

GasP model in Archsim

Igor Benko

UCIB #: 2005-ib02

September 2005

Abstract

Archsim is an event-based behavioral simulator intended for architectural explorations of Fleet [?] structures. This document outlines a model framework that we built for Archsim. The framework follows the notation from [?]. Building blocks of the model map to GasP control circuits [?]. Consequently, we expect that it should not be too difficult to automate translation from a model to a schematic, at least for control structures.

References

- [1] Ebergen: “Revised notation for designing GasP Circuits”, SML 2000-0379
- [2] Benko: “Archsim Structure”, SML 2005-0103
- [3] Benko, Sutherland: “StateWires, ActionBoxes, and TerminalArrows”, SML-2005-is02
- [4] Coates, Lexau, Jones: FLEETzero: An asynchronous switching experiment”, SML 2000-0768
- [5] Sutherland, Fairbanks: “GasP: A minimal FIFO control”, SML 2000-0756
- [6] Ebergen, Sutherland, Tourancheau: “Arbiters with Preferential Enables”, SML2003-0552

UC Berkeley Computer Science

This document is a product of a collaboration between Sun Microsystems and the University of California at Berkeley. The ideas contained herein are freely available for any academic purpose.

1 Introduction

The layered nature of Archsim [?] requires that one builds their own model that uses the provided simulation engine. The model we chose for simulations of the Fleet architecture is based on Jo Ebergen’s notation for GasP circuits [?]. This notation has a simple translation into GasP circuit modules. This memo describes a framework for building models of GasP circuits that can be executed in Archsim. The goal of the memo is to make reading and understanding of the framework easier, and to provide simple directions for using the framework. A complete specification of the framework is beyond the scope of this memo.

2 Overview of Ebergen’s notation

The notation presented in [?] represents states and transitions between the states in a collection of finite state machines. At the same time, the notation can easily be translated into an implementation using GasP modules. For this reason, many perceive this notation as being specific to GasP modules. In this section I make an attempt at a very brief overview of the notation introduced by Ebergen; I strongly encourage the reader to refer to [?] for more information. An even better approach is to talk to Jo Ebergen.

Figure ?? depicts the components of the notation: an action, a state, and a terminal. A state can be either active or inactive. The state also stores data. Because the state is depicted with a line that is usually used to represent a wire, building the intuition that data storage may require a bit of an effort.

An action, depicted with a box, represents a transition from one state to another. A simple rule holds for any action: the action can fire when all its input states are active. When an action fires, all its output states become active. Depending on the connection, input states either become inactive, or are not affected. Note that there is no memory in an action.

What do we mean by “input” and “output” when we refer to the states with respect to an action? One view is that output states are the states that can be set active by the action, and input states will only be set inactive by the action. Because an action can fire only when all of the input states are active, one can say that the input states are “controlled” by the environment. The control over activity of output states provides an action with the ability to enable or disable firing of other actions.

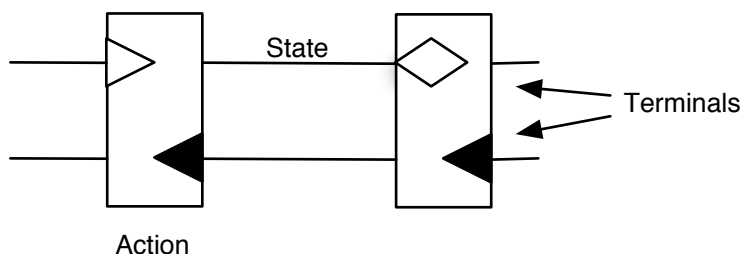


Figure 1: Notation components: state, action, and terminal

Actions and states are connected via terminals. Both, a state or an action can be connected to any number of terminals. A terminal, however, is always connected to one state and one action. A terminal has a direction with respect to the action: For input terminals we use either an arrow (triangle) or a diamond, and we do not use any specific symbol to depict an output terminal.

A triangle represents a self-resetting input terminal. This simply means that when the action fires the state connected to the terminal will become inactive. Recall that an action can only fire when all states attached to input terminals are active.

A diamond represents an input terminal that is not self-resetting. That is, when the action fires, the state attached to a non-self-resetting input terminal does not become inactive.

Finally, a blackened input terminal means that the state attached to that terminal is initially active. This means that all input terminals attached to a state that is initially active must be blackened.

Figure ?? demonstrates how to represent two stages of a FIFO. The dashed boxes represent stage boundaries - note that stage boundaries cross actions. In the more common representation of a FIFO, the boundaries tend to be drawn across the "wires", centering on transitions between the stages.

State names correspond to the status of the corresponding FIFO stage. For example, when state "A full" is active, FIFO stage A is storing data and protecting that data from being overwritten by previous stages. Similarly, when state "A empty" is active, then FIFO stage A does not prevent its stored data from being overwritten. Initially, no data has been passed into the FIFO, thus all of the "empty" states are active.

Notice that data transition from stage A to B can only happen when stage A is full and stage B is empty.

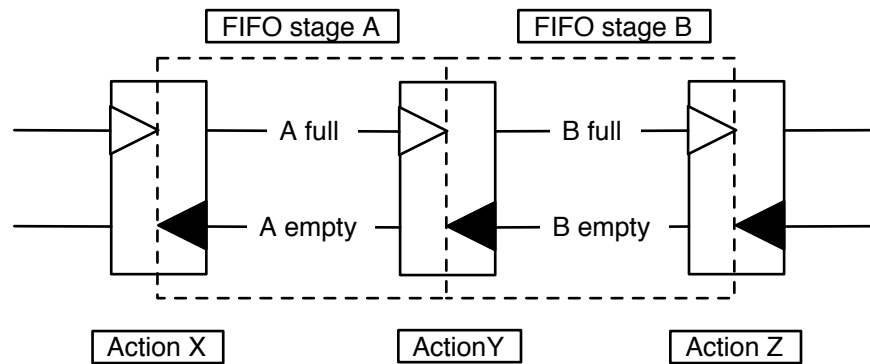


Figure 2: FIFO

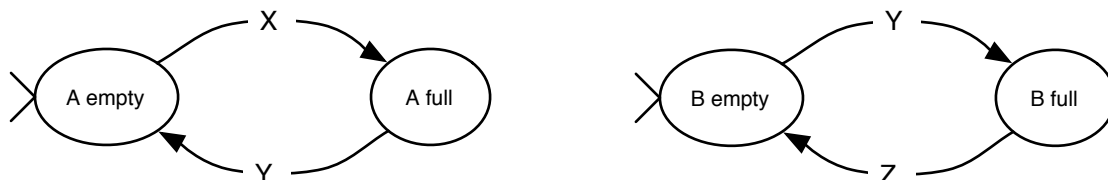


Figure 3: State graph for one FIFO stage

Figure ?? shows the state graphs corresponding to FIFO stages from Figure ?. Transition on X leads from state "A empty" to state "A full". Looking at Figure ?? we notice that this transition corresponds to firing of action X. Notice also, that firing of action Y corresponds to a transition in both state graphs: Stage A becomes empty, and stage B becomes full. This gives us a hint of how Ebergen's notation can lead to composition of behaviors described by state graphs. For more details, we refer the reader to [?].

2.1 Implementation

The model used in Archsim has a close correspondence to an implementation of GasP shown in Figure ?. That figure shows a circuit for two actions connected in the same manner as shown in Figure ?. The wire marked with letter F and its keeper correspond to state "full" and the wire marked with letter E, and its keeper, correspond to the state "empty". In this implementation the "request" (full) and acknowledge

(empty) signals are separated. In pipeline configurations, the requests and acknowledge signal can both use the same wire, as shown in Figure ???. I contemplated a model that would correspond to this optimized circuit. At the moment I decided to stick with the more general model and leave optimizations for later.

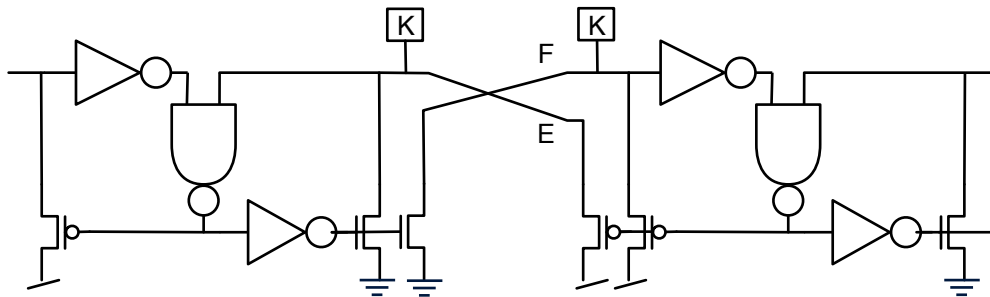


Figure 4: GasP module circuit: separate request and acknowledge

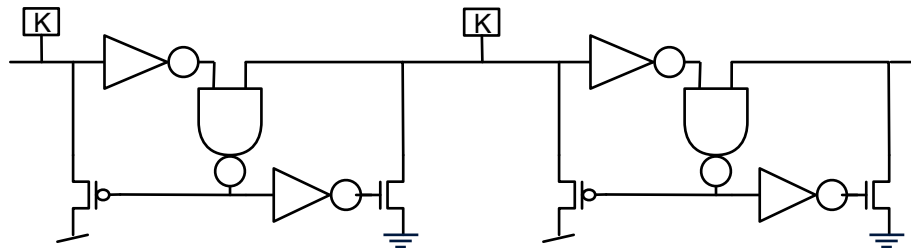


Figure 5: GasP module implementation: merged request and acknowledge

3 Class hierarchy

Figure ?? shows the class hierarchy of the GasP model framework for Archsim. Only most important attributes and methods are shown in the class diagram of Figure ??.

All of the classes extend the abstract class Entity, which defines the basic common attributes and methods. That is, each entity has a name, ID, and a Group object as a parent. Moreover, note that each entity has a selfCheck method. Invocation of this method causes the entity to check consistency of its connections and consistency of its internal structure.

3.1 Group

Class Group provides us with means to associate related objects and to organize the model in a hierarchical manner. For example, all of the entities comprising a FIFO can belong to the same group. Moreover, a factory object that can make FIFOs would return a subclass of a group that contains all objects comprising a FIFO of desired length. Such a subclass would also provide additional methods that would, for example, allow one to access the head and the tail of the FIFO.

Internally, a group contains a set of entities. Note that a group could belong to another larger group.

A group has a name. The full name of each object is a concatenation of names of its parents and the name of the object itself. For example, an action object with the base name “Head” that belongs to the group named “LongFifo” has a full name “LongFifo.Head”.

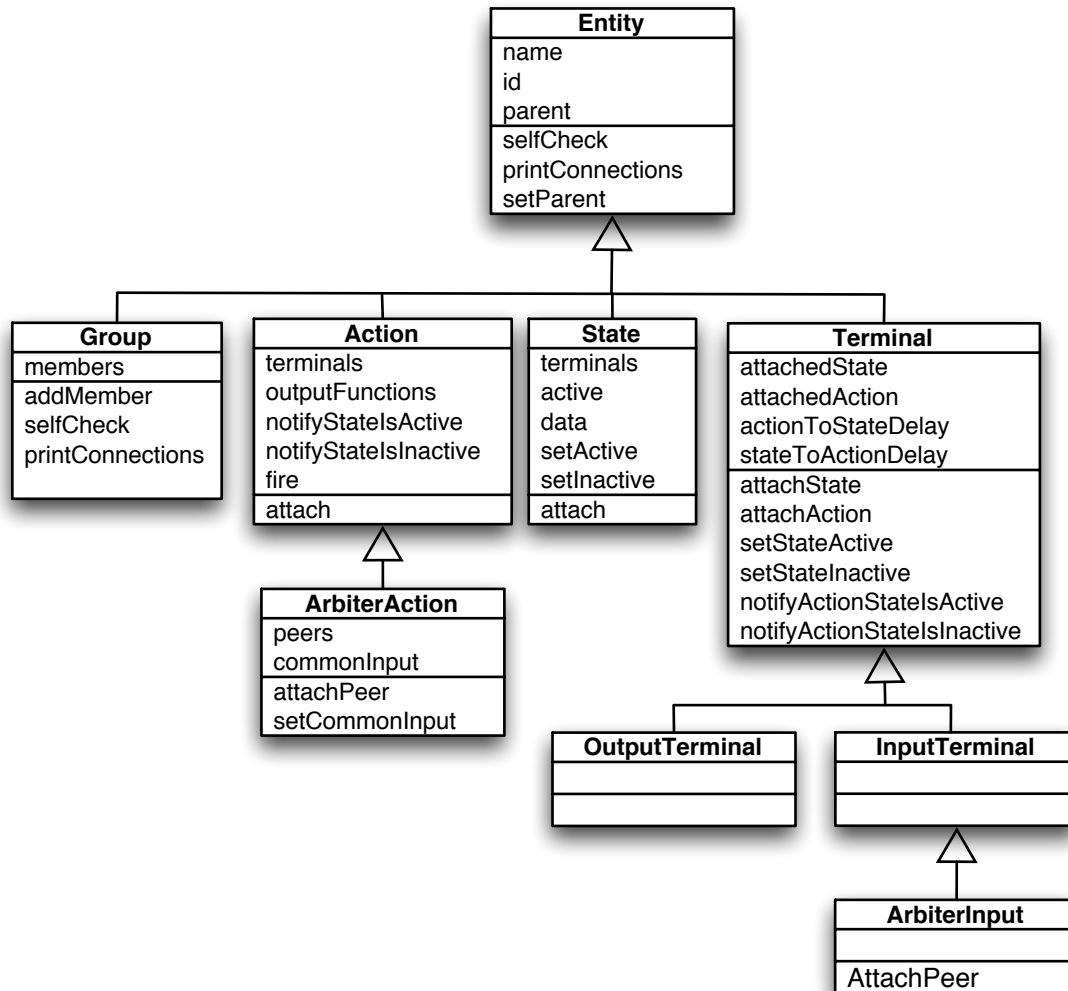


Figure 6: GasP framework class hierarchy

3.2 State

Class State directly corresponds to the notion of state in Ebergen's notation. A state contains the set of terminals that are connected to the state, and the state can store a data item, which is of class Object.

A State object has methods for setting the State object active or inactive. These methods are intended for initialization, or for manipulation of the state during a simulation for debugging purposes.

A State also contains two commands, one for making the state active and one for making it inactive. These commands are used in simulation events that are scheduled by terminals that are attached to the state. When a state changes from active to inactive and vice-versa, it notifies all of the attached actions of the change. Note that even the action that fired and caused the change of state receives a notification.

The general principle followed in our implementation of the GasP model framework is that an entity communicates only with directly attached entities. This means, for example, that an action will never directly invoke a state, but will always use the common terminal as the intermediary.

3.3 Terminal

Class Terminal directly corresponds to the notion of terminal in Ebergen's notation. We distinguish input and output terminals through subclassing. Each terminal has an attached state and an attached action. Moreover, a terminal contains two delays, one for communication from the attached state to the attached action, and one for communication in the reverse direction. The two delays can be different.

The terminal knows of four commands, which are passed to the terminal when it connects to a state and an action. Two commands are used for communication from the attached action to the attached state - one command makes the state active, and the other makes it inactive. The other two commands are used for notifying the action that the state has become either active or inactive.

A input terminal is, by default, self-resetting, but can be turned into non-self-resetting through invocation of method `beNonResetting`.

3.3.1 ArbiterInput

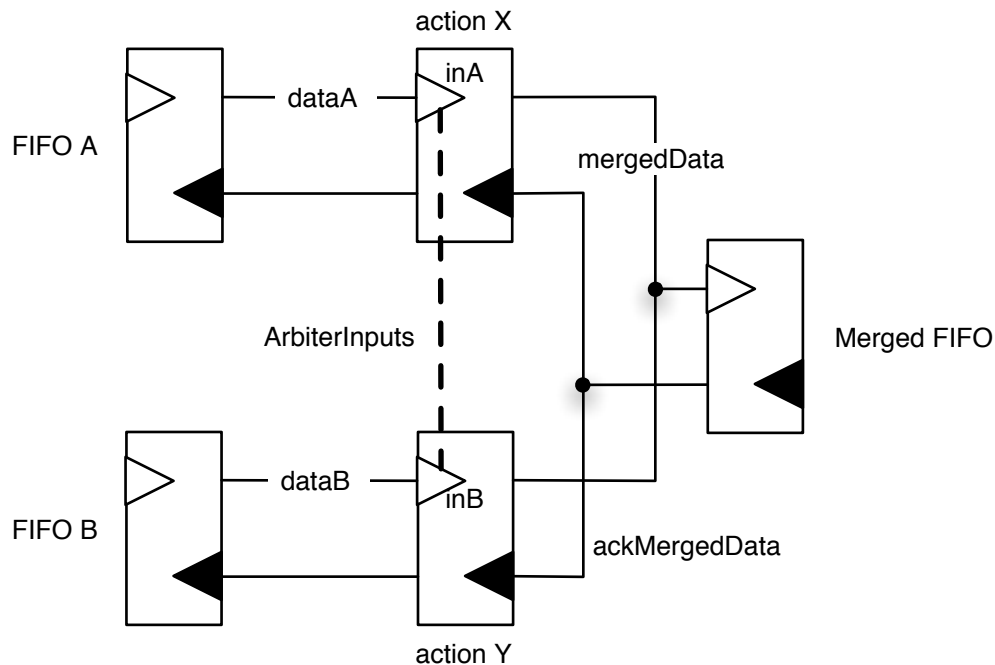


Figure 7: Arbiter input terminals used in merging two FIFOs

An arbiter input terminal extends Ebergen's notation by adding arbitration to GasP structures. See Figure 7 in [?] for a diagram of a circuit modeled by an arbiter input terminal.

In isolation, an arbiter input operates exactly the same as the ordinary input terminal. However, when a set of ArbiterInputs are connected as peers, arbitration is included in their behavior. More precisely, arbitration prevents more than one ArbiterInput at a time informing its attached action that the attached state has become active. For example, consider merging of two FIFOs shown in Figure ???. The two input terminals connected with a dashed lines are peer ArbiterInputs. Suppose both FIFO A and FIFO B are ready to pass data to the merged FIFO. This means that both state dataA and dataB are active, both requesting to transfer data to merged FIFO. However, both action X and Y cannot fire at the same time. That is, data must be transferred from either state dataA or dataB to state mergedData, then the data must be consumed by the

merged FIFO, and only then the next data transfer takes place.

Adding arbitration to input terminals inA and inB can help us introduce the desired mutual exclusion. The dashed line indicates that these two terminals are peer ArbiterInputs. If both state dataA and dataB become active at the same time, arbitration takes place between the two terminals. Only the winner of the arbitration will inform its action that the attached state became active. Assume the winner was terminal inA. This means that action X may fire as soon as the state connected to its other input terminal is active. At the same time, terminal inB is blocked and it cannot inform action Y that state dataB became active. Terminal inB remains blocked until action X fires and terminal inA caused state dataA to become inactive. At that time, the block for terminal inB is removed and action Y is informed that state dataB is active.

Note that the behavior as described in the scenario above depends on the outcome of the following race: State ackMergedData must become inactive before the block of terminal inB is released and action Y receives the signal indicating that state dataB is active. To make sure this race has the desired outcome, we must set the arbitration delay to an appropriate value.

Note that this style of arbitration can lead to deadlock. Deadlock can take place because the arbitration takes place before adjacent actions are ready to fire. That is, the winner of the arbitration may be adjacent to an action that cannot fire before the action attached to the loser of the arbitration has fired. Despite this possibility of deadlock, this form of arbitration may be practical in simple configurations such as the one shown in Figure ??.

3.4 Action

Class Action directly corresponds to the notion of action in Ebergen's notation. The basic premise in the operation of an action is that an action can fire when all of its inputs are active. By "inputs" we mean states attached to input terminals attached to the action. In order to know when to fire, an Action object keeps a count of active inputs. Every time an input becomes active, the count is incremented, and every time an input becomes inactive, the count is decremented. When the count is equal to the number of inputs, the action can fire. A additional safety check is built in to make sure that an action fires only when appropriate: Before an action object fires, it probes all of the input states to check whether they are active. Why is this necessary? An input state may have been rendered inactive by some other action and the notification that the state is inactive has not yet reached the action that is about to fire. If such a case is detected, it indicates a timing problem with the simulated GasP structure: a runtime error is reported and the simulation is terminated.

When an action fires, it makes its input states attached to self-resetting input terminals inactive, and all of the output states become active. What also happens is that data is moved from input states to output states. In fact, the data is not simply moved from input states to output states - there may be, for example, a different number of input states than there are output states. For each output terminal, and consequently for each output state, we must define an output function. The output function takes as input the data from all input states and computes the resulting output. Examples of outputs are a simple unary value, the sum of all inputs, a simple copy of a specific input, the largest value of all inputs, n lower bits from a specific input and so on.

We use private classes to define output functions. For each type of an action, we construct a subclass of Action, and within that subclass, we create a private class for each output function. These private classes are subclasses of Function, described in [?]. Each output function is associated with a name of an output terminal. The self-check method that is provided for the generic Action class verifies that each of the output terminals that are attached to the action has a corresponding output function.

Because processing of data may cause additional delay, a delay is associated with each output function. This means that firing of an action consists of several steps: First, values are computed for each output function, and completion of the computation is scheduled as an event with the delay specific to each output function. When the computation is complete, the corresponding output terminal is notified that the state

must be made active with the computed value passed to the state. The terminal schedules an event to make the state active with the delay specific to that terminal. That event then reaches the state, which becomes active. Moreover, the state schedules notifications for all attached actions. These notifications are scheduled as events with delays specific to each of the terminals that connect the state and one of its attached actions.

3.4.1 ArbiterAction

An arbiter action is our second approach at introducing arbitration in GasP. This approach is based mostly on suggestions made by Jo Ebergen, and on [?].

Note that the arbitration approach presented in Section [?] offers a more general approach to arbitration than the ArbiterActions, where there is no ability to express preference on the outcome of the arbitration.

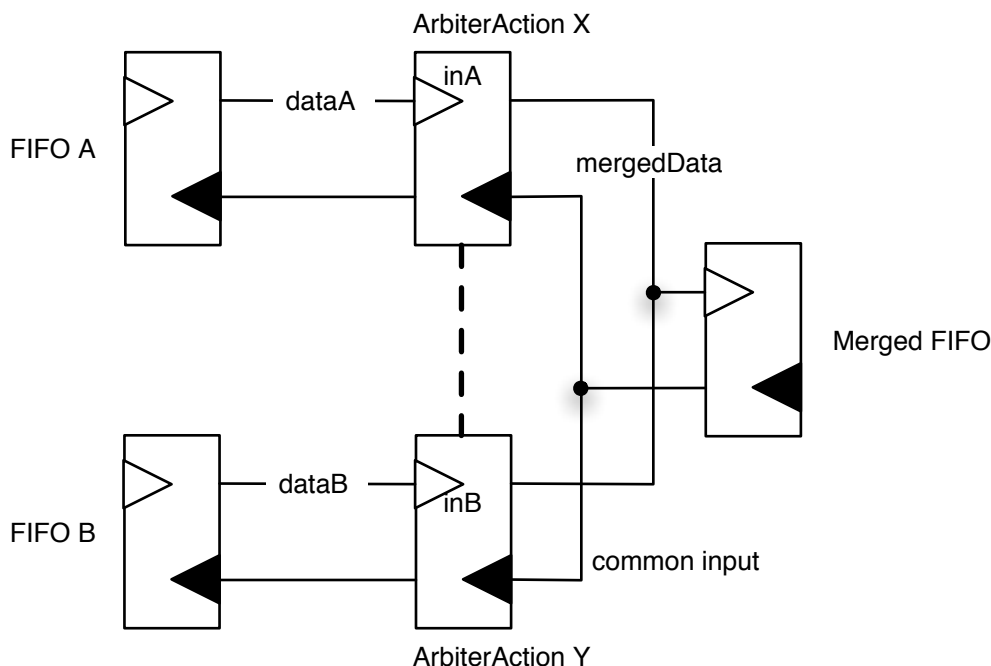


Figure 8: Arbiter actions used in merging two FIFOs

In isolation, an ArbiterAction behaves exactly the same as an ordinary Action. However, a group of ArbiterActions can share a common arbiter, which enables only one ArbiterAction to fire at a time. The arbiter is an object that is shared by the ArbiterAction peers, but is not visible outside the ArbiterAction. An ArbiterAction can be a member of only one group of peers. That is, internally, an arbiter action has access to only one arbiter.

Let us examine operation of an arbiter action more closely. When all inputs to an ArbiterAction are active, then the ArbiterAction issues a request to an arbiter. The arbiter action cannot fire until the request is granted. When the arbiter grants the outstanding request, the arbiter action can fire. No other peer ArbiterAction can fire until the arbiter is released, and their request granted. The ArbiterAction releases the arbiter when in input marked as the “done input” becomes active. The name “done input” comes signal naming in and RGD arbiter, where a separate signal denotes that the arbiter may grant a subsequent request. That is, when the “done input” becomes active it is known that the “critical section” that is protected by the arbiter is completed.

The configuration shown in Figure ?? illustrates that the “done input” may be common to all peer ArbiterActions. The “done input” in Figure ?? indicates that the data has been passed to the merged FIFO. The next data item may be consumed from either of the input FIFOs, depending on which one wins the arbitration.