April 3, 1995

SRC Research Report 136

A Functional Specification of the Alpha AXP™ Shared Memory Model

Manfred Broy

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

A Functional Specification of the Alpha AXP^{TM} Shared Memory Model

Manfred Broy

April 3, 1995

Manfred Broy is at the Institut für Informatik, Technische Universität München, D–80290 München, Germany.

E-mail: `broy@informatik.tu-muenchen.de`

This work was partially sponsored by the German Sonderforschungsbereich 342 "Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen"

Abstract

We give a functional specification of the Alpha AXP architecture with special emphasis on the Alpha Shared Memory Model. We keep the specification as abstract as possible and modular in the sense that we provide an independent description of the processors and the memory. We show how to handle a number of critical aspects of the Alpha architecture within the functional model, such as the specification of basic assumptions about the behavior of the processors and the exclusion of causal loops. We use the model for specifying the notion of lookahead and shortcut optimization for the behaviors of the processors. This allows us to define the concept of correct processor behavior by using the conservative sequential behavior as a reference. Finally, we extend the model to the constructs for synchronization in the Alpha architecture and include the instructions “read locked” as well as “store conditional”.

Contents

1 Introduction

The Alpha AXP architecture is a RISC architecture that was designed for high performance and longevity (see [AARM 92]). A major design goal was to avoid any elements that would become limitations during a 15-25 year design horizon. The architecture allows and supports a factor-of-1000 increase in performance. It allows multiple instruction streams and the execution of many instructions per clock cycle, as well as multiple data streams and instruction stream memory management. The interaction between the processors and the memory is highly underspecified to allow implementations to be very flexible in adopting future speed-up techniques.

We give a functional specification of the Alpha architecture and in particular of the Alpha Shared Memory using a data flow model. The "nondeterminism" in the architecture is modeled by underspecification.

The mathematical basis for the functional model is "streams" and "stream processing functions" (for a short introduction see the appendix B or, for more details, [Broy 90]). Every device in the Alpha architecture is described by a logical formula that characterizes the stream processing functions representing the behavior of the components and the streams flowing between the components. This way the behavior of the processors and the behavior of the memory are described by separate logical formulas. The behavior of the overall system is obtained by a composition of the formulas describing the behaviors of the subsystems. This structure supports independent reasoning about the different devices. It is possible to reason about the behavior of the complete system by using the specification of its components.

There are numerous papers that deal with sequential consistency, serializability and related aspects of concurrent access to memory and data base systems (see [Shasha, Smir 88]). For modern machine architectures with caches and concurrent execution of processors difficult issues of programming arise (see, for example, [Attiya, Friedmann 94]). Our main motivation is a simple and powerful model of such architectures. It also allows us to clarify a number of issues of high practical relevance, such as the exclusion of causal loops and the definition of correct optimization. These issues have not been addressed sufficiently in the literature, so far.

The paper is structured as follows. Section 2 describes the basic structure of the model of the Alpha AXP architecture and its components. In sections 3 and 4 we give the description of the behaviors of the memory and of the processors. Section 5 summarizes the description. The remaining part of the paper shows how to make use of the description. In section 6 we study the exclusion of causal loops and analyze basic assumptions about the behavior of processors. We characterize speed-ups of the processors by advanced lookahead when issuing memory requests. We define the concept of lookahead and shortcut optimization of processor behaviors. This allows us to define the correctness for processor behavior using the strictly sequential behavior as the reference. Roughly speaking, the behavior of a processor is correct if it is an optimization of the behavior of a strictly sequential processor. We prove that for such optimizations the results of programs that run without shared memory are effectively equivalent to the results that we obtain for the processors with strictly sequential behavior and a memory that does not reorder memory accesses.

In section 7 we extend the model to memory access with locked loads and conditional stores. We show for a simple example how to prove properties of programs using synchronization protocols.

In appendix A we give a more liberal scheduling strategy for the memory requests of the processors than the one described in [AARM 92]. In appendix B we include the essentials of the mathematics of the functional system model.

2 The Model

In this section we describe the structure of the mathematical model that we suggest for the Alpha architecture.

2.1 Basic Components

The Alpha architecture consists of a number of *processors* and a *memory*. The memory consists of a set of *locations* in which data can be stored. By PRC we denote the set of processors, by LOC we denote the set of locations, by $DATA$ we denote the set of data values. The set $DATA$ includes instructions.

2.2 Actions

In the Alpha architecture the relevant actions for the interaction between the processors and the memory are read and write actions, instruction fetches, and memory and instruction barriers. The table given in Figure 1 lists the syntax of the actions and introduces some useful selector functions.

However, we prefer to think about the execution of these actions not as one atomic step but in terms of an interaction between the processors and the memory. To execute an action, the processor issues a request (similarly to a procedure call in a conventional programming language) and the memory responds to it by a memory response (similarly to a returned result for a procedure call). Following this concept we decompose each of the actions into memory requests and memory responses. Processors issue *memory requests* and receive *read response messages*. The memory receives memory requests and issues read response messages.

Action	Syntax	processor	location	data
Read action	`P:R(x,a)`	P	x	a
Write action	`P:W(x,a)`	P	x	a
Instruction fetch	`P:I(x,b)`	P	x	b
Memory barrier	`P:MB`	P	–	–
Instruction memory barrier	`P:IMB`	P	–	–

Figure 1: Table of memory actions and selector functions processor, location and data

Action name	Syntax	Memory Request	Response
Read action	`P:R(x,a)`	`P:R?x`	`P:R(x,a)`
Write action	`P:W(x,a)`	`P:W(x,a)`	–
Instruction fetch	`P:I(x,b)`	`P:I?x`	`P:I(x,b)`
Memory barrier	`P:MB`	`P:MB`	–
Instruction memory barrier	`P:IMB`	`P:IMB`	–

Figure 2: Table of actions and their split into requests and responses

Let P be a processor, x be a location, and a be some data element. A memory request is one of the following:

- a write request, represented by the action `P:W(x,a)`;
- a read request, represented by the action `P:R?x`;
- a memory barrier request, represented by the action `P:MB`;
- an instruction fetch request, represented by the action `P:I?x`;
- or an instruction memory barrier request, represented by the action `P:IMB`.

By REQ we denote the set of memory requests.

The memory replies to a read memory request by a *read response message*. A read response message is represented by `P:R(x,a)`. The memory replies to an instruction fetch memory request by an *instruction fetch response message*. An instruction fetch response message is represented by `P:I(x,a)`. By RFR we denote the set of read response and instruction fetch response messages.

The table given in Figure 2 summarizes the decomposition of actions into memory requests and responses to them. For write actions and barrier requests, responses are not needed.

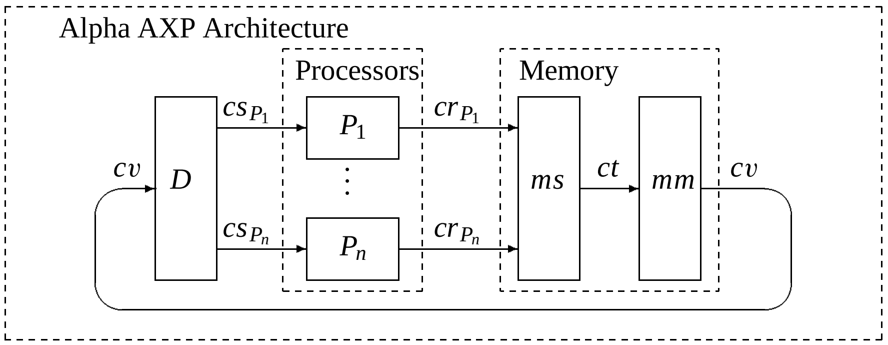

Figure 3: Data flow graph of the Alpha architecture

The decomposition of actions into requests and responses has a number of advantages. This way we are able to distinguish between the *issue order* of memory requests, determined by the processors, the *access order* of memory requests, determined by the memory, and the *reception order* of the responses by the processors. This allows us to model the scheduling discipline of memory access more explicitly and modularize the model more effectively.

2.3 The Model of the Architecture

We model the Alpha architecture by a data flow model as given in Figure 3. In the data flow model the processors and the memory are independent units that communicate asynchronously via message exchange. The processors are connected to the memory by channels. The channels are denoted by cv, cs_P, cr_P , and ct where P is a processor.

Each processor sends a stream of memory requests to the memory and receives a stream of read response messages from the memory. In a computation of the Alpha architecture the behavior of each processor $P \in PRC$ is modeled by a stream processing function

$$f_P : RFR^{\omega} \to REQ^{\omega}$$

A stream processing function is a prefix continuous function on streams. A short introduction to the mathematics of streams and stream processing functions is given in appendix B. There we also introduce the notation we are using throughout the paper for writing logical formulas for streams and stream processing functions.

In a computation a stream is associated with each of the channels cv, cs_P, cr_P, and ct. These streams will be denoted by $\vec{cv}$, $\vec{cs}_P$, $\vec{cr}_P$, and $\vec{ct}$ respectively.

The memory receives the streams of memory requests from the processors and sends back in response a stream of read response messages. Mathematically the memory is modeled by a stream processing function

$$md : (PRC \rightarrow REQ^{\omega}) \rightarrow RFR^{\omega}$$

Since each response message is labeled by the identifier of the processor that has requested it, the stream of read response messages produced by the memory can easily be split into individual streams of responses for each of the processors.

The Alpha architecture is modeled as follows: for every processor P we represent the history of messages exchanged between the processor and the memory by the input stream $\vec{cs}_P \in RFR^{\omega}$ and the output stream $\vec{cr}_P \in REQ^{\omega}$ of the processor. Of course, the stream $\vec{cr}_P$ is the result of the processor function f_P applied to the stream $\vec{cs}_P$ of memory responses. This is expressed mathematically as:

$$\vec{cr}_P = f_P(\vec{cs}_P)$$

By $\vec{cr}$ we denote the mapping that associates with each processor its request stream produced by the processors in PRC. By $\vec{cs}$ we denote the mapping that associates with each processor its response stream produced by memory for the processor in PRC. Formally $\vec{cr}$ is a mapping that associates a stream of memory requests with every processor:

$$\vec{cr} : PRC \rightarrow REQ^{\omega}$$

and $\vec{cs}$ is a mapping that associates the input stream of memory request responses with every processor:

$$\vec{cs} : PRC \rightarrow RFR^{\omega}$$

We specify the history of messages between the memory and the processors by a stream $\vec{cv} \in RFR^{\omega}$ that represents the stream of all read request responses produced by the memory. We do not assume any responses for write or barrier requests.

By the split component called D in the data flow graph given in Figure 3 we obtain the input streams $\vec{cs}_P$ for each of the processors from $\vec{cv}$.

In the functional approach, a distributed system is modeled by associating a set of monotonic stream-processing functions with each component. An instance of the behavior of the system, a computation, is obtained as follows: for each component one function is chosen out of the set of stream processing functions modelling that component. By these functions we associate a stream with each channel that connects the components. These streams are determined as the least fixpoint for the functions associated with each of the components. In the case of the scheduler, the choice of the function representing the behavior of the scheduler is constrained by specific properties of the streams that form the fixpoints. This modelling technique, where there is a dependency between the selected behavior function and its actual input stream, is called "input choice specification" and is described in detail in [Broy 93].

Causality is a decisive notion in information processing and message passing systems. It imposes a "physical law" on the flow of information. Roughly speaking, we understand by the phrase "action *a* is causal for action *b*" that *b* cannot take place before *a* has happened. In distributed systems that consist of a set of components that exchange messages via channels between the components, we distinguish two forms of causality:

- *component causality*: a component cannot issue an output message before it has received the input required for computing the content of the output messages. This form of causality between input and output for a component is captured in the functional model by the *monotonicity* constraint.
- *information exchange causality*: a message is not received before it has been sent. This is captured in the functional model by the least fixpoint principle for the recursive stream equations for the channels.

These two forms of causality are an integral part of the functional system model. They are the basis for ruling out "causal loops".

3 The Memory Architecture

We formalize the requirements for the behavior of the memory by characterizing the relation between its input streams and its output stream. The requirements for the memory fall into the following two categories:

- *Proper memory access scheduling*: the memory requests of the individual processors are rescheduled and merged; the scheduling is restricted with respect to memory barriers and memory instruction barriers as well as read and write requests for the same location.
- *Read/write consistency*: in response to read requests, those data elements are sent that have been written by the most recent write requests for that location in the access stream.

Both requirements are described in [AARM 92] in a semiformal way that does not provide explicit answers to a number of critical questions. For instance, it is not clear whether so-called *causal loops* are implicitly excluded. Certainly they are not explicitly excluded in [AARM 92]. We shall come back to the issue of causal loops after we have introduced the mathematical model.

To keep the model simple and to achieve a good separation of concerns, we decompose the function md modeling the behavior of the memory into two prefix continuous stream processing functions, the memory scheduler

$$ms : (PRC \to REQ^{\omega}) \to REQ^{\omega}$$

which takes care of the proper memory access scheduling and determines the access stream and thus the access order, and the memory manager

$$mm : REQ^{\omega} \to RFR^{\omega}$$

which models read/write consistency (see Figure 3). The function md then is simply obtained by composing ms and mm. Mathematically we specify the result stream of the function md for all the streams of memory requests $r : PRC \to REQ^{\omega}$ by the following equation:

$$md(r) = mm(ms(r))$$

Precise specifications of the functions ms and mm are given in the following section.

As shown in the data flow diagram given in Figure 3, we specify the history of messages between the memory scheduler and the memory service by an access stream

$$\vec{ct} \in REQ^{\omega}$$

and the history of messages between the memory service and the processors by a stream of read and instruction fetch response messages

$$\vec{cv} \in RFR^{\omega}$$

The stream $\vec{ct}$ denotes sequence of scheduled requests for the memory and determines the outcome of the function ms. Mathematically, this is expressed by the following equation:

$$\vec{ct} = ms(\vec{cr})$$

The stream $\vec{ct}$ denotes the access order of the memory requests.

The stream $\vec{cv}$ denotes the outcome of the function mm. Mathematically, this is expressed by the equation:

$$\vec{cv} = mm(\vec{ct})$$

1st ↓ \ 2nd →	P:I?x	P:R?x	P:W(x,b)	P:MB	P:IMB
P:I?x	$<$		$<$	$<$	$<$
P:R?x		$<$	$<$	$<$	$<$
P:W(x,a)		$<$	$<$	$<$	$<$
P:MB		$<$	$<$	$<$	$<$
P:IMB	$<$	$<$	$<$	$<$	$<$

Figure 4: Table of relations between requests in the issue order and the access order

¿From the stream $\vec{cv}$ the input streams $\vec{cs}_P \in RFR^{\omega}$ of memory responses for the individual processors $P \in PRC$ can easily be computed.

3.1 Scheduling the Memory Requests

We give the specification for the function ms by the relation between the streams $\vec{cr}_P$ and $\vec{ct}$. The function ms merges and reschedules its input streams. This merge and rearrangement follows the rules described in [AARM 92]. In this section we formalize these rules.

The table given by Figure 4 is taken from [AARM 92]. It shows the reordering restrictions that are imposed by the issue streams $\vec{cr}_P$ onto the access stream $\vec{ct}$. The table expresses the following requirement in terms of the mathematical model: for every pair of memory requests c_1, c_2 for process P for which we have $c_1 < c_2$ in the table in Figure 4, their relative order in the issue stream $\vec{cr}_P$ and in access stream $\vec{ct}$ coincide. Mathematically we write

$$c_1 < c_2 \Rightarrow [c_1 < c_2]\, in\, [\vec{cr}_P, \vec{ct}\ ]$$

Here (for arbitrary streams r, t) the proposition

$$[c_1 < c_2]\, in\, [r, t]$$

stands for the following two conditions: condition (1) expresses that all requests c_1 and c_2 in r are eventually scheduled in t, and only requests that have been issued are scheduled. Its formalization is rather straightforward. Every memory request c in each of the issue streams r also appears in the access stream t. Since all requests are labeled by the processors, all elements in issue streams of different processors are distinct. Mathematically expressed condition (1) reads as follows (for the definition of the filter function "$x|_M$" see the appendix B):

$$r|_{\{c_i\}} = t|_{\{c_i\}} \quad for\ i \in \{1, 2\}$$

Condition (2) expresses that the requests c_1 compared to the requests c_2 may not come later in the stream t than in stream r. In other words, the scheduler may not move the requests c_2 to the left over the requests c_1 to obtain the substream of the requests c_1 and c_2 in t from that in r. Mathematically expressed, condition (2) reads as follows:

$$\forall k \in \mathbb{N} : h_k(r)|_{\{c_1\}} \sqsubseteq h_k(t)|_{\{c_1\}}$$

where

$$h_k(s) = (s|_{\{c_1,c_2\}})[1:k]$$

Here we use the following notation. For a stream s we denote by $s[1:k]$ the first k elements of the stream. If a stream s has less than k elements then $s[1:k] = s$.

The relationship between the issue streams and the access stream as formalized above has the following consequence: for every set M of requests of the processor P that are all pairwise in the $<$-relation, the substreams of the requests in M in the issue streams and the access stream are identical. Mathematically expressed:

$$(\forall c, d \in M : c < d \vee d < c) \Rightarrow \vec{cr}_P|_M = \vec{ct}\,|_M$$

This can be shown for finite streams $\vec{cr}_P$ and $\vec{ct}$ by induction on the length of the stream $\vec{ct}$. For infinite streams it follows by the continuity of the function that filters out a substream.

Based on the notation introduced above, we can now formulate the correctness requirement of the scheduling function. We decompose the requirements for the scheduler into the safety and liveness properties. In a first step we give just the safety property. It is represented by a simple predicate characterizing the set of scheduling functions that are correct with respect to safety. The liveness condition for the function ms is not a simple predicate on ms, but depends also on the particular request streams in the data flow model. A function

$$ms : (PRC \to REQ^{\omega}) \to REQ^{\omega}$$

is called a *safe* scheduling function if for all request streams $r \in (PRC \to REQ^{\omega})$ for which r_P contains only requests issued by processor P we have

$$\begin{gathered} \exists t \in REQ^{\omega} : ms(r) \sqsubseteq t \wedge \\ \forall c_1, c_2, P : c_1 < c_2 \Rightarrow [c_1 < c_2]\, in\, [r_P, t] \end{gathered}$$

This requirement is a safety condition for the scheduler. It expresses that the requests are in a proper relationship, if they are scheduled at all. It does not express the liveness property that all requests are eventually scheduled. The liveness condition for the scheduler is not a simple predicate on ms, but depends also on the particular request streams in the data flow model. It will therefore be added as a constraint for the scheduler functions ms with respect to the stream $\vec{cr}$ in the network modeling the Alpha architecture in section 5 where we summarize the model.

3.2 Responding to the Scheduled Memory Requests

The behavior of the memory manager is represented by the function mm, which executes the memory requests in the order produced by the memory request scheduler ms. Its specification is rather simple. In response to each read request the memory manager sends the data element that has been written by the most recently executed write request. According to the rescheduling of requests, the execution order is not necessarily identical to the issue order.

We ignore here the possibility that the memory may be initialized by other means than memory write requests. Explicit initialization can easily be included, however.

For our mathematical model, we express read/write consistency by the following formula: for all streams $u, w \in REQ^*, s \in REQ^\omega$ we assume:

$$\begin{gathered}(u \frown w)|_{RI} = \langle\rangle \wedge w|_{W(x)} = \langle\rangle \\ \Rightarrow \\ mm(u \frown \texttt{Q:W(x,a)} \frown w \frown \texttt{P:R?x} \frown s) = \texttt{P:R(x,a)} \frown mm(u \frown \texttt{Q:W(x,a)} \frown w \frown s) \\ \wedge \\ mm(u \frown \texttt{Q:W(x,a)} \frown w \frown \texttt{P:I?x} \frown s) = \texttt{P:I(x,a)} \frown mm(u \frown \texttt{Q:W(x,a)} \frown w \frown s)\end{gathered}$$

where $W(x)$ is the set of all write requests for location x:

$$W(x) = \{\texttt{P:W(x,a)} \in REQ : P \in PRC \wedge a \in DATA\}$$

and RI is the set of all the read requests and instruction fetch requests:

$$RI = \{\texttt{P:R?x} : P \in PRC \wedge x \in LOC\} \cup \{\texttt{P:I?x} : P \in PRC \wedge x \in LOC\}$$

The stream $\vec{cv}$ is the result of applying the memory function mm to the stream $\vec{ct}$ of memory requests.

$$\vec{cv} = mm(\vec{ct})$$

The stream $\vec{cv}$ of read request and instruction fetch responses produced by the memory manager mm can easily be decomposed into one memory request response stream $\vec{cs}_P$ for each processor P. We specify:

$$\vec{cs}_P = D_P(\vec{cv}) \quad \text{where} \quad \forall v : D_P(v) = v|_{R(P)}$$

where the set $R(P)$ of read request responses of the processor P is specified as follows:

$$R(P) = \{e \in RFR : processor(e) = P\}$$

Of course, there are other ways to decompose the memory function md. For instance, we may introduce a merge function that interleaves all streams of read requests produced by the processors in a stream of read requests. Using this merge function we can then either introduce for each processor an individual memory scheduler that reschedules all memory requests before they are merged, or merge all streams of read requests produced by the processors in a stream of read requests and then do the memory scheduling by rescheduling this stream.

4 The Processors

The behavior of a processor is modeled by a prefix continuous stream processing function. In this section we show only schematically how we model the instruction cycle of an Alpha processor by such a function.

A processor has a local state that consists of all the entries in its registers and maybe additional information. The set of states of a processor is denoted by PRC-State. Initially, a processor starts from an initial state by issuing a finite sequence of memory requests.

Whenever a processor receives a memory request response, it changes its local state and issues a finite (possibly empty) sequence of memory requests. For formalizing this behavior of a processor, we introduce the following two functions:

$$mr : PRCState \times RFR \to REQ^*$$

$$sc : PRCState \times RFR \to PRCState$$

The function mr yields the sequence of memory requests issued by the processor in a state when receiving a memory response. The function sc yields the successor state of the processor. The behavior of the processor P is defined by the function f_P. We specify this function by the following equation:

$$f_P(s) = init^\frown exec(\sigma_P, s)$$

where σ_P is the initial state of the processor P, $init$ is the initial sequence of memory requests, and $exec$ is the function

$$exec : PRCState \times RFR^\omega \to REQ^\omega$$

specified by

$$exec(\sigma, c \frown s) = mr(\sigma, c) \frown exec(sc(\sigma, c), s)$$

Of course, a processor P issues only memory requests labeled by P.

We do not give a more detailed description of the individual instructions and their execution here. We come back to this issue in section 6.

5 Summary of the Model

In this section we summarize the functional model of the Alpha architecture as a reminder for the readers.

As indicated in Figure 3, we assume that the streams $\vec{cr}_P$, $\vec{cs}_P$ and $\vec{ct}$ are the least fixpoints of the following equations:

$$\begin{array}{rcll} \vec{cr}_P & = & f_P(\vec{cs}_P) & \text{for all processors } P \in PRC \\ \vec{cs}_P & = & \vec{cv}|_{R(P)} & \text{for all processors } P \in PRC \\ \vec{cv} & = & mm(ms(\vec{cr}\,)) & \end{array}$$

where for all processors $P \in PRC$ the function f_P and also the functions mm and ms are stream processing functions. This means in particular that they are prefix continuous. The function f_P is assumed to be a processor behavior. The function ms is required to be a safe scheduling function. The function mm fulfills the following requirement: for all streams $u, w \in REQ^*, s \in REQ^\omega$:

$$\begin{array}{c} u \frown w|_{RI} = \langle\rangle \wedge w|_{W(x)} = \langle\rangle \\ \Rightarrow \\ mm(u \frown \texttt{Q:W(x,a)} \frown w \frown \texttt{P:R?x} \frown s) = \texttt{P:R(x,a)} \frown mm(u \frown \texttt{Q:W(x,a)} \frown w \frown s) \\ \wedge \\ mm(u \frown \texttt{Q:W(x,a)} \frown w \frown \texttt{P:I?x} \frown s) = \texttt{P:I(x,a)} \frown mm(u \frown \texttt{Q:R(x,a)} \frown w \frown s) \end{array}$$

For the function ms we require the following liveness condition: every request c is eventually scheduled, mathematically expressed:

$$\vec{cr}_P|_{\{c\}} = ms(\vec{cr}_P)|_{\{c\}}$$

This is a liveness condition that restricts the choice of the scheduling function ms in addition to the safety properties required for ms making sure that all requests are eventually scheduled.

The prefix monotonicity requirement for the functions models the causality within the processors and thus the causality within their programs. The least fixpoint property of the streams described by the recursive equations models the causality between the sending and receiving of messages. If either of these requirements is dropped, then causal loops are no longer excluded.

6 The Model at Work

In this section we start with a short analysis of the functional model and then show how it can be used to formalize further properties of the processors and their execution of instruction streams.

6.1 Mathematics of the Model

The model given in the previous sections is based on the following mathematical concepts:

- *prefix continuous stream processing functions,*
- *recursive stream equations and least fixpoint interpretations for them,*
- *liveness constraints for the scheduling function*.

These concepts are well suited as a mathematical basis for data flow models. Hardware systems can also be understood as data flow systems. A large number of examples have demonstrated that the concepts work well for both.

The purpose and benefits of *mathematical* or *formal* models for information processing systems are manifold:

- Mathematical system models provide a consistent and precise description of the properties of a system, but nevertheless give freedom by leaving certain aspects deliberately unspecified; we speak of *underspecification*.
- In the process of deriving mathematical system models from informal descriptions, flaws, inaccuracies and omissions can be detected and clarified.
- Mathematical system models provide a reference basis for understanding and discussion.
- Mathematical system models provide a basis for a formal reasoning about a system, by which specific properties can be derived (and therefore verified).
- Mathematical system models provide a precise requirement specification for implementations by hardware or software systems.

The functional model given for the Alpha AXP architecture exhibits a number of typical properties. For instance, every response in the response stream is triggered by a request. This property is formalized by the following definition.

Definition 1 (Feasible Response Streams) A response stream $s \in RFR^{\omega}$ is called *feasible* for a processor with behavior f_P, if all the instances of responses in s are triggered by requests in the request streams $f_P(s)$. A response stream s is triggered by the request stream $f_P(s)$, if every prefix $\tilde{s}$ of the stream s contains for every instance of a read request and every instance of an instruction fetch request in $f_P(\tilde{s})$ at most one corresponding response. Mathematically expressed, a stream s is a feasible response stream for the behavior f_P of processor P, if for all its prefixes $\tilde{s} \sqsubseteq s$ the following two conditions are fulfilled:

$$\#\tilde{s}|_{\{\texttt{P:R?x}\}} \leq \# f_P(\tilde{s})|_{R(x)}$$

and

$$\#\tilde{s}|_{\{\texttt{P:I?x}\}} \leq \# f_P(\tilde{s})|_{I(x)}$$

where

$$R(x) = \{\texttt{P:R(x,a)} : a \in DATA\}$$

$$I(x) = \{\texttt{P:I(x,a)} : a \in DATA\}$$

We then write:

$$f_P \hookrightarrow_P s$$

If the inequalities above are strengthened to equalities for $\tilde{s} = s$, then the stream s is called a *complete and feasible* response stream for the processor behavior f_P.

$\square$

It is a straightforward exercise to show that in our model for the Alpha architecture according to the definition of the memory device for every processor P, every stream $\vec{cs}_P$ is a feasible response stream for f_P.

6.2 Causal Loops

In this subsection we study the problem of *causal loops*. It is widely accepted that there is a natural causality flow in information processing systems. More technically speaking, a particular message value cannot be sent by an interactive message passing system before all values on which it depends have been received.

In the Alpha architecture there are two kinds of processing units, the processors and the memory. The principle of causality can be applied to both of them. A processor cannot issue a memory write request before it receives the data to be written. The memory cannot respond to a read request before it receives the memory write request that supplies that data.

To make our discussion more concrete, let us look at a simple example. We assume that the processors P and Q execute the following two little programs. We use some pidgin assembler language here which has the two commands LD (for load) and ST (for store) only and works with the local registers of the processors. The registers are denoted by $R1$, $R2$, etc. Consider for the processor P the following program

$$\begin{array}{lll} LD & R1 & x \\ ST & R1 & y \end{array}$$

and for the processor Q the following program

$$\begin{array}{lll} LD & R2 & y \\ ST & R2 & x \end{array}$$

Let us assume that initially the value 0 is stored both in location x and in location y.

Of course, we may expect that, independent of the scheduling, the effect of executing these programs is that both the locations x and y invariantly have the value 0. Now let us consider the following access stream

```
P:R?x Q:R?y P:W(y,1) Q:W(x,2)
```

and the corresponding response stream:

```
P:R(x,2) Q:R(y,1)
```

This sequence of actions is certainly read/write consistent (only values are read that have been written before). It also fulfills all the requirements of memory access scheduling. So one may argue that it is a feasible access stream according to what is required in [AARM 92] for the memory. However, if we consider in addition the processor functions f_P and f_Q, we realize that this access order violates the causality requirements of the processors. For the processor P the write request can be issued only after the response to the read request has been received.

One may argue whether the paradoxical behavior of the causal loop as demonstrated above is actually admitted or not by the [AARM 92]. Such an exegesis is not very productive, however. As soon as one assumes a proper causality flow for the processors, causal loops are ruled out anyway. We claim that to any realistic processor, whatever advanced concepts it includes, the law of causality flow applies. Therefore, for any hardware, causal loops are excluded.

6.3 Sequential Execution of Processors

We do not want go into the definition of all the details of the execution of particular instructions in the Alpha AXP architecture. We therefore introduce as a reference for the behavior of the processors the behavior of a processor with a conservative sequential execution strategy. Such a behavior is obtained, if the processor sequentially executes the classical instruction cycle. This cycle first fetches the instruction indicated by the program counter, then computes the operand address and reads the operand from the memory or writes a value into the memory or issues a barrier request and then starts the cycle again.

This sequential execution strategy corresponds in our model to a particular stream processing function for the processors $P \in PRC$ which we denote by

$$f_P^{seq} : RFR^{\omega} \to REQ^{\omega}$$

This function will be used as a point of reference for formalizing the correctness of the behaviors of processors with a nonsequential execution strategy in the following.

6.4 Speeding Up Executions

In a most conservative implementation every processor issues just one memory request, then waits until it gets the response to this request, and only then issues the next request. This is the behavior represented by the function f_P^{seq}.

In contrast to this conservative sequential behavior, a more aggressive processor may send several memory requests before it receives some of the responses from the memory. This can lead to a speed-up in the interaction between the memory and the processor. We call such behavior, where several requests can be issued before a response is received, *issuing lookahead requests*.

We distinguish between the following strategies of processors in issuing lookahead requests.

If a processor issues only memory requests whose responses are certainly needed for the execution, we speak of *issuing conservative lookahead requests*. In this case, any missing response from the memory will eventually bring the processor to a waiting state.

A lookahead request processor may even issue memory read requests or instruction fetch requests whose responses it might not need. We call this strategy *issuing speculative lookahead requests*. It may lead to a considerable speed-up in accesses to the memory. The price is that some requests may be processed that turn out to be unnecessary. Clearly speculative write requests are not a safe concept and not considered therefore.

A further optimization can be obtained in cases where it can be recognized in advance that responses to certain read and instruction fetch requests that appear in the strictly sequential behavior are irrelevant for the further course of computation. Such requests need not even be issued. We call this *shortcut optimization.* One may argue that in a well–written program such irrelevant requests should not occur. However, even when doing multiplication of two values read from the memory, one of the values may turn out to be not relevant, if the other one is zero.

Shortcut optimization leads to a more radical change in the behavior of processors than lookahead optimizations. Responses to read and instruction fetch requests are generally needed by a processor to get the information (instructions and operands) required to continue its execution. Moreover, the arrival of responses is used to trigger further requests. This is most obvious in the conservative sequential behavior. The processor issues a request only when the response to the request issued previously has been received. So there is a causal relation between responses and the following requests. In lookahead optimizations, read and instruction fetch requests are issued earlier and more of them may be issued, but at least all the requests that appear in the nonoptimized behavior are eventually issued. Write requests and barrier requests are issued before the corresponding responses are received only as long as there are no actual data dependencies. In shortcut optimizations fewer requests may be issued by avoiding ones whose responses would be irrelevant. This may change the causality flow of a processor more radically.

It is one of the basic ideas of the Alpha AXP architecture that processors may issue lookahead memory requests, in order to speed up the general execution by parallelizing the memory accesses. In the following section we give a definition of the properties required for a processor to make sure that the replacement of a sequential processor by a processor with lookahead and shortcut optimizations does not change the effects of the executed programs as long as the program runs without any access to shared memory.

6.5 Optimizations of Processor Behavior

In this section we define the concept of valid optimization of the behavior of processors. Not all read request responses in a response stream coming from the memory are actually needed for further computation by the processor. Some read requests may serve only lookahead purposes and the responses to those might turn out to be obsolete after they have been requested. Similarly, certain responses may be irrelevant, since the transmitted values do not really influence the further computation.

For instance, if a value of a location is requested from the memory and later the transmitted value is multiplied by 0, its value is certainly not relevant. Of course, in practice it is very difficult to determine whether a response is relevant, since irrelevant responses may even trigger further read requests that lead to responses that turn out to be irrelevant, too. Optimized processors nevertheless may make use of the fact that a response is irrelevant. In this case a response might even not be requested.

Request streams have two effects on the memory: write requests change the state of the memory locations and therefore can be effectively observed by other processors, read and instruction fetch requests trigger responses by the memory and this way allow the processor to observe the current state of locations. Memory and instruction barrier requests restrict the memory access order. So it is not important for the effect of a request stream on the memory, how many barrier requests are created, but only how they restrict the access order. This leads us to the following definition.

Definition 2 (Feasible Schedulings of Memory Request Streams) For a request stream $r_1 \in REQ^{\omega}$ of processor $P \in PRC$ we call a request stream $r_2 \in (REQ\text{``}\{\texttt{P:MB}, \texttt{P:IMB}\})^{\omega}$ a *feasible scheduling* for the stream r_1, if there exists a stream $r_0 \in REQ^{\omega}$ such that for all requests $c_1, c_2 \in REQ$ we have:

$$c_1 < c_2 \Rightarrow [c_1 < c_2]\, in\, [r_1, r_0]$$

and

$$r_2 = r_0|_M \quad \text{where} \quad M = REQ\text{``}\{\texttt{P:MB}, \texttt{P:IMB}\}$$

□

For the correctness of optimizations it is decisive to identify under which conditions two request streams have the same effects for the memory such that they lead to the same set of possible observations by the processors. We may change the behavior of a processor such that it issues a different request stream as long as this is observably equivalent to the previous request stream. Along these lines we define the effective equivalence of request streams. The memory barrier and the instruction fetch barrier requests restrict only the rescheduling of the memory requests. Two request streams of a processor P are considered to be effectively equivalent, if they lead to the same effects and therefore can be called *observation equivalent*.

In the following definition, we do not require that effectively equivalent request streams have exactly the same substreams of barrier requests. We just require that they have the same read, instruction fetch and write requests and they include barrier requests that impose the same scheduling restrictions for them.

Definition 3 (Effective Equivalence of Memory Request Streams) Two request streams $r_1, r_2 \in REQ^{\omega}$ are called *effectively equivalent* for processor P, if their sets of feasible schedulings coincide. We write then $r_1 \overset{e}{\sim} r_2$.

□

To have a basis to speak about the substreams of relevant and irrelevant responses, we use the concept of the decomposition of streams.

Definition 4 (Decomposition of Streams) For an arbitrary stream $s \in M^{\omega}$ two streams s_1 and s_2 are called a *decomposition* of s, if the stream s can be split into the substreams s_1 and s_2. Mathematically expressed, if there exists an oracle $\beta \in \{1, 2\}^{\infty}$ such that for $i \in \{1, 2\}$

$$s_i = dis_i(s, \beta)$$

where the functions $dis_i : M^{\omega} \times \{1, 2\}^{\omega} \to M^{\omega}$ are specified by

$$dis_i(m^\frown s, i^\frown \beta) = m^\frown dis_i(s, \beta)$$

and

$$i \neq j \Rightarrow dis_i(m^\frown s, j^\frown \beta) = dis_i(s, \beta)$$

□

Based on this definition we can now define when for a processor a behavior is a lookahead optimization and when it is a shortcut optimization of another behavior.

Definition 5 (Lookahead Optimization) We consider the two behavior functions

$$f_1,\ f_2 : RFR^{\omega} \to REQ^{\omega}$$

for the processor P. The behavior f_2 is called a *lookahead optimization* of f_1, if, for every response stream $s_2 \in RFR^{\omega}$ that is feasible for f_2, the following condition holds: there is a decomposition of s_2 into a response stream $s_1 \in RFR^{\omega}$ and a response stream u_1 such that s_1 is a feasible response stream for f_1 and the request streams $f_1(s_1)$ and $f_2(s_2)$ are effectively equivalent.

□

Note that by the definition above, the processor with behavior f_2 is a refinement of the processor with behavior f_1 since every behavior that f_2 shows is an optimized behavior of a behavior of f_1.

Shortcut optimizations are more difficult to define. In a shortcut optimizations certain requests are not issued. This can be done if the effects achieved that way are equivalent to a behavior of the architecture for the nonoptimized issue stream.

Definition 6 (Shortcut Refinement of Issue Streams) We define a relation

$$\overset{e}{\rightsquigarrow}: REQ^{\omega} \times REQ^{\omega}$$

that defines the allowed shortcut optimizations. We give axiomatic rules:

$$\begin{gathered}
(\forall c_1, c_2 : c_1 < c_2 \Rightarrow [c_1 < c_2]\, in\, [r_1, r_2]) \Rightarrow r_1 \overset{e}{\rightsquigarrow} r_2 \\
r_1 \overset{e}{\rightsquigarrow} r_2 \wedge r_2 \overset{e}{\rightsquigarrow} r_3 \rightarrow r_1 \overset{e}{\rightsquigarrow} r_3 \\
r_1 \frown \texttt{MB} \frown \texttt{MB} \frown r_2 \overset{e}{\rightsquigarrow} r_1 \frown \texttt{MB} \frown r_2 \\
r_1 \frown \texttt{P:W(x,a)} \frown \texttt{P:W(x,b)} \frown r_2 \overset{e}{\rightsquigarrow} r_1 \frown \texttt{P:W(x,b)} \frown r_2 \\
r_1 \frown \texttt{P:R?x} \frown r_2 \overset{e}{\rightsquigarrow} r_1 \frown r_2
\end{gathered}$$

□

This definition essentially expresses that in a shortcut optimization we may leave out superfluous write requests, read requests and memory barriers. Of course, a read request may not be left out if the requested value is needed by the processor.

Definition 7 (Shortcut Optimization) Let us consider the two behavior functions

$$f_1,\ f_2 : RFR^{\omega} \rightarrow REQ^{\omega}$$

for the processor P. The behavior f_2 is called a *shortcut optimization* of f_1, if, for every response stream $s_2 \in RFR^{\omega}$ that is feasible for f_2, the following condition holds: there exists a response stream s_1 that is feasible for f_1 such that $f_1(s_1) \overset{e}{\rightsquigarrow} f(s_2)$.

□

The definition essentially says that in a lookahead optimization, for every response stream feasible for f_2 we can find a response stream for f_1 such that the response streams coincide after we get rid of some irrelevant responses for f_2 and the remaining requests are effectively equivalent.

A shortcut optimization can lead to a processor behavior with a quite different processor causality. This allows sophisticated optimizations where certain requests and the corresponding responses are recognized as unnecessary and therefore avoided even though they are relevant in the control flow of f_1, since there they trigger further relevant requests.

Conservative sequential execution is very restricted. In every state of the processor, the number of issued read and instruction fetch requests is at most one larger than the number of received responses. Mathematically, for a processor P with the behavior function f_P^{seq} the following property holds:

$$\tilde{s} \sqsubseteq \vec{cs}_P \Rightarrow \tilde{s} = \vec{cs}_P \vee 1 + \#\tilde{s} = \# f_P^{seq}(\tilde{s})|_{RI}$$

where RI is the set of read and instruction fetch requests. To get a less restricted behavior, we allow that certain requests are issued earlier. We extend the notion of processor correctness by shortcut and lookahead optimization. Based on the concept of sequential behavior and optimization, we can now define a notion of correctness of processor functions.

Definition 8 (Correct Processor Function) A processor P with processor function

$$f_P : RFR^{\omega} \rightarrow REQ^{\omega}$$

is called *correct*, if it is a shortcut optimization of a lookahead optimization of f_P^{seq}.

$\square$

Based on this definition, it is possible to prove that the observable behavior of the Alpha AXP architecture does not depend on the particular choice of the processor functions as long as all of the functions are correct.

Theorem 1 (Scheduling Robustness) When all processors $P \in PRC$ execute the sequential behavior represented by f_P^{seq}, the access stream is effectively equivalent to the access stream obtained, as long as all processor functions are correct,

- the processors do not share any memory, and instruction fetch locations and
- write locations are disjoint

Sketch of Proof: Since every processor function is correct, we can assume a response stream s_P and a request stream r_P corresponding to a lookahead optimization for the response stream s_P^{seq} and the request stream r_P^{seq} that we obtain for the sequential behavior such that the following holds: we can decompose the response stream s_P and the request stream r_P on one hand into substreams of the response stream $\vec{cs}_P$ and the request stream $\vec{cr}_P$ that we obtain for the considered behavior. On the other hand, we can decompose the response stream s_P and the request stream r_P into substreams of the response stream s_P^{seq} and the request stream r_P^{seq} that we obtain for the sequential behavior. Hence the request streams r_P^{seq} and request streams $\vec{cr}_P$ are effectively equivalent.

□

The theorem shows that, although different scheduling strategies can be used as well as different strategies for read and instruction fetch lookahead, as long as shared memory is not used and no writes occur to locations that occur in instruction fetch requests, the produced sequences of memory states for each of the processors coincide.

6.6 Assumptions about Executions

We have given only a very schematic description of the behavior of the processors, so far. In this section, we introduce a fundamental assumption about the behavior of processors. In a first definition, we introduce the notion of the equivalence of response streams.

Definition 9 (Equivalence of Response Streams) Two memory response streams $s_1, s_2 \in \mathit{RFR}$ are called *equivalent* and we write:

$$s_1 \sim s_2$$

if read response messages and instruction fetch response messages for the same locations are identical and in the same order: mathematically expressed, if for all locations x, the following proposition holds:

$$s_1|_{L(x)} = s_2|_{L(x)}$$

where

$$L(x) = \{\texttt{P:R(x,a)} : a \in \mathit{DATA}\} \cup \{\texttt{P:I(x,a)} : a \in \mathit{DATA}\}$$

□

Based on the notion of equivalence of response streams, we next define what it means that a processor is robust against reorderings of its response stream.

Definition 10 (Response Delay Robustness) For a processor P, a processor function

$$f_P : \mathit{RFR}^{\omega} \to \mathit{REQ}^{\omega}$$

is called *response delay robust* if, for all short cut optimizations g of f_P and all response streams s_1 and s_2 that are feasible for f_P and for which $s_1 \sim s_2$ holds, we have:

$$g(s_1) \overset{e}{\sim} g(s_2)$$

□

This leads to the following basic assumption about the behavior of processors:

- *Assumption: Response Delay Robustness*: for every processor P, its behavior function f_P is response delay robust.

Response delay robustness makes sure that the behavior of a processor does not critically depend on the order in which its memory requests are executed, as long as the scheduling constraints are fulfilled.

Let us next briefly analyze how significant the requirement is that the streams $\vec{cs}_P, \vec{cr}_P, \vec{ct}, \vec{cv}$ are least fixpoints. To answer this question, we also ask whether there exist solutions to the recursive equations for these streams that are not least fixpoints.

To be able to answer this question we need a further assumption, however. We did not say anything about the behavior of a processor in the case where it gets a response for which it did not send a request. We may assume, however, that a processor that has terminated its execution and comes to a halt does not issue any further requests even when it receives further (unrequested) responses.

Definition 11 (Response Satisfaction Property) A processor P with the behavior function f_P fulfills the *response satisfaction property* if, for all response streams $s \in RFR$, the following proposition is fulfilled:

$$f_P \hookrightarrow_P s \wedge s \text{ complete for } f_P \wedge s \sqsubseteq \tilde{s} \Rightarrow f_P(s) = f_P(\tilde{s})$$

This proposition expresses that the processor P with behavior f_P does not issue further memory requests after all its requests have been satisfied even if it gets further memory responses that it has not requested.

□

By assuming the response satisfaction property for all processors, we can prove that all fixpoints of the recursive equations are unique.

Theorem 2 (Uniqueness of Fixpoints) *Let us assume the response satisfaction property for all processors. If the streams $\vec{cs}_P, \vec{cr}_P, \vec{ct}, \vec{cv}$ are fixpoints of the defining equations and fulfill the constraints, then they are least fixpoints.*

Proof: Assume the streams $\vec{cs}_P, \vec{cr}_P, \vec{ct}, \vec{cv}$ fulfill the equations and constraints listed in section 5, but are not necessarily least fixpoints of the equations. Assume further that the streams $\tilde{s}_P$, $\tilde{r}_P$, $\tilde{t}$, $\tilde{v}$ also fulfill the equations and constraints for the same functions f_P, ms, md and furthermore:

$$\tilde{s}_P \sqsubseteq \vec{cs}_P \ \wedge \tilde{r}_P \sqsubseteq \vec{cr}_P \ \wedge \tilde{t} \sqsubseteq \vec{ct} \ \wedge \tilde{v} \sqsubseteq \vec{cv}$$

The streams $\vec{cs}_P$ are feasible responses for the streams f_P. Formally expressed, we have $f_P \hookrightarrow_P \vec{cs}_P$ and $\vec{cs}_P$ is complete for f_P. According to the satisfaction property of the processors from $\tilde{s}_P \sqsubseteq \vec{cs}_P$ and $f_P \hookrightarrow_P \vec{cs}_P$ we may conclude $\tilde{r}_P = \vec{cr}_P$. By straightforward equational reasoning we obtain

$$\tilde{t} = \vec{ct} \ \wedge \tilde{v} = \vec{cv} \ \wedge \tilde{s}_P = \vec{cs}_P$$

This shows that every set of streams that fulfills both the equations and the constraints is a least fixpoint as long as we assume the response satisfaction property for all processors.

$\square$

This theorem is of some importance for proofs about the execution of Alpha programs using the functional model. It indicates that in proofs we do not have to rely on least fixpoint properties, but we may just work with fixpoint properties. Hence we can work with purely equational reasoning.

The theorem does not say that there is only one fixpoint, it says that there is only one fixpoint for each feasible scheduler function ms. Due to the underspecification in the scheduler, there are many different scheduler functions ms that may have different fixpoints.

7 Locking

In order to synchronize programs properly, we need more sophisticated concepts than the ones treated so far. For the Alpha architecture load locked instructions and store conditional instructions are available. So far we have not said anything about locking. In this section, we briefly show how loads with locks and conditional stores can be treated in our model.

7.1 Extension of the Model to Locking

A locked load action is represented by `P:K(x,a)`. Roughly speaking, a locked load is a read action that in addition sets a lock flag. A following conditional store succeeds, only if the lock flag is set. A lock flag is also cleared, if a write request is executed for the location for which the lock flag was set. A successful conditional store request is a write request.

Action name	Syntax	Memory Request	Response
Load locked	`P:K(x,a)`	`P:K?x`	`P:R(x,a)`
Store conditional successfully	`P:S(x,a)`	`P:S(x,a)`	`P:S(x,L)`
Store conditional failed	`P:S(x,a)`	`P:S(x,a)`	`P:S(x,0)`
Store conditional failed	`P:S(x,0)`	`P:S(x,0)`	`P:S(x,0)`

Figure 5: Table actions for locking and their split into requests and responses

Conditional stores can be successful or they may fail. A successful conditional store action is represented by `P:S(x,a)`. A failed conditional store action is represented by `P:S(x,0)`. A memory request issued by the processor *P* for executing a locked read on location *x* is represented by `P:K?x`. A memory request issued by the processor *P* for executing a successful conditional store is represented by `P:S(x,a)`. A memory request issued by the processor *P* for executing a failed conditional store is represented by `P:S(x,0)`.

It looks strange that a processor can issue a request for a failing store action, but this is used to model the following situation. Let us assume that the instruction stream contains a conditional store instruction. When the processor executes this instruction, the failure of the corresponding conditional store may depend on the situation inside the processor. Then the processor issues an instruction store request that is condemned to failure.

Memory requests issued by the processor *P* for executing locked read or conditional store instructions both require responses. A response to a memory request issued by the processor *P* for executing a locked read on location *x* is represented by `P:R(x,a)`. A response to a memory request issued by the processor *P* for executing a conditional store is represented by `P:S(x,L)`, if it was successful. A memory request issued by the processor *P* for executing a conditional store is represented by the response message `P:S(x,0)`, if it failed.

The table given in Figure 5 shows these additional actions and their decomposition into memory requests and responses to them.

Let us assume from now on that the sets of requests *REQ* and responses *RFR* also contain these additional requests and responses. A load locked request behaves like a read request, but in addition may interfere with the execution of conditional store requests. The conditional store request is like a write but, since it may fail its failure or success needs to be indicated to the processor. With respect to the memory scheduling there is no difference between locked load and read or between conditional store and write. There is a difference, however, with respect to the read/write consistency.

For convenience, when writing the specification, we use an additional artificial request message with syntax `P:K!x`. It stands for locked loads in the access stream that have been executed already. The execution of a locked load request `P:K?x` in the access stream leads to a read response `P:R(x,a)` with the most recent written value a. In the access stream, the marker `P:K!x` indicates the position in the access stream, in which the lock has been executed. All other locks of processor P get cleared.

In our mathematical model, we express read/write consistency for locked loads by the following formula: for all streams $u, w \in REQ^*$, $s \in REQ^\omega$ we assume:

$$\begin{gathered}(u \frown w)|_{RISK} = \langle\rangle \wedge w|_{W(x)} = \langle\rangle \wedge w|_{NOL(P)} = \tilde{w} \wedge u|_{NOL(P)} = \tilde{u} \\ \Rightarrow \\ mm(u \frown \texttt{Q:W(x,a)} \frown w \frown \texttt{P:K?x} \frown s) = \\ \texttt{P:R(x,a)} \frown mm(\tilde{u} \frown \texttt{Q:W(x,a)} \frown \tilde{w} \frown \texttt{P:K!x} \frown s)\end{gathered}$$

where $W(x)$ is the set of all the write requests for location x:

$$W(x) = \{\texttt{P:W(x,a)} \in REQ : P \in PRC \wedge a \in DATA\}$$

and $NOL(P)$ is the set of all the requests different from lock markers for processor P:

$$NOL(P) = REQ \text{“} \{\texttt{P:K!x} \in REQ : x \in LOC\}$$

and $RISK$ is the set of all requests that require responses, namely the set of all the read requests, instruction fetch requests, store conditional requests, and load locked requests:

$$\begin{aligned} RISK = \; & \{\texttt{P:R?x} \in REQ : P \in PRC \wedge x \in LOC\} \cup \\ & \{\texttt{P:I?x} \in REQ : P \in PRC \wedge x \in LOC\} \cup \\ & \{\texttt{P:K?x} \in REQ : P \in PRC \wedge x \in LOC\} \cup \\ & \{\texttt{P:S(x,a)} \in REQ : P \in PRC \wedge x \in LOC \wedge a \in DATA \cup \{\texttt{O}\}\}\end{aligned}$$

This shows that a locked load request is processed like a read request, but after execution a marker is kept indicating the place in the access stream where the lock occurred.

A conditional store on location y is successful if it is marked as successful by the issuing processor P and if there is an executed lock `P:K!x` in the access stream and there are no writes to the location x between this lock and the conditional store in the access stream. The success of a conditional store is expressed by the following formula (with $a \in DATA$):

$$\begin{gathered}(u \frown w)|_{RISK} = \langle\rangle \wedge w|_{W(x)} = \langle\rangle \wedge w|_{K(P)} = \langle\rangle \wedge u|_{NOL(P)} = \tilde{u} \\ \Rightarrow \\ mm(u \frown \texttt{P:K!x} \frown w \frown \texttt{P:S(y,a)} \frown s) = \texttt{P:S(y,L)} \frown mm(\tilde{u} \frown w \frown \texttt{P:W(y,a)} \frown s)\end{gathered}$$

where $K(P)$ is the set of all the lock markers for the process P:

$$K(P) = \{\mathtt{P:K!x} \in REQ : x \in LOC\}$$

and $W(x)$ is the set of all the write requests to location x as specified above.

A conditional store to location x fails if there is a write to location x between the last locked load and the execution of the store request.

$$\begin{gathered}(u^\frown w)|_{RISK} = \langle\rangle \wedge w|_{W(x)} \neq \langle\rangle \wedge w|_{K(P)} = \langle\rangle \wedge u|_{NOL(P)} = \tilde{u} \\ \Rightarrow \\ mm(u^\frown \mathtt{P:K!x}^\frown w^\frown \mathtt{P:S(y,a)}^\frown s) = \mathtt{P:S(y,0)}^\frown mm(\tilde{u}^\frown w^\frown s)\end{gathered}$$

A conditional store also fails if there is no lock marker that was issued by the processor still valid.

$$\begin{gathered}u|_{RISK} = \langle\rangle \wedge u|_{K(P)} = \langle\rangle \\ \Rightarrow \\ mm(u^\frown \mathtt{P:S(x,a)}^\frown s) = \mathtt{P:S(y,0)}^\frown mm(u^\frown s)\end{gathered}$$

The failure of a conditional store request marked as failing is expressed by the following formula:

$$\begin{gathered}u|_{RISK} = \langle\rangle \wedge u|_{NOL(P)} = \tilde{u} \\ \Rightarrow \\ mm(u^\frown \mathtt{P:S(x,0)}^\frown s) = \mathtt{P:S(x,0)}^\frown mm(\tilde{u}^\frown s)\end{gathered}$$

Whether a conditional store request issued by a processor is marked as condemned to failure or as successful if not interfered with by another processor is left unspecified. In [AARM 92], page (I) 4-9, this is called *unpredictable*. In the functional model of a processor, it is part of the specification of the behavior of the processor to describe under which conditions a conditional store request is condemned to failure or as successful. In the mathematical model these conditions can also be left unspecified, and thus unpredictable. However, more sophisticated fairness conditions can also be formulated.

7.2 Analysis of a Simple Locking Discipline

In this section we show how a simple mutual exclusion scheme does work with the described memory scheduling discipline.

We consider the following mutual exclusion scheme as given in [AARM 92], page (I) 5-6. Every processor executes the following program, before it gets into its critical phase:

```
try_again :  LDL            R1           x
             <  modify y  >
             STC            R1           x
             BEQ            R1           no_store
             ...
no_store :   BR             try_again
```

Our specification allows us to conclude that the store conditional request can be successful only if it is issued as successful by the processor, and if in the access stream there is no write to location x between the load locked and the store conditional. The specification of the memory function allows us to prove that according to these facts this simple protocol works.

8 Conclusion

The purpose of this report is to give a mathematical model of the Alpha shared memory system. It provides a consistent description of the Alpha shared system and of the locking rules as given in [AARM 92]. It is, of course, still a simplification, since it does not consider interrupts or exceptions.

It is nevertheless helpful for analyzing some of the properties of the Alpha architecture and the programs that run on it. It moreover clarifies some issues not treated explicitly in [AARM 92] such as the treatment of causal loops.

This report also shows the usefulness and flexibility of functional system models. It demonstrates that functional system specifications can be used to describe system architectures and properties thereof of considerable complexity.

Acknowledgement: This work was mainly carried out during my stay at DIGITAL Systems Research Center in April 1993. The excellent working environment and stimulating discussions with the colleagues at SRC, in particular Jim Horning, Yuan Yu, and Jim Saxe are gratefully acknowledged. I thank Jim Horning for his careful reading of a version of the manuscript and his most useful comments.

A Appendix: A More Liberal Scheduling Concept

In the model given so far we have a coherent memory in the sense that all processors make observations that are consistent with the assumption of a sequential global memory. Therefore, if two locations x and y that are initially 1 are updated to 2, then it is impossible that one process receives the response stream

```
R(y,2) R(x,1) R(x,2)
```

as reaction to its request stream

```
R?y MB R?x MB R?x
```

and the other process obtains

```
R(x,2) R(y,1) R(y,2)
```

as reaction to its request stream

```
R?x MB R?y MB R?y
```

This is excluded by the assumption of consistent updates of a global memory leading to serializability for the access stream.

We model a more liberal memory scheduling for the Alpha architecture by a data flow model as given in Figure 6. In this data flow model the memory is distributed. Every processor has its own copy of the memory. The processors are connected to their memory by channels. The channels are denoted by cs_P, cr_P, and ct_P where P denotes the corresponding processor.

Each processor sends a stream of memory requests to the scheduler and receives a stream of read response messages from its memory. In a computation of the Alpha architecture, the behavior of each processor $P \in PRC$ and the function mm are modeled as before. In a computation, a stream is associated with each of the channels cs_P, cr_P , and ct_P. These streams will be denoted by $\vec{cs}_P$, $\vec{cr}_P$, and $\vec{ct}_P$ respectively.

We formalize the requirements for the behavior of the scheduler component by a function

$$ms : (PRC \to REQ^{\omega}) \to (PRC \to REQ^{\omega})$$

which models the proper memory access scheduling and determines the access streams and thus the access orderings and the memory service

$$mm : REQ^{\omega} \to RFR^{\omega}$$

which models read/write consistency. The function md then is simply obtained by composing ms and mm.

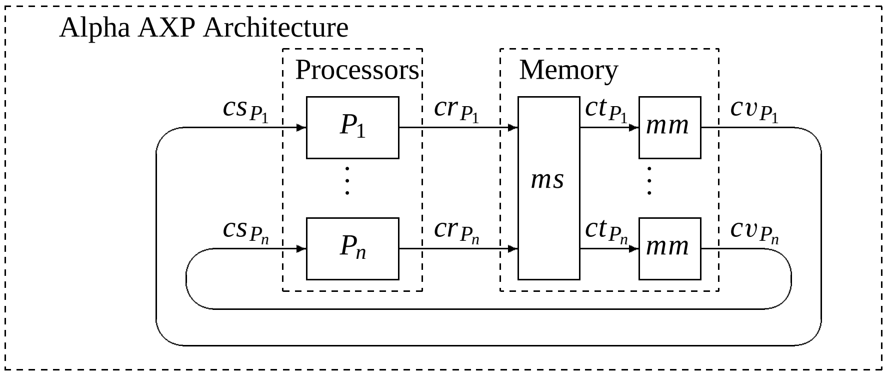

Figure 6: Data flow graph of the Alpha architecture

We give the specification for the function ms by the relation between the streams $\vec{cr}_P$ and $\vec{ct}_P$. The function ms merges and reschedules its input streams.

Two streams s_1 and s_2 are called *consistent* and we write

$$s_1 \bowtie s_2$$

if

$$s_1 \sqsubseteq s_2 \vee s_2 \sqsubseteq s_1$$

For the function ms we require the following safety conditions:

- for all locations x the access streams are consistent for all processors P and Q:

  $$ms(r)_P|_{W(x)} \bowtie ms(r)_Q|_{W(x)}$$

  where

  $$W(x) = \{\texttt{P:W(x,a)} \in REQ \, : \, P \in PRC \wedge a \in DATA\}$$

- for each processor P, the restriction of the access order with respect to the issue order is obeyed in its access stream; for all $r \in (PRC \to REQ^{\omega})$ we have

  $$\exists t \in REQ^{\omega} : ms(r)_P \sqsubseteq t \wedge$$
  $$\forall c_1, c_2 \in R(P) : c_1 < c_2 \Rightarrow [c_1 < c_2] \, in \, [r_P, t]$$

  where

  $$R(P) = \{c \in REQ \, : \, processor(c) = P\}$$

- for each processor P in its access stream $ms(r)_P$ only read requests for P occur.

For the scheduler we require the following liveness condition: for any pair of memory requests c_1, c_2 of processor P for which we have $c_1 < c_2$ in the table given in section 3, their relative order in the issue stream $\vec{cr}_P$ and in access stream $ms(\vec{cr})$ coincide. Expressed mathematically the liveness condition is

$$c_1 < c_2 \Rightarrow [c_1 < c_2]\, in\, [\vec{cr}_P, ms(\vec{cr})_P\ ]$$

This is a liveness condition that restricts the choice of the scheduling function ms in addition to the safety properties required for ms making sure that all requests are eventually scheduled.

B Appendix: Mathematical Basis

A *stream* represents a communication history for a channel. A stream of messages over a given message set M is a finite or infinite sequence of messages. We define

$$M^\omega =_{df} M^* \cup M^\infty$$

We briefly repeat the concepts from the theory of streams that are used in the specifications. More comprehensive explanations can be found in [Broy 90].

- By $x\frown y$ we denote the result of concatenating two streams x and y. We assume that $x\frown y = x$, if x is infinite.

- By $\langle\rangle$ we denote the empty stream.

- By $ft(x)$ we denote the first element in a stream; if the stream x is empty, it is undefined.

- By $rt(x)$ we denote the stream obtained from x by dropping its first element; if the stream x is empty, the resulting stream is empty.

- If a stream x is a *prefix* of a stream y, we write $x \sqsubseteq y$. The relation $\sqsubseteq$ is called *prefix order*. It is formally specified by

$$x \sqsubseteq y \equiv_{df} \exists z \in M^\omega : x\frown z = y$$

The behavior of deterministic interactive systems with n input channels and m output channels is modeled by (n, m)*-ary stream processing functions*

$$f : (M^\omega)^n \to (M^\omega)^m$$

We use some notions from domain and fixpoint theory that are briefly listed:

- A stream processing function is called *prefix monotonic*, if for all tuples of streams $x, y \in (M^\omega)^n$ we have

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

- By $\sqcup S$ we denote a least upper bound of a set S, if it exists.
- A set S is called *directed,* if for any pair of elements x and y in S there exists an upper bound in S.
- A stream processing function f is called *prefix continuous* if f is prefix monotonic and for every directed set $S \subseteq M^\omega$ we have:

$$f(\sqcup S) = \sqcup\{f(x) : x \in S\}$$

- A partially ordered set is called *complete* if every directed set has a least upper bound.

The set of streams and the set of tuples of streams are complete. Note that every directed set of streams has a least upper bound.

In specifications we use the filter function in infix notation. Let S be an arbitrary subset of the set M and $x \in M^\omega$ be a stream over M. We specify:

$$(m\frown x)|_S = x|_{M_0} \Leftarrow \neg(m \in S)$$

$$(m\frown x)|_S = m\frown(x|_{M_0}) \Leftarrow m \in S$$

$$\langle\rangle|_{M_0} = \langle\rangle$$

Furthermore we use the function

$$\# : M^\omega \to I\!N \cup \{\infty\}$$

that yields the length of a stream. It is specified by (for $m \in M$):

$$\#\langle\rangle = 0$$

$$\#(m^\frown x) = 1 + \#x$$

We model the behavior of interactive components by sets of continuous (and therefore by definition also monotonic) stream processing functions. Monotonicity models causality between input and output. Continuity models the fact that for every behavior the systems reaction to infinite input can be predicted from the reactions of the component to all finite prefixes of this input[1]. Monotonicity reflects the fact that in an interactive system previous output cannot be changed when further input arrives. The empty stream represents the information "further communication unspecified".

A specification describes a set of stream processing functions that represent the behaviors of the specified systems. If this set is empty, the specification is called *inconsistent*. If the set contains exactly one element, then the specification is called *determined*. If this set has more then one element, then the specification is called *underdetermined* and we also speak of *underspecification*. An underdetermined specification can also be used to describe hardware or software units that are *nondeterministic*. An executable system description is called *nondeterministic*, if it is underdetermined. Then the underspecification in the description of the behaviors of a nondeterministic system allows nondeterministic choices carried out during the execution of the system. In the functional modeling of interactive systems there is no difference in principle between underspecification and the operational notion of nondeterminism. In particular, it does not make any difference in such a framework whether these nondeterministic choices are taken before the execution starts or step by step during the execution.

[1]This does not exclude the specification of more elaborate liveness properties including fairness. Note, fairness is, in general, a property that has to do with "fair" choices between an infinite number of behaviors.

References

[AARM 92]	R.L. Sites (ed.): Alpha Architecture Reference Manual. DIGITAL Press 1992
[Attiya, Friedmann 94]	H. Attiya, R. Friedmann: Programming Alpha Based Multiprocessors the Easy Way. Unpublished Manuscript.
[Broy 90]	M. Broy: Functional Specification of Time Sensitive Communicating Systems. REX Workshop. In: J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): Stepwise Refinement of Distributed Systems. Lecture Notes in Computer Science 430, Springer 1990, 153-179
[Broy 93]	M. Broy: Functional Specification of Time-Sensitive Communicating Systems. In: ACM Transactions on Software Engineering and Methodology 2:1, 1993, 1-46
[Shasha, Smir 88]	D. Shasha, M. Smir: Efficient and Correct Execution of Parallel Programs that Share Memory. In: ACM Transactions on Programming Languages and Systems. 10:2, 1988, 282–312.