Bus Arbitration Modelling and Design in DAME: an Expert Microprocessor-Based-Systems Designer

by

Marco Antonio Escalante
B.Sc., Universidad Iberoamericana, 1987

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

We accept this thesis as conforming
to the required standard

[redacted]

Dr. N.J. Dimopoulos, Supervisor
(Department of Electrical and Computer Engineering)

[redacted]

Dr. K.F. Li, Departmental Member
(Department of Electrical andComputer Engineering)

[redacted]

Dr. Z. Dong, Outside Member
(Department of Mechanical Engineering)

[redacted]

Dr. M. van Emden, External Examiner
(Department of Computer Science)

University of Victoria

Supervisor: Dr. Nikitas J. Dimopoulos

ABSTRACT

The automation of computer hardware design has received considerable attention in recent years. Expert systems, that incorporate explicit domain knowledge into problem-solving programs, have been successfully applied in design problems. DAME (**D**esign **A**utomation of **M**icroprocessor-based systems using an **E**xpert system approach) is a system capable of designing microprocessor-based systems from original specifications.

In this thesis, the bus arbitration interface design module, a module of the DAME designer, is presented to illustrate the design process in DAME.

The design philosophy followed by DAME requires that components include reference to abstractions of their interfacing capabilities. The design rules utilize this information to produce an abstract design which is subsequently instantiated. In particular, components are functionally modelled by their capabilities. These capabilities follow protocols to guarantee the proper communication. Actions are the elementary operations that describe the protocol sequence. Finally signals are the agents that convey the actions.

The interface design strategy in DAME divides the interface problem into subproblems that deal with the component capabilities, and applies generic rules that instantiate an abstract interface design with the particular information from the participating components. The abstract interface design is carried out by merging action graphs that represent the involved protocols. The methodology outlined above is illustrated through an example involving the VMEbus.

Examiners:

[redacted]

Dr. Nikitas J. Dimopoulos, Supervisor
(Department of Electrical and Computer Engineering)

[redacted]

Dr. Kin F. Li, Departmental Member
(Department of Electrical and Computer Engineering)

[redacted]

Dr. Zuomin Dong, Outside Member
(Department of Mechanical Engineering)

[redacted]

Dr. Maarten van Emden, External Examiner
(Department of Computer Science)

Table of Contents

List of Figures

Acknowledgements

I would like to thank to all the people that contributed, either academically or otherwise, to the successful completion of this thesis, with special consideration to:

Dr. Nikitas Dimopoulos, for his infinite patience and understanding, as well as for his central directions whose imprint can be found throughout these pages,

Dr. Kin Li, for his kind assistance and enthusiasm, that significantly enriched the contents of this work,

the other members of the DAME group, Dongni Li, and Ben Huber, for their quotidian charm, and for the endless discussions that make this thesis better,

my other fellow grad students, for those moments that we altogether shared at school and in recreation, especially Al Keddy for his valuable insight on UNIX,

my brother Carlos, for his concern and thoughtful reflections,

and finally, my parents, whose immense love initiated this wonderful experience.

A mis padres,
A mi abuelita Rosa.

1. Introduction.

This thesis work forms part of a broader project, DAME (**D**esign **A**utomation of **M**icroprocessor-based systems using an **E**xpert system approach). In this introductory part, the motivation of this thesis is proffered. As well, the contents of the various chapters that constitute this work are portrayed. The trademark acknowledgment of the software used to develop a prototype that demonstrates the concepts in this thesis is given at the end of this chapter.

1.1 Thesis Rationale.

Semiconductor technology is supporting high performance, low-cost, and highly reliable microprocessor-based systems. The problem that such a rapid advance poses to designers is the specialized knowledge required to build up those sophisticated systems. Computer-Aided Design (CAD) tools become valuable aids throughout the design process.

In many system design problems, the lack of a comprehensive theory of system integration and design choices has led to a more or less empirical set of rules [26]. Several expert systems were developed that are capable of synthesis, such as R1 [60, 62] and MICON [7, 8]. DAME is a system that attempts to automate the hardware design in the context of microprocessor systems. It is DAME's tenet that the design can be carried out by utilizing very powerful rules dealing with abstract objects, instead of just intuitive rules that capture the interrelations of the objects in the design space. It is this viewpoint that differentiates DAME from other systems.

DAME practices a top-down approach that breaks the design problem down into subproblems easier to be worked out. When two components in a microprocessor system need to be interfaced, several groups of signals come into action that embody their functionality, such as the data transfer lines, the interrupt lines, or the bus arbitration lines. This thesis illustrates the design procedure in DAME by exploring the bus

arbitration capability.

1.2 Organization of the Thesis.

This thesis is organized in seven chapters, including this introduction, and several appendices.

Chapter two discusses the concepts behind design automation, and specifically hardware synthesis. Some systems that have been developed in the field of hardware synthesis are briefly described. An overview of expert systems closes the chapter.

Chapter three deals with the bus arbitration capability. Bus arbitration is inscribed in the framework of exclusive access systems. Several techniques of allocation, of which arbitration is one example, are presented. Two frequently encountered bus arbitration protocols are delineated.

Chapter four introduces the notion of actions as a means of describing protocols. An action graph is used to encapsulate the sequence of operations in a protocol. A notation that expresses the mapping of actions into physical signals is presented. Finally action graphs for the two protocols introduced in Chapter three are developed.

Chapter five describes the bus arbitration designer module. Representation plays an important role in this module. The design problem of interfacing two components is translated into a graph merging problem. Once a solution is incorporated into the designer module, it uses this abstract design to instantiate a particular interface with the information in the component library.

Chapter six presents a design example of a multi-master system. The output of the designer module in the functional phase produces a technology independent description of the interface. A hardware implementation was carried out manually from this description to demonstrate that it is both complete and correct.

Chapter seven enumerates some conclusions, and makes recommendations regarding further work.

In the appendices, diverse material is included to complement the main text, especially Chapter six.

1.3 Trademarks.

Software packages have been employed in developing and testing the designer's module. The following terms are trademarks of their respective owners:

Knowledge Craft is a trademark of Carnegie Group Inc.

Silos II is a trademark of Simucad Inc.

2. The Expert System Approach in Hardware Design Automation.

2.1 Introduction.

This work forms part of DAME (**D**esign **A**utomation of **M**icroprocessor-based systems using an **E**xpert system approach), a system that will be capable of designing microprocessor-based systems. In this chapter, the ideas behind hardware design are discussed, and some examples of systems that, like DAME, attack the problem of automating the hardware design are presented. All those systems, including DAME, use an expert system as their designer module. A brief introduction to expert systems concludes this chapter.

2.2 Design Automation.

It is synthesis, the art of building new artifacts, either a chemical process or a computer system, the unique contribution of engineering that sustains our industrial civilization [84]. The automatic synthesis of digital hardware from high-level specifications has been a research topic early on, but it is until recently that some progress has been made and systems addressing aspects of the design process have been developed.

There is a strong motivation for the automation of the computer design process. Semiconductor technology is advancing by leaps and bounds, producing high-end chips that allow the designer to build highly sophisticated machines with off-the-shelf components. However there is a price to pay. High-performance systems require expert designers that can keep conversant of a rapidly evolving technology. On the other hand, companies are involved in an intense competition for marketing their products, whose life cycle is measured in months. Thus, the hardware designer is being squeezed by three forces: a reduced design time, the ceaseless demand for increased performance at a lower

price, and a constantly evolving technology [8]. It is difficult to foresee future hardware designs without the intervention of Computer-Aided Design (CAD) tools.

2.3 Hardware Design.

Hardware synthesis is the process of mapping an input specification for a hardware design into a hardware implementation [71]. The input specification normally contains behavior information that describes the expected functionality. The implementation contains the structural information, and more detailed behavior of the structure. Structural information may take several forms, such as logic gates, registers, net lists, and standard cells.

Although it is difficult to formalize the hardware process, it can be stated as the application of successive transformations from general specifications that produce a final implementation of the system, unless it is discovered that the original specifications are not feasible. A divide-and-conquer strategy, that breaks the problem into a number of interconnected sub-problems, has been widely used in the design field.

2.3.1 Hierarchy in the Design Process.

Hardware synthesis can be viewed as a search in the design space, comprising all the possible interconnections of components, for the optimal design. Because of the combinatorial explosion of design possibilities, an exhaustive search is not feasible.

Hierarchical design methodologies have been used that break the design down into several steps. These methodologies have the advantage of reducing the amount of information needed at a particular level, making the design process less error prone and the designer more productive. The phases in the hardware design process, that correspond to levels of abstraction of hardware systems, are [92]:

• Architectural level. At this level, the main concern is the overall structure of the system. Designers are interested in components such as processors, memories, I/O devices, and in their interconnections. In addition to cost and functional correctness, designers start paying attention to system properties as fault tolerance and reliability. The focus of attention is on the global flow of information among the various components. An optimal design is usually the configuration that best satisfies the specifications at a minimal cost. The PMS notation [2], for example, has been used as a system description

tool at this level.

• Instruction-set level. The main concern at this level is the functionality of the system rather than its physical structure. Bell and Newell's ISP notation can be used to describe the system at this level [5].

• Register-transfer level (RTL). At this level, sequences of operations that define the functionality of the modules that comprise the system are usually specified as transfers of information between the registers, defined in the functional design. Simulation at this stage helps to discover logical errors in the design.

• Logic design level. The goal is to map the micro-operations and control structures defined in the register-transfer level into physical hardware elements.

• Physical design level. The main problem at this level is the layout procedure, that consists of placing the final hardware components and routing their interconnections.

• Circuit level. Nowadays, with digital logic primitives being the smallest component of digital systems, circuit level design is less critical than in early computer designs, unless a special-purpose IC needs to be designed or a new circuit advances the state of the art.

• Documentation. Although considered as a time-consuming task with little creative challenge, documentation is very important in order to ensure an effective maintenance of the system.

A system level simulation validates the final design. Circuit-level simulators, such as SPICE, produce a more realistic verification of the functionality of the system. Formal and timing verification of digital systems is still under development [92]. Because logic design has become less important with the introduction of complex functions into a single IC, modern hardware design is mostly carried out at the system level.

2.3.2 High-Level Hardware Design.

The most sophisticated digital systems require more attention at the highest phases of the design. This assumes that the lower design levels have somehow been standardized. The three developmental choices for such a high-level digital design are [41]:

• Standard single-board computers. The main concern of the system design is the production of the software for the specific problem. This approach is called the software method.

• Microprocessor-based systems. The demand for compactness, reliability, capacity, and cost-efficiency led to the production of customized circuits. The main drawback is the high development cost. One solution is the production of the microprocessor, a general-purpose digital circuit that can be utilized for a variety of tasks.

• VLSI circuits. Another approach facilitated a complete IC customized design procedure. This approach organizes the whole design process into a number of tasks that the user can master in a short period of time. This is called the hardware or VLSI approach.

Our work will explore the second approach. A microprocessor-based design can be outlined as follows [41]: from the variety of microprocessors, memories and peripherals, the designer chooses the appropriate components, combines them on a board, and writes the program that will solve her specific problem.

2.3.3 Design Representation.

The main motivation of design languages is to have the tools for describing a system's behavior to allow a more precise designer-user communication, to use it as an input language to a simulator, and/or to use it as an input to a hardware compiler. For instance, Silos II™ Behavioral Language (SBL) has been used in DAME to simulate and verify the design.

Designers deliver a description of the system to the production stage, as a list of hardware primitives and their interconnection. Connectivity description tools are used to map the list of hardware blocks with a list of interconnections between these blocks, called net list. The net list remains a key element in capturing the design and managing the logical network data.

Different representations can be used for different phases of the design, each emphasizing the important aspects of a particular phase. One approach inclines the design representation and procedures towards certain technology. Another approach produces a technology-independent description of the hardware functionality and uses

transformation rules that correctly implement the design according to a selected technology. The DAME designer subsystem, the subject of this work, generates an intermediate abstract description of the hardware so that the final implementation can be modified by including new modules in the implementor subsystem as new technologies become available.

2.3.4 Automation of the Hardware Design.

The traditional functions of Design Automation (DA) have been to replace the designer in the routine tasks that require no design decisions, to assist the designer in making design decisions by evaluating the merits of design alternatives, and to assist designers in verifying the correctness of their design.

The evolution of design automation tools has been tightly coupled to the evolution of the design process, which, in turn, has been influenced by the evolving technology [92]. The first applications of computers as hardware designer aids were developed in the fifties. They focused on the clerical tasks of the design, such as the wire-list preparation program developed at IBM [17]. When integrated circuits became available, programs and algorithms appeared for the design of printed circuit boards (PCB). Several tools for IC layout were developed in the seventies such as artwork editors, design rule checkers, and automated placement and routing. Simulators also helped in the verification of logic designs.

Attempts were made to automate the logic design phase by using register-transfer languages that, through a hardware compiler, would produce a complete logic design. More recently the primary concern is in the integration of CAD tools into a complete system. Data representation has become an important issue in integrated system design, while user interface considerations have led to an increased use of interactive graphics.

There are two schools of thought about the general synthesis method [71]: the first one generates a correct solution and transforms it in order to optimize certain objectives; the second school performs the optimizations as the design is synthesized. DAME belongs to the first school. The main concern is the correctness of the design, the rationale being that it is more important to generate a correct design rather than to spend effort optimizing designs that may not function correctly.

2.4 Hardware Design Systems.

Some examples of systems that attempt to solve the design problem using off-the-shelf components, in particular microprocessors, are presented in the following. Other systems that automate the design from specifications given at the register-transfer level are not covered here, and can be found in [71].

2.4.1 R1 (XCON).

This system, developed by Digital Equipment Corp., was one of the first successful applications of an AI technology (expert systems) in industry [60, 61, 62]. XCON is able to configure a suitable VAX computer arrangement given the customer's particular requirements. Despite there is no hardware design in this system, it shares some similarities with other high-level hardware design systems, such as the hierarchical division of the design, and the use of complex blocks as building blocks of the system to be designed.

XCON incorporates the necessary knowledge to configure a system in the form of a production system, as described in the next section. XCON breaks the configuration into six task [62]:

1. To determine if there are any inconsistencies in the configuration.

2. To put the correct components in CPU subassemblies.

3. To develop the arrangement for the UNIBUS cabinets.

4. To put names in the UNIBUS expansion cabinets.

5. To design the system and lay out its diagram.

6. To design the cabling.

No backtracking is required between these tasks.

2.4.2 MAPLE.

MAPLE (**M**icroprocessor **ApPL**ications **E**xpert) is an expert system prototype, developed at the University of Reading, England, which takes the role of an expert

consultant in the field of hardware design [85]. The first prototype was limited to design and documentation from post-analysis specification.

A MAPLE *consultation* consists of:

• an interview, in which the user is asked for hardware requirements (speed) and constraints (power, cost);

• a design stage, in which MAPLE uses its knowledge to design a system that meets the requirements and fulfill with the constraints; and

• a report stage, in which MAPLE produces the design documentation.

MAPLE uses the following databases to generate the design:

• COMPONENTS, that contains information about the blocks used to construct microprocessor-based systems (microprocessors, memories, peripherals, discrete logic).

• BOARDS, that contains pre-designed boards.

• REPORTS, that contains past designs.

This last database enables MAPLE to learn from its experience. The present version of MAPLE considers boards as the primitive building blocks of the design. Microprocessor application design using IC-level components is recognized to be a more difficult problem [85] and it is left for further investigation.

2.4.3 MICON.

The MICON system (**MI**crocomputer **CON**figurer) is an integrated collection of programs which automatically synthesizes single-board computers from high-level specifications, using standard microprocessor technology [6, 7, 8, 9]. MICON was developed at the Carnegie-Mellon University.

The block diagram of MICON is shown in figure 2.1. M1 embodies the system designer expert system. M1 accepts as input a set of high-level specifications that describes the required functionality of the computer system to be designed. Examples of information contained in the specification include the family of microprocessor, the amount and type of memory (RAM static and dynamic, and ROM), and the number and type of input/output devices. An additional set of parameters is used to evaluate a cost

function that characterizes the design constraints (such as cost, board size, etc.) During the design process, M1 interacts with the user to ask for directions whenever different alternatives appear.

ASSURE (Automated Synthesis for Reliability) is a sub-module of M1 that analyzes and modifies designs for improved reliability. Although ASSURE is closely interacting with M1 through the design process, it has an algorithmic structure.

CGEN (Code Generator) is the knowledge acquisition tool for the M1 module. Inputs to CGEN consist of schematic drawings and simple boolean equations, allowing a hardware designer, not familiar with the M1 internal representation, to increase the number of components. Therefore the rule base can be updated with new components and design techniques as they are being developed.

A group of physical design tools implements the final design. Standard software does the placement and routing, and the board fabrication is done at a local PCB manufacturer. The database centralizes the component information used in the design.

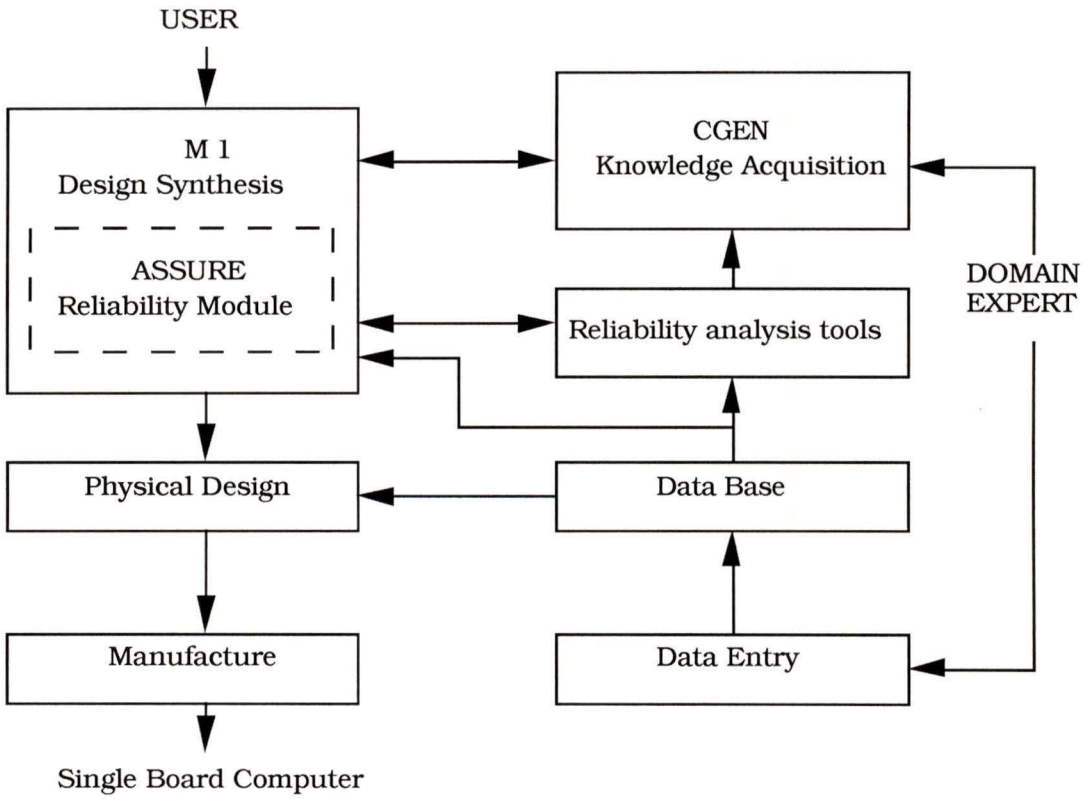


Figure 2.1 The MICON system.

The construction of the design in M1 is based upon the integration of the following

subsystems:

- processor
- memory
- peripheral devices

Design subtasks that deal with each subsystem can be considered as incremental steps towards the completion of the design. The basic design cycle in M1 consists of the following stages:

• Specification. The user is requested to enter the features for a desired subsystem. The user may not provide some of the features asked. In that case, M1 chooses the subsystem without constraints.

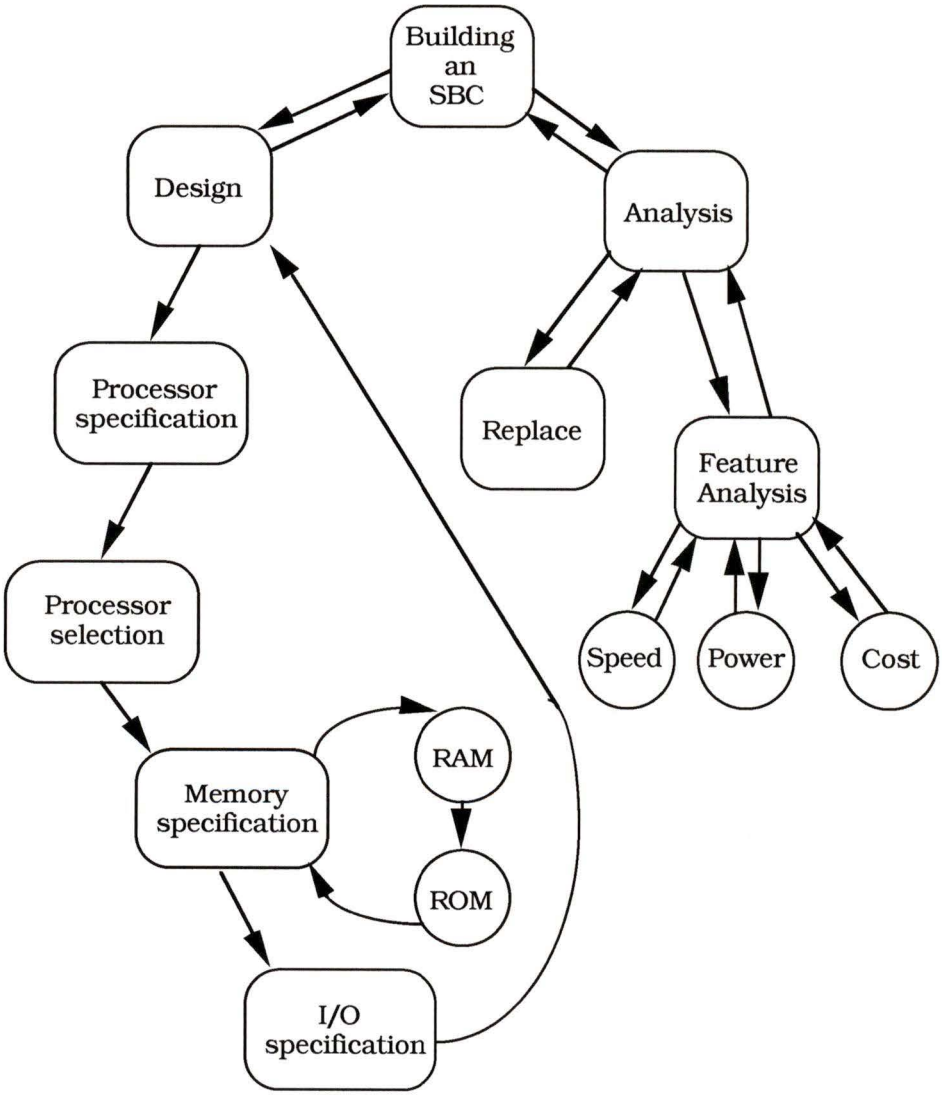


Figure 2.2 Subtasks in MICON.

• Selection. The closest component in the data base that matches the specification is chosen. In case that no selection is found, the user is asked to modify her specifications.

• Instantiation. Once the parts are selected, a copy of the templates in the data base is produced. Finally the required interconnections are made.

The hierarchy of subtasks in MICON is shown in figure 2.2.

In the instantiation phase, the inclusion of any support chips, such as memory controllers, occurs automatically without the intervention of the designer. *Templates*[1] are activated to fulfill this goal. A template is a complex structure that accomplish one of the basic subsystems (processor, memory, I/O). It might consist of several chips. In figure 2.3 the template for the Z80 processor is shown. It consists of several units that complement the functionality of the Z80 CPU in order to convert the CPU into an integral processor. In templates, extra glue logic that implements a standard interface bus is attached to the individual components. MICON eliminates the interface problem by defining as its primitive elements, objects with the same interconnection structure.

2.4.4 DAME.

DAME (**D**esign **A**utomation of **M**icroprocessor-based systems using an **E**xpert system approach) is an expert system prototype currently being developed at the University of Victoria [24, 26, 27, 42, 43]. DAME will be capable of interpreting the input design specifications to produce a final customized microprocessor-based system.

The designer module goes through a formal design process consisting of the following phases:

• Design specification phase. At this phase, the system responsibilities, design constraints, and system environment are established. Specifications include type of application, communication and computational requirements, as well as economical criteria. This phase involves considerable consultation between the designer and the user.

• Configuration phase. The gross system architecture is established during this phase. The modules that form the system at this level are processor, memory, I/O devices,

---

[1] MICON terminology is used in this case. Templates have another connotation in semantic networks, as discussed in a subsequent section.

and bus.

• Behavior description phase. This phase defines the capabilities of the subsystems produced by the configuration phase. For example, it is during this phase that the system bus is selected.

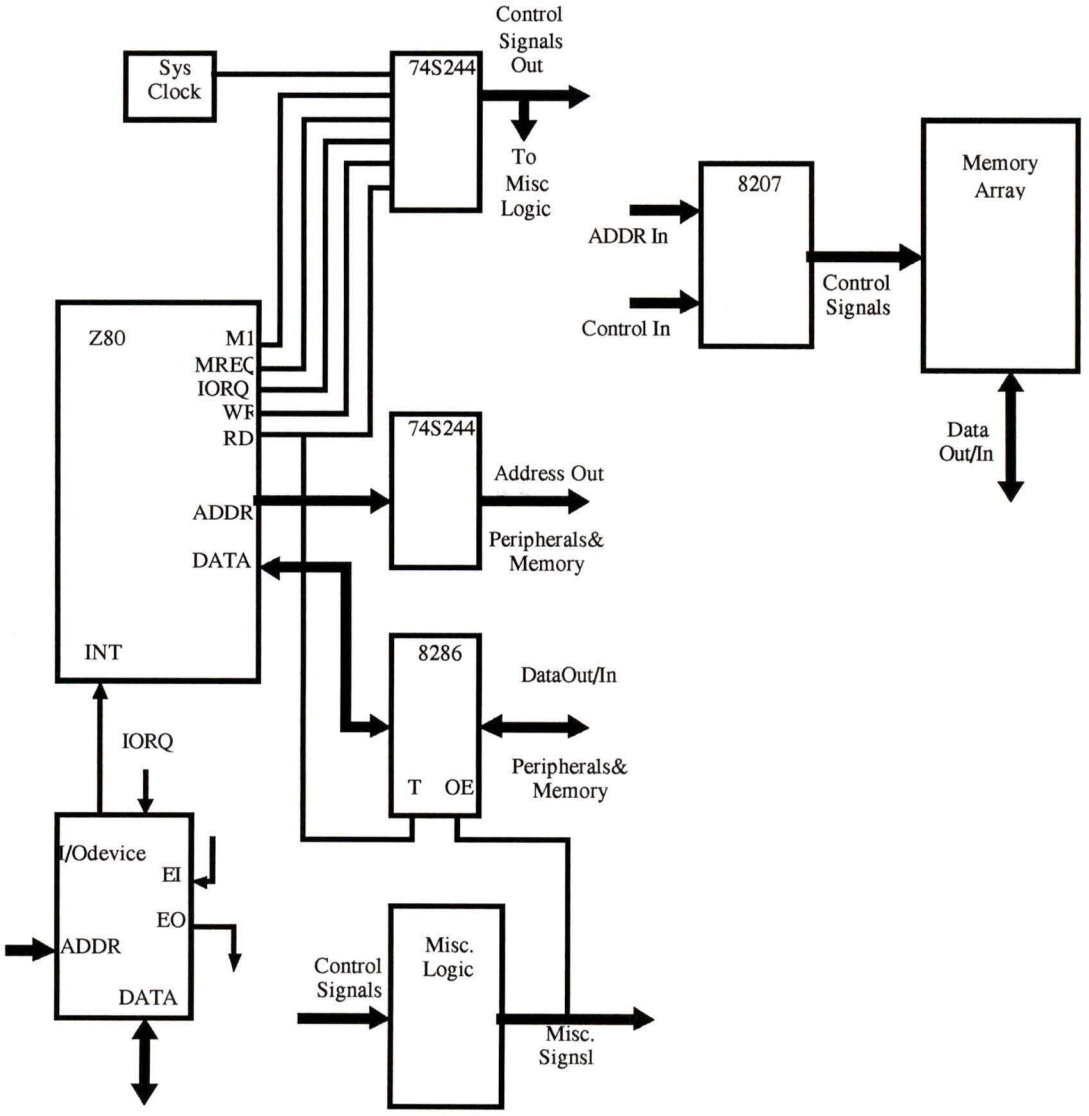


Figure 2.3. Template for the Z80 CPU in the MICON system.

• Functional block design phase. During this phase, the capabilities of the subsystems are mapped into available components and the related interfaces are created to interconnect those components.

• Implementation and integration phase. The modules obtained in the previous

stage are connected together. Functions that do not directly correspond to a hardware component are synthesized at this point using random logic.

Each hierarchical level represents an abstraction of the given design problem. As the levels are transversed, the abstraction of the design is refined, until the complete design is produced.

DAME focuses on the interface problem (i.e., how to interconnect components that do not necessarily use the same protocol). MICON eliminates this problem by defining a universal interface for all the components so that they can be hooked up directly. On the other hand, in DAME, the components' interfacing capabilities and their respective protocols are incorporated in the data library, and knowledge about the required interfaces at the protocol level is included in the designer's rules, so that the design is carried out at the protocol level instead of at the component level.

The topic of this thesis is the bus arbitration interface design subsystem, part of the DAME designer system. This module produces the arbitration interfaces required in a multi-master situation during the functional block phase of the design.

2.4.5 Expert System Approach.

The success of the XCON project illustrated the potential of AI technology in solving design problems. The success of these systems depends not only on a good formulation of the synthesis problem but also on the encoding of large amounts of domain knowledge by rules that selectively use the database information [61]. Knowledge-based systems, or expert systems, facilitate the hardware design by making the representation of general design expertise possible. Expert systems are discussed in the following section.

2.5 Expert System Concepts.

The area of expert systems investigates methods and techniques for constructing computer systems with specialized problem-solving expertise. Expertise consists of knowledge about a particular domain, understanding of domain problems, and skill at solving some of these problems [36]. There are some reasons that account for the emphasis on knowledge representation rather than on formal reasoning methods, mainly the non-existence of feasible algorithmic solutions for many important problems (i.e., design), the demonstration of the power of knowledge to solve certain class of problems

(mineral prospecting [28, 29], computer configuration [60], chemical structure identification [14], symbolic mathematics [58], medical diagnosis [83, 53], and electronic analysis [86]), and the recognition that knowledge is the key element in certain problems.

2.5.1 Knowledge in Expert Systems.

Knowledge is considered as the descriptions, relationships and procedures in certain domain [36]. Descriptions identify and differentiate objects and classes. Dependencies and associations of the elements in the knowledge base are the relationships, usually expressed as taxonomic descriptions. Procedures specify the operations to be performed to the knowledge base elements.

Knowledge engineering addresses the problem of building skilled computer systems, aiming first at extracting the expert's knowledge and then organizing it in an effective implementation. Skill is used in the sense of having the right knowledge and using it effectively [54].

Expert systems can be classified according to the type of problem it addresses:

Category	Problem addressed
Interpretation	Inferring situations descriptions from sensor data
Prediction	Inferring likely consequences of given situations
Diagnosis	Inferring system malfunctions from observations
Design	Configuring objects under constraints
Planning	Designing actions
Monitoring	Comparing observations to plan vulnerabilities
Debugging	Prescribing remedies for malfunctions
Repair	Executing a plan to administer a prescribed remedy
Instruction	Diagnosing, debugging, and repairing student behavior
Control	Interpreting, predicting, repairing, and monitoring system behaviors

Design systems develop configurations of objects that satisfy the constraints of the design problem. Design systems are different from other types of systems in that they are concerned with a hypothetical rather than a physical system[2] [84]. Non-monotonic reasoning, in which assumptions and further inferences based on them must be regarded as tentative, is important in design problems. The solution space usually is very large and it is impossible to predict the outcome of an early design choice.

Many expert systems are written as a collection of knowledge, represented as a large number of IF-THEN clauses, called production systems or rule-based systems. The expert system shell that embodies DAME uses a forward chaining production system for the encapsulation of knowledge, as discussed in the following.

2.5.2 Production Systems.

A production system is a program composed of conditional statements called production or rules together with a database of state information, sometimes organized as a collection of frames, and a procedure for invoking the production rules called the inference engine [31]. Knowledge base is the term used to describe the facts and the rules of the production system.

These rules separate actions, that specify what to do, from conditions elements, that specify when to do it. One example of a rule used in a diagnostic expert system is given in figure 2.4.

IF (the infection is primary-bacteremia) AND
(the site of the culture is one of the sterile sites) AND
(the suspected portal of entry of the organism is the gastrointestinal tract)
THEN
(there is suggestive evidence that the identity of the organism is bacteroids).

Figure 2.4 One example of a rule from a diagnostic production system.

Production systems are particularly useful when the solution of the problem can

---

[2] Other expert systems represent a real physical system (i.e., a diagnostic or classification system) and therefore, the inputs are inherently consistent, unless an error is made in observing them. However, the inputs of a design system are hypothetical, and not necessarily convey to a solution.

be expressed as IF-THEN relationships between inputs and results.

Because the inference engine can be separated from the knowledge base, it is possible to use expert system tools, also called shells, that contain the inference engine together with knowledge base management functions, saving time in the development of the expert system. Knowledge Craft™ (KC) has been chosen as the shell in which DAME is embedded. KC's powerful set of commands has been exploited to build DAME's semantic network, and KC's built-in forward chaining production system implements the designer.

2.5.3 Frames and Semantic Networks.

A frame provides a representation for an object in terms of a set of attribute names, called slots, and values for the attributes [88]. Frames consist of a non-empty list of attribute-value pairs[3]. A frame is similar to a record structure in algorithmic high-level programming languages. Schemata or templates are general frames that provide the attributes for a class of objects.

Knowledge can be represented using rules. Other means of knowledge representation are semantic networks. Semantic nets represent knowledge by denoting objects and describing relations among them. The value of a slot in a frame could be the name of another frame, so that frames can be used to build a semantic network.

The IS-A and INSTANCE relations are used to group objects in the semantic network into classes, super-classes, etc. The INSTANCE relation links an object with a class. For example, the object U1, describing a MC68000 microprocessor chip, is an element of the class of MC68000 microprocessors. This relation is described by writing U1 INSTANCE MC68000. The IS-A relation links a class with a super-class. For instance, MC68000 IS-A MICROPROCESSOR means that the class of MC68000 microprocessors is a (proper) sub-set of the super-class of microprocessors. In DAME, the INSTANCE relation is used to link an object to its template.

2.5.4 The Knowledge Craft Shell.

The Knowledge Craft shell is a workbench consisting of a number of workcenters, that comprise an integral system with knowledge base editors, debugging environments,

---

[3] The name slot, always present in the attribute list, contains the name of the frame.

file manager tools, and a forward chaining production system. Workcenters that are accessible from the shell include:

• CRL workcenter, that consists of several knowledge base editors, a Lisp evaluator, and the toolbox that permits to save the knowledge base.

• CRL-OPS workcenter, that contains the Knowledge Craft production system, with a forward chaining inference engine, and a knowledge base, structured as working memory (the facts) and production memory (the rules).

The use of the Knowledge Craft shell allowed the engineer to focus on the fundamental problems of DAME by making use of the rapid prototyping facility.

2.5.5 Deep Reasoning in Expert Systems.

Most of the current generation of expert systems use knowledge which does not represent a deep understanding of the domain, but it is instead a collection of condition-action rules, which correspond to the problem-solving heuristics of the expert in the domain. Recognizing the importance of a better insight in the solution of certain problems, such as diagnosis and synthesis, expert systems have been categorized as *deep* and *shallow* systems, according to the depth at which knowledge is represented and used in problem-solving [35, 65].

Consider the following approaches of building a system that attacks a synthesis problem:

1. A straightforward system will contain all the possible design solutions. The applicability of this approach is limited to very small systems.

2. The next best approach devises some problem-solving mechanisms which operates on a data base of partial patterns (i.e., patterns relating only a small set of data to design states, described by rules).

The above two approaches fall into the category of shallow systems.

There is less agreement about a precise characterization of a deep expert system. It is suggested that deep systems will solve problems of significantly greater complexity than shallow systems by adding to the heuristic relationships represented by the rules mechanisms to exploit a physical model of the problem [84]. The major intuition behind

the use of deep models in expert systems is the observation that often human experts recur to first principles when they are confronted with a difficult problem [16].

There exists another knowledge and problem-solving paradigm which:

- has the relevant model compiled into it in such a way that it can handle the design problems that a deep expert system is supposed to solve,
- solves the problem more efficiently than the deep system,
- but cannot solve other type of problems that the deep knowledge structure potentially could handle.

A medical diagnosis system that follows this scheme has been proposed in [16]. The DAME designer also conforms to this paradigm by incorporating an abstract model in its knowledge base. In this way, a set of powerful rules can produce the design by instantiating an abstract interface corresponding to the involved protocols with the particularities of the components that comprise the system, as it will be evident in the following chapters.

2.6 Summary.

Expert systems have been applied successfully to design applications because of their capacity of expressing knowledge in the form of rules and semantic networks. This work constitutes part of the DAME designer, an expert system that will perform the automation of the hardware design process in the particular area of microprocessor-based systems. This project is inscribed in the new generation of expert systems that incorporate deeper knowledge about the model of the system to attack the problem with generic rules. This paradigm is called *compiled* structure for generic problem-solving [16]. The term compiled is used in the sense of compiling knowledge in a form ready to be used for a class of problems of a given type, rather than in the context of compiling solutions to specific problems, as it is done in shallow expert systems.

3. Bus Arbitration in Multiple-Master Systems.

3.1 Introduction.

Computer systems comprise several interconnected modules that include processors, memory and I/O. These modules interchange information through private or shared communication links called buses. The bus is a fundamental structure in a computer system [1]. Whenever more than one master device can take over the bus, an arbitration mechanism which selects only one master is essential. This chapter presents bus arbitration techniques embedded into the general exclusive access framework. Finally two distinct protocols for bus arbitration that have been identified and incorporated in DAME are discussed.

3.2 Bus Concepts.

In order to build up a functioning microprocessor-based system, modules, such as processors, memory and I/O, must be properly interconnected. The bus is the collection of pathways that permits the various modules in the system to interact with one another. The advantage of a bus is that it needs fewer lines and fewer interfaces than a set of point-to-point links among the modules. Although serial buses play an important role in long-distance communication systems, parallel buses are preferred in microprocessor systems because of their increased capacity.

Buses may be physically limited within a module, or span several modules. There exists a hierarchy of buses in microprocessor systems according to the different characteristics that match the requirements of the devices they connect. Normally, one can expect a CPU bus, which is used for high speed transfers between the CPU and devices such as cache and co-processors, and a system bus on which slower devices, such as I/O

or system memory, exist. Depending on the complexity of the architecture, only one of the CPU or system bus, or additional point-to-point buses may be included. Figure 3.1 shows the diversity of buses that can coexist in such a system. Modern buses incorporate advanced protocols and technology so that efficient bus utilization and optimum transfer rates are achieved [34].

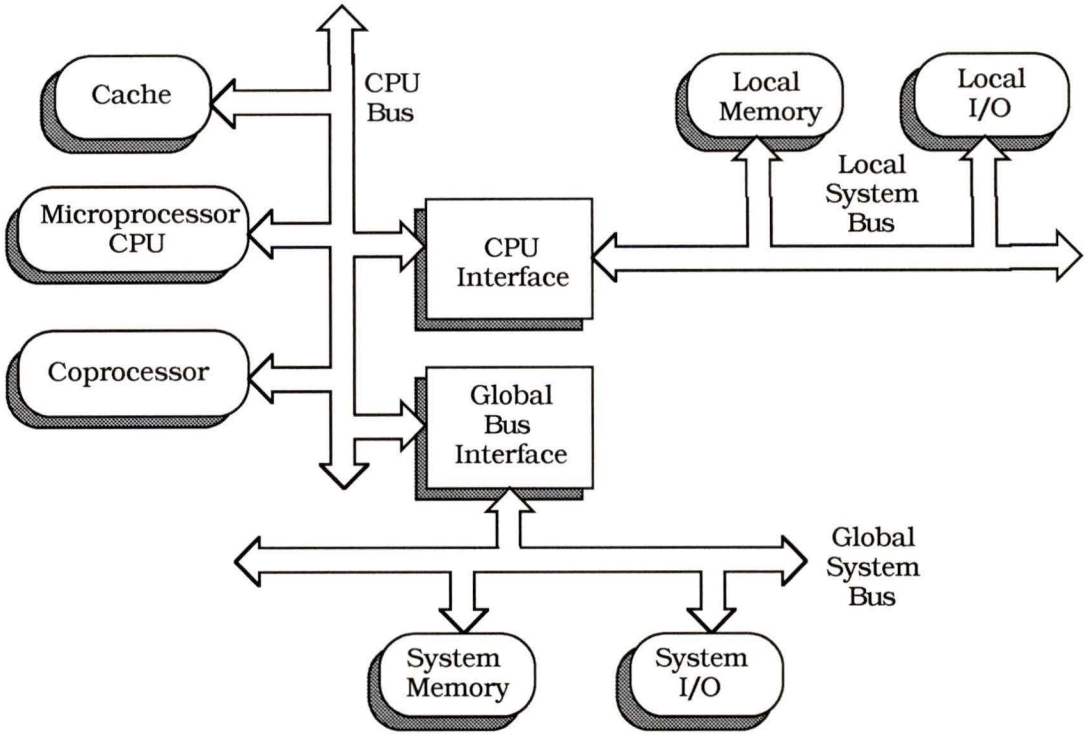
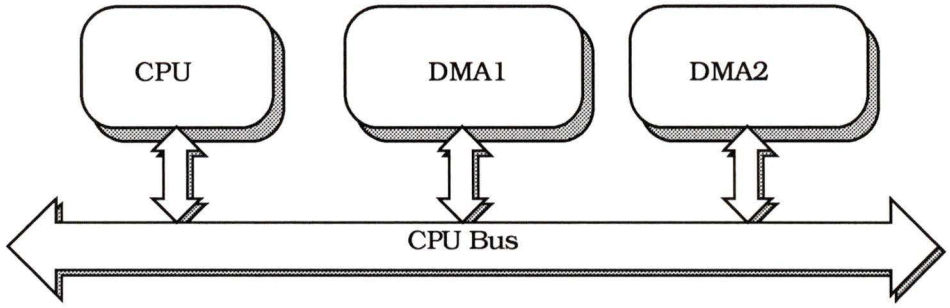


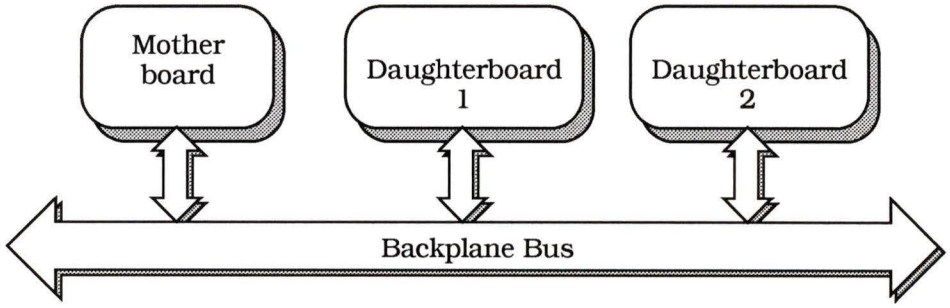
Figure 3.1 Bus hierarchy: (a) CPU bus; (b) local bus; (c) system bus.

The devices connected to the bus can be classified according to their roles as masters and slaves. Masters can initiate information transfer operations, while slaves merely respond to them. A multiple-master bus is a communication structure in which several units, masters and slaves, can be connected in order to perform independent information transfers. Two common examples of multi-master systems are a single CPU system with DMA devices, and a multiple-board system, as depicted in figure 3.2.

Since the bus is a single resource which carries only one information packet at any time, the multi-master bus protocol must include a sequence of operations that selects a unique master, called the current master (CM) or commander, to take over the bus at any given time. In the following, the bus allocation is presented as a particular case of resource sharing. Various methods for the selection of the bus commander in a multiple-master bus system are described.



(a)



(b)

Figure 3.2 Multi-master systems:
(a) CPU system with DMA devices;
(b) multi-board system.

3.3 Resource Allocation Model.

In this section, a general framework for the utilization of the bus by several masters is introduced. Independent-request, token-passing and collision-detection techniques to allocate the bus are discussed.

3.3.1 Single Resource Allocation Problem.

A logical model of the resource-requestor system is shown in figure 3.3. No information exchange is possible until an *allocator* gives exclusive access of the data transfer bus to a single master. The actual implementation of the allocator might be different from the logical structure in figure 3.3 because it might be composed of various sub-units which are either centralized in a single module or distributed among several modules. However the allocator must work as a single entity.

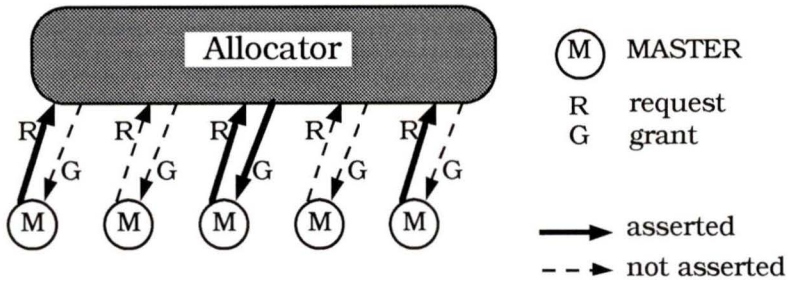


Figure 3.3 Logic structure of the allocator.

Allocation comes into play whenever several modules request the use of the bus simultaneously. This is called *contention*. The allocator must grant the resource to only one of the requestors. If the resource is used by more than one requestor at a time, a *collision* occurs. In some methods, collisions are legal during some part of the protocol. As shown in figure 3.3, each master that needs the bus sends a *request* to the allocator. Only one of them will receive a *grant* of the bus.

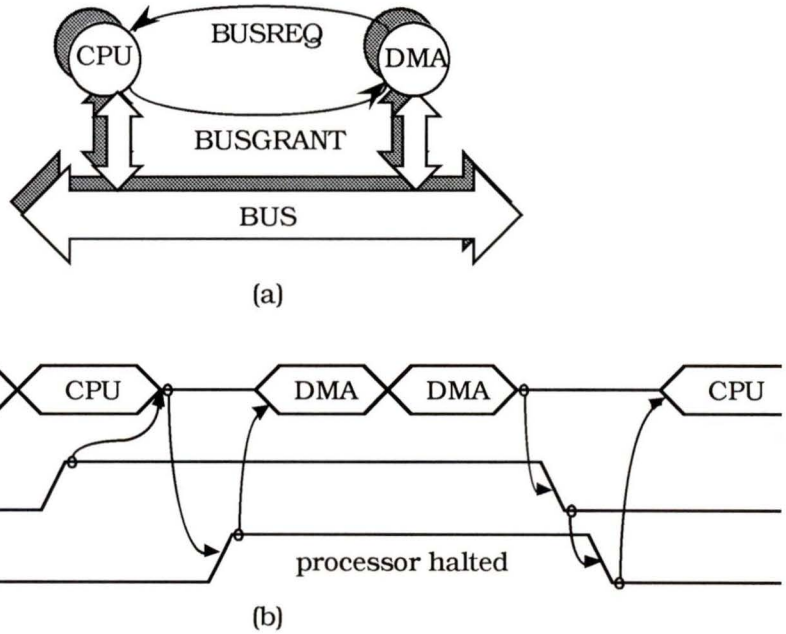


Figure 3.4 CPU bus exchange sequence.
(a) Block diagram; (b) Timing diagram.

3.3.2 Simple Exclusive Access System.

Consider a bussed system with only two masters. In such a system, the resource can be assigned to one user by default[1], and only be switched upon an explicit request

of the other master. This is called *default assignment*. In this case, the allocator may be physically part of the default user. The allocator only needs to guarantee that the bus is transferred at the right time to the other master. A typical system that follows this idea is a microprocessor local bus with a DMA device. The CPU has control of the bus, but releases it after an explicit request of the other master. The request is honored at the end of a bus transaction, as it is shown in figure 3.4.

The default assignment method can be used with more than 2 masters, but an exclusive access resolution technique must be incorporated to discriminate among the requests of the non-default requestors.

3.3.3 Model of the Exclusive Access System.

The N-way allocation may follow complex rules and is handled in most of the cases by a hierarchy of sub-units. A detailed view of the exclusive access process is given in figure 3.5[23].

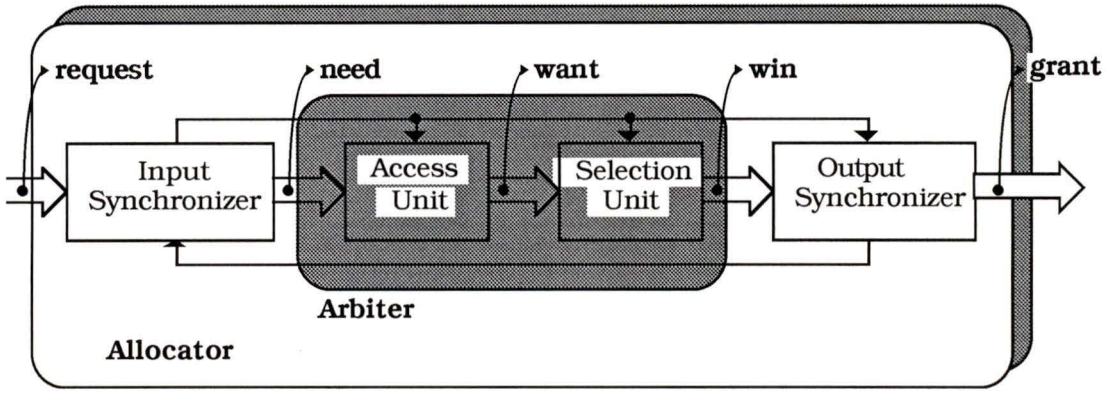


Figure 3.5 Detailed model of an exclusive access system.

Since the requests may come from asynchronous masters, or may be asynchronous because of propagation delays, the input synchronizer prevents the arrival of new requests while the arbitration logic is making a decision.

The arbiter selects the user which will be served in the next access cycle. This subsystem is composed of the access unit, and the selection unit. The access unit orders the requests according to the *access policy* (e.g.: based on the arrival time of the requests, or according to the identity of the requestors). The access policy also gives some

---

[1] Usually to the device that utilizes the resource most frequently.

information on how long a device will wait for the resource. The selection unit picks the WANT (request) on top of the service queue and activates the corresponding WIN output (resource).

The output synchronizer generates the grant only after the bus is available. In other words, the mastership of a bus must not be changed while a transfer is carried on. In general, a BUSY signal, or a combination of signals encoding the status of the bus, indicates whether one of the masters is using the resource.

3.3.4 Exclusive Access Techniques.

The basic techniques used for the exclusive access arbitration are the following:

1) Each requestor has an independent request signal, so that the allocator knows the identity of the active requestors. The arbiter makes a decision on which request to serve. This is called *independent-request arbitration.*

2) The allocator does not know the identity of the requestors. Therefore the grant is routed through all possible requestors. The grant is carried by a token. This technique is called *token passing arbitration.*

3) The resource requests bypass the arbitration unit and go directly to the output synchronizer. If many masters attempt to use the bus at the same time, a collision is detected. In order to avoid loss of information, the transfer is re-attempted after a delay. If after a specified time no collision is detected in the bus, the active requestor has been granted the bus. This is called *collision detection arbitration.*

3.3.5 Independent-Request Arbitration.

The allocator detects pending requests and the identities of the requestors and assigns the bus to only one of them. Possible access policies are:

FIFO.

The requests are served in the order of arrival. In principle this scheme would give all participants an equal opportunity to use the bus, provided that no device keeps the resource continuously.

Fixed Priority.

Because real systems have a finite time resolution, it might occur that the allocator cannot distinguish which one of two (or more) requests comes first. The *priority* is an ordering of the requestors such that each requestor is assigned a unique level. The highest ranked one will be served. The fixed priority arbitration presents the problem of *live-lock* that occurs when high priority devices keep alternating the use of the bus excluding the low priority requestors from using the resource. In this case it is impossible to guarantee an upper limit to the service time of any request.

Variable Priority.

In order to overcome the problem of live-lock in fixed priority schemes, a *fairness* rule can be used, such as:

- to raise the priority of requestors which have been waiting too long (*time-out*);
- to lower the priority of requestors that have already been served (e.g., *round robin*);
- to redistribute all priorities after every service (either randomly or according to a policy);
- to inhibit new requests until all pending requests are served (*masking*).

These techniques are fair, assuring an upper limit to the service time for all requestors.

3.3.6 Token Passing Arbitration.

In a token-passing scheme, the arbiter gives out the grant to each one of the requestors in a serial manner, until one of them takes the token. Possible strategies are described below.

Sequential Token Passing.

This simple allocation technique is an *a priori* assignment of the priority of the requestors. It can be visualized as a token that passes through the system. Since only one token can circulate in the system, the resource is automatically granted to only one

requestor and collisions are avoided.

We can model the sequential token passing allocator as an exclusive access system in which the requests are generated within the allocator, so that the input synchronizer is bypassed. This is shown in figure 3.6.

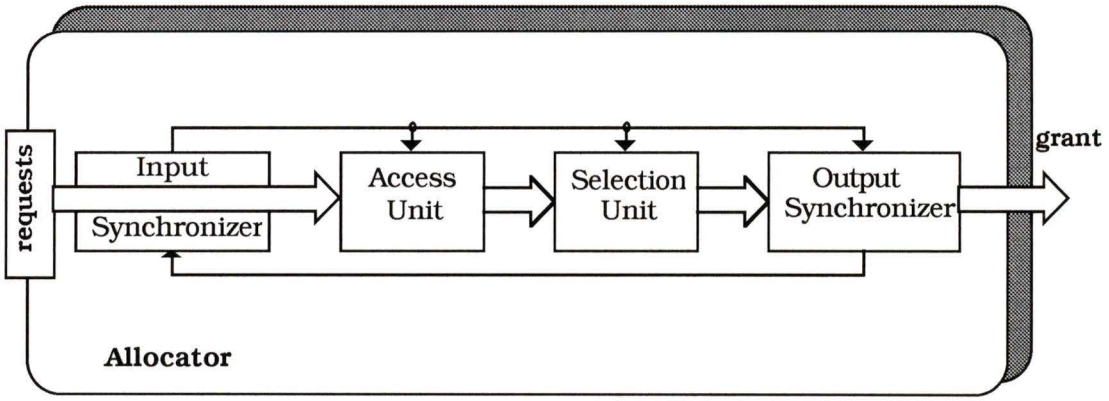


Figure 3.6 Model of sequential token passing allocator.

A fixed time-slot bus is an example of a centralized token-passing bus control (see figure 3.7). The token is usually passed in the same order (following a round robin scheme), but other sequences are possible. The round robin sequence guarantees a maximum service time which is equal to the sum of all the participants' maximum resource use time. If the worst case time is too long for certain devices, a strict priority could be used, in which the token is always returned to the highest priority device after its use (see figure 3.8).

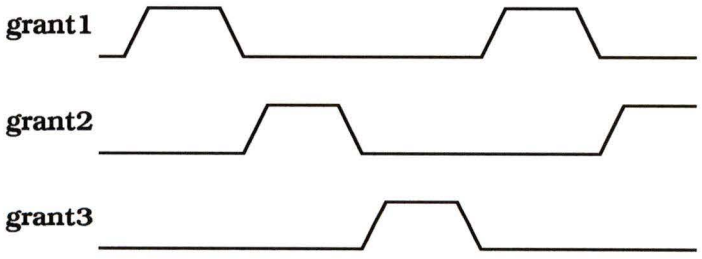


Figure 3.7 Assignment of the bus to 3 users in fixed time slots.

Switch-On-Demand Token Passing.

In sequential token passing, the granted user has no control of the token and it cannot pass it back if no request is pending. A first improvement uses a grant-acknowledge signal that is activated by the user at the end of the transfer. This technique is called *handshake grant.*

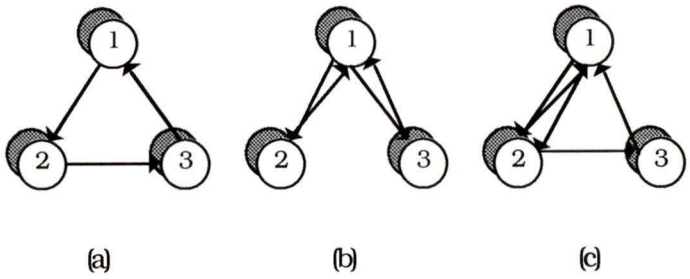


Figure 3.8 Token passing policies: (a) Round robin (123 ...); (b) strict priority (1213 ...); (c) strict priority II (12123).

An improved method does not pass the token unless there is a request for it. This method allows a device that repeatedly requests the resource to use the bus without giving back the token and waiting to get it again if no other master needs it. This technique is called *release-on-demand*. A combination of the release-on-demand procedure and the priority operation can reduce the average access time of some units.

3.3.7 Collision Detection Arbitration.

With the collision detection technique, requests are mapped directly into WIN signals and synchronized only with resource activity, as shown in figure 3.9. A request can be sent only if there is no activity in the bus. Due to propagation delays and the amount of time to make a decision, a requestor may see the resource as being available even if another one has already started to use it. This may cause a collision which would be detected by the users which, in turn, would release the bus.

An elementary sequence for the collision detection method consists of the following steps:

- a requestor checks the status of the bus;
- if busy, the requestor waits; if free, it starts its transaction and monitors for possible collisions;
- if a collision is detected, the access is aborted and the requestor will try again after certain delay;
- if no collision is detected after a fixed time, the requestor has been granted the bus.

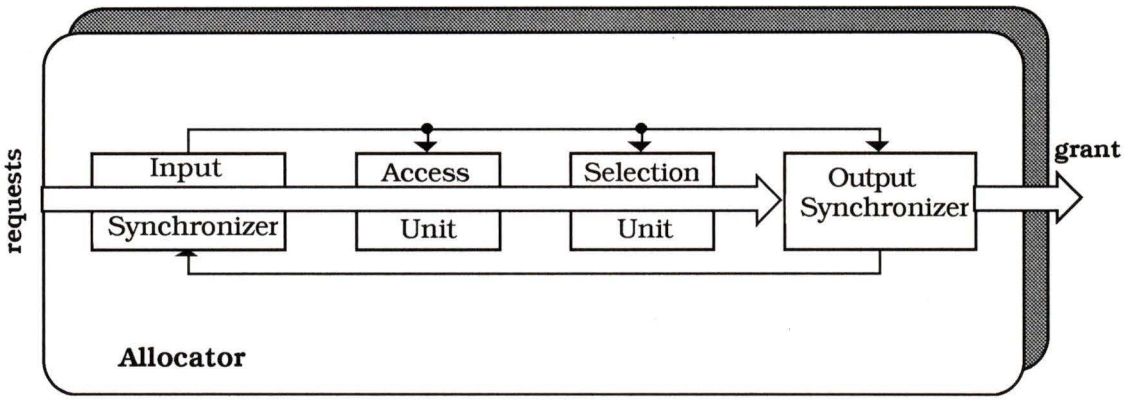


Figure 3.9 Collision detection allocator.

A collision may occur only at the beginning of a transaction. A *collision window* can be defined and depends on the maximum propagation delay of the system and on the speed of the interfaces. Since this method involves an abort-and-retry process, a message buffer and trial count are needed. For this reason the method is mostly used on serial communication systems. A collision detection is used in the arbitration resolution of the IEEE P896 (Futurebus) standard bus.

3.4 Arbitration Structures.

3.4.1 Centralized and Distributed Structures.

The bus allocator can be implemented using either a *centralized* or a *distributed* structure. Centralized arbiters are selected mainly when the number of users is fixed or limited. Distributed arbiters use a set of identical units and are generally selected for more complex systems. Figure 3.10 depicts the two approaches.

In some cases it is possible to find combinations of fully centralized and fully distributed arbitration structures. For example, a distributed access logic may be tied to a centralized selection unit.

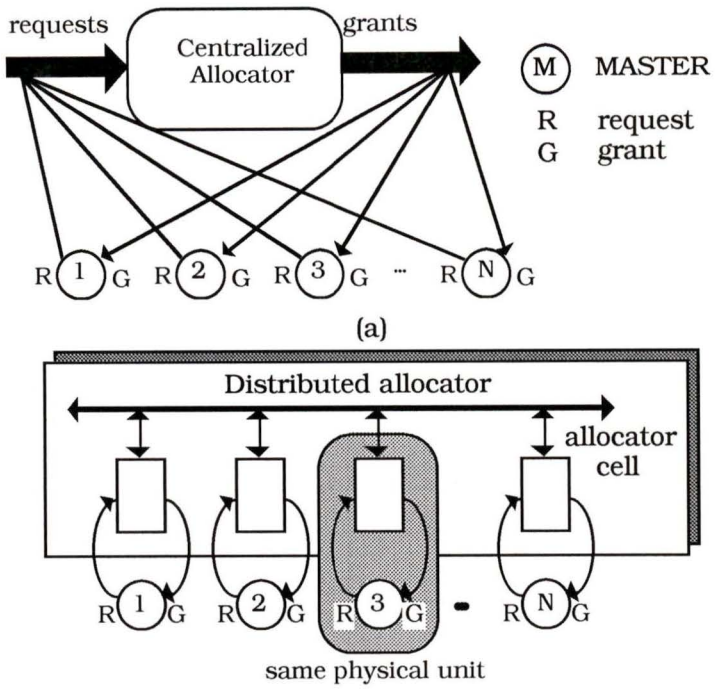


Figure 3.10 Arbitration structures.
(a) Centralized allocator; (b) distributed allocator.

3.4.2 Arbitration Circuits.

An overview of the basic implementation techniques for arbitration systems is presented, according to the degree of centralization. The possible access policies are also discussed for each technique.

Basic Priority Circuits.

The selection logic of a centralized arbiter is a standard priority circuit. Ripple or look-ahead circuits can be used as shown in figure 3.11. Look-ahead circuits are faster but the hardware structure is more complicated.

A variable-priority system is achieved by modifying the mapping of NEED signals into WANT signals according to a suitable priority change algorithm. Despite its simplicity and speed, the centralized selection unit requires 2N point-to-point connections for the WANT/WIN pairs in a star topology. It is possible to reduce the number of lines by encoding the WIN signals. An 8-requestor selection logic is available in a single IC (i.e., 74456). An example of a standard bus that uses a centralized arbiter is

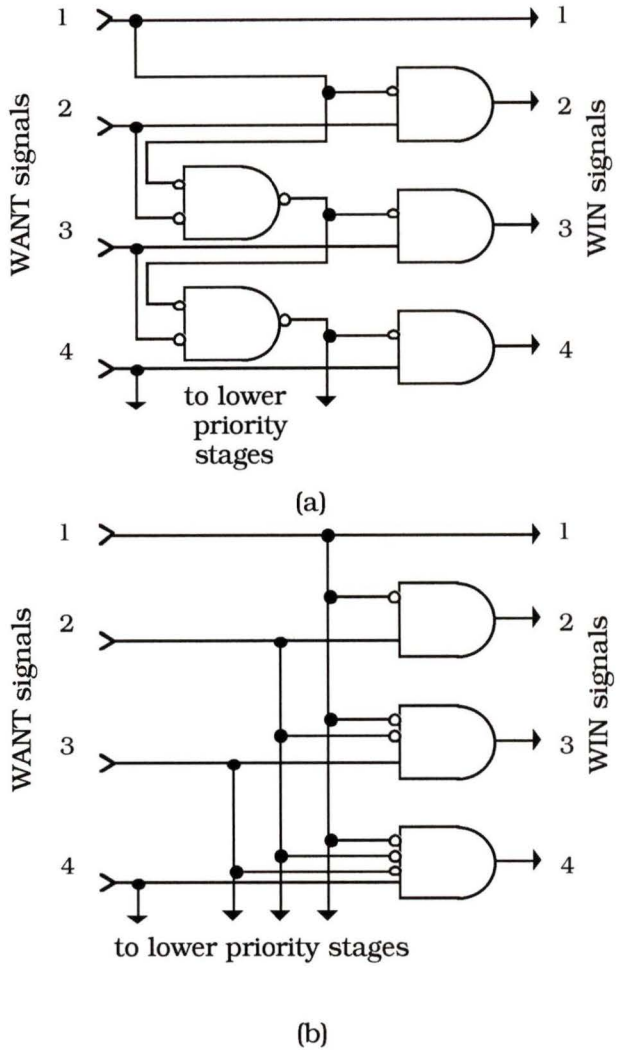


Figure 3.11 Basic priority circuits:
a) ripple logic; b) look-ahead logic.

Daisy-Chain Circuits.

The ripple priority logic depicted in figure 3.11a can be distributed as separate cells among the requestors as it is shown in figure 3.12. This is an example of a distributed arbiter obtained directly from the centralized version.

The daisy chain lines constitute a group different from the bussed lines but they can be incorporated easily in a backplane. The chain must be continuous. Boards that do not use the arbitration system must connect the daisy chain IN directly to the output,

and dummy boards with the same direct connection must be inserted in the empty slots of the backplane.

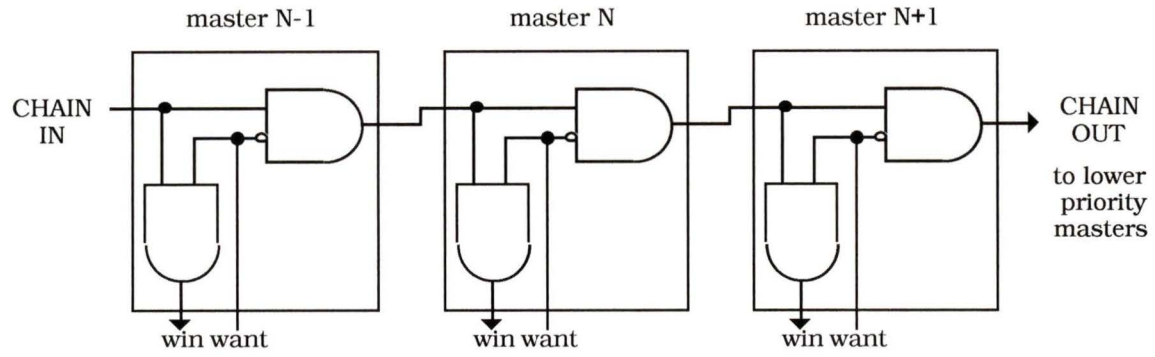


Figure 3.12 Daisy chain circuit.

The problem of live-lock exists in daisy chain arbitration systems because the priority of the requestors is fixed by their position in the chain. This problem can be avoided by using:

a) closed loop chains with rotating priorities where the access logic becomes a sequential machine in which the top priority level is routed sequentially to all requestors (round-robin); and

b) a distributed fairness logic so that new requests from served requestors are masked until all pending requests are served (masking).

The arbitration speed of a daisy chain is proportional to the number of requestors in the chain. On the other hand, the daisy-chain arbiter uses very few lines in the backplane, has simple circuitry and is fully modular.

Linear Self-selection Arbiter.

When each CHAIN-IN/CHAIN-OUT pair is assigned to a bus line, the structure is called a *linear distributed arbiter* or linear self-selection arbiter, as shown in figure 3.13. The access unit is the connection between the module requests and the bussed request lines. A fairness structure can be added at this stage.

Coded Self-selection arbiter.

The number of bussed lines in figure 3.13 can be reduced by encoding the requests. These encoded lines form the priority bus (PRBUS). Each requestor drives the PRBUS

through open collector gates and compares them with its own priority code (PRCOD). A more detailed explanation can be found in [89].

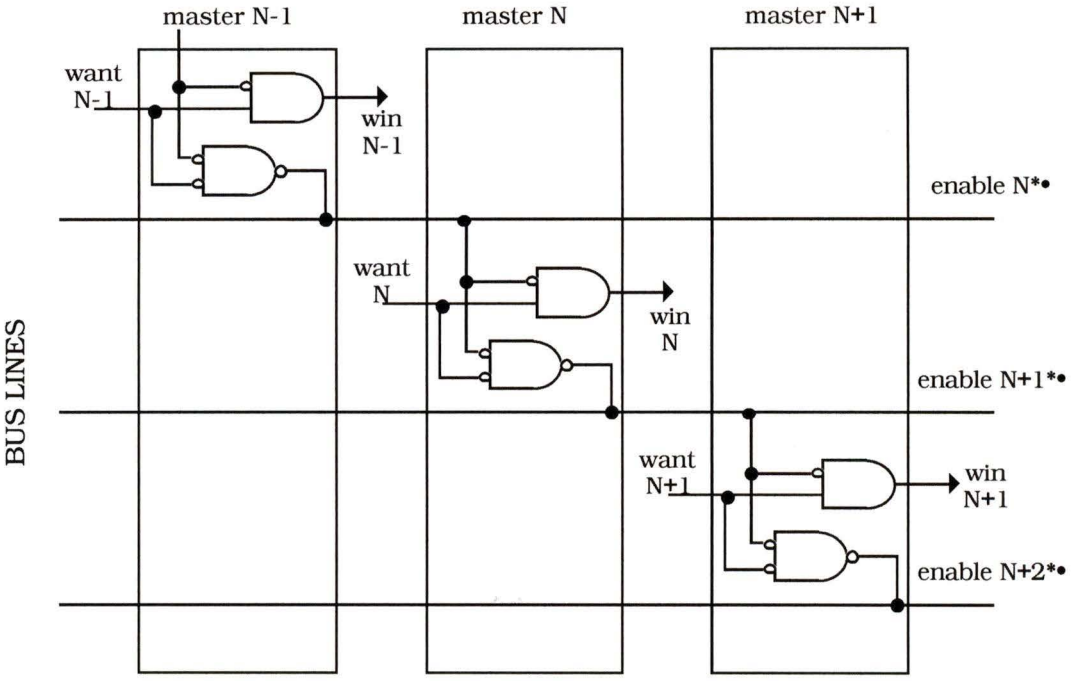


Figure 3.13 Linear self-selection distributed arbiter (ripple circuit).

3.4.3 Mixed Arbitration Techniques.

The methods presented above can be combined together in order to avoid some drawbacks and take advantage of the benefits. A widely used combination is the multi-level daisy-chain (see figure 3.14). This structure is more flexible than that of a single chain because the priority of a requestor can be changed (either by moving it to another chain or by changing the priority of its chain in the central arbiter).

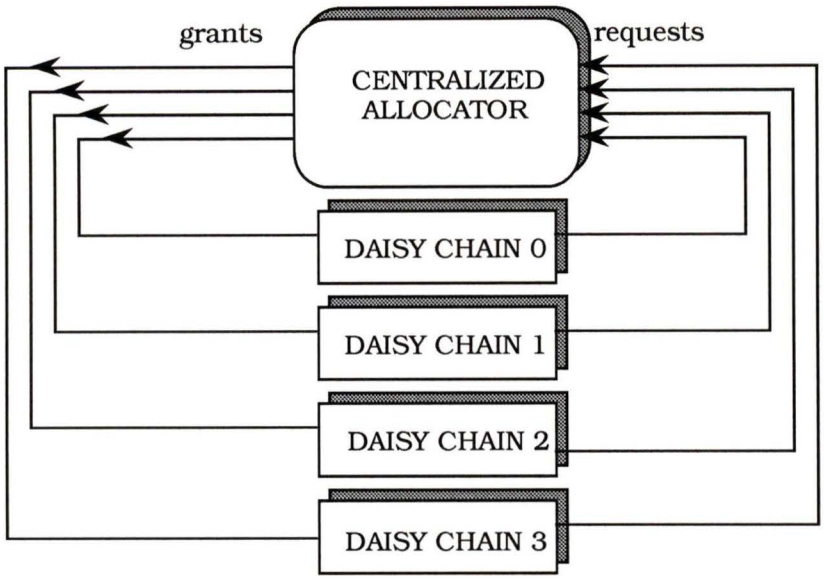


Figure 3.14 Multi-level daisy chain.

3.5 Input Synchronization.

The asynchronous nature of the arbitration makes it vulnerable to the synchronization problem [63] in which, owing to the fact that physical systems responds in a finite amount of time, two simultaneous requests to an arbiter may yield a metastable state in which the output is unpredictable. Although this unstable state will decay, there is no upper bound for that moment.

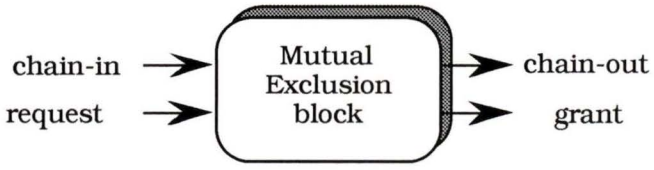


Figure 3.15 Mutual Exclusion block.

There are circuits that minimize the likelihood of the metastable state and avoid the hazards that arise in this kind of situation [23,91] (i.e., emitting two grants to two different requestors that will take over the bus, producing a collision that may even damage the hardware). The basic element in those circuits is the Mutual Exclusion block (ME) [81,11], shown in figure 3.15. Even if the **chain-in** and **request** transitions occur simultaneously, eventually only one of the two outputs will be asserted. If **request** occurs

earlier than **chain-in**, the requestor is granted the bus via the **grant** signal, otherwise the grant is propagated through the daisy chain to the lower-priority requestors.

3.5.1 Daisy Chain Model.

A daisy chain can be modelled by cascading the mutual exclusion (ME) blocks, as shown in figure 3.16. The token is fed into the **chain-in** of the ME corresponding to the highest-priority interface, and it propagates through the **chain-out** to the lower-priority devices. The **request** terminal receives the WANT signal from the requestor, while the **grant** terminal carries the WIN signal from the allocator.

In this model, the synchronization problem has been isolated in the ME block. It is noted that in the ME element both the synchronization and the distributed arbiter logic are combined together. It is possible to guarantee the arbitration principle: only one requestor will obtain the bus at a time, the performance penalty being the decaying time of the metastable state.

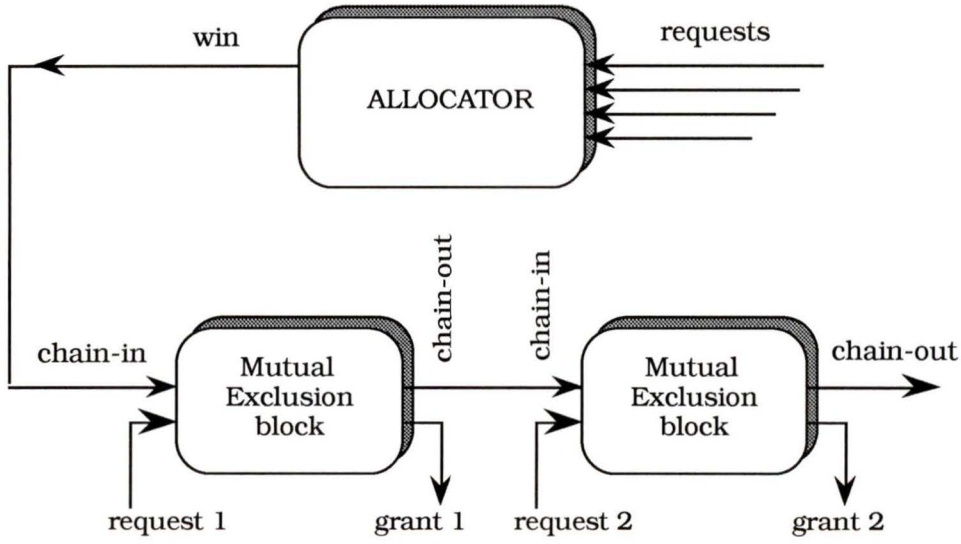


Figure 3.16 Daisy chain model using ME blocks.

3.6 Concurrence in Allocation and Transfer Operations.

For each transaction, the allocation of the bus must be performed before the data transfer can be carried out, as shown in figure 3.17a. To increase the throughput, a default mastership policy can be used, that generates the release-on-demand sequencing

in figure 3.17b. The allocation can be performed concurrently with the use of the bus as shown in figure 3.17c.

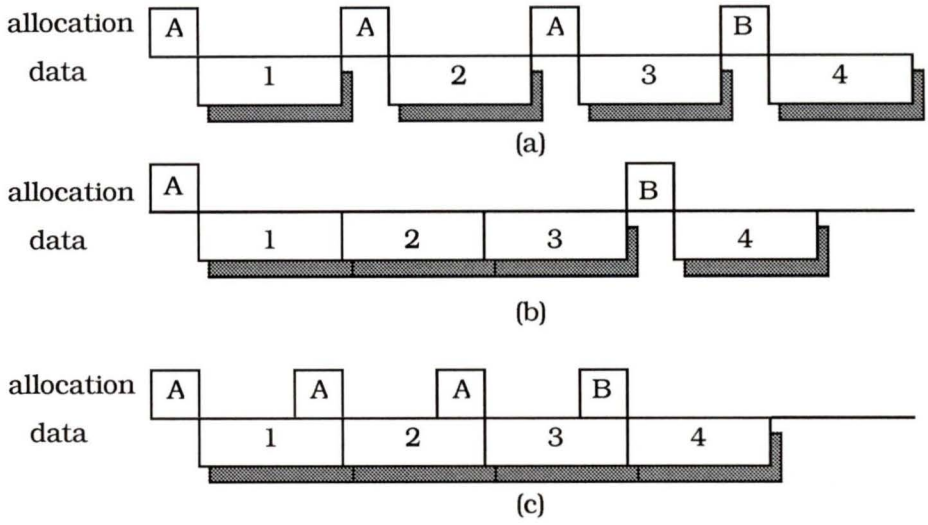


Figure 3.17 Sequencing of allocation and transfer in a multi-master bus:
(a) sequential allocation; (b) sequential, release-on-demand;
(c) concurrent allocation.

Sequential allocation is less expensive in hardware. In parallel buses where an extra arbitration channel is available, allocation and data transfer can be done concurrently. The VMEbus [45] provides separate bus arbitration lines that follow a protocol that permits the allocation to take place while the commander is performing the last transaction.

3.7 Bus Arbitration Protocols.

A bus uses a protocol through which the arbiter can select the current master or commander of the bus. In the subsequent, we shall introduce briefly two commonly used protocols.

The basic handshake used by a master to request the bus is shown in figure 3.4. Two control lines are required, a request line from the master to the arbiter, and an acknowledge line in the opposite direction. This is called a two-signal protocol.

A timing diagram for the two-signal protocol is shown in figure 3.18. In this timing diagram, active-low signals are used to implement the protocol. A widely used

notation to represent active-low signals is to write a trailing star after the name of the signal. A requestor asserts REQ* to request the bus, and negates it at the end of its transaction. The ACK* signal, when asserted, indicates that the bus has been granted to the requestor, and when negated, acknowledges the end of the operation. This conforms to a fully responsive handshake protocol, as discussed in the next chapter.

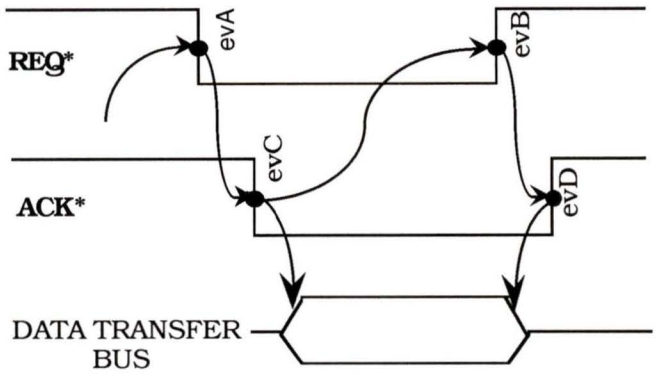


Figure 3.18 Two-signal bus arbitration protocol.

As mentioned in section 3.6, the allocation can be performed in parallel with the data transfer. The three-signal protocol provides an extra line to accomplish concurrence. A corresponding timing diagram, taken from the VMEbus specifications, is used to explain this protocol, as shown in figure 3.19. This figure defines the arbiter part of the protocol.

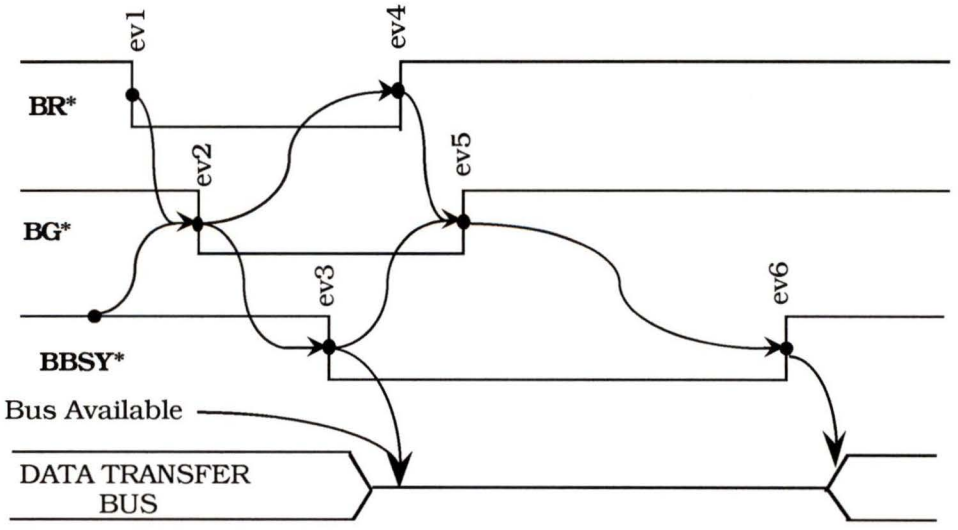


Figure 3.19 Three-signal bus arbitration protocol.

A device requests the bus by asserting BR*. The arbiter responds by asserting BG*

if **BGACK*** is not active. When the device receives the grant, it must release **BR*** and has to wait until the bus is available before starting using the bus, allowing the previous master to end its last operation. The device informs that it will become the current master (CM) by asserting **BGACK***. After it detects that a device has acknowledged the grant, the arbiter releases **BG***. Finally the CM allows the arbiter to start another arbitration cycle while it is performing its last transaction by negating **BGACK***.

The power of the three-signal protocol lies to the fact that the determination of the new master can proceed while the previous master is performing its last transaction, speeding thus the transactions on the bus.

The three-signal protocol has been used for the N-partner broadcast and broadcall data transfer operations in the P896 Futurebus [20].

3.8 Summary.

All modern microprocessors incorporate in their functionality a bus arbitration capability that allows to build multiple-master systems. In this chapter, the bus allocation concepts were discussed, in the framework of the general resource exclusive access problem. Several structures were studied, mainly token passing, arbitration, and collision detection. Because arbitration is the most widely spread technique used in multi-master microprocessor systems, this work focuses on its particular features.

According to the structure, arbitration can be centralized or distributed. The latter is generally preferred, due to its modularity. The daisy-chain scheme can be found in most of the microprocessor configurations. According to the algorithm used to perform the arbitration, fixed priority and variable priority techniques are available. While fixed priority is simple to implement, it has the problem of live-lock, in which the lower priority devices can starve as the higher priority devices take turn in taking over the resource. Variable techniques, such as round robin, strives for a fair arbitration, in which a maximum service time is guaranteed for every master.

The relation between the synchronization problem and arbitration was established, for the arbiter must select the unique commander of the bus even when all the requests occur simultaneously. A mutual exclusion mechanism must be ensured in the arbitration system to prevent malfunctions. Finally, two bus arbitration protocols were identified, that will serve as a primary example in the rest of this work.

4. Protocol Description.

4.1 Introduction.

In its classical meaning, a protocol is a code prescribing strict adherence to correct etiquette and precedence [94]. Protocols play an important role in computer systems by establishing the actions necessary for the exchange of information between modules comprising the system. In this chapter, action graphs are introduced as a means of describing protocols. In order to get a flavor of the basic protocols, they are first addressed in the context of a point-to-point connection. A notation that connects the logical actions in the protocol with the actual signals is also presented. Finally, the two bus arbitration protocols which were identified in the previous chapter are modelled using action graphs.

4.2 Basic Information Transfer Protocols.

In a *point-to-point* connection, information moves from a source to a destination, through a link (figure 4.1). The link consists of a data path, which carries the information, and a control path, which directs the operation of the data path. The data path and control path may be multiplexed over the same physical medium, although in computer systems they are implemented on distinct media.

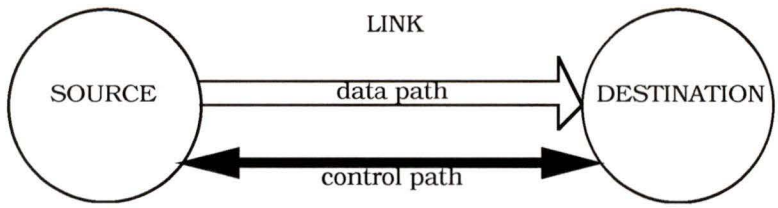


Figure 4.1 Point-to-point transfer representation.

The *protocol* of the link specifies the sequence of *actions*, or elementary indivisible operations, which correctly transfer the data from the source to the

destination through the link. The *activation* of each action of the protocol is associated with the occurrence of other actions that must precede it.

In a computer system, the complete information transfer from the source to the destination is called a *transaction*. In a point-to-point link, the transaction consists of a single data transfer, while in a multiple-master system, the transaction can be divided into three steps: arbitration, addressing and data transfer.

A *cycle* is the sequence of elementary actions that completes one aspect of the transaction (i.e., arbitration). The different levels of abstraction of the information flow, as represented by the hierarchy of transactions, cycles and actions, is shown in figure 4.2. From now on, cycles are the abstraction level in which the basic protocols will be introduced.

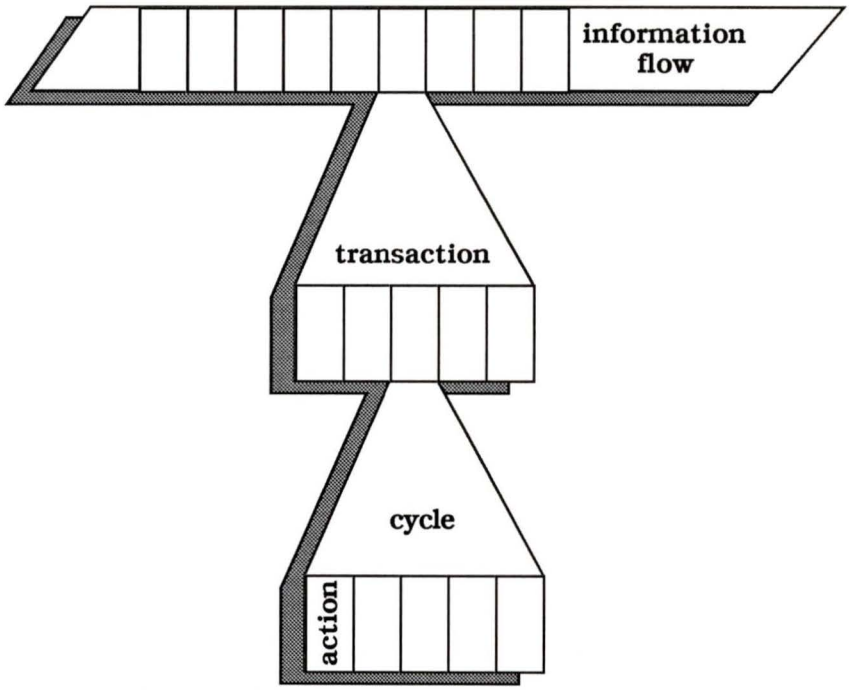


Figure 4.2 Hierarchy of transfer operations.

Before presenting the basic protocols for transfer information, the mapping of actions into signals is discussed in order to give a flavor of the concepts that will be introduced in the subsequent.

4.2.1 Strobe Action.

An action is associated with a state or change of state of a signal. A signal is a

boolean variable which can take two values: TRUE or FALSE, or equivalently, active or inactive. A one-to-one correspondence between actions and signals is not indispensable. For example, 2^N mutually exclusive actions can be encoded using N signals. These mapping choices produce numerous possible implementations of a protocol.

Consider a transfer of several bits of information from the source to the destination. A strobe action is required when encoded data are transferred because the voltage levels in the lines settle down at different times (skewing effect). The encoded strobe signal provides a time reference for the settling of all the data lines as shown in figure 4.3. The set-up time t_{su} in the transmitter guarantees a minimum set-up time at the receiver t_{su}' for a minimum propagation time t_p and a maximum propagation time $t_p+\Delta t_p$.

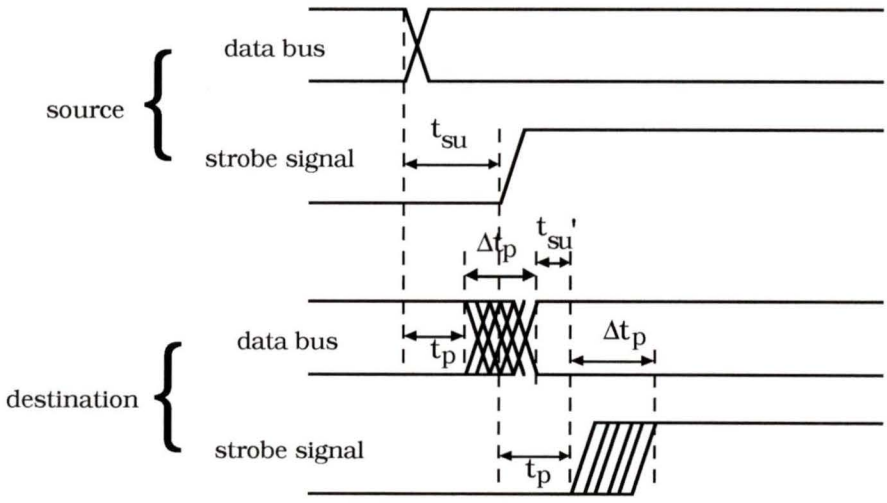


Figure 4.3 Strobe signal propagation through the link.

4.2.2 Synchronous and Asynchronous Transfers.

When the source initiates a transaction, it needs to perform at least two actions:

DATA action.- Place the data in the data path.

STROBE action.- Inform that the data lines are valid.

There are two basic approaches to initiate another cycle:

a) The source assumes that the STROBE action has been received and acted upon by the destination within a fixed time t_a, after which it may start another cycle. Because the source has total control of the timing and thus synchronizes the link to its clock,

this technique is called *synchronous*.

b) A new ACCEPT action is inserted by the destination which acknowledges the STROBE action. In this case both source and destination actively cooperate and inform each other. Since time is not controlled solely by either of the two participants, this technique is called *asynchronous* or *handshake*.

Figure 4.4 shows the action sequence for a synchronous transfer. The source places the data in the link (DATA action A), signals action STROBE (B), and waits t_a for the destination to read the data before starting a new cycle (C).

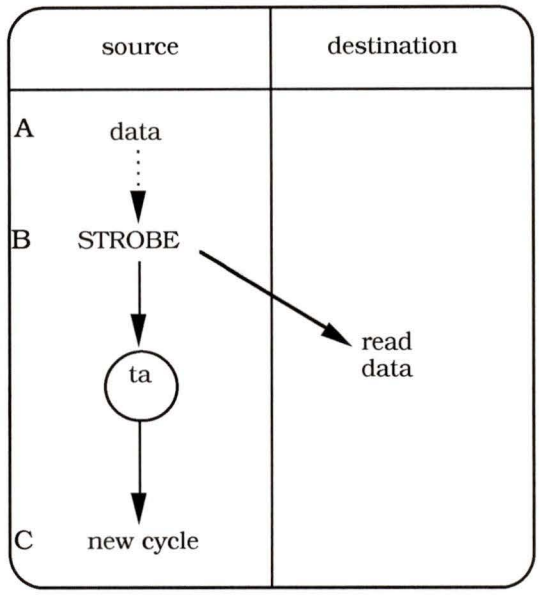


Figure 4.4 Synchronous sequence of actions.

A possible implementation uses the active (high) value of a **strobe** signal to encode the STROBE action as depicted in figure 4.5. In the synchronous technique, the speed of the destination device must match the speed of the source (i.e., it must be at least as fast as the source) to ensure correct operation.

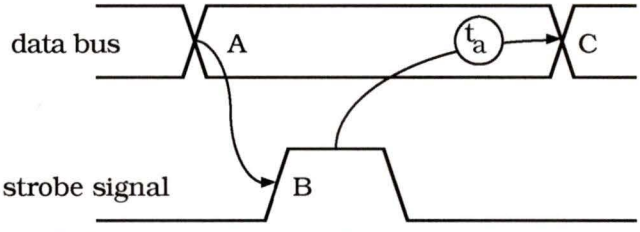


Figure 4.5 Timing diagram for the synchronous sequence.

Similarly for the asynchronous case, the sequence of actions is shown in figure

4.6. In this case, the source waits for action ACCEPT to take place before starting a new cycle. In other words, the cycle adjusts its speed to both the source and destination devices.

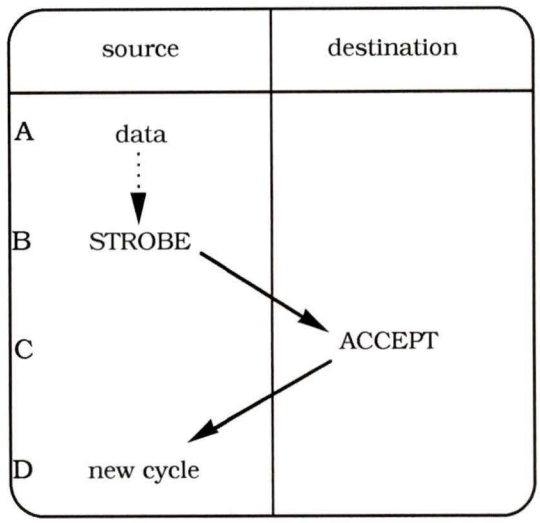


Figure 4.6 Handshake sequence of actions.

An usual implementation of the handshake actions (see figure ## NO TEXT IN MARKER: "Simple handsh timing diag" ##) uses two signals: the **strobe** driven by the source, and the **accept** driven by the destination. In this example the STROBE action is encoded as leading edge of **strobe** and the ACCEPT action as the leading edge of **accept**.

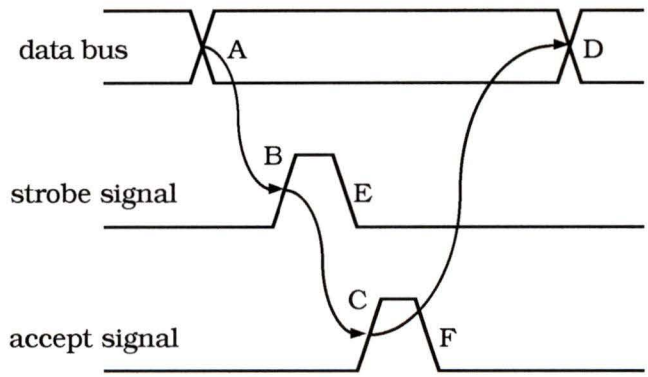


Figure 4.7 Timing diagram for the handshake sequence.

4.2.3 Fully Interlocked Cycles.

In figure ## NO TEXT IN MARKER: "Simple handsh timing diag" ##, it is possible to associate another pair of actions to trailing edges E and F. The new action NOT-VALID is used to inform that the data is no longer valid, while READY notifies that the destination is ready for the next cycle. A cycle that includes these new actions is called a

fully-interlocked, or *fully-responsive*. The sequence of actions for this full handshake is shown in figure 4.8.

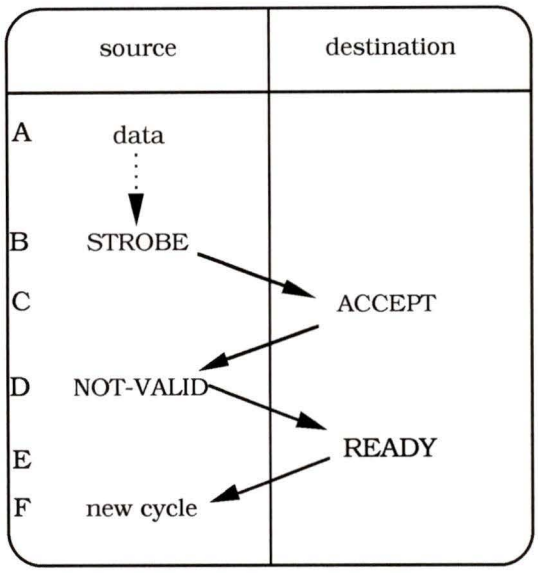


Figure 4.8 Four-action handshake protocol.

Figure 4.9 shows how the full handshake is accomplished between two signals **strobe** and **accept**.

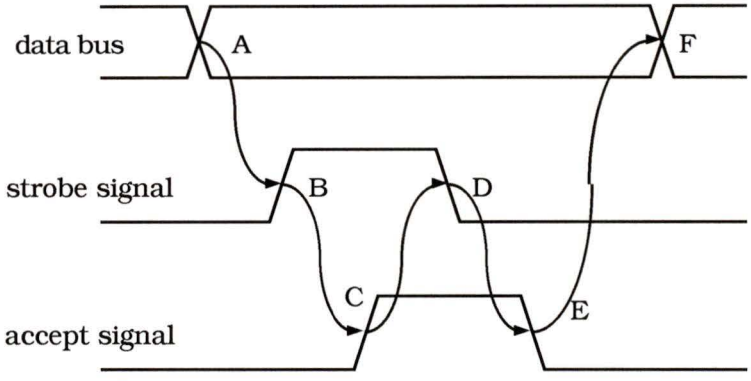


Figure 4.9 Fully interlocked four-action handshake protocol.

The fully interlocked handshake protocol avoids the following hazard which may arise if the partial handshake were to be used. Assume that a second strobe action happened before the destination were ready to receive it. This is the case when the **strobe** is very narrow while the **accept** is wide, as shown in figure 4.10. The source may mistakenly take the asserted value of **accept** for a response to the next transfer and remove the data lines. As a result one transaction is lost.

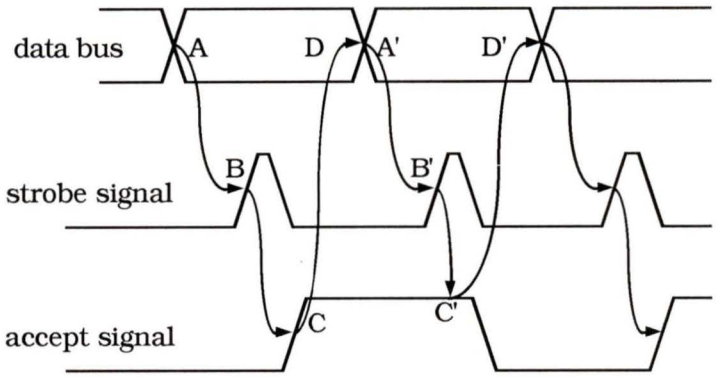


Figure 4.10 Possible hazard in the partial handshake protocol.

The partial handshake can be made safe provided that adherence to a strict set of timing constraints on the non-interlocked transitions E and F (see figure ## NO TEXT IN MARKER: "Simple handsh timing diag" ##) is observed. The main advantage of using the partial handshake is that the bus bandwidth can be increased by allowing edges E and F to represent a second pair of strobe and accept actions. This method is used frequently in block transfer operations. The transfer cycle in figure 4.11 shows a block write transaction as defined by the P896 Futurebus, for an even number of data[1]. While the addressing actions, encoded by the strobe and acknowledge signals AS* and AK*, comply to a fully responsive handshake protocol, the data actions, represented by the strobe and acknowledge signals DS* and DK*, observe a partial handshake protocol. The speed of the transaction is roughly increased by two by using the partial handshake.

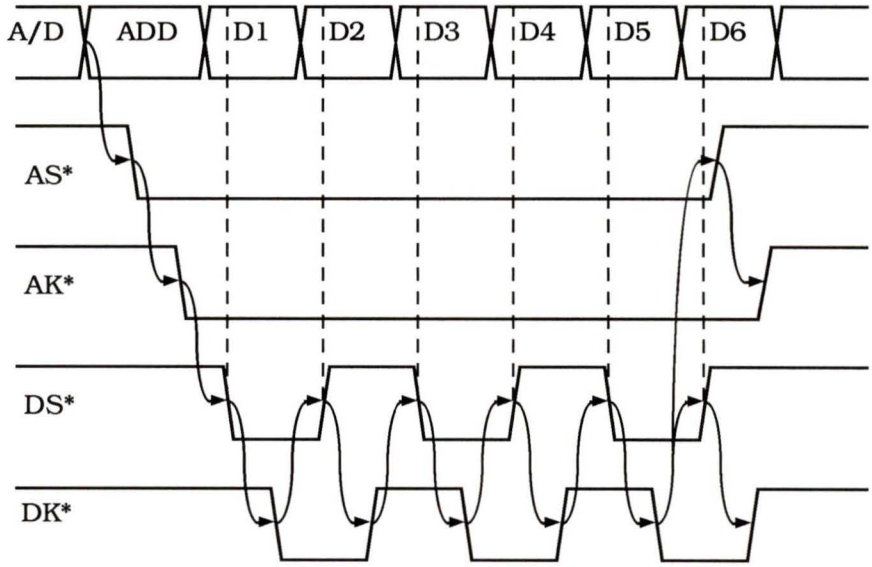


Figure 4.11 Block write transfer in the Futurebus.

[1] Odd transfers have a different end-of-cycle sequence.

Closed Cycles.

In order to allow the chaining of identical transactions, a cycle must be closed. A cycle is said to be complete or *closed* when the actions in the sequence cause their associated signals to return to their original level at the end of the cycle [20]. In the partial handshake, the closure of the cycle is accomplished either by inserting two dummy transitions or by using those transitions for another cycle, while in the fully-responsive handshake, its four actions form a close cycle.

4.2.4 Synchronous Cycles.

In synchronous operation mode, the destination does not supply the corresponding actions of the sequence, as discussed for the asynchronous case. The source assumes that within a fixed time the operations of the implicit actions will be performed by the destination. Figure 4.12 displays the sequence of explicit and implicit actions and delays for a *closed* synchronous cycle with an additional NOT-VALID action and a corresponding delay t_b.

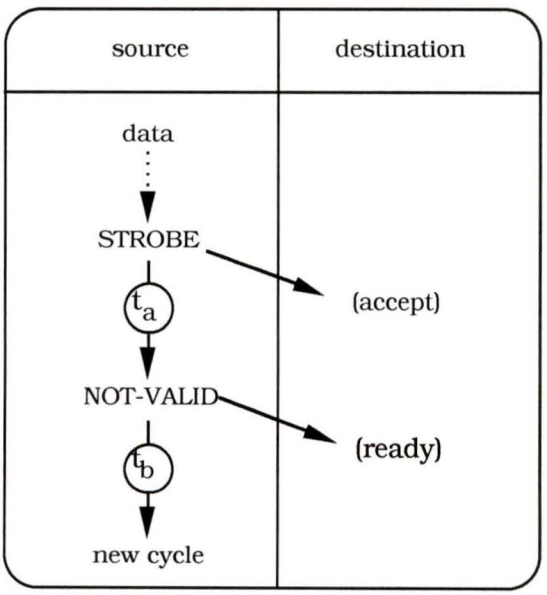


Figure 4.12 Synchronous closed cycle.

4.2.5 Semi-Synchronous Cycles.

Although synchronous protocols are simple, they are too rigid for applications in which the responders have different operation speeds. The handshake protocol is

more versatile but uses more control lines and requires more complex interfaces. An intermediate solution consists in designing a protocol that operates synchronously by default but transmutes into handshake following an explicit request. This is called a *semi-synchronous* protocol. The action sequence of a semi-synchronous write cycle is shown in figure 4.13. In this case only if a NOT-ACCEPT action is not generated by the responder, the NOT-VALID action can proceed after a delay t_a. Otherwise NOT-ACCEPT and NOT-VALID are interlocked.

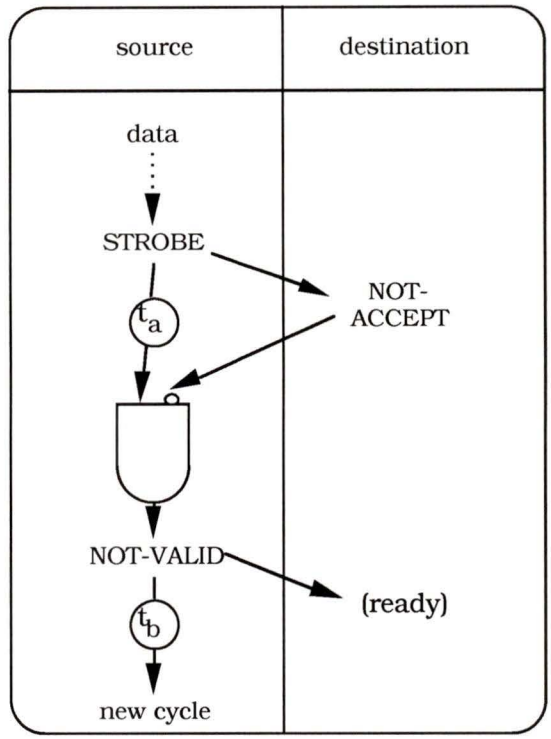


Figure 4.13 Semi-synchronous write cycle.

The semi-synchronous protocol can be described as the agglomeration of two protocols: one synchronous and one asynchronous. The designer is left with the choice of selecting the protocol that fits more her requirements.

4.3 Source- and Destination-Activated Cycles.

The cycles considered so far have the characteristic of having been initiated by the source unit. Such cycles are called *source-activated* cycles. Similarly, a cycle is called a *destination-activated* cycle when it is commenced by the destination unit. In the context of data transfer operations, source- and destination-activated cycles are also called write

and read cycles, respectively. The sequence of actions for a read asynchronous cycle is shown in figure 4.14. A new action REQUEST by the destination unit starts the cycle. A corresponding timing diagram, in which the actions are mapped into a **request** and a **strobe** signals in the destination and source units respectively, is shown in figure 4.15.

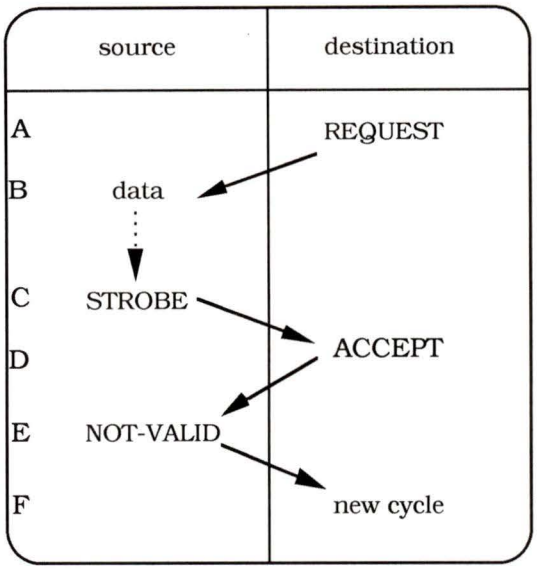


Figure 4.14 Actions in the handshake read cycle.

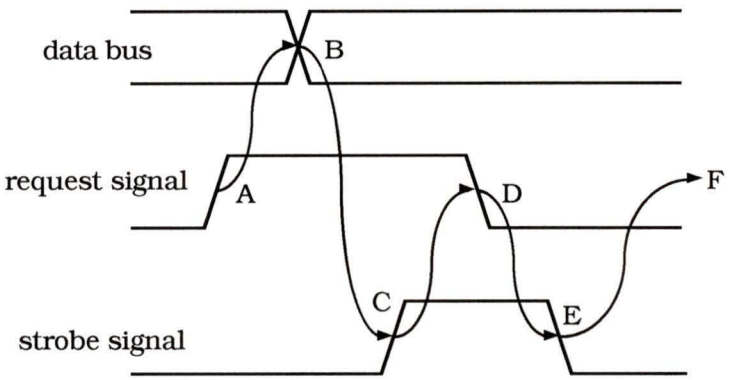


Figure 4.15 Timing diagram for the handshake read cycle.

That brings into consideration two types of communicating units and two roles for them. A commander is a module that can start a cycle. The other unit selected by the commander to participate in the transaction is called the responder. Modules that can act as commanders are called masters, while modules that only participate if requested are called slaves. The quality of being a master or a slave is a property of the module [20] (i.e., a module is designed to be a master or a slave, or both, and it becomes a commander or a responder operatively).

Figure 4.16 shows the implicit actions in a read synchronous cycle. The

destination acts as the commander, while the source becomes the responder.

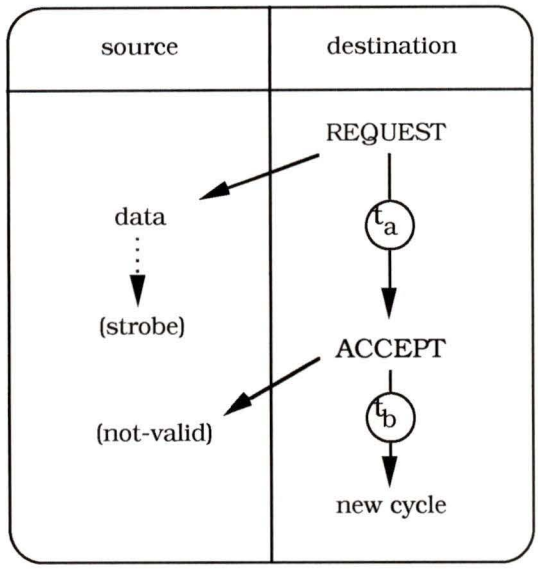


Figure 4.16 Synchronous read cycle.

4.3.1 Clocked Interfaces and Protocols.

A processing logic is usually synchronized by a clock. All actions in the protocol are referenced to this clock. In synchronous and semi-synchronous protocols in a clocked system, the delays are related to the clock edges instead of being a continuous variable. Also a handshake protocol can be modified into clocked mode if, for instance, the acknowledge signal is sampled at the clock's falling edge.

If all the system components are timed with the same clock, there is no synchronization problem and one can speak of a clocked protocol. If different clocks are used in the units, care should be taken to minimize metastability problems inherent of synchronization [63].

4.4 Protocol Conversion.

When the units used to build up a system follow different protocols, the control signals of each interface must be adapted to the common protocol utilized by the interconnection link.

There are two levels of protocol conversion. At a lower level, one protocol may

change the encoding of the actions, but it follows basically the same sequence. At a second level, the protocol modifies the synchronization of the actions.

At the re-encoding level, the conversion is achieved by changing the correspondence between actions and signals. This process usually requires only combinatorial logic [23]. Figure 4.17 shows the interface between a Intel 8086 CPU and static RAM. The protocols used in both the microprocessor and the memory are synchronous. Without considering the addressing aspect, the interface must generate the R/$\overline{W}$ and $\overline{CE}$ signals from the $\overline{RD}$ and $\overline{WR}$ pulses.

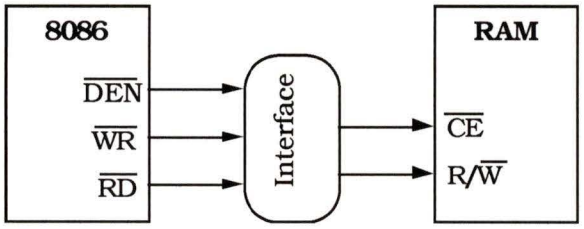


Figure 4.17 An example of action encoding in protocol conversion.

The second level usually demands a more involved design. An example of synchronization of actions is shown in figure 4.18. A synchronous responder is connected to a handshake link. In this case because the responder does not signal back to the commander, the interface must produce the $\overline{ACK}$ signal needed in the link's side.

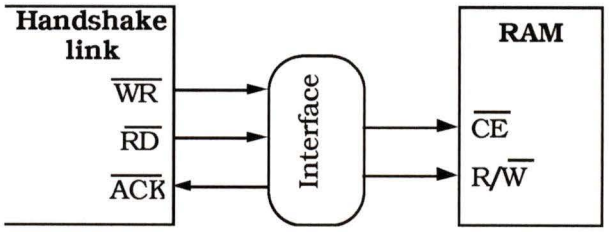


Figure 4.18 An example of action synchronization in protocol conversion.

With regard to the speed of operation, the following can be stated: Synchronous commanders must run at a fixed speed with an upper bound given by the slower of the responders connected to the link. A synchronous responder can be easily interfaced to slower commanders or to any commander with a semi-synchronous or handshake protocol. Only when the link is synchronous, must the speed of commanders and responders match (the slowest responder must be faster than the fastest commander).

4.5 Signals, States, and Transitions.

Protocols are described by elementary actions at the logical level. At the physical level, actions are encoded into signals. A notation is required that permits the encapsulation of the temporal information of physical signals and to relate this information with the logical actions of the protocol. This section defines the syntax used to model the behavior of the various signals associated with the components used in the design of microprocessor-based systems. The Backus-Naur Form (BNF) notation is used[2].

4.5.1 Signal State.

A signal associated with a particular device or bus is assigned a name which consists of a unique string of characters.

signal-name::= {CHAR}$^{+}$.

Signals are found to exist at well defined states. The state of a signal can be defined as a triplet as follows:

signal-state::= "(" value directionality validity ")".

value::= enabled | disabled.

enabled::= ACTIVE | INACTIVE.

disabled::= HIGH-IMPEDANCE | OPEN-COLLECTOR.

directionality::= INPUT | OUTPUT | BIDIRECTIONAL.

validity::= VALID | INVALID.

A boolean signal can take only the *values* true and false, or similarly active and inactive. When a signal is either active or inactive, it is said to be enabled, otherwise it is disabled. As a result of multiplexing several signals in the same line, it is necessary to be able to disconnect output signals. Two approaches are either to use tri-state logic or WIRED-OR logic. A tri-stated signal has a third floating state called HIGH-IMPEDANCE. An "open-collector" signal (i.e., open-collector in TTL, open drain in MOS, or open-emitter in ECL technology) can be tied together in an ORed fashion because it

---

[2] The following symbols have a special meaning in the BNF notation:

::=	definition
{ }	repetition of zero or more times
{ }$^{+}$	repetition of one or more times
[ ]	optional
|	exclusive-OR
" "	string

can take only two states: ACTIVE and OPEN-COLLECTOR. Open-collector signals are common in bus arbitration because of their OR capability without additional gates.

The *directionality* of a signal with reference to a module indicates if the signal is conveying information to the module (INPUT signal), the outside world (OUTPUT signal), or both (BIDIRECTIONAL). Input signals can have only enabled values.

Sometimes it is not important to know the precise value of the signal but its validity. This situation is expressed by the *valid* dimension. For instance one may want to describe the instant when the address lines in a microprocessor become stable, independently of their value. This example is illustrated subsequently in this section.

Not all the combinations represent meaningful signal states. For instance, there is no case in assigning to an invalid signal a state value. It is possible to attach a mnemonic to a state. A list of the most frequently used states and their mnemonics are given in the following, where *any* indicates a don't care value for this particular dimension.

ASSERTED:= (ACTIVE *any* VALID).
NEGATED:= (INACTIVE *any* VALID).
UNKNOWN:= (enabled *any* VALID).
VALIDO:= (enabled OUTPUT VALID).
INVALIDO:= (*any* OUTPUT INVALID).
VALIDI:= (enabled INPUT VALID).
INVALIDI:= (*any* INPUT INVALID).
ENABLED:= (enabled OUTPUT *any*).
DISABLED:= (disabled OUTPUT *any*).
TRI-STATED:= (HIGH-IMPEDANCE OUTPUT *any*).

Voltage levels associated to the states listed below are shown in figure 4.19.

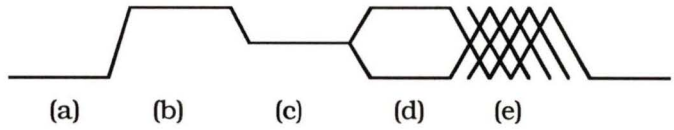


Figure 4.19 Output signal states: (a) asserted (enabled); (b) negated (enabled); (c) tri-stated; (d) unknown (valido, enabled); (e) invalido (enabled).

4.5.2 State Expressions.

The state of a signal is expressed by giving the signal state followed by the signal name. This is called a single state expression. It is convenient to assign to a group of signals certain state. The single state expression allows this by specifying a signal class.

single-state-expression::= signal-state (signal-name | signal-class).

signal-class::= "("signal-name {"," signal-name}$^{+}$")".

For example, the following single state expression expresses that the address lines in a system with an address space of 512 words do not have a valid value.

ADDRESS-LINES:= (A0, A1 A2, A3, A4, A5, A6, A7, A8)

INVALIDO ADDRESS-LINES

The state of a situation may depend on the state of several signals. There is a general state-expression that incorporates the logical operators NOT, OR, and AND. Any boolean function can be described using these operators. The operator AND is true whenever all of its arguments are true. The operator OR is true if at least one of its arguments is true. The NOT operator is true when its argument is false.

state-expression::= single-state-expression |
"(NOT" state-expression ")" |
"(AND" state-expression {"," state-expression}$^{+}$ ")" |
"(OR" state-expression {"," state-expression}$^{+}$ ")".

4.5.3 Transitions.

If a signal does not change state, it conveys no information[3]. When a signal changes state, a transition occurs. The transition of a signal is denoted through a transition expression.

transition-expression::= [state-expression] "!" state-expression signal-name.

When state 1 is the opposite of state 2 in "state-1 ! state-2 signal-name", state 1 can be omitted (i.e., ! ASSERTED READ is equivalent to NEGATED ! ASSERTED READ).

At several instances, one needs to specify set-up and hold times for a group of signals so that they will be stable during the transition of another signal. The state-transition expression is used for this purpose.

state-transition-expression::=

"[" t_{su} "," t_{h} "]" "(#" transition-expression , state-expression ")".

In some cases an action depends on transitions in several signals. An event expression describes transitions in various signals.

---

[3] Similar situation to that of transmitting the same pattern in a communication system.

event-expression::= transition-expression | state-transition-expression | and-event-expression | or-event-expression.

and-event-expression::= "(^" event-expression {"," event-expression}$^{+}$ ")".

or-event-expression::= "(+" event-expression {"," event-expression}$^{+}$ ")".

A more thorough overview about the modelling of signal behavior can be found in [42, 43].

4.5.4 Actions and Signals.

An output signal may encode an action in several ways (see figure 4.20):

- its (a) asserted or (b) negated static level;

- its (c) positive (negated to asserted) or (d) negative (asserted to negated) dynamic transition;

- its duration (e).

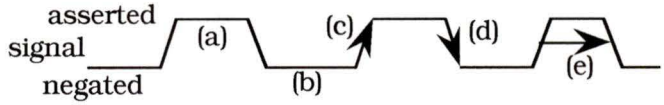


Figure 4.20 Signal encoding of actions.

An action can be written down as a state expression or an event expression. The notation discussed above together with the action graph to be introduced in the next section can encapsulate the temporal information of a protocol. In the subsequent section, the attention is focused on how to describe the sequence of actions in the protocol, once the mapping of these actions in terms of signals has been done.

4.6 Action Graphs.

An extension for the description of protocols using timing diagrams is first introduced. Then the equivalent representation in the action graphs is described.

4.6.1 Protocol Abstraction.

The progression of a digital system in response to changes of external conditions can be represented by an algorithmic state machine [33]. Although algorithmic state

machines can represent synchronous, asynchronous, and mixed systems, timing diagrams are preferred because they can be used not only to describe the functionality of the system but also to display physical and timing constraints, and attracts the attention of the designer to propagation delays, synchronization problems, etc. An algorithmic state machine and its corresponding timing diagram are shown in figures 4.21 and 4.22.

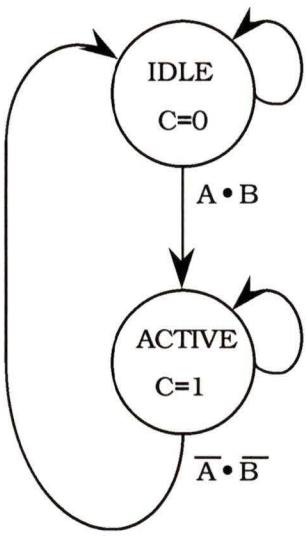


Figure 4.21 An algorithmic state machine.

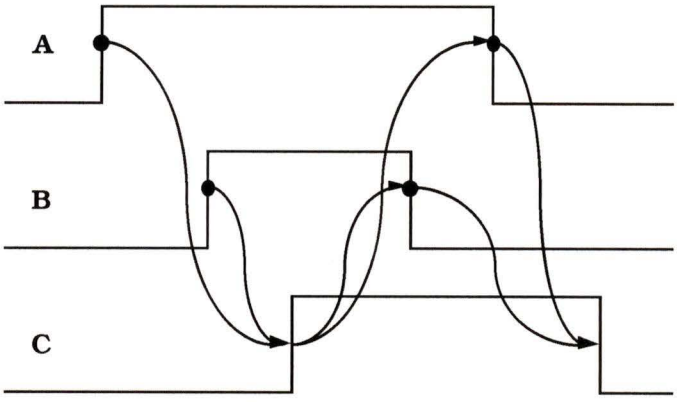


Figure 4.22 Timing diagram corresponding to the algorithmic state machine of figure 4.21.

State transition graphs (STG) [93] have been used to capture the protocol information from timing diagrams. The state transition graph of figure 4.23 is a graph representation of the timing diagram shown in figure 4.22. In this figure the underlined variables denote input signals. An equivalent Petri net of the STG can be drawn by transforming the edges into transitions and the states into links, and adding a place in the middle of each connected transition. The resultant Petri net, shown in figure 4.24, is also called a marked graph [78]. In DAME, action graphs are used to abstract protocols

as discussed in the following.

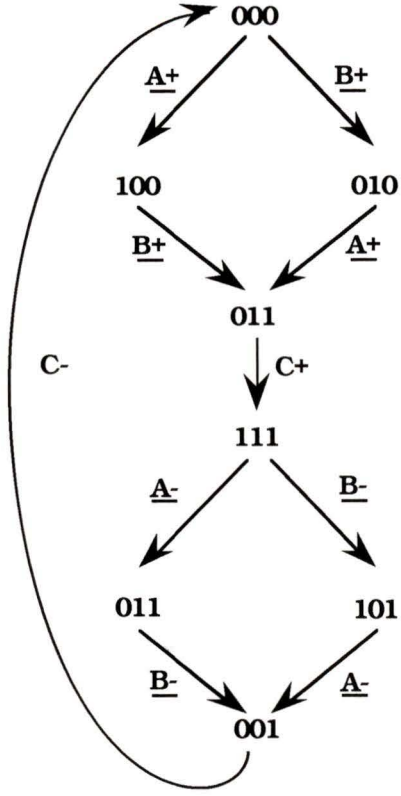


Figure 4.23 State transition graph of the timing diagram in figure 4.22.

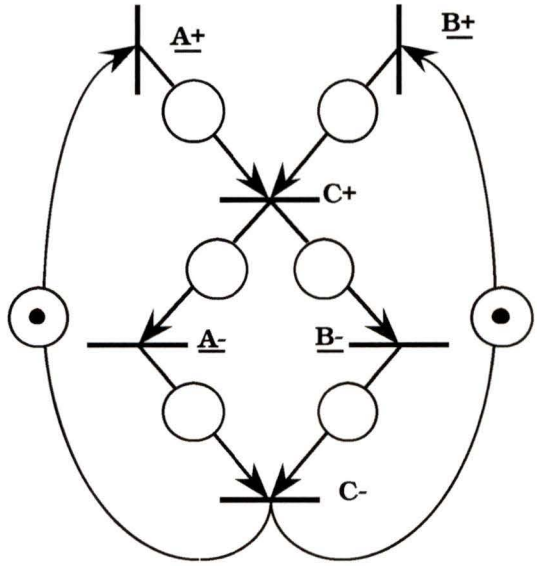


Figure 4.24 Petri net of the STG in figure 4.23.

4.6.2 Action Graph Definition.

Let **A** be the set of actions in the protocol. We define the relation **P** (for precedes) for any two actions a, b ∈ **A**, such that (a, b) ∈ **P** *iff* action b is preceded by action a in the protocol cycle. A directed graph (**A**, **P**, t_p) [56] can be used to represent the protocol, where **A** is the set of vertices of the graph, **P** is the set of edges, and t_p is a function on **P** that associates a label (t_{min}, t_{max}), with the minimum and maximum timing between actions, to the edges. For self-timed circuits, the corresponding label is (0, ∞). In figure 4.25, A and B are actions, (A, B) ∈ **P**, A is the initial vertex, B is the terminal vertex, and (10ns, 100ns) is the label associated with edge (A,B).

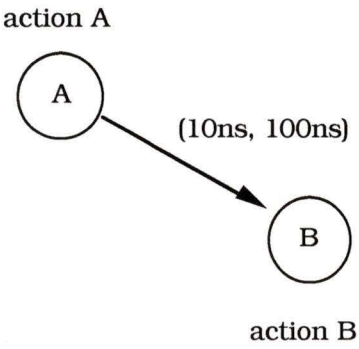


Figure 4.25 Two actions in an action graph.

In the context of the association of an action with an event or a state, the negation of an action is understood to correspond to the negation of the event or the state associated with that particular action. Thus, if an action **a** corresponds to a certain transition of a given signal, the negation of **a**, denoted by $\bar{\mathbf{a}}$, can be described as the opposite transition of the same signal.

4.6.3 Properties of Action Graphs.

Certain properties that action graphs share with directed graphs and in particular Petri nets, are presented in the following.

Strongly Connected Graph.

A directed path is a sequence of edges (e_{i_1}, e_{i_2}, ..., e_{i_k}) such that the terminal vertex of e_{i_j} coincides with the initial vertex of $e_{i_{(j+1)}}$ for $1 \leq j \leq k-1$. A directed path is said to be simple if it does not include the same edge twice. A directed circuit is a directed path (e_{i_1}, e_{i_2}, ..., e_{i_k}) in which the terminal vertex of e_{i_k} coincides with the initial vertex of e_{i_1}. A

directed circuit is said to be simple if it does not include the same edge twice. A directed graph is said to be strongly connected if every pair of vertices is contained in a directed circuit.

Petri Net Equivalence.

An action graph for self-timed events is equivalent to a marked graph [76], if transitions of the Petri net are made to correspond to the vertices in the action graph, and a place is added in the middle of each arc between transitions.

Petri net models have been studied intensively in the area of asynchronous, concurrent systems [76, 69]. There exist several types of Petri nets. For the more general ones, although they have a powerful description capability, the analysis is quite difficult. For the simpler ones, some properties are known, but they have a limited description power. The action graph for asynchronous protocols belongs to a sub-class of Petri nets called marked graphs [76].

A system is called *decision-free* [78] if it has a Petri net model that is decision free. A Petri net is decision-free (also called a *marked graph*) *iff* for each place in the net, there is one input arc and one output arc. For a decision-free graph, the number of tokens in a circuit remains constant after any firing sequence (conservative system).

If an asynchronous system goes back to its initial configuration (state) after each cycle, then that system is live and consistent [78]. In principle it can repeat its operation an unlimited number of times. Otherwise, either the system produces an infinite number of tokens (the number of requests grows unboundedly in bus arbitration) or consumes tokens and eventually stops (no more grants are given out by the arbiter). For the latter, systems are called inconsistent.

A marked graph is live *iff* the directed graph has no token-free directed circuits. A live marking is safe *iff* every edge in the directed graph belongs to a directed circuit with a token count of one [69]. It is easy to check these conditions in the two protocol graphs described below.

4.6.4 Timing Dependencies and Precedence.

In section 4.2, an informal notation was used to express the dependencies between signals in timing diagrams. The precise meaning of the notation will be established in

order to reveal the link with precedence in action graphs.

The following cases of dependencies are considered [23]:

1) A transition in a signal depends on a transition of some other signal(s).

2) A transition in a signal depends on the state of some other signal(s).

3) A transition in a signal depends on a logical combination of transitions and states of other signals.

Dependencies in timing diagrams are denoted by linking the respective signals using arrows with small circles at their *tail*, as shown in figure 4.26. If the dependency is a state dependency, the circle is placed on the level of the signal (figure 4.26a). Similarly if it is a transition dependency, the circle is placed in the corresponding transition (figure 4.26b). Figure 4.26c specifies that the leading transition in E must occur after the trailing transition of signal C while signal D is high.

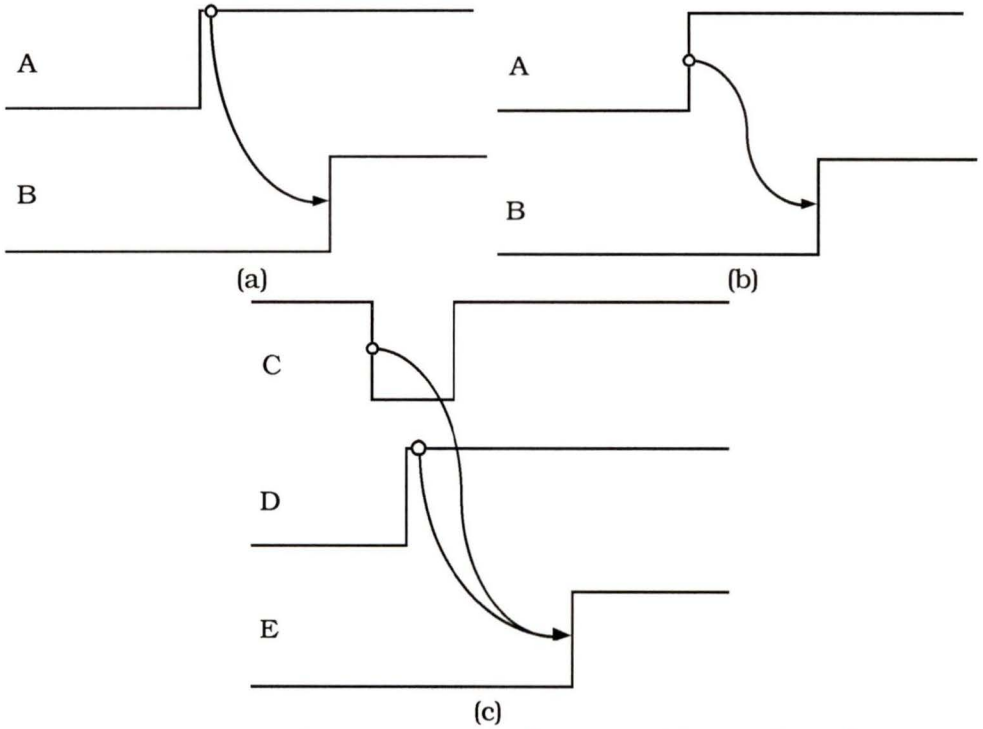


Figure 4.26 Dependencies in timing diagrams: (a) state dependency; (b) transition dependency; (c) logical combinations of a) and b).

The difference between a state dependency and a transition dependency is illustrated in figure 4.26c. Normally there is a delay between the dependency conditions

being met and the resulting transition taking place. In a state dependency, the state condition must be held until the dependent transition occurs, otherwise a race situation may occur. For example, in figure 4.26c, signal D must remain high until E occurs. In a transition dependency, after the transition has happened even if the signal return to the original state immediately (as it occurs with pulses) the resultant transition will be triggered. In the example, signal C is driven low but it is allowed to go high before E changes. Transition detection usually requires a memory device that stores the occurrence of the transition.

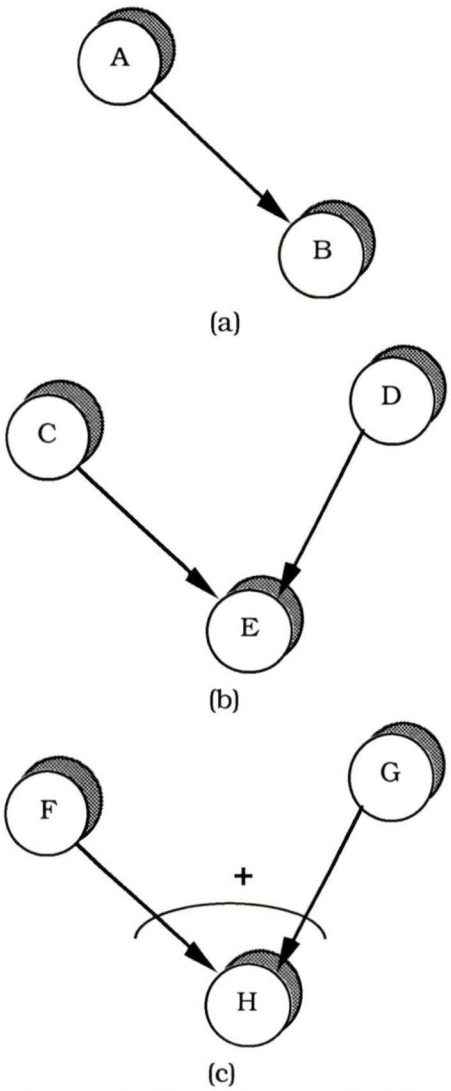


Figure 4.27 Dependencies in the action graph: (a) single dependency; (b) AND dependency; (c) OR dependency.

Dependencies denote precedence in the context of actions. Figure 4.27 shows how the dependencies of figure 4.26 are represented in action graphs. A state or an event can

be associated with an action. Thus the cases illustrated by figures 4.26a, and b, are shown in figure 4.27a. When two incident edges reach a node in the action graph, this corresponds to an AND expression. If an action is activated when at least one of its preceding actions is enabled, a + (OR) symbol is placed nearby an arc enclosing the edges describing the precedence, as shown in figure 4.27c.

4.6.5 Action Graph Construction.

In this subsection, an example of creating an action graph from timing diagrams is presented.

The action graph for the handshake write cycle (see figure 4.8) is shown in figure 4.28. Because actions STROBE and NOT-VALID are encoded in a strobe signal (see figure 4.9), it is possible to rename NOT-VALID as $\overline{\text{STROBE}}$ as shown in figure 4.28. The completion of the cycle is portrayed by the edge from $\overline{\text{ACK}}$ to STROBE, indicating that in order to initiate a new cycle, the previous one must be completed. This is important in modelling conservative systems, as discussed before. There are also two nodes that are attached to the status of the data bus, called DatVal and $\overline{\text{DatVal}}$.

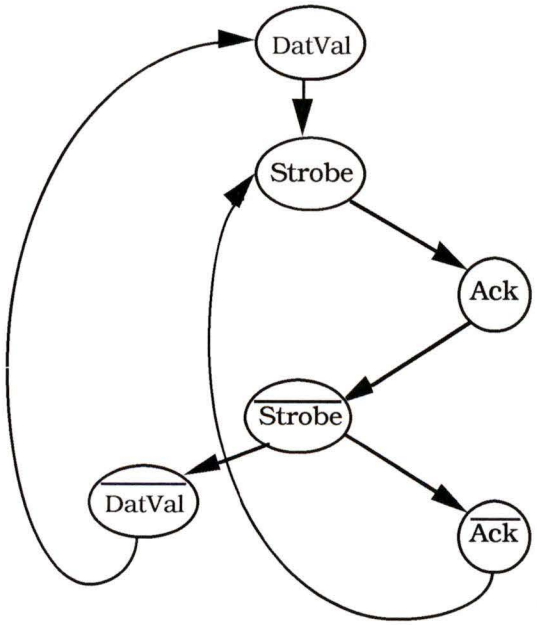


Figure 4.28 Action graph of the handshake write cycle.

4.6.6 Timing Constraints.

In addition to strict precedence relations, which define the protocol's action sequence, timing constraints can also be described by precedence links. Consider the synchronous read protocol of figure 4.16. The corresponding action graph is shown in figure 4.29. The Accept action is the negation of Request, and it is represented as $\overline{\text{Request}}$. DatVal and $\overline{\text{DatVal}}$ actions indicate when the data is being retrieved by the source. Control actions in the source's side are missing as no handshake exists.

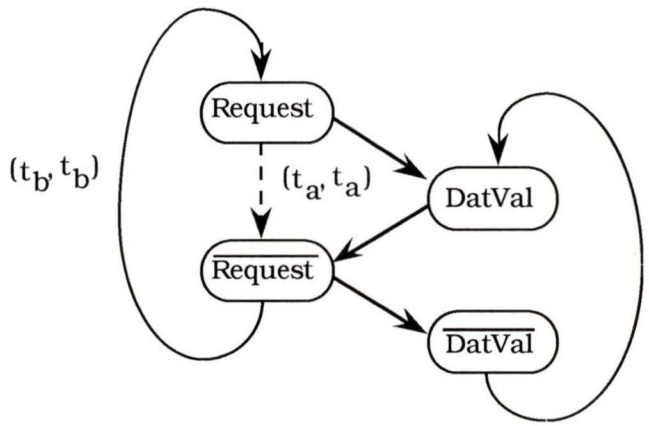


Figure 4.29 Timing constraints in a synchronous read protocol.

The delays t_a and t_b inserted by the synchronous master are denoted by the edges from (Request, $\overline{\text{Request}}$) and ($\overline{\text{Request}}$, Request). Although the precedence indicated by (Request, $\overline{\text{Request}}$) can be inferred from edges (Request, DatVal) and (DatVal, $\overline{\text{Request}}$), and therefore that edge is redundant, to include the t_a label on that edge, rather than assign the label to the other two edges, provides extra constraint information.

It is not until an implementation is worked out that it is possible to make use of the constraint information. Timing constraints do not modify the abstract protocol (because they convey redundant precedence information) but restrict the occurrence of certain actions more directly than a transitive precedence.

The interrelation between the abstract design and the implementation can be sketched as follows: after a design is produced, one implementation is carried out. A functional and timing verification of the resulting hardware is performed. If the implementation does not meet the timing constraints, another technological approach is attempted until a final design is produced or the implementor decides that no implementation is possible within DAME's scope (meaning that the specifications might be too tight for the available incorporated techniques). The constraint sub-graphs of

action graphs, used by the designer system, are not drawn in the subsequent.

4.6.7 Action Graph Example.

In this subsection, another example of creating an action graph from timing diagrams is presented. Figure 4.30 shows the timing diagram of the write cycle in a static RAM memory chip (2114). After the address becomes stable, both CS* and WE* must be driven low to initiate the cycle. Data can be placed on the data bus. Then the WE* strobe is pulsed high to latch the data. Finally the data lines are removed from the data bus and the address lines may change for the next cycle.

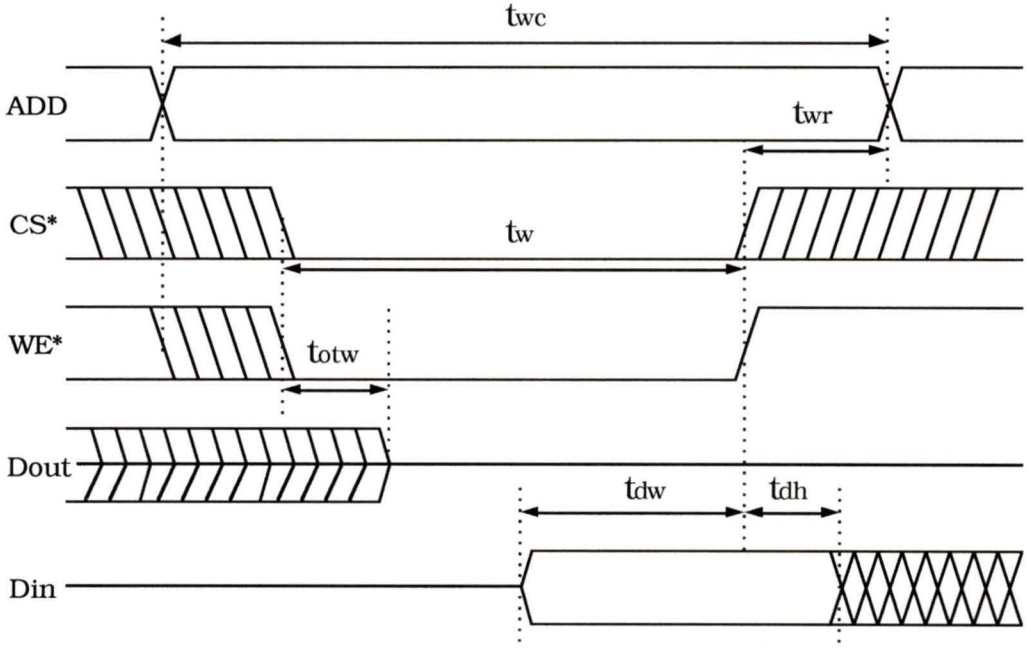


Figure 4.30 Write cycle in the 2114 static RAM memory.

Consider the case in which CS* occurs before WE* (the other case in which WE* occurs before CS* can be drawn in another graph). The corresponding action graph is shown in figure 4.31. The actions are labelled as follows:

Action	Signal
(Select, $\overline{\text{Select}}$)	CS*
(Write, $\overline{\text{Write}}$)	WE*
(Put-Address, $\overline{\text{Put-Address}}$)	ADD
(Data-Valid, $\overline{\text{Data-Valid}}$)	Din/Dout

The continuous edges correspond to the strict precedence relation **P**, while the dashed edges represent timing constraints. It is noted that Select is modelled by a state, while Write corresponds to a transition. In this sense, $\overline{\text{Select}}$ could not happen at all but the protocol will still work properly.

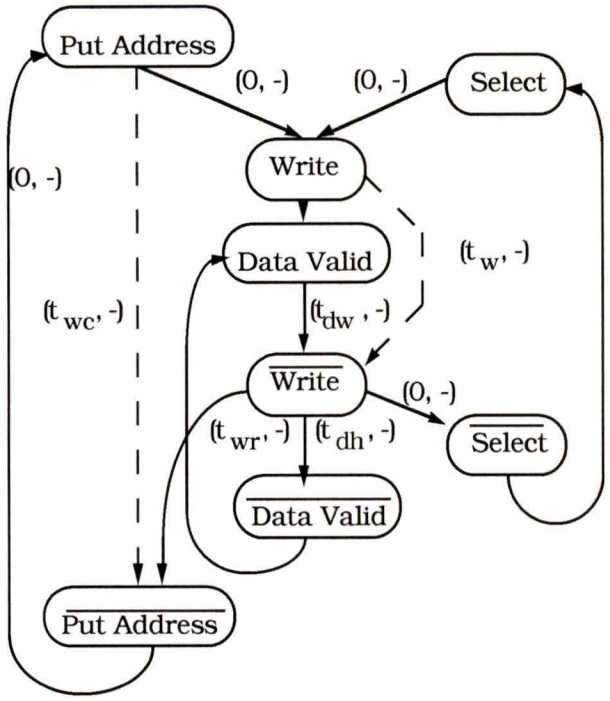


Figure 4.31 Action graph for the timing diagram of figure 4.30.

4.7 Action Graphs for Bus Arbitration Protocols.

In the previous chapter two protocols used in bus arbitration were identified: the two-signal protocol and the three-signal protocol. In this section, their timing diagrams are transformed into action graphs.

The two-signal bus arbitration timing diagram of figure 3.18 can be described by the graph in figure 4.32. The REQ* signal is associated to the action pair $(\mathbf{r}, \overline{\mathbf{r}})$. Action **r** corresponds to ! ASSERTED REQ*, and action $\overline{\mathbf{r}}$ corresponds to ! NEGATED REQ*. Similarly, actions **a** and $\overline{\mathbf{a}}$ are associated to signal ACK*. Actions **b** and $\overline{\mathbf{b}}$ indicate the actual use of the bus by the requestor. The control lines of this protocol follow a fully-responsive handshake, described by the sequence $(\mathbf{r}, \mathbf{a}, \overline{\mathbf{r}}, \overline{\mathbf{a}})$.

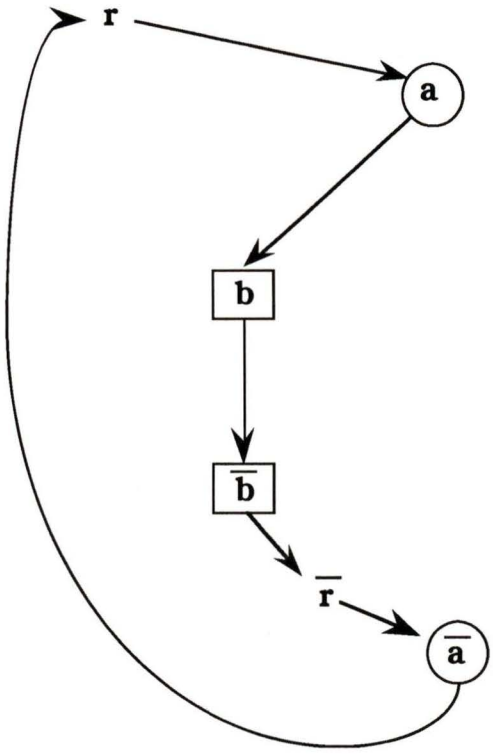


Figure 4.32 Two-signal bus arbitration protocol graph.

The directionality of an action is an intrinsic property. Input control actions monitor events generated by external devices, while output actions inform that the component's part of the protocol has taken place. Accordingly, input control actions are differentiated from output control actions in action graphs by placing circles on them, as shown in figure 4.32. Actions that correspond to the status of the resource being used and do not convey control information, are marked with square blocks.

The three-signal protocol is modelled by the graph shown in figure 4.33. The arbiter's side is described in the action graph. Signals BR*, BG* and BBSY* encode in their transitions the action pairs (**R**, $\overline{\mathbf{R}}$), (**G**, $\overline{\mathbf{G}}$), and (**GA**, $\overline{\mathbf{GA}}$) respectively. The first and last pairs correspond to input actions, while (**G**, $\overline{\mathbf{G}}$) are the output actions.

A request action (**R**) from the requestor eventually will produce a grant (**G**) from the arbiter. The new current master (CM) acknowledges the grant (**GA**) and releases its request (**R**). The new CM can take over the bus (**B**) after the resource is available (**B**). Remember that the last transaction of the previous bus commander may still be taking place at the time the new CM receives the grant. The full handshake between the grant and the grant acknowledge action pairs can be seen in the graph.

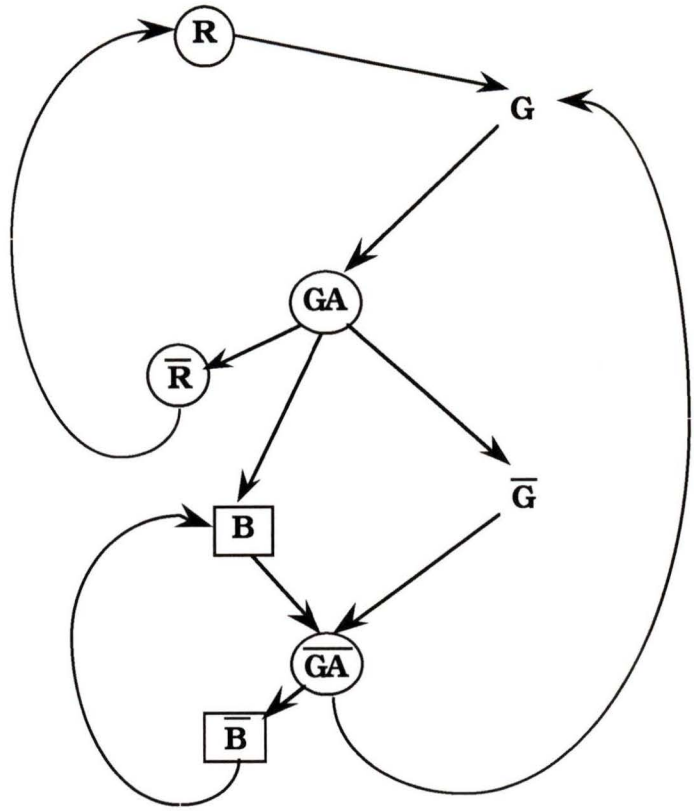


Figure 4.33 Three-signal bus arbitration protocol graph.

In the above action graphs, there must exist at least one simple circuit that contains an action and its complement. This guarantees that the signal associated with a particular action pair will eventually return to its initial state (return-to-zero condition).

The action graphs can also be viewed as data flow graphs. Thus, when an action takes place, a token is placed on all its outgoing edges. An action cannot happen unless all its incident edges have tokens. The initial state is represented by placing tokens on certain edges. The initial marking from figure 3.19 calls for tokens on the edges (**R**, **R**), (**B**, **B**), and (**GA**, **G**).

4.8 Summary.

It is the basic tenet of the DAME project that the interface signals found in various microprocessor families follow a limited number of well defined protocols for information exchange. In this chapter, the basic synchronous and asynchronous protocols were presented in the context of actions. Furthermore, a notation that allows

expressing actions in terms of the interface signals was defined. Finally, action graphs were used to represent the sequence of actions that describes a protocol.

5. Bus Arbitration Designer Subsystem.

5.1 Introduction.

It has been recognized that the role of representation is a fundamental issue in AI [95]. In this chapter, the bus arbitration interface design problem is formulated in terms of merged action graphs. The representation model of the interface within the DAME designer subsystem contains structural and functional information that can be derived naturally from those graphs. The structural model of the interface involves the grouping of action pairs. The functional design of the interface utilizes sequential machines as basic building blocks to implement the functionality required to interconnect the system modules. The semantic network that embodies the component information is manipulated by the designer system to instantiate an abstract design using generic rules, based primarily on the identification of the intervening protocols.

5.2 Bus Arbitration Interface Problem.

In a multi-master bus system, several masters have access to the bus and the allocator solves any conflict caused by contention. Figure 5.1 gives a general picture of the bus arbitration interface design problem. The interface block transforms the protocol used by the requestor that solicits the resource to the one used by the arbiter so that the latter can issue a grant to the unique commander.

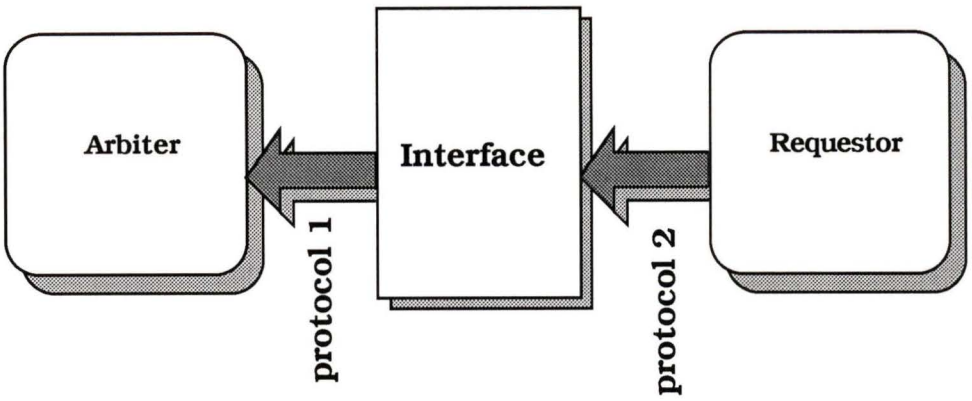


Figure 5.1 Bus arbitration interface between a master and the arbiter.

The designer system in DAME contains the necessary knowledge to construct the interface blocks to interconnect the modules that comprise a microprocessor-based system. In particular, the bus arbitration subsystem in the designer will produce the interface depicted in figure 5.1, using the interfacing information stored in the component library. But before tackling the problem of the representation of such a interface, the design in the abstract space is presented as a transformation on the action graphs that were introduced in the previous chapter.

5.3 Abstract Interface Design.

The interface problem can be solved in the abstract using action graphs. In this section, the general interface design procedure is presented through an example for the bus arbitration capability.

5.3.1 Interface Design Problem.

The interface design procedure involving two protocols can be stated as the problem of finding an action graph that incorporates both protocols in which the causal relations between all the edges are satisfied. In the general case there may be several solutions. This combined graph is called the interface merged action graph, or simply the *merged* graph.

The starting point for the construction of the merged graph are the two complementary protocols that correspond to identical capabilities in the devices to be

interconnected. Two protocols, corresponding to one master and one slave devices, are shown in figure 5.2. Control *output* actions are the actions generated by the device, while *input* actions are generated externally. In the protocols described in figure 5.2, actions associated to the information path are enclosed by squares, input actions are encircled, and output actions are left unmarked.

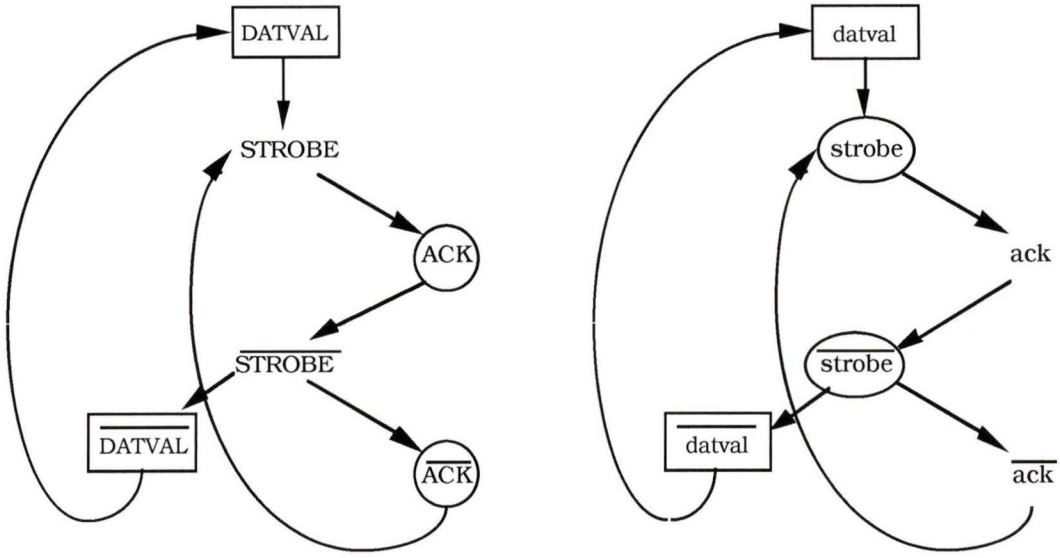


Figure 5.2 Two complementary protocols for the same capability in two devices that are to be interconnected.

Specific operations are attached to the protocol actions. For instance, the Strobe action signifies that a group of lines are valid. *Matching* actions, that correspond to the same type of operation in both protocols, are then identified. In figure 5.2, matching actions in both protocols have the same name. In order for the protocols to be complementary, matching control actions must have opposite directionality[1]. That constrains the matching action of an output action to be an input action.

Paths that connect an output action to its corresponding matching input action must be established. This can be done either by placing a direct link between matching actions (from output to input), or by ensuring that a path exist between them in the merged graph. Also, the precedence between the data path actions and the control actions must be satisfied. By correctly using the information provided by the original graphs, the correct precedence of actions is guaranteed. Figure 5.3 presents the resulting merged

---

[1] For example, it is not possible to have two protocols that both assert the same Strobe action at the same time. An example taken from bus arbitration is the meaningless situation of having two protocols, both of them awarding the bus.

graph for the simple example discussed above.

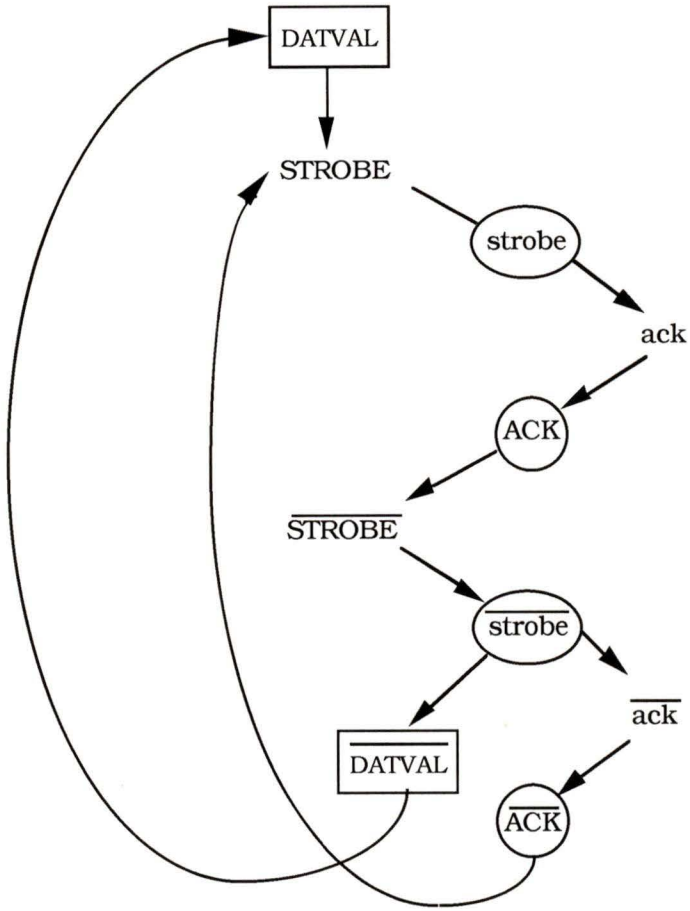
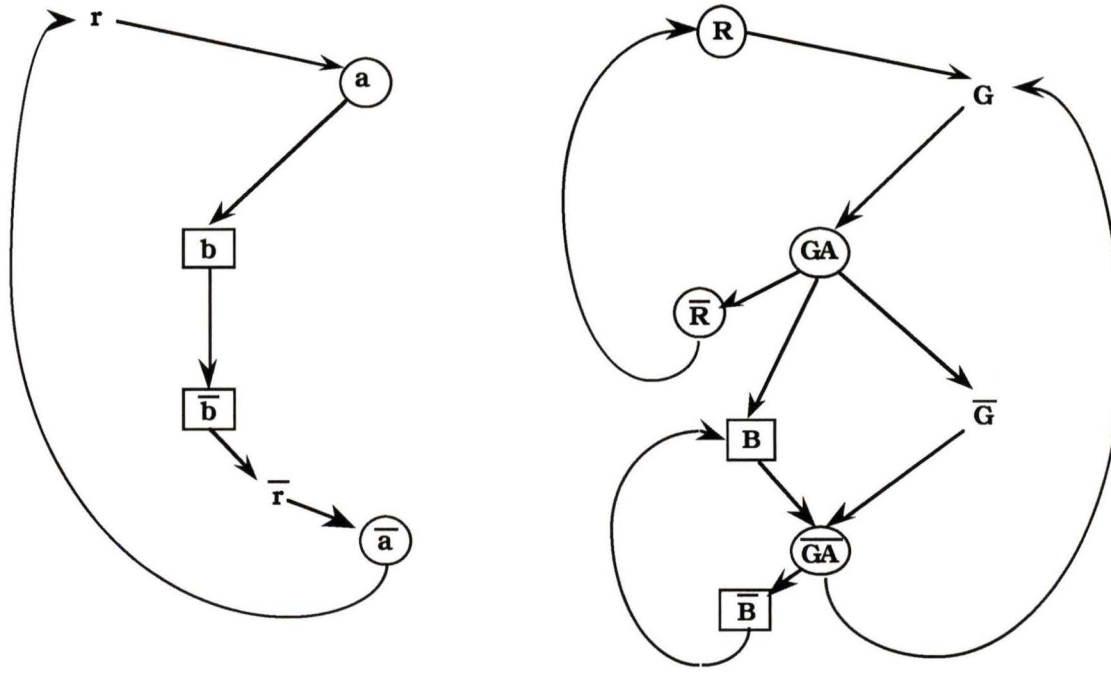


Figure 5.3 Merged graph that describes the interconnection of the protocols shown in figure 5.2.

The construction procedure for the merged graph outlined above is used to produce the interface of figure 5.1, involving the protocols for bus arbitration identified in chapter 3. Not only does this example illustrate a widely used interface, but it also presents the basic ideas in a non-trivial situation.

In bus arbitration, complementary roles for the devices are arbiter and master (or requestor). Figure 5.4 shows two action graphs corresponding to a requestor using a two-signal protocol in its bus arbitration capability, and an arbiter following a three-signal protocol. Actions in the two-signal protocol are labelled using lowercase characters, and capitals are used for the three-signal protocol. The square blocks mark the actions related with the use of the bus.



(a) (b)

Figure 5.4 Bus arbitration protocols for (a) a requestor (two-signal protocol), and (b) an arbiter (three-signal protocol).

Figure 5.5 shows a merged graph that is a solution of the interface of the protocols described in figure 5.4. The steps to construct this merged graph are sketched below. It is the responsibility of the interface to generate the encircled actions (**R**, $\overline{\mathbf{R}}$), (**GA**, $\overline{\mathbf{GA}}$), and (**a**, $\overline{\mathbf{a}}$) using the other actions[2].

The actions of the two original protocol graphs share three basic functions: request, to solicit the resource; grant, to award the bus; and the use of the resource. In the requestor's side, these functions correspond to the action pairs (**r**, $\overline{\mathbf{r}}$), (**a**, $\overline{\mathbf{a}}$) and (**b**, $\overline{\mathbf{b}}$), respectively. In the arbiter's protocol, the corresponding pairs of actions are (**R**, $\overline{\mathbf{R}}$) for the request function, (**G**, $\overline{\mathbf{G}}$) and (**GA**, $\overline{\mathbf{GA}}$) for the grant function, and (**B**, $\overline{\mathbf{B}}$) for the use of the bus.

[2] It is possible to generate an encircled action from other encircled action, as far as no simple circuit with only inputs actions is created in the merged graph, i.e., an input action A that is generated from and generates another input action B.

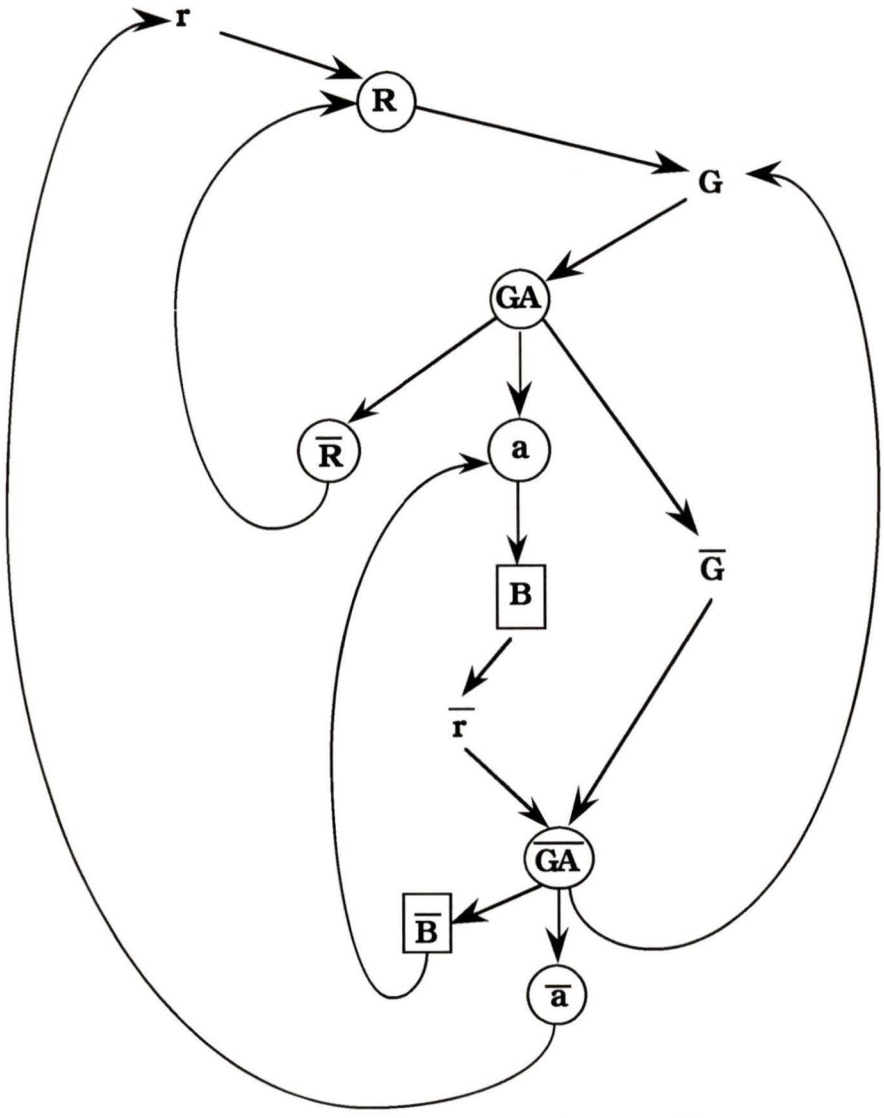


Figure 5.5 Merged graph of a two-signal protocol (requestor) with a three-signal protocol (arbiter).

The appropriate edges between *matching* actions in the protocol are created. For instance, starting with action **r**, an edge is created from **r** to input action **R**. This input action is one condition for **G** to occur. One solution places an edge between **G** and input action **a**. The solution presented in figure 5.5 keeps the direct precedence between **G** and **GA**, as shown in the three-signal protocol in figure 5.4, and creates an edge (**GA**, **a**). In both cases the precedence required by the two-signal protocol is fulfilled. Different solutions are the result of the alternatives for the paths between corresponding actions.

By following the above procedure for all the matching actions, the merged graph

of figure 5.5 can be produced. The only edge that may not seem obvious at first glance is edge (**B**, **a**). The reader must convince her/himself that the acknowledge to the requestor should not take place unless the global bus action indicates the resource is available. If all the precedences in the original protocols are satisfied, it is possible to abstract both protocols from the merged graph. Note that the action pair (**b**, $\overline{\mathbf{b}}$) is not shown in figure 5.5 as it is discussed in the following paragraphs.

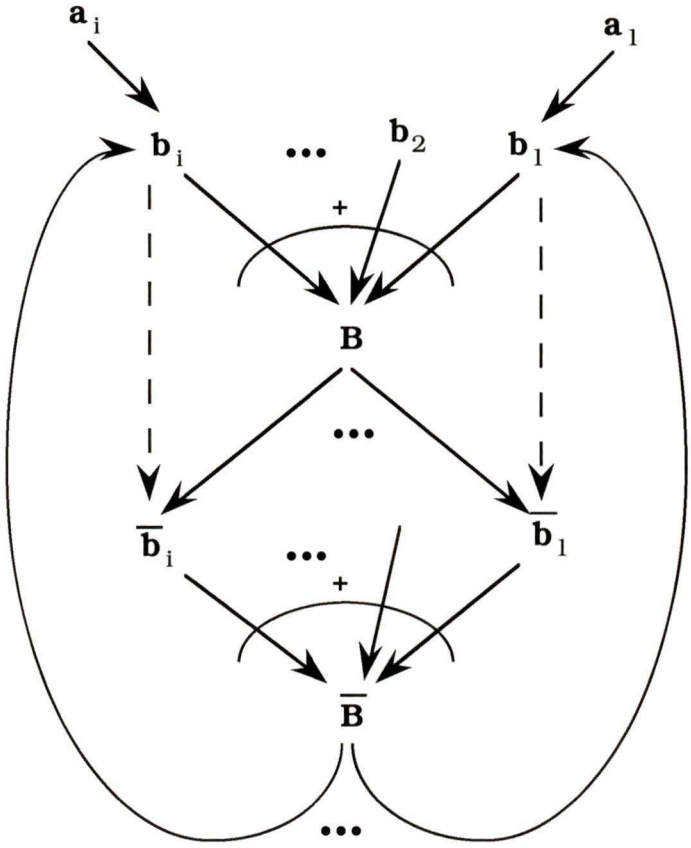


Figure 5.6 Action sub-graph for the use of the bus by the requestors.

Use of the bus.

Figure 5.6 shows the relation between ($\mathbf{b}_i$, $\overline{\mathbf{b}}_i$) and (**B**, $\overline{\mathbf{B}}$), the former pair of actions corresponding to the use by the potential master i, and the latter corresponding to the global bus resource. In figure 5.5, the use of the bus by the requestor has been omitted not only for clarity but also because the bus is a unique resource and thus actions ($\mathbf{b}_i$, $\overline{\mathbf{b}}_i$) must be replaced by the global actions (**B**, $\overline{\mathbf{B}}$). In the case of bus arbitration this is particularly important because the status of the global resource visible to all the masters is indicated by the action pair (**B**, $\overline{\mathbf{B}}$) and not by the individual action pairs ($\mathbf{b}_i$, $\overline{\mathbf{b}}_i$).

It can be seen that when any of the requestors takes over the bus, the resource is marked as busy, because of the ORing of the incident edges into **B**. Similarly after a requestor gives up the bus (action $\bar{\mathbf{b}}_i$ takes place), the bus becomes available to the other masters (**B**).

The mutually exclusive function of the arbiter that ensures that only one requestor will take over the bus at any given time can be modelled using mutual exclusion blocks as shown in the next section. In this section, it is sufficient to consider the arbitration process as a black box that receives the requests from several potential commanders and awards the bus to only one of them, as shown in figure 5.7.

Figure 5.7 Black box representing the mutual exclusive function of the arbitration procedure.

This function is implicitly symbolized by the (**R**, **G**) edge. The grant action $\mathbf{a}_i$ in figure 5.6 emphasizes that only one of the possible requestors shall be granted the bus, and only one of the ORed $\mathbf{b}_i$ actions will be active at any given time. The precedence link (dashed lines in figure 5.6) from $\mathbf{b}_i$ to $\bar{\mathbf{b}}_i$ signifies that the current master will eventually release the resource.

5.4 Interface Model.

In this section, the interface model in the DAME designer subsystem that contains the structural and functional information of the designed interface is presented. The structural model comprises the static information about the actions, and it is obtained from the nodes of the merged graph shown in figure 5.8. The functional model includes information about the generation of the input (encircled) actions in the merged graph. The functional model bases its description on a particular form of sequential machines. Other blocks complete the design.

5.4.1 Structural Model.

Several structures exist for bus allocation, as described in chapter 3. Mainly, the basic mechanisms used in bus allocation are: token-passing, arbitration, and collision detection. According to the structure, the allocation can be centralized or distributed. Furthermore, there exist several algorithms to perform the selection required by the arbitration mechanism, such as fixed priority, round-robin, etc.

Throughout this work, a widely used technique for distributed arbitration is considered: the daisy-chain structure. One configuration of a multi-master system in a daisy-chained fashion includes an arbiter that generates the unique grant, and several requestors that pass the grant through a daisy-chain. The closest requestor to the arbiter has the highest priority and it can take the grant before any of the other requestors. A block diagram of the interface is depicted in figure 5.8. The arbiter follows a three-signal protocol. Although in the general case, any valid protocol can be attached to the arbiter, this particular case is used in the subsequent. Requestors may use a two-signal protocol so that an interface for protocol conversion is necessitated.

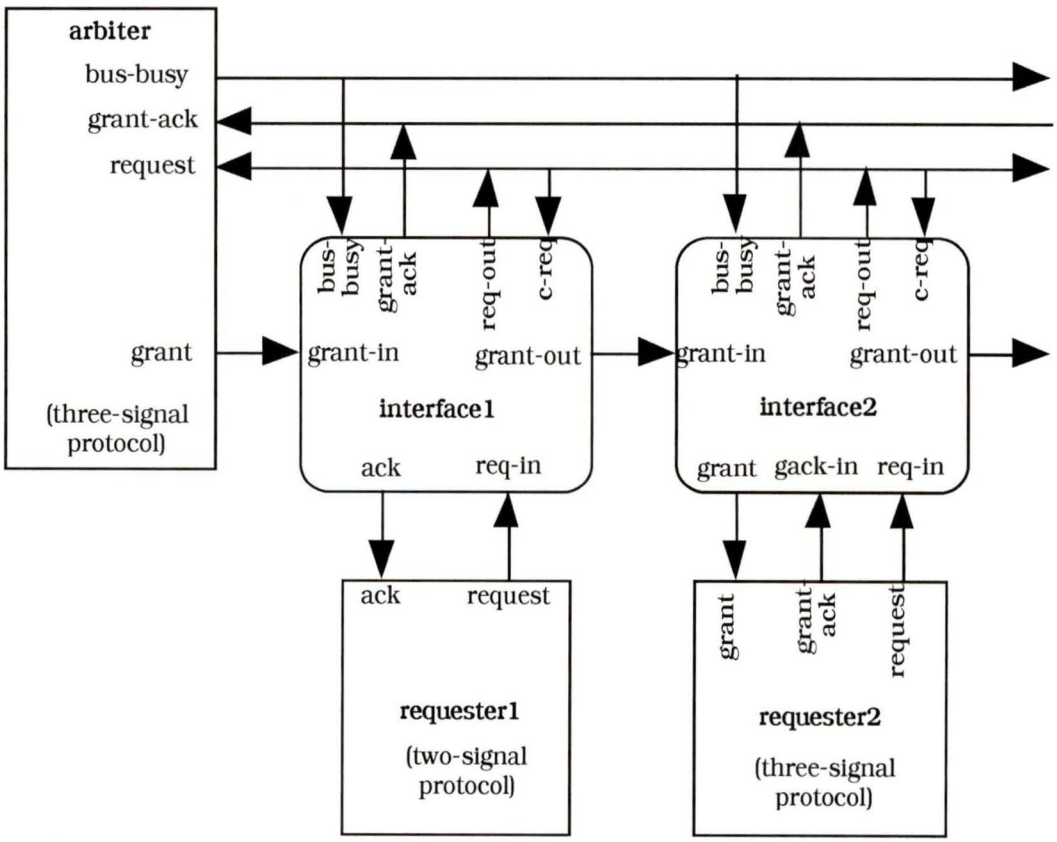


Figure 5.8 Bus arbitration model for a multi-master system with a daisy-chain scheme.

Interface 1 shown in figure 5.8 captures the structural information of the merged protocol graph of figure 5.5. An action and its complement are encoded within the interface as a single line. For instance, the action pairs of the two-signal protocol (**r**, $\bar{\mathbf{r}}$) and (**a**, $\bar{\mathbf{a}}$) are represented by signals **req-in** and **ack** respectively. Similarly for the three-signal sub-graph, the action pairs (**R**, $\bar{\mathbf{R}}$), (**GA**, $\overline{\mathbf{GA}}$) and (**B**, $\bar{\mathbf{B}}$) are represented by **req-out**, **grant-ack** and **bus-busy** respectively. The grant pair (**G**, $\bar{\mathbf{G}}$) is related to the daisy-chain structure represented in the structural model by the **grant-in** and **grant-out** signals, as explained in the following subsection.

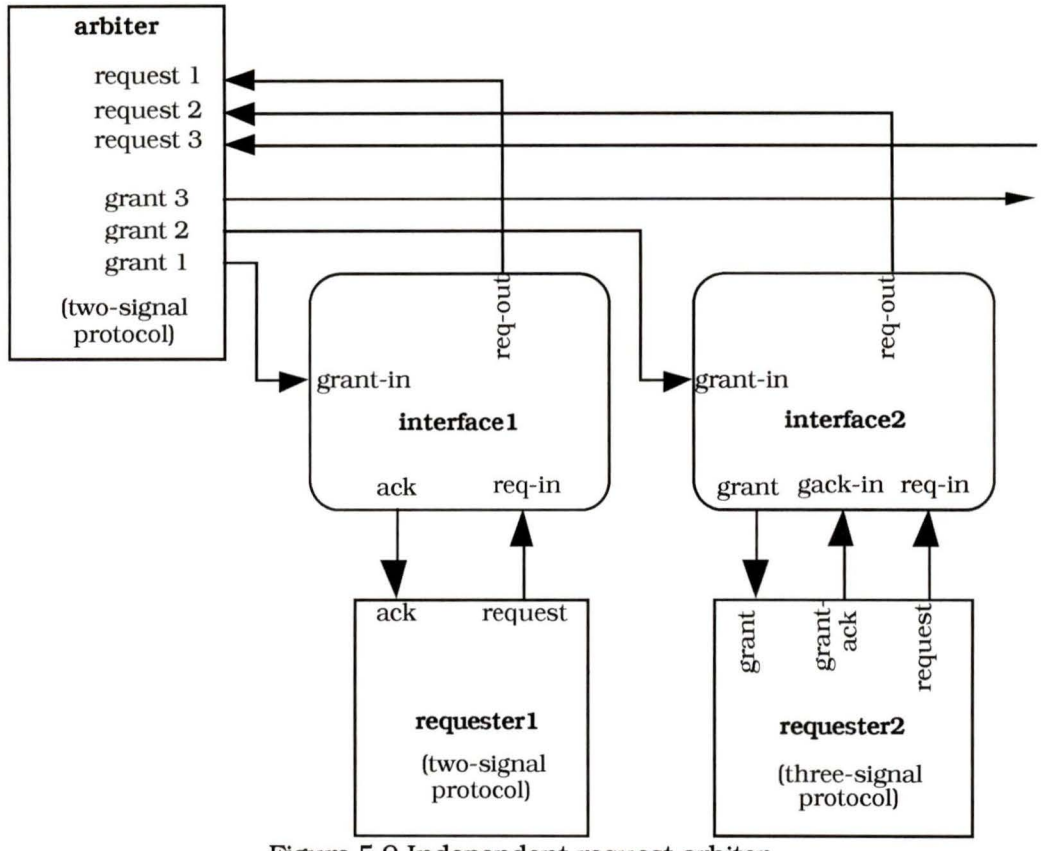


Figure 5.9 Independent request arbiter.

A bare daisy-chain scheme of distributed arbitration has the problem of live-lock, in which the lower priority devices can starve because of the possibility that the higher priority devices take turn in keeping the resource busy ad infinitum. A fair design has been incorporated in the daisy-chain interface as described in [23]. The fairness implementation is described below. According to this fairness scheme, a requestor that has just released the bus cannot start another request until all pending requests have been served. For this purpose the interface includes an input line **c-req** to monitor the

requests from other devices (see figure 5.8).

For the case of an independent-request arbitration scheme, the arbiter has separate request and grant lines for each requestor, as shown in figure 5.9. It is noted that the same merged graph of figure 5.5 applies for the design of the signals corresponding to the control actions of the protocol, the only difference being the absence of the daisy-chain structure described below. In the subsequent the internal structure of the interface is addressed.

5.4.2 Daisy Chain.

The arbitration process belongs to the general exclusive access problem for resource allocation, the allocated resource being the bus. The selection of the current master in a daisy-chained fashion corresponds to a distributed arbiter structure. Therefore the arbitration takes place in the interfaces. A model of the daisy-chain structure that uses the mutual exclusion blocks described in chapter 3 is shown in figure 5.10. Basically, the grant given out by the arbiter, after it detects at least one of the masters' request action, is received by the requestor's interface through the **grant-in** input. The interface then has to decide if it will take the token or pass it through the daisy chain. The decision is based upon if a request from its master had been received (in **req-in**) before the grant reached the interface. If that is the case, the **grant** is directed to the master using **grant**, otherwise the grant is propagated down the daisy-chain to the other masters through **grant-out**.

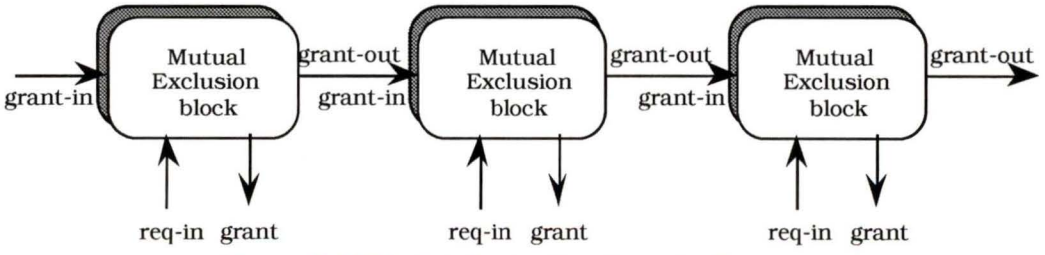


Figure 5.10 Daisy chain distributed allocator.

The actions captured by **grant-in** and **req-in** are independent and they may happen simultaneously. The mutual exclusion circuit guarantees that even in this situation, only one of its outputs will be activated at any time[3]. Figure 5.11 shows the timing behavior of the mutual exclusion block, a two-input/two-output device.

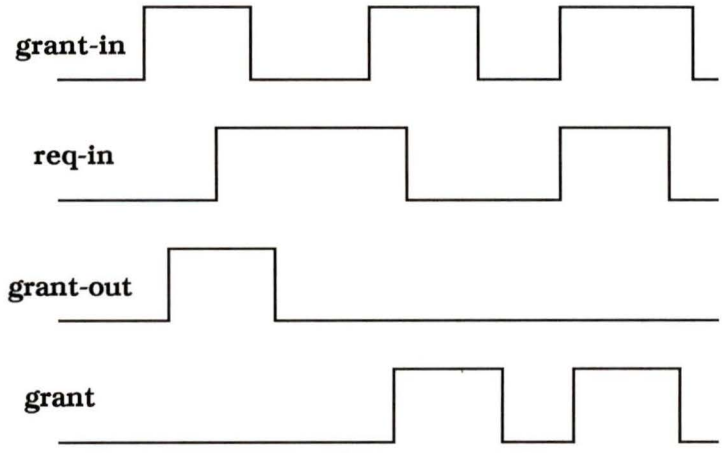


Figure 5.11 Timing behavior of the Mutual Exclusion element.

An implementation based on the design of a sequential state machine is presented in Appendix D. The mutual exclusion element allows us to isolate the synchronization problem inherent in concurrent systems.

5.4.3 Asynchronous State Machine.

The asynchronous state machine (ASM) described in this section is a basic building block used in the interface design. The Mealy machine representation of the two-state ASM in figure 5.12 describes the following behavior: a change in state from 0 to 1 occurs when the input event $!I_1$ takes place, resulting in an output event $!O_1$. While in state 1, an event $!I_0$ will reset the state machine to the initial state 0, causing an event output $!O_0$.

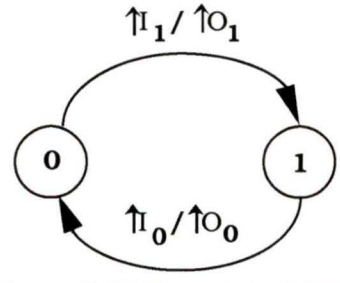


Figure 5.12 Two-state ASM.

In the timing diagram of figure 5.13, it is shown that this ASM can be implemented

---

[3] It is stressed that the decision of which output, either **grant** or **grant-out**, should be active when the inputs occur simultaneously is not important, but the fact that only one becomes active.

as a two-input/one-output device, in which each of the input signals represents one event, and the output encodes the two outputs events. The input signals corresponding to events $!I_1$ and $!I_0$ are called the set and reset inputs, respectively.

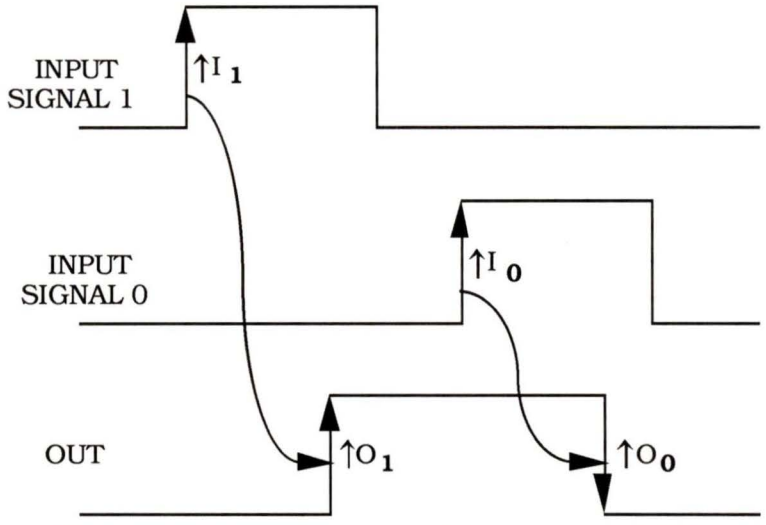


Figure 5.13 Timing behavior of the two-state ASM.

Suppose that $!O_1$ and $!O_0$ represent a pair of actions $(\mathbf{a}, \overline{\mathbf{a}})$. If $!I_1$ and $!I_0$ indicate the moment in which all the actions that must precede **a** and $\overline{\mathbf{a}}$, respectively, have occurred, the two-state ASM can be used to generate $(\mathbf{a}, \overline{\mathbf{a}})$. In this manner, the DAME designer subsystem is able to generate the encircled actions in figure 5.5 using two-state ASMs.

The design of an implementation for the described state machine is carried out in Appendix D. Other blocks required for this design are event detectors, that transform input events into signals shown in figure 5.13, as discussed below.

5.4.4 Event Detectors.

The general two-state sequential machine described in the previous subsection accepts two events as inputs and produces two event outputs. Actual implementations deal with signals. Event detectors are the blocks that translate events into signals.

An event detector is a device that monitors transitions and states of some signals as described by an event expression. When the event expression is satisfied, the output of the event detector becomes active. Because only the positive transition is relevant, the output of this event detector can be a pulse.

A general event detector must be able to handle AND, OR and NOT combinations of transitions and states. Figure 5.14 shows the block diagram for the general event detector.

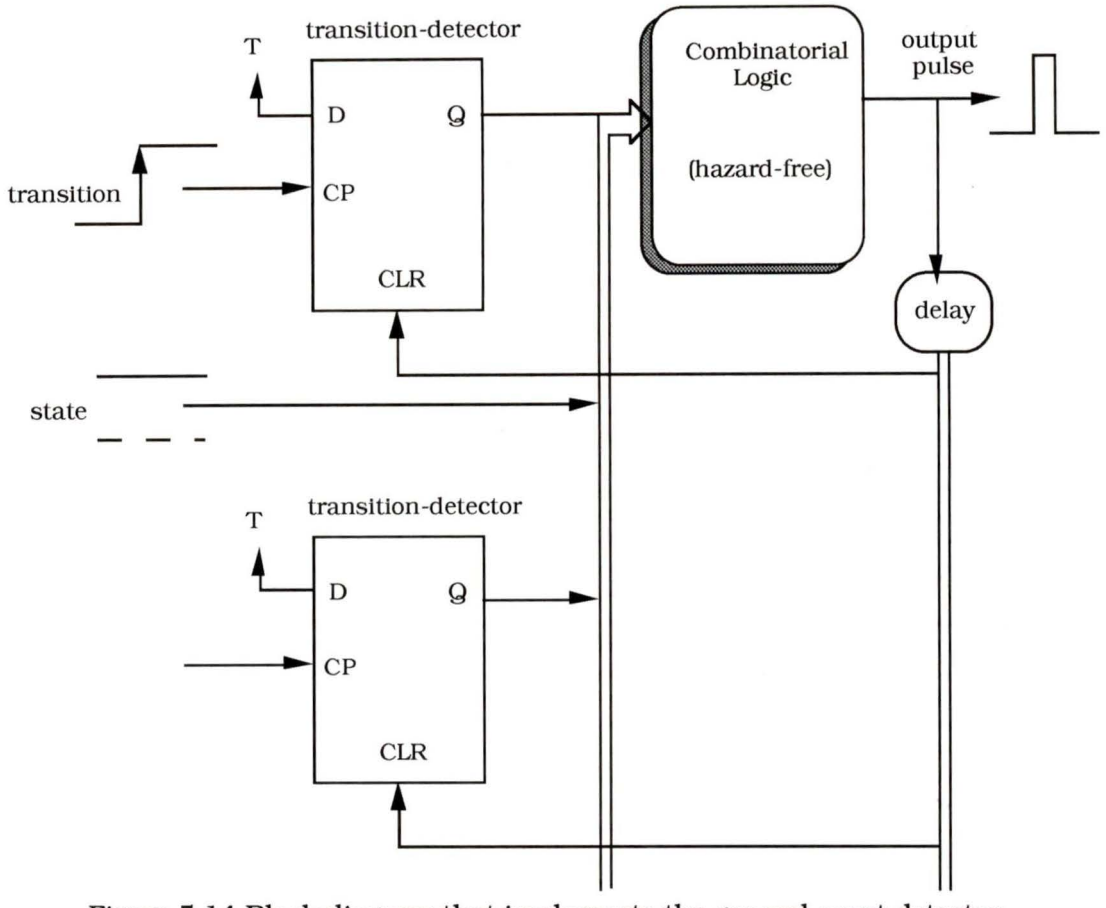


Figure 5.14 Block diagram that implements the general event detector.

5.4.5 Functional Design.

The sequential machine described in subsection 5.4.3 can be used to generate the outputs of the interface that correspond to the encircled actions in the merged graph shown in figure 5.5. In particular, a two-state machine is used because the actions in the protocol are cluster in pairs.

Mutual exclusion blocks, two-state sequential machines, and other blocks are the basic blocks used to completely specify the interface functional description. One mutual exclusion block is included to implement the arbitration. Three sequential machines produce the outputs **req-out**, **grant-ack**, and **ack** of interface 1 in figure 5.8.

Figure 5.15 describes such sequential machines.

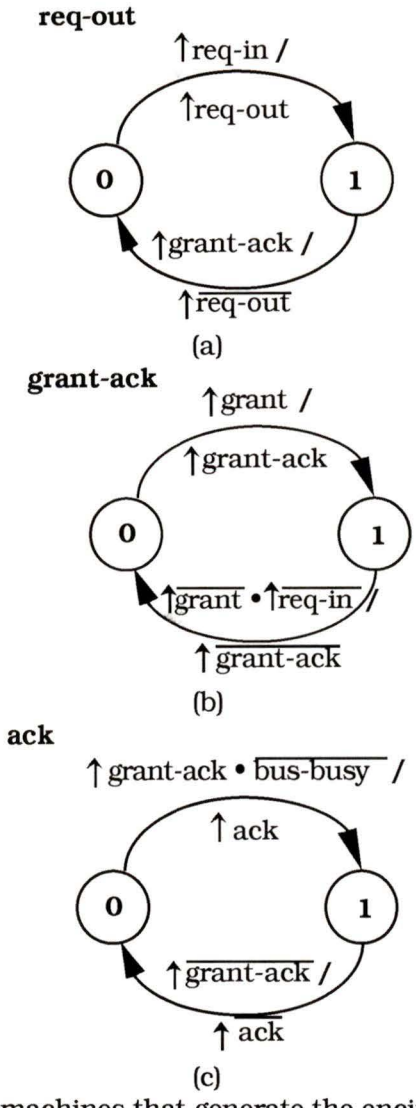


Figure 5.15 The sequential machines that generate the encircled actions of the merged protocol in interface 1.

The output events in a sequential machine are associated to the pair of actions this ASM is generating. For example, the **req-out** ASM produces the action pair (**R**, **R**), represented by signal **req-out** within the interface (see figure 5.8).

$!O_1$:= ! ASSERTED **req-out**

$!O_0$:= ! NEGATED **req-out**

The input events $!I_1$ and $!I_0$ contain all the dependencies of the corresponding

pair of actions. For instance, the **ack** ASM, that generates the pair of actions (**a**, $\overline{\mathbf{a}}$), contains the following input events, indicating that action **a** must be preceded by **GA** (**grant-ack**) and **B** ($\overline{\textbf{bus-busy}}$), and action $\overline{\mathbf{a}}$ must be preceded by $\overline{\mathbf{GA}}$.

$!I_1$:= (AND (! ASSERTED **grant-ack**), (NEGATED **bus-busy**))

$!I_0$:= ! NEGATED **grant-ack**

5.4.6 Interface Block Diagram.

The proposed interface that implements the merged action graph is shown in figure 5.16. The action pairs that are inputs to the interface are drawn at the left, while the generated actions are at the right. The blocks that conform the interface are the sequential machines (ASM), the mutual exclusion block (ME) and event detectors in two flavors, single event detectors (D) and AND event detectors (^D), as explained in previous subsections. The small circles represent inverters.

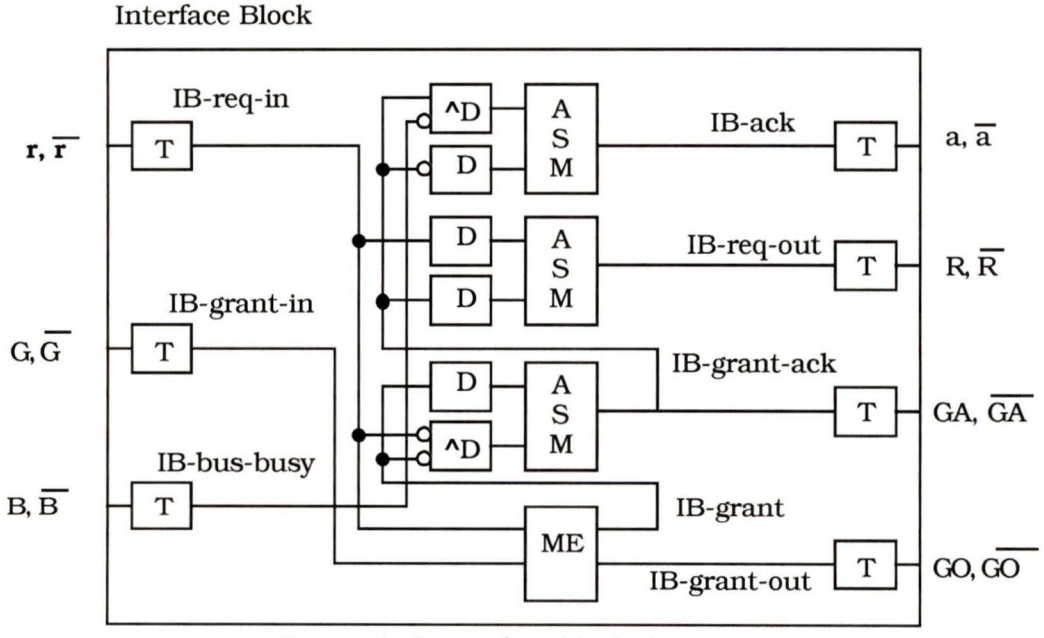


Figure 5.16 Interface block diagram.

T blocks map an action pair into a single signal. Usually it is the case that a pair of actions are encoded in the devices as opposite transitions of a single signal. In this case, T blocks reduce to connect blocks. A connect block is a one-signal-to-one-signal block, either a buffer, an inverter, an open-collector gate, or even a wire, depending on

the technology, polarity, fan-out, etc., of the signals. The flavor of the connect block is determined at the implementation stage. In the general case, in which both actions correspond to independent events, two-state ASM, as described in 5.4.3, can be used as T blocks.

5.4.7 Fairness Design.

As described in the previous chapter, masking is one possible solution to the problem of starvation in a daisy-chain structure. This technique is equivalent to a round-robin scheme. Basically this technique inhibits further requests from the interface's requestor until no other requests are in progress in the bus. In order to be able to support this policy, the interface must know if at least one request is still pending. This information can be retrieved from the common request line **c-req**, that is the OR of the requests from all the interfaces (usually implemented as a wired-OR). After a requestor has used and released the bus, its further requests are frozen while **c-req** is active. Only when the last pending module has been served, **req-out** is enabled again.

A two-state ASM can be used to describe the fairness bit as shown in figure 5.17. The state of this *fairness* bit is a pre-condition for **req-out** to become active. The modified **req-out** sequential machine is shown in figure 5.18. The initial state for the fairness bit is asserted, and only becomes negated after the leading edge of **req-out**. The negative transition of the ORed **c-req** signal drives the bit back to its original state.

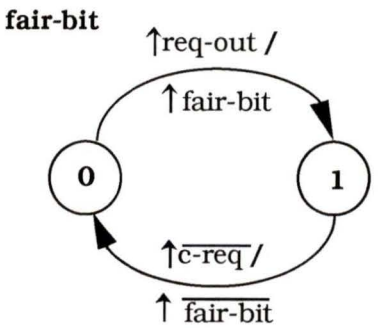


Figure 5.17 Fairness bit described by a two-state ASM.

The fair design shown in this subsection is an example of the descriptive power of the two-state ASM and of the simplicity to incorporate other blocks in the designer subsystem.

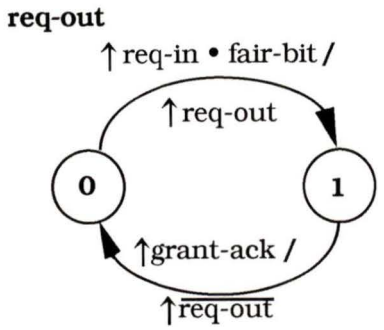


Figure 5.18 Modified req-out ASM in a fair design.

5.5 Data Representation.

The interface representation within the designer subsystem, covered in the preceding section, was obtained from an abstract design represented by the interface merged action graph. In actual systems, interface signals encode the actions found in the protocols. Instantiated protocols, where the actions are mapped to the particular event/state expressions according to the component used, are included in the component library in DAME. The DAME designer system uses these instantiations of the protocols to produce a tailored design. The problem of the component description is addressed in this section, concentrating on the bus arbitration case.

5.5.1 Component Representation.

Components incorporate both structural and function specifications. Typical structure information include the name of the component, number and name of pins, package, etc. Function specifications relate to the behavior of its interface signals, and it is described by the protocols.

Components are represented in DAME as networks of schemata called *semantic networks* ([88]. These networks are in the form of a tree that incorporates in sub-trees the description of the participating protocols. Figure 5.19 shows a partial semantic network describing the MC68000 microprocessor with emphasis on the bus arbitration sub-tree.

Three levels of the hierarchy are represented in figure 5.19:

Module Level, that organizes all the available modules that exist in a microprocessor system into different categories. The classes of modules defined in DAME are: CPU (microprocessors), memory devices, I/O devices, buses, etc.

Capability Level, that describes the interfacing capabilities the device uses to communicate with other system modules. Standard capabilities associated with CPU chips are Data Transfer, Interrupt, Bus Arbitration, and Utility (that includes such signals as clock and reset). Devices have at least a Data Transfer capability (i.e., memory devices). I/O devices may have Interrupt as one of their capabilities. Multi-master buses comprise all the four mentioned capabilities.

Protocol level, that defines the protocol the device utilizes to exercise the capability. This level captures the functional information of the interfacing capabilities of the devices. As a matter of fact, the interface design is based on the information contained at this level. However, it is the semantic network as a whole that maintains the component database.

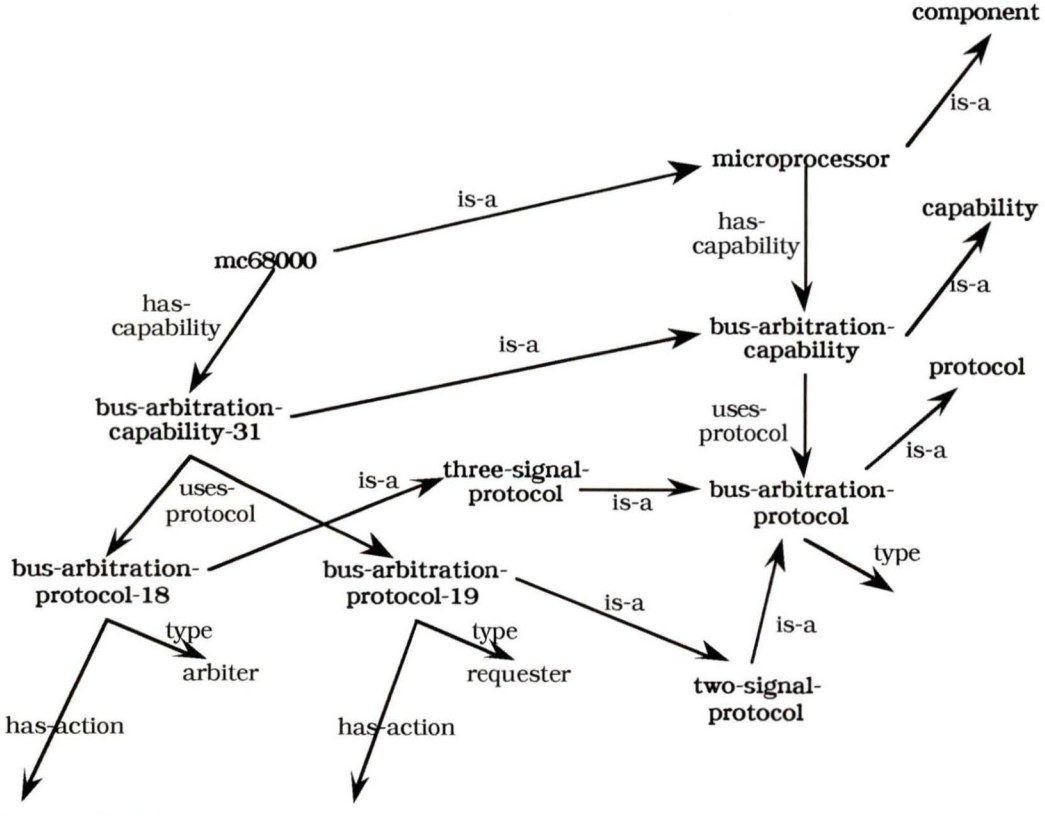


Figure 5.19 Partial semantic network describing the MC68000 microprocessor.

In figure 5.19, the bus arbitration capability comprises both a requestor and a responder protocols, expressing that the MC68000 can be used as a default-master/arbiter and as independent requestor. These protocols are related through the IS-A relation to the protocol templates that define them.

There are two features of semantic networks that are essential to the organization of the component library in DAME: inheritance and information hiding.

Inheritance.

Each node in the semantic network is an object. Objects are linked to each other through relations. The IS-A relation is the means of providing inheritance in our semantic network. For instance, all the microprocessor components possess common characteristics, that are described in the **microprocessor** object. The MC68000 inherits those properties through the IS-A relation. Inheritance can be exploited so that it is not necessary to duplicate information in each object. This is particularly useful when the data base contains large amounts of information.

A template in DAME encapsulates the abstract features of an object. In this sense, the **microprocessor** object is the template for the MC68000. A microprocessor has capabilities that are described using protocols. Accordingly, the MC68000 replicates the microprocessor sub-tree in its structure, but adding specific information pertaining only to itself. Similarly, there exist but a limited number of protocols, and the protocol sub-trees incorporated in the semantic network of the component are instantiations of these well defined protocols.

Information Hiding.

Objects are structured hierarchically in such a way that the traversing of the tree through different relations brings up more detail about the ancestor nodes. At the top level, only components are visible. When the HAS-CAPABILITY relation is considered, schemata that describe the interfacing capabilities of the components become apparent. In order to get the functional specification of a particular capability, contained in the associated the protocol, the USES-PROTOCOL relation must be transversed.

At different stages of the design, not all the information is relevant. For example, when deciding about the overall structure of the system, it is not important to know about the capabilities of the components. Hiding of information is achieved by allowing

only certain relations to be active. The mechanisms used by the designer to access the required information from the semantic network are called notebooks. Notebooks are discussed in the following section.

5.5.2 Protocol Representation.

The protocol template consists of a group of schemata that represent the actions of the protocol, linked together through the PRECEDES relation according to the protocol sequence. Figure 5.20 shows the protocol template corresponding to the two-signal bus arbitration protocol. Each action has a distinct name. The type property characterizes the function of the action in the protocol.

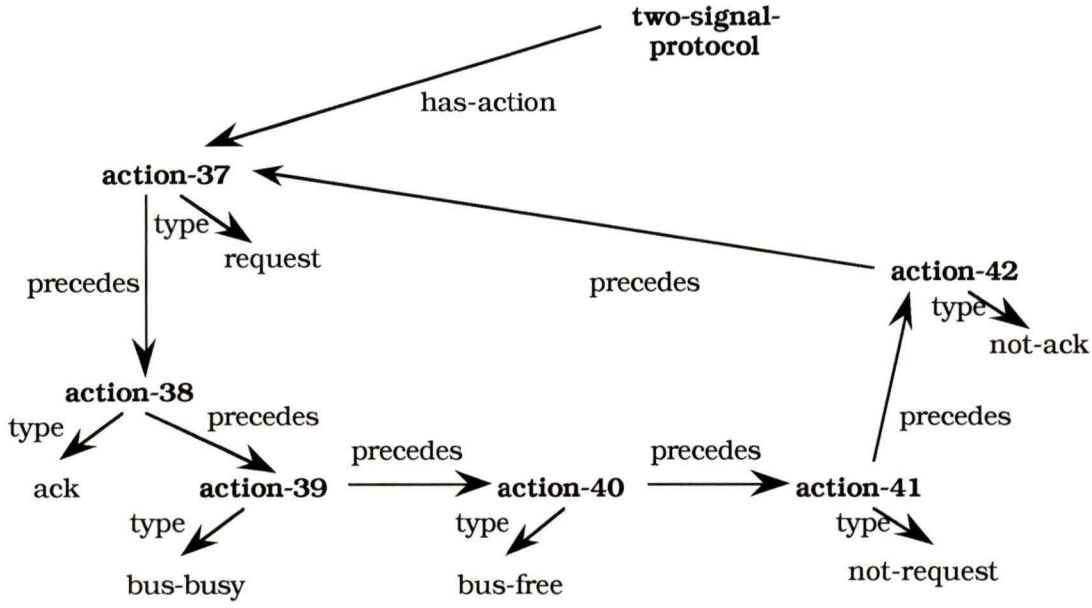


Figure 5.20 Protocol schemata network in DAME.

Figure 5.21 illustrates the schema corresponding to the request action. Each action inherits the properties of the ACTION template. The type information is a slot whose value identifies the function of the action. The HAS-DESCRIPTION slot contains the event expression that defines the action. Because the value in this slot is component-dependent, it is left empty in the template and it is filled accordingly in each instantiation of the template. The PRECEDES relation slot points to the following action schemata. In figure 5.21, action-38, corresponding to ack (see figure 5.20), is contained in the PRECEDES slot.

```
(DEFSCHEMA ACTION-37
 (IS-A          ACTION)
 (TYPE          REQUEST)
 (HAS-DESCRIPTION )
 (PRECEDES      ACTION-38))
```

Figure 5.21 Schema that represents the request action in the protocol graph.

5.6 Design Knowledge Base.

The design knowledge base is structured in clusters of rules pertaining to specific aspects of the design. The design process is organized in a hierarchical manner [26] as described in chapter 2, and proceeds through a continuous refinement of the structures arrived at the previous layers of the hierarchy. Clusters of rules at particular layers are activated to carry the design forward. For example, in the functional block design layer, there exists a cluster of rules dealing with the design of the arbitration subsystem, as shown in figure 5.22. The supplied bus arbitration specification, generated in the Configuration phase, is taken by the bus arbitration interface design module in order to produce the structural and functional specification of the desired interfaces. The subsequent implementation layers are responsible of producing the final hardware circuitry.

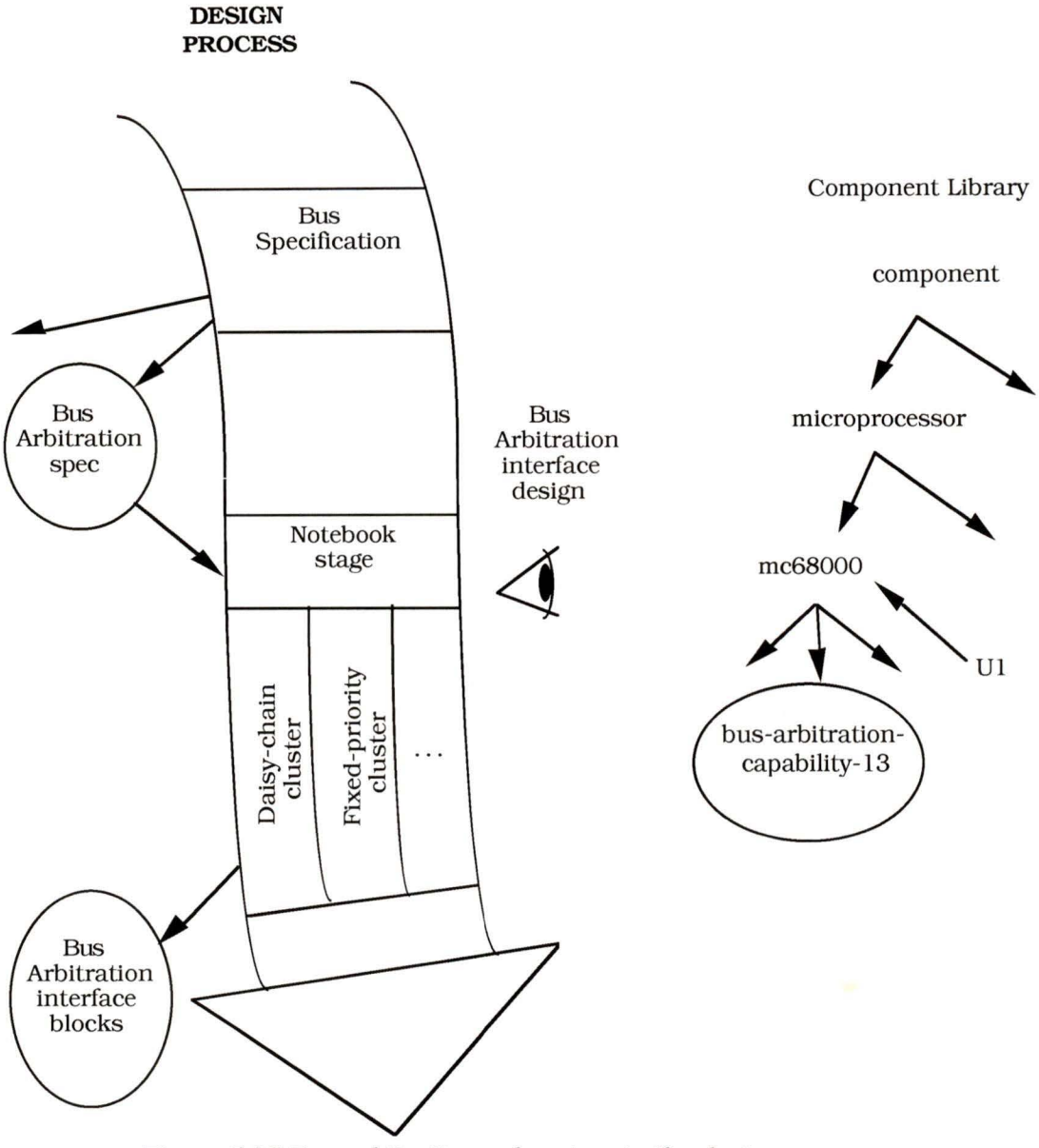


Figure 5.22 Bus arbitration subsystem in the design process.

The procedure followed by the bus arbitration interface design stage to refine the design is depicted in figure 5.23. The pertinent information required by this module is retrieved from the component library with the help of notebooks. Empty interface blocks are placed in the working memory for each master that can access the bus. Different clusters deal with specific arbitration structures, such as independent-request, daisy-chain, round-robin, etc. Finally the structural and functional specification of the interface is produced so that additional stages can carry out the hardware design.

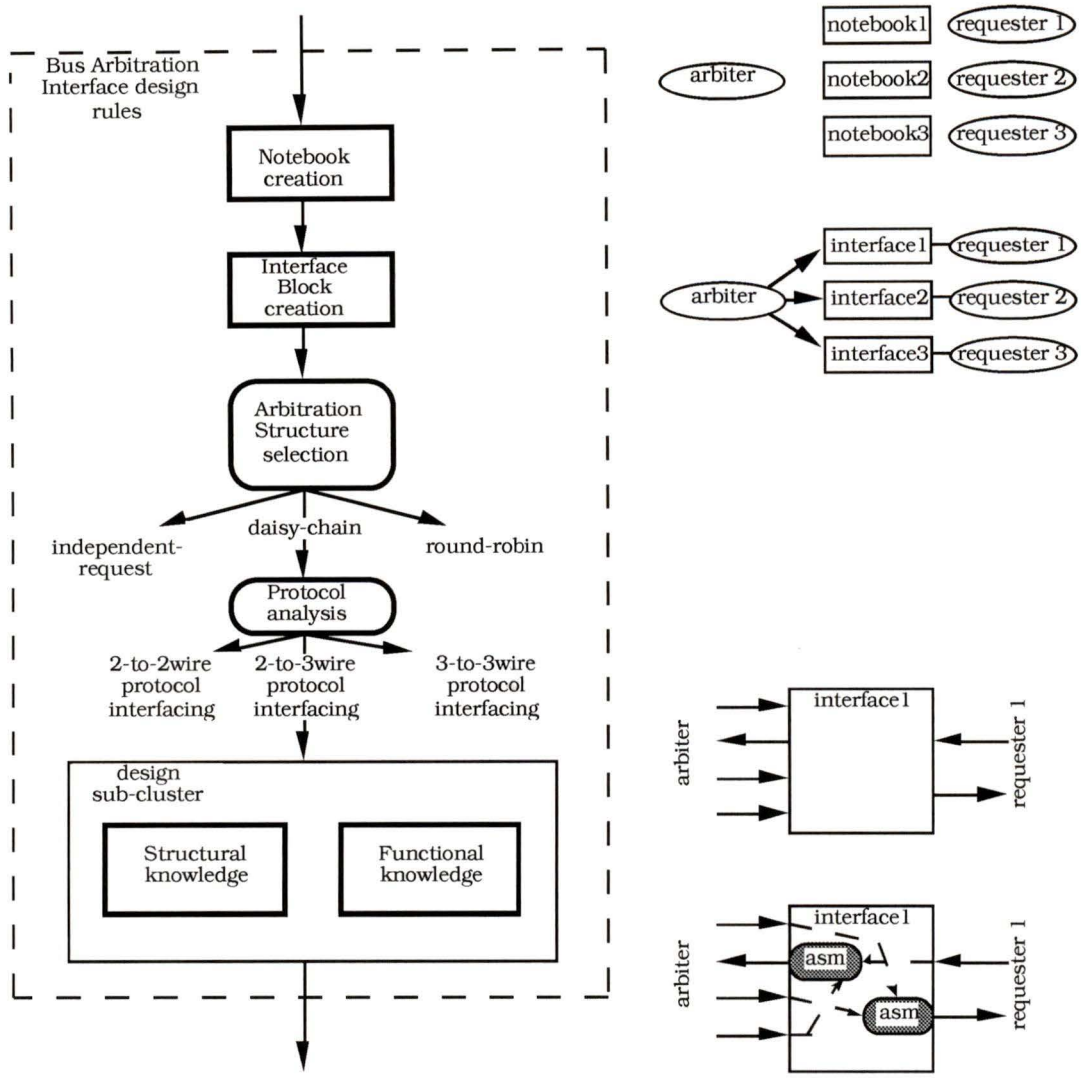


Figure 5.23 The structure of the cluster of rules that accomplishes the bus arbitration Subsystem Design.

5.6.1 Notebooks.

As illustrated in figure 5.22, the cluster is allowed to look at the component library. Since clusters of rules focus into specific aspects of the design process, they access only pertinent information from the network of schemata. Each cluster of rules employs a private notebook in which it copies the relevant parts of the networks of schemata of the participating components, ignoring the remaining information. Hence, notebooks are mechanisms used by the designer to transverse the component library in order to retrieve the specific information necessary to implement a particular design. The power

of inheritance is employed here in order to efficiently access the necessary information from the data base. Knowledge Craft™ provides inheritance implicitly in the schemata and through the CRL™ language.

Notebooks are created at the beginning of every design stage. Figure 5.24 shows a notebook creation rule, part of the bus arbitration interface design cluster. At this stage of the design, such a rule, in the presence of a bus arbitration specification that involves a requestor device, creates a notebook containing a link to the requestor's bus arbitration protocol together with other related information (such as the bus arbitration spec and component names). In order to get the protocol link, the rule is allowed to use the functions provided by the CRL language.

```
(p create-BA-requestor-notebook
 (bus-arbitration-spec ^schema-name <ba-spec> ^requestors <reqs>)
 (component ^schema-name {<name> (member <> <reqs>)}
        ^has-capability {<ba> (is-BA <>)} )
 (step ^phase bus-arbitration-interface)
-->
 (let((ba-prot(get-value-if (is-ba <ba>) 'uses-protocol
        #'(lambda (x) (equal (get-value x 'role) 'requestor)):start 0 :end nil)))
 (cschema (create-name 'notebook)
        ('instance      'notebook)
        ('spec          <ba-spec>)
        ('component     <name>)
        ('role          'requestor)
        ('protocol      ba-prot))))
```

Figure 5.24 Notebook creation rule in the bus arbitration interface design cluster.

Notebooks can be destroyed at the end of each design stage. Otherwise they can be kept as records. The notebook concept gives to the design clusters clarity, and independence of structure.

5.6.2 Design Rules.

Consistent with the design philosophy of DAME, the design rules apply to the abstract design necessitated by the presence of particular types of protocols in the

participating modules rather than the specific instantiations of the protocols. The knowledge base is capable of instantiating the abstract design based on the instantiations of the protocols in the modules.

This technique of abstraction allows us to use but a limited number of powerful general rules rather than a plethora of specific rules which apply to specific instances of components. Interface blocks are produced complete, with the description of their function and their interconnection. The subsequent implementation layers are responsible of implementing the functionality of these blocks in a particular technology.

Design rules firstly create interface blocks, as empty instantiations of the INTERFACE-BLOCK template, in the working memory, for each pair of devices (master, arbiter) in the arbitration subsystem. According to the arbitration structure included in the bus arbitration input specification, a specific cluster is activated. Possible arbitration structures are independent-request (fixed priority, centralized), daisy-chain (fixed priority, distributed), round-robin (fair selection), etc. Depending on the protocols of the involved devices, one sub-cluster is invoked, the one that contains the required protocol interface design knowledge. This sub-cluster will produce the structural and functional specifications of the interface.

Consider the sub-cluster that deals with the interface of a two-signal protocol requestor with a three-signal protocol arbiter, in a daisy-chain arbitration structure. The corresponding rule that defines the req-out ASM is shown in figure 5.25. Basically this rule states that if there exists an interface block for a daisy chain structure between a two-signal protocol requestor and a three-signal protocol arbiter, and the respective ASM has not been created yet, but the interface signals that encode the required precedent actions exist in the working memory, the rule is enabled and the instantiation of the req-out ASM is imminent.

Whenever no other rule in this cluster can fire, switching to the next stage of the design is performed, meaning either that the design is complete or that this cluster is not required for the particular design under consideration. The rules for the daisy-chain cluster are contained in Appendix B.

```
(p design-2to3-req-out-asm
 (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^type asm ^has-asm <asms>)
 -(asm ^schema-name (member <> <asms>) ^function request)
 (notebook ^schema-name <req-out> ^block <block> ^function req-out)
 (notebook ^schema-name <req-in> ^block <block> ^function req-in)
 (notebook ^schema-name <grant-ack> ^block <block>
        ^function grant-ack)
 (step ^phase bus-arbitration-interface)
-->
 (cschema (create-name 'two-state-asm)
        ('instance          'two-state-asm)
        ('function          'request)
        ('has-asm+inv       <block>)
        ('input1            (get-value <grant-ack>      'active))
        ('input0            (get-value <req-in>         'active))
        ('output1           (get-value <req-out>        'inactive))
        ('output0           (get-value <req-out>        'active)))
)
```

Figure 5.25 Typical design rule in the bus arbitration interface design cluster.

5.7 Summary.

The representation problem attacked in the previous chapter, was exploited in the construction of the bus arbitration designer subsystem in DAME. A hierarchy of signals, actions, protocols, capabilities and components represents different levels of abstraction in the interface problem. An interface allows two components to communicate to each other. The interface can be divided further down so that each component's capability can be tackled separately. Protocols describe the sequence of operations of a capability. Interfacing protocols can be translated into the operation of merging action graphs. This abstract interface model can then be used by the designer to instantiate the interface with the corresponding signals from the components.

The distributed arbitration, as represented by a daisy-chain scheme, has been selected in this work to demonstrate the designer subsystem, because the distributed

nature of the arbitration puts some of the complexity in the interface design. Particularly, a mixture of the protocol handling together with the arbitration synchronization problem are mixed together in the interface model. Nevertheless it is thought that the proposed methodology not only applies to this seemingly quite restrictive case, but also to the other allocation structures. In particular, the centralized arbitration design is close to the daisy-chain, the main difference being that the arbitration has been removed from the interface, simplifying its design.

Currently, the daisy-chain cluster has been incorporated into the DAME designer in two flavors: fixed priority and fair design (masking mechanism). This cluster includes sub-clusters that deal with the possible combinations of two-signal and three-signal protocols for arbiter and requestors. It is planned to integrate the independent-request cluster as well in the near future. It is noted that the fair design implemented is equivalent to a distributed round-robin arbitration scheme.

6. A Design Example.

6.1 Introduction

Microcomputers are evolving rapidly, and new microprocessor families are being developed by various companies to support them. Very sophisticated designs are possible based on off-the-shelf components. However, this continuous advancement requires sophisticated hardware designers. One solution that attempts to resolve the demand for new computer systems with increased performance, reduced price, and short design period, is to automate the design.

One of the crucial structures in the design of computer systems is the bus. This communication path allows the transfer of information between masters at different levels of complexity. In this chapter the design principles in DAME are illustrated through an example involving a standard bus in the scope of bus arbitration.

6.2 The Example.

As an example we present the design produced by the DAME designer for a direct memory access (DMA) device in a VME-based multi-board system. The VMEbus standard was developed in 1981 by Mostek, Motorola and Signetics/Phillips to support the new generation of 16- and 32-bit microprocessors, with special attention in the Motorola MC68000 microprocessor. The design goal was to produce a high speed and high performance bus with flexible interrupt management and multiprocessor capability.

6.2.1 VMEbus Description.

The bus is fully parallel and it uses a fully-responsive asynchronous data transfer protocol. The address space is 16 Mbytes and the bus can accommodate backplane configurations with both 16 and 32 data lines. The basic structure of the bus consists of

4 groups of signals, called buses: Data Transfer bus (DTB), DTB Arbitration bus, Priority Interrupt bus and Utility bus. The bus signals are mapped on a 96-pin connector. The original VMEbus specifications became IEEE standard 1014 in 1987. In the following, VME is used to refer to the published version of the standard [45].

Each information transfer on VME consists of two sequential cycles: 1) arbitration, and 2) addressing and data transfer. The arbitration cycle takes place only in multiple-master systems.

The arbitration is based on a daisy-chain scheme. As shown in figure 6.1, four daisy-chains are arranged in a fixed priority fashion, daisy-chain 3 having the highest priority. For each daisy-chain there are 3 dedicated lines: BRx*, BGxIN*, and BGxOUT*, where x=0,1,2,3 refers to one of the daisy-chains.

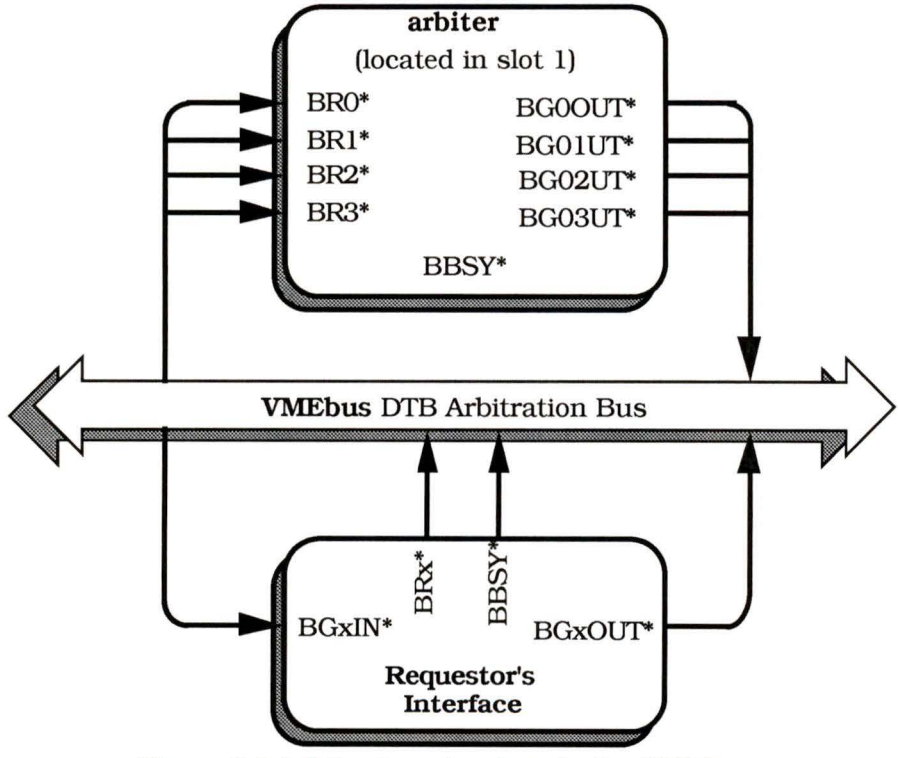


Figure 6.1 Arbitration structure in the VMEbus.

The requests from the masters in daisy-chain *x* are directed to the arbiter through the wired-OR bus request line BRx*. BGxIN* and BGxOUT* are the bus grant lines that implement the daisy-chain. The arbiter awards use of the bus by asserting the proper daisy-chained line BGxIN*. The grant propagates down the daisy-chain, typically passing through several boards in the process, from BGxIN* to BGxOUT*. If a board receives a grant but has not requested the bus, it routes the grant through its BGxOUT* line; otherwise

BGxIN* is gated on the board, and the corresponding BGxOUT* line is disabled. Once a requestor has been granted the use of the DTB, it asserts the bus busy (BBSY*) line. The release of BBSY* allows the arbiter to grant the DTB to some other requestor. There are two restrictions upon the release of BBSY*:

1. After being granted the bus, the requestor shall drive BBSY* low for at least 90 ns, even it does not use the bus.

2. In response to a falling edge on BGxIN*, the requestor shall maintain BBSY* low for at least 30 ns after it releases BRx*. The 30 ns delay between the negation of BRx* and the negation of BBSY* ensures that the arbiter does not mistakenly interpret an old bus request as a new one.

VME supports the following modes of arbitration:

• Prioritized arbitration (PRI), that assigns the bus according to a fixed priority scheme, where each of four bus requests lines has a priority, from highest (BR3*) to lowest (BR0*).

• Round-robin arbitration (RRS), that assigns the bus on a rotating priority basis, in such a way that when the bus is granted to the requestor on bus request line BR(n)*, then the highest priority for the next arbitration cycle is assigned to bus request line BR((n-1) modulo 4)*.

• Single-level arbitration (SGL), that only accepts requests on BR3*, and relies on the propagation of the grant through the daisy-chain 3 to arbitrate the requests.

However, other scheduling algorithms other than the ones mentioned above can be used. For example, another arbitration mode might assign daisy-chain 3 the highest priority, but use a round-robin scheme for the other daisy-chain levels.

Two important rules must be observed:

1. The arbiter shall be located in slot 1.

2. If a backplane slot is not occupied by a board, and if there are boards further down the daisy chain, then jumpers shall be installed at the empty slot to pass through the daisy-chain signal.

Three types of requestors are described by VME: a release-when-done (RWD) requestor, a release-on-request (ROR) requestor, and a FAIR requestor. The RWD

requestor's interface releases BBSY* as soon as the requestor ends its transaction.

The ROR interface does not release BBSY* when the requestor ends its data transfer unless some other master wants the bus. It monitors the four requests lines and negates BBSY* only if at least one of the lines is active. ROR masters reduce the number of arbitrations initiated by a master that is generating a large percentage of the bus traffic.

A round-robin (RRS) arbitration scheme guarantees a fairness arbitration for at most 4 masters. In systems with more than four masters, fairness is provided by designing FAIR requestors. After it has been granted the bus, a FAIR requestor's interface refrains from requesting the bus again as long as there is at least one active bus request pending on its own request level.

A Single Level (SGL) design is considered in the example presented in this chapter in order to focus on the daisy-chain interface problem. The other cases derive directly from this simple example. For instance, the Prioritized (PRI) mode can be considered as an arbitration structure that consists of four daisy-chains. The design of the interfaces for the requestors in a particular daisy-chain will follow the same procedure as in SGL mode. In addition, a PRI's arbiter must have a priority circuit to handle the four independent requests, as explained in chapter 3.

In the SGL Arbitration the arbiter drives BG3IN* at slot 1; BG3IN* and BG3OUT* are the bussed lines for the daisy-chain. All the requestors share lines BR3* to initiate a request, and BBSY* to acknowledge the grant and to take over the bus. The status of the bus can be obtained from monitoring lines BBSY* and AS* in the bus, as explained in the next section.

The arbitration sequence for two requestors that send simultaneous requests to a SGL arbiter is shown in figure 6.2. The arbiter is located in slot 1, while the two requestors A and B occupy slots 2 and 3 respectively.

At the beginning of the sequence, both requestors assert BR3* simultaneously. The arbiter responds by asserting BG3IN*. Because there is no other master in slot 1, there must be a jumper that connects BG3IN* to BG3OUT*. The grant is monitored by requestor A (slot 2) in its BG3IN* line, which then asserts BBSY*, and releases BR3*. Requestor B keeps driving BR3* low. After detecting BBSY* low, the arbiter negates BG3IN*. When requestor A has completed (or is about to complete) its last data transfer, and the 30 ns delay since its release of BR1* has been satisfied, it releases BBSY*. Thus the arbitration and the data transfer operations can proceed concurrently.

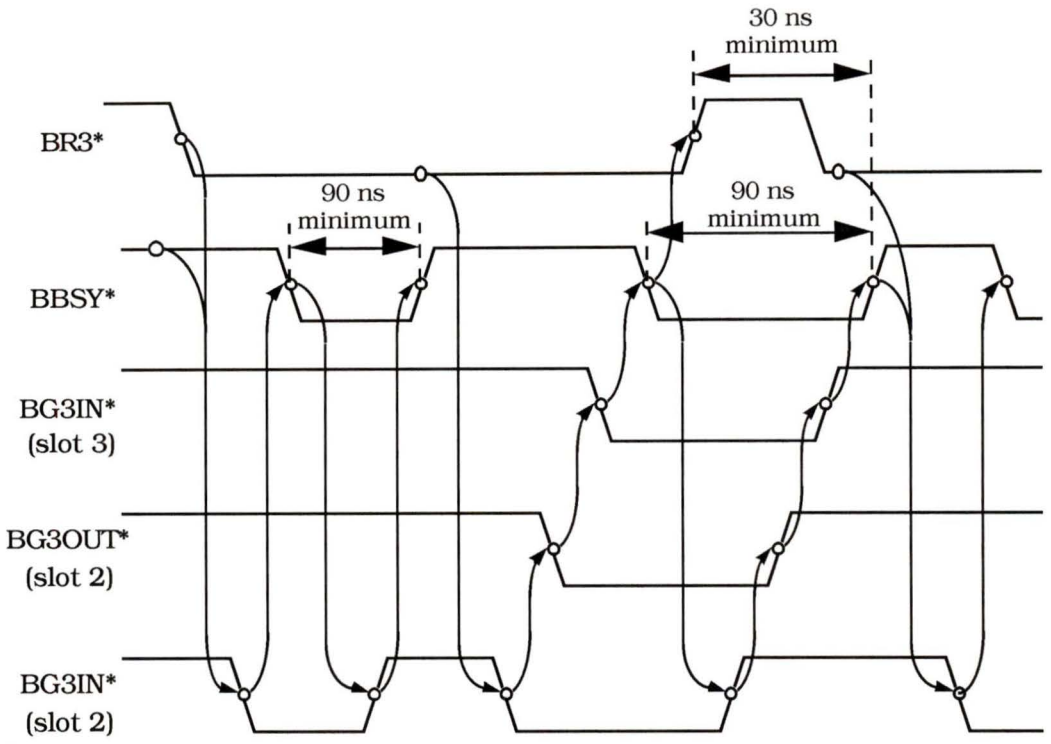


Figure 6.2 Arbitration sequence diagram for two requestors in VME SGL mode.

The arbiter interprets the release of BBSY* as a signal to arbitrate any current requests. Since BR3* is still asserted, the arbiter drives BG3IN* active. Requestor A passes the grant from its BG3IN* to BG3OUT*. Requestor B detects its BG3IN* asserted and responds by asserting BBSY* and releasing BR1*. When the arbiter detects BBSY* inactive, it releases BG3IN*, which propagates down the daisy-chain. After requestor B ends using the bus, it releases BBSY*. Because there is no other request, the bus remains idle until another master asserts the request line. Note that lines BG3IN* and BBSY* are fully interlocked.

Race Conditions.

Suppose there are two requestors A and B that share a common bus request line, and requestor B is further down in the daisy-chain. Requestor B asserts BRx*, and the arbitration generates a grant that propagates through the daisy-chain. When the grant reaches requestor A, this master just decides it wants the bus. Requestor A is permitted to take the grant. However, if requestor A is improperly designed, it may assert BGxOUT* momentarily, resulting in a transient that will propagate down to the lower priority requestors. The VME standard prohibit explicitly such a design. The cause of this malfunction can be traced back to the synchronization problem discussed in chapter 3.

For example, when an interface is designed so that it latches the request from the master upon the falling edge of BGxIN*, there is a possibility that both events occur simultaneously. In this case, the set-up times in the latch are not respected and a metastable state may occur. In VME, there is no maximum time specified for the requestor to pass along the bus grant so that the interface can avoid this race situation by inserting delays that allow any undesirable state to disappear.

6.2.2 Problem Statement.

In the subsequent, the design carried out by the DAME bus arbitration subsystem is presented. The system consists of a DMA device in a multi-board system connected to the VMEbus in Single (SGL) arbitration mode, as shown in Figure 6.3. The DMA device used in this example is Intel 8257.

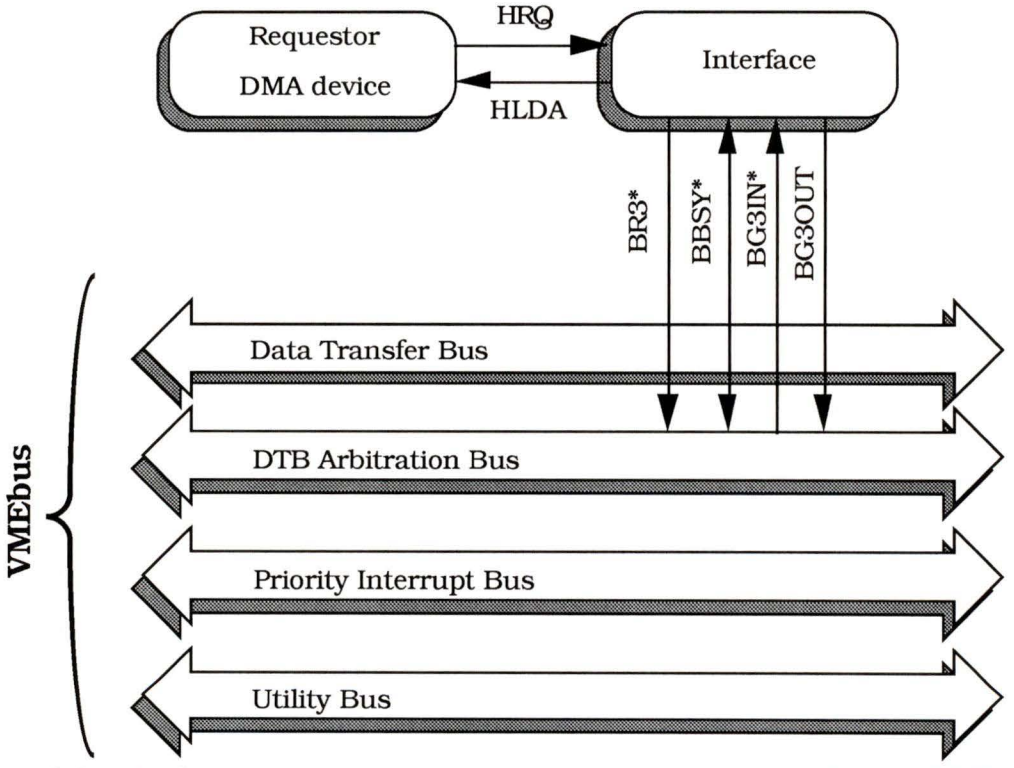


Figure 6.3 DMA device as a requestor in a multi-board system based on the VMEbus.

6.3 Data Representation.

The DAME designer uses the component library to retrieve the necessary information in order to complete the interface design. The arbitration subsystem focuses

on the bus arbitration capabilities of the devices to be connected. In this section, the protocols for the VMEbus and the DMA device as they are represented in the component library are described.

6.3.1 Protocol Identification.

From the discussion in the previous section, it is apparent that the bus arbitration in the VMEbus follows a three-signal protocol, as described in chapter 3 (cf. figure 3.19), and it can be modelled by the action graph depicted in figure 4.33. Lines BR3*, BG3IN* and BBSY* correspond to action pairs (**R**, $\overline{\mathbf{R}}$), (**G**, $\overline{\mathbf{G}}$), and (**GA**, $\overline{\mathbf{GA}}$), respectively. BG3OUT* is the daisy-chain bus **grant-out** signal (see figure 5.8).

Similarly, the DMA device used, Intel 8257, uses a two-signal bus arbitration protocol, with HRQ as the request signal, describing action pair (**r**, $\overline{\mathbf{r}}$), and HLDA as the acknowledge signal, describing action pair (**a**, $\overline{\mathbf{a}}$), as illustrated by the action graph in figure 4.32.

6.3.2 Bus Status Actions.

In the VMEbus there is no dedicated line for the status of the bus. Both signals BBSY* and AS* serve this purpose. The state diagram of figure 6.4 describes the different states the bus can take. State A is the initial state, in which both BBSY* and AS* are inactive. After the assertion of BBSY* and AS* by the current master, the bus changes to states B and C, respectively. While in state F, the bus is idle, but owned by the current master (CM). State D takes place when the current master negates BBSY* during its last transaction so that the arbitration can take place concurrently. In state E, a new requestor acknowledges the grant given out by the arbiter, indicating it will become the next CM; in the meanwhile, the bus is still executing the last transaction of the previous master. Finally the bus changes to state G when the previous master gives up the bus so that the newly selected commander is able to take over.

Action **B**, in which the bus is being used by the current master, corresponds to states B, C, D, E, and F, while action $\overline{\mathbf{B}}$ corresponds to states A and G, in which the bus is either idle, not being owned by any master, or being transferred from the previous CM to the new CM.

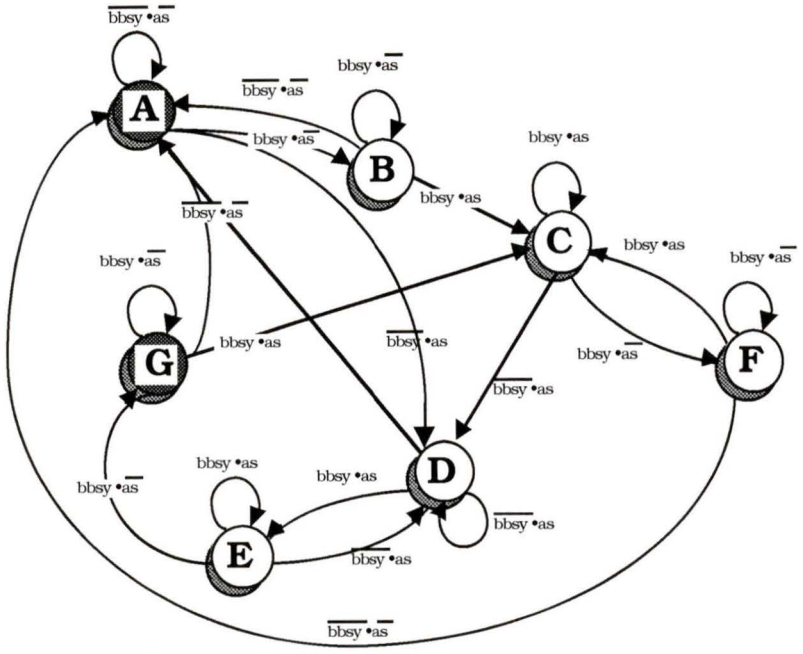


Figure 6.4 Bus utilization state diagram.

To the effect that the prescribed actions **B** and $\overline{\mathbf{B}}$ included in the template specifying the three-signal bus arbitration protocol, as described in section 4.7, do not correspond directly to any signal in the VMEbus, thus, the observer discussed above is included in the instantiation of the protocol, and the actions **B** and $\overline{\mathbf{B}}$ are mapped to states (B, C, D, E, F) and (A, G) respectively.

6.3.3 Bus Arbitration Capabilities.

The bus arbitration capability of a device is a sub-tree in the semantic network that contains the functional description of the bus arbitration lines. The USES-PROTOCOL relation allows the designer system to transverse the device's network of schemata in order to look at the protocol description.

Protocols are stored in the component library as a collection of schemata representing the actions, linked together by the PRECEDES relation, as shown in figure 5.20. These protocols are instantiations of the abstract protocols that were discussed in chapter 4. The design rules apply to the abstract protocols, but also use the instantiations to complete the abstract design with the particular signals that effect the protocol.

Figure 6.5 presents the schema corresponding to the **R** action in the bus-arbitration protocol for the VMEbus. This schema is an instantiation of the ACTION template. The distinctive slot is the description of this action in terms of the signals in

the bus. This particular action is mapped into the event expression (! ASSERTED VME-BR3)[1].

```
(DEFSCHEMA ACTION-7
 (IS-A ACTION)
 (FUNCTION    REQUEST)
 (HAS-DESCRIPTION (! ASSERTED VME-BR3))
 (PRECEDES    ACTION-8))
```

Figure 6.5 Schema that represents the request action in the VMEbus bus arbitration protocol.

The accompanying schemata for the rest of the actions that constitute the bus arbitration protocols for the VMEbus and the DMA device are included in Appendix A. It suffices to present the description of the actions for the two participating protocols. The four actions that participate in the protocol of the DMA device are described by the following event expressions:

r:= ! ASSERTED HRQ $\quad$ $\overline{\mathbf{r}}$:= ! NEGATED HRQ

a:= ! ASSERTED HLDA $\quad$ $\overline{\mathbf{a}}$:= ! NEGATED HLDA

Similarly the event expressions that describe the actions in the three-signal protocols are:

R:= ! ASSERTED BR3 $\quad$ $\overline{\mathbf{R}}$:= ! NEGATED BR3

G:= ! ASSERTED BG3IN $\quad$ $\overline{\mathbf{G}}$:= ! NEGATED BG3IN

GA:= ! ASSERTED BBSY $\quad$ $\overline{\mathbf{GA}}$:= ! NEGATED BBSY

The actions corresponding to the status of the bus are described by the bus monitor of figure 6.4, as discussed above. The active state of the monitor's output, BUSY, indicates that the bus is currently being used by the current master and corresponds to states (B, C, D, E, F) of the observer. The inactive state corresponds to states (A, G). Actions **B** and $\overline{\mathbf{B}}$ can be described by the following state expressions:

B:= ASSERTED BUSY $\quad$ $\overline{\mathbf{B}}$:= NEGATED BUSY

---

[1] Note that the * (negative logic, asserted low) symbol is not needed in the event expression containing BR3, because this information is included in the description of the signal.

6.4 Designer Session.

The Configuration phase, in which the gross system architecture is established, will produce, for the case of a multiple-master system, a bus arbitration specification. The specification contains the basic information about the allocation structure: type, participants, required details for each structure. For instance, if arbitration is to be used, more information is required such as if it is centralized (independent request) or distributed (daisy-chain), which device is the arbiter, and a list of masters, and the algorithm to perform the selection (fixed priority, round-robin, etc.). For a fixed priority algorithm, the priority of the masters is also necessitated.

The bus arbitration designer subsystem initially checks if there is a bus arbitration specification before it proceeds to carry out the design. Notebooks are then created that retrieve the pertinent information from the semantic networks. Empty templates for each interface block are placed in the working memory, to be instantiated by the design rules. These rules are organized into clusters that specialize in different arbitration structures. Clusters have sub-clusters. These sub-clusters are the ones that contain the knowledge to define the interface structure and produce the functional specification, according to the nature of the participating protocols. The schemata comprising each interface embody the description that will be translated into hardware by the implementor subsystem.

6.4.1 Input Specifications.

The bus arbitration specification produced in the Configuration phase of the design hierarchy contains basic information about the devices (masters) that will share a common communication path (bus).

Figure 6.6 shows the bus arbitration specification for the example under consideration. The bus allocation structure instantiated is a daisy-chain arbitration structure. For a daisy-chain scheme, the identification of the arbiter and a list of the requestors is demanded. The order of the list indicates the priority of the requestors, the first in the list having the highest priority. Because in the board there is only one master, the schema's REQUESTORS slot contains only one device, U5, but in the general case, this slot can collect a list of several masters. B1 represents the instantiation of the VMEbus for this particular board.

```
(DEFSCHEMA BUS-ARBITRATION-SPEC-1
 (INSTANCE DAISY-CHAIN)
 (ARBITER B1)
 (REQUESTORS U5))

(DEFSCHEMA B1
 (INSTANCE VME-BUS))

(DEFSCHEMA U5
 (INSTANCE DMA-DEVICE))
```

Figure 6.6 Schema defining the input bus arbitration specification.

6.4.2 Interface Design.

The present example exercises the sub-cluster of the daisy-chain rules dealing with the interfacing of a two-signal protocol device into a three-signal protocol daisy-chained arbitration structure (as defined by the VMEbus). The corresponding structural model is represented in figure 6.7.

Starting from the general input specification of the bus allocation structure, until the complete output description of the interface is produced, several transformations are carried out by the designer whose objective is to progressively refine the design. As mentioned in chapter 5, in the first place, notebook creation rules are activated that retrieve the protocol information corresponding to the bus arbitration capability of the arbiter and the requestors. In the case that one device has more than one protocol describing the capability[2], the notebook would have to get the appropriate one.

After the notebooks are created, the interface-block creation rules place copies of the INTERFACE-BLOCK template in the working memory, one for each pair of devices (master, arbiter) that needs be interconnected.

---

[2] In chapter 5, the MC68000 was shown that contains two bus arbitration protocols, one for the case it functions as a default master/arbiter, and the other when it is a two-signal requestor.

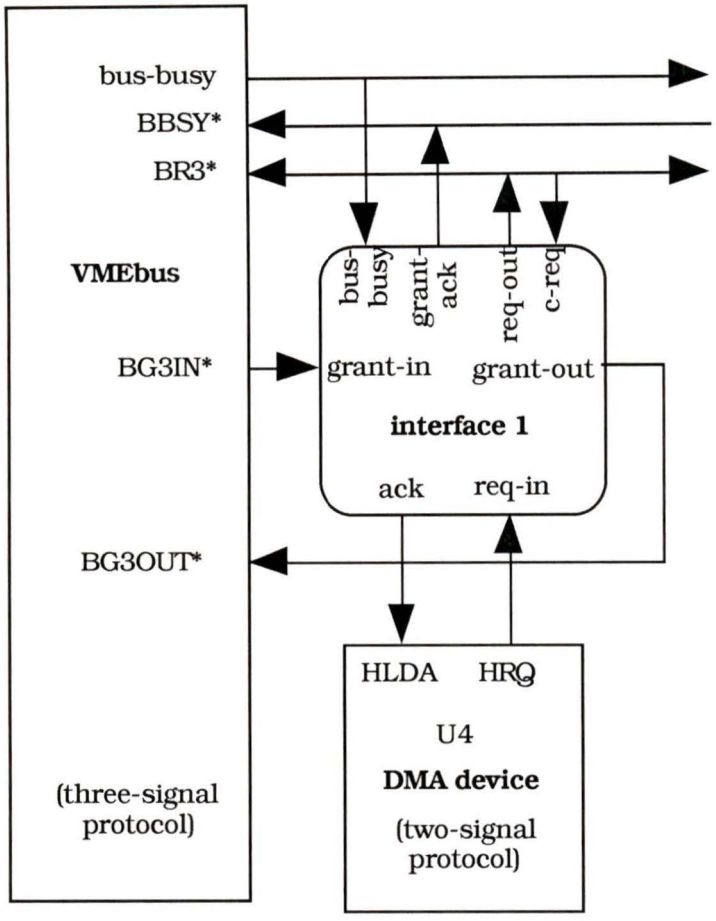


Figure 6.7 Interface model for the DMA device connected to the VMEbus.

According to the bus allocation structure defined in the input specification, only the respective cluster of rules is selected. Some general information, such as the priority in the daisy-chain, is determined at this level.

The specific sub-cluster that accommodates the knowledge to construct the structural and functional description of the interface is organized in the following groups:

• Rules that create notebooks for the structural signals. For this example, notebooks are created for the internal signals of the interface: **req-in**, **ack**, **req-out**, **bus-busy**, **ack**, **grant-in**, and **grant-out**. The notebooks will contain the event descriptions of the corresponding actions (i.e., **r** and $\bar{\mathbf{r}}$ for the **req-in** signal notebook).

• Rules that decide about the T (transform) blocks that convert the actions into signals. In most of the cases those rules produce a C (connect) block that connects an external signal comprising the action pair into the internal signal. C blocks can be buffers, inverters, open-collector drivers, or wires, according to the nature of the

interconnecting signals. For instance, the bus observer of figured 6.4 is associated to signal **bus-busy**.

• Rules that create the sequential machines and mutual exclusion (ME) blocks. These rules instantiate the inputs and outputs of the two-state ASM and ME blocks with the respective internal signals. The input and output slots of the schemata representing these basic blocks can accommodate event and state expressions, embracing the corresponding event detectors.

• Rules that detect the presence of errors. This group monitors the working memory for certain situations that may indicate a problem in the design. For instance, if one of the listed requestors does not have a bus arbitration capability, the associated error rule will fire, stopping the process, asking for a corrective measure.

These rules fire as soon as their antecedents are satisfied, sharing the typical opportunistic behavior of data-flow driven systems. When no more can be activated, the output description of the interface is contained in the working memory, ready to be processed by subsequent designer's subsystems.

6.4.3 Output Description.

Figure 6.8 shows the description of one of the two-state ASM comprising interface 1. The rest of the functional blocks produced by the designer subsystem are included in Appendix C. In order to verify that these schemata correctly captures the interface's hardware specification, the implementation was carried out manually, as described in the next section.

```
{{ TWO-STATE-ASM-2
 INSTANCE: TWO-STATE-ASM
 HAS-BLOCK+INV: INTERFACE-BLOCK-1
 ACTION: REQ-OUT
 INPUT0: (! ASSERTED INTERFACE-BLOCK-1-GRANT-ACK)
 INPUT1: (! ASSERTED INTERFACE-BLOCK-1-REQ-IN)
 OUTPUT0: (! NEGATED INTERFACE-BLOCK-1-REQ-OUT)
 OUTPUT1: (! ASSERTED INTERFACE-BLOCK-1-REQ-OUT)}}
```

Figure 6.8 Frame describing the two-state ASM for **req-out**.

The important information contained in the schema of figure 6.8 is: type of block (two-state ASM), which interface it belongs to (INTERFACE-BLOCK-1), and for this particular block, its inputs and outputs. Although the action slot (req-out) is redundant, it makes this schema easy to read[3].

6.4.4 Designed Interface.

The complete description of the designed interface is shown in figure 6.9. The two-state ASM and the Mutual Exclusion (ME) block generate the required actions. T blocks have been restricted to connect blocks (C), that translate the logic level and technology of a single signal. Two kinds of detectors blocks are used: single detectors (D), and AND detectors (^D). Finally, the observer O monitors the bussed signals BBSY* and AS* to infer the availability of the data transfer bus.

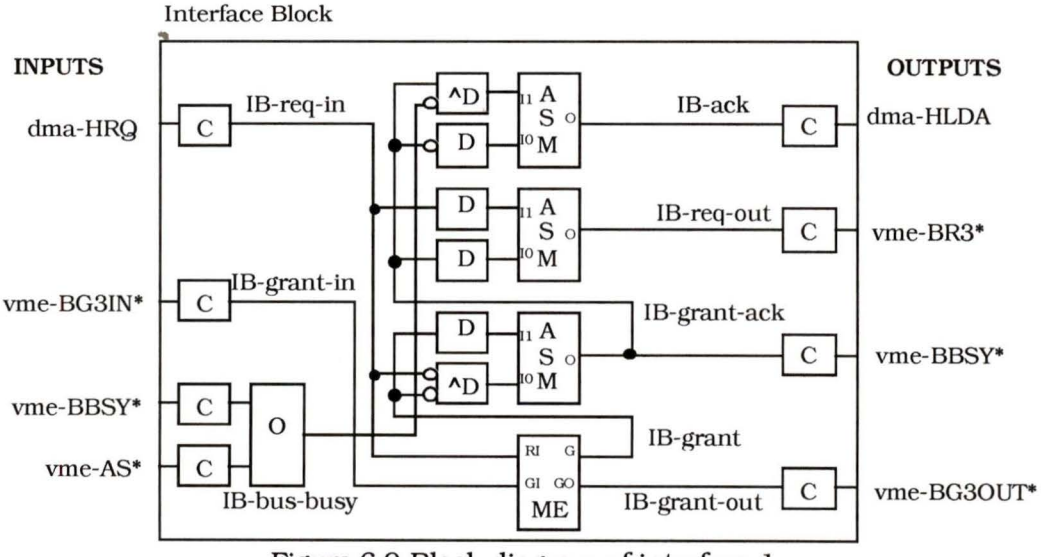


Figure 6.9 Block diagram of interface 1.

6.5 Hardware Implementation.

The final VME bus arbitration design was implemented manually from the design blocks produced by the DAME bus arbitration design subsystem, by following conventional digital design procedures. The procedure followed for the design of the two state ASM can be found in Appendix D. Connection blocks were transformed into buffers.

---

[3] One can know immediately that this is the req-out ASM without looking at the information in the other slots.

State transition diagrams were developed for the asynchronous state machines required in this design, which were in turn minimized and converted to a combinatorial logic implementation. A circuit diagram of the implemented circuit is shown in figure 6.10. This figure also shows the general behavior of the two state machines required.

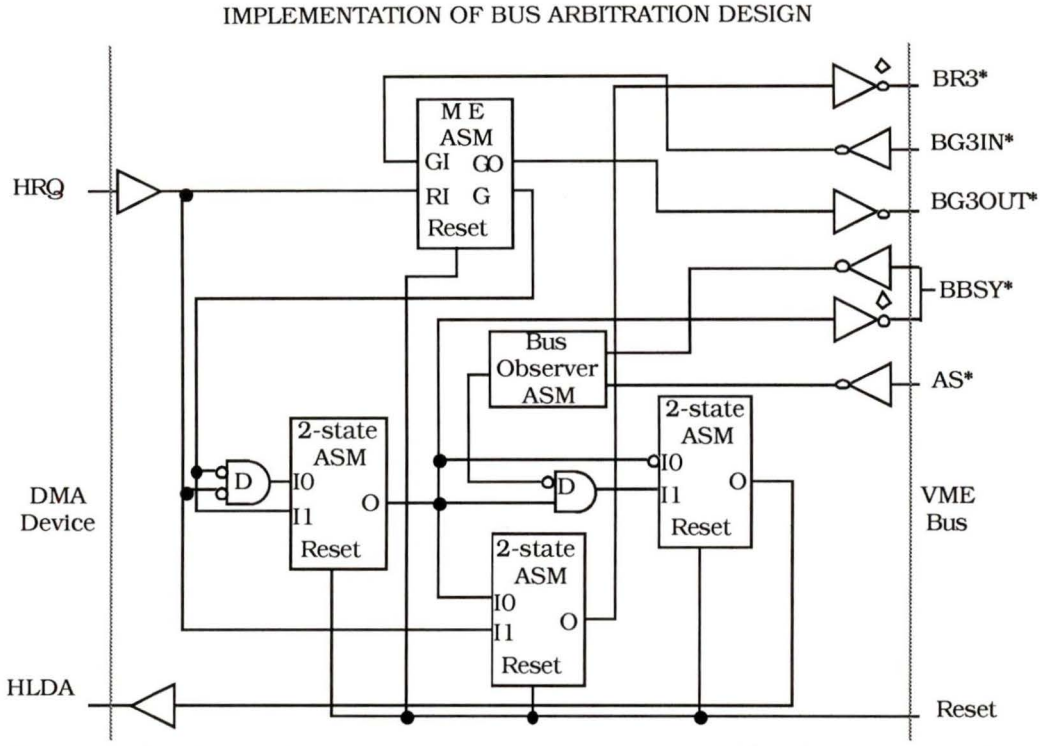


Figure 6.10 Circuit schematic for the implementation of the designed interface.

The final design was implemented using discrete logic. The finished design was transferred to the SILOS logic simulator to perform logical and timing verifications. The test vectors applied to the circuit consisted of all the expected input conditions. Under all conditions the circuit performed as expected. Appendices E and F contain the hardware description as needed by SILOS. Figure 6.11 shows the result of the simulation.

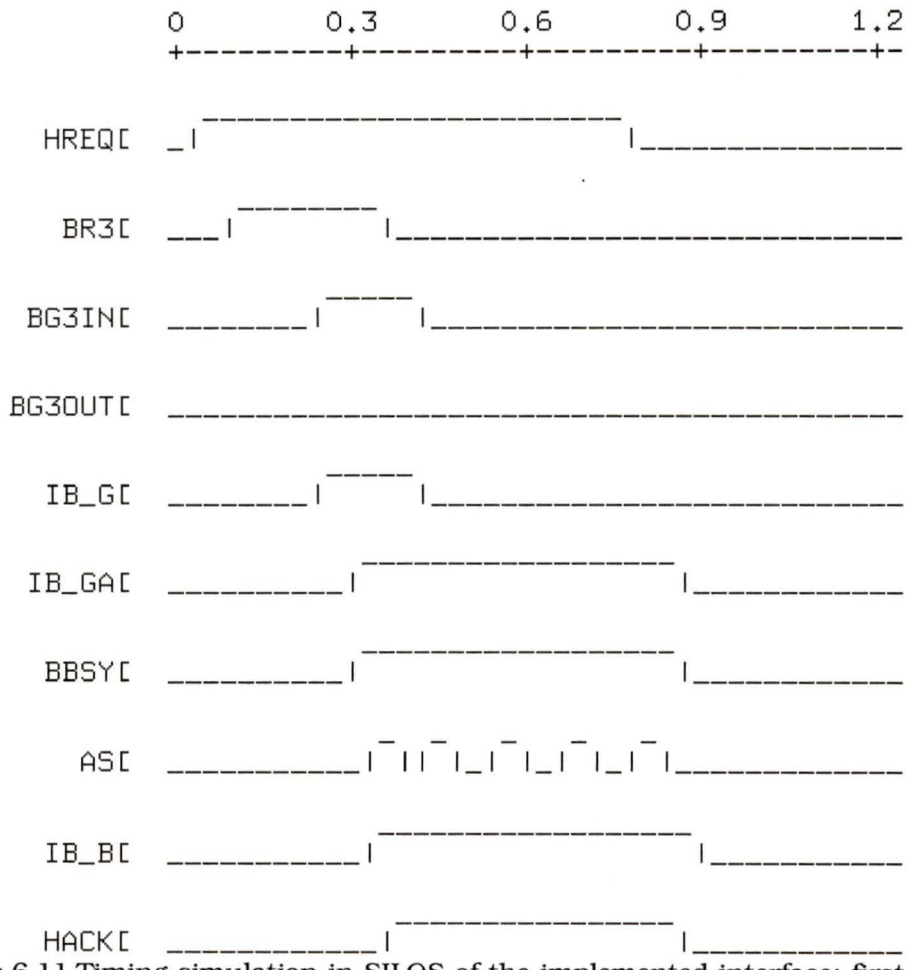


Figure 6.11 Timing simulation in SILOS of the implemented interface: first cycle.

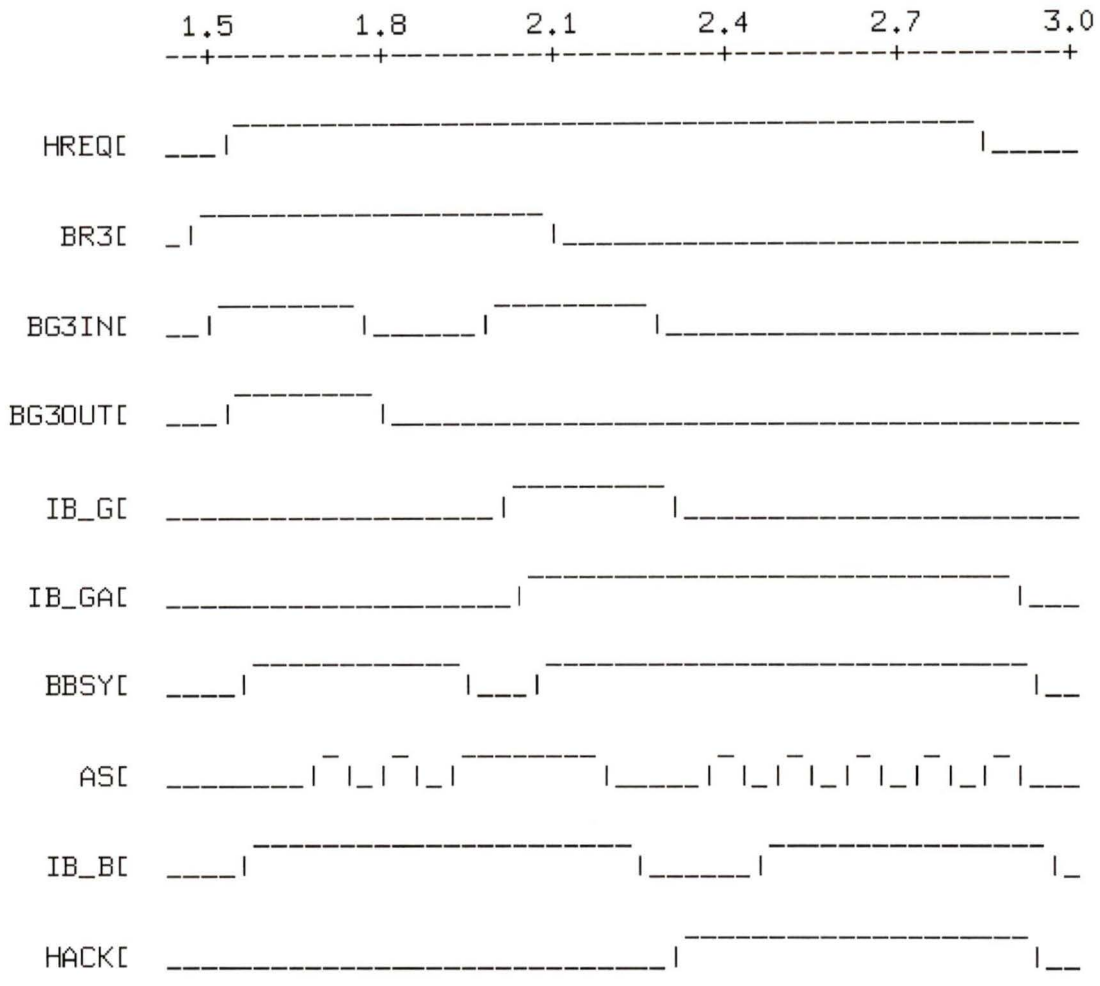


Figure 6.11 Timing simulation in SILOS of the implemented interface: second cycle.

The order of the signals in figure 6.11 intents to give an idea of the time flow in the circuit. Through HRQ, the DMA device requests the bus. The wired-OR BR3 line accepts the master's request. BG3IN and BG3OUT constitutes the daisy chain signals. Through the internal signals IB_G and IB_GA the interface receives and acknowledges the grant. IB_GA is connected in a wired-OR fashion to BBSY. Both BBSY and AS are used by the observer of the bus to induce the status of the resource, described by IB_B. HLDA carries the acknowledge to the DMA device, controlling its access to the bus.

In the first cycle, the bus is available when the DMA device puts the requests in the arbitration lines. It takes only the time to perform the handshake for the requestor to have the bus granted.

During the second cycle, another device has become the current master when the DMA device asserts HRQ. The DMA's interface gets the grant as soon as the other device

signals by releasing BBSY that the arbiter can start another arbitration. The interface acknowledges the grant and waits until the previous master gives up the bus before allowing the DMA device to use the bus. The last data transfer cycle of the previous master has been stretched in order to make the bus transfer more appreciable.

6.6 Summary.

This chapter closes the design cycle that started with the interface problem, stated in the abstract, by producing the hardware that synthesizes the functionality of that interface. Nevertheless, the main idea is centered around the representation of capabilities and protocols of devices, and how to utilize it to automate the design.

A suitable methodology of hardware design that breaks the interface problem into hierarchical levels of representation was presented through an example brought up from the realm of multi-masters systems. Despite this work explored only one facet of the design process, it substantiates the hypothesis behind DAME, that an expert system capable of configuring and designing a customized microprocessor system from original specifications can be constructed.

7. Conclusions

This chapter completes this work by giving conclusions and recommendations based on the experiences gained during this thesis. As well, suggestions are made regarding further work in the area of research.

7.1 Results.

The result of this work is the construction of the DAME's bus arbitration designer subsystem. This module was used to demonstrate the principles in DAME, an expert system that designs microprocessor-based systems from original specifications. The approach in DAME, unlike other previously developed systems, consists of designing in the abstract with powerful generic rules. The bus arbitration module participates in the functional phase of the design, taking as input the bus arbitration requirements, and producing a technology independent description of the designed interfaces. It is the implementor's responsibility, in the implementation and integration stage, to produce the final hardware according to a selected technology.

The bus arbitration capability was chosen because it is an integral part of any computer system. The partition was feasible because of the hierarchical methodology that breaks the design problem down not only in levels of hardware abstraction but also in several interfacing capabilities in the components. The design concepts developed for the bus arbitration module applies to the other capabilities used by components to interconnect to one another, such as data transfer and interrupt.

Representation turned to be crucial for the concretization of the ideas in this work. Several facets of representation were delineated:

- Description of components as a hierarchy of capabilities and protocols.
- Description of protocols by action graphs.

• Description of actions in terms of signals.

Altogether these representational issues allow the formulation of the interface design problem in the abstract design space. The generic rules identify the protocols used by the participating components and instantiate the abstract design with the particular information found in the component library.

In summary, the following methodology was proposed:

1. Break down the interface problem by identifying the interfacing capabilities of the involved components and work out the design for each capability.

2. Generate the abstract designs according to the protocols followed by the capabilities. The abstract design can be formulated as the merging of the action graphs corresponding to the protocols.

3. Produce a particular design by instantiating the abstract design with the specific information contained in the protocol description of the participating components.

4. Enter new components by describing their capabilities and protocols as instantiations of the templates that exist in the component library. The actions of the protocol are described in terms of the signals used by the component. The notation described in chapter 4 can be used for this purpose.

7.3 Future Developments

Beside the proposed methodology, the aim of DAME is to configure and design actual systems. The proposed methodology can be applied to complete the DAME designer subsystem that carries out the functional phase of the design process.

Because of lack of time, the developed prototype covered only some arbitration structures: distributed arbitration with fixed-priority (daisy-chain) and distributed arbitration using a fair scheme (similar to a round robin arbitration resolution). Other arbitration structures can be incorporated fairly easily, in particular centralized arbitration with various schemes (fixed priority, round-robin, etc.) and self-arbitration as described in [89]. Also other capabilities, such as data transfer, interrupts, etc., could also be modelled and included into the DAME designer.

An expert system's performance relies heavily on its data acquisition sub-system that will maintain the database updated with the new components available [7, 85]. At this moment, the protocol representation has been incorporated manually into DAME, but it is envisioned that this part will be automated.

It was shown that DAME is a not a shallow system, as the first generation of expert systems, but includes an abstract model of the interface problem. It is tantalizing to increase DAME's knowledge depth by adding to it the capability of manipulating the action graphs directly in order to produce the design. The main problems are the identification of *matching* actions, and how to deal with the merging of graphs with actions whose functions do not have a direct correspondence.

Bibliography.

[1] Alexandridis, N.A., *Microprocessor System Design Concepts*, Computer Science Press, Maryland, 1984.

[2] Baer, J.L., *Computer Systems Architecture*, Computer Science Press, Maryland, 1980.

[3] Baldwin, D., *A Model for Automatic Design of Digital Circuits*, Technical Report 188, Department of Computer Science, University of Rochester, July 1986.

[4] Barbacci, M.R. and Uehara, T., "Computer Hardware Description Languages: The Bridge between Software and Hardware," *Computer*, pp. 6-8, February 1985.

[5] Bell, C.G., and Newell, A., "The PMS and ISP Descriptive Systems for Computer Structures," In *Proceedings AFIPS*, AFIPS Press, Reston, VA., pp. 351-374, 1970.

[6] Birmingham, W.P., *MICON: A Knowledge Based Single Board Computer Designer*, Research Report CMUCAD-83-21, Department of Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburgh, December 1983.

[7] Birmingham, W.P., Gupta, A.P., and Siewiorek, D.P. "The MICON System for Computer Design," In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pp. 135-140, 1989.

[8] Birmingham, W.P., Gupta, A.P., and Siewiorek, D.P., "The MICON System for Computer Design," *IEEE Micro*, pp. 61-67, October 1989.

[9] Birmingham, W.P. and Siewiorek, D.P. "Capturing Designer Expertise: The CGEN System," In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pp. 610-613, 1989.

[10] Blakeslee, T.R., *Digital Design with Standard MSI and LSI Design Techniques for the Microcomputer Age*, John Wiley & Sons, New York, second edition, 1979.

[11] Bochmann, G.V., "Hardware Specification with Temporal Logic: An Example," *IEEE Transactions on Computers*, vol. C-31, no. 3, pp. 223-230, March 1982.

[12] Booth, T.L., *Sequential Machines and Automata Theory*, John Wiley & Sons, New York, 1967.

[13] Brofferio, S.C. and Scire, G., *A Structured Approach to Intra-Digital-System Communication on a Common Path: Principles and Applications*, North-Holland (1982), Microsystems: Architecture, Integration and Use.

[14] Buchanan, B.G. and Feigenbaum, E.A., "DENDRAL and meta-Dendral: Their Applications Dimension," *Artificial Intelligence*, no. 11, pp. 5-24, January 1978.

[15] Butler, W.L., "Hints for Computer Design," *IEEE Software*, pp. 11-28, January 1984.

[16] Chandrasekaran, B. and Mittal, S., *Deep versus Compiled Knowledge Approaches to Diagnostic Problem-Solving*, Academic Press: London (1984), Development in Expert Systems.

[17] Case, P.W., Graff, H.H., and Kloomok, M. "The Recording, Checking, and Printing of Logical Diagrams," In *Proceedings Eastern Joint Computer Conference*, pp. 108-118, 1958.

[18] Chaney, T.J. and Molnar, C.E., "Anomolous Behavior of Synchronizer and Arbiter Circuits," *IEEE Transactions on Computers*, vol. C-22, no. 4, pp. 421-422, April 1973.

[19] Chorafas, D.N., *Systems Architecture & Systems Design*, McGraw-Hill, New York, 1989.

[20] Conte, G. and del Corso, D., *Multi-microprocessor Systems for Real-Time Applications*, D. Reidel, Dordrecht, 1985.

[21] *Developments in Expert Systems*, Academic Press, London, Editor: Coombs, M.J., 1984.

[22] Del Corso, D. and Maddaleno, F., *Extension of Bus Protocols: A Technique for Modular Upgrade of Processing Systems*, North-Holland (1982), Microsystems: Architecture, Integration and Use.

[23] Del Corso, D., Kirmann, H., and Nicoud, J.D., *Microcomputer Buses and Links*, Academic Press, London, 1986.

[24] Dimopoulos, N.J., Lee, H.C., and Galatis, N. "DAME: Automated Design of Microprocessor-based Systems, an Expert Systems Approach," In *Proceedings of the Canadian Conference on Industrial Computer Systems*, Montreal, Canada, pp. 20-1/20-7, May 1986.

[25] Dimopoulos, N.J. and Lee, C.H. "Experiments in Designing with DAME: Design Automation of Microprocessor Based Systems using an Expert Systems Approach," In *Proceedings of the International Computer Symposium 1986*, CCICS, Tainan, Taiwan, pp. 1858-1867, Dec. 1986.

[26] Dimopoulos, N.J., Li, K.F., and Manning, E.G. "DAME: A Rule Based Designer of Microprocessor Based Systems," In *Proceedings of the 2nd International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems*, Tullahoma, Tennessee, pp. 486-492, June 6-9 1989.

[27] Dimopoulos, N.J., Huber, B., Li, K.F., Caughey, D., Escalante, M., Li, D., Burnett, R., and Manning, E. "Modelling Components in DAME," In *Proceedings of the 3rd International Conference on Industrial & Engineering Applications of Artificial Intelligence and Expert Systems*, Charleston, South Carolina, pp. 716-725, July 15-18 1990.

[28] Duda, R.O., Gaschnig, J.G., and Hart, P.E. Model Design in the PROSPECTOR Consultant System for Mineral Exploration. In *Expert Systems in the Micro-Electronic Age*. Edinburgh University Press, Michie, D., Edinburgh, 1979.

[29] Duda, R.O., Hart, P.E., Konolige, K., and Reboh, R., *A Computer-Based Consultant for Mineral Exploration*, Technical Report, SRI International, September 1979.

[30] Escalante, M., Dimopoulos, N.J., Huber, B., Li, K.F., Li, D., and Manning, E.G. "Generic Design Rules for the Design of Microprocessor Based Systems in DAME: Bus Arbitration Subsystem," In *Proceedings of the 1991 IEEE International Symposium on Circuit and Systems*, Singapore, pp. 3166-3169, June 11-14 1991.

[31] Forgy, C.L., *OPS5 User's Manual*, Carnegie-Mellon University, Pittsburgh, 1981.

[32] Friedman, A.D. and Menon, P.R., *Theory & Design of Switching Circuits*, Computer Science Press, California, Digital System Design series, 1975.

[33] Green, D., *Modern Logic Design*, Addison-Wesley, Wokingham, England, 1986.

[34] Gustavson, D.B., "Computer Buses - A Tutorial," *IEEE Micro*, vol. 4, no. 4, pp. 7-22, August 1984.

[35] Hart, P.E., "Direction for AI in the eighties," *SIGART Newsletter*, no. 79, pp. 79, January 1982.

[36] *Building Expert Systems*, Addison-Wesley, Reading, Ma., Editors: Hayes-Roth, F., Waterman, D.A., and Lenat, D.B., 1983.

[37] Hayes, J.P., *Digital System Design and Microprocessors*, McGraw-Hill, New York, Series in Computer Organization and Architecture, 1984.

[38] Hill, F.J. and Peterson, G.R., *Introduction to Switching Theory & Logical Design*, John Wiley & Sons, New York, third edition, 1981.

[39] Hill, F.J. and Peterson, G.R., *Hardware Organization and Design*, John Wiley & Sons, New York, third edition, 1987.

[40] Horbst, E., *Advances in CAD for VLSI*, Springer-Verlag, Amsterdam, Vol. 2, Logic Design and Simulation, 1986.

[41] Horbst, E., Muller, C., and Schwartzel, H., *Design of VLSI circuits*, Springer-Verlag, Berlin, 1987.

[42] Huber, B., Li, K.F., Dimopoulos, N.J., Li, D., Burnett, R., and Manning, E.G. "Modelling Signal Behavior in DAME," In *Proceedings of the1990 Interantional Symposium on Circuits and Systems*, New Orleans, Louisiana, pp. 1497-1500, April 29-May 3 1990.

[43] Huber, B., Escalante, M., Caughey, D., Dimopoulos, N.J., Li, K.F., Li, D., and Manning, E.G. "Microprocessor Components and Signal Behavior Modelling in DAME," In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, Ottawa, pp. 19.4.1-19.4.4, Sept. 1990.

[44] Hudson, D.L. and Estrin, T., "EMERGE: A Data-driven Medical Decision Making Aid," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, pp. 87-91, January 1984.

[45] IEEE, *IEEE Standard for a Versatile Backplane Bus: VMEbus*, IEEE, New York, 1988.

[46] Intel, *Component Data Catalog*, Santa Clara, Ca., January.

[47] Jensen, K. and Wirth, N., *Pascal: User Manual and Report*, Springer-Verlag, 2nd. edition, 1978.

[48] Johnson, J.R. and Kassel, S., *The MULTIBUS Design Guidebook: Structures, Architectures, and Applications*, McGraw-Hill, New York, 1984.

[49] Green, P.E., "Protocol Conversion," *IEEE Transactions on Communications*, vol. COM-34, no. 3, pp. 257-268, March 1986.

[50] Kelly, V.E. "The CRITTER System: Automated Critiquing of Digital Circuit Designs," In *Proceedings of the 21th ACM/IEEE Design Automation Conference*, pp. 419-425, 1984.

[51] Kowalski, T.J., *The VLSI Design Automation Assistant: A Knowledge-Based Expert System*, Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, PA, April 1984.

[52] Krutz, R.L., *Interfacing Techniques in Digital Design with Emphasis on Microprocessors*, Wiley, New York, 1988.

[53] Kulikowski, C.A., "Artificial Intelligence Methods and Systems for Medical Consultation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 464-476, 1980.

[54] Lenat, D.B., Hayes-Roth, F., and Klahr, P., *Cognotive Economy*, Technical Report, HPP-79-15, Computer Science Department, Stanford University, Stanford, CA, 1979.

[55] Liu, Y.C. and Gibson, G.A., *Microcomputer Sustems: The 8086/8088 Family.Architecture, Programming, and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[56] Liu, C.L., *Elements of Discrete Mathematics*, McGraw-Hill, New York, Computer Science series, 2nd. edition, 1985.

[57] Mano, M.M., *Digital Design*, Prentice-Hall, Englewood Cliffs, N.J., 1984.

[58] Martin, W.A. and Fateman, R.J. "The MACSYMA System," In *Proceedings 2nd. Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, pp. 59-75, 1971.

[59] Maruyama, F. and Fujita, M., "Hardware Verification," *Computer*, pp. 22-32, February 1985.

[60] McDermott, J., *R1: An Expert Configurer*, Technical Report, CMU-CS-80-119, Carnegie Mellon University, Pittsburgh, 1980.

[61] McDermott, J. "Domain Knowledge and the Design Process," In *18th Design Automation Conference*, 1981.

[62] McDermott, J., "R1: A Rule-Based Configurer of Computer Systems," *Artificial Intelligence*, vol. 19, pp. 39-88, September 1982.

[63] Mead, C. and Conway, L., *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachussetts, Series in Computer Science, 1980.

[64] *Expert Systems 85*, The University Press, Cambridge, Editor: Merry, M., 1985.

[65] Michie, D., "High-road and low-road programs," *AI Magazine*, vol. 3, no. 1, pp. 21-22, 1982.

[66] Moszkowski, B., "A Temporal Logic for Multilevel Reasoning about Hardware," *Computer*, pp. 10-19, February 1985.

[67] Motorola, *MC68000 16-bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, New Jersey, third edition, 1982.

[68] Motorola, *MC68020 32-bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[69] Murata, T., *Petri Nets and Their Application*, Plenum Press: New York (1984), Management and Office Information Systems, pp. , 351-367, Chapter 20.

[70] Norton, S.W. and Kelly, K.M. "Learning Preference Rules for a VLSI Design Problem Solver," In *Proceedings of the Fourth Conference on Artificial Intelligence Applications*, pp. 152-158, March 1988.

[71] Parker, A.C., "Automated Synthesis of Digital Systems," *IEEE Design and Test*, pp. 75-81, November 1984.

[72] Parker, A.C. and Hayati, S., "Automating the VLSI Design Process Using Expert Systems and Silicon Compilation," *Proceedings of the IEEE*, vol. 75, no. 6, pp. 777-785, June 1987.

[73] *The Foundations of Artificial Intelligence*, Cambridge University Press, Cambridge, Editors: Partridge, D. and Wilks, Y., 1990.

[74] Pawlak, A. "Microprocessor Systems Modeling with MODLAN," In *20th Design Automation Conference*, IEEE, pp. 804-811, 1983.

[75] Peels, A.J.H.M., *On the Design of Digital Systems*, North-Holland (1982), Microsystems: Architecture, Integration and Use.

[76] Petterson, J.L., "Petri Nets," *Computing Surveys*, vol. 9, no. 3, pp. 223-252, September 1977.

[77] Piloty, R. and Borrione, D., "The Conlan Project: Concepts, Implementations, and Applications," *Computer*, pp. 81-92, February 1985.

[78] Ramamoorthy, C.V. and Ho, G.S., "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 5, pp. 440-449, September 1980.

[79] Rector, R. and Alexy, G., *The 8086 Book*, OSBORNE/McGraw-Hill, Berkeley, California, 1980.

[80] Ronald, C.G., *PECOS - An Expert Hardware Synthesis Systems*, Technical Report, Technical Report, US Army Research Office, Triangle Park, NC, 1985.

[81] Seitz, C.L., "Ideas about Arbiters," *Lambda*, pp. 10-14, First Quarter 1980.

[82] Shahdad, M., Lipsett, R., Marschner, E., Sheehan, K., Cohen, H., Waxman, R., and Ackley, D., "VHSIC Hardware Description Language," *Computer*, pp. 94-103, February 1985.

[83] Shortliffe, E.H., *Computer-based Medical Consultation: MYCIN*, American Elsevier, New York, 1976.

[84] Siddall, J.N., *Expert Systems for Engineers*, Marcel Dekker, New York, 1990.

[85] Smith, M.F. and Bowen, J.A., "Knowledge and Experience-based Systems for Analysis and Design of Microprocessor Applications Hardware," *Microprocessors and Microsystems*, vol. 6, no. 10, pp. 515-518, December 1982.

[86] Stallman, R. and Sussman, G.J., "Forward Reasoning and Dependency-directed Backtracking in a System for Computer-aided Circuit Analysis," *Artificial Intelligence*, no. 9, pp. 135-196, 1977.

[87] Stone, H.S., *Microcomputer Interfacing*, Addison-Wesley, Reading, Massachussetts, 1982.

[88] Tanimoto, S.L., *The Elements of Artificial Intelligence*, Computer Science Press, Maryland, 1987.

[89] Taub, D.M., "Arbitration and Control Acquisition in the Proposed IEEE 896 Futurebus," *IEEE Micro*, vol. 4, no. 4, pp. 28-41, August 1984.

[90] Uehara, T., "A Knowledge-Based Logic Design System," *IEEE Design & Test*, vol. 2, no. 5, pp. 27-34, October 1985.

[91] Unger, S.H., *The Essence of Logic Circuits*, Prentice-Hall, Englewoods Cliffs, New Jersey, 1989.

[92] Van Cleemput, W.M. and Ofek, H., "Design Automation for Digital Systems," *Computer*, pp. 114-122, October 1984.

[93] Vanbekbergen, P., Catthoor, F., van Meerbergen, J., and de Man, H., *Race-Free Time-Optimised Synthesis of Asynchronous Interface Circuits*, 1989, IMEC Lab., Leuven, Belgium and Philips Research Labs, Eindhoven, the Netherlands.

```
((STATE-B (INPUT1-A INPUT1-B)) STATE-C)
((STATE-C (INPUT0-A INPUT1-B)) STATE-F)
((STATE-C (INPUT1-A INPUT0-B)) STATE-D)
((STATE-D (INPUT0-A INPUT0-B)) STATE-A)
((STATE-D (INPUT1-A INPUT1-B)) STATE-E)
((STATE-E (INPUT0-A INPUT1-B)) STATE-G)
((STATE-E (INPUT1-A INPUT0-B)) STATE-D)
((STATE-F (INPUT0-A INPUT0-B)) STATE-A)
((STATE-F (INPUT1-A INPUT1-B)) STATE-C)
((STATE-G (INPUT0-A INPUT0-B)) STATE-A)
((STATE-G (INPUT1-A INPUT1-B)) STATE-C))))

(DEFSCHEMA PRIORITY-INTERRUPT-4
        (IS-A PRIORITY-INTERRUPT))

(DEFSCHEMA UTILITY-4
        (IS-A UTILITY))
```

A.2 DMA Device Bus Arbitration Frames.

```
(DEFSCHEMA DMA-DEVICE
        (IS-A IO-DEVICE)
        (HAS-CAPABILITY
                DATA-TRANSFER-3 BUS-ARBITRATION-2
                PRIORITY-INTERRUPT-3 UTILITY-3)
        (HAS-SIGNAL
        DMA-DEVICE-HREQ DMA-DEVICE-HACK
        DMA-DEVICE-D0  DMA-DEVICE-D1 DMA-DEVICE-D7 DMA-DEVICE-D8
        DMA-DEVICE-A1 DMA-DEVICE-A2 DMA-DEVICE-A3 DMA-DEVICE-A4
        DMA-DEVICE-A5)
        (HAS-BUS
        DMA-DEVICE-DATA-BUS DMA-DEVICE-ADDRESS-BUS
        DMA-DEVICE-CONTROL-BUS DMA-DEVICE-DATA-TRANSFER-BUS))

(DEFSCHEMA DMA-DEVICE-HREQ
        (IS-A SIGNAL)
        (NAME HREQ)
        (PIN-NUMBER )
        (I-O  OUTPUT))
```

```
(DEFSCHEMA DMA-DEVICE-HACK
        (IS-A SIGNAL)
        (NAME HACK)
        (PIN-NUMBER )
        (I-O  INPUT))

(DEFSCHEMA DMA-DEVICE-A1
        (IS-A SIGNAL)
        (NAME A1)
        (PIN-NUMBER 1)
        (I-O  INPUT))

(DEFSCHEMA DMA-DEVICE-A2
        (IS-A SIGNAL)
        (NAME A2)
        (PIN-NUMBER 2)
        (I-O  INPUT))

(DEFSCHEMA DMA-DEVICE-A3
        (IS-A SIGNAL)
        (NAME A3)
        (PIN-NUMBER 3)
        (I-O  INPUT))

(DEFSCHEMA DMA-DEVICE-A4
        (IS-A SIGNAL)
        (NAME A4)
        (PIN-NUMBER 4)
        (I-O  INPUT))

(DEFSCHEMA DMA-DEVICE-A5
        (IS-A SIGNAL)
        (NAME A5)
        (PIN-NUMBER 5)
        (I-O  INPUT))

(DEFSCHEMA DMA-DEVICE-D0
        (IS-A SIGNAL)
        (NAME D0)
        (PIN-NUMBER )
        (I-O  INPUT OUTPUT))

(DEFSCHEMA DMA-DEVICE-D1
        (IS-A SIGNAL)
        (NAME D1)
        (PIN-NUMBER )
        (I-O  INPUT OUTPUT))
```

```
(DEFSCHEMA DMA-DEVICE-D7
        (IS-A SIGNAL)
        (NAME D7)
        (PIN-NUMBER )
        (I-O  INPUT OUTPUT))

(DEFSCHEMA DMA-DEVICE-D8
        (IS-A SIGNAL)
        (NAME D8)
        (PIN-NUMBER )
        (I-O  INPUT OUTPUT))


(DEFSCHEMA DMA-DEVICE-DATA-TRANSFER-BUS
        (IS-A SIGNAL-BUS)
        (HAS-SIGNAL
         DMA-DEVICE-D0 DMA-DEVICE-D1 DMA-DEVICE-D7 DMA-DEVICE-D8
         DMA-DEVICE-A1 DMA-DEVICE-A2 DMA-DEVICE-A3 DMA-DEVICE-A4
         DMA-DEVICE-A5))

(DEFSCHEMA DMA-DEVICE-DATA-BUS
        (IS-A SIGNAL-BUS)
        (HAS-SIGNAL DMA-DEVICE-D0 DMA-DEVICE-D1 DMA-DEVICE-D7
                DMA-DEVICE-D8))

(DEFSCHEMA DMA-DEVICE-ADDRESS-BUS
        (IS-A SIGNAL-BUS)
        (HAS-SIGNAL DMA-DEVICE-A1 DMA-DEVICE-A2 DMA-DEVICE-A3
                DMA-DEVICE-A4 DMA-DEVICE-A5))

(DEFSCHEMA DMA-DEVICE-CONTROL-BUS
        (IS-A SIGNAL-BUS)
        (HAS-SIGNAL DMA-DEVICE-HREQ DMA-DEVICE-DHACK))


(DEFSCHEMA DATA-TRANSFER-3
        (IS-A  DATA-TRANSFER)
        (USES-EXTRA-TIMING)
        (USES-WRITE-PROTOCOL)
        (USES-READ-PROTOCOL))


(DEFSCHEMA BUS-ARBITRATION-2
        (IS-A BUS-ARBITRATION)
         (USES-PROTOCOL HANDSHAKE-PROTOCOL-1))
```

```
(DEFSCHEMA HANDSHAKE-PROTOCOL-1
        (IS-A HANDSHAKE-PROTOCOL)
        (ROLE REQUESTER)
        (HAS-ACTION
                ACTION-1 ACTION-2 ACTION-3 ACTION-4
                ACTION-5 ACTION-6))


(DEFSCHEMA ACTION-1
        (IS-A ACTION)
        (FUNCTION   REQUEST)
        (TYPE OUTPUT)
        (HAS-DESCRIPTION (! ASSERTED DMA-DEVICE-HREQ))
        (PRECEDES   ACTION-2))

(DEFSCHEMA ACTION-2
        (IS-A ACTION)
        (FUNCTION   ACK)
        (TYPE INPUT)
        (HAS-DESCRIPTION (! ASSERTED DMA-DEVICE-HACK))
        (PRECEDES   ACTION-3))

(DEFSCHEMA ACTION-3
        (IS-A ACTION)
        (FUNCTION   BUS-BUSY)
        (TYPE RESOURCE)
        (HAS-DESCRIPTION )
        (PRECEDES   ACTION-4))

(DEFSCHEMA ACTION-4
        (IS-A ACTION)
        (FUNCTION   BUS-FREE)
        (TYPE RESOURCE)
        (HAS-DESCRIPTION )
        (PRECEDES   ACTION-5))

(DEFSCHEMA ACTION-5
        (IS-A ACTION)
        (FUNCTION   NOT-REQUEST)
        (TYPE OUTPUT)
        (HAS-DESCRIPTION (! NEGATED DMA-DEVICE-HREQ))
        (PRECEDES   ACTION-6))

(DEFSCHEMA ACTION-6
        (IS-A ACTION)
        (FUNCTION   NOT-ACK)
        (TYPE INPUT)
        (HAS-DESCRIPTION (! NEGATED DMA-DEVICE-HACK))
        (PRECEDES   ACTION-2))
```

```
(DEFSCHEMA PRIORITY-INTERRUPT-3
        (IS-A PRIORITY-INTERRUPT))

(DEFSCHEMA UTILITY-3
        (IS-A UTILITY))
```

Appendix B. Rules in the Bus Arbitration Design Interface Module.

The rules written in CRL-OPS that implement the bus arbitration interface design module are described in this appendix. Only the daisy-chain structured is considered.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Bus Arbitration interface design
;;; Daisy chain cluster
;;;
;;; Marco Antonio Escalante
;;; July 22, 91
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(in-package 'crl-user)
;;; Specificity-only conflict resolution strategy
(load "/home/dame/designer/strategy")
(reset-ops :strategy spec-only)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; CRL-OPS rules that generate the asynchronous state-machine
;;; description of the interface for the Bus Arbitration (BA)
;;; control signals of 2 devices with a 2- or 3-wire BA protocol
;;; in the arbiter and
;;; a handshake (2-wire) BA protocol in the requestor.
;;;
;;; Daisy chain version
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Declaration of the Schema Memory Elements
;;;
;;; These slots will contain a list of elements
;;; (instead of a single value)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(list-attribute has-capability has-block has-signal requestors)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Schema Definition
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Input Schemata
(schema-literalize bus-arbitration-spec instance requestors arbiter fairness)
(schema-literalize component has-signal has-capability)
(schema-literalize handshake-protocol)
(schema-literalize three-signal-protocol)
(schema-literalize step phase)

;;; Output Schemata
(schema-literalize interface-block type
 cluster spec priority
 arbiter arbiter-protocol requestor requestor-protocol
 has-block has-signal
 grant-out fairness merged-grant-ack sync)
(schema-literalize signal name)
(schema-literalize block function action)

;;; Internal Schemata
;;; Create signal
(schema-literalize notebook type
;;; Interface blocks
  cluster spec component role protocol fairness sync
;;; Action filler
  block function direction)
(schema-literalize priority-counter spec count)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; CREATE-NOTEBOOK rule
;;;
;;; This rule creates the notebook for schemata, instance of
;;; COMPONENT, that has a BUS-ARBITRATION capability
;;; and it is included in the BUS-ARBITRATION-SPEC
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(p create-BA-arbiter-notebook
  :specificity 30
  (bus-arbitration-spec ^schema-name <ba-spec> ^arbiter <name>
        ^instance <arb-structure> ^fairness <boolean>)
  (component ^schema-name <name> ^has-capability {<ba> (is-BA <>)} )
  (step ^phase bus-arbitration-interface)
-->
 (let
  ((ba-prot
         (get-value-if (is-ba <ba>) 'uses-protocol
          #'(lambda (x) (equal (get-value x 'role) 'arbiter))
          :start 0 :end nil)))
  (cschema
        (create-name 'notebook)
        ('instance                    'notebook)
```

```
        ('type                          'ba-notebook)
        ('cluster                       <arb-structure>)
        ('spec                          <ba-spec>)
        ('component                     <name>)
        ('role                          'arbiter)
        ('protocol                      ba-prot)
        ('sync                          (when (local-slot-p ba-prot 'clock)
                                              (get-value ba-prot 'clock)))
        ('fairness                      <boolean>)))
)

(p create-BA-requestor-notebook
  :specificity 30
  (bus-arbitration-spec ^schema-name <ba-spec> ^requestors <reqs>)
  (component ^schema-name {<name> (member <> <reqs>)}
        ^has-capability {<ba> (is-BA <>)} )
  (step ^phase bus-arbitration-interface)
-->
  (let
   ((ba-prot
         (get-value-if (is-ba <ba>) 'uses-protocol
          #'(lambda (x) (equal (get-value x 'role) 'requestor))
          :start 0 :end nil)))
   (cschema (create-name 'notebook)
        ('instance                      'notebook)
        ('type                          'ba-notebook)
        ('spec                          <ba-spec>)
        ('component                     <name>)
        ('role                          'requestor)
        ('protocol                      ba-prot)
        ('sync                          (when (local-slot-p ba-prot 'clock)
                                              (get-value ba-prot 'clock)))))
)

(p create-BA-notebook-demon
  :specificity 32
  (notebook ^component <comp> ^type ba-notebook ^protocol nil)
  (step ^schema-name <step> ^phase bus-arbitration-interface)
-->
  (print <comp> 'does 'not 'have 'bus 'arbitration 'capability)
  (new-value <step> 'phase 'error)
)

(defun is-BA (list)
  (cond
   ((null list) nil)
   ((relatedp (car list) 'is-a 'bus-arbitration) (car list))
   (t (is-BA (cdr list)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; BUS ARBITRATION DESIGN INTERFACE
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; These rules design the interface block between
;;; a processor/arbiter and an io-device/requestor
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Priority counter
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p create-BA-priority-counter
  (notebook ^cluster <cluster> ^spec <ba-spec> ^role arbiter)
 -(priority-counter ^schema-name <counter> ^spec <ba-spec>)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (create-name 'priority-counter)
   ('instance     'priority-counter)
   ('spec         <ba-spec>)
   ('count        1))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Interface block for each requestor
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p create-BA-interface-block
  :specificity 29
  (notebook ^cluster <cluster> ^spec <ba-spec> ^component <name1>
        ^role arbiter ^protocol <prot1> ^fairness <boolean> ^sync <clock1>)
  (notebook ^spec <ba-spec> ^component <name2>
        ^role requestor ^protocol <prot2> ^sync <clock2>)
  (priority-counter ^schema-name <counter> ^spec <ba-spec> ^count <num>)
 -(interface-block ^spec <ba-spec> ^arbiter <name1> ^requestor <name2>)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (create-name 'interface-block)
   ('instance                'interface-block)
   ('cluster                 <cluster>)
   ('spec                    <ba-spec>)
   ('priority                <num>)
   ('arbiter                 <name1>)
   ('arbiter-protocol    <prot1>)
   ('requestor               <name2>)
   ('requestor-protocol <prot2>)
   ('fairness                <boolean>)
   ('grant-out               'non)
   ('sync                    (if <clock1> <clock1> <clock2>)))
  (cschema <counter>
   ('count                   (+ <num> 1)))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; BUS ARBITRATION DESIGN INTERFACE
;;; DAISY CHAIN CLUSTER
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p daisy-chain-basic-info
  :specificity 29
  (interface-block ^schema-name <block> ^grant-out non ^cluster daisy-chain
        ^arbiter-protocol <prot>)
  (step ^phase bus-arbitration-interface)
-->
  (cschema <block>
   ('grant-out              (get-action <prot> 'grant-out)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; BUS ARBITRATION DESIGN INTERFACE
;;; DAISY CHAIN CLUSTER
;;; 2-to-2-wire BA protocol case
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p daisy-chain-2to2-interface
  :specificity 28
  (interface-block ^schema-name <block> ^cluster daisy-chain
        ^arbiter-protocol <prot1> ^requestor-protocol <prot2>)
  (handshake-protocol ^schema-name <prot1>)
  (handshake-protocol ^schema-name <prot2>)
  (step ^phase bus-arbitration-interface)
-->
  (cschema <block>
   ('cluster        'daisy-chain-2to2)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; BUS ARBITRATION DESIGN INTERFACE
;;; DAISY CHAIN CLUSTER
;;; 2-to-3-wire BA protocol case
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p daisy-chain-2to3-interface
  :specificity 28
  (interface-block ^schema-name <block> ^cluster daisy-chain
        ^arbiter-protocol <prot1> ^requestor-protocol <prot2>)
  (three-signal-protocol ^schema-name <prot1>)
  (handshake-protocol ^schema-name <prot2>)
  (step ^phase bus-arbitration-interface)
-->
  (cschema <block>
   ('cluster                          'daisy-chain-2to3)
   ('merged-grant-ack  (action-equivalence
                              (get-action <prot1> 'bus-busy)
                              (get-action <prot1> 'grant-ack)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; BUS ARBITRATION DESIGN INTERFACE
;;; DAISY CHAIN CLUSTER
;;; 2-to-2-wire BA protocol case
;;; INTERFACE FILLER NOTEBOOKS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; REQUEST-OUT Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to2-req-out-notebook
  :specificity 20
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to2
        ^arbiter <name1> ^arbiter-protocol <prot1>)
 -(notebook ^block <block> ^function req-out)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block> 'req-out-notebook)
   ('instance          'notebook)
   ('block                    <block>)
   ('function          'req-out)
   ('direction         'output)
   ('active                   (get-event <name1> <prot1> 'request))
   ('inactive          (get-event <name1> <prot1> 'not-request)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; COMMON REQUEST Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to2-c-req-notebook
  :specificity 20
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to2
        ^arbiter <name1> ^arbiter-protocol <prot1> ^fairness <> nil)
 -(notebook ^block <block> ^function c-req)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block> 'c-req-notebook)
   ('instance          'notebook)
   ('block                    <block>)
   ('function          'c-req)
   ('direction         'input)
   ('active                   (get-event <name1> <prot1> 'request))
   ('inactive          (get-event <name1> <prot1> 'not-request)))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; REQUEST-IN Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to2-req-in-notebook
  :specificity 20
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to2
        ^requestor <name2> ^requestor-protocol <prot2>)
 -(notebook ^block <block> ^function req-in)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block> 'req-in-notebook)
   ('instance          'notebook)
   ('block             <block>)
   ('function          'req-in)
   ('direction         'input)
   ('active            (get-event <name2> <prot2> 'request))
   ('inactive          (get-event <name2> <prot2> 'not-request)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; ACK Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to2-ack-notebook
  :specificity 20
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to2
        ^requestor <name2> ^requestor-protocol <prot2>)
 -(notebook ^block <block> ^function ack)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block> 'ack-notebook)
   ('instance          'notebook)
   ('block             <block>)
   ('function          'ack)
   ('direction         'output)
   ('active            (get-event <name2> <prot2> 'ack))
   ('inactive          (get-event <name2> <prot2> 'not-ack)))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; GRANT-IN &
;;; GRANT-OUT Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to2-grant-notebook-1a
  :specificity 21
  (interface-block ^schema-name <block1> ^cluster daisy-chain-2to2
        ^spec <ba-spec> ^priority {<num> > 1}
        ^arbiter <name1> ^arbiter-protocol <prot1> ^grant-out <> nil)
 -(notebook ^block <block1> ^function grant-in)
 -(interface-block ^cluster daisy-chain-2to2 ^spec <ba-spec>
        ^priority (lower-priority <> <num>))
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block1> 'grant-in-notebook)
   ('instance          'notebook)
   ('block             <block1>)
   ('function          'grant-in)
   ('direction         'input))
  (cschema (make-name <block1> 'grant-out-notebook)
   ('instance          'notebook)
   ('block             <block1>)
   ('function          'grant-out)
   ('direction         'output)
   ('active            (get-event <name1> <prot1> 'grant-out))
   ('inactive          (get-event <name1> <prot1> 'not-grant-out)))
)

(p design-2to2-grant-notebook-1b
  :specificity 21
  (interface-block ^schema-name <block1> ^cluster daisy-chain-2to2
        ^spec <ba-spec> ^priority {<num> > 1})
 -(notebook ^block <block1> ^function grant-in)
 -(interface-block ^cluster daisy-chain-2to2 ^spec <ba-spec>
        ^priority (lower-priority <> <num>))
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block1> 'grant-in-notebook)
   ('instance          'notebook)
   ('block             <block1>)
   ('function          'grant-in)
   ('direction         'input))
)
```

```
(p design-2to2-grant-notebook-2
  :specificity 20
  (interface-block ^schema-name <block1> ^cluster daisy-chain-2to2
        ^spec <ba-spec> ^priority {<num> > 1})
 -(notebook ^block <block1> ^function grant-in)
  (interface-block ^schema-name <block3> ^cluster daisy-chain-2to2
        ^spec <ba-spec> ^priority (lower-priority <> <num>))
  (notebook ^schema-name <nb3> ^block <block3> ^function grant-in)
  (step ^phase bus-arbitration-interface)
-->
 (let ((signal (make-name <block1> 'grant-out)))
  (cschema signal
   ('instance          'signal)
   ('name              'grant-out)
   ('has-signal+inv    <block1>))
  (cschema <nb3>
   ('active            (list '! 'asserted signal))
   ('inactive          (list '! 'negated  signal)))
  (cschema (make-name <block1> 'grant-in-notebook)
   ('instance          'notebook)
   ('block             <block1>)
   ('function          'grant-in)
   ('direction         'input))
  (cschema (make-name <block1> 'grant-out-notebook)
   ('instance          'notebook)
   ('block             <block1>)
   ('function          'grant-out)
   ('direction         'output)
   ('active            (list '! 'asserted signal))
   ('inactive          (list '! 'negated  signal))
   ('internal-active   (list '! 'asserted signal))
   ('internal-inactive (list '! 'negated  signal)))
))

(p design-2to2-grant-notebook-3
  :specificity 20
  (interface-block ^schema-name <block1> ^cluster daisy-chain-2to2
        ^spec <ba-spec> ^priority 1
        ^arbiter <name1> ^arbiter-protocol <prot1>)
 -(notebook ^block <block1> ^function grant-in)
  (interface-block ^schema-name <block3> ^cluster daisy-chain-2to2
        ^spec <ba-spec> ^priority 2)
  (notebook ^schema-name <nb3> ^block <block3> ^function grant-in)
  (step ^phase bus-arbitration-interface)
-->
 (let ((signal (make-name <block1> 'grant-out)))
  (cschema signal
   ('instance          'signal)
   ('name              'grant-out)
   ('has-signal+inv    <block1>))
```

```
 (cschema <nb3>
  ('active              (list '! 'asserted signal))
  ('inactive            (list '! 'negated  signal)))
 (cschema (make-name <block1> 'grant-in-notebook)
  ('instance            'notebook)
  ('block               <block1>)
  ('function            'grant-in)
  ('direction           'input)
  ('active              (get-event <name1> <prot1> 'ack))
  ('inactive            (get-event <name1> <prot1> 'not-ack)))
 (cschema (make-name <block1> 'grant-out-notebook)
  ('instance            'notebook)
  ('block               <block1>)
  ('function            'grant-out)
  ('direction           'output)
  ('active              (list '! 'asserted signal))
  ('inactive            (list '! 'negated  signal))
   ('internal-active    (list '! 'asserted signal))
   ('internal-inactive  (list '! 'negated  signal)))
))

(p design-2to2-grant-notebook-4a
 :specificity 21
 (interface-block ^schema-name <block1> ^cluster daisy-chain-2to2
       ^spec <ba-spec> ^priority 1
       ^arbiter <name1> ^arbiter-protocol <prot1> ^grant-out <> nil)
 -(notebook ^block <block1> ^function grant-in)
 -(interface-block ^cluster daisy-chain-2to2 ^spec <ba-spec>
       ^priority 2)
 (step ^phase bus-arbitration-interface)
-->
 (cschema (make-name <block1> 'grant-in-notebook)
  ('instance            'notebook)
  ('block               <block1>)
  ('function            'grant-in)
  ('direction           'input)
  ('active              (get-event <name1> <prot1> 'ack))
  ('inactive            (get-event <name1> <prot1> 'not-ack)))
 (cschema (make-name <block1> 'grant-out-notebook)
  ('instance            'notebook)
  ('block               <block1>)
  ('function            'grant-out)
  ('direction           'output)
  ('active              (get-event <name1> <prot1> 'grant-out))
  ('inactive            (get-event <name1> <prot1> 'not-grant-out)))
)
```

```
(p design-2to2-grant-notebook-4b
  :specificity 20
  (interface-block ^schema-name <block1> ^cluster daisy-chain-2to2
        ^spec <ba-spec> ^priority 1
        ^arbiter <name1> ^arbiter-protocol <prot1>)
 -(notebook ^block <block1> ^function grant-in)
 -(interface-block ^cluster daisy-chain-2to2 ^spec <ba-spec>
        ^priority 2)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block1> 'grant-in-notebook)
   ('instance          'notebook)
   ('block             <block1>)
   ('function          'grant-in)
   ('direction         'input)
   ('active            (get-event <name1> <prot1> 'ack))
   ('inactive          (get-event <name1> <prot1> 'not-ack)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; FAIRNESS Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to2-fairness-notebook
  :specificity 20
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to2
        ^fairness <> nil)
 -(notebook ^block <block> ^function fairness)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block> 'fairness-notebook)
   ('instance          'notebook)
   ('block             <block>)
   ('function          'fairness)
   ('direction         'internal))
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; BUS ARBITRATION DESIGN INTERFACE
;;; DAISY CHAIN CLUSTER
;;; 2-to-2-wire BA protocol case
;;; ASM DESIGN
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; INTERFACE INPUT-SIGNAL ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to2-interface-input-signal-asm
 :specificity 6
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to2
        ^has-block <blks>)
  (notebook ^schema-name <nb> ^type asm ^block <block>
        ^function <fn> ^direction input)
 -(block ^schema-name (member <> <blks>) ^function <fn>)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (create-name 'two-state-asm)
   ('instance          'two-state-asm)
   ('function          <fn>)
   ('has-block+inv     <block>)
   ('input0            (get-value <nb>      'inactive))
   ('input1            (get-value <nb>      'active))
   ('output0           (get-value <nb>      'internal-inactive))
   ('output1           (get-value <nb>      'internal-active)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; INTERFACE INTERNAL-SIGNAL ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; FAIRNESS ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to2-fairness-asm
 :specificity 8
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to2
        ^has-block <blks>)
 -(block ^schema-name (member <> <blks>) ^function fairness)
  (notebook ^schema-name <fair>   ^block <block> ^function fairness)
  (notebook ^schema-name <c-req>  ^block <block> ^function c-req)
  (notebook ^schema-name <req-in> ^block <block> ^function req-in)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (create-name 'two-state-asm)
   ('instance          'two-state-asm)
   ('function          'fairness)
   ('has-block+inv     <block>)
   ('input0            (get-value <c-req>   'internal-inactive))
   ('input1            (get-value <req-in>  'internal-active))
   ('output0           (get-value <fair>    'internal-inactive))
   ('output1           (get-value <fair>    'internal-active)))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; INTERFACE OUTPUT-SIGNAL ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; REQUEST-OUT ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to2-request-asm-1a
  :specificity 5
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to2
        ^has-block <blks> ^fairness <> nil)
  (block ^schema-name {<asm> (member <> <blks>)} ^function request)
  (notebook ^schema-name <fair> ^block <block> ^function fairness)
  (step ^phase bus-arbitration-interface)
-->
  (new-value <asm>
   'input1
   (add-signal '^ (get-state (get-value <fair> 'internal-inactive))
        (get-value <asm> 'input1)))
)

(p design-2to2-request-asm-1b
  :specificity 5
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to2
        ^has-block <blks>)
 -(block ^schema-name (member <> <blks>) ^function request)
  (notebook ^schema-name <req-out> ^block <block> ^function req-out)
  (notebook ^schema-name <req-in>  ^block <block> ^function req-in)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (create-name 'two-state-asm)
   ('instance          'two-state-asm)
   ('function          'request)
   ('has-block+inv     <block>)
   ('input0            (get-value <req-in>       'internal-inactive))
   ('input1            (get-value <req-in>       'internal-active))
   ('output0           (get-value <req-out>  'internal-inactive))
   ('output1           (get-value <req-out>  'internal-active)))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; ACK & GRANT-OUT ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to2-ack-asm
  :specificity 7
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to2
        ^has-block <blks>)
 -(block ^schema-name (member <> <blks>) ^function grant)
  (notebook ^schema-name <ack> ^block <block> ^function grant)
  (notebook ^schema-name <grant-in> ^block <block> ^function grant-in)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (create-name 'connect-block)
   ('instance          'connect-block)
   ('has-block+inv     <block>)
   ('function          'grant)
   ('input0            (get-value <grant-in> 'internal-inactive))
   ('input1            (get-value <grant-in> 'internal-active))
   ('output0           (get-value <ack>      'internal-inactive))
   ('output1           (get-value <ack>      'internal-active)))
)

(p design-2to2-ack-and-grant-out-asm
  :specificity 8
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to2
        ^has-block <blks>)
 -(block ^schema-name (member <> <blks>) ^function grant)
  (notebook ^schema-name <grant-out> ^block <block> ^function grant-out)
  (notebook ^schema-name <ack>       ^block <block> ^function grant)
  (notebook ^schema-name <req-in>    ^block <block> ^function req-in)
  (notebook ^schema-name <grant-in>  ^block <block> ^function grant-in)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (create-name 'mutual-exclusion-asm)
   ('instance          'mutual-exclusion-asm)
   ('function          'grant 'grant-out)
   ('has-block+inv     <block>)
   ('condition-A       (get-state (get-value <req-in> 'internal-active)))
   ('condition-B(get-state (get-value <req-in> 'internal-inactive)))
   ('input0            (get-value <grant-in> 'internal-inactive))
   ('input1            (get-value <grant-in> 'internal-active))
   ('output0-A         (get-value <ack>             'internal-inactive))
   ('output1-A         (get-value <ack>             'internal-active))
   ('output0-B         (get-value <grant-out>       'internal-inactive))
   ('output1-B         (get-value <grant-out>       'internal-active)))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; BUS ARBITRATION DESIGN INTERFACE
;;; DAISY CHAIN CLUSTER
;;; 2-to-3-wire BA protocol case
;;; INTERFACE FILLER NOTEBOOKS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; BUS-FREE Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-bus-free-notebook
  :specificity 20
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^arbiter <name1> ^arbiter-protocol <prot1>)
 -(notebook ^block <block> ^function bus-free)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block> 'bus-free-notebook)
   ('instance          'notebook)
   ('block             <block>)
   ('function          'bus-free)
   ('direction         'input)
   ('active            (get-event <name1> <prot1> 'bus-free))
   ('inactive          (get-event <name1> <prot1> 'bus-busy)))
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; REQUEST-OUT Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-req-out-notebook
  :specificity 20
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^arbiter <name1> ^arbiter-protocol <prot1>)
 -(notebook ^block <block> ^function req-out)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block> 'req-out-notebook)
   ('instance          'notebook)
   ('block             <block>)
   ('function          'req-out)
   ('direction         'output)
   ('active            (get-event <name1> <prot1> 'request))
   ('inactive          (get-event <name1> <prot1> 'not-request)))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; COMMON REQUEST Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-c-req-notebook
  :specificity 20
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^arbiter <name1> ^arbiter-protocol <prot1> ^fairness <> nil)
 -(notebook ^block <block> ^function c-req)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block> 'c-req-notebook)
   ('instance          'notebook)
   ('block             <block>)
   ('function          'c-req)
   ('direction         'input)
   ('active            (get-event <name1> <prot1> 'request))
   ('inactive          (get-event <name1> <prot1> 'not-request)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; REQUEST-IN Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-req-in-notebook
  :specificity 20
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^requestor <name2> ^requestor-protocol <prot2>)
 -(notebook ^block <block> ^function req-in)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block> 'req-in-notebook)
   ('instance          'notebook)
   ('block             <block>)
   ('function          'req-in)
   ('direction         'input)
   ('active            (get-event <name2> <prot2> 'request))
   ('inactive          (get-event <name2> <prot2> 'not-request)))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; ACK Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-ack-notebook
  :specificity 20
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^requestor <name2> ^requestor-protocol <prot2>)
 -(notebook ^block <block> ^function ack)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block> 'ack-notebook)
   ('instance          'notebook)
   ('block             <block>)
   ('function          'ack)
   ('direction         'output)
   ('active            (get-event <name2> <prot2> 'ack))
   ('inactive          (get-event <name2> <prot2> 'not-ack)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; GRANT-ACK Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-grant-ack-notebook
  :specificity 20
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^arbiter <name1> ^arbiter-protocol <prot1>)
 -(notebook ^block <block> ^function grant-ack)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block> 'grant-ack-notebook)
   ('instance          'notebook)
   ('block             <block>)
   ('function          'grant-ack)
   ('direction         'output)
   ('active            (get-event <name1> <prot1> 'grant-ack))
   ('inactive          (get-event <name1> <prot1> 'not-grant-ack)))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; GRANT-IN &
;;; GRANT-OUT Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-grant-notebook-1a
  :specificity 22
  (interface-block ^schema-name <block1> ^cluster daisy-chain-2to3
         ^spec <ba-spec> ^priority {<num> > 1}
         ^arbiter <name1> ^arbiter-protocol <prot1> ^grant-out <> nil)
 -(notebook ^block <block1> ^function grant-in)
 -(interface-block ^cluster daisy-chain-2to3 ^spec <ba-spec>
         ^priority (lower-priority <> <num>))
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block1> 'grant-in-notebook)
   ('instance             'notebook)
   ('block                <block1>)
   ('function             'grant-in)
   ('direction            'input))
  (cschema (make-name <block1> 'grant-out-notebook)
   ('instance             'notebook)
   ('block                <block1>)
   ('function             'grant-out)
   ('direction            'output)
   ('active               (get-event <name1> <prot1> 'grant-out))
   ('inactive             (get-event <name1> <prot1> 'not-grant-out)))
)

(p design-2to3-grant-notebook-1b
  :specificity 21
  (interface-block ^schema-name <block1> ^cluster daisy-chain-2to3
         ^spec <ba-spec> ^priority {<num> > 1})
 -(notebook ^block <block1> ^function grant-in)
 -(interface-block ^cluster daisy-chain-2to3 ^spec <ba-spec>
         ^priority (lower-priority <> <num>))
  (step ^phase bus-arbitration-interface)
-->
  (cschema (make-name <block1> 'grant-in-notebook)
   ('instance             'notebook)
   ('block                <block1>)
   ('function             'grant-in)
   ('direction            'input))
)

(p design-2to3-grant-notebook-2
  :specificity 20
  (interface-block ^schema-name <block1> ^cluster daisy-chain-2to3
         ^spec <ba-spec> ^priority {<num> > 1})
 -(notebook ^block <block1> ^function grant-in)
```

```
  (interface-block ^schema-name <block3> ^cluster daisy-chain-2to3
        ^spec <ba-spec> ^priority (lower-priority <> <num>))
  (notebook ^schema-name <nb3> ^block <block3> ^function grant-in)
  (step ^phase bus-arbitration-interface)
-->
 (let ((signal (make-name <block1> 'grant-out)))
  (cschema signal
   ('instance              'signal)
   ('name                  'grant-out)
   ('has-signal+inv        <block1>))
  (cschema <nb3>
   ('active                (list '! 'asserted signal))
   ('inactive              (list '! 'negated  signal)))
  (cschema (make-name <block1> 'grant-in-notebook)
   ('instance              'notebook)
   ('block                 <block1>)
   ('function              'grant-in)
   ('direction             'input))
  (cschema (make-name <block1> 'grant-out-notebook)
   ('instance              'notebook)
   ('block                 <block1>)
   ('function              'grant-out)
   ('direction             'output)
   ('active                (list '! 'asserted signal))
   ('inactive              (list '! 'negated  signal))
   ('internal-active       (list '! 'asserted signal))
   ('internal-inactive     (list '! 'negated  signal)))
))

(p design-2to3-grant-notebook-3
  :specificity 20
  (interface-block ^schema-name <block1> ^cluster daisy-chain-2to3
        ^spec <ba-spec> ^priority 1
        ^arbiter <name1> ^arbiter-protocol <prot1>)
 -(notebook ^block <block1> ^function grant-in)
  (interface-block ^schema-name <block3> ^cluster daisy-chain-2to3
        ^spec <ba-spec> ^priority 2)
  (notebook ^schema-name <nb3> ^block <block3> ^function grant-in)
  (step ^phase bus-arbitration-interface)
-->
 (let ((signal (make-name <block1> 'grant-out)))
  (cschema signal
   ('instance              'signal)
   ('name                  'grant-out)
   ('has-signal+inv        <block1>))
  (cschema <nb3>
   ('active                (list '! 'asserted signal))
   ('inactive              (list '! 'negated  signal)))
  (cschema (make-name <block1> 'grant-in-notebook)
   ('instance              'notebook)
   ('block                 <block1>)
```

```
  ('function              'grant-in)
  ('direction             'input)
  ('active                (get-event <name1> <prot1> 'grant))
  ('inactive              (get-event <name1> <prot1> 'not-grant)))
 (cschema (make-name <block1> 'grant-out-notebook)
  ('instance              'notebook)
  ('block                 <block1>)
  ('function              'grant-out)
  ('direction             'output)
  ('active                (list '! 'asserted signal))
  ('inactive              (list '! 'negated  signal))
  ('internal-active       (list '! 'asserted signal))
  ('internal-inactive     (list '! 'negated  signal)))
))

(p design-2to3-grant-notebook-4a
 :specificity 21
 (interface-block ^schema-name <block1> ^cluster daisy-chain-2to3
        ^spec <ba-spec> ^priority 1
        ^arbiter <name1> ^arbiter-protocol <prot1> ^grant-out <> nil)
 -(notebook ^block <block1> ^function grant-in)
 -(interface-block ^cluster daisy-chain-2to3 ^spec <ba-spec>
        ^priority 2)
 (step ^phase bus-arbitration-interface)
-->
 (cschema (make-name <block1> 'grant-in-notebook)
  ('instance              'notebook)
  ('block                 <block1>)
  ('function              'grant-in)
  ('direction             'input)
  ('active                (get-event <name1> <prot1> 'grant))
  ('inactive              (get-event <name1> <prot1> 'not-grant)))
 (cschema (make-name <block1> 'grant-out-notebook)
  ('instance              'notebook)
  ('block                 <block1>)
  ('function              'grant-out)
  ('direction             'output)
  ('active                (get-event <name1> <prot1> 'grant-out))
  ('inactive              (get-event <name1> <prot1> 'not-grant-out)))
)

(p design-2to3-grant-notebook-4b
 :specificity 20
 (interface-block ^schema-name <block1> ^cluster daisy-chain-2to3
        ^spec <ba-spec> ^priority 1
        ^arbiter <name1> ^arbiter-protocol <prot1>)
 -(notebook ^block <block1> ^function grant-in)
 -(interface-block ^cluster daisy-chain-2to3 ^spec <ba-spec>
        ^priority 2)
 (step ^phase bus-arbitration-interface)
-->
```

```
 (cschema (make-name <block1> 'grant-in-notebook)
  ('instance            'notebook)
  ('block               <block1>)
  ('function            'grant-in)
  ('direction           'input)
  ('active              (get-event <name1> <prot1> 'grant))
  ('inactive            (get-event <name1> <prot1> 'not-grant)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; GRANT Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-grant-notebook
  :specificity 18
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3)
 -(notebook ^block <block> ^function grant)
  (step ^phase bus-arbitration-interface)
-->
 (cschema (make-name <block> 'grant-notebook)
  ('instance            'notebook)
  ('block               <block>)
  ('function            'grant)
  ('direction           'internal))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; FAIRNESS Notebook
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-fairness-notebook
  :specificity 20
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^fairness <> nil)
 -(notebook ^block <block> ^function fairness)
  (step ^phase bus-arbitration-interface)
-->
 (cschema (make-name <block> 'fairness-notebook)
  ('instance            'notebook)
  ('block               <block>)
  ('function            'fairness)
  ('direction           'internal))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; BUS ARBITRATION DESIGN INTERFACE
;;; DAISY CHAIN CLUSTER
;;; 2-to-2 and 2-to-3-wire BA protocol case
;;; SIGNAL DESIGN
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Create interface signal
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p create-interface-signal
  :specificity 20
  (interface-block ^schema-name <block> ^has-signal <sigs>)
  (notebook ^schema-name <nb> ^block <block> ^function <fn>)
 -(signal ^schema-name (member <> <sigs>) ^name <fn>)
  (step ^schema-name <step> ^phase bus-arbitration-interface)
-->
 (let
  ((signal (make-name <block> <fn>)))
  (cschema signal
   ('instance                    'signal)
   ('name                        <fn>)
   ('has-signal+inv              <block>))
  (cschema <nb>
   ('internal-active    (list '! 'asserted signal))
   ('internal-inactive  (list '! 'negated signal))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; BUS ARBITRATION DESIGN INTERFACE
;;; DAISY CHAIN CLUSTER
;;; 2-to-2 and 2-to-3-wire BA protocol case
;;; T-BLOCK DEFINITION
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; T-block definition for output signal
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p define-output-signal-demon
  :specificity 30
  (interface-block ^schema-name <block>)
  (notebook ^schema-name <nb> ^type << asm customized >> ^block <block>
        ^function <fn> ^direction output)
  (step ^schema-name <step> ^phase bus-arbitration-interface)
-->
  (print <fn> 'could 'not 'be 'complex)
  (new-value <step> 'phase 'error)
)
```

```
(p define-single-output-signal
  :specificity 10
  (interface-block ^schema-name <block>)
  (notebook ^schema-name <nb> ^type nil ^block <block> ^direction output)
  (step ^phase bus-arbitration-interface)
-->
  (let
   ((active (get-value <nb> 'active))
         (inactive (get-value <nb> 'inactive)))
    (if (single-signal active inactive)
         (cschema <nb>
          ('type           'single))
         (if (asm-signal active inactive)
          (cschema <nb>
           ('type           'asm))
          (cschema <nb>
           ('type           'customized)))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; T-block definition for input signal
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p define-input-signal
  :specificity 10
  (interface-block ^schema-name <block>)
  (notebook ^schema-name <nb> ^type nil ^block <block> ^direction input)
  (step ^phase bus-arbitration-interface)
-->
  (let
   ((active (get-value <nb> 'active))
         (inactive (get-value <nb> 'inactive)))
    (if (single-signal active inactive)
         (cschema <nb>
          ('type           'single))
         (if (asm-signal active inactive)
          (cschema <nb>
           ('type           'asm))
          (cschema <nb>
           ('type           'customized)))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; T-block definition for internal signal
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p define-internal-signal
  :specificity 10
  (interface-block ^schema-name <block>)
  (notebook ^schema-name <nb> ^type nil ^block <block> ^direction internal)
```

```
 (step ^phase bus-arbitration-interface)
-->
 (cschema <nb>
  ('type                   'asm))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; BUS ARBITRATION DESIGN INTERFACE
;;; DAISY CHAIN CLUSTER
;;; 2-to-2 and 2-to-3-wire BA protocol case
;;; CONNECT BLOCK DESIGN (T-BLOCK = C-BLOCK)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Output C-block
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;  ^cluster daisy-chain-2to3
(p create-output-connect-block
  :specificity 9
 (interface-block ^schema-name <block> ^has-block <blks>)
 (notebook ^schema-name <nb> ^type single ^block <block>
        ^function <fn> ^direction output)
 -(block ^schema-name (member <> <blks>) ^action <fn>)
 (step ^phase bus-arbitration-interface)
-->
 (cschema (create-name 'connect-block)
  ('instance             'connect-block)
  ('has-block+inv        <block>)
  ('action               <fn>)
  ('output0              (get-value <nb> 'inactive))
  ('output1              (get-value <nb> 'active))
  ('input0               (get-value <nb> 'internal-inactive))
  ('input1               (get-value <nb> 'internal-active)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Input C-block
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p create-input-connect-block
  :specificity 9
 (interface-block ^schema-name <block> ^has-block <blks>)
 (notebook ^schema-name <nb> ^type single ^block <block>
        ^function <fn> ^direction input)
 -(block ^schema-name (member <> <blks>) ^action <fn>)
 (step ^phase bus-arbitration-interface)
-->
 (cschema (create-name 'connect-block)
  ('instance             'connect-block)
  ('has-block+inv        <block>)
```

```
  ('action            <fn>)
  ('input0            (get-value <nb> 'inactive))
  ('input1            (get-value <nb> 'active))
  ('output0           (get-value <nb> 'internal-inactive))
  ('output1           (get-value <nb> 'internal-active)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; BUS ARBITRATION DESIGN INTERFACE
;;; DAISY CHAIN CLUSTER
;;; 2-to-3-wire BA protocol case
;;; CUSTOMIZED DESIGN
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; INTERFACE INPUT-SIGNAL CUSTOMIZED ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-interface-input-signal-customized-asm
  :specificity 9
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^has-block <blks>)
  (notebook ^schema-name <nb> ^type customized ^block <block>
        ^function <fn> ^direction input)
 -(block ^schema-name (member <> <blks>) ^function <fn>)
  (step ^phase bus-arbitration-interface)
-->
 (let
  ((asm-template (get-value <nb> 'active))
   (asm-instance (create-name 'asm)))
  (cschema asm-instance
   ('instance          asm-template)
   ('function          <fn>)
   ('has-block+inv     <block>))
  (instantiate-asm     asm-instance asm-template 'u1))
)

(defun instantiate-asm (instance template comp)
  (dolist (slot (append (get-values template 'inputs)
                        (get-values template 'outputs)))
   (cschema instance
        (slot    (transform (get-value template slot) comp)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; BUS ARBITRATION DESIGN INTERFACE
;;; DAISY CHAIN CLUSTER
;;; 2-to-3-wire BA protocol case
;;; ASM DESIGN
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; INTERFACE INPUT-SIGNAL ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-interface-input-signal-asm
  :specificity 9
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^has-block <blks>)
  (notebook ^schema-name <nb> ^type asm ^block <block>
        ^function <fn> ^direction input)
 -(block ^schema-name (member <> <blks>) ^function <fn>)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (create-name 'two-state-asm)
   ('instance          'two-state-asm)
   ('function          <fn>)
   ('has-block+inv     <block>)
   ('input0            (get-value <nb>        'inactive))
   ('input1            (get-value <nb>        'active))
   ('output0           (get-value <nb>        'internal-inactive))
   ('output1           (get-value <nb>        'internal-active)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; INTERFACE INTERNAL-SIGNAL ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; FAIRNESS ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-fairness-asm
  :specificity 8
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^has-block <blks>)
 -(block ^schema-name (member <> <blks>) ^function fairness)
  (notebook ^schema-name <fair>   ^block <block> ^function fairness)
  (notebook ^schema-name <c-req>  ^block <block> ^function c-req)
  (notebook ^schema-name <req-in> ^block <block> ^function req-in)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (create-name 'two-state-asm)
   ('instance          'two-state-asm)
   ('function          'fairness)
   ('has-block+inv     <block>)
   ('input0            (get-value <c-req>     'internal-inactive))
   ('input1            (get-value <req-in>    'internal-active))
   ('output0           (get-value <fair>      'internal-inactive))
   ('output1           (get-value <fair>      'internal-active)))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; GRANT & GRANT-OUT ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-grant-asm
  :specificity 7
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
         ^has-block <blks>)
 -(block ^schema-name (member <> <blks>) ^function grant)
  (notebook ^schema-name <grant> ^block <block> ^function grant)
  (notebook ^schema-name <grant-in> ^block <block> ^function grant-in)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (create-name 'connect-block)
   ('instance          'connect-block)
   ('has-block+inv     <block>)
   ('function          'grant)
   ('input0            (get-value <grant-in> 'internal-inactive))
   ('input1            (get-value <grant-in> 'internal-active))
   ('output0           (get-value <grant>    'internal-inactive))
   ('output1           (get-value <grant>    'internal-active)))
)

(p design-2to3-grant-and-grant-out-asm
  :specificity 8
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
         ^has-block <blks>)
 -(block ^schema-name (member <> <blks>) ^function grant)
  (notebook ^schema-name <grant-out> ^block <block> ^function grant-out)
  (notebook ^schema-name <grant>     ^block <block> ^function grant)
  (notebook ^schema-name <req-in>    ^block <block> ^function req-in)
  (notebook ^schema-name <grant-in>  ^block <block> ^function grant-in)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (create-name 'mutual-exclusion-asm)
   ('instance          'mutual-exclusion-asm)
   ('function          'grant 'grant-out)
   ('has-block+inv     <block>)
   ('condition-A       (get-state (get-value <req-in> 'internal-active)))
   ('condition-B(get-state (get-value <req-in> 'internal-inactive)))
   ('input0            (get-value <grant-in> 'internal-inactive))
   ('input1            (get-value <grant-in> 'internal-active))
   ('output0-A         (get-value <grant>              'internal-inactive))
   ('output1-A         (get-value <grant>              'internal-active))
   ('output0-B         (get-value <grant-out>          'internal-inactive))
   ('output1-B         (get-value <grant-out>          'internal-active)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; INTERFACE OUTPUT-SIGNAL ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; REQUEST-OUT ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-request-asm-1a
  :specificity 5
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^has-block <blks> ^fairness <> nil)
  (block ^schema-name {<asm> (member <> <blks>)} ^function request)
  (notebook ^schema-name <fair> ^block <block> ^function fairness)
  (step ^phase bus-arbitration-interface)
-->
  (new-value <asm>
   'input1
   (add-signal '^ (get-state (get-value <fair> 'internal-inactive))
        (get-value <asm> 'input1)))
)

(p design-2to3-request-asm-1b
  :specificity 5
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^has-block <blks>)
 -(block ^schema-name (member <> <blks>) ^function request)
  (notebook ^schema-name <req-out>   ^block <block> ^function req-out)
  (notebook ^schema-name <req-in>    ^block <block> ^function req-in)
  (notebook ^schema-name <grant-ack> ^block <block> ^function grant-ack)
  (step ^phase bus-arbitration-interface)
-->
  (cschema (create-name 'two-state-asm)
   ('instance          'two-state-asm)
   ('function          'request)
   ('has-block+inv     <block>)
   ('input0            (get-value <grant-ack>     'internal-active))
   ('input1            (get-value <req-in>        'internal-active))
   ('output0           (get-value <req-out>  'internal-inactive))
   ('output1           (get-value <req-out>  'internal-active)))
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; GRANT-ACK ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-grant-ack-asm-1
  :specificity 5
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^has-block <blks> ^merged-grant-ack <> nil)
 -(block ^schema-name (member <> <blks>) ^function grant-ack)
  (notebook ^schema-name <grant-ack> ^block <block> ^function grant-ack)
  (notebook ^schema-name <req-in>    ^block <block> ^function req-in)
```

```
 (notebook ^schema-name <grant>    ^block <block> ^function grant)
 (notebook ^schema-name <bus-free>  ^block <block> ^function bus-free)
 (step ^phase bus-arbitration-interface)
-->
 (cschema (create-name 'two-state-asm)
  ('instance          'two-state-asm)
  ('function          'grant-ack)
  ('has-block+inv     <block>)
  ('input0            (list '^
                           (get-value <req-in>    'internal-inactive)
                           (get-value <grant>     'internal-inactive)))
  ('input1            (list '^
                           (get-value <grant>     'internal-active)
                           (get-value <bus-free>  'internal-active)))
  ('output0           (get-value <grant-ack>      'internal-inactive))
  ('output1           (get-value <grant-ack>      'internal-active)))
)

(p design-2to3-grant-ack-asm-2
 :specificity 5
 (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
       ^has-block <blks> ^merged-grant-ack nil)
 -(block ^schema-name (member <> <blks>) ^function grant-ack)
 (notebook ^schema-name <grant-ack> ^block <block> ^function grant-ack)
 (notebook ^schema-name <req-in>    ^block <block> ^function req-in)
 (notebook ^schema-name <grant>     ^block <block> ^function grant)
 (step ^phase bus-arbitration-interface)
-->
 (cschema (create-name 'two-state-asm)
  ('instance          'two-state-asm)
  ('function          'grant-ack)
  ('has-block+inv     <block>)
  ('input0            (list '^
                           (get-value <req-in>    'internal-inactive)
                           (get-value <grant>      'internal-inactive)))
  ('input1                 (get-value <grant>     'internal-active))
  ('output0           (get-value <grant-ack>      'internal-inactive))
  ('output1           (get-value <grant-ack>      'internal-active)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; ACK ASM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p design-2to3-ack-asm-1
 :specificity 5
 (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
       ^has-block <blks> ^merged-grant-ack nil)
 -(block ^schema-name (member <> <blks>) ^function ack)
 (notebook ^schema-name <ack>       ^block <block> ^function ack)
 (notebook ^schema-name <grant-ack> ^block <block> ^function grant-ack)
```

```
 (notebook ^schema-name <bus-free>  ^block <block> ^function bus-free)
 (step ^phase bus-arbitration-interface)
-->
 (cschema (create-name 'two-state-asm)
  ('instance           'two-state-asm)
  ('function           'ack)
  ('has-block+inv      <block>)
  ('input0                      (get-value <grant-ack>         'internal-inactive))
  ('input1             (list '^
                                (get-value <grant-ack>'internal-active)
                                (get-value <bus-free> 'internal-active)))
  ('output0            (get-value <ack>              'internal-inactive))
  ('output1            (get-value <ack>              'internal-active)))
)

(p design-2to3-ack-asm-2
  :specificity 5
  (interface-block ^schema-name <block> ^cluster daisy-chain-2to3
        ^has-block <blks> ^merged-grant-ack <> nil)
 -(block ^schema-name (member <> <blks>) ^function ack)
 (notebook ^schema-name <ack>       ^block <block> ^function ack)
 (notebook ^schema-name <grant-ack> ^block <block> ^function grant-ack)
 (step ^phase bus-arbitration-interface)
-->
 (cschema (create-name 'two-state-asm)
  ('instance           'two-state-asm)
  ('function           'ack)
  ('has-block+inv      <block>)
  ('input0             (get-value <grant-ack>        'internal-inactive))
  ('input1             (get-value <grant-ack>        'internal-active))
  ('output0            (get-value <ack>              'internal-inactive))
  ('output1            (get-value <ack>              'internal-active)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Lisp Pseudo-predicates and Functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun create-name (class)
 (make-name class (new-value class 'subclass#
        (1+ (get-value class 'subclass#)))))

(defun lower-priority (hi pri)
 (if (and (numberp hi) (numberp pri))
          (= hi (+ pri 1))
          nil))

(defun get-signal-name (component function)
 (get-value-if component 'has-signal
  #'(lambda (x) (equal (get-value x 'name) function))
  :start 0 :end nil))
```

```
(defun get-signal (event)
  (if (atom event)
   nil
   (third event)))

(defun single-signal (event1 event2)
  (when
   (and
         (not (null (get-signal event1)))
         (not (null (get-signal event2)))
         (equal (get-signal event1) (get-signal event2)))
   t))

(defun asm-signal (event1 event2)
  (and
   (not (null (get-signal event1)))
   (not (null (get-signal event2)))))

(defun get-state (event)
  (rest event))

(defun get-event (component protocol function)
  (let ((expr (get-action protocol function)))
   (transform (get-value expr 'has-description) component)))

(defun get-action (protocol function)
  (get-value-if protocol 'has-action
   #'(lambda (x) (equal (get-value x 'function) function))
   :start 0 :end nil))

(defun transform (expr component)
  (cond
   ((null expr) nil)
   ((atom expr) expr)
   ((equal (first expr) '!)
         (list (first expr) (second expr)
                  (make-name component (get-value (third expr) 'name))))
   (t (cons (transform (first expr) component)
                        (transform (rest expr) component)))))

(defun action-equivalence (action1 action2)
  (and
   (relatedp action1 'precedes action2)
   (relatedp action2 'precedes action1)))

(defun add-signal (oper sign expr)
  (list oper expr sign))

(defun simplify (expr)
  expr)
```

Appendix C. Output Description.

The designer produces a technology-independent description of the blocks that comprise the interface. A sample output for the example discussed in chapter 6 is listed in this appendix.

Schemata from context $ROOT-CONTEXT:

```
{{ CONNECT-BLOCK-1
   INSTANCE: CONNECT-BLOCK
   HAS-BLOCK+INV: INTERFACE-BLOCK-1
   ACTION: REQ-OUT
   INPUT0: (! DISABLED U1-BR3)
   INPUT1: (! ASSERTED U1-BR3)
   OUTPUT0: (! NEGATED INTERFACE-BLOCK-1-REQ-OUT)
   OUTPUT1: (! ASSERTED INTERFACE-BLOCK-1-REQ-OUT)
   TO: U1-BR3 }}

{{ CONNECT-BLOCK-2
   INSTANCE: CONNECT-BLOCK
   HAS-BLOCK+INV: INTERFACE-BLOCK-1
   ACTION: GRANT-ACK
   OUTPUT0: (! DISABLED U1-BBSY)
   OUTPUT1: (! ASSERTED U1-BBSY)
   INPUT0: (! NEGATED INTERFACE-BLOCK-1-GRANT-ACK)
   INPUT1: (! ASSERTED INTERFACE-BLOCK-1-GRANT-ACK)
   TO: U1-BBSY }}

{{ CONNECT-BLOCK-3
   INSTANCE: CONNECT-BLOCK
   HAS-BLOCK+INV: INTERFACE-BLOCK-1
   ACTION: BUS-FREE
   INPUT0: (! ASSERTED U1-AS)
   INPUT1: (! NEGATED U1-AS)
   OUTPUT0: (! NEGATED INTERFACE-BLOCK-1-BUS-FREE)
   OUTPUT1: (! ASSERTED INTERFACE-BLOCK-1-BUS-FREE)
   FROM: U1-AS }}
```

{{ CONNECT-BLOCK-4
INSTANCE: CONNECT-BLOCK
HAS-BLOCK+INV: INTERFACE-BLOCK-1
ACTION: REQ-IN
INPUT0: (! NEGATED U5-HREQ)
INPUT1: (! ASSERTED U5-HREQ)
OUTPUT0: (! NEGATED INTERFACE-BLOCK-1-REQ-IN)
OUTPUT1: (! ASSERTED INTERFACE-BLOCK-1-REQ-IN)
FROM: U5-HREQ }}

{{ CONNECT-BLOCK-5
INSTANCE: CONNECT-BLOCK
HAS-BLOCK+INV: INTERFACE-BLOCK-1
ACTION: ACK
OUTPUT0: (! NEGATED U5-HACK)
OUTPUT1: (! ASSERTED U5-HACK)
INPUT0: (! NEGATED INTERFACE-BLOCK-1-ACK)
INPUT1: (! ASSERTED INTERFACE-BLOCK-1-ACK)
TO: U5-HACK }}

{{ CONNECT-BLOCK-6
INSTANCE: CONNECT-BLOCK
HAS-BLOCK+INV: INTERFACE-BLOCK-1
ACTION: GRANT-IN
INPUT0: (! NEGATED U1-BG3IN)
INPUT1: (! ASSERTED U1-BG3IN)
OUTPUT0: (! NEGATED INTERFACE-BLOCK-1-GRANT-IN)
OUTPUT1: (! ASSERTED INTERFACE-BLOCK-1-GRANT-IN)
FROM: U1-BG3IN }}

{{ CONNECT-BLOCK-7
INSTANCE: CONNECT-BLOCK
HAS-BLOCK+INV: INTERFACE-BLOCK-1
ACTION: GRANT-OUT
OUTPUT1: (! ASSERTED U1-BG3OUT)
OUTPUT0: (! NEGATED U1-BG3OUT)
INPUT1: (! ASSERTED INTERFACE-BLOCK-1-GRANT-OUT)
INPUT0: (! NEGATED INTERFACE-BLOCK-1-GRANT-OUT)
TO: U1-BG3OUT }}

```
{{ MUTUAL-EXCLUSIVE-ASM-1
   INSTANCE: MUTUAL-EXCLUSIVE-ASM
   HAS-BLOCK+INV: INTERFACE-BLOCK-1
   ACTION: GRANT GRANT-OUT
   CONDITION-A: (ASSERTED INTERFACE-BLOCK-1-REQ-IN)
   CONDITION-B: (NEGATED INTERFACE-BLOCK-1-REQ-IN)
   INPUT0: (! NEGATED INTERFACE-BLOCK-1-GRANT-IN)
   INPUT1: (! ASSERTED INTERFACE-BLOCK-1-GRANT-IN)
   OUTPUT0-A: (! NEGATED INTERFACE-BLOCK-1-GRANT)
   OUTPUT1-A: (! ASSERTED INTERFACE-BLOCK-1-GRANT)
   OUTPUT0-B: (! NEGATED INTERFACE-BLOCK-1-GRANT-OUT)
   OUTPUT1-B: (! ASSERTED INTERFACE-BLOCK-1-GRANT-OUT)}}

{{ TWO-STATE-ASM-1
   INSTANCE: TWO-STATE-ASM
   HAS-BLOCK+INV: INTERFACE-BLOCK-1
   ACTION: ACK
   INPUT0: (! NEGATED INTERFACE-BLOCK-1-GRANT-ACK)
   INPUT1: (^ (! ASSERTED INTERFACE-BLOCK-1-GRANT-ACK)
          (ASSERTED INTERFACE-BLOCK-1-BUS-FREE))
   OUTPUT0: (! NEGATED INTERFACE-BLOCK-1-ACK)
   OUTPUT1: (! ASSERTED INTERFACE-BLOCK-1-ACK)}}

{{ TWO-STATE-ASM-2
   INSTANCE: TWO-STATE-ASM
   HAS-BLOCK+INV: INTERFACE-BLOCK-1
   ACTION: GRANT-ACK
   INPUT0: (^ (! NEGATED INTERFACE-BLOCK-1-REQ-IN)
          (! NEGATED INTERFACE-BLOCK-1-GRANT))
   INPUT1: (! ASSERTED INTERFACE-BLOCK-1-GRANT)
   OUTPUT0: (! NEGATED INTERFACE-BLOCK-1-GRANT-ACK)
   OUTPUT1: (! ASSERTED INTERFACE-BLOCK-1-GRANT-ACK)}}

{{ TWO-STATE-ASM-3
   INSTANCE: TWO-STATE-ASM
   HAS-BLOCK+INV: INTERFACE-BLOCK-1
   ACTION: REQUEST
   INPUT0: (! ASSERTED INTERFACE-BLOCK-1-GRANT-ACK)
   INPUT1: (! ASSERTED INTERFACE-BLOCK-1-REQ-IN)
   OUTPUT0: (! NEGATED INTERFACE-BLOCK-1-REQ-OUT)
   OUTPUT1: (! ASSERTED INTERFACE-BLOCK-1-REQ-OUT)}}
```

Appendix D. Sequential Design.

D.1 Introduction.

In this appendix, a standard procedure of designing the various blocks required for the interface design presented in chapter 5, is outlined. The motivation is twofold. Firstly, it is important to prove that the design produced by the DAME's designer is complete and correct. The implementation of the bus arbitration interface example analyzed in chapter 6 served this purpose. Another important issue is linked to the automation of the design. The formal procedure of designing asynchronous circuits using standard logic gates, is well known in the literature [32, 57]. As a matter of fact, there exist some programs that can produce a hardware design from input specifications at the register-transfer level [3, 42, 51, 70, 71, 72, 80, 82, 93].

D.2 Basic concepts for Asynchronous Sequential Circuits.

An overview of the basic ideas in asynchronous sequential circuits (ASC) design is presented in this section, giving especial attention to the SR latch, an important constituent of asynchronous circuits. A procedure for designing ASC from behavioral specifications is described. In the next section, this procedure is exploited to create the hardware for the general two-state ASM necessitated as a sub-block in the interface block for the bus arbitration design.

D.2.1 Asynchronous Sequential Circuit Model.

A sequential circuit is specified by a time sequence of inputs, outputs, and internal states. In synchronous sequential circuits the change of internal state is regulated by the synchronized clock pulses. In an asynchronous sequential circuit, a change of

internal state occurs when there is a change in the input variables.

Asynchronous sequential circuits are useful when:

• the speed of operation is important,

• the input signals to the system may change at any time, independently of any clock signal, and/or

• there is no common clock between two units that need to communicate to one another.

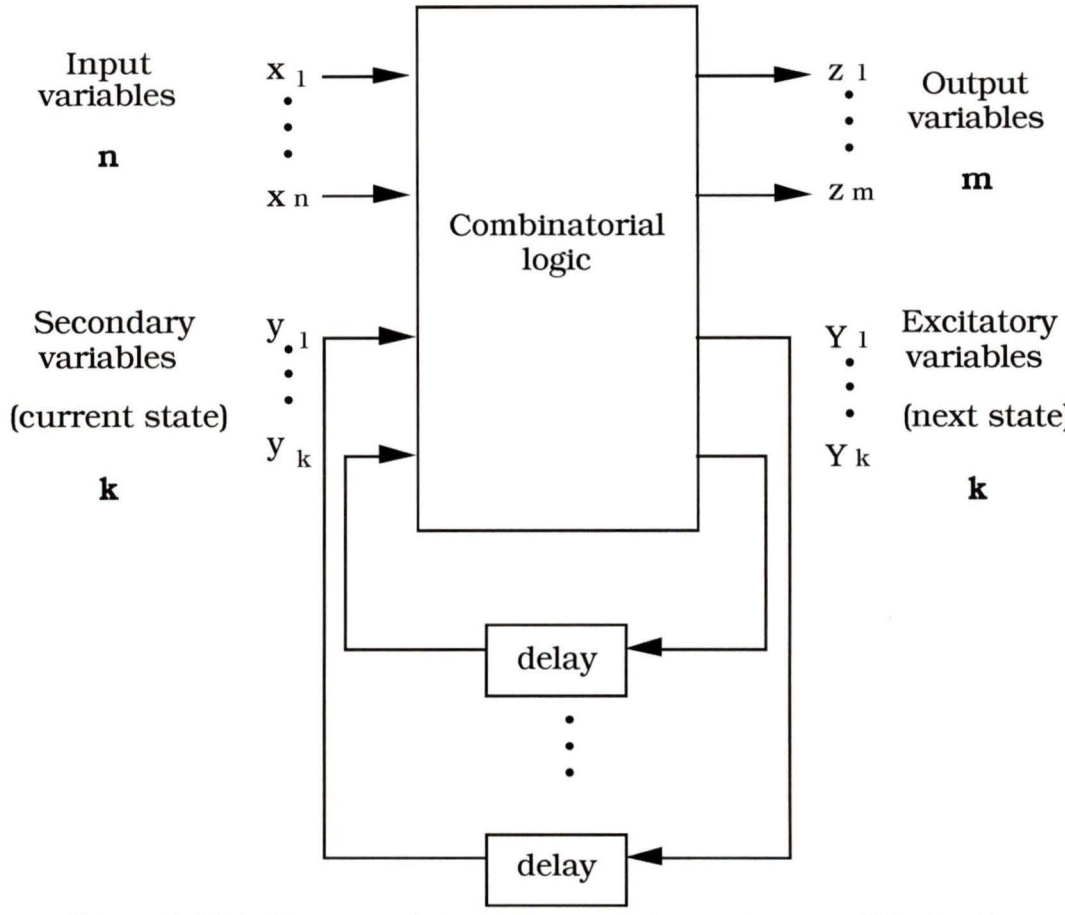


Figure D.1 Block representation of an asynchronous sequential circuit.

Figure D.1 shows the block diagram of an asynchronous sequential circuit. It consists of a combinational circuit and delay time elements connected to form a feedback path. There are n input variables, m output variables, and k internal states. The delay elements play the role of short-term memories for the sequential circuit.

To ensure proper operation, asynchronous sequential circuits must be allowed to attain a stable state before one of its inputs changes to a new value. Usually only one signal is allowed to change at a time, although this condition may be relaxed in certain cases. This type of operation is called the *fundamental mode.*

We will introduce the techniques and descriptions used for the analysis of asynchronous sequential circuits using the Set-Reset (SR) latch.

D.2.2 SR Latch.

The memory elements in synchronous sequential circuits are clocked flip-flops, while in asynchronous sequential circuits, they are either combinatorial logic with feedback or unclocked flip-flops together with time delay elements. Sometimes the delay is obtained by using the gates' delays from the combinatorial block. The Set-Reset, or SR, latch can be used as the building block of asynchronous sequential circuits. The use of SR latches in asynchronous circuits produces a more regular design[1].

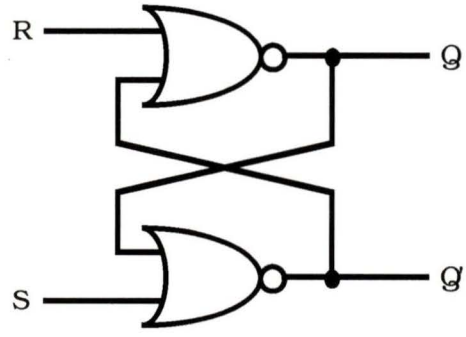


Figure D.2 SR latch using NOR gates.

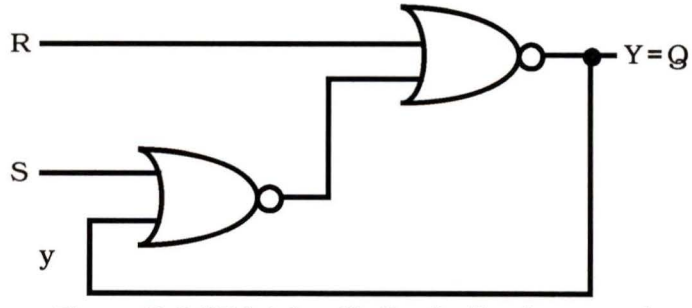


Figure D.3 SR latch with the feedback exposed.

The SR latch is a digital circuit that consists of two cross-coupled NOR gates (or

---

[1] Not necessarily an optimal design, (i.e., one with the least number of components).

NAND gates for negative logic) as shown in figure D.2. The same circuit is shown in figure D.3 to reveal the feedback. The SR latch has two inputs S and R, one output Q that corresponds to the excitation state variable Y, and one secondary variable y. There is a direct interconnection between Y and y. The corresponding delay shown in figure D.1 is considered to be the internal delays from the NOR gates.

The Boolean function for Y is

$$Y = [(S + y)' + R]' = SR' + R'y \qquad \text{Eq. (D.1)}$$

The corresponding transition table for equation D.1 is shown in figure D.4. The entries in the table in which Y=y are encircled. Those entries correspond to stable states of the circuit. We can interpret the transition table as follows (figure D.5): Suppose the initial state is y = 0, and the input is SR = 00. While the input remains at 00, y will stay at 0. If the input changes to SR = 10 (it is not allowed to change to SR = 11 for the fundamental mode assumption), then Y = 1, and because of the feedback path y turns into 1 (stable state at the right-down corner). The input must not change until the stable condition Y = y is reached.

	SR			
y	00	01	11	10
0	(0)	(0)	(0)	1
1	(1)	0	0	(1)

Figure D.4 Transition table for Y in the SR latch.

In a synchronous system the present state is totally specified by the flip-flop values and does not change if the inputs change while the clock pulse is inactive. In an asynchronous circuit the internal state can change immediately after a change in the input. That is the reason to consider the internal state and the input value together as the *total state* of the circuit.

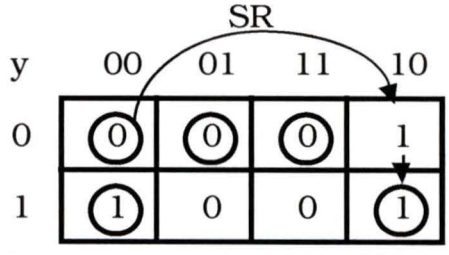


Figure D.5 State transition from SRy = 000 to SRy = 101.

The transition table of asynchronous systems is equivalent to the one used for synchronous circuits if we regard the secondary variables as the present state and the excitation variables as the next state. There is one restriction in asynchronous circuits for the system behavior to be stable: there must be at least one entry for the *next* state that is the same as the *present* state in each row of the table. Otherwise all the total states in that row would be unstable. The other condition for stability calls for the presence of a stable state in each column of the transition table. Otherwise if the input corresponding to that column were present, the circuit would not converge to any stable state.

D.2.3 State Assignment and the Flow Table.

A flow table is the abstraction of the transition table in which the internal states are represented as symbols. The step that transforms the flow table into the transition table is called state assignment.

A primitive flow table has only one stable state in each row. A design procedure suggests to start from the behavior specifications in order to create a primitive flow table.

A race condition occurs in an asynchronous system when two or more state variables change in response to an input change. Because of unequal delays in the gates, the state sequence is unpredictable. If the final stable state that the circuit reaches does not depend on the order of the state sequence, the race is called *non-critical*. Otherwise it is a *critical race*. For proper operation critical races must be avoided.

There exist several techniques to eliminate races. Two of them are:

- binary assignment of the state variables such that only one state variable changes when a state transition occurs; this is not always possible to achieve in the general case; and

- directing the state sequence through unstable states with unique state variable change; the main drawback is the delay to reach the stable state that affects the speed of the circuit.

Once the state transition table is known, standard techniques can be used to produce the corresponding combinatorial logic (i.e., Karnaugh maps, the Quine-

McCluskey algorithm).

D.2.4 Hazards in Combinatorial Circuits.

Hazards occur in combinatorial circuits and may cause a temporary false output value. Hazards are unwanted switching transients that may appear at the output of a circuit because different paths exhibit different propagation delays. The presence of hazards in ASC may cause a wrong state transition.

Combinatorial hazards are classified into static and dynamic hazards. A static hazard is present when the output is supposed to remain constant during a transition but it changes momentarily because of propagation delays in the gates. It is called a static 0-hazard, or static 1-hazard, if the output should stay at level 0, or 1, respectively. A dynamic hazard produces a sequence of several output changes when only one was expected.

There exist methods of eliminating hazards, that usually involve removing unnecessary connections or adding redundant gates to the circuit [57]. One method of avoiding static hazards in ASC is to implement the circuit with SR latches using a two-level[2] logic strategy [94]. SR latches are insensitive to 1-hazards because a momentary 0 signal applied to the S or R inputs of a NOR latch (figure D.2) has no effect on the state of the circuit. In a 2-level sum-of-products realization, 0-hazards or dynamic hazards cannot occur [32].

Another type of hazard is the essential hazard, that is caused by different delays along two or more paths that originate from the same signal. Essential hazards can be corrected by adjusting the amount of delay in the affected path. One technique is based on ensuring that each feedback loop does have approximately the same delay.

D.2.5 Design Procedure for Asynchronous Sequential Circuits.

The procedural steps for designing ASC can be summarized as follows [57]:

• Obtain a primitive flow table from the given design specifications. It is recognized that this is the most difficult part of the design.

---

[2] Two level refers to the maximum number of gates an input signal must pass through.

• Reduce the flow table by merging rows in the primitive flow table. There exists a formal procedure to accomplish this phase.

• Assign binary state variables to each row of the reduced flow table to obtain a transition table. The main concern is to eliminate any possible critical race.

• Simplify the Boolean functions of the excitation and output variables and draw the logic diagrams using either logic gates or SR latches as basic blocks. Remove hazards in the circuit design.

In the following section we will generate the design of the two-state ASM defined in the bus arbitration interface design phase of the DAME's designer.

D.3 Two-state ASM Implementation.

Firstly, the given specifications for the asynchronous state machines are presented. The primitive flow table is constructed from these specifications. The number of states is minimized by using standard merging rules. A state assignment is carried out to ensure a race-free operation of the circuit. Finally a hazard-free logic circuit that implements the asynchronous state machines is produced.

D.3.1 Specifications for the Asynchronous Sequential Circuit.

A frame description of the asynchronous state machine will be used to construct a primitive flow table. The Mealy machine corresponding to the two-state ASM is shown in figure D.6. Each of the inputs $!I_1$ and $!I_0$ conveys a state/event expression, using the notation developed in chapter 4. The output events $!O_1$ and $!O_0$ are restricted to be opposite transitions of an output signal O.

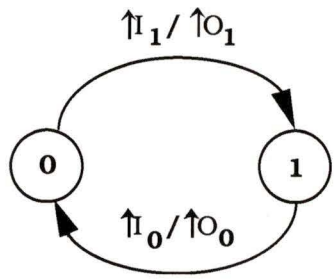


Figure D.6 Two-state ASM (Mealy machine).

The two-state ASM description in DAME is given in figure D.7. Transitions from inactive to active (! asserted) of the signals GRANT-ACK and REQ-IN correspond to $!I_1$ and $!I_0$ respectively. The expressions OUTPUT1 and OUTPUT0 represent complementary transitions of the output, in this case REQ-OUT.

```
{{ TWO-STATE-ASM-5
 INSTANCE: TWO-STATE-ASM
 HAS-SUB-BLOCK+INV: INTERFACE-BLOCK-1
 FUNCTION: REQUEST
 INPUT1: (! ASSERTED INTERFACE-BLOCK-1-GRANT-ACK)
 INPUT0: (! ASSERTED INTERFACE-BLOCK-1-REQ-IN)
 OUTPUT1: (!NEGATED INTERFACE-BLOCK-1-REQ-OUT)
 OUTPUT0: (!ASSERTED INTERFACE-BLOCK-1-REQ-OUT)}}
```

Figure D.7 Request ASM frame description.

One possible implementation involves two input signals, as shown in figure D.8, whose leading transitions register whenever the event expressions described by INPUT1 and INPUT0 are satisfied. This implementation requires the use of event detectors, as explained in chapter 5.

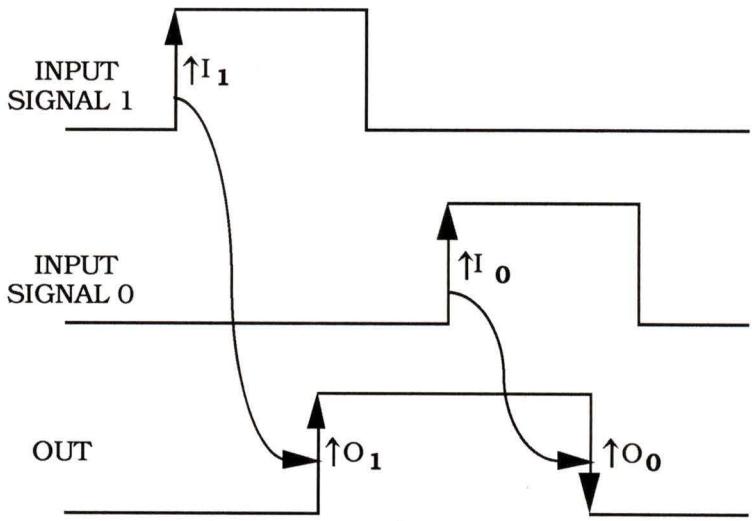


Figure D.8 Timing waveforms for the events and the inputs of the ASM.

The expected behavior of the circuit is described as follows: initially the machine is in state 0, yielding output O_0. When $!I_1$ occurs, the output changes to O_1. Similarly, if while the ASM's output is O_1, $!I_0$ occurs, the output changes back to O_0. Otherwise, no

change in the output is expected.

There are systems that can be completely defined by the values of their inputs and outputs at any given moment. It is possible to describe these systems using rules of the form:

if $[outputs]_0$ and $[inputs]_0$ and $[inputs]_1$ then $[outputs]_1$.

The meaning of this rule can be explained as follows: given a state denoted by certain values of the inputs and outputs of the system at time t_0, when $[inputs]_0$ changes to $[inputs]_1$, another set of outputs $[outputs]_1$ is reached. It is noted that the difference between $[inputs]_0$ and $[inputs]_1$ must be in only one entry because of the fundamental mode assumption.

The following rules describe the behavior of the two-state ASM in terms of its inputs and outputs:

1. If output=O_0 and !I_1 then output=O_1.

2. If output=O_1 and !I_0 then output=O_0.

3. Otherwise do not change the output.

The last rule groups several situations in which $[outputs]_0$ is the same as $[outputs]_1$. Also a short notation is used to represent changes in the input. For instance, rule 1 could be written as:

1. If $[output]_0$=O_0 and $[input0,input1]_0$=X0 and $[input0,input1]_1$=X1
then $[output]_1$=O_1.

Symbol X denotes a don't care value. However, input0 should remain unchanged. From a compact description like the one mentioned above, it is possible to derive all the possible states the system can hold, by constructing a *state tree*. The state tree can be used to build up the corresponding primitive flow table.

A state consists of a unique name tag, and its corresponding set of inputs, and outputs. A state is denoted as a triplet ([inputs], name-tag, [outputs]). For instance, in the two-state ASM, the initial state is called **a**, corresponding to input0=0, input1=0, and output=O_0. It is written (00, **a**, O_0), where 00 refers to [input0,input1]. The procedure to construct a state tree involves the application of the rules starting with the initial state

to expand all the possible states. There are two lists used by the procedure, a StateList that contains all the states that have been found, and the BranchList that holds the states being under consideration (i.e., states in the boundary of the generated tree that are not leaves).

1. Put the initial state in BranchList and StateList.

2. If BranchList is not empty, take one state S out of the list.

3. Generate the $[inputs]_1$ and $[outputs]_1$ of all possible successors of S, by changing one input at a time in $[inputs]_0$ of S, and by applying the rules of the system to obtain the new outputs.

4. Check if each successors exists in StateList. If it does not, give the successor a new name, and place it in StateList and BranchList. Otherwise, the successor is a leaf in the state tree.

5. Repeat steps 2 to 4 until BranchList is empty.

Figure D.9 shows the resulting state tree for the two-state ASM. Because there are two inputs, the state tree is a binary tree. The root of the tree is the initial state. In this state, two successors are obtained by changing either input1 or input2, yielding states **b** and **c** respectively. Rule **a** applies to the change from state **a** to state **b**, thus the corresponding output of state **b** is O_1. Of the two successors of **c**, one is exactly the same as state **a**; in this case a leaf in the tree has been reached. The leaves of the tree are shown encircled in figure D.9.

D.3.2 Construction of the Primitive Flow Table.

The state tree contains not only all the possible states exhibited by the system, but also the state sequences. A primitive flow table can be built from the information conveyed by the state tree by observing the following steps:

1. Form a primitive flow table by assigning one row to each of the states in the state tree. There are 2^N columns, where N is the number of inputs; each column corresponds to one input combination.

2. Fill in for each row, the information of the corresponding state S. Put the name and output of S in the column associated to its input.

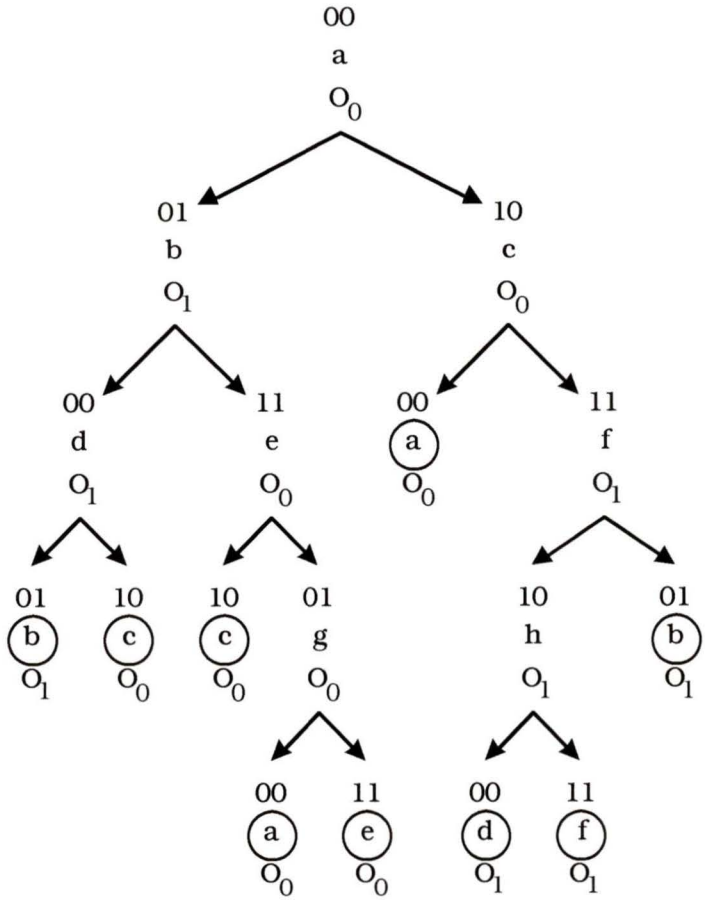


Figure D.9 Two-state ASM state tree.

3. Fill in the next state information. Put the name of the successors of S in the columns given by the successors' inputs, leaving the output empty.

4. Leave the remaining slots empty. Those slots correspond to forbidden changes (more than one input change at a given time).

The primitive flow table obtained from the state tree shown in figure D.9 is presented in figure D.10.

After state reduction and assignment, the transition table of figure D.11 is produced. A hazard-free circuit using logic gates was used in the implementation of the bus arbitration interface explained in chapter 6.

	$I_0 I_1$ = 00	01	11	10
a	(a), O_0	b , -	- , -	c , -
b	d , -	(b), O_1	e , -	- , -
c	a , -	- , -	f , -	(c), O_0
d	(d), O_1	b , -	- , -	c , -
e	- , -	g , -	(e), O_0	c , -
f	- , -	b , -	(f), O_1	h , -
g	a , -	(g), O_0	e , -	- , -
h	d , -	- , -	f , -	(h), O_1

Figure D.10 Primitive flow table.

	$I_0 I_1$ = 00	01	11	10
000	(000), 0	001 , -	010 , -	(000), 0
001	(001), 1	(001), 1	011 , -	000 , -
011	111 , 0	(011), 0	(011), 0	111 , 0
010	110 , 1	110 , 1	(010), 1	(010), 1
110	(110), 1	(110), 1	100 , -	111 , -
111	(111), 0	110 , -	101 , -	(111), 0
101	001 , 1	001 , 1	(101), 1	(101), 1
100	000 , 0	(100), 0	(100), 0	000 , 0

Figure D.11 State transition table for the ASM.

D.4 Mutual Exclusion Implementation.

The procedure outlined above was used to implement the Mutual Exclusion (ME) element. The timing behavior of the ME is shown in figure D.12. This element has two inputs, **grant-in** and **req-in**, and two outputs, **grant-out** and **grant**. Whenever **grant-in** is asserted, either **grant-out** or **grant** will be asserted, if **req-in** is inactive or active, respectively. When **grant-in** is negated, the activated output is pulled down.

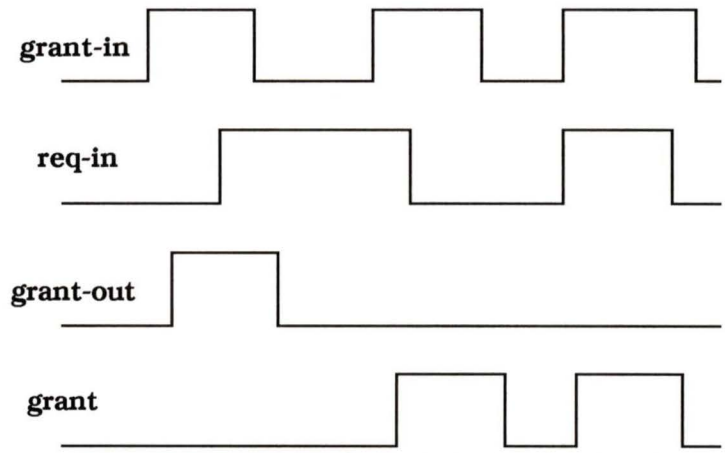


Figure D.12 Timing diagram of the mutual exclusion element.

In the initial state, all inputs and outputs are inactive. Four rules describe the behavior of the ME element:

1. If [**grant, grant-out**]=00 and **req-in**=0 and !**grant-in** then **grant-out**=1.

2. If [**grant, grant-out**]=00 and **req-in**=1 and !**grant-in** then **grant**=1.

3. If [**grant, grant-out**]=01 and !$\overline{\textbf{grant-in}}$ then **grant-out**=0.

3. If [**grant, grant-out**]=10 and !$\overline{\textbf{grant-in}}$ then **grant**=0.

The expanded state tree for the ME block is shown in figure D.13. The total number of possible states is six. From the state tree it is easy to construct the corresponding primitive flow table, depicted in figure D.14.

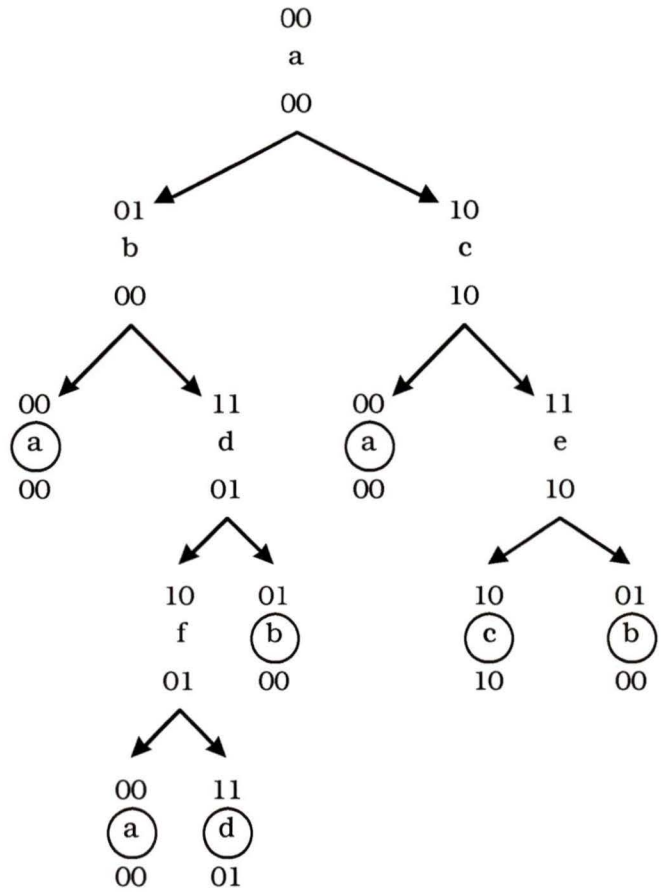


Figure D.13 State tree for the mutual exclusion element.

grant-in/req-in

	00	01	11	10
a	(a), 00	b , -	- , -	c , -
b	a , -	(b), 00	d , -	- , -
c	a , -	- , -	e , -	(c), 10
d	- , -	b , -	(d), 01	f , -
e	- , -	b , -	(e), 10	c , -
f	a , -	- , -	d , -	(f), 01

Figure D.14 Flow table for the mutual exclusion element.

Standards procedures of state minimization and state assignment can be carried

through. A simplified transition table for the ME element with only two states was accomplished, as presented in figure D.15. A two-level sum-of-products implementation is made hazard-free using well known procedures. Such a hardware implementation for the ME block using logic gates is shown in figure D.16.

	grant-in/req-in			
	00	01	11	10
0	(0), 00	1 , 00	(0), 10	(0), 10
1	0 , 00	(1), 00	(1), 01	(1), 01

Figure D.15 Transition table for the mutual exclusion element.

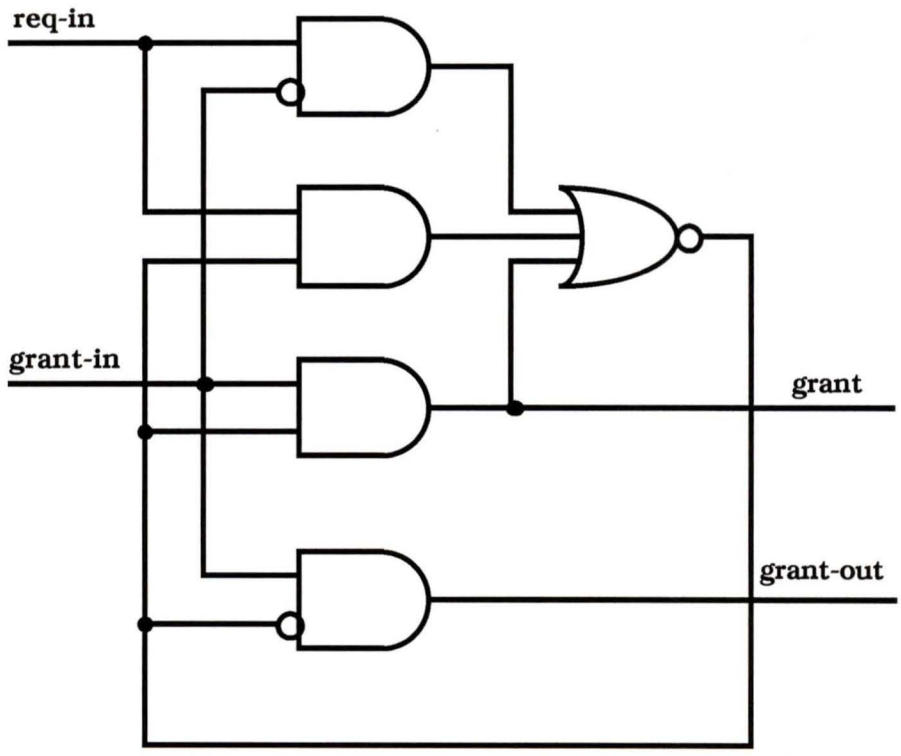


Figure D.16 ASC implementation for the mutual exclusion element.

D.5 Bus Status Observer Implementation.

Actions in the protocol are described by event or state expressions. There are cases in which the relationship between the action and the signals obeys a complicated rule. One example was shown in relation to the VMEbus bus arbitration interface. As described in chapter 5, in order to interconnect two devices with a two-wire and a three-

wire protocols, certain actions are required. In the VMEbus, all the actions are mapped into single signals, except bus-busy. A state machine, shown in figure 6.4, describes the status of the bus using two signals: BBSY and AS.

A corresponding primitive flow table can be obtained by assigning one row to each state of figure 6.4, as shown in figure D.17. Afterwards a formal procedure can be carried out to get a final implementation. Figure D.18 shows the transition table for the bus monitor. In the simulation, mentioned in chapter 6, a two-level sum-of-product realization of this transition table was used.

bbsy/as

	00	01	11	10
a	(a), 0	d, -	-, -	b, -
b	a, -	-, -	c, -	(b), 1
c	-, -	d, -	(c), 1	g, -
d	a, -	(d), 1	e, -	-, -
e	-, -	d, -	(e), 1	f, -
f	a, -	-, -	c, -	(f), 0
g	a, -	-, -	c, -	(g), 1

Figure D.17 Bus status flow table.

bbsy/as

	00	01	11	10
00	(00), 0	01, -	(00), 1	(00), 1
01	00, -	(01), 1	(01), 1	(11), -
11	10, 0	-,-	10, -	11, 0
10	00, 0	-,-	00, -	-,-

Figure D.18 Bus status transition table.

D.6 Summary.

This thesis introduces a methodology used in the bus arbitration interface design module. From an abstract design model, an instantiation for the particular components involved in the interface is created. The description is not the final hardware but a behavioral specification of the desired interface, in terms of functional blocks. Some examples of functional blocks are two-state ASM, and Mutual Exclusion blocks. This appendix presented one possible translation from the behavioral description to the actual hardware implementation. The procedure is prone to be automated. Actually systems that produce a hardware design from functional or behavioral specifications, such as the register-transfer description, have been described in the literature.

The hardware developed in this appendix was used in the implementation and verification of the bus arbitration interface example introduced in chapter 6. It was shown that the description of the blocks that are produced by the interface design stage contains enough information to complete the design. This stage of the design process can be automated.

Appendix E. Behavioral Description of the Functional Blocks.

After the abstract design for the bus arbitration example was carried out, a simulation to verify the correct behavior was performed. At the first level, the blocks that comprise the interface were modelled using the behavioral language (SBL) provided by SILOS II. In this appendix, the behavioral description of such blocks is presented.

E.1 Two-state ASM.

The two-state ASM description is described in figure E.1.

```
.module two_state_asm input_set input_reset / output ;
bit input_set, input_reset, output;
initial
{       output = #b0;}
forever
{       waitfor rise(input_set);
        if (output == #b0)
        {       output = #b1;}
}
forever
{       waitfor rise(input_reset);
        if (output == #b1)
        {       output = #b0;}
}
.eom ;
```

Figure E.1 Behavioral description of the two-state ASM using Silos II™ Behavioral Language.

The module has two binary inputs, **input-set** and **input-reset**, and one binary output, called **output**. The **initial** statement block describes the initial state of the circuit. There are two **forever** blocks that run concurrently and continuously. The first one waits for the input set transition, and the second one waits for the input reset transition. A binary number is preceded by **#b**.

E.2 Mutual Exclusion element.

The description of the Mutual Exclusion block using Silos II™ Behavioral Language is given in figure E.2.

```
.module mutual_exclusive GI RI / GO G;
bit GI, RI, GO, G;
initial
{       GO = #b0;
        G = #b0;}
forever
{       waitfor rise(GI);
        if ((G==#b0) and (GO==#b0) and (RI==#b0))
        {GO = #b1;}
        elseif ((G==#b0) and (GO==#b0) and (RI==#b1))
        {G = #b1;}
}
forever
{       waitfor fall(GI);
        if (G==#b1)
        {G = #b0;}
        elseif (GO==#b1)
        {GO = #b0;}
}
.eom;
```

Figure E.2 Behavioral description of the Mutual Exclusion block using Silos II™ Behavioral Language.

The module has two inputs, **GI** and **RI**, and two outputs, **GO** and **G**. Initially both outputs are negated. When a positive transition on **GI** is detected, if there is no recorded request from the requestor (**RI**=0), **GO** is asserted passing the grant through the daisy chain, otherwise (**RI**=1) the grant is given to the requestor. At the falling edge of **GI**, the previously asserted output is negated.

E.3 Event Detectors.

The behavioral description of two simple cases, the single event detector, and the two-event AND detector are shown in figures E.3 and E.4 respectively.

```
.module single_event_detector EVENT / OUT;
bit EVENT, OUT;
initial
{       OUT = #b0;}
forever
{       waitfor rise(EVENT);
        if (OUT == #b0)
        {OUT = #b1;}
        waitfor +1;
        OUT = #b0;}
.eom ;
```

Figure E.3 Behavioral description of a single event detector in SBL.

```
.module and_event_detector EVENT1 EVENT2 / OUT;
bit EVENT1, EVENT2, OUT;
int flag;
initial
{       OUT = #b0;
        flag = 0;}
forever
{       waitfor rise(EVENT1);
        if (flag == 0)
        {flag = 1;}
        elseif (flag == 2)
```

```
        {       OUT = #b1;
                waitfor +1;
                OUT = #b0;
        flag = 0;}
}
.eom ;
```

Figure E.4 Behavioral description of an AND event detector blocks using SBL.

The **waitfor+1** statement limits the width of the pulse to one simulation time unit. Initially the output of the detector is zero. In the event AND detector, an internal flag remembers if a positive transition in the inputs has been detected.

Appendix F. Hardware Specification of the Interface.

In this appendix, the hardware implementation of the interface discussed in chapter 6 is described. This implementation was tested in Silos II, using standard gates and flip-flops. Macros are used to represent the building blocks of the interface: two-state ASMs, mutual exclusion elements, event-detectors, and the bus observer.

```
.title VMEbus Bus Arbitration Interface : HARDWARE IMPLEMENTATION

.MACRO two_state_asm S R O RESET

NS      .inv    S
NR      .inv    R
ny0     .inv    y0
ny1     .inv    y1
ny2     .inv    y2

t1      .and    NR NS y1 -RESET
t2      .and    NR y1 ny0 -RESET
t3      .and    y2 y1 -RESET
t4      .and    R S y2 -RESET
t5      .and    R y2 y0 -RESET
t6      .and    S y2 ny1 y0 -RESET
t7      .and    NS y1 y0 -RESET

t8      .and    R S ny2 -RESET
t9      .and    ny2 y1 -RESET
t10     .and    NR y1 -RESET
t11     .and    R NS y1 -RESET

t12     .and    NR S ny2 ny1 -RESET
t13     .and    NR ny2 y0 -RESET
t14     .and    S ny2 y0 -RESET
t15     .and    NR ny1 y0 -RESET
t16     .and    R S y0 -RESET
t17     .and    NR NS y0 -RESET
```

```
t18     .and    NS y1 y0 -RESET
t19     .and    R NS y2 y1 -RESET
t20     .and    R y2 y0 -RESET
t21     .and    y2 ny1 y0 -RESET
t22     .and    NS y2 y0 -RESET

y2      .or     t1 t2 t3 t4 t5 t6 t7
y1      .or     t8 t9 t10 t11
y0      .or     t12 t13 t14 t15 t16 t17 t18 t19 t20 t21 t22

O       .xor    y1 y0
.EOM two_state_asm


.MACRO single_event_detector EVENT OUT RESET VDD

OUT     .dff    VDD EVENT RES
RES     .or     RESET OUT

.EOM single_event_detector


.MACRO and_event_detector EVENT1 EVENT2 OUT RESET VDD

OUT     .and    OT1 OT2
RES     .or     RESET OUT
OT1     .dff    VDD EVENT1 RES
OT2     .dff    VDD EVENT2 RES

.EOM and_event_detector

.MACRO and_ev_st_detector EVENT1 STATE2 OUT RESET VDD

OUT     .and    OT1 STATE2
RES     .or     RESET OUT
OT1     .dff    VDD EVENT1 RES

.EOM and_ev_st_detector


.MACRO mutual_exclusive GI RI GO G RESET

ME1 .inv GI
ME2 .inv ME5
ME3 .and RI ME1 -RESET
ME4 .and RI ME5 -RESET
G   .and GI ME5 -RESET
GO  .and GI ME2 -RESET
ME5 .or  ME3 ME4 G
.EOM mutual_exclusive;
```

```
.MACRO bus_observer bbsy as busy RESET

n_bbsy      .inv    bbsy
n_as  .inv  as
n_y2  .inv  y2
n_y1  .inv  y1

t1      .and    y2 y1 -RESET
t2      .and    bbsy n_as y1 -RESET

t3      .and    n_bbsy as -RESET
t4      .and    as n_y2 y1 -RESET
t5      .and    bbsy n_y2 y1 -RESET

t6      .and    bbsy n_y2
t7      .and    n_y2 y1
t8       .and   as n_y2

y2      .or     t1 t2
y1      .or     t3 t4 t5 t2

busy    .or     t6 t7 t8
.EOM bus_observer



.DELAY      .DEFAULT = 5,.5 5,.5
.DELAY      .INV = 2,.3 2,.3
VDD         .clk    0 S1

ib_ri       .buf    HRQ
ib_not_ri   .inv    ib_ri
ib_gi       .inv    BG3IN*

HLDA        .buf    ib_a
BR3OUT*     .inv    ib_ro
BR3*        .and        BR3OUT* BR3IN*
BBSYOUT*    .inv    ib_ga
ib_not_ga   .inv    ib_ga
BG3OUT*     .inv    ib_go

BBSY*       .and    BBSYIN* BBSYOUT*
```

```
(asm_B          bus_observer            -BBSY* -AS* ib_b RESET
ib_not_b        .inv    ib_b

(ibaset         and_ev_st_detector      ib_ga ib_not_b ib_a_set RESET VDD
$(ibareset      single_event_detector   ib_not_ga ib_a_reset RESET VDD
(ibareset       and_event_detector      ib_not_ri ib_not_g ib_a_reset RESET VDD
(asm_A          two_state_asm           ib_a_set ib_a_reset ib_a RESET
ib_not_a        .inv    ib_a

(ibroset        single_event_detector   ib_ri ib_ro_set RESET VDD
(ibroreset      single_event_detector   ib_ga ib_ro_reset RESET VDD
(asm_RO         two_state_asm           ib_ro_set ib_ro_reset ib_ro RESET

(ibgaset        single_event_detector   ib_g ib_ga_set RESET VDD
(ibgareset      and_event_detector      ib_not_ri ib_not_g ib_ga_reset RESET
VDD
(asm_GA         two_state_asm           ib_ga_set ib_ga_reset ib_ga RESET

(asm_GO         mutual_exclusive        ib_gi ib_ri ib_go ib_g RESET
ib_not_g        .inv    ib_g
```

Vita

Surname:	Escalante	Given Names:	Marco Antonio
Place of Birth:	Mexico City, Mexico	Date of Birth:	1961-07-22

Educational Institutions Attended:

Universidad Iberoamericana (Mexico City, Mexico)	1979 to 1983

Degrees Awarded:

B.Sc. (Honours)	Universidad Iberoamericana, Mexico	1987

Honours and Awards:

University of Victoria Fellowship	1990-91

Publications:

1. Dimopoulos, N.J., Huber, B., Li, K.F., Caughey, D., Escalante, M., Li, D., Burnett, R., and Manning, E., "Modelling Components in DAME," In Proceedings of the Third International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, Vol. II, pp. 716-725, Charleston, South Carolina, July 15-18, 1990.

2. Huber, B., Escalante, M., Caughey, D., Dimopoulos, N.J., Li, K.F., Li, D., and Manning, E.G., "Microprocessor Components and Signal Behavior Modelling in DAME," In Proceedings of the 1990 Canadian Conference on Electrical and Computer Engineering, Vol. 1, pp. 27.1.1-27.1.4, Ottawa, Canada, Sep. 4-6, 1990.

3. Dimopoulos, N.J., Huber, B., Li, K.F., Caughey, D., Escalante, M., Li, D., Burnett, R., and Manning, E., "DAME: An Expert Microprocessor-Based-Systems-Designer. An Overview and Status Report," In Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing", Victoria, Canada, May 9-10, 1991.

4. Escalante, M., Dimopoulos, N.J., Huber, B., Li, K.F., Li, D., and Manning, E.G., "Generic Design Rules for the Design of Microprocessor Based Systems in DAME: Bus Arbitration Subsystem," In Proceedings of the 1991 IEEE International Symposium on Circuit and Systems, Singapore, Vol. 5, pp. 3166-3169, June 11-14, 1991.

5. Escalante, M.A., Huber, B., Dimopoulos, N.J., Li, K.F., Li, D., and Manning, E.G.,"Bus Arbitration Modelling and Design in Dame: an Expert Microprocessor-Based-Systems-Designer", To be published in the Proceedings of the International Symposium on Artificial Intelligence, Cancun, Mexico, November 11-14, 1991.

Partial Copyright License

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its user. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

Bus Arbitration Modelling and Design in DAME:
an Expert Microprocessor-Based-Systems Designer

Author

(Signature)

Marco Antonio Escalante

(Name in Block Letters)

September 26, 1991

(Date)