant the bus twice to that master.

These properties are realized in the following way: The first one is obtained by restricting the $startreq$ input of $m2$ and adjusting the $busy$ output to the host. The latter has to be done in order to preserve the property that a transfer is handled correctly if the $busy$ signal is inactive. The second property is achieved by restricting the $grant$ input of $m2$.

Next we define the $InpTrans$ function which splits the inputs of the master and applies the described restrictions. Then we specify the pipelined master in terms of a transition function using the $InpTrans$ function.

Definition 7 (Input Transformation) *Given a master* m_{pipe}. *Let* $inp = (startreq, hreq, rdata, rdy, grant)$ *denote its inputs with* $hreq = (hwr, haddr, hwdata)$. *Then we define the function InpTrans by:*

$$\begin{array}{l} InpTrans\ inp = let \\ \quad startreq_2 = startreq \wedge grant \wedge (m1.state = addr) \\ \quad grant_2 = grant \wedge (m1.state = addr) \\ in \\ \quad (inp, (startreq_2, hreq, rdata, rdy, grant_2)) \end{array}$$

Definition 8 (Master for Pipelined Transfers) *Given a pipelined master* $m_{pipe} = (m1, m2)$ *and let* dM_{seq} *denote the transition function from the sequential master as defined in Section III-C. Moreover, let* $inp = (startreq, hreq, rdata, rdy, grant)$ *denote the inputs of* m_{pipe} *with* $hreq = (hwr, haddr, hwdata)$. *Then, the pipelined master is defined by the following transition function.*

$$\begin{array}{l} dM_{pipe}\ m_{pipe}\ inp = let \\ \quad (inp_1, inp_2) = InpTrans\ inp \\ in \\ \quad (dM_{seq}\ m1\ inp_1, dM_{seq}\ m2\ inp_2) \end{array}$$

Finally, the outputs are obtained in a straight forward way. The sequential master which is currently in the $addr$ or $data$ state provides the corresponding bus outputs. Additionally, the req_{out} signal to the arbiter is only triggered by $m1$.

The only non-obvious computation is the $busy$ signal. Obviously, the pipelined master has to be busy if both sequential masters are busy. Additionally, the second sequential master is not allowed to request the bus. Therefore we have to enable the busy signal in cases where the first master is busy and the second master would have to request the bus after a $startreq$ signal. There are three of those cases, namely (i) both sequential masters are in the $idle$ state and the first master is waiting for a $grant$ signal, (ii) $m1$ is in the address phase and the bus is no more granted to the master, and (iii) $m1$ is in the data phase and neither the bus is granted anymore nor the rdy signal is active.

Definition 9 (Output Transformation) *Given a master m_{pipe}. Then its outputs are obtained from the outputs of the sequential masters $m1$ and $m2$ in the following way:*

$$\begin{aligned}
req^t_{out} &= m1.req^t_{out}\\
x^t_{out} &= \mathit{if}\ (m2.state^t = addr)\\
&\quad \mathit{then}\ m2.x^t_{out}\ \mathit{else}\ m1.x^t_{out}\\
wdata^t_{out} &= \mathit{if}\ (m2.state^t = data)\\
&\quad \mathit{then}\ m2.wdata^t\ \mathit{else}\ m1.wdata^t\\
busy^t_{out} &= (m1.busy^t \wedge m2.busy^t)\\
&\quad \vee\ (m1.busy^t \wedge m1.state^t = idle\ \wedge\\
&\qquad m2.state^t = idle)\\
&\quad \vee\ (m1.state^t = addr \wedge \neg grant^t)\\
&\quad \vee\ (m1.state^t = data \wedge \neg(grant \wedge rdy))
\end{aligned}$$

where $x \in \{trans, addr, wr\}$.

Recall that the $grant$ vector only changes in a cycle when rdy is active. Thus if the the bus is granted to a master this holds until the next active rdy signal and vice versa.

Next we argue about the local correctness of the pipelined master. Similar to the local correctness of the sequential master, we split cases into host interface correctness and bus correctness. The correctness with respect to the host interface reads similar to for the sequential system. The only difference is that we also have to ensure that there is a cycle when $busy$ is inactive before the data phase (i. e. at $a(i)$) if the bus is still granted. This is required to allow pipelined data transfers. The proof is slightly more complex because one has to argue about two consecutive transfers instead of one as they are executed partially in parallel.

The correctness with respect to the bus is also the same as for the sequential system. This maybe surprising fact results from the pipelined transfer definition which encapsulates the parallel execution property. This leads to clean correctness statements but has to be considered during the proof.

Lemma 8 (Local Master Correctness wrt. Bus) *Given a master m_{pipe}, a pipelined transfer $tr_{pipe}(i)$, and a corresponding host request $hreq^{trt(i)}$. Then the local correctness predicate for the sequential system from Definition 4 also holds for the pipelined master, thus $lcorr(m_{pipe}, tr_{pipe}(i), hreq^{trt(i)})$.*

Proof: This lemma is obtained from: (i) the correctness properties for $m1$ and $m2$, (ii) $m1$ always executes *in advance* to $m2$, i. e. for a sequence of transfers for which the master is constantly granted the bus, $m1$ always executes the odd transfers within this sequence and $m2$ the even ones. (iii) induction on the length of the sequence of consecutive granted transfers.

■

B. Global Correctness

The global correctness of the pipelined system is obtained completely analogous to the sequential argumentation. The two invariants have to be slightly modified.

Invariant 1 (master uniqueness) needs to be adopted to take the parallel execution of address and data phase into account. Moreover, the new state space has to be considered and whether a master granted the bus for consecutive transfers.

In Invariant 2 (slave uniqueness) not all slaves are in the idle state during address phase since the address phase of a transfer is equal to the data phase of the previous transfer. Hence, it has to be modified to state that during the address phase of a transfer, the only slave not in the $idle$ state is the slave address by the master specified by the old grand vector $dgrant$ of the arbiter at time $a(i)$.

Finally, the global correctness statement reads the same as for the sequential design in Lemma 7. Similarly to the local correctness, this results from the abstract transfer definition but additionally from the fact that the properties regarding the address and data phase in the global correctness statement are formulated completely separately. The difficulties arising from the transfer pipelining has already been dealt with in the previous local correctness properties and the invariants.

V. Burst Protocol

In this Section we specify a transformation providing burst transfers applicable to either the sequential or the pipelined system. The presented transformation is general enough that there are only few cases where one has to distinguish whether a sequential or a pipelined design is extended. Therefore, we present the general transformation and accept the slightly increased complexity at some places.

A burst transfer of size $(bsize + 1)$ is a read or write transfer supposed to start a some base address $baddr$ transferring all data from address $baddr$ to $baddr{+}bsize$. We support burst transfers of arbitrary but fixed length up to a maximum denoted BM. The actual length of a transfer is specified during the corresponding host request. We assume that $BM = 2^b - 1$ for some $b \in \mathbb{N}$ and thus we use $bsize \in \mathbb{B}^b$.

The communication bus between masters and slaves stays as before, but to realize burst transfers, we need to introduce two additional signals from the master to the arbiter. Hence, besides the req signal, the master provides the outputs $bst \in \mathbb{B}$ and $bsize \in \mathbb{B}^b$ to the arbiter. These signals are used to signal a burst transfer request to the arbiter that has to ensure that the grant signal stays constant during a burst transfer.

As in the previous section, we focus on the master transformation and therefore, we only describe the arbiter and slave shortly.

The arbiter is obtained from the corresponding previous arbiter, i. e. sequential or pipelined, by adding two additional registers for every master to sample burst request data. That is upon an active $startreq$ signal from some master, the corresponding flag indicating a burst request is set if the bst bus signal is active. At the same time, the $bsize$ signal is sampled. When the bus is granted to a master with a pending burst request, the arbiter keeps the $grant$ vector stable for $bsize$ may transfers.

The slave is the same as in the corresponding sequential or pipelined system.

Finally, we adapt the transfer notation to support burst. These changes are more complex than the changes from sequential to pipelined transfers. Again, the basic principle is to add a component $bst(i)$ to indicate a burst transfer together with a field $bsize(i)$. The end of the address phase is now not only a single time point but a partial function assigning a address phase end time to every *sub-transfer* $n \in [0 : bsize(i) - 1]$.

Definition 10 (Abstract Burst Transfer) *The i-th abstract burst transfer $tr_{burst}(i)$ within the burst communication system is defined as the tuple*

$$\begin{aligned} &tr_{burst}(i) \\ &\quad = (gnt(i), id(i), g(i), a(i), d(i), bst(i), bsize(i)) \\ &\quad \in \mathbb{B}^{NM} \times \mathbb{B} \times \mathbb{N} \times (\mathbb{N} \rightarrow \mathbb{N}) \times \mathbb{N} \times \mathbb{B} \times \mathbb{B}^b \end{aligned}$$

where the components are defined as:

$$\begin{aligned} gnt(i) &= grant^{g(i)} \\ id(i) &= m[\langle gnt(i)\rangle_u].trans_{out}^{a(i)} \\ g(i) &= \begin{cases} 0 & : i = 0 \\ a(i-1, bsize(i-1)) & \\ \quad : i > 0 \wedge (m\ \textit{pipelined}) & \\ d(i-1) & : i > 0 \wedge (m\ \textit{sequential}) \end{cases} \\ a(i,n) &= \begin{cases} \min\{t > g(i) \mid rdy^t\} & \\ \quad : n = 0 \vee \neg bst(i) & \\ \min\{t > a(i, n-1) \mid rdy^t\} & \\ \quad : n \in [1 : bsize(i)] \wedge bst(i) & \\ \textit{undefined} : \textit{otherwise} & \end{cases} \\ d(i) &= \begin{cases} \min\{t > a(i,0) \mid rdy^t\} & \\ \quad : \neg bst(i) & \\ \min\{t > a(i, bsize(i)) \mid rdy^t\} & \\ \quad : bst(i) & \end{cases} \\ bst(i) &= m[\langle gnt(i)\rangle_u].bst^{a(i,0)} \\ bsize(i) &= m[\langle gnt(i)\rangle_u].bsize^{a(i,0)} \end{aligned}$$

The main difference between this definition and the previous is the case split on actual burst transfers. In case of a non-burst transfer the above definition resolves to one of the previous transfer definitions depending on the kind of master we are extending by burst support.

A. Master

In the following we present the transformation to add burst transfer support to one of the previous masters. The basic idea is simple: We add a counter for the *burst sub-transfers* and *simulate* a sequence of $bsize$ many standard transfers. Thereby, the arbiter correctness ensures that the bus is granted during the whole burst transfer. We specify the whole transformation again in terms of an input transformation $\mathit{InpTrans}$ and an output transformation $\mathit{TransOut}$.

The interface from host to master has to be extended by two new signals: Upon a host request, $\mathit{hbst} \in \mathbb{B}$ signals that the current transfer is a burst request of size $\mathit{hbsize} \in \mathbb{B}^b$. Additionally, we introduce a signal to the host called $\mathit{bdataupd} \in \mathbb{B}$. It is used to signal that the data in $wdata$ has to be updated for a burst write access or a data chunk from a burst read access can be read from the $rdata$ bus. Hence, we require that the host updates the $wdata$ register at the time $\mathit{bdataupd}$ is active for a burst write access and the host hast to read the data from $rdata$ in the cycle after $\mathit{bdataupd}$ was active. We define $\mathit{bdataupd}$ in detail within the specification of $\mathit{TransOut}$.

Note that this host interface requirement could be discharged by extending the master with a data memory of address width b. This would not complicate the subsequent correctness argumentation significantly but lengthen the specification of the transformations. Therefore we omit it here.

We also have to extend the configuration of the master by three new components to handle burst requests: $\mathit{bst} \in \mathbb{B}$ and $\mathit{bsize} \in \mathbb{B}^b$ are used to sample the corresponding host signals upon a request. $\mathit{bfst} \in \mathbb{B}$ indicates if the first burst sub-transfer has to be sent. This flag is needed because the local address register has to be incremented before every burst sub-transfer except the very first. Thus, the configuration of a burst master is defined as the tuple

$$m_{burst} = m_x \times (\mathit{bst}, \mathit{bsize}, \mathit{bfst})$$

where x is either seq or $pipe$.

In contrast to the $\mathit{InpTrans}$ for the pipelined master, this transformation is not only a combinatorial circuit but a state-representing extension to the master containing the newly introduced configuration parts. Thus, we describe the $\mathit{InpTrans}$ box by a next state function $dInpTrans$ and denote its output function by $\mathit{InpTrans}$ which is defined afterwards.

Definition 11 (Input Transformation) *Given a master m_{burst} supporting burst transfers. Let $inp = (startreq, hreq, rdata, rdy, grant)$ denote its inputs with $hreq = (hwr, haddr, hwdata, hbst, hbsize)$ and let m_x denote the master which is extended. Moreover, let $done = (bsize = 0)$ and*

$$bupd = \begin{cases} rdy \wedge \neg bfst & : x = pipe \\ rdy \wedge (state = data) & : x = seq \end{cases}$$

Then we define $dInpTrans$ by:

$$\begin{aligned}
& dInpTrans\ (bst, bsize, bfst)\ inp = let \\
& \quad bfst' = \begin{cases} 1 & : startreq \wedge \neg m_x.busy \\ & \quad \wedge\ hbst \wedge \neg(grant \wedge rdy) \\ 0 & : rdy \wedge grant \\ bfst & : otherwise \end{cases} \\
& \quad bst' = \begin{cases} hbst & : startreq \wedge \neg m_x.busy \\ 0 & : bupd \wedge done \\ bst & : otherwise \end{cases} \\
& \quad bsize' = \begin{cases} hbsize & : startreq \wedge \neg m_x.busy \\ bsize - 1 & : bupd \wedge bst \wedge \neg done \\ bsize & : otherwise \end{cases} \\
& in \\
& \quad (bst', bsize', bfst')
\end{aligned}$$

The outputs generated by the $\mathit{InpTrans}$ box are specified by the $\mathit{InpTrans}$ function defined in the following.

Definition 12 (Input Transformation Signals) *Let $(bst, bsize, bfst)$ denote the current state of the InpTrans box and let $inp = (startreq, hreq, rdata, rdy, grant)$ denote its inputs with $hreq = (hwr, haddr, hwdata, hbst, hbsize)$. Additionally, let m_x denote the master which is extended, $done = (bsize = 0)$, and*

$$bupd = \begin{cases} rdy \wedge \neg bfst & : x = pipe \\ rdy \wedge (state = data) & : x = seq \end{cases}$$

Moreover let $nextbst$ be a shorthand for $bupd \wedge bst \wedge$

$\neg done$. Then, the output function $InpTrans$ is given by:

$$
\begin{aligned}
&InpTrans\ (m_x, bst, bsize, bfst)\ inp = let\\
&\quad startreq_{burst} = startreq \vee nextbst\\
&\quad wr_{burst} = \begin{cases} m_x.wr_{out}: & nextbst\\ hwr: & otherwise\end{cases}\\
&\quad addr_{burst} = \begin{cases} m_x.addr_{out}+1: & nextbst\\ haddr: & otherwise\end{cases}\\
&in\\
&\quad (startreq_{burst}, wr_{burst}, addr_{burst})
\end{aligned}
$$

Now, we can define the master supporting burst transfers based on a previous master and the presented input transformation.

Definition 13 (Master for Burst Transfers) *Given a master $m_{burst} = (m_x, bst, bsize, bfst)$ and the transition function dM_x for x either seq or pipe. Moreover, let $inp = (startreq, hreq, rdata, rdy, grant)$ denote its inputs with $hreq = (hwr, haddr, hwdata, hbst, hbsize)$. Then, the burst master is defined by the following transition function.*

$$
\begin{aligned}
&dM_{burst}\ m_{burst}\ inp = let\\
&\quad (startreq_{burst}, wr_{burst}, addr_{burst}) =\\
&\qquad\qquad InpTrans\ (m_x, bst, bsize, bfst)\ inp\\
&\quad m'_x = dM_x\ m_x\ (startreq_{burst}, wr_{burst}, addr_{burst},\\
&\qquad\qquad hdata, grant, rdy)\\
&in\\
&\quad (m'_x, dInpTrans\ (bst, bsize, bfst)\ inp)
\end{aligned}
$$

Finally, we specify the output transformation. Except for the $busy$ and $bdataupd$ outputs, the outputs remain the same as for the previous masters. The signal to update the burst data is given by $bdataupd^t = bupd^t \wedge bst^t \wedge \neg done^t$ where $done$ denotes $bsize = 0$ as before. The $busy$ signal has to be adapted for the case a burst access is in progress. Usually the $busy$ signal turns off at the end of a request if the bus is still granted, i. e. at time $d(i)$ for a single non-burst transfer. The $busy$ signal is modified such that it remains active during a burst transfer. Thus we obtain $busy^t = m_x.busy^t \vee (bst^t \wedge \neg done)$. For all other outputs, $TransOut$ is just the identity.

In the remaining of this section, we argue about the local correctness of the presented construction. The local correctness of the master supporting burst transfers is again split into the correctness regarding the host interface and regarding the bus. The correctness with respect to the host interface is very similar to the previous ones, extended by the correct sampling of the $wdata$ for the sub-transfers during a burst request. Since the sequential or pipelined master can only update all host data registers at the same time, the transition function of the burst master simulates a completely new transfer with the same wr, the increased $addr$, and the new $wdata$.

As before, the local correctness with respect to the bus is the more interesting case. Here we spilt cases on burst transfers: (i) in case of a non-burst transfer, the master has to obey the local correctness statement from the sequential (Lemma 4) or pipelined (Lemma 8) master. (ii) in case of a burst transfer, it has to simulate a sequence of locally correct single transfers as defined in Definition 14.

Definition 14 (Local Master Correctness) *Given a master m_{burst}, a transfer $tr_{burst}(i)$, and a burst host request $hreq^{trt(i)}$ on m_{burst}. Moreover, let $0 < n \wedge n < bsize(i) - 1$. In order to increase readability, we omit the transfer index i in the following. Then, we define the predicate indicating local burst correctness $lbcorr(m_{burst}, tr_{burst}, hreq^{trt})$ by:*

$lbcorr(m_{burst}, tr_{burst}, hreq^{trt})$ holds iff either $m_{burst} \neq \langle gnt\rangle_u$ or $bst = 0$ or $m_{burst} = \langle gnt\rangle_u$, $bst = 1$, and

$$
\begin{aligned}
&lcorr(m_{burst}, (1, gnt, g, a(0), a(1)),\\
&\quad (hwr^{trt}, haddr^{trt}, hwdata^{trt})) \quad \wedge\\
&lcorr(m_{burst}, (1, gnt, a(n-1), a(n), a(n+1)),\\
&\quad (hwr^{trt}, haddr^{trt}+n, hwdata^{a(n-1)})) \quad \wedge\\
&lcorr(m_{burst}, (1, gnt, a(bsize-2), a(bsize-1), d),\\
&\quad (hwr^{trt}, haddr^{trt}+bsize-1, hwdata^{a(bsize-2)}))
\end{aligned}
$$

Finally, the argumentation about the global correctness is straight forward analogous to the global correctness of the sequential or pipelined design. As for the local correctness one splits cases if the requested transfer is a single or a burst transfer. For single transfer the global correctness of the corresponding design holds. For a burst transfer the local correctness statement from Definition 14 is lifted to the global level the same way as done for the sequential or pipelined design.

VI. Conclusion and Future Work

As on-chip communication systems have become more complex to meet performance requirements, their formal verification has become infeasible in reasonable time and effort with current post-hoc verification methodologies. Our new approach tries to resolve this by introducing a new methodology to design and verify communication protocols in an algebraic fashion. This approach is based on correctness-preserving or correctness-inheriting transformations to a simpler design providing enhanced functionality.

The results presented here are only initial work. Besides tackling more transformations, future work will also have to address the following issues. First, we need to find a more systematic method for applying transformations. For example, the pipelined design presented here is obtained by a parallel composition of two sequential masters and imposing extra constraints of the inputs (Definition 7) and combining the outputs in an appropriate way (Definition 9). In this initial work these constraints were devised by hand, but we want to provide a more systematic approach to this, either through a formal analysis such as model checking or by adopting a notational structure that makes them evident.

Second, we will need to devise a range of refinement relations to link different abstraction levels of a model; either temporal, logical, or a combination of them. We need to find methods to reason about the relationships between communications at an implementation level, e. g. register transfers, and a more abstract, system level. This is needed, for example, to relate an atomic, high-level transfer to non-atomic communications possibly spread over space and time on a cycle-accurate register transfer level.

Third, we must address the problem of incorporating methods to tackle problems evolving from high-performance aspects on gate-level or even physical layers. For example, a desirable goal is to develop a method to argue about low-level timing constraints or distributed, asynchronous clocks within our methodology.

Finally, we emphasise that our main interest for future development of these ideas is not to analyse AMBA or standard system-on-chip bus protocols. Our aim is develop a framework for the design and formal verification of scalable, high-performance communication platforms that allow customisation.

References

[1] *AMBA Specification Revision 2.0*, ARM, 1999. [Online]. Available: http://www.arm.com

[2] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[3] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. R. Marco Pistore, R. Sebastiani, and A. Tacchella, “NuSMV 2: An open source tool for symbolic model checking,” in *CAV ’02*. Springer-Verlag, 2002, pp. 359–364.

[4] S. Tverdyshev, “Combination of Isabelle/HOL with automatic tools,” in *FroCoS 2005, Vienna Austria*, ser. LNCS, vol. 3717. Springer, 2005, pp. 302–309.

[5] A. Roychoudhury, T. Mitra, and S. Karri, “Using formal techniques to debug the amba system-on-chip bus protocol,” *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pp. 828–833, 2003.

[6] H. Amjad, “Model checking the AMBA protocol in HOL,” University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-602, Sep. 2004. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-602.pdf

[7] J. Schmaltz and D. Borrione, “Towards a formal theory of on chip communications in the acl2 logic,” in *ACL2 ’06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*. New York, NY, USA: ACM, 2006, pp. 47–56.

[8] F. Müffke, “A better way to design communication protocols,” Ph.D. dissertation, University of Bristol, May 2004. [Online]. Available: http://www.cs.bris.ac.uk/Publications/Papers/2000199.pdf

[9] C. Seger, “The design of a floating point unit using the integrated design and verification (idv) system,” in *Sixth International Workshop on Designing Correct Circuits: Participants’ Proceedings*, M. Sheeran and T. Melham, Eds., March 2006.

[10] M. Aagaard, B. Cook, N. A. Day, and R. B. Jones, “A framework for microprocessor correctness statements,” in *CHARME ’01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. London, UK: Springer-Verlag, 2001, pp. 433–448.

[11] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan, “Shared memory consistency protocol verification against weak memory models: Refinement via model-checking,” in *CAV ’02: Proceedings of the 14th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 2002, pp. 123–136.

[12] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic, “Abstraction and refinement in protocol derivation,” in *CSFW ’04: Proceedings of the 17th IEEE workshop on Computer Security Foundations*. Washington, DC, USA: IEEE Computer Society, 2004, p. 30.